SPRING 3

ДЛЯ ПРОФЕССИОНАЛОВ

Pro SPRING 3

Clarence Ho
Rob Harrop

Apress®

Spring_title.indd 2 21.08.2012 19:30:23

SPRING 3

ДЛЯ ПРОФЕССИОНАЛОВ

Кларенс Хо
Роб Харроп

ВИЛЬЯМС

Москва • Санкт-Петербург • Киев
2013

Spring_title.indd 3 21.08.2012 19:30:23

ББК 32.973.26-018.2.75
Х68
УДК 681.3.07

Издательский дом “Вильямс”
Зав. редакцией *С.Н. Тригуб*
Перевод с английского *Ю.Н. Артеменко*
Под редакцией *Ю.Н. Артеменко*

По общим вопросам обращайтесь в Издательский дом “Вильямс” по адресу:
info@williamspublishing.com, http://www.williamspublishing.com

Хо, Кларенс, **Харроп**, Роб.

Х68 Spring 3 для профессионалов. : Пер. с англ. — М. : ООО “И.Д. Вильямс”, 2013. — 880 с. : ил. — Парал. тит. англ.

ISBN 978-5-8459-1803-1 (рус.)

ББК 32.973.26-018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства APress, Berkeley, CA.

Authorized translation from the English language edition published by APress, Inc., Copyright © 2012 by Clarence Ho and Rob Harrop.

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

Russian language edition is published by Williams Publishing House according to the Agreement with R&I Enterprises International, Copyright © 2013.

Научно-популярное издание

Кларенс Хо, Роб Харроп

Spring 3 для профессионалов

Верстка *Т.Н. Артеменко*
Художественный редактор *В.Г. Павлютин*

Подписано в печать 28.08.2012. Формат 70×100/16.
Гарнитура Times. Печать офсетная.
Усл. печ. л. 70,95. Уч.-изд. л. 57,3.
Тираж 1500 экз. Заказ № 0000.

Первая Академическая типография “Наука”
199034, Санкт-Петербург, 9-я линия, 12/28

ООО “И. Д. Вильямс”, 127055, г. Москва, ул. Лесная, д. 43, стр. 1

ISBN 978-5-8459-1803-1 (рус.)
ISBN 978-1-43-024107-2 (англ.)

© Издательский дом “Вильямс”, 2013
© by Clarence Ho and Rob Harrop, 2012

Оглавление

Об авторах 17

Глава 1. Введение в Spring 19

Глава 2. Начало работы 33

Глава 3. Пример приложения 57

Глава 4. Введение в IoC и DI в Spring 73

Глава 5. Детальные сведения о конфигурации Spring 135

Глава 6. Введение в аспектно-ориентированное программирование с использованием Spring 203

Глава 7. Дополнительные сведения об аспектно-ориентированном программировании в Spring и аннотациях 253

Глава 8. Поддержка JDBC в Spring 293

Глава 9. Использование Hibernate в Spring 339

Глава 10. Доступ к данным в Spring с использованием JPA 2 365

Глава 11. Использование MyBatis в Spring 415

Глава 12. Проектирование и реализация Spring-приложений 453

Глава 13. Управление транзакциями 475

Глава 14. Проверка достоверности с преобразованием типов и форматированием 509

Глава 15. Планирование задач в Spring 537

Глава 16. Использование удаленной обработки в Spring 553

Глава 17. Разработка веб-приложений в Spring 595

Глава 18. Проект Spring Web Flow и JSF 669

Глава 19. Тестирование в Spring 707

Глава 20. Проекты Spring: Spring Batch, Spring Integration и Spring Roo 735

Глава 21. Дополнительные сведения о примере приложения 769

Глава 22. Поддержка написания сценариев в Spring 809

Глава 23. Мониторинг приложений Spring 829

Приложение A. SpringSource Tool Suite (STS) 853

Предметный указатель 871

Содержание

Об авторах 17

О техническом рецензенте 18
От издательства 18

Глава 1. Введение в Spring 19

Что такое Spring? 19
Инверсия управления или внедрение зависимостей? 20
Эволюция внедрения зависимостей 21
По ту сторону внедрения зависимостей 23
Проект Spring 29
Происхождение Spring 29
Сообщество Spring 29
Spring для Microsoft .NET 30
Комплект SpringSource Tool Suite / IDE-среда Spring 30
Проект Spring Security 30
Проекты Spring Batch и Spring Integration 31
Другие проекты 31
Альтернативы Spring 31
JBoss Seam Framework 31
Google Guice 31
PicoContainer 32
Контейнер JEE 6 32
Резюме 32

Глава 2. Начало работы 33

Получение Spring Framework 34
Загрузка стандартного дистрибутива 34
Извлечение Spring из GitHub 34
Упаковка Spring 35
Модули Spring 35
Выбор модулей для разрабатываемого приложения 37
Модули Spring в репозитории Maven 38
Анализ зависимостей Spring 39
Примеры приложений 43
Получение исходного кода примеров Spring 43
Приложение Petclinic 44
Приложение Petclinic Groovy 44
Приложение jPetStore 44
Приложение Spring Configuration Basic 45
Приложение Spring Task and Scheduling 45
Приложение Spring MVC Showcase 45
Приложения Spring MVC Basic и Spring MVC Ajax 45
Приложение Spring Petcare 46
Примеры приложений для Spring Web Flow 46
Документация Spring 46
Помещение Spring в приложение “Hello World!” 47
Построение примера приложения “Hello World!” 47
Создание проекта Spring в STS 51
Резюме 55

Глава 3. Пример приложения 57

Требования для приложения SpringBlog 58
Безопасность и аутентификация 58
Просмотр записей блога 59
Помещение записей в блог 60
Комментирование записи блога 60
Фильтрация нежелательного контента 62
Присоединение файлов к записи блога или к ее комментарию 63
Аудит действий в блоге 63
RSS-лента 63
Загрузка блога из XML-файла 64
Реализация приложения SpringBlog 64
Инструменты разработки и управление зависимостями 64
Проектное решение для приложения 65
Управление конфигурацией приложения 65
Многоуровневая архитектура приложения SpringBlog 66
Реализация уровня постоянства 66
Реализация уровня обслуживания 68
Использование АОП для фильтрации нежелательного контента 68
Резюме 71

Глава 4. Введение в IoC и DI в Spring 73

Инверсия управления и внедрение зависимостей 74
Типы инверсии управления 74
Тип Dependency Pull 74
Тип Contextualized Dependency Lookup 75
Тип Constructor Dependency Injection 76
Тип Setter Dependency Injection 77
Выбор между внедрением и поиском 77
Выбор между Setter Injection и Constructor Injection 78
Инверсия управления в Spring 81
Внедрение зависимостей с помощью Spring 82
Бины и фабрики бинов 82
Реализации `BeanFactory` 83
Интерфейс `ApplicationContext` 85
Конфигурирование `ApplicationContext` 85
Варианты конфигурации Spring (XML и Java-аннотации) 85
Объявление компонентов Spring 87
Использование внедрения через метод установки 91
Использование внедрения через конструктор 92
Параметры внедрения 97
Использование внедрения через метод 112
Именование бинов 121
Режим создания экземпляров бинов 123
Разрешение зависимостей 127
Автосвязывание бина 128
Различные режимы автосвязывания 128
Когда используется автосвязывание 130
Наследование бинов 130
Резюме 132

Глава 5. Детальные сведения о конфигурации Spring 135

Влияние Spring на переносимость приложений 136
Управление жизненным циклом бинов 137
Привязка к созданию бина 138
Привязка к уничтожению бина 146
Превращение бинов в “осведомленные о платформе Spring” 153
Использование интерфейса `BeanNameAware` 154
Использование интерфейса `ApplicationContextAware` 155
Использование фабрик бинов 157
Пример специальной фабрики бинов: класс `MessageDigestFactoryBean` 158
Доступ к фабрике бинов напрямую 161
Использование атрибутов `factory-bean` и `factory-method` 162
Редакторы свойств для компонентов JavaBean 163
Встроенные редакторы свойств 164
Создание специального редактора свойств 169
Дополнительные сведения о конфигурации `ApplicationContext` 171
Интернационализация с помощью интерфейса `MessageSource` 172
Использование `MessageSource` в автономных приложениях 176
Интерфейс `MessageSourceResolvable` 176
События приложений 176
Доступ к ресурсам 179
Конфигурация, использующая Java-классы 181
Конфигурирование `ApplicationContext` в Java 181
Выбор между конфигурациями Java и XML 187
Профили 187
Пример использования средства профилей Spring 187
Соображения по поводу использования профилей 193
Абстракции `Environment` и `PropertySource` 193
Конфигурация, использующая аннотации JSR-330 197
Резюме 201

Глава 6. Введение в аспектно-ориентированное программирование с использованием Spring 203

Концепции АОП 205
Типы АОП 206
Статическое АОП 206
Динамическое АОП 206
Выбор типа АОП 206
АОП в Spring 207
Альянс АОП 207
Пример “Hello World!” в АОП 208
Архитектура АОП в Spring 210
Класс `ProxyFactory` 211
Создание совета в Spring 212
Советы и срезы в Spring 228
Интерфейс `Pointcut` 229
Что собой представляют прокси 245
Использование динамических прокси JDK 245
Использование прокси CGLIB 246
Сравнение производительности прокси 246
Какой прокси использовать? 250
Резюме 251

Глава 7. Дополнительные сведения об аспектно-ориентированном программировании в Spring и аннотациях 253

Расширенное использование срезов 254
Использование срезов потока управления 254
Использование компонуемых срезов 257
Компоновка и интерфейс `Pointcut` 260
Резюме по созданию срезов 260
Работа с введениями 261
Основы введения 261
Обнаружение модификации объекта с помощью введений 264
Резюме по введениям 269
Службы платформы, предназначенные для АОП 269
Декларативное конфигурирование АОП 269
Использование `ProxyFactoryBean` 270
Использование пространства имен `aop` 275
Использование аннотаций в стиле `@AspectJ` 280
Соображения по поводу декларативного конфигурирования АОП в Spring 284
Интеграция с AspectJ 285
Что собой представляет AspectJ 285
Использование одиночных экземпляров аспектов 286
АОП в примере приложения 290
Фильтрация нежелательного контента в SpringBlog 290
Резюме 292

Глава 8. Поддержка JDBC в Spring 293

Модель данных для кода примеров 294
Исследование инфраструктуры JDBC 297
Инфраструктура JDBC в Spring 302
Обзор пакетов JDBC в Spring 303
Подключения к базе данных и источники данных 303
Поддержка встроенной базы данных 307
Использование источников данных в классах DAO 308
Обработка исключений 309
Класс `JdbcTemplate` 311
Инициализация `JdbcTemplate` в классе DAO 311
Извлечение одиночного значения с использованием класса `JdbcTemplate` 312
Использование именованных параметров с помощью `NamedParameterJdbcTemplate` 313
Извлечение объектов предметной области с помощью `RowMapper<T>` 314
Извлечение вложенных объектов предметной области с помощью `ResultSetExtractor` 316
Классы Spring, моделирующие операции JDBC 318
Настройка DAO-классов JDBC с использованием аннотаций 318
Запрашивание данных с использованием `MappingSqlQuery<T>` 321
Обновление данных с использованием `SqlUpdate` 325
Вставка данных и извлечение сгенерированного ключа 327
Объединение операций в пакеты с помощью `BatchSqlUpdate` 329
Вызов хранимых функций с использованием `SqlFunction` 333
Использование Java-конфигурации 336
Проект Spring Data: расширения JDBC 337
Соображения по поводу использования JDBC 337
Резюме 338

Глава 9. Использование Hibernate в Spring 339

Создание служебного проекта Hibernate в STS 340
Модель данных для кода примера 342
Конфигурирование фабрики сеансов Hibernate 344
Объектно-реляционное отображение с использованием аннотаций Hibernate 346
Простое отображение 347
Отображение “один ко многим” 351
Отображение “многие ко многим” 352
Интерфейс `Session` в Hibernate 354
Выполнение операций базы данных с помощью Hibernate 354
Запрашивание данных с использованием языка запросов Hibernate 355
Вставка данных 360
Обновление данных 362
Удаление данных 362
Соображения по поводу использования Hibernate 363
Резюме 364

Глава 10. Доступ к данным в Spring с использованием JPA 2 365

Введение в JPA 2 366
Создание простого служебного проекта Spring JPA в STS 367
Модель данных для кода примеров 369
Конфигурирование `EntityManagerFactory` в JPA 369
Отображение ORM с использованием аннотаций JPA 371
Устранение уровня DAO 371
Внедрение диспетчера сущностей в классы уровня обслуживания 372
Операции базы данных в JPA 373
Запрашивание данных с использованием языка запросов постоянства Java 374
Вставка данных 381
Обновление данных 383
Удаление данных 384
Собственный запрос 385
Запрос с критерием, использующий API-интерфейс критериев JPA 2 387
Введение в проект Spring Data JPA 392
Добавление библиотечных зависимостей Spring Data JPA 392
Операции базы данных, использующие абстракцию `Repository` из проекта Spring Data JPA 394
Отслеживание изменений в сущностном классе 398
Отслеживание версий сущностей с использованием Hibernate Envers 405
Добавление зависимостей Hibernate Envers 406
Добавление таблиц для отслеживания версий сущностей 406
Конфигурирование `EntityManagerFactory` для отслеживания версий сущностей 407
Кодирование изменений для отслеживания версий сущностей и извлечения хронологии 409
Тестирование отслеживания версий сущностей 411
Соображения по поводу того, когда использовать JPA 412
Использование JPA в примере приложения 412
База данных серверной части 412
Использование JPA для реализации уровня постоянства 413
Аудит и отслеживание версий сущностей 413
Резюме 413

Глава 11. Использование MyBatis в Spring 415

Начало работы с MyBatis в Spring 416
Введение в MyBatis 416
Создание простого служебного проекта с поддержкой MyBatis в STS 416
Модель данных для кода примеров 418
Конфигурирование `SqlSessionFactory` и `MapperScannerConfigurer` в MyBatis 419
SQL-отображение в MyBatis 421
Интерфейсы отображателей и файлы SQL-отображения 422
XML-конфигурация SQL-отображения 424
Операции базы данных в MyBatis 424
Запрашивание данных 424
Вставка данных 440
Обновление данных 443
Удаление данных 448
Соображения по поводу того, когда использовать MyBatis 450
Использование MyBatis в примере приложения 450
База данных серверной части 450
Использование MyBatis для реализации уровня постоянства 451
Аудит и отслеживание версий сущностей 451
Резюме 451

Глава 12. Проектирование и реализация Spring-приложений 453

Проектирование интерфейсов 454
Причины выбора подхода с проектированием интерфейсов 454
Шаблон “Фабрика” 455
Влияние Spring на проектирование, основанное на интерфейсах 458
Построение объектной модели предметной области 459
Spring и объектная модель предметной области 459
DOM — это не то же самое, что объект значения 459
Причины создания объектной модели предметной области 460
Моделирование объектов предметной области 461
Инкапсулировать поведение или нет? 462
Объектная модель предметной области для примера приложения SpringBlog 463
Итоги по объектной модели предметной области 468
Проектирование и построение уровня доступа к данным 469
Практические соображения по поводу проектирования 470
Итоги по уровню доступа к данным 471
Проектирование уровня обслуживания 472
Для чего нужен уровень обслуживания? 472
Проектирование бизнес-интерфейсов 473
Итоги по уровню обслуживания 474
Резюме 474

Глава 13. Управление транзакциями 475

Исследование уровня абстракции транзакций Spring 476
Типы транзакций 476
Реализации интерфейса `PlatformTransactionManager` 478
Анализ свойств транзакций 479
Интерфейс `TransactionDefinition` 479
Интерфейс `TransactionStatus` 481
Модель данных и инфраструктура для кода примеров 481
Создание простого служебного проекта Spring JPA с зависимостями 481

Модель данных и общие классы 483
Декларативные и программные транзакции в Spring 485
Использование аннотаций для управления транзакциями 486
Использование XML-конфигурации для управления транзакциями 494
Использование программных транзакций 497
Соображения по поводу управления транзакциями 500
Глобальные транзакции в Spring 500
Инфраструктура для реализации примера применения JTA 500
Реализация глобальных транзакций с помощью JTA 501
Соображения по поводу использования диспетчера транзакций JTA 508
Резюме 508

Глава 14. Проверка достоверности с преобразованием типов и форматированием 509

Создание проекта для примеров в STS 510
Система преобразования типов Spring 510
Преобразование из String с использованием редакторов свойств 510
Появление в Spring 3 системы преобразования типов 514
Форматирование полей в Spring 3 519
Реализация специального форматировщика 519
Конфигурирование `ConversionServiceFactoryBean` 520
Проверка достоверности в Spring 522
Использование интерфейса `Validator` в Spring 523
Использование JSR-303: Bean Validation API 525
Какой API-интерфейс проверки достоверности использовать? 532
Проверка достоверности с преобразованиями типов и форматированием в примере приложения 533
Резюме 535

Глава 15. Планирование задач в Spring 537

Создание проекта для примеров в STS 537
Реализация планирования задач в Spring 538
Введение в абстракцию `TaskScheduler` 538
Пример задачи 540
Планирование задач с использованием пространства имен `task` 544
Планирование задач с использованием аннотаций 546
Асинхронное выполнение задач в Spring 547
Планирование задач в примере приложения 550
Резюме 551

Глава 16. Использование удаленной обработки в Spring 553

Создание проекта для примеров в STS 554
Реализация уровня обслуживания для примеров 554
Добавление обязательных зависимостей для серверной части JPA 555
Проверка работоспособности проекта 556
Модель данных для примеров 557
Реализация и конфигурирование интерфейса `ContactService` 559
Использование HTTP-активатора Spring 564
Отображение службы 564
Вызов службы 565
Использование JMS в Spring 566
Установка сервера ActiveMQ 567

Реализация слушателя JMS в Spring 569
Отправка сообщений JMS в Spring 571
Использование веб-служб REST в Spring 573
Введение в веб-службы REST 574
Добавление обязательных зависимостей для примеров 575
Веб-службы REST для контактной информации 575
Использование Spring MVC для отображения веб-служб REST 576
Использование `curl` для тестирования веб-служб REST 583
Использование класса `RestTemplate` для доступа к веб-службам REST 584
Защита веб-служб REST с помощью Spring Security 588
Использование JSR-303 с веб-службами REST 592
Удаленная обработка в примере приложения 593
Резюме 594

Глава 17. Разработка веб-приложений в Spring 595

Создание проекта для примеров в STS 597
Реализация уровня обслуживания для примеров 597
Модель данных для примеров 597
Реализация и конфигурирование интерфейса `ContactService` 598
Введение в MVC и Spring MVC 603
Введение в MVC 603
Введение в Spring MVC 605
Создание первого представления в Spring MVC 610
Конфигурирование сервлета диспетчера 611
Реализация класса `ContactController` 611
Реализация представления списка контактов 612
Тестирование представления списка контактов 613
Обзор структуры проекта Spring MVC 614
Интернационализация 616
Конфигурирование интернационализации в сервлете диспетчера 616
Модификация представления списка контактов для поддержки интернационализации 618
Оформление темами и применение шаблонов 619
Поддержка оформления темами 620
Применение шаблонов представлений с помощью Apache Tiles 622
Реализация представлений для информации о контактах 628
Отображение URL на представления 628
Реализация представления просмотра контакта 629
Реализация представления редактирования контакта 632
Реализация представления добавления контакта 636
Включение проверки достоверности бинов JSR-303 637
Использование jQuery и jQuery UI 640
Введение в jQuery и jQuery UI 641
Активизация jQuery и jQuery UI в представлении 641
Редактирование форматированного текста с помощью CKEditor 643
Построение сетки данных, поддерживающей разбиение на страницы, с использованием jqGrid 645
Обработка загрузки файлов 651
Конфигурирование поддержки загрузки файлов 652
Изменение представлений для поддержки загрузки файлов 653
Изменение контроллера для поддержки загрузки файлов 654

Защита веб-приложения с помощью Spring Security 656
Конфигурирование Spring Security 657
Добавление к приложению функций входа 659
Использование аннотаций для защиты методов контроллера 662
Поддержка конфигурации на основе кода для Servlet 3 663
Spring MVC в примере приложения 665
Реализация MVC для приложения SpringBlog 665
Насыщенный пользовательский интерфейс и Ajax 666
Поддержка безопасности 666
Поддержка Servlet 3.0 667
Резюме 667

Глава 18. Проект Spring Web Flow и JSF 669

Проект примера серверной части 669
Уровень обслуживания для примера серверной части 670
Импорт проекта примера серверной части в STS 670
Введение в Spring Web Flow 672
Модули Spring Web Flow 672
Функциональные возможности Spring Web Flow 673
Введение в JSF 674
Представление 675
Взаимодействие с моделью 675
Навигация 676
Жизненный цикл приложения 676
Пример приложения Spring Web Flow 677
Проектирование примера потока 677
Структура проекта 678
Конфигурация Spring Web Flow и JSF 679
Добавление обязательных зависимостей 679
Конфигурирование JSF 679
Конфигурирование дескриптора веб-развертывания 680
Конфигурирование Spring Web Flow и Spring MVC 682
Реализация примера потока 684
Создание определения потока 684
Реализация шаблонной страницы 687
Реализация специального преобразователя 688
Реализация контроллера и поддерживающего бина 689
Реализация представления просмотра контакта 695
Реализация потока добавления контакта 697
Шаг 1: ввод базовой информации 697
Шаг 2: выбор хобби 701
Шаг 3: просмотр введенной информации 703
Шаг 4: добавление контакта завершено 704
Резюме 706

Глава 19. Тестирование в Spring 707

Проект для примера веб-приложения 708
Импорт проекта примера серверной части в STS 708
Введение в корпоративную инфраструктуру тестирования 710
Реализация модульного тестирования логики 712
Добавление обязательных зависимостей 712
Модульное тестирование контроллера Spring MVC 713

Реализация модульного тестирования взаимодействия 718
Добавление обязательных зависимостей 718
Конфигурирование профиля для тестирования уровня обслуживания 719
Реализация классов инфраструктуры 720
Модульное тестирование уровня обслуживания 724
Реализация модульного тестирования интерфейсной части 728
Добавление обязательных зависимостей 728
Введение в Selenium 728
Реализация тестового сценария для пользовательского интерфейса 729
Контроль покрытия кода тестовыми сценариями 731
Резюме 733

Глава 20. Проекты Spring: Spring Batch, Spring Integration и Spring Roo 735

Проект для примеров этой главы 736
Импорт проекта примера серверной части в STS 736
Введение в Spring Batch 737
Поток и процесс пакетного задания 737
Компоненты инфраструктуры Spring Batch 738
Метаданные Spring Batch 738
Политики выполнения заданий 740
Реализация пакетного задания 741
Добавление обязательных зависимостей 741
Конфигурирование инфраструктуры Spring Batch 742
Реализация задания импорта контактов 743
Использование Spring Batch и Spring Integration 751
Введение в Spring Integration 752
Добавление обязательных зависимостей 753
Реализация механизма опроса файлов 753
Введение в Spring Roo 757
Конфигурирование Spring Roo в STS 758
Создание проекта Spring Roo 758
Настройка уровня постоянства и сущностного класса 758
Настройка уровня обслуживания 763
Настройка уровня презентаций 764
Дополнение Spring Roo 767
Соображения по поводу Spring Roo 768
Spring Batch и Spring Integration в примере приложения 768
Резюме 768

Глава 21. Дополнительные сведения о примере приложения 769

Настройка примера приложения 770
Настройка проекта 770
Переключение между базами данных MySQL и H2 773
Переключение между реализациями JPA и MyBatis 775
Проект приложения 776
Модель данных 776
Объектная модель предметной области 777
Модель UML 779
Подробности конфигурирования 785
Детали реализации 792
Реализация уровня обслуживания 792
Фильтрация нежелательного контента с использованием АОП 797

Планирование задания для очистки данных аудита 800
Уровень презентаций 801
Резюме 807

Глава 22. Поддержка написания сценариев в Spring 809

Проект для примеров этой главы 810
Создание простого служебного проекта Spring 810
Установка подключаемого модуля Groovy для Eclipse 810
Поддержка написания сценариев в Java 813
Введение в Groovy 815
Динамическая типизация 815
Упрощенный синтаксис 816
Замыкание 817
Использование Groovy в Spring 818
Добавление обязательных зависимостей 819
Предметная область, связанная с контактами 819
Реализация процессора правил 820
Реализация фабрики правил как обновляемого бина Spring 823
Тестирование правила возрастной категории 825
Резюме 827

Глава 23. Мониторинг приложений Spring 829

Проект для примеров этой главы 830
Поддержка JMX в Spring 832
Экспорт бина Spring в JMX 832
Настройка VisualVM для мониторинга JMX 833
Мониторинг вошедших пользователей 835
Мониторинг статистики Hibernate 840
Мониторинг пакетных заданий Spring 842
Мониторинг приложения с помощью Spring Insight 845
Введение в Spring Insight 845
Конфигурирование Spring Insight 846
Использование Spring Insight 848
Резюме 851

Приложение A. SpringSource Tool Suite (STS) 853

Введение в STS 853
Установка STS 854
Установка автономной версии STS 854
Установка STS в существующей среде Eclipse 858
Настройка проекта и управление зависимостями 862
Создание простого служебного проекта Spring 862
Управление зависимостями для проекта 866
Использование STS 867
Установка расширений STS 867
Конфигурирование VMware tc Server в STS 869
Резюме 870
Предметный указатель 871
Содержание 872

Предметный указатель 871

Всем моим друзьям и коллегам, с которыми я работал, за предоставленную неоценимую поддержку, помощь и великолепный опыт работы. Моей семье, особенной моей маме Енг, за настоящую любовь и горячую поддержку всех моих решений. Также команде авторов, включая Криса, Мануэля и Брента, за необыкновенную поддержку при написании этой книги.

— Кларенс Хо

Об авторах

Кларенс Хо — ведущий Java-архитектор в фирме SkywideSoft Technology Limited (`www.skywidesoft.com`), занимающейся консультациями по программному обеспечению и расположенной в Гонконге. Работая в сфере информационных технологий более 20 лет, Кларенс выступал в качестве главы команды во многих проектах по разработке приложений на дому, а также предоставлял услуги консультаций по производственным решениям своим клиентам. Кларенс начал программировать на Java в 2001 г., но уже с 2005 г. был вовлечен в проектирование и разработку JEE-приложений с такими технологиями, как EJB, Spring Framework, Hibernate, JMS, WS и т.д. С тех пор он выступал в качестве Java-архитектора корпоративных приложений.

В настоящее время Кларенс работает консультантом в международном финансовом учреждении, принимая участие в различных областях, в числе которых архитектурное проектирование Java EE, обучение, предоставление рекомендаций по технологическим решениям и рекомендуемые приемы разработки приложений.

В свободное от работы время Кларенс занимается спортом (бегом трусцой, плаванием, футболом, пешеходным туризмом), чтением, просмотром фильмов, общением с друзьями и т.п.

Роб Харроп является соучредителем SpringSource — компании, которая стоит за успешным проектом Spring Framework. В настоящее время он занимает должность руководителя технического отдела в First Banco. До SpringSource Роб был соучредителем и техническим директором в Cake Solutions (Манчестер, Соединенное Королевство). Он специализируется на крупных масштабируемых корпоративных системах. В настоящее время он занимает должность руководителя технического отдела в First Banco. До SpringSource Роб был соучредителем и техническим директором в Cake Solutions (Манчестер, Соединенное Королевство). Он специализируется на крупных масштабируемых корпоративных системах.

Роб выступал автором и соавтором в более чем 5 книгах. Он доступен в Твиттере как `@robertharrop`.

О техническом рецензенте

Мануэль Джордан Элера — разработчик-самоучка и исследователь, которому нравится изучать новые технологии для собственных экспериментов и создавать новые интеграции.

Мануэль выиграл звание 2010 Springy Award — Community Champion. В свободное от работы время (которого не особенно много) он читает Библию и сочиняет музыку на своей гитаре. Он является ведущим участником на форумах сообщества Spring (Spring Community Forums) и известен там как *dr_pompeii*.

Мануэль выступал техническим рецензентом в следующих книгах, выпущенных издательством Apress:

- *Pro SpringSource dm Server* (2009 г.)
- *Spring Enterprise Recipes* (2009 г.)
- *Spring Recipes (Second Edition)* (2010 г.)
- *Pro Spring Integration* (2011 г.)
- *Pro Spring Batch* (2011 г.)
- *Pro Spring MVC: With Web Flow* (2012 г.)

Найти его можно в собственном блоге по адресу `http://manueljordan.wordpress.com/`, а также в Твиттере как `@dr_pompeii`.

От издательства

Вы, читатель этой книги, и есть главный ее критик и комментатор. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать и любые другие замечания, которые вам хотелось бы высказать в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо, либо просто посетить наш веб-сервер и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится или нет вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию последующих книг.

Наши координаты:

E-mail: `info@williamspublishing.com`
WWW: `http://www.williamspublishing.com`

Информация для писем из:

России: 127055, г. Москва, ул. Лесная, д. 43, стр. 1
Украины: 03150, Киев, а/я 152

ГЛАВА 1

Введение в Spring

Когда мы думаем о сообществе разработчиков на языке Java, мы вспоминаем полчища старателей, которые в конце 1840-х годов неистово прочесывали реки Северной Америки в поисках золотых самородков. Наши "реки" — реки Java-разработчиков — изобилуют проектами с открытым кодом, но, в отличие от истории со старателями, нахождение действительно полезного проекта может оказаться длительным по времени и трудным.

Многим проектам с открытым кодом Java характерна общая особенность — они были призваны просто заполнить пробел в реализации последней "модной" технологии или шаблона. С учетом сказанного, многие высококачественные и полезные проекты предназначены для решения реальных потребностей в реальных приложениях, и в этой книге вы встретите ряд таких проектов. С одним из них вы ознакомитесь достаточно подробно — и это, как не трудно догадаться, Spring.

На протяжении всей книги вы будете сталкиваться со многими применениями разнообразных технологий с открытым кодом, причем все они объединены в платформу Spring Framework. Благодаря Spring, разработчик приложений может пользоваться широким спектром инструментов с открытым кодом, не занимаясь написанием больших объемов кода и не привязывая создаваемое приложение слишком тесно к какому-то конкретному инструменту.

В этой главе вы ознакомитесь с платформой Spring Framework без просмотра крупных примеров или изучения подробных объяснений. Если вы уже знакомы с проектом Spring, можете сразу переходить к чтению главы 2.

Что такое Spring?

Возможно, самой трудной частью объяснения Spring как технологии является точная классификация того, что Spring собой представляет. Обычно Spring описывают как облегченную платформу для построения Java-приложений, но с этим утверждением связаны два интересных момента. Во-первых, Spring можно использовать для построения любого приложения на языке Java (т.е. автономных, веб-приложений, приложений JEE и т.д.), что отличает Spring от многих других платформ, таких как Apache Struts, которая ограничена только веб-приложениями. Во-вторых, характеристика "облегченная" в действительности не имеет никакого отношения к количеству классов или размеру дистрибутива; напротив, она определяет принцип всей философии Spring — минимальное воздействие. Платформа Spring является облегченной в том смысле, что для использования ядра Spring вы должны вносить минимальные (если вообще какие-либо) изменения в код своего приложения, а если в какой-то момент вы решите больше не пользоваться Spring, то и это сделать очень просто. Обратите внимание, что речь идет только о ядре Spring — многие дополнительные компоненты Spring, такие как доступ к данным, требуют более тесной привязки к Spring Framework. Однако польза от такой

привязки вполне очевидна, и практически повсеместно в книге мы будет представлять приемы минимизации влияния этой привязки на разрабатываемые приложения.

Инверсия управления или внедрение зависимостей?

Ядро Spring Framework основано на принципе инверсии управления (Inversion of Control — IoC), при котором создание и управление зависимостями между компонентами становятся внешними. Рассмотрим пример, в котором класс `Foo` при выполнении обработки определенного вида зависит от экземпляра класса `Bar`. Традиционно `Foo` создает экземпляр `Bar`, используя операцию `new` или получая нужный экземпляр от некоторого фабричного класса. Согласно подходу IoC, экземпляр класса `Bar` (или его подкласса) предоставляется классу `Foo` во время выполнения некоторым внешним процессом. Такое поведение, т.е. внедрение зависимостей во время выполнения, было переименовано Мартином Фаулером на более описательное — внедрение зависимостей (Dependency Injection — DI). Точная природа зависимостей, управляемых DI, обсуждается в главе 4.

На заметку! Как будет показано в главе 4, использование термина внедрение зависимостей при ссылке на инверсию управления всегда корректно. В контексте Spring эти термины можно применять взаимозаменяемо, совершенно не теряя при этом смысла.

Реализация DI в Spring основана на двух ключевых концепциях Java — компонентах JavaBean и интерфейсах. При использовании Spring в качестве поставщика DI вы получаете гибкость определения конфигурации зависимостей внутри своих приложений разнообразными путями (т.е. внешне в XML-файлах, с помощью конфигурационных Java-классов Spring или посредством аннотаций Java в коде). Компоненты JavaBean (также называемые POJO (Plain Old Java Object — простой старый объект Java)) предоставляют стандартный механизм для создания ресурсов Java, которые являются конфигурируемыми множеством способов. В главе 4 вы увидите, как Spring использует спецификацию JavaBean для формирования ядра своей модели конфигурации DI; фактически на любой ресурс, управляемый Spring, ссылаются как на *бин* (bean). На тот случай, если вы еще не знакомы с технологией JavaBean, в начале главы 4 будет приведен краткий пример.

Интерфейсы и DI — это взаимовыгодные технологии. Мы уверены, что любой читатель этой книги согласится с утверждением о том, что проектирование и написание кода интерфейсов предназначено для гибких приложений, но сложность сборки вместе приложения, использующего интерфейсы, довольно высока и требует от разработчиков дополнительных усилий по кодированию. За счет применения DI объем кода, который необходим при проектировании приложения на основе интерфейсов, снижается почти до нуля. Кроме того, с помощью интерфейсов можно получить максимальную отдачу от DI, потому что бины могут использовать любую реализацию интерфейса для удовлетворения их зависимости.

В контексте внедрения зависимостей Spring действует больше подобно контейнеру, чем платформе, предоставляя экземпляры классов вашего приложения со всеми необходимыми зависимостями, но делает это гораздо менее навязчивым способом. Использование Spring для DI основано всего лишь на следовании внутри своих классов соглашениям об именовании, принятым для компонентов JavaBean (требование, которое, как будет показано в главе 5, можно обойти, воспользовавшись поддержкой внедрения через метод в Spring); ни специальных классов, предназначенных для наследования, ни собственных схем именования, которые должны соблюдаться, не предусмотрено. Во всяком случае, единственным изменением, которое делается в приложении, использующем DI, является открытие доступа к дополнительным свойствам ваших компонентов JavaBean, что позволяет внедрять больше зависимостей во время выполнения.

На заметку! В дополнение к конфигурационным XML-файлам, платформа Spring Framework версии 3.0 (и последующих версий) поддерживает метаданные Java-бинов.

Эволюция внедрения зависимостей

Благодаря популярности Spring и других платформ DI, за последние несколько лет технология внедрения зависимостей обрела широкое признание в сообществе разработчиков на Java. В то же время, разработчики были уверены, что использование DI было наилучшей практикой в разработке приложений, и преимущества применения DI были также хорошо изучены.

Широкое распространение DI также оказала влияние на развитие JCP (Java Community Process — процесс сообщества Java), возглавляемого компанией Sun Microsystems (которая была приобретена Oracle в 2009 г.). В 2009 г. внедрение зависимостей для Java ("Dependency Injection for Java") стало формальным запросом спецификации Java (Java Specification Request (JSR-330)) и, как и можно было предположить, автором одной из лидирующих спецификаций был Род Джонсон, создатель Spring Framework.

В Java Enterprise Edition версии 6 (или, сокращенно, JEE 6), JSR-330 стала одной из спецификаций, включенных в стек технологий. Тем временем, архитектура Enterprise JavaBeans (EJB), начиная с версии 3.0, также была значительно модернизирована; с целью упрощения разработки разнообразных приложений Enterprise JavaBeans была внедрена модель DI.

Несмотря на то что подробное обсуждение DI откладывается до главы 4, полезно взглянуть на преимущества, которые обеспечивает использование DI по сравнению с более традиционным подходом.

- Сокращение объема связующего кода. Одним из самых больших плюсов DI является возможность значительного сокращения объема кода, который должен быть написан для связывания вместе различных компонентов приложения. Зачастую этот код очень прост — при создании зависимости должен создаваться новый экземпляр соответствующего объекта. Однако связующий код может оказаться довольно сложным, когда нужно искать зависимости в репозитории JNDI или если зависимости не могут вызываться напрямую, как в случае с удаленными (дистанционно) ресурсами. В таких ситуациях DI может действительно упростить связующий код, предоставляя автоматический поиск в JNDI и автоматическое создание прокси для удаленных ресурсов.
- Упрощенная конфигурация приложения. За счет применения DI процесс конфигурирования приложения значительно упрощается. Для конфигурирования классов, которые могут быть внедрены в другие классы, можно использовать аннотации или XML-файлы. Те же самые приемы можно применять для формулирования требований зависимостей для внедряющего объекта при внедрении соответствующего экземпляра бина или свойства. Вдобавок DI намного упрощает обмен одной реализации зависимости с другой. Рассмотрим случай, когда есть компонент объекта доступа к данным (data access object — DAO), который выполняет операции над данными в базе данных PostgreSQL, и нужно модернизировать его до Oracle. С помощью DI соответствующую зависимость ваших бизнес-объектов можно просто реконфигурировать, чтобы она использовала реализацию Oracle вместо PostgreSQL.
- Возможность управления общими зависимостями в единственном репозитории. При традиционном подходе к управлению зависимостями в общих службах, к ко-

торым относятся, например, подключение к источнику данных, транзакция, удаленные службы и т.п., вы создаете экземпляры (или получаете их из определенных фабричных классов) зависимостей там, где они нужны — внутри зависимого класса. Это приводит к распространению зависимостей по множеству классов в приложении, что может затруднить их изменение. В случае использования DI вся информация об общих зависимостях содержится в единственном репозитории (в Spring есть возможность хранить эту информацию в XML-файлах или Java-классах), что существенно упрощает управление зависимостями и снижает количество возможных ошибок.

- Улучшенная возможность тестирования. Когда классы проектируются для DI, становится возможной простая замена зависимостей. Это особенно полезно при тестировании приложения. Рассмотрим бизнес-объект, который осуществляет определенную сложную обработку; в рамках этого он использует DAO для доступа к данным, хранящимся в реляционной базе данных. При тестировании вас не интересует проверка DAO — вам нужно протестировать бизнес-объект с разнообразными наборами данных. При традиционном подходе, когда бизнес-объект самостоятельно отвечает за получение экземпляра DAO, тестирование этого довольно затруднительно, поскольку нет возможности заменить реализацию DAO пробной реализацией, которая возвращала бы наборы тестовых данных. Вместо этого нужно будет удостовериться, что тестовая база данных содержит корректные данные и применять для прогона тестов полную реализацию DAO. В случае использования DI можно создать пробную реализацию объекта DAO, которая возвращает наборы тестовых данных, и затем передавать их бизнес-объекту с целью проверки. Этот механизм может быть расширен для тестирования любого уровня приложения и особенно полезен при тестировании веб-компонентов, для которых создаются пробные реализации `HttpServletRequest` и `HttpServletResponse`.
- Стимулирование качественных проектных решений для приложений. Вообще говоря, проектирование для DI означает проектирование с использованием интерфейсов. Типовое приложение, ориентированное на внедрение зависимостей, построено так, что все основные компоненты определяются как интерфейсы, после чего создаются конкретные реализации этих интерфейсов и затем связываются вместе с применением контейнера DI. Такая разновидность проектного решения была возможна в Java и до появления DI и основанных на DI контейнеров, подобных Spring, но, используя Spring, вы получаете в свое распоряжение целый ряд средств DI и можете сосредоточиться на построении логики приложения, а не на поддерживающей DI платформе.

Как видите, технология DI предоставляет немало преимуществ приложению, однако она отнюдь не лишена недостатков. В частности, DI может затруднить выяснение, какая реализация конкретной зависимости к каким объекта привязана, для тех, кто не особенно хорошо ориентируется в коде. Как правило, эта проблема возникает только при отсутствии опыта работы с DI. С обретением опыта и следуя рекомендациям по программированию для DI (например, помещение всех встраиваемых классов на каждом уровне приложения в один и тот же пакет), разработчики получат возможность легко получить представление о картине в целом. Почти всегда преимущества перевешивают этот небольшой недостаток, тем не менее, о нем следует помнить при планировании приложения.

По ту сторону внедрения зависимостей

Само по себе ядро Spring со своими развитыми возможностями DI является вполне достойным инструментом, но чем Spring действительно выделяется — так это широчайшим набором дополнительных средств, все их которых элегантно спроектированы и построены с использованием основных принципов DI. Платформа Spring предлагает средства для всех уровней приложения, от вспомогательных API-интерфейсов для доступа к данным до расширенных возможностей MVC (Model View Controller — модель–представление–контроллер). Упомянутые возможности Spring хороши тем, что хотя платформа Spring часто предоставляет собственный подход, их можно легко интегрировать с другими инструментами Spring, делая их полноправными членами семейства Spring.

Аспектно-ориентированное программирование с использованием Spring

Аспектно-ориентированное программирование (АОП) — это одна из "моделей программирования текущего момента" в мире Java. АОП предоставляет возможность реализации сквозной логики — т.е. логики, которая применяется к множеству частей приложения — в одном месте и обеспечения автоматического применения этой логики по всему приложению. В настоящее время АОП находится в центре повышенного внимания, однако, несмотря на поднятую шумиху, АОП является очень полезной технологией, которая должна поддерживаться любым инструментальным набором для разработчиков на Java.

Подход Spring к АОП заключается в создании "динамических прокси" для целевых объектов и "привязывании" объектов к сконфигурированному совету для выполнения сквозной логики.

Еще одной популярной библиотекой АОП является проект Eclipse AspectJ (`www.eclipse.org/aspectj`), который предоставляет более мощные средства, включая конструирование объектов, загрузку классов и большую возможность выделения сквозной функциональности.

Хорошей новостью для разработчиков, использующих Spring и АОП, станет тот факт, что, начиная с версии 2.0, в Spring поддерживается более тесная интеграция с AspectJ. Ниже перечислено несколько особенностей.

- Поддержка выражений со срезами в стиле AspectJ.
- Поддержка стиля аннотаций `@AspectJ` при использовании для привязывания АОП из Spring.
- Поддержка аспектов, реализованных в AspectJ для DI.
- Поддержка привязывания во время загрузки внутри объекта `ApplicationContext` в Spring.

Оба вида АОП имеют свои области применения, и в большинстве случаев АОП из Spring достаточно для удовлетворения требований сквозной функциональности в приложении. Однако для более сложных требований может использоваться AspectJ, и в одном и том же приложении Spring можно смешивать оба вида АОП.

АОП имеет много применений. Типичное применение, демонстрируемое во множестве традиционных примеров АОП, связано с выполнением некоторого вида регистрации, но возможности АОП выходят далеко за рамки тривиальных приложений регистрации. На самом деле внутри самой платформы Spring Framework АОП используется для многих целей, в частности, при управлении транзакциями. АОП в Spring подробно рассматривается в главах 6 и 7, где демонстрируются типичные применения АОП в рамках Spring Framework и примерах приложений, обсуждаются вопросы производительности АОП и области, в которых традиционные технологии подходят лучше, чем АОП.

Язык выражений Spring (SpEL)

Язык выражений (Expression Language — EL) — это технология, позволяющая приложению манипулировать объектами Java во время выполнения. Однако с EL связана одна проблема: различные технологии предоставляют собственные реализации и синтаксис EL. Например, Java Server Pages (JSP) и Java Server Faces (JSF) имеют собственные языки EL, синтаксис которых отличается друг от друга. Для решения этой проблемы и был создан унифицированный язык выражений (Unified Expression Language).

Поскольку платформа Spring Framework развивалась слишком быстро, существовала потребность в стандартном языке выражений, который мог бы совместно использоваться всеми модулями Spring Framework, а также другими проектами Spring. Поэтому, начиная с версии 3.0, в Spring появился язык выражений Spring (Spring Expression Language — SpEL), который предлагает мощные средства для вычисления выражений, а также для доступа к объектам Java и бинам Spring во время выполнения. Результаты вычислений могут использоваться в приложении или внедряться в другие компоненты JavaBean.

В настоящей книге не предусмотрено отдельной главы, полностью посвященной SpEL. Тем не менее, по всей книге мы будем применять язык SpEL и при необходимости давать детальные объяснения.

Проверка достоверности в Spring

Проверка достоверности — еще одна обширная тема в приложениях любого вида. В идеальном сценарии правила проверки достоверности атрибутов внутри компонентов JavaBean, содержащих данные, могут быть применены согласованным образом вне зависимости от того, где инициирован запрос на манипуляцию данными — в пользовательском интерфейсе, пакетном задании или же удаленно (например, веб-службами, веб-службами, поддерживающими REST, RPC и т.д.).

Руководствуясь необходимостью, JCP была разработана спецификация Bean Validation API (JSR-303). API-интерфейс Bean Validation предлагает стандартный путь определения правил проверки достоверности для бинов. Например, когда к свойству бина применяется аннотация `@NotNull`, это означает, что атрибут не должен содержать нулевое значение перед тем, как он может быть сохранен в базе данных.

Начиная с версии 3.0, платформа Spring предоставляет встроенную поддержку спецификации JSR-303. Для использования этого API-интерфейса просто объявите `ValidatorFactoryBean` и внедрите интерфейс `Validator` в любые бины, управляемые Spring. Платформа Spring подготовит лежащую в основе реализацию. По умолчанию Spring сначала ищет Hibernate Validator (`hibernate.org/subprojects/validator`), который представляет собой популярную реализацию JSR-303. Многие технологии пользовательских интерфейсов (например, JSF 2 и Google Web Toolkit), включая Spring MVC, также поддерживают применение проверки достоверности JSR-303 в пользовательском интерфейсе. Время, когда разработчикам приходилось писать одну и ту же логику проверки достоверности как в пользовательском интерфейсе, так и на уровне работы с базой данных, прошло. Более подробно об этом пойдет речь в главе 14.

Доступ к данным в Spring

Доступ к данным и постоянство являются, пожалуй, наиболее обсуждаемыми темами в мире Java. Похоже, вам вряд ли удастся посетить веб-сайт сообщества, такой как `www.theserverside.com`, не попав под шквал статей и записей в блогах об очередном новейшем инструменте доступа к данным. Платформа Spring обеспечивает великолепную интеграцию с любым выбранным инструментом доступа к данным подобного рода. В дополнение к этому, для многих проектов Spring предлагает в качестве жизнеспособ-

ного решения простое средство JDBC (Java Database Connectivity — подключение к базам данных Java) с его упрощенными API-интерфейсами.

Начиная с версии 3.*x*, модуль доступа к данным Spring обеспечивает встроенную поддержку JDBC, Hibernate, MyBatis (ранее iBATIS), Java Data Object (JDO) и Java Persistence API (JPA).

Однако по причине безудержного роста Интернета и облачных вычислений в последние несколько лет, вдобавок к реляционным базам данных были разработаны другие базы "специального назначения". Примерами могут служить базы данных, основанные на парах "ключ-значение" и предназначенные для обработки исключительно больших объемов данных (в общем случае на них ссылаются как на NoSQL), графовые базы данных, документные базы данных и т.д. Чтобы помочь разработчикам в поддержке таких баз данных и не усложнять модель доступа к данным Spring, был создан отдельный проект под названием Spring Data (`www.springsource.org/spring-data`). Этот проект в дальнейшем был разделен на различные категории с целью поддержки более специфических требований к доступу к данным.

На заметку! Поддержка нереляционных баз данных в Spring в этой книге не рассматривается. Для тех, кто интересуется этой темой, упомянутый ранее проект Spring Data будет хорошей отправной точкой. На странице проекта указаны нереляционные базы данных, которые он поддерживает, а также предоставлены ссылки на домашние страницы для этих баз данных.

Поддержка JDBC в Spring делает построений приложения на основе JDBC вполне реальным, даже в особо сложных случаях. Поддержка Hibernate, MyBatis, JDO и JPA еще более упрощает и без того простые API-интерфейсы, сокращая затраты на кодирование со стороны разработчиков. При использовании API-интерфейсов Spring для доступа к данным через любой инструмент имеется возможность получить все преимущества великолепной поддержки транзакций, предлагаемой Spring. Более подробные сведения будут предоставлены в главе 13.

Одним из самых полезных средств Spring является возможность простого комбинирования технологий доступа к данным в рамках приложения. Например, приложение может работать с базой данных Oracle, но использовать Hibernate для большей части логики доступа к данным. Однако если требуются некоторые специфичные для Oracle средства, очень просто реализовать соответствующую часть уровня доступа к данным с помощью API-интерфейсов JDBC, поддерживаемых Spring.

Поддержка Object to XML Mapping (OXM) в Spring

Многие приложения должны интегрироваться или предоставлять службы для других приложений. Одним из распространенных требований является обмен данными с другими системами, либо на регулярной основе, либо в реальном времени. Наиболее общим форматом данных считается XML. В результате возникает необходимость в преобразовании компонента JavaBean в формат XML и наоборот.

Платформа Spring поддерживает много общепринятых инфраструктур отображения Java в XML и, как обычно, устраняет потребность в непосредственной привязке к любой специфической реализации. Spring предоставляет общие интерфейсы маршализации (преобразования компонентов JavaBean в XML) и демаршализации (преобразования XML в объекты Java) для DI в любые бины Spring. Поддерживаются такие общие библиотеки, как Java API for XML Binding (JAXB), Castor, XStream и XMLBeans. При рассмотрении в главе 16 удаленного доступа приложения Spring к бизнес-данным в формате XML будет показано, как использовать в своих приложениях поддержку OXM (Object to XML Mapping — отображение объектов в XML), предлагаемую Spring.

Управление транзакциями

Платформа Spring предоставляет великолепный уровень абстракции для управления транзакциями, который позволяет осуществлять программный и декларативный контроль над транзакциями. За счет использования уровня абстракции для транзакций упрощается внесение изменений в лежащий в основе протокол транзакций и диспетчеры ресурсов. Вы можете начать с простых, локальных, специфичных для ресурса транзакций и перейти к глобальным, многоресурсным транзакциям, не изменяя уже написанного кода.

Транзакции подробно рассматриваются в главе 13.

Упрощение и интеграция с JEE

Как указывалось ранее, за последние несколько лет платформы DI наподобие Spring получили широкое распространение, и многие разработчики решили строить приложения, используя платформы DI для поддержки подхода EJB из JEE. В результате сообщество JCP также столкнулось со сложностью EJB, и в версиях 3.0 и 3.1 спецификации EJB соответствующий API-интерфейс был упрощен и теперь включает многие концепции DI.

Однако для приложений, которые были построены на EJB или должны развертывать Spring-приложения в контейнере JEE и пользоваться службами корпоративного уровня на сервере приложений (например, JTA Transaction Manager (диспетчер транзакций JTA), пул подключений к источникам данных, фабрики подключений JMS и т.д.), Spring также предлагает упрощенную поддержку упомянутых технологий. Для EJB платформа Spring предоставляет простое объявление для выполнения поиска JNDI и внедрения в бины Spring. На противоположной стороне Spring также обеспечивает простую аннотацию для внедрения бинов Spring в компоненты EJB.

Для любых ресурсов, сохраненных в местоположении, которое доступно через JNDI, платформа Spring позволяет избавиться от сложного кода поиска и иметь управляемые JNDI ресурсы как внедренные зависимости в другие объекты во время выполнения. В качестве побочного эффекта приложение становится не привязанным к JNDI, увеличивая степень многократного использования кода в будущем.

MVC на веб-уровне

Хотя платформа Spring может использоваться практически в любых конфигурациях, от настольной системы до веб, она предлагает широкий спектр классов, предназначенных для поддержки создания веб-приложений. Благодаря Spring, вы получаете максимальную гибкость при выборе способа реализации пользовательского интерфейса для веб-приложения.

Наиболее популярным шаблоном, применяемым при разработке веб-приложений, является MVC. В последних версиях Spring постепенно развилась от простой веб-платформы до полноценной реализации MVC.

Первой следует отметить обширную поддержку представлений в Spring MVC. В дополнение к стандартной поддержке JSP, которая значительно укреплена библиотеками дескрипторов Spring, в распоряжении разработчиков имеется полностью интегрированная поддержка Apache Velocity, FreeMarker, Apache Tiles и XSLT. Вдобавок доступен набор базовых классов представлений, которые упрощают добавление к приложениям вывода Excel и PDF.

Во многих случаях вы сочтете поддержку Spring MVC вполне достаточной для удовлетворения требований к разработке веб-приложений. Однако Spring может также интегрироваться с другими популярными веб-платформами, такими как Struts, JSF, Google Web Toolkit (GWT) и т.д.

В последние годы технология веб-платформ стремительно развивалась. Пользователи требовали более быстрого отклика и высокой интерактивности, и это привело к появлению Ajax — широко распространенной технологии для разработки насыщенных Интернет-приложений (Rich Internet Application — RIA). С другой стороны, пользователи также хотят иметь доступ к своим приложениям на любом устройстве, включая смартфоны и планшетные ПК. В результате для удовлетворения существующим требованиям веб-платформы должны поддерживать HTML5, JavaScript и CSS3. В главе 17 будет рассматриваться разработка веб-приложений с использованием Spring MVC, jQuery (популярная библиотека JavaScript, поддерживающая Ajax и многие другие средства) и JSP.

Еще одним проектом Spring, ориентированным на разработку веб-приложений, который следовало бы упомянуть, является Spring Web Flow (`www.springframework.org/webflow`). Этот проект был создан для удовлетворения потребностей в управлении потоком страницы, а также для предоставления более точного контроля над данными веб-приложения, которые должны передаваться между серией страниц (т.н. *диалоговая область*). В главе 18 мы кратко рассмотрим проект Spring Web Flow и его интеграцию с PrimeFaces (`www.primefaces.org`) — популярной платформой JSF.

Поддержка удаленных технологий

Доступ или отображение удаленных компонентов в Java никогда не было простой работой. Используя Spring, вы можете получить в свое распоряжение обширную поддержку широкого спектра удаленных технологий для быстрого отображения и доступа к удаленным службам.

Платформа Spring предлагает поддержку разнообразных механизмов удаленного доступа, в том числе Java RMI, JAXRPC, Caucho Hessian и Caucho Burlap. В дополнение к этим удаленным протоколам, Spring также предоставляет собственный вызывающий объект на основе HTTP, базирующийся на стандартной сериализации Java. Пользуясь возможностями динамического создания прокси Spring, вы получаете прокси для удаленного ресурса, внедренный в виде зависимости в ваши классы; это избавляет от необходимости привязывать приложение к специфической реализации удаленной технологии, а также сокращает объем кода, который должен быть написан для приложения.

Еще одной технологией, которая получила широкое признание в наши дни, являются веб-службы REST (RESTful Web Services — RESTful-WS). Службы RESTful-WS спроектированы на основе HTTP и существенно упрощают механизм вызова служб и получения от них результатов. Например, когда клиент выдает HTTP-запрос GET к URL-адресу `http://somedomain.com/someapp/customer/123`, это означает, что ему необходимо извлечь информацию о заказчике, идентификатор которого равен 123. Возвращаемое значение может быть в формате XML, JSON (JavaScript Object Notation — нотация объектов JavaScript) или других форматах, поддерживаемых клиентами, как указано в заголовке HTTP-запроса. Начиная с версии 3.0, модуль Spring MVC обеспечивает всестороннюю поддержку RESTful-WS. Поддержка удаленных технологий в Spring будет рассматриваться в главе 16.

На заметку! В рамках JCP стандарт RESTful-WS формализован как JAX-RS — API-интерфейс Java для веб-служб REST (Java API for Restful Web Services (JSR 311)).

Поддержка электронной почты

Отправка электронной почты — это типовое требование для многих разновидностей приложений, которое полноценно поддерживается в Spring Framework. Платформа Spring предоставляет упрощенный API-интерфейс для отправки сообщений электронной почты, который хорошо вписывается в возможности Spring DI. Кроме того, Spring поддерживает стандартный API-интерфейс Java Mail.

Spring позволяет создать прототипное сообщение в контейнере DI и применять его в качестве основы для всех сообщений, отправляемых из приложения. Это упрощает настройку параметров почтового сообщения, таких как тема и адрес отправителя. Вдобавок для настройки тела сообщения платформа Spring интегрируется с механизмами шаблонов наподобие Apache Velocity, которые дают возможность генерировать содержимое сообщений с помощью Java-кода.

Поддержка планирования заданий

Большинство нетривиальных приложений требуют определенных возможностей планирования. Является это отправкой обновленной информации заказчикам или выполнением служебных задач, возможность планирования запуска кода в заранее определенный момент времени представляет собой очень полезное средство для разработчиков.

Платформа Spring предлагает собственную поддержку планирования, которая удовлетворяет большинству общих сценариев. Задача может быть запланирована на запуск либо с фиксированным интервалом, либо с использованием выражения для Unix-утилиты `cron`.

С другой стороны, для выполнения и планирования задач Spring также интегрируется с другими библиотеками планирования. Например, в среде сервера приложений Spring может делегировать выполнение библиотеке CommonJ, которая применяется многими распространенными серверами приложений. Для целей планирования задач Spring также поддерживает ряд библиотек, в числе которых JDK Timer API и Quartz, представляющие собой известные библиотеки планирования с открытым кодом.

Поддержка планирования в Spring подробно рассматривается в главе 15.

Поддержка динамических сценариев

Начиная с JDK 6, в Java появилась поддержка динамического языка, которая позволяет запускать сценарии, написанные на других языках, в среде JVM. Примерами таких языков могут служить Groovy, JRuby, JavaScript и т.д.

Платформа Spring также поддерживает выполнение динамических сценариев в Spring-приложении, или же можно определить Spring-бин, написанный на языке динамических сценариев, и внедрить его в нужные компоненты JavaBean. В число языков динамических сценариев, поддерживаемых Spring, входят Groovy, JRuby и BeanShell. Более детально поддержка динамических сценариев в Spring рассматривается в главе 22.

Упрощенная обработка исключений

Существует область, в которой Spring действительно помогает сократить объем повторяющегося рутинного кода, и это — обработка исключений. Основой философии Spring в этом отношении является тот факт, что проверяемых исключений в Java слишком много, и данная платформа не заставляет перехватывать исключения, после которых вряд ли будет возможность провести восстановление; мы полностью согласны с такой точкой зрения.

В действительности многие платформы спроектированы так, чтобы уменьшить потребность в написании кода для обработки проверяемых исключений. Тем не менее, во многих таких платформах принято придерживаться проверяемых исключений, но искусственно снижать степень детализации в иерархии классов исключений. В отношении Spring вы отметите интересный момент: вследствие удобства, которое получают разработчики благодаря использованию непроверяемых исключений, иерархия классов исключений является в высшей степени детальной.

Повсеместно в этой книге вы будете встречать примеры, в которых механизмы обработки исключений Spring могут сокращать объем необходимого кода и, в то же время, совершенствовать возможности идентификации, классификации и диагностики ошибок внутри приложения.

Проект Spring

Одной из самых подкупающих особенностей проекта Spring является уровень активности, наблюдаемой в настоящее время в сообществе, и степень взаимодействия между другими проектами, такими как CGLIB, Apache Geronimo и AspectJ. Наиболее расхваливаемое преимущество открытого кода связано с тем, что если даже проект будет неожиданно свернут, у вас останется исходный код; но давайте посмотрим правде в глаза — вряд ли вы захотите остаться с кодовой базой размера Spring и впоследствии поддерживать и улучшать ее. По этой причине отрадно знать, что дела идут хорошо, а сообщество Spring сохраняет высокую активность.

Происхождение Spring

Истоки Spring можно проследить вплоть до книги *Expert One-to-One J2EE Design and Development*, написанной Родом Джонсоном (Wrox, 2002 г.). В этой книге Род представляет собственную платформу под названием Interface 21 Framework, которую он разработал для использования в своих приложениях. После выпуска в мир открытого кода эта платформа сформировала основу Spring Framework, которую мы знаем сейчас.

Платформа Spring быстро прошла через стадии бета-тестирования и предвыпускной версии, и ее первый официальный выпуск 1.0 был сделан доступным 24 марта 2004 г. С тех пор платформа Spring значительно разрослась и на время написания этой книги она доступна в виде версии Spring Framework 3.1.

Сообщество Spring

Сообщество Spring — одно из лучших сообществ из всех проектов с открытым кодом, с которыми мы встречались. Списки рассылки и форумы всегда активны, а скорость добавления новых средств, как правило, высока. Команда разработчиков действительно сосредоточена на том, чтобы сделать Spring самой успешной платформой для построения Java-приложений, и об этом можно судить по качеству производимого кода. Большая часть непрерывно продолжающейся разработки Spring связана с переделкой существующего кода, чтобы сделать его быстрее, короче и аккуратнее.

Как упоминалось ранее, Spring также получает преимущества от тесных взаимоотношений с другими проектами с открытым кодом, и этот факт чрезвычайно важен, если вы полагаетесь на значительную зависимость от полного дистрибутива Spring.

С точки зрения пользователя, возможно, наилучшими средствами Spring являются великолепная документация и тестовый комплект, входящие в состав дистрибутива. В документации описаны практически все возможности Spring, и это упрощает освоение платформы новыми пользователями. Тестовый комплект Spring является всеобъемлющим — команда разработчиков пишет тесты для любого аспекта. Когда они обнаруживают ошибку, то фиксируют ее, сначала написав тест, который выявляет ошибку, и затем прогнав его.

Что все это означает для вас? Выражаясь просто, это значит, что вы можете быть уверены в высоком качестве Spring Framework, а также в том, что в обозримом будущем команда разработчиков Spring продолжит совершенствовать и без того замечательную платформу.

Spring для Microsoft .NET

Главный проект Spring Framework на 100% основан на Java. Однако из-за успеха Java-версии разработчики из мира .NET почувствовали себя несколько обделенными; по этой причине Марк Поллак и Род Джонсон запустили проект Spring .NET. За исключением Рода, этими двумя проектами занимаются совершенно разные команды разработчиков, так что проект для .NET должен оказывать лишь минимальное влияние на проект Spring на Java. На самом деле мы считаем это очень хорошей новостью. Вопреки распространенному мнению в мире Java, .NET вовсе не является нагромождением ненужного хлама, и мы сами можем подтвердить это, имея за плечами множество успешно поставленных нашим клиентам приложений .NET. Этот проект открывает совершенно новые возможности для взаимодействия, особенно с тех пор, как платформа .NET заняла лидирующие позиции в некоторых областях, таких как метаданные на уровне исходного кода, и должен привести к получению наилучшего продукта на обоих фронтах. Еще один побочный эффект от этого проекта связан с тем, что он делает переход между платформами более простым для разработчиков, поскольку и там, и там можно использовать Spring. Это приобретает даже еще больший вес с учетом того, что другие проекты, вроде Hibernate и MyBATIS, уже имеют свои эквиваленты для .NET. Дополнительные сведения по платформе Spring .NET можно получить на веб-сайте `www.springframework.net`.

Комплект SpringSource Tool Suite / IDE-среда Spring

Для упрощения процесса разработки основанных на Spring приложений в Eclipse (наиболее часто используемая IDE-среда для разработки Java-приложений), в рамках Spring создан проект Spring IDE. Вскоре после этого компания SpringSource, которую Род Джонсон основал после Spring, создала интегрированное инструментальное средство под названием SpringSource Tool Suite (STS). Хотя это был платный продукт, теперь данный инструмент доступен бесплатно. Он интегрирован в Eclipse IDE, Spring IDE, Mylyn (среда разработки в Eclipse, основанная на задачах), Maven for Eclipse, AspectJ Development Tool и множество других полезных подключаемых модулей Eclipse в едином пакете. В каждой новой версии появляются дополнительные средства, такие как поддержка языка сценариев Groovy, поддержка Spring Roo и поддержка SpringSource tcServer (сервер приложений с платной поддержкой, предлагаемый SpringSource и построенный поверх сервера Tomcat). Исходный код примеров для каждой главы, а также код примера приложения, будут разработаны в STS, поэтому вам понадобится загрузить STS и импортировать проекты. (Комплект STS подробно рассматривается в приложении A.) В случае, если вы хотите получить дополнительные сведения об использовании STS для разработки Spring-приложений немедленно, обратитесь к приложению A прямо сейчас.

Проект Spring Security

Проект Spring Security (`http://static.springsource.org/spring-security/site/index.html`), ранее известный как Acegi Security System for Spring — это еще один важный проект в рамках Spring. Проект Spring Security предоставляет всеобъемлющую поддержку безопасности на уровне веб-приложения и отдельных методов. Он тесно интегрирован с платформой Spring Framework и другими распространенными механизмами аутентификации, такими как базовая аутентификация HTTP, вход с помощью форм, сертификаты X.509, продукт SSO (например, SiteMinder) и т.д. Проект Spring Security предлагает управление доступом на основе ролей к ресурсам приложения, а в приложениях с более сложными требованиями к безопасности (скажем, разделение данных)

поддерживается ACL (Access Control List — список управления доступом). Тем не менее, Spring Security в основном применяется в защищенных веб-приложениях, о чем речь пойдет в главе 17.

Проекты Spring Batch и Spring Integration

Не стоит и говорить, что выполнение пакетных заданий и интеграция являются распространенными сценариями использования в приложениях. Чтобы удовлетворить этим требованиям и упростить решение для разработчиков в этих областях, в рамках Spring созданы проекты Spring Batch и Spring Integration. Проект Spring Batch предоставляет общую инфраструктуру и разнообразные политики для реализации пакетных заданий, сокращая объем рутинного кодирования. За счет реализации шаблона EIP (Enterprise Integration Pattern — шаблон интеграции корпоративных приложений), проект Spring Integration помогает упростить интеграцию Spring-приложений с внешними системами. Соответствующие детали ищите в главе 20.

Другие проекты

Выше мы описали ключевые модули Spring и некоторые важные проекты внутри Spring, но существует и множество других проектов, предназначенных для удовлетворения других требований сообщества. В качестве примеров можно упомянуть Spring BlazeDS для интеграции с Flex, Spring Mobile, Spring Dynamic Modules, Spring Social, Spring AMQP и т.д. Эти проекты в настоящей книге не рассматриваются. За дополнительной информацией обращайтесь на веб-сайт SpringSource (`www.springsource.org/projects`).

Альтернативы Spring

С учетом предыдущих замечаний относительно количества проектов с открытым кодом, не должен вызывать удивление тот факт, что Spring не является единственной платформой, которая предлагает средства внедрения зависимостей (DI) или полноценных сквозных решений для построения приложений. На самом деле проектов настолько много, что их сложно даже упомянуть вскользь. Соблюдая дух открытости, мы включили здесь краткое описание нескольких таких платформ, но, по нашему убеждению, ни одна из этих платформ не предлагает столь всеобъемлющее решение, как это делает Spring.

JBoss Seam Framework

Основанная Гэвином Кингом (создателем библиотеки Hibernate ORM), Seam Framework (`www.seamframework.org`) представляет собой еще одну зрелую основанную на DI платформу; она содержит уровни из пользовательских интерфейсов веб-приложений (JSF), уровень бизнес-логики (EJB 3) и JPA для поддержки постоянства. Как можно заметить, основное отличие между Seam и Spring состоит в том, что платформа Seam построена полностью на стандартах JEE. Платформа JBoss также способствует развитию идей Seam в JCP и появлению JSR-299 (Contexts and Dependency Injection for the Java EE Platform — Контексты и внедрение зависимостей для платформы Java EE).

Google Guice

Следующая популярная платформа DI — это Google Guice (`http://code.google.com/p/google-guice`). Поддерживаемая поисковым гигантом Google, Guice представляет собой облегченную платформу, которая сосредоточена на обеспечении DI для управления конфигурацией приложений. Она также была опорной реализацией спецификации JSR-330 (Dependency Injection for Java — Внедрение зависимостей для Java).

PicoContainer

PicoContainer (`www.picocontainer.org`) — это исключительно малый (около 300 Кбайт) контейнер DI, который позволяет использовать DI для своего приложения, не вводя никаких зависимостей, кроме PicoContainer. Поскольку PicoContainer является всего лишь контейнером DI, вы можете обнаружить, что по мере роста приложения приходится вводить другую платформу, такую как Spring; в этом случае лучше применять Spring с самого начала. Однако если все, что нужно — это небольшой контейнер DI, то PicoContainer будет хорошим вариантом. Тем не менее, поскольку пакеты контейнера DI отделены от остальной части платформы, имеет смысл воспользоваться ими, обеспечив более высокую гибкость в будущем.

Контейнер JEE 6

Как упоминалось ранее, концепция DI получила широкое распространение и также реализована JCP. В результате в версии JEE 6 стек технологий (EJB 3.1, JPA 2.1, JSR-299, JSR-330 и т.д.) был существенно модернизирован для принятия DI и упрощения разработки приложений JEE. Это значит, что при разработке приложения для серверов приложений, совместимых с JEE 6, стандартные приемы DI можно использовать на всех уровнях. На время написания этой книги популярными серверами приложений, совместимыми с JEE 6, являлись JBoss AS 7, Oracle Glassfish 3.1 и WebSphere 8.

Резюме

В этой главе мы представили высокоуровневый взгляд на платформу Spring Framework с обсуждением всех ее основных возможностей, а также указали на те части книги, где эти возможности рассматриваются более подробно. После чтения этой главы вы должны получить определенное представление о том, что Spring может делать; все, что нам осталось — узнать, *как* это реализовать. Итак, самое время продолжить.

В следующей главе мы посмотрим, что необходимо для построения и запуска базового Spring-приложения. Мы покажем, как получить Spring Framework, а также обсудим варианты упаковки, тестовый комплект и документацию. Кроме того, в главе 2 будет представлен некоторый базовый код Spring, включая вездесущий пример “Hello World!”, реализованный в контексте DI.

ГЛАВА 2

Начало работы

Часто самым сложным аспектом при освоении нового инструмента разработки является выяснение того, с чего следует начать. Как правило, проблема усугубляется, если инструмент предлагает настолько много опций, как это делает Spring. К счастью, начало работы с платформой Spring на самом деле не так трудно, если знать, где и что искать в первую очередь. В этой главе приведены все основные сведения о том, как выйти на старт. В частности, будут рассмотрены следующие вопросы.

- *Получение Spring.* Первый логический шаг заключается в получении и в сборке JAR-файлов Spring. Если вы хотите быстро запустить стандартный дистрибутив Spring, просто загрузите последний дистрибутив Spring из веб-сайта Spring по адресу `www.springframework.org`. Однако если вы хотите находиться на переднем крае разработки с помощью Spring, поищите последнюю версию исходного кода в репозитории GitHub для Spring. В случае использования при разработке приложений среды Maven можете просто добавить зависимости для Spring в файл `pom.xml` проекта (project object model — объектная модель проекта), и Maven загрузит необходимые JAR-файлы самостоятельно. Более подробно это рассматривается в разделе “Модули Spring в репозитории Maven” далее в главе.
- *Варианты упаковки Spring.* Упаковка Spring является модульной; она позволяет выбирать компоненты, которые должны использоваться в приложении, и при распространении готового приложения включать только эти компоненты.
- *Зависимости Spring.* Полный дистрибутив Spring включает обширный набор зависимостей, но во многих случаях необходимо только некоторое подмножество этих зависимостей. В этом разделе мы посмотрим, какие средства Spring каких зависимостей требуют; эти сведения помогут свести размер окончательного приложения к минимуму.
- *Примеры Spring.* Платформа Spring поступает с широким выбором примеров приложений, которые могут стать идеальной опорной точкой для построения собственных приложений. В этом разделе мы заглянем внутрь примеров приложений, чтобы вы смогли оценить объем доступного кода примеров. Вместе с примером приложения, которое будет разрабатываться на протяжении этой книги, вы получаете более чем достаточную кодовую базу, на основе которой можно приступать к построению собственных приложений.
- *Тестовый комплект и документация.* Одним из предметов особой гордости членов сообщества Spring является всеобъемлющий тестовый комплект и набор документации. Тестирование занимает значительную часть работы команды. Используя Clover (`www.atlassian.com/software/clover`), команда осуществляет активный мониторинг процента покрытия тестами и постоянно стремиться увеличить этот

процент. Набор документации, входящий в состав стандартного дистрибутива, также великолепен.

- *Пример приложения "Hello World!" в Spring.* Как бы то ни было, мы считаем, что наилучшим способом начала работы с любым новым инструментов для программирования является написание простого кода. Мы собираемся рассмотреть некоторые простые примеры, включая основанную на DI реализацию хорошо известного примера "Hello World!". Не переживайте, если сходу не поняли все примеры кода; далее в этой книге будут даны исчерпывающие пояснения.

Если вы уже знакомы с основами Spring Framework, можете переходить прямо к главе 3, где обсуждается пример приложения, которое будет строиться на протяжении данной книги. Однако, даже зная основы Spring, вы наверняка найдете интересную информацию в этом разделе, особенно касающуюся упаковки и зависимостей.

Получение Spring Framework

Перед тем, как можно будет приступить к написанию кода для Spring, необходимо получить код самой платформы Spring. Для получения кода есть несколько вариантов: можно загрузить упакованный дистрибутив из веб-сайта Spring или же извлечь код из репозитория Spring GitHub. Еще один вариант предусматривает использование инструмента для управления зависимостями приложений, такого как Maven либо Ivy; понадобится объявить зависимость в конфигурационном файле и позволить инструменту получить требуемые библиотеки самостоятельно.

Загрузка стандартного дистрибутива

Разработка платформы Spring хранится в центре загрузки SpringSource по адресу `www.springsource.org/download`. Для загрузки последнего выпуска Spring (версии 3.1 на время написания этой книги) зайдите на указанную страницу. В центре загрузки также доступны промежуточные экранные снимки для планируемых выпусков и предыдущих версий.

Начиная с версии 3.0, выпуски Spring Framework поступают в двух разновидностях: одна с включенной документацией и одна — без. До версии 3.0 платформа Spring использовалась для предоставления еще одного пакета, который включал все библиотеки третьих сторон (например, `commons-logging`, `hibernate` и т.д.). Однако в настоящее время Spring полагается на инструменты управления зависимостями, подобные Maven и Ivy, для выражения зависимости от библиотек третьих сторон в каждом своем модуле. Поэтому, когда вы объявите свой проект зависимым от любого модуля Spring (скажем, `spring-context`), все обязательные зависимости будут включены автоматически. Более подробно это обсуждается позже в данной главе.

Извлечение Spring из GitHub

В случае если вы хотите получить доступ к новым средствам даже до того, как они будут отражены в экранных снимках, можете извлечь исходный код непосредственно из репозитория GitHub в SpringSource.

Для получения последней версии кода Spring сначала установите средство GitHub, которое можно загрузить из веб-сайта `http://git-scm.com/`, после чего откройте инструмент Git Bash и запустите следующую команду:

```
git clone git://github.com/SpringSource/spring-framework.git
```

Упаковка Spring

После загрузки и извлечения пакета в папке `dist` вы найдете целый список JAR-файлов, которые представляют каждый модуль Spring. После того, как вы поймете назначение каждого модуля, вы сможете выбрать модули, необходимые для проекта, и включить их в свой код. На рис. 2.1 показано содержимое папки `dist` после извлечения загруженного пакета Spring Framework.

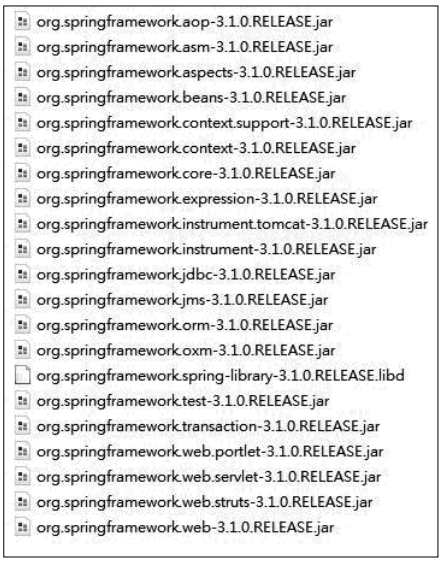


Рис. 2.1. Библиотеки Spring Framework после извлечения

Модули Spring

В версии 3.1 платформа Spring поступает с 20 модулями, упакованными в 20 файлов JAR. Эти JAR-файлы и соответствующие им модули описаны в табл. 2.1. Действительный формат JAR-файлов выглядит, например, как `org.springframework.aop-3.1.0.`*`ВЫПУСК`*`.jar`, но для простоты мы включили в табл. 2.1 только часть, специфичную для модуля (вроде `aop`).

Таблица 2.1. Модули Spring

JAR-файл	Описание
`aop`	Этот модуль содержит все классы, необходимые для использования в приложении средств аспектно-ориентированного программирования (АОП) из Spring. Этот JAR-файл также должен быть включен в приложение, если планируется работать с другими средствами Spring, которые используют АОП, такие как декларативное управление транзакциями. Кроме того, в этот модуль упакованы и классы, поддерживающие интеграцию с AspectJ
`asm`	ASM (`asm.ow2.org`) — это инфраструктура для манипуляций байт-кодом Java. Платформа Spring зависит от этой библиотеки при анализе байт-кода бинов Spring, их динамической модификации и генерировании нового байт-кода во время выполнения
`aspects`	Этот модуль содержит все классы, предназначенные для расширенной интеграции с библиотекой АОП в AspectJ. Например, если вы используете Java-классы для своей конфигурации Spring и нуждаетесь в управлении транзакциями с помощью аннотаций в стиле AspectJ, то вам определенно понадобится этот модуль
`beans`	Этот модуль содержит все классы, предназначенные для поддержки манипуляций бинами Spring. Большинство классов здесь поддерживают реализацию фабрики бинов в Spring. Например, в этот модуль упакованы классы, требуемые для анализа XML-файла конфигурации Spring и Java-аннотаций
`context`	Этот модуль содержит классы, которые предоставляют многие расширения для ядра Spring. Вы увидите, что все классы должны использовать средство `ApplicationContext` из Spring (описанное в главе 5), а также классы для интеграции с EJB, Java Naming and Directory Interface (JNDI) и Java Management Extensions (JMX). В этом модуле также содержатся удаленные классы Spring, классы для интеграции с языками динамических сценариев (например, JRuby, Groovy, BeanShell), классы API-интерфейса Beans Validation (JSR-303), классы для планирования и выполнения задач и т.д.

Продолжение табл. 2.1

JAR-файл	Описание
`context.support`	Этот модуль содержит дополнительные расширения для модуля `spring-context`. На стороне пользовательского интерфейса имеются классы для поддержки электронной почты и интеграции с механизмами шаблонов, такими как Velocity, FreeMarker и JasperReports. Кроме того, здесь же упакованы классы для интеграции с различными библиотеками выполнения и планирования задач, в числе которых CommonJ и Quartz
`core`	Это ключевой модуль, необходимый для каждого приложения Spring. В данном JAR-файле находятся все классы, разделяемые всеми другими модулями Spring, например, классы для доступа к файлам конфигурации. Также в этом JAR-файле вы найдете набор исключительно полезных служебных классов, которые используются по всей кодовой базе Spring и которые можно применять в собственных приложениях
`expression`	Этот модуль содержит все классы поддержки для языка SpEL (Spring Expression Language — язык выражений Spring)
`instrument`	Этот модуль включает агент инструментальных средств Spring для начальной загрузки виртуальной машины Java (Java Virtual Machine — JVM). Данный JAR-файл обязателен для использования в приложении Spring привязки к AspectJ во время загрузки
`instrument.tomcat`	Этот модуль включает агент инструментальных средств Spring для начальной загрузки JVM на сервере Tomcat
`jdbc`	Этот модуль включает все классы, предназначенные для поддержки JDBC. Данный модуль необходим для всех приложений, которым требуется доступ к базам данных. В модуле упакованы классы для поддержки источников данных, типов данных JDBC, шаблонов JDBC, встроенных подключений JDBC и т.д.
`jms`	Этот модуль включает все классы, предназначенные для поддержки JMS
`orm`	Этот модуль расширяет стандартный набор средств JDBC, имеющийся в Spring, поддержкой популярных инструментов ORM, включая Hibernate, iBATIS (но не MyBatis), JDO и JPA. Многие из классов в данном JAR-файле зависят от классов в `spring-jdbc.jar`, поэтому вы определенно должны включить его в свое приложение
`oxm`	Этот модуль предоставляет поддержку OXM (отображение объектов на XML). В данный модуль упакованы классы, предназначенные для абстрагирования маршализации и демаршализации XML, а также для поддержки популярных инструментов, подобных Castor, JAXB, XMLBeans, XStream и т.д.
`web.struts`	Этот модуль включает все классы, предназначенные для интеграции между Spring и веб-платформой Struts
`test`	Как упоминалось ранее, Spring предоставляет набор пробных (mock) классов для помощи в тестировании приложения. Многие из этих пробных классов используются внутри тестового комплекта Spring, так что они хорошо проверены и существенно упростят тестирование разрабатываемых приложений. Безусловно, в модульных тестах для веб-приложений применяются пробные классы `HttpServletRequest` и `HttpServletResponse`. С другой стороны, Spring обеспечивает тесную интеграцию с платформой модульного тестирования JUnit, и в этом модуле предоставлены многие классы, которые поддерживают разработку тестовых сценариев JUnit; например, класс `SpringJUnit4ClassRunner` предлагает простой способ загрузки средства `ApplicationContext` в среду модульного тестирования

Окончание табл. 2.1

JAR-файл	Описание
transaction	Этот модуль предоставляет все классы, предназначенные для поддержки инфраструктуры транзакций Spring. Здесь вы найдете классы из уровня абстракции транзакций, поддерживающие Java Transaction API (JTA) и интеграцию с серверами приложений от ведущих производителей
web	Этот модуль содержит ключевые классы для использования Spring в веб-приложениях, включая классы для автоматической загрузки средства ApplicationContext, классы для поддержки загрузки файлов и набор полезных классов для выполнения повторяющихся задач, таких как извлечение целочисленных значений из строки запроса
web.servlet	Этот модуль содержит все классы для собственной инфраструктуры MVC в Spring. В случае использования отдельной инфраструктуры MVC классы из этого JAR-файла не нужны. Spring MVC рассматривается более подробно в главах 17 и 18
web.portlet	Этот модуль предоставляет поддержку для использования Spring MVC при разработке портлетов для развертывания в среде сервера портала

Выбор модулей для разрабатываемого приложения

Без IDE-среды, подобной Eclipse, либо инструмента управления зависимостями, такого как Maven или Ivy, выбор модулей для использования в приложении может быть несколько затруднен. Например, если требуется только фабрика бинов Spring и поддержка DI, нужны такие модули, как spring-core, spring-beans, spring-context, spring-aop и spring-asm. Если необходима поддержка веб-приложений Spring, понадобится также добавить spring-web. Для интеграции с платформой Struts потребуется spring-struts и т.д.

Однако когда применяется IDE-среда, в особенности SpringSource Tool Suite (STS), которая будет считаться IDE-средой по умолчанию во всех примерах настоящей книги, управление/визуализация упомянутых выше зависимостей становится намного проще. В среде STS доступна возможность создать шаблонный проект Spring и произвести выбор из некоторого числа шаблонов проектов варианта, подходящего для разрабатываемого приложения. Шаблонный проект Spring использует Maven для управления зависимостями, и среда STS также связана с m2e — проектом подключаемого модуля для интеграции с Maven. В случае выбора этого шаблона проекта Spring создаст проект с объявленными соответствующими зависимостями. Благодаря поддержке транзитивных зависимостей Maven, все обязательные библиотеки третьих сторон также будут включены автоматически.

На рис. 2.2 показано окно STS с простым служебным проектом Spring. Экран получен с применением файла pom.xml (Maven Project Object Model — объектная модель проекта Maven) проекта и средства просмотра иерархии зависимостей из подключаемого модуля m2e, которое отображает все зависимости, требующиеся проекту. На диаграмме иерархии зависимостей можно заметить, что проект зависит от модуля spring-context, который, в свою очередь, требует модулей spring-aop, spring-beans, spring-core, spring-expression и spring-asm. Кроме того, spring-aop дополнительно зависит от aopalliance, в то время как spring-core зависит от commons-logging. По умолчанию проект Spring использует log4j для регистрации в журнале. Наконец, имеются и зависимости для тестирования (spring-test, junit).

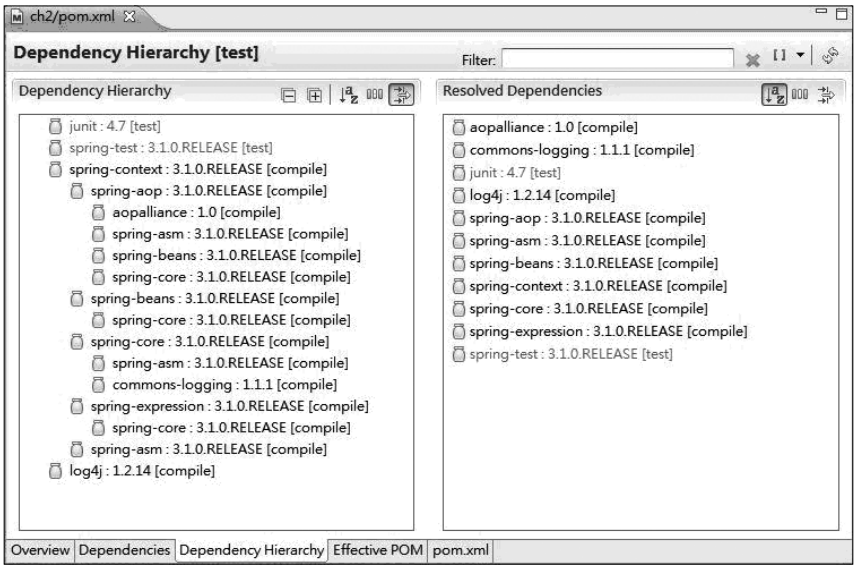


Рис. 2.2. Иерархия зависимостей Spring для простого служебного проекта Spring, который использует фабрику бинов Spring и средства DI

Модули Spring в репозитории Maven

Кроме загрузки из Интернета, библиотеками Spring можно управлять с помощью инструмента управления зависимостями приложения, такого как Ivy и Maven. В этом разделе мы посмотрим на модули Spring в репозитории Maven.

Созданные Apache Software Foundation, инструменты Maven (`http://maven.apache.org`) стали одними из наиболее популярных средств, используемых при управлении зависимостями для Java-приложений, от сред с открытым кодом до корпоративных сред.

Maven представляет собой мощный инструмент сборки, упаковки и управления зависимостями для приложений. Он поддерживает полный цикл сборки приложения, начиная с обработки ресурсов и компиляции и заканчивая тестированием и упаковкой. Вдобавок существует большое количество подключаемых модулей Maven для решения разнообразных задач, вроде обновления баз данных и развертывания упакованного приложения на специфическом сервере (например, Tomcat, JBoss, WebSphere и т.д.).

Практически все проекты с открытым кодом поддерживают распространение своих библиотек через репозиторий Maven. Наиболее популярным является репозиторий Maven Central, обслуживаемый веб-сервером Apache; на веб-сайте Maven Central (`http://search.maven.org`) можно выполнять поиск на предмет существования артефакта и получать связанную с ним информацию. После загрузки и установки Maven на машине разработки автоматически появляется доступ к репозиторию Maven Central. Ряд других сообществ открытого кода (например, JBoss, SpringSource и т.д.) также предоставляют своим пользователям доступ к собственным репозиториям Maven. Однако для доступа к таким репозиториям понадобится добавить необходимый репозиторий в свой файл настроек Maven.

Детальное обсуждение Maven выходит за рамки этой книги, но вы всегда можете обратиться к онлайновой документации или книгам, которые предоставят подробные сведения о Maven. Тем не менее, по причине столь широкого распространения Maven, полезно упомянуть структуру упаковки Spring в репозитории Maven.

С каждым артефактом Maven связан идентификатор группы, идентификатор артефакта, тип упаковки и версия. Например, для `log4j` идентификатором группы является `log4j`, идентификатором артефакта — `log4j`, а типом упаковки — `jar`. Далее определяется версия. Скажем, для версии 1.2.16 файл артефакта будет иметь имя `log4j-1.2.16.jar` и располагаться в папке для конкретного идентификатора группы, идентификатора артефакта и версии.

Подобно другим библиотекам с открытым кодом, артефакты для Spring в Maven могут быть найдены на веб-сайте Apache Maven Central. Однако SpringSource также обслуживает собственный репозиторий Maven и предоставляет библиотеки Spring в форме EBR (Enterprise Bundle Repository — корпоративный репозиторий пакетов), которые являются совместимыми с OSGi. Соглашения об именовании Spring EBR отличаются от таковых в Maven Central.

Чтобы снять причины для замешательства, полезно упомянуть об отличиях в назначении имен артефактам Spring в репозитории Maven Central и собственном репозитории Maven, поскольку команда разработчиков должна придерживаться стандартов одного из них. В общем случае предпочтительнее использовать артефакты из Maven Central. Однако если планируется развертывать приложение в контейнере OSGi (например, в динамических модулях Spring), то имеет смысл применять Spring EBR.

В табл. 2.2 показаны имена артефактов Spring, соответственно, в Maven Central и SpringSource EBR. В обоих репозиториях идентификаторы группы совпадают, а отличаются лишь идентификаторы артефакта. Как и в табл. 2.1, в имена JAR-файлов включена только часть, касающаяся модуля.

Анализ зависимостей Spring

В Spring имеются многочисленные зависимости от библиотек третьих сторон. При сборке Spring из исходного кода необходимы все эти зависимости. Тем не менее, во время выполнения, скорее всего, потребуется только подмножество зависимостей; включая только необходимые зависимости, можно действительно минимизировать размер дистрибутива созданного приложения.

Из-за большого количества зависимостей Spring группирует их вместе, тем самым упрощая работу с ними. Эти группы описаны в табл. 2.3; в ней также перечислены JAR-файлы для каждой группы и указаны используемые зависимости.

В табл. 2.3 отражены только базовые и общие зависимости для модулей платформы Spring. В разрабатываемом приложении могут потребоваться дополнительные зависимости. Например, для поддержки Spring XML (т.е. модуля `oxm`), если в качестве лежащей в основе библиотеки отображения объектов на XML используется Castor, то понадобится добавить библиотеку Castor. В случае применения XStream нужно добавить библиотеку XStream и т.д. По мере перехода к последующим главам и обсуждения каждой отдельной темы, будут упоминаться дополнительные библиотеки третьих сторон, которые при этом используются. Поэтому воспринимайте табл. 2.3 как общий обзор, а не как полный справочник.

Таблица 2.2. Модули Spring

JAR-файл модуля Spring	Идентификатор группы	Идентификатор артефакта (Maven Central)	Идентификатор артефакта (Spring EBR)
aop	org.springframework	spring-aop	org.springframework.aop
asm	org.springframework	spring-asm	org.springframework.asm
aspects	org.springframework	spring-aspects	org.springframework.aspects
beans	org.springframework	spring-beans	org.springframework.beans
context	org.springframework	spring-context	org.springframework.context
context.support	org.springframework	spring-context-support	org.springframework.context.support
core	org.springframework	spring-core	org.springframework.core
expression	org.springframework	spring-expression	org.springframework.expression
instrument	org.springframework	spring-instrument	org.springframework.instrument
instrument.tomcat	org.springframework	spring-instrument-tomcat	org.springframework.instrument.tomcat
jdbc	org.springframework	spring-jdbc	org.springframework.jdbc
jms	org.springframework	spring-jms	org.springframework.jms
orm	org.springframework	spring-orm	org.springframework.orm
oxm	org.springframework	spring-oxm	org.springframework.oxm
web.struts	org.springframework	spring-struts	org.springframework.struts
test	org.springframework	spring-test	org.springframework.test
transaction	org.springframework	spring-tx	org.springframework.transaction
web	org.springframework	spring-web	org.springframework.web
web.servlet	org.springframework	spring-webmvc	org.springframework.web.servlet
web.portlet	org.springframework	spring-webmvc-portlet	org.springframework.web.portlet

Таблица 2.3. Зависимости от библиотек третьих сторон в Spring Framework

Группа зависимостей	JAR-файлы	Описание
aopalliance	aopalliance-1.0.jar	Альянс АОП (http://aopalliance.sourceforge.net) — это объединенная координация совместной деятельности над множеством проектов с открытым кодом с целью предоставления стандартного набора интерфейсов для АОП в Java. Реализация АОП в Spring основана на стандартных API-интерфейсах Альянса АОП. Этот JAR-файл необходим, только если планируется использовать какие-либо средства АОП или основанные на АОП в Spring
aspectj	aspectjweaver-1.6.8.jar	Платформа Spring тесно интегрирована с AspectJ для поддержки более мощных средств АОП. Всякий раз, когда используется AspectJ, нужна эта библиотека
caucho	com.springsource.com.caucho-3.2.1.jar	Удаленное взаимодействие Spring предоставляет поддержку для широкого спектра различных протоколов, включая протоколы Burlap и Hessian от Caucho Technology. JAR-файл в этой группе нужен только в случае использования в приложении соответствующих протоколов
cglib	cglib-2.2.jar	CGLIB — это библиотека для генерации кода, от которой зависит модуль aop из Spring. Библиотека CGLIB способна генерировать прокси для классов и для интерфейсов Java
dom4j	dom4j-1.6.1.jar	Зависимость dom4j нужна при использовании Hibernate, так что если для поддержки ORM в приложении планируется применять Hibernate, то понадобится включить этот JAR-файл
easymock	easymock-2.5.1.jar	EasyMock используется в тестовом комплекте Spring, поэтому данный JAR-файл должен включаться только при сборке и запуске упомянутого тестового комплекта; при распространении своего приложения этот JAR-файл не нужен
freemarker	freemarker-2.3.15.jar	Spring предоставляет классы-оболочки для механизма шаблонов FreeMarker и также обеспечивает поддержку использования шаблонов FreeMarker как представлений в разрабатываемых веб-приложениях. Данный JAR-файл необходим всякий раз, когда применяется FreeMarker
hibernate	hibernate-core-3.6.10.Final.jar, hibernate-commons-annotations-3.2.0.Final.jar, hibernate-entitymanager-3.6.10.Final.jar, hibernate-jpa-2.0-api-1.0.1.Final.jar, hibernate-validator-4.1.0.GA.jar	Эти JAR-файлы требуются в случае использования интеграции с Hibernate и классов ее поддержки в Spring. Если применяется другой инструмент ORM, такой как MyBatis, можете не включать указанные JAR-файлы в приложение. Если используется Hibernate, в приложение понадобится также включить файл javassist.jar. В случае применения JPA и Hibernate в качестве поставщиков постоянства требуется диспетчер сущностей и библиотека JPA. Если нужен JSR-303 Beans Validation API или JPA 2.0, то также понадобится средство проверки достоверности Hibernate

Продолжение табл. 2.3

Группа зависимостей	JAR-файлы	Описание
javassist	javassist-3.12.0.GA.jar	Это библиотека для манипуляции байт-кодом
mybatis	mybatis-3.0.6.jar, mybatis-spring-1.0.2.jar	Эти JAR-файлы требуются в случае использования MyBatis
itext	itextpdf-5.1.2.jar	Spring использует iText для обеспечения поддержки формата PDF на веб-уровне. Этот JAR-файл следует включать, только если веб-приложения должны генерировать вывод в формате PDF
jee	activation-1.1.0.jar, connector-api-1.5.jar, jaxws-api-2.2.6.jar, ejb-api-3.0.jar, jms-1.1.jar, jstl-1.2.jar, jta-1.1.jar, mail-1.4.4.jar, servlet-2.5.jar, xml-apis-2.0.2.jar	Как видите, список различных JAR-файлов, относящихся к JEE, довольно обширен. Файлы activation.jar и mail.jar нужны, если планируется работать с реализацией JavaMail почтовой поддержки Spring. Файл connector-api.jar необходим для использования JCA Connector для Hibernate, ejb.jar — поддержки Spring EJB, а jms.jar — поддержки Spring JMS. Если в веб-приложениях должна применяться поддержка Spring JSTL, то потребуются файлы servlet.jar и jstl.jar. Файл jaxws-api.jar нужен для поддержки JAX-WS при удаленном взаимодействии Spring, а jta.jar используется для поддержки транзакций JTA
apache-commons	commons-attributes-api-2.1.jar, commons-attributes-compiler-2.2.jar, commons-beanutils-1.8.3.jar, commons-collections-3.2.0.jar, commons-dbcp-1.4.jar, commons-digester-2.1.jar, commons-discovery-0.5.jar, commons-fileupload-1.2.0.jar, commons-lang-2.5.jar, commons-logging-1.1.1.jar, commons-pool-1.5.4.jar, httpclient-4.1.1.jar	Многие компоненты проекта Apache Commons применяются платформой Spring. Файл commons-attributes-api.jar необходим, если планируется использовать в приложении метаданные на уровне исходного кода, а также нужен JAR-файл компилятора для компиляции атрибутов в приложение. JAR-файлы BeanUtils, Collections, Digester и Discovery JAR применяются Struts, а Hibernate вдобавок использует Collections. DBCP используется поддержкой JDBC в Spring, когда приложение пользуется пулом подключений DBCP, при этом организация пула требуется рядом примеров приложений. JAR-файл FileUpload необходим, если в веб-приложениях планируется применение соответствующей оболочки Spring для обработки загрузок файлов. Наконец, регистрация в журнале используется в Spring повсеместно, поэтому ее нужно включать в каждое приложение, основанное на Spring
junit	junit-4.9.jar	JUnit вообще не требуется во время выполнения, а только при сборке и запуске тестового комплекта
log4j	log4j-1.2.14.jar	Это необходимо, если планируется использовать Spring для конфигурирования регистрации в журнале с помощью log4j

Окончание табл. 2.3

Группа зависимостей	JAR-файлы	Описание
poi	poi-3.6.jar	Это добавляет к инфраструктуре Spring MVC поддержку вывода Microsoft Excel
quartz	quartz-1.6.2.jar	Это используется для поддержки основанного на Quartz планирования в Spring
struts	struts-1.2.9.jar	JAR-файл Struts требуется каждый раз, когда для построения веб-приложений планируется использовать Struts в сочетании с платформой Spring
velocity	velocity-1.5.0.jar	Spring предоставляет классы-оболочки для механизма шаблонов Velocity, чтобы сделать его поддерживающим DI и также сократить объем кода, необходимого для использования Velocity в разрабатываемом приложении. В дополнение к этому, Spring предоставляет классы, предназначенные для поддержки использования Velocity в качестве поставщика представлений на веб-уровне. В случае применения любого из этих средств в дистрибутив приложения понадобится включить JAR-файл Velocity

Как видите, зависимости Spring весьма разнообразны, и для большинства приложений требуется только часть полного набора зависимостей. Полезно потратить время на определение, какие точно зависимости нужны, и добавить в приложение только их. При таком подходе можно обеспечить минимальный размер приложения, что особенно полезно тем, кто часто должен выполнять развертывание в удаленные местоположения. Сохранение размера приложения на минимально возможном уровне очень важно, когда планируется распространять приложение через веб пользователям, у которых могут быть медленные соединения с Интернетом.

Примеры приложений

Существует одна область, в которой многие продукты с открытым кодом, в том числе и коммерческие, терпят неудачу — обеспечение хорошо документированного кода примеров, который помог бы новым пользователям быстро приступить к работе. К счастью, Spring поставляется с полным набором полезных примеров приложений, которые демонстрируют широкий выбор функциональных средств, доступных в Spring. Команда разработчиков рассматривает примеры приложений как полноценные части платформы, которые должны постоянно совершенствоваться. По этой причине вы обнаружите, что после полной отработки тестового комплекта примеры являются идеальным местом для изучения новых возможностей.

Получение исходного кода примеров Spring

Весь исходный код примеров Spring (за исключением проекта Spring MVC Showcase) хранится в репозитории SpringSource SVN. Чтобы получить исходный код, введите следующую команду, находясь в любом пустом каталоге:

```
svn co https://src.springframework.org/svn/spring-samples/
```

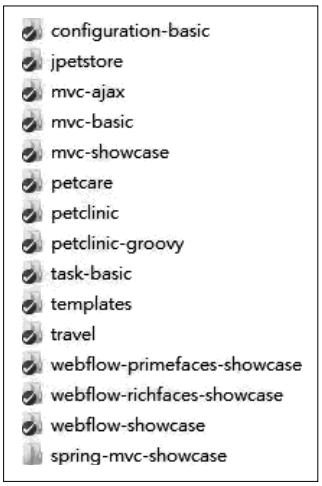


Рис. 2.3. Примеры приложений Spring

На рис. 2.3 показана папка примеров приложений после ее извлечения по указанной ссылке.

Приложение Petclinic

Приложение Petclinic (расположенное в папке `petclinic`) — это интересный пример, который построен для демонстрации поддержки доступа к данным в Spring. В нем вы найдете веб-приложение для запроса и обновления базы данных в вымышленном офисе ветеринарной клиники. С этим приложением связан один момент: оно поставляется с набором взаимозаменяемых реализаций DAO, которые подчеркивают простоту развязывания приложения и логики доступа к данным в случае использования Spring.

Реализация DAO для Hibernate действительно демонстрирует поддержку Hibernate в Spring за счет реализации каждого из восьми методов DAO с помощью одной или двух строк кода. Реализация JDBC не менее интересна. Первым делом, она показывает, что использование `SimpleJdbcTemplate` и `SimpleJdbcInsert` позволяет избежать необходимости в избыточном коде для общих операций JDBC. С другой стороны, весьма познавательно узнать, как обрабатывается доступ к данным в более объектно-ориентированной манере. Применяя интерфейс `RowMapper` и его многочисленные реализации, результирующий набор запроса может быть напрямую отображен на объекты значений и возвращен вызывающему коду.

Этот проект также содержит весьма показательный пример построения веб-приложения с использованием поддержки MVC в Spring, так что если вы планируете применять Spring MVC в своих приложениях, внимательно изучите этот пример.

Поддержка JDBC рассматривается в главе 8, Hibernate — в главе 9, а Spring MVC — в главе 17.

Приложение Petclinic Groovy

Пример приложения Petclinic Groovy (расположенный в папке `petclinic-groovy`) демонстрирует реализацию того же самого приложения Petclinic, но с использованием Groovy и Grails. Groovy — это динамический язык сценариев, выполняющихся на машине JVM, а Grails — платформа быстрой разработки приложений, которая позволяет разрабатывать веб-приложения на основе Groovy. Все контроллеры на уровне Spring MVC написаны на языке Groovy.

Обсуждение Grails выходит за рамки этой книги, но интеграция Spring и Groovy будет рассматриваться в главе 22.

Приложение jPetStore

Пример приложения jPetStore (расположенный в папке `jpetstore`) основан на примере jPetStore, созданном Клинтоном Бегином для iBATIS. Это один из крупных примеров приложений. Он содержит полный уровень DAO, созданный с использованием Spring и iBATIS, с реализациями для Oracle, Microsoft SQL Server, MySQL и PostgreSQL. Уровень бизнес-логики полностью управляется Spring, а в связке с уровнем DAO он представляет хороший пример транзакций, управляемых Spring.

В состав этого приложения также входит показательный пример применения сразу Spring MVC и Struts. Это приложение также показывает, как использовать удаленное взаимодействие Spring с помощью JAXRPC.

Spring MVC рассматривается в главе 17, iBATIS (MyBatis) — в главе 11, а удаленное взаимодействие Spring — в главе 16.

Приложение Spring Configuration Basic

Проект базовой конфигурации Spring Configuration Basic (расположенный в папке `configuration-basic`) демонстрирует определение конфигурации приложения Spring с использованием Java-классов, что является новой возможностью, начиная с версии Spring 3.0. В этом проекте XML-файлы отсутствуют; вся конфигурация, относящаяся к Spring, объявлена в Java-классе (`AppConfig.java`) с разнообразными аннотациями.

Конфигурирование Spring с помощью Java-классов рассматривается в главе 5.

Приложение Spring Task and Scheduling

Проект работы с базовыми задачами Spring Task and Scheduling (расположенный в папке `task-basic`) демонстрирует использование поддержки запуска и планирования задач в Spring.

Поддержка планирования в Spring рассматривается в главе 15.

Приложение Spring MVC Showcase

Проект Spring MVC Showcase (расположенный в папке `spring-mvc-showcase`) показывает возможности инфраструктуры Spring MVC Web Framework с помощью набора небольших простых примеров. Приложение Spring MVC Showcase построено на Spring MVC 3 и демонстрирует функционирование основных средств Spring MVC 3, как описано ниже.

- Использование аннотации `@Controller` для объявления контроллеров Spring MVC.
- Отображение запроса получение данных запроса и генерация ответа.
- Преобразователи сообщений и визуализация представлений.
- Формы, проверка достоверности типов, загрузка файлов и обработка исключений.

Инфраструктура Spring MVC детально рассматривается в главе 17.

На заметку! В отличие от других примеров проектов Spring, проект Spring MVC Showcase хранится в GitHub (`www.github.com`) — очень популярном веб-сайте для совместно разрабатываемого кода. Чтобы извлечь исходный код из GitHub, сначала понадобится получить инструмент Git (`http://git-scm.com`). После установки Git введите следующую команду для копирования репозитория проекта из GitHub на вашу рабочую станцию: `git clone https://github.com/SpringSource/spring-mvc-showcase`.

Приложения Spring MVC Basic и Spring MVC Ajax

Помимо проекта Spring MVC Showcase, в Spring предоставляются два других проекта, демонстрирующих разнообразные сценарии использования Spring MVC 3. Проект Spring MVC Basic (расположенный в папке `mvc-basic`) — это очень простой проект Spring MVC с одним контроллером. Проект Spring MVC Ajax (расположенный в папке `mvc-ajax`) показывает, как использовать Spring MVC для построения веб-приложений с поддержкой Ajax. Серверная сторона построена с помощью Spring MVC, и за счет

применения ее встроенной поддержки RESTful-WS запросы отображаются, а данные возвращаются в формате JSON. На клиентской стороне для взаимодействия с инфраструктурой Spring MVC с целью обеспечения развитого пользовательского интерфейса применяется jQuery — популярная библиотека JavaScript с поддержкой Ajax.

Приложение Spring Petcare

Приложение Spring Petcare (расположенное в папке `petcare`) является еще одним интересным проектом. Это полномасштабное веб-приложение, которое демонстрирует множество разных средств Spring Framework и других проектов Spring. Со стороны веб оно использует Spring MVC вместе с проектом Spring Security для защиты необходимых ресурсов. Для представлений оно интегрируется с Tiles с целью поддержки шаблонов. Здесь также можно найти поддержку электронной почты.

Следующим интересным средством является интеграция с проектом Spring Integration для широковещательной рассылки сообщений приложения. Проект Spring под названием Spring Roo используется для генерации базы компонентов JavaBean на основе схемы базы данных серверной стороны.

Защита ресурсов веб-приложения с применением Spring Security рассматривается в главе 17. Высокоуровневый обзор по использованию Spring Framework с другими проектами Spring, включая Spring Integration и Spring Roo, представлен в главе 20.

Примеры приложений для Spring Web Flow

Возможности Spring и Spring Web Flow демонстрируются во множестве примеров проектов. Приложение Travel (расположенное в папке `travel`) относится к Spring Framework 3 с версией Spring Web Flow 2.1.

Проект Spring Web Flow интегрируется со многими технологиями представлений, одной из которых является Java Server Faces (JSF). Spring Faces — это модуль в Spring Web Flow, обеспечивающий тесную интеграцию с JSF. Для демонстрации этого средства предусмотрено два примера приложения. Приложение Webflow Primefaces Showcase (расположенное в папке `webflow-primefaces-showcase`) показывает интеграцию с PrimeFaces, а приложение Webflow Richfaces Showcase (расположенное в папке `webflow-richfaces-showcase`) — интеграцию с JBoss Richfaces. И PrimeFaces, и JBoss RichFaces являются популярными библиотеками JSF, причем их последние версии соответствуют стандартам JSF 2.0.

Разработка веб-приложений с использованием Spring Web Flow на примере с PrimeFaces рассматривается в главе 18.

Документация Spring

Один из аспектов Spring, который делает ее настолько полезной платформой для разработчиков, строящих реальные приложения — это изобилие аккуратно написанной документации. Команда, отвечающая за подготовку документации, в каждом выпуске Spring Framework прилагает все усилия, чтобы документация была исчерпывающей и согласованной с командой разработки. Это значит, что каждое функциональное средство Spring не только полностью документировано в JavaDoc, но также описано в справочном руководстве, включаемом в каждый выпуск. Если вы еще не знакомы с документацией Spring JavaDoc и справочным руководством, самое время сделать это сейчас. Эта книга не задумана в качестве замены любого из упомянутых ресурсов; напротив, она должна служить дополняющим справочным пособием, демонстрирующим построение приложений Spring с самого начала.

Помещение Spring в приложение “Hello World!”

К этому моменту вы уже должны понимать, что Spring является серьезным, хорошо поддерживаемым проектом, обладающим всем необходимым для того, чтобы считаться великолепным инструментом для разработки приложений. Тем не менее, мы пока еще не показывали никакого кода. Наверняка вы с нетерпением ждете того момента, когда платформа Spring будет продемонстрирована в действии, и поскольку это нельзя сделать, не прибегая к коду, мы займемся этим прямо сейчас. Не переживайте, если не полностью поймете код, приводимый в этом разделе; по мере продвижения по материалу настоящей книги будут представлены дополнительные подробности.

Построение примера приложения “Hello World!”

Вы определенно должны быть знакомы с традиционным примером “Hello World!”, но на всякий случай в листинге 2.1 приведен код Java-версии во всей своей красе.

Листинг 2.1. Типовой пример “Hello World!”

```
package com.apress.prospring3.ch2;

public class HelloWorld {

   public static void main(String[] args) {
      System.out.println("Hello World!");
   }
}
```

В рамках примеров приложений данный пример очень прост — он делает свою работу, но не является особо расширяемым. Что если необходимо изменить сообщение? А что если нужно выводить сообщение разными способами, возможно, в `stderr` вместо `stdout`, или заключить его в HTML-дескрипторы, а не представлять как простой текст?

Мы собираемся переопределить требования для этого примера приложения, указав, что приложение должно поддерживать простой и гибкий механизм изменения выводимого сообщения, а также несложный способ изменения поведения вывода из приложения. В базовом примере “Hello World!” оба изменения можно произвести быстро и легко, внеся соответствующие правки в код. Однако более крупное приложение требует времени на перекомпиляцию и повторное тестирование. Поэтому лучшее решение заключается в вынесении содержимого сообщения наружу и чтении его во время выполнения, возможно из аргументов командной строки, как показано в листинге 2.2.

Листинг 2.2. Использование аргументов командной строки в “Hello World!”

```
package com.apress.prospring3.ch2;

public class HelloWorldWithCommandLine {

   public static void main(String[] args) {
      if(args.length > 0) {
         System.out.println(args[0]);
      } else {
         System.out.println("Hello World!");
      }
   }
}
```

Этот пример делает то, что нам было нужно — теперь можно изменить сообщение, не меняя код. Однако в данном приложении по-прежнему остается проблема: компонент, отвечающий за визуализацию сообщения, также отвечает и за получение этого сообщения. Изменение способы получения сообщения означает изменение кода визуализации. Добавьте к этому тот факт, что мы не можем легко изменить код визуализации; для этого понадобится изменять класс, который запускает приложение.

Забегая несколько вперед, отметим, что лучшее решение предусматривает рефакторинг логики визуализации и получения сообщений в отдельные компоненты. Вдобавок, чтобы сделать приложение действительно гибким, эти компоненты должны реализовать интерфейсы, с использованием которых определяются взаимные зависимости между компонентами и запускающим классом.

Выполняя рефакторинг логики получения сообщений, мы можем определить простой интерфейс `MessageProvider` с единственным методом `getMessage()`, как показано в листинге 2.3.

Листинг 2.3. Интерфейс `MessageProvider`

```
package com.apress.prospring3.ch2;
public interface MessageProvider {
   public String getMessage();
}
```

Как видно в листинге 2.4, интерфейс `MessageRenderer` реализован всеми компонентами, которые могут визуализировать сообщения.

Листинг 2.4. Интерфейс `MessageRenderer`

```
package com.apress.prospring3.ch2;
public interface MessageRenderer {
   public void render();
   public void setMessageProvider(MessageProvider provider);
   public MessageProvider getMessageProvider();
}
```

Легко заметить, что интерфейс `MessageRenderer` имеет метод `render()` и свойство `MessageProvider` в стиле JavaBean. Любые реализации `MessageRenderer` отделены от получения сообщений и делегируют это интерфейсу `MessageProvider`, с которым они поставляются. Здесь `MessageProvider` представляет собой зависимость от `MessageRenderer`. Создать простые реализации этих интерфейсов довольно легко (листинг 2.5).

Листинг 2.5. Класс `HelloWorldMessageProvider`

```
package com.apress.prospring3.ch2;
public class HelloWorldMessageProvider implements MessageProvider {
   public String getMessage() {
      return "Hello World!";
   }
}
```

В листинге 2.5 мы создаем простой интерфейс `MessageProvider`, который всегда возвращает в качестве сообщения строку “Hello World!”. Класс `StandardOutMessageRenderer` (показанный в листинге 2.6) также прост.

Листинг 2.6. Класс StandardOutMessageRenderer

```
package com.apress.prospring3.ch2;
public class StandardOutMessageRenderer implements MessageRenderer {
   private MessageProvider messageProvider = null;
   public void render() {
      if (messageProvider == null) {
         throw new RuntimeException(
            "You must set the property messageProvider of class:"
            + StandardOutMessageRenderer.class.getName());
            // Должно быть установлено свойство messageProvider класса
      }
      System.out.println(messageProvider.getMessage());
   }
   public void setMessageProvider(MessageProvider provider) {
      this.messageProvider = provider;
   }
   public MessageProvider getMessageProvider() {
      return this.messageProvider;
   }
}
```

Теперь осталось только переписать метод `main()` основного класса, как показано в листинге 2.7.

Листинг 2.7. Приложение “Hello World!” после рефакторинга

```
package com.apress.prospring3.ch2;
public class HelloWorldDecoupled {
   public static void main(String[] args) {
      MessageRenderer mr = new StandardOutMessageRenderer();
      MessageProvider mp = new HelloWorldMessageProvider();
      mr.setMessageProvider(mp);
      mr.render();
   }
}
```

Приведенный код довольно прост: мы создаем экземпляры `HelloWorldMessageProvider` и `StandardOutMessageRenderer`, хотя объявленными типами являются, соответственно, `MessageProvider` и `MessageRenderer`.

Причина в том, что нам нужно взаимодействовать в коде только с методами, предоставленными этими интерфейсами, а интерфейсы уже реализованы классами `HelloWorldMessageProvider` и `StandardOutMessageRenderer`. Затем мы передаем `MessageProvider` в `MessageRenderer` и вызываем `MessageRenderer.render()`. Вывод из этого примера показан на рис. 2.4.

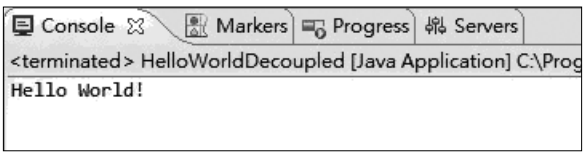


Рис. 2.4. Вывод из `HelloWorldDecoupled`

Теперь этот пример приобрел вид, похожий на требуемый нами, но в нем осталась еще одна небольшая проблема. Изменение реализации либо интерфейса `MessageRenderer`, либо интерфейса `MessageProvider` означает модификацию кода. Чтобы обойти данную проблему, мы можем создать простой фабричный класс, который будет читать имена классов реализации из файла свойств и создавать их экземпляры в интересах приложения (листинг 2.8).

Листинг 2.8. Класс `MessageSupportFactory`

```
package com.apress.prospring3.ch2;

import java.io.FileInputStream;
import java.util.Properties;

public class MessageSupportFactory {

   private static MessageSupportFactory instance = null;
   private Properties props = null;
   private MessageRenderer renderer = null;
   private MessageProvider provider = null;

   private MessageSupportFactory() {
      props = new Properties();

      try {
         props.load(new FileInputStream("ch2/src/conf/msf.properties"));

         // Получить классы реализации.
         String rendererClass = props.getProperty("renderer.class");
         String providerClass = props.getProperty("provider.class");
         renderer =
            (MessageRenderer) Class.forName(rendererClass).newInstance();
         provider =
            (MessageProvider) Class.forName(providerClass).newInstance();
      } catch (Exception ex) {
         ex.printStackTrace();
      }
   }

   static {
      instance = new MessageSupportFactory();
   }

   public static MessageSupportFactory getInstance() {
      return instance;
   }

   public MessageRenderer getMessageRenderer() {
      return renderer;
   }

   public MessageProvider getMessageProvider() {
      return provider;
   }
}
```

Приведенная реализация элементарна и несколько наивна, обработка ошибок упрощена, а имя конфигурационного файла жестко закодировано, но мы уже имеем значимый объем кода. Конфигурационный файл для этого класса предельно прост:

```
renderer.class=com.apress.prospring3.ch2.StandardOutMessageRenderer
provider.class=com.apress.prospring3.ch2.HelloWorldMessageProvider
```

Осталось еще внести простое изменение в метод `main()`, как показано в листинге 2.9.

Листинг 2.9. Использование `MessageSupportFactory`

```
package com.apress.prospring3.ch2;

public class HelloWorldDecoupledWithFactory {

  public static void main(String[] args) {
    MessageRenderer mr = MessageSupportFactory.getInstance().getMessageRenderer();
    MessageProvider mp = MessageSupportFactory.getInstance().getMessageProvider();
    mr.setMessageProvider(mp);
    mr.render();
  }
}
```

Перед тем, как перейти к введению Spring в это приложение, давайте вспомним, что было сделано. Начав с простого приложения “Hello World!”, мы определили два дополнительных требования, которым приложение должно удовлетворять. Первое требование: изменение сообщения должно осуществляться просто. Второе требование: изменение механизма визуализации также должно быть простым. Для удовлетворения этих требований мы ввели два интерфейса: `MessageProvider` и `MessageRenderer`. При получении сообщения с целью визуализации интерфейс `MessageRenderer` полагается на реализацию интерфейса `MessageProvider`. Наконец, мы добавили простой фабричный класс для извлечения имен классов реализации и создания их экземпляров для приложения.

Создание проекта Spring в STS

Перед обсуждением рефакторинга приложения “Hello World!” с учетом Spring давайте посмотрим, как использовать STS для разработки основанных на Spring приложений. STS может сохранить немало времени при создании проектов Spring различных типов (например, простых приложений Spring DI, веб-приложений, приложений JPA и т.д.), и STS создаст проект со всеми объявленными обязательными зависимостями. Итак, давайте первым делом загрузим и установим STS и создадим проект для рефакторинга приложения “Hello World!”.

Для начала загрузите IDE-среду STS из веб-сайта SpringSource (`www.springsource.com/downloads/sts`), выберите среду разработки и установите ее. На момент написания этой книги текущей версией STS была 2.8.1. С помощью этой версии разрабатывались все примеры в книге, а также пример приложения.

После установки запустите STS и создайте новый шаблонный проект Spring. В качестве шаблона выберите Simple Spring Utility Project (Простой служебный проект Spring), как показано на рис. 2.5.

Впоследствии STS запросит базовую информацию о проекте, такую как базовое имя пакета и т.п. (рис. 2.6).

Затем STS создаст проект с включенной поддержкой зависимостей Maven. Обязательные зависимости (`spring-context`, `spring-test`, `log4j` и т.д.) будут объявлены и включены автоматически. Включать библиотеки Spring вручную для проекта не понадобится.

Теперь все готово к рефакторингу ранее показанного приложения “Hello World!” с применением Spring.

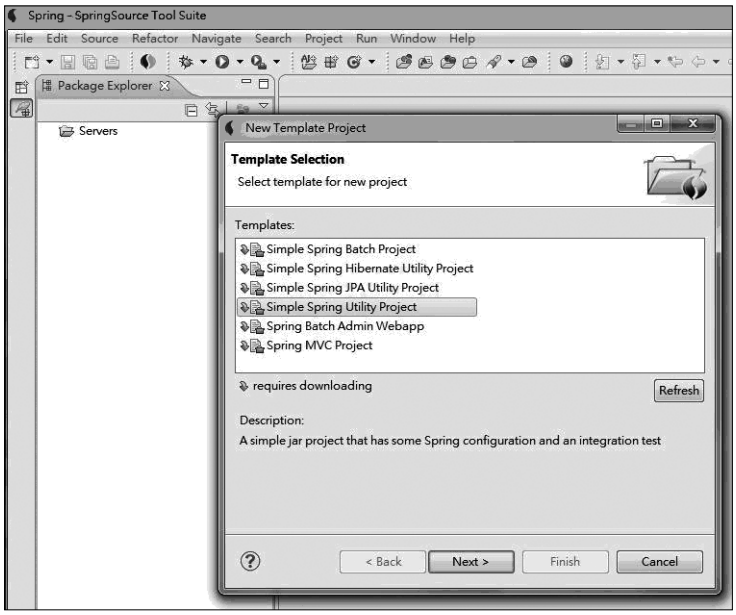


Рис. 2.5. Создание простого служебного проекта Spring в STS

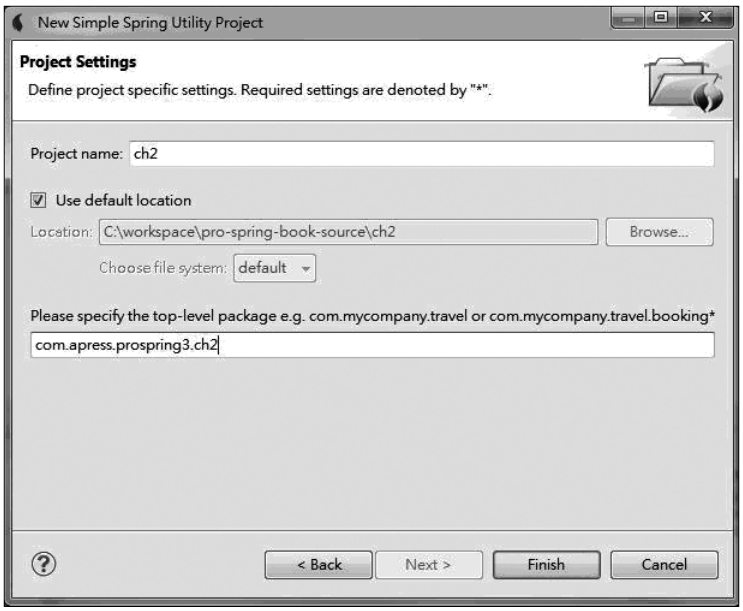


Рис. 2.6. Определение базовых атрибутов для проекта Spring

Рефакторинг с помощью Spring

Приведенный ранее финальный пример соответствует целям, намеченным для нашего примера приложения, но с ним по-прежнему связаны проблемы. Первая проблема состоит в том, что мы должны написать немалый объем связующего кода для сборки вместе всего приложения, в то же время сохраняя компоненты слабо связанными. Вторая проблема заключается в том, что мы все еще должны предоставлять реализацию `MessageRenderer` с интерфейсом `MessageProvider` вручную. Используя Spring, мы можем решить обе эти проблемы.

Для решения проблемы с большим количеством связующего кода мы можем полностью удалить класс `MessageSupportFactory` из приложения и заменить его интерфейсом Spring по имени `ApplicationContext`. Не переживайте особо по поводу этого интерфейса; пока вполне достаточно знать, что он используется Spring для сохранения всей информации о среде, относящейся к приложению, которое управляется Spring. Данный интерфейс расширяет другой интерфейс, `ListableBeanFactory`, который действует в качестве поставщика для любого экземпляра бинов, управляемых Spring (листинг 2.10).

Листинг 2.10. Использование интерфейса `ApplicationContext` из Spring

```
package com.apress.prospring3.ch2;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class HelloWorldSpringDI {

   public static void main(String[] args) {

      // Инициализация ApplicationContext.
      ApplicationContext ctx = new ClassPathXmlApplicationContext
         ("META-INF/spring/app-context.xml");

      MessageRenderer mr = ctx.getBean("renderer", MessageRenderer.class);
      mr.render();
   }

}
```

Как показано в листинге 2.10, метод `main()` получает экземпляр `ClassPathXmlApplicationContext` (информация о конфигурации приложения загружается из файла `META-INF/spring/app-context.xml`, расположенного в пути классов проекта), типизированный как `ApplicationContext`, и далее получает экземпляры `MessageRenderer` с использованием метода `ApplicationContext.getBean()`. Сейчас не стоит беспокоиться о методе `getBean()`; достаточно знать, что он читает конфигурацию приложения (в рассматриваемом случае из XML-файла), инициализирует среду `ApplicationContext` из Spring и затем возвращает сконфигурированный экземпляр бина. Этот XML-файл (`app-context.xml`) служит тем же целям, что и аналогичный файл, который применялся для `MessageSupportFactory` (листинг 2.11).

Листинг 2.11. Конфигурация приложения в Spring, представленная с помощью XML

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
   xmlns:p="http://www.springframework.org/schema/p"
     xmlns:context="http://www.springframework.org/schema/context"
```

```
    xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context-3.0.xsd">
  <bean id="provider" class="com.apress.prospring3.ch2.HelloWorldMessageProvider"/>
  <bean id="renderer" class="com.apress.prospring3.ch2.StandardOutMessageRenderer"
       p:messageProvider-ref="provider"/>
</beans>
```

В листинге 2.11 было приведено содержимое типового конфигурационного файла `ApplicationContext` из Spring. Сначала объявляется пространство имен Spring; при этом пространством имен по умолчанию является `beans`, используемое для объявления бинов, которые должны управляться Spring. Затем объявляются требования к зависимостям (в данном примере свойство `messageProvider` бина `renderer` ссылается на бин `provider`) для Spring, чтобы разрешить и внедрить эти зависимости.

Далее мы объявляем бин с идентификатором `"provider"` и соответствующий класс реализации. Когда во время инициализации `ApplicationContext` платформа Spring обнаруживает это определение бина, она создает экземпляр класса и сохраняет его с указанным идентификатором.

Затем объявляется бин “визуализатора” (с идентификатором `"renderer"`) с соответствующим классом реализации. Вспомните, что этот бин полагается на интерфейс `MessageProvider` при получении сообщения для визуализации. Чтобы информировать Spring о требовании DI, мы используем атрибут пространства имен `p`. Атрибут `p:messageProvider-ref="provider"` сообщает Spring, что свойство `messageProvider` бина должно быть внедрено с помощью другого бина. Внедряемый в это свойство бин должен ссылаться на бин с идентификатором `"provider"`. Когда платформа Spring встречает это определение, она создает экземпляр класса, находит в бине свойство по имени `messageProvider` и внедряет его с экземпляром бина, имеющим идентификатор `"provider"`.

Как видите, во время инициализации `ApplicationContext` метод `main()` теперь просто получает бин `MessageRenderer`, используя безопасный к типам метод `getBean()` (с передачей ему идентификатора и ожидаемого возвращаемого типа, которым является интерфейс `MessageRenderer`), и вызывает `render()`; Spring создает реализацию `MessageProvider` и внедряет ее в реализацию `MessageRenderer`. Обратите внимание, что мы не должны вносить какие-либо изменения в классы, которые связаны вместе с использованием Spring. В сущности, эти классы не имеют никакого отношения к платформе Spring вообще и совершенно не зависят от ее существования. Тем не менее, это не всегда так. Ваши классы могут реализовать интерфейсы Spring для взаимодействия с контейнером DI разнообразными путями.

Давайте теперь посмотрим на приложение “Hello World!”, оснащенное Spring DI, в действии. В IDE-среде STS щелкните правой кнопкой мыши на классе `HelloWorldSpringDI` и выберите в контекстном меню пункт Run As (Запустить как), а затем пункт Java Application (Java-приложение). Результаты выполнения будут отображены на вкладке Console (Консоль), как показано на рис. 2.7.

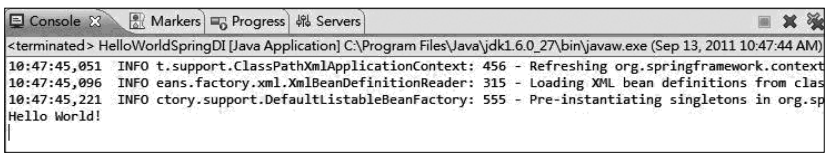


Рис. 2.7. Вывод, полученный в результате выполнения `HelloWorldSpringDI`

Резюме

В этой главе мы представили основные сведения, необходимые для подготовки и запуска Spring. Было показано, как получить дистрибутив выпуска Spring и текущую разрабатываемую версию непосредственно из GitHub. Мы описали, как упакована платформа Spring, и перечислили зависимости, необходимые для каждого ее функционального средства. Располагая этими сведениями, вы можете принимать обоснованные решения о том, какие JAR-файлы Spring нужны для приложения и какие зависимости должны распространяться с приложением. Документация Spring, примеры приложений и тестовый комплект обеспечивают пользователям Spring идеальную основу для начала разработки в Spring, поэтому мы уделили определенное время анализу того, что доступно в рамках дистрибутива Spring. Наконец, был представлен пример применения Spring DI для превращения традиционного приложения “Hello World!” в слабо связанное, расширяемое приложение визуализации сообщений.

Важно понимать, что в этой главе мы только слегка коснулись поверхности Spring DI, да и в целом платформы Spring. В следующей главе мы подробно рассмотрим пример приложения, которое будет разрабатываться в этой книге, уделяя особое внимание тому, как использовать Spring для решения общих задач проектирования и как с помощью Spring сделать приложение более простым и управляемым.

ГЛАВА 3

Пример приложения

Примеры, приводимые в каждой главе этой книги, подогнаны под текущее обсуждение и предназначены для детальной разработки каждого функционального средства в легкой для понимания манере. По большей части, эти примеры дают достаточное объяснение темы, которую они демонстрируют. Тем не менее, примеры выглядят изолированными и не основаны на сценариях реального мира, и это может затруднить понимание того, как различные средства Spring функционируют вместе. Чтобы преодолеть это, мы построим базовое приложение блога под названием SpringBlog, которое отразит большинство тем, обсуждаемых в этой книге, и покажет, как разнообразные средства Spring функционируют совместно друг с другом.

Следует отметить, что это приложение намеренно сделано очень простым, и на самом деле многие из его возможностей были задуманы для отражения отдельных порций функциональности Spring. Несмотря на простоту, приложение SpringBlog демонстрирует, как сконструировано основанное на Spring приложение и каким образом его компоненты связываются вместе.

В этой главе вы получите возможность взглянуть на готовое приложение SpringBlog. Затем мы обсудим средства Spring, используемые для реализации различных частей этого приложения. В этой главе также объясняются некоторые решения, которые были приняты при проектировании приложения SpringBlog, а также соображения, которыми мы руководствовались. По большому счету, эта глава послужит “дорожной картой” для оставшейся части книги, позволяя выделить области, которые важны именно для данного приложения, и сразу же выяснить, где они подробно рассматриваются в книге.

В частности, в этой главе раскрываются следующие две основных темы.

- *Требования для приложения SpringBlog.* В этом разделе мы обсудим требования для приложения SpringBlog и взглянем на завершенный продукт, реализующий эти требования. Мы также объясним причины, почему определенные требования были включены, а другие — проигнорированы во время построения примера приложения.
- *Реализация приложения SpringBlog.* В этом разделе мы рассмотрим на высоком уровне, как описанные в предыдущем разделе требования реализованы с использованием Spring. Здесь мы не будем вдаваться в детали индивидуальных средств Spring; вместо этого мы обсудим средства в общем и укажем главы, в которых содержатся более развернутые описания.

Если вы уже знакомы с проектным решением, положенным в основу приложений Spring, или знаете темы, которые являются наиболее важными в вашем конкретном приложении, можете пропустить данную главу. Если же вы — новичок в Spring, то чтение этой главы даст хорошее представление о том, как выбрать разнообразные компоненты Spring, подходящие для разрабатываемого приложения.

Требования для приложения SpringBlog

При определении требований для приложения SpringBlog нашей главной целью было отражение работы ряда функциональных средств Spring Framework в контексте полноценного приложения. В связи с этим мы включили несколько средств, таких как аудит и фильтрация нежелательного контента, которые обычно отсутствуют в традиционном приложении блога, но, тем не менее, являются полезными функциями, позволяющими подчеркнуть интересные возможности платформы Spring Framework.

В этом разделе приводится полное изложение функциональных средств, включенных в приложение SpringBlog.

Безопасность и аутентификация

Подобно большинству других приложений блогов, приложение SpringBlog поддерживает контрольные меры, связанные с безопасностью, которые не позволяют неавторизованным пользователям создавать и редактировать записи блога.

Как показано на рис. 3.1, приложение SpringBlog предоставляет форму входа для пользователей, которая дает возможность пользователям идентифицировать себя как зарегистрированных пользователей приложения.

Однако для просмотра записей блога вход не требуется. В приложении SpringBlog новому пользователю автоматически назначается идентичность Anonymous (Анонимный). Используя функцию входа, можно сверить свои сведения со списком пользователей в базе данных и назначить себе другую идентичность.

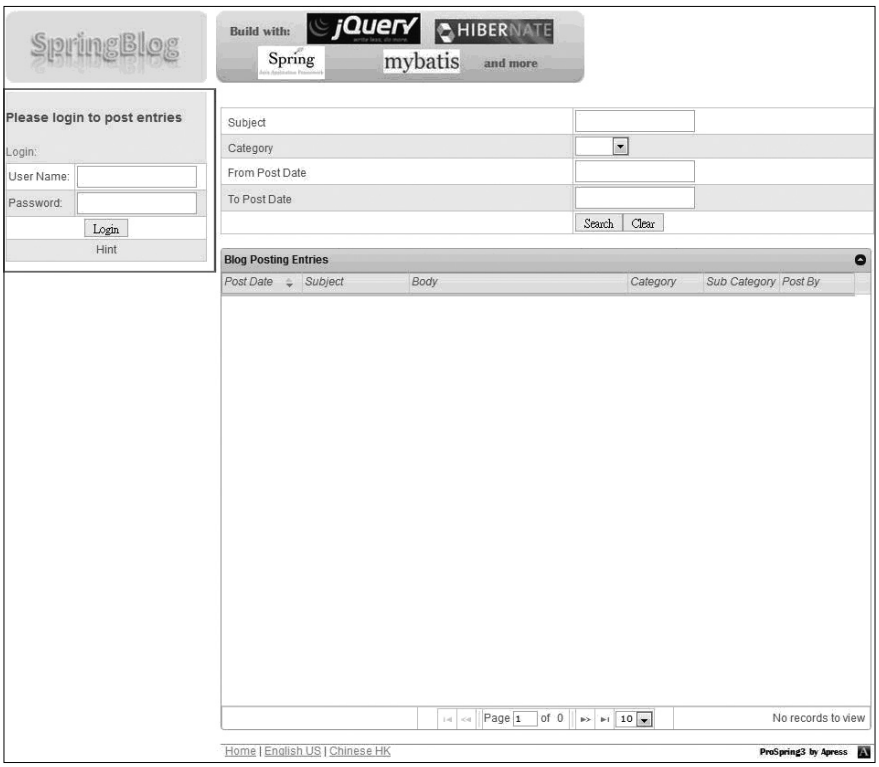


Рис. 3.1. Идентификация пользователей в SpringBlog

Приложение SpringBlog внутренне использует эту идентичность для контроля доступа, а информация о пользователе будет применяться также в процессе аудита.

Ниже перечислены требования к контролю доступа.

- Анонимным пользователям разрешено только просматривать записи блога.
- В рамках приложения предусмотрены две пользовательских роли — обычный пользователь (`ROLE_USER`) и администратор (`ROLE_ADMIN`).
- Пользователи с назначенной ролью `ROLE_USER` могут выполнять следующие действия:
  - помещать в блог новую запись или комментировать существующую запись;
  - редактировать запись блога или созданный ранее комментарий.
- Пользователи с назначенной ролью `ROLE_ADMIN` могут выполнять следующие действия:
  - просматривать данные аудита;
  - осуществлять обслуживание пользователей.

Просмотр записей блога

Очевидным требованием для любой системы блогов является возможность отображения пользователям записей из блогов. Как показано на рис. 3.2, приложение SpringBlog отображает все сообщения, отправленные в блог, на домашней странице в обратном хронологическом порядке. Записи блога представлены в табличном формате.

Рис. 3.2. Просмотр последних записей блога

Пользователи могут конфигурировать количество записей, отображаемых на странице, фильтровать записи блога по темам, категориям и дате создания, а также управлять порядком отображения записей блога.

Щелчок на отдельной записи блога приводит к отображению только этой записи вместе со списком ее комментариев и файлами, присоединенными к ней (рис. 3.3).

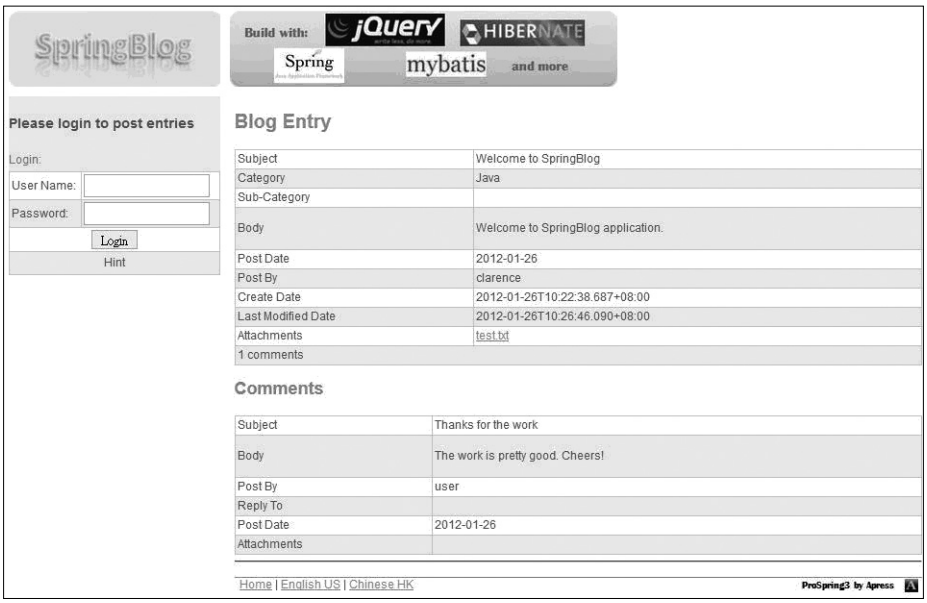


Рис. 3.3. Просмотр записи блога

Помещение записей в блог

Без возможности создания записей в блоге нечего будет и отображать. После входа в приложение зарегистрированный пользователь может создать новую запись в блоге, используя форму ввода на рис. 3.4, доступ к которой осуществляется с помощью ссылки New Blog Posting (Новое сообщение в блоге) на домашней странице.

При создании/редактировании записи блога можно вводить детальные сведения, а также выбирать категорию и подкатегорию записи блога, что позволит посетителям легко отфильтровывать те записи, которые они не желают видеть. Вдобавок во время создания к записи блога будут применены правила проверки достоверности.

После того, как запись блога создана, ее можно отредактировать, щелкнув на ссылке Edit (Редактировать) внутри страницы подробностей записи (запись блога может редактировать только пользователь, который ее создал). Внутренне для создания и для редактирования записи блога приложение SpringBlog использует одну и ту же HTML-форму, но разные методы контроллеров для обработки каждого из этих действий.

Комментирование записи блога

Подобно большинству приложений блогов, SpringBlog позволяет пользователям выражать свое мнение по поводу отдельных записей путем отправки на них комментариев (для комментирования также требуется вход). Пользователи могут отправлять комментарии, как показано на рис. 3.5, используя ссылку Post a comment (Отправить комментарий) на странице подробностей записи.

Рис. 3.4. Создание записи в блоге

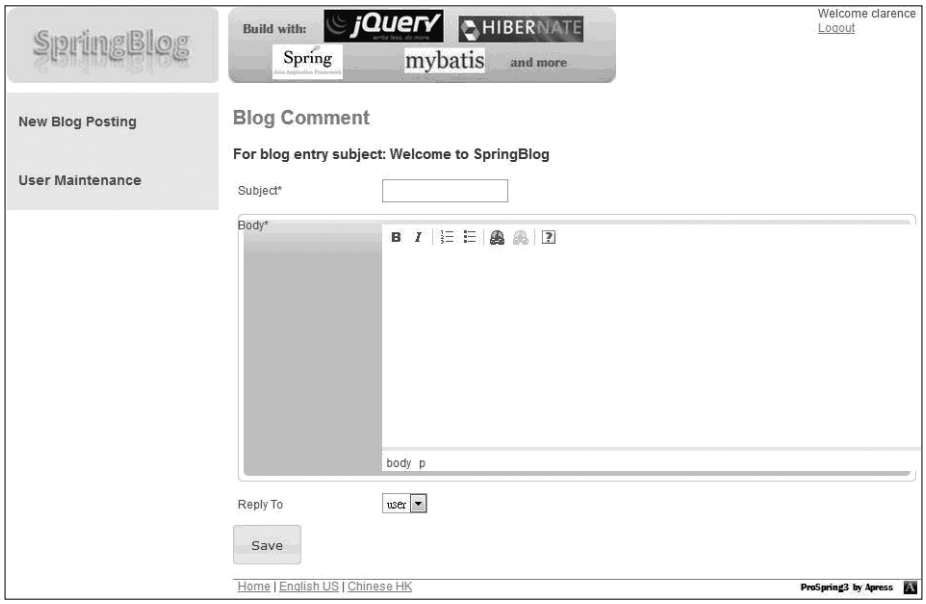


Рис. 3.5. Отправка комментария

Как и в случае с функциональностью создания записи, пользователям также доступна возможность редактирования своих комментариев.

Фильтрация нежелательного контента

Одним из функциональных средств платформы Spring Framework, которое мы хотели подчеркнуть с помощью приложения SpringBlog, является аспектно-ориентированное программирование (АОП), но мы не хотели использовать традиционный пример регистрации, а основанное на АОП управление транзакциями уже встроено в Spring Framework. Хотя в большинстве блогов какая-либо фильтрация нежелательного контента не применяется, мы решили реализовать ее в приложении SpringBlog. Во время проектирования нам показалось, что АОП предоставляет наилучший способ применения этого средства по всему приложению.

Благодаря этой функциональности, при попытке отправить запись вроде показанной на рис. 3.6 в действительности получается сообщение, подобное приведенному на рис. 3.7. Для выполнения фильтрации применяется алгоритм ROT13, который будет подробно рассматриваться в главе 21.

Рис. 3.6. Попытка отправить нежелательный контент

Рис. 3.7. Работа фильтра нежелательного контента

Присоединение файлов к записи блога или к ее комментарию

В отличие от многих приложений блогов, используемых в Интернете, приложение SpringBlog позволяет загружать файлы вместе с записями блога и комментариями к ним. В действительности эта функция порождает исключительно высокий риск для безопасности, однако позволяет продемонстрировать великолепную поддержку загрузки файлов, предлагаемую Spring Framework. На рис. 3.8 показана загрузка файла для существующей записи.

Рис. 3.8. Загрузка файла в приложении SpringBlog

Аудит действий в блоге

Аудит был включен в приложение чисто для целей демонстрации отдельного функционального средства Spring. Из-за необходимости в регистрации всех операций в блоге для целей аудита каждая операция в блоге требует нескольких операций базы данных; в свою очередь, возникает необходимость в транзакции базы данных, которая очевидным образом управляется с помощью средств управления транзакциями, доступных в Spring.

Просмотреть хронологию аудита для сообщения блога (рис. 3.9) можно, щелкнув на ссылке View Audit History (Просмотреть хронологию аудита) внутри страницы подробностей записи блога. Как упоминалось ранее, видеть эту ссылку и, соответственно, просматривать хронологию аудита могут только пользователи с ролью администратора.

RSS-лента

Вместо регулярного посещения сайта некоторые пользователи могут получать RSS-ленты (Really Simple Syndication — действительно простая синдикация) от своих самостоятельно разработанных веб-приложений или через собственный персонализированный портальный сайт (например, от iGoogle). Приложение SpringBlog будет предоставлять записи блога через RESTful-WS. Поскольку это служба, ее потребители могут извлекать записи блога в формате XML или JSON.

Entry Audit

Entry Audit History

Modified Date	Modified By	Post Date	Subject	Body	Category	Sub Category	Post By
2012-01-26 10:54:05	clarence	2012-01-26	Welcome to Spring	Welcome to SpringBlog applicat	Spring		clarence
2012-01-26 10:26:46	clarence	2012-01-26	Welcome to Spring	Welcome to SpringBlog applicat	Java		clarence
2012-01-26 10:22:38	clarence	2012-01-26	Welcome to Spring	Welcome to SpringBlog applicat	Java		clarence
2012-01-26 10:22:38	clarence	2012-01-26	Welcome to Spring	Welcome to SpringBlog applicat	Java		clarence

Рис. 3.9. Просмотр данных аудита

Загрузка блога из XML-файла

Некоторые пользователи могут предпочесть написание своих блогов в автономном режиме с последующей отправкой их в SpringBlog. Приложение SpringBlog также обеспечивает поддержку этого подхода. Пользователи могут записывать свои блоги, сохранять их в XML-файле с предварительно определенными дескрипторами и загружать этот файл в заданную папку на сервере. Приложение SpringBlog будет опрашивать эту папку на регулярной основе, импортируя обнаруженные в ней новые файлы.

Реализация приложения SpringBlog

Одной из ключевых причин использования платформы Spring является тот факт, что она существенно упрощает проектирование и построение приложений с применением традиционных подходов объектно-ориентированного программирования (ООП). Кроме того, зрелый и всеобъемлющий набор средств в последней версии Spring создает мощную основу для разработки приложений JEE. С помощью Spring вы можете проектировать приложения по своему усмотрению, а Spring позаботится о связывании различных компонентов друг с другом. Платформа Spring устраняет необходимость в применении таких шаблонов, как Factory и Singleton, которые препятствуют тестированию и лишь частично решают задачу, для которой они предназначены.

В этом разделе мы представляем высокоуровневый обзор проектных решений и реализации приложения SpringBlog с указанием мест, где каждая тема обсуждается более подробно.

Инструменты разработки и управление зависимостями

Перед тем, как переходить к проектированию, давайте решим, как мы собираемся реализовать приложение SpringBlog. Для разработки основанных на Spring приложений наиболее подходящим инструментом является SpringSource Tool Suite (STS). Комплект STS связывает Eclipse IDE, Spring IDE и множество полезных инструментов (таких как подключаемый модуль Maven для Eclipse, Mylyn, AspectJ Development Tool for

Eclipse и т.д.) в единый пакет. Если на вашей машине разработки уже установлена какая-нибудь версия Eclipse, можете также установить подключаемые модули STS через сайт обновлений STS Eclipse (`www.springsource.com/downloads/sts`). Приложение SpringBlog будет разрабатываться с использованием STS. Дополнительные сведения об STS приведены в приложении A.

Как обсуждалось ранее, платформа Spring Framework состоит из множества модулей, и каждый модуль также зависит от тех или иных Java-библиотек. Хотя вы можете отсортировать требуемые модули Spring и зависимости третьих сторон и включить их в проект вручную, намного проще для этого воспользоваться инструментом управления зависимостями. STS располагает готовой поддержкой Maven — популярного инструмента для управления зависимостями. Приложение SpringBlog будет использовать Maven для управления зависимостями, а также жизненным циклом сборки. Для управления версиями будет применяться Git, и кроме загрузки исходного кода примера приложения со страницы книги, последнюю версию приложения можно также получить из GitHub (`http://github.com`).

Проектное решение для приложения

Проектное решение для приложения SpringBlog очень простое, при этом каждый его уровень определен в терминах интерфейсов, а не конкретных классов. На каждом уровне соответствующие ему интерфейсы определяют только методы, которые этот уровень делает доступными другим уровням, называемым клиентскими уровнями. Параметры для конфигурирования SpringBlog не будут жестко кодироваться в приложении. Вместо этого в классах, реализующих интерфейсы на каждом уровне, объявлены методы конфигурирования, и данные конфигурации внедрены с использованием конфигурационных механизмов, основанных на Spring IoC. Детальное обсуждение интерфейсов приложения SpringBlog, их связывания друг с другом при помощи Spring, а также факторов, влияющих на модульность интерфейсов, приводится в главе 21.

Приложение SpringBlog также содержит базовую объектную модель предметной области (Domain Object Model — DOM), которая инкапсулирует данные и поведение. В главе 12 мы рассмотрим разновидности DOM, которые могут встречаться в рабочих проектах, и обсудим факторы, которые должны приниматься во внимание во время принятия решения о том, инкапсулировать поведение в DOM или же в отдельные служебные объекты.

Управление конфигурацией приложения

В приложении SpringBlog мы будем использовать последнюю версию платформы Spring Framework (на момент написания этой книги — версию 3.1) для конфигурирования DI. Начиная с версии 3.0, платформа Spring поддерживает управление конфигурацией с помощью XML-файлов или Java-аннотаций.

В приложении SpringBlog будет применяться смешанный подход. Во-первых, для упрощения обслуживания все настройки инфраструктуры (например, источника данных, диспетчера транзакций и т.д.) будут определены в разных XML-файлах конфигурации. Для внедряемых бинов и бинов, требующих DI, будут использоваться Java-аннотации для выражения требований DI. Во-вторых, требования поддержки транзакций будут определены также с применением Java-аннотаций. После этого мы полагаемся на средства сканирования и автосвязывания компонентов платформы Spring для сканирования этих классов на предмет соответствующих аннотаций и управления бинами и их требованиями для разнообразных ресурсов.

Вопросы, касающиеся конфигурации Spring, подробно рассматриваются в главах 4 и 5.

Многоуровневая архитектура приложения SpringBlog

Перед тем, как погружаться в детали реализации каждого уровня, давайте в общем рассмотрим уровни, которые собираемся реализовать в SpringBlog. На рис. 3.10 показана многоуровневая архитектура приложения SpringBlog, а ниже приведено описание этих уровней (от серверной части до интерфейсной части).

- *Уровень постоянства.* Этот уровень взаимодействует непосредственно с лежащим в основе хранилищем данных (в этом случае реляционной базой данных) и преобразует извлеченные данные в Java-объекты предметной области для использования уровнем обслуживания. С целью упрощения развертывания в качестве базы данных серверной части будет применяться механизм H2 (`www.h2database.com`). Тем не менее, будет поддерживаться также MySQL, и в примере приложения предоставляются инструкции на случай использования базы данных MySQL.
- *Уровень обслуживания.* Этот уровень является основным в приложении. На данном уровне будет реализована вся бизнес-логика. Любой запрос к службе приложения, независимо от того, по какому каналу он поступает (например, интерфейс браузера, запрос к RESTful-WS, пакетное задание и т.п.), должен проходить через этот уровень для выполнения необходимой обработки с помощью бизнес-логики. Например, здесь происходит проверка достоверности атрибутов бина. Кроме того, для доступа к базе данных этот уровень полагается на уровень постоянства через DI.
- *Уровень пакетных заданий и интеграции.* Этот уровень обеспечивает интеграцию с внешними сторонами. Например, он будет опрашивать XML-файлы (из папки с конфигурируемым местоположением), которые содержат записи блога от пользователей, и импортировать их в приложение SpringBlog с помощью пакетной обработки. Данный уровень взаимодействует с уровнем обслуживания для загрузки записей блога.
- *Уровень презентаций.* Этот уровень предназначен для веб-приложения, которое предоставляет пользовательский интерфейс в рамках SpringBlog. Он также поддерживает RSS-ленту для генерации XML-данных потребителям. Уровень презентаций взаимодействует с уровнем обслуживания для обработки данных и выполнения бизнес-логики.
- *Уровень безопасности.* Этот уровень предохраняет защищенные ресурсы. Например, анонимным пользователям разрешено только просматривать записи блога. Будут предоставлены функции входа, после чего к пользователям будет применен доступ на основе ролей, гарантирующий работу пользователей с функциями согласно назначенным им ролям.

Ознакомившись с многоуровневой архитектурой, теперь давайте посмотрим, как каждый уровень будет реализован в приложении SpringBlog.

Реализация уровня постоянства

Доступ к данным — это тема, близкая сердцам многих разработчиков, и она же является предметом острых дискуссий на форумах разработчиков по всему Интернету. В последнее время в центре внимания мира Java были инструменты объектно-реляционного отображения (object-relational mapping — ORM) как замена JDBC; эти инструменты многие считают чрезмерно запутанными.

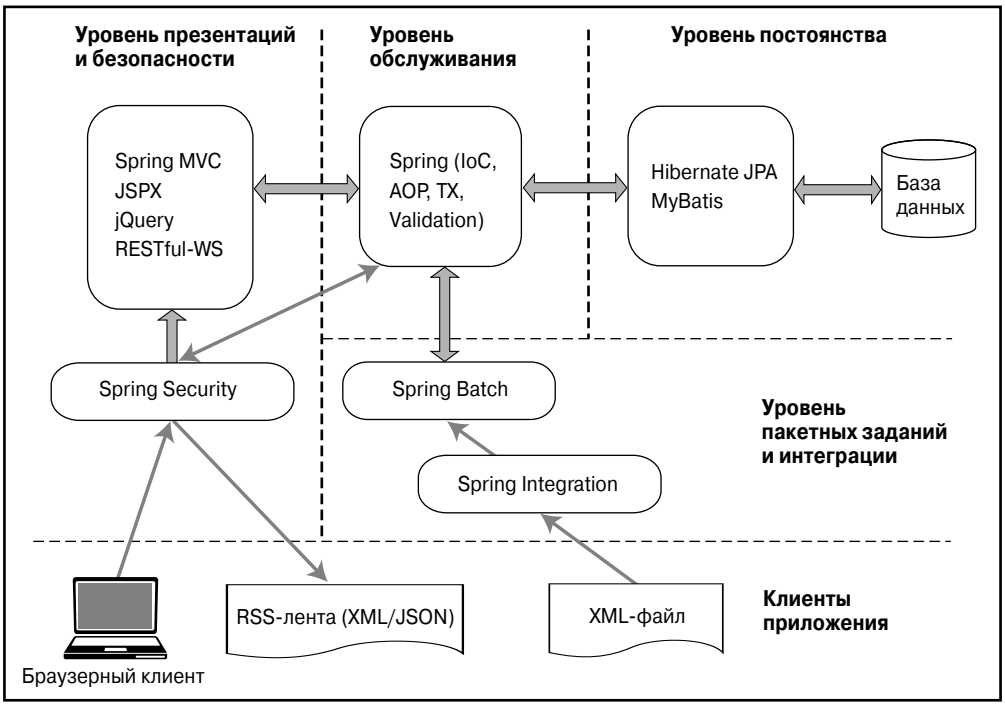


Рис. 3.10. Многоуровневая архитектура приложения SpringBlog

Крупная часть набора функциональных средств Spring основана на поддержке этой платформой технологий доступа к данным. В общем и целом, Spring поддерживает множество различных механизмов доступа к данным: простой JDBC, Hibernate, MyBatis (iBATIS), Java Data Objects (JDO) и JPA.

Посредством приложения SpringBlog мы хотим показать, насколько просто переключаться между реализациями доступа к данным, поэтому построим две разных реализации уровня постоянства — одну с использованием JPA (выбрав Hibernate в качестве лежащего в основе поставщика службы постоянства), а другую с применением MyBatis.

Hibernate, вероятно, является наиболее известным инструментом ORM в мире Java, и он пользовался большим успехом из-за легкости его применения для разработки высокопроизводительной логики постоянства. Несмотря на то что Hibernate имеет довольно простой API-интерфейс, код обработки ошибок, необходимый при использовании Hibernate, достаточно обширен; Spring существенно упрощает этот аспект, сводя обработку ошибок для большинства операций Hibernate до всего лишь одной строки кода. Кроме того, из-за значительного успеха инструментов ORM, подобных Hibernate, и широкого признания разработчиками Hibernate как средства доступа к данным, в JCP формально поместили эту технологию в стек технологий JEE. Начиная с версии EJB 3.0, спецификация бина сущности была заменена Java Persistence API (JPA). В своей текущей версии 2.0 API-интерфейс JPA является частью стандарта JEE 6, и он вобрал в себя многие идеи из популярных инфраструктур ORM. В JPA определен общий шаблон ORM для объектно-реляционного отображения в Java-приложении с целью доступа к реляционным базам данных, а конкретные реализации (называемые *поставщиками постоянства*) оставлены на усмотрение сообществ открытого кода и коммерческих поставщиков. Популярные поставщики постоянства JPA включают Hibernate,

Eclipselink и OpenJPA. Поддержка Hibernate в Spring обсуждается в главе 9, а поддержка JPA в Spring — в главе 10. В этих двух главах показано, как использовать Spring и JPA 2, применяя Hibernate в качестве поставщика службы постоянства, для построения реализации нашего уровня доступа к данным. Кроме того, для Spring с JPA и Hibernate будет объясняться, как использовать Spring Data JPA при реализации регистрации для пользователей, которые создают запись в блоге или комментарий, и как с помощью Hibernate Envers (Entity Versioning System) реализовать функцию аудита в журнале.

Строго говоря, MyBatis — это не инструмент ORM, но инфраструктура DataMapper, которая им является, не снимает всей ответственности с разработчика за создание SQL-операторов, необходимых для отображения Java-объектов на данные в реляционной базе данных. В MyBatis вводится концепция SQL-карты, которая позволяет указать множество SQL-запросов и определить, как эти запросы отображаются на входные и выходные параметры. В некоторых аспектах доступа к данным инструмент MyBatis является достаточно мощным. В главе 11 детально рассматривается MyBatis и показано, как построить вторую реализацию доступа к данным.

Во всех случаях мы используем классы инфраструктуры Spring, предназначенные для каждого инструмента доступа к данным. Эти классы интегрированы с архитектурой транзакций Spring, которая позволяет транзакциям быть управляемыми в независимой от платформы и поставщика ресурсов манере на уровне обслуживания приложения SpringBlog.

Реализация уровня обслуживания

Приложение SpringBlog достаточно простое, и помимо базового сохранения и извлечения данных блога, в системе определено совсем немного бизнес-правил. Тем не менее, в приложении SpringBlog имеются две отдельных бизнес-функции, которые используют два наиболее интересных средства Spring: фильтр нежелательного контента, основанный на АОП, и журнал аудита.

Использование АОП для фильтрации нежелательного контента

На данный момент АОП является горячей темой в мире Java и, как следствие, Java-разработчикам повезло получить в свое распоряжение широкий спектр доступных реализаций АОП. Поддержка АОП в Spring имеет две формы: встроенная в Spring инфраструктура АОП и интеграция с инфраструктурой АОП из AspectJ. Обе формы детально рассматриваются в главах 6 и 7.

В приложении SpringBlog мы хотим предоставить практический пример использования АОП вместо традиционного (и довольно скучного) примера регистрации. Одним из функциональных средств, разрабатываемых в рамках примера приложения, является фильтр нежелательного контента, и во время проектирования стало очевидным, что АОП представляет собой идеальный механизм для реализации этого фильтра. В главе 21 будет показано, как построить фильтр нежелательного контента, и каким образом использовать АОП для выборочного применения этого фильтра к подходящим методам.

Использование поддержки транзакций Spring

Как разработчики, одним из наиболее впечатляющих средств Spring мы считаем поддержку транзакций. Поддержка транзакций в Spring предлагает простой механизм для управления транзакциями между одним или большим числом поставщиков ресурсов, либо программно, либо декларативно. Инфраструктура поддержки транзакций подробно рассматривается в главе 13, при этом особое внимание уделяется транзакциям в базе данных — как локальным, так и распределенным.

Для приложения SpringBlog мы определяем требование о том, что все операции в блоге будут подвержены аудиту и зарегистрированы в базе данных. Чтобы обеспечить возможность отката операции в случае сбоя процесса аудита, мы используем инфраструктуру транзакций Spring для инкапсуляции каждой операции и связанного с ней процесса аудита в одиночную транзакцию. Более подробно это рассматривается в главе 13.

Проверка достоверности бинов

Разработчики больше всего стремятся получить централизованный механизм правил проверки достоверности, при котором все правила проверки применяются и проверяются на данных вне зависимости от того, откуда эти данные поступают (например, когда пользователь вводит информацию через веб-приложение, либо из XML-файла во время обработки пакетного задания). В идеальном случае такие правила проверки достоверности должны определяться только однажды, после чего данные могут проверяться с помощью этих правил на любом уровне, когда в этом возникает необходимость. Для этих целей был создан интерфейс Bean Validation API (JSR-303), который будет использоваться в приложении SpringBlog для централизованной проверки достоверности. Детали поддержки Bean Validation API в Spring обсуждаются в главе 13.

В приложении SpringBlog мы будем применять Hibernate Validator (`http://hibernate.org/subprojects/validator`) — популярную реализацию Bean Validation API.

Две разных реализации уровня обслуживания

Традиционно на уровне доступа к данным разработчики будут реализовать объекты доступа к данным (data access object — DAO), отделяя логику доступа к данным от бизнес-логики. Эти объекты DAO будут внедряться в объекты уровня обслуживания. Цель заключается в том, чтобы упростить переключение с одной реализации доступа к данным на другую. Тем не менее, шаблон DAO вводит еще один уровень между уровнем обслуживания и базой данных серверной части, который оказывается довольно громоздким в большинстве сценариев. Кроме того, с применением стандартов, подобных JPA, контекст постоянства может быть внедрен напрямую в уровень обслуживания для доступа к данным, используя стандартный API-интерфейс, который определен JPA. После этого появляется гибкость благодаря простому переключению одного поставщика постоянства JPA на другой (например, с Hibernate на Eclipselink). Как результат, в большинстве современных приложений шаблон DAO не используется, и вся бизнес-логика и логика доступа к данным инкапсулируется на уровне обслуживания.

В приложении SpringBlog уровень обслуживания включает несколько интерфейсов, отражающих бизнес-службы, которые может предоставить этот уровень. И, как упоминалось ранее, мы обеспечим две разных реализации уровня обслуживания, используя отличающиеся технологии постоянства — одну с JPA и Hibernate, а другую с MyBatis.

Мы также покажем, как выбирать реализацию уровня обслуживания для применения в приложении, с помощью нового средства конфигурирования версии Spring 3.1, которое называется *профилями*. Конфигурация и профили рассматриваются в главе 5.

Реализация уровня пакетных заданий и интеграции

Большинство приложений должны интегрироваться с другими системами для обмена информацией, либо через пакет, либо в режиме реального времени. Два проекта Spring — Spring Batch и Spring Integration — могут работать совместно, обеспечивая мощную и стандартную платформу для реализации пакетных заданий и базового шаблона интеграции корпоративных приложений (Enterprise Integration Pattern — EIP).

Эта два проекта Spring будут использоваться для реализации пакетного задания, отвечающего за импорт записей блога из XML-файла. Для запуска пакетного задания в

случае, когда в специфической папке появляется файл, будет применяться поддержка опроса файлов, доступная в Spring Integration. Краткое введение в Spring Batch и Spring Integration можно найти в главе 20.

Реализация уровня презентаций

Помимо поддержки технологий доступа к данным, платформа Spring также известна своей поддержкой широкого спектра различных инфраструктур и инструментов для веб-приложений. В главах 17 и 18 приводится детальное обсуждение инфраструктуры Spring MVC и ряда общих технологий представлений, поддерживаемых Spring, в том числе Java Server Pages (JSP) и Java Server Faces (JSF).

Инфраструктура Spring MVC

В настоящее время на рынке доступно множество инфраструктур для веб-приложений, и каждой из них присущи достоинства и недостатки. Практически нереально выбрать какую-то одну инфраструктуру в надежде, что она удовлетворит всем требованиям веб-приложений, имеющих разную природу и предназначенных для разных бизнес-целей.

В приложении SpringBlog мы реализуем веб-уровень, воспользовавшись более общим и простым подходом. Будет принят шаблон MVC, а в качестве инфраструктуры — Spring MVC. На стороне представлений будут использоваться стандартные JSP-страницы. Для реализации функций Ajax будет применяться JavaScript-библиотека jQuery и всеобъемлющая поддержка RESTful-WS, обеспечиваемая Spring MVC. За соответствующими подробностями обращайтесь в главу 17.

Использование Tiles

В большинстве веб-приложений при обработке очередного запроса изменяется только некоторая область экрана, а общие элементы, такие как заголовок и панель навигации, остаются теми же самыми. С помощью Tiles страницы можно собирать из отдельных частей, называемых *мозаикой* (tile), что позволяет определить общие элементы один раз и затем многократно использовать по всему приложению. В главе 17 показано, как работать с Tiles в Spring-приложениях вообще и в примере приложения SpringBlog в частности.

RESTful-WS и OXM

Еще одной возможностью, обеспечиваемой Spring, является RSS-лента в формате XML. Для этого будет использоваться поддержка RESTful-WS в Spring MVC и OXM (Object to XML Mapping — отображение объектов в XML). Для маршализации и демаршализации XML будет применяться инфраструктура Castor (`www.castor.org`). Платформа Spring поддерживает и другие инфраструктуры привязки к XML, включая JAXB, XStream и XMLBeans. Дополнительные сведения по этому поводу также ищите в главе 16.

Реализация уровня безопасности

Безопасность является еще одной серьезной проблемой в любом приложении, особенно это касается веб-приложений. Без надлежащего контроля безопасности веб-приложение может быть подвержено атакам и потерять важные бизнес-данные.

Наиболее стандартной инфраструктурой безопасности для приложений, основанных на Spring, можно считать Spring Security (ранее называемую Acegi Security Framework for Spring). Ее всеобъемлющая поддержка декларативного и программного контроля безопасности и тесная интеграция с платформой Spring значительно упрощает код, которые разработчики должны написать.

В приложении SpringBlog инфраструктура Spring Security 3.1 будет использоваться для защиты веб-приложения и обеспечения того, что пользователям разрешено выполнять только действия, предусмотренные назначенными им ролями. Инфраструктура Spring Security рассматривается в главах 16 и 17.

Поскольку информация о пользователе и назначенных ему ролям хранится в базе данных SpringBlog, мы реализуем класс службы подробностей о пользователе для Spring Security, чтобы извлекать упомянутую информацию и соответствующим образом применять настройки безопасности.

Резюме

В этой главе вы взглянули на приложение SpringBlog, которое обсуждается в книге, и получили понятие о разнообразных функциональных возможностях SpringBlog, о том, как они реализованы, и где они обсуждаются в книге.

В следующей главе речь пойдет о ядре платформы Spring Framework — ее контейнере инверсии управления (Inversion of Control — IoC). В главе 4 расширены примеры, рассмотренные в главе 2, а также описаны разновидности IoC и их поддержка в Spring.

ГЛАВА 4

Введение в IoC и DI в Spring

В главе 1 были раскрыты базовые принципы инверсии управления (Inversion of Control — IoC) и внедрения зависимостей (Dependency Injection — DI). С практической точки зрения DI является специализированной формой IoC, хотя вы часто будете обнаруживать, что эти два термина используются взаимозаменяемо. В данной главе мы более подробно рассмотрим IoC и DI, формализуя отношения между этими двумя концепциями и демонстрируя, каким образом платформа Spring вписывается в общую картину.

После определения IoC и DI и их взаимоотношения с платформой Spring мы рассмотрим концепции, которые являются существенными для реализации DI в Spring. В этой главе показаны только основы реализации DI в Spring; более сложные средства DI будут описаны в главе 5, а в той же главе 5 и в главе 12 мы посмотрим на DI в контексте проектирования приложений. Итак, в данной главе рассматриваются следующие темы.

- *Концепции инверсии управления.* В этом разделе мы обсудим разновидности IoC, включая инверсию управления (Dependency Injection) и инверсию поиска (Dependency Lookup). Здесь мы опишем отличия между разнообразными подходами IoC и представим доводы за и против каждого из них.
- *Инверсия управления в Spring.* В этом разделе рассмотрены возможности IoC, доступные в Spring, и показано, как они реализованы. В частности, здесь описаны службы внедрения зависимостей, предлагаемые Spring, включая Setter Injection (Внедрение через метод установки), Constructor Injection (Внедрение через конструктор) и Method Injection (Внедрение через метод).
- *Внедрение зависимостей в Spring.* В этом разделе показана реализация контейнера IoC в Spring. Для определения бинов и требований DI основным интерфейсом, с которым будет взаимодействовать приложение, является `BeanFactory`. Однако, за исключением нескольких начальных примеров, все остальные примеры кода в этой главе будут сконцентрированы на использовании Spring-интерфейса `ApplicationContext`, который представляет собой расширение `BeanFactory` и обеспечивает намного более мощные средства, требуемые для корпоративных приложений. Отличия между `BeanFactory` и `ApplicationContext` будут описаны в последующих разделах.
- *Конфигурирование контекста приложения Spring.* В финальном разделе этой главы рассматривается применение обоих подходов к конфигурированию `ApplicationContext` — на основе XML-файлов и с помощью Java-аннотаций.

Данный раздел начинается с обсуждения конфигурации DI, после чего внимание переключается на дополнительные службы, предоставляемые `BeanFactory` (часть Spring-интерфейса `ApplicationContext`), такие как наследование, управление жизненным циклом и автосвязывание.

Инверсия управления и внедрение зависимостей

По своей сути IoC, а, следовательно, и DI, направлены на то, чтобы предложить простой механизм для предоставления зависимостей компонента (часто называемых *коллаборaторами* объекта) и управления этими зависимостями на протяжении всего их жизненного цикла. Компонент, который требует определенных зависимостей, зачастую называют *зависимым объектом* или, в случае IoC, целевым объектом. Вполне уместно сейчас заявить, что IoC предоставляет службы, через которые компоненты могут получать доступ к своим зависимостям, и службы для взаимодействия с зависимостями в течение их времени жизни. В общем случае IoC может быть расщеплена на два подтипа: инверсия управления (Dependency Injection) и инверсия поиска (Dependency Lookup). Эти подтипы в дальнейшем разделяются на конкретные реализации служб IoC. Из этого определения можно четко увидеть, что когда речь ведется о DI, мы всегда говорим об IoC, но когда речь идет об IoC, не всегда имеется в виду DI (например, Dependency Lookup — это также форма IoC).

Типы инверсии управления

Вас может интересовать, почему имеются два различных типа IoC, которые, к тому же, дополнительно разделены на различные реализации. Похоже, что четкий ответ на этот вопрос отсутствует; конечно, различные типы предоставляют определенный уровень гибкости, но нам кажется, что IoC — это более чем смесь старых и новых идей; это и представлено двумя разными типами IoC.

Dependency Lookup является намного более традиционным подходом и на первый взгляд выглядит более знакомым Java-программистам. Dependency Injection — новый, но менее устоявшийся подход, который хотя и кажется поначалу нелогичным, но в действительности обеспечивает намного большую гибкость и удобство использования, чем Dependency Lookup.

В случае IoC в стиле Dependency Lookup компонент должен получить ссылку на зависимость, тогда как в случае Dependency Injection зависимости внедряются в компонент контейнером IoC. Dependency Lookup поступает в виде двух типов: Dependency Pull (Извлечение зависимостей) и Contextualized Dependency Lookup (Контекстуализированный поиск зависимостей), или CDL. Dependency Injection также принимает две общих формы: Constructor Dependency Injection (Внедрение зависимостей через конструктор) и Setter Dependency Injection (Внедрение зависимостей через метод установки).

На заметку! При обсуждениях в этом разделе мы не заботимся о том, каким образом вымышленный контейнер IoC узнает обо всех разных зависимостях, а только о том, что в определенной точке он выполняет действия, описанные для каждого механизма.

Тип Dependency Pull

Для Java-разработчика Dependency Pull является самым узнаваемым типом IoC. В Dependency Pull зависимости извлекаются из реестра по мере необходимости. Любой, кто хотя бы раз писал код для доступа к EJB (2.1 или предшествующих версий), использовал Dependency Pull (т.е. через API-интерфейс JNDI для поиска компонента EJB).

На рис. 4.1 показан сценарий Dependency Pull через механизм поиска.

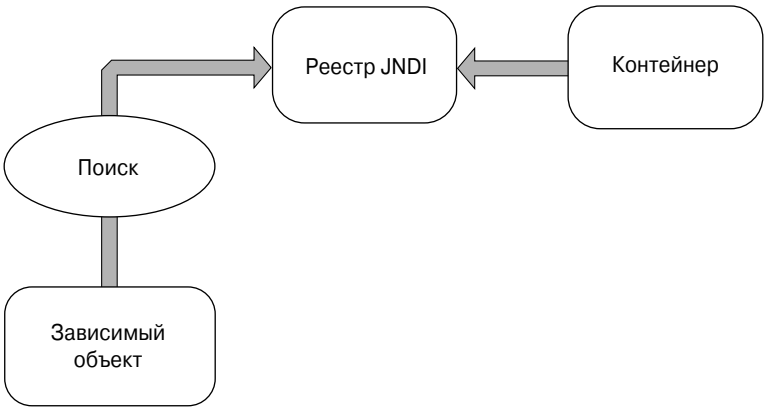


Рис. 4.1. Извлечение зависимостей через поиск JNDI

Платформа Spring также предлагает Dependency Pull в качестве механизма для извлечения компонентов, которыми она управляет; вы видели это в действии в главе 2. В листинге 4.1 показан типовой поиск Dependency Pull в основанном на Spring приложении.

Листинг 4.1. Dependency Pull в Spring

```
public static void main(String[] args) throws Exception {
    // Получение фабрики бинов.
    BeanFactory factory = getBeanFactory();
    MessageRenderer mr = (MessageRenderer) factory.getBean("renderer");
    mr.render();
}
```

Этот тип IoC распространен не только в JEE-приложениях (использующих EJB 2.1 или предшествующих версий), в которых широко применяются поиски JNDI для получения зависимостей из реестра, он также имеет важнейшее значение для работы с платформой Spring во многих средах.

Тип Contextualized Dependency Lookup

Тип Contextualized Dependency Lookup (CDL) в некоторых отношениях похож на Dependency Pull, но в CDL поиск осуществляется в контейнере, который управляет ресурсом, а не только в каком-то центральном реестре, и обычно он производится в установленной точке. Механизм CDL показан на рис. 4.2.

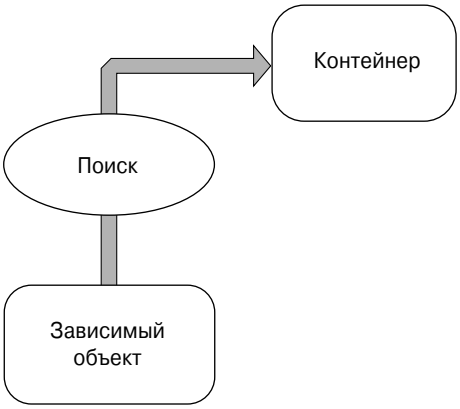


Рис. 4.2. Контекстуализированный поиск зависимостей

CDL работает за счет реализации компонентом интерфейса, подобного показанному в листинге 4.2.

Листинг 4.2. Интерфейс компонента для CDL

```
package com.apress.prospring3.ch4;

public interface ManagedComponent {
   public void performLookup(Container container);
}
```

Реализуя этот интерфейс, компонент сигнализирует контейнер о том, что он желает получить зависимость. `Container` обычно предоставляется лежащим в основе сервером приложений (например, Tomcat, JBoss) или платформой (вроде Spring).

В листинге 4.3 показан простой интерфейс `Container`, который предоставляет службу Dependency Lookup.

Листинг 4.3. Простой интерфейс `Container`

```
package com.apress.prospring3.ch4;

public interface Container {

   public Object getDependency(String key);
}
```

Когда контейнер готов передать зависимости компоненту, он вызывает метод `performLookup()` на каждом компоненте по очереди. Затем компонент может поискать свои зависимости с использованием интерфейса `Container`, как показано в листинге 4.4.

Листинг 4.4. Получение зависимостей в CDL

```
package com.apress.prospring3.ch4;

public class ContextualizedDependencyLookup implements ManagedComponent {

   private Dependency dependency;

   public void performLookup(Container container) {
      this.dependency = (Dependency) container.getDependency("myDependency");
   }
}
```

В листинге 4.4 обратите внимание на то, что `Dependency` является пустым классом.

Тип Constructor Dependency Injection

Constructor Dependency Injection — это тип внедрения зависимостей, при котором зависимости компонента предоставляются ему в его конструкторе (или конструкторах). Компонент объявляет конструктор или набор конструкторов, получающих в качестве аргументов его зависимости, и контейнер IoC передает зависимости компоненту при создании его экземпляра, как показано в листинге 4.5.

Листинг 4.5. Внедрение зависимостей через конструктор

```
package com.apress.prospring3.ch4;

public class ConstructorInjection {
   private Dependency dependency;

   public ConstructorInjection(Dependency dependency) {
      this.dependency = dependency;
   }
}
```

Тип Setter Dependency Injection

В случае Setter Dependency Injection контейнер IoC внедряет зависимости компонента в компонент через методы установки в стиле JavaBean. Методы установки компонента отражают набор зависимостей, которыми контейнер может управлять. В листинге 4.6 показан типовой компонент, основанный на Setter Dependency Injection.

Листинг 4.6. Внедрение зависимостей через метод установки

```
package com.apress.prospring3.ch4;

public class SetterInjection {
   private Dependency dependency;

   public void setDependency(Dependency dependency) {
      this.dependency = dependency;
   }
}
```

Внутри контейнера на требование зависимости, отраженное методом `setDependency()`, ссылаются по имени в стиле JavaBean — т.е. `dependency`. На практике Setter Injection является наиболее широко используемым механизмом внедрения, и он относится к одному из простейших в реализации механизмов IoC.

Выбор между внедрением и поиском

Выбор используемого стиля IoC — внедрение или поиск — не всегда является сложной задачей. Во многих случаях применяемый тип IoC определяется используемым контейнером. Например, в случае EJB 2.1 или предшествующих версий вы должны выбрать IoC в стиле поиска (через JNDI), чтобы получать EJB из контейнера JEE. В Spring, помимо первоначальных поисков бинов, компоненты и их зависимости всегда связываются друг с другом с помощью IoC в стиле внедрения.

На заметку! При использовании Spring доступ к ресурсам EJB можно производить без необходимости в явном поиске. Spring может действовать в качестве адаптера между системами IoC в стиле поиска и внедрения, таким образом позволяя управлять всеми ресурсами с применением внедрения.

Реальный вопрос заключается в следующем: когда есть выбор, какой метод должен использоваться — внедрение или поиск? Ответ: определенно внедрение. Взглянув на код в листингах 4.4 и 4.5, вы четко увидите, что применение внедрения оказывает нулевое влияние на код компонентов. Код Dependency Pull, с другой стороны, должен активно получать ссылку на реестр и взаимодействовать с ним при получении зависимостей, а использование CDL требует от классов реализации специфического интерфейса и поис-

ка зависимостей вручную. Когда применяется внедрение, все, что классы должны сделать — это позволить зависимостям внедряться либо через конструкторы, либо через методы установки.

В случае внедрения вы можете пользоваться своими классами полностью отдельно от контейнера IoC, который поставляет зависимые объекты вручную, тогда как в случае поиска ваши классы всегда являются зависимыми от классов и интерфейсов, определенных контейнером. Еще один недостаток поиска связан с чрезмерным затруднением процесса тестирования классов в изоляции от контейнера. При использовании внедрения тестирование компонентов становится тривиальным, поскольку вы можете просто предоставить зависимости самостоятельно с помощью соответствующего конструктора или метода установки.

На заметку! Более полное обсуждение тестирования с использованием Dependency Injection и Spring приведено в главе 19.

Решения на основе поиска являются неизбежно более сложными, чем решения, основанные на внедрении. Хотя такой сложности не следует бояться, мы ставим под сомнение допустимость добавления ненужной сложности к процессу, поскольку считаем этот аспект ключевым при управлении зависимостями в приложении.

Оставив все второстепенные причины в стороне, главной причиной выбора внедрения, а не поиска, является значительное упрощение работы. При использовании внедрения приходится писать существенно меньше кода, этот код прост и в общем случае может быть автоматизирован с помощью хорошей IDE-среды. Вы заметите, что весь код в примерах внедрения является пассивным в том смысле, что он не пытается активно выполнять какую-то задачу. Самым замечательным моментом, который вы отметите в коде внедрения, будет то, что получаемые объекты хранятся только в полях; отсутствует какой-либо код для извлечения зависимости из какого-то реестра или контейнера. Таким образом, код оказывается намного проще и менее подвержен ошибкам. Пассивный код легче сопровождать, чем активный код, поскольку в нем мало что может пойти на так. Обратите внимание на следующий фрагмент кода из листинга 4.4:

```
public void performLookup(Container container) {
  this.dependency = (Dependency) container.getDependency("myDependency");
}
```

В этом коде много чего может пойти не так, как было задумано: ключ зависимости может измениться, экземпляр контейнера может быть `null` или возвращенная зависимость может оказаться неподходящего типа. Мы привели этот код, т.к. он содержит большое количество перемещающихся частей, и здесь много чего может быть нарушено. Применение поиска может разъединить компоненты приложения, но при этом добавится сложность в дополнительный код, который требуется для соединения этих компонентов друг с другом с целью выполнения каких-либо полезных задач.

Выбор между Setter Injection и Constructor Injection

Теперь, когда стало ясно, какой метод IoC является предпочтительным, осталось выбрать применяемую разновидность внедрения — Setter Injection или Constructor Injection. Тип Constructor Injection особенно удобен, когда перед использованием компонента обязательно должен существовать экземпляр класса зависимости. Многие контейнеры, включая Spring, предоставляют механизм проверки, все ли зависимости определены, предназначенный для случая применения Setter Injection, но за счет использования Constructor Injection вы формально задаете требование для зависимости в независимой от контейнера манере.

Тип Setter Injection полезен в ряде случаев. Если компонент отражает свои зависимости контейнеру, но готов предоставить для них стандартные настройки, то обычно лучшим способом достичь этого является Setter Injection. Еще одно достоинство Setter Injection в том, что данный тип внедрения позволяет объявлять зависимости в интерфейсе, хотя это не настолько удобно, как могло показаться на первый взгляд. Предположим, что есть типовой бизнес-интерфейс с одним бизнес-методом по имени `defineMeaningOfLife()`. Если в дополнение к этому методу вы определите метод установки для внедрения, такой как `setEncylopedia()`, то тем самым обяжете все реализации использовать или хотя бы учитывать наличие этой зависимости. Тем не менее, вы не должны определять `setEncylopedia()` в бизнес-интерфейсе. Вместо этого вы можете определить данный метод в классах, реализующих бизнес-интерфейс. При таком подходе все современные контейнеры IoC, в том числе и Spring, могут работать с компонентом в терминах бизнес-интерфейса, но по-прежнему предоставлять зависимости реализованного класса. Несколько прояснить сказанное поможет пример. Пусть имеется бизнес-интерфейс, приведенный в листинге 4.7.

Листинг 4.7. Интерфейс `Oracle`

```
package com.apress.prospring3.ch4;

public interface Oracle {
   public String defineMeaningOfLife();
}
```

Обратите внимание, что этот бизнес-интерфейс не определяет никаких методов установки для Dependency Injection. Реализовать такой интерфейс можно так, как показано в листинге 4.8.

Листинг 4.8. Реализация интерфейса `Oracle`

```
package com.apress.prospring3.ch4;

public class BookwormOracle implements Oracle {

   private Encyclopedia encyclopedia;

   public void setEncyclopedia(Encyclopedia encyclopedia) {
      this. encyclopedia = encyclopedia;
   }

   public String defineMeaningOfLife() {
      return "Encyclopedias are a waste of money - use the Internet";
   }
}
```

Как видите, класс `BookwormOracle` не только реализует интерфейс `Oracle`, но также определяет метод установки для Dependency Injection. Платформа Spring более чем удобна при работе со структурой подобного рода, и нет никакой необходимости определять зависимости в бизнес-интерфейсе. Возможность использовать интерфейсы для определения зависимостей — часто рекламируемое преимущество Setter Injection, однако в действительности вы должны стремиться к применению методов установки исключительно для внедрения из своих бизнес-интерфейсов и интерфейсов DAO. Если только вы не абсолютно уверены в том, что все реализации отдельного бизнес-интерфейса требуют специфической зависимости, позвольте каждому классу реализации определить собственные зависимости и поддерживать в бизнес-интерфейсе только бизнес-методы.

Хотя вы не всегда должны помещать методы установки для зависимостей в бизнес-интерфейс, добавление в упомянутый интерфейс методов установки и извлечения для параметров конфигурации является удачным решением, которое делает Setter Injection полезным инструментом. Мы рассматриваем параметры конфигурации как специальный случай зависимостей. Безусловно, компоненты зависят от данных конфигурации, однако конфигурационные данные значительно отличаются от тех типов зависимостей, которые были показаны до сих пор. Мы вскоре обсудим отличия, а пока взгляните на бизнес-интерфейс, показанный в листинге 4.9.

Листинг 4.9. Интерфейс `NewsletterSender`

```
package com.apress.prospring3.ch4;

public interface NewsletterSender {

   public void setSmtpServer(String smtpServer);
   public String getSmtpServer();

   public void setFromAddress(String fromAddress);
   public String getFromAddress();

   public void send();
}
```

Интерфейс `NewsletterSender` реализуется классами, которые отправляют по электронной почте набор информационных бюллетеней. Единственным бизнес-методом является `send()`, но обратите внимание на определение в интерфейсе двух свойств JavaBean. Почему это было сделано, ведь только что говорилось о том, что не следует определять зависимости в бизнес-интерфейсе? Причина в том, что эти значения — адрес SMTP-сервера и почтовый адрес, по которому отправляются сообщения — в практическом смысле не являются зависимостями; они представляют собой конфигурационные детали, влияющие на функционирование всех реализаций `NewsletterSender`. Возможности Dependency Injection в Spring формируют идеальное решение для поддержки внешней конфигурации компонентов приложения, причем не для предоставления зависимостей, а в качестве механизма для вынесения конфигурационных настроек компонентов за пределы приложения. И здесь возникает вопрос: в чем же тогда состоит отличие между параметром конфигурации и любым другим видом зависимости? В большинстве случаев можно четко определить, что зависимость должна классифицироваться как параметр конфигурации, но если вы не уверены, то проверьте следующие три характеристики, которые указывают на параметр конфигурации.

- Параметры конфигурации являются пассивными. В интерфейсе `NewsletterSender`, показанном в листинге 4.8, параметр SMTP-сервера может служить примером пассивной зависимости. Пассивные зависимости не используются напрямую для выполнения какого-то действия; для запуска своих действий они применяются внутренне или другими зависимостями. В примере `MessageRenderer` из главы 2 зависимость `MessageProvider` не является пассивной — она выполняет функцию, которая была необходима `MessageRenderer` для завершения своей задачи.
- Параметры конфигурации — это обычно информация, а не другие компоненты. Под этим мы понимаем, что параметр конфигурации представляет собой определенную порцию информации, которая необходима компоненту для завершения своей работы. Очевидно, что SMTP-сервер — порция информации, требуемая `NewsletterSender`, однако `MessageProvider` — это в действительности другой компонент, который необходим для корректного функционирования `MessageRenderer`.

- Параметры конфигурации обычно представляют собой простые значения или коллекции простых значений. В действительности это является побочным продуктом первых двух характеристик: параметры конфигурации — обычно простые значения. В Java это означает, что они относятся к примитивному типу (или соответствующему классу-оболочке), к типу `String` или к коллекции таких значений. Простые значения, как правило, пассивны. Другими словами, со значениями `String` мало что можно сделать, кроме манипулирования данными, которые они представляют; почти всегда эти значения используются для информационных целей: например, значение `int` может представлять номер порта для прослушивания сетевым сокетом, а значение `String` — SMTP-сервер, через который будет производиться отправка почтовых сообщений.

При рассмотрении вопроса, определять ли конфигурационные настройки в бизнес-интерфейсе, также подумайте о том, применим ли конкретный параметр конфигурации ко всем реализациям этого бизнес-интерфейса либо же только к одной. Например, в случае `NewsletterSender` очевидно, что все реализации должны знать, какой SMTP-сервер использовать для отправки почтовых сообщений. Однако такую конфигурационную настройку, как признак защищенной отправки сообщений, имеет смысл не включать в бизнес-интерфейс, потому что данную возможность поддерживают не все почтовые API-интерфейсы, и вполне уместно предположить, что многие реализации вообще не будут принимать во внимание эту настройку.

На заметку! Вспомните, что в главе 2 мы решили определять зависимости на уровне бизнес-логики. Это делалось только в целях иллюстрации и ни в коем случае не должно рассматриваться как рекомендуемый подход.

Внедрение зависимостей через метод установки также позволяет менять зависимости для различных реализаций на лету, не создавая нового экземпляра родительского компонента. Это становится возможным благодаря поддержке JMX в Spring. Пожалуй, самое значительное преимущество типа Setter Injection в том, что он представляет собой наименее навязчивый механизм внедрения из всех существующих.

Если вы определяете конструкторы для внедрения в классе, который иначе имел бы только конструктор по умолчанию, то тем самым оказываете влияние на весь код, использующий этот код в среде без IoC. Дополнительные методы установки, определяемые в классе для целей IoC, не влияют на возможность взаимодействия с ним со стороны других классов.

В общем, внедрение через метод установки является более удачным вариантом, потому что оно оказывает наименьшее влияние на возможность использования вашего кода в средах без IoC. Внедрение через конструктор будет хорошим выбором, когда нужно гарантировать передачу зависимостей компоненту, но имейте в виду, что многие контейнеры предлагают для этого собственные механизмы с Setter Injection. В большей части кода примера приложения используется Setter Injection, хотя есть и несколько примеров применения Constructor Injection.

Инверсия управления в Spring

Как упоминалось ранее, инверсия управления — это крупная часть того, делает Spring, и ядро реализации Spring основано на инверсии управления, хотя также предоставляются и средства Dependency Lookup. Когда платформа Spring предоставляет коллаборaторы зависимому объекту автоматически, она делает это с использованием инверсии управления (Dependency Injection). В приложении, основанном на Spring, все-

гда предпочтительнее применять Dependency Injection для передачи коллаборaторов зависимым объектам вместо того, чтобы заставлять зависимые объекты получать коллаборaторы через поиск. Механизм Dependency Injection в Spring показан на рис. 4.3 (механизм Dependency Lookup был представлен на рис. 4.2).

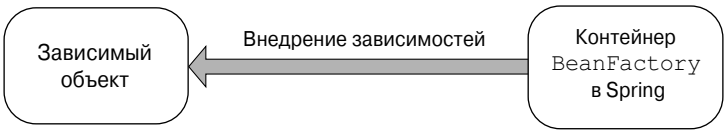


Рис. 4.3. Механизм внедрения зависимостей в Spring

Несмотря на то что Dependency Injection является предпочтительным механизмом для связывания вместе коллаборaторов и зависимых объектов, для доступа к зависимым объектам понадобится Dependency Lookup. Во многих средах Spring не может автоматически связать все компоненты приложения с использованием Dependency Injection, поэтому для доступа к начальному набору компонентов придется применять Dependency Lookup. Например, в автономных Java-приложениях необходимо выполнить начальную загрузку контейнера Spring в методе `main()` и получить зависимости (через интерфейс `ApplicationContext`) для программной обработки. Однако при построении веб-приложений с применением поддержки MVC в Spring этого можно избежать за счет автоматического связывания всего приложения. Всегда, когда возможно использовать Dependency Injection вместе с платформой Spring, это следует делать; в противном случае можно прибегнуть к средствам Dependency Lookup. В этой главе вы увидите примеры обоих подходов в действии, и мы укажем на них, как только они встретятся впервые.

Интересной функцией контейнера IoC в Spring является возможность действовать в качестве адаптера между его собственным контейнером Dependency Injection и внешними контейнерами Dependency Lookup. Эта функция рассматривается далее в главе.

Платформа Spring поддерживает Constructor Injection и Setter Injection, укрепляя стандартный набор средств IoC рядом полезных дополнений и упрощая разработку в целом. В оставшейся части главы приводятся основы контейнера DI в Spring с множеством иллюстративных примеров.

Внедрение зависимостей с помощью Spring

Поддержка Dependency Injection в Spring является всеобъемлющей и, как вы увидите в главе 5, выходит за рамки стандартного набора средств IoC, обсуждавшегося до сих пор. В оставшейся части текущей главы рассматриваются основы контейнера Dependency Injection в Spring, в том числе Setter Injection, Constructor Injection и Method Injection, вместе с детальным описанием способов конфигурирования Dependency Injection в Spring.

Бины и фабрики бинов

Ядром контейнера Dependency Injection в Spring является интерфейс фабрики бинов под названием `BeanFactory`. Этот интерфейс отвечает за управление компонентами, в том числе их зависимостями и жизненными циклами. Термин *бин* в Spring используется для ссылки на любой компонент, управляемый контейнером. Обычно бины на определенном уровне придерживаются спецификации JavaBean, но это не обязательно, особенно если для связывания бинов друг с другом планируется применять Constructor Injection.

Если приложению требуется только поддержка DI, с контейнером Spring DI можно взаимодействовать через интерфейс BeanFactory. В этом случае приложение должно создать экземпляр класса, реализующего интерфейс BeanFactory, и сконфигурировать его в соответствие с информацией о бине и зависимостях. После того, как это сделано, приложение может получать доступ к бинам через BeanFactory и пользоваться их обработкой. В ряде случаев вся настройка подобного рода производится автоматически (например, в веб-приложении ApplicationContext будет загружаться веб-контейнером во время начальной загрузки приложения с помощью класса ContextLoaderListener, предоставляемого Spring, который объявлен в файле дескрипторов web.xml). Но во многих случаях кодировать настройку приходится самостоятельно. Все примеры, приведенные в этой главе, требуют ручной настройки реализации BeanFactory.

Несмотря на то что BeanFactory можно сконфигурировать программно, более распространен подход с внешним конфигурированием, при котором используется какая-то разновидность файла конфигурации. Внутренне конфигурация бина представлена экземплярами классов, которые реализуют интерфейс BeanDefinition.

Конфигурация бина хранит не только информацию о самом бине, но также и о бинах, от которых он зависит. Для любых классов реализации BeanFactory, которые также реализуют интерфейс BeanDefinitionRegistry, можно читать данные BeanDefinition из файла конфигурации, используя либо PropertiesBeanDefinitionReader, либо XmlBeanDefinitionReader. Две основные реализации BeanFactory, поставляемые в составе платформы Spring, реализуют BeanDefinitionRegistry.

Класс PropertiesBeanDefinitionReader читает определение бина из файла свойств, а XmlBeanDefinitionReader — из XML-файла.

Итак, бины можно идентифицировать внутри BeanFactory, и каждому бину может быть назначен либо идентификатор, либо имя, либо то и другое. Бин можно также создать без идентификатора и имени (такой бин называется *анонимным*) или как внутренний бин в рамках указанного бина. Каждый бин имеет, по крайней мере, одно имя, но может иметь их любое количество (дополнительные имена разделяются запятыми). Все имена после первого считаются псевдонимами одного и того же бина. Для извлечения бина из BeanFactory, а также для установки отношений, связанных с зависимостями (т.е. когда бин X зависит от бина Y), используются идентификаторы или имена.

Реализации BeanFactory

Столь развернутое описание BeanFactory может привести к тому, что его использование покажется излишне сложным, однако на практике это не так. Давайте рассмотрим простой пример.

Пусть имеется реализация, имитирующая оракула, рассказывающего о смысле жизни. В листингах 4.10 и 4.11 показаны интерфейс и простая реализация.

Листинг 4.10. Интерфейс Oracle

```
package com.apress.prospring3.ch4;

public interface Oracle {

   public String defineMeaningOfLife();
}
```

Листинг 4.11. Простая реализация интерфейса Oracle

```
package com.apress.prospring3.ch4;

public class BookwormOracle implements Oracle {
```

```
    public String defineMeaningOfLife() {
        return "Encyclopedias are a waste of money - use the Internet";
    }
}
```

А теперь посмотрим, как в автономной Java-программе можно инициализировать BeanFactory и получить бин oracle для работы с ним (листинг 4.12).

Листинг 4.12. Использование фабрики бинов

```
package com.apress.prospring3.ch4;
import org.springframework.beans.factory.support.DefaultListableBeanFactory;
import org.springframework.beans.factory.xml.XmlBeanDefinitionReader;
import org.springframework.core.io.FileSystemResource;

public class XmlConfigWithBeanFactory {

    public static void main(String[] args) {

        DefaultListableBeanFactory factory = new DefaultListableBeanFactory();

        XmlBeanDefinitionReader rdr = new XmlBeanDefinitionReader(factory);
        rdr.loadBeanDefinitions(new
            FileSystemResource("src/main/resources/xmlBeanFactory.xml"));

        // Oracle oracle = (Oracle)factory.getBean("oracle");
        Oracle oracle = factory.getBean("oracle", Oracle.class);
        System.out.println(oracle.defineMeaningOfLife());
    }
}
```

В листинге 4.12 мы используем DefaultListableBeanFactory — одну из двух основных реализаций BeanFactory, поставляемых Spring — и читаем информацию BeanDefinition из XML-файла с помощью XmlBeanDefinitionReader. После того, как реализация BeanFactory создана и сконфигурирована, мы извлекаем бин Oracle, используя его имя oracle, которое настроено в XML-файле конфигурации.

В листинге 4.13 приведено содержимое XML-файла для начальной загрузки BeanFactory (xmlBeanFactory.xml).

Листинг 4.13. Простая XML-конфигурация для Spring

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.1.xsd">

    <!-- Бин oracle, используемый в нескольких примерах -->
    <bean id="oracle" name="wiseworm"
          class="com.apress.prospring3.ch4.BookwormOracle"/>

</beans>
```

В показанном выше файле объявлен бин Spring, которому назначен идентификатор "oracle" и имя "wiseworm", а также лежащий в основе класс реализации com.apress.prospring3.ch4.BookwormOracle. На данный момент этих сведений о конфигурации вполне достаточно; мы обсудим детали в последующих разделах.

Имея такую конфигурацию, запустите программу из листинга 4.12, и вы увидите фразу, возвращаемую методом `defineMeaningOfLife()`, в выводе консоли STS.

В дополнение к `XmlBeanDefinitionReader` платформа Spring также предоставляет класс `PropertiesBeanDefinitionReader`, который позволяет управлять конфигурацией бина с использованием свойств, отличных от XML. Хотя свойства почти идеальны для небольших, простых приложений, они могут быстро стать громоздкими, когда приходится иметь дело с большим количеством бинов. По этой причине предпочтительнее применять XML-формат конфигурации для всех приложений кроме самых тривиальных.

Разумеется, можно объявлять собственные реализации `BeanFactory`, хотя следует помнить, что это требует довольно много работы: чтобы получить тот же уровень функциональности, который предлагают готовые реализации `BeanFactory`, понадобится реализовать намного больше интерфейсов, нежели просто `BeanFactory`. Если все, что требуется — это определение нового механизма конфигурации, создайте свое средство чтения определений, разработав класс, расширяющий класс `DefaultListableBeanFactory`, который реализует интерфейс `BeanFactory`.

Интерфейс `ApplicationContext`

В Spring интерфейс `ApplicationContext` представляет собой расширение `BeanFactory`. В дополнение к службам DI, интерфейс `ApplicationContext` также предлагает другие службы, такие как служба транзакций и служба АОП, источник сообщений для интернационализации, обработка событий приложения и многое другое.

При разработке приложений, основанных на Spring, рекомендуется взаимодействовать с платформой Spring через интерфейс `ApplicationContext`. Spring поддерживает начальную загрузку `ApplicationContext` за счет ручного кодирования (создание экземпляра вручную и загрузка соответствующей конфигурации) или в среде веб-контейнера через `ContextLoaderListener`. Начиная с этого места, все приводимые примеры кода будут использовать `ApplicationContext`.

Конфигурирование `ApplicationContext`

Обсудив базовые концепции IoC и DI, а также ознакомившись с простым примером использования интерфейса `BeanFactory` в Spring, давайте более подробно рассмотрим детали конфигурирования Spring-приложения.

В последующих разделах мы пройдемся по различным аспектам конфигурирования Spring-приложений. В частности, мы сосредоточим внимание на интерфейсе `ApplicationContext`, который предоставляет намного больше вариантов конфигурации, чем традиционный интерфейс `BeanFactory`.

Варианты конфигурации Spring (XML и Java-аннотации)

Перед тем, как мы погрузимся в детали конфигурирования `ApplicationContext`, давайте посмотрим, какие варианты доступны для определения конфигурации приложения в рамках Spring.

Изначально платформа Spring поддерживала определение бинов либо через свойства, либо через XML-файл, и вариант с XML-файлом использовался большинством разработчиков Spring-приложений в течение довольно продолжительного времени. С момента выхода JDK 5 и появления в Spring поддержки Java-аннотаций, платформа Spring (начиная с версии Spring 2.5) также позволяет применять для конфигурирования `ApplicationContext` аннотации Java.

Так что же лучше — XML или аннотации? На эту тему велось множество дискуссий, и часть из них можно найти в Интернете (например, просмотрите форум сообщества Spring по адресу `http://forum.springsource.org`). Тем не менее, строго определенный ответ на этот вопрос отсутствует, и каждый из подходов характеризуется как положительными, так и отрицательными чертами. Подход с XML-файлом позволяет вынести всю конфигурацию за пределы Java-кода, в то время как аннотации дают возможность разработчику определять и просматривать настройку DI внутри кода. Платформа Spring также поддерживает смесь из этих двух подходов в единственном `ApplicationContext` (конфигурация из XML-файла будет переопределять конфигурацию, заданную аннотациями). В настоящее время часто применяется подход с определением инфраструктуры приложения (например, источника данных, диспетчера транзакций, фабрики подключений JMS, JMX и т.д.) в XML-файле, а конфигурации DI (внедряемых бинов и зависимостей для бинов) — в аннотациях. Однако независимо от того, какой вариант выбран, его необходимо придерживаться и довести это до сведения всех членов команды разработки. Следование используемому стилю и согласованное его применение во всем приложении намного упростит последующие действия по разработке и сопровождению.

Чтобы облегчить понимание обоих подходов к конфигурации, мы по возможности будем приводить примеры кода XML и аннотаций рядом друг с другом.

Обзор базовой конфигурации

Для конфигурации XML понадобится объявить обязательное базовое пространство имен, предоставляемое Spring, которое требуется приложению. В листинге 4.13 приведен простой пример, в котором объявлено только пространство имен `beans` для определения бинов Spring.

Помимо `beans` платформа Spring предоставляет большое количество других пространств имен, предназначенных для разных целей. В листинге 4.14 показано объявление пространства имен для конфигурации Spring, которая будет использоваться примерами в этой главе. В разделе, посвященном конфигурации Spring, мы подготовим два конфигурационных файла для всех примеров. Один из них, `app-context-xml.xml`, предназначен для конфигурации в стиле XML, в другой — `app-context-annotation.xml` — для конфигурации в стиле аннотаций.

Листинг 4.14. XML-конфигурация Spring

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:context="http://www.springframework.org/schema/context"
   xmlns:p="http://www.springframework.org/schema/p"
   xmlns:c="http://www.springframework.org/schema/c"
   xmlns:util="http://www.springframework.org/schema/util"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context-3.1.xsd
      http://www.springframework.org/schema/util
      http://www.springframework.org/schema/util/spring-util-3.1.xsd">

</beans>
```

В предыдущем объявлении пространства имен `beans` сделано пространством имен по умолчанию. Кроме того, были также объявлены следующие пространства имен.

- context. Пространство имен context обеспечивает поддержку для конфигурирования ApplicationContext.
- p. Пространство имен p предоставляет более простую конфигурацию DI для Setter Injection.
- c. Появившееся в версии Spring 3.1, пространство имен c предлагает более простую конфигурацию DI для Constructor Injection.
- util. Пространство имен util предоставляет ряд полезных утилит для конфигурации DI.

Платформа Spring также предлагает множество пространств имен, предназначенных для разных целей, среди которых aop для поддержки АОП, tx для поддержки транзакций и т.д. Эти пространства имен будут рассмотрены в соответствующих главах.

Для использования поддержки аннотаций в Spring необходимо поместить в XML-файл конфигурации (app-context-annotation.xml) дескрипторы, показанные в листинге 4.15.

Листинг 4.15. XML-конфигурация Spring с поддержкой аннотаций

```
// Объявления пространств имен пропущены.
<context:annotation-config/>
<context:component-scan base-package="com.apress.prospring3.ch4.annotation" />
```

Дескриптор <context:annotation-config> сообщает Spring о необходимости просканировать кодовую базу на предмет требований зависимостей, а дескриптор <context:component-scan> заставляет Spring сканировать код для поиска внедряемых бинов в рамках указанного пакета (и всех его подпакетов). В дескрипторе <context:component-scan> можно определить множество пакетов, используя символ запятой, точки запятой или пробела в качестве разделителя. Кроме того, этот дескриптор поддерживает включение и исключение сканирования компонентов для более точного управления. Например, взгляните на конфигурацию в листинге 4.16.

Листинг 4.16. Сканирование компонентов в XML-конфигурации Spring

```
<context:component-scan base-package="com.apress.prospring3.ch4.annotation">
  <context:exclude-filter type="assignable" expression="com.example.NotAService"/>
</context:component-scan>
```

Приведенный выше дескриптор сообщает Spring о необходимости сканирования указанного пакета, но с пропуском классов, которым был назначен тип, заданный в выражении (либо класс, либо интерфейс). Помимо исключающего фильтра можно использовать включающий фильтр. Для типа можно указывать annotation, regex, assignable, AspectJ или custom (с собственным классом фильтра, который реализует org.springframework.core.type.filter.TypeFilter) в качестве критерия фильтрации. Формат выражения зависит от указанного типа.

Объявление компонентов Spring

После разработки некоторой разновидности классов служб, предназначенных для использования в базовом приложении Spring, необходимо сообщить Spring, что эти бины пригодны для внедрения в другие бины, и позволить Spring управлять ими. Вспомните пример из главы 2, в котором интерфейс MessageRender выводил сообще-

ние и полагался на интерфейс `MessageProvider` в получении сообщения для вывода. В листинге 4.17 показаны интерфейсы и реализации этих двух служб.

Листинг 4.17. `MessageRenderer` и `MessageProvider`

```
package com.apress.prospring3.ch4;

public interface MessageRenderer {

   public void render();

   public void setMessageProvider(MessageProvider provider);

   public MessageProvider getMessageProvider();

}

package com.apress.prospring3.ch4.xml;

import com.apress.prospring3.ch4.MessageProvider;
import com.apress.prospring3.ch4.MessageRenderer;

public class StandardOutMessageRenderer implements MessageRenderer {

   private MessageProvider messageProvider = null;

   public void render() {
      if (messageProvider == null) {
         throw new RuntimeException(
         "You must set the property messageProvider of class:"
         + StandardOutMessageRenderer.class.getName());
      }

      System.out.println(messageProvider.getMessage());
   }

   public void setMessageProvider(MessageProvider provider) {
      this.messageProvider = provider;
   }

   public MessageProvider getMessageProvider() {
      return this.messageProvider;
   }

}

package com.apress.prospring3.ch4;

public interface MessageProvider {

   public String getMessage();

}

package com.apress.prospring3.ch4.xml;

import com.apress.prospring3.ch4.MessageProvider;

public class HelloWorldMessageProvider implements MessageProvider {

   public String getMessage() {
      return "Hello World!";
   }

}
```

Чтобы объявить бины в XML-файле `app-context-xml.xml`, к базовой конфигурации (показанной ранее в листинге 4.14) потребуется добавить дескрипторы `<bean>` из листинга 4.18.

Листинг 4.18. Объявление бинов Spring (стиль XML)

```
<bean id="messageRenderer"
      class="com.apress.prospring3.ch4.xml.StandardOutMessageRenderer"/>

<bean id="messageProvider"
      class="com.apress.prospring3.ch4.xml.HelloWorldMessageProvider"/>
```

Дескрипторы, приведенные в листинге 4.18, объявляют два бина, один с идентификатором "messageProvider" и реализацией HelloWorldMessageProvider, а другой с идентификатором "messageRenderer" и реализацией StandardOutMessageRenderer.

Для определения бинов Spring через аннотации модифицировать XML-файл конфигурации (app-context-annotation.xml) не понадобится; нужно просто добавить соответствующую аннотацию к классам реализации служб в пакете com.apress.prospring3.ch4.annotation (листинг 4.19).

Листинг 4.19. Объявление бинов Spring (стиль аннотаций)

```
package com.apress.prospring3.ch4.annotation;

import org.springframework.stereotype.Service;

// Остальная часть кода опущена.
@Service("messageRenderer")
public class StandardOutMessageRenderer implements MessageRenderer {

package com.apress.prospring3.ch4.annotation;

import org.springframework.stereotype.Service;

// Остальная часть кода опущена.
@Service("messageProvider")
public class HelloWorldMessageProvider implements MessageProvider {
```

В предыдущем примере кода аннотация @Service используется для указания на то, что этот бин предоставляет службы, которые могут требоваться другим бинам; в качестве параметра аннотации передается имя бина. Во время начальной загрузки ApplicationContext с XML-конфигурацией, показанной в листинге 4.15, платформа Spring будет искать эти компоненты и создавать бины с указанными именами.

Выбранный подход никак не влияет на способ получения бинов из ApplicationContext. В листинге 4.20 приведен пример кода для получения поставщика сообщений.

Листинг 4.20. Объявление бинов Spring (тестирование)

```
package com.apress.prospring3.ch4;

import org.springframework.context.support.GenericXmlApplicationContext;

public class DeclareSpringComponents {
   public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:META-INF/spring/app-context-annotation.xml");
      ctx.refresh();

      MessageProvider messageProvider = ctx.getBean("messageProvider",
         MessageProvider.class);
      System.out.println(messageProvider.getMessage());
   }
}
```

В листинге 4.20 вместо DefaultListableBeanFactory создается экземпляр GenericXmlApplicationContext. Класс GenericXmlApplicationContext реализует интерфейс ApplicationContext и способен выполнить начальную загрузку ApplicationContext из конфигураций, определенных в XML-файлах.

Если поменять местами файлы app-context-xml.xml и app-context-annotation.xml в предоставленном исходном коде для этой главы, можно заметить, что оба случая дают один и тот же результат — вывод строки “Hello World!”.

В листингах 4.21 (app-context-xml.xml) и 4.22 (app-context-annotation.xml) приведено содержимое конфигурационных файлов для конфигурации в стиле XML и в стиле аннотаций, которые обсуждались ранее.

Листинг 4.21. Конфигурация в стиле XML (app-context-xml.xml)

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:context="http://www.springframework.org/schema/context"
      xmlns:p="http://www.springframework.org/schema/p"
      xmlns:c="http://www.springframework.org/schema/c"
      xmlns:util="http://www.springframework.org/schema/util"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-3.1.xsd
        http://www.springframework.org/schema/util
        http://www.springframework.org/schema/util/spring-util-3.1.xsd">

   <bean id="messageProvider"
        class="com.apress.prospring3.ch4.xml.HelloWorldMessageProvider"/>

</beans>
```

Листинг 4.22. Конфигурация в стиле аннотаций (app-context-annotation.xml)

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:context="http://www.springframework.org/schema/context"
      xmlns:p="http://www.springframework.org/schema/p"
      xmlns:c="http://www.springframework.org/schema/c"
      xmlns:util="http://www.springframework.org/schema/util"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-3.1.xsd
        http://www.springframework.org/schema/util
        http://www.springframework.org/schema/util/spring-util-3.1.xsd">

   <context:annotation-config/>

   <context:component-scan
        base-package="com.apress.prospring3.ch4.annotation"/>

</beans>
```

Использование внедрения через метод установки

Для конфигурирования Setter Injection с использованием XML необходимо указать дескрипторы `<property>` внутри `<bean>` для каждого дескриптора `<property>`, в который должна быть внедрена зависимость. Например, чтобы присвоить бин поставщика сообщений свойству `messageProvider` бина `messageRenderer`, понадобится просто изменить дескриптор `<bean>` для бина `messageRenderer`, как показано в листинге 4.23.

Листинг 4.23. Внедрение через метод установки (стиль XML)

```
<bean id="messageRenderer" class="com.apress.prospring3.ch4.xml.
StandardOutMessageRenderer">
   <property name="messageProvider">
      <ref bean="messageProvider"/>
   </property>
</bean>
```

В этом коде мы присваиваем бин `messageProvider` свойству `messageProvider`. С помощью дескриптора `<ref>` свойству можно присвоить ссылку на бин (более подробно об этом позже). Если используется Spring 2.5 или последующие версии и в XML-файле конфигурации объявлено пространство имен `p`, то объявить внедрение можно следующим образом:

```
<bean id="messageRenderer"
      class="com.apress.prospring3.ch4.xml.StandardOutMessageRenderer"
      p:messageProvider-ref="messageProvider"/>
```

Пространство имен `p` предлагает упрощенный способ объявления Setter Injection.

В случае аннотаций все даже еще проще. Нужно только добавить аннотацию `@Autowired` к методу установки, как показано в листинге 4.24.

Листинг 4.24. Внедрение через метод установки (стиль аннотаций)

```
package com.apress.prospring3.ch4.annotation;

import org.springframework.beans.factory.annotation.Autowired;

// Остальная часть кода опущена.
@Service("messageRenderer")
public class StandardOutMessageRenderer implements MessageRenderer {
   ...
   @Autowired
   public void setMessageProvider(MessageProvider provider) {
      this.messageProvider = provider;
   }
   ...
}
```

После добавления дескриптора `<context:annotation-config>` в XML-файл конфигурации во время инициализации `ApplicationContext` платформа Spring обнаружит эти аннотации `@Autowired` и внедрит зависимость (найденную через дескриптор `<context:component-scan>`) по мере надобности.

На заметку! Для получения тех же результатов вместо `@Autowired` можно также использовать `@Resource(name="messageProvider")`. Аннотация `@Resource` присутствует в стандарте JSR-250, который определяет общий набор Java-аннотаций для применения в рамках платформ JSE и JEE. Отличающаяся от `@Autowired`, аннотация `@Resource` поддерживает параметр `name` для реализации более точных требований DI.

А теперь давайте проверим результат с помощью кода, показанного в листинге 4.25.

Листинг 4.25. Использование внедрения через метод установки (тестирование)

```
package com.apress.prospring3.ch4;

import org.springframework.context.support.GenericXmlApplicationContext;

public class UsingSetterInjection {

   public static void main(String[] args) {

      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:app-context-xml.xml");
      ctx.refresh();

      MessageRenderer messageRenderer = ctx.getBean("messageRenderer",
         MessageRenderer.class);
      messageRenderer.render();
   }

}
```

Как и в предыдущем разделе, файлы `app-context-xml.xml` и `app-context-annotation.xml` в предоставленном для главы исходном коде можно поменять местами; в обоих случаях получается один и тот же результат — вывод строки “Hello World!”.

Использование внедрения через конструктор

В предыдущем примере реализация `MessageProvider` — `HelloWorldMessageProvider` — возвращает одно и то же жестко закодированное сообщение для каждого вызова метода `getMessage()`. В файле конфигурации Spring можно просто создать конфигурируемый `MessageProvider`, который позволит определять сообщение внешне, как показано в листинге 4.26.

Листинг 4.26. Класс `ConfigurableMessageProvider` (стиль XML)

```
package com.apress.prospring3.ch4.xml;

import com.apress.prospring3.ch4.MessageProvider;

public class ConfigurableMessageProvider implements MessageProvider {

   private String message;

   public ConfigurableMessageProvider(String message) {
      this.message = message;
   }

   public String getMessage() {
      return message;
   }
}
```

Как видите, создать экземпляр `ConfigurableMessageProvider`, не предоставив значение для сообщения, невозможно (если только не указать `null`). Это в точности то, что требуется, и данный класс идеально подходит для использования с Constructor Injection. В листинге 4.27 показано, как переделать определение бина `messageProvider` для создания экземпляра `ConfigurableMessageProvider`, внедряя сообщение с помощью Constructor Injection.

Листинг 4.27. Использование внедрения через конструктор (стиль XML)

```
<bean id="messageProvider"
      class="com.apress.prospring3.ch4.xml.ConfigurableMessageProvider">
   <constructor-arg>
      <value>This is a configurable message</value>
   </constructor-arg>
</bean>
```

В этом коде вместо дескриптора `<property>` используется дескриптор `<constructor-arg>`. Поскольку в этот раз другой бин не передается, а только литерал `String`, для указания значения аргумента конструктора применяется дескриптор `<value>`, а не `<ref>`.

Когда есть более одного аргумента конструктора или класс имеет несколько конструкторов, каждому дескриптору `<constructor-arg>` нужно предоставить атрибут `index` для указания индекса аргумента, начинающегося с 0, в сигнатуре конструктора. Имея дело с конструкторами, принимающими несколько аргументов, всегда лучше использовать атрибут `index`, чтобы избежать путаницы между параметрами и обеспечить выбор платформой Spring корректного конструктора.

Подобно пространству имен `p`, в версии Spring 3.1 можно также использовать пространство имен `c`, как показано ниже:

```
<bean id="messageProvider"
      class="com.apress.prospring3.ch4.xml.ConfigurableMessageProvider"
      c:message="This is a configurable message"/>
```

Чтобы использовать аннотацию для Constructor Injection, мы также применяем аннотацию `@Autowired` в методе конструктора целевого бина (листинг 4.28), что представляет собой альтернативу использованию Setter Injection, продемонстрированному в листинге 4.24.

Листинг 4.28. Использование внедрения через конструктор (стиль аннотаций)

```
package com.apress.prospring3.ch4.annotation;

import org.springframework.beans.factory.annotation.Value;
...
@Service("messageProvider")
public class ConfigurableMessageProvider implements MessageProvider {

   private String message;

   @Autowired
   public ConfigurableMessageProvider(@Value("This is a configurable message")
String message) {
      this.message = message;
   }
...
}
```

В предыдущем листинге используется еще одна аннотация — `@Value` — для определения значения, внедряемого в конструктор. Таким способом в Spring внедряются значения в бин. Кроме простых строк, можно использовать мощный язык SpEL для динамического внедрения значений (об этом пойдет речь далее в этой главе).

Однако жесткое кодирование значения не является удачной идеей, поскольку для его изменения понадобится перекомпилировать программу. Даже если выбран вариант DI с аннотациями, значения, предназначенные для внедрения, рекомендуется выносить за пределы кода. Чтобы вынести сообщение за пределы кода, давайте определим его как бин Spring в конфигурационном файле для случая аннотаций (листинг 4.29).

Листинг 4.29. Определение сообщения как бина Spring

```
<bean id="message" class="java.lang.String"
      c:_0="This is a configurable message"/>
```

В листинге 4.29 определен бин с идентификатором "`message`" и типом `java.lang.String`. Обратите внимание на использование пространства имен `c` для Constructor Injection при установке строкового значения и конструкции `_0`, указывающей индекс аргумента в конструкторе.

Имея объявленный бин, можно избавиться от аннотации `@Value` в целевом бине, как показано в листинге 4.30.

Листинг 4.30. Избавление от аннотации `@Value`

```
package com.apress.prospring3.ch4.annotation;
...
@Service("messageProvider")
public class ConfigurableMessageProvider implements MessageProvider {

   private String message;

   @Autowired
   public ConfigurableMessageProvider(String message) {
      this.message = message;
   }
...
}
```

Поскольку объявленный бин `message` и его идентификатор совпадают с именем аргумента в конструкторе, Spring обнаружит аннотацию и внедрит значение в метод конструктора.

Запустив тест (класс `UsingSetterInjection`, показанный в листинге 4.25) для конфигураций с XML (`app-context.xml.xml`) и аннотациями (`app-context-annotation.xml`), мы получаем в обоих случаях вывод сконфигурированного сообщения. Пример вывода выглядит следующим образом:

```
This is a configurable message
```

Избежание путаницы с конструкторами

В некоторых случаях Spring не имеет возможности сообщить, какой конструктор будет использоваться для Constructor Injection. Как правило, такое случается в ситуации, когда есть два конструктора с одинаковым количеством аргументов, а типы, используемые для аргументов, представлены в точности тем же самым способом. Взгляните на код в листинге 4.31.

Листинг 4.31. Путаница с конструкторами

```
package com.apress.prospring3.ch4.xml;

import org.springframework.context.support.GenericXmlApplicationContext;

public class ConstructorConfusion {

   private String someValue;

   public ConstructorConfusion(String someValue) {
      System.out.println("ConstructorConfusion(String) called");
      this.someValue = someValue;
   }

   public ConstructorConfusion(int someValue) {
      System.out.println("ConstructorConfusion(int) called");
      this.someValue = "Number: " + Integer.toString(someValue);
   }

   public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:app-context-xml.xml");
      ctx.refresh();

      ConstructorConfusion cc =
          (ConstructorConfusion) ctx.getBean("constructorConfusion");
      System.out.println(cc);
   }

   public String toString() {
      return someValue;
   }
}
```

Здесь четко видно, что этот код делает — он просто извлекает бин типа `ConstructorConfusion` из `ApplicationContext` и записывает значение в консольный вывод. Теперь посмотрим на код в листинге 4.32 (`app-context-xml.xml`).

Листинг 4.32. Файл `app-context-xml.xml`

```
<bean id="constructorConfusion"
      class="com.apress.prospring3.ch4.xml.ConstructorConfusion">
   <constructor-arg>
      <value>90</value>
   </constructor-arg>
</bean>
```

Какой конструктор будет вызван в этом случае? Запуск примера дает следующий вывод:

```
ConstructorConfusion(String) called
90
```

Вывод показывает, что был вызван конструктор с аргументом `String`. Судя по тому, что любое целочисленное значение, переданное с использованием Constructor Injection, должно предваряться `Number:`, как можно видеть в коде конструктора с аргументом `int`, это не тот результат, который ожидался. Чтобы обойти эту проблему, понадобится внести небольшое изменение в конфигурацию, как показано в листинге 4.33 (`app-context-xml.xml`).

Листинг 4.33. Ликвидация путаницы с конструкторами

```
<bean id="constructorConfusion"
     class="com.apress.prospring3.ch4.xml.ConstructorConfusion">
   <constructor-arg type="int">
      <value>90</value>
   </constructor-arg>
</bean>
```

Обратите внимание, что дескриптор `<constructor-arg>` имеет дополнительный атрибут `type`, указывающий тип аргумента, который должна искать Spring. Запуск примера теперь дает правильный вывод:

```
ConstructorConfusion(int) called
Number: 90
```

При внедрении через конструктор с использованием аннотаций аналогичной путаницы можно избежать, применяя аннотацию прямо к целевому методу конструктора, что и сделано в листинге 4.34.

Листинг 4.34. Путаница с конструкторами (стиль аннотаций)

```
package com.apress.prospring3.ch4.annotation;

// Некоторый код опущен.

@Service("constructorConfusion")
public class ConstructorConfusion {

   private String someValue;

   public ConstructorConfusion(String someValue) {
      System.out.println("ConstructorConfusion(String) called");
      this.someValue = someValue;
   }

   @Autowired
   public ConstructorConfusion(@Value("90") int someValue) {
      System.out.println("ConstructorConfusion(int) called");
      this.someValue = "Number: " + Integer.toString(someValue);
  }

// Некоторый код опущен.

}
```

За счет применения аннотации `@Autowired` к целевому методу конструктора платформа Spring будет использовать этот метод для создания бина и внедрения указанного значения. Как и ранее, это значение должно быть вынесено во внешнюю конфигурацию.

На заметку! Аннотацию `@Autowired` можно применить только к одному из методов конструктора. В случае ее применения сразу к нескольким методам конструктора, во время начальной загрузки `ApplicationContext` платформа Spring выдаст соответствующее сообщение.

Параметры внедрения

В предыдущих двух примерах было показано, как внедрять другие компоненты и значения в бин, используя Setter Injection и Constructor Injection. Платформа Spring поддерживает большое количество опций для параметров внедрения, позволяя внедрять не только другие компоненты и простые значения, но также коллекции Java, внешне определенные свойства и даже бины в другой фабрике. Все эти типы параметров внедрения могут использоваться в Setter Injection и Constructor Injection за счет добавления соответствующего дескриптора в рамках дескрипторов `<property>` и `<constructor-args>`.

Внедрение простых значений

Внедрение простых значений в бины осуществляется просто. Для этого нужно лишь указать в дескрипторе конфигурации значение, заключенное в дескриптор `<value>`. По умолчанию дескриптор `<value>` может не только читать значения `String`, но также и преобразовывать их в любой примитивный класс или оболочку над ним. В листинге 4.35 показан простой бин, который имеет разнообразные свойства, доступные для внедрения.

Листинг 4.35. Внедрение простых значений (стиль XML)

```
package com.apress.prospring3.ch4.xml;

import org.springframework.context.support.GenericXmlApplicationContext;

public class InjectSimple {

   private String name;

   private int age;

   private float height;
   private boolean programmer;
   private Long ageInSeconds;

   public static void main(String[] args) {

      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:app-context-xml.xml");
      ctx.refresh();

      InjectSimple simple = (InjectSimple)ctx.getBean("injectSimple");
      System.out.println(simple);
   }
   public void setAgeInSeconds(Long ageInSeconds) {
      this.ageInSeconds = ageInSeconds;
   }
   public void setProgrammer(boolean programmer) {
      this.programmer = programmer;
   }
   public void setAge(int age) {
      this.age = age;
   }
   public void setHeight(float height) {
      this.height = height;
   }
   public void setName(String name) {
      this.name = name;
   }
```

```
    public String toString() {
        return    "Name :" + name + "\n"
                + "Age:" + age + "\n"
                + "Age in Seconds: " + ageInSeconds + "\n"
                + "Height: " + height + "\n"
                + "Is Programmer?: " + programmer;
    }
}
```

В дополнение к свойствам в классе `InjectSimple` также определен метод `main()`, который создает `ApplicationContext` и затем извлекает бин `InjectSimple` из Spring. Значения свойств этого бина затем записываются в консольный вывод. В листинге 4.36 показана конфигурация (`app-context-xml.xml`) для этого бина.

Листинг 4.36. Конфигурирование внедрения простых значений

```
<bean id="injectSimple" class="com.apress.prospring3.ch4.xml.InjectSimple">
   <property name="name">
      <value>John Smith</value>
   </property>
   <property name="age">
      <value>35</value>
   </property>
   <property name="height">
      <value>1.78</value>
   </property>
   <property name="programmer">
      <value>true</value>
   </property>
   <property name="ageInSeconds">
      <value>1103760000</value>
   </property>
</bean>
```

В листингах 4.35 и 4.36 видно, что допускается определять свойства бина, которые принимают значения `String`, примитивные значения или оболочки примитивных значений, и затем внедрять значения для этих свойств с использованием дескриптора `<value>`. Ниже показан вывод, полученный в результате запуска этого примера:

```
Name: John Smith
Age: 35
Age in Seconds: 1103760000
Height: 1.78
Is Programmer?: true
```

Для внедрения простых значений в стиле аннотаций можно применить аннотацию `@Value` к свойствам бина. В этот раз вместо метода установки мы применим указанную аннотацию к оператору объявления свойства, как показано в листинге 4.37. (Платформа Spring поддерживает аннотации либо в методе установки, либо в свойствах.)

Листинг 4.37. Внедрение простых значений (стиль аннотаций)

```
package com.apress.prospring3.ch4.annotation;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.support.GenericXmlApplicationContext;
```

```
import org.springframework.stereotype.Service;

@Service("injectSimple")
public class InjectSimple {

   @Value("John Smith")
   private String name;

   @Value("35")
   private int age;

   @Value("1.78")
   private float height;

   @Value("true")
   private boolean programmer;

   @Value("1103760000")
   private Long ageInSeconds;

   public static void main(String[] args) {

      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:app-context-annotation.xml");
      ctx.refresh();

      InjectSimple simple = (InjectSimple)ctx.getBean("injectSimple");
      ctx.getBean("injectRef");
      System.out.println(simple);
   }

   // Методы установки/извлечения опущены.

   public String toString() {
      return   "Name :" + name + "\n"
            + "Age:" + age + "\n"
            + "Age in Seconds: " + ageInSeconds + "\n"
            + "Height: " + height + "\n"
            + "Is Programmer?: " + programmer;
   }

}
```

Полученный результат будет в точности таким же, как при использовании конфигурации с помощью XML.

Внедрение значений с использованием SpEL

В Spring 3 появилось мощное функциональное средство — язык выражений Spring (Spring Expression Language — SpEL). Язык SpEL позволяет вычислять выражения динамически и затем применять их в `ApplicationContext`. Одним из сценариев использования является внедрение результата в бины Spring. В этом разделе мы посмотрим, как применять SpEL для внедрения свойств из других бинов, используя ранее представленный пример.

Предположим, что значения, предназначенные для внедрения в бин Spring, необходимо вынести в конфигурационный класс, показанный в листинге 4.38.

Листинг 4.38. Внедрение значений с использованием SpEL (стиль XML)

```
package com.apress.prospring3.ch4.xml;

public class InjectSimpleConfig {
```

```
    private String name = "John Smith";
    private int age = 35;
    private float height = 1.78f;
    private boolean programmer = true;
    private Long ageInSeconds = 1103760000L;
    // Методы установки/извлечения опущены.
}
```

После этого можно определить бин в XML-конфигурации и применить SpEL для внедрения свойств бина в зависимый бин, как сделано в листинге 4.39 (`app-context-xml.xml`).

Листинг 4.39. Использование SpEL для внедрения значений (стиль XML)

```
<bean id="injectSimpleConfig"
      class="com.apress.prospring3.ch4.xml.InjectSimpleConfig"/>
<bean id="injectSimpleSpel"
      class="com.apress.prospring3.ch4.xml.InjectSimpleSpel">
   <property name="name">
      <value>#{injectSimpleConfig.name}</value>
   </property>
   <property name="age">
      <value>#{injectSimpleConfig.age + 1}</value>
   </property>
   <property name="height">
      <value>#{injectSimpleConfig.height}</value>
   </property>
   <property name="isProgrammer">
      <value>#{injectSimpleConfig.programmer}</value>
   </property>
   <property name="ageInSeconds">
      <value>#{injectSimpleConfig.ageInSeconds}</value>
   </property>
</bean>
```

Обратите внимание на использование SpEL-выражения `#{injectSimpleConfig.name}` в ссылке на свойство другого бина. Для возраста (`age`) мы добавляем 1 к значению из бина, чтобы продемонстрировать возможность применения SpEL для манипулирования свойством и внедрения в зависимый бин. Давайте протестируем конфигурацию с помощью программы, показанной в листинге 4.40.

Листинг 4.40. Тестирование внедрения значений с использованием SpEL

```
package com.apress.prospring3.ch4.xml;
import org.springframework.context.support.GenericXmlApplicationContext;
public class InjectSimpleSpel {
    private String name;
    private int age;
    private float height;
    private boolean programmer;
```

```
    private Long ageInSeconds;

    public static void main(String[] args) {

        GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
        ctx.load("classpath:app-context-xml.xml");
        ctx.refresh();

        InjectSimpleSpel simple =
           (InjectSimpleSpel)ctx.getBean("injectSimpleSpel");
        System.out.println(simple);
    }
// Остальной код опущен.
}
```

Вывод этой программы выглядит следующим образом:

```
Name: John Smith
Age:36
Age in Seconds: 1103760000
Height: 1.78
Is Programmer?: true
```

При использовании внедрения значений в стиле аннотаций понадобится только добавить аннотации `@Value` с выражениями SpEL (листинг 4.41).

Листинг 4.41. Внедрение значений с использованием SpEL (стиль аннотаций)

```
package com.apress.prospring3.ch4.annotation;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.support.GenericXmlApplicationContext;
import org.springframework.stereotype.Service;

@Service("injectSimpleSpel")
public class InjectSimpleSpel {

    @Value("#{injectSimpleConfig.name}")
    private String name;

    @Value("#{injectSimpleConfig.age + 1}")
    private int age;

    @Value("#{injectSimpleConfig.height}")
    private float height;

    @Value("#{injectSimpleConfig.programmer}")
    private boolean programmer;

    @Value("#{injectSimpleConfig.ageInSeconds}")
    private Long ageInSeconds;

// Остальной код опущен.
}
```

В листинге 4.42 приведена версия класса `InjectSimpleConfig` с аннотациями.

Листинг 4.42. Класс `InjectSimpleConfig` (версия с аннотациями)

```
package com.apress.prospring3.ch4.annotation;

import org.springframework.stereotype.Component;

@Component("injectSimpleConfig")
```

```
public class InjectSimpleConfig {
   private String name = "John Smith";
   private int age = 35;
   private float height = 1.78f;
   private boolean programmer = true;
   private Long ageInSeconds = 1103760000L;

   // Методы установки/извлечения опущены.
}
```

В листинге 4.42 вместо аннотации `@Service` используется аннотация `@Component`. В основном применение `@Component` дает тот же самый эффект, что и `@Service`. Обе аннотации инструктируют Spring о том, что аннотированный класс является кандидатом на автоматическое обнаружение с применением конфигурации, основанной на аннотациях, и сканирования в пути классов. Однако поскольку класс `InjectSimpleConfig` хранит конфигурацию приложения, а не предоставляет бизнес-службу, использование `@Component` имеет больший смысл. На практике `@Service` является специализацией `@Component`, отражающей тот факт, что аннотированный класс предоставляет бизнес-службу другим уровням в рамках приложения.

Тестирование полученной программы даст тот же результат. Используя SpEL, можно получать доступ к любым управляемым Spring бинам и свойствам, а также манипулировать ими в приложении за счет поддержки в Spring развитых языковых средств и синтаксиса.

Внедрение бинов в одной и той же единице XML

Как уже было показано, один бин можно внедрять в другой, используя для этого дескриптор `<ref>`. В листинге 4.43 приведен класс, который предлагает метод установки, чтобы разрешить внедрение бина.

Листинг 4.43. Внедрение бина

```
package com.apress.prospring3.ch4.xml;

import org.springframework.context.support.GenericXmlApplicationContext;
import com.apress.prospring3.ch4.Oracle;

public class InjectRef {
   private Oracle oracle;
   public void setOracle(Oracle oracle) {
      this.oracle = oracle;
   }
   public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:app-context-xml.xml");
      ctx.refresh();
      InjectRef injectRef = (InjectRef) ctx.getBean("injectRef");
      System.out.println(injectRef);
   }
   public String toString() {
      return oracle.defineMeaningOfLife();
   }
}
```

Чтобы настроить Spring для внедрения одного бина в другой, сначала понадобится сконфигурировать два бина: один в качестве внедряемого и еще один в качестве

цели для внедрения. После того, как это сделано, останется только сконфигурировать внедрение с использованием дескриптора `<ref>` для целевого бина. Вспомните, что дескриптор `<ref>` должен располагаться либо внутри `<property>`, либо внутри `<constructor-arg>`, в зависимости от типа применяемого внедрения — Setter Injection или Constructor Injection.

В листинге 4.44 показан пример такой конфигурации (`app-context-xml.xml`).

Листинг 4.44. Конфигурирование внедрения бина

```
<bean id="oracle" name="wiseworm"
      class="com.apress.prospring3.ch4.BookwormOracle"/>
<bean id="injectRef" class="com.apress.prospring3.ch4.xml.InjectRef">
   <property name="oracle">
      <ref local="oracle"/>
   </property>
</bean>
```

Запуск класса, показанного в листинге 4.43, даст следующий вывод:

```
Encyclopedias are a waste of money - use the Internet
```

Здесь важно отметить один момент: внедряемый тип не должен в точности совпадать с тем типом, который определен в качестве целевого; эти типы просто должны быть совместимыми. *Совместимость* означает, что если объявленный в качестве целевого тип является интерфейсом, то внедряемый тип должен реализовать этот интерфейс. Если объявленный тип является классом, то внедряемый тип должен относиться либо к тому же типу, либо к его подтипу. В этом примере класс `InjectRef` определяет метод `setOracle()`, предназначенный для получения экземпляра `Oracle`, который представляет собой интерфейс, а внедряемым типом является `BookwormOracle` — класс, реализующий `Oracle`. Этот момент может привести в замешательство некоторых разработчиков, хотя в действительности здесь нет ничего сложного. Внедрение подчиняется тем же правилами типизации, что и любой Java-код, поэтому если вы знаете, как работает типизация в Java, то должны понять, что типизация при внедрении довольно проста.

В предыдущем примере идентификатор (`id`) бина, предназначенного для внедрения, был указан с использованием атрибута `local` дескриптора `<ref>`. Как будет показано позже, в разделе “Именование бинов”, бину можно назначить более одного имени, поэтому ссылаться на него можно с помощью множества псевдонимов. Применение атрибута `local` означает, что дескриптор `<ref>` всегда просматривает только идентификатор бина и никогда не принимает во внимание его псевдонимы. Более того, определение бина должно существовать в том же самом XML-файле конфигурации. Чтобы внедрить бин по любому имени или импортировать бин из другого XML-файла конфигурации, вместо атрибута `local` в дескрипторе `<ref>` необходимо применять атрибут `bean`. В листинге 4.45 показана альтернативная конфигурация для предыдущего примера, в которой используется альтернативное имя для внедряемого бина.

Листинг 4.45. Внедрение с использованием псевдонимов бина

```
<bean id="oracle" name="wiseworm"
      class="com.apress.prospring.ch4.BookwormOracle"/>
<bean id="injectRef" class="com.apress.prospring3.ch4.xml.InjectRef">
   <property name="oracle">
      <ref bean="wiseworm"/>
   </property>
</bean>
```

В этом примере бину oracle с помощью атрибута name назначается псевдоним, после чего этот бин внедряется в бин injectRef с указанием псевдонима в атрибуте bean дескриптора <ref>. Не следует особо переживать по поводу семантики именования — мы подробно обсудим ее далее в этой главе. Запуск класса InjectRef, как показано в листинге 4.43, даст тот же результат, что и предыдущий пример.

Внедрение и вложение ApplicationContext друг в друга

До сих пор внедряемые бины располагались в том же самом контексте Application Context (и, следовательно, в той же самой фабрике BeanFactory), что и бины, в которые производилось внедрение. Тем не менее, Spring поддерживает иерархическую структуру для ApplicationContext, когда один контекст (и связанная с ним фабрика BeanFactory) является родительским для другого контекста. Разрешая вложение ApplicationContext, платформа Spring позволяет разделять конфигурацию на отдельные файлы — настоящая находка для крупных проектов, содержащих множество бинов.

При вложении ApplicationContext платформа Spring позволяет бинам в контексте, который считается дочерним, ссылаться на бины в родительском контексте. Вложение ApplicationContext с использованием GenericXmlApplicationContext осуществляется очень просто. Чтобы вложить один контекст GenericXmlApplicationContext в другой, необходимо просто вызвать метод setParent() в дочернем контексте ApplicationContext, как показано в листинге 4.46.

Листинг 4.46. Вложение GenericXmlApplicationContext

```
package com.apress.prospring3.ch4;

import org.springframework.context.support.GenericXmlApplicationContext;

public class HierarchicalAppContextUsage {

   public static void main(String[] args) {

      GenericXmlApplicationContext parent =
        new GenericXmlApplicationContext();
      parent.load("classpath:parent.xml");
      parent.refresh();

      GenericXmlApplicationContext child =
        new GenericXmlApplicationContext();
      child.load("classpath:app-context-xml.xml");
      child.setParent(parent);
      child.refresh();

      SimpleTarget target1 = (SimpleTarget) child.getBean("target1");
      SimpleTarget target2 = (SimpleTarget) child.getBean("target2");
      SimpleTarget target3 = (SimpleTarget) child.getBean("target3");

      System.out.println(target1.getVal());
      System.out.println(target2.getVal());
      System.out.println(target3.getVal());

   }

}
```

В листинге 4.47 представлен класс SimpleTarget.

Листинг 4.47. Класс SimpleTarget

```
package com.apress.prospring3.ch4;

public class SimpleTarget {

   private String val;

   public void setVal(String val) {
      this.val = val;
   }

   public String getVal() {
      return val;
   }
}
```

Внутри конфигурационного файла для дочернего контекста ApplicationContext ссылка на какой-нибудь бин в родительском контексте ApplicationContext работает в точности подобно ссылке на бин в дочернем контексте ApplicationContext, если только в дочернем контексте ApplicationContext отсутствует бин с таким же именем. В таком случае нужно просто заменить атрибут bean дескриптора <ref> атрибутом parent. В листинге 4.48 показан пример конфигурационного файла для родительского контекста BeanFactory (parent.xml).

Листинг 4.48. Конфигурация родительского контекста ApplicationContext

```
<bean id="injectBean" class="java.lang.String">
   <constructor-arg>
      <value>Bean In Parent</value>
   </constructor-arg>
</bean>
<bean id="injectBeanParent" class="java.lang.String">
   <constructor-arg>
      <value>Bean In Parent</value>
   </constructor-arg>
</bean>
```

Как видите, в этой конфигурации просто определены два бина: injectBean и injectBeanParent. Оба представляют собой объекты String со значениями Bean In Parent (Бин в родительском контексте). В листинге 4.49 приведен пример конфигурации для дочернего контекста ApplicationContext (app-context-xml.xml).

Листинг 4.49. Конфигурация дочернего контекста ApplicationContext

```
<bean id="target1" class="com.apress.prospring3.ch4.SimpleTarget">
   <property name="val">
      <ref bean="injectBeanParent"/>
   </property>
</bean>

<bean id="target2" class="com.apress.prospring3.ch4.SimpleTarget">
   <property name="val">
      <ref bean="injectBean"/>
   </property>
</bean>
```

```
<bean id="target3" class="com.apress.prospring3.ch4.SimpleTarget">
   <property name="val">
      <ref parent="injectBean"/>
   </property>
</bean>

<bean id="injectBean" class="java.lang.String">
   <constructor-arg>
      <value>Bean In Child</value>
   </constructor-arg>
</bean>
```

Обратите внимание, что здесь определены четыре бина. Бин `injectBean` в этом листинге похож на бин в `injectBean` в родительском контексте за исключением отличия в значении `String`, которое отражает его размещение в дочернем контексте `ApplicationContext` (строка `Bean In Child`).

Бин `target1` использует атрибут `bean` дескриптора `<ref>` для ссылки на бин по имени `injectBeanParent`. Поскольку этот бин существует только в родительской фабрике `BeanFactory`, бин `target1` получает ссылку на него. Здесь интересно отметить два момента. Первый: атрибут `bean` можно использовать для ссылки на бины как в дочернем, так и в родительском контекстах `ApplicationContext`. Это позволяет ссылаться на бины прозрачным образом, позволяя перемещать бины между конфигурационными файлами по мере роста приложения. Второй момент связан с невозможностью применения атрибута `local` для ссылки на бины в родительском контексте `ApplicationContext`. Анализатор XML проверяет, существует ли в том же самом файле допустимый элемент, который соответствует значению, указанному в атрибуте `local`, предотвращая его использование для ссылки на бины в родительском контексте.

Бин `target2` применяет атрибут `bean` дескриптора `<ref>` для ссылки на `injectBean`. Поскольку этот бин определен в обоих контекстах `ApplicationContext`, бин `target2` получает ссылку на `injectBean` в собственном контексте `ApplicationContext`.

Бин `target3` использует атрибут `parent` дескриптора `<ref>` для ссылки на бин `injectBean` напрямую в родительском контексте `ApplicationContext`. Так как в `target3` применяется атрибут `parent` дескриптора `<ref>`, бин `injectBean`, объявленный в дочернем контексте `ApplicationContext`, полностью игнорируется.

Код в листинге 4.46 также демонстрирует обсуждаемую здесь семантику, извлекая каждый из трех бинов `targetX` из дочерней фабрики `BeanFactory` и в каждом случае выводя значение свойства `val`.

Ниже показан вывод, полученный в результате запуска класса `HierarchicalAppContextUsage` (см. листинг 4.46):

```
Bean In Parent
Bean In Child
Bean In Parent
```

Как и ожидалось, бины `target1` и `target3` получают ссылку на бины в родительском контексте `ApplicationContext`, тогда как бин `target2` — ссылку на бин в дочернем контексте `ApplicationContext`.

Использование коллекций для внедрения

Часто бинам необходим доступ к коллекциям объектов, а не только к отдельным бинам или значениям. Таким образом, не должно стать сюрпризом, что Spring позволяет внедрять коллекцию объектов в бины. Использовать коллекцию несложно: необходимо выбрать дескриптор `<list>`, `<map>`, `<set>` или `<props>` для представления экземпля-

ра List, Map, Set или Properties, а затем передавать индивидуальные элементы, как это делается при любом другом внедрении. Дескриптор <props> позволяет передавать в качестве значений только String, потому что класс Properties разрешает только значения свойств типа String. Для <list>, <map> или <set> можно использовать любой дескриптор при внедрении в свойство, даже дескриптор другой коллекции. Это позволяет передавать List из Map, Map из Set или даже List, состоящий из элементов Map, каждый из которых состоит из элементов Set, а те, в свою очередь, из элементов List!

В листинге 4.50 показан класс, в который можно внедрять все четыре типа коллекций.

Листинг 4.50. Внедрение коллекции (стиль XML)

```
package com.apress.prospring3.ch4.xml;

import java.util.List;
import java.util.Map;
import java.util.Properties;
import java.util.Set;

import org.springframework.context.support.GenericXmlApplicationContext;
public class CollectionInjection {

   private Map<String, Object> map;

   private Properties props;

   private Set set;

   private List list;

   public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:app-context-xml.xml");
      ctx.refresh();

      CollectionInjection instance =
         (CollectionInjection) ctx.getBean("injectCollection");
      instance.displayInfo();
   }

   public void setList(List list) {
      this.list = list;
   }

   public void setSet(Set set) {
      this.set = set;
   }

   public void setMap(Map <String, Object> map) {
      this.map = map;
   }

   public void setProps(Properties props) {
      this.props = props;
   }

   public void displayInfo() {

      // Отобразить карту.
      System.out.println("Map contents:\n");
      for (Map.Entry<String, Object> entry: map.entrySet()) {
         System.out.println("Key: " + entry.getKey() + " - Value: "
                            + entry.getValue());
      }
```

```
        // Отобразить свойства.
        System.out.println("\nProperties contents:\n");
        for (Map.Entry<Object, Object> entry: props.entrySet()) {
            System.out.println("Key: " + entry.getKey() + " - Value: "
                               + entry.getValue());
        }
        // Отобразить набор.
        System.out.println("\nSet contents:\n");
        for (Object obj: set) {
            System.out.println("Value: " + obj);
        }
        // Отобразить список.
        System.out.println("\nList contents:\n");
        for (Object obj: list) {
            System.out.println("Value: " + obj);
        }
    }
}
```

Здесь довольно много кода, но в действительности он мало что делает. Метод `main()` извлекает бин `CollectionInjection` из Spring и затем вызывает метод `displayInfo()`. Этот метод просто выводит содержимое экземпляров `Map`, `Properties`, `Set` и `List`, которые будут внедрены из Spring. В листинге 4.51 приведена конфигурация, требуемая для внедрения значений каждого свойства класса `CollectionInjection`.

Также обратите внимание на объявление свойства `Map<String,Object>`. Для JDK 5 и более новых версий Spring дополнительно поддерживает строго типизированное определение `Collection`, и будет выполнять преобразование из XML-конфигурации в соответствующий указанный тип (`app-context-xml.xml`).

Листинг 4.51. Конфигурирование внедрения коллекции (стиль XML)

```
<bean id="oracle" name="wiseworm" class="com.apress.prospring3.ch4.
BookwormOracle"/>

<bean id="injectCollection" class="com.apress.prospring3.ch4.xml.
CollectionInjection">
        <property name="map">
            <map>
                <entry key="someValue">
                    <value>Hello World!</value>
                </entry>
                <entry key="someBean">
                    <ref local="oracle"/>
                </entry>
            </map>
        </property>
        <property name="props">
            <props>
                <prop key="firstName">Clarence</prop>
                <prop key="secondName">Ho</prop>
            </props>
        </property>
```

```
    <property name="set">
       <set>
          <value>Hello World!</value>
          <ref local="oracle"/>
       </set>
    </property>
    <property name="list">
       <list>
          <value>Hello World!</value>
          <ref local="oracle"/>
       </list>
    </property>
</bean>
```

В этом коде значение внедряется во все четыре метода установки, доступные в классе `CollectionInjection`. Для свойства `map` мы внедряем экземпляр `Map`, используя дескриптор `<map>`. Каждый элемент карты указывается с помощью дескриптора `<entry>`, и каждый элемент имеет ключ `String` и значение. Этим значением может быть любое значение, которое можно внедрить в свойство отдельно; в приведенном примере демонстрируется использование дескрипторов `<value>` и `<ref>` для добавления к `Map` значения `String` и ссылки на бин. Для свойства `props` мы применяем дескриптор `<props>`, чтобы создать экземпляр `java.util.Properties` и наполнить его с помощью дескрипторов `<prop>`. Обратите внимание, что хотя дескриптор `<prop>` снабжается ключами подобно дескриптору `<entry>`, указывать значение `String` можно только для каждого свойства, относящегося к экземпляру `Properties`.

Оба дескриптора, `<list>` и `<set>`, работают совершенно одинаково: каждый элемент указывается с использованием любых дескрипторов значений, таких как `<value>` и `<ref>`, которые предназначены для внедрения одиночного значения в свойство. В листинге 4.51 к `List` и `Set` были добавлены значение `String` и ссылка на бин.

Ниже показан вывод, сгенерированный классом из листинга 4.50. Как и ожидалось, он просто выводит список элементов, добавленных в коллекцию в конфигурационном файле.

```
Map contents:

Key: someValue - Value: Hello World!
Key: someBean - Value: com.apress.prospring3.ch4.xml.BookwormOracle@2a0ab444

Properties contents:

Key: secondName - Value: Ho
Key: firstName - Value: Clarence

Set contents:

Value: Hello World!
Value: com.apress.prospring3.ch4.xml.BookwormOracle@2a0ab444

List contents:

Value: Hello World!
Value: com.apress.prospring3.ch4.xml.BookwormOracle@2a0ab444
```

Вспомните, что с элементами `<list>`, `<map>` и `<set>` можно использовать любой из дескрипторов, которые применяются при установке значений свойств, отличных от коллекций, для указания значения одного из элементов в коллекции. Это очень мощная концепция, поскольку вы не ограничены одним лишь внедрением коллекций примитивных значений; вы можете также внедрять коллекции бинов или другие коллекции.

Используя эту функциональность, намного проще разбить разрабатываемое приложение на модули и предоставить различные выбираемые пользователем реализации ключевых частей логики приложения. Подумайте о системе, которая позволяет корпоративным сотрудникам создавать, корректировать и заказывать персонализированные канцелярские принадлежности в онлайновом режиме. В этой системе построенный изображение каждого заказа отправляется на подходящий принтер, когда он готов к работе. Единственная сложность связана с тем, что некоторые принтеры ожидают получения изображений по электронной почте, другие — по FTP, а большинство используют протокол защищенного копирования (Secure Copy Protocol — SCP). С помощью внедрения коллекций Spring можно создать стандартный интерфейс для этой функциональности, который приведен в листинге 4.52.

Листинг 4.52. Интерфейс `ArtworkSender`

```
package com.apress.prospring3.ch4;

public interface ArtworkSender {

   public void sendArtwork(String artworkPath, Recipient recipient);

   public String getFriendlyName();

   public String getShortName();
}
```

В листинге 4.52 класс `Recipient` является пустым. Для интерфейса `ArtworkSender` можно создавать множество реализаций, одна из которых показана в листинге 4.53.

Листинг 4.53. Класс `FtpArtworkSender`

```
package com.apress.prospring3.ch4;

public class FtpArtworkSender implements ArtworkSender {

   public void sendArtwork(String artworkPath, Recipient recipient) {
      // Логика работы с ftp...
   }
   public String getFriendlyName() {
      return "File Transfer Protocol";
   }
   public String getShortName() {
      return "ftp";
   }
}
```

Предположим, что затем разрабатывается класс `ArtworkManager`, который поддерживает все доступные реализации интерфейса `ArtworkSender`. Имея такие реализации, вы просто передаете `List` в класс `ArtworkManager`. Используя метод `getFriendlyName()`, системному администратору можно отобразить список вариантов доставки для выбора. Вдобавок ваше приложение может оставаться полностью несвязанным, если вы просто кодируете интерфейс `ArtworkSender`. Реализацию класса `ArtworkManager` мы оставляем в качестве упражнения для самостоятельной проработки.

Кроме XML-конфигурации для внедрения коллекций можно также использовать аннотации. Однако для простоты сопровождения имеет смысл также вынести значения коллекций в конфигурационный файл. В листинге 4.54 показана конфигурация для четырех разных бинов Spring, которые имитируют те же свойства коллекций, что и предыдущий пример (`app-context-annotation.xml`).

Листинг 4.54. Конфигурирование внедрения коллекции (стиль аннотаций)

```
<util:map id="map" map-class="java.util.HashMap">
   <entry key="someValue">
      <value>Hello World!</value>
   </entry>
   <entry key="someBean">
      <ref bean="oracle"/>
   </entry>
</util:map>

<util:properties id="props">
   <prop key="firstName">Clarence</prop>
   <prop key="secondName">Ho</prop>
</util:properties>

<util:set id="set">
   <value>Hello World!</value>
   <ref bean="oracle"/>
</util:set>

<util:list id="list">
   <value>Hello World!</value>
   <ref bean="oracle"/>
</util:list>
```

Давайте также разработаем версию класса `BookwormOracle` с аннотациями. Код показано в листинге 4.55.

Листинг 4.55. Класс `BookwormOracle`

```
package com.apress.prospring3.ch4.annotation;

import org.springframework.stereotype.Service;

import com.apress.prospring3.ch4.Oracle;

@Service("oracle")
public class BookwormOracle implements Oracle {

   public String defineMeaningOfLife() {
      return "Encyclopedias are a waste of money - use the Internet";
   }

}
```

В конфигурации, показанной в листинге 4.54, мы используем пространство имен `util`, предоставляемое Spring, для объявления бинов, хранящих свойства коллекций. По сравнению с предшествующими версиями Spring конфигурация значительно упростилась. В тестовом классе мы внедряем ранее показанные бины и применяем аннотации `@Resource` из стандарта JSR-250 с указанием в них имен (листинг 4.56).

Листинг 4.56. Применение аннотации `@Resource` при внедрении коллекций

```
package com.apress.prospring3.ch4.annotation;

// Остальной код опущен.
@Service("injectCollection")
```

```
public class CollectionInjection {

    @Resource(name="map")
    private Map<String, Object> map;

    @Resource(name="props")
    private Properties props;

    @Resource(name="set")
    private Set set;

    @Resource(name="list")
    private List list;

// Остальной код опущен.
}
```

Запуск тестовой программы дает тот же самый результат, что и пример, в котором используется XML-конфигурация.

На заметку! Вас может удивить, почему вместо `@Autowired` использовалась аннотация `@Resource`. Причина в том, что аннотация `@Autowired` семантически определена так, что она всегда трактует массивы, коллекции и карты как наборы соответствующих бинов, с целевым типом бина, производным от объявленного типа значений в коллекции. Таким образом, например, если класс имеет атрибут типа `List<Oracle>` и определена аннотация `@Autowired`, платформа Spring попытается внедрить все бины типа `Oracle` из текущего `ApplicationContext` в этот атрибут (вместо `<util:list>`, объявленного в конфигурационном файле), что приведет либо к внедрению неожиданных зависимостей, либо к генерации платформой Spring исключения при отсутствии бинов типа `Oracle`. В результате для внедрения типа коллекции понадобится явно сообщить Spring о необходимости выполнить внедрение, указав имя бина, что и поддерживается аннотацией `@Resource`.

Использование внедрения через метод

Кроме внедрения через конструктор и метод установки Spring предоставляет еще одно менее часто используемое средство DI, которое называется внедрением через метод (Method Injection). Возможности Method Injection доступны в двух слабо связанных формах — Lookup Method Injection (Внедрение через метод поиска) и Method Replacement (Замена метода). Форма Lookup Method Injection поддерживает еще один механизм, с помощью которого бин может получить одну из своих зависимостей, а форма Method Replacement позволяет заменять реализацию любого метода бина произвольным образом, без необходимости в изменении первоначального исходного кода.

Для обеспечения этих двух средств Spring использует возможности динамического расширения байт-кода, предлагаемые библиотекой CGLIB. Чтобы работать с Lookup Method Injection или Method Replacement в приложении, понадобится включить JAR-файл CGLIB в путь классов.

Тип Lookup Method Injection

Тип Lookup Method Injection был добавлен в Spring версии 1.1 для преодоления проблем, возникающих, когда один бин зависит от другого бина с отличающимся жизненным циклом — в частности, когда одиночный объект (singleton) зависит от неодиночного объекта (nonsingleton). В такой ситуации Setter Injection и Constructor Injection приводят к тому, что одиночный объект хранится в одном экземпляре того, что должно быть неодиночным бином. В некоторых случаях одиночный бин должен при необходимости получать новый экземпляр неодиночного бина.

Рассмотрим сценарий, при котором имеется класс LockOpener, предоставляющий службу открытия любого шкафчика. Для открытия шкафчика LockOpener полагается на класс KeyHelper, который внедрен в LockOpener. Однако проектное решение класса KeyHelper предусматривает наличие ряда внутренних состояний, что делает этот класс непригодным для повторного использования. Каждый раз, когда вызывается метод openLock(), требуется новый экземпляр KeyHelper. В этом случае LockOpener будет одиночным объектом. Однако при внедрении класса KeyHelper с применением нормального механизма будет повторно использоваться тот же самый экземпляр класса KeyHelper (который был создан, когда платформа Spring выполняла внедрение в первый раз). Чтобы обеспечить передачу нового экземпляра KeyHelper в метод openLock() при каждом его вызове, необходимо применять внедрение типа Lookup Method Injection.

Обычно этого можно добиться, заставив одиночный бин реализовать интерфейс ApplicationContextAware (который рассматривается в следующей главе). После этого, используя экземпляр ApplicationContext, одиночный бин может искать новый экземпляр неодиночной зависимости каждый раз, когда он в ней нуждается. Lookup Method Injection позволяет одиночному бину объявить, что ему требуется неодиночная зависимость, и что он получает новый экземпляр неодиночного бина всегда, когда ему нужно взаимодействовать с ним, без необходимости в реализации любого интерфейса, специфичного для Spring.

Lookup Method Injection работает за счет объявления в одиночном бине метода поиска, который возвращает экземпляр неодиночного бина. При получении ссылки на одиночный бин внутри приложения в действительности получается ссылка на динамически созданный подкласс, в котором Spring реализует метод поиска. Типовая реализация включает определение метода поиска как абстрактного. Это предотвращает возникновение странных ошибок в ситуации, когда вы забыли сконфигурировать Method Injection и работаете напрямую с классом бина, имеющим пустую реализацию метода, а не с подклассом, расширенным Spring. Это достаточно сложная тема, которую лучше проиллюстрировать примером.

В этом примере мы создаем неодиночный бин и два одиночных бина, реализующие тот же самый интерфейс. Первый одиночный бин получает экземпляр неодиночного бина, используя “традиционный” подход с Setter Injection, а второй одиночный бин применяет Method Injection. В листинге 4.57 показан бин MyHelper, который в данном примере играет роль неодиночного бина.

Листинг 4.57. Бин MyHelper

```
package com.apress.prospring3.ch4.mi;
public class MyHelper {
   public void doSomethingHelpful() {
      // Какая-то обработка.
   }
}
```

Этот бин не является особо интересным, но он прекрасно подходит для целей рассматриваемого примера. В листинге 4.58 приведен интерфейс DemoBean, который реализуют оба одиночных бина.

Листинг 4.58. Интерфейс DemoBean

```
package com.apress.prospring3.ch4.mi;
public interface DemoBean {
   public MyHelper getMyHelper();
   public void someOperation();
}
```

Интерфейс DemoBean имеет два метода: getMyHelper() и someOperation(). В примере приложения метод getMyHelper() применяется для получения ссылки на экземпляр MyHelper и, в случае бина с методом поиска, для выполнения действительного поиска. someOperation() — это простой метод, который при своей работе зависит от класса MyHelper.

В листинге 4.59 показан класс StandardLookupDemoBean, который использует Setter Injection для получения экземпляра класса MyHelper.

Листинг 4.59. Класс StandardLookupDemoBean

```
package com.apress.prospring3.ch4.mi;

public class StandardLookupDemoBean implements DemoBean {

   private MyHelper myHelper;

   public void setMyHelper(MyHelper myHelper) {
      this.myHelper = myHelper;
   }

   public MyHelper getMyHelper() {
      return this.myHelper;
   }

   public void someOperation() {
      myHelper.doSomethingHelpful();
   }
}
```

Этот код должен выглядеть знакомо, но обратите внимание, что метод someOperation() для реализации своей обработки использует сохраненный экземпляр MyHelper. В листинге 4.60 приведен класс AbstractLookupDemoBean, который применяет Method Injection для получения экземпляра класса MyHelper.

Листинг 4.60. Класс AbstractLookupDemoBean

```
package com.apress.prospring3.ch4.mi;

public abstract class AbstractLookupDemoBean implements DemoBean {

   public abstract MyHelper getMyHelper();

   public void someOperation() {
      getMyHelper().doSomethingHelpful();
   }
}
```

Метод getMyHelper() объявлен как абстрактный, и он вызывается методом someOperation() для получения экземпляра MyHelper. В листинге 4.61 показана конфигурация, необходимая для этого примера (lookup.xml).

Листинг 4.61. Конфигурирование внедрения через метод поиска

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd">
```

```
   <bean id="helper"
         class="com.apress.prospring3.ch4.mi.MyHelper" scope="prototype"/>
   <bean id="abstractLookupBean"
         class="com.apress.prospring3.ch4.mi.AbstractLookupDemoBean">
      <lookup-method name="getMyHelper" bean="helper"/>
   </bean>
   <bean id="standardLookupBean"
         class="com.apress.prospring3.ch4.mi.StandardLookupDemoBean">
      <property name="myHelper">
         <ref local="helper"/>
      </property>
   </bean>
</beans>
```

Конфигурация для вспомогательного бина и бина `standardLookupBean` должна уже выглядеть знакомой. Для `abstractLookupBean` необходимо сконфигурировать метод поиска, используя дескриптор `<lookup-method>`.

Атрибут `name` дескриптора `<lookup-method>` сообщает платформе Spring имя метода бина, который ей следует переопределить. Этот метод не должен принимать аргументы, а возвращаемым типом должен быть бин, который нужно вернуть из метода. В данном случае метод должен возвращать класс `MyHelper` или какой-то из его подклассов. Атрибут `bean` указывает Spring, какой бин должен возвращать метод поиска.

Оставшаяся часть кода этого примера приведена в листинге 4.62.

Листинг 4.62. Класс `LookupDemo`

```
package com.apress.prospring3.ch4.mi;

import org.springframework.context.support.GenericXmlApplicationContext;
import org.springframework.util.StopWatch;

public class LookupDemo {
   public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:lookup.xml");
      DemoBean abstractBean = (DemoBean) ctx.getBean("abstractLookupBean");
      DemoBean standardBean = (DemoBean) ctx.getBean("standardLookupBean");
      displayInfo(standardBean);
      displayInfo(abstractBean);
   }
   public static void displayInfo(DemoBean bean) {
      MyHelper helper1 = bean.getMyHelper();
      MyHelper helper2 = bean.getMyHelper();
      System.out.println("Helper Instances the Same?: "
                     + (helper1 == helper2));
      StopWatch stopWatch = new StopWatch();
      stopWatch.start("lookupDemo");
      for (int x = 0; x < 100000; x++) {
         MyHelper helper = bean.getMyHelper();
         helper.doSomethingHelpful();
      }
      stopWatch.stop();
      System.out.println("100000 gets took "
                        + stopWatch.getTotalTimeMillis() + " ms");
   }
}
```

В показанном коде мы извлекаем abstractLookupBean (создание экземпляра этого абстрактного класса поддерживается только в случае использования внедрения через метод поиска, при котором Spring применяет CGLIB для генерации подкласса AbstractLookupDemoBean, динамически переопределяющего этот метод) и standardLookupBean из GenericXMLApplicationContext и передаем каждую ссылку методу displayInfo(). В первой части метода displayInfo() создаются две локальных переменных MyHelper, каждой из которых присваивается значение за счет вызова метода getMyHelper() на переданном бине. Используя эти две переменных, он записывает в stdout сообщение, которое позволяет понять, указывают ли две ссылки на один и тот же объект. В случае класса abstractLookupBean новый экземпляр MyHelper должен извлекаться для каждого вызова getMyHelper(), поэтому ссылки не должны быть одинаковыми. В случае класса standardLookupBean бину передается одиночный экземпляр MyHelper с помощью Setter Injection, и данный экземпляр сохраняется и возвращается для каждого вызова getMyHelper(), поэтому две ссылки должны совпадать.

На заметку! Класс StopWatch, используемый в предыдущем примере, является служебным классом, доступным в Spring. Вы сочтете класс StopWatch очень полезным в ситуациях, когда необходимо выполнять простые тесты производительности разрабатываемых приложений.

Финальная часть метода displayInfo() запускает простой тест производительности для выяснения, какой из бинов быстрее. Очевидно, что standardLookupBean должен быть быстрее, потому что каждый раз он возвращает один и тот же экземпляр, но нам интересно увидеть эту разницу.

Перед запуском примера понадобится добавить к проекту зависимость CGLIB, как показано в табл. 4.1. За детальными сведениями о добавлении зависимостей к проекту в STS обращайтесь в приложение A.

Таблица 4.1. Зависимость для внедрения через метод

Идентификатор группы	Идентификатор артефакта	Версия	Описание
cglib	cglib	2.2.2	Библиотека генерации кода, требуемая Spring для внедрения через метод

Теперь можно запустить класс LookupDemo (см. листинг 4.62) для тестирования. Ниже показан результирующий вывод:

```
Helper Instances the Same?: true
100000 gets took 3 ms
Helper Instances the Same?: false
100000 gets took 367 ms
```

Как видите, экземпляры вспомогательного бина, как и ожидалось, одинаковы в случае использования standardLookupBean и отличаются в случае применения abstractLookupBean. Разница в производительности значительна, но это также ожидалось.

Соображения по поводу внедрения через метод поиска

Внедрение через метод поиска (Method Lookup Injection) предназначено для использования в ситуации, когда нужно работать с двумя бинами, имеющими разные жизненные циклы. Не поддавайтесь искушению применять Method Lookup Injection, если бины разделяют один и тот же жизненный цикл, особенно когда они являются одиночными. Класс в листинге 4.62 демонстрирует заметную разницу в производительности между случаями использования Method Injection для получения новых экземпляров зависимо-

сти и применения стандартного DI для получения одиночного экземпляра зависимости. Кроме того, внедрение через метод поиска не следует применять без нужды, даже если имеются бины с разными жизненными циклами.

Представим ситуацию, при которой имеются три одиночных бина, которые разделяют общую зависимость. Каждый одиночный бин должен иметь собственный экземпляр зависимости, поэтому зависимость создается как неодиночный класс, но требуется, чтобы каждый одиночный бин использовал один и тот же экземпляр коллаборатора на протяжении всего времени жизни. В этом случае идеальным решением будет Setter Injection, а Method Lookup Injection просто привнесет ненужные накладные расходы.

При использовании Method Lookup Injection имеется несколько принципов проектирования, которые должны быть приняты во внимание во время построения классов. В показанных ранее примерах мы объявляли метод поиска в интерфейсе. Единственная причина такого решения состоит в том, что мы не хотим повторять метод `displayInfo()` дважды в двух разных типах бинов. Как уже упоминалось, в общем случае вы не должны засорять бизнес-интерфейс ненужными определениями, которые предназначены исключительно для целей IoC. Другой момент, о котором следует помнить, заключается в том, что хотя делать метод поиска абстрактным не обязательно, это поможет не забыть сконфигурировать метод поиска, чтобы случайно не пользоваться пустой его реализацией.

Тип Method Replacement

Несмотря на то что в документации Spring замена метода (Method Replacement) классифицируется как форма внедрения, этот прием сильно отличается от того, что было показано до сих пор. Ранее внедрение применялось исключительно для поставки бинов с их коллабораторами. Используя Method Replacement, можно заменить реализацию любого метода любого бина произвольным образом, без необходимости в изменении исходного кода этого бина. Например, пусть в приложении используется библиотека третьей стороны, и требуется изменить логику определенного метода. Однако, поскольку изменить исходный код не получится, т.к. библиотека поставляется третьей стороной, можно воспользоваться Method Replacement для замены логики конкретного метода новой реализацией.

В коде это достигается динамическим созданием подкласса для класса бина. В данном случае используется библиотека CGLIB и вызовы метода, подлежащего замене, перенаправляются другому бину, который реализует интерфейс `MethodReplacer`.

В листинге 4.63 показан простой бин, в котором объявлены две перегруженных версии метода `formatMessage()`.

Листинг 4.63. Класс `ReplacementTarget`

```
package com.apress.prospring3.ch4.mi;

public class ReplacementTarget {

    public String formatMessage(String msg) {
        return "<h1>" + msg + "</h1>";
    }

    public String formatMessage(Object msg) {
        return "<h1>" + msg + "</h1>";
    }
}
```

С помощью функциональности Method Replacement можно заменить любой из методов класса `ReplacementTarget`. В этом примере мы покажем, как заменить метод

formatMessage(String), а также сравним производительность замененного и исходного методов.

Чтобы заменить метод, сначала нужно создать реализацию интерфейса MethodReplacer, которая приведена в листинге 4.64.

Листинг 4.64. Реализация интерфейса MethodReplacer

```
package com.apress.prospring3.ch4.mi;

import java.lang.reflect.Method;

import org.springframework.beans.factory.support.MethodReplacer;

public class FormatMessageReplacer implements MethodReplacer {

   public Object reimplement(Object arg0, Method method, Object[] args)
         throws Throwable {

      if (isFormatMessageMethod(method)) {

         String msg = (String) args[0];

         return "<h2>" + msg + "</h2>";
      } else {
         throw new IllegalArgumentException("Unable to reimplement method "
               + method.getName());
      }
   }

   private boolean isFormatMessageMethod(Method method) {

      // Проверить правильность количества параметров.
      if (method.getParameterTypes().length != 1) {
         return false;
      }

      // Проверить имя метода.
      if (!("formatMessage".equals(method.getName()))) {
         return false;
      }

      // Проверить возвращаемый тип.
      if (method.getReturnType() != String.class) {
         return false;
      }

      // Проверить тип параметра.
      if (method.getParameterTypes()[0] != String.class) {
         return false;
      }

      return true;
   }
}
```

Интерфейс MethodReplacer имеет единственный метод reimplement(), который должен быть определен. Этому методу передаются три аргумента: бин, на котором вызывается исходный метод, экземпляр Method, представляющий переопределяемый метод, и массив аргументов, передаваемых методу. Метод reimplement() должен возвращать результат замененной логики и, очевидно, тип возвращаемого значения должен быть совместим с возвращаемым типом заменяемого метода. В листинге 4.64 класс FormatMessageReplacer сначала проверяет, переопределяется ли ме-

тод formatMessage(String); если это так, он вызывает заменяющую логику — в данном случае она окружает сообщение парой <h2> и </h2> — и возвращает сформатированное сообщение вызывающему коду. Проверять правильность сообщения не обязательно, однако полезно в случае использования нескольких реализаций интерфейсов MethodReplacer с похожими аргументами. Применение проверки помогает избежать ситуации, при которой случайно используется другой MethodReplacer с совместимыми аргументами и возвращаемыми типами.

В листинге 4.65 приведен ApplicationContext, который определяет два бина типа ReplacementTarget — один имеет замененный метод formatMessage(String), а другой — нет (replacement.xml).

Листинг 4.65. Конфигурирование замены метода

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd">

   <bean id="methodReplacer"
         class="com.apress.prospring3.ch4.mi.FormatMessageReplacer"/>

   <bean id="replacementTarget"
         class="com.apress.prospring3.ch4.mi.ReplacementTarget">
      <replaced-method name="formatMessage" replacer="methodReplacer">
         <arg-type>String</arg-type>
      </replaced-method>
   </bean>

   <bean id="standardTarget"
         class="com.apress.prospring3.ch4.mi.ReplacementTarget"/>

</beans>
```

В листинге 4.65 видно, что реализация MethodReplacer объявлена как бин в ApplicationContext. Затем с использованием дескриптора <replaced-method> производится замена метода formatMessage(String) бина replacementTargetBean. В атрибуте name дескриптора <replaced-method> указывается имя заменяемого метода, а в атрибуте replacer — имя бина MethodReplacer, который должен заменить реализацию метода. При наличии перегруженных методов, как в классе ReplacementTarget, с помощью дескриптора <arg-type> можно указать необходимую сигнатуру метода. Дескриптор <arg-type> поддерживает сопоставление по образцу, поэтому String будет соответствовать java.lang.String и также java.lang.StringBuffer.

В листинге 4.66 приведено простое демонстрационное приложение, которое извлекает бины standardTarget и replacementTarget из ApplicationContext, запускает их методы formatMessage(String) и затем прогоняет несложный тест производительности, чтобы определить, какой из методов быстрее.

Листинг 4.66. Замена метода в действии

```
package com.apress.prospring3.ch4.mi;

import org.springframework.context.support.GenericXmlApplicationContext;
import org.springframework.util.StopWatch;
```

```
public class MethodReplacementExample {

   public static void main(String[] args) {

      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:replacement.xml");
      ctx.refresh();

      ReplacementTarget replacementTarget = (ReplacementTarget) ctx
            .getBean("replacementTarget");
      ReplacementTarget standardTarget = (ReplacementTarget) ctx
            .getBean("standardTarget");

      displayInfo(replacementTarget);
      displayInfo(standardTarget);
   }

   private static void displayInfo(ReplacementTarget target) {
      System.out.println(target.formatMessage("Hello World!"));

      StopWatch stopWatch = new StopWatch();
      stopWatch.start("perfTest");

      for (int x = 0; x < 1000000; x++) {
         String out = target.formatMessage("foo");
      }

      stopWatch.stop();

      System.out.println("1000000 invocations took: "
            + stopWatch.getTotalTimeMillis() + " ms");
   }
}
```

Этот код должен выглядеть очень знакомым, поэтому мы не будем вдаваться в какие-либо детали. На нашей машине запуск этого примера дал следующий вывод:

```
<h2>Hello World!</h2>
1000000 invocations took: 363 ms
<h1>Hello World!</h1>
1000000 invocations took: 107 ms
```

Как и ожидалось, вывод из бина `replacementTarget` показывает, что выполнялась переопределенная реализация, предоставленная `MethodReplacer`. Однако интересно, что динамически замененный метод более чем в три раза медленнее, чем метод, определенный статически. Удаление кода проверки правильности метода в `MethodReplacer` показало незначительную разницу на множестве запусков, поэтому можно сделать заключение о том, что бо́льшая часть накладных расходов кроется в подклассе CGLIB.

Когда использовать замену метода

Замена метода может оказаться весьма полезной в различных обстоятельствах, особенно когда нужно переопределить только отдельный метод в единственном бине, а не во всех бинах одного типа. Тем не менее, для переопределения методов мы по-прежнему предпочитаем пользоваться стандартными механизмами Java, а не полагаться на динамическое расширение байт-кода.

Если вы собираетесь применять замену метода как часть своего приложения, мы рекомендуем использовать один интерфейс `MethodReplacer` на метод или группу перегруженных методов. Не поддавайтесь искушению применить единственную реализацию интерфейса `MethodReplacer` для множества несвязанных друг с другом методов; это приведет к выполнению ненужных сравнений `String` с целью выяснения, какой

метод должен получить новую реализацию. Мы сочли, что проведение простых проверок того, что MethodReplacer работает с правильным методом, является полезным и не добавляет слишком много накладных расходов. Если вы заботитесь о производительности, можете предусмотреть в MethodReplacer булевское свойство, которое позволит включать и отключать такие проверки с использованием Dependency Injection.

Именование бинов

Платформа Spring поддерживает довольно сложную структуру именования бинов, которая предоставляет в ваше распоряжение высокую гибкость в обработке множества разных ситуаций. Каждый бин должен иметь, по крайней мере, одно имя, которое является уникальным в контексте ApplicationContext, содержащем бин. При определении имени, применяемого для бина, Spring следует простому процессу разрешения. Если в дескрипторе <bean> предусмотрен атрибут id, его значение используется в качестве имени. В случае, когда атрибут id не указан, Spring ищет атрибут name, и если он определен, то применяется первое имя, заданное в атрибуте name. (Мы говорим "первое имя", потому что в атрибуте name допускается определять множество имен; этот момент вскоре будет рассматриваться более подробно.) Если не указан ни атрибут id, ни атрибут name, Spring использует в качестве имени имя класса бина, разумеется, при условии отсутствия других бинов с тем же самым именем класса. В случае если множество бинов, не имеющих атрибутов id и name, используют одно и то же имя класса, Spring сгенерирует исключение (типа org.springframework.beans.factory.NoSuchBeanDefinitionException) во время инициализации ApplicationContext. В листинге 4.67 показан пример конфигурации, в которой применяются все три схемы именования.

Листинг 4.67. Именование бинов

```
<bean id="string1" class="java.lang.String"/>
<bean name="string2" class="java.lang.String"/>
<bean class="java.lang.String"/>
```

Формально все эти подходы одинаково допустимы, но какой из них лучше всего избрать для разрабатываемого приложения? Для начала: старайтесь избегать использования автоматических имен, создаваемых на основе поведения классов. Это существенно снижает гибкость в случае определения множества бинов одного типа, и потому гораздо лучше назначить им имена самостоятельно. В таком случае, если стандартное поведение Spring в будущем изменится, ваше приложение продолжит свое нормальное функционирование. Выбирая, какой атрибут лучше использовать — id или name, всегда отдавайте предпочтение id для указания имени по умолчанию для бина. До выхода версии Spring 3.1 атрибут id являлся тем же самым, что и XML-сущность (т.е. xsd:ID), которая накладывала ограничение на символы, допустимые для применения. Начиная с версии Spring 3.1, для атрибута id используется xsd:String, поэтому существовавшее ранее ограничение на применяемые символы снято. Однако Spring продолжает контролировать уникальность id в рамках всего контекста ApplicationContext. В общем случае вы должны назначать своему бину имя с использованием атрибута id и затем ассоциировать этот бин с другими именами за счет создания псевдонимов имен, как объясняется в следующем разделе.

Создание псевдонимов для имен бинов

Платформа Spring позволяет назначать бину более одного имени. Для этого в атрибуте name дескриптора <bean> бина необходимо указать список имен, разделенных

пробелами, запятыми или точками с запятой. Это можно сделать вместо использования атрибута id или же в сочетании с ним.

Чтобы определить псевдонимы для имен бинов Spring, кроме атрибута name можно воспользоваться дескриптором <alias>. В листинге 4.68 приведена простая конфигурация <bean>, в которой для единственного бина определено множество имен (app-context-xml.xml).

Листинг 4.68. Конфигурирование множества имен для бина

```
<bean id="name1" name="name2 name3,name4;name5" class="java.lang.String"/>
<alias name="name1" alias="name6"/>
```

Здесь для бина определены шесть имен. Первое имя указано в атрибуте id и еще четыре имени заданы в атрибуте name в виде списка, в котором используются все разрешенные разделители (мы поступили подобным образом только в демонстрационных целях; в реальных приложениях так делать не рекомендуется). Во время разработки реальных приложений рекомендуется придерживаться единого стандарта при разделении имен бинов в объявлениях. Последний псевдоним определен с помощью дескриптора <alias>. В листинге 4.69 приведен пример Java-программы, которая извлекает один и тот же бин из ApplicationContext шесть раз, используя разные имена и проверяя идентичность полученного бина.

Листинг 4.69. Доступ к бинам с применением псевдонимов

```
package com.apress.prospring3.ch4.xml;

import org.springframework.context.support.GenericXmlApplicationContext;

public class BeanNameAliasing {

public static void main(String[] args) {

      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:app-context-xml.xml");
      ctx.refresh();
      String s1 = (String) ctx.getBean("name1");
      String s2 = (String) ctx.getBean("name2");
      String s3 = (String) ctx.getBean("name3");
      String s4 = (String) ctx.getBean("name4");
      String s5 = (String) ctx.getBean("name5");
      String s6 = (String) ctx.getBean("name6");

      System.out.println((s1 == s2));
      System.out.println((s2 == s3));
      System.out.println((s3 == s4));
      System.out.println((s4 == s5));
      System.out.println((s5 == s6));
   }
}
```

Для конфигурации, показанной в листинге 4.68, этот код выводит на консоль значение true пять раз, проверяя, что бины, доступ к которым осуществляется с использованием разных имен, в действительности являются одним и тем же бином.

Для извлечения списка псевдонимов бина необходимо вызвать метод ApplicationContext.getAliases(String), передав ему любое из имен бина или его идентификатор. Этот метод возвращает в виде массива String список псевдонимов, содержащий все псевдонимы кроме указанного при вызове метода.

Создание псевдонимов для имен бинов не является средством, которое будет обязательно использоваться при построении приложения. Если заранее спланировано, что в приложении будет множество бинов, которые внедряются в другой бин, то внедряемые бины могут для доступа к целевому бину применять одно и то же имя. Тем не менее, после передачи приложения на эксплуатацию, по мере его сопровождения могут вноситься модификации, и вот тут вполне может пригодиться создание псевдонимов для имен.

Рассмотрим следующий сценарий: имеется приложение, в котором все 50 разных бинов, сконфигурированных с использованием Spring, требуют реализации интерфейса `Foo`. При этом 25 бинов пользуются реализацией `StandardFoo` с именем бина `standardFoo`, а другие 25 — реализацией `SuperFoo` с именем бина `superFoo`. Через полгода после передачи приложения в эксплуатацию принимается решение о переводе первых 25 бинов на реализацию `SuperFoo`. Есть три варианта, как сделать это.

- Первый вариант — изменение реализации класса бина `standardFoo` на `SuperFoo`. Недостаток этого подхода в том, что появляются два экземпляра класса `SuperFoo`, когда в действительности необходим только один. Вдобавок, когда изменится конфигурация, придется вносить изменения в два бина, а не в один.
- Второй вариант — обновление конфигурации внедрения для 25 изменяемых бинов, в которой имена бинов меняются с `standardFoo` на `superFoo`. Это не самый элегантный подход — можно воспользоваться поиском и заменой, но последующий откат изменений, если решение окажется неприемлемым, означает необходимость извлечения старой версии кода из системы управления версиями.
- Третий, и наиболее идеальный, вариант — удаление (или комментирование) определения бина `standardFoo` и назначение `standardFoo` в качестве псевдонима для `superFoo`. Такое изменение требует минимальных усилий и упрощает восстановление системы в ее прежней конфигурации.

Режим создания экземпляров бинов

По умолчанию все бины в Spring являются одиночными. Это значит, что Spring обслуживает одиночный экземпляр бина, все зависимые объекты используют один и тот же экземпляр и все вызовы `ApplicationContext.getBean()` возвращают тот же самый экземпляр. Мы демонстрировали это в примере, показанном в листинге 4.64, где для проверки идентичности бинов имели возможность применять операцию сравнения ==, а не `equals()`.

Термин *одиночный* (singleton) используется в Java взаимозаменяемо для ссылки на две отличающиеся концепции: объект, который имеет единственный экземпляр в рамках приложения, и шаблон проектирования Singleton. Первую концепцию мы будем называть одиночным объектом, а вторую без изменений — шаблоном проектирования Singleton. Этот шаблон проектирования стал популярным благодаря книге *Design Patterns: Elements of Reusable Object Oriented Software* Эриха Гаммы и др. (Addison-Wesley, 1995 г.). Проблема возникает, когда начинают путать потребность в одиночных экземплярах и необходимость в применении шаблона проектирования Singleton. Типовая реализация упомянутого шаблона в Java приведена в листинге 4.70.

Листинг 4.70. Шаблон проектирования Singleton

```
package com.apress.prospring3.ch4;

public class Singleton {

   private static Singleton instance;
```

```
    static {
        instance = new Singleton();
    }

    public static Singleton getInstance() {
        return instance;
    }
}
```

Этот шаблон достигает своих целей, позволяя оперировать единственным экземпляром класса по всему приложению, но делает это за счет увеличения степени связанности. Код приложения должен всегда явно знать о классе `Singleton`, чтобы получить его экземпляр, и это приводит к полному исчезновению возможности вынесения кода в интерфейсы. В действительности шаблон Singleton представляет собой два шаблона проектирования в одном. Первый, и он же желаемый, шаблон включает обслуживание единственного экземпляра объекта. Второй, менее желательный, шаблон касается поиска объекта, и это полностью устраняет возможность использования интерфейсов. Применение шаблона Singleton также существенно затрудняет замену реализаций произвольным образом, поскольку большинство объектов, которым требуется экземпляр `Singleton`, получают доступ к объекту `Singleton` напрямую. Это приведет к возникновению разнообразных сложностей при попытке модульного тестирования приложения, т.к. заменить объект Singleton пробным объектом в целях тестирования не получится.

К счастью, в Spring можно воспользоваться моделью создания одиночного экземпляра, не прибегая непосредственно к шаблону проектирования Singleton. Все бины в Spring по умолчанию создаются как экземпляры `Singleton`, и для обработки всех запросов к конкретному бину Spring применяется один и тот же экземпляр. Естественно, платформа Spring не ограничивается использованием только одиночного экземпляра; можно по-прежнему создавать новый экземпляр бина для удовлетворения каждой зависимости и каждого обращения к методу `getBean()`. Все это делается без какого-либо влияния на код приложения, и по этой причине мы предпочитаем говорить, что платформа Spring является *независимой от режима создания экземпляров*. На самом деле, это очень мощная концепция. Если в начале разработки вы считаете, что некоторый объект является одиночным, но впоследствии выясняете, что это не годится для многопоточного доступа, то можете спокойно изменить объект на неодиночный, не оказывая влияния на код приложения.

На заметку! Хотя изменение режима создания экземпляров не влияет на код приложения, оно может привести к возникновению ряда проблем, ели вы полагаетесь на интерфейсы жизненного цикла Spring. Более детально об этом пойдет речь в главе 5.

Изменить режим создания экземпляров с одиночного на неодиночный довольно просто, как показано в листинге 4.71 (`app-context-xml.xml`).

Листинг 4.71. Конфигурация неодиночного бина

```
<bean id="nonSingleton" class="java.lang.String" scope="prototype">
    <constructor-arg>
        <value>Clarence Ho</value>
    </constructor-arg>
</bean>
```

Как видите, единственным отличием между этим объявлением бина и объявлениями, которые приводились ранее, является добавление атрибута `scope` и установка его значения в `prototype` (прототип). По умолчанию Spring устанавливает этот атрибут контекста в `singleton` (одиночный). Область действия на уровне прототипа заставляет Spring создавать новый экземпляр бина каждый раз, когда он запрашивается приложением. В листинге 4.72 можно видеть, какой эффект эта установка оказывает на приложение.

Листинг 4.72. Неодиночные бины в действии

```
package com.apress.prospring3.ch4;

import org.springframework.context.support.GenericXmlApplicationContext;

public class NonSingleton {

   public static void main(String[] args) {

        GenericXmlApplicationContext ctx =
             new GenericXmlApplicationContext();
        ctx.load("classpath:app-context-xml.xml");
        ctx.refresh();

        String s1 = (String) ctx.getBean("nonSingleton");
        String s2 = (String) ctx.getBean("nonSingleton");

        System.out.println("Identity Equal?: " + (s1 ==s2));
        System.out.println("Value Equal:? " + s1.equals(s2));
        System.out.println(s1);
        System.out.println(s2);
   }
}
```

Запуск этого примера дает следующий вывод:

```
Identity Equal?:  false
Value Equal:? true
Clarence Ho
Clarence Ho
```

Вывод показывает, что хотя значения двух объектов `String` одинаковы, сущности различаются, несмотря на то, что оба экземпляра были получены с использованием одного и того же имени бина.

Выбор режима создания экземпляров

В большинстве сценариев довольно просто выяснить, какой режим создания экземпляров подойдет для них. Обычно мы определяем одиночный режим в качестве режима по умолчанию для наших бинов. В общем случае одиночные экземпляры должны использоваться в следующих сценариях.

- *Разделяемые объекты без состояния.* Ситуация, когда имеется объект, не поддерживающий состояния, и множество зависимых от него объектов. Поскольку состояние не поддерживается и, следовательно, синхронизация не нужна, не понадобится и создавать новый экземпляр бина каждый раз, когда он требуется зависимому объекту для какой-то обработки.
- *Разделяемый объект с состоянием только для чтения.* Этот сценарий похож на предыдущий, но здесь поддерживается состояние, предназначенное только для

чтения. В данном случае синхронизация по-прежнему не нужна, поэтому создание экземпляра для каждого запроса бина просто добавляет дополнительные накладные расходы.

- *Разделяемый объект с совместно используемым состоянием.* Если нужен бин, состояние которого должно использоваться совместно, то одиночный экземпляр будет идеальным выбором. В этом случае потребуется обеспечить как можно более мелкомодульную синхронизацию при записи состояния.
- *Объекты с высокой пропускной способностью и записываемым состоянием.* Если есть бин, который интенсивно используется в приложении, вы можете обнаружить, что сохранение его одиночным и синхронизация всего доступа к нему для записи позволяет обеспечить более высокую производительность, чем постоянное создание сотен его экземпляров. Используя такой подход, старайтесь обеспечить как можно более мелкомодульную синхронизацию без ущерба для согласованности. Этот подход особенно полезен в ситуациях, когда приложение создает большое количество экземпляров в течение длительного периода времени, когда разделяемый объект имеет лишь небольшую часть записываемого состояния или когда создание нового экземпляра сопряжено с большими расходами ресурсов.

В перечисленных ниже сценариях должны применяться неодиночные экземпляры.

- *Объекты с записываемым состоянием.* Если имеется бин с большим объемом записываемого состояния, может выясниться, что затраты на синхронизацию превышают затраты на создание нового экземпляра для обработки каждого запроса от зависимого объекта.
- *Объекты с закрытым состоянием.* В ряде случаев зависимым объектам нужен бин, который имеет закрытое состояние, так что зависимые объекты могут проводить свою обработку отдельно друг от друга. В такой ситуации одиночный экземпляр точно не подходит, поэтому необходим неодиночный экземпляр.

Главное преимущество, которое вы получаете от управления созданием экземпляров Spring, связано с уменьшением потребления памяти приложениями благодаря одиночным экземплярам, причем для этого понадобятся весьма небольшие усилия с вашей стороны. Кроме того, если окажется, что режим одиночных экземпляров не соответствует потребностям приложения, довольно легко перенастроить конфигурацию на использование режима неодиночных экземпляров.

Области действия бинов

В дополнение к областям действия на уровне одиночного экземпляра и прототипа, при определении бина Spring, предназначенного для более специфических целей, доступны и другие области действия. Можно также реализовать собственную специальную область действия и зарегистрировать ее в `ApplicationContext`. Ниже приведен список областей действия, которые поддерживаются в версии Spring 3.1.

- *Одиночный экземпляр.* Область действия, устанавливаемая по умолчанию.
- *Прототип.* Новый экземпляр будет создаваться Spring при его запросе приложением.
- *Запрос.* Для использования в веб-приложениях. В случае применения Spring MVC при построении веб-приложения бины с областью действия на уровне запроса будут создаваться для каждого HTTP-запроса и уничтожаться после завершения его обработки.

- *Сеанс*. Для использования в веб-приложениях. В случае применения Spring MVC при построении веб-приложения бины с областью действия на уровне сеанса будут создаваться для каждого HTTP-сеанса и уничтожаться после его завершения.
- *Глобальный сеанс*. Для использования в веб-приложениях, основанных на портлетах. Бины с областью действия на уровне глобального сеанса могут разделяться всеми портлетами в рамках портального приложения Spring MVC.
- *Поток*. Новый экземпляр бина будет создаваться Spring в случае его запроса новым потоком, в то время как при его запросе тем же самым потоком будет возвращаться один тот же экземпляр бина. Обратите внимание, что по умолчанию эта область действия не зарегистрирована.
- *Специальная*. Специальная область действия бина, которую можно создать путем реализации интерфейса `org.springframework.beans.factory.config.Scope` и затем зарегистрировать в конфигурации Spring (в случае XML используйте класс `org.springframework.beans.factory.config.CustomScopeConfigurer`).

Разрешение зависимостей

Во время нормального функционирования платформа Spring способна разрешать зависимости, просто просматривая конфигурационный файл или аннотации в классах. Таким образом, Spring может гарантировать, что все бины сконфигурированы в правильном порядке, так что каждый бин имеет корректно настроенные зависимости. Если платформа Spring не выполнит этого, а просто создаст бины и сконфигурирует их в произвольном порядке, может оказаться, что какой-нибудь бин будет создан и настроен до его зависимостей. Очевидно, что при этом в приложении могут возникать различные проблемы.

К сожалению, Spring ничего не известно о зависимостях, которые существуют между бинами в коде. Например, пусть один бин под названием `beanA` получает экземпляр другого бина под названием `beanB` в своем конструкторе через вызов `getBean()`. Например, в конструкторе бина `beanA` получается экземпляр бина `beanB` путем вызова `ctx.getBean("beanB")` без уведомления Spring о необходимости внедрения зависимости. В этом случае платформа Spring не знает о том, что `beanA` зависит от `beanB`, в результате чего она может создать экземпляр `beanA` перед созданием экземпляра `beanB`. Предоставить Spring дополнительную информацию о зависимостях между бинами можно с помощью атрибута `depends-on` дескриптора `<bean>`. В листинге 4.73 показано, как можно было бы сконфигурировать `beanA` и `beanB`.

Листинг 4.73. Ручное определение зависимостей

```
<bean id="beanA" class="com.apress.prospring.ch4.BeanA" depends-on="beanB"/>
<bean id="beanB" class="com.apress.prospring.ch4.BeanB"/>
```

В этой конфигурации мы утверждаем, что бин по имени `beanA` зависит от бина по имени `beanB`. Платформа Spring примет это внимание при создании экземпляров бинов и обеспечит создание экземпляра `beanB` раньше `beanA`.

При проектировании приложений такого подхода лучше избегать; в противном случае зависимости следует определять с помощью контрактов Setter Injection и Constructor Injection. Однако если Spring интегрируется с унаследованным кодом, может оказаться, что определенные в нем зависимости требуют предоставления дополнительной информации для Spring Framework.

Автосвязывание бина

Во всех показанных до сих пор примерах с помощью конфигурационного файла явно определялось, каким образом отдельные бины связаны друг с другом. Если вы не желаете это делать самостоятельно, можете предоставить Spring попытку связать компоненты автоматически. По умолчанию автосвязывание отключено. Чтобы включить его, в атрибуте `autowire` для бина понадобится указать используемый режим автосвязывания.

Различные режимы автосвязывания

Платформа Spring поддерживает следующие режимы автосвязывания: `byName`, `byType`, `constructor`, `default` и `no` (автосвязывание отключено; устанавливается по умолчанию). При использовании автосвязывания `byName` платформа Spring пытается связать каждое свойство с бином, имеющим такое же имя. Таким образом, если целевой бин имеет свойство по имени `foo` и в `ApplicationContext` определен бин `foo`, то этот бин `foo` назначается свойству `foo` целевого бина.

Когда применяется автосвязывание `byType`, платформа Spring пытается связать каждое свойство целевого бина с бином того же самого типа, что и свойство, в `ApplicationContext`. Это значит, что если в целевом бине имеется свойство типа `String` и в `ApplicationContext` определен некоторый бин типа `String`, то Spring свяжет бин `String` со свойством `String` целевого бина. Если в `ApplicationContext` определено более одного бина типа `String`, то Spring не сможет решить, какой из них использовать для автосвязывания, и сгенерирует исключение (типа `org.springframework.beans.factory.NoSuchBeanDefinitionException`).

Режим автосвязывания `constructor` функционирует подобно режиму `byType` за исключением того, что для выполнения внедрения он использует конструкторы, а не методы установки. Spring пытается найти соответствие с как можно большим числом аргументов в конструкторе. Таким образом, если бин имеет два конструктора, первый из которых принимает аргумент `String`, а второй — аргументы `String` и `Integer`, и при этом в `ApplicationContext` определены бины `String` и `Integer`, Spring будет использовать конструктор с двумя аргументами.

В режиме `default` платформа Spring будет автоматически делать выбор между режимами `constructor` и `byType`. Если бин имеет конструктор по умолчанию (без аргументов), то Spring использует `byType`; в противном случае Spring применяет `constructor`.

В листинге 4.74 показана простая конфигурация, которая автоматически связывает три бина одного и того же типа с использованием всех описанных режимов (`autowiring.xml`).

Листинг 4.74. Конфигурирование автосвязывания

```
<bean id="foo" class="com.apress.prospring3.ch4.autowiring.Foo"/>
<bean id="bar1" class="com.apress.prospring3.ch4.autowiring.Bar"/>

<bean id="targetByName" autowire="byName"
     class="com.apress.prospring3.ch4.autowiring.Target"
     lazy-init="true"/>
<bean id="targetByType" autowire="byType"
     class="com.apress.prospring3.ch4.autowiring.Target"
     lazy-init="true"/>
<bean id="targetConstructor" autowire="constructor"
     class="com.apress.prospring3.ch4.autowiring.Target"
     lazy-init="true"/>
```

Эта конфигурация должна выглядеть очень знакомо. `Foo` и `Bar` — пустые классы. Каждый из бинов `Target` имеет отличающееся значение в атрибуте `autowire`. Кроме того, атрибут `lazy-init` установлен в `true`, чтобы проинформировать Spring о необходимости создания экземпляра бина только при первом его запросе, а не во время начальной загрузки, поэтому мы можем выводить результаты в правильном месте внутри тестовой программы. В листинге 4.75 показано простое Java-приложение, которое извлекает каждый бин `Target` из `ApplicationContext`.

Листинг 4.75. Автосвязывание коллабораторов

```
package com.apress.prospring3.ch4.autowiring;

import org.springframework.context.support.GenericXmlApplicationContext;

public class Target {

   private Foo foo;
   private Foo foo2;

   private Bar bar;

   public Target() {
   }

   public Target(Foo foo) {
      System.out.println("Target(Foo) called"); // вызван метод Target(Foo)
   }

   public Target(Foo foo, Bar bar) {
      System.out.println("Target(Foo, Bar) called");
                        // вызван метод Target(Foo, Bar)
   }

   public void setFoo(Foo foo) {
      this.foo = foo;
      System.out.println("Property foo set"); // установлено свойство foo
   }

   public void setFoo2(Foo foo) {
      this.foo2 = foo;
      System.out.println("Property foo2 set");// установлено свойство foo2
   }

   public void setBar(Bar bar) {
      this.bar = bar;
      System.out.println("Property bar set"); // установлено свойство bar
   }

   public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:autowiring.xml");
      ctx.refresh();

      Target t = null;
      System.out.println("Using byName:\n");      // использование byName
      t = (Target) ctx.getBean("targetByName");
      System.out.println("\nUsing byType:\n");    // использование byType
      t = (Target) ctx.getBean("targetByType");
      System.out.println("\nUsing constructor:\n");
                        // использование конструктора
      t = (Target) ctx.getBean("targetConstructor");
   }
}
```

В этом коде класс Target имеет три конструктора: конструктор без аргументов, конструктор, принимающий экземпляр Foo, и конструктор, который принимает экземпляры Foo и Bar. В дополнение к перечисленным конструкторам, бин Target имеет три свойства: два из них типа Foo и одно типа Bar. Все свойства и конструкторы при вызове выводят сообщения на консоль. Метод main() просто извлекает все бины Target, объявленные в ApplicationContext, инициируя процесс автосвязывания. Ниже показан вывод, полученный в результате запуска этого примера:

```
Using byName:

Property foo set

Using byType:

Property bar set
Property foo set
Property foo2 set

Using constructor:

Target(Foo, Bar) called
```

В выводе видно, что когда Spring использует byName, единственным устанавливаемым свойством является foo, потому что одно это свойство имеет соответствующую запись бина в конфигурационном файле. В случае применения byType платформа Spring устанавливает значения трех свойств. Свойства foo и foo2 установлены бином foo, а свойство bar — бином bar1. При использовании constructor платформа Spring выбирает конструктор с двумя аргументами, т.к. Spring может предоставить бины для обоих аргументов и не нуждается в обращении к другому конструктору.

Когда используется автосвязывание

В большинстве случаев ответ на вопрос, следует ли применять автосвязывание, будет определенно отрицательным. Автосвязывание может сэкономить время в небольших приложениях, но во многих ситуациях оно приводит к неудачным решениям и утере гибкости в крупных приложениях. Использование byName выглядит неплохой идеей, но может потребовать назначения свойствам в классах искусственных имен, чтобы воспользоваться функциональностью автосвязывания. Вся идея Spring заключается в том, что вы создаете свои классы так, как вам нравится, и позволяете Spring работать с ними, а не наоборот. Искушение применять byType может возникать до тех пор, пока вы не поймете, что в ApplicationContext может существовать только один бин для каждого типа — ограничение, которое создаст проблемы, когда необходимо обслуживать бины одного и того же типа, но с различными конфигурациями. Аналогичное замечание касается также использования автосвязывания constructor.

В ряде случаев автосвязывание может сберечь время, однако явное определение связывания не требует настолько много дополнительных усилий, чтобы отказываться от преимуществ явной семантики и полной свободы в именовании свойств и произвольном количестве экземпляров бина одного и того же типа, которыми можно управлять. При разработке любого нетривиального приложение держитесь от автосвязывания подальше любой ценой.

Наследование бинов

В некоторых ситуациях может понадобиться множество определений бинов одного и того же типа или реализация разделяемого интерфейса. Это может стать проблематичным, если бины должны разделять между собой одни параметры конфигурации

и не разделять другие. Процесс сохранения разделяемых параметров конфигурации в синхронизированном состоянии подвержен ошибкам, а в крупных проектах еще и может отнимать довольно много времени. Чтобы обойти это, Spring позволяет задавать определения `<bean>`, которые наследуют свои свойства от других бинов в том же самом контексте `ApplicationContext`. При необходимости можно переопределить значения любых свойств дочернего бина, что позволяет иметь над ними полный контроль, но родительский бин может предоставить каждому дочернему бину базовую конфигурацию. В листинге 4.76 показана простая конфигурация с двумя бинами, один из которых является дочерним по отношению к другому (`app-context-xml.xml`).

Листинг 4.76. Конфигурирование наследования бинов

```
<bean id="inheritParent" class="com.apress.prospring3.ch4.inheritance.SimpleBean">
   <property name="name">
      <value>Clarence Ho</value>
   </property>
   <property name="age">
      <value>22</value>
   </property>
</bean>

<bean id="inheritChild"
      class="com.apress.prospring3.ch4.inheritance.SimpleBean"
      parent="inheritParent">
   <property name="age">
      <value>35</value>
   </property>
</bean>
```

В приведенном коде дескриптор `<bean>` для бина `inheritChild` имеет дополнительный атрибут `parent`, который указывает, что платформа Spring должна считать бин `inheritParent` родительским для определяемого бина. В случае если определение родительского бина не должно быть доступным для поиска из `ApplicationContext`, к дескриптору `<bean>`, объявляющему родительский бин, можно добавить атрибут `abstract="true"`. Поскольку бин `inheritChild` имеет собственное значение в свойстве `age`, Spring передает это значение бину. Однако `inheritChild` не имеет установленного значения в свойстве `name`, поэтому Spring использует значение, заданное для бина `inheritParent`. В листинге 4.77 приведен код класса `SimpleBean`, который применялся в предыдущей конфигурации.

Листинг 4.77. Класс `SimpleBean`

```
package com.apress.prospring3.ch4.inheritance;

import org.springframework.context.support.GenericXmlApplicationContext;

public class SimpleBean {

   public String name;

   public int age;

   public static void main(String[] args) {

      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:app-context-xml.xml");
```

```
        SimpleBean parent = (SimpleBean) ctx.getBean("inheritParent");
        SimpleBean child = (SimpleBean) ctx.getBean("inheritChild");

        System.out.println("Parent:\n" + parent);
        System.out.println("Child:\n" + child);
    }

    public void setName(String name) {
        this.name = name;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public String toString() {
        return    "Name: " + name + "\n"
                + "Age: " + age;
    }
}
```

Здесь метод `main()` класса `SimpleBean` извлекает бины `inheritChild` и `inheritParent` из `ApplicationContext` и записывает содержимое их свойств в `stdout`. Ниже показан вывод, полученный в результате запуска этого примера:

```
Parent:
Name: Clarence Ho
Age: 22
Child:
Name: Clarence Ho
Age: 35
```

Как и ожидалось, бин `inheritChild` унаследовал значение для своего свойства `name` от бина `inheritParent`, но предоставил собственное значение для свойства `age`.

Дочерние бины наследуют аргументы конструкторов и значения свойств от родительских бинов, поэтому при наследовании бинов можно использовать оба стиля внедрения. Такой уровень гибкости превращает наследование бинов в мощный инструмент для построения приложений с большим количеством бинов. При объявлении множества бинов с одинаковыми значениями в разделяемых свойствах избегайте соблазна применять копирование и вставку для размножения этих значений; вместо этого настройте в своей конфигурации подходящую иерархию наследования.

Не следует забывать, что наследование бинов не обязательно должно совпадать с иерархией наследования Java. Вполне допустимо использовать наследование бинов, скажем, для пяти бинов одного и того же типа. Наследование бинов больше похоже на средство создания шаблонов, чем на собственно наследование. Однако при изменении типа дочернего бина имейте в виду, что этот тип должен быть совместим с типом родительского бина.

Резюме

В этой главе мы раскрыли много общих понятий, связанных с ядром Spring и IoC. Были продемонстрированы примеры различных типов IoC и приведены обсуждения достоинств и недостатков использования каждого механизма в разрабатываемых приложениях. Мы посмотрели, какие механизмы IoC предлагает Spring, и объяснили, когда их применять в приложениях, а когда нет. Во время исследования IoC был представлен интерфейс `BeanFactory`, являющийся ключевым компонентом для средств IoC в Spring, а

также интерфейс `ApplicationContext`, который расширяет `BeanFactory` и предоставляет дополнительную функциональность. Во время рассмотрения `ApplicationContext` мы сосредоточили внимание на классе `GenericXmlApplicationContext`, который делает возможным применение внешней конфигурации Spring с помощью XML. Также обсуждался еще один способ объявления требований DI для `ApplicationContext` — использование Java-аннотаций.

В главе также были даны основы набора функциональных средств IoC в Spring, включая Setter Injection, Constructor Injection, Method Injection, автосвязывание и наследование бинов. При обсуждении конфигурации мы продемонстрировали настройку свойств бинов с применением широкого разнообразия значений, в том числе других бинов, конфигураций в стиле XML и аннотаций, а также `GenericXmlApplicationContext`.

В данной главе мы лишь слегка коснулись поверхности Spring и встроенного в Spring контейнера IoC. В следующей главе мы рассмотрим ряд связанных с IoC возможностей, специфичных для Spring, и взглянем более внимательно на другую функциональность, доступную в ядре Spring.

ГЛАВА 5

Детальные сведения о конфигурации Spring

В предыдущей главе была рассмотрена концепция инверсии управления (Inversion of Control — IoC) и ее реализация в Spring Framework. Но, как было сказано в конце той главы, в действительности мы лишь слегка коснулись поверхности ядра Spring.

Платформа Spring предоставляет широкий массив служб, которые дополняют и расширяют базовые возможности IoC. Контейнеры IoC предлагаются многими проектами, но до сих пор не появилось ни одного проекта, который бы обеспечил такой же всеобъемлющий набор средств, какой предоставляет Spring. В этой главе мы собираемся детально рассмотреть ряд дополнительных средств, связанных с IoC, которые предлагает Spring, вместе с другой функциональностью, обеспечиваемой ядром Spring. Кроме того, мы посмотрим, как использовать Java-классы при конфигурировании `ApplicationContext`, и взглянем на новые средства, доступные в версиях Spring 3.0 и 3.1, которые предназначены для более сложных вариантов конфигурирования. В частности, в главе будут рассмотрены следующие темы.

- *Управление жизненным циклом бинов.* Все бины, которые вы видели до сих пор, было довольно простыми и совершенно не привязанными к контейнеру Spring. В данном разделе вы ознакомитесь со стратегиями, которыми можно воспользоваться, чтобы позволить бинам получать уведомления от контейнера Spring в различных точках их жизненного цикла. Это можно сделать за счет реализации специфических интерфейсов, предлагаемых Spring, путем указания методов, которые Spring может вызывать через рефлексию, или с использованием аннотаций жизненного цикла компонентов JavaBean согласно JSR-250.
- *Превращение бинов в "осведомленные о платформе Spring".* В некоторых случаях необходимо, чтобы бин имел возможность взаимодействовать со сконфигурировавшим его контекстом `ApplicationContext`. Для этих целей Spring предлагает два интерфейса, `BeanNameAware` и `ApplicationContextAware`, которые позволяют бину получать, соответственно, назначенное ему имя (т.е. значение атрибута `id` или `name` дескриптора `<bean>`, причем `id` имеет более высокий приоритет) и ссылку на его контекст `ApplicationContext`. В данном разделе рассматриваются реализации этих интерфейсов и приводятся некоторые соображения по поводу их практического использования в приложении.
- *Использование фабрик бинов.* Интерфейс `FactoryBean` предназначен для реализации любым бином, который действует в качестве фабрики для других бинов. Интерфейс `FactoryBean` предоставляет механизм, с помощью которого легко интегрировать собственные фабрики с `BeanFactory` из Spring.

- *Работа с редакторами свойств для компонентов JavaBean.* Интерфейс `PropertyEditor` — это стандартный интерфейс, предоставляемый в пакете `java.beans`. Редакторы свойств (реализующие `PropertyEditor`) используются для преобразования значений свойств в и из представлений `String`. Платформа Spring применяет редакторы свойств повсеместно, в основном для чтения значений, указанных в конфигурации `BeanFactory`, и преобразования их в корректные типы. В этом разделе главы мы обсудим набор редакторов свойств, поставляемых Spring, и покажем, как их использовать в рамках приложения. Также будет кратко объясняться реализация собственных редакторов свойств.
- *Дополнительные сведения о расширении* `ApplicationContext`. Как обсуждалось в предыдущей главе, `ApplicationContext` — это расширение `BeanFactory`, предназначенное для использования в полнофункциональных приложениях. Интерфейс `ApplicationContext` предоставляет полезный набор дополнительной функциональности, включая обеспечение интернационализированными сообщениями, загрузку ресурсов и публикацию событий. В этой главе мы детально рассмотрим средства, дополняющие возможности IoC, которые предлагает `ApplicationContext`. Мы также забежим немного вперед и покажем, как `ApplicationContext` упрощает использование Spring при построении веб-приложений.
- *Конфигурирование с использованием Java-классов.* До версии платформа Spring поддерживала только базовую конфигурацию XML с аннотациями для конфигурации бинов и зависимостей. Начиная с версии 3.0, платформа Spring предлагает разработчикам еще один вариант конфигурирования `ApplicationContext` с использованием Java-классов. Мы кратко рассмотрим этот новый вариант конфигурации Spring-приложений.
- *Новые возможности конфигурации в Spring 3.1.* Версия Spring 3.1 предоставляет ряд полезных новых средств, связанных с конфигурированием, такие как управление профилями, абстракция `Environment` и `PropertySource` и т.д. В этом разделе мы рассмотрим эти средства и покажем, как использовать их для решения специфических потребностей в конфигурировании.

Влияние Spring на переносимость приложений

Большинство функциональных средств, обсуждаемых в этой главе, являются специфичными для Spring и во многих случаях не доступными в других контейнерах IoC. Хотя многие контейнеры IoC предлагают функциональность управления жизненным циклом, возможно, они делают это через набор интерфейсов, отличный от Spring. Если переносимость приложения между разными контейнерами IoC действительно важна, вы можете решить не использовать некоторые из средств, которые привязывают приложение к платформе Spring.

Однако помните, что установив ограничение, связанное с переносимостью приложения между контейнерами IoC, вы теряете все богатство функциональности, предлагаемой Spring. Поскольку, скорее всего, будет принято стратегическое решение использовать платформу Spring, имеет смысл извлечь максимум из ее возможностей.

Остерегайтесь создавать требования переносимости на пустом месте. Во многих случаях конечных пользователей не волнует тот факт, что приложение может выполняться в трех разных контейнерах IoC; им просто нужно его запустить. Согласно нашему опыту, частой ошибкой является попытка построить приложение на основе “наименьшего общего знаменателя” для функциональных средств, доступных в рамках выбранной технологии. Это может поставить создаваемое приложение в невыгодные условия с самого начала. Однако если приложение должно быть переносимым между контейнерами IoC,

не следует трактовать это требование как недостаток — это нормальное требование, которому, помимо прочих, приложение должно удовлетворять. В книге *Expert One-on-One: J2EE Development without EJB* (Wrox, 2004 г.) такие типы требований названы фантомными требованиями; там же можно найти более развернутое обсуждение их влияния на разрабатываемые проекты.

Несмотря на то что использование этих функциональных средств может привязать приложение к платформе Spring Framework, на самом деле переносимость приложения увеличивается в гораздо большей степени. Подумайте о том, что вы пользуетесь свободно доступной платформой с открытым кодом, не принадлежащей какому-то конкретному поставщику. Приложение, построенное с применением контейнера IoC из Spring, запускается везде, где функционирует машина Java. Для корпоративных Java-приложений Spring открывает новые возможности в плане переносимости. Платформа Spring предоставляет многие из средств, предлагаемых JEE, и также поддерживает классы для абстрагирования и упрощения множества других аспектов JEE. В большинстве ситуаций с помощью Spring можно построить веб-приложение, которое выполняется в простом контейнере типа сервлета, но с таким же уровнем сложности, как и приложение, ориентированное на полномасштабный сервер приложений JEE. Привязываясь к Spring, вы увеличиваете степень переносимости своего приложения, заменяя многие функциональные средства, которые являются либо специфическими для поставщика, либо зависят от специфической для поставщика конфигурации, эквивалентными средствами Spring.

Управление жизненным циклом бинов

Важным аспектом любого контейнера IoC, включая Spring, является возможность конструирования бинов так, что они будут получать уведомления в определенных точках их жизненного цикла. Это позволяет бинам выполнять соответствующую обработку в этих точках. В общем случае непосредственное отношение к бинам имеют два события жизненного цикла: событие после инициализации и событие перед уничтожением.

В контексте Spring событие после инициализации генерируется, когда Spring завершает установку всех значений свойств бина и все проверки зависимостей, которые были сконфигурированы для выполнения. Событие перед уничтожением инициируется непосредственно перед тем, как Spring приступает к уничтожению экземпляра бина. Однако для бинов с областью действия на уровне прототипа событие перед уничтожением не будет генерироваться Spring. Проектное решение, лежащее в основе платформы Spring, предполагает выполнение методов обратного вызова, инициализирующих жизненный цикл, для объектов независимо от области действия бинов, но для бинов с областью действия на уровне прототипа методы обратного вызова, разрушающие жизненный цикл, инициироваться не будут. Платформа Spring предоставляет три механизма, которые бин может использовать для привязки к каждому из указанных событий и выполнения дополнительной обработки: основанный на интерфейсах, основанный на методах и основанный на аннотациях.

Используя механизм, основанный на интерфейсах, бин реализует интерфейс, специфичный для типа уведомлений, которые необходимо получать, и Spring уведомляет бин через метод обратного вызова, определенный в этом интерфейсе. В случае механизма, основанного на методах, Spring позволяет указать в конфигурации `ApplicationContext` имя метода, который должен быть вызван при инициализации бина, и имя метода, который должен быть вызван при уничтожении бина. Механизм, основанный на аннотациях, предполагает использование аннотаций JSR-250 для указания методов, которые платформа Spring должна вызывать после конструирования и перед уничтожением.

Для обоих событий эти механизмы достигают в точности тех же самых целей. Механизм, основанный на интерфейсах, широко используется в самой платформе Spring, поэтому при работе с компонентами Spring нет необходимости помнить о том, что нужно указывать инициализацию и уничтожение. Тем не менее, в собственных бинах может быть лучше пользоваться механизмом, основанным на методах или аннотациях, поскольку ваши бины не должны реализовывать интерфейсы, специфичные для Spring. Хотя мы утверждаем, что переносимость часто не является настолько важной, как это заявлено во многих книгах, это не означает, что вы должны жертвовать переносимостью, когда существует вполне хорошая альтернатива. Тем не менее, если вы привязали свое приложение к Spring другими путями, то использование метода интерфейса позволит определить обратный вызов только один раз, после чего забыть о нем. В ситуации, когда определяется множество однотипных бинов, которым необходимы уведомления жизненного цикла, применение механизма, основанного на интерфейсах, может помочь избежать необходимости в указании методов обратного вызова жизненного цикла для каждого бина в XML-файле конфигурации. Использование аннотаций JSR-250 — также приемлемый вариант, поскольку это стандарт, определенный JCP, вдобавок вы не привязываетесь к аннотациям, специфичным для Spring. Просто удостоверьтесь, что контейнер IoC, в котором выполняется приложение, поддерживает стандарт JSR-250.

В целом, выбор механизма, который будет применяться для получения уведомлений жизненного цикла, зависит от требований вашего приложения. Если вас интересует переносимость либо вы просто определяете один или два бина определенного типа, которым нужны обратные вызовы, используйте механизм, основанный на методах. Если вы применяете конфигурацию в стиле аннотаций и уверены, что целевой контейнер IoC поддерживает JSR-250, используйте механизм, основанный на аннотациях. Если вы не слишком обеспокоены переносимостью или определяете множество однотипных бинов, которым нужны уведомления жизненного цикла, то применение механизма, основанного на интерфейсах, будет наилучшим способом обеспечить получение бинами ожидаемых уведомлений. Если же вы планируете использовать какой-то бин во множестве разных проектов Spring, то наверняка хотите, чтобы функциональность этого бина была как можно более самодостаточной, поэтому в таком случае определенно понадобится применять механизм, основанный на интерфейсах.

На рис. 5.1 приведен высокоуровневый обзор того, как Spring управляет жизненным циклом бинов внутри их контейнера.

Привязка к созданию бина

Зная, когда он инициализирован, бин может проверить, все ли требуемые зависимости удовлетворены. Хотя платформа Spring может проверить зависимости самостоятельно, применяемый ею подход можно охарактеризовать как “все или ничего”, и она не предлагает каких-либо возможностей для применения дополнительной логики в процедуре разрешения зависимостей. Рассмотрим бин, который имеет четыре зависимости, объявленные как методы установки, две из которых являются обязательными, а одна обеспечивает подходящее значение по умолчанию на случай, когда зависимость не предоставлена. Используя обратный вызов инициализации, бин может проверить требующиеся ему зависимости и при необходимости сгенерировать исключение либо предоставить значение по умолчанию.

Бин не может выполнить такие проверки в своем конструкторе, т.к. в этой точке Spring не имеет возможности предоставить значения для зависимостей. Обратный вызов инициализации инициируется после того, как Spring завершает предоставление зависимостей и может выполнить любые запрашиваемые проверки зависимостей.

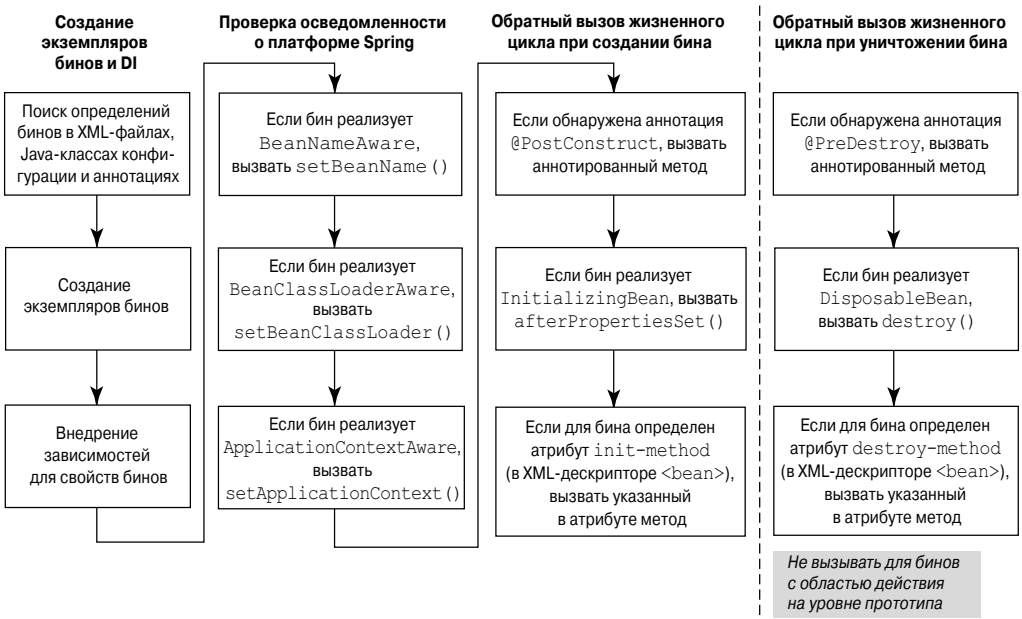


Рис. 5.1. Жизненный цикл бинов Spring

Обратный вызов инициализации не ограничивается одной лишь проверкой зависимостей; в нем можно делать все, что угодно, но наиболее полезен этот обратный вызов как раз для той цели, которую мы описали выше. Во многих случаях обратный вызов инициализации также является местом для запуска любых действий, которые бин должен предпринимать автоматически в ответ на свою конфигурацию. Например, если вы строите бин для выполнения запланированных задач, то обратный вызов инициализации будет идеальным местом для запуска планировщика — ведь конфигурационные данные устанавливаются для бина.

На заметку! Вам не придется разрабатывать бин для запуска запланированных задач, поскольку Spring может это делать автоматически с помощью встроенного средства планирования или через интеграцию с планировщиком Quartz. Более детально эти вопросы рассматриваются в главе 15.

Запуск метода, когда бин создается

Как упоминалось ранее, один из способов получения обратного вызова инициализации предусматривает назначение некоторого метода бина в качестве метода инициализации и сообщение об этом платформе Spring. Этот механизм обратного вызова полезен, когда имеется всего один бин определенного типа или когда необходимо отвязать приложение от Spring. Еще одна причина применения этого механизма заключается в том, чтобы позволить Spring-приложению работать с бинами, которые были построены ранее или предоставлены независимыми поставщиками.

Определение метода обратного вызова сводится просто к указанию имени в атрибуте `init-method` дескриптора `<bean>` для бина. В листинге 5.1 показан код бина с двумя зависимостями.

Листинг 5.1. Класс SimpleBean

```
package com.apress.prospring3.ch5.lifecycle;
import org.springframework.beans.factory.BeanCreationException;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.GenericXmlApplicationContext;
public class SimpleBean {

   private static final String DEFAULT_NAME = "Luke Skywalker";

   private String name = null;

   private int age = Integer.MIN_VALUE;

   public void setName(String name) {
      this.name = name;
   }
   public void setAge(int age) {
      this.age = age;
   }
   public void init() {
      System.out.println("Initializing bean"); // инициализация бина
     if (name == null) {
         System.out.println("Using default name");
                          // использование имени по умолчанию
         name = DEFAULT_NAME;
      }
      if (age == Integer.MIN_VALUE) {
         throw new IllegalArgumentException(
               "You must set the age property of any beans of type "
             // Должно быть установлено свойство age любого бина этого типа.
               + SimpleBean.class);
      }
   }
   public String toString() {
      return "Name: " + name + "\nAge: " + age;
   }
   public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:lifecycle/initMethod.xml");
      ctx.refresh();   // Обновить ApplicationContext после
                       // загрузки XML-файла конфигурации

      SimpleBean simpleBean1 = getBean("simpleBean1", ctx);
      SimpleBean simpleBean2 = getBean("simpleBean2", ctx);
      SimpleBean simpleBean3 = getBean("simpleBean3", ctx);
   }
   private static SimpleBean getBean(String beanName, ApplicationContext ctx) {
      try {
         SimpleBean bean =(SimpleBean) ctx.getBean(beanName);
         System.out.println(bean);
         return bean;
      } catch (BeanCreationException ex) {
         System.out.println("An error occured in bean configuration: "
               + ex.getMessage()); // В конфигурации бина обнаружена ошибка.
         return null;
      }
   }
}
```

Обратите внимание, что здесь определен метод `init()`, который будет служить обратным вызовом инициализации. Метод `init()` проверяет, установлено ли свойство `name`, и если это не так, использует значение по умолчанию, хранящееся в константе `DEFAULT_NAME`. Кроме того, этот метод также проверяет, установлено ли свойство `age`, и генерирует исключение `IllegalArgumentException`, если оно не установлено.

Метод `main()` класса `SimpleBean` пытается получить три бина из `GenericXmlApplicationContext`, имеющие тип `SimpleBean`, с использованием собственного метода `getBean()`. Если бин получен успешно, в методе `getBean()` его детали выводятся на консоль. Если в методе `init()` возникает исключение, как и будет в случае, когда свойство `age` не установлено, то Spring поместит это исключение в оболочку `BeanCreationException`. Метод `getBean()` перехватывает такие исключения и записывает в консольный вывод сообщение, информирующее о возникшей ошибке, а затем возвращает значение `null`.

В листинге 5.2 приведена конфигурация `ApplicationContext`, в которой определены бины, используемые в листинге 5.1 (`lifecycle/initMethod.xml`).

Листинг 5.2. Конфигурирование бинов `SimpleBean`

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd"
      default-lazy-init="true">
   <bean id="simpleBean1"
      class="com.apress.prospring3.ch5.lifecycle.SimpleBean"
      init-method="init">
      <property name="name">
         <value>Clarence Ho</value>
      </property>
      <property name="age">
         <value>100</value>
      </property>
   </bean>
   <bean id="simpleBean2"
      class="com.apress.prospring3.ch5.lifecycle.SimpleBean"
       init-method="init">
      <property name="age">
         <value>100</value>
      </property>
   </bean>
   <bean id="simpleBean3"
      class="com.apress.prospring3.ch5.lifecycle.SimpleBean"
      init-method="init">
      <property name="name">
         <value>Clarence Ho</value>
      </property>
   </bean>
</beans>
```

Как видите, дескрипторы `<bean>` для всех трех бинов содержат атрибут `init-method`, который сообщает платформе Spring, что она должна вызвать метод `init()`, как только завершит конфигурирование бина. Бин `simpleBean1` имеет значения для обоих свойств `name` и `age`, поэтому он проходит через метод `init()` совершенно без изменений.

В бине simpleBean2 не задано значение для свойства name, а это означает, что в методе init() свойство name получит значение по умолчанию. Наконец, в бине simpleBean3 не указано значение для свойства age. Логика, реализованная в методе init(), трактует это как ошибку, поэтому генерируется исключение IllegalArgumentException. Также обратите внимание, что в дескриптор <beans> мы добавили атрибут default-lazy-init="true", который инструктирует Spring создавать экземпляры бинов, определенных в конфигурационном файле, только при их запросе приложением. Если не указать этот атрибут, Spring попытается инициализировать все бины во время начальной загрузки ApplicationContext и потерпит неудачу при инициализации simpleBean3. Запуск приведенного выше примера дает следующий вывод:

```
Initializing bean
Name: Clarence Ho
Age: 100
Initializing bean
Using default name
Name: Luke Skywalker
Age: 100
Initializing bean
An error occured in bean configuration: Error creating bean with name
'simpleBean3' defined in class path resource [lifecycle/initMethod.xml]:
Invocation of init method failed; nested exception is java.lang.
IllegalArgumentException: You must set the age property of any beans of
type class com.apress.prospring3.ch5.lifecycle.SimpleBean
В конфигурации бина обнаружена ошибка: Ошибка создания бина с именем
'simpleBean3', отпределенного в ресурсе пути классов [lifecycle/initMethod.xml]:
Сбой при вызове метода init; вложенным исключением является java.lang.
IllegalArgumentException: Должно быть установлено свойство age любого бина
типа com.apress.prospring3.ch5.lifecycle.SimpleBean
```

В этом выводе показано, что бин simpleBean1 был сконфигурирован корректно с применением значений, указанных в конфигурационном файле. В бине simpleBean2 для свойства name было использовано значение по умолчанию, поскольку в конфигурации никакого значения предусмотрено не было. Наконец, экземпляр бина simpleBean3 не был создан, т.к. метод init() сгенерировал ошибку из-за отсутствия значения для свойства age.

Как видите, использование метода инициализации является идеальным способом удостовериться, что бины сконфигурированы корректно. Благодаря этому механизму, вы можете получить все преимущества IoC, не теряя контроля, который дает определение зависимостей вручную. Единственное ограничение метода инициализации связано с тем, что он не может принимать аргументы. Можно определять любой возвращаемый тип, хотя он игнорируется Spring, и можно даже использовать статический метод, но этот метод не должен принимать аргументы.

Когда применяется статический метод инициализации, преимущества этого механизма сводятся на нет, поскольку отсутствует возможности получить доступ к состоянию любого бина для его проверки. Если бин использует статическое состояние с целью снижения расходов памяти, и для его проверки предусмотрен статический метод инициализации, вы должны рассмотреть возможность перемещения статического состояния в состояние экземпляра и применения нестатического метода инициализации. Когда используются возможности управления одиночными экземплярами, встроенные в Spring, конечный эффект будет таким же, но вы получаете бин, который намного проще тестировать, и также имеете дополнительную возможность создавать при необходимости множество экземпляров бина с собственными состояниями.

Конечно, бывают случаи, когда нужно использовать статическое состояние, разделяемое между множеством экземпляров бина, и вот тогда должен применяться статический метод инициализации.

Реализация интерфейса InitializingBean

Интерфейс InitializingBean, предлагаемый в Spring, позволяет определять внутри бина код, который будет выполняться, когда бин получает уведомление о том, что платформа Spring завершила его конфигурирования. Аналогично методу инициализации, это дает возможность проверить конфигурацию бина на предмет ее правильности, предоставляя при необходимости любые значения по умолчанию. Интерфейс InitializingBean определяет единственный метод afterPropertiesSet(), который предназначен для той же цели, что и метод init() в листинге 5.1. В листинге 5.3 показан переделанный предыдущий пример, в котором вместо метода инициализации применяется интерфейс InitializingBean.

Листинг 5.3. Использование интерфейса InitializingBean

```
package com.apress.prospring3.ch5.lifecycle;

import org.springframework.beans.factory.BeanCreationException;
import org.springframework.beans.factory.InitializingBean;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.GenericXmlApplicationContext;

public class SimpleBeanWithInterface implements InitializingBean {

   private static final String DEFAULT_NAME = "Luke Skywalker";

   private String name = null;

   private int age = Integer.MIN_VALUE;

   public void setName(String name) {
      this.name = name;
   }

   public void setAge(int age) {
      this.age = age;
   }

   public void myInit() {
      System.out.println("My Init");
   }
   public void afterPropertiesSet() throws Exception {
      System.out.println("Initializing bean");

      if (name == null) {
         System.out.println("Using default name");
         name = DEFAULT_NAME;
      }

      if (age == Integer.MIN_VALUE) {
         throw new IllegalArgumentException(
               "You must set the age property of any beans of type "
               + SimpleBean.class);
      }
   }

   public String toString() {
      return "Name: " + name + "\nAge: " + age;
   }
```

```
    public static void main(String[] args) {
        GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
        ctx.load("classpath:lifecycle/initInterface.xml");
        ctx.refresh();

        SimpleBeanWithInterface simpleBean1 = getBean("simpleBean1", ctx);
        SimpleBeanWithInterface simpleBean2 = getBean("simpleBean2", ctx);
        SimpleBeanWithInterface simpleBean3 = getBean("simpleBean3", ctx);
    }

    private static SimpleBeanWithInterface getBean(String beanName,
            ApplicationContext ctx) {
        try {
            SimpleBeanWithInterface bean =
                    (SimpleBeanWithInterface) ctx.getBean(beanName);
            System.out.println(bean);
            return bean;
        } catch (BeanCreationException ex) {
            System.out.println("An error occured in bean configuration: "
                    + ex.getMessage());
            return null;
        }
    }
}
```

Как видите, по сравнению с предыдущим примером изменилось немногое (измененный код выделен полужирным). Кроме очевидного изменения имени класса, отличия касаются того, что этот класс реализует интерфейс `InitializingBean`, и логика инициализации перемещена в метод `InitializingBean.afterPropertiesSet()`.

В листинге 5.4 приведена конфигурация для данного примера (`lifecycle/initInterface.xml`).

Листинг 5.4. Конфигурация для примера, использующего `InitializingBean`

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.1.xsd"
        default-lazy-init="true">

    <bean id="simpleBean1"
        class="com.apress.prospring3.ch5.lifecycle.SimpleBeanWithInterface">
        <property name="name">
            <value>Clarence Ho</value>
        </property>
        <property name="age">
            <value>100</value>
        </property>
    </bean>

    <bean id="simpleBean2"
        class="com.apress.prospring3.ch5.lifecycle.SimpleBeanWithInterface">
        <property name="age">
            <value>100</value>
        </property>
    </bean>
```

```
    <bean id="simpleBean3"
        class="com.apress.prospring3.ch5.lifecycle.SimpleBeanWithInterface">
        <property name="name">
            <value>Clarence Ho</value>
        </property>
    </bean>
</beans>
```

И снова между конфигурациями в листингах 5.4 и 5.2 отличий не очень много. Заметная разница связана с отсутствием атрибута `init-method`. Поскольку класс `SimpleBeanWithInterface` реализует интерфейс `InitializingBean`, платформе Spring известно, какой метод инициировать в качестве обратного вызова инициализации, поэтому отпадает потребность в любой дополнительной конфигурации. Ниже показан вывод, полученный в результате запуска этого примера:

```
Initializing bean
Name: Clarence Ho
Age: 100
Initializing bean
Using default name
Name: Luke Skywalker
Age: 100
Initializing bean
An error occured in bean configuration: Error creating bean with name
'simpleBean3' defined in class path resource [lifecycle/initInterface.xml]:
Invocation of init method failed; nested exception is java.lang.
IllegalArgumentException: You must set the age property of any beans of
type class com.apress.prospring3.ch5.lifecycle.SimpleBean
```

Несложно заметить, что вывод совпадает с выводом примера, в котором применялся механизм, основанный на методах.

Использование аннотации `@PostConstruct` стандарта JSR-250

Еще одним методом, с помощью которого можно достичь той же самой цели, является использование аннотации жизненного цикла `@PostConstruct`, соответствующей стандарту JSR-250. Начиная с версии 2.5, платформа Spring также поддерживает аннотации JSR-250 для указания метода, который Spring должна вызвать, если соответствующая аннотация, связанная с жизненным циклом бина, существует в классе. В листинге 5.5 представлен код программы, в котором применяется аннотация `@PostConstruct`.

Листинг 5.5. Использование аннотации `@PostConstruct`

```
package com.apress.prospring3.ch5.lifecycle;

import javax.annotation.PostConstruct;

// Остальные операторы import опущены.
public class SimpleBeanWithJSR250 {

    // Некоторый код опущен.
    @PostConstruct
    public void init() throws Exception {
        // Некоторый код опущен.
    }
}
```

Программа в точности соответствует ранее показанной программе `SimpleBean`; здесь просто к методу `init()` применяется аннотация `@PostConstruct`. Следует отметить, что этому методу можно назначить любое имя.

Что касается конфигурации, то можно воспользоваться конфигурацией, которая применяется механизмом, основанным на интерфейсах (см. листинг 5.4). Однако поскольку используется аннотация, понадобится добавить к дескриптору `<context:annotation-driven>` в конфигурационном файле пространство имен `context`.

Запустив программу, вы увидите тот же самый вывод, что и производимый при использовании других механизмов.

Как обсуждалось ранее, все три подхода характеризуются своими достоинствами и недостатками. Применение метода инициализации обладает достоинством отсутствия привязки приложения к Spring, но при этом понадобится сконфигурировать метод инициализации для каждого бина, которому он необходим. Использование `InitializingBean` дает возможность указывать обратный вызов инициализации один раз для всех экземпляров класса бина, но при этом приложение привязывается к Spring. Применение аннотаций к методу требует проверки, что контейнер IoC поддерживает JSR-250. В конечном счете, вы должны позволить требованиям приложения управлять решением относительно используемого подхода. Если проблемой является переносимость, применяйте подход с инициализацией или аннотациями; в противном случае используйте интерфейс `InitializingBean`, чтобы сократить объем потребностей в конфигурировании для приложения и снизить вероятность возникновения ошибок в приложении из-за неверной конфигурации.

Порядок разрешения

На одном и том же экземпляре бина можно использовать все механизмы. В таком случае Spring вызывает метод, аннотированный `@PostConstruct`, затем `InitializingBean.afterPropertiesSet()` и, наконец, ваш метод инициализации. Это полезно в ситуации, когда существующий бин выполняет некоторую инициализацию в определенном методе, но необходимо предусмотреть дополнительную инициализацию, если используется Spring.

Привязка к уничтожению бина

При использовании реализации `ApplicationContext`, которая помещает интерфейс `DefaultListableBeanFactory` в оболочку (такую как `GenericXmlApplicationContext`, через метод `GenericApplicationContext.getDefaultListableBeanFactory()`), можно сигнализировать фабрике бинов о том, что требуется уничтожить все одиночные экземпляры, с помощью вызова `ConfigurableBeanFactory.destroySingletons()`. Обычно это делается, когда приложение завершается, и такое действие позволяет освободить любые ресурсы, которые бины могут удерживать, аккуратно завершая приложение. Указанный обратный вызов является удобным местом для сброса данных, находящихся в памяти, в постоянное хранилище, а также для завершения длительно выполняющихся процессов, которые были запущены ранее.

Разрешить бинам получать уведомления о факте вызова метода `destroySingletons()` можно тремя способами, которые похожи на механизмы, предназначенные для получения обратного вызова инициализации. Обратный вызов уничтожения часто используется в сочетании с обратным вызовом инициализации. Во многих случаях в обратном вызове инициализации создается и конфигурируется некоторый ресурс, а в обратном вызове уничтожения этот ресурс освобождается.

Запуск метода, когда бин уничтожается

Чтобы назначить метод для вызова во время уничтожения бина, нужно просто указать имя соответствующего метода в атрибуте `destroy-method` дескриптора `<bean>` для бина. Платформа Spring вызывает этот метод непосредственно перед уничтожением одиночного экземпляра бина (как упоминалось ранее, Spring не будет вызывать этот метод для бинов с областью действия на уровне прототипа). В листинге 5.6 показан простой класс, реализующий `InitializingBean`, который в методе `afterPropertiesSet()` создает экземпляр `FileInputStream` и сохраняет его в закрытом поле.

Листинг 5.6. Использование обратного вызова `destroy-method`

```
package com.apress.prospring3.ch5.lifecycle;

import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;

import org.springframework.beans.factory.InitializingBean;
import org.springframework.context.support.GenericXmlApplicationContext;

public class DestructiveBean implements InitializingBean {

   private InputStream is = null;

   public String filePath = null;

   public void afterPropertiesSet() throws Exception {

      System.out.println("Initializing Bean"); // инициализация бина

      if (filePath == null) {
         throw new IllegalArgumentException(
               "You must specify the filePath property of "
            // Свойство filePath должно быть установлено.
               + DestructiveBean.class);
       }

      is = new FileInputStream(filePath);
   }

   public void destroy() {

      System.out.println("Destroying Bean"); // уничтожение бина

      if (is != null) {
         try {
            is.close();
            is = null;
         } catch (IOException ex) {
            // Во время закрытия InputStream возникло исключение IOException.
            System.err.println("WARN: An IOException occured"
                  + " trying to close the InputStream");
         }
      }
   }

   public void setFilePath(String filePath) {
      this.filePath = filePath;
   }

   public static void main(String[] args) throws Exception {

      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:lifecycle/disposeMethod.xml");
```

```
        ctx.refresh();   // Обновить ApplicationContext после
                         // загрузки XML-файла конфигурации.

        DestructiveBean bean = (DestructiveBean) ctx.getBean("destructiveBean");

        System.out.println("Calling destroy()");    // вызов destroy()
        ctx.destroy();
        System.out.println("Called destroy()");     // destroy() вызван
    }
}
```

В этом классе также определен метод `destroy()`, в коде которого `FileInputStream` закрывается и устанавливается в `null`, освобождая ресурс и делая его доступным для сборщика мусора. Метод `main()` извлекает бин типа `DestructiveBean` из `GenericXmlApplicationContext` и затем вызывает его метод `destroy()` (который, в свою очередь, вызывает `ConfigurableBeanFactory.destroySingletons()`, находящийся в оболочке `ApplicationContext`), сообщая платформе Spring о том, что нужно уничтожить все одиночные экземпляры, которыми она управляет. Обратные вызовы инициализации и уничтожения выводят на консоль сообщение, информирующее о том, что они были вызваны. В листинге 5.7 показана конфигурация для бина `destructiveBean` (`lifecycle/disposeMethod.xml`).

Листинг 5.7. Конфигурирование обратного вызова `destroy-method`

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd">

   <bean id="destructiveBean"
      class="com.apress.prospring3.ch5.lifecycle.DestructiveBean"
      destroy-method="destroy">
      <property name="filePath">
         <value>d:/temp/test.txt</value>
      </property>
   </bean>

</beans>
```

Обратите внимание на указание метода `destroy()` в качестве обратного вызова уничтожения с использованием атрибута `destroy-method`. Запуск этого примера на выполнение дает следующий вывод:

```
Initializing Bean
Calling destroy()
Destroying Bean
Called destroy()
```

В этом выводе видно, что Spring сначала инициирует обратный вызов инициализации, после чего экземпляр `DestructiveBean` создает экземпляр `FileInputStream` и сохраняет его. Далее, во время вызова `destroy()`, платформа Spring проходит по набору управляемых ею одиночных экземпляров (в данном случае есть только один такой экземпляр) и инициирует любые обратные вызовы уничтожения, которые были определены. Именно здесь экземпляр `DestructiveBean` закрывает `FileInputStream` и устанавливает соответствующую ссылку в `null`.

Реализация интерфейса DisposableBean

Как и для обратных вызовов инициализации, Spring предоставляет интерфейс, в данном случае DisposableBean, который бины могут реализовать в качестве механизма для получения обратных вызовов уничтожения. В интерфейсе DisposableBean определен единственный метод destroy(), который вызывается непосредственно перед уничтожением бина. Этот интерфейс используется подобно интерфейсу InitializingBean. В листинге 5.8 приведена модифицированная реализация класса DestructiveBean, реализующего интерфейс DisposableBean.

Листинг 5.8. Реализация DisposableBean

```
package com.apress.prospring3.ch5.lifecycle;

import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;

import org.springframework.beans.factory.DisposableBean;
import org.springframework.beans.factory.InitializingBean;
import org.springframework.context.support.GenericXmlApplicationContext;

public class DestructiveBeanWithInterface implements InitializingBean,
      DisposableBean {

   private InputStream is = null;

   public String filePath = null;

   public void afterPropertiesSet() throws Exception {

      System.out.println("Initializing Bean");    // инициализация бина

      if (filePath == null) {
         throw new IllegalArgumentException(
               "You must specify the filePath property of "
                + DestructiveBean.class);
      }
      is = new FileInputStream(filePath);
   }

   public void destroy() {

      System.out.println("Destroying Bean");      // уничтожение бина

      if (is != null) {
         try {
            is.close();
            is = null;
         } catch (IOException ex) {
            System.err.println("WARN: An IOException occured"
                               + " trying to close the InputStream");
         }
      }
   }

   public void setFilePath(String filePath) {
      this.filePath = filePath;
   }

   public static void main(String[] args) throws Exception {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:lifecycle/disposeInterface.xml");
      ctx.refresh();
```

```
        DestructiveBeanWithInterface bean = (DestructiveBeanWithInterface)
            ctx.getBean("destructiveBean");

        System.out.println("Calling destroySingletons()");
                          // вызов destroySingletons()
        ctx.destroy();
        System.out.println("Called destroySingletons()");
                          // destroySingletons() вызван

    }
}
```

И снова разница между кодом, использующим метод обратного вызова, и кодом, в котором применяется интерфейс обратного вызова, незначительна (отличия выделены полужирным). В этом случае мы даже использовали одинаковые имена методов. В листинге 5.9 показана уточненная конфигурация для данного примера (`lifecycle/disposeInterface.xml`).

Листинг 5.9. Конфигурация для примера использования интерфейса `DisposableBean`

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd">

   <bean id="destructiveBean"
     class="com.apress.prospring3.ch5.lifecycle.DestructiveBeanWithInterface">
       <property name="filePath">
          <value>d:/temp/test.txt</value>
       </property>
   </bean>

</beans>
```

Как видите, кроме отличающегося имени класса, единственная разница заключается в отсутствии атрибута `destroy-method`.

Запуск этого примера дает следующий вывод:

```
Initializing Bean
Calling destroy()
Destroying Bean
Called destroy()
```

Легко заметить, что вывод, полученный в результате использования двух разных механизмов, в точности совпадает.

Использование аннотации `@PreDestroy` стандарта JSR-250

Третий способ предусматривает применение аннотации жизненного цикла `@PreDestroy`, соответствующей стандарту JSR-250, которая похожа на `@PostConstruct`. В листинге 5.10 представлена версия класса `DestructiveBean`, в которой используются аннотации `@PostConstruct` и `@PreDestroy` для выполнения действий, связанных с инициализацией и уничтожением.

Листинг 5.10. Реализация `DisposableBean` с использованием `@PreDestroy`

```
package com.apress.prospring3.ch5.lifecycle;

import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;

import org.springframework.context.support.GenericXmlApplicationContext;

public class DestructiveBeanWithJSR250 {

   private InputStream is = null;

   public String filePath = null;

   @PostConstruct
   public void afterPropertiesSet() throws Exception {

      System.out.println("Initializing Bean");    // инициализация бина

      if (filePath == null) {
         throw new IllegalArgumentException(
                "You must specify the filePath property of "
                 + DestructiveBean.class);
      }

      is = new FileInputStream(filePath);
   }

   @PreDestroy
   public void destroy() {

      System.out.println("Destroying Bean");      // уничтожение бина

      if (is != null) {
         try {
            is.close();
            is = null;
         } catch (IOException ex) {
            System.err.println("WARN: An IOException occured"
                   + " trying to close the InputStream");
         }
      }
   }

   public void setFilePath(String filePath) {
      this.filePath = filePath;
   }

   public static void main(String[] args) throws Exception {

      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:lifecycle/disposeJSR250.xml");
      ctx.refresh();

      DestructiveBeanWithJSR250 bean = (DestructiveBeanWithJSR250)
         ctx.getBean("destructiveBean");

      System.out.println("Calling destroy()");    // вызов destroy()
      ctx.destroy();
      System.out.println("Called destroy()");     // destroy() вызван
   }
}
```

В листинге 5.11 показана XML-конфигурация для этого бина, в которую добавлен дескриптор `<context:annotation-config>` (`lifecycle/disposeJSR250.xml`).

Листинг 5.11. Конфигурация для примера использования `DisposableBean` с аннотацией JSR-250

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:context="http://www.springframework.org/schema/context"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context-3.1.xsd">

   <context:annotation-config/>

   <bean id="destructiveBean"
      class="com.apress.prospring3.ch5.lifecycle.DestructiveBeanWithJSR250">
      <property name="filePath">
         <value>d:/temp/test.txt</value>
      </property>
   </bean>

</beans>
```

Запуск этой программы на выполнение даст тот же самый вывод, что и программы, в которых применялись предыдущие два механизма.

Обратный вызов уничтожения — это идеальный механизм для аккуратного завершения приложений, при котором ресурсы гарантированно не останутся в незакрытом или несогласованном состоянии. Однако по-прежнему приходится решать, какой подход использовать: метод обратного вызова при уничтожении, интерфейс `DisposableBean` или аннотацию `@PreDestroy`. Как и ранее, позвольте требованиям приложения управлять решением этого вопроса; применяйте метод обратного вызова, если важна переносимость, а интерфейс `DisposableBean` или аннотацию JSR-250, чтобы снизить объем необходимой конфигурации.

Порядок разрешения

Как и в случае с созданием бина, при уничтожении бина можно использовать все три механизма на одном и том же экземпляре бина. В такой ситуации Spring сначала вызывает метод, аннотированный `@PreDestroy`, затем метод `DisposableBean.destroy()` и, наконец, ваш метод уничтожения.

Использование перехватчика завершения

Единственный недостаток обратных вызовов уничтожения в Spring состоит в том, что они не запускаются автоматически; нужно не забыть вызвать `AbstractApplicationContext.destroy()` перед закрытием приложения. Когда приложение выполняется как сервлет, указанный метод `destroy()` можно вызвать в методе `destroy()` сервлета. Однако в автономном приложении все не так просто, особенно при наличии множества точек выхода из приложения. К счастью, решение есть. Java позволяет создать перехватчик завершения (shutdown hook) — поток, который выполняется непосредственно перед завершением приложения. Это отличный способ вызвать метод `destroy()` вашего `AbstractApplicationContext` (который расширяется всеми конкретными реализациями `ApplicationContext`). Проще всего задейство-

вать данный механизм, воспользовавшись методом `registerShutdownHook()` класса `AbstractApplicationContext`. Этот метод автоматически заставляет Spring зарегистрировать перехватчик завершения лежащей в основе исполняющей среды JVM. В листинге 5.12 приведен пример.

Листинг 5.12. Регистрация перехватчика завершения

```
package com.apress.prospring3.ch5.lifecycle;

import org.springframework.context.support.GenericXmlApplicationContext;

public class ShutdownHookExample {

   public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:lifecycle/disposeInterface.xml");
      ctx.registerShutdownHook();
      ctx.refresh();

      DestructiveBeanWithInterface bean = (DestructiveBeanWithInterface)
         ctx.getBean("destructiveBean");
   }
}
```

Запуск этого примера дает в результате следующий вывод:

```
Initializing Bean
Destroying Bean
```

Как видите, метод `destroy()` был вызван, даже несмотря на то, что мы не написали ни единой строки кода для его явного вызова при завершении приложения.

Превращение бинов в “осведомленные о платформе Spring”

Одно из крупнейших преимуществ внедрения зависимостей перед поиском зависимостей как механизма инверсии управления заключается в том, что бины не обязаны знать детали реализации контейнера, который ими управляет. Для бина, который использует внедрение зависимостей через конструктор или метод установки, контейнер Spring ничем не отличается от контейнера, предоставляемого Google Guice, или, скажем, контейнера PicoContainer. Однако при определенных обстоятельствах может потребоваться бин, который применяет внедрение зависимостей для получения своих зависимостей, так что он может взаимодействовать с контейнером по какой-то другой причине. Примером может служить бин, который автоматически конфигурирует перехватчик завершения, поэтому ему необходим доступ к `ApplicationContext`. В других случаях бину может понадобиться узнать свое имя (т.е. имя бина, назначенное внутри текущего `ApplicationContext`), чтобы на основе этого предпринимать то или иное действие.

В действительности это средство предназначено для внутреннего употребления Spring. Предоставление имени некоторого бизнес-смысла в общем случае является неудачной задумкой и может привести к проблемам конфигурирования, когда именами бинов придется искусственно манипулировать с целью поддержки их бизнес-смысла. Тем не менее, мы считаем, что возможность бина выяснить свое имя во время выполнения действительно полезна для целей регистрации в журналах. Представьте ситуацию, в которой есть множество бинов одного и того же типа, выполняющихся с различны-

ми конфигурациями. Имя бина может быть включено в журнальные сообщения, что поможет отличать бины, генерирующие ошибки, от бинов, которые функционировали нормально, когда что-то пошло не так, как было задумано.

Использование интерфейса BeanNameAware

Интерфейс BeanNameAware, который бин может реализовать, чтобы получить свое имя, имеет единственный метод: setBeanName(String). Платформа Spring вызывает метод setBeanName() после завершения конфигурирования бина, но перед любыми обратными вызовами жизненного цикла (инициализации или уничтожения), как было показано на рис. 5.1. В большинстве случаев реализация метода setBeanName() сводится к одной строке, в которой значение, переданное контейнером, сохраняется в каком-то поле для дальнейшего использования. В листинге 5.13 приведен код простого бина, который получает свое имя с помощью BeanNameAware и затем применяет это имя при записи сообщений в журнал.

Листинг 5.13. Реализация BeanNameAware

```
package com.apress.prospring3.ch5.interaction;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.BeanNameAware;

public class LoggingBean implements BeanNameAware {

   private static final Log log = LogFactory.getLog(LoggingBean.class);

   private String beanName = null;

   public void setBeanName(String beanName) {
      this.beanName = beanName;
   }

   public void someOperation() {
      if(log.isInfoEnabled()) {
         log.info("Bean [" + beanName + "] - someOperation()");
      }
   }
}
```

Представленная реализация довольно проста. Вспомните, что BeanNameAware.setBeanName() вызывается перед тем, как первый экземпляр бина будет возвращен приложению через вызов ApplicationContext.getBean(), поэтому в методе someOperation() нет необходимости в проверке, доступно ли имя бина. В листинге 5.14 показана простая конфигурация для этого примера (interaction/logging.xml).

Листинг 5.14. Конфигурация для примера LoggingBean

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
     http://www.springframework.org/schema/beans/spring-beans-3.1.xsd">

  <bean id="loggingBean" class="com.apress.prospring3.ch5.interaction.LoggingBean"/>
</beans>
```

Как видите, никакой специальной конфигурации для использования интерфейса BeanNameAware не требуется. В листинге 5.15 приведен код простого приложения, которое извлекает экземпляр LoggingBean из ApplicationContext и затем вызывает метод someOperation() (interaction/logging.xml).

Листинг 5.15. Класс LoggingBeanExample

```
package com.apress.prospring3.ch5.interaction;
import org.springframework.context.support.GenericXmlApplicationContext;
public class LoggingBeanExample {
   public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:interaction/logging.xml");
      ctx.refresh();
      LoggingBean bean = (LoggingBean) ctx.getBean("loggingBean");
      bean.someOperation();
   }
}
```

Этот пример генерирует следующий вывод — обратите внимание на наличие имени бина в журнальном сообщении для вызова someOperation():

```
Loading XML bean definitions from class path resource [interaction/logging.xml]
Refreshing org.springframework.context.support.GenericXmlApplicationCont
ext@5dccce3c: startup date [Sun Sep 25 13:40:04 CST 2011]; root of context
hierarchy
Pre-instantiating singletons in org.springframework.beans.factory.support.
DefaultListableBeanFactory@7ec5495e: defining beans [loggingBean]; root of
factory hierarchy
Bean [loggingBean] - someOperation()
```

Использовать интерфейс BeanNameAware действительно просто, и он очень полезен, когда нужно улучшить качество журнальных сообщений. Не поддавайтесь искушению придавать именам бинов бизнес-смысл просто потому, что к ним можно получить доступ; поступая так, вы привязываете классы к Spring для получения функционального средства, которое приносит незначительную пользу. Если бины внутренне нуждаются в имени, они должны реализовать такой интерфейс, как Nameable (специфичный для приложения), имеющий метод setName(), и затем каждому бину назначается имя с применением внедрения зависимостей. В этом случае можно сохранить имена, используемые при конфигурировании, краткими, и не манипулировать конфигурацией с целью придания именам бинов бизнес-смысла.

Использование интерфейса ApplicationContextAware

За счет использования интерфейса ApplicationContextAware бины имеют возможность получать ссылку на контекст ApplicationContext, который их сконфигурировал. Основной причиной, по которой созданы эти интерфейсы, была необходимость предоставить бину доступ к ApplicationContext внутри приложения, например, для получения других бинов Spring программным образом, используя getBean(). Однако вы должны избегать подобной практики и применять внедрение зависимостей для предоставления своих бинов, а также их коллабораторов. Если вы используете основанный на поиске подход getBean() для получения зависимостей, когда можно применять внедрение зависимостей, то тем самым привносите ненужную сложность в свои бины и привязываете их к Spring Framework без веских причин.

Конечно, контекст ApplicationContext используется не только для поиска бинов; он решает множество других задач. Как упоминалось ранее, одной из задач является уничтожение всех одиночных экземпляров с предварительным уведомлением их по очереди. В предыдущем разделе было показано, как создать перехватчик завершения, чтобы сообщить ApplicationContext о необходимости уничтожения всех одиночных экземпляров перед завершением приложения. За счет использования интерфейса ApplicationContextAware можно построить бин, который может быть сконфигурирован в ApplicationContext на автоматическое создание и настройку перехватчика завершения. Код этого бина приведен в листинге 5.16.

Листинг 5.16. Класс ShutdownHookBean

```
package com.apress.prospring3.ch5.interaction;

import org.springframework.beans.BeansException;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationContextAware;
import org.springframework.context.support.GenericApplicationContext;

public class ShutdownHookBean implements ApplicationContextAware {

   private ApplicationContext ctx;
   public void setApplicationContext(ApplicationContext ctx)
      throws BeansException {
      if (ctx instanceof GenericApplicationContext) {
         ((GenericApplicationContext) ctx).registerShutdownHook();
      }
   }
}
```

Большая часть кода должна выглядеть знакомой.

В интерфейсе ApplicationContextAware определен единственный метод setApplicationContext(ApplicationContext), который Spring вызывает для передачи вашему бину ссылки на его ApplicationContext. В листинге 5.16 класс ShutdownHookBean проверяет, относится ли ApplicationContext к типу GenericApplicationContext, что означает его поддержку метода registerShutdownHook(); если это так, перехватчик завершения для ApplicationContext будет зарегистрирован. В листинге 5.17 показано, как сконфигурировать этот бин для работы с бином DestructiveBeanWithInterface, который использовался в предыдущем разделе (interaction/shutdownHook.xml).

Листинг 5.17. Конфигурирование ShutdownHookBean

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd">
   <bean id="destructiveBean"
      class="com.apress.prospring3.ch5.lifecycle.DestructiveBeanWithInterface">
      <property name="filePath">
         <value>d:/temp/test.txt</value>
      </property>
   </bean>
   <bean id="shutdownHook"
      class="com.apress.prospring3.ch5.interaction.ShutdownHookBean"/>
</beans>
```

Обратите внимание, что никакой специальной конфигурации не требуется. В листинге 5.18 приведен простой пример приложения, в котором ShutdownHookBean применяется для управления уничтожением одиночных экземпляров бинов.

Листинг 5.18. Использование ShutdownHookBean

```
package com.apress.prospring3.ch5.interaction;

import org.springframework.context.support.GenericXmlApplicationContext;

import com.apress.prospring3.ch5.lifecycle.DestructiveBeanWithInterface;

public class ShutdownHookBeanExample {

   public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("interaction/shutdownHook.xml");
      ctx.refresh();
      DestructiveBeanWithInterface bean =
         (DestructiveBeanWithInterface) ctx.getBean("destructiveBean");
   }
}
```

Показанный код также должен выглядеть знакомым. Когда платформа Spring производит начальную загрузку ApplicationContext, и бин destructiveBean определен в конфигурации (поскольку этот бин реализует интерфейс ApplicationContextAware), она передает ссылку на ApplicationContext бину shutdownHook для регистрации перехватчика завершения. Запуск этого примера, как и ожидалось, дает следующий вывод:

```
Initializing Bean
Destroying Bean
```

Как видите, даже при отсутствии обращений к destroy() в главном приложении, ShutdownHookBean зарегистрирован как перехватчик завершения, и он вызывает destroy() непосредственно перед завершением приложения.

Использование фабрик бинов

При использовании Spring вы столкнетесь с проблемой создания и внедрения зависимостей, которые нельзя создать просто с использованием операции new. Для решения этой проблемы Spring предоставляет интерфейс фабрики бинов FactoryBean, действующий в качестве адаптера для объектов, которые не могут быть созданы и управляемы стандартной семантикой Spring. Обычно фабрики бинов применяются для конструирования бинов, создать которые с помощью операции new невозможно, путем доступа к статическим фабричным методам, хотя это не всегда так. Попросту говоря, FactoryBean — это бин, действующий в качестве фабрики для других бинов. Фабрики бинов конфигурируются внутри ApplicationContext подобно обычным бинам, но когда платформа Spring использует интерфейс FactoryBean для удовлетворения запроса зависимости или поиска, не производится возврат FactoryBean; вместо этого вызывается метод FactoryBean.getObject() и возвращается результат этого вызова.

Фабрики бинов применяются в Spring для решения важных задач, наиболее заметными из которых является создание прокси для транзакций (см. главу 13) и автоматическое извлечение ресурсов из контекста JNDI. Тем не менее, фабрики бинов полезны не только для реализации внутренних функциональных средств Spring; вы сочтете их действительно удобными и при построении собственных приложений, поскольку они позволяют управлять намного большим числом ресурсов с использованием IoC, чем это было бы в противном случае.

Пример специальной фабрики бинов: класс MessageDigestFactoryBean

Часто разрабатываемые проекты требуют той или иной разновидности криптографической обработки; как правило, это связано с генерацией дайджеста сообщения или хеша пользовательского пароля для его сохранения в базе данных. В Java имеется класс MessageDigest, который предлагает функциональность для создания дайджеста из произвольных данных. Класс MessageDigest сам по себе является абстрактным; его конкретные реализации получаются вызовом метода MessageDigest.getInstance() с передачей ему имени алгоритма построения дайджеста, который должен использоваться. Например, если для создания дайджеста необходимо применять алгоритм MD5, то получить экземпляр MessageDigest можно с помощью следующего кода:

```
MessageDigest md5 = MessageDigest.getInstance("MD5");
```

Если для управления созданием объекта MessageDigest должна использоваться платформа Spring, то лучшее, что можно сделать без FactoryBean — предусмотреть в бине определенное свойство, algorithmName, и затем применять обратный вызов инициализации для обращения к MessageDigest.getInstance(). Используя FactoryBean, можно инкапсулировать эту логику внутри бина. После этого любые бины, которым требуется экземпляр MessageDigest, могут просто объявлять свойство, скажем, messageDigest, и с помощью FactoryBean получать нужный экземпляр. В листинге 5.19 показана реализация FactoryBean, которая делает как раз то, что описано выше.

Листинг 5.19. Класс MessageDigestFactoryBean

```
package com.apress.prospring3.ch5.factory;

import java.security.MessageDigest;

import org.springframework.beans.factory.FactoryBean;
import org.springframework.beans.factory.InitializingBean;

public class MessageDigestFactoryBean implements
     FactoryBean<MessageDigest>, InitializingBean {

   private String algorithmName = "MD5";

   private MessageDigest messageDigest = null;

   public MessageDigest getObject() throws Exception {
      return messageDigest;
   }

   public Class<MessageDigest> getObjectType() {
      return MessageDigest.class;
   }


   public boolean isSingleton() {
      return true;
   }

   public void afterPropertiesSet() throws Exception {
      messageDigest = MessageDigest.getInstance(algorithmName);
   }

   public void setAlgorithmName(String algorithmName) {
      this.algorithmName = algorithmName;
   }
}
```

В интерфейсе FactoryBean объявлены три метода: getObject(), getObjectType() и isSingleton(). Платформа Spring вызывает метод getObject() для извлечения объекта, созданного FactoryBean. Это действительный объект, который передается другим бинам, использующим FactoryBean в качестве коллаборатора.

В листинге 5.19 видно, что MessageDigestFactoryBean передает клон сохраненного экземпляра MessageDigest, который создан в обратном вызове InitializingBean.afterPropertiesSet().

Метод getObjectType() позволяет сообщить Spring, какой тип объекта фабрика бинов будет возвращать. В качестве типа можно указать null, если тип заранее не известен (например, фабрика бинов создает объекты разных типов в зависимости от конфигурации, что может быть определено только после инициализации фабрики), но если тип задан, то Spring может использовать его для автосвязывания. В рассматриваемом случае возвращаемым типом является MessageDigest (в данной ситуации это класс, но можно попробовать вернуть тип интерфейса и заставить фабрику бинов создавать экземпляр конкретного класса реализации, если только это необходимо). Причина в том, что нам не известно, как конкретный тип будет возвращен (это не так уж важно, поскольку в любом случае все бины будут определять свои зависимости с использованием MessageDigest).

Свойство isSingleton() позволяет информировать Spring о том, что фабрика бинов управляет одиночным экземпляром. Не забывайте, что за счет установки атрибута singleton дескриптора <bean> для фабрики бинов вы сообщаете Spring о поддержке одиночных экземпляров самой фабрикой, но не возвращаемыми ею объектами.

А теперь давайте посмотрим, как задействовать фабрику бинов в приложении. В листинге 5.20 приведен код простого бина, который обслуживает два экземпляра MessageDigest и отображает дайджесты сообщения, передаваемого методу digest().

Листинг 5.20. Класс MessageDigester

```
package com.apress.prospring3.ch5.factory;
import java.security.MessageDigest;
public class MessageDigester {
   private MessageDigest digest1 = null;
   private MessageDigest digest2 = null;
   public void setDigest1(MessageDigest digest1) {
      this.digest1 = digest1;
   }
   public void setDigest2(MessageDigest digest2) {
      this.digest2 = digest2;
   }
   public void digest(String msg) {
      System.out.println("Using digest1");   // использование digest1
      digest(msg, digest1);
      System.out.println("Using digest2");   // использование digest2
      digest(msg, digest2);
   }
   private void digest(String msg, MessageDigest digest) {
      // Вывести используемый алгоритм.
      System.out.println("Using alogrithm: " + digest.getAlgorithm());
      digest.reset();
      byte[] bytes = msg.getBytes();
      byte[] out = digest.digest(bytes);
      System.out.println(out);
   }
}
```

В листинге 5.21 показана конфигурация BeanFactory, которая настраивает два бина MessageDigestFactoryBean, один для использования алгоритма SHA1, а другой — алгоритма по умолчанию, т.е. MD5 (factory/factory.xml).

Листинг 5.21. Конфигурирование бинов FactoryBean

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd">

   <bean id="shaDigest"
         class="com.apress.prospring3.ch5.factory.MessageDigestFactoryBean">
      <property name="algorithmName">
         <value>SHA1</value>
      </property>
   </bean>

   <bean id="defaultDigest"
         class="com.apress.prospring3.ch5.factory.MessageDigestFactoryBean"/>

   <bean id="digester"
      class="com.apress.prospring3.ch5.factory.MessageDigester">
      <property name="digest1">
         <ref local="shaDigest"/>
      </property>
      <property name="digest2">
         <ref local="defaultDigest"/>
      </property>
   </bean>
</beans>
```

Как видите, были сконфигурированы не только два бина MessageDigestFactoryBean, но также и MessageDigester, с использованием двух MessageDigestFactoryBean для предоставления значений свойствам digest1 и digest2. В случае бина defaultDigest, поскольку свойство algorithmName не указано, никакого внедрения не произойдет, и будет применяться алгоритм по умолчанию (т.е. MD5), закодированный в классе. В листинге 5.22 приведен базовый пример класса, который извлекает бин MessageDigester из фабрики бинов и создает дайджест простого сообщения.

Листинг 5.22. Использование MessageDigester

```
package com.apress.prospring3.ch5.factory;

import org.springframework.context.support.GenericXmlApplicationContext;

public class MessageDigestExample {

   public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:factory/factory.xml");
      ctx.refresh();

      MessageDigester digester = (MessageDigester) ctx.getBean("digester");
      digester.digest("Hello World!");

   }
}
```

Запуск этого примера дает в результате следующий вывод:

```
Using digest1
Using alogrithm: SHA1
[B@786c730
Using digest2
Using alogrithm: MD5
[B@217f242c
```

Здесь видно, что бин `MessageDigest` снабжен двумя реализациями `MessageDigest`, SHA1 и MD5, несмотря на то, что бины `MessageDigest` не были сконфигурированы в фабрике бинов. Именно так функционирует фабрика бинов.

Фабрики бинов не являются идеальным решением, когда вы работаете с классами, экземпляры которых не могут быть созданы с использованием операции `new`. Если вы работаете с объектами, созданными с помощью фабричного метода, и хотите применять эти классы в Spring-приложении, создайте фабрику бинов для действия в качестве адаптера, позволив классам пользоваться всеми преимуществами средств IoC, встроенных в Spring.

Доступ к фабрике бинов напрямую

Учитывая тот факт, что Spring автоматически отвечает на любые ссылки на фабрику бинов объектами, созданными этой фабрикой, возникает вопрос: можно ли получить доступ к фабрике бинов напрямую? Ответ: да, можно.

Доступ к фабрике бинов в действительности очень прост: нужно просто предварить имя бина амперсандом в вызове `getBean()`, как показано в листинге 5.23.

Листинг 5.23. Доступ к фабрикам бинов напрямую

```
package com.apress.prospring3.ch5.factory;

import java.security.MessageDigest;

import org.springframework.context.support.GenericXmlApplicationContext;

public class AccessingFactoryBeans {

   public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:factory/factory.xml");
      ctx.refresh();

      MessageDigest digest = (MessageDigest) ctx.getBean("shaDigest");

      MessageDigestFactoryBean factoryBean =
          (MessageDigestFactoryBean) ctx.getBean("&shaDigest");

      try {
         MessageDigest shaDigest = factoryBean.getObject();
         System.out.println(shaDigest.digest("Hello world".getBytes()));
      } catch (Exception ex) {
         ex.printStackTrace();
      }
   }
}
```

Это средство используется в нескольких местах кода Spring, но для применения его в приложении не должно возникать каких-либо причин. Интерфейс `FactoryBean` предназначен для использования в качестве части поддерживающей инфраструктуры, что-

бы дать возможность применять большее число классов приложения в настройке IoC. Избегайте прямого доступа к FactoryBean и затем ручного вызова метода getObject(), позволив это делать Spring; выполняя это вручную, вы делаете излишнюю работу и неизбежно привязываете свое приложение к специфическим деталям реализации, которые вполне могут быть изменены в будущем.

Использование атрибутов factory-bean и factory-method

Иногда нужно создавать экземпляры компонентов JavaBean, которые были предоставлены приложением третьей стороны, не поддерживающим Spring. Вы не знаете, как создавать экземпляр этого класса, но вам известно, что приложение третьей стороны предлагает класс, который можно использовать для получения экземпляра JavaBean, требующегося вашему Spring-приложению. В этом случае также могут применяться атрибуты factory-bean и factory-method дескриптора <bean> для бина Spring.

Чтобы проверить это в работе, в листинге 5.24 приведена еще одна версия MessageDigestFactory, которая предоставляет метод для возврата бина MessageDigest.

Листинг 5.24. Класс MessageDigestFactory

```
package com.apress.prospring3.ch5.factory;

import java.security.MessageDigest;

public class MessageDigestFactory {

   private String algorithmName = "MD5";

   public MessageDigest createInstance() throws Exception {
      return MessageDigest.getInstance(algorithmName);
   }

   public void setAlgorithmName(String algorithmName) {
      this.algorithmName = algorithmName;
   }
}
```

В листинге 5.25 показано, как сконфигурировать фабричный метод для получения соответствующего экземпляра бина MessageDigest (factory/factoryMethod.xml).

Листинг 5.25. Конфигурирование MessageDigestFactory

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd">

   <bean id="shaDigestFactory"
      class="com.apress.prospring3.ch5.factory.MessageDigestFactory">
      <property name="algorithmName">
         <value>SHA1</value>
      </property>
   </bean>

   <bean id="defaultDigestFactory"
      class="com.apress.prospring3.ch5.factory.MessageDigestFactory"/>
```

```
    <bean id="shaDigest"
         factory-bean="shaDigestFactory"
         factory-method="createInstance">
    </bean>

    <bean id="defaultDigest"
         factory-bean="defaultDigestFactory"
         factory-method="createInstance"/>

    <bean id="digester"
       class="com.apress.prospring3.ch5.factory.MessageDigester">
       <property name="digest1">
          <ref local="shaDigest"/>
       </property>
       <property name="digest2">
          <ref local="defaultDigest"/>
       </property>
    </bean>
</beans>
```

Обратите внимание, что в конфигурации определены два бина-фабрики дайджестов, один из которых использует SHA1, а другой — алгоритм по умолчанию. Затем для бинов `shaDigest` и `defaultDigest` мы инструктируем Spring о необходимости создавать их экземпляры, используя соответствующий бин-фабрику дайджеста сообщений (`factory-bean`), и указываем метод для получения экземпляра бина (`factory-method`). В листинге 5.26 приведен код класса, предназначенного для тестирования.

Листинг 5.26. Класс `MessageDigestFactory` в действии

```
package com.apress.prospring3.ch5.factory;

import org.springframework.context.support.GenericXmlApplicationContext;

public class MessageDigestFactoryExample {

   public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:factory/factoryMethod.xml");
      ctx.refresh();
      MessageDigester digester = (MessageDigester) ctx.getBean("digester");
      digester.digest("Hello World!");
   }
}
```

Запуск этой программы сгенерирует следующий вывод:

```
Using digest1
Using alogrithm: SHA1
[B@877ef83
Using digest2
Using alogrithm: MD5
[B@7a0d85cc
```

Редакторы свойств для компонентов JavaBean

Для тех, кто не очень хорошо знаком с концепциями компонентов JavaBean, сообщаем, что `PropertyEditor` (редактор свойств) — это интерфейс, который преобразует значение свойства в и из внутреннего представления типа String. Первоначально это

задумывалось как способ ввода значений свойств в виде строк (String) в редакторе с последующим их преобразованием в корректный тип. Однако, поскольку редакторы свойств по своей природе являются облегченными классами, они нашли применение во многих различных ситуациях, включая Spring.

Так как изрядная часть значений свойств в приложении, основанном на Spring, задается в файле конфигурации BeanFactory, в сущности, они являются строками. Тем не менее, свойства, для которых указываются эти значения, могут и не относиться к типу String. Таким образом, платформа Spring позволяет вместо создания множества искусственных свойств типа String определить редакторы свойств, которые будут управлять преобразованием значений String, заданных для свойств, в подходящие типы.

Встроенные редакторы свойств

В версии Spring 3.1 доступно 13 встроенных реализаций PropertyEditor, предварительно зарегистрированных в BeanFactory. В листинге 5.27 приведен код простого бина, в котором объявлено 13 свойств, по одному для каждого типа, поддерживаемого встроенными редакторами свойств.

Листинг 5.27. Использование встроенных редакторов свойств

```
package com.apress.prospring3.ch5.pe;

import java.io.File;
import java.io.InputStream;
import java.net.URL;
import java.util.Date;
import java.util.List;
import java.util.Locale;
import java.util.Properties;
import java.util.regex.Pattern;

import org.springframework.context.support.GenericXmlApplicationContext;

public class PropertyEditorBean {
   private byte[] bytes;                  // ByteArrayPropertyEditor

   private Class cls;                     // ClassEditor

   private Boolean trueOrFalse;           // CustomBooleanEditor

   private List<String> stringList;       // CustomCollectionEditor

   private Date date;                     // CustomDateEditor

   private Float floatValue;              // CustomNumberEditor

   private File file;                     // FileEditor

   private InputStream stream;            // InputStreamEditor

   private Locale locale;                 // LocaleEditor

   private Pattern pattern;               // PatternEditor

   private Properties properties;         // PropertiesEditor

   private String trimString;             // StringTrimmerEditor

   private URL url;                       // URLEditor

   public void setCls(Class cls) {
      System.out.println("Setting class: " + cls.getName());
      this.cls = cls;
   }
```

```
public void setFile(File file) {
    System.out.println("Setting file: " + file.getName());
    this.file = file;
}

public void setLocale(Locale locale) {
    System.out.println("Setting locale: " + locale.getDisplayName());
    this.locale = locale;
}

public void setProperties(Properties properties) {
    System.out.println("Loaded " + properties.size() + " properties");
    this.properties = properties;
}

public void setUrl(URL url) {
    System.out.println("Setting URL: " + url.toExternalForm());
    this.url = url;
}

public void setBytes(byte[] bytes) {
    System.out.println("Adding " + bytes.length + " bytes");
    this.bytes = bytes;
}

public void setTrueOrFalse(Boolean trueOrFalse) {
   System.out.println("Setting Boolean: " + trueOrFalse);
   this.trueOrFalse = trueOrFalse;
}

public void setStringList(List<String> stringList) {
   System.out.println("Setting string list with size: "
                      + stringList.size());
   this.stringList = stringList;
   for (String string: stringList) {
      System.out.println("String member: " + string);
   }
}

public void setDate(Date date) {
   System.out.println("Setting date: " + date);
   this.date = date;
}

public void setFloatValue(Float floatValue) {
   System.out.println("Setting float value: " + floatValue);
   this.floatValue = floatValue;
}

public void setStream(InputStream stream) {
   System.out.println("Setting stream: " + stream);
   this.stream = stream;
}

public void setPattern(Pattern pattern) {
   System.out.println("Setting pattern: " + pattern);
   this.pattern = pattern;
}

public void setTrimString(String trimString) {
   System.out.println("Setting trim string: " + trimString);
   this.trimString = trimString;
}
```

```
   public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:pe/builtin.xml");
      ctx.refresh();

      PropertyEditorBean bean =
         (PropertyEditorBean) ctx.getBean("builtInSample");
   }
}
```

В листинге 5.27 видно, что `PropertyEditorBean` имеет 13 свойств, при этом каждому свойству соответствует один из встроенных редакторов свойств. В листинге 5.28 показана простая конфигурация `BeanFactory`, задающая значения для всех этих свойств (`pe/builtin.xml`).

Листинг 5.28. Конфигурирование редакторов свойств

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:util="http://www.springframework.org/schema/util"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
      http://www.springframework.org/schema/util
      http://www.springframework.org/schema/util/spring-util-3.1.xsd">

   <bean class="org.springframework.beans.factory.config.CustomEditorConfigurer">
      <property name="customEditors">
         <map>
            <entry key="java.util.Date">
               <bean
        class="org.springframework.beans.propertyeditors.CustomDateEditor">
                  <constructor-arg>
                     <bean class="java.text.SimpleDateFormat">
                        <constructor-arg value="yyyy-MM-dd" />
                     </bean>
                  </constructor-arg>
                  <constructor-arg value="true" />
               </bean>
            </entry>
            <entry key="java.lang.String">
               <bean
      class="org.springframework.beans.propertyeditors.StringTrimmerEditor">
                  <constructor-arg value="true" />
               </bean>
            </entry>
         </map>
      </property>
   </bean>

   <bean id="builtInSample" class="com.apress.prospring3.ch5.pe.PropertyEditorBean">
      <property name="bytes">
         <value>Hello World</value>
      </property>
      <property name="cls">
         <value>java.lang.String</value>
      </property>
```

```
        <property name="trueOrFalse">
           <value>true</value>
        </property>
        <property name="stringList">
           <util:list>
              <value>String member 1</value>
              <value>String member 2</value>
           </util:list>
        </property>
        <property name="date">
           <value>2011-12-29</value>
        </property>
        <property name="floatValue">
           <value>123.45678</value>
        </property>
        <property name="file">
           <value>classpath:test.txt</value>
        </property>
        <property name="stream">
           <value>classpath:test.txt</value>
        </property>
        <property name="locale">
           <value>en_US</value>
        </property>
        <property name="pattern">
           <value>a*b</value>
        </property>
        <property name="properties">
           <value>
              name=foo
              age=19
           </value>
        </property>
        <property name="trimString">
           <value>   String need trimming   </value>
        </property>
        <property name="url">
           <value>http://www.springframework.org</value>
        </property>
    </bean>
</beans>
```

Как видите, несмотря на то, что все свойства `PropertyEditorBean` не относятся к типу `String`, значения для свойств указаны как простые строки. Также обратите внимание на регистрацию `CustomDateEditor` и `StringTrimmerEditor`, поскольку эти два редактора по умолчанию не зарегистрированы в Spring. Запуск данного примера дает следующий вывод:

```
Adding 11 bytes
Setting class: java.lang.String
Setting Boolean: true
Setting string list with size: 2
String member: String member 1
String member: String member 2
Setting date: Thu Dec 29 00:00:00 CST 2011
```

```
Setting float value: 123.45678
Setting file: test.txt
Setting stream: java.io.BufferedInputStream@6490832e
Setting locale: English (US)
Setting pattern: a*b
Loaded 2 properties
Setting trim string: String need trimming
Setting URL: http://www.springframework.org
```

Согласно этому выводу, Spring использует встроенные редакторы свойств для преобразования строковых представлений различных свойств в корректные типы. В табл. 5.1 приведена сводка по встроенным редакторам свойств, которые доступны в Spring.

Таблица 5.1. Встроенные редакторы свойств Spring

Редактор свойств	Описание
`ByteArrayPropertyEditor`	Преобразует значение `String` в массив байт
`ClassEditor`	Преобразует полностью определенное имя класса в экземпляр `Class`. При использовании этого редактора свойств следите за тем, чтобы не включить лишние пробелы перед или после имени класса, когда применяется `GenericXmlApplicationContext`; в противном случае будет сгенерировано исключение `ClassNotFoundException`
`CustomBooleanEditor`	Преобразует строку в Java-тип `Boolean`
`CustomCollectionEditor`	Преобразует исходную коллекцию (например, доступную через пространство имен `util` в Spring) в целевой тип `Collection`
`CustomDateEditor`	Преобразует строковое представление даты в значение `java.util.Date`. Этот редактор с желаемым форматом даты понадобится зарегистрировать в `ApplicationContext`
`CustomNumberEditor`	Преобразует строку в числовое значение, которым может быть `Integer`, `Long`, `Float` и `Double`
`FileEditor`	Преобразует строковый путь к файлу в экземпляр `File`. Spring не проверяет существование файла
`InputStreamEditor`	Преобразует строковое представление ресурса (например, файлового ресурса вида `file:D:/temp/test.txt` или `classpath:test.txt`) в свойство входного потока
`LocaleEditor`	Преобразует строковое представление локали, такое как `en-GB`, в экземпляр `java.util.Locale`
`Pattern`	Преобразует строку в JDK-объект `Pattern` или наоборот
`PropertiesEditor`	Преобразует строку в формате `ключ1=значение1 ключ2=значение2 ключ*n*=значение*n*` в экземпляр `java.util.Properties` с настройкой соответствующих свойств
`StringTrimmerEditor`	Выполняет усечение строковых значений перед внедрением. Этот редактор свойств понадобится явно зарегистрировать
`URLEditor`	Преобразует строковое представление URL в экземпляр `java.net.URL`

Приведенный в таблице набор редакторов свойств обеспечивает хорошую основу для работы с платформой Spring и существенно упрощает конфигурирование приложения с такими общими компонентами, как файлы и URL.

Создание специального редактора свойств

Хотя встроенные редакторы свойств покрывают некоторые стандартные случаи преобразования типов свойств, может возникнуть потребность в создании собственного редактора свойств для поддержки класса или набора классов, используемых в приложении.

Spring полностью поддерживает регистрацию специальных редакторов свойств; единственный недостаток заключается в том, что в интерфейсе `java.beans.PropertyEditor` присутствует большое количество методов, многие из которых не имеют отношения к решаемой задаче — преобразованию типов свойств. К счастью, в JDK 5 или более новых версиях предлагается класс `PropertyEditorSupport`, который специальные редакторы свойств могут расширять, оставляя вам реализацию только одного метода: `setAsText()`.

Давайте рассмотрим простой пример реализации специального редактора свойств. Предположим, что имеется класс `Name` с двумя свойствами — именем и фамилией. Код этого класса приведен в листинге 5.29.

Листинг 5.29. Класс `Name`

```
package com.apress.prospring3.ch5.pe;

public class Name {

   private String firstName;

   private String lastName;

   public Name() {
   }

   public Name(String firstName, String lastName) {
      this.firstName = firstName;
      this.lastName = lastName;
   }

   public String getFirstName() {
      return firstName;
   }

   public void setFirstName(String firstName) {
      this.firstName = firstName;
   }

   public String getLastName() {
      return lastName;
   }

   public void setLastName(String lastName) {
      this.lastName = lastName;
   }

   public String toString() {
      return "First name: " + firstName + " - Last name: " + lastName;
   }
}
```

Для упрощения конфигурации приложения разработаем специальный редактор, который преобразует строку с разделителем-пробелом в имя и фамилию для класса `Name`. Код этого специального редактора свойств показан в листинге 5.30.

Листинг 5.30. Класс NamePropertyEditor

```
package com.apress.prospring3.ch5.pe;

import java.beans.PropertyEditorSupport;

public class NamePropertyEditor extends PropertyEditorSupport {

   public void setAsText(String text) throws IllegalArgumentException {

      String[] name = text.split("\\s");
      Name result = new Name(name[0], name[1]);
      setValue(result);
   }
}
```

Как видите, редактор очень прост. Он расширяет класс `PropertyEditorSupport` из JDK и реализует метод `setAsText()`. В этом методе мы просто расщепляем `String` на массив строк, используя в качестве разделителя символ пробела. Затем создается экземпляр класса `Name`, при этом для имени передается строка до пробела, а для фамилии — строка после пробела. Наконец, преобразованное значение возвращается с помощью вызова метода `setValue()`.

Чтобы использовать `NamePropertyEditor` в приложении, его понадобится зарегистрировать в `ApplicationContext`. В листинге 5.31 приведена конфигурация `ApplicationContext`, в которой настраиваются `CustomEditorConfigurer` и `NamePropertyEditor` (`pe/custom.xml`).

Листинг 5.31. Использование CustomEditorConfigurer

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd">

   <bean name="customEditorConfigurer"
    class="org.springframework.beans.factory.config.CustomEditorConfigurer">
      <property name="customEditors">
         <map>
            <entry key="com.apress.prospring3.ch5.pe.Name">
               <bean
                 class="com.apress.prospring3.ch5.pe.NamePropertyEditor"/>
            </entry>
         </map>
      </property>
   </bean>
   <bean id="exampleBean"
         class="com.apress.prospring3.ch5.pe.CustomEditorExample">
      <property name="name">
         <value>Clarence Ho</value>
      </property>
   </bean>
</beans>
```

Относительно этой конфигурации следует отметить три момента. Во-первых, специальные редакторы свойств внедряются в класс `CustomEditorConfigurer` с применением свойства `customEditors` типа `Map`. Во-вторых, каждая запись в `Map` представля-

ет одиночный редактор свойств с ключом записи — именем класса, для которого этот редактор используется. Как видно в коде, ключом для NamePropertyEditor является com.apress.prospring3.ch5.pe.Name, т.е. класс, для которого этот редактор должен применяться. В-третьих, в качестве значения единственной записи Map используется объявление анонимного бина. Поскольку к этому бину никакие другие бины не получают доступ, имя ему не нужно, в результате чего бин можно объявить внутри дескриптора <entry>.

В листинге 5.32 приведен код класса CustomEditorExample, который регистрировался в качестве бина в листинге 5.31.

Листинг 5.32. Класс CustomEditorExample

```
package com.apress.prospring3.ch5.pe;

import org.springframework.context.support.GenericXmlApplicationContext;

public class CustomEditorExample {

   private Name name;

   public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:pe/custom.xml");
      ctx.refresh();

      CustomEditorExample bean =
         (CustomEditorExample) ctx.getBean("exampleBean");

      System.out.println(bean.getName());
   }
   public Name getName() {
      return name;
   }
   public void setName(Name name) {
      this.name = name;
   }
}
```

В этом коде нет ничего особенного. Запуск примера дает следующий вывод:

```
First name: Clarence - Last name: Ho
```

Это вывод из метода toString(), который был реализован в классе Name; он показывает, что имя и фамилия правильно заполнены Spring с помощью сконфигурированного редактора свойств NamePropertyEditor.

Начиная с версии 3, платформа Spring предлагает Converter SPI (service provider interface — интерфейс поставщика служб) и Formatter SPI, которые предоставляют простой и хорошо структурированный API-интерфейс для выполнения преобразований типов и форматирования полей. Это особенно полезно при разработке веб-приложений. Converter SPI и Formatter SPI подробно рассматриваются в главе 14.

Дополнительные сведения о конфигурации ApplicationContext

Хотя мы уже обсуждали интерфейс ApplicationContext, по большому счету, мы касались возможностей, связанных с интерфейсом BeanFactory, оболочкой для которого он является. В Spring различные реализации интерфейса BeanFactory отвечают

за создание экземпляров бина, обеспечивая внедрение зависимостей и поддержку жизненного цикла для бинов, управляемых Spring. Тем не менее, как уже утверждалось, помимо расширения BeanFactory, интерфейс ApplicationContext предлагает также и другую полезную функциональность.

Главная функция ApplicationContext заключается в предоставлении намного более развитой инфраструктуры, на основе которой строятся приложения. Интерфейс ApplicationContext гораздо больше (чем интерфейс BeanFactory) осведомлен о бинах, которые сконфигурированы внутри него, и подобно многим классам и интерфейсам инфраструктуры Spring, таким как BeanFactoryPostProcessor, ApplicationContext взаимодействует с бинами от вашего имени, сокращая объем кода, необходимого для использования Spring.

Самое большое преимущество от работы с ApplicationContext связано с тем, что он позволяет конфигурировать и управлять платформой Spring и управляемыми ею ресурсами полностью декларативным путем. Это означает, что по мере возможности Spring предоставляет классы поддержки для автоматической загрузки ApplicationContext в приложение, устраняя необходимость в написании какого-либо кода для доступа к ApplicationContext. На практике это средство доступно пока что только при построении веб-приложений с помощью Spring, позволяя инициализировать ApplicationContext в дескрипторе развертывания веб-приложения. В автономном приложении инициализация ApplicationContext требует простого кодирования.

Вдобавок к предоставлению модели, которая больше сосредоточена на декларативной конфигурации, интерфейс ApplicationContext поддерживает следующие дополнительные средства:

- интернационализация;
- публикация событий;
- управление и доступ к ресурсам;
- дополнительные интерфейсы жизненного цикла;
- улучшенное автоматическое конфигурирование компонент инфраструктуры.

В последующих разделах мы обсудим некоторые наиболее важные средства в ApplicationContext кроме DI.

Интернационализация с помощью интерфейса MessageSource

Одной из областей, в которых Spring действительно превосходит другие платформы, является поддержка интернационализации (i18n). Используя интерфейс MessageSource, приложение может получать доступ к строковым ресурсам, называемых *сообщениями*, которые сохранены для множества разных языков. Для каждого языка, который должен поддерживаться в приложении, обеспечивается список сообщений, имеющих ключи для соответствия сообщениям на других языках. Например, если нужно отобразить фразу “Быстрая бурая лиса перепрыгивает ленивую собаку” на английском и чешском языках, понадобится создать два сообщения, имеющих ключ msg; на английском она выглядит как “The quick brown fox jumped over the lazy dog” (панграмма, включающая все буквы алфавита), а на чешском — “Příšerne žlutoucký kun úpel dábelské ódy”.

Хотя для работы с MessageSource нет необходимости в использовании Application Context, этот интерфейс в действительности расширяет MessageSource и предоставляет специальную поддержку для загрузки сообщений и для обеспечения их доступности в рамках среды. Автоматическая загрузка сообщений доступна в любой среде, но автоматический доступ предоставляется только в определенных управляемых Spring сценариях, таких как применение инфраструктуры Spring MVC для построения веб-приложе-

ния. Несмотря на то что любой класс может реализовать ApplicationContextAware и таким образом получить доступ к автоматически загруженным приложениям, мы предложим лучшее решение в разделе “Использование MessageSource в автономных приложениях” далее в этой главе.

Перед тем, как продолжить, рекомендуем просмотреть документацию по классам Locale и ResourceBundle (http://download.oracle.com/javase/6/docs/api/index.html), если вы еще не знакомы с поддержкой интернационализации в Java.

Использование ApplicationContext и MessageSource

Помимо ApplicationContext, в Spring предоставляются три реализации MessageSource: ResourceBundleMessageSource, ReloadableResourceBundle MessageSource и StaticMessageSource. Реализация StaticMessageSource на самом деле не предназначена для использования в корпоративном приложении, поскольку ее нельзя конфигурировать внешне, а это, как правило, одно из главных требований при добавлении возможностей интернационализации к приложению. Реализация ResourceBundleMessageSource загружает сообщения с использованием Java-класса ResourceBundle. Реализация ReloadableResourceBundleMessageSource в основном такая же, но только поддерживает запланированную перезагрузку лежащих в основе исходных файлов.

Все три реализации MessageSource также реализуют еще один интерфейс по имени HierarchicalMessageSource, который позволяет вкладывать друг в друга экземпляры MessageSource. Это ключ к тому, как ApplicationContext работает с множеством экземпляров MessageSource.

Чтобы получить поддержку MessageSource, предлагаемую ApplicationContext, в конфигурации понадобится определить бин типа MessageSource с именем messageSource. Контекст ApplicationContext берет этот MessageSource и вкладывает его внутрь себя самого, позволяя доступ к сообщениям с использованием ApplicationContext. Для лучшего понимания того, что происходит, имеет смысл рассмотреть пример.

В листинге 5.33 показан код простого приложения, которое имеет доступ к набору сообщений для английской и чешской локалей.

Листинг 5.33. Исследование MessageSource

```
package com.apress.prospring3.ch5.context;

import java.util.Locale;

import org.springframework.context.support.GenericXmlApplicationContext;

public class MessageSourceDemo {
   public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:appContext/messageSource.xml");
      ctx.refresh();

      Locale english = Locale.ENGLISH;
      Locale czech = new Locale("cs", "CZ");

      System.out.println(ctx.getMessage("msg", null, english));
      System.out.println(ctx.getMessage("msg", null, czech));

      System.out.println(ctx.getMessage("nameMsg", new Object[] { "Clarence",
            "Ho" }, english));
   }
}
```

Пока что не переживайте по поводу вызовов `getMessage()`; мы вернемся к ним очень скоро. Сейчас достаточно знать, что они извлекают сообщения с ключами для указанной локали. В листинге 5.34 представлена конфигурация, используемая этим приложением (`appContext/messageSource.xml`).

Листинг 5.34. Конфигурирование бина `MessageSource`

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-3.1.xsd">

   <bean id="messageSource"
     class="org.springframework.context.support.ResourceBundleMessageSource">
       <property name="basenames">
          <list>
             <value>buttons</value>
             <value>labels</value>
          </list>
       </property>
   </bean>
</beans>
```

Здесь мы определяем бин `ResourceBundleMessageSource` по имени `messageSource`, как и требовалось, и конфигурируем его с набором имен для формирования основы его набора файлов. Java-класс `ResourceBundle`, который используется `ResourceBundleMessageSource`, работает на наборе файлов свойств, указанных с помощью базовых имен. При поиске сообщения для конкретной локали `ResourceBundle` ищет файл, имя которого состоит из комбинации базового имени и имени локали. Например, если базовым именем является `foo` и производится поиск сообщения для локали `en-GB` (британский английский), то `ResourceBundle` ищет файл по имени `foo_en_GB.properties`.

Содержимое файлов свойств для английского (`labels_en.properties`) и чешского (`labels_cs_CZ.properties`) языков, используемых в предыдущем примере, приведено, соответственно, в листингах 5.35 и 5.36.

Листинг 5.35. Файл `labels_en.properties`

```
msg=The quick brown fox jumped over the lazy dog
nameMsg=My name is {0} {1}
```

Листинг 5.36. Файл `labels_cs_CZ.properties`

```
msg=Príšerne žlutoucký kun úpel dábelské ódy
```

Запуск класса `MessageSourceDemo` из листинга 5.33 дает следующий вывод:

```
The quick brown fox jumped over the lazy dog
Príšerne žlutoucký kun úpel dábelské ódy
My name is Clarence Ho
```

Теперь этот пример стал вызывать даже больше вопросов. Что означают эти вызовы `getMessage()`? Почему мы использовали `ApplicationContext.getMessage()`, а не

получали доступ к бину ResourceBundleMessageSource напрямую? Мы ответим на эти вопросы по очереди.

Метод getMessage()

В интерфейсе MessageSource определены три перегруженных версии метода getMessage(), которые описаны в табл. 5.2.

Таблица 5.2. Перегруженные версии MessageSource.getMessage()

Сигнатура метода	Описание
getMessage (String, Object[], Locale)	Это стандартный метод getMessage(). Аргумент типа String представляет собой ключ сообщения, соответствующий ключу в файле свойств. В листинге 5.33 первый вызов getMessage() использует в качестве ключа msg, и это соответствует следующей записи в файле свойств для локали en: msg=The quick brown fox jumped over the lazy dog. Аргумент типа массива Object[] предназначен для хранения замен в сообщении. В третьем вызове getMessage() в листинге 5.33 мы передаем массив из двух элементов String. Сообщением, имеющим ключ nameMsg, было My name is {0} {1}. Числа в фигурных скобках являются заполнителями, каждый из которых замещается соответствующим элементом в массиве, переданном как аргумент. Последний аргумент, Locale, указывает ResourceBundleMessageSource, какой файл свойств искать. Несмотря на то что в первом и втором вызовах getMessage() в примере использовался один и тот же ключ, вызовы возвращали разные сообщения, соответствующие значению Locale, которое было передано getMessage()
getMessage (String, Object[], String, Locale)	Эта перегруженная версия работает аналогично getMessage(String, Object[], Locale), но принимает второй аргумент типа String, который позволяет передать значение по умолчанию на случай, если указанный ключ не доступен для заданной в Locale локали
getMessage (MessageSourceResolvable, Locale)	Эта перегруженная версия представляет собой специальный случай, который детально обсуждается в разделе “Интерфейс MessageSourceResolvable”

Зачем использовать ApplicationContext в качестве MessageSource?

Для ответа на этот вопрос мы должны немного забежать вперед и взглянуть на поддержку веб-приложений в Spring. Ответ на вопрос, в общем, может быть сформулирован так: вы не должны использовать ApplicationContext как MessageSource, когда это приводит к нежелательной привязке вашего бина к ApplicationContext (более подробно об этом речь пойдет в следующем разделе). Применять ApplicationContext необходимо при построении веб-приложения с помощью инфраструктуры MVC, поддерживаемой Spring.

Ключевым интерфейсом в Spring MVC является Controller. В отличие от платформ, подобных Struts, которые требуют реализации контроллеров путем наследования от конкретного класса, в Spring просто необходимо реализовать интерфейс Controller (либо аннотировать разрабатываемый класс контроллера с помощью аннотации @Controller). С учетом сказанного, Spring предоставляет коллекцию полезных базовых классов, которые вы будете использовать для реализации собственных контролле-

ров. Все эти базовые классы сами являются подклассами (прямо или косвенно) класса ApplicationObjectSupport, представляющего собой удобный суперкласс для любых объектов приложения, которые должны быть осведомлены об ApplicationContext.

Не забывайте, что в конфигурации веб-приложения ApplicationContext загружается автоматически. Класс ApplicationObjectSupport обращается к этому ApplicationContext, помещает его в оболочку объекта MessageSourceAccessor и делает доступным контроллеру через защищенный метод getMessageSourceAccessor(). Класс MessageSourceAccessor предлагает широкий спектр удобных методов для работы с MessageSource. Эта форма "автоматического внедрения" исключительно полезна; она устраняет необходимость в наличии открытого свойства MessageSource во всех контроллерах.

Однако это не самая веская причина использования ApplicationContext как MessageSource в веб-приложении. Главная причина применения ApplicationContext вместо ручного определения бина MessageSource заключается в том, что Spring, где это возможно, открывает ApplicationContext как MessageSource для уровня презентаций. Это означает, что при использовании библиотеки дескрипторов JSP, поддерживаемой Spring, дескриптор <spring:message> автоматически читает сообщения из ApplicationContext, и в случае применения JSTL дескриптор <fmt:message> делает то же самое.

Все указанные преимущества означают, что при разработке веб-приложения лучше использовать поддержку MessageSource в ApplicationContext, чем управлять экземпляром MessageSource отдельно. Это особенно верно, если принять во внимание, что для работы с упомянутым средством нужно всего лишь сконфигурировать бин MessageSource с именем messageSource.

Использование MessageSource в автономных приложениях

Когда бины MessageSource используются в автономных приложениях, в которых Spring не предлагает никакой дополнительной поддержки, кроме автоматического вложения бина MessageSource в ApplicationContext, делать доступными бины MessageSource лучше с помощью внедрения зависимостей. Вы можете решить сделать бины ApplicationContextAware, но это воспрепятствует их использованию в контексте BeanFactory. Добавьте к этому факт усложнения тестирования без какой-либо видимой пользы, и необходимость придерживаться внедрения зависимостей для доступа к объектам MessageSource в автономных приложениях станет очевидной.

Интерфейс MessageSourceResolvable

При поиске сообщения в MessageSource на месте ключа можно использовать объект, реализующий интерфейс MessageSourceResolvable, и набор аргументов. Наиболее широко этот интерфейс применяется в библиотеках проверки достоверности Spring для связывания объектов Error с их интернационализированными сообщениями об ошибках. Пример использования MessageSourceResolvable будет приведен в главе 17, при рассмотрении обработки ошибок в библиотеке Spring MVC.

События приложений

Еще одним средством ApplicationContext, отсутствующим в BeanFactory, является возможность публиковать и получать события, используя ApplicationContext в качестве брокера. В этом разделе мы посмотрим, как это делается.

Использование событий приложения

Событие — это класс, производный от ApplicationEvent, который сам является производным от java.util.EventObject. Любой бин может прослушивать события, реализовав интерфейс ApplicationListener<T>; при этом ApplicationContext автоматически регистрирует любой сконфигурированный бин, который реализует данный интерфейс, в качестве слушателя. События публикуются с помощью метода ApplicationEventPublisher.publishEvent(), поэтому публикующий их класс должен быть осведомлен об интерфейсе ApplicationContext (расширяющем интерфейс ApplicationEventPublisher). В веб-приложении это достигается просто, поскольку многие классы являются производными от классов Spring Framework, которые позволяют доступ к ApplicationContext через защищенный метод. В автономном приложении, чтобы публиковать события, бин должен реализовать интерфейс ApplicationContextAware.

В листинге 5.37 показан пример базового класса события.

Листинг 5.37. Создание класса события

```
package com.apress.prospring3.ch5.event;

import org.springframework.context.ApplicationEvent;

public class MessageEvent extends ApplicationEvent {

    private String msg;

    public MessageEvent(Object source, String msg) {
        super(source);
        this.msg = msg;
    }

    public String getMessage() {
        return msg;
    }
}
```

Этот код довольно прост; единственным моментом, который в нем следует отметить, является наличие в ApplicationEvent одного конструктора, принимающего ссылку на источник события. Это отражено в конструкторе MessageEvent. В листинге 5.38 приведен код для слушателя.

Листинг 5.38. Класс MessageEventListener

```
package com.apress.prospring3.ch5.event;

import org.springframework.context.ApplicationListener;

public class MessageEventListener implements ApplicationListener<MessageEvent> {

    public void onApplicationEvent(MessageEvent event) {
        MessageEvent msgEvt = (MessageEvent) event;
        System.out.println("Received: " + msgEvt.getMessage());
    }
}
```

В интерфейсе ApplicationListener определен единственный метод onApplicationEvent, который вызывается Spring, когда событие сгенерировано. Класс MessageEventListener заинтересован только в событиях типа MessageEvent (или его подклассов), для чего он реализует строго типизированный интерфейс ApplicationListener. При получе-

нии MessageEvent он выводит сообщение в stdout. Публикация событий осуществляется просто; нужно лишь создать экземпляр класса события и передать его методу ApplicationEventPublisher.publishEvent(), как показано в листинге 5.39.

Листинг 5.39. Публикация события

```
package com.apress.prospring3.ch5.event;

import org.springframework.beans.BeansException;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationContextAware;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Publisher implements ApplicationContextAware {
   private ApplicationContext ctx;
   public static void main(String[] args) {
      ApplicationContext ctx = new ClassPathXmlApplicationContext(
            "classpath:events/events.xml");
      Publisher pub = (Publisher) ctx.getBean("publisher");
      pub.publish("Hello World!");
      pub.publish("The quick brown fox jumped over the lazy dog");
   }

   public void setApplicationContext(ApplicationContext applicationContext)
         throws BeansException {
      this.ctx = applicationContext;

   }

   public void publish(String message) {
      ctx.publishEvent(new MessageEvent(this, message));
   }
}
```

В приведенном коде класс Publisher извлекает из ApplicationContext экземпляр самого себя и затем, используя метод publish(), публикует два события MessageEvent для ApplicationContext. Экземпляр бина Publisher получает доступ к ApplicationContext за счет реализации ApplicationContextAware. В листинге 5.40 показана конфигурация для этого примера (events/events.xml).

Листинг 5.40. Конфигурирование бинов ApplicationListener

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd">

   <bean id="publisher" class="com.apress.prospring3.ch5.event.Publisher"/>
   <bean id="messageEventListener"
      class="com.apress.prospring3.ch5.event.MessageEventListener"/>
</beans>
```

Обратите внимание, что для регистрации MessageEventListener с ApplicationContext никакой специальной конфигурации не требуется; платформа Spring делает это автоматически. Запуск примера дает в результате следующий вывод:

```
Received: Hello World!
Received: The quick brown fox jumped over the lazy dog
```

Соображения по поводу использования событий

В приложении нередко возникает необходимость в уведомлении определенных компонентов о тех или иных событиях. Часто это делается путем написания кода для явного уведомления каждого компонента или за счет применения какой-нибудь технологии обмена сообщениями вроде JMS. Недостаток подхода с написанием кода для уведомления каждого компонента по очереди состоит в том, что в этом случае компоненты привязываются к публикующему классу, что во многих случаях нежелательно.

Представьте ситуацию, когда вы кешируете подробную информацию о товаре внутри приложения, чтобы избежать лишних обращений к базе данных. Еще один компонент позволяет модифицировать подробную информацию о товаре и сохранять в базе данных. Чтобы данные в кеше не стали недействительными, компонент обновления явно уведомляет кеш о том, что информация о товаре изменилась. В этом примере компонент обновления связан с компонентом, который в действительности не имеет никакого отношения к его бизнес-ответственности. Более удачное решение может выглядеть так: компонент обновления публикует событие всякий раз, когда подробная информация о товаре изменяется, а заинтересованные компоненты, такие как кеш, прослушивают это событие. Преимущество этого решения заключается в том, что компоненты сохраняются несвязанными, а это упрощает удаление кеша, когда он больше не нужен, или добавление другого слушателя, который заинтересован в получении сведений об изменении подробной информации о товаре.

Применение JMS в этом случае было бы излишним, поскольку процесс объявления записи о товаре в кеше недействительной является быстрым и некритичным. Использование инфраструктуры для работы с событиями в Spring приводит к добавлению в приложение весьма незначительных накладных расходов.

Как правило, события применяются для реактивной логики, которая выполняется быстро и не является частью главной логики приложения. В предшествующем примере объявление записи о товаре в кеше недействительной происходит в ответ на обновление подробной информации о товаре, это осуществляется быстро (во всяком случае, так должно быть) и не относится к главной функции приложения. Для процессов, которые выполняются долго и формируют часть основной бизнес-логики, рекомендуется использовать JMS или аналогичную систему обмена сообщениями, такую как RabbitMQ. Основные преимущества применения JMS связаны с тем, что она больше подходит для длительно выполняющихся процессов, и по мере роста системы управляемую JMS обработку сообщений, которые содержат бизнес-информацию, можно при необходимости вынести на отдельную машину.

Доступ к ресурсам

Часто приложение нуждается в доступе к разнообразным ресурсам в той или иной форме. Может понадобиться доступ к конфигурационным данным, хранящимся в файле внутри файловой системы, данным образа, записанным в JAR-файле в пути классов, или каким-нибудь данным, расположенным на произвольном сервере. Платформа Spring предоставляет унифицированный механизм для доступа к ресурсам независимым от протокола способом. Это означает, что приложение может работать с файловым ресурсом одинаковым образом, где бы он ни находился: в файловой системе, в пути классов или на удаленном сервере.

В основе всей поддержки ресурсов Spring находится интерфейс `org.springframework.core.io.Resource`. В интерфейсе `Resource` определены десять самоочевидных методов: `contentLength()`, `exists()`, `getDescription()`, `getFile()`, `getFileName()`, `getURI()`, `getURL()`, `isOpen()`, `isReadable()` и `lastModified()`. В дополнение к этим десяти методам имеется еще один, не столь самоочевидный: `createRelative()`.

Метод `createRelative()` создает новый экземпляр `Resource`, используя путь относительно экземпляра, на котором он вызывается. Можно построить собственные реализации `Resource`, хотя это и выходит за рамки контекста этой главы, но в большинстве случаев будет применяться одна из встроенных реализаций для доступа к файлу (класс `FileSystemResource`), пути классов (класс `ClassPathResource`) или URL-ресурсам (класс `UrlResource`). Для поиска и создания экземпляров `Resource` платформа Spring внутренне использует другой интерфейс, `ResourceLoader`, и его стандартную реализацию `DefaultResourceLoader`. Однако обычно вы не будете взаимодействовать с `DefaultResourceLoader`, а вместо этого работать с другой реализацией `ResourceLoader` — `ApplicationContext`.

В листинге 5.41 приведен пример приложения, которое обращается с тремя ресурсами, используя `ApplicationContext`.

Листинг 5.41. Доступ к ресурсам

```
package com.apress.prospring3.ch5.resource;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.core.io.Resource;

public class ResourceDemo {

   public static void main(String[] args) throws Exception{
      ApplicationContext ctx = new ClassPathXmlApplicationContext(
      "classpath:events/events.xml");

      Resource res1 = ctx.getResource("file:///d:/temp/test.txt");
      displayInfo(res1);
      Resource res2 = ctx.getResource("classpath:test.txt");
      displayInfo(res2);
      Resource res3 = ctx.getResource("http://www.google.co.uk");
      displayInfo(res3);
   }

   private static void displayInfo(Resource res) throws Exception{
      System.out.println(res.getClass());
      System.out.println(res.getURL().getContent());
      System.out.println("");
   }
}
```

Следует отметить, что конфигурационный файл в этом примере не важен. Обратите внимание, что в каждом вызове `getResource()` передается URI для каждого ресурса. Вы должны узнать общепринятые протоколы `file:` и `http:`, указанные для `res1` и `res3`. Протокол `classpath:`, используемый для `res2`, является специфичным для Spring и указывает, что `ResourceLoader` должен искать ресурс в пути классов. Запуск этого примера дает в результате следующий вывод:

```
class org.springframework.core.io.UrlResource
sun.net.www.content.text.PlainTextInputStream@709446e4

class org.springframework.core.io.ClassPathResource
sun.net.www.content.text.PlainTextInputStream@16ba5c7a

class org.springframework.core.io.UrlResource
sun.net.www.protocol.http.HttpURLConnection$HttpInputStream@6825c828
```

Как видите, для протоколов `file:` и `http:` платформа Spring возвращает экземпляр `UrlResource`. Она включает класс `FileSystemResource`, но `DefaultResourceLoader` вообще не использует этот класс. Причина в том, что стандартная стратегия загрузки ресурсов в Spring трактует URL и файл как один и тот же тип ресурса, но с отличающимися протоколами (т.е. `file:` и `http:`). Если экземпляр `FileSystemResource` обязателен, применяйте `FileSystemResourceLoader`. Получив экземпляр `Resource`, вы можете работать с содержимым ресурса по своему усмотрению, используя `getFile()`, `getInputStream()` или `getURL()`. В ряде случаев, например, когда применяется протокол `http:`, вызов `getFile()` дает в результате исключение `FileNotFoundException`. По этой причине для доступа к содержимому ресурсов рекомендуется использовать метод `getInputStream()`, потому что он, скорее всего, будет функционировать со всеми возможными типами ресурсов.

Конфигурация, использующая Java-классы

Кроме XML для конфигурирования `ApplicationContext` можно также использовать Java-классы. Ранее для этого применялся отдельный проект Spring JavaConfig, но, начиная с версии Spring 3.0, его основные средства, касающиеся конфигурирования с помощью Java-классов, были объединены с ядром Spring Framework.

В этом разделе мы посмотрим, как использовать Java-классы для конфигурирования `ApplicationContext`, и приведем эквивалентную XML-конфигурацию.

Конфигурирование `ApplicationContext` в Java

Мы рассмотрим конфигурирование `ApplicationContext` с использованием Java-классов на том же самом примере поставщика и визуализатора сообщений, который был представлен в главе 4. Для удобства в листинге 5.42 повторно приведен код интерфейса поставщика сообщений и конфигурируемого поставщика сообщений.

Листинг 5.42. `MessageProvider` и `ConfigurableMessageProvider`

```
package com.apress.prospring3.ch5.javaconfig;

public interface MessageProvider {

    public String getMessage();

}

package com.apress.prospring3.ch5.javaconfig;

public class ConfigurableMessageProvider implements MessageProvider {

    private String message = "Default message";
    public ConfigurableMessageProvider() {
    }

    public ConfigurableMessageProvider(String message) {
        this.message = message;
    }

    public void setMessage(String message) {
        this.message = message;
    }

    public String getMessage() {
        return message;
    }
}
```

В листинге 5.43 показаны интерфейс `MessageRenderer` и реализация `StandardOutMessageRenderer`.

Листинг 5.43. `MessageRenderer` и `StandardOutMessageRenderer`

```
package com.apress.prospring3.ch5.javaconfig;

public interface MessageRenderer {

   public void render();

   public void setMessageProvider(MessageProvider provider);

   public MessageProvider getMessageProvider();

}

package com.apress.prospring3.ch5.javaconfig;

public class StandardOutMessageRenderer implements MessageRenderer {

   private MessageProvider messageProvider = null;

   public void render() {
      if (messageProvider == null) {
         throw new RuntimeException(
            "You must set the property messageProvider of class:"
            + StandardOutMessageRenderer.class.getName());
      }
      System.out.println(messageProvider.getMessage());
   }

   public void setMessageProvider(MessageProvider provider){
      this.messageProvider = provider;
   }

   public MessageProvider getMessageProvider() {
      return this.messageProvider;
   }

}
```

В листинге 5.44 представлена XML-конфигурация для `ApplicationContext` (`app-context.xml`).

Листинг 5.44. XML-конфигурация `ApplicationContext`

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:p="http://www.springframework.org/schema/p"
   xmlns:c="http://www.springframework.org/schema/c"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd">

   <bean id="messageRenderer"
     class="com.apress.prospring3.ch5.javaconfig.StandardOutMessageRenderer"
     p:messageProvider-ref="messageProvider"/>

   <bean id="messageProvider"
     class="com.apress.prospring3.ch5.javaconfig.ConfigurableMessageProvider"
     c:message="This is a configurable message"/>

</beans>
```

Наконец, в листинге 5.45 приведен код тестовой программы.

Листинг 5.45. Тестовая программа для XML-конфигурации

```
package com.apress.prospring3.ch5.javaconfig;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class JavaConfigSimpleExample {

   public static void main(String[] args) {

      ApplicationContext ctx = new
        ClassPathXmlApplicationContext("classpath:app-context.xml");

      MessageRenderer renderer =
         ctx.getBean("messageRenderer", MessageRenderer.class);

      renderer.render();

   }
}
```

При использовании Java-класса вместо XML для конфигурирования показанных выше поставщика и визуализатора сообщений необходимо просто реализовать обычный компонент JavaBean, но с соответствующими аннотациями для Java-конфигурации в Spring. В листинге 5.46 показан Java-класс, который эквивалентен XML-конфигурации из листинга 5.44.

Листинг 5.46. Java-конфигурация

```
package com.apress.prospring3.ch5.javaconfig;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class AppConfig {

   // XML:
   // <bean id="messageProvider"
   // class="com.apress.prospring3.ch5.javaconfig.ConfigurableMessageProvider"/>
   @Bean
   public MessageProvider messageProvider() {
      return new ConfigurableMessageProvider();
   }

   // XML:
   // <bean id="messageRenderer"
   // class="com.apress.prospring3.ch5.javaconfig.StandardOutMessageRenderer"
   //       p:messageProvider-ref="messageProvider"/>
   @Bean
   public MessageRenderer messageRenderer() {
      MessageRenderer renderer = new StandardOutMessageRenderer();

      // Внедрение через метод установки.
      renderer.setMessageProvider(messageProvider());

      return renderer;
   }
}
```

В приведенном выше классе AppConfig сначала с помощью аннотации @Configuration платформе Spring сообщается, что это конфигурационный файл, основанный на Java. После этого для объявления бина Spring и требований DI применяется аннотация @Bean. В комментариях кода указаны XML-эквиваленты Java-аннотаций. Таким образом, вы видите, что аннотация @Bean эквивалентна дескриптору <bean>, имя метода эквивалентно атрибуту id дескриптора <bean>, а при создании экземпляра бина MessageRender внедрение зависимости через метод установки достигается вызовом соответствующего метода для получения поставщика сообщений, и это дает тот же самый результат, что и применение дескриптора <ref> в XML-конфигурации. В листинге 5.47 показано, как инициализировать ApplicationContext в Java-конфигурации.

Листинг 5.47. Тестирование Java-конфигурации

```
package com.apress.prospring3.ch5.javaconfig;

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotationAnnotationConfigApplicationCont
ext;

public class JavaConfigSimpleExample {
   public static void main(String[] args) {
      ApplicationContext ctx = new
         AnnotationConfigApplicationContext(AppConfig.class);

      MessageRenderer renderer =
         ctx.getBean("messageRenderer", MessageRenderer.class);

      renderer.render();
   }
}
```

В приведенном выше листинге используется класс AnnotationConfigApplication Context, конструктору которого передается в качестве аргумента класс конфигурации (ему также можно передавать множество классов конфигурации с помощью средства varargs из JDK). После этого возвращенный ApplicationContext можно использовать обычным образом.

Обратите внимание, что для поддержки классов конфигурации Spring требует CGLIB. Для этого понадобиться добавить к проекту в STS зависимость, описанную в табл. 5.3.

Таблица 5.3. Зависимость для Java-конфигурации

Идентификатор группы	Идентификатор артефакта	Версия	Описание
cglib	cglib	2.2.2	Библиотека генерации кода, требуемая для поддержки Java-конфигурации в Spring

Запуск этой программы даст следующий результат:

```
Default message
```

Ознакомившись с базовым применением Java-класса конфигурации, давайте перейдем к рассмотрению других вариантов конфигурации и их XML-эквивалентов. Пусть для поставщика сообщений необходимо вынести сообщение в файл свойств (message.properties) и затем внедрять его в ConfigurableMessageProvider с помощью Constructor Injection. Содержимое файла message.properties таково:

```
message=Spring 3 Java Configuration Rocks!
```

Взглянем на переделанную тестовую программу, которая загружает файлы свойств, используя аннотацию `@PropertySource`, и затем внедряет их в реализацию поставщика сообщений. В листинге 5.48 также добавлено много различных аннотаций, которые Spring поддерживает для базовой Java-конфигурации, и приведены соответствующие XML-эквиваленты. Обратите внимание, что для работы аннотации `@EnableTransactionManagement` к проекту необходимо добавить зависимость от модуля `spring-tx` (табл. 5.4).

Таблица 5.4. Зависимость для поддержки транзакций в Spring

Идентификатор группы	Идентификатор артефакта	Версия	Описание
org.springframework	spring-tx	3.1.0.RELEASE	Модуль Spring для поддержки транзакций

Листинг 5.48. Java-класс конфигурации (переделанный)

```
package com.apress.prospring3.ch5.javaconfig;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.DependsOn;
import org.springframework.context.annotation.Import;
import org.springframework.context.annotation.ImportResource;
import org.springframework.context.annotation.Lazy;
import org.springframework.context.annotation.PropertySource;
import org.springframework.context.annotation.Scope;
import org.springframework.core.env.Environment;
import org.springframework.transaction.annotation.
EnableTransactionManagement;

@Configuration
@Import(OtherConfig.class)
// XML: <import resource="classpath:events/events.xml")
@ImportResource(value="classpath:events/events.xml")
// XML: <context:property-placeholder location="classpath:message.properties"/>
@PropertySource(value="classpath:message.properties")
// XML: <context:component-scan base-package="com.apress.prospring3.ch5.context"/>
@ComponentScan(basePackages={"com.apress.prospring3.ch5.context"})
@EnableTransactionManagement
public class AppConfig {

    @Autowired
    Environment env;

    // XML:
    // <bean id="messageProvider"
    // class="com.apress.prospring3.ch5.javaconfig.ConfigurableMessageProvider"/>
    @Bean
    @Lazy(value=true)   // XML: <bean .... lazy-init="true"/>
    public MessageProvider messageProvider() {
        // Внедрение через конструктор.
        return new ConfigurableMessageProvider(env.getProperty("message"));
    }
```

```
    // XML:
    // <bean id="messageRenderer"
    // class="com.apress.prospring3.ch5.javaconfig.StandardOutMessageRenderer"
    //       p:messageProvider-ref="messageProvider"/>
    @Bean(name="messageRenderer")
    @Scope(value="prototype")
       // XML: <bean ... scope="prototype"/>
    @DependsOn(value="messageProvider")
       // XML: <bean ... depends-on="messageProvider"/>
    public MessageRenderer messageRenderer() {
       MessageRenderer renderer = new StandardOutMessageRenderer();

       // Внедрение через метод установки.
       renderer.setMessageProvider(messageProvider());

       return renderer;
    }
}
```

В листинге 5.48 класс `OtherConfig` — это просто пустой класс, аннотированный с помощью `@Configuration`. Здесь вы видите множество распространенных аннотаций для конфигурирования с помощью Java-классов; кроме того, для справочных целей приведены также их XML-эквиваленты. Ниже перечислены основные моменты, связанные с листингом 5.48.

- Аннотация `@PropertySource` используется для загрузки файлов свойств в `ApplicationContext` и принимает в качестве аргумента местоположение (допускается указывать более одного местоположения). Той же самой цели в XML служит дескриптор `<context:property-placeholder>`.
- Аннотация `@Import` позволяет импортировать другие классы конфигурации, а это означает, что можно иметь множество Java-классов конфигурации для поддержки разных конфигураций (например, один класс может быть выделен для объявления бинов DAO, один — для объявления бинов служб и т.д.).
- Помимо `@Import`, можно также использовать аннотацию `@ImportResource` для импортирования конфигурации из XML-файлов, а это означает возможность применять XML и Java-классы конфигурации, сочетая и совмещая их, хотя поступать подобным образом не рекомендуется. Смешивание конфигураций XML и Java усложнит сопровождение приложения, поскольку в поисках специфического бина нужно будет просматривать и XML-файлы, и Java-классы.
- Аннотация `@ComponentScan` определяет пакеты, которые платформа Spring должна сканировать на предмет аннотаций для определений бинов. Аналогичную роль играет дескриптор `<context:component-scan>` в XML-конфигурации.
- Остальные аннотации самоочевидны. Аннотация `@Lazy` заставляет Spring создавать экземпляры бина только при его запрашивании (то же, что и `lazy-init="true"` в XML), а `@DependsOn` сообщает Spring о том, что определенный бин зависит от ряда других бинов, поэтому Spring должна обеспечить создание их экземпляров первыми. Аннотация `@Scope` определяет область действия бина.
- Службы инфраструктуры приложения также могут быть определены в Java-классах. Например, `@EnableTransactionManagement` указывает, что будет использоваться средство управления транзакциями Spring, которое подробно рассматривается в главе 12.

- Вы могли также заметить свойство `@Autowired` переменной `env` типа `Environment`. Это средство абстрагирования `Environment`, предоставляемое Spring 3.1. Мы обсудим его далее в этой главе.

Запуск тестовой программы дает в результате следующий вывод:

```
Spring 3 Java Configuration Rocks!
```

Как видите, отобразилось сообщение, определенное в файле `message.properties`.

Выбор между конфигурациями Java и XML

Как уже должно быть понятно, применение Java-классов может обеспечить тот же самый уровень конфигурирования `ApplicationContext`, что и XML. Так каким же подходом воспользоваться? Соображение по этому поводу очень похоже на то, что принималось во внимание при выборе конфигурации DI — в стиле XML или в стиле аннотаций Java-аннотаций. Ясно, что каждый подход имеет свои плюсы и минусы. Тем не менее, рекомендация остается такой же: решив в команде, какой подход использовать, придерживайтесь его и сохраняйте стиль конфигурирования неизменным, не разбрасываясь между Java-классами и XML-файлами. В противном случае работа по сопровождению существенно усложнится.

Профили

Еще одной интересной возможностью, которая стала доступной разработчикам в версии Spring 3.1, является концепция профилей конфигурации. В сущности, профиль заставляет Spring конфигурировать только контекст `ApplicationContext`, который определялся, когда указанный профиль был активен. В этом разделе мы продемонстрируем использование профилей в простой программе.

Пример использования средства профилей Spring

Пусть имеется служба по имени `FoodProviderService`, которая отвечает за обеспечение едой учебных заведений, включая детский сад (kindergarten) и среднюю школу (high school). В интерфейсе `FoodProviderService` определен только один метод `provideLunchSet()`, который формирует обеденный набор для каждого учащегося вызываемого учебного заведения. Обеденный набор представляет собой список объектов `Food` — очень простого класса, который имеет единственный атрибут `name`. Код класса `Food` показан в листинге 5.49.

Листинг 5.49. Класс `Food`

```
package com.apress.prospring3.ch5.profile;

public class Food {

    private String name;

    public Food() {
    }

    public Food(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
```

```
    public void setName(String name) {
        this.name = name;
    }
}
```

В листинге 5.50 приведен интерфейс `FoodProviderService`.

Листинг 5.50. Интерфейс `FoodProviderService`

```
package com.apress.prospring3.ch5.profile;

import java.util.List;

public interface FoodProviderService {

    public List<Food> provideLunchSet();

}
```

Теперь предположим, что существуют два разных поставщика обеденных наборов, один для детского сада и один для средней школы. Хотя производимые ими обеденные наборы отличаются, предлагаемая ими служба является той же самой — поставка обедов учащимся. Итак, давайте создадим две разных реализации `FoodProviderService`, назначив им одно и то же имя, но поместив их в разные пакеты для идентификации целевых учебных заведений. Код для этих двух классов показан в листингах 5.51 и 5.52.

Листинг 5.51. Реализация `FoodProviderService` для детского сада

```
package com.apress.prospring3.ch5.profile.kindergarten;

import java.util.ArrayList;
import java.util.List;

import com.apress.prospring3.ch5.profile.Food;
import com.apress.prospring3.ch5.profile.FoodProviderService;

public class FoodProviderServiceImpl implements FoodProviderService {

    public List<Food> provideLunchSet() {
        List<Food> lunchSet = new ArrayList<Food>();
        lunchSet.add(new Food("Milk"));
        lunchSet.add(new Food("Biscuits"));

        return lunchSet;
    }
}
```

Листинг 5.52. Реализация `FoodProviderService` для средней школы

```
package com.apress.prospring3.ch5.profile.highschool;

import java.util.ArrayList;
import java.util.List;

import com.apress.prospring3.ch5.profile.Food;
import com.apress.prospring3.ch5.profile.FoodProviderService;

public class FoodProviderServiceImpl implements FoodProviderService {

    public List<Food> provideLunchSet() {
        List<Food> lunchSet = new ArrayList<Food>();
```

```
        lunchSet.add(new Food("Coke"));
        lunchSet.add(new Food("Hamburger"));
        lunchSet.add(new Food("French Fries"));

        return lunchSet;
    }
}
```

В этих листингах видно, что две разных реализации предоставляют один и тот же интерфейс `FoodProviderService`, но производят отличающиеся комбинации еды в обеденных наборах. А теперь предположим, что в детском саду хотят, чтобы поставщик доставлял обеденные наборы их воспитанникам; давайте посмотрим, как можно использовать конфигурацию профиля Spring для достижения этого. Сначала мы рассмотрим XML-конфигурацию.

Мы создадим два разных XML-файла конфигурации, один для профиля детского сада и один для профиля средней школы. В листингах 5.53 и 5.54 приведена конфигурация поставщиков еды для детского сада и средней школы, соответственно.

Листинг 5.53. XML-конфигурация для детского сада (`kindergarten-config.xml`)

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.1.xsd"
        profile="kindergarten">

    <bean id="foodProviderService"
class="com.apress.prospring3.ch5.profile.kindergarten.FoodProviderServiceImpl"/>

</beans>
```

Листинг 5.54. XML-конфигурация для средней школы (`highschool-config.xml`)

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.1.xsd"
        profile="highschool">

    <bean id="foodProviderService"
class="com.apress.prospring3.ch5.profile.highschool.FoodProviderServiceImpl"/>

</beans>
```

В предыдущих двух конфигурациях обратите внимание на использование, соответственно, атрибутов `profile="kindergarten"` и `profile="highschool"` в дескрипторе `<beans>`. Это сообщает Spring о том, что экземпляры бинов должны создаваться, только если активен указанный профиль. Давайте посмотрим, как активизировать корректный профиль, когда `ApplicationContext` применяется в автономном приложении. Тестовая программа приведена в листинге 5.55.

Листинг 5.55. Пример XML-конфигурации профиля

```
package com.apress.prospring3.ch5.profile;

import java.util.List;

import org.springframework.context.support.GenericXmlApplicationContext;

public class ProfileXmlConfigExample {

   public static void main(String[] args) {

      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.getEnvironment().setActiveProfiles("kindergarten");
      ctx.load("classpath:profile/*-config.xml");
      ctx.refresh();

      FoodProviderService foodProviderService =
         ctx.getBean("foodProviderService", FoodProviderService.class);
      List<Food> lunchSet = foodProviderService.provideLunchSet();

      for (Food food: lunchSet) {
         System.out.println("Food: " + food.getName());
      }
   }
}
```

В этом листинге мы используем метод `ctx.getEnvironment().setActiveProfiles()` для активизации определенного профиля. Обратите внимание, что для одного и того же контекста можно активизировать несколько профилей (т.е. методу `setActiveProfile()` можно передавать множество имен профилей, используя для этого поддержку varargs). В листинге 5.55 активизирован профиль `kindergarten` (детский сад). После этого метод `ctx.load()` загрузит оба файла `kindergarten-config.xml` и `highschool-config.xml`, т.к. ему передается групповой символ в виде префикса. В этом случае Spring создаст экземпляры только бинов из файла `kindergarten-config.xml`, основываясь на атрибуте `profile`. Запуск программы даст следующий вывод:

```
Food: Milk
Food: Biscuits
```

Это в точности то, что реализация поставщика для детского сада формирует для обеденного набора. А теперь изменим строку профиля в предыдущем листинге на `"highschool"`; вывод окажется таким:

```
Food: Coke
Food: Hamburger
Food: French Fries
```

Итак, с использованием профилей вы ознакомились. Еще один интересный момент состоит в том, что вместо программной активизации профиля его можно указать, передавая следующий аргумент JVM:

```
-Dspring.profiles.active="kindergarten"
```

На рис. 5.2 показано, как сконфигурировать это аргумент JVM в STS.

За счет передачи этого аргумента можно убрать строку кода, устанавливающую активный профиль, и запустить программу. Результат будет тем же самым, как показано выше. Можете изменить аргумент JVM на `"highschool"` и запустить программу снова.

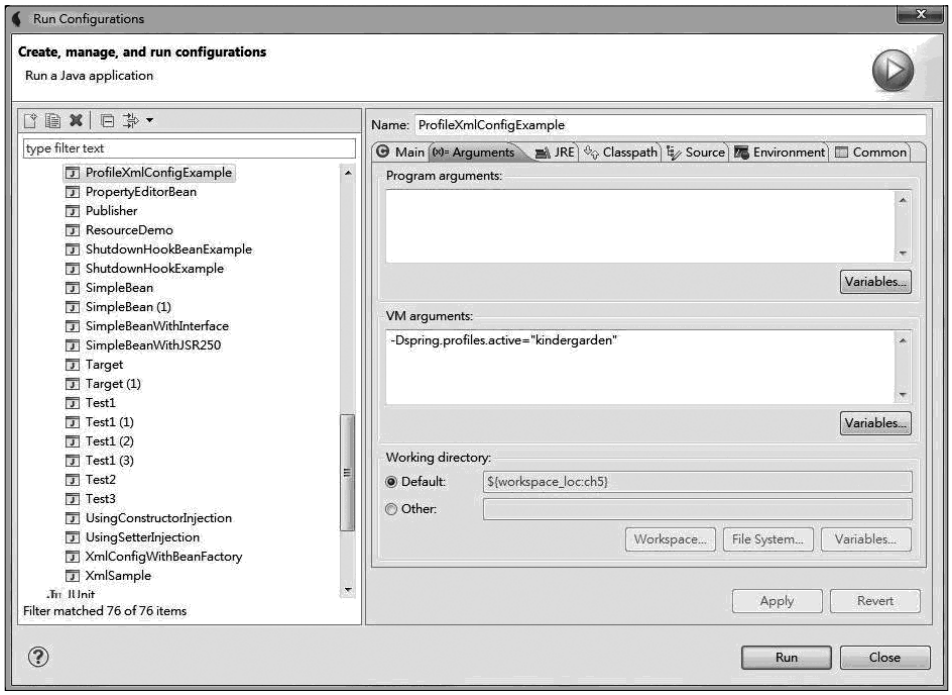


Рис. 5.2. Конфигурирование аргументов JVM в STS

В главе 21 мы обсудим, как использовать и активизировать профили в веб-приложениях.

Профили можно применять также при использовании Java-конфигурации вместо XML. Давайте посмотрим, каким образом это делается. Как и в версии с XML, мы разработаем два разных Java-файла конфигурации и определим профиль, используя аннотацию `@Profile`. В листингах 5.56 и 5.57 показан файл Java-конфигурации для профилей детского сада и средней школы.

Листинг 5.56. Java-конфигурация профиля для детского сада

```
package com.apress.prospring3.ch5.profile;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;

import com.apress.prospring3.ch5.profile.kindergarten.FoodProviderServiceImpl;

@Configuration
@Profile(value="kindergarten")
public class KindergartenConfig {

   @Bean
   public FoodProviderService foodProviderService() {
      return new FoodProviderServiceImpl();
   }
}
```

Листинг 5.57. Java-конфигурация профиля для средней школы

```
package com.apress.prospring3.ch5.profile;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;

import com.apress.prospring3.ch5.profile.highschool.FoodProviderServiceImpl;

@Configuration
@Profile(value="highschool")
public class HighschoolConfig {

   @Bean
   public FoodProviderService foodProviderService() {
   return new FoodProviderServiceImpl();
   }
}
```

Как видите, эти два класса просто имитируют два XML-файла, которые были представлены ранее, с аннотацией `@Profile`, указывающей профиль приложения, к которому тот или иной класс принадлежит. Теперь реализуем еще одну тестовую программу, чтобы увидеть Java-конфигурацию профиля в действии. Код этой тестовой программы показан в листинге 5.58.

Листинг 5.58. Пример Java-конфигурации профиля

```
package com.apress.prospring3.ch5.profile;

import java.util.List;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class ProfileJavaConfigExample {

   public static void main(String[] args) {

      AnnotationConfigApplicationContext ctx =
         new AnnotationConfigApplicationContext();
      ctx.getEnvironment().setActiveProfiles("kindergarten");
      ctx.register(KindergartenConfig.class, HighschoolConfig.class);
      ctx.refresh();

      FoodProviderService foodProviderService =
         ctx.getBean("foodProviderService", FoodProviderService.class);
      List<Food> lunchSet = foodProviderService.provideLunchSet();

      for (Food food: lunchSet) {
         System.out.println("Food: " + food.getName());
      }
   }
}
```

В листинге 5.58 мы используем реализацию `AnnotationConfigApplicationContext` и ее метод `register()` для регистрации определений бинов из классов `KindergartenConfig` и `HighschoolConfig`. Остальной код остается без изменений. Запустив программу, вы увидите вывод обеденного набора для детского сада.

Аргумент JVM по имени `spring.profiles.active` применяется точно так же, как в примере с XML. Опробуйте его самостоятельно. Соответствующий исходный код доступен для загрузки в составе примеров, рассмотренных в книге.

Соображения по поводу использования профилей

Средство профилей в Spring 3.1 предлагает разработчикам еще один способ управления конфигурацией запуска приложений, осуществляемого в инструментах сборки (например, поддержка профилей в Maven). Инструменты сборки на основе переданным им аргументов упаковывают нужные файлы конфигурации/свойств в Java-архив (в зависимости от типа приложения это может быть JAR или WAR) и затем развертывают его в целевой среде. Средство профилей Spring позволяет разработчикам приложений самостоятельно определять профили и активизировать их либо программно, либо с помощью аргумента JVM. Поддержка профилей в Spring позволяет иметь один архив приложения и развертывать его во всех различных средах, передавая подходящие профили в качестве аргумента во время начальной загрузки JVM. Например, могут существовать приложения с разными профилями, такими как `(dev, hibernate)`, `(prd, jdbc)` и т.д., при этом каждая комбинация представляет среду запуска (среда разработки или производственная среда) и применяемую библиотеку доступа к данным (Hibernate или JDBC). Это переносит управление профилями приложения на сторону программирования.

Тем не менее, описанному подходу также присущи и недостатки. Например, некоторые могут возразить, что помещение всей конфигурации для различных сред в конфигурационные файлы приложения или в Java-классы с последующим связыванием их вместе будет подвержено ошибкам в случае небрежной обработки (например, администратор может забыть об установке подходящего аргумента JVM в среде сервера приложений). Упаковка файлов сразу для всех профилей также приводит к увеличению размеров пакета. Здесь снова следует напомнить: позвольте требованиям к приложению и конфигурации управлять вашим выбором подхода, который наилучшим образом подходит для разрабатываемого проекта.

Абстракции `Environment` и `PropertySource`

В предыдущем разделе было показано, как использовать средство профилей Spring. Для установки активного профиля понадобится доступ к интерфейсу `Environment`. Этот интерфейс представляет собой новый уровень абстракции, появившийся в Spring 3.1; он предназначен для инкапсуляции среды выполняющегося Spring-приложения.

Кроме профилей, интерфейс `Environment` также инкапсулирует другие ключевые порции информации — свойства. Свойства служат для сохранения лежащей в основе приложения конфигурации среды, в которую входит местоположение папки приложения, параметры подключения к базе данных и т.д.

Абстракции `Environment` и `PropertySource` в Spring 3.1 содействуют разработчикам в доступе к разнообразной конфигурационной информации, связанной с платформой запуска. В рамках этих абстракций все свойства системы, переменные среды и свойства приложения обслуживаются интерфейсом `Environment`, который Spring наполняет во время начальной загрузки `ApplicationContext`. В листинге 5.59 показан простой пример.

Листинг 5.59. Пример работы с абстракцией среды Spring

```
package com.apress.prospring3.ch5.env;

import java.util.HashMap;
import java.util.Map;

import org.springframework.context.support.GenericXmlApplicationContext;
import org.springframework.core.env.ConfigurableEnvironment;
import org.springframework.core.env.MapPropertySource;
```

```
import org.springframework.core.env.MutablePropertySources;

public class EnvironmentSample {

   public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.refresh();
      ConfigurableEnvironment env = ctx.getEnvironment();
      MutablePropertySources propertySources = env.getPropertySources();
      Map appMap = new HashMap();
      appMap.put("application.home", "/etc/prospring3/home");
      propertySources.addLast(new MapPropertySource("PROSPRING3_MAP", appMap));
      System.out.println("user.home: " + System.getProperty("user.home"));
      System.out.println("JAVA_HOME: " + System.getenv("JAVA_HOME"));
      System.out.println("user.home: " + env.getProperty("user.home"));
      System.out.println("JAVA_HOME: " + env.getProperty("JAVA_HOME"));
      System.out.println("application.home: "
        + env.getProperty("application.home"));
   }
}
```

В листинге 5.59 после инициализации `ApplicationContext` мы получаем ссылку на интерфейс `ConfigurableEnvironment`. Через этот интерфейс получается обработчик `MutablePropertySources` (стандартная реализация интерфейса `PropertySources`, которая позволяет манипулировать содержащимися источниками свойств). Затем мы создаем карту, помещаем в нее свойства приложения и конструируем экземпляр класса `MapPropertySource` (подкласс `PropertySource`, который читает ключи и значения из экземпляра `Map`) для этой карты. Наконец, мы добавляем класс `MapPropertySource` к `MutablePropertySources` с помощью метода `addLast()`.

Запустив программу, мы получим следующий вывод:

```
user.home: C:\Users\Clarence
JAVA_HOME: C:\Program Files\Java\jdk1.6.0_27
user.home: C:\Users\Clarence
JAVA_HOME: C:\Program Files\Java\jdk1.6.0_27
application.home: /etc/prospring3/home
```

Показанные в первых двух строках системное свойство JVM по имени `user.home` и переменная среды `JAVA_HOME` извлекались так, как это делалось ранее (с использованием класса `System` в JVM). Тем не менее, как видно в последних трех строках, доступ ко всем свойствам системы, переменным среды и свойствам приложения может быть получен через интерфейс `Environment`. Это является хорошей иллюстрацией того, как абстракция `Environment` может помочь в управлении и доступе ко всему многообразию свойств внутри среды запуска приложения.

По умолчанию в рамках абстракции `PropertySource` платформа Spring обращается к свойствам в следующем порядке:

- свойства системы для выполняющейся машины JVM;
- переменная среды;
- свойства, определяемые приложением.

Для примера предположим, что вы определили то же самое свойство приложения, `user.home`, и добавили его к интерфейсу `Environment` через класс `MutablePropertySources`. Запустив программу, вы увидите, что по-прежнему извлекается свойство `user.home` из JVM, а не определенное вами свойство.

Тем не менее, Spring позволяет управлять порядком, в соответствии с которым интерфейс Environment должен извлекать свойства. Давайте немного изменим код в листинге 5.59 и посмотрим, как это работает. Модифицированная версия показана в листинге 5.60 (отличия выделены полужирным).

Листинг 5.60. Модифицированный пример работы с абстракцией среды Spring

```
// Некоторый код опущен.
public class EnvironmentSample {

   public static void main(String[] args) {

     GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
     ctx.refresh();

     ConfigurableEnvironment env = ctx.getEnvironment();
     MutablePropertySources propertySources = env.getPropertySources();
     Map appMap = new HashMap();
     appMap.put("user.home", "/etc/prospring3/home");
     propertySources.addFirst(new MapPropertySource("PROSPRING3_MAP", appMap));

     System.out.println("user.home: " + System.getProperty("user.home"));
     System.out.println("JAVA_HOME: " + System.getenv("JAVA_HOME"));

     System.out.println("user.home: " + env.getProperty("user.home"));
     System.out.println("JAVA_HOME: " + env.getProperty("JAVA_HOME"));
   }
}
```

В листинге 5.60 мы определили свойство приложения по имени user.home и добавили его в качестве первого при поиске с помощью метода addFirst() класса MutablePropertySources. Запуск программы в результате дает следующий вывод:

```
user.home: C:\Users\Clarence
JAVA_HOME: C:\Program Files\Java\jdk1.6.0_27
user.home: /etc/prospring3/home
JAVA_HOME: C:\Program Files\Java\jdk1.6.0_27
```

Первые две строки остались такими же, как ранее, поскольку для извлечения показанных в них свойств по-прежнему применялись методы getProperty() и getenv() класса System из JVM. Однако при использовании интерфейса Environment вы видите, что определенное нами свойство user.home имеет больший приоритет, т.к. оно определено как первое для поиска значений свойств.

В реальных приложениях необходимость в непосредственном взаимодействии с интерфейсом Environment возникает редко. Чаще всего будет использоваться заполнитель свойства в форме ${} (например, ${application.home}) и внедрение полученного значения в бины Spring. Давайте рассмотрим этот прием в действии.

Предположим, что имеется класс AppProperty для сохранения всех свойств приложения, загруженных из файла свойств. Код этого класса показан в листинге 5.61.

Листинг 5.61. Пример использования заполнителей свойств Spring

```
package com.apress.prospring3.ch5.env;

public class AppProperty {

   private String applicationHome;

   private String userHome;
```

```
    public String getApplicationHome() {
        return applicationHome;
    }

    public void setApplicationHome(String applicationHome) {
        this.applicationHome = applicationHome;
    }

    public String getUserHome() {
        return userHome;
    }

    public void setUserHome(String userHome) {
        this.userHome = userHome;
    }
}
```

В этом классе не ничего особо интересного. В листинге 5.62 представлено содержимое файла `application.properties`, в котором хранятся свойства выполняющегося приложения.

Листинг 5.62. Файл `application.properties`

```
application.home=/etc/prospring3/home
user.home=/clarence/home
```

Обратите внимание, что в этом файле свойств также объявлено свойство `user.home`. Давайте взглянем на XML-конфигурацию Spring, показанную в листинге 5.63 (`env/env.xml`).

Листинг 5.63. Конфигурация для примера использования заполнителей свойств Spring

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context-3.1.xsd">

    <context:property-placeholder
     location="classpath:env/application.properties"/>

    <bean id="appProperty" class="com.apress.prospring3.ch5.env.AppProperty">
        <property name="applicationHome" value="${application.home}"/>
        <property name="userHome" value="${user.home}"></property>
    </bean>

</beans>
```

Мы применяем дескриптор `<context:property-placeholder>` для загрузки свойств в Spring-интерфейс `Environment`, который помещен в оболочку интерфейса `ApplicationContext`. Кроме того, мы используем заполнители для внедрения значений в бин `AppProperty`. Код тестовой программы показан в листинге 5.64.

Листинг 5.64. Тестирование использования заполнителей свойств Spring

```
package com.apress.prospring3.ch5.env;

import org.springframework.context.support.GenericXmlApplicationContext;

public class PlaceHolderSample {
   public static void main(String[] args) {

     GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
     ctx.load("classpath:env/env.xml");
     ctx.refresh();
     AppProperty appProperty = ctx.getBean("appProperty", AppProperty.class);
     System.out.println("application.home: " + appProperty.getApplicationHome());
     System.out.println("user.home: " + appProperty.getUserHome());
   }
}
```

Запуск этой программы приводит к получению следующего вывода:

```
application.home: /etc/prospring3/home
user.home: C:\Users\Clarence
```

В выводе видно, что заполнитель `application.home` был корректно разрешен, в то время как свойство `user.home` по-прежнему извлекалось из свойств JVM, что вполне правильно, поскольку таково стандартное поведение абстракции `PropertySource`. Чтобы заставить Spring назначить более высокий приоритет значениям из файла `application.properties`, к дескриптору `<context:property-placeholder>` добавляется атрибут `local-override="true"`.

```
<context:property-placeholder local-override="true"
   location="classpath:env/application.properties"/>
```

Атрибут `local-override` сообщает Spring о необходимости переопределить существующие свойства свойствами, определенными в этом заполнителе. Запустив программу, вы увидите, что теперь извлекается свойство `user.home` из файла `application.properties`:

```
application.home: /etc/prospring3/home
user.home: /clarence/home
```

В предшествующих двух разделах было показано, что профили и абстракция источника свойств с помощью интерфейса `Environment` в Spring 3.1 обеспечивают мощный и централизованный способ управления конфигурациями среды приложения.

Конфигурация, использующая аннотации JSR-330

Как было указано в главе 1, в JEE 6 предлагается поддержка спецификации JSR-330 (внедрение зависимостей для Java), которая представляет собой коллекцию аннотаций для выражения конфигурации DI приложения в рамках контейнера JEE или другой совместимой инфраструктуры IoC. Платформа Spring также поддерживает и распознает эти аннотации, поэтому даже если приложение не запускается в контейнере JEE 6, все равно можно использовать аннотации JSR-330 внутри Spring. Применение аннотаций JSR-330 помогает упростить перенос приложения из Spring в контейнер JEE 6 или другой совместимый контейнер IoC (например, Google Guice).

Давайте снова вернемся к примеру поставщика и визуализатора сообщений и реализуем его с использованием аннотаций JSR-330.

Для поддержки аннотаций JSR-330 понадобится добавить к проекту зависимость, описанную в табл. 5.5.

Таблица 5.5. Зависимость для поддержки JSR-330

Идентификатор группы	Идентификатор артефакта	Версия	Описание
javax.inject	javax.inject	1	Стандартная библиотека JSR-330

Интерфейсные классы (MessageRenderer и MessageProvider) остались такими же, поэтому код их не показан. В листинге 5.65 представлена реализация класса ConfigurableMessageProvider.

Листинг 5.65. Класс ConfigurableMessageProvider (JSR-330)

```
package com.apress.prospring3.ch5.jsr330;

import javax.inject.Inject;
import javax.inject.Named;

@Named("messageProvider")
public class ConfigurableMessageProvider implements MessageProvider {
   private String message = "Default message";

   public ConfigurableMessageProvider() {
   }

   @Inject
   @Named("message")
   public ConfigurableMessageProvider(String message) {
      this.message = message;
   }

   public void setMessage(String message) {
      this.message = message;
   }

   public String getMessage() {
      return message;
   }
}
```

Как видите, все аннотации принадлежать пакету javax.inject, который представляет собой стандарт JSR-330. Аннотация @Named может применяться в классе в двух местах. Во-первых, мы используем ее для объявления внедряемого бина (аналогично аннотации @Component или @Service в Spring). В листинге аннотация @Named("messageProvider") указывает, что ConfigurableMessageProvider является внедряемым бином, и назначает ему имя messageProvider, т.е. делает то же самое, что и атрибут name дескриптора <bean> в Spring. Во-вторых, мы применяем внедрение через конструктор, помещая аннотацию @Inject перед конструктором, который принимает строковое значение. Затем мы используем аннотацию @Named для указания, что нужно внедрять значение, имеющее имя message. Взгляните на класс StandardOutMessageRenderer, код которого показан в листинге 5.66.

Листинг 5.66. Класс `StandardOutMessageRenderer` (JSR-330)

```
package com.apress.prospring3.ch5.jsr330;

import javax.inject.Inject;
import javax.inject.Named;
import javax.inject.Singleton;

@Named("messageRenderer")
@Singleton
public class StandardOutMessageRenderer implements MessageRenderer {

   @Inject
   @Named("messageProvider")
   private MessageProvider messageProvider = null;

   public void render() {
      if (messageProvider == null) {
         throw new RuntimeException(
            "You must set the property messageProvider of class:"
            + StandardOutMessageRenderer.class.getName());
      }
      System.out.println(messageProvider.getMessage());
   }
   public void setMessageProvider(MessageProvider provider) {
      this.messageProvider = provider;
   }
   public MessageProvider getMessageProvider() {
      return this.messageProvider;
   }
}
```

В листинге 5.66 мы используем атрибут `@Named` для того, чтобы определить внедряемый бин. Обратите внимание на аннотацию `@Singleton`. Следует отметить, что в стандарте JSR-330 бин по умолчанию является неодиночным, что похоже на область действия на уровне прототипа в Spring. Таким образом, если вы хотите, чтобы ваш бин был одиночным в среде JSR-330, то должны применять аннотацию `@Singleton`. Однако использование этой аннотации в Spring не дает никакого эффекта, потому что режимом создания экземпляров бинов в Spring уже является одиночный. Мы поместили эту аннотацию только для демонстрационных целей, чтобы подчеркнуть отличие между Spring и другими контейнерами, совместимыми с JSR-330.

В свойстве `messageProvider` в этот раз мы применяем аннотацию `@Inject` для внедрения через метод установки и указываем, что для такого внедрения должен использоваться бин с именем `messageProvider`.

В листинге 5.67 определена простая XML-конфигурация Spring для приложения (`jsr330/jsr330.xml`).

Листинг 5.67. XML-конфигурация Spring (JSR-330)

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:context="http://www.springframework.org/schema/context"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
     http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
     http://www.springframework.org/schema/context
     http://www.springframework.org/schema/context/spring-context-3.1.xsd">
```

```
    <context:component-scan base-package="com.apress.prospring3.ch5.jsr330"/>
    <bean id="message" class="java.lang.String">
        <constructor-arg value="You are running JSR330!"/>
    </bean>
</beans>
```

Для использования JSR-330 не нужны никакие специальные дескрипторы; просто сконфигурируйте свое приложение подобно обычному приложению Spring. Дескриптор `<context:component-scan>` заставляет Spring искать относящиеся к DI аннотации, и Spring распознает заданные аннотации JSR-330. Кроме того, здесь объявлен бин Spring по имени `message` для внедрения через конструктор в класс `ConfigurableMessageProvider`. В листинге 5.68 приведен код тестовой программы.

Листинг 5.68. Тестирование примера использования JSR-330

```
package com.apress.prospring3.ch5.jsr330;

import org.springframework.context.support.GenericXmlApplicationContext;

public class Jsr330Example {
    public static void main(String[] args) {
        GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
        ctx.load("classpath:jsr330/jsr330.xml");
        ctx.refresh();
        MessageRenderer renderer =
            ctx.getBean("messageRenderer", MessageRenderer.class);
        renderer.render();
    }
}
```

Код аналогичен коду из предыдущих примеров. Запуск этой программы приводит к генерации следующего вывода:

```
You are running JSR330!
```

За счет использования спецификации JSR-330 можно упростить миграцию в другие контейнеры IoC, совместимые с JSR-330 (например, серверы приложений, совместимые с JEE 6, отличные от таких контейнеров DI, как Google Guice, и т.д.). Тем не менее, аннотации, встроенные в Spring, обладают намного большими возможностями и гибкостью, чем аннотации JSR-330.

Ниже описаны некоторые основные отличия между ними.

- При использовании аннотации `@Autowired` из Spring можно задавать атрибут `required`, указывающий на то, что DI должно быть выполнено (для объявления этого требования можно также применять аннотацию `@Required` из Spring), тогда как для аннотации `@Inject` из JSR-330 подобный эквивалент отсутствует. Более того, Spring предоставляет аннотацию `@Qualifier`, которая обеспечивает более точное управление выполнением автосвязывания зависимостей на основе имени квалификатора.
- Спецификация JSR-330 поддерживает только одиночный и неодиночный режимы создания экземпляров бинов, в то время как Spring поддерживает большее число режимов, что очень полезно для веб-приложений.
- В Spring можно использовать аннотацию `@Lazy`, чтобы заставить Spring создавать экземпляры бина только при запрашивании приложением. В JSR-330 эквивалентная возможность отсутствует.

Аннотации Spring и JSR-330 можно также смешивать в одном приложении. Тем не менее, рекомендуется придерживаться какого-то одного стиля аннотаций, что упростит разработку и сопровождение приложения. Один из возможных подходов предусматривает использование аннотаций JSR-330, когда это только возможно, и аннотаций Spring при необходимости. Однако это даст лишь незначительную выгоду, поскольку придется выполнять большой объем работы при переходе на другой контейнер DI. В заключение отметим, что с учетом того факта, что аннотации Spring являются более мощными, рекомендуется отдавать предпочтение им, а не подходу с аннотациями JSR-330, если только не существует требования относительно того, что приложение должно быть независимым от контейнеров IoC.

Резюме

В этой главе вы ознакомились с широким спектром средств Spring, которые дополняют ключевые возможности IoC. Вы узнали, как подключаться к жизненному циклу бина и делать его осведомленным о среде Spring. Были представлены фабрики бинов в качестве решения для включения IoC в обширный набор классов. Было также показано, как использовать редакторы свойств для упрощения конфигурации приложения и устранения потребности в искусственных свойствах типа `String`. Кроме того, в главе было завершено подробное рассмотрение ряда дополнительных средств, предлагаемых `ApplicationContext`, среди которых интернационализация, публикация событий и доступ к ресурсам.

Мы также описали некоторые последние средства, появившиеся в версиях Spring 3.0 и 3.1, такие как использование Java-классов вместо XML-конфигурации, поддержка профилей, а также уровень абстракции среды и источников свойств. Наконец, мы обсудили применение стандартных аннотаций JSR-330 в Spring-приложениях.

Итак, мы рассмотрели основные концепции платформы Spring Framework и ее функционирования в качестве контейнера DI, а также другие службы, которые предоставляет ядро Spring Framework. В следующей главе и далее мы обсудим использование Spring в различных специальных областях, таких как АОП, доступ к данным, поддержка транзакций, поддержка веб-приложений и т.д.

ГЛАВА 6

Введение в аспектно-ориентированное программирование с использованием Spring

Помимо внедрения зависимостей (Dependency Injection — DI), платформа Spring Framework предлагает сообществу разработчиков еще одно ключевое средство — аспектно-ориентированное программирование (Aspect-Oriented Programming — AOP), или АОП. Хотя АОП трудно изучать, понимать и реализовывать, благодаря интенсивному использованию АОП в рамках платформы и упрощенной модели программирования, предоставляемой Spring, аспектно-ориентированное программирование стало технологией, которую разработчики применяют ежедневно, особенно при создании приложений, основанных на Spring. На АОП часто ссылаются как на инструмент для реализации сквозной функциональности. Когда вы сталкиваетесь с неизвестной предметной областью, вы используете АОП для разбиения на модули отдельных порций логики, которые называются *сквозной функциональностью*, и применяете эту функциональность ко многим частям приложения. Типичными примерами сквозной функциональности являются регистрация в журнале и безопасность, присутствующие во многих приложениях. Представьте себе приложение, которое в целях отладки регистрирует в журнале начало и завершение каждого метода. Возможно, вы вынесете код регистрации в журнале в специальный класс, однако впоследствии для выполнения регистрации все равно понадобится вызывать методы этого класса по два раза для каждого метода приложения. Используя АОП, вы можете просто указать, что методы регистрирующего класса должны вызываться перед и после вызова каждого метода в приложении.

Важно понимать, что АОП дополняет объектно-ориентированное программирование (ООП), а не соперничает с ним. ООП очень хорошо подходит для решения широкого разнообразия задач, с которыми сталкиваются программисты. Однако если взглянуть на пример с регистрацией в журнале еще раз, вполне ясно, что возможностей ООП не хватает, когда доходит до реализации сквозной функциональности в больших масштабах. Использование для разработки всего приложения одного лишь АОП практически нереально, учитывая, что функции АОП основаны на ООП. Аналогично, хотя определенно возможно строить полные приложения с применением ООП, более интеллектуальный подход предусматривает использование АОП для решения определенных задач, связанных со сквозной функциональностью.

Мы собираемся раскрыть АОП в этой и следующей главах. В частности, ниже перечислены темы, которые рассматриваются в данной главе.

- *Основы АОП*. Перед началом обсуждения реализации АОП в Spring мы опишем основы АОП как технологии. Большинство концепций в этом разделе не являются специфичными для Spring и могут быть найдены в любой реализации АОП. Если вы уже знакомы с другими реализациями АОП, можете спокойно пропустить этот раздел.
- *Типы АОП*. Различают два типа АОП: статическое и динамическое. При статическом АОП, таком как предоставляемое механизмами связывания во время компиляции AspectJ (`http://eclipse.org/aspectj/`), сквозная функциональность применяется к коду на этапе компиляции, и ее нельзя изменить без модификации кода и повторной компиляции. При динамическом АОП, вроде АОП в Spring, сквозная функциональность применяется динамически, во время выполнения. Это позволяет вносить изменения в распространение сквозной функциональности без перекомпиляции приложения. Эти типы АОП дополняют друг друга и в случае совместного использования формируют мощное сочетание, которое можно применять в приложениях.

На заметку! Статическое и динамическое АОП отличается от статической и динамической сквозной функциональности. Различия между статической и динамической сквозной функциональностью в основном теоретические и не имеют никакого отношения к АОП в Spring. За дополнительными сведениями по этой теме и по АОП в целом рекомендуем обратиться к книге *AspectJ in Action: Enterprise AOP with Spring Applications*, 2-е изд. (Manning, 2010 г.).

- *Архитектура АОП в Spring*. В этом разделе мы рассмотрим мельчайшие подробности реализации АОП в Spring, которая представляет собой только подмножество полного набора функциональных возможностей АОП, доступного в других реализациях вроде AspectJ. Здесь мы на высоком уровне выясним, какие средства присутствуют в Spring, как они реализованы и почему некоторые возможности исключены из реализации Spring.
- *Прокси АОП в Spring*. Прокси — это крупная часть рабочей среды АОП в Spring, и вы должны хорошо понять их, чтобы извлечь максимум пользы от этой реализации АОП. В данном разделе мы рассмотрим два вида прокси: динамический прокси JDK и прокси CGLIB. В частности, мы представим различные сценарии использования каждого вида прокси, оценим производительность двух типов прокси и дадим ряд простых рекомендаций, следуя которым, приложение может эффективно пользоваться АОП в Spring.
- *Использование АОП в Spring*. В этом разделе будут приведены некоторые практические примеры применения АОП. Мы начнем с простого примера “Hello World!”, чтобы упростить понимание кода АОП, и продолжим детальным описанием разнообразных функциональных средств АОП, доступных в Spring, сопровождая их подходящими примерами.

В этой главе мы рассматриваем АОП в Spring изолированно от остальной части платформы. В главе 7 будет дано более ориентированное на платформу представление АОП в Spring, включая конфигурирование АОП с использованием `ApplicationContext`.

Концепции АОП

Как и большинство технологий, АОП имеет собственный специфичный набор концепций и терминов. Перед тем, как переходить к применению АОП в приложениях, очень важно понять, что означают эти термины. Ниже перечислены ключевые концепции АОП.

- *Точки соединения.* Точка соединения (joinpoint) — это четко определенная точка во время выполнения приложения. Типовые примеры точек соединения включают обращение к методу, собственно Method Invocation, инициализацию класса и создание экземпляра объекта. Точки соединения являются ключевой концепцией АОП и определяют места в приложении, в которые можно вставлять дополнительную логику с использованием АОП.
- *Советы.* Фрагмент кода, который должен выполняться в отдельной точке соединения, представляет собой совет (advice). Существует несколько типов советов, среди которых *перед*, когда совет выполняется до точки соединения, и *после*, когда совет выполняется после точки соединения. В ООП совет принимает форму метода внутри класса.
- *Срезы.* Срез (pointcut) — это коллекция точек соединения, которая используется для определения ситуации, когда совет должен быть выполнен. Создавая срезы, вы получаете точный контроль над тем, как применять совет к компонентам приложения. Как упоминалось ранее, типичной точкой соединения является Method Invocation. А типичный срез представляет собой коллекцию всех точек соединения Method Invocation в отдельном классе. Часто между срезами можно установить сложные отношения, чтобы обеспечить дополнительные ограничения на то, когда будет выполнен совет. Композиция срезов более подробно рассматривается в следующей главе.
- *Аспекты.* Аспект (aspect) — это комбинация совета и срезов. Такая комбинация дает в результате определение логики, которая должна быть включена в приложение, и указание мест, где она должна выполняться.
- *Связывание.* Связывание (weaving) представляет собой процесс действительной вставки аспектов в определенную точку кода приложения. Для решений АОП времени компиляции это делается на этапе компиляции, обычно в виде дополнительного шага процесса сборки. Аналогично, для решений АОП времени выполнения связывание происходит динамически во время выполнения. В AspectJ поддерживается еще один механизм связывания под названием *связывание во время загрузки* (load-time weaving — LTW), который перехватывает лежащий в основе загрузчик классов JVM и обеспечивает связывание с байт-кодом, когда он загружается загрузчиком классов.
- *Цель.* Цель (target) — это объект, поток выполнения которого изменяется каким-то процессом АОП. На целевой объект часто ссылаются как на объект, снабженный советом.
- *Введение.* Введение (introduction) представляет собой процесс, посредством которого можно изменить структуру объекта за счет введения в него дополнительных методов или полей. Введение можно использовать для обеспечения реализации любым объектом определенного интерфейса без необходимости в том, чтобы класс этого объекта реализовывал такой интерфейс явно.

Не переживайте, если сразу не поняли эти концепции; они станут яснее, когда мы представим ряд примеров. Кроме того, многие из перечисленных выше концепций скрыты в реализации АОП, предлагаемой Spring, а некоторые даже и не важны. По мере продвижения по этой главе, мы обсудим все необходимые средства в контексте платформы Spring.

Типы АОП

Как упоминалось ранее, различают два типа аспектно-ориентированного программирования: статическое и динамическое. Разница между ними касается точки, в которой происходит процесс связывания, а также того, каким образом этот процесс осуществляется.

Статическое АОП

Многие из первых реализаций АОП были статическими. При статическом АОП процесс связывания формирует еще один шаг в процессе сборки приложения. В терминах Java процесс связывания при статической реализации АОП предполагает модификацию действительного байт-кода приложения, должным образом изменяя и расширяя код приложения. Очевидно, что это хороший способ выполнения процесса связывания, поскольку конечным результатом является просто байт-код Java, и нет необходимости предпринимать специальные трюки во время выполнения, чтобы определить, когда должен быть применен совет.

Недостаток этого механизма связан с тем, что любые изменения, вносимые в аспекты, даже если они касаются всего лишь добавления еще одной точки соединения, требуют перекомпиляции всего приложения. Связывание во время компиляции в AspectJ представляет собой прекрасный пример статической реализации АОП.

Динамическое АОП

Динамические реализации АОП, подобные АОП в Spring, отличаются от статических реализаций в том, что процесс связывания происходит динамически во время выполнения. Способ достижения этого зависит от реализации, но, как вы увидите, применяемый в Spring подход состоит в создании прокси для всех целевых объектов, позволяя совету вызываться требуемым образом. Небольшой недостаток динамического АОП связан с тем, что оно обычно не выполняется настолько хорошо, как статическое АОП, однако производительность неуклонно растет. Основное преимущество применения динамических реализаций АОП заключается в простоте изменения целого набора аспектов в приложении, причем без необходимости в перекомпиляции кода приложения.

Выбор типа АОП

Выбор между статическим и динамическим АОП в действительности является довольно трудным решением. Нет никаких причин для выбора исключительно одной реализации, т.к. обе обладают своими достоинствами. На самом деле, начиная с версии 2.0, платформа Spring обеспечивала тесную интеграцию с AspectJ, позволяя с легкостью использовать оба типа АОП. В общем случае статические реализации АОП несколько длиннее и обычно обладают более широким набором функциональных средств и большим количеством доступных точек соединения. На самом деле Spring поддерживает только подмножество средств, которыми располагает AspectJ. Обычно если производительность является особенно критичной или если нужно средство АОП, которое в Spring не реализовано, необходимо применять AspectJ.

В большинстве других ситуаций АОП в Spring является идеальным выбором для решения всех задач. Удостоверьтесь, что вам известен спектр функциональных возможностей АОП, которые доступны в Spring, такие как декларативное управление транзакциями. Повторная их реализация с помощью AspectJ — пустая трата сил и времени, особенно с учетом того, что в Spring доступны готовые проверенные реализации.

Очень важно позволить требованиям приложения управлять выбором реализации АОП и не ограничиваться единственной реализацией, если для нужд приложения больше подходит комбинация из нескольких реализаций. В общем, мы обнаружили, что реализация АОП в Spring отличается меньшей сложностью, чем AspectJ, поэтому она должна рассматриваться в качестве первого варианта. Если потом окажется, что АОП в Spring не делает то, что нужно, или во время настройки приложения выяснится, что его производительность недостаточно высока (например, результаты работы профилировщика Java для приложения покажут, что львиная доля времени в Spring тратится на генерацию динамического прокси для определенных аспектов), следует переходить на AspectJ.

АОП в Spring

Реализацию АОП в Spring удобно представлять как состоящую из двух логических частей. Первая часть — это ядро АОП, которое обеспечивает совершенно несвязанную, чисто программную функциональность АОП (это также называется Spring AOP API). Вторая часть реализации АОП — это набор служб платформы, которые упрощают использование АОП в приложениях. Сверх того, другие компоненты Spring, такие как диспетчер транзакций и вспомогательные классы EJB, поддерживают основанные на АОП службы, упрощающие разработку приложений. В этой главе мы сосредоточим внимание только на основах ядра АОП. Службы платформы и расширенная функциональность ядра рассматриваются в главе 7.

АОП в Spring на самом деле является подмножеством полного набора функциональных возможностей АОП, реализуя лишь небольшое число конструкций, доступных в таких реализациях, как AspectJ. Тем не менее, это не должно быть основанием считать АОП в Spring бесполезным. В действительности одной из самых примечательных характеристик АОП в Spring является простота его использования, что объясняется отсутствием посторонних средств, которые часто совершенно не нужны. Реализация только подмножества функциональных возможностей АОП — это специфическая проектная цель Spring, позволяющая сконцентрироваться на простом доступе к наиболее общим средствам АОП. Вдобавок проектировщики Spring обеспечили полноценную интеграцию этой платформы с AspectJ.

Альянс АОП

Альянс АОП (AOP Alliance; `http://aopalliance.sourceforge.net/`) является результатом совместных усилий представителей многих проектов АОП с открытым кодом, в том числе Рода Джонсона из Spring, по определению стандартного набора интерфейсов для реализаций АОП. Будучи очень консервативным и сопротивляясь соблазну чрезмерно ограничить АОП во время его постоянного роста, Альянс АОП в конечном счете определил интерфейсы только для подмножества средств АОП. Где это возможно, в Spring вместо собственных применяются интерфейсы, определенные Альянсом АОП. Это позволяет многократно использовать отдельный совет во множестве реализаций АОП, которые поддерживают интерфейсы от Альянса АОП.

Пример “Hello World!” в АОП

Перед тем, как погрузиться в детали реализации АОП в Spring, мы представим простой пример, чтобы обеспечить некоторый контекст для дальнейших обсуждений. В этом примере мы возьмем простой класс, выводящий сообщение “World”, и с применением АОП трансформируем экземпляр этого класса во время выполнения для вывода сообщения “Hello World!”. Базовый класс `MessageWriter` показан в листинге 6.1.

Листинг 6.1. Класс `MessageWriter`

```
package com.apress.prospring3.ch6;

public class MessageWriter {

   public void writeMessage() {
      System.out.print("World");
   }
}
```

В классе `MessageWriter` нет ничего особо интересного; он имеет всего лишь один метод, который записывает сообщение “World” в консольный вывод. Мы хотим добавить к этому классу совет, чтобы метод `writeMessage()` выводил вместо “World” сообщение “Hello World!”.

Для этого перед телом метода должен быть выполнен код для записи в консольный вывод строки “Hello ”, а после тела — код для записи строки “!”. В терминах АОП нам требуется совет, выполняющийся *вместо* точки соединения. В этом случае точкой соединения является вызов метода `writeMessage()`. В листинге 6.2 приведена реализация совета “вместо” — класса `MessageDecorator`.

Листинг 6.2. Реализация совета “вместо”

```
package com.apress.prospring3.ch6;

import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;

public class MessageDecorator implements MethodInterceptor {

   public Object invoke(MethodInvocation invocation) throws Throwable {
      System.out.print("Hello ");
      Object retVal = invocation.proceed();
      System.out.println("!");
      return retVal;
   }
}
```

Интерфейс `MethodInterceptor` — это стандартный интерфейс Альянса АОП для реализации совета “вместо” для точек соединения Method Invocation. Объект `MethodInvocation` представляет вызов метода, для которого предназначен совет, и с помощью этого объекта мы управляем тем, когда вызову метода фактически разрешено продолжаться. Поскольку это совет “вместо”, в сущности, мы способны выполнить некоторые действия перед вызовом метода и некоторые действия после его вызова, но до того, как произойдет возврат из метода.

В листинге 6.2 мы просто записываем в консольный вывод строку “Hello ”, вызываем метод с помощью `MethodInvocation.proceed()` и затем записываем в консольный вывод строку “!”.

Финальный шаг этого примера заключается в связывании совета MessageDecorator (а точнее — метода invoke()) с кодом. Для этого мы создаем экземпляр MessageWriter, т.е. цель, после чего создаем прокси этого экземпляра, инструктируя фабрику прокси выполнить связывание с советом MessageDecorator. Соответствующий код показан в листинге 6.3.

Листинг 6.3. Связывание с советом MessageDecorator

```
package com.apress.prospring3.ch6;

import org.springframework.aop.framework.ProxyFactory;

public class HelloWorldAOPExample {

   public static void main(String[] args) {
      MessageWriter target = new MessageWriter();

      // Создать прокси.
      ProxyFactory pf = new ProxyFactory();

      pf.addAdvice(new MessageDecorator());
      pf.setTarget(target);

      MessageWriter proxy = (MessageWriter) pf.getProxy();

      // Записать сообщения target.writeMessage();
      target.writeMessage();
      System.out.println("");
      proxy.writeMessage();
   }
}
```

Важным моментом, который следует отметить в коде, является использование класса ProxyFactory для создания прокси целевого объекта и одновременного его связывания с советом. Совет MessageDecorator передается в ProxyFactory с помощью вызова addAdvice(), а цель для связывания указывается посредством вызова setTarget(). После того, как цель установлена, а некоторый совет добавлен к ProxyFactory, с помощью вызова getProxy() мы генерируем прокси. Наконец, мы вызываем writeMessage() на исходном целевом объекте и на прокси-объекте. Перед запуском программы в проект понадобится добавить зависимость для CGLIB, как показано в табл. 6.1.

Таблица 6.1. Зависимость для CGLIB

Идентификатор группы	Идентификатор артефакта	Версия	Описание
cglib	cglib	2.2.2	Библиотека генерации кода, требуемая для АОП в Spring

После добавления этой зависимости программу из листинга 6.3 можно запустить. Результаты ее выполнения выглядят следующим образом:

```
World
Hello World!
```

Как видите, вызов метода writeMessage() на незатронутом целевом объекте приводит к стандартному обращению без выдачи на консоль дополнительной информации. Однако при вызове прокси выполняется код в MessageDecorator, создавая желаемый вывод сообщения “Hello World!”. В этом примере целевой класс не имеет никаких зави-

симостей от Spring или интерфейсов Альянса АОП; прелесть АОП в Spring и, следовательно, АОП в общем заключается в том, что совет можно предусмотреть для любого класса, даже если класс создавался без учета АОП. Единственное ограничение, по крайней мере, в АОП из Spring, связано с невозможностью добавления советов к финальным классам, поскольку они не могут быть переопределены и, таким образом, для них нельзя создавать прокси.

Архитектура АОП в Spring

Ключевая архитектура АОП в Spring основана на прокси. Когда вы хотите создать экземпляр класса, снабженный советом, вы должны использовать класс `ProxyFactory` для создания прокси этого экземпляра, первым делом предоставив `ProxyFactory` со всеми аспектами, которые необходимо связать с прокси. Применение `ProxyFactory` — это чисто программный подход к созданию прокси АОП. По большей части, использовать такой подход в своем приложении не обязательно; вместо этого можно положиться на механизмы декларативной конфигурации АОП, предоставляемые Spring (класс `ProxyFactoryBean`, пространство имен `aop` и аннотации в стиле `@AspectJ`), которые обеспечат декларативное создание прокси. Тем не менее, важно понимать, как работает создание прокси. В оставшейся части этой главы для создания прокси будет применяться программный подход. Использование декларативных конфигураций АОП обсуждается в следующей главе.

Во время выполнения Spring будет анализировать сквозную функциональность, определенную для бинов в `ApplicationContext`, и динамически генерировать прокси-бины (которые помещают внутрь себя лежащие в основе целевые бины). Вместо обращения к целевому бину напрямую, вызывающий объект внедряет прокси-бин, и все обращения к целевому бину получает прокси-бин. После этого прокси-бин анализирует текущие условия (т.е. точку соединения, срез, совет и т.д.) и соответствующим образом связывает подходящий совет. На рис. 6.1 показано высокоуровневое представление прокси АОП в Spring в действии.

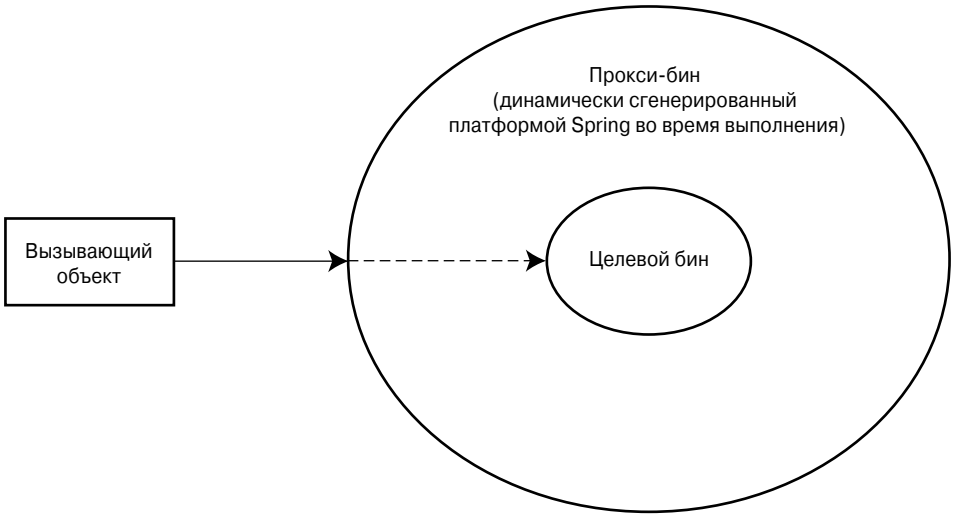


Рис. 6.1. Прокси АОП в Spring в действии

Внутренне Spring поддерживает две реализации прокси: динамический прокси JDK и прокси CGLIB. По умолчанию, когда целевой объект, оснащаемый советом, реализует некоторую разновидность интерфейса, Spring будет использовать динамический прокси JDK для создания экземпляров прокси целевого объекта. Однако если целевой объект, оснащаемый советом, не реализует интерфейс (например, из-за того, что является конкретным классом), для создания экземпляров прокси будет применяться CGLIB. Одна из основных причин заключается в том, что динамический прокси JDK поддерживает только создание прокси для интерфейсов. Прокси подробно обсуждаются в разделе "Что собой представляют прокси" далее в главе.

Точки соединения в Spring

Одним из наиболее заметных упрощений АОП в Spring следует считать поддержку только одного типа точек соединения: Method Invocation (вызов метода). На первый взгляд, это может показаться серьезным ограничением, если вы знакомы с другими реализациями АОП, такими как AspectJ, где поддерживается намного больше типов точек соединения, но в действительности это делает платформу Spring более доступной.

Точка соединения Method Invocation на сегодняшний день является наиболее часто используемой из всех доступных точек соединения, и с ее применением можно решить многие из задач, которые делают АОП полезным при повседневном программировании. Не забывайте, что если нужно снабдить советом некоторый код в точке соединения, отличной от Method Invocation, всегда можно использовать Spring и AspectJ вместе.

Аспекты в Spring

В контексте АОП в Spring аспект представляется экземпляром класса, который реализует интерфейс `Advisor`. Платформа Spring предлагает набор удобных реализаций `Advisor`, которые можно применять в своих приложениях, устраняя необходимость в самостоятельном создании множества разных реализаций `Advisor`. Существуют два подчиненных интерфейса `Advisor`: `IntroductionAdvisor` и `PointcutAdvisor`. Интерфейс `PointcutAdvisor` реализован всеми `Advisor`, которые используют срезы для управления применимостью совета к точкам соединения.

В Spring введение трактуется как специальная разновидность совета. Используя интерфейс `IntroductionAdvisor`, вы можете управлять классами, к которым применяется введение. Более подробно об этом пойдет речь в следующей главе.

Различные реализации `PointcutAdvisor` детально обсуждаются в разделе "Советы и срезы в Spring" далее в этой главе.

Класс `ProxyFactory`

Класс `ProxyFactory` управляет процессом связывания и создания прокси АОП в Spring. Перед действительным созданием прокси понадобится указать целевой объект. Как было показано ранее, это можно сделать с помощью метода `setTarget()`. Внутренне `ProxyFactory` делегирует выполнение задачи создания прокси экземпляру `DefaultAopProxyFactory`, который, в свою очередь, делегирует ее либо `Cglib2AopProxy`, либо `JdkDynamicAopProxy`, в зависимости от параметров приложения. Создание прокси более подробно рассматривается далее в этой главе.

Используя класс `ProxyFactory`, вы управляете тем, какие аспекты должны связываться с прокси. Ранее уже упоминалось, что с кодом, оснащенным советом, можно связать только один аспект — аспект, скомбинированный со срезом. Однако в ряде случаев совет должен применяться к вызовам всех методов в классе, а не только к избранным. Для этой цели в классе `ProxyFactory` предусмотрен метод `addAdvice()`, который использовался в листинге 6.3. Внутри метод `addAdvice()` помещает переданный ему со-

вет в экземпляр DefaultPointcutAdvisor, который является стандартной реализацией PointcutAdvisor, и конфигурирует его со срезом, который по умолчанию включает все методы. Если необходим дополнительный контроль над созданием Advisor или нужно добавить введение к прокси, создайте Advisor самостоятельно и используйте метод addAdvisor() класса ProxyFactory.

Один и тот же экземпляр ProxyFactory можно применять для создания множества разных прокси, каждый с отличающимся аспектом. Чтобы помочь в этом, в ProxyFactory предусмотрены методы removeAdvice() и removeAdvisor(), позволяющие удалять из ProxyFactory любой совет или Advisor, которые ранее были добавлены. Для проверки, имеет ли ProxyFactory конкретный присоединенный к нему совет, вызовите метод adviceIncluded(), передав ему проверяемый объект совета.

Создание совета в Spring

Платформа Spring поддерживает шесть разновидностей советов, которые описаны в табл. 6.2.

Таблица 6.2. Типы советов в Spring

Название совета	Интерфейс	Описание
Перед (before)	org.springframework.aop.MethodBeforeAdvice	Используя совет “перед”, можно осуществлять специальную обработку перед входом в точку соединения. Поскольку в Spring точкой соединения всегда является Method Invocation, это, в сущности, позволяет реализовать предварительную обработку до выполнения метода. Совет “перед” имеет полный доступ к цели Method Invocation, а также к аргументам, переданным методу, но не имеет никакого контроля над самим выполнением метода. Если совет “перед” генерирует исключение, дальнейшее выполнение цепочки перехватчика (а также целевой метод) прерывается, и исключение распространяется обратно по цепочке перехватчика
После возврата (after returning)	org.springframework.aop.AfterReturningAdvice	Совет “после возврата” выполняется после завершения выполнения Method Invocation в точке соединения и возврата значения. Совет “после возврата” имеет доступ к цели Method Invocation, к аргументам, переданным методу, а также к возвращаемому значению. Поскольку когда вызывается совет этого типа, метод уже выполнен, совет не имеет никакого контроля над Method Invocation. Если целевой метод генерирует исключение, совет “после возврата” выполняться не будет, а исключение распространится вверх по стеку вызовов, как обычно
После (after (finally))	org.springframework.aop.AfterAdvice	Совет “после возврата” выполняется только в случае нормального завершения метода, снабженного советом. Тем не менее, совет “после” будет выполняться вне зависимости от результата соответствующего метода. Совет этого типа выполняется, даже когда метод, снабженный советом, дает сбой или когда генерируется исключение

Продолжение табл. 6.2

Название совета	Интерфейс	Описание
Вместо (around)	`org.aopalliance.intercept.MethodInterceptor`	В Spring совет “вместо” моделируется с использованием стандарта Альянса АОП для перехватчика метода. Совету разрешено выполняться перед и после вызова метода, и есть возможность управления точкой, в которой вызов метода может быть продолжен. При необходимости можно вообще пропустить выполнение метода, предоставив собственную реализацию его логики
Перехват (throws)	`org.springframework.aop.ThrowsAdvice`	Совет “перехват” выполняется после возврата из Method Invocation, но только в случае, если во время вызова было сгенерировано исключение. Совет этого типа может перехватывать только специфичные исключения, и тогда возможен доступ к методу, сгенерировавшему исключение, к аргументам, переданным вызову, и к цели Method Invocation
Введение (introduction)	`org.springframework.aop.IntroductionInterceptor`	Платформа Spring моделирует введения как специальные типы перехватчиков. Используя перехватчик введения, можно указать реализацию методов, которые должны быть введены советом. Введения рассматриваются более подробно в следующей главе

Мы обнаружили, что все эти типы советов в сочетании с точкой соединения Method Invocation позволяют выполнить около 90% задач, решаемых с помощью АОП. Для решения остальных 10% задач, возникающих довольно редко, мы возвращались к AspectJ.

Интерфейсы для совета

Вспомните из предыдущего обсуждения класса `ProxyFactory`, что совет добавляется к прокси либо прямо, используя метод `addAdvice()`, либо косвенно с применением `Advisor`, с помощью метода `addAdvisor()`. Основная разница между `Advice` и `Advisor` заключается в том, что `Advisor` содержит в себе `Advice` и связанный `Pointcut`, что обеспечивает более точный контроль над тем, какие точки соединения `Advice` будет перехватывать. Для представления совета в Spring предусмотрена четко определенная иерархия интерфейсов советов. Эта иерархия основана на интерфейсах Альянса АОП и показана на рис. 6.2.

Такая иерархия обладает не только преимуществом своей объектно-ориентированной природы, но также и возможностью работы с типами советов обобщенным образом, например, используя единственный метод `addAdvice()` класса `ProxyFactory` для простого добавления новых типов советов без необходимости в модификации класса `ProxyFactory`.

Создание совета “перед”

Совет “перед” является одним из наиболее часто применяемых типов советов, доступных в Spring. Совет этого типа может изменять аргументы, переданные методу, и предотвращать выполнение метода путем генерации исключения. В следующей главе вы столкнетесь с частым использованием совета “перед” при рассмотрении применения АОП в приложении SpringBlog.

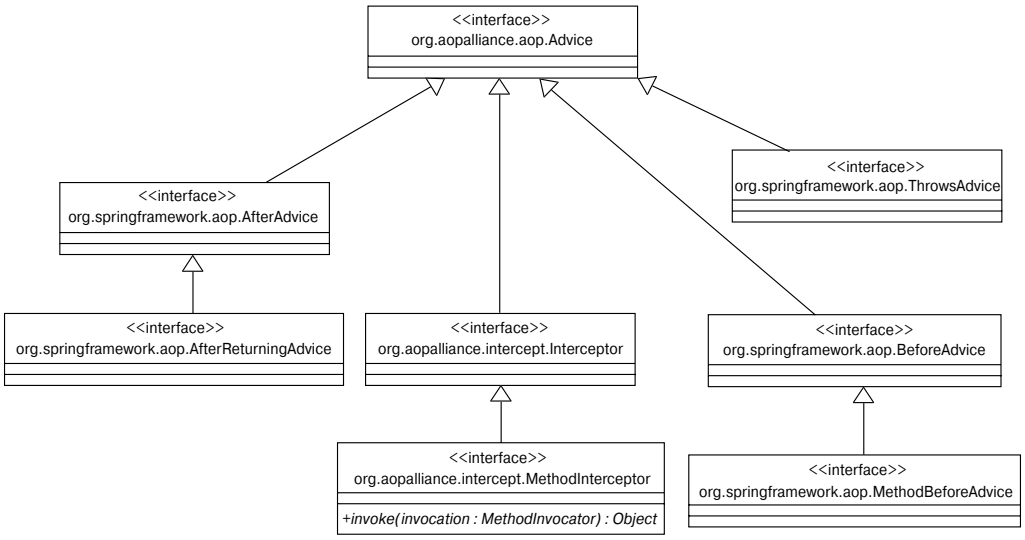


Рис. 6.2. Интерфейсы, предназначенные для типов советов Spring

В данном разделе мы покажем два примера использования совета “перед”: простой пример, который записывает в консольный вывод сообщение, содержащее имя метода, перед его выполнением, и простой совет безопасности, который можно применять для ограничения доступа к методам объекта.

В листинге 6.4 приведен код класса `SimpleBeforeAdvice`.

Листинг 6.4. Класс `SimpleBeforeAdvice`

```
package com.apress.prospring3.ch6;

import java.lang.reflect.Method;

import org.springframework.aop.MethodBeforeAdvice;
import org.springframework.aop.framework.ProxyFactory;

public class SimpleBeforeAdvice implements MethodBeforeAdvice {

   public static void main(String[] args) {
      MessageWriter target = new MessageWriter();

      // Создать прокси.
      ProxyFactory pf = new ProxyFactory();

      pf.addAdvice(new SimpleBeforeAdvice());
      pf.setTarget(target);

      MessageWriter proxy = (MessageWriter) pf.getProxy();

      // Записать сообщения.
      proxy.writeMessage();
   }

   public void before(Method method, Object[] args, Object target)
         throws Throwable {
      System.out.println("Before method: " + method.getName());
   }
}
```

В этом коде мы снабжаем советом созданный ранее экземпляр класса `MessageWriter` с помощью экземпляра класса `SimpleBeforeAdvice`. В интерфейсе `MethodBeforeAdvice`, который реализует класс `SimpleBeforeAdvice`, определен единственный метод `before()`, вызываемый инфраструктурой АОП перед запуском метода в точке соединения. Вспомните, что в данный момент мы используем срез по умолчанию, предоставляемый методом `addAdvice()`, который соответствует всем методам класса. Методу `before()` передаются три аргумента: метод для запуска, аргументы, которые будут переданы этому методу, и экземпляр `Object`, служащий целью вызова. Класс `SimpleBeforeAdvice` применяет аргумент `Method` метода `before()` для записи в консольный вывод сообщения, которое содержит имя запускаемого метода. В результате выполнения этого примера генерируется следующий вывод:

```
Before method: writeMessage
World
```

Как видите, вывод, полученный из вызова `writeMessage()`, присутствует, но перед ним находится вывод, сгенерированный `SimpleBeforeAdvice`.

Защита доступа к методам с использованием совета “перед”

Предыдущий пример был настолько тривиальным, что вряд ли продемонстрировал всю мощь АОП. В этом разделе мы собираемся построить совет “перед”, который проверяет полномочия пользователя перед тем, как разрешить вызову метода продолжаться. Если полномочия пользователя не достаточны, совет генерирует исключение, предотвращая выполнение метода. Пример, рассматриваемый в данном разделе, несколько упрощен. Он позволяет пользователям аутентифицироваться с произвольным паролем и также предоставляет одному жестко закодированному пользователю доступ к защищенным методам. Тем не менее, пример иллюстрирует простоту применения АОП для реализации сквозной функциональности, такой как средства безопасности.

На заметку! Это всего лишь пример демонстрации использования совета “перед”. Всеобъемлющая поддержка защиты запуска методов, относящихся к бинам Spring, предлагается проектом Spring Security, поэтому самостоятельно реализовать такие средства не понадобится.

В листинге 6.5 показан код класса `SecureBean`. Этот класс будет защищен с применением АОП.

Листинг 6.5. Класс `SecureBean`

```
package com.apress.prospring3.ch6.security;

public class SecureBean {

    public void writeSecureMessage() {
        System.out.println("Every time I learn something new, "
            + "it pushes some old stuff out of my brain");
    }
}
```

Класс `SecureBean` отображает одно из изречений Гомера Симпсона. Поскольку в этом примере требуется аутентификация пользователей, так или иначе, мы должны предусмотреть хранение ряда деталей. В листинге 6.6 приведен код класса `UserInfo`, используемого для хранения полномочий пользователя.

Листинг 6.6. Класс `UserInfo`

```
package com.apress.prospring3.ch6.security;

public class UserInfo {
   private String userName;

   private String password;

   public UserInfo(String userName, String password) {
      this.userName = userName;
      this.password = password;
   }

   public String getPassword() {
      return password;
   }

   public String getUserName() {
      return userName;
   }
}
```

В этом классе нет ничего особо примечательного; он просто хранит данные о пользователе, с которыми впоследствии можно делать что-нибудь полезное. В листинге 6.7 представлен код класса `SecurityManager`, отвечающего за аутентификацию пользователей и сохранение их полномочий с целью дальнейшего извлечения.

Листинг 6.7. Класс `SecurityManager`

```
package com.apress.prospring3.ch6.security;

public class SecurityManager {

   private static ThreadLocal<UserInfo> threadLocal = new
ThreadLocal<UserInfo>();

   public void login(String userName, String password) {
      // Предполагается, что все полномочия
      // достаточны для входа.
      threadLocal.set(new UserInfo(userName, password));
   }

   public void logout() {
      threadLocal.set(null);
   }

   public UserInfo getLoggedOnUser() {
      return threadLocal.get();
   }
}
```

Приложение использует класс `SecurityManager` для аутентификации пользователя и последующего извлечения деталей, связанных с аутентифицированным пользователем. Пользователь аутентифицируется с помощью метода `login()`. В реальном приложении метод `login()` мог бы проверять пользователей по базе данных или каталогу LDAP, но здесь мы предполагаем, что все пользователи аутентифицированы. Метод `login()` создает для пользователя объект `UserInfo` и сохраняет его в текущем потоке с применением `ThreadLocal`. Метод `logout()` устанавливает любые значения, которые могут быть сохранены в `ThreadLocal`, в `null`. Наконец, метод `getLoggedOnUser()` воз-

вращает объект UserInfo для текущего аутентифицированного пользователя. Если аутентифицированные пользователи отсутствуют, этот метод возвращает null.

Чтобы проверить, аутентифицирован ли пользователь, и если это так, разрешен ли ему доступ к методам SecureBean, мы должны создать совет, который выполняется перед методом и сравнивает объект UserInfo, возвращенный SecurityManager.getLoggedOnUser(), с набором полномочий для разрешенных пользователей. Код класса SecurityAdvice, представляющего этот совет, показан в листинге 6.8.

Листинг 6.8. Класс SecurityAdvice

```
package com.apress.prospring3.ch6.security;

import java.lang.reflect.Method;

import org.springframework.aop.MethodBeforeAdvice;

public class SecurityAdvice implements MethodBeforeAdvice {

   private SecurityManager securityManager;

   public SecurityAdvice() {
      this.securityManager = new SecurityManager();
   }

   public void before(Method method, Object[] args, Object target)
         throws Throwable {
      UserInfo user = securityManager.getLoggedOnUser();

      if (user == null) {
         System.out.println("No user authenticated");
         throw new SecurityException(
            "You must login before attempting to invoke the method: "
            + method.getName());
      } else if ("clarence".equals(user.getUserName())) {
         System.out.println("Logged in user is clarence - OKAY!");
      } else {
         System.out.println("Logged in user is " + user.getUserName()
            + " NOT GOOD :(");
         throw new SecurityException("User " + user.getUserName()
            + " is not allowed access to method " + method.getName());
      }
   }
}
```

Класс SecurityAdvice создает экземпляр SecurityManager в своем конструкторе и затем сохраняет этот экземпляр в поле. Следует отметить, что приложение и SecurityAdvice не нуждаются в разделении одного и того же экземпляра SecurityManager, т.к. все данные сохранены в текущем потоке с использованием ThreadLocal. В методе before() мы предпринимаем простую проверку, является ли clarence именем аутентифицированного пользователя. Если это так, мы предоставляем пользователю доступ, а в противном случае генерируем исключение. Также обратите внимание, что объект UserInfo проверяется на равенство null, означающее, что текущий пользователь не аутентифицирован.

В листинге 6.9 приведен пример приложения, которое использует класс SecurityAdvice для защиты класса SecureBean.

Листинг 6.9. Класс SecurityExample

```
package com.apress.prospring3.ch6.security;

import org.springframework.aop.framework.ProxyFactory;

public class SecurityExample {
   public static void main(String[] args) {
      // Получить диспетчер безопасности.
      SecurityManager mgr = new SecurityManager();

      // Получить бин.
      SecureBean bean = getSecureBean();

      // Попробовать войти как пользователь clarence.
      mgr.login("clarence", "pwd");
      bean.writeSecureMessage();
      mgr.logout();

      // Попробовать войти как пользователь janm.
      try {
         mgr.login("janm", "pwd");
         bean.writeSecureMessage();
      } catch(SecurityException ex) {
         System.out.println("Exception Caught: " + ex.getMessage());
      } finally {
         mgr.logout();
      }
      // Попробовать войти без полномочий.
      try {
         bean.writeSecureMessage();
      } catch(SecurityException ex) {
         System.out.println("Exception Caught: " + ex.getMessage());
      }
   }

   private static SecureBean getSecureBean() {
      // Создать цель.
      SecureBean target = new SecureBean();

      // Создать совет.
      SecurityAdvice advice = new SecurityAdvice();

      // Получить прокси.
      ProxyFactory factory = new ProxyFactory();
      factory.setTarget(target);
      factory.addAdvice(advice);
      SecureBean proxy = (SecureBean)factory.getProxy();
      return proxy;
   }
}
```

В методе `getSecureBean()` мы создаем прокси для класса `SecureBean`, который снабжается советом с применением экземпляра `SecurityAdvice`. Этот прокси возвращается вызывающему объекту. Когда вызывающий объект запускает любой метод на этом прокси, этот вызов сначала направляется экземпляру `SecurityAdvice` для проверки, связанной с безопасностью. В методе `main()` мы тестируем три различных сценария, вызывая метод `SecureBean.writeSecureMessage()` с двумя разными наборами пользовательских полномочий и затем вообще без полномочий. Поскольку `SecurityAdvice`

разрешает вызовам метода продолжаться, только если текущим аутентифицированным пользователем является clarence, ожидается, что единственным успешным сценарием в листинге 6.9 будет первый из трех. Запуск этого примера дает следующий вывод:

```
Logged in user is clarence - OKAY!
Every time I learn something new, it pushes some old stuff out my brain
Logged in user is janm NOT GOOD :(
Exception Caught: User janm is not allowed access to method writeSecureMessage
No user authenticated
Exception Caught: You must login before attempting to invoke the method:
writeSecureMessage
```

Как видите, продолжение было разрешено только первому вызову SecureBean.writeSecureMessage(). Остальные вызовы были прекращены по исключению SecurityException, сгенерированному SecurityAdvice.

Несмотря на простоту, этот пример подчеркивает полезность совета "перед". Безопасность — типичный пример совета "перед", но мы также сочли его удобным в сценарии, который требует модификации аргументов, поступающих в метод.

В главе 7 мы покажем, как использовать совет "перед" для создания фильтра нежелательного контента в приложении SpringBlog.

Создание совета "после возврата"

Как и следовало ожидать, совет "после возврата" выполняется после того, как из вызова метода в точке соединения происходит возврат. Учитывая, что метод уже выполнен, переданные ему аргументы модифицировать не получится. Хотя эти аргументы можно прочитать, нельзя изменить путь выполнения, равно как и предотвратить собственно запуск метода. Это вполне ожидаемые ограничения; однако, несколько неожиданным является тот факт, что в совете "после возврата" невозможно модифицировать возвращаемое значение. Вы ограничены только выполнением некоторой дополнительной обработки. Несмотря на то что совет "после возврата" не позволяет изменять возвращаемое значение из вызова метода, можно сгенерировать исключение, которое будет передано по стеку вверх взамен возвращаемого значения.

В этом разделе мы рассмотрим два примера применения в приложении совета "после возврата". В первом примере после вызова метода просто осуществляется запись сообщения в консольный вывод. Во втором примере показано, как можно использовать совет "после возврата" для добавления к методу дополнительной проверки ошибок. Предположим, что имеется класс KeyGenerator, который генерирует ключи для криптографических целей. Многие криптографические алгоритмы страдают из-за проблемы, заключающейся в том, что небольшое количество ключей в пространстве ключей считаются слабыми. Слабый ключ — это любой ключ, характеристики которого делают его существенно более простым для вывода исходного сообщения, не зная этого ключа. Для алгоритма DES всего существуют 256 возможных ключей. В этом пространстве ключей 4 ключа считаются слабыми, а еще 12 ключей — полуслабыми. Хотя шансы получить один из этих ключей во время их случайной генерации весьма невелики (1 из 252), проверка ключей настолько проста, что ее игнорирование выглядит как небрежность. Во втором примере этого раздела мы построим совет "после возврата", который проверяет ключи, сгенерированные KeyGenerator, на предмет слабости и инициирует исключение при обнаружении такого ключа.

На заметку! Для получения более полных сведений о слабых ключах и криптографии в целом рекомендуем обратиться к книге *Applied Cryptography* Брюса Шнайера (Wiley, 1995 г.).

В листинге 6.10 приведен код класса SimpleAfterReturningAdvice, который демонстрирует применение совета “после возврата”, записывая сообщение в консольный вывод после возврата из выполненного метода.

Листинг 6.10. Класс SimpleAfterReturningAdvice

```
package com.apress.prospring3.ch6;

import java.lang.reflect.Method;

import org.springframework.aop.AfterReturningAdvice;
import org.springframework.aop.framework.ProxyFactory;

public class SimpleAfterReturningAdvice implements AfterReturningAdvice {

   public static void main(String[] args) {
      MessageWriter target = new MessageWriter();

      // Создать прокси.
      ProxyFactory pf = new ProxyFactory();

      pf.addAdvice(new SimpleAfterReturningAdvice());
      pf.setTarget(target);

      MessageWriter proxy = (MessageWriter) pf.getProxy();

      // Записать сообщения.
      proxy.writeMessage();
   }

   public void afterReturning(Object returnValue, Method method,
         Object[] args, Object target) throws Throwable {
      System.out.println("");
      System.out.println("After method: " + method.getName());
   }
}
```

Этот пример не так уж сильно отличается от класса SimpleBeforeAdvice, который был показан ранее. Обратите внимание, что в интерфейсе AfterReturningAdvice объявлен единственный метод afterReturning(), которому передается возвращаемое значение из вызова метода, ссылка на вызванный метод, аргументы, переданные этому методу, и цель вызова. Запуск этого примера дает в результате следующий вывод:

```
World
After method: writeMessage
```

Вывод очень похож на полученный в примере с советом “перед”, за исключением того, что сообщение, записываемое советом, выводится после сообщения из метода writeMessage().

Совет “после возврата” удобно применять для выполнения дополнительной проверки ошибок, когда это возможно, для метода, возвращающего недопустимое значение. В описанном ранее сценарии генератор криптографических ключей может выдать ключ, который считается слабым для определенного алгоритма. В идеальном случае генератор ключей должен самостоятельно проверять получение слабых ключей, но поскольку шансы на их возникновение очень малы, многие генераторы такой проверки не содержат. Используя совет “после возврата”, мы можем обеспечить эту дополнительную проверку для метода генерации ключей. В листинге 6.11 показан очень простой генератор ключей.

Листинг 6.11. Класс KeyGenerator

```
package com.apress.prospring3.ch6.crypto;

import java.util.Random;

public class KeyGenerator {

   public static final long WEAK_KEY = 0xFFFFFFF0000000L;
   public static final long STRONG_KEY = 0xACDF03F590AE56L;

   private Random rand = new Random();

   public long getKey() {
      int x = rand.nextInt(3);

      if(x == 1) {
         return WEAK_KEY;
      } else {
         return STRONG_KEY;
      }
   }
}
```

Легко заметить, что этот генератор ключей совершенно не защищен, но мы не хотим ожидать годами появления слабого ключа, поэтому создали такой генератор, который в одном случае из трех должен сгенерировать слабый ключ. В листинге 6.12 приведен код класса `WeakKeyCheckAdvice`, который проверяет, не является ли результат метода `getKey()` слабым ключом.

Листинг 6.12. Проверка на слабые ключи

```
package com.apress.prospring3.ch6.crypto;

import java.lang.reflect.Method;

import org.springframework.aop.AfterReturningAdvice;

public class WeakKeyCheckAdvice implements AfterReturningAdvice {

   public void afterReturning(Object returnValue, Method method,
         Object[] args,Object target) throws Throwable {

      if ((target instanceof KeyGenerator)
            && ("getKey".equals(method.getName()))) {
         long key = ((Long) returnValue).longValue();

         if (key == KeyGenerator.WEAK_KEY) {
            throw new SecurityException(
               "Key Generator generated a weak key. Try again");
            // Генератор выдал слабый ключ.
         }
      }
   }
}
```

В методе `afterReturning()` мы сначала проверяем, является ли метод, который был выполнен в точке соединения, методом `getKey()`. Если это так, мы проверяем результирующее значение на предмет слабого ключа. Если обнаруживается, что результат метода `getKey()` — слабый ключ, мы генерируем исключение `SecurityException`, чтобы информировать об этом вызывающий код. В листинге 6.13 показано простое приложение, демонстрирующее использование этого совета.

Листинг 6.13. Тестирование класса `WeakKeyCheckAdvice`

```
package com.apress.prospring3.ch6.crypto;

import org.springframework.aop.framework.ProxyFactory;

public class AfterAdviceExample {

   public static void main(String[] args) {
      KeyGenerator keyGen = getKeyGenerator();

      for(int x = 0; x < 10; x++) {
         try {
            long key = keyGen.getKey();
            System.out.println("Key: " + key);
         } catch(SecurityException ex) {
            System.out.println("Weak Key Generated!");// Сгенерирован
                                                      // слабый ключ
         }
      }
   }

   private static KeyGenerator getKeyGenerator() {

      KeyGenerator target = new KeyGenerator();

      ProxyFactory factory = new ProxyFactory();
      Factory.setTarget(target);
      factory.addAdvice(new WeakKeyCheckAdvice());

      return (KeyGenerator)factory.getProxy();
   }
}
```

После создания прокси для цели `KeyGenerator` класс `AfterAdviceExample` пробует сгенерировать десять ключей. Если во время одиночной генерации произошло исключение `SecurityException`, в консольный вывод записывается сообщение, информирующее пользователя о том, что был получен слабый ключ; иначе отображается сгенерированный ключ. Однократный запуск этого приложения на нашей машине дал следующий вывод:

```
Key: 48658904092028502
Key: 48658904092028502
Weak Key Generated!
Key: 48658904092028502
Key: 48658904092028502
Key: 48658904092028502
Weak Key Generated!
Key: 48658904092028502
Key: 48658904092028502
Key: 48658904092028502
```

Вывод показывает, что класс `KeyGenerator`, как и ожидалось, временами генерирует слабые ключи, а `WeakKeyCheckAdvice` гарантирует, что при каждом обнаружении слабого ключа будет сгенерировано исключение `SecurityException`.

Создание совета “вместо”

Совет “вместо” функционирует похоже на комбинацию советов “перед” и “после”, но с одним большим отличием — имеется возможность модифицировать возвращаемое

значение. Кроме того, можно предотвратить действительное выполнение метода. Это значит, что посредством совета "вместо" можно, в сущности, заменить всю реализацию метода новым кодом. Совет "вместо" в Spring моделируется как перехватчик с использованием интерфейса `MethodInterceptor`. Существует множество применений совета данного типа, и вы обнаружите, что многие функциональные средства Spring, такие как поддержка удаленных прокси и управление транзакциями, реализованы с применением перехватчиков методов. Перехватчики методов также являются удобным механизмом для построения профилей выполнения приложений, и они формируют основу примеров в данном разделе.

Мы не планируем строить простой пример для перехвата метода, а сошлемся на первый пример в листинге 6.2, демонстрирующий использование базового перехватчика метода для записи сообщения перед или после вызова метода. В этом примере следует отметить, что метод `invoke()` класса `MethodInterceptor` не имеет такой же набор аргументов, как у методов `MethodBeforeAdvice` и `AfterReturningAdvice`, т.е. ему не передаются цель вызова, вызываемый метод и аргументы для вызываемого метода. Тем не менее, доступ к этим данным можно получить через объект `MethodInvocation`, передаваемый `invoke()`. Мы покажем все это в следующем примере.

В этом примере мы хотим получить базовую информацию о производительности методов класса, в частности, узнать, сколько времени выполняется тот или иной метод. Для достижения этой цели мы воспользуемся классом `StopWatch`, включенным в Spring, и нам, безусловно, нужен класс, реализующий `MethodInterceptor`, т.к. необходимо запустить `StopWatch` перед вызовом метода и остановить `StopWatch` сразу после вызова.

В листинге 6.14 показан код класса `WorkerBean`, который мы собираемся профилировать с применением класса `StopWatch` и совета "вместо".

Листинг 6.14. Класс `WorkerBean`

```
package com.apress.prospring3.ch6.profiling;

public class WorkerBean {

   public void doSomeWork(int noOfTimes) {
      for(int x = 0; x < noOfTimes; x++) {
         work();
      }
   }

   private void work() {
      System.out.print("");
   }
}
```

Это очень простой класс. Метод `doSomeWork()` принимает единственный аргумент `noOfTimes` и вызывает метод `work()` указанное в `noOfTimes` количество раз. Метод `work()` просто вызывает `System.out.print()`, передавая пустую строку. Это предотвратит проведение компилятором оптимизации путем отбрасывания метода `work()` и его вызова.

В листинге 6.15 приведен код класса `ProfilingInterceptor`, который использует класс `StopWatch` для получения времени вызова метода. Этот перехватчик применяется для профилирования класса `WorkerBean`, показанного в листинге 6.14.

Листинг 6.15. Класс `ProfilingInterceptor`

```
package com.apress.prospring3.ch6.profiling;

import java.lang.reflect.Method;

import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;
import org.springframework.util.StopWatch;

public class ProfilingInterceptor implements MethodInterceptor {
   public Object invoke(MethodInvocation invocation) throws Throwable {
      // Запустить StopWatch.
      StopWatch sw = new StopWatch();
      sw.start(invocation.getMethod().getName());

      Object returnValue = invocation.proceed();

      sw.stop();
      dumpInfo(invocation, sw.getTotalTimeMillis());
      return returnValue;
   }
   private void dumpInfo(MethodInvocation invocation, long ms) {
      Method m = invocation.getMethod();
      Object target = invocation.getThis();
      Object[] args = invocation.getArguments();

      System.out.println("Executed method: " + m.getName());
                        // Выполняемый метод
      System.out.println("On object of type: " +
            target.getClass().getName());        // класс объекта

      System.out.println("With arguments:");      // аргументы
      for (int x = 0; x < args.length; x++) {
         System.out.print("      > " + args[x]);
      }
      System.out.print("\n");

      System.out.println("Took: " + ms + " ms");   // время выполнения
   }
}
```

В методе `invoke()`, который является единственным методом интерфейса `MethodInterceptor`, мы создаем экземпляр `StopWatch` и сразу же запускаем его, после чего позволяем вызову метода продолжаться с помощью `MethodInvocation.proceed()`. После завершения вызова метода и получения возвращаемого значения мы останавливаем `StopWatch` и передаем общее количество миллисекунд вместе с объектом `MethodInvocation` методу `dumpInfo()`. Наконец, мы возвращаем `Object`, полученный из `MethodInvocation.proceed()`, чтобы вызывающий код мог извлечь корректное возвращаемое значение. В этом случае мы не хотим разрушать стек вызовов каким-либо образом; код просто действует как перехватчик вызова метода. При желании можно было бы изменить стек вызовов полностью, переадресовав вызов метода другому объекту или удаленной службе, либо просто заменить реализацию логики метода внутри перехватчика и возвратить другое значение.

Метод `dumpInfo()` просто записывает в консольный вывод некоторую информацию о вызове метода, а также время, затраченное на выполнение метода. В первых трех строках `dumpInfo()` вы видите, как можно использовать объект `MethodInvocation` для выяснения метода, который был вызван, исходной цели вызова и переданных аргументов.

В листинге 6.16 приведен код класса `ProfilingExample`, который сначала снабжает советом экземпляр `WorkerBean` с помощью `ProfilingInterceptor` и затем профилирует метод `doSomeWork()`.

Листинг 6.16. Класс `ProfilingExample`

```
package com.apress.prospring3.ch6.profiling;

import org.springframework.aop.framework.ProxyFactory;

public class ProfilingExample {

   public static void main(String[] args) {
      WorkerBean bean = getWorkerBean();
      bean.doSomeWork(10000000);
   }

   private static WorkerBean getWorkerBean() {
      WorkerBean target = new WorkerBean();

      ProxyFactory factory = new ProxyFactory();
      factory.setTarget(target);
      factory.addAdvice(new ProfilingInterceptor());

      return (WorkerBean)factory.getProxy();
   }
}
```

Этот код должен быть уже хорошо знаком. Запуск этого кода на нашей машине дал следующий вывод:

```
Executed method: doSomeWork
On object of type: com.apress.prospring3.ch6.profiling.WorkerBean
With arguments:
    > 10000000
Took: 874 ms
```

В этом выводе видно, какой метод был выполнен, какой класс служил целью, какие аргументы передавались, и сколько времени заняло выполнение.

Создание совета “перехват”

Совет “перехват” похож на совет “после возврата” тем, что выполняется после точки соединения, которая всегда имеет тип Method Invocation, но совет “перехват” инициируется, только если метод генерирует исключение. Совет “перехват” также подобен совету “после возврата” в том, что он имеет небольшой контроль над выполнением программы. При использовании совета “перехват” нельзя проигнорировать возникшее исключение и вернуть вместо этого какое-то значение. Единственной модификацией, которую можно внести в поток управления программы, является изменение типа сгенерированного исключения. На самом деле, это довольно мощная идея, которая существенно упрощает разработку приложений. Представьте себе ситуацию, когда имеется API-интерфейс, который генерирует целое множество плохо определенных исключений. Применяя совет “перехват”, вы снабжаете им все классы в этом API-интерфейсе и превращаете иерархию исключений в что-то более управляемое и описательное. Разумеется, совет “перехват” можно также использовать для обеспечения централизованной регистрации ошибок в приложении, тем самым снижая объем кода регистрации ошибок, который разбросан по всему приложению.

Как было показано на рис. 6.1, совет “перехват” реализован интерфейсом `ThrowsAdvice`. В отличие от интерфейсов, с которыми вы сталкивались до сих пор, в

ThrowsAdvice не определено ни одного метода; он просто является маркерным интерфейсом, используемым Spring. Причина подобного решения состоит в том, что в Spring разрешен типизированный совет “перехват”, позволяющий определять точные типы исключений, которые совет должен перехватывать. Это достигается в Spring обнаружением методов с заданными сигнатурами с помощью рефлексии. Платформа Spring ищет две отличающихся сигнатуры методов. Сказанное лучше всего продемонстрировать на простом примере. В листинге 6.17 представлен простой бин с двумя методами, которые генерируют исключения разных типов.

Листинг 6.17. Класс ErrorBean

```
package com.apress.prospring3.ch6;

public class ErrorBean {

   public void errorProneMethod() throws Exception {
      throw new Exception("Foo");
   }

   public void otherErrorProneMethod() throws IllegalArgumentException {
      throw new IllegalArgumentException("Bar");
   }
}
```

В листинге 6.18 показан код класса SimpleThrowsAdvice, отображающего обе сигнатуры методов, которые Spring ищет для совета “перехват”.

Листинг 6.18. Класс SimpleThrowsAdvice

```
package com.apress.prospring3.ch6;

import java.lang.reflect.Method;

import org.springframework.aop.ThrowsAdvice;
import org.springframework.aop.framework.ProxyFactory;

public class SimpleThrowsAdvice implements ThrowsAdvice {

   public static void main(String[] args) throws Exception {
      ErrorBean errorBean = new ErrorBean();

      ProxyFactory pf = new ProxyFactory();
      pf.setTarget(errorBean);
      pf.addAdvice(new SimpleThrowsAdvice());

      ErrorBean proxy = (ErrorBean) pf.getProxy();

      try {
         proxy.errorProneMethod();
      } catch (Exception ignored) {

      }

      try {
         proxy.otherErrorProneMethod();
      } catch (Exception ignored) {
      }
   }

   public void afterThrowing(Exception ex) throws Throwable {
      System.out.println("***");
      System.out.println("Generic Exception Capture");
                        // Перехват обобщенного исключения
```

```
    System.out.println("Caught: " + ex.getClass().getName()); // перехвачено
    System.out.println("***\n");
  }

  public void afterThrowing(Method method, Object[] args, Object target,
        IllegalArgumentException ex) throws Throwable {
    System.out.println("***");
    System.out.println("IllegalArgumentException Capture");
                    // Перехват исключения IllegalArgumentException
    System.out.println("Caught: " + ex.getClass().getName()); // перехвачено
    System.out.println("Method: " + method.getName());        // имя метода
    System.out.println("***\n");
  }
}
```

Мы уверены, что с пониманием кода в методе `main()` не должно возникать никаких проблем, поэтому сосредоточим внимание на двух методах `afterThrowing()`. Первое, что Spring ищет для совета "перехват" — это один или более открытых методов по имени `afterThrowing()`. Возвращаемые типы этих методов не важны, хотя мы считаем, что лучше указывать для них тип `void`, поскольку такие методы не могут возвращать значащие данные. Первый метод `afterThrowing()` в классе `SimpleThrowsAdvice` принимает единственный аргумент типа `Exception`. В этом аргументе можно указывать любой тип исключения, и этот метод идеально подходит, когда вас не интересует метод, сгенерировавший исключение, или переданные ему аргументы. Обратите внимание, что этот метод перехватывает `Exception` и любые подтипы `Exception`, если только для них не предусмотрены собственные методы `afterThrowing()`.

Во втором методе `afterThrowing()` принимает четыре аргумента для указания метода, сгенерировавшего исключение, аргументов, переданных этому методу, и цели вызова метода. Порядок следования аргументов в этом методе важен, к тому же должны быть указаны все четыре. Обратите внимание, что второй метод `afterThrowing()` перехватывает исключения типа `IllegalArgumentException` (или его подтипов). Запуск этого примера на выполнение дает следующий вывод:

```
***
Generic Exception Capture
Caught: java.lang.Exception
***

***
IllegalArgumentException Capture
Caught: java.lang.IllegalArgumentException
Method: otherErrorProneMethod
***
```

Как видите, когда генерируется простое исключение типа `Exception`, вызывается первый метод `afterThrowing()`, но при генерации `IllegalArgumentException` вызывается второй метод `afterThrowing()`. Платформа Spring вызывает одиночный метод `afterThrowing()` по одному разу для каждого исключения, к тому же, как было показано в примере из листинга 6.18, Spring использует метод, сигнатура которого в наибольшей степени соответствует типу исключения. В ситуации, когда совет "перехват" имеет дело с двумя методами `afterThrowing()`, причем оба объявлены с тем самым типом `Exception`, но один принимает единственный аргумент, а другой — четыре аргумента, платформа Spring вызывает метод `afterThrowing()` с четырьмя аргументами.

Как упоминалось ранее, совет "перехват" полезен во многих ситуациях; он позволяет реклассифицировать целые иерархии исключений, а также строить централизованную

регистрацию исключений для приложения. В частности, мы обнаружили, что совет "перехват" удобен при отладке готовых приложений, поскольку он позволяет добавлять дополнительный код регистрации в журнале, не изменяя код самого приложения.

Выбор типа совета

В общем случае выбор типа совета для использования управляется требованиями к приложению, но вы должны выбирать наиболее специфический тип совета. Другими словами, не используйте совет "вместо", если подходит совет "перед". В большинстве ситуаций совет "вместо" может обеспечить все то, что поддерживают остальные три типа советов, однако он может оказаться излишним для реализации задуманного. Применяя наиболее специфический тип совета, вы делаете назначение кода более ясным, а также сокращаете возможности возникновения ошибок. Предположим, что в совете должно подсчитываться количество вызовов метода. Если использовать для этого совет "перед", то понадобится лишь соответствующим образом закодировать счетчик, но в случае совета "вместо" вы должны не забыть о вызове самого метода и о возврате значения вызывающему коду. Такие мелочи способствуют проникновению ошибок в приложение. Поддерживая тип совета максимально целенаправленным, вы уменьшаете возможность возникновения ошибок.

Советы и срезы в Spring

Во всех примерах, показанных до сих пор, при конфигурировании совета для прокси использовался метод `ProxyFactory.addAdvice()`. Ранее упоминалось, что этот метод "за кулисами" делегирует свою работу методу `addAdvisor()`, создающему экземпляр `DefaultPointcutAdvisor` и конфигурирующему его со срезом, который указывает на все методы. В такой ситуации совет считается применяемым ко всем методам целевого объекта. В ряде случаев, как, например, при использовании АОП для регистрации в журнале, это может быть желательно, но в других случаях может понадобиться ограничить круг методов, к которым применяется данный совет.

Конечно, можно было бы просто выполнять проверку, подходит ли метод, внутри самого совета, но такому подходу присущи многие недостатки. Первый из них: жесткое кодирование принимаемых методов в совете снижает возможности его многократного использования. С помощью срезов можно конфигурировать перечень методов, к которым будет применяться совет, не помещая этот код внутрь совета; это очевидным образом увеличит степень повторного использования совета. Второй и третий недостатки жесткого кодирования списка методов в совете связаны с производительностью. Если снабжаемый советом метод проверяется внутри совета, это происходит каждый раз, когда вызывается любой метод целевого объекта. В результате производительность приложения снижается. При использовании срезов проверка выполняется по одному разу для каждого метода, а результаты кешируются для дальнейшего использования. Еще один недостаток отказа от применения срезов для ограничения списка снабжаемых советом методов, также связанный с производительностью, заключается в том, что при создании прокси Spring может оптимизировать методы, не снабженные советом, и это ускоряет их вызовы. Более подробно вопросы, касающиеся такой оптимизации, будут рассматриваться далее в этой главе.

Мы настоятельно рекомендуем не поддаваться искушению жестко кодировать проверки методов внутри совета, а взамен этого, где только возможно, пользоваться срезами для управления применимостью совета к методам целевого объекта. Тем не менее, в некоторых случаях жесткое кодирование проверок в совете является необходимым. Вспомните рассмотренный ранее пример совета "после возврата", который был предназначен для перехвата слабых ключей, генерируемых классом `KeyGenerator`. Этот вид

совета тесно связан с классом, для которого он назначен, и вполне разумно внутри него проверять, к правильному ли типу он применяется. Такое связывание совета и целевого объекта мы называем *родственностью цели*. В общем случае срезы должны использоваться, когда совет имеет небольшую родственность цели или вообще ее не имеет — т.е. может быть применен к любому типу или к широкому диапазону типов. Если совет имеет строгую родственность цели, попробуйте проверить корректность использования совета в нем самом; это поможет уменьшить ошибки, связанные с неправильным применением совета. Мы также рекомендуем избегать снабжения методов советами без нужды. Как вы увидите, это приведет к существенному снижению скорости вызовов, что может оказать серьезное влияние на общую производительность приложения.

Интерфейс `Pointcut`

Срезы в Spring создаются путем реализации интерфейса `Pointcut`, который показан в листинге 6.19.

Листинг 6.19. Интерфейс `Pointcut`

```
package org.springframework.aop;

public interface Pointcut {

   ClassFilter getClassFilter();

   MethodMatcher getMethodMatcher();
}
```

В этом коде видно, что в интерфейсе `Pointcut` определены два метода, `getClassFilter()` и `getMethodMatcher()`, которые возвращают экземпляры `ClassFilter` и `MethodMatcher` соответственно. При создании собственных срезов с нуля должны быть реализованы также оба интерфейса `ClassFilter` и `MethodMatcher`. К счастью, как будет показано в следующем разделе, обычно это не требуется, поскольку Spring предоставляет на выбор реализации `Pointcut`, покрывающие большинство, если не все, сценарии использования.

При выяснении, применим ли `Pointcut` к конкретному методу, Spring сначала проверяет, применим ли `Pointcut` к классу этого метода, используя экземпляр `ClassFilter`, который возвращается вызовом `Pointcut.getClassFilter()`. Интерфейс `ClassFilter` показан в листинге 6.20.

Листинг 6.20. Интерфейс `ClassFilter`

```
org.springframework.aop;

public interface ClassFilter {

   boolean matches(Class<?> clazz);
}
```

Как видите, в интерфейсе `ClassFilter` определен единственный метод `matches()`, принимающий экземпляр `Class`, который представляет класс, предназначенный для проверки. Метод `matches()` возвращает `true`, если срез применим к классу, и `false` — в противном случае.

Интерфейс `MethodMatcher` немного сложнее интерфейса `ClassFilter` и он показан в листинге 6.21.

Листинг 6.21. Интерфейс MethodMatcher

```
package org.springframework.aop;
public interface MethodMatcher {
    boolean matches(Method m, Class<?> targetClass);
    boolean isRuntime();
    boolean matches(Method m, Class<?> targetClass, Object[] args);
}
```

В Spring поддерживаются два разных типа MethodMatcher, статический и динамический, что определяется по возвращаемому значению метода isRuntime(). Перед использованием MethodMatcher платформа Spring вызывает isRuntime() для выяснения, является ли MethodMatcher статическим, на что указывает возвращенное значение false, или же динамическим, что отражается значением true.

Для статического среза Spring вызывает метод matches(Method, Class<T>) интерфейса MethodMatcher один раз для каждого метода целевого объекта, кешируя возвращаемое значение для последующих обращений к этому методу. Таким образом, проверка применимости метода производится только раз для каждого метода, и последующие обращения к методу не приводят к вызову matches().

Для динамических срезов Spring по-прежнему выполняет статическую проверку с помощью matches(Method, Class<T>) при вызове метода в первый раз, чтобы определить общую применимость этого метода. Однако в дополнение к этому и при условии, что статическая проверка вернула true, Spring осуществляет дальнейшую проверку для каждого вызова метода, используя matches(Method, Class<T>, Object[]). Таким образом, динамический MethodMatcher может выяснить, должен ли применяться срез, на основе конкретного вызова метода, а не только самого метода. Например, срез необходимо применять, только если аргумент представляет собой значение типа Integer, больше 100. В этом случае в методе matches(Method, Class<T>, Object[]) может быть предусмотрен код для дополнительной проверки аргумента при каждом вызове.

Очевидно, что статические срезы — т.е. срезы со статическим MethodMatcher — выполняются намного быстрее динамических, поскольку не требуют дополнительной проверки при каждом вызове. Тем не менее, динамические срезы обеспечивают больший уровень гибкости при решении о том, применять ли совет. В общем случае мы рекомендуем применять статические срезы везде, где только возможно. Однако в случаях, когда совет добавляет значительные накладные расходы, может иметь смысл избегать любых необязательных обращений к совету за счет использования динамического среза.

В целом вам редко придется создавать собственные реализации интерфейса Pointcut с нуля, т.к. Spring предоставляет абстрактные базовые классы и для статических, и для динамических срезов. Эти базовые классы, а также другие реализации Pointcut, рассматриваются в последующих разделах.

Доступные реализации интерфейса Pointcut

В версии Spring 3.1 предлагаются восемь реализаций интерфейса Pointcut: два абстрактных класса, служащие вспомогательными классами для создания статических и динамических срезов, и шесть конкретных классов, которые предназначены для решения следующих задач:

- компоновка множества срезов в одно целое;
- поддержка срезов потока управления;
- выполнение простых сопоставлений на основе имени;

- определение срезов с использованием регулярных выражений;
- определение срезов с использованием выражений AspectJ;
- определение срезов, которые ищут специфические аннотации на уровне классов или методов.

В табл. 6.3 приведена сводка по восьми реализациям интерфейса `Pointcut`.

Таблица 6.3. Сводка по реализациям интерфейса `Pointcut`

Класс реализации	Описание
`org.springframework.aop.support.annotation.AnnotationMatchingPointcut`	Срез, который ищет специфическую Java-аннотацию в классе или методе. Этот класс требует JDK 5 или более новой версии
`org.springframework.aop.aspectj.AspectJExpressionPointcut`	Срез, который использует средство связывания AspectJ для оценки выражения среза, представленного с помощью синтаксиса AspectJ
`org.springframework.aop.support.ComposablePointcut`	Класс `ComposablePointcut` применяется для компоновки двух и более срезов с помощью таких операций, как `union()` и `intersection()`. Более подробно этот класс рассматривается в следующей главе
`org.springframework.aop.support.ControlFlowPointcut`	Класс `ControlFlowPointcut` представляет срез, предназначенный для специального случая, который соответствует всем методам в потоке управления другого метода — т.е. любому методу, который вызван прямо или косвенно в результате выполнения другого метода. Класс `ControlFlowPointcut` более подробно описан в следующей главе
`org.springframework.aop.support.DynamicMethodMatcherPointcut`	Класс `DynamicMethodMatcherPointcut` служит базовым классом для построения динамических срезов
`org.springframework.aop.support.JdkRegexpMethodPointcut`	Класс `JdkRexepMethodPointcut` позволяет определять срезы с использованием поддержки регулярных выражений JDK 1.4. Этот класс требует JDK 1.4 или более новой версии
`org.springframework.aop.support.NameMatchMethodPointcut`	С помощью класса `NameMatchMethodPointcut` можно создать срез, который выполняет простое сопоставление со списком имен методов
`org.springframework.aop.support.StaticMethodMatcherPointcut`	Класс `StaticMethodMatcherPointcut` служит базовым классом для построения статических срезов

На рис. 6.3 показана UML-диаграмма для классов реализации `Pointcut`.

Шесть базовых реализаций подробно рассматриваются в последующих разделах. Классы `ComposablePointcut` и `ControlFlowPointcut` будут обсуждаться в следующей главе.

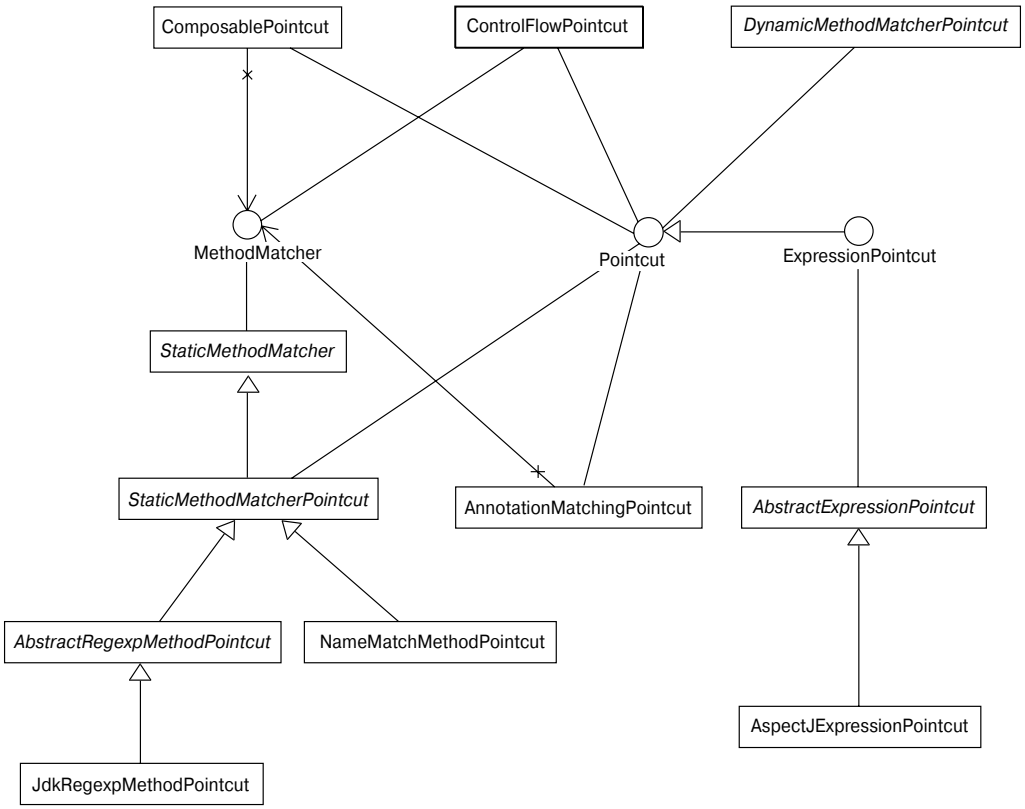


Рис. 6.3. Классы реализации `Pointcut`

Использование `DefaultPointcutAdvisor`

Перед тем, как можно будет использовать любую реализацию `Pointcut`, потребуется реализовать интерфейс `Advisor` или, точнее, интерфейс `PointcutAdvisor`. Как упоминалось ранее (в разделе “Аспекты в Spring” этой главы), интерфейс `Advisor` предназначен для представления аспекта в Spring — связки совета и срезов, управляющей тем, какие методы должны снабжаться советом, и как это должно делаться. Платформа Spring предоставляет ряд реализаций `PointcutAdvisor`, но сейчас мы займемся только одной из них — `DefaultPointcutAdvisor` (этот класс расширяет абстрактный класс `AbstractGenericPointcutAdvisor`, который реализует интерфейс `PointcutAdvisor`). Класс `DefaultPointcutAdvisor` — это просто `PointcutAdvisor` для связывания одного `Pointcut` с одним `Advice`.

Создание статического среза с использованием класса `StaticMethodMatcherPointcut`

В этом разделе мы создадим простой статический срез, расширив абстрактный класс `StaticMethodMatcherPointcut`. Поскольку класс `StaticMethodMatcherPointcut` расширяет класс `StaticMethodMatcher` (также абстрактный), который реализует интерфейс `MethodMatcher`, потребуется реализовать метод `matches(Method, Class<?>)`; остальная часть реализации `Pointcut` поддерживается автоматически. Хотя это единственный метод, который должен быть реализован (при расширении класса

StaticMethodMatcherPointcut), может также понадобиться переопределить метод getClassFilter(), как и делается в этом примере, чтобы гарантировать снабжение советом только методов подходящего типа.

В рассматриваемом примере имеются два класса, BeanOne и BeanTwo, в которых определены идентичные методы. В листинге 6.22 показан код класса BeanOne.

Листинг 6.22. Класс BeanOne

```
package com.apress.prospring3.ch6.staticpc;

public class BeanOne {

   public void foo() {
      System.out.println("foo");
   }

   public void bar() {
      System.out.println("bar");
   }
}
```

Класс BeanTwo имеет те же методы, что и класс BeanOne. В этом примере нам нужна возможность создать прокси для обоих классов, используя тот же самый класс DefaultPointcutAdvisor, но применить совет только к методу foo() класса BeanOne. Чтобы сделать это, мы создали класс SimpleStaticPointcut, код которого приведен в листинге 6.23.

Листинг 6.23. Класс SimpleStaticPointcut

```
package com.apress.prospring3.ch6.staticpc;

import java.lang.reflect.Method;

import org.springframework.aop.ClassFilter;
import org.springframework.aop.support.StaticMethodMatcherPointcut;

public class SimpleStaticPointcut extends StaticMethodMatcherPointcut {

   public boolean matches(Method method, Class<?> cls) {
      return ("foo".equals(method.getName()));
   }

   public ClassFilter getClassFilter() {
      return new ClassFilter() {
         public boolean matches(Class<?> cls) {
            return (cls == BeanOne.class);
         }
      };
   }
}
```

В коде видно, что мы реализовали метод matches(Method, Class<?>) согласно требованию абстрактного класса StaticMethodMatcher. Реализация просто возвращает true, если именем метода является foo, и false — в противном случае. Обратите внимание, что метод getClassFilter() также был переопределен; он возвращает экземпляр ClassFilter, метод matches() которого возвращает true только для класса BeanOne. Благодаря этому статическому срезу, обеспечивается соответствие только для методов класса BeanOne и, более того — только для метода foo() этого класса.

В листинге 6.24 показан код класса `SimpleAdvice`, который просто выводит сообщения до и после вызова метода.

Листинг 6.24. Класс `SimpleAdvice`

```
package com.apress.prospring3.ch6.staticpc;

import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;
public class SimpleAdvice implements MethodInterceptor {
   public Object invoke(MethodInvocation invocation) throws Throwable {
      System.out.println(">> Invoking " + invocation.getMethod().getName());
      Object retVal = invocation.proceed();
      System.out.println(">> Done");
      return retVal;
   }
}
```

В листинге 6.25 представлен код простого управляющего приложения для этого примера, которое создает экземпляр класса `DefaultPointcutAdvisor` с применением классов `SimpleAdvice` и `SimpleStaticPointcut`.

Листинг 6.25. Класс `StaticPointcutExample`

```
package com.apress.prospring3.ch6.staticpc;

import org.aopalliance.aop.Advice;
import org.springframework.aop.Advisor;
import org.springframework.aop.Pointcut;
import org.springframework.aop.framework.ProxyFactory;
import org.springframework.aop.support.DefaultPointcutAdvisor;
public class StaticPointcutExample {
   public static void main(String[] args) {
      BeanOne one = new BeanOne();
      BeanTwo two = new BeanTwo();
      BeanOne proxyOne;
      BeanTwo proxyTwo;
      // Создать срез, совет и аспект.
      Pointcut pc = new SimpleStaticPointcut();
      Advice advice = new SimpleAdvice();
      Advisor advisor = new DefaultPointcutAdvisor(pc, advice);
      // Создать прокси BeanOne.
      ProxyFactory pf = new ProxyFactory();
      pf.addAdvisor(advisor);
      pf.setTarget(one);
      proxyOne = (BeanOne)pf.getProxy();
      // Создать прокси BeanTwo.
      pf = new ProxyFactory();
      pf.addAdvisor(advisor);
      pf.setTarget(two);
      proxyTwo = (BeanTwo)pf.getProxy();
      proxyOne.foo();
      proxyTwo.foo();
      proxyOne.bar();
      proxyTwo.bar();
   }
}
```

Обратите внимание, что экземпляр DefaultPointcutAdvisor затем используется при создании двух прокси: для экземпляра BeanOne и для экземпляра BeanTwo. Наконец, на этих двух прокси вызываются методы foo() и bar().

Запуск этого примера дает в результате следующий вывод:

```
>> Invoking foo
foo
>> Done
foo
bar
bar
```

Как видите, единственным методом, к которому применялся SimpleAdvice, был foo() класса BeanOne, что и ожидалось. Ограничение методов, к которым применяется совет, реализуется довольно просто и, как будет показано при обсуждении различных вариантов прокси, является ключевым фактором получения наилучшей производительности приложения.

Создание динамического среза с использованием класса DynamicMethodMatcherPointcut

В этом разделе будет показано, что создание динамического среза не особенно отличается от создания статического среза. В рассматриваемом примере мы создаем динамический срез для класса, представленного в листинге 6.26.

Листинг 6.26. Класс SampleBean

```
package com.apress.prospring3.ch6.dynamicpc;

public class SampleBean {

   public void foo(int x) {
      System.out.println("Invoked foo() with: " + x);
   }

   public void bar() {
      System.out.println("Invoked bar()");
   }
}
```

В этом примере мы хотим снабдить советом только метод foo(), но в отличие от предыдущего примера, сделать это тогда, когда передаваемый ему аргумент int имеет значение, отличное от 100.

Как и в случае со статическими срезами, для создания динамических срезов Spring предлагает удобный базовый класс — DynamicMethodMatcherPointcut. Этот класс имеет единственный абстрактный метод matches(Method, Class<?>, Object[]) (благодаря реализуемому им интерфейсу MethodMatcher), который должен быть реализован, но, как вы увидите, также разумно реализовать метод matches(Method, Class<?>) для управления поведением статических проверок. Класс SimpleDynamicPointcut показан в листинге 6.27.

Листинг 6.27. Класс SimpleDynamicPointcut

```
package com.apress.prospring3.ch6.dynamicpc;

import java.lang.reflect.Method;

import org.springframework.aop.ClassFilter;
```

```
import org.springframework.aop.support.DynamicMethodMatcherPointcut;

public class SimpleDynamicPointcut extends DynamicMethodMatcherPointcut {

   public boolean matches(Method method, Class<?> cls) {
      // Выполнить статическую проверку.
      System.out.println("Static check for " + method.getName());
      return ("foo".equals(method.getName()));
   }

   public boolean matches(Method method, Class<?> cls, Object[] args) {
      // Выполнить динамическую проверку.
      System.out.println("Dynamic check for " + method.getName());

      int x = ((Integer) args[0]).intValue();

      return (x != 100);
   }

   public ClassFilter getClassFilter() {
      return new ClassFilter() {
         public boolean matches(Class<?> cls) {
            return (cls == SampleBean.class);
         }
      };
   }
}
```

В коде, приведенном в листинге 6.27, легко заметить, что мы переопределяем метод `getClassFilter()` подобно предыдущему примеру из листинга 6.23. Это устраняет необходимость в проверке класса в методах сопоставления имен методов — то, что особенно важно для динамической проверки. Хотя обязательной является только реализация динамической проверки, мы также реализуем и статическую проверку. Причина заключается в том, что, как нам известно, метод `bar()` никогда не будет снабжаться советом. Если указать на данный факт с использованием статической проверки, то Spring никогда не будет выполнять динамическую проверку для этого метода. Это объясняется тем, что когда метод статической проверки реализован, Spring сначала воспользуется им, и если результат проверки покажет несоответствие, то дальнейшая динамическая проверка не предпринимается. Вдобавок результат статической проверки кешируется для улучшения производительности. Но если пренебречь статической проверкой, Spring будет выполнять динамическую проверку при каждом вызове метода `bar()`. Рекомендуется применять следующий подход: выполнять проверку класса в методе `getClassFilter()`, проверку метода — в методе `matches(Method, Class<?>)` и проверку аргумента — в методе `matches(Method, Class<?>, Object[])`. Это сделает срез намного проще для понимания и сопровождения, а также улучшит показатели производительности.

В методе `matches(Method, Class<?>, Object[])` мы возвращаем `true`, если значение, переданное в аргументе `int` методу `foo()`, не равно 100, и `false` — в противном случае. Обратите внимание, что при выполнении динамической проверки известно, что мы имеем дело с методом `foo()`, поскольку никакие другие методы не прошли статической проверки.

В листинге 6.28 приведен пример этого среза в действии.

Листинг 6.28. Класс `DynamicPointcutExample`

```
package com.apress.prospring3.ch6.dynamicpc;

import org.springframework.aop.Advisor;
import org.springframework.aop.framework.ProxyFactory;
```

```
import org.springframework.aop.support.DefaultPointcutAdvisor;
import com.apress.prospring3.ch6.staticpc.SimpleAdvice;
public class DynamicPointcutExample {
   public static void main(String[] args) {
      SampleBean target = new SampleBean();
      // Создать экземпляр DefaultPointcutAdvisor.
      Advisor advisor = new DefaultPointcutAdvisor(
         new SimpleDynamicPointcut(), new SimpleAdvice());
      // Создать прокси.
      ProxyFactory pf = new ProxyFactory();
      pf.setTarget(target);
      pf.addAdvisor(advisor);
      SampleBean proxy = (SampleBean)pf.getProxy();
      proxy.foo(1);
      proxy.foo(10);
      proxy.foo(100);
      proxy.bar();
      proxy.bar();
      proxy.bar();
   }
}
```

Обратите внимание, что мы используем тот же самый класс совета, что и в примере со статическим срезом. Однако в текущем примере должны быть снабжены советом только первые два вызова `foo()`. Динамическая проверка предотвращает снабжением советом третий вызов `foo()`, а статическая проверка не позволяет снабдить советом метод `bar()`. Запуск этого примера дает следующий вывод:

```
Static check for foo
Static check for bar
Static check for toString
Static check for clone
Static check for foo
Dynamic check for foo
>> Invoking foo
Invoked foo() with: 1
>> Done
Dynamic check for foo
>> Invoking foo
Invoked foo() with: 10
>> Done
Dynamic check for foo
Invoked foo() with: 100
Static check for bar
Invoked bar()
Invoked bar()
Invoked bar()
```

Как и ожидалось, советом были снабжены только первые два обращения к методу `foo()`. Обратите внимание, что благодаря статической проверке метода `bar()`, ни один из вызовов `bar()` не подвергался динамической проверке. Интересно отметить также и то, что метод `foo()` участвовал в двух статических проверках: во время начальной фазы, когда проверялись все методы, и при его вызове в первый раз.

Пример показал, что динамические проверки обеспечивают более высокую гибкость, чем статические проверки, но из-за дополнительных накладных расходов во время выполнения они должны использоваться только в случае абсолютной необходимости.

Использование простого сопоставления имен

Часто при создании среза требуется выполнять сопоставление на основе лишь имени метода, игнорируя его сигнатуру и возвращаемый тип. В таком случае можно избежать необходимости в создании подкласса `StaticMethodMatcherPointcut` и применять вместо него `NameMatchMethodPointcut` (подкласс `StaticMethodMatcherPointcut`) для сопоставления со списком имен методов. Когда используется `NameMatchMethodPointcut`, никакого внимания сигнатуре метода не уделяется, поэтому если есть, например, методы `foo()` и `foo(int)`, то они оба соответствуют имени `foo`.

Продемонстрируем сказанное на примере. В листинге 6.29 показан простой класс с четырьмя методами.

Листинг 6.29. Класс `NameBean`

```
package com.apress.prospring3.ch6.namepc;

public class NameBean {
   public void foo() {
      System.out.println("foo");
   }
   public void foo(int x) {
      System.out.println("foo " + x);
   }
   public void bar() {
      System.out.println("bar");
   }
   public void yup() {
      System.out.println("yup");
   }
}
```

В этом примере мы хотим реализовать сопоставление с методами `foo()`, `foo(int)` и `bar()` с помощью `NameMatchMethodPointcut`; это сводится к сопоставлению с именами `foo` и `bar`. Код показан в листинге 6.30.

Листинг 6.30. Использование `NameMatchMethodPointcut`

```
package com.apress.prospring3.ch6.namepc;

import org.springframework.aop.Advisor;
import org.springframework.aop.framework.ProxyFactory;
import org.springframework.aop.support.DefaultPointcutAdvisor;
import org.springframework.aop.support.NameMatchMethodPointcut;

import com.apress.prospring3.ch6.staticpc.SimpleAdvice;

public class NamePointcutExample {

   public static void main(String[] args) {
      NameBean target = new NameBean();
      // Создать экземпляр DefaultPointcutAdvisor.
      NameMatchMethodPointcut pc = new NameMatchMethodPointcut();
      pc.addMethodName("foo");
      pc.addMethodName("bar");
      Advisor advisor = new DefaultPointcutAdvisor(pc, new SimpleAdvice());
```

```
        // Создать прокси.
        ProxyFactory pf = new ProxyFactory();
        pf.setTarget(target);
        pf.addAdvisor(advisor);
        NameBean proxy = (NameBean)pf.getProxy();
        proxy.foo();
        proxy.foo(999);
        proxy.bar();
        proxy.yup();
    }
}
```

Потребность в проектировании класса для среза отсутствует; достаточно просто создать экземпляр `NameMatchMethodPointcut`. Обратите внимание, что к срезу было добавлено два имени, `foo` и `bar`, с использованием метода `addMethodName()`. Запуск этого примера дает следующий вывод:

```
>> Invoking foo
foo
>> Done
>> Invoking foo
foo 999
>> Done
>> Invoking bar
bar
>> Done
yup
```

Как и ожидалось, благодаря срезу, методы `foo()`, `foo(int)` и `bar()` оказались снабжены советами, но метод `yup()` остался незатронутым.

Создание срезов с помощью регулярного выражения

В предыдущем разделе мы обсуждали, как выполнить простое сопоставление с предварительно определенным списком методов. Но как быть в ситуации, если все имена методов заранее не известны, но существует шаблон, которому эти имена следуют? Например, что если необходимо обеспечить соответствие всем методам, имена которых начинаются на `get`? В этом случае можно воспользоваться срезом с регулярным выражением `JdkRegexpMethodPointcut`, который позволяет выполнять сопоставление на основе регулярного выражения.

В листинге 6.31 показан код простого класса с тремя методами.

Листинг 6.31. Класс `RegexpBean`

```
package com.apress.prospring3.ch6.regexppc;

public class RegexpBean {
    public void foo1() {
        System.out.println("foo1");
    }
    public void foo2() {
        System.out.println("foo2");
    }
    public void bar() {
        System.out.println("bar");
    }
}
```

Используя срез с регулярным выражением, можно реализовать соответствие всех методов этого класса, имена которых начинаются с foo. Необходимый код приведен в листинге 6.32.

Листинг 6.32. Использование регулярных выражений для срезов

```
package com.apress.prospring3.ch6.regexppc;

import org.springframework.aop.Advisor;
import org.springframework.aop.framework.ProxyFactory;
import org.springframework.aop.support.DefaultPointcutAdvisor;
import org.springframework.aop.support.JdkRegexpMethodPointcut;

import com.apress.prospring3.ch6.staticpc.SimpleAdvice;

public class RegexpPointcutExample {

   public static void main(String[] args) {
      RegexpBean target = new RegexpBean();

      // Создать экземпляр DefaultPointcutAdvisor.
      JdkRegexpMethodPointcut pc = new JdkRegexpMethodPointcut();
      pc.setPattern(".*foo.*");
      Advisor advisor = new DefaultPointcutAdvisor(pc, new SimpleAdvice());

      // Создать прокси.
      ProxyFactory pf = new ProxyFactory();
      pf.setTarget(target);
      pf.addAdvisor(advisor);
      RegexpBean proxy = (RegexpBean)pf.getProxy();

      proxy.foo1();
      proxy.foo2();
      proxy.bar();
   }
}
```

Обратите внимание, что нет необходимости в создании класса для среза; вместо этого мы просто создаем экземпляр JdkRegexpMethodPointcut и указываем шаблон для сопоставления. Что интересно отметить, так это шаблон. При сопоставлении имен методов Spring использует полностью определенное имя метода, т.е. для foo1() производится сопоставление с com.apress.prospring3.ch6.regexppc.RegexpBean.foo1, чем и объясняется наличие .* в начале шаблона. Это мощная концепция, поскольку она позволяет производить сопоставление со всеми методами внутри заданного пакета без необходимости в знании точного перечня классов в пакете, а также имен имеющихся методов. Запуск этого примера дает следующий вывод:

```
>> Invoking foo1
foo1
>> Done
>> Invoking foo2
foo2
>> Done
bar
```

Как и можно было ожидать, только методы foo1() и foo2() были снабжены советом, т.к. метод bar() не соответствовал шаблону регулярного выражения.

Создание срезов с помощью выражения для срезов AspectJ

Кроме регулярных выражений JDK, срезы можно также объявлять с использованием языка выражений для срезов AspectJ. В следующей главе вы увидите, что при объявлении среза в XML-конфигурации с применением пространства имен `aop` платформа Spring по умолчанию использует язык срезов AspectJ. Кроме того, когда задействована поддержка АОП в стиле аннотаций `@AspectJ`, предлагаемая Spring, также необходимо применять язык срезов AspectJ. Таким образом, при объявлении срезов с использованием языка выражений лучше всего подойдут выражения срезов AspectJ. Платформа Spring предоставляет класс `AspectJExpressionPointcut` для определения срезов с помощью языка выражений AspectJ. Для работы с выражениями срезов AspectJ в Spring понадобится включить в путь классов проекта два библиотечных файла AspectJ — `aspectjrt.jar` и `aspectjweaver.jar`. В STS можно просто добавить к проекту зависимости, показанные в табл. 6.4.

Таблица 6.4. Зависимости Maven для AspectJ

Идентификатор группы	Идентификатор артефакта	Версия	Описание
`org.aspectj`	`aspectjrt`	1.6.11	Библиотека времени выполнения AspectJ
`org.aspectj`	`aspectjweaver`	1.6.11	Библиотека связывания AspectJ

Возьмем предыдущий пример применения регулярного выражения JDK еще раз и посмотрим, как воспользоваться выражением AspectJ для достижения тех же целей. В листинге 6.33 показан бин, который в точности такой же, как в листинге 6.31; отличается только имя.

Листинг 6.33. Класс `AspectjexpBean`

```
package com.apress.prospring3.ch6.aspectjexppc;

public class AspectjexpBean {
   public void foo1() {
      System.out.println("foo1");
   }
   public void foo2() {
      System.out.println("foo2");
   }
   public void bar() {
      System.out.println("bar");
   }
}
```

С помощью среза, основанного на выражении AspectJ, можно также обеспечить сопоставление со всеми методами этого класса, имена которых начинаются с `foo`. Соответствующий код показан в листинге 6.34.

Листинг 6.34. Использование выражений AspectJ для срезов

```
package com.apress.prospring3.ch6.aspectjexppc;

import org.springframework.aop.Advisor;
import org.springframework.aop.aspectj.AspectJExpressionPointcut;
import org.springframework.aop.framework.ProxyFactory;
import org.springframework.aop.support.DefaultPointcutAdvisor;
```

```
import com.apress.prospring3.ch6.staticpc.SimpleAdvice;

public class AspectjexpPointcutExample {

   public static void main(String[] args) {
      AspectjexpBean target = new AspectjexpBean();

      // Создать экземпляр DefaultPointcutAdvisor.
      AspectJExpressionPointcut pc = new AspectJExpressionPointcut();
      pc.setExpression("execution(* foo*(..))");
      Advisor advisor = new DefaultPointcutAdvisor(pc, new SimpleAdvice());

      // Создать прокси.
      ProxyFactory pf = new ProxyFactory();
      pf.setTarget(target);
      pf.addAdvisor(advisor);
      AspectjexpBean proxy = (AspectjexpBean) pf.getProxy();

      proxy.foo1();
      proxy.foo2();
      proxy.bar();
   }
}
```

Обратите внимание на использование метода `setExpression()` класса `AspectJExpressionPointcut` для установки критерия совпадения. Выражение `"execution(* foo*(..))"` означает, что совет должен применяться к выполнению любых методов, которые имеют имена, начинающиеся с `foo`, принимают любые аргументы и возвращают значение любого типа. Запуск программы даст те же результаты, что и предыдущий пример применения регулярных выражений JDK.

Создание срезов, соответствующих аннотациям

Если приложение основано на аннотациях, может возникнуть желание использовать для определения срезов собственные аннотации, т.е. применять логику совета ко всем или типам со специфическими аннотациями. В Spring доступен класс `AnnotationMatchingPointcut` для определения срезов с применением аннотаций. Мы снова воспользуемся предыдущим примером и посмотрим, как в нем применить аннотацию в качестве среза.

Определим интерфейс аннотации по имени `AdviceRequired`, который представляет аннотацию, используемую для объявления среза. В листинге 6.35 приведен код.

Листинг 6.35. Использование аннотации для срезов

```
package com.apress.prospring3.ch6.annotationpc;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.METHOD})
public @interface AdviceRequired {
}
```

В предыдущем листинге объявлен интерфейс как аннотация с указанием `@interface` в качестве типа, при этом аннотация `@Target` определяет, что может быть применена либо на уровне типа, либо на уровне метода.

Простой бин с аннотациями показан в листинге 6.36.

Листинг 6.36. Класс SampleAnnotationBean

```
package com.apress.prospring3.ch6.annotationpc;

public class SampleAnnotationBean {

   @AdviceRequired
    public void foo(int x) {
        System.out.println("Invoked foo() with: "  +x);
    }

    public void bar() {
        System.out.println("Invoked bar()");
    }
}
```

В представленном выше бине метод `foo()` аннотирован с помощью `@AdviceRequired` и к нему должен быть применен совет.

В листинге 6.37 приведен код тестовой программы.

Листинг 6.37. Тестирование среза, использующего аннотацию

```
package com.apress.prospring3.ch6.annotationpc;

import org.springframework.aop.Advisor;
import org.springframework.aop.framework.ProxyFactory;
import org.springframework.aop.support.DefaultPointcutAdvisor;
import org.springframework.aop.support.annotation.AnnotationMatchingPointcut;

import com.apress.prospring3.ch6.staticpc.SimpleAdvice;

public class AnnotationPointcutExample {

    public static void main(String[] args) {
        SampleAnnotationBean target = new SampleAnnotationBean();

        // Создать экземпляр DefaultPointcutAdvisor.
        AnnotationMatchingPointcut pc = AnnotationMatchingPointcut
            .forMethodAnnotation(AdviceRequired.class);
        Advisor advisor = new DefaultPointcutAdvisor(pc, new SimpleAdvice());

        // Создать прокси.
        ProxyFactory pf = new ProxyFactory();
        pf.setTarget(target);
        pf.addAdvisor(advisor);
        SampleAnnotationBean proxy = (SampleAnnotationBean) pf.getProxy();

        proxy.foo(100);
        proxy.bar();
    }
}
```

В этом коде запрашивается экземпляр класса `AnnotationMatchingPointcut` за счет вызова его статического метода `forMethodAnnotation()`, которому передается тип аннотации. Это указывает, что мы хотим применить совет ко всем методам, аннотированным заданной аннотацией. Также возможно указать аннотации на уровне типа, вызвав метод `forClassAnnotation()`.

Ниже показан вывод, полученный в результате запуска этой программы:

```
>> Invoking foo
Invoked foo() with: 100
>> Done
Invoked bar()
```

Как видите, поскольку метод `foo()` был аннотирован, только к нему и применялся совет.

Удобные реализации `Advisor`

Для многих реализаций `Pointcut` в Spring также предлагаются удобная реализация `Advisor`, которая действует в качестве среза. Например, вместо использования в предыдущем примере `NameMatchMethodPointcut` совместно с `DefaultPointcutAdvisor` можно было бы просто применить `NameMatchMethodPointcutAdvisor`, как показано в листинге 6.38.

Листинг 6.38. Использование `NameMatchMethodPointcutAdvisor`

```
package com.apress.prospring3.ch6.namepc;

import org.springframework.aop.framework.ProxyFactory;
import org.springframework.aop.support.NameMatchMethodPointcutAdvisor;

import com.apress.prospring3.ch6.staticpc.SimpleAdvice;

public class NamePointcutUsingAdvisor {

   public static void main(String[] args) {
      NameBean target = new NameBean();

      // Создать экземпляр DefaultPointcutAdvisor.
      NameMatchMethodPointcutAdvisor advisor = new
         NameMatchMethodPointcutAdvisor(new SimpleAdvice());
      advisor.addMethodName("foo");
      advisor.addMethodName("bar");

      // Создать прокси.
      ProxyFactory pf = new ProxyFactory();
      pf.setTarget(target);
      pf.addAdvisor(advisor);
      NameBean proxy = (NameBean) pf.getProxy();

      proxy.foo();
      proxy.foo(999);
      proxy.bar();
      proxy.yup();
   }
}
```

В листинге 6.38 вместо создания экземпляра `NameMatchMethodPointcut` мы конфигурируем детали среза с помощью экземпляра `NameMatchMethodPointcutAdvisor`. В этом случае `NameMatchMethodPointcutAdvisor` действует и как `Advisor`, и как `Pointcut`.

Узнать подробные сведения о различных удобных реализациях `Advisor` можно в документации JavaDoc по пакету `org.springframework.aop.support`. Между этими двумя подходами нет заметной разницы в производительности, и помимо чуть меньшего объема кода во втором подходе, отличия при кодировании незначительны. Мы предпочитаем придерживаться первого подхода, поскольку считаем, что он дает более

ясный код. В конце концов, выбор того или иного стиля является только персональным предпочтением.

Что собой представляют прокси

До сих пор мы только вскользь упоминали о прокси, которые генерируются `ProxyFactory`. Мы указывали, что в Spring доступны два типа прокси: прокси JDK, создаваемые с использованием класса `Proxy` из JDK, и прокси на основе CGLIB, создаваемые с помощью класса `Enhancer` из библиотеки CGLIB. Могут возникнуть вопросы, в чем заключаются отличия между этими двумя типами прокси, и для чего они применяются в Spring. В этом разделе мы детально рассмотрим отличия между прокси.

Основная цель прокси — перехватить вызов метода и, где необходимо, выполнить цепочки совета, который применяется к отдельному методу. Управление и вызов совета в основном не зависит от прокси, и этим занимается инфраструктура АОП в Spring. Однако прокси отвечает за перехват вызовов всех методов и передачу их при необходимости инфраструктуре АОП для применения совета.

Вдобавок к этой ключевой функциональности, прокси должен также поддерживать набор дополнительных средств. Прокси можно сконфигурировать так, чтобы он был доступен через класс `AopContext` (который является абстрактным); это позволит извлечь прокси и вызвать оснащенные советом методы из целевого объекта. Прокси отвечает за обеспечение того, что если эта опция включена с помощью `ProxyFactory.setExposeProxy()`, то класс прокси становится соответствующим образом доступным. В дополнение к этому все классы прокси по умолчанию реализуют интерфейс `Advised`, который, помимо прочего, позволяет изменять цепочку совета после того, как прокси был создан. Прокси должен также гарантировать, чтобы любой метод, который возвращает `this` — т.е. возвращает цель с прокси, — фактически возвращал прокси, но не целевой объект.

Как видите, типичный прокси должен выполнять немало работы, и вся нужная логика реализована в прокси типов JDK и CGLIB.

Использование динамических прокси JDK

Прокси JDK представляют собой наиболее базовый тип прокси, доступный в Spring. В отличие от прокси CGLIB, прокси JDK генерируются только для интерфейсов, но не классов. Таким образом, любой объект, для которого необходим прокси, должен реализовывать хотя бы один интерфейс. В общем случае использование интерфейсов для классов является удачным проектным решением, однако это не всегда возможно, особенно если приходится работать с унаследованным кодом или кодом от третьих сторон. В таком случае должен применяться прокси CGLIB.

При использовании прокси JDK все вызовы методов перехватываются JVM и направляются методу `invoke()` прокси. Затем `invoke()` выясняет, снабжен ли вызываемый метод советом (согласно правилам, определяемым срезом), и если это так, вызывает цепочку совета и сам метод с помощью рефлексии. Вдобавок метод `invoke()` выполняет всю логику, которая обсуждалась в предыдущем разделе.

Прокси JDK не разделяет методы на снабженные и не снабженные советом вплоть до вызова метода `invoke()`. Это означает, что для методов, не снабженных советом, метод `invoke()` на прокси по-прежнему вызывается, все проверки выполняются, и методы запускаются с применением рефлексии. Очевидно, что при каждом вызове метода возникают накладные расходы времени выполнения, хотя часто прокси не реализует никакой дополнительной обработки, кроме вызова метода, не снабженного советом, через рефлексию.

Чтобы сообщить ProxyFactory о необходимости использования прокси JDK, необходимо указать список интерфейсов для прокси с помощью метода setInterfaces() (в классе AdvisedSupport, который класс ProxyFactory косвенно расширяет).

Использование прокси CGLIB

В случае прокси JDK все решения относительно обработки конкретного вызова метода принимаются во время выполнения при каждом таком вызове. Библиотека CGLIB динамически генерирует байт-код нового класса для каждого прокси, по возможности повторно используя ранее сгенерированные классы.

Когда прокси CGLIB создается в первый раз, библиотека CGLIB запрашивает Spring о том, как желательно поддерживать каждый метод. Это означает, что многие решения, принимаемые при каждом обращении к invoke() в прокси JDK, в случае прокси CGLIB производятся только один раз. Поскольку CGLIB генерирует действительный байт-код, появляется намного больше гибкости в том, как поддерживать методы. Например, прокси CGLIB генерирует соответствующий байт-код для вызова любых методов, не снабженных советом, напрямую, сокращая накладные расходы, привносимые прокси. Вдобавок прокси CGLIB определяет, может ли метод возвращать this, и если нет, то позволяет вызову метода выполняться напрямую, снова сокращая накладные расходы времени выполнения.

Прокси CGLIB также поддерживает цепочки фиксированных советов отличным от прокси JDK образом. Цепочка фиксированного совета — это такая цепочка, которая не будет изменяться после генерации прокси. По умолчанию имеется возможность изменить средства снабжения советом и сам совет даже после создания прокси, хотя такое требуется редко. Прокси CGLIB обрабатывает цепочки фиксированных советов специальным образом, сокращая накладные расходы времени выполнения, которые связаны с цепочкой совета.

Сравнение производительности прокси

До сих пор мы занимались обсуждением в свободной форме отличий в реализации разных типов прокси. В этом разделе мы собираемся прогнать простой тест для сравнения производительности прокси CGLIB и прокси JDK.

Давайте создадим интерфейс ISimpleBean и реализующий его класс SimpleBean, который будет использоваться в качестве целевого объекта при создании прокси. Код интерфейса ISimpleBean и класса SimpleBean показан в листингах 6.39 и 6.40 соответственно.

Листинг 6.39. Интерфейс ISimpleBean

```
package com.apress.prospring3.ch6.proxies;

public interface ISimpleBean {
   public void advised();
   public void unadvised();
}
```

Листинг 6.40. Класс SimpleBean

```
package com.apress.prospring3.ch6.proxies;

public class SimpleBean implements ISimpleBean {

   private long dummy = 0;
```

```
    public void advised() {
        dummy = System.currentTimeMillis();
    }

    public void unadvised() {
        dummy = System.currentTimeMillis();
    }
}
```

В листинге 6.41 приведен код класса `TestPointcut`, который обеспечивает статическую проверку метода, снабжаемого советом.

Листинг 6.41. Класс `TestPointcut`

```
package com.apress.prospring3.ch6.proxies;

import java.lang.reflect.Method;

import org.springframework.aop.support.StaticMethodMatcherPointcut;

public class TestPointcut extends StaticMethodMatcherPointcut {

    public boolean matches(Method method, Class cls) {
        return ("advised".equals(method.getName()));
    }
}
```

В листинге 6.42 показан код класса `NoOpBeforeAdvice`, который представляет собой простой совет “перед”, не имеющий операций.

Листинг 6.42. Класс `NoOpBeforeAdvice`

```
package com.apress.prospring3.ch6.proxies;

import java.lang.reflect.Method;

import org.springframework.aop.MethodBeforeAdvice;

public class NoOpBeforeAdvice implements MethodBeforeAdvice {

    public void before(Method method, Object[] args, Object target)
        throws Throwable {
            // операции отсутствуют
    }
}
```

Наконец, в листинге 6.43 приведен код для теста производительности.

Листинг 6.43. Тестирование производительности прокси

```
package com.apress.prospring3.ch6.proxies;

import org.springframework.aop.Advisor;
import org.springframework.aop.framework.Advised;
import org.springframework.aop.framework.ProxyFactory;
import org.springframework.aop.support.DefaultPointcutAdvisor;

public class ProxyPerfTest {
    public static void main(String[] args) {
        ISimpleBean target = new SimpleBean();
        Advisor advisor = new DefaultPointcutAdvisor(new TestPointcut(),
                new NoOpBeforeAdvice());
```

```
    runCglibTests(advisor, target);
    runCglibFrozenTests(advisor, target);
    runJdkTests(advisor, target);
}

private static void runCglibTests(Advisor advisor, ISimpleBean target) {
    ProxyFactory pf = new ProxyFactory();
    pf.setProxyTargetClass(true);
    pf.setTarget(target);
    pf.addAdvisor(advisor);

    ISimpleBean proxy = (ISimpleBean)pf.getProxy();
    // Запуск тестов для стандарного прокси CGLIB.
    System.out.println("Running CGLIB (Standard) Tests");
    test(proxy);
}

private static void runCglibFrozenTests(Advisor advisor, ISimpleBean target) {
    ProxyFactory pf = new ProxyFactory();
    pf.setProxyTargetClass(true);
    pf.setTarget(target);
    pf.addAdvisor(advisor);
    pf.setFrozen(true);

    ISimpleBean proxy = (ISimpleBean)pf.getProxy();
    // Запуск тестов для прокси CGLIB с цепочкой фиксированного совета.
    System.out.println("Running CGLIB (Frozen) Tests");
    test(proxy);
}

private static void runJdkTests(Advisor advisor, ISimpleBean target) {
    ProxyFactory pf = new ProxyFactory();
    pf.setTarget(target);
    pf.addAdvisor(advisor);
    pf.setInterfaces(new Class[]{ISimpleBean.class});

    ISimpleBean proxy = (ISimpleBean)pf.getProxy();
    // Запуск тестов для прокси JDK.
    System.out.println("Running JDK Tests");
    test(proxy);
}

private static void test(ISimpleBean bean) {
    long before = 0;
    long after = 0;

    // Тест 1: проверка метода, снабженного советом
    System.out.println("Testing Advised Method");
    before = System.currentTimeMillis();
    for(int x = 0; x < 500000; x++) {
        bean.advised();
    }
    after = System.currentTimeMillis();;

    System.out.println("Took " + (after - before) + " ms");

    // Тест 2: проверка метода, не снабженного советом
    System.out.println("Testing Unadvised Method");
    before = System.currentTimeMillis();
    for(int x = 0; x < 500000; x++) {
        bean.unadvised();
    }
```

```
        after = System.currentTimeMillis();;

        System.out.println("Took " + (after - before) + " ms");

        // Тест 3: проверка метода equals()
        System.out.println("Testing equals() Method");
        before = System.currentTimeMillis();
        for(int x = 0; x < 500000; x++) {
           bean.equals(bean);
        }
        after = System.currentTimeMillis();;

        System.out.println("Took " + (after - before) + " ms");

        // Тест 4: проверка метода hashCode()
        System.out.println("Testing hashCode() Method");
        before = System.currentTimeMillis();
        for(int x = 0; x < 500000; x++) {
           bean.hashCode();
        }
        after = System.currentTimeMillis();;

        System.out.println("Took " + (after - before) + " ms");

        // Тест 5: проверка метода Advised
        Advised advised = (Advised)bean;

        System.out.println("Testing Advised.getProxyTargetClass() Method");
        before = System.currentTimeMillis();
        for(int x = 0; x < 500000; x++) {
           advised.getTargetClass();
        }
        after = System.currentTimeMillis();;

        System.out.println("Took " + (after - before) + " ms");

        System.out.println(">>>\n");
    }
}
```

В этом коде мы тестируем три разновидности прокси: стандартный прокси CGLIB, прокси CGLIB с цепочкой фиксированного совета (т.е., когда прокси зафиксирован вызовом метода `setFrozen()` в классе `ProxyConfig`, который класс `ProxyFactory` косвенно расширяет, библиотека CGLIB предпримет дальнейшую оптимизацию, но последующее изменение совета не разрешается), и прокси JDK. Для каждого типа прокси запускаются следующие пять тестовых сценариев.

- *Метод, снабженный советом (тест 1).* Метод, который снабжен советом. В этом тесте используется совет “перед”, не осуществляющий никакой обработки, что снижает влияние совета на тесты производительности.
- *Метод, не снабженный советом (тест 2).* Метод прокси, который не снабжен советом. Часто прокси имеет много методов, не снабженных советом. Этот тест позволяет выяснить, насколько хорошо методы, не снабженные советом, выполняются для различных прокси.
- *Метод* `equals()` *(тест 3).* Данный тест позволяет выяснить накладные расходы, связанные с вызовом метода `equals()`. Это особенно важно, когда прокси используются в качестве ключей в `HashMap` или подобной коллекции.

- *Метод* `hashcode()` *(тест 4).* Как и `equals()`, метод `hashCode()` важен, когда используются `HashMap` или похожие коллекции.
- *Выполнение методов интерфейса* `Advised` *(тест 5).* Как упоминалось ранее, прокси по умолчанию реализует интерфейс `Advised`, позволяя модифицировать прокси после создания и запрашивать информацию, связанную с прокси. Этот тест показывает, насколько быстро происходит обращение к методам интерфейса `Advised` с использованием различных типов прокси.

Мы прогоняли тесты на машине с процессором Intel Core i7 2,8 ГГц и объемом ОЗУ в 8 Гбайт. При этом начальный размер кучи JVM был установлен равным 2048 Мбайт, чтобы уменьшить влияние на результаты тестов со стороны изменения размера кучи. Результаты тестирования сведены в табл. 6.5.

Таблица 6.5. Результаты тестирования производительности прокси (значения указаны в миллисекундах)

	CGLIB (стандартный)	CGLIB (фиксированный)	JDK
Метод, снабженный советом	148	67	129
Метод, не снабженный советом	41	22	50
`equals()`	9	24	63
`hashCode()`	16	13	30
`Advised.getProxyTargetClass()`	7	14	15

Приведенные в таблице результаты показывают, что производительность стандартного прокси CGLIB и динамического прокси JDK не особенно отличается для методов, снабженных и не снабженных советом. Если вы читали предыдущее издание этой книги, то знаете, что производительность прокси JDK была намного ниже для методов, не снабженных советом. Причина в том, что в предыдущем издании тесты запускались под управлением текущей на тот момент версии JDK 1.4, а производительность рефлексии в ней была чрезвычайно низкой. В настоящее время мы запускали тесты под управлением версии JDK 6, производительность механизма рефлексии в которой существенно улучшилась.

Однако наблюдается заметная разница при использовании прокси CGLIB с цепочкой фиксированного совета. Похожая ситуация имеет место также с методами `equals()` и `hashCode()`, которые оказываются значительно быстрее, когда применяется прокси CGLIB. Методы интерфейса `Advised` также быстрее в случае прокси CGLIB. Это объясняется тем, что методы `Advised` обрабатываются раньше в методе `intercept()`, поэтому для них не выполняется большая часть логики, которая требуется остальным методам.

Какой прокси использовать?

Решение вопроса о том, какой прокси использовать, обычно не требует больших усилий. Прокси CGLIB предназначен как для классов, так и для интерфейсов, тогда как прокси JDK — только для интерфейсов. В плане производительности нет заметной разницы между стандартным режимом прокси JDK и CGLIB (во всяком случае, при выполнении методов, снабженных и не снабженных советом). В фиксированном режиме прокси CGLIB цепочка совета не может быть изменена и CGLIB проводит дополнительную оптимизацию. Когда создается прокси для класса, по умолчанию устанавливается

прокси CGLIB, поскольку это единственный прокси, который может быть сгенерирован для класса. Чтобы использовать прокси CGLIB для интерфейса, понадобится установить флаг `optimize` класса `ProxyFactory` в `true` с помощью метода `setOptimize()`.

На заметку! Кроме CGLIB имеется еще одна библиотека манипулирования байт-кодом, называемая Javassist, которая применяется в ряде других популярных проектов (например, Hibernate). Некоторые разработчики отдают предпочтение Javassist перед CGLIB и даже подняли вопрос о переносе прокси АОП в Spring с CGLIB на Javassist (`http://jira.springsource.org/browse/SPR-5654`). Команда разработки Spring рассмотрит его во время построения Spring 3.2. Существует также промежуточное решение, предоставленное сообществом. Те, кому интересны детали использования Javassist с АОП в Spring, могут проследовать по указанной выше ссылке.

Резюме

В этой главе мы предоставили введение в ключевые концепции АОП и показали, как эти концепции были воплощены в реализации АОП в Spring. Мы обсудили, какие функциональные средства реализованы в рамках АОП в Spring, а какие — нет, и указали на AspectJ как на решение АОП, поддерживающее возможности, которые отсутствуют в реализации Spring. Некоторое время было уделено объяснению деталей типов советов, доступных в Spring, и рассмотрению примеров четырех типов советов в действии. Мы также показали, как с помощью срезов ограничить методы, к которым применяется совет. В частности, были продемонстрированы шесть базовых реализаций срезов, предлагаемые Spring. Наконец, мы раскрыли детали конструирования прокси АОП, различные их варианты, а также отличия между ними. Обсуждение прокси было завершено сравнением производительности трех разных типов прокси и рассмотрением основных отличий и ограничений при выборе между прокси JDK и CGLIB.

В главе 7 мы завершим обсуждение срезов, доступных в Spring, детальным описанием `ComposablePointcut` и `ControlFlowPointcut`. Мы также уделим некоторое время использованию интеграции с AspectJ в Spring для расширения набора функциональных средств АОП, предназначенного для приложений. Мы также посмотрим, как АОП поддерживается службами Spring Framework, что означает возможность определения и конфигурирования совета декларативно, а не программно.

ГЛАВА 7

Дополнительные сведения об аспектно-ориентированном программировании в Spring и аннотациях

В этой главе мы рассмотрим дополнительные детали, связанные со средствами АОП, доступными в Spring. В частности, мы рассмотрим эту тему в контексте реальных приложений: исследуем службы инфраструктуры АОП в Spring, которые позволяют прозрачное применение АОП, раскроем реальное использование АОП на нашем примере приложения и обсудим способы преодоления ограничений АОП в Spring за счет интеграции Spring с AspectJ. В главе будут освещены следующие темы.

- *Расширенное использование срезов.* Эта глава завершает обсуждение применения срезов рассмотрением классов `ComposablePointcut` и `ControlFlowPointcut`. В данном разделе также приведено резюме по теме срезов и по соответствующим технологиям, которые должны применяться в случае использования срезов в приложении.
- *Введения.* Упомянутые в предыдущей главе введения позволяют добавлять реализации интерфейсов динамическим образом к любому объекту на лету, используя знакомую концепцию перехватчика.
- *Службы инфраструктуры АОП.* В предыдущей главе эта тема не рассматривалась, а все внимание было уделено ручной сборке прокси АОП и цепочек совета. Тем не менее, платформа Spring полностью поддерживает конфигурирование АОП в прозрачной и декларативной манере. В этом разделе мы рассмотрим три способа (класс `ProxyFactoryBean`, пространство имен `aop` и аннотации в стиле `@AspectJ`) внедрения декларативно определенных прокси АОП в объекты приложения в качестве коллаборaторов, делая приложение полностью не осведомленным о том, что оно работает с объектами, которые снабжены советами.

- *Интеграция с AspectJ.* Расширение AspectJ является полнофункциональной реализацией АОП. Основное отличие между AspectJ и АОП в Spring заключается в том, что AspectJ применяет совет к целевым объектам через связывание (либо во время компиляции, либо во время выполнения), тогда как АОП в Spring основано на использовании прокси. Набор функциональных возможностей AspectJ намного шире, чем в АОП в Spring, но им значительно сложнее пользоваться по сравнению со Spring. Расширение AspectJ будет подходящим решением, если обнаруживается, что АОП в Spring не поддерживает средство, которое необходимо для приложения. Начиная с версии 2.0, при конфигурировании аспектов AspectJ можно в полной мере пользоваться средствами Spring.

Для запуска некоторых примеров, рассмотренных в данной главе, понадобится получить AspectJ. Это расширение доступно для загрузки на веб-сайте `http://eclipse.org/aspectj`. В примерах этой главы мы применяли версию AspectJ 1.6.11. Можно также добавить в проект STS зависимости Maven, описанные в табл. 7.1.

Таблица 7.1. Зависимости Maven для AspectJ

Идентификатор группы	Идентификатор артефакта	Версия	Описание
`org.aspectj`	`aspectjrt`	1.6.11	Библиотека времени выполнения AspectJ
`org.aspectj`	`aspectjweaver`	1.6.11	Библиотека связывания AspectJ

Расширенное использование срезов

В предыдущей главе мы взглянули на шесть базовых реализаций интерфейса `Pointcut`, предоставляемых Spring; по большей части, это удовлетворяет потребностям многих приложений. Тем не менее, иногда возникает необходимость в большей гибкости при определении срезов. В Spring доступны две дополнительных реализации `Pointcut`, `ComposablePointcut` и `ControlFlowPointcut`, которые как раз и предоставляют нужную гибкость.

Использование срезов потока управления

Срезы потока управления Spring, реализованные классом `ControlFlowPointcut`, подобны конструкции `cflow`, доступной во многих других реализациях АОП, хотя они не настолько мощные. В сущности, срез потока управления в Spring перехватывает все вызовы методов, выполненные из заданного метода либо из всех методов в классе. Это довольно трудно представить визуально, поэтому рассмотрим, как работает такой срез, на простом примере.

В листинге 7.1 показан код класса `SimpleBeforeAdvice`, который выводит описание метода, снабжаемого советом.

Листинг 7.1. Класс `SimpleBeforeAdvice`

```
package com.apress.prospring3.ch7.cflow;
import java.lang.reflect.Method;
import org.springframework.aop.MethodBeforeAdvice;
public class SimpleBeforeAdvice implements MethodBeforeAdvice {
   public void before(Method method, Object[] args, Object target)
         throws Throwable {
      System.out.println("Before method: " + method);
   }
}
```

Этот класс совета позволяет увидеть, какой метод попадает в срез `ControlFlowPointcut`. В листинге 7.2 представлен простой класс с одним методом — методом, который необходимо снабдить советом.

Листинг 7.2. Класс `TestBean`

```
package com.apress.prospring3.ch7.cflow;

public class TestBean {

   public void foo() {
      System.out.println("foo()");
   }
}
```

В листинге 7.2 вы видите простой метод `foo()`, который нужно снабдить советом. Однако при этом должно быть удовлетворено специальное требование — этот метод необходимо снабдить советом, только когда он вызывается из другого заданного метода. В листинге 7.3 приведен код управляющей программы для рассматриваемого примера.

Листинг 7.3. Использование класса `ControlFlowPointcut`

```
package com.apress.prospring3.ch7.cflow;

import org.springframework.aop.Advisor;
import org.springframework.aop.Pointcut;
import org.springframework.aop.framework.ProxyFactory;
import org.springframework.aop.support.ControlFlowPointcut;
import org.springframework.aop.support.DefaultPointcutAdvisor;

public class ControlFlowExample {
   public static void main(String[] args) {
      ControlFlowExample ex = new ControlFlowExample();
      ex.run();
   }
   public void run() {
       TestBean target = new TestBean();

      // Создать экземпляр DefaultPointcutAdvisor.
      Pointcut pc = new ControlFlowPointcut(ControlFlowExample.class, "test");
      Advisor advisor = new DefaultPointcutAdvisor(pc,
         new SimpleBeforeAdvice());

      // Создать прокси.
      ProxyFactory pf = new ProxyFactory();
      pf.setTarget(target);
      pf.addAdvisor(advisor);
      TestBean proxy = (TestBean) pf.getProxy();

      // Попытка нормального вызова.
      System.out.println("Trying normal invoke");
      proxy.foo();
      // Попытка вызова из ControlFlowExample.test().
      System.out.println("Trying under ControlFlowExample.test()");
      test(proxy);
   }
   private void test(TestBean bean) {
      bean.foo();
   }
}
```

В листинге 7.3 прокси связывается с ControlFlowPointcut и метод foo() вызывается дважды: один раз напрямую из метода run() и один раз из метода test(). Особый интерес представляет следующая строка кода:

```
Pointcut pc = new ControlFlowPointcut(ControlFlowExample.class, "test");
```

В этой строке создается экземпляр ControlFlowPointcut для метода test() класса ControlFlowExample. В сущности, она говорит: "Включить в срез все методы, вызванные из метода ControlFlowExample.test()". Обратите внимание, что хотя формулировка выглядит как "Включить в срез все методы", на самом деле она означает "Включить в срез все методы объекта прокси, который снабжен советом с использованием Advisor, соответствующего этому экземпляру ControlFlowPointcut".

Кроме того, в проект понадобится добавить зависимость для библиотеки CGLIB, которая описана в табл. 7.2.

Таблица 7.2. Зависимость для CGLIB

Идентификатор группы	Идентификатор артефакта	Версия	Описание
cglib	cglib	2.2.2	Библиотека генерации кода, требуемая для АОП в Spring

Запуск на выполнение примера из листинга 7.3 дает следующий вывод:

```
Trying normal invoke
foo()
Trying under ControlFlowExample.test()
Before method: public void com.apress.prospring3.ch7.cflow.TestBean.foo()
foo()
```

Как видите, когда метод foo() вызывается в первый раз за пределами потока управления метода test(), он не снабжен советом. Когда foo() вызывается во второй раз, уже внутри потока управления метода test(), экземпляр ControlFlowPointcut показывает, что связанный с ним совет применяется к foo(), поэтому метод превращается в снабженный советом. Обратите внимание, что если бы мы вызвали внутри метода test() еще один метод, не относящийся к прокси, он не был бы снабжен советом.

Срезы потока управления могут быть исключительно полезны, позволяя снабжать объект советом выборочно, только когда объект выполняется в контексте другого объекта. Однако помните, что применение среза потока управления связано с существенным снижением производительности по сравнению с другими видами срезов. Данные, полученные из документации по Spring, показывают, что в среде JVM 1.4 срез потока управления обычно функционирует в пять раз медленнее, чем другие срезы.

Давайте рассмотрим пример. Предположим, что имеется система обработки транзакций, которая поддерживает интерфейсы TransactionService и AccountService. Мы хотим применить совет "после", чтобы в ситуации, когда метод AccountService.updateBalance() вызывается методом TransactionService.reverseTransaction(), после обновления баланса на счету клиенту отправлялось уведомление по электронной почте. Однако уведомление не должно посылаться ни при каких других обстоятельствах. В этом случае удобно воспользоваться срезом потока управления. На рис. 7.1 показана диаграмма последовательностей UML для описанного сценария.

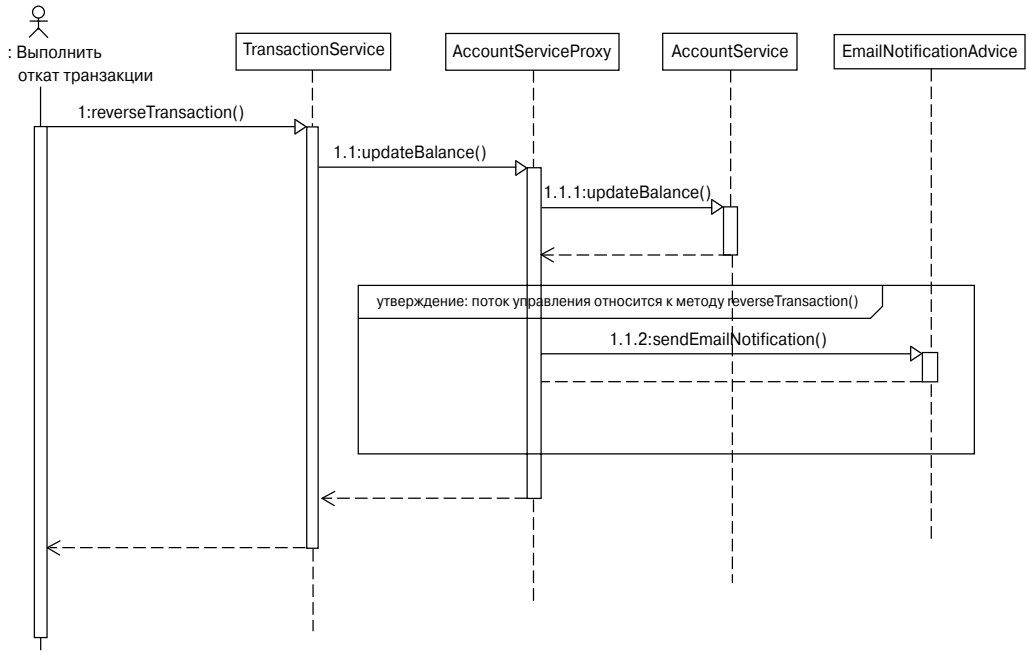


Рис. 7.1. Диаграмма последовательностей UML для среза потока управления

Использование компонуемых срезов

В предыдущих примерах создания срезов мы использовали по одному срезу для каждого экземпляра `Advisor`. В большинстве случаев этого вполне достаточно, но в некоторых ситуациях для достижения намеченной цели может понадобиться скомпоновать вместе два или более срезов. Представьте ситуацию, когда нужно включить в срез все методы получения и установки в классе бина. У вас есть срез для методов получения и срез для методов установки, но нет среза для обеих разновидностей методов. Разумеется, можно было бы создать еще один срез с новой логикой, но более удачный подход предполагает комбинирование упомянутых двух срезов в единый срез с использованием `ComposablePointcut`.

Класс `ComposablePointcut` поддерживает два метода: `union()` и `intersection()`. По умолчанию `ComposablePointcut` создается с реализацией интерфейса `ClassFilter`, которая соответствует всем классам, и реализацией интерфейса `MethodMatcher`, которая соответствует всем методам, хотя при конструировании можно указать собственные начальные `ClassFilter` и `MethodMatcher`. Методы `union()` и `intersection()` перегружены для принятия аргументов `ClassFilter` и `MethodMatcher`.

Метод `ComposablePointcut.union()` может быть вызван передачей экземпляра реализации одного из интерфейсов `ClassFilter`, `MethodMatcher` или `Pointcut`. Результат операции объединения заключается в том, что `ComposablePointcut` добавит условие “или” в свою цепочку вызова для поиска соответствия срезам. То же касается и метода `ComposablePointcut.intersection()`, но в этом случае вместо условия “или” добавляется условие “и”, которое означает, что все определения `ClassFilter`, `MethodMatcher` и `Pointcut` внутри `ComposablePointcut` должны давать соответствие, чтобы совет был применен. Это удобно представлять как конструкцию `WHERE` в SQL-запросе, причем метод `union()` похож на операцию “или”, а метод `intersection()` — на операцию “и”.

Как и в случае со срезами потока управления, это сложно воспроизвести визуально, а намного проще рассмотреть на примере. В листинге 7.4 показан простой бин с тремя методами.

Листинг 7.4. Класс `SampleBean`

```
package com.apress.prospring3.ch7.composable;

public class SampleBean {

   public String getName() {
      return "Clarence Ho";
   }

   public void setName(String name) {
   }

   public int getAge() {
      return 100;
   }
}
```

В этом примере мы собираемся сгенерировать три разных прокси, используя один и тот же экземпляр `ComposablePointcut`, но каждый раз модифицировать `ComposablePointcut` с применением метода `union()` или `intersection()`. После этого мы вызовем все три метода на прокси `SampleBean` и посмотрим, какие из них будут снабжены советом. В листинге 7.5 приведен необходимый код.

Листинг 7.5. Исследование `ComposablePointcut`

```
package com.apress.prospring3.ch7.composable;

import java.lang.reflect.Method;

import org.springframework.aop.Advisor;
import org.springframework.aop.ClassFilter;
import org.springframework.aop.framework.ProxyFactory;
import org.springframework.aop.support.ComposablePointcut;
import org.springframework.aop.support.DefaultPointcutAdvisor;
import org.springframework.aop.support.StaticMethodMatcher;

import com.apress.prospring3.ch7.cflow.SimpleBeforeAdvice;

public class ComposablePointcutExample {

   public static void main(String[] args) {
      // Создать целевой объект.
      SampleBean target = new SampleBean();

      ComposablePointcut pc = new ComposablePointcut(ClassFilter.TRUE,
         new GetterMethodMatcher());

      System.out.println("Test 1");      // тест 1
      SampleBean proxy = getProxy(pc, target);
      testInvoke(proxy);

      System.out.println("Test 2");      // тест 2
      pc.union(new SetterMethodMatcher());
      proxy = getProxy(pc, target);
      testInvoke(proxy);

      System.out.println("Test 3");      // тест 3
      pc.intersection(new GetAgeMethodMatcher());
```

```
        proxy = getProxy(pc, target);
        testInvoke(proxy);
    }

    private static SampleBean getProxy(ComposablePointcut pc,
            SampleBean target) {
        // Создать экземпляр DefaultPointcutAdvisor.
        Advisor advisor = new DefaultPointcutAdvisor(pc,
            new SimpleBeforeAdvice());

        // Создать прокси.
        ProxyFactory pf = new ProxyFactory();
        pf.setTarget(target);
        pf.addAdvisor(advisor);
        return (SampleBean) pf.getProxy();
    }

    private static void testInvoke(SampleBean proxy) {
        proxy.getAge();
        proxy.getName();
        proxy.setName("Clarence Ho");
    }

    private static class GetterMethodMatcher extends StaticMethodMatcher {

        public boolean matches(Method method, Class<?> cls) {
            return (method.getName().startsWith("get"));
        }
    }

    private static class GetAgeMethodMatcher extends StaticMethodMatcher {

        public boolean matches(Method method, Class<?> cls) {
            return "getAge".equals(method.getName());
        }
    }

    private static class SetterMethodMatcher extends StaticMethodMatcher {

        public boolean matches(Method method, Class<?> cls) {
            return (method.getName().startsWith("set"));
        }

    }
}
```

Первый момент, который следует отметить в этом примере, касается набора из трех закрытых реализаций интерфейса `MethodMatcher`. Класс `GetterMethodMatcher` соответствует всем методам, имена которых начинаются на `get`. Это `MethodMatcher` по умолчанию, который мы используем для сборки `ComposablePointcut`. В связи с этим мы ожидаем, что первый цикл обращений к методам `SampleBean` приведет к снабжению советом только методов `getAge()` и `getName()`.

Класс `SetterMethodMatcher` соответствует всем методам, имена которых начинаются с `set`, и он комбинируется с `ComposablePointcut`, используя `union()`, для второго цикла обращений. В этой точке мы имеем объединение двух `MethodMatcher`: один соответствует всем методам, начинающимся на `get`, а другой — всем методам, начинающимся на `set`. Здесь мы ожидаем, что будут снабжены советом все вызовы методов.

Класс `GetAgeMethodMatcher` является специализированным и соответствует только методу `getAge()`. Этот `MethodMatcher` комбинируется с `ComposablePointcut`, используя `intersection()`, для третьего цикла обращений. Поскольку `GetAgeMethodMatcher`

скомпонован с применением intersection(), в третьем цикле обращений мы ожидаем снабжения советом только метода getAge(), т.к. он является единственным методом, который соответствует всем скомпонованным MethodMatcher.

Запуск этого примера на выполнение дает в результате следующий вывод:

```
Test 1
Before method: public int com.apress.prospring3.ch7.composable.SampleBean.getAge()
Before method: public java.lang.String com.apress.prospring3.ch7
↳.composable.SampleBean.getName()
Test 2
Before method: public int com.apress.prospring3.ch7.composable.SampleBean.getAge()
Before method: public java.lang.String com.apress.prospring3.ch7
↳.composable.SampleBean.getName()
Before method: public void com.apress.prospring3.ch7
↳.composable.SampleBean.setName(java.lang.String)
Test 3
Before method: public int com.apress.prospring3.ch7.composable.SampleBean.getAge()
```

Как и ожидалось, при первом цикле обращений к методам прокси только методы getAge() и getName() были снабжены советом. На втором цикле, когда SetterMethodMatcher был скомпонован с помощью union(), советом оказались снабжены все методы. На третьем цикле в результате пересечения GetAgeMethodMatcher был снабжен советом один лишь метод getAge().

Хотя этот пример демонстрирует использование реализаций MethodMatcher только в процессе компоновки, не менее просто применять ClassFilter при построении среза. На самом деле, при построении составного среза можно использовать комбинацию реализаций MethodMatcher и ClassFilter.

Компоновка и интерфейс Pointcut

В предыдущем разделе было показано, как создавать составной срез, используя множество реализаций MethodMatcher и ClassFilter. Создавать составные срезы можно также с помощью других объектов, которые реализуют интерфейс Pointcut.

Другой способ построения составного среза предполагает применение класса org.springframework.aop.support.Pointcuts. Этот класс предоставляет три статических метода. Методы intersection() и union() принимают два среза в качестве аргументов и конструируют составной срез. С другой стороны, доступен также метод matches(Pointcut, Method, Class, Object[]), предназначенный для выполнения быстрой проверки, соответствует ли срез предоставленному методу, классу и аргументам метода.

Класс Pointcuts поддерживает операции только с двумя срезами. Таким образом, если нужно скомбинировать MethodMatcher и ClassFilter с Pointcut, следует использовать класс ComposablePointcut. Однако когда необходимо лишь скомбинировать два среза, класс Pointcuts применять удобнее.

Резюме по созданию срезов

Обсуждения в этой и предыдущей главах показывают, что Spring предлагает мощный набор реализаций Pointcut, которые должны удовлетворять большинству, если не всем, требованиям приложения. Не забывайте, что когда не удается найти срез, подходящий для ваших потребностей, вы можете создать собственную реализацию с нуля, реализовав Pointcut, MethodMatcher и ClassFilter.

Для комбинирования срезов и советов предусмотрены два подхода. Первый подход, который применялся до сих пор, предполагает отделение реализации среза от сове-

та. В показанном до настоящего момента коде мы создавали экземпляры реализаций Pointcut и затем использовали DefaultPointcutAdvisor для добавления совета вместе с Pointcut к прокси.

При втором подходе, который задействован во многих примерах в документации по Spring, Pointcut инкапсулируется внутри собственной реализации Advisor. Таким образом, имеется класс, который реализует Pointcut и PointcutAdvisor, с методом PointcutAdvisor.getPointcut(), просто возвращающим this. Этот подход используется многими классами Spring, такими как StaticMethodMatcherPointcutAdvisor.

Мы считаем первый подход более гибким, т.к. он позволяет применять разные реализации Pointcut с разными реализациями Advisor. Однако второй подход удобен в ситуациях, когда вы собираетесь использовать одну и ту же комбинацию Pointcut и Advisor в различных частях приложения или даже между разными приложениями. Второй подход также удобен в случае, если каждый Advisor должен иметь отдельный экземпляр Pointcut; вы достигаете этого, делая Advisor ответственным за создание Pointcut.

Вспомните из обсуждения производительности прокси в предыдущей главе, что методы, не снабженные советом, выполняются намного эффективнее, чем методы, снабженные советом. По данной причине вы должны удостовериться, что при использовании Pointcut вы снабжаете советом только те методы, для которых это абсолютно необходимо. Тем самым вы сократите объем ненужных накладных расходов, появляющихся в приложении из-за применения АОП.

Работа с введениями

Введения представляют собой важную часть набора функциональных возможностей АОП, доступных в Spring. За счет использования введений можно динамически добавлять новую функциональность к существующему объекту. В Spring можно вводить в существующий объект реализацию любого интерфейса. Возникает вполне закономерный вопрос: зачем может понадобиться добавление функциональности динамически во время выполнения, если ее легко добавить при проектировании? Ответ на этот вопрос прост. Функциональность добавляется динамически, если она является сквозной и ее довольно сложно реализовать с применением традиционного совета.

Основы введения

Платформа Spring трактует введения как специальный тип совета, точнее — как специальный тип совета “вместо”. Поскольку введения применяются исключительно на уровне классов, для них нельзя использовать срезы; они не совпадают семантически. Введение добавляет к классу новые реализации интерфейсов, а срез определяет, к каким методам применяется совет. Введение создается реализацией интерфейса IntroductionInterceptor, который расширяет интерфейсы MethodInterceptor и DynamicIntroductionAdvice. На рис. 7.2 показана эта структура с методами обоих интерфейсов.

Как видите, в интерфейсе MethodInterceptor определен метод invoke(). С помощью этого метода вы предоставляете реализацию для вводимых интерфейсов и выполняете перехват любых дополнительных методов по мере необходимости. Реализация всех методов для интерфейса внутри одного метода может оказаться затруднительной и, скорее всего, приведет к большому объему кода, с которым придется иметь дело при решении, какой из методов вызывать. К счастью, Spring предоставляет стандартную реализацию IntroductionInterceptor по имени DelegatingIntroductionInterceptor, которая намного упрощает создание введений.

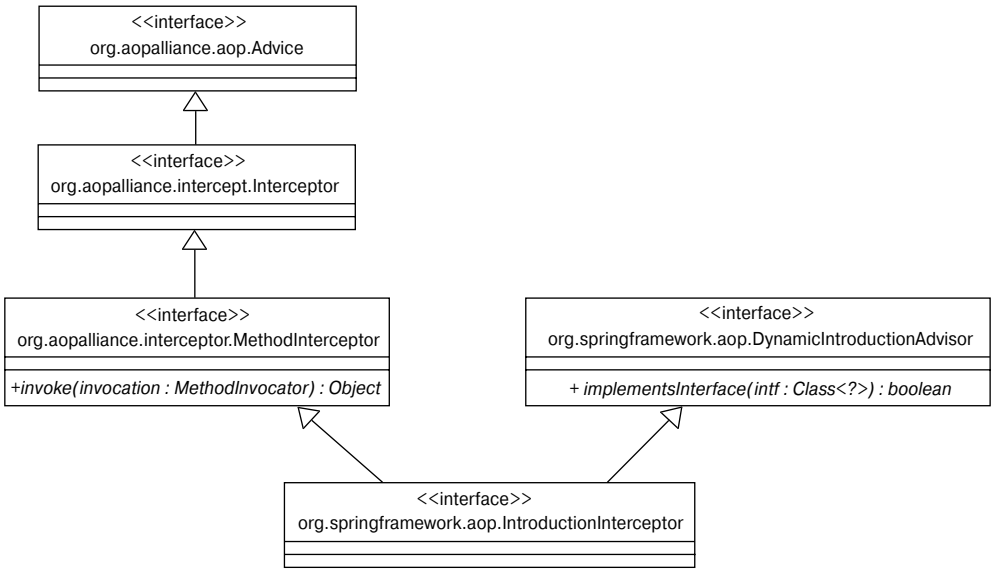


Рис. 7.2. Структура интерфейсов для введений

Чтобы построить введение с использованием `DelegatingIntroductionInterceptor`, вы создаете класс, который унаследован от `DelegatingIntroductionInterceptor` и реализует интерфейсы, предназначенные для введения.

Класс `DelegatingIntroductionInterceptor` затем просто делегирует все вызовы введенных методов соответствующему методу внутри себя. Не беспокойтесь, если это кажется не особенно понятным; в следующем разделе будет приведен пример.

Точно так же, как необходимо использовать `PointcutAdvisor` при работе с советом среза, потребуется применять `IntroductionAdvisor` для добавления введений к прокси. Стандартной реализацией `IntroductionAdvisor` является `DefaultIntroductionAdvisor`, которая должна удовлетворять большинству, если не всем, потребностям введения. Учтите, что добавление введения с помощью `ProxyFactory.addAdvice()` не разрешено и приводит к генерации исключения `AopConfigException`. Вместо него должен использоваться метод `addAdvisor()` с передачей ему экземпляра реализации интерфейса `IntroductionAdvisor`.

Стандартный совет — не введение — допускает применение одного и того же экземпляра для множества разных объектов. В документации по Spring это называется жизненным циклом на основе классов, хотя одиночный экземпляр совета можно использовать для многих разных классов. Совет типа введения формирует часть состояния объекта, снабженного советом, в результате чего для каждого такого объекта должен быть предусмотрен отдельный экземпляр совета. В документации это называется жизненным циклом на основе экземпляров. Так как необходимо гарантировать, что каждый снабженный советом объект имеет отдельный экземпляр введения, часть предпочтительнее создавать подкласс `DefaultIntroductionAdvisor`, отвечающий за создание совета типа введения. Таким образом, нужно лишь обеспечить создание нового экземпляра класса совета для каждого объекта, потому что это автоматически создает новый экземпляр введения.

Например, пусть требуется применить совет “перед” к методу `setFirstName()` на всех экземплярах класса `Contact`. На рис. 7.3 показан один и тот же совет, который применяется ко всем объектам типа `Contact`.

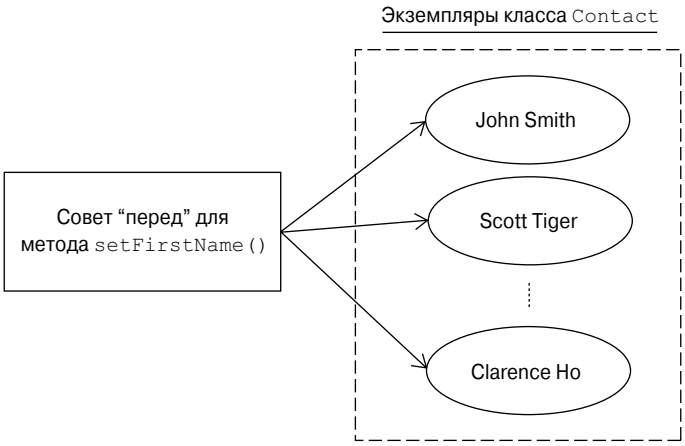


Рис. 7.3. Жизненный цикл на основе экземпляров для совета

А теперь предположим, что необходимо смешать это с введением во все экземпляры класса `Contact`, и это введение будет сопровождаться информацией для каждого экземпляра `Contact` (например, атрибутом `isModified`, который указывает, был ли экземпляр модифицирован). В этом случае введение будет создаваться для каждого экземпляра класса `Contact` и привязываться к этому экземпляру, как показано на рис. 7.4.

На этом рассмотрение основ введения завершено. Теперь мы обсудим, как использовать введения для решения задачи с обнаружением модификации объекта.

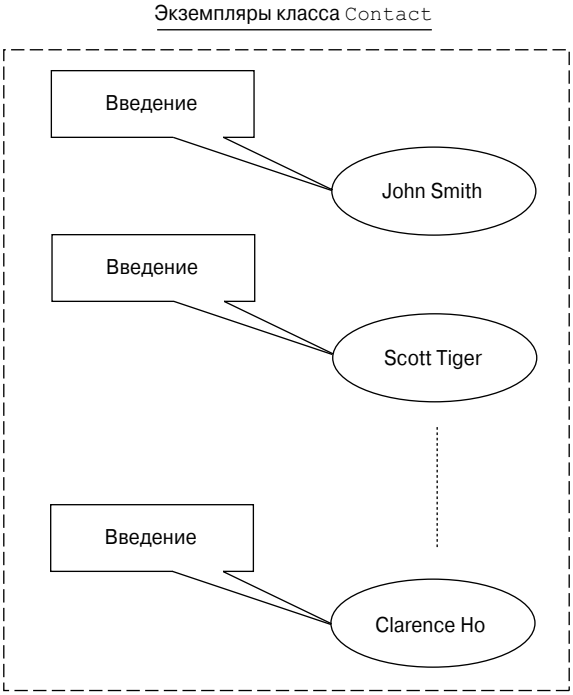


Рис. 7.4. Введение на основе экземпляров

Обнаружение модификации объекта с помощью введений

Обнаружение модификации объекта — полезный прием для достижения многих целей. Обычно обнаружение модификации применяется для предотвращения ненужного доступа к базе данных во время сохранения содержимого объекта. Если объект передается методу для модификации, но возвращается обратно неизмененным, не имеет смысла выполнять оператор обновления в базе данных. Использование проверки модификации подобным образом может увеличить пропускную способность приложения, особенно когда база данных уже работает с высокой нагрузкой или расположена в удаленной сети, увеличивая стоимость коммуникаций.

К сожалению, такой вид функциональности трудно реализовать вручную, потому что он требует добавления к каждому методу, который может модифицировать состояние объекта, проверки, действительно ли состояние было изменено. Если подсчитать все проверки на `null` и проверки, изменилось ли значение, получится примерно восемь строк кода на метод. Конечно, можно вынести этот код в отдельный метод, но все равно данный метод придется вызывать каждый раз, когда необходимо выполнить проверку. Распространение такого кода по всему приложению с множеством разных классов, требующих проверки модификации — мина замедленного действия.

В описанной ситуации помогут введения. Мы не хотим, чтобы каждый класс, требующий проверок модификации, наследовался от какой-то базовой реализации, в результате теряя свой единственный шанс на наследование, равно как и не хотим добавлять код проверки ко всем методам, которые могут изменять состояние объекта. С помощью введений мы можем обеспечить гибкое решение задачи обнаружения модификации, не требующее написания множества повторяющегося и подверженного ошибкам кода.

В рассматриваемом примере мы собираемся построить полноценную инфраструктуру проверок модификации, используя введения. Логика проверки модификации инкапсулирована в интерфейсе `IsModified`, реализация которого будет введена в соответствующие объекты вместе с логикой перехвата для автоматического выполнения проверок модификации. Для целей данного примера мы используем соглашения JavaBean, по которым считается, что модификация происходит при любом вызове метода установки. Разумеется, мы не трактуем все вызовы метода установки как модификацию — мы проверяем, отличается ли значение, переданное методу установки, от текущего значения, хранящегося в объекте. Единственным недостатком этого решения является то, что возврат объекта в первоначальное состояние будет считаться модификацией, если какое-то одно из значений объекта изменилось во время обработки. Например, пусть имеется объект `Contact` с атрибутом `firstName`. Предположим, что во время обработки атрибут `firstName` изменил свое значение с Peter на John. В результате объект помечается как модифицированный. Однако он все равно будет помечен как модифицированный, даже если при последующей обработке значение атрибута вернется обратно к исходному состоянию Peter. Один из способов учета такой ситуации предусматривает сохранение полной хронологии изменений в рамках жизненного цикла объекта. Тем не менее, предлагаемая нами реализация нетривиальна и достаточна для удовлетворения большинства требований. Реализация более полного решения могла бы привести к чрезмерному усложнению примера.

Интерфейс `IsModified`

Главным компонентом в решении проверки модификаций является интерфейс `IsModified`, который рассматриваемый пример приложения использует для принятия интеллектуальных решений о сохранении содержимого объектов. Мы не будем касаться того, как приложение могло бы использовать `IsModified`, а вместо этого сосредоточим внимание на реализации введения. Интерфейс `IsModified` показан в листинге 7.6.

Листинг 7.6. Интерфейс IsModified

```
package com.apress.prospring3.ch7.introductions;

public interface IsModified {
   public boolean isModified();
}
```

Здесь нет ничего особо интересного — просто один метод isModified(), отражающий, был ли объект модифицирован.

Создание смеси

Следующий шаг заключается в создании кода, который реализует интерфейс IsModified и будет введен в объекты; это называется смесью (mixin). Как упоминалось ранее, создавать смеси намного проще путем создания подклассов DelegatingIntroductionInterceptor, чем непосредственной реализацией интерфейса IntroductionInterceptor. Наш класс смеси, IsModifiedMixin, является подклассом DelegatingIntroductionInterceptor и также реализует интерфейс IsModified. Код класса IsModifiedMixin приведен в листинге 7.7.

Листинг 7.7. Класс IsModifiedMixin

```
package com.apress.prospring3.ch7.introductions;

import java.lang.reflect.Method;
import java.util.HashMap;
import java.util.Map;

import org.aopalliance.intercept.MethodInvocation;
import org.springframework.aop.support.DelegatingIntroductionInterceptor;

public class IsModifiedMixin extends DelegatingIntroductionInterceptor
      implements IsModified {

   private boolean isModified = false;

   private Map<Method, Method> methodCache = new HashMap<Method, Method>();

   public boolean isModified() {
      return isModified;
   }

   public Object invoke(MethodInvocation invocation) throws Throwable {

      if (!isModified) {
         if ((invocation.getMethod().getName().startsWith("set"))
            && (invocation.getArguments().length == 1)) {

            // Вызвать соответствующий метод get, чтобы посмотреть,
            // изменилось ли значение.
            Method getter = getGetter(invocation.getMethod());

            if (getter != null) {
               // Для методов, предназначенных только для записи,
               // проверка модификации не важна.
               Object newVal = invocation.getArguments()[0];
               Object oldVal = getter.invoke(invocation.getThis(), null);

               if((newVal == null) && (oldVal == null)) {
                  isModified = false;
               } else if((newVal == null) && (oldVal != null)) {
                  isModified = true;
```

```
                } else if((newVal != null) && (oldVal == null)) {
                    isModified = true;
                } else {
                    isModified = (!newVal.equals(oldVal));
                }
            }
        }
    }

    return super.invoke(invocation);
}

private Method getGetter(Method setter) {
    Method getter = null;

    // Попытка извлечения из кеша.
    getter = (Method) methodCache.get(setter);

    if (getter != null) {
        return getter;
    }

    String getterName = setter.getName().replaceFirst("set", "get");
    try {
        getter = setter.getDeclaringClass().getMethod(getterName, null);

        // Метод извлечения из кеша.
        synchronized (methodCache) {
            methodCache.put(setter, getter);
        }

        return getter;
    } catch (NoSuchMethodException ex) {
        // Должен быть только для записи.
        return null;
    }
}
}
```

Первым моментом, который следует отметить в коде, является реализация интерфейса `IsModified`, которая состоит из закрытого поля `isModified` и метода `isModified()`. Этот пример подчеркивает главную причину, по которой должен существовать только один экземпляр смеси на объект, снабженный советом — смесь вводит в объект не только методы, но также и состояние. Совместное использование единственного экземпляра этой смеси множеством разных объектов приведет также к разделению его состояния, и в результате окажется, что после первого же изменения какого-то одного объекта модифицированными будут считаться все объекты.

В действительности реализация метода `invoke()` для смеси не является обязательной, но в данном случае она позволяет обнаруживать модификацию автоматически. Реализация проверки модификации довольно проста. Мы начинаем с выполнения проверки, только если объект все еще не изменялся; проверять на предмет модификаций объект, в отношении которого известно, что он изменялся, необходимости нет. Затем мы выясняем, является ли метод установщиком, и если это так, извлекаем соответствующий метод получателя. Обратите внимание, что пара получателя/установщика кешируется для ускорения будущих извлечений. Наконец, мы сравниваем значение, возвращенное получателем, со значением, переданным установщику, чтобы определить, произошла ли модификация. При этом выполняются проверки с возможными комбинациями значения `null` с соответствующей установкой `isModified`.

Важно помнить, что в случае использования DelegatingIntroductionInterceptor при переопределении метода invoke() потребуется вызвать super.invoke(), потому что DelegatingIntroductionInterceptor в действительности направляет обращения корректному адресату — либо объекту, снабженному советом, либо самой смеси.

В смеси можно реализовать произвольное количество интерфейсов, и каждый из них будет автоматически введен в объект, снабженный советом.

Создание аспекта

На следующем шаге создается Advisor, служащий оболочкой для создания класса смеси. Этот шаг не обязателен, но он поможет обеспечить использование для каждого объекта, снабженного советом, нового экземпляра класса смеси. Код класса IsModifiedAdvisor показан в листинге 7.8.

Листинг 7.8. Создание совета для смеси

```
package com.apress.prospring3.ch7.introductions;

import org.springframework.aop.support.DefaultIntroductionAdvisor;

public class IsModifiedAdvisor extends DefaultIntroductionAdvisor {

    public IsModifiedAdvisor() {
        super(new IsModifiedMixin());
    }
}
```

Обратите внимание, что для создания IsModifiedAdvisor был расширен DefaultIntroductionAdvisor. Полученная реализация довольно проста и не требует дополнительных пояснений.

Собираем все вместе

Теперь, имея класс смеси и класс Advisor, мы можем протестировать инфраструктуру проверки модификации. В листинге 7.9 приведен простой класс, предназначенный для тестирования IsModifiedMixin.

Листинг 7.9. Класс TargetBean

```
package com.apress.prospring3.ch7.introductions;

public class TargetBean {

    private String name;

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

Этот бин имеет единственное свойство name, которое мы используем при тестировании смеси проверки модификации. В листинге 7.10 показано, как собрать прокси для совета и затем провести тестирование кода проверки модификаций.

Листинг 7.10. Использование IsModifiedMixin

```
package com.apress.prospring3.ch7.introductions;

import org.springframework.aop.IntroductionAdvisor;
import org.springframework.aop.framework.ProxyFactory;
public class IntroductionExample {
   public static void main(String[] args) {
      // Создать целевой объект.
      TargetBean target = new TargetBean();
      target.setName("Clarence Ho");

      // Создать аспект.
      IntroductionAdvisor advisor = new IsModifiedAdvisor();

      // Создать прокси.
      ProxyFactory pf = new ProxyFactory();
      pf.setTarget(target);
      pf.addAdvisor(advisor);
      pf.setOptimize(true);
      TargetBean proxy = (TargetBean)pf.getProxy();
      IsModified proxyInterface = (IsModified)proxy;

      // Протестировать интерфейсы.
      System.out.println("Is TargetBean?: " + (proxy instanceof TargetBean));
      System.out.println("Is IsModified?: " + (proxy instanceof IsModified));

      // Протестировать реализацию проверки модификаций.
      System.out.println("Has been modified?: " +
         proxyInterface.isModified());
      proxy.setName("Clarence Ho");
      System.out.println("Has been modified?: " +
         proxyInterface.isModified());
      proxy.setName("Rob Harrop");
      System.out.println("Has been modified?: " +
         proxyInterface.isModified());
   }
}
```

Обратите внимание, что при создании прокси флаг оптимизации устанавливается в `true`, чтобы обеспечить использование прокси CGLIB. Причина в том, что если выбрать для введения смеси прокси JDK, то результирующий прокси не будет экземпляром класса (в данном случае `TargetBean`); такой прокси реализует только интерфейсы смеси, но не исходный класс. В случае прокси CGLIB исходный класс реализуется прокси наряду с интерфейсами смеси.

В коде сначала осуществляется проверка, что прокси является экземпляром `TargetBean`, а затем — что это экземпляр `IsModified`. Обе проверки возвращают `true`, если используется прокси CGLIB, но для прокси JDK значение `true` дает только проверка на `IsModified`. Наконец, мы тестируем код проверки модификаций, сначала устанавливая свойство `name` в его текущее значение, а потом — в новое значение, каждый раз проверяя значение флага `isModified`. Запуск этого примера дает в результате следующий вывод:

```
Is TargetBean?: true
Is IsModified?: true
Has been modified?: false
Has been modified?: false
Has been modified?: true
```

Как и ожидалось, обе проверки `instanceof` возвращают `true`. Обратите внимание, что первый вызов `isModified()`, до возникновения всех модификаций, возвращает `false`. Следующий вызов, после установки свойства `name` в то же самое значение, также возвращает `false`. Однако финальный вызов метода `isModified()`, после того, как `name` установлено в новое значение, возвращает `true`, указывая на то, что объект на самом деле был изменен.

Резюме по введениям

Введения являются одним из наиболее мощных средств АОП в Spring; они позволяют не только расширять функциональность существующих методов, но также динамически расширять набор интерфейсов и реализаций объектов. Использование введений — это великолепный способ реализации сквозной логики, с которой приложение взаимодействует через четко определенные интерфейсы. В общем, это такая разновидность логики, которую желательно применять декларативно, а не программно. Используя класс `IsModifiedMixin`, построенный в рассмотренном примере, и службы платформы, обсуждаемые в следующем разделе, мы можем определить декларативно, какие объекты обладают возможностью проверки модификаций, не изменяя реализации этих объектов.

Очевидно, что поскольку введения работают через прокси, они добавляют определенный объем накладных расходов, при этом все методы прокси считаются снабженными советом, т.к. применять срезы вместе с введениями не допускается. Тем не менее, с учетом широты спектра служб, которые можно реализовать с помощью введений (в их числе и проверка модификации объекта), накладные расходы, связанные с производительностью — это лишь небольшая плата за сокращение объема кода, требуемого для реализации служб, а также за повышенную устойчивость и улучшенные возможности сопровождения, обеспечиваемые централизацией логики служб.

Службы платформы, предназначенные для АОП

К этому моменту мы написали немало кода для снабжения объектов советами и генерации прокси для них. Хотя само по себе это не является большой проблемой, но все же означает, что вся конфигурация советов жестко закодирована в приложении, убирая некоторые преимущества возможности прозрачного снабжения советом реализации метода. К счастью, Spring предоставляет дополнительные службы платформы, позволяющие создать прокси для совета в конфигурации приложения и затем внедрить этот прокси в целевой бин подобно любым другим зависимостям.

Использование декларативного подхода к конфигурации АОП предпочтительнее ручного программного подхода. Применение декларативного механизма не только позволяет вынести вовне конфигурацию совета, но и снижает шансы на допущение ошибок при кодировании. Можно также получить преимущество от совместного использования внедрения зависимостей и АОП, получая, в конечном счете, полностью прозрачную среду.

Декларативное конфигурирование АОП

Для декларативного конфигурирования АОП в Spring доступны три опции.

- *Использование `ProxyFactoryBean`.* В АОП, реализованном Spring, класс `ProxyFactoryBean` предоставляет декларативный способ конфигурирования `ApplicationContext` (и, следовательно, лежащего в основе `BeanFactory`) при создании прокси АОП на основе определенных бинов Spring.
- *Использование пространства имен `aop` в Spring.* Появившееся в версии Spring 2.0, пространство имен `aop` предоставляет упрощенный (по сравнению с

ProxyFactoryBean) способ определения аспектов и их требований DI в Spring-приложениях. Тем не менее, пространство имен aop использует "за кулисами" ProxyFactoryBean.

- *Использование аннотация в стиле @AspectJ*. Кроме пространства имен aop, основанного на XML, для конфигурирования АОП внутри классов можно также применять аннотации в стиле @AspectJ. Хотя используемый в этом случае синтаксис основан на AspectJ и понадобится включить ряд библиотек AspectJ, Spring по-прежнему применяет механизм прокси, т.е. создает прокси-объекты для целей во время начальной загрузки ApplicationContext.

Все эти опции по очереди рассматриваются в последующих разделах.

Использование ProxyFactoryBean

Класс ProxyFactoryBean — реализация FactoryBean, позволяющая указать целевой бин и предоставляющая для этого бина набор советов и аспектов, которые впоследствии объединяются в прокси АОП. Поскольку с ProxyFactoryBean можно использовать совет и аспект, декларативное конфигурирование доступно не только для совета, но также и для срезов.

Класс ProxyFactoryBean разделяет общий интерфейс (org.springframework.aop.framework.Advised) с ProxyFactory (оба класса косвенно расширяют класс org.springframework.aop.framework.AdvisedSupport, который реализует интерфейс Advised), в результате чего в нем доступны многие из тех же самых флагов — frozen, optimize и exposeProxy. Значения для этих флагов передаются прямо лежащему в основе классу ProxyFactory, который также позволяет декларативно конфигурировать фабрику.

Класс ProxyFactoryBean в действии

Работать с классом ProxyFactoryBean в действительности очень просто. Вы определяете бин, который будет целевым, затем используете ProxyFactoryBean и определяете бин, к которому приложение будет получать доступ, применяя целевой бин в качестве цели прокси. Где только возможно, определяйте целевой бин как анонимный бин внутри объявления бина прокси. Это предотвратит приложение от случайного доступа к бину, не снабженному советом. Однако в ряде случаев, таком как рассматриваемый здесь пример, может потребоваться создание более чем одного прокси для того же самого бина, и тогда должен использоваться нормальный бин верхнего уровня.

В листингах 7.11 и 7.12 показаны два класса, один из которых зависит от другого.

Листинг 7.11. Класс MyDependency

```
package com.apress.prospring3.ch7.pfb;

public class MyDependency {

    public void foo() {
        System.out.println("foo()");
    }

    public void bar() {
        System.out.println("bar()");
    }
}
```

Листинг 7.12. Класс MyBean

```
package com.apress.prospring3.ch7.pfb;

public class MyBean {

   private MyDependency dep;

   public void execute() {
      dep.foo();
      dep.bar();
   }

   public void setDep(MyDependency dep) {
      this.dep = dep;
   }
}
```

В этом примере мы собираемся создать два прокси для одного экземпляра MyDependency, оба с тем же самым базовым советом, который приведен в листинге 7.13.

Листинг 7.13. Класс MyAdvice

```
package com.apress.prospring3.ch7.pfb;

import java.lang.reflect.Method;
import org.springframework.aop.MethodBeforeAdvice;

public class MyAdvice implements MethodBeforeAdvice {

   public void before(Method method, Object[] args, Object target)
         throws Throwable {
      System.out.println("Executing: " + method);
   }
}
```

Первый прокси будет просто снабжать целевой объект советом напрямую; таким образом, будут снабжены все методы. Для второго прокси мы сконфигурируем AspectJExpressionPointcut и DefaultPointcutAdvisor так, чтобы советом снабжался только метод foo() класса MyDependency. Чтобы протестировать совет, мы создадим два определения бинов типа MyBean, каждое из которых будет внедрено с помощью различного прокси. Затем мы вызовем метод execute() на каждом из этих бинов и посмотрим, что происходит при вызове методов, снабженных советом, на зависимости.

В листинге 7.14 показана конфигурация для этого примера (pfb.xml).

Листинг 7.14. Декларативная конфигурация АОП

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-3.1.xsd">

   <bean id="myBean1" class="com.apress.prospring3.ch7.pfb.MyBean">
      <property name="dep">
         <ref local="myDependency1"/>
      </property>
   </bean>
```

```
   <bean id="myBean2" class="com.apress.prospring3.ch7.pfb.MyBean">
      <property name="dep">
         <ref local="myDependency2"/>
      </property>
   </bean>

   <bean id="myDependencyTarget" class="com.apress.prospring3.ch7.pfb.MyDependency"/>

   <bean izZd="myDependency1"
         class="org.springframework.aop.framework.ProxyFactoryBean">
      <property name="target">
         <ref local="myDependencyTarget"/>
      </property>
      <property name="interceptorNames">
         <list>
            <value>advice</value>
         </list>
      </property>
   </bean>

   <bean id="myDependency2"
        class="org.springframework.aop.framework.ProxyFactoryBean">
      <property name="target">
         <ref local="myDependencyTarget"/>
      </property>
      <property name="interceptorNames">
         <list>
            <value>advisor</value>
         </list>
      </property>
   </bean>

   <bean id="advice" class="com.apress.prospring3.ch7.pfb.MyAdvice"/>

   <bean id="advisor"
        class="org.springframework.aop.support.DefaultPointcutAdvisor">
      <property name="advice">
         <ref local="advice"/>
      </property>
      <property name="pointcut">
         <bean class="org.springframework.aop.aspectj.AspectJExpressionPointcut">
            <property name="expression">
               <value>execution(* foo*(..))</value>
            </property>
         </bean>
      </property>
   </bean>
</beans>
```

Этот код должен выглядеть вполне знакомым. Обратите внимание, что мы не делаем в нем ничего особенного; мы просто устанавливаем свойства, которые устанавливали в коде с использованием возможностей DI в Spring. Единственный примечательный момент связан с использованием анонимного бина для среза и класса `ProxyFactoryBean`. Мы предпочитаем применять анонимные бины для срезов, когда они не являются совместно используемыми, поскольку это позволяет удерживать набор бинов, к которым разрешен прямой доступ, насколько возможно малым и релевантным приложению. Применяя `ProxyFactoryBean`, важно понимать, что объявление `ProxyFactoryBean` будет видно приложению, и оно будет использоваться при удовлетворении зависимо-

стей. Объявление лежащего в основе целевого бина не снабжается советом, так что это бин должен применяться, только когда нужно обойти инфраструктуру АОП, хотя в общем случае приложение не обязано быть осведомленным об инфраструктуре АОП и, следовательно, пытаться обойти ее. По этой причине анонимные бины должны использоваться везде, где это возможно, чтобы избежать случайного доступа к ним со стороны приложения.

В листинге 7.15 приведен код простого класса, который захватывает два экземпляра `MyBean` из `ApplicationContext` и затем вызывает для каждого из них метод `execute()`.

Листинг 7.15. Класс `ProxyFactoryBeanExample`

```
package com.apress.prospring3.ch7.pfb;

import org.springframework.context.support.GenericXmlApplicationContext;

public class ProxyFactoryBeanExample {

   public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:pfb.xml");
      ctx.refresh();

      MyBean bean1 = (MyBean)ctx.getBean("myBean1");
      MyBean bean2 = (MyBean)ctx.getBean("myBean2");

      System.out.println("Bean 1");
      bean1.execute();

      System.out.println("\nBean 2");
      bean2.execute();
   }
}
```

Запуск примера из листинга 7.15 дает в результате следующий вывод:

```
Bean 1
Executing: public void com.apress.prospring3.ch7.pfb.MyDependency.foo()
foo()
Executing: public void com.apress.prospring3.ch7.pfb.MyDependency.bar()
bar()

Bean 2
Executing: public void com.apress.prospring3.ch7.pfb.MyDependency.foo()
foo()
bar()
```

Как и ожидалось, оба метода — `foo()` и `bar()` — в первом прокси были снабжены советом, т.к. в их конфигурации никакие срезы не применялись. Однако во втором прокси был снабжен советом только метод `foo()`, потому что в конфигурации использовался срез.

Использование класса `ProxyFactoryBean` для введений

Применение класса `ProxyFactoryBean` не ограничивается одним лишь снабжением объекта советом, он также может использоваться для введения смесей в объекты. Вспомните из предшествующего обсуждения введений, что для добавления введения должен применяться интерфейс `IntroductionAdvisor`; добавить введение напрямую нельзя. То же самое правило применяется и при использовании `ProxyFactoryBean` с введениями. Как упоминалось ранее, когда применяется `ProxyFactoryBean`, прокси

намного проще конфигурировать, если создан специальный `Advisor` для смеси. В листинге 7.16 показан пример конфигурации для введения `IsModifiedMixin`, которое рассматривалось ранее (`introductions.xml`).

Листинг 7.16. Конфигурирование введений с помощью `ProxyFactoryBean`

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://www.springframework.org/schema/beans
     http://www.springframework.org/schema/beans/spring-beans-3.1.xsd">

   <bean id="bean" class="org.springframework.aop.framework.ProxyFactoryBean">
      <property name="target">
         <bean  class="com.apress.prospring3.ch7.introductions.TargetBean">
            <property name="name">
               <value>Clarence Ho</value>
            </property>
         </bean>
      </property>
      <property name="interceptorNames">
         <list>
            <value>advisor</value>
         </list>
      </property>
      <property name="proxyTargetClass">
         <value>true</value>
      </property>
   </bean>

   <bean id="advisor" class="com.apress.prospring3.ch7.introductions.
IsModifiedAdvisor"/>
</beans>
```

В конфигурации видно, что класс `IsModifiedAdvisor` используется в качестве аспекта для `ProxyFactoryBean`, и поскольку не нужно создавать другой прокси для того же самого целевого объекта, мы применяем анонимное объявление для целевого бина. В листинге 7.17 показано изменение предыдущего примера введения, в котором прокси получается из `ApplicationContext`.

Листинг 7.17. Класс `IntroductionConfigExample`

```
package com.apress.prospring3.ch7.introductions;

import org.springframework.context.support.GenericXmlApplicationContext;

public class IntroductionConfigExample {

   public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:introductions.xml");
      ctx.refresh();

      TargetBean bean = (TargetBean) ctx.getBean("bean");
      IsModified mod = (IsModified) bean;

      // Протестировать интерфейсы.
      System.out.println("Is TargetBean?: " + (bean instanceof TargetBean));
      System.out.println("Is IsModified?: " + (bean instanceof IsModified));
```

```
        // Протестировать реализацию проверки модификаций.
        System.out.println("Has been modified?: " + mod.isModified());
        bean.setName("Clarence Ho");
        System.out.println("Has been modified?: " + mod.isModified());
        bean.setName("Rob Harrop");
        System.out.println("Has been modified?: " + mod.isModified());
    }
}
```

Запуск этого примера на выполнение дает в точности такой же вывод, как и предыдущий пример введения, но на этот прокси получается из `ApplicationContext` и в коде приложения отсутствует какая-либо конфигурация.

Резюме по классу `ProxyFactoryBean`

При использовании класса `ProxyFactoryBean` можно конфигурировать прокси АОП, что обеспечивает гибкость программного метода без необходимости в привязке кода приложения к конфигурации АОП. Если не требуется принятие решений о том, как должны создаваться прокси, во время выполнения, то лучше придерживаться декларативного, а не программного, подхода к конфигурированию прокси. А теперь давайте перейдем к рассмотрению других двух опций для декларативного конфигурирования АОП в Spring, которые являются рекомендуемыми опциями для приложений, основанных на Spring 2.0 и выше с JDK 5 или более новой версии.

Использование пространства имен `aop`

Пространство имен `aop` предлагает значительно упрощенный синтаксис для декларативного конфигурирования АОП в Spring. Для демонстрации его работы воспользуемся предыдущим примером с `ProxyFactoryBean` и соответствующим образом модифицируем его.

В листингах 7.18 и 7.19 показаны несколько измененные классы `MyDependency` и `MyBean`.

Листинг 7.18. Класс `MyDependency`

```
package com.apress.prospring3.ch7.aopns;

public class MyDependency {

    public void foo(int intValue) {
        System.out.println("foo(int): " + intValue);
    }

    public void bar() {
        System.out.println("bar()");
    }
}
```

Листинг 7.19. Класс `MyBean`

```
package com.apress.prospring3.ch7.aopns;

public class MyBean {

    private MyDependency dep;

    public void execute() {
        dep.foo(100);
```

```
        dep.foo(101);
        dep.bar();
    }

    public void setDep(MyDependency dep) {
        this.dep = dep;
    }
}
```

В предыдущем листинге метод `foo()` класса `MyDependency` был модифицирован для принятия целочисленного значения в качестве аргумента. И в классе `MyBean` метод `foo()` был вызван два раза с разными параметрами.

Давайте посмотрим, как выглядит класс совета. В листинге 7.20 приведен пересмотренный код для класса `MyAdvice`.

Листинг 7.20. Класс `MyAdvice`

```
package com.apress.prospring3.ch7.aopns;

import org.aspectj.lang.JoinPoint;

public class MyAdvice {

    public void simpleBeforeAdvice(JoinPoint joinPoint) {
        System.out.println("Executing: " +
                joinPoint.getSignature().getDeclaringTypeName() + " "
                + joinPoint.getSignature().getName());
    }
}
```

Вы увидите, что класс совета больше не нуждается в реализации интерфейса `MethodBeforeAdvice`. Кроме того, совет “перед” принимает в качестве аргумента `JoinPoint`, но не метод, объект и аргументы. В действительности, для класса совета этот аргумент является необязательным, так что можете оставить этот метод без аргументов. Однако если в совете необходим доступ к деталям среза (в этом случае мы хотим получить копию информации о вызывающем типе и имени метода), то аргумент должен приниматься. Когда аргумент для метода определен, Spring будет автоматически передавать методу срез для дальнейшей обработки.

В листинге 7.21 приведена XML-конфигурация АОП в Spring с пространством имен `aop` (`aopns.xml`).

Листинг 7.21. Конфигурирование АОП в Spring с использованием пространства имен `aop`

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns:aop="http://www.springframework.org/schema/aop"
     xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop-3.1.xsd">

    <aop:config>

        <aop:pointcut id="fooExecution"
            expression="execution(* com.apress.prospring3.ch7..foo*(int))"/>

        <aop:aspect ref="advice">
```

```
        <aop:before pointcut-ref="fooExecution"
           method="simpleBeforeAdvice"/>
      </aop:aspect>

   </aop:config>

   <bean id="advice" class="com.apress.prospring3.ch7.aopns.MyAdvice"/>

   <bean id="myDependency"
      class="com.apress.prospring3.ch7.aopns.MyDependency"/>

   <bean id="myBean" class="com.apress.prospring3.ch7.aopns.MyBean">
      <property name="dep" ref="myDependency"/>
   </bean>
</beans>
```

Во-первых, мы должны объявить пространство имен `aop` в дескрипторе `<beans>`. Во-вторых, вся конфигурация АОП в Spring помещена в дескриптор `<aop:config>`. В рамках этого дескриптора мы можем определить срез, аспекты, советы и т.д., а также ссылаться на другие бины Spring обычным образом.

В предыдущей конфигурации был определен срез с идентификатором `fooExecution`. Выражение `"execution(* com.apress.prospring3.ch7..foo*(int))"` означает, что необходимо снабдить советом все методы с префиксом `foo` в классах, определенных в пакете `com.apress.prospring3.ch7` (включая все его подпакеты). Кроме того, метод `foo()` должен принимать один аргумент целочисленного типа. Далее был объявлен аспект в дескрипторе `<aop:aspect>` со ссылкой на класс совета как на бин Spring с идентификатором `"advice"`, который представляет собой класс `MyAdvice`.

В атрибуте `pointcut-ref` указана ссылка на определенный срез с идентификатором `fooExecution`, а совет “перед” (объявленный с помощью дескриптора `<aop:before>`) реализован методом `simpleBeforeAdvice()` внутри бина совета.

В листинге 7.22 приведена тестовая программа.

Листинг 7.22. Класс AopNamespaceExample

```
package com.apress.prospring3.ch7.aopns;

import org.springframework.context.support.GenericXmlApplicationContext;

public class AopNamespaceExample {
   public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:aopns.xml");
      ctx.refresh();

      MyBean myBean = (MyBean) ctx.getBean("myBean");
      myBean.execute();
   }
}
```

Этот класс не требует особых пояснений. Здесь просто инициализируется `ApplicationContext`, как обычно, извлекается бин и вызывается его метод `execute()`. Запуск этой программы на выполнение дает следующий вывод:

```
Executing: com.apress.prospring3.ch7.aopns.MyDependency foo
foo(int): 100
Executing: com.apress.prospring3.ch7.aopns.MyDependency foo
foo(int): 101
bar()
```

Как видите, два вызова метода foo() были снабжены советом, но не вызов метода bar(). Пример работает в точности так, как ожидалось, и вы видите, что конфигурация в нем была значительно упрощена по сравнению с конфигурацией ProxyFactoryBean.

Давайте снова займемся переделкой предыдущего примера для учета более сложного случая. Предположим, что необходимо снабдить советом только методы бинов Spring с идентификаторами, начинающимися с myDependency, которые принимают целочисленный аргумент со значением, не равным 100.

Чтобы запустить совет только когда значение аргумента не равно 100, мы должны модифицировать совет. Переделанный класс MyAdvice показан в листинге 7.23.

Листинг 7.23. Класс MyAdvice (переделанный для проверки аргумента)

```
package com.apress.prospring3.ch7.aopns;

import org.aspectj.lang.JoinPoint;

public class MyAdvice {

    // Для извлечения среза и аргумента.
    public void simpleBeforeAdvice(JoinPoint joinPoint, int intValue) {

        // Выполнять, только если intValue не равно 100.
        if (intValue != 100) {
            System.out.println("Executing: " +
                joinPoint.getSignature().getDeclaringTypeName() + " "
                + joinPoint.getSignature().getName()
                + " argument: " + intValue);
        }
    }
}
```

Изменения в классе были внесены в двух местах. Во-первых, к сигнатуре совета “перед” был добавлен аргумент intValue. Во-вторых, внутри совета производится проверка и логика запускается, только если значение аргумента не равно 100.

Для передачи аргумента совету также понадобится немного подкорректировать XML-конфигурацию. В этом случае нужно модифицировать выражение среза. Измененное выражение среза показано ниже:

```
<aop:pointcut id="fooExecution" expression="execution(* foo*(int))
    and args(intValue) and bean(myDependency*)"/>
```

К выражению среза были добавлены две дополнительных директивы. Первая из них, args(intValue), сообщает Spring о необходимости передачи аргумента по имени intValue в совет “перед”. Вторая директива, bean(myDependency*), указывает Spring снабжать советом только бины, имеющие префикс идентификатора myDependency. Это весьма мощное средство; при наличии четко определенной структуры имен бинов Spring, вы можете очень легко снабжать советом только необходимые объекты. Например, используя bean(*DAO*), совет можно применять ко всем бинам DAO, а с помощью bean(*Service*) — применять его ко всем бинам, относящимся к уровню обслуживания, не указывая полностью определенные имена классов для сопоставления.

Запуск той же самой тестовой программы (AopNamespaceExample) даст следующий вывод:

```
foo(int): 100
Executing: com.apress.prospring3.ch7.aopns.MyDependency foo argument: 101
foo(int): 101
bar()
```

В выводе видно, что советом был снабжен только вызов метода `foo()` с аргументом, имеющим значение, не равное 100.

Давайте рассмотрим еще один пример применения пространства имен `aop` для совета "вместо". Взамен создания другого класса для реализации интерфейса `MethodInterceptor` мы можем просто добавить новый метод к классу `MyAdvice`. Переделанный класс `MyAdvice` с новым методом `simpleAroundAdvice()` показан в листинге 7.24.

Листинг 7.24. Класс `MyAdvice` (переделанный для проверки аргумента) с методом `simpleAroundAdvice()`

```
package com.apress.prospring3.ch7.aopns;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.ProceedingJoinPoint;

public class MyAdvice {

    // Для извлечения среза и аргумента.
    public void simpleBeforeAdvice(JoinPoint joinPoint, int intValue) {

        // Выполнять, только если intValue не равно 100.
        if (intValue != 100) {
        System.out.println("Executing: " +
            joinPoint.getSignature().getDeclaringTypeName() + " "
            + joinPoint.getSignature().getName()
            + " argument: " + intValue);
        }
    }

    public Object simpleAroundAdvice(ProceedingJoinPoint pjp, int intValue)
            throws Throwable {

        // Вывести информацию перед выполнением.
        System.out.println("Before execution: " +
            pjp.getSignature().getDeclaringTypeName() + " "
            + pjp.getSignature().getName()
            + " argument: " + intValue);

        Object retVal = pjp.proceed();

        // Вывести информацию после выполнения.
        System.out.println("After execution: " +
            pjp.getSignature().getDeclaringTypeName() + " "
            + pjp.getSignature().getName() + " argument: " + intValue);

        return retVal;
    }
}
```

Вновь добавленный метод `simpleAroundAdvice()` должен получить, по крайней мере, один аргумент типа `ProceedingJoinPoint`, чтобы он смог продолжить обращением к целевому объекту. Кроме того, к нему также добавлен аргумент `intValue` для отображения его значения внутри совета.

В XML-конфигурацию потребуется добавить один дескриптор. Соответствующий фрагмент показан в листинге 7.25.

Листинг 7.25. Конфигурирование АОП в Spring с использованием пространства имен aop (совет “вместо”)

```
...
      <aop:aspect ref="advice">
         <aop:before pointcut-ref="fooExecution"
            method="simpleBeforeAdvice"/>
         <aop:around pointcut-ref="fooExecution"
            method="simpleAroundAdvice"/>
      </aop:aspect>
...
```

Как видите, мы только добавили новый дескриптор `<aop:around>` для объявления совета “вместо” и ссылки на тот же самый срез. Запуск тестовой программы на этот раз дает следующий вывод:

```
Before execution: com.apress.prospring3.ch7.aopns.MyDependency foo argument: 100
foo(int): 100
After execution: com.apress.prospring3.ch7.aopns.MyDependency foo argument: 100
Executing: com.apress.prospring3.ch7.aopns.MyDependency foo argument: 101
Before execution: com.apress.prospring3.ch7.aopns.MyDependency foo argument: 101
foo(int): 101
After execution: com.apress.prospring3.ch7.aopns.MyDependency foo argument: 101
bar()
```

В выводе есть два интересных момента. Во-первых, совет “вместо” был применен к обоим вызовам метода `foo()`, поскольку он не проверяет аргумент. Во-вторых, для вызова метода `foo()` с передачей значения 101 в качестве аргумента были выполнены оба совета, “перед” и “вместо”, причем по умолчанию совет “перед” получает преимущество.

На заметку! При использовании пространства имен `aop` или аннотаций в стиле `@AspectJ` доступны два типа совета “после”. Совет “после возврата” (дескриптор `<aop:after-returning>`) применяется, только если целевой метод завершился нормально. Другой тип — обычный совет “после” (дескриптор `<aop:after>`) — выполняется как в случае нормального завершения метода, так и в ситуации, когда в методе возникла ошибка, из-за которой было сгенерировано исключение. Если необходимо, чтобы совет применялся независимо от результата выполнения целевого метода, следует использовать обычный совет “после”.

Использование аннотаций в стиле @AspectJ

Как и можно было ожидать, в случае применения АОП в Spring с JDK 5 или последующей версии для объявления совета можно также использовать аннотации. В Spring поддерживаются аннотации в стиле `@AspectJ`, которые соответствуют аннотациям и синтаксису AspectJ. Однако, как было указано ранее, для снабжения советами целевых методов Spring по-прежнему применяет собственный механизм создания прокси, а не механизм связывания AspectJ.

В этом разделе мы посмотрим, как реализовать такие же аспекты, как построенные с помощью пространства имен `aop`, но с использованием аннотаций в стиле `@AspectJ`. В примерах данного раздела аннотации также будут применяться и для других бинов Spring.

В листингах 7.26 и 7.27 показаны классы `MyDependency` и `MyBean` с аннотациями DI из Spring.

Листинг 7.26. Класс MyDependency

```
package com.apress.prospring3.ch7.aspectjannotation;

import org.springframework.stereotype.Component;

@Component("myDependency")
public class MyDependency {

   public void foo(int intValue) {
      System.out.println("foo(int): " + intValue);
   }

   public void bar() {
      System.out.println("bar()");
   }
}
```

Листинг 7.27. Класс MyBean

```
package com.apress.prospring3.ch7.aspectjannotation;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component("myBean")
public class MyBean {

   private MyDependency myDependency;

   public void execute() {
      myDependency.foo(100);
      myDependency.foo(101);
      myDependency.bar();
   }

   @Autowired
   public void setDep(MyDependency myDependency) {
      this.myDependency = myDependency;
   }
}
```

Оба класса аннотированы с помощью аннотации `@Component` с назначением соответствующих имен. Кроме того, в классе `MyBean` метод установки свойства `myDependency` аннотирован посредством `@Autowired` для автоматического внедрения со стороны Spring.

А теперь посмотрим, как класс `MyAdvice` может использовать аннотации в стиле `@AspectJ`. Мы реализуем срезы, а также советы “перед” и “вместо”. Код класса `MyAdvice` приведен в листинге 7.28.

Листинг 7.28. Класс MyAdvice

```
package com.apress.prospring3.ch7.aspectjannotation;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;
```

```
import org.springframework.stereotype.Component;

@Component
@Aspect
public class MyAdvice {

    @Pointcut("execution(* com.apress.prospring3.ch7..foo*(int)) && args(intValue)")
    public void fooExecution(int intValue) {
    }

    @Pointcut("bean(myDependency*)")
    public void inMyDependency() {
    }

    @Before("fooExecution(intValue) && inMyDependency()")
    public void simpleBeforeAdvice(JoinPoint joinPoint, int intValue) {
        // Выполнять, только если intValue не равно 100.
        if (intValue != 100) {
        System.out.println("Executing: " +
            joinPoint.getSignature().getDeclaringTypeName() + " "
            + joinPoint.getSignature().getName() + " argument: " + intValue);
        }
    }

    @Around("fooExecution(intValue) && inMyDependency()")
    public Object simpleAroundAdvice(ProceedingJoinPoint pjp, int intValue)
            throws Throwable {

        System.out.println("Before execution: " +
            pjp.getSignature().getDeclaringTypeName() + " "
            + pjp.getSignature().getName()
            + " argument: " + intValue);

        Object retVal = pjp.proceed();

        System.out.println("After execution: " +
            pjp.getSignature().getDeclaringTypeName() + " "
            + pjp.getSignature().getName()
            + " argument: " + intValue);

        return retVal;
    }
}
```

Вы заметите, что структура кода очень похожа на ту, что использовалась в примере с пространством имен `aop`, только в этом случае применяются аннотации. Тем не менее, здесь важно отметить несколько моментов.

- Для аннотирования класса `MyAdvice` использовались аннотации `@Component` и `@Aspect`. Аннотация `@Aspect` объявляет его как класс аспекта. Чтобы позволить Spring сканировать компонент, когда в XML-конфигурации применяется дескриптор `<context:component-scan>`, класс также необходимо аннотировать с помощью `@Component`.
- Срезы определены как методы, возвращающие `void`. В классе было определено два среза; оба они аннотированы посредством `@Pointcut`. Мы намеренно разделили выражение среза в примере с пространством имен `aop` на два. Первое выражение (указанное методом `fooExecution(int intValue)`) определяет срез для выполнения методов `foo*()` с целочисленным аргументом внутри всех классов из пакета `com.apress.prospring3.ch7`, и этот аргумент (`intValue`) будет также пере-

дан в совет. Второе выражение (указанное методом `inMyDependency()`) определяет другой срез, предназначенный для выполнения всех методов из бинов Spring, имена которых имеют префикс `myDependency`. Также обратите внимание, что для представления условия "и" в выражении среза применяется `&&`, тогда как в случае пространства имен `aop` для этого использовалась операция `and`.

- Метод совета "перед" аннотирован с помощью `@Before`, а метод совета "вместо" — посредством `@Around`. Для обоих советов передается значение, которое участвует в двух срезах, определенных в классе. Конструкция `"fooExecution(intValue) && inMyDependency()"` означает, что для применения совета должны быть удовлетворены условия обоих срезов, и это то же самое, что и операция "пересечения" в `ComposablePointcut`.
- Логика советов "перед" и "вместо" совпадает с логикой этих же советов в примере с пространством имен `aop`.

При наличии всех этих аннотаций XML-конфигурация становится очень простой. В листинге 7.29 показана конфигурация для рассматриваемого примера (`aspectjannotation.xml`).

Листинг 7.29. Конфигурирование АОП в Spring с использованием аннотаций `@AspectJ`

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:aop="http://www.springframework.org/schema/aop"
   xmlns:context="http://www.springframework.org/schema/context"
   xsi:schemaLocation="http://www.springframework.org/schema/aop
     http://www.springframework.org/schema/aop/spring-aop-3.1.xsd
     http://www.springframework.org/schema/beans
     http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
     http://www.springframework.org/schema/context
     http://www.springframework.org/schema/context/spring-context-3.1.xsd">

   <aop:aspectj-autoproxy/>

   <context:component-scan base-package=
                        "com.apress.prospring3.ch7.aspectjannotation"/>
</beans>
```

Здесь объявлено только два дескриптора. Дескриптор `<aop:aspect-autorpoxy>` предназначен для сообщения Spring о том, что нужно сканировать аннотации в стиле `@AspectJ`, а дескриптор `<context:component-scan>` по-прежнему требуется Spring для сканирования бинов Spring внутри пакета, в котором находится совет. Также необходимо аннотировать класс совета с помощью аннотации `@Component`, указывая, что он является компонентом Spring.

Дескриптор `<aop:aspectj-autoproxy>` имеет атрибут `proxy-target-class`. По умолчанию он установлен в `false`, а это означает, что Spring будет создавать стандартные прокси, основанные на интерфейсах, с использованием динамического прокси JDK. В случае установки `proxy-target-class` в `true` Spring будет применять библиотеку CGLIB для создания прокси, которые основаны на классах.

В листинге 7.30 показана тестовая программа в форме класса `AspectJAnnotationExample`.

Листинг 7.30. Тестирование аннотаций AspectJ

```
package com.apress.prospring3.ch7.aspectjannotation;

import org.springframework.context.support.GenericXmlApplicationContext;

public class AspectJAnnotationExample {

   public static void main(String[] args) {

      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:aspectjannotation.xml");
      ctx.refresh();

      MyBean myBean = (MyBean) ctx.getBean("myBean");
      myBean.execute();

   }
}
```

Запуск этой программы на выполнение дает те же самые результаты, что и пример с пространством имен `aop`:

```
Before execution: com.apress.prospring3.ch7.aspectjannotation.MyDependency
↳foo argument: 100
foo(int): 100
After execution: com.apress.prospring3.ch7.aspectjannotation.
MyDependency foo argument: 100
Executing: com.apress.prospring3.ch7.aspectjannotation.MyDependency
↳foo argument: 101
Before execution: com.apress.prospring3.ch7.aspectjannotation.MyDependency
↳foo argument: 101
foo(int): 101
After execution: com.apress.prospring3.ch7.aspectjannotation.MyDependency
↳foo argument: 101
bar()
```

Соображения по поводу декларативного конфигурирования АОП в Spring

К этому моменту мы обсудили три различных способа определения конфигурации АОП в Spring, включая класс `ProxyFactoryBean`, пространство имен `aop` и аннотации в стиле `@AspectJ`. Мы уверены, вы согласитесь с тем, что применять пространство имен `aop` намного проще, чем класс `ProxyFactoryBean`. Итак, остается главный вопрос: что использовать — пространство имен `aop` или аннотации в стиле `@AspectJ`?

Если приложение Spring основано на XML-конфигурации, применение пространства имен `aop` является естественным выбором, поскольку оно обеспечит согласованность стилей конфигураций АОП и DI. С другой стороны, если приложение в основном базируется на аннотациях, следует использовать аннотации `@AspectJ`. Подход с аннотациями `@AspectJ` также обладает тем преимуществом, что вся связанная с аспектом информация инкапсулирована в одном модуле, поэтому ею легко управлять.

Кроме того, существуют и другие отличия между подходами с пространством имен `aop` и аннотациями `@AspectJ`.

- Синтаксис для выражений срезов имеет небольшие отличия (например, как упоминалось ранее, необходимо применять `and` при подходе с пространством имен `aop`, но `&&` — при подходе с аннотациями `@AspectJ`).

- Подход с пространством имен `aop` поддерживает только модель создания одиночных экземпляров аспектов.
- При подходе с пространством имен `aop` нельзя "комбинировать" множество выражений срезов. Например, в рассмотренном ранее примере использования аннотаций `@AspectJ` мы скомбинировали два определения срезов (т.е. `fooExecution(intValue) && inMyDependency()`) в советах "перед" и "вместо". Однако поступить так в случае применения пространства имен `aop` не получится, поэтому придется создать новое выражение среза, комбинирующее условия сопоставления.

На заметку! В документации по Spring вы также обнаружите упоминание еще одного механизма, который называется *автоматическим созданием прокси* (automatic proxying). В сущности, он предоставляет несколько способов определения и применения совета к множеству бинов Spring в рамках `ApplicationContext`. Например, класс `BeanNameAutoProxyCreator` позволяет снабдить советом несколько бинов Spring по имени, а класс `DefaultAdvisorAutoProxyCreator` дает возможность применить аспект к бинам внутри `ApplicationContext` с логикой среза, определенной в классе реализации `Advisor`. Тем не менее, оба механизма теперь могут поддерживаться мощными средствами выражений для срезов как при использовании пространства имен `aop`, так и в случае применения аннотаций в стиле `@AspectJ`. В связи с этим мы решили не обсуждать в книге автоматическое создание прокси, но если вам интересно, то в документации по Spring вы найдете подробные объяснения.

Интеграция с AspectJ

АОП предлагает мощное решение для многих общих задач, которые возникают при объектно-ориентированной разработке приложений. При использовании АОП в Spring в вашем распоряжении имеется подмножество функциональности АОП, с помощью которого в большинстве случаев можно решить задачи, возникающие во время построения приложений. Однако в ряде ситуаций могут понадобиться средства АОП, которые выходят за рамки доступных в реализации АОП из Spring.

С позиции точки соединения АОП в Spring поддерживает только срезы, соответствующие выполнению открытых нестатических методов. Тем не менее, в некоторых случаях совет необходимо применять к защищенным/закрытым методам — во время создания объекта или доступа к полям и т.д.

В таких ситуациях следует обратиться к реализации АОП с более полным набором функциональных возможностей. Мы предпочитаем использовать AspectJ, и поскольку теперь можно конфигурировать аспекты AspectJ с помощью Spring, AspectJ становится великолепным дополнением АОП в Spring.

Что собой представляет AspectJ

AspectJ — это полнофункциональная реализация АОП, которая использует процесс связывания (либо во время компиляции, либо во время выполнения) для введения аспектов в код. В AspectJ аспекты и срезы построены с применением Java-подобного синтаксиса, который упрощает изучение разработчикам на Java. Мы не собираемся уделять слишком много времени на описание работы AspectJ, потому что это выходит за рамки контекста настоящей книги. Взамен мы представим ряд простых примеров AspectJ и покажем, как их сконфигурировать в Spring. За дополнительными сведениями по AspectJ обращайтесь к книге *AspectJ in Action: Enterprise AOP with Spring Applications*, 2-е издание (Manning, 2010 г.).

На заметку! Мы не будет описывать, каким образом связать аспекты AspectJ с приложением. За соответствующими деталями обращайтесь к документации по AspectJ. В качестве альтернативы пользователи Eclipse могут загрузить инструменты разработки AspectJ для Eclipse (Eclipse AspectJ Development Tools — AJDT) и получить в свое распоряжение полную интеграцию с IDE-средой и автоматическую компиляцию. AJDT также входит в состав набора инструментов SpringSource (SpringSource Tool Suite — STS), поэтому в случае использования STS отдельно загружать его не понадобится.

Использование одиночных экземпляров аспектов

По умолчанию аспекты AspectJ являются одиночными экземплярами в том смысле, что вы получаете единственный экземпляр на загрузчик класса. Проблема, возникающая в платформе Spring с каждым аспектом AspectJ, состоит в том, что она не может создать экземпляр аспекта, поскольку это обрабатывается самой инфраструктурой AspectJ. Однако в каждом аспекте доступен метод `aspectOf()` (в классе `org.aspectj.lang.Aspects` внутри JAR-файла `aspectjrt`), который можно использовать для доступа к экземпляру аспекта. Применяя метод `aspectOf()` и специальную возможность конфигурирования Spring, вы можете поручить конфигурирование аспекта платформе Spring. Преимущества этой возможности трудно переоценить. Вы получаете полную отдачу от мощного набора функциональных средств АОП, предлагаемого AspectJ, не теряя великолепных возможностей DI и конфигурирования, которые доступны в Spring. Это также означает, что в приложении вам не понадобятся два отдельных метода конфигурирования; один и тот же подход с `ApplicationContext` можно использовать для всех бинов, управляемых Spring, и для аспектов AspectJ.

Как показано в следующем примере, с конфигурированием аспектов AspectJ при помощи Spring не связано ничего специфического или сложного. В листинге 7.31 приведен код базового класса `MessageWriter`, который будет снабжен советом с применением AspectJ.

Листинг 7.31. Класс `MessageWriter`

```
package com.apress.prospring3.ch7.aspectj;

public class MessageWriter {

   public void writeMessage() {
      System.out.println("foobar!");
   }
   public void foo() {
      System.out.println("foo");
   }
}
```

В этом примере мы собираемся использовать AspectJ для снабжения советом метода `writeMessage()` и вывода сообщений до и после вызова метода. Выводимые сообщения будут иметь возможность конфигурирования с помощью Spring.

В состав STS входит AspectJ Development Tools, который представляет собой подключаемый модуль Eclipse для разработки с применением AspectJ. Мы будем его использовать для построения аспекта AspectJ в рассматриваемом примере. Чтобы включить поддержку AspectJ для проекта, щелкните правой кнопкой мыши на проекте и выберите в контекстном меню пункт Configure⇨Convert to AspectJ Project (Конфигурировать⇨Преобразовать в проект AspectJ). Набор меню STS показан на рис. 7.5.

Рис. 7.5. Включение поддержки AspectJ в STS

После этого можно создать аспект, щелкнув правой кнопкой мыши на имени пакета `com.apress.prospring3.ch7.aspectj` и выбрав в контекстном меню пункт New⇨Aspect (Создать⇨Аспект), как показано на рис. 7.6.

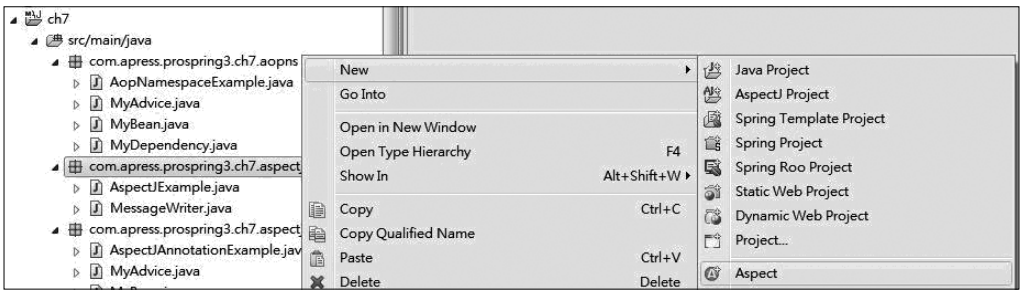


Рис. 7.6. Создание нового аспекта

Однако если скомпилировать проект прямо сейчас, возникнут ошибки, связанные с классом `MyAdvice` из листинга 7.28. Вкладка с сообщениями об ошибках в STS показана на рис. 7.7.

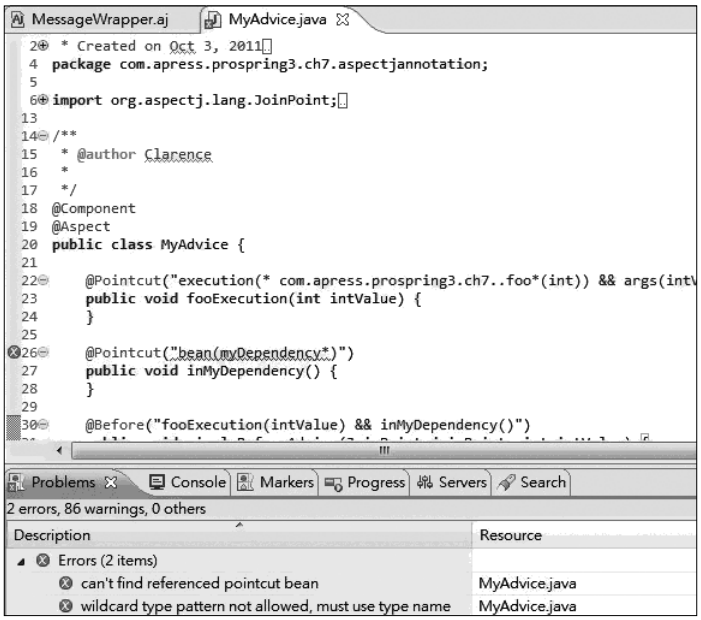


Рис. 7.7. Ошибки, связанные с классом `MyAdvice`

Ошибка заключается в том, что в примере применения аннотаций `@AspectJ` мы использовали спецификатор `bean()` среза, который AspectJ не поддерживает. Спецификатор `bean()` среза — это расширение Spring, добавленное в AspectJ, и он предназначен для поддержки предоставления имени бина как среза для аннотации в стиле `@AspectJ`. Таким образом, когда поддержка AspectJ в проекте включена, инструмент AspectJ сообщит об ошибке. Для нормального сосуществования инструмента AspectJ и спецификатора `bean()` среза имеются два сценария.

- В случае использования спецификатора `bean()` среза применяйте пространство имен `aop` в XML-конфигурации, а не аннотации в стиле `@AspectJ`.
- Не используйте спецификатор `bean()` среза, если хотите работать в проекте с AspectJ.

Чтобы избавиться от ошибки, пока просто удалите класс `MyAdvice`.

В листинге 7.32 показан аспект `MessageWrapper` (имя файла выглядит как `MessageWrapper.aj`, т.е. это файл AspectJ, а не стандартный Java-класс). Этот аспект создан с помощью пункта меню New⇨Aspect, как было показано на рис. 7.6.

Листинг 7.32. Аспект `MessageWrapper`

```
package com.apress.prospring3.ch7.aspectj;

public aspect MessageWrapper {
   private String prefix;
   private String suffix;
   public void setPrefix(String prefix) {
      this.prefix = prefix;
   }
   public void setSuffix(String suffix) {
      this.suffix = suffix;
   }
```

```
    pointcut doWriting() :
        execution(*
            com.apress.prospring3.ch7.aspectj.MessageWriter.writeMessage());

    before() : doWriting() {
        System.out.println(prefix);
    }

    after() : doWriting() {
        System.out.println(suffix);
    }
}
```

Большая часть кода должна выглядеть знакомой. Здесь мы создаем аспект по имени `MessageWrapper` и, как нормальному Java-классу, предоставляем ему два свойства, `suffix` и `prefix`, которые будут использоваться при снабжении советом метода `writeMessage()`. Затем мы определяем именованный срез `doWriting()` для единственной точки соединения, которой в этом случае является выполнение метода `writeMessage()`. (AspectJ поддерживает большое количество точек соединения, но перечисление их всех не входит в задачи данного примера.) Наконец, мы определяем советы двух видов: один выполняется перед срезом `doWriting()`, а другой — после него. Совет "перед" выводит строку, содержащую значение `prefix`, а совет "после" — строку, содержащую значение `suffix`. В листинге 7.33 показано, как этот аспект конфигурируется в Spring (`aspectj.xml`).

Листинг 7.33. Конфигурирование аспекта AspectJ

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd">

    <bean id="aspect" class="com.apress.prospring3.ch7.aspectj.MessageWrapper"
        factory-method="aspectOf">
        <property name="prefix">
            <value>Ha Ha!</value>
        </property>
        <property name="suffix">
            <value>Ho Ho!</value>
        </property>
    </bean>
</beans>
```

Как видите, большая часть конфигурации бина аспекта очень похожа на конфигурацию стандартного бина. Единственное отличие заключается в использовании атрибута `factory-method` дескриптора `<bean>`. Этот атрибут позволяет классам, следующим традиционному шаблону Factory (Фабрика), плавно интегрироваться в Spring. Например, если есть класс `Foo` с закрытым конструктором и статическим методом фабрики `getInstance()`, то применение атрибута `factory-method` позволяет бину этого класса быть управляемым со стороны Spring. Метод `aspectOf()`, имеющийся в каждом аспекте AspectJ, предоставляет доступ к экземпляру аспекта, позволяя Spring устанавливать свойства этого аспекта. В листинге 7.34 приведен код простого управляющего приложения для рассматриваемого примера.

Листинг 7.34. Конфигурация AspectJ в действии

```
package com.apress.prospring3.ch7.aspectj;

import org.springframework.context.support.GenericXmlApplicationContext;

public class AspectJExample {

   public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:aspectj.xml");
      ctx.refresh();

      MessageWriter writer = new MessageWriter();
      writer.writeMessage();
      writer.foo();
   }
}
```

Обратите внимание, что сначала мы загружаем `ApplicationContext`, чтобы дать возможность Spring сконфигурировать аспект. Затем мы создаем экземпляр `MessageWriter` и вызываем методы `writeMessage()` и `foo()`. Запуск этого примера на выполнение дает следующий вывод:

```
Ha Ha!
foobar!
Ho Ho!
foo
```

Здесь видно, что совет в аспекте `MessageWrapper` был применен к методу `writeMessage()`, а значения свойств `prefix` и `suffix`, указанные в конфигурации `ApplicationContext`, использовались советом при выводе сообщений до и после вызова метода.

АОП в примере приложения

До сих пор вы видели множество небольших примеров, демонстрирующих применение функциональных возможностей АОП в Spring, но пока еще не встречались с более реальными сценариями использования АОП в каком-нибудь приложении. К типичным сценариям в этой области относятся регистрация в журнале и безопасность, и мы кратко взглянули на них в этой и предшествующей главах. Тем не менее, применение АОП не ограничивается лишь регистрацией и безопасностью; оно может принести большую пользу при реализации любой специфичной для приложения логики, которая является сквозной — т.е. любой логики в приложении, которая должна вызываться большим числом отдельных компонент. В этом разделе мы дадим обзор того, как АОП в Spring используется в примере приложения SpringBlog для решения задачи, связанной со сквозной функциональностью.

Фильтрация нежелательного контента в SpringBlog

Одной из задач, с которыми мы столкнулись при построении приложения SpringBlog, была необходимость унифицированной фильтрации нежелательного контента из сообщений, отправляемых в блог. Фильтрация должна применяться как к записям верхнего уровня в блоге, так и к любым комментариям, которые добавляются к отдельной записи. Мы хотим гарантировать, что создаваемые записи или комментарии не могут содержать нежелательный контент, и такой контент не может быть внесен при последующем

редактировании любой записи или комментария. В частности, нам нужна возможность автоматической замены ненормативной лексики в сообщениях, отправляемых в блог, неоскорбительными альтернативами. Развивая эту мысль дальше, мы решили, что в некоторых случаях владелец блога должен иметь возможность добавления ненормативной лексики к своим записям, но читатели блога ограничены в отправке комментариев, содержащих такую лексику.

Традиционный подход к решению описанной задачи предусматривает определение интерфейса (скажем, `ObscenityFilter`), затем построение реализации этого интерфейса и обеспечение доступа к ней через какой-нибудь фабричный класс. После этого в каждом методе, где запись либо комментарий создается или модифицируется, необходимо обращаться к `ObscenityFilter` с целью удаления ненормативной лексики из отправляемого сообщения. Однако основная проблема с этим подходом состоит в том, что необходимо не забыть реализовать данную проверку внутри всей бизнес-логики, так или иначе связанной с обработкой записей и комментариев блога.

Используя АОП в Spring, мы можем построить намного более элегантное решение этой задачи, выделив проверку на предмет нежелательного контента в совет "перед", который будет применяться к любому методу, принимающему объект записи блога или комментария в качестве аргумента. Интересной особенностью этой реализации является то, что, в сущности, мы просто следуем рекомендуемой практике ООП, как это предлагается при традиционном подходе. Мы определяем интерфейс `ObscenityFilter` и затем строим его реализацию. Благодаря механизму внедрения зависимостей в Spring, мы можем избежать необходимости в создании фабричного класса, но за счет следования рекомендациям мы смогли построить фильтр нежелательного контента, который одинаково хорошо используется в условиях с и без АОП.

Интерфейс `BlogPosting`

В рамках блога существуют два разных типа отправляемых сообщений: основная запись блога, представленная объектом `Entry`, и комментарий к записи, представленный объектом `Comment`. Несмотря на представление разных видов сообщений, эти два объекта разделяют похожие характеристики, такие как тело, тема, вложения и дата отправки сообщения. В связи с этим мы создали интерфейс по имени `BlogPosting`, который позволяет манипулировать объектами `Comment` и `Entrie` одновременно. Поскольку все свойства с типами `String` объектов `Comment` и `Entry` доступны через интерфейс `BlogPosting`, мы используем этот интерфейс в совете фильтра нежелательного контента.

Реализация интерфейса `ObscenityFilter`

В приложении SpringBlog мы решили создать реализацию интерфейса `ObscenityFilter`, которая позволяет указывать список фильтруемой ненормативной лексики в виде `List` и заменять эту ненормативную лексику с применением алгоритма ROT13. Имея готовую реализацию интерфейса `ObscenityFilter` (класс `ListBasedObscenityFilter`), мы создаем совет (`ObscenityFilterAdvice`), который позволит применять средства фильтрации нежелательного контента декларативно к любому методу, принимающему аргументы `Entry` или `Comment`. В основе `ObscenityFilterAdvice` лежит изменение аргументов, переданных методу, при котором любые свойства `String` объектов `Entry` или `Comment` заменяются неоскорбительными альтернативами. Поскольку нам требуется только просматривать аргументы, передаваемые методу, и, возможно, модифицировать их, для этого идеально подойдет совет "перед". Кроме того, для реализации этого совета мы решили выбрать подход с аннотациями в стиле `@AspectJ`.

Резюме по фильтру нежелательного контента

Пример, приведенный в настоящем разделе, показал, что АОП имеет множество практических применений в реальном приложении. Мы обнаружили, что за счет использования АОП для создания фильтра нежелательного контента мы смогли сохранить код на уровне обслуживания приложения SpringBlog более ясным, а также сумели сократить объем дублированного кода в рамках всего приложения. При построении собственного приложения с помощью Spring полезно уделить время на идентификацию сквозной функциональности. Сделав это, определите интерфейсы для взаимодействия с ней, постройте реализации и затем вместо применения фабрики и встраивания обращений к этим интерфейсам по всему коду воспользуйтесь АОП в Spring для связывания сквозной функциональности с приложением прозрачным образом. Фильтр нежелательного контента и его исходный код детально рассматриваются в главе 21.

Резюме

В этой главе мы завершили обсуждение АОП. Мы продемонстрировали дополнительные возможности для создания срезов и показали, как расширять набор интерфейсов, реализуемых объектом, с использованием механизма введения. Основное внимание в этой главе было сосредоточено на применении служб платформы Spring для декларативного конфигурирования АОП, позволяющего избежать жесткого кодирования логики конструирования прокси АОП. Также мы кратко рассмотрели интеграцию Spring и AspectJ, которая дает возможность получить всю мощь AspectJ, не теряя преимуществ гибкости Spring. В следующей главе мы переходим к совершенно другой теме — как использовать поддержку JDBC, предлагаемую Spring, для радикального упрощения процесса кодирования доступа к данным на основе JDBC.

ГЛАВА 8

Поддержка JDBC в Spring

В предшествующих главах вы видели, насколько просто строить приложение, полностью управляемое Spring. Вы получили хорошее представление о конфигурации бинов и об аспектно-ориентированном программировании (АОП) — другими словами, вы знаете, как увязывать отдельные части приложения в одно целое с использованием Spring. Тем не менее, отсутствует еще одна часть головоломки: как получить данные, которые управляют приложением?

Кроме простейших одноразовых утилит командной строки, почти каждое приложение нуждается в сохранении данных в каком-нибудь хранилище. Наиболее часто используемым и удобным хранилищем является реляционная база данных.

Самыми примечательными системами управления реляционными базами данных (СУРБД) с открытым кодом, пожалуй, можно считать MySQL (`www.mysql.com`) и PostgreSQL (`www.postgresql.org`). С точки зрения предоставляемых ими возможностей СУРБД они примерно одинаковы. Как правило, MySQL более широко используется при разработке веб-приложений, особенно на платформе Linux. С другой стороны, система PostgreSQL является более дружественной для разработчиков под Oracle, поскольку ее процедурный язык PL/pgSQL очень близок к языку PL/SQL в Oracle.

Даже в случае выбора самой быстрой и надежной СУРБД приложение может потерять в скорости и гибкости, если в нем присутствует плохо спроектированный и реализованный уровень доступа к данным. Приложения, как правило, работают с уровнем доступа к данным весьма интенсивно, в результате чего любые узкие места в коде доступа к данным влияют на все приложение независимо от того, насколько хорошо спроектированы остальные уровни.

В этой главе мы покажем, как с помощью Spring упростить реализацию кода доступа к данным, используя JDBC. Мы начнем с рассмотрения того ужасающего объема кода, который пришлось бы писать, не имея Spring, и затем сравним его с классом доступа к данным, реализованным с применением соответствующих возможностей Spring. Результат действительно впечатляет: Spring позволяет использовать всю мощь настроенных человеком SQL-запросов, сводя к минимуму объем кода поддержки, который придется написать. В частности, в главе будут рассматриваться следующие темы.

- *Сравнение традиционного кода JDBC и поддержки JDBC в Spring.* Мы исследуем, каким образом Spring упрощает код JDBC старого стиля, в то же время обеспечивая аналогичную функциональность. Вы также увидите, как Spring обращается к низкоуровневому API-интерфейсу JDBC и как этот API-интерфейс отображается на удобные классы вроде `JdbcTemplate`.

- *Подключение к базе данных.* Хотя мы не собираемся вникать в каждую мелкую деталь управления подключениями к базам данных, мы продемонстрируем фундаментальные отличия между `Connection` и `DataSource`. Естественно, мы обсудим, каким образом Spring управляет источниками данных (`DataSource`), и какие источники данных можно использовать в приложениях.
- *Извлечение и отображение данных на Java-объекты.* Мы покажем, как извлекать данные и затем эффективно отображать выбранные данные на Java-объекты. Вы также узнаете, что Spring JDBC является жизнеспособной альтернативой инструментам объектно-реляционного отображения (object-relational mapping — ORM).
- *Вставка, обновление и удаление данных.* Наконец, мы обсудим, как реализовать операции вставки, обновления и удаления таким образом, чтобы любые изменения в используемой базе данных не оказывали разрушительного влияния на написанный код.

Что такое база данных?

Разработчики иногда с трудом описывают такое понятие, как база данных. В одном случае база данных представляет действительные данные, а в других — программное обеспечение (ПО), управляющее данными, экземпляр процесса этого ПО или даже физическую машину, на которой выполняется этот управляющий процесс. Формально база данных — это коллекция данных; ПО базы данных (такое как Oracle, PostgreSQL, MySQL и т.д.) называется *программным обеспечением управления базами данных* или, более точно, *системой управления реляционными базами данных (СУРБД)*; экземпляр СУРБД называется *механизмом баз данных*; и, наконец, машина, на которой выполняется механизм баз данных, называется *сервером баз данных*. Тем не менее, большинство разработчиков сходу понимают значение термина база данных из контекста, в котором он употребляется. Именно поэтому мы будем использовать этот термин в дальнейшем для представления всех четырех описанных значений.

Из-за бурного роста Интернета и технологий облачных вычислений в последние несколько лет, появилось множество специализированных веб-приложений, реализующих такую функциональность, как социальные сети, поисковые механизмы, карты, видео и т.п. Для обслуживания специальных требований доступа к данным, которые предъявляются этими приложениями, также был разработан ряд дополнительных категорий “баз данных”. В число их примеров входят базы данных пар “ключ-значение” (в общем случае называемые базами данных NoSQL), базы данных графики, базы данных документов и т.д. Таким образом, *база данных* теперь является намного более широким термином. Однако обсуждение таких нереляционных баз данных выходит за рамки настоящей книги, поэтому везде, где упоминаются базы данных, мы имеем в виду СУРБД.

Модель данных для кода примеров

Перед тем, как продолжить, мы должны представить простую модель данных, которая будет использоваться во всех примерах этой главы, а также в нескольких последующих главах при обсуждении других технологий доступа к данным (мы будем соответствующим образом расширять модель для удовлетворения потребностей, связанных с каждой такой технологией).

Модель представляет очень простую базу данных контактов. В ней имеются две таблицы. Первая — таблица `CONTACT`, которая хранит контактную информацию о персоне, а вторая — таблица `CONTACT_TEL_DETAIL`, хранящая подробности о телефонах контакта. Каждый контакт может иметь ноль и более телефонных номеров; другими словами, между таблицами `CONTACT` и `CONTACT_TEL_DETAIL` существует отношение “один ко многим”. Информация о контакте включает имя, фамилию и дату рождения, а сведения

о телефонах — тип телефона (мобильный, домашний и т.д.) и соответствующий телефонный номер. На рис. 8.1 показана диаграмма "сущность-отношение" (entity-relationship — ER) для этой базы данных.

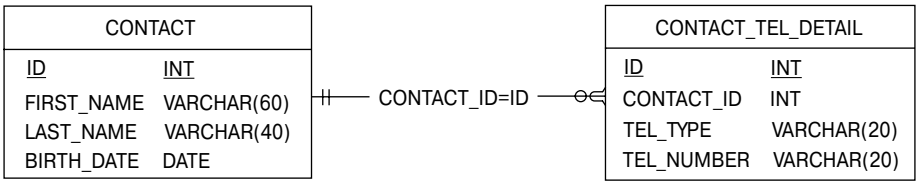


Рис. 8.1. Простая модель данных для кода примеров

Как видите, обе таблицы содержат столбец `ID`, значение которого автоматически устанавливается базой данных во время вставки. В таблице `CONTACT_TEL_DETAIL` имеется отношение внешнего ключа с таблицей `CONTACT`, которое поддерживает связь по столбцу `CONTACT_ID` с первичным ключом таблицы `CONTACT` (т.е. столбцом `ID`).

На заметку! Рассматриваемая модель данных была создана с использованием подключаемого модуля для Eclipse под названием Clay Mark II. Нелицензированной версией можно пользоваться свободно для создания моделей данных, предназначенных для бесплатных баз данных с открытым кодом, в числе которых MySQL, PostgreSQL, HSQL, Derby и т.д. Чтобы запустить код примера, этот подключаемый модуль не нужен, поскольку в состав кода входят сценарии создания необходимых таблиц. Тем не менее, файл с диаграммой модели (`ch8/data-model/prospring3-ch8-datamodel.clay`) также включен в код примера, поэтому заинтересованные читатели могут установить подключаемый модуль Clay Mark II и просмотреть диаграмму (детальные сведения о модуле доступны на веб-сайте `www.azzurri.co.jp`).

В листинге 8.1 показан код сценария для создания базы данных (он является совместимым с MySQL).

Листинг 8.1. Сценарий для создания простой модели данных (`schema.sql`)

```
CREATE TABLE CONTACT (
      ID INT NOT NULL AUTO_INCREMENT
    , FIRST_NAME VARCHAR(60) NOT NULL
    , LAST_NAME VARCHAR(40) NOT NULL
    , BIRTH_DATE DATE
    , UNIQUE UQ_CONTACT_1 (FIRST_NAME, LAST_NAME)
    , PRIMARY KEY (ID)
);
CREATE TABLE CONTACT_TEL_DETAIL (
      ID INT NOT NULL AUTO_INCREMENT
    , CONTACT_ID INT NOT NULL
    , TEL_TYPE VARCHAR(20) NOT NULL
    , TEL_NUMBER VARCHAR(20) NOT NULL
    , UNIQUE UQ_CONTACT_TEL_DETAIL_1 (CONTACT_ID, TEL_TYPE)
    , PRIMARY KEY (ID)
    , CONSTRAINT FK_CONTACT_TEL_DETAIL_1 FOREIGN KEY (CONTACT_ID)
              REFERENCES CONTACT (ID)
);
```

В листинге 8.2 представлен код сценария для наполнения таблиц `CONTACT` и `CONTACT_TEL_DETAIL` некоторыми тестовыми данными.

Листинг 8.2. Сценарий для наполнения данными (test-data.sql)

```
insert into contact (first_name, last_name, birth_date)
       values ('Clarence', 'Ho', '1980-07-30');
insert into contact (first_name, last_name, birth_date)
       values ('Scott', 'Tiger', '1990-11-02');
insert into contact (first_name, last_name, birth_date)
       values ('John', 'Smith', '1964-02-28');
insert into contact_tel_detail (contact_id, tel_type, tel_number)
       values (1, 'Mobile', '1234567890');
insert into contact_tel_detail (contact_id, tel_type, tel_number)
       values (1, 'Home', '1234567890');
insert into contact_tel_detail (contact_id, tel_type, tel_number)
       values (2, 'Home', '1234567890');
```

В последующих разделах этой главы будут приведены примеры извлечения данных из базы данных через JDBC и отображения результирующего набора прямо на Java-объекты (т.е. объекты POJO). В листингах 8.3 и 8.4 показаны классы предметной области `Contact` и `ContactTelDetail`.

Листинг 8.3. Класс предметной области Contact

```
package com.apress.prospring3.ch8.domain;
import java.io.Serializable;
import java.sql.Date;
import java.util.List;
public class Contact implements Serializable {
   private Long id;
   private String firstName;
   private String lastName;
   private Date birthDate;
   private List<ContactTelDetail> contactTelDetails;

   // Методы извлечения и установки опущены.
   public String toString() {
      return "Contact - Id: " + id + ", First name: " + firstName
         + ", Last name: " + lastName + ", Birthday: " + birthDate;
   }
}
```

Листинг 8.4. Класс предметной области ContactTelDetail

```
package com.apress.prospring3.ch8.domain;
import java.io.Serializable;
public class ContactTelDetail implements Serializable {
   private Long id;
   private Long contactId;
   private String telType;
   private String telNumber;

   // Методы извлечения и установки опущены.
   public String toString() {
      return "Contact Tel Detail - Id: " + id + ", Contact id: " + contactId
         + ", Type: " + telType + ", Number: " + telNumber;
   }
}
```

Давайте начнем с очень простого интерфейса для ContactDao, который инкапсулирует все службы доступа к данным для информации о контакте. В листинге 8.5 показан этот интерфейс.

Листинг 8.5. Интерфейс ContactDao

```
package com.apress.prospring3.ch8.dao;

import java.util.List;

import com.apress.prospring3.ch8.domain.Contact;

public interface ContactDao {

    public List<Contact> findAll();

    public List<Contact> findByFirstName(String firstName);

    public void insert(Contact contact);

    public void update(Contact contact);

    public void delete(Long contactId);
}
```

В интерфейсе ContactDao определены два метода поиска, а также методы вставки, обновления и удаления. Все вместе они объединяются в термин CRUD (Create, Read, Update, Delete — создание, чтение, обновление и удаление).

Наконец, чтобы упростить тестирование, мы модифицируем свойства log4j, указав DEBUG в качестве уровня регистрации в журнале для всех классов. При уровне DEBUG модуль JDBC в Spring будет выводить все лежащие в основе SQL-операторы, выполняемые в базе данных, поэтому вы будете знать, что точно происходит; это особенно удобно во время поиска и устранения синтаксических ошибок в SQL-операторах. В листинге 8.6 приведено содержимое файла log4j.properties (находящегося в папке /src/main/resources файлов исходного кода проекта для главы 8) с включенным уровнем DEBUG.

Листинг 8.6. Файл log4j.properties

```
log4j.rootCategory=DEBUG, stdout

log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE} %5p %40.40c:%4L - %m%n
```

На заметку! В среде STS после создания шаблонного проекта Spring файл log4j.properties сгенерируется в папке src/test/resources. Вы можете просто переместить его в папку src/main/resources и соответствующим образом модифицировать или же удалить этот файл в src/test/resources и затем создать новый файл log4j.properties в папке /src/main/resources.

Исследование инфраструктуры JDBC

JDBC предоставляет Java-приложениям стандартный способ доступа к данным, хранящимся в базе данных. В основе инфраструктуры JDBC лежит драйвер, специфичный для каждой базы данных; именно этот драйвер позволяет Java-коду получать доступ к базе данных.

После загрузки драйвер регистрирует себя с помощью класса `java.sql.DriverManager`. Этот класс управляет списком драйверов и предоставляет статические методы для установления подключений к базе данных. Метод `getConnection()` класса `DriverManager` возвращает интерфейс `java.sql.Connection`, реализованный драйвером. Этот интерфейс позволяет запускать SQL-операторы в базе данных.

Инфраструктура JDBC сложна и хорошо протестирована; тем не менее, эта сложность затрудняет разработку. Первый уровень сложности связан с обеспечением в коде средств управления подключениями к базе данных. Подключение является дефицитным ресурсом, который к тому же весьма накладен в установлении. Обычно база данных создает поток или порождает дочерний процесс для каждого подключения. Однако количество параллельных подключений, как правило, ограничено, и большое число открытых подключений замедляет работу базы данных.

Мы покажем, как Spring помогает справиться с этой сложностью, но сначала нужно посмотреть, каким образом выбираются, удаляются и обновляются данные в чистом JDBC.

Давайте создадим простую форму реализации интерфейса `ContactDao` для взаимодействия с базой данных через чистый JDBC. Памятуя о том, что нам уже известно о подключениях к базе данных, мы возьмем всю ответственность на себя и применим дорогой (в смысле производительности) подход с созданием подключения для каждого оператора. Это приведет к значительному снижению производительности Java и добавит дополнительную нагрузку к базе данных, поскольку подключение должно устанавливаться для каждого запроса. Однако если оставлять подключение открытым, сервер базы данных может даже прекратить работу. В листинге 8.7 приведен код, необходимый для управления подключением JDBC, в котором в качестве примера используется MySQL.

Листинг 8.7. Управление подключением JDBC

```
public class PlainContactDao implements ContactDao {

   static {
      try {
         Class.forName("com.mysql.jdbc.Driver");
      } catch (ClassNotFoundException ex) {
         // обработка отсутствует
      }
   }

   private Connection getConnection() throws SQLException {
      return DriverManager.getConnection(
         "jdbc:mysql://localhost:3306/prospring3_ch8",
         "prospring3", "prospring3");
   }

   private void closeConnection(Connection connection) {
      if (connection == null) return;

      try {
         connection.close();
      } catch (SQLException ex) {
         // обработка отсутствует
      }
   }

   ...
```

Хотя этот код далек от завершения, он дает представление о том, какие действия должны предприниматься для управления подключением JDBC. Код даже не работает с пулом подключений, который является распространенным способом более эффективного управления подключениями к базе данных. Мы не будем обсуждать организацию пула подключений в этот момент (это будет сделано в разделе "Подключения к базе данных и источники данных" далее в главе), а взамен покажем в листинге 8.8 реализацию методов `findAll()`, `insert()` и `delete()` интерфейса `ContactDao` с использованием простого JDBC.

Листинг 8.8. Реализация DAO с использованием простого JDBC

```
package com.apress.prospring3.ch8.dao.plain;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.List;

import com.apress.prospring3.ch8.dao.ContactDao;
import com.apress.prospring3.ch8.domain.Contact;

public class PlainContactDao implements ContactDao {
   static {
      try {
         Class.forName("com.mysql.jdbc.Driver");
      } catch (ClassNotFoundException ex) {
         // обработка отсутствует
      }
   }

   private Connection getConnection() throws SQLException {
      return DriverManager.getConnection(
         "jdbc:mysql://localhost:3306/prospring3_ch8",
         "prospring3", "prospring3");
   }

   private void closeConnection(Connection connection) {
      if (connection == null) return;

      try {
         connection.close();
      } catch (SQLException ex) {
         // обработка отсутствует
      }
   }

   public List<Contact> findAll() {
      List<Contact> result = new ArrayList<Contact>();
      Connection connection = null;
      try {
         connection = getConnection();
         PreparedStatement statement =
            connection.prepareStatement("select * from contact");
         ResultSet resultSet = statement.executeQuery();
         while (resultSet.next()) {
            Contact contact = new Contact();
```

```
                contact.setId(resultSet.getLong("id"));
                contact.setFirstName(resultSet.getString("first_name"));
                contact.setLastName(resultSet.getString("last_name"));
                contact.setBirthDate(resultSet.getDate("birth_date"));
                result.add(contact);
            }
        } catch (SQLException ex) {
            ex.printStackTrace();
        } finally {
            closeConnection(connection);
        }
        return result;
    }

    public List<Contact> findByFirstName(String firstName) {
        return null;
    }

    public void insert(Contact contact) {
        Connection connection = null;
        try {
            connection = getConnection();
            PreparedStatement statement = connection.prepareStatement(
"insert into Contact (first_name, last_name, birth_date) values (?, ?, ?)"
, Statement.RETURN_GENERATED_KEYS);
            statement.setString(1, contact.getFirstName());
            statement.setString(2, contact.getLastName());
            statement.setDate(3, contact.getBirthDate());
            statement.execute();

            ResultSet generatedKeys = statement.getGeneratedKeys();
            if (generatedKeys.next()) {
                contact.setId(generatedKeys.getLong(1));
            }
        } catch (SQLException ex) {
            ex.printStackTrace();
        } finally {
            closeConnection(connection);
        }
    }

    public void update(Contact contact) {
    }

    public void delete(Long contactId) {
        Connection connection = null;
        try {
            connection = getConnection();
            PreparedStatement statement =
                connection.prepareStatement("delete from contact where id=?");
            statement.setLong(1, contactId);
            statement.execute();
        } catch (SQLException ex) {
            ex.printStackTrace();
        } finally {
            closeConnection(connection);
        }
    }
}
```

В листинге 8.9 приведена главная тестовая программа, демонстрирующая показанную выше реализацию DAO в действии.

Листинг 8.9. Тестирование реализации, использующей чистый JDBC

```
package com.apress.prospring3.ch8;

import java.sql.Date;
import java.util.GregorianCalendar;
import java.util.List;

import com.apress.prospring3.ch8.dao.ContactDao;
import com.apress.prospring3.ch8.dao.plain.PlainContactDao;
import com.apress.prospring3.ch8.domain.Contact;

public class PlainJdbcSample {

   private static ContactDao contactDao = new PlainContactDao();

   public static void main(String[] args) {

      // Вывести список всех контактов.
      System.out.println("Listing initial contact data:");
      listAllContacts();

      System.out.println();

      // Вставить новый контакт.
      System.out.println("Insert a new contact");
      Contact contact = new Contact();
      contact.setFirstName("Jacky");
      contact.setLastName("Chan");
      contact.setBirthDate(
        new Date((new GregorianCalendar(2001, 10, 1)).getTime().getTime()));
      contactDao.insert(contact);
      System.out.println("Listing contact data after new contact created:");
      listAllContacts();

      System.out.println();

      // Удалить только что созданный контакт.
      System.out.println("Deleting the previous created contact");
      contactDao.delete(contact.getId());
      System.out.println("Listing contact data after new contact deleted:");
      listAllContacts();

   }

   private static void listAllContacts() {

      List<Contact> contacts = contactDao.findAll();

      for (Contact contact: contacts) {
         System.out.println(contact);
      }
   }
}
```

Чтобы запустить программу, необходимо добавить к проекту зависимость для MySQL Java, как описано в табл. 8.1.

Таблица 8.1. Зависимость для MySQL

Идентификатор группы	Идентификатор артефакта	Версия	Описание
mysql	mysql-connector-java	5.1.18	Java-библиотека драйвера MySQL

Запуск программы из листинга 8.9 дает следующие результаты (предполагается, что вы имеете локально установленную СУБД MySQL с базой данных prospring3_ch8, а также располагаете именем пользователя и паролем, которые установлены в prospring3; вы должны иметь возможность доступа к схеме базы данных и запустить сценарии schema.sql и test-data.sql для создания таблиц и наполнения их начальными данными):

```
Listing initial contact data:
Contact - Id: 1, First name: Clarence, Last name: Ho, Birthday: 1980-07-30
Contact - Id: 2, First name: Scott, Last name: Tiger, Birthday: 1990-11-02
Contact - Id: 3, First name: John, Last name: Smith, Birthday: 1964-02-28

Insert a new contact
Listing contact data after new contact created:
Contact - Id: 1, First name: Clarence, Last name: Ho, Birthday: 1980-07-30
Contact - Id: 2, First name: Scott, Last name: Tiger, Birthday: 1990-11-02
Contact - Id: 3, First name: John, Last name: Smith, Birthday: 1964-02-28
Contact - Id: 4, First name: Jacky, Last name: Chan, Birthday: 2001-11-01

Deleting the previous created contact
Listing contact data after new contact deleted:
Contact - Id: 1, First name: Clarence, Last name: Ho, Birthday: 1980-07-30
Contact - Id: 2, First name: Scott, Last name: Tiger, Birthday: 1990-11-02
Contact - Id: 3, First name: John, Last name: Smith, Birthday: 1964-02-28
```

В первом блоке строк внутри вывода отображаются начальные данные. Второй блок строк показывает, что была добавлена новая запись. Финальный блок строк отражает факт удаления ранее созданной записи.

В листинге 8.8 видно, что большой объем кода должен быть перемещен во вспомогательный класс или — что еще хуже — продублирован в каждом классе DAO. Это основной недостаток JDBC с точки зрения разработчика приложений — ему просто не хватит терпения повторять один и тот код в каждом классе DAO. Вместо этого лучше сосредоточиться на написании кода, который действительно делает то, что должен делать класс DAO: выборка, обновление и удаление данных. Чем больше вспомогательного кода нужно написать, тем больше проверяемых исключений понадобится проверить и тем больше ошибок можно внести в код.

Именно здесь вступает в игру инфраструктура DAO и Spring. Платформа исключает код, не выполняющий какую-либо специальную логику, позволяя забыть о служебных действиях, которые должны быть выполнены. Кроме того, широкая поддержка JDBC в Spring также значительно упрощает решение задач.

Инфраструктура JDBC в Spring

Код, который обсуждался в первой части этой главы, не был особенно сложным, но его довольно утомительно писать, к тому же при его вводе высока вероятность допущения ошибок. Самое время взглянуть, каким образом Spring упрощает реализацию и делает ее более элегантной.

Обзор пакетов JDBC в Spring

Поддержка JDBC в Spring разделена на пять пакетов, которые описаны в табл. 8.2; каждый пакет обрабатывает те или иные аспекты доступа JDBC.

Таблица 8.2. Пакеты JDBC в Spring

Пакет	Описание
`org.springframework.jdbc.core`	Содержит ядро для классов JDBC в Spring. Пакет включает класс JDBC по имени `JdbcTemplate`, который упрощает программирование операций для базы данных с помощью JDBC. Множество его подпакетов предоставляют поддержку доступа к данным JDBC с более специализированным назначением (например, класс `JdbcTemplate`, который поддерживает именованные параметры), а также связанные служебные классы
`org.springframework.jdbc.datasource`	Содержит вспомогательные классы и реализации `DataSource`, которые можно использовать для запуска кода JDBC вне контейнера JEE. Множество подпакетов предоставляют поддержку для встроенных баз данных, инициализации баз данных и разнообразных механизмов поиска в источниках данных
`org.springframework.jdbc.object`	Содержит классы, которые помогают преобразовывать данные, возвращаемые из базы, в объекты или списки объектов. Эти объекты и списки являются простыми Java-объектами, следовательно, они отключены от базы данных
`org.springframework.jdbc.support`	Наиболее важным средством в этом пакете является поддержка трансляции `SQLException`. Это позволяет Spring распознавать коды ошибок, используемые базой данных, и отображать их на исключения более высокого уровня
`org.springframework.jdbc.config`	Содержит классы, которые поддерживают конфигурацию JDBC внутри `ApplicationContext`. Например, этот пакет включает класс обработчика для пространства имен `jdbc` (например, для дескрипторов `<jdbc:embedded-database>`)

Начнем обсуждение поддержки JDBC в Spring с рассмотрения функциональности самого низкого уровня. Первое, что понадобится сделать перед тем, как можно будет запускать SQL-запросы — это установление подключения к базе данных.

Подключения к базе данных и источники данных

Для управления подключением к базе данных можно использовать Spring, предоставив бин, который реализует `javax.sql.DataSource`. Отличие между `DataSource` и `Connection` состоит в том, что `DataSource` предоставляет и управляет набором `Connection`.

Простейшей реализацией `DataSource` является `DriverManagerDataSource` (из пакета `org.springframework.jdbc.datasource`). Глядя на имя класса, можно предположить, что он просто обращается к `DriverManager` для получения подключения. Тот факт, что `DriverManagerDataSource` не поддерживает пул подключений к базе

данных, делает этот класс неподходящим ни для каких целей, кроме тестирования. Конфигурация DriverManagerDataSource очень проста, как легко заметить в листинге 8.10; нужно лишь указать имя класса драйвера, URL подключения, имя пользователя и пароль (datasource-drivermanager.xml).

Листинг 8.10. Бин dataSource типа DriverManagerDataSource, управляемый Spring

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:context="http://www.springframework.org/schema/context"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context-3.1.xsd">

   <bean id="dataSource"
      class="org.springframework.jdbc.datasource.DriverManagerDataSource">
      <property name="driverClassName">
         <value>${jdbc.driverClassName}</value>
      </property>
      <property name="url">
         <value>${jdbc.url}</value>
      </property>
      <property name="username">
         <value>${jdbc.username}</value>
      </property>
      <property name="password">
         <value>${jdbc.password}</value>
      </property>
   </bean>

   <context:property-placeholder location="jdbc.properties"/>
</beans>
```

Наверняка вы узнали свойства, выделенные в листинге полужирным. Они представляют значения, которые обычно передаются JDBC для получения реализации интерфейса Connection. Информация подключения к базе данных обычно сохраняется в файле свойств для упрощения обслуживания и модификации под нужды разных сред развертывания. В листинге 8.11 показано содержимое файла jdbc.properties, из которого заполнитель свойств Spring будет загружать информацию о подключении.

Листинг 8.11. Файл jdbc.properties

```
jdbc.driverClassName=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/prospring3_ch8
jdbc.username=prospring3
jdbc.password=prospring3
```

В реальных приложениях можно использовать доступный в Apache класс BasicDataSource (http://commons.apache.org/dbcp/) или класс источника данных, реализованный сервером приложений JEE (например, JBoss, WebSphere, WebLogic, GlassFish и т.д.), который может дополнительно увеличить производительность приложения. Источник данных можно было бы применить в простом коде JDBC и получить те же преимущества организации пула; однако в большинстве случаев будет отсутство-

вать возможность централизованного конфигурирования источника данных. С другой стороны, Spring позволяет объявлять бин `dataSource` и устанавливать свойства подключения в файлах определений `ApplicationContext` (листинг 8.12; файл называется `datasource-dbcp.xml`).

На заметку! Кроме класса `BasicDataSource` из Apache, доступны и другие популярные библиотеки с открытым кодом для поддержки пула подключений к базам данных, в числе которых C3P0 (`www.mchange.com/projects/c3p0/index.html`) и BoneCP (`http://jolbox.com/`).

Листинг 8.12. Бин `dataSource`, управляемый Spring

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:context="http://www.springframework.org/schema/context"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context-3.1.xsd">

   <bean id="dataSource"
      class="org.apache.commons.dbcp.BasicDataSource"
      destroy-method="close">
      <property name="driverClassName">
         <value>${jdbc.driverClassName}</value>
      </property>
      <property name="url">
         <value>${jdbc.url}</value>
      </property>
      <property name="username">
         <value>${jdbc.username}</value>
      </property>
      <property name="password">
         <value>${jdbc.password}</value>
      </property>
   </bean>
   <context:property-placeholder location="jdbc.properties"/>
</beans>
```

Показанный источник данных, управляемый Spring, реализован в `org.apache.commons.dbcp.BasicDataSource`. Наиболее важный момент заключается в том, что бин `dataSource` реализует `javax.sql.DataSource`, и его можно непосредственно применять в своих классах доступа к данным.

Другой способ конфигурирования бина `dataSource` предполагает использование JNDI. Если разрабатываемое приложение должно выполняться в контейнере JEE, можно извлечь преимущества от пула подключений, управляемого контейнером. Для работы с источником данных, основанным на JNDI, понадобится изменить объявление бина `dataSource`, как показано в листинге 8.13 (`datasource-jndi.xml`).

Листинг 8.13. Бин `dataSource`, основанный на JNDI и управляемый Spring

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd">

   <bean id="dataSource"
         class="org.springframework.jndi.JndiObjectFactoryBean">
      <property name="jndiName">
         <value>java:comp/env/jdbc/prospring3ch8</value>
      </property>
   </bean>
</beans>
```

В предыдущем примере мы применяли бин `JndiObjectFactoryBean` из Spring для получения источника данных с помощью поиска JNDI. Начиная с версии 2.5, платформа Spring предоставляет пространство имен `jee`, которое дополнительно упрощает конфигурирование. В листинге 8.14 приведена та же самая конфигурация источника данных JNDI, использующая пространство имен `jee` (`datasource-jee.xml`).

Листинг 8.14. Бин `dataSource`, основанный на JNDI и управляемый Spring (используется пространство имен `jee`)

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:jee="http://www.springframework.org/schema/jee"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
      http://www.springframework.org/schema/jee
      http://www.springframework.org/schema/jee/spring-jee-3.1.xsd">

   <jee:jndi-lookup jndi-name="java:comp/env/jdbc/prospring3ch8"/>

</beans>
```

В показанном листинге мы объявляем пространство имен `jee` в дескрипторе `<beans>` и затем объявляем источник данных с помощью дескриптора `<jee:jndi-lookup>`.

Принимая подход JNDI, нужно не забыть о добавлении ссылки на ресурс (`resource-ref`) в файл описателя приложения (листинг 8.15).

Листинг 8.15. Ссылка на ресурс в файле описателя приложения

```
<root-node>
   <resource-ref>
      <res-ref-name>jdbc/prospring3ch8</res-ref-name>
      <res-type>javax.sql.DataSource</res-type>
      <res-auth>Container</res-auth>
   </resource-ref>
</root-node>
```

Дескриптор `<root-node>` является заполнителем; его необходимо изменить в зависимости от того, как упакован ваш модуль. Например, он становится `<web-app>` в описателе веб-развертывания (`WEB-INF/web.xml`), если приложение представляет собой веб-модуль. Скорее всего, потребуется также сконфигурировать `resource-ref` в файле описателя, специфичном для сервера приложений. Однако обратите внимание, что в элементе `resource-ref` указано имя ссылки `jdbc/prospring3ch8`, а свойство `jndiName` бина `dataSource` установлено в `java:comp/env/jdbc/prospring3ch8`.

Как видите, платформа Spring позволяет конфигурировать источник данных практически любым желаемым образом, и она скрывает действительную реализацию или местоположение источника данных от остальной части кода приложения. Другими словами, ваши классы DAO не знают, да и не должны знать, на что указывает источник данных.

Управление подключениями также делегировано бину `dataSource`, который, в свою очередь, осуществляет управление самостоятельно либо использует контейнер JEE для выполнения всей работы.

Поддержка встроенной базы данных

Начиная с версии 3.0, платформа Spring также предлагает поддержку встроенных баз данных, которая автоматически запускает нужную встроенную базу данных и делает ее доступной как источник данных для приложения. В листинге 8.16 показана конфигурация встроенной базы данных (`app-context-xml.xml`).

Листинг 8.16. Поддержка встроенной базы данных в Spring

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:jdbc="http://www.springframework.org/schema/jdbc"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
      http://www.springframework.org/schema/jdbc
      http://www.springframework.org/schema/jdbc/spring-jdbc-3.1.xsd">

   <jdbc:embedded-database id="dataSource" type="H2">
      <jdbc:script location="classpath:schema.sql"/>
      <jdbc:script location="classpath:test-data.sql"/>
   </jdbc:embedded-database>

</beans>
```

В предыдущем листинге мы сначала объявляем пространство имен `jdbc` в дескрипторе `<beans>`. После этого мы используем дескриптор `<jdbc:embedded-database>` для объявления встроенной базы данных и назначаем ей идентификатор `dataSource`. Внутри этого дескриптора мы также сообщаем Spring о необходимости выполнения указанных сценариев, которые предназначены для создания схемы базы данных и наполнения ее тестовыми данными. Обратите внимание, что порядок следования сценариев является важным: файл с командами DDL (Data Definition Language — язык определения данных) всегда должен указываться первым, а за ним — файл с командами DML (Data Manipulation Language — язык манипулирования данными). В атрибуте `type` задается тип используемой встроенной базы данных. В версии Spring 3.1 поддерживаются типы HSQL (по умолчанию), H2 и DERBY.

Поддержка встроенной базы данных исключительно полезна при локальной разработке и модульном тестировании. В оставшейся части этой главы мы будем применять встроенную базу данных для запуска кода примеров, так что устанавливать какую-либо СУБД на вашей машине не понадобится.

Использование источников данных в классах DAO

Давайте начнем с пустого интерфейса ContactDao и простой его реализации. По мере освоения классов JDBC в Spring мы будем добавлять к нему дополнительные возможности. Пустой интерфейс ContactDao показан в листинге 8.17.

Листинг 8.17. Интерфейс и реализация ContactDao

```
public interface ContactDao { }
public class JdbcContactDao implements ContactDao { }
```

Первым делом мы добавим к этой простой реализации свойство dataSource. Причина, по которой мы хотим добавить свойство dataSource в класс реализации, а не в интерфейс, должна быть довольно очевидной: интерфейс не обязан знать, каким образом данные будут извлекаться и обновляться. Добавляя методы getDataSource/setDataSource в интерфейс, мы в лучшем случае принуждаем реализации к объявлению заглушек для методов получения и установки. Очевидно, это не особенно удачный подход. Взгляните на простой класс JdbcContactDao в листинге 8.18.

Листинг 8.18. Класс JdbcUserDao со свойством dataSource

```
package com.apress.prospring3.ch8.dao.jdbc.xml;

import javax.sql.DataSource;

import com.apress.prospring3.ch8.dao.ContactDao;

public class JdbcContactDao implements ContactDao {

    private DataSource dataSource;

    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }
}
```

Теперь мы можем сообщить Spring о необходимости конфигурирования бина contactDao с использованием реализации JdbcContactDao и установки свойства dataSource (листинг 8.19; файл называется app-context-xml.xml).

Листинг 8.19. Файл контекста приложения Spring с бинами dataSource и contactDao

```
// Объявление пространства имен опущено.

<jdbc:embedded-database id="dataSource" type="H2">
    <jdbc:script location="classpath:schema.sql"/>
    <jdbc:script location="classpath:test-data.sql"/>
</jdbc:embedded-database>

<bean id="contactDao" class="com.apress.prospring3.ch8.dao.jdbc.xml.JdbcContactDao">
    <property name="dataSource">
        <ref local="dataSource"/>
    </property>
</bean>
```

Для поддержки базы данных H2 необходимо добавить к проекту соответствующую зависимость, как показано в табл. 8.3.

Таблица 8.3. Зависимость для базы данных H2

Идентификатор группы	Идентификатор артефакта	Версия	Описание
com.h2database	h2	1.3.160	Java-библиотека для базы данных H2

Платформа Spring теперь создает бин contactDao за счет получения экземпляра класса JdbcContactDao со свойством dataSource, установленным в бин dataSource.

Рекомендуется удостоверяться в том, что все обязательные свойства бина были должным образом установлены. Проще всего это сделать, реализовав интерфейс InitializingBean и предоставив реализацию для метода afterPropertiesSet() (листинг 8.20). В таком случае вы гарантируете, что все обязательные свойства JdbcContactDao правильно установлены. Более подробные сведения об инициализации бинов ищите в главе 5.

Листинг 8.20. Класс JdbcContactDao с реализацией интерфейса InitializingBean

```
package com.apress.prospring3.ch8.dao.jdbc.xml;

import javax.sql.DataSource;

import org.springframework.beans.factory.BeanCreationException;
import org.springframework.beans.factory.InitializingBean;

import com.apress.prospring3.ch8.dao.ContactDao;

public class JdbcContactDao implements ContactDao, InitializingBean {

   private DataSource dataSource;
   public void setDataSource(DataSource dataSource) {
      this.dataSource = dataSource;
   }
   public void afterPropertiesSet() throws Exception {
      if (dataSource == null) {
        throw new BeanCreationException("Must set dataSource on ContactDao");
      }
   }
}
```

В коде, показанном до сих пор, для управления источником данных использовалась платформа Spring, а также вводился интерфейс ContactDao и его реализация JDBC. Мы также устанавливали свойство dataSource класса JdbcContactDao в файле для ApplicationContext. А теперь мы расширим код, добавив к интерфейсу и реализации действительные операции DAO.

Обработка исключений

Поскольку в Spring принято использовать исключения времени выполнения, а не проверяемые исключения, необходим механизм трансляции проверяемого исключения SQLException в исключение времени выполнения Spring JDBC. Так как исключения, связанные с SQL в Spring, представляют собой исключения времени выполнения, они могут быть намного более детализированными, чем проверяемые исключения. (По опре-

делению это не является функциональной возможностью исключений времени, однако весьма неудобно объявлять длинный список проверяемых исключений в конструкции throws; следовательно, проверяемые исключения, как правило, менее детализированы, чем их эквиваленты времени выполнения.)

Платформа Spring предоставляет стандартную реализацию интерфейса SQLExceptionTranslator, которая берет на себя обязанности по трансляции обобщенных кодов ошибок SQL в исключения Spring JDBC. В большинстве случаев этой реализации вполне достаточно, но мы можем расширить стандартную реализацию Spring и указать новую реализацию SQLExceptionTranslator для использования в JdbcTemplate, как показано в листинге 8.21.

В то же время к проекту нужно добавить зависимость для spring-jdbc, как описано в табл. 8.4.

Таблица 8.4. Зависимость для spring-jdbc

Идентификатор группы	Идентификатор артефакта	Версия	Описание
org.springframework	spring-jdbc	3.1.0.RELEASE	Модуль Spring JDBC

Листинг 8.21. Специальный класс SQLExceptionTranslator

```
package com.apress.prospring3.ch8.exception.translator;

import java.sql.SQLException;

import org.springframework.dao.DataAccessException;
import org.springframework.dao.DeadlockLoserDataAccessException;
import org.springframework.jdbc.support.SQLErrorCodeSQLExceptionTranslator;

public class MySQLErrorCodesTranslator extends
   SQLErrorCodeSQLExceptionTranslator  {

   protected DataAccessException customTranslate(String task,
         String sql, SQLException sqlex) {
      if (sqlex.getErrorCode() == -12345)
         return new DeadlockLoserDataAccessException(task, sqlex);
      return null;
   }
}
```

Чтобы использовать специальный транслятор, его необходимо передать JdbcTemplate в классах DAO. В листинге 8.22 представлен соответствующий фрагмент кода.

Листинг 8.22. Использование специального класса SQLExceptionTranslator в Spring JDBC

```
   // Внутри любого класса DAO.

   JdbcTemplate jdbcTemplate = new JdbcTemplate();
   jdbcTemplate.setDataSource(dataSource);

   // Создать специальный транслятор и установить источник
   // данных для поиска стандартной трансляции.
   MySQLErrorCodesTranslator errorTranslator =
      new MySQLErrorCodesTranslator();
   errorTranslator.setDataSource(dataSource);
   jdbcTemplate.setExceptionTranslator(errorTranslator);
```

```
// Использовать JdbcTemplate для SqlUpdate.
SqlUpdate sqlUpdate = new SqlUpdate();
sqlUpdate.setJdbcTemplate(jdbcTemplate);
sqlUpdate.setSql("update contact set first_name = 'Clarence'");
sqlUpdate.compile();
sqlUpdate.update();
```

Платформа Spring будет вызывать этот специальный транслятор исключений SQL при обнаружении исключений SQL во время выполнения SQL-операторов в базе данных и трансляция произойдет, когда код ошибки равен `-12345`. Для других ошибок Spring применит свой стандартный механизм трансляции исключений.

Очевидно, ничто не мешает создать `SQLExceptionTranslator` в виде управляемого Spring бина и применять бин `JdbcTemplate` в своих классах DAO. Не волнуйтесь, если не помните прочитанное о классе `JdbcTemplate`; мы собираемся обсудить его более подробно.

Класс JdbcTemplate

Этот класс представляет ядро поддержки JDBC в Spring. Он способен выполнять все типы SQL-операторов. Выражаясь упрощенно, операторы могут быть разделены на определяющие данные и манипулирующие данными. Операторы определения данных отвечают за создание разнообразных объектов базы данных (таблиц, представлений, хранимых процедур и т.п.). Операторы манипулирования данными взаимодействуют с данными и могут быть классифицированы на операторы выборки и обновления. Оператор выборки в общем случае возвращает набор строк, каждая из которых содержит один и тот же набор столбцов. Оператор обновления модифицирует данные в базе, но никакого результата не возвращает.

Класс `JdbcTemplate` позволяет отправлять базе данных SQL-оператор любого типа и возвращать результат также любого типа.

В этом разделе мы рассмотрим общие сценарии использования поддержки JDBC в Spring с помощью класса `JdbcTemplate`.

Инициализация JdbcTemplate в классе DAO

Перед обсуждением использования `JdbcTemplate` давайте посмотрим, как подготовить `JdbcTemplate` для применения в классе DAO. Делается это очень просто; в большинстве случаев нужно лишь сконструировать экземпляр класса, передав объект источника данных (который должен быть внедрен платформой Spring в класс DAO). В листинге 8.23 приведен фрагмент кода, инициализирующего объект `JdbcTemplate`.

Листинг 8.23. Инициализация JdbcTemplate

```
private JdbcTemplate jdbcTemplate;

private DataSource dataSource;

public void setDataSource(DataSource dataSource) {
    this.dataSource = dataSource;
    this.jdbcTemplate = new JdbcTemplate(dataSource);
}
```

Общая практика заключается в инициализации `JdbcTemplate` внутри метода установки источника данных, поэтому как только источник данных внедряется Spring, `JdbcTemplate` будет также инициализирован и готов к использованию.

После конфигурирования JdbcTemplate является безопасным в отношении потоков. Это означает, что можно также инициализировать одиночный экземпляр JdbcTemplate в XML-файле конфигурации Spring и внедрить его во все бины DAO.

На заметку! В Spring-модуле Jdbc имеется класс по имени JdbcDaoSupport. Он является оболочкой для класса JdbcTemplate и ваши классы DAO могут расширять класс JdbcDaoSupport. В этом случае после внедрения такого класса DAO с источником данных JdbcTemplate будет инициализирован автоматически.

Извлечение одиночного значения с использованием класса JdbcTemplate

Начнем с простого запроса, возвращающего одиночное значение. Например, мы хотим иметь возможность извлекать имя контакта по его идентификатору. Давайте сначала добавим метод в интерфейс ContactDao:

```
public String findFirstNameById(Long id);
```

Извлекать значение с применением JdbcTemplate несложно. В листинге 8.24 показана реализация метода findFirstNameById() в классе JdbcContactDao. Для других методов предоставлены пустые реализации.

Листинг 8.24. Использование JdbcTemplate для извлечения одиночного значения

```
package com.apress.prospring3.ch8.dao.jdbc.xml;

// Операторы импорта опущены.

public class JdbcContactDao implements ContactDao, InitializingBean {

    public String findFirstNameById(Long id) {
        String firstName = jdbcTemplate.queryForObject(
            "select first_name from contact where id = ?",
            new Object[]{id}, String.class);
        return firstName;
    }

    public List<Contact> findAll() {
        return null;
    }

    public List<Contact> findByFirstName(String firstName) {
        return null;
    }

    public void insert(Contact contact) {
    }

    public void update(Contact contact) {
    }

    public void delete(Long contactId) {
    }

}
```

В предыдущем листинге для извлечения значения имени используется метод queryForObject() класса JdbcTemplate. Первый аргумент — это строка с SQL-оператором, а второй — параметры, передаваемые SQL-оператору для связывания, в

формате массива объектов. Последний аргумент представляет возвращаемый тип, которым в рассматриваемом случае является `String`. Кроме `Object` можно также запрашивать другие типы, например, `Long` и `Integer`. Давайте посмотрим на результат. В листинге 8.25 приведен код тестовой программы.

Листинг 8.25. Использование `JdbcTemplate`

```
package com.apress.prospring3.ch8;

import org.springframework.context.support.GenericXmlApplicationContext;

import com.apress.prospring3.ch8.dao.ContactDao;

public class JdbcContactDaoSample {

   public static void main(String[] args) {

      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:app-context-xml.xml");
      ctx.refresh();

      ContactDao contactDao = ctx.getBean("contactDao", ContactDao.class);

      // Найти имя по идентификатору.
      System.out.println("First name for contact id 1 is: " +
         contactDao.findFirstNameById(1l));

   }
}
```

Как и можно было ожидать, запуск этой программы дает следующий вывод:

```
First name for contact id 1 is: Clarence
```

Использование именованных параметров с помощью `NamedParameterJdbcTemplate`

В предыдущем примере мы использовали обычный заполнитель (символ `?`) в качестве параметров запроса. Вдобавок вы видели, что нужно было передавать параметры в виде массива `Object`. Когда применяется обычный заполнитель, порядок становится очень важным, и порядок помещения параметров в массив должен быть таким же, как и порядок указания параметров в запросе.

Некоторые разработчики предпочитают использовать именованные параметры, чтобы гарантировать точную привязку параметров. Соответствующую поддержку в Spring обеспечивает разновидность класса `JdbcTemplate` по имени `NamedParameterJdbcTemplate` (из пакета `org.springframework.jdbc.core.namedparam`). Давайте посмотрим, как он работает.

Например, нам необходим еще один метод для поиска фамилии по идентификатору, поэтому добавим его в интерфейс `ContactDao`:

```
public String findLastNameById(Long id);
```

Инициализация `NamedParameterJdbcTemplate` такая же, как для `JdbcTemplate`, т.е. мы просто должны объявить переменную типа `NamedParameterJdbcTemplate` и добавить в метод `setDataSource()` класса DAO следующую строку:

```
this.namedParameterJdbcTemplate = new NamedParameterJdbcTemplate(dataSource);
```

Теперь рассмотрим реализацию этого метода, которая показана в листинге 8.26.

Листинг 8.26. Использование NamedParameterJdbcTemplate для извлечения одиночного значения

```
package com.apress.prospring3.ch8.dao.jdbc.xml;

// Операторы импорта опущены.

public class JdbcContactDao implements ContactDao, InitializingBean {

   // Код для других методов не показан.

   public String findLastNameById(Long id) {
       String sql = "select last_name from contact where id = :contactId";

      SqlParameterSource namedParameters =
         new MapSqlParameterSource("contactId", id);
      return namedParameterJdbcTemplate.queryForObject(sql,
         namedParameters, String.class);
   }
}
```

Во-первых, здесь вместо заполнителя ? используется именованный параметр (предваряемый двоеточием). Во-вторых, экземпляр реализации `SqlParameterSource` инициализирован как источник SQL-параметров, основанный на `Map`, с ключом — именем параметра и значением — значением параметра. Вместо `SqlParameterSource` можно также просто построить карту для хранения именованных параметров. В листинге 8.27 приведен один из вариантов предыдущего метода.

Листинг 8.27. Использование NamedParameterJdbcTemplate для извлечения одиночного значения (другой вариант)

```
package com.apress.prospring3.ch8.dao.jdbc.xml;

// Операторы импорта опущены.

public class JdbcContactDao implements ContactDao, InitializingBean {

   // Код для других методов не показан.

   public String findLastNameById(Long id) {
   String sql = "select last_name from contact where id = :contactId";

   Map<String, Object> namedParameters = new HashMap<String, Object>();
   namedParameters.put("contactId", id);
   return namedParameterJdbcTemplate.queryForObject(sql,
      namedParameters, String.class);
   }
}
```

Для тестирования этого кода просто добавьте метод в главный тестовый класс, представленный в листинге 8.25, и запустите его. Полученный вывод здесь не показан.

Извлечение объектов предметной области с помощью RowMapper<T>

Большую часть времени вместо извлечения одиночного значения вам требуется запрашивать одну или более строк и затем трансформировать каждую строку в соответствующий объект предметной области.

Интерфейс `RowMapper<T>` в Spring (из пакета `org.springframework.jdbc.core`) поддерживает простой способ отображения результирующего набора JDBC на объек-

ты POJO. Давайте посмотрим на него в действии, реализовав метод findAll() интерфейса ContactDao с использованием интерфейса RowMapper<T>. Реализация метода findAll() показана в листинге 8.28.

Листинг 8.28. Использование RowMapper<T> для запроса объектов предметной области

```
package com.apress.prospring3.ch8.dao.jdbc.xml;

// Операторы импорта опущены.
public class JdbcContactDao implements ContactDao, InitializingBean {

   // Код для других методов не показан.

   public List<Contact> findAll() {
     String sql = "select id, first_name, last_name, birth_date from contact";
     return jdbcTemplate.query(sql, new ContactMapper());
   }
   private static final class ContactMapper implements RowMapper<Contact> {

      public Contact mapRow(ResultSet rs, int rowNum) throws SQLException {
         Contact contact = new Contact();
         contact.setId(rs.getLong("id"));
         contact.setFirstName(rs.getString("first_name"));
         contact.setLastName(rs.getString("last_name"));
         contact.setBirthDate(rs.getDate("birth_date"));
         return contact;
      }
   }
}
```

В предыдущем листинге мы определяем статический внутренний класс по имени ContactMapper, который реализует интерфейс RowMapper<T>. Класс должен предоставить реализацию mapRow(), которая трансформирует значения конкретной записи результирующего набора в необходимый объект предметной области. Объявление его как статического внутреннего класса позволяет разделять RowMapper<T> между множеством методов поиска.

Затем метод findAll() просто должен вызвать метод запроса, передав ему строку запроса и экземпляр отображения строки. На случай, когда запрос требует параметры, для метода query() предоставляется перегруженная версия, которая принимает параметры запроса.

Добавим фрагмент кода, показанный в листинге 8.29, в тестовую программу (класс JdbcContactDaoSample).

Листинг 8.29. Фрагмент кода для вывода списка контактов с применением RowMapper<T>

```
      // Найти и вывести список всех контактов.
      List<Contact> contacts = contactDao.findAll();
      for (Contact contact: contacts) {
         System.out.println(contact);
         if (contact.getContactTelDetails() != null) {
            for (ContactTelDetail contactTelDetail:
               contact.getContactTelDetails()) {
               System.out.println("---" + contactTelDetail);
            }
         }
         System.out.println();
      }
```

Запуск этой программы дает следующий результат (показан не весь вывод):

```
Contact - Id: 1, First name: Clarence, Last name: Ho, Birthday: 1980-07-30
Contact - Id: 2, First name: Scott, Last name: Tiger, Birthday: 1990-11-02
Contact - Id: 3, First name: John, Last name: Smith, Birthday: 1964-02-28
```

Извлечение вложенных объектов предметной области с помощью ResultSetExtractor

Давайте перейдем к рассмотрению более сложного примера, в котором необходимо извлечь данные из родительской (CONTACT) и дочерней (CONTACT_TEL_DETAIL) таблиц с помощью соединения, а затем соответствующим образом трансформировать данные во вложенный объект (ContactTelDetail внутри Contact).

Упомянутый ранее интерфейс RowMapper<T> подходит только для отображения строки на одиночный объект предметной области. Для более сложной объектной структуры должен использоваться интерфейс ResultSetExtractor. Чтобы продемонстрировать его применение, мы добавим к интерфейсу ContactDao еще один метод по имени findAllWithDetail(). Этот метод должен наполнить список контактов информацией о телефонах:

```
public List<Contact> findAllWithDetail();
```

В листинге 8.30 показана реализация метода findAllWithDetail() с использованием ResultSetExtractor.

Листинг 8.30. Использование ResultSetExtractor для запроса объектов предметной области

```
package com.apress.prospring3.ch8.dao.jdbc.xml;

// Операторы импорта опущены.
public class JdbcContactDao implements ContactDao, InitializingBean {

   public List<Contact> findAllWithDetail() {
      String sql = "select c.id, c.first_name, c.last_name, c.birth_date"
        + ", t.id as contact_tel_id, t.tel_type, t.tel_number from contact c "
        + "left join contact_tel_detail t on c.id = t.contact_id";
      return jdbcTemplate.query(sql, new ContactWithDetailExtractor());
   }
   private static final class ContactWithDetailExtractor
          implements ResultSetExtractor<List<Contact>> {
      public List<Contact> extractData(ResultSet rs)
         throws SQLException, DataAccessException {
          Map<Long, Contact> map = new HashMap<Long, Contact>();
          Contact contact = null;
          while (rs.next()) {
             Long id = rs.getLong("id");
             contact = map.get(id);
             if (contact == null) {  // новая запись контакта
               contact = new Contact();
               contact.setId(id);
               contact.setFirstName(rs.getString("first_name"));
               contact.setLastName(rs.getString("last_name"));
               contact.setBirthDate(rs.getDate("birth_date"));
               contact.setContactTelDetails(new ArrayList<ContactTelDetail>());
               map.put(id, contact);
             }
```

```
            // Обработать информацию о телефонах контакта (если есть).
            Long contactTelDetailId = rs.getLong("contact_tel_id");
            if (contactTelDetailId > 0) {
               ContactTelDetail contactTelDetail = new ContactTelDetail();
               contactTelDetail.setId(contactTelDetailId);
               contactTelDetail.setContactId(id);
               contactTelDetail.setTelType(rs.getString("tel_type"));
               contactTelDetail.setTelNumber(rs.getString("tel_number"));
               contact.getContactTelDetails().add(contactTelDetail);
            }
         }
         return new ArrayList<Contact> (map.values());
      }
   }
}
```

Код похож на пример `RowMapper`, но в этот раз мы объявляем внутренний класс, который реализует `ResultSetExtractor`. Затем мы реализуем метод `extractData()` для трансформации результирующего набора в список объектов `Contact`.

В методе `findAllWithDetail()` запрос использует левое соединение двух таблиц, так что контакты без телефонов также будут извлечены. Результатом является декартово произведение двух таблиц. Наконец, мы вызываем метод `JdbcTemplate.query()`, передавая ему строку запроса и экземпляр реализации `ResultSetExtractor`.

Добавим фрагмент кода, приведенный в листинге 8.31, в тестовую программу (класс `JdbcContactDaoSample`).

Листинг 8.31. Фрагмент кода для вывода списка контактов с применением `ResultSetExtractor`

```
      // Найти и вывести список всех контактов вместе с деталями.
      List<Contact> contactsWithDetail = contactDao.findAllWithDetail();
      for (Contact contact: contactsWithDetail) {
         System.out.println(contact);
         if (contact.getContactTelDetails() != null) {
            for (ContactTelDetail contactTelDetail:
                  contact.getContactTelDetails()) {
               System.out.println("---" + contactTelDetail);
            }
         }
         System.out.println();
      }
```

Запустив тестовую программу еще раз, получаем следующий результат (показан не весь вывод):

```
Contact - Id: 1, First name: Clarence, Last name: Ho, Birthday: 1980-07-30
---Contact Tel Detail - Id: 2, Contact id: 1, Type: Home, Number: 1234567890
---Contact Tel Detail - Id: 1, Contact id: 1, Type: Mobile, Number: 1234567890

Contact - Id: 2, First name: Scott, Last name: Tiger, Birthday: 1990-11-02
---Contact Tel Detail - Id: 3, Contact id: 2, Type: Home, Number: 1234567890

Contact - Id: 3, First name: John, Last name: Smith, Birthday: 1964-02-28
```

Как видите, контакты и сведения о связанных с ними телефонах выведены соответствующим образом. Данные были сгенерированы сценарием из листинга 8.2.

До сих пор вы видели, как применять JdbcTemplate для выполнения ряда общих операций запросов. Класс JdbcTemplate (а также класс NamedParameterJdbcTemplate) предоставляет несколько перегруженных методов update(), которые поддерживают операции обновления данных, включая вставку, обновление, удаление и т.д. Поскольку метод update() довольно самоочевиден, мы не будем его здесь рассматривать.

С другой стороны, как будет показано в последующих разделах, мы будем использовать класс SqlUpdate, предоставляемый Spring, для выполнения операций обновления данных.

Классы Spring, моделирующие операции JDBC

В предыдущем разделе вы видели, что JdbcTemplate и связанные служебные классы для отображения данных существенно упрощают программную модель при разработке логики доступа к данным с помощью JDBC. Построенная поверх JdbcTemplate платформа Spring также предоставляет множество полезных классов, которые моделируют операции над данными JDBC и позволяют разработчикам реализовывать логику запроса и трансформации результирующего набора в объекты предметной области в более объектно-ориентированной манере. Как уже упоминалось, эти классы упакованы в org.springframework.jdbc.object. В частности, мы обсудим следующие классы.

- *MappingSqlQuery<T>*. Класс MappingSqlQuery<T> позволяет поместить строку запроса вместе с методом mapRow() в оболочку единственного класса.
- *SqlUpdate*. Класс SqlUpdate позволяет поместить внутрь себя любой SQL-оператор обновления. Он также предлагает множество полезных функций для привязки SQL-параметров, извлечения сгенерированного СУРБД ключа после вставки новой записи и т.д.
- *BatchSqlUpdate*. Класс BatchSqlUpdate позволяет выполнять пакетные операции обновления. Например, можно проходить в цикле по Java-объекту List и с помощью BatchSqlUpdate заносить в очередь записи, для которых затем будут выданы операторы обновления в пакете. Можно устанавливать размер пакета и сбрасывать операцию в любой момент.
- *SqlFunction<T>*. Класс SqlFunction<T> позволяет вызывать хранимые функции в базе данных с аргументами и возвращаемым типом. Существует еще один класс, StoredProcedure, который помогает вызывать хранимые процедуры.

На заметку! Во всех примерах кода, которые рассматривались в предыдущих разделах, применялась XML-конфигурация. Поэтому в последующих разделах для конфигурирования ApplicationContext мы будем использовать аннотации Spring. Если же вы решите, что в разрабатываемом приложении больше подходит XML-конфигурация, мы уверены, что вы справитесь с этой задачей самостоятельно.

Настройка DAO-классов JDBC с использованием аннотаций

Первым делом давайте посмотрим, как настроить класс реализации DAO, используя аннотации. В листинге 8.32 представлен интерфейс ContactDao с более полным списком предоставляемых им служб доступа к данным.

Листинг 8.32. Интерфейс ContactDao

```
package com.apress.prospring3.ch8.dao;

// Операторы импорта опущены.
public interface ContactDao {

   public List<Contact> findAll();

   public List<Contact> findAllWithDetail();

   public List<Contact> findByFirstName(String firstName);

   public String findFirstNameById(Long id);

   public String findLastNameById(Long id);

   public void insert(Contact contact);

   public void update(Contact contact);

   public void delete(Long contactId);

}
```

В листинге 8.33 показано начальное объявление и внедрение свойства источника данных с применением аннотации JSR-250. Именем класса осталось JdbcContactDao, но в этот раз он помещен в пакет com.apress.prospring3.ch8.dao.jdbc.annotation.

Листинг 8.33. Объявление JdbcContactDao с использованием аннотаций

```
package com.apress.prospring3.ch8.dao.jdbc.annotation;

import javax.annotation.Resource;
import javax.sql.DataSource;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.stereotype.Repository;

import com.apress.prospring3.ch8.dao.ContactDao;

@Repository("contactDao")
public class JdbcContactDao implements ContactDao {

   private Log log = LogFactory.getLog(JdbcContactDao.class);

   private DataSource dataSource;

   @Resource(name="dataSource")
   public void setDataSource(DataSource dataSource) {
      this.dataSource = dataSource;
   }

   public DataSource getDataSource() {
      return dataSource;
   }
}
```

В предыдущем листинге мы используем аннотацию @Repository для объявления бина Spring с именем contactDao, и поскольку класс содержит код доступа к данным, @Repository также инструктирует Spring транслировать исключения SQL, специфичные от базы данных, в более дружественную к приложению иерархию DataAccessException, поддерживаемую платформой.

Мы также объявляем переменную `log`, используя пакет `commons-logging` из Apache, предназначенную для записи сообщений в журнал внутри программы. Наконец, мы применяем к свойству источника данных аннотацию `@Resource` стандарта JSR-250, чтобы позволить Spring внедрить источник данных по имени `dataSource`.

Мы собираемся реализовать методы интерфейса `ContactDao` друг за другом, но начнем с создания пустых реализаций всех методов в классе `JdbcContactDao`. Проще всего воспользоваться STS для генерации пустых реализаций. В среде STS щелкните правой кнопкой мыши на имени класса и выберите в контекстном меню пункт Source⇨Override/Implement Methods (Исходный код⇨Переопределить/реализовать методы), как показано на рис. 8.2.

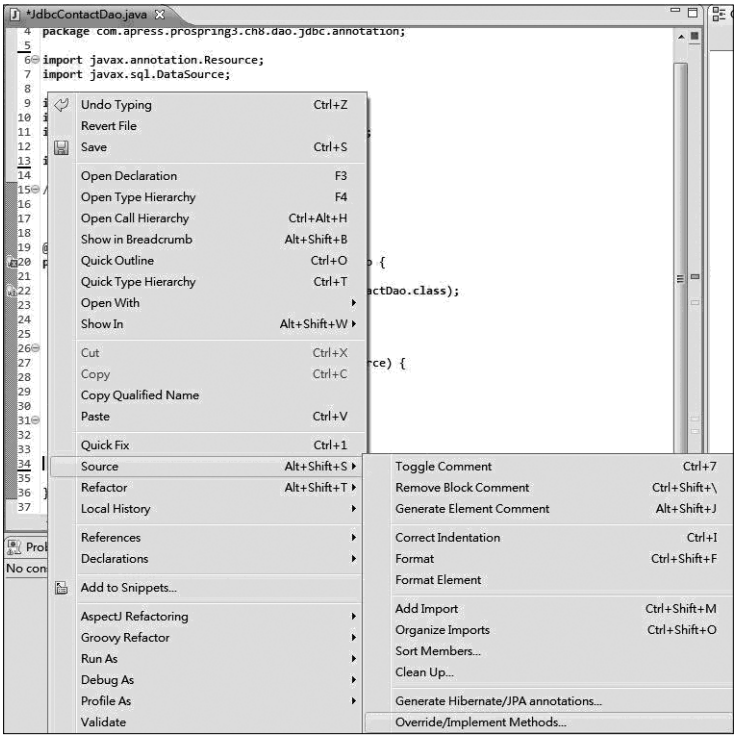


Рис. 8.2. Реализация методов в STS

На следующем экране уже должны быть выделены все методы интерфейса `ContactDao` (рис. 8.3). Просто щелкните на кнопке OK и для всех выбранных методов будут автоматически созданы пустые реализации.

Затем вы увидите сгенерированные пустые реализации этих методов. После этого можете приступать к постепенной реализации методов.

В листинге 8.34 представлена XML-конфигурация для Spring, использующая аннотации (`app-context-annotation.xml`).

Листинг 8.34. XML-конфигурация для Spring, использующая аннотации

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:context="http://www.springframework.org/schema/context"
```

```
   xmlns:jdbc="http://www.springframework.org/schema/jdbc"
   xmlns:jee="http://www.springframework.org/schema/jee"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context-3.1.xsd
      http://www.springframework.org/schema/jdbc
      http://www.springframework.org/schema/jdbc/spring-jdbc-3.1.xsd
      http://www.springframework.org/schema/jee
      http://www.springframework.org/schema/jee/spring-jee-3.1.xsd">
   <context:component-scan base-package=
      "com.apress.prospring3.ch8.dao.jdbc.annotation"/>
   <context:annotation-config/>
   <jdbc:embedded-database id="dataSource" type="H2">
      <jdbc:script location="classpath:schema.sql"/>
      <jdbc:script location="classpath:test-data.sql"/>
   </jdbc:embedded-database>
</beans>
```

В этой конфигурации нет ничего особо примечательного; мы просто объявляем встроенную базу данных типа H2 и применяем `<context:component-scan>` для автоматического обнаружения бинов Spring. Построив такую инфраструктуру, мы можем переходить к реализации операций JDBC.

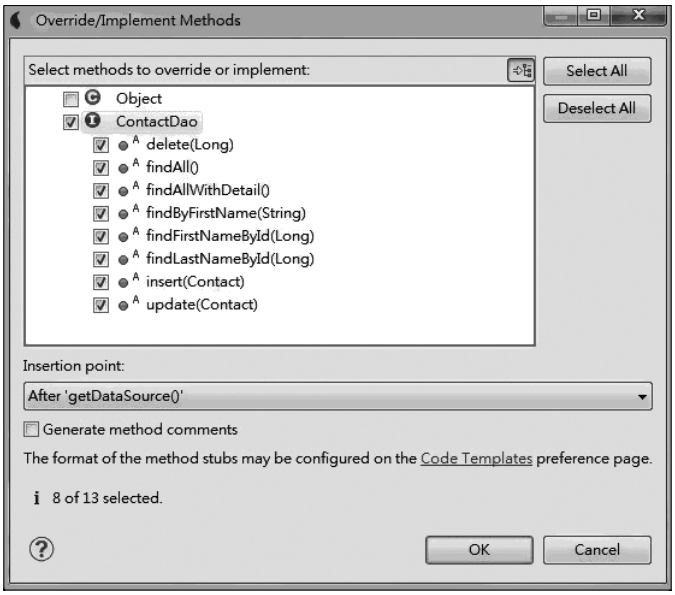


Рис. 8.3. Выбор методов для реализации в STS

Запрашивание данных с использованием `MappingSqlQuery<T>`

Для моделирования операций запросов платформа Spring предоставляет класс `MappingSqlQuery<T>`. В сущности, мы конструируем класс `MappingSqlQuery<T>`, используя источник данных и строку запроса. С другой стороны, мы реализуем метод `mapRow()` для отображения каждой записи результирующего набора в соответствующий объект предметной области.

Давайте первым реализуем метод findAll(). Мы начнем с создания класса SelectAllContacts (который представляет операцию запроса для выборки всех контактов), расширяющего абстрактный класс MappingSqlQuery<T>. В листинге 8.35 приведен код класса SelectAllContacts.

Листинг 8.35. Класс SelectAllContacts

```
package com.apress.prospring3.ch8.dao.jdbc.annotation;

import java.sql.ResultSet;
import java.sql.SQLException;

import javax.sql.DataSource;

import org.springframework.jdbc.object.MappingSqlQuery;

import com.apress.prospring3.ch8.domain.Contact;

public class SelectAllContacts extends MappingSqlQuery<Contact> {

   private static final String SQL_SELECT_ALL_CONTACT =
      "select id, first_name, last_name, birth_date from contact";

   public SelectAllContacts(DataSource dataSource) {
      super(dataSource, SQL_SELECT_ALL_CONTACT);
   }

   protected Contact mapRow(ResultSet rs, int rowNum) throws SQLException {
      Contact contact = new Contact();

      contact.setId(rs.getLong("id"));
      contact.setFirstName(rs.getString("first_name"));
      contact.setLastName(rs.getString("last_name"));
      contact.setBirthDate(rs.getDate("birth_date"));

      return contact;
   }
}
```

В листинге 8.35 внутри класса SelectAllContacts объявляется SQL-оператор для выборки всех контактов. В конструкторе класса вызывается метод super() для создания экземпляра класса, используя источник данных и SQL-оператор. Кроме того, метод MappingSqlQuery<T>.mapRow() реализован для отображения результирующего набора на объект предметной области Contact.

Имея класс SelectAllContacts, мы можем реализовать метод findAll() в классе JdbcContactDao, как показано в листинге 8.36.

Листинг 8.36. Реализация метода findAll()

```
package com.apress.prospring3.ch8.dao.jdbc.annotation;

import java.util.List;

import javax.annotation.Resource;
import javax.sql.DataSource;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

import org.springframework.stereotype.Repository;

import com.apress.prospring3.ch8.dao.ContactDao;
import com.apress.prospring3.ch8.domain.Contact;

@Repository("contactDao")
```

```
public class JdbcContactDao implements ContactDao {
   private Log log = LogFactory.getLog(JdbcContactDao.class);
   private DataSource dataSource;
   private SelectAllContacts selectAllContacts;
   @Resource(name="dataSource")
   public void setDataSource(DataSource dataSource) {
      this.dataSource = dataSource;
      selectAllContacts = new SelectAllContacts(dataSource);
   }
   public DataSource getDataSource() {
      return dataSource;
   }
   public List<Contact> findAll() {
      return selectAllContacts.execute();
   }
   // Другие методы с пустой реализацией не показаны.
}
```

В листинге 8.36 внутри метода `setDataSource()` при внедрении источника данных создается экземпляр класса `SelectAllContacts`. В методе `findAll()` мы просто вызываем метод `SelectAllContacts.execute()`, который непрямо унаследован из абстрактного класса `SqlQuery<T>`. Это все, что нужно было сделать. В листинге 8.37 приведен пример программы для тестирования полученной логики.

Листинг 8.37. Тестирование `MappingSqlQuery`

```
package com.apress.prospring3.ch8;
// Операторы импорта опущены.
public class AnnotationJdbcDaoSample {
   public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:app-context-annotation.xml");
      ctx.refresh();
      ContactDao contactDao = ctx.getBean("contactDao", ContactDao.class);
      // Найти и вывести список всех контактов.
      List<Contact> contacts = contactDao.findAll();
      listContacts(contacts);
   }
   private static void listContacts(List<Contact> contacts) {
      for (Contact contact: contacts) {
         System.out.println(contact);
         if (contact.getContactTelDetails() != null) {
           for (ContactTelDetail contactTelDetail: contact.getContactTelDetails())
           {
              System.out.println("---" + contactTelDetail);
           }
         }
         System.out.println();
      }
   }
}
```

Запуск тестовой программы дает следующий вывод:

```
Contact - Id: 1, First name: Clarence, Last name: Ho, Birthday: 1980-07-30
Contact - Id: 2, First name: Scott, Last name: Tiger, Birthday: 1990-11-02
Contact - Id: 3, First name: John, Last name: Smith, Birthday: 1964-02-28
```

Из-за того, что в STS мы указали для свойств регистрации в журнале уровень DEBUG, в выводе консоли будет также присутствовать запрос, отправленный Spring (рис. 8.4).

```
JdbcTemplate: 635 - Executing prepared SQL query
JdbcTemplate: 570 - Executing prepared SQL statement [select id, first_name, last_name, birth_date from contact
```

Рис. 8.4. Вывод в STS при включенном уровне DEBUG

Теперь перейдем к реализации метода findByFirstName(), принимающего один именованный параметр. Подобно предыдущему примеру, мы создаем для операции класс SelectContactByFirstName, код которого показан в листинге 8.38.

Листинг 8.38. Класс SelectContactByFirstName

```
package com.apress.prospring3.ch8.dao.jdbc.annotation;

// Операторы импорта опущены.

public class SelectContactByFirstName extends MappingSqlQuery<Contact> {

    private static final String SQL_FIND_BY_FIRST_NAME =
        "select id, first_name, last_name, birth_date from contact
where first_name = :first_name";

    public SelectContactByFirstName(DataSource dataSource) {
        super(dataSource, SQL_FIND_BY_FIRST_NAME);
        super.declareParameter(new SqlParameter("first_name", Types.VARCHAR));
    }

    protected Contact mapRow(ResultSet rs, int rowNum) throws SQLException {
        Contact contact = new Contact();

        contact.setId(rs.getLong("id"));
        contact.setFirstName(rs.getString("first_name"));
        contact.setLastName(rs.getString("last_name"));
        contact.setBirthDate(rs.getDate("birth_date"));

        return contact;
    }
}
```

Класс SelectContactByFirstName похож на класс SelectAllContacts (отличия выделены полужирным). В нем используется другой SQL-оператор с именованным параметром first_name.

В методе конструктора вызывается метод declareParameter() (который непрямо унаследован от абстрактного класса org.springframework.jdbc.object.RdbmsOperation). Теперь давайте займемся реализацией метода findByFirstName() в классе JdbcContactDao. Соответствующий фрагмент кода показан в листинге 8.39.

Листинг 8.39. Реализация метода findByFirstName()

```
package com.apress.prospring3.ch8.dao.jdbc.annotation;

// Операторы импорта опущены.
```

```
@Repository("contactDao")
public class JdbcContactDao implements ContactDao {

    private SelectContactByFirstName selectContactByFirstName;

    @Resource(name="dataSource")
    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;

        selectAllContacts = new SelectAllContacts(dataSource);
        selectContactByFirstName = new SelectContactByFirstName(dataSource);
    }

    public List<Contact> findByFirstName(String firstName) {
        Map<String, Object> paramMap = new HashMap<String, Object>();
        paramMap.put("first_name", firstName);
        return selectContactByFirstName.executeByNamedParam(paramMap);
    }

    // Остальной код не показан.
}
```

В листинге 8.39 при внедрении источника данных создается экземпляр `SelectContactByFirstName` (строки кода выделены полужирным). После этого в методе `findByFirstName()` конструируется `HashMap` с именованными параметрами и значениями. Наконец, вызывается метод `executeByNamedParam()` (непрямо унаследованный от абстрактного класса `SqlQuery<T>`). Чтобы протестировать этот метод, добавьте фрагмент кода, показанный в листинге 8.40, в класс `AnnotationJdbcDaoSample`.

Листинг 8.40. Тестирование метода `findByFirstName()`

```
    // Найти контакты по имени.
    contacts = contactDao.findByFirstName("Clarence");
    listContacts(contacts);
```

Запуск этой программы дает следующий вывод из метода `findByFirstName()`:

```
Contact - Id: 1, First name: Clarence, Last name: Ho, Birthday: 1980-07-30
```

Здесь важно отметить один момент: `MappingSqlQuery<T>` подходит только для отображения одиночной строки на объект предметной области. Для вложенного объекта по-прежнему нужно применять `JdbcTemplate` с `ResultSetExtractor`, как это делалось в примере метода `findAllWithDetail()` в классе `JdbcTemplate`.

Обновление данных с использованием `SqlUpdate`

Для обновления данных Spring предоставляет класс `SqlUpdate`. В листинге 8.41 приведен код класса `UpdateContact`, который расширяет класс `SqlUpdate` для добавления операции обновления.

Листинг 8.41. Класс `UpdateContact`

```
package com.apress.prospring3.ch8.dao.jdbc.annotation;

import java.sql.Types;

import javax.sql.DataSource;

import org.springframework.jdbc.core.SqlParameter;
import org.springframework.jdbc.object.SqlUpdate;
```

```
public class UpdateContact extends SqlUpdate {

   private static final String SQL_UPDATE_CONTACT =
      "update contact set first_name=:first_name, last_name=:last_name,
birth_date=:birth_date where id=:id";

   public UpdateContact(DataSource dataSource) {
      super(dataSource, SQL_UPDATE_CONTACT);
      super.declareParameter(new SqlParameter("first_name", Types.VARCHAR));
      super.declareParameter(new SqlParameter("last_name", Types.VARCHAR));
      super.declareParameter(new SqlParameter("birth_date", Types.DATE));
      super.declareParameter(new SqlParameter("id", Types.INTEGER));
   }
}
```

Код в листинге 8.41 должен выглядеть знакомым. Здесь создается экземпляр класса `SqlUpdate` с запросом, а также объявляются именованные параметры.

В листинге 8.42 показана реализация метода `update()` в классе `JdbcContactDao`.

Листинг 8.42. Использование `SqlUpdate`

```
package com.apress.prospring3.ch8.dao.jdbc.annotation;

// Операторы импорта опущены.
@Repository("contactDao")
public class JdbcContactDao implements ContactDao {

   private UpdateContact updateContact;

   @Resource(name="dataSource")
   public void setDataSource(DataSource dataSource) {
      this.dataSource = dataSource;

      selectAllContacts = new SelectAllContacts(dataSource);
      selectContactByFirstName = new SelectContactByFirstName(dataSource);
      updateContact = new UpdateContact(dataSource);
   }

   public void update(Contact contact) {
      Map<String, Object> paramMap = new HashMap<String, Object>();
      paramMap.put("first_name", contact.getFirstName());
      paramMap.put("last_name", contact.getLastName());
      paramMap.put("birth_date", contact.getBirthDate());
      paramMap.put("id", contact.getId());
      updateContact.updateByNamedParam(paramMap);
      log.info("Existing contact updated with id: " + contact.getId());
   }

   // Остальной код не показан.

}
```

В листинге 8.42 при внедрении источника данных конструируется экземпляр `UpdateContact` (строки кода выделены полужирным). В методе `update()` на основе переданного объекта `Contact` создается `HashMap` из именованных параметров, после чего вызывается метод `updateByNamedParam()` для обновления записи о контакте. Чтобы протестировать операцию обновления, добавьте в класс `AnnotationJdbcDaoSample` фрагмент кода, показанный в листинге 8.43.

Листинг 8.43. Тестирование метода `update()`

```
Contact contact;

// Обновить контакт.
contact = new Contact();
contact.setId(1l);
contact.setFirstName("Clarence");
contact.setLastName("Peter");
contact.setBirthDate(
  new Date((new GregorianCalendar(1977, 10, 1)).getTime().getTime()));
contactDao.update(contact);
contacts = contactDao.findAll();
listContacts(contacts);
```

В листинге 8.43 мы просто создаем объект `Contact` и затем вызываем метод `update()`. Запуск этой программы дает следующий вывод из последнего метода `listContacts()`:

```
11:12:27,020  INFO 3.ch8.dao.jdbc.annotation.JdbcContactDao:
87 - Existing contact updated with id: 1
Contact - Id: 1, First name: Clarence, Last name: Peter, Birthday: 1977-11-01

Contact - Id: 2, First name: Scott, Last name: Tiger, Birthday: 1990-11-02

Contact - Id: 3, First name: John, Last name: Smith, Birthday: 1964-02-28
```

В выводе видно, что контакт с идентификатором 1 был соответствующим образом обновлен.

Вставка данных и извлечение сгенерированного ключа

Для вставки данных мы также используем класс `SqlUpdate`. Однако в этом случае присутствует один интересный момент: первичный ключ, т.е. столбец `ID`, будет доступен только после завершения оператора вставки, когда СУРБД сгенерирует значение, идентифицирующее запись. Столбец `ID` объявлен с атрибутом `AUTO_INCREMENT` и является первичным ключом, а это означает, что значение ему присваивается СУРБД во время выполнения операции вставки.

Если вы работаете с Oracle, то, скорее всего, сначала получите уникальный идентификатор из последовательности Oracle и затем запустите оператор вставки с помощью запроса. Но как в нашем случае можно извлечь ключ, сгенерированный СУРБД, после того, как запись вставлена?

В старых версиях JDBC это делалось не так-то просто. Например, для MySQL пришлось бы выдать SQL-оператор `select last_insert_id()`, а для Microsoft SQL Server — `select @@IDENTITY`.

К счастью, начиная с версии JDBC 3.0, появилась новая функциональная возможность, которая позволяет извлекать сгенерированные СУРБД ключи унифицированным образом. В листинге 8.37 приведена реализация метода `insert()`, которая также извлекает сгенерированный ключ для вставленной записи контакта. Она будет работать в большинстве баз данных (если только не во всех); просто удостоверьтесь, что используете драйвер JDBC, который совместим с JDBC 3.0 и последующими версиями.

Мы начинаем с создания класса `InsertContact` для операции вставки, расширяющего класс `SqlUpdate`. Код показан в листинге 8.44.

Листинг 8.44. Класс `InsertContact`

```
package com.apress.prospring3.ch8.dao.jdbc.annotation;

import java.sql.Types;

import javax.sql.DataSource;

import org.springframework.jdbc.core.SqlParameter;
import org.springframework.jdbc.object.SqlUpdate;

public class InsertContact extends SqlUpdate {

   private static final String SQL_INSERT_CONTACT =
      "insert into contact (first_name, last_name, birth_date) values
(:first_name, :last_name, :birth_date)";

   public InsertContact(DataSource dataSource) {
      super(dataSource, SQL_INSERT_CONTACT);
      super.declareParameter(new SqlParameter("first_name", Types.VARCHAR));
      super.declareParameter(new SqlParameter("last_name", Types.VARCHAR));
      super.declareParameter(new SqlParameter("birth_date", Types.DATE));
      super.setGeneratedKeysColumnNames(new String[] {"id"});
      super.setReturnGeneratedKeys(true);
   }
}
```

Класс `InsertContact` почти такой же, как и класс `UpdateContact`. В этом случае понадобится сделать две дополнительных вещи. При конструировании экземпляра класса `InsertContact` мы вызываем метод `SqlUpdate.setGeneratedKeysColumnNames()` для объявления имени столбца идентификации.

Метод `SqlUpdate.setReturnGeneratedKeys()` заставляет лежащий в основе драйвер JDBC извлечь сгенерированный ключ.

В листинге 8.45 приведена реализация метода `insert()` в классе `JdbcContactDao`.

Листинг 8.45. Использование `SqlUpdate` для операции вставки

```
package com.apress.prospring3.ch8.dao.jdbc.annotation;

// Операторы импорта опущены.
@Repository("contactDao")
public class JdbcContactDao implements ContactDao {
   private InsertContact insertContact;

   @Resource(name="dataSource")
   public void setDataSource(DataSource dataSource) {
      this.dataSource = dataSource;

      selectAllContacts = new SelectAllContacts(dataSource);
      selectContactByFirstName = new SelectContactByFirstName(dataSource);
      updateContact = new UpdateContact(dataSource);
      insertContact = new InsertContact(dataSource);
   }

   public void insert(Contact contact) {
      Map<String, Object> paramMap = new HashMap<String, Object>();
      paramMap.put("first_name", contact.getFirstName());
      paramMap.put("last_name", contact.getLastName());
      paramMap.put("birth_date", contact.getBirthDate());
```

```
        KeyHolder keyHolder = new GeneratedKeyHolder();
        insertContact.updateByNamedParam(paramMap, keyHolder);
        contact.setId(keyHolder.getKey().longValue());
        log.info("New contact inserted with id: " + contact.getId());
    }

    // Остальной код не показан.
}
```

В листинге 8.45 при внедрении источника данных создается экземпляр `InsertContact` (строки кода выделены полужирным). Внутри `insert()` мы также используем метод `SqlUpdate.updateByNamedParam()`. Однако мы передаем методу экземпляр `KeyHolder`, который должен будет хранить сгенерированный идентификатор. После вставки данных сгенерированный ключ можно будет извлечь из `KeyHolder`.

Чтобы протестировать операцию вставки, добавьте в класс `AnnotationJdbcDaoSample` фрагмент кода, показанный в листинге 8.46.

Листинг 8.46. Тестирование метода `insert()`

```
        // Вставить контакт.
        contact = new Contact();
        contact.setFirstName("Rod");
        contact.setLastName("Johnson");
        contact.setBirthDate(
          new Date((new GregorianCalendar(2001, 10, 1)).getTime().getTime()));
        contactDao.insert(contact);
        contacts = contactDao.findAll();
        listContacts(contacts);
```

Запуск программы дает следующий вывод из последнего метода `listContacts()`:

```
11:36:08,871  INFO 3.ch8.dao.jdbc.annotation.JdbcContactDao:
88 - New contact inserted with id: 4
Contact - Id: 1, First name: Clarence, Last name: Peter, Birthday: 1977-11-01

Contact - Id: 2, First name: Scott, Last name: Tiger, Birthday: 1990-11-02

Contact - Id: 3, First name: John, Last name: Smith, Birthday: 1964-02-28

Contact - Id: 4, First name: Rod, Last name: Johnson, Birthday: 2001-11-01
```

Как видите, новый контакт был корректно вставлен с идентификатором 4 и затем корректно извлечен.

Объединение операций в пакеты с помощью `BatchSqlUpdate`

Для пакетной операции используется класс `BatchSqlUpdate`. Работа с ним производится в основном так же, как с классом `SqlUpdate`; просто нужно предпринять несколько дополнительных действий. Для демонстрации применения этого класса добавим в интерфейс `ContactDao` новый метод:

```
public void insertWithDetail(Contact contact);
```

Новый метод `insertWithDetail()` будет вставлять информацию о контакте и его телефонах в базу данных.

Чтобы можно было вставить запись о телефоне, понадобится создать класс `InsertContactTelDetail`, который показан в листинге 8.47.

Листинг 8.47. Класс InsertContactTelDetail

```
package com.apress.prospring3.ch8.dao.jdbc.annotation;

import java.sql.Types;

import javax.sql.DataSource;

import org.springframework.jdbc.core.SqlParameter;
import org.springframework.jdbc.object.BatchSqlUpdate;

public class InsertContactTelDetail extends BatchSqlUpdate {

   private static final String SQL_INSERT_CONTACT_TEL =
      "insert into contact_tel_detail (contact_id, tel_type, tel_number)
values (:contact_id, :tel_type, :tel_number)";

   private static final int BATCH_SIZE = 10;

   public InsertContactTelDetail(DataSource dataSource) {
      super(dataSource, SQL_INSERT_CONTACT_TEL);
      declareParameter(new SqlParameter("contact_id", Types.INTEGER));
      declareParameter(new SqlParameter("tel_type", Types.VARCHAR));
      declareParameter(new SqlParameter("tel_number", Types.VARCHAR));
      setBatchSize(BATCH_SIZE);
   }
}
```

Обратите внимание, что в конструкторе мы вызываем метод `BatchSqlUpdate.setBatchSize()` для установки размера пакета для операции вставки JDBC.

В листинге 8.48 приведена реализация метода `insertWithDetail()` в классе `JdbcContactDao`.

Листинг 8.48. Пакетная SQL-операция обновления

```
package com.apress.prospring3.ch8.dao.jdbc.annotation;

// Операторы импорта опущены.
@Repository("contactDao")
public class JdbcContactDao implements ContactDao {

   private InsertContactTelDetail insertContactTelDetail;

   public void insertWithDetail(Contact contact) {

      insertContactTelDetail = new InsertContactTelDetail(dataSource);

      // Вставить контакт.
      Map<String, Object> paramMap = new HashMap<String, Object>();
      paramMap.put("first_name", contact.getFirstName());
      paramMap.put("last_name", contact.getLastName());
      paramMap.put("birth_date", contact.getBirthDate());
      KeyHolder keyHolder = new GeneratedKeyHolder();
      insertContact.updateByNamedParam(paramMap, keyHolder);
      contact.setId(keyHolder.getKey().longValue());
      log.info("New contact inserted with id: " + contact.getId());

      // Пакетная операция вставки телефонов для контакта.
      List<ContactTelDetail> contactTelDetails =
         contact.getContactTelDetails();
      if (contactTelDetails != null) {
         for (ContactTelDetail contactTelDetail: contactTelDetails) {
            paramMap = new HashMap<String, Object>();
```

```
            paramMap.put("contact_id", contact.getId());
            paramMap.put("tel_type", contactTelDetail.getTelType());
            paramMap.put("tel_number", contactTelDetail.getTelNumber());
            insertContactTelDetail.updateByNamedParam(paramMap);
        }
    }
    insertContactTelDetail.flush();
}

public List<Contact> findAllWithDetail() {
    JdbcTemplate jdbcTemplate = new JdbcTemplate(getDataSource());
    String sql = "select c.id, c.first_name, c.last_name, c.birth_date" +
", t.id as contact_tel_id, t.tel_type, t.tel_number from contact c " +
"left join contact_tel_detail t on c.id = t.contact_id";
    return jdbcTemplate.query(sql, new ContactWithDetailExtractor());
}

private static final class ContactWithDetailExtractor
    implements ResultSetExtractor<List<Contact>> {

    public List<Contact> extractData(ResultSet rs) throws
        SQLException, DataAccessException {

        Map<Long, Contact> map = new HashMap<Long, Contact>();
        Contact contact = null;
        while (rs.next()) {
            Long id = rs.getLong("id");
            contact = map.get(id);
           if (contact == null) {  // новая запись контакта
               contact = new Contact();
               contact.setId(id);
               contact.setFirstName(rs.getString("first_name"));
                contact.setLastName(rs.getString("last_name"));
                contact.setBirthDate(rs.getDate("birth_date"));
                contact.setContactTelDetails(new ArrayList<ContactTelDetail>());
                map.put(id, contact);
            }

           // Обработать телефоны контакта (если есть).
           Long contactTelDetailId = rs.getLong("contact_tel_id");
           if (contactTelDetailId > 0) {
               ContactTelDetail contactTelDetail = new ContactTelDetail();
               contactTelDetail.setId(contactTelDetailId);
               contactTelDetail.setContactId(id);
               contactTelDetail.setTelType(rs.getString("tel_type"));
               contactTelDetail.setTelNumber(rs.getString("tel_number"));
               contact.getContactTelDetails().add(contactTelDetail);
           }
        }
        return new ArrayList<Contact> (map.values());
    }
}
// Остальной код не показан.
}
```

Как показано в листинге 8.48, каждый раз, когда метод `insertWithDetail()` вызывается, создается новый экземпляр `InsertContactTelDetail`. Причина в том, что класс `BatchSqlUpdate` не является безопасным в отношении потоков. Далее он используется

так же, как SqlUpdate. Однако класс BatchSqlUpdate будет помещать операции вставки в очередь и отправлять их базе данных в виде пакета. Каждый раз, когда количество записей равно размеру пакета, Spring запускает в базе данных операцию групповой вставки для ожидающих записей. С другой стороны, по завершении мы вызываем метод BatchSqlUpdate.flush(), чтобы заставить Spring сбросить все ожидающие операции (т.е. операции вставки, находящиеся в очереди, количество которых пока не достигло размера пакета). Наконец, мы проходим в цикле по списку объектов ContactTelDetail в объекте Contact и вызываем метод BatchSqlUpdate.updateByNamedParam().

Для упрощения тестирования был также реализован метод findAllWithDetail(). В листинге 8.49 приведен фрагмент кода, который нужно добавить в класс AnnotationJdbcDaoSample, чтобы протестировать пакетную операцию вставки.

Листинг 8.49. Тестирование метода insertWithDetail()

```
// Вставить контакт вместе с деталями.
contact = new Contact();
contact.setFirstName("Michael");
contact.setLastName("Jackson");
contact.setBirthDate(
  new Date((new GregorianCalendar(1964, 10, 1)).getTime().getTime()));

List<ContactTelDetail> contactTelDetails =
  new ArrayList<ContactTelDetail>();

ContactTelDetail contactTelDetail = new ContactTelDetail();
contactTelDetail.setTelType("Home");
contactTelDetail.setTelNumber("11111111");
contactTelDetails.add(contactTelDetail);
contactTelDetail = new ContactTelDetail();
contactTelDetail.setTelType("Mobile");
contactTelDetail.setTelNumber("22222222");
contactTelDetails.add(contactTelDetail);
contact.setContactTelDetails(contactTelDetails);
contactDao.insertWithDetail(contact);
contacts = contactDao.findAllWithDetail();
listContacts(contacts);
```

Запуск этой программы даст следующий вывод из последнего метода listContacts():

```
Contact - Id: 1, First name: Clarence, Last name: Peter, Birthday: 1977-11-01
---Contact Tel Detail - Id: 2, Contact id: 1, Type: Home, Number: 1234567890
---Contact Tel Detail - Id: 1, Contact id: 1, Type: Mobile, Number: 1234567890

Contact - Id: 2, First name: Scott, Last name: Tiger, Birthday: 1990-11-02
---Contact Tel Detail - Id: 3, Contact id: 2, Type: Home, Number: 1234567890

Contact - Id: 3, First name: John, Last name: Smith, Birthday: 1964-02-28

Contact - Id: 4, First name: Rod, Last name: Johnson, Birthday: 2001-11-01

Contact - Id: 5, First name: Michael, Last name: Jackson, Birthday: 1964-11-01
---Contact Tel Detail - Id: 4, Contact id: 5, Type: Home, Number: 11111111
---Contact Tel Detail - Id: 5, Contact id: 5, Type: Mobile, Number: 22222222
```

Как видите, новые контакты со сведениями о телефонах были вставлены в базу данных.

Вызов хранимых функций с использованием SqlFunction

Платформа Spring также предлагает несколько классов, предназначенных для упрощения запуска хранимых процедур/функций с применением JDBC. В этом разделе мы рассмотрим простую функцию, использующую класс SqlFunction для вызова SQL-функции в базе данных. В качестве примера будет выбрана база данных MySQL; мы создадим хранимую функцию и вызовем ее с помощью класса SqlFunction<T>.

Мы предполагаем, что имеется база данных MySQL со схемой prospring3_ch8, а также именем пользователя и паролем, которые установлены в prospring3 (та же самая база данных, которая применялась в примере из раздела "Исследование инфраструктуры JDBC"). Создадим хранимую функцию по имени getFirstNameById(), которая принимает идентификатор контакта и возвращает имя контакта. В листинге 8.50 показан сценарий для создания хранимой функции в MySQL (store-function.sql). Запустите этот сценарий в указанной выше базе данных MySQL.

Листинг 8.50. Хранимая функция для MySQL

```
DELIMITER //
CREATE FUNCTION getFirstNameById(in_id INT)
   RETURNS VARCHAR(60)
BEGIN
   RETURN (SELECT first_name FROM contact WHERE id = in_id);
END //
DELIMITER ;
```

Хранимая функция должна быть самоочевидной. Она просто принимает идентификатор и возвращает имя из записи контакта с этим идентификатором.

Давайте создадим для этого примера новый интерфейс по имени ContactSfDao. Его код приведен в листинге 8.51.

Листинг 8.51. Интерфейс ContactSfDao

```
package com.apress.prospring3.ch8.dao;

public interface ContactSfDao {

   public String getFirstNameById(Long id);

}
```

Следующий шаг заключается в создании класса SfFirstNameById, расширяющего класс SqlFunction<T>, который будет представлять операцию вызова хранимой функции. Код для этого класса показан в листинге 8.52.

Листинг 8.52. Интерфейс SfFirstNameById

```
package com.apress.prospring3.ch8.dao.jdbc.annotation;

import java.sql.Types;

import javax.sql.DataSource;

import org.springframework.jdbc.core.SqlParameter;
import org.springframework.jdbc.object.SqlFunction;

public class SfFirstNameById extends SqlFunction<String> {

   private static final String SQL = "select getfirstnamebyid(?)";
```

```
    public SfFirstNameById(DataSource dataSource) {
        super(dataSource, SQL);
        declareParameter(new SqlParameter(Types.INTEGER));
        compile();
    }
}
```

Класс в листинге 8.52 расширяет `SqlFunction<T>` и передает тип `String`, который указывает возвращаемый тип функции. Затем мы объявляем строку SQL для вызова хранимой функции MySQL. После этого в конструкторе объявляется параметр и формируется операция. Теперь класс готов для использования в качестве класса реализации. В листинге 8.53 показан класс `JdbcContactSfDao`, который реализует интерфейс `ContactSfDao`.

Листинг 8.53. Класс `JdbcContactSfDao`

```
package com.apress.prospring3.ch8.dao.jdbc.annotation;

import java.util.List;

import javax.annotation.Resource;
import javax.sql.DataSource;

import org.springframework.stereotype.Repository;

import com.apress.prospring3.ch8.dao.ContactSfDao;

@Repository("contactSfDao")
public class JdbcContactSfDao implements ContactSfDao {

    private DataSource dataSource;

    private SfFirstNameById sfFirstNameById;

    @Resource(name="dataSource")
    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;

        sfFirstNameById = new SfFirstNameById(dataSource);
    }

    public DataSource getDataSource() {
        return dataSource;
    }

    public String getFirstNameById(Long id) {
        List<String> result = sfFirstNameById.execute(id);
        return result.get(0);
    }
}
```

В листинге 8.53 при внедрении источника данных создается экземпляр `SfFirstNameById`. После этого внутри `getFirstNameById()` вызывается метод `execute()` с передачей ему идентификатора контакта. Этот вызов вернет список `String`, из которого нужен лишь первый элемент, поскольку результирующий набор должен содержать только одну запись.

В листинге 8.54 приведено содержимое файла конфигурации Spring для подключения к MySQL (`app-context-sf.xml`).

Листинг 8.54. Конфигурация Spring для MySQL

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:context="http://www.springframework.org/schema/context"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context-3.1.xsd">

   <import resource="datasource-dbcp.xml"/>

   <context:component-scan base-package=
        "com.apress.prospring3.ch8.dao.jdbc.annotation"/>

   <context:annotation-config/>
</beans>
```

В листинге 8.54 импортируется файл `datasource-dbcp.xml`, который содержит конфигурацию источника данных для базы данных MySQL. Для запуска программы к проекту должна быть добавлена зависимость `commons-dbcp`, как показано в табл. 8.5.

Таблица 8.5. Зависимость для `commons-dbcp`

Идентификатор группы	Идентификатор артефакта	Версия	Описание
`commons-dbcp`	`commons-dbcp`	1.4	Библиотека поддержки пула подключений к базе данных `commons-dbcp` из Apache

В листинге 8.55 показана тестовая программа.

Листинг 8.55. Тестирование хранимой функции в MySQL

```
package com.apress.prospring3.ch8;

import org.springframework.context.support.GenericXmlApplicationContext;

import com.apress.prospring3.ch8.dao.ContactSfDao;

public class JdbcContactSfDaoSample {

  public static void main(String[] args) {

    GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
    ctx.load("classpath:app-context-sf.xml");
    ctx.refresh();

    ContactSfDao contactSfDao = ctx.getBean("contactSfDao", ContactSfDao.class);

    System.out.println(contactSfDao.getFirstNameById(1l));
  }
}
```

В тестовой программе хранимой функции передается идентификатор, равный 1. Это вернет Clarence в качестве имени, если ранее для базы данных MySQL запускался сценарий `test-data.sql`. В результате выполнения программы получен следующий вывод:

```
15:16:11,990 DEBUG g.springframework.jdbc.core.JdbcTemplate:
635 - Executing prepared SQL query
15:16:11,991 DEBUG g.springframework.jdbc.core.JdbcTemplate:
570 - Executing prepared SQL statement [select firstnamebyid(?)]
15:16:11,998 DEBUG ramework.jdbc.datasource.DataSourceUtils:
110 - Fetching JDBC Connection from DataSource
15:16:12,289 DEBUG ramework.jdbc.datasource.DataSourceUtils:
332 - Returning JDBC Connection to DataSource
Clarence
```

В выводе видно, что имя было извлечено корректно.

Здесь был представлен лишь простой пример для демонстрации функций модуля Spring JDBC. Платформа Spring также предлагает и другие классы (например, `StoredProcedure`), предназначенные для вызова сложных хранимых процедур, которые возвращают составные типы данных. Если вас интересует доступ к хранимым процедурам с использованием JDBC, мы рекомендуем обратиться к справочному руководству Spring.

Использование Java-конфигурации

На тот случай, если вы предпочитаете пользоваться конфигурацией на основе Java-классов, а не XML-конфигурацией, в листинге 8.56 показан класс конфигурации Spring.

Листинг 8.56. Пример использования Java-конфигурации

```
package com.apress.prospring3.ch8.javaconfig;

import javax.sql.DataSource;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabase;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseBuilder;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType;

@Configuration
@ComponentScan(basePackages="com.apress.prospring3.ch8.dao.jdbc.annotation")
public class AppConfig {

   @Bean
   public DataSource dataSource() {
      EmbeddedDatabaseBuilder builder = new EmbeddedDatabaseBuilder();
      EmbeddedDatabase db = builder.setType(EmbeddedDatabaseType.H2).
         addScript("schema.sql").
         addScript("test-data.sql").build();
      return db;
   }
}
```

В приведенном листинге мы применяем `EmbeddedDatabaseBuilder` для конструирования встроенной базы данных H2; эффект будет тем же самым, как и при использовании дескриптора `<jdbc:embedded-database>` в XML-конфигурации. Можно также с помощью аннотации `@Profile` указать, что данная конфигурация предназначена только для конкретной среды.

Проект Spring Data: расширения JDBC

Как упоминалось в начале этой главы, технологии баз данных развивались настолько быстрыми темпами, порождая огромное количество специализированных баз данных, что в наши дни СУРБД не является единственным выбором при разработке приложений. В ответ на эту эволюцию технологий баз данных, а также с целью удовлетворения потребностей сообщества разработчиков, был организован проект Spring Data (`www.springsource.org/spring-data`). Главная цель этого проекта — предоставление удобных расширений к ключевой функциональности доступа к данным Spring для решения потребностей разработчиков, которые взаимодействуют с базами данных, отличными от реляционных. Кроме того, также предоставляются расширенные средства поддержки стандартов доступа к данным (например, JDBC и JPA).

Проект Spring Data предлагает множество расширений. Одно из расширений стоит отметить особо — JDBC Extensions (`www.springsource.org/spring-data/jdbc-extensions`). Как вы уже наверняка догадались, это расширение предоставляет ряд средств для упрощения разработки приложений JDBC с использованием Spring.

На момент написания этой книги первый выпуск (версия 1.0.0) находился на промежуточном этапе разработке. Ниже перечислены основные функциональные возможности этого расширения.

- *Поддержка QueryDSL.* Упомянутый здесь QueryDSL (`www.querydsl.com`) — это специфичный для предметной области язык, который предоставляет инфраструктуру для разработки запросов, безопасных к типам. Расширение JDBC Extensions в проекте Spring Data предлагает класс `QueryDslJdbcTemplate`, упрощающий написание приложений JDBC за счет применения QueryDSL вместо SQL-операторов.
- *Расширенная поддержка Oracle Database.* Расширение JDBC Extensions делает доступными множество развитых средств для пользователей Oracle Database. С точки зрения подключений к базе данных оно поддерживает параметры сеансов, специфичные для Oracle, а также технологию быстрого устранения отказа подключения (Fast Connection Failover) при работе с Oracle RAC. Также предлагаются классы, которые интегрируются с расширенной поддержкой очередей Oracle (Oracle Advanced Queueing). С точки зрения типов данных обеспечивается встроенная поддержка XML-типов Oracle, `STRUCT` и `ARRAY` и т.д.

Если вы занимаетесь разработкой приложений JDBC, используя Spring с Oracle Database, определенно стоит взглянуть на JDBC Extensions.

Соображения по поводу использования JDBC

Из предыдущих обсуждений вы должны были понять, что Spring может существенно упростить разработку, когда для взаимодействия с лежащей в основе СУРБД нужно применять JDBC. Тем не менее, по-прежнему приходится писать много кода, особенно при трансформации результирующих наборов в соответствующие объекты предметной области.

Для JDBC разработано множество библиотек с открытым кодом, призванных ликвидировать разрыв между структурой реляционных данных и объектно-ориентированной моделью Java. Например, MyBatis (ранее известная как iBATIS) является популярной инфраструктурой отображения данных, которая также основана на SQL-отображении. MyBatis позволяет отображать объекты с хранимыми процедурами или запросами на XML-файл описателя (Java-аннотации тоже поддерживаются). Подобно Spring, инфраструктура MyBatis предоставляет декларативный способ для отображения запросов на

объекты, значительно сокращая время на обслуживание SQL-запросов, которые могут быть разбросаны по различным классам DAO.

Доступны также и другие инфраструктуры объектно-реляционного отображения, которые ориентированы на объектную модель, а не на запрос. В число популярных инфраструктур входят Hibernate, EclipseLink (также называется TopLink) и OpenJPA. Все они соответствуют спецификации JPA в JCP.

За последние годы инструменты и инфраструктуры объектно-реляционного отображения стали намного более зрелыми, поэтому большинство разработчиков будут пользоваться одним из таких инструментов, а не напрямую JDBC. Однако в случаях, когда для обеспечения максимальной производительности необходим полный контроль над запросом, отправляемым в базу данных (к примеру, когда применяется иерархический запрос в Oracle), Spring JDBC оказывается действительно жизнеспособным вариантом. Вдобавок при использовании Spring появляется преимущество в том, что можно смешивать различные технологии доступа к данным. Например, можно применять Hibernate в качестве главного инструмента объектно-реляционного отображения, а JDBC — как дополнение для некоторой логики сложных запросов или пакетных операций; их допускается смешивать в рамках одиночной бизнес-операции и затем помещать в одну и ту же транзакцию базы данных. Платформа Spring помогает легко справляться с ситуациями подобного рода.

Резюме

В этой главе было показано, как использовать Spring для упрощения программирования JDBC. Вы научились подключаться к базе данных и выполнять операции выборки, обновления, удаления и вставки, а также вызывать хранимые функции. Подробно обсуждался ключевой класс Spring JDBC по имени `JdbcTemplate`. Кроме того, были рассмотрены другие классы Spring, построенные поверх `JdbcTemplate`, которые помогают моделировать различные операции JDBC.

В последующих главах речь пойдет о том, как применять Spring совместно с популярными технологиями объектно-реляционного отображения для разработки логики доступа к данным.

ГЛАВА 9

Использование Hibernate в Spring

В предыдущей главе было показано, как использовать JDBC в Spring-приложениях. Тем не менее, хотя Spring обеспечивает существенное упрощение разработки для JDBC, по-прежнему приходится писать большой объем кода.

В этой главе мы рассмотрим одну из библиотек объектно-реляционного отображения (object-relational mapping — ORM), которая имеет широкую поддержку в Spring — Hibernate.

Если вы обладаете опытом разработки приложений с доступом данных, используя сущностные бины EJB (до появления EJB 3.0), то наверняка должны помнить этот довольно болезненный процесс. Утомительное конфигурирование отображения, установление границ транзакций и наличие в каждом бине множества стереотипного кода, предназначенного для управления его жизненным циклом, значительно снижало продуктивность разработки корпоративных Java-приложений.

Поскольку платформа Spring была создана для охвата разработки на базе POJO и поддержки декларативной конфигурации вместо трудной и неуклюжей настройки EJB, в сообществе разработчиков начали понимать, что более простая, облегченная и основанная на POJO инфраструктура может снизить сложность написания логики доступа к данным. С тех пор появилось множество разных библиотек, которые все вместе называют *библиотеками ORM*. Главная предназначение библиотеки ORM — ликвидация разрыва между структурой реляционных данных в СУРБД и объектно-ориентированной моделью в Java, что позволит разработчикам сосредоточиться на программировании с применением объектной модели и в то же самое время легко выполнять действия, касающиеся постоянства.

Среди всех библиотек ORM, доступных в сообществе открытого кода, Hibernate является одной из наиболее успешных. Ее функциональные возможности, такие как подход на основе POJO, простота разработки и поддержка определений сложных отношений, завоевали сердца многих участников сообщества передовых разработчиков на Java.

Популярность Hibernate также повлияла на процесс JCP, в рамках которого была разработана спецификация Java-объектов данных (Java Data Objects — JDO) в качестве одной из стандартных технологий ORM в Java EE. Начиная с EJB 3.0, сущностный бин EJB был даже заменен интерфейсом Java Persistence API (JPA), на многие идеи которого оказали влияние популярные библиотеки ORM, в том числе Hibernate, TopLink и JDO.

Отношения между Hibernate и JPA очень тесные. Гэвин Кинг (Gavin King), основатель Hibernate, представил JBoss как один из членов экспертной группы JCP в определении спецификации JPA. Начиная с версии 3.2, Hibernate предоставляет реализацию JPA.

Таким образом, при разработке приложений с помощью Hibernate можно выбирать между собственным API-интерфейсом Hibernate и JPA с Hibernate в качестве поставщика службы постоянства.

После обсуждения истории развития Hibernate в этой главе будет показано, как использовать Spring с Hibernate при разработке логики доступа к данным. Hibernate является настолько обширной библиотекой ORM, что раскрытие каждого ее аспекта в единственной главе попросту невозможно; тематике Hibernate посвящены отдельные книги.

В этой главе рассматриваются базовые идеи и основные сценарии использования Hibernate в Spring. В частности, мы обсудим следующие темы.

- *Конфигурирование фабрики сеансов Hibernate.* Ключевая концепция Hibernate связана с интерфейсом `Session`, который управляется `SessionFactory`. Мы покажем, как конфигурировать фабрику сеансов Hibernate для работы в Spring-приложении.
- *Основные концепции объектно-реляционного отображения с использованием Hibernate.* Мы рассмотрим основные концепции, связанные с применением Hibernate для отображения POJO на структуру лежащей в основе реляционной базы данных. Также будут обсуждаться некоторые часто используемые отношения, включая “один ко многим” и “многие ко многим”.
- *Операции над данными.* Мы продемонстрируем несколько примеров выполнения операций над данными (запроса, вставки, обновления, удаления) с использованием Hibernate в среде Spring. При работе с библиотекой Hibernate мы в основном будем взаимодействовать с ее интерфейсом `Session`.

На заметку! При определении объектно-реляционных отображений Hibernate поддерживает два стиля конфигурирования. Один из них предусматривает помещение информации отображения в XML-файлы, а другой предполагает применение Java-аннотаций внутри сущностных классов (в мире ORM или JPA класс Java, который отображается на структуру лежащей в основе реляционной базы данных, называется *сущностным классом* (entity class)). В настоящее время намного более популярным является подход с аннотациями. Таким образом, в этой главе мы сосредоточим внимание на использовании подхода с аннотациями для объектно-реляционного отображения. Для аннотирования отображений будут применяться стандарты JPA (например, определенные в пакете `javax.persistence`), поскольку они взаимозаменяемы с собственными аннотациями Hibernate, что поможет при будущей миграции в среду JPA.

Создание служебного проекта Hibernate в STS

Как и Spring, инфраструктура Hibernate представляет собой большую библиотеку, упакованную в нескольких модулях (таких как Hibernate Core, Entity Manager и т.д.). Иногда трудно определить, какие модули Hibernate требуются приложению. К счастью, STS предоставляет простой способ создания общих типов проектов, основанных на Spring, в число которых входит и шаблонный проект с Hibernate.

Чтобы создать в STS проект, использующий Spring с поддержкой Hibernate, просто создайте новый проект с применением шаблона Simple Spring Hibernate Utility Project (Простой служебный проект Spring с поддержкой Hibernate), который можно найти, выбрав пункт меню New Project⇨Spring Template Project (Новый проект⇨Шаблонный проект Spring); открывшееся диалоговое окно New Template Project (Новый шаблонный проект) показано на рис. 9.1.

После этого введите имя проекта и имя пакета верхнего уровня, как показано на рис. 9.2.

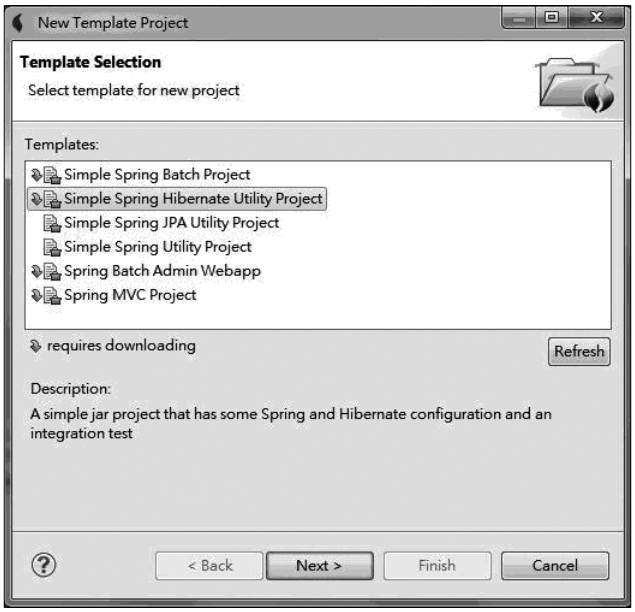


Рис. 9.1. Создание проекта, основанного на шаблоне Simple Spring Hibernate Utility Project, в STS

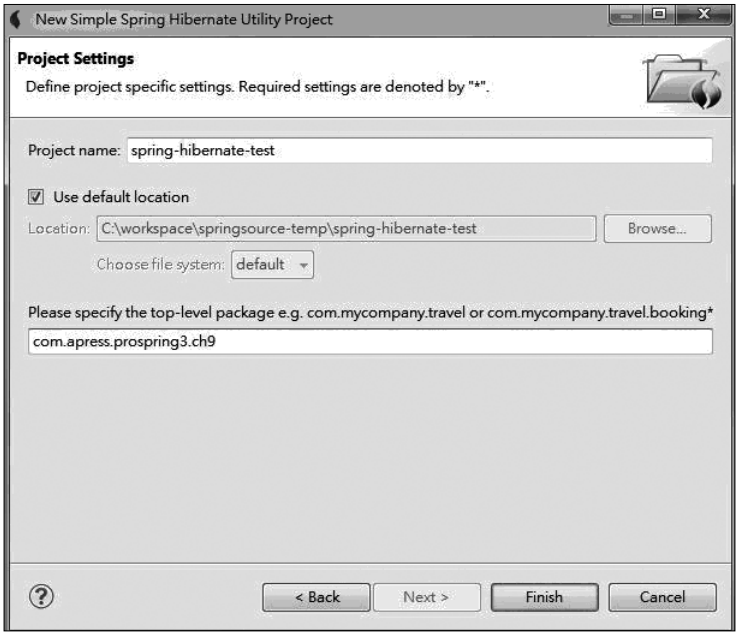


Рис. 9.2. Детали служебного проекта Spring Hibernate

По завершении STS создаст основанный на Maven проект с обязательными зависимостями. Если вы заглянете в файл `pom.xml` (файл объектной модели проекта Maven), находящийся в корневой папке проекта, то увидите, что в проект была добавлена зависимость. Соответствующий фрагмент кода показан в листинге 9.1.

Листинг 9.1. Зависимость для Hibernate

```
<dependencies>
   <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-entitymanager</artifactId>
      <version>3.6.0.Final</version>
   </dependency>
...
```

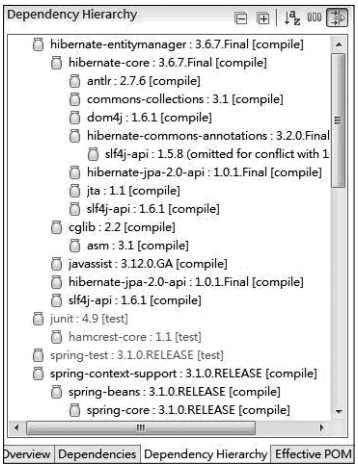


Рис. 9.3. Вкладка Dependency Hierarchy

Если сравнить это с простым служебным проектом Spring, то добавлена лишь одна зависимость. Однако, благодаря средству транзитивных зависимостей Maven, все другие обязательные зависимости будут автоматически обнаружены и добавлены Maven в проект. При открытии файла `pom.xml` в STS отображается редактор файлов POM, на вкладке Dependencies Hierarchy (Иерархия зависимостей) которого можно просмотреть остальные зависимости, требуемые Hibernate (рис. 9.3).

На рис. 9.3 видно, что модуль `hibernate-entitymanager` требует модуля `hibernate-core`, который, в свою очередь, требует модуля `hibernate-commons-annotations`, и т.д. Таким образом, вы быстро сможете выяснить, что было включено в проект.

Тем не менее, для производственной версии приложения по-прежнему может быть нужна точная настройка файла `pom.xml`, что поможет гарантировать включение только необходимых версий библиотек.

Модель данных для кода примера

В главе 8 была представлена простая модель данных для демонстрации примеров кода. В настоящей главе для демонстрации некоторых более сложных отношений мы расширим эту модель данных. Модель данных, которая будет использоваться в этой главе, показана на рис. 9.4.

В эту модель данных были добавлены две новых таблицы, `HOBBY` и `CONTACT_HOBBY_DETAIL` (таблица соединения), которые моделируют отношения “многие ко многим” между таблицами `CONTACT` и `HOBBY`. Кроме того, в таблицы `CONTACT` и `CONTACT_TEL_DETAIL` был добавлен столбец `VERSION`, предназначенный для оптимистичной блокировки, которая будет подробно обсуждаться позже. В примерах этой главы мы будем работать со встроенной базой данных H2, поэтому имя базы данных не требуется.

В листингах 9.2 и 9.3 приведены сценарии для создания схемы и наполнения тестовыми данными.

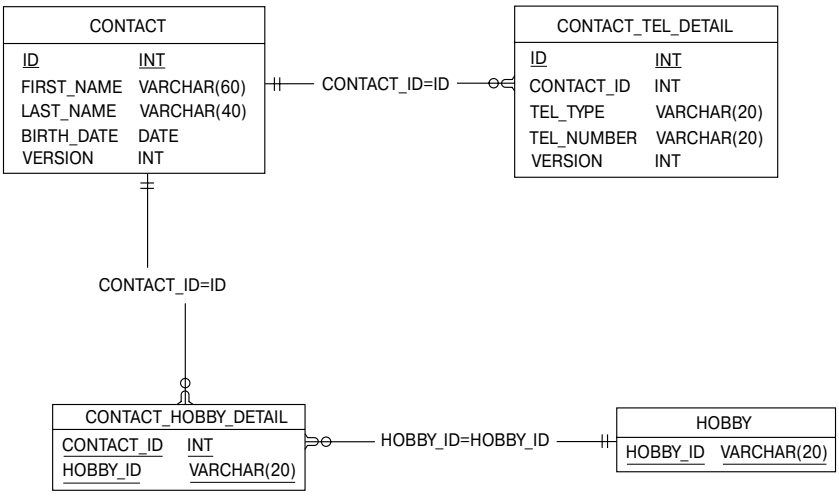


Рис. 9.4. Модель данных для демонстрации работы с Hibernate

Листинг 9.2. Сценарий для создания примера модели данных (`schema.sql`)

```
CREATE TABLE CONTACT (
   ID INT NOT NULL AUTO_INCREMENT
   , FIRST_NAME VARCHAR(60) NOT NULL
   , LAST_NAME VARCHAR(40) NOT NULL
   , BIRTH_DATE DATE
   , VERSION INT NOT NULL DEFAULT 0
   , UNIQUE UQ_CONTACT_1 (FIRST_NAME, LAST_NAME)
   , PRIMARY KEY (ID)
);
CREATE TABLE HOBBY (
    HOBBY_ID VARCHAR(20) NOT NULL
    , PRIMARY KEY (HOBBY_ID)
);
CREATE TABLE CONTACT_TEL_DETAIL (
ID INT NOT NULL AUTO_INCREMENT
    , CONTACT_ID INT NOT NULL
    , TEL_TYPE VARCHAR(20) NOT NULL
    , TEL_NUMBER VARCHAR(20) NOT NULL
    , VERSION INT NOT NULL DEFAULT 0
    , UNIQUE UQ_CONTACT_TEL_DETAIL_1 (CONTACT_ID, TEL_TYPE)
    , PRIMARY KEY (ID)
    , CONSTRAINT FK_CONTACT_TEL_DETAIL_1 FOREIGN KEY (CONTACT_ID)
       REFERENCES CONTACT (ID)
);
CREATE TABLE CONTACT_HOBBY_DETAIL (
    CONTACT_ID INT NOT NULL
    , HOBBY_ID VARCHAR(20) NOT NULL
    , PRIMARY KEY (CONTACT_ID, HOBBY_ID)
    , CONSTRAINT FK_CONTACT_HOBBY_DETAIL_1 FOREIGN KEY (CONTACT_ID)
       REFERENCES CONTACT (ID) ON DELETE CASCADE
    , CONSTRAINT FK_CONTACT_HOBBY_DETAIL_2 FOREIGN KEY (HOBBY_ID)
    REFERENCES HOBBY (HOBBY_ID)
);
```

Листинг 9.3. Сценарий для наполнения тестовыми данными (`test-data.sql`)

```
insert into contact (first_name, last_name, birth_date) values
↳ ('Clarence', 'Ho', '1980-07-30');
insert into contact (first_name, last_name, birth_date) values
↳ ('Scott', 'Tiger', '1990-11-02');
insert into contact (first_name, last_name, birth_date) values
↳ ('John', 'Smith', '1964-02-28');

insert into contact_tel_detail (contact_id, tel_type, tel_number) values
↳ (1, 'Mobile', '1234567890');
insert into contact_tel_detail (contact_id, tel_type, tel_number) values
↳ (1, 'Home', '1234567890');
insert into contact_tel_detail (contact_id, tel_type, tel_number) values
↳ (2, 'Home', '1234567890');

insert into hobby (hobby_id) values ('Swimming');
insert into hobby (hobby_id) values ('Jogging');
insert into hobby (hobby_id) values ('Programming');
insert into hobby (hobby_id) values ('Movies');
insert into hobby (hobby_id) values ('Reading');

insert into contact_hobby_detail(contact_id, hobby_id) values (1, 'Swimming');
insert into contact_hobby_detail(contact_id, hobby_id) values (1, 'Movies');
insert into contact_hobby_detail(contact_id, hobby_id) values (2, 'Swimming');
```

Конфигурирование фабрики сеансов Hibernate

Как упоминалось ранее, ключевая концепция Hibernate основана на интерфейсе `Session`, который получается из `SessionFactory`. В Spring доступно несколько классов для поддержки конфигурирования фабрики сеансов Hibernate в качестве бина Spring с заданными свойствами. Поскольку мы собираемся использовать стиль аннотаций, мы будем работать с классом `AnnotationSessionFactoryBean`. В листинге 9.4 показано содержимое соответствующего XML-файла конфигурации (`app-context.xml`).

Листинг 9.4. Конфигурация Spring для `AnnotationSessionFactoryBean`

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:context="http://www.springframework.org/schema/context"
   xmlns:tx="http://www.springframework.org/schema/tx"
   xmlns:jdbc="http://www.springframework.org/schema/jdbc"
   xsi:schemaLocation="
     http://www.springframework.org/schema/jdbc
     http://www.springframework.org/schema/jdbc/spring-jdbc-3.1.xsd
     http://www.springframework.org/schema/beans
     http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
     http://www.springframework.org/schema/tx
     http://www.springframework.org/schema/tx/spring-tx-3.1.xsd
     http://www.springframework.org/schema/context
     http://www.springframework.org/schema/context/spring-context-3.1.xsd">

   <jdbc:embedded-database id="dataSource" type="H2">
      <jdbc:script location="classpath:schema.sql"/>
      <jdbc:script location="classpath:test-data.sql"/>
   </jdbc:embedded-database>
```

```
   <bean id="transactionManager"
     class="org.springframework.orm.hibernate3.HibernateTransactionManager">
     <property name="sessionFactory" ref="sessionFactory"/>
   </bean>

   <tx:annotation-driven/>

   <context:component-scan base-package="com.apress.prospring3.ch9" />

   <bean id="sessionFactory"
class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean">
      <property name="dataSource" ref="dataSource"/>
      <property name="packagesToScan"
         value="com.apress.prospring3.ch9.domain"/>
      <property name="hibernateProperties">
        <props>
          <prop key="hibernate.dialect">org.hibernate.dialect.H2Dialect</prop>
          <prop key="hibernate.max_fetch_depth">3</prop>
          <prop key="hibernate.jdbc.fetch_size">50</prop>
          <prop key="hibernate.jdbc.batch_size">10</prop>
          <prop key="hibernate.show_sql">true</prop>
        </props>
      </property>
   </bean>
</beans>
```

В предыдущей конфигурации было объявлено множество бинов для обеспечения поддержки фабрики сеансов Hibernate. Ниже перечислены основные конфигурации.

- *Бин* `dataSource`. Мы объявляем источник данных со встроенной базой данных H2.
- *Бин* `transactionManager`. Фабрика сеансов Hibernate требует диспетчера транзакций для транзакционного доступа к данным. В Spring предоставляется диспетчер транзакций, специфичный для Hibernate 3 (`org.springframework.orm.hibernate3.HibernateTransactionManager`). Бин объявлен с идентификатором `transactionManager`. По умолчанию всегда, когда требуется диспетчер транзакций, Spring будет искать бин с именем `transactionManager` внутри `ApplicationContext`. Транзакции подробно рассматриваются в главе 13. Вдобавок мы объявляем дескриптор `<tx:annotation-driven>` для поддержки требований к установлению границ транзакций, используя аннотации.
- *Дескриптор* `<context:component-scan>`. Этот дескриптор должен уже быть хорошо знаком. Мы указываем Spring на необходимость сканирования компонентов в пакете `com.apress.prospring3.ch9`.
- *Бин* `sessionFactory`. Бин `sessionFactory` является самой важной частью. Так как мы пользуемся поддержкой аннотаций Hibernate, мы имеем дело с бином `AnnotationSessionFactoryBean`. Внутри этого бина определено множество свойств. Во-первых, как и можно было ожидать, мы должны внедрить бин источника данных в фабрику сеансов. Во-вторых, мы инструктируем Hibernate относительно сканирования объектов предметной области с аннотацией ORM в пакете `com.apress.prospring3.ch9.domain`. Наконец, свойство `hibernateProperties` предоставляет детали конфигурации для Hibernate. Существует множество конфигурационных параметров, но мы определяем только несколько важных свойств, которые должны предоставляться для каждого приложения. В табл. 9.1 перечислены основные конфигурационные параметры для фабрики сеансов Hibernate.

Таблица 9.1. Основные конфигурационные параметры

Свойство	Описание
hibernate.dialect	Диалект базы данных для запросов, которые должны использоваться Hibernate. Библиотека Hibernate поддерживает диалекты SQL для многих баз данных. Эти диалекты являются подклассами org.hibernate.dialect.Dialect. В число основных диалектов входят H2Dialect, Oracle10gDialect, PostgreSQLDialect, MySQLDialect, SQLServerDialect и т.д.
hibernate.max_fetch_depth	Объявляет "глубину" для внешних соединений, когда отображаемые объекты имеют ассоциации с другими отображенными объектами. Этот параметр позволяет предотвратить выборку Hibernate слишком большого объема данных при наличии множества вложенных ассоциаций. Обычно используется значение 3
hibernate.jdbc.fetch_size	Количество записей из лежащего в основе результирующего набора JDBC, который библиотека Hibernate должна использовать для извлечения записей из базы данных в каждой выборке. Например, запрос был отправлен в базу данных и результирующий набор содержит 500 записей. Если размер выборки равен 50, то для получения всех данных Hibernate понадобится выполнить 10 выборок
hibernate.jdbc.batch_size	Указывает Hibernate количество операций обновления, которые должны быть сгруппированы в пакет. Это очень удобно для выполнения пакетных операций в Hibernate. Очевидно, когда выполняется пакетное задание, обновляющее сотни записей, мы хотели бы, чтобы библиотека Hibernate сгруппировала запросы в пакеты, а не отправляла запросы обновления по одному за раз
hibernate.show_sql	Указывает, должна ли библиотека Hibernate выводить SQL-запросы в файл журнала или на консоль. Этот режим имеет смысл включать в среде разработки, потому что он помогает в тестировании и устранении ошибок

За полным списком свойств, поддерживаемых Hibernate, обращайтесь в справочное руководство по Hibernate (http://docs.jboss.org/hibernate/core/3.6/reference/en-US/html/session-configuration.html).

Объектно-реляционное отображение с использованием аннотаций Hibernate

Имея показанную ранее конфигурацию, следующий шаг заключается в моделировании сущностных Java-классов POJO и их отображение на структуру лежащей в основе базы данных.

Отображение можно реализовать с помощью двух подходов. При первом подходе сначала проектируется объектная модель, а затем на ее основе генерируются сценарии для базы данных. Например, для конфигурации фабрики сеансов можно передать свойство hibernate.hbm2ddl.auto, чтобы заставить Hibernate автоматически экспортировать DDL-сценарий схемы в базу данных.

Второй подход заключается в том, чтобы начать с модели данных, а затем построить объекты POJO для нужного отображения. Мы предпочитаем второй подход, поскольку он обеспечивает больший контроль над моделью данных, что очень полезно при оптимизации производительности доступа к данным. Предложенной ранее модели данных соответствует объектно-ориентированная модель, диаграмма классов которой показана на рис. 9.5.

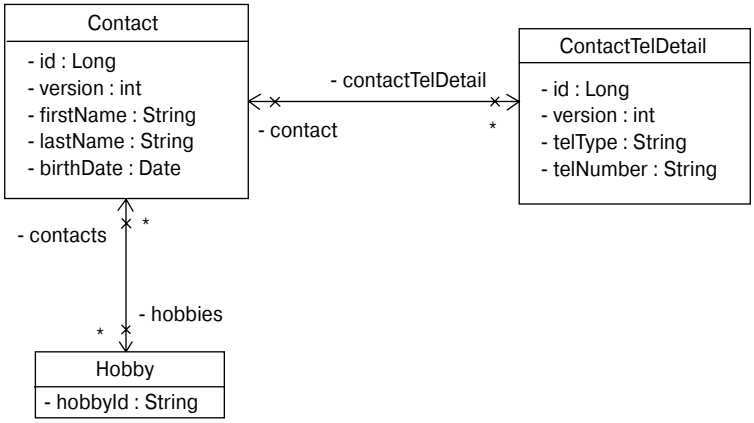


Рис. 9.5. Диаграмма классов для примера модели данных

Как видите, здесь имеется отношение “один ко многим” между объектами `Contact` и `ContactTelDetail`, а также отношение “многие ко многим” между объектами `Contact` и `Hobby`.

Простое отображение

Давайте начнем с отображения простых атрибутов. Как было показано в диаграмме классов, простые атрибуты присутствуют во всех трех классах. Займемся ими по очереди. В листинге 9.5 представлен класс `Contact` с аннотациями отображения.

Листинг 9.5. Класс `Contact`

```
package com.apress.prospring3.ch9.domain;

import static javax.persistence.GenerationType.IDENTITY;

import java.io.Serializable;
import java.util.Date;
import java.util.HashSet;
import java.util.Set;
import javax.persistence.*;

@Entity
@Table(name = "contact")
public class Contact implements Serializable {

   private Long id;
   private int version;
   private String firstName;
   private String lastName;
   private Date birthDate;
```

```
    @Id
    @GeneratedValue(strategy = IDENTITY)
    @Column(name = "ID")
    public Long getId() {
        return this.id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    @Version
    @Column(name = "VERSION")
    public int getVersion() {
        return this.version;
    }

    public void setVersion(int version) {
        this.version = version;
    }

    @Column(name = "FIRST_NAME")
    public String getFirstName() {
        return this.firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    @Column(name = "LAST_NAME")
    public String getLastName() {
        return this.lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    @Temporal(TemporalType.DATE)
    @Column(name = "BIRTH_DATE")
    public Date getBirthDate() {
        return this.birthDate;
    }

    public void setBirthDate(Date birthDate) {
        this.birthDate = birthDate;
    }

    public String toString() {
        return "Contact - Id: " + id + ", First name: " + firstName
            + ", Last name: " + lastName + ", Birthday: " + birthDate;
    }
}
```

Сначала мы аннотируем тип с помощью `@Entity`, указывая, что это отображенный сущностный класс. Аннотация `@Table` определяет имя таблицы в базе данных, на которую эта сущность отображается. Каждый отображенный атрибут аннотируется посредством `@Column` с указанием имени столбца. В случае если имена типа и атрибутов совпадают с именами таблицы и столбцов, аннотации `@Table` и `@Column` можно не указывать.

Относительно отображения необходимо отметить несколько моментов.

- Атрибут даты рождения аннотирован с помощью `@Temporal` с назначением ему типа `TemporalType.DATE`. Это объясняется тем, что мы хотим отобразить тип данных из Java-даты (`java.util.Date`) на SQL-тип даты (`java.sql.Date`). В результате мы сможем обращаться к атрибуту `birthDate` в объекте `Contact`, используя `java.util.Date`, как обычно делается в приложении.
- Атрибут `id` аннотирован с помощью `@Id`. Это означает, что он является первичным ключом объекта. Hibernate будет использовать его как уникальный идентификатор при управлении экземплярами сущностей контактов в рамках сеанса. Кроме того, аннотация `@GeneratedValue` сообщает Hibernate, как сгенерировано значение `id`. Стратегия `IDENTITY` (мы можем применять ее прямо внутри аннотации, поскольку присутствует оператор `import static`) говорит о том, что `id` был сгенерирован СУРБД во время вставки (столбец `ID` таблицы `CONTACT` является первичным ключом с атрибутом `AUTO_INCREMENT`, т.е. его значение будет генерироваться и назначаться базой данных при выполнении операции вставки).
- Атрибут `version` аннотирован с помощью `@Version`. Это сообщает Hibernate о том, что мы хотим использовать механизм оптимистичной блокировки, применяя для управления атрибут `version`. Каждый раз, когда инфраструктура Hibernate обновляет запись, она сравнивает версию экземпляра сущности с версией записи в базе данных. Если версии совпадают, значит, данные перед этим не обновлялись, поэтому Hibernate обновит данные и увеличит столбец версии. Однако если версии отличаются, это значит, что кто-то ранее обновил запись, и Hibernate сгенерирует исключение `StaleObjectStateException`, которое Spring будет транслировать в `HibernateOptimisticLockingFailureException`. В приведенном примере для управления версиями используется целочисленное значение. Помимо целого числа, Hibernate также поддерживает временн*ую* метку. Тем не менее, рекомендуется применять для управления версиями именно целочисленное значение, т.к. в этом случае Hibernate будет всегда увеличивать значение версии на 1 после каждого обновления. Когда же используется временная метка, после каждого обновления Hibernate будет заменять значение этой метки текущим показанием времени. Временные метки менее безопасны, поскольку две параллельных транзакции могут загрузить и обновить один и тот же элемент данных практически одновременно, в рамках миллисекунды.

В листинге 9.6 показан еще один отображенный объект — `ContactTelDetail`.

Листинг 9.6. Класс `ContactTelDetail`

```
package com.apress.prospring3.ch9.domain;

// Операторы импорта опущены.
@Entity
@Table(name = "contact_tel_detail")
public class ContactTelDetail implements Serializable {

   private Long id;
   private int version;
   private String telType;
   private String telNumber;

   public ContactTelDetail() {
   }
```

```
    public ContactTelDetail(String telType, String telNumber) {
        this.telType = telType;
        this.telNumber = telNumber;
    }

    @Id
    @GeneratedValue(strategy = IDENTITY)
    @Column(name = "ID")
    public Long getId() {
        return this.id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    @Version
    @Column(name = "VERSION")
    public int getVersion() {
        return this.version;
    }

    public void setVersion(int version) {
        this.version = version;
    }

    @Column(name = "TEL_TYPE")
    public String getTelType() {
        return this.telType;
    }

    public void setTelType(String telType) {
        this.telType = telType;
    }

    @Column(name = "TEL_NUMBER")
    public String getTelNumber() {
        return this.telNumber;
    }

    public void setTelNumber(String telNumber) {
        this.telNumber = telNumber;
    }

}
```

В этом классе нет ничего особо примечательного. В листинге 9.7 представлен класс `Hobby`.

Листинг 9.7. Класс `Hobby`

```
package com.apress.prospring3.ch9.domain;

// Операторы импорта опущены.
@Entity
@Table(name = "hobby")
public class Hobby implements Serializable {

    private String hobbyId;

    @Id
    @Column(name = "HOBBY_ID")
```

```
    public String getHobbyId() {
        return this.hobbyId;
    }

    public void setHobbyId(String hobbyId) {
        this.hobbyId = hobbyId;
    }

    public String toString() {
        return "Hobby :" + getHobbyId();
    }
}
```

Здесь также нет ничего нового. Давайте теперь посмотрим, как моделируются ассоциации между классами `Contact` b `ContactTelDetail`.

Отображение “один ко многим”

Инфраструктура Hibernate обладает возможностью моделировать много разновидностей ассоциаций. Наиболее общими ассоциациями являются “один ко многим” и “многие ко многим”. Каждый контакт будет иметь ноль или больше телефонных номеров, так что здесь получается ассоциация “один ко многим” (в терминах ORM ассоциация “один ко многим” используется для моделирования отношений “ноль ко многим” и “один ко многим” в структуре данных). В листинге 9.8 приведен фрагмент кода для класса `Contact`, отвечающий за отображение с классом `ContactTelDetail`.

Листинг 9.8. Ассоциация “один ко многим”

```
package com.apress.prospring3.ch9.domain;

// Операторы импорта опущены.
@Entity
@Table(name = "contact")
public class Contact implements Serializable {

    // Остальной код не показан.
    private Set<ContactTelDetail> contactTelDetails =
        new HashSet<ContactTelDetail>();

    @OneToMany(mappedBy = "contact", cascade=CascadeType.ALL,
        orphanRemoval=true)
    public Set<ContactTelDetail> getContactTelDetails() {
        return this.contactTelDetails;
    }

    public void setContactTelDetails(Set<ContactTelDetail> contactTelDetails)
{
        this.contactTelDetails = contactTelDetails;
    }

    public void addContactTelDetail(ContactTelDetail contactTelDetail) {
        contactTelDetail.setContact(this);
        getContactTelDetails().add(contactTelDetail);
    }

    public void removeContactTelDetail(ContactTelDetail contactTelDetail) {
        getContactTelDetails().remove(contactTelDetail);
    }

}
```

Как показано в листинге 9.8, метод извлечения атрибута contactTelDetails аннотирован с помощью аннотации @OneToMany, указывающей на наличие отношения "один ко многим" с классом ContactTelDetail. Этой аннотации передается несколько атрибутов. Атрибут mappedBy задает свойство в классе ContactTelDetail, которое предоставляет ассоциацию (т.е. связано с определением внешнего ключа в таблице CONTACT_TEL_DETAIL). Атрибут cascade означает, что операция обновления должна распространяться на дочерние записи. Атрибут orphanRemoval указывает, что после обновления деталей телефонных номеров контакта записи, которые больше не существуют в наборе, должны быть удалены из базы данных. В листинге 9.9 приведен соответствующий фрагмент кода в классе ContactTelDetail, предназначенный для отображения ассоциации.

Листинг 9.9. Отображение "один ко многим" в ContactTelDetail

```
package com.apress.prospring3.ch9.domain;

// Операторы импорта опущены.
@Entity
@Table(name = "contact_tel_detail")
public class ContactTelDetail implements Serializable {

   // Остальной код не показан.
   private Contact contact;

   @ManyToOne
   @JoinColumn(name = "CONTACT_ID")
   public Contact getContact() {
      return this.contact;
   }

   public void setContact(Contact contact) {
      this.contact = contact;
   }

   public String toString() {
   return "Contact Tel Detail - Id: " + id + ", Contact id: "
         + getContact().getId() + ", Type: "
         + telType + ", Number: " + telNumber;
   }
}
```

В предыдущем листинге мы аннотировали метод извлечения атрибута contact с помощью аннотации @ManyToOne, которая задает другую сторону ассоциации с Contact. Мы также указали аннотацию @JoinColumn с именем столбца внешнего ключа. Наконец, метод toString() был переопределен для упрощения тестирования кода примера в будущем.

Отображение "многие ко многим"

Давайте перейдем к рассмотрению отображения "многие ко многим" между классами Contact и Hobby. Каждый контакт имеет ноль или большее количество хобби, а каждое хобби связано с нулем или большим числом контактов. Таким образом, получается отображение "многие ко многим". Такое отображение требует таблицу соединения, которой является таблица CONTACT_HOBBY_DETAIL, показанная на рис. 9.4. В листинге 9.10 представлен фрагмент кода в классе Contact, предназначенный для моделирования этого отношения.

Листинг 9.10. Ассоциация “многие ко многим”

```
package com.apress.prospring3.ch9.domain;

// Операторы импорта опущены.
@Entity
@Table(name = "contact")
public class Contact implements Serializable {

    // Остальной код не показан.
    private Set<Hobby> hobbies = new HashSet<Hobby>();

    @ManyToMany
    @JoinTable(name = "contact_hobby_detail",
        joinColumns = @JoinColumn(name = "CONTACT_ID"),
        inverseJoinColumns = @JoinColumn(name = "HOBBY_ID"))
    public Set<Hobby> getHobbies() {
        return this.hobbies;
    }

    public void setHobbies(Set<Hobby> hobbies) {
        this.hobbies = hobbies;
    }
}
```

Как показано в листинге 9.10, метод извлечения атрибута `hobbies` в классе `Contact` аннотирован посредством `@ManyToMany`. Мы также предоставляем `@JoinTable` для указания таблицы соединения, которую должна просматривать инфраструктура Hibernate. В `name` задается имя таблицы соединения, в `joinColumns` определяется столбец, являющийся внешним ключом в таблице `CONTACT`, а в `inverseJoinColumns` указывается столбец, который представляет внешний ключ на другой стороне ассоциации, т.е. в таблице `HOBBY`. В листинге 9.11 показан соответствующий фрагмент кода в классе `Hobby`.

Листинг 9.11. Отображение “многие ко многим” в классе `Hobby`

```
package com.apress.prospring3.ch9.domain;

// Операторы импорта опущены.

@Entity
@Table(name = "hobby")
public class Hobby implements Serializable {

    // Остальной код не показан.

    private Set<Contact> contacts = new HashSet<Contact>();

    @ManyToMany
    @JoinTable(name = "contact_hobby_detail",
        joinColumns = @JoinColumn(name = "HOBBY_ID"),
        inverseJoinColumns = @JoinColumn(name = "CONTACT_ID"))
    public Set<Contact> getContacts() {
        return this.contacts;
    }

    public void setContacts(Set<Contact> contacts) {
        this.contacts = contacts;
    }
}
```

Отображение более-менее похоже на приведенное в листинге 9.10, но атрибуты `joinColumns` и `inverseJoinColumns` обращены, чтобы отразить новую ассоциацию.

Интерфейс `Session` в Hibernate

В Hibernate при взаимодействии с базой данных в основном приходится иметь дело с интерфейсом `Session`, который получается из `SessionFactory`.

В листинге 9.12 показано, что класс `ContactDaoImpl` содержит примеры из этой главы. Сконфигурированная реализация `SessionFactory` внедрена в этот класс.

Листинг 9.12. Внедрение `SessionFactory`

```
package com.apress.prospring3.ch9.dao.hibernate;

// Операторы импорта опущены.

@Repository("contactDao")
@Transactional
public class ContactDaoImpl implements ContactDao {

    // Остальной код не показан.

    private Log log = LogFactory.getLog(ContactDaoImpl.class);

    private SessionFactory sessionFactory;

    public SessionFactory getSessionFactory() {
        return sessionFactory;
    }

    @Resource(name="sessionFactory")
    public void setSessionFactory(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }
}
```

Как обычно, мы объявляем класс DAO как бин Spring с логикой доступа данных, используя аннотацию `@Repository`. Аннотация `@Transactional` определяет требования к транзакциям, которые будут обсуждаться в главе 13. Атрибут `sessionFactory` указан для внедрения с применением аннотации `@Resource`.

Выполнение операций базы данных с помощью Hibernate

В этом разделе мы обсудим, как выполнять операции базы данных в Hibernate. В листинге 9.13 показан интерфейс `ContactDao`, который задает службы доступа к данным контактов, которые мы собираемся предоставлять.

Листинг 9.13. Интерфейс `ContactDao`

```
package com.apress.prospring3.ch9.dao;

import java.util.List;

import com.apress.prospring3.ch9.domain.Contact;

public interface ContactDao {

    // Найти все контакты.
    public List<Contact> findAll();
```

```
    // Найти все контакты с заданным телефоном и хобби.
    public List<Contact> findAllWithDetail();

    // Найти контакт со всеми деталями по идентификатору.
    public Contact findById(Long id);

    // Вставить или обновить контакт.
    public Contact save(Contact contact);

    // Удалить контакт.
    public void delete(Contact contact);
}
```

Интерфейс очень прост; он имеет три метода поиска, один метод сохранения и один метод удаления. Метод `save()` будет выполнять операции вставки и обновления.

Изначально класс реализации `com.apress.prospring3.ch9.dao.hibernate.ContactDaoImpl` представляет собой класс с пустыми реализациями всех методов интерфейса `ContactDao`. В среде STS это делается очень просто. За соответствующими подробностями обращайтесь в раздел “Настройка DAO-классов JDBC с использованием аннотаций” главы 8.

Запрашивание данных с использованием языка запросов Hibernate

Инфраструктура Hibernate, как и другие инструменты ORM вроде JDO и JPA, спроектирована на основе объектной модели. Это значит, что после определения всех отображений конструировать SQL-операторы для взаимодействия с базой данных не придется. Вместо этого для определения запросов в Hibernate используется язык запросов Hibernate (Hibernate Query Language — HQL). Во время взаимодействия с базой данных Hibernate транслирует HQL-запросы в SQL-операторы.

Синтаксис HQL очень похож на синтаксис SQL. Тем не менее, при этом нужно мыслить категориями объектов, а не данных. В последующих разделах будет приведено несколько примеров.

Простой запрос с отложенной выборкой

Давайте начнем с метода `findAll()`, который просто извлекает из базы данных все контакты. Фрагмент кода приведен в листинге 9.14.

Листинг 9.14. Реализация метода `findAll`

```
package com.apress.prospring3.ch9.dao.hibernate;

// Операторы импорта опущены.

@Repository("contactDao")
@Transactional
public class ContactDaoImpl implements ContactDao {

    // Остальной код не показан.

    @Transactional(readOnly=true)
    public List<Contact> findAll() {
        return sessionFactory.getCurrentSession().
              createQuery("from Contact c").list();
    }
}
```

Как показано в листинге 9.14, метод `SessionFactory.getCurrentSession()` получает реализацию интерфейса `Session` из Hibernate. Затем вызывается метод `Session.createQuery()`, которому передается оператор HQL. Оператор `from Contact c` просто извлекает все контакты из базы данных. Альтернативный синтаксис этого оператора выглядит следующим образом: `select c from Contact c`.

Конструкция `@Transactional(readOnly=true)` означает, что транзакция должна быть установлена как предназначенная только для чтения. Установка этого атрибута для методов поиска обеспечит лучшую производительность.

В листинге 9.15 приведена простая тестовая программа для `ContactDaoImpl`.

Листинг 9.15. Тестирование класса `ContactDaoImpl`

```
package com.apress.prospring3.ch9;

import java.util.List;

import org.springframework.context.support.GenericXmlApplicationContext;

import com.apress.prospring3.ch9.dao.ContactDao;
import com.apress.prospring3.ch9.domain.Contact;

public class SpringHibernateSample {

   public static void main(String[] args) {

      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:app-context.xml");
      ctx.refresh();

      ContactDao contactDao = ctx.getBean("contactDao", ContactDao.class);
      List<Contact> contacts = contactDao.findAll();
      listContacts(contacts);

   }

   private static void listContacts(List<Contact> contacts) {
      System.out.println("");
      System.out.println("Listing contacts without details:");
      for (Contact contact: contacts) {
         System.out.println(contact);
         System.out.println();
      }
   }
}
```

Запуск тестовой программы дает в результате следующий вывод:

```
Listing contacts without details:
Contact - Id: 1, First name: Clarence, Last name: Ho, Birthday: 1980-07-30

Contact - Id: 2, First name: Scott, Last name: Tiger, Birthday: 1990-11-02

Contact - Id: 3, First name: John, Last name: Smith, Birthday: 1964-02-28
```

В выводе видно, что записи о контактах были успешно извлечены. Однако что насчет деталей, связанных с телефонами и хобби? Давайте создадим в тестовом классе новый метод для вывода детальной информации. В листинге 9.16 показан фрагмент кода.

Листинг 9.16. Тестирование класса `ContactDaoImpl` (добавлен метод `listContactsWithDetail()`)

```
// Остальной код не показан.
public class SpringHibernateSample {

   public static void main(String[] args) {

      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:app-context.xml");
      ctx.refresh();

      ContactDao contactDao = ctx.getBean("contactDao", ContactDao.class);
      List<Contact> contacts = contactDao.findAll();
      listContactsWithDetail (contacts);

   }

   private static void listContactsWithDetail(List<Contact> contacts) {
      System.out.println("");
      System.out.println("Listing contacts with details:");
      for (Contact contact: contacts) {
         System.out.println(contact);
         if (contact.getContactTelDetails() != null) {
            for (ContactTelDetail contactTelDetail:
                   contact.getContactTelDetails()) {
               System.out.println(contactTelDetail);
            }
         }
         if (contact.getHobbies() != null) {
            for (Hobby hobby: contact.getHobbies()) {
               System.out.println(hobby);
            }
         }
         System.out.println();
      }
   }
}
```

Запустив эту программу еще раз, вы получите следующее исключение:

```
Listing contacts with details:
Contact - Id: 1, First name: Clarence, Last name: Ho, Birthday: 1980-07-30
Exception in thread "main" org.hibernate.LazyInitializationException:
Исключение в потоке "main" org.hibernate.LazyInitializationException:
...
```

При попытке доступа к ассоциации инфраструктура Hibernate генерирует исключение `LazyInitializationException`. Дело в том, что по умолчанию Hibernate будет выбирать ассоциацию отложенным (“ленивым”) образом, а это означает, что соединение для наполнения записями таблицы ассоциации (т.е. `CONTACT_TEL_DETAIL`) осуществляться не будет. Причина принятия такого подхода связана с производительностью, поскольку если для запроса, извлекающего тысячи записей, извлекать также и ассоциации, то объем передаваемых данных будет чрезмерно велик.

Запрос с выборкой ассоциаций

Для того чтобы инфраструктура Hibernate извлекала данные из ассоциаций, можно воспользоваться одним из двух способов. При первом способе ассоциация определяется с режимом выборки EAGER. Ниже показан пример аннотации:

```
@ManyToMany(fetch=FetchType.EAGER)
```

Это сообщает Hibernate о необходимости выборки связанных данных в каждом запросе. Однако, как уже упоминалось, такой подход оказывает влияние на производительность извлечения данных.

Второй способ предполагает обеспечение выборки связанных данных в запросе только по требованию. Если используется запрос Criteria, можно вызвать метод Criteria.setFetchMode() и заставить Hibernate незамедлительно выполнить выборку ассоциации. В случае применения NamedQuery для инструктирования Hibernate о необходимости незамедлительной выборки ассоциации можно использовать операцию fetch.

Давайте взглянем на реализацию метода findAllWithDetail(), который будет извлекать все контакты вместе с телефонами и хобби. Мы воспользуемся подходом с именованным запросом (NamedQuery). Именованный запрос может быть вынесен в XML-файл или объявлен с применением аннотации в сущностном классе. В листинге 9.17 показан пересмотренный объект предметной области Contact с именованным запросом, который определен посредством аннотаций.

Листинг 9.17. Использование NamedQuery

```
@Entity
@Table(name = "contact")
@NamedQueries({
@NamedQuery(name="Contact.findAllWithDetail",
query="select distinct c from Contact c left join fetch c.contactTelDetails t
left join fetch c.hobbies h")
})
public class Contact implements Serializable {
    // Остальной код не показан.
}
```

В листинге 9.17 был определен именованный запрос Contact.findAllWithDetail. Затем мы определяем собственно запрос в HQL. Обратите внимание на конструкцию left join fetch, которая сообщает Hibernate о необходимости незамедлительно произвести выборку ассоциации. Также должна использоваться конструкция select distinct, иначе Hibernate возвратит дублированные объекты (например, если какой-то контакт имеет два телефона, будут возвращены два объекта Contact).

В листинге 9.18 приведена реализация метода findAllWithDetail().

Листинг 9.18. Реализация метода findAllWithDetail

```
package com.apress.prospring3.ch9.dao.hibernate;

// Операторы импорта опущены.
@Repository("contactDao")
@Transactional
public class ContactDaoImpl implements ContactDao {

    // Остальной код не показан.
    @Transactional(readOnly=true)
```

```
    public List<Contact> findAllWithDetail() {
        return sessionFactory.getCurrentSession().
        getNamedQuery("Contact.findAllWithDetail").list();
    }
}
```

В этот раз мы используем метод `Session.getNamedQuery()`, передавая ему имя `NamedQuery`. После добавления в тестовую программу вызова `ContactDao.findAllWithDetail()` и ее запуска получается следующий вывод:

```
Listing contacts with details:
Contact - Id: 1, First name: Clarence, Last name: Ho, Birthday: 1980-07-30
Contact Tel Detail - Id: 2, Contact id: 1, Type: Home, Number: 1234567890
Contact Tel Detail - Id: 1, Contact id: 1, Type: Mobile, Number: 1234567890
Hobby :Swimming
Hobby :Movies

Contact - Id: 2, First name: Scott, Last name: Tiger, Birthday: 1990-11-02
Contact Tel Detail - Id: 3, Contact id: 2, Type: Home, Number: 1234567890
Hobby :Swimming

Contact - Id: 3, First name: John, Last name: Smith, Birthday: 1964-02-28
```

Теперь все контакты со своими деталями извлекаются корректно.

Рассмотрим еще один пример с именованным запросом, принимающим параметры. На этот раз мы реализуем метод `findById()`, который также должен производить выборку ассоциации. В листинге 9.19 показан класс `Contact` с добавленным новым именованным запросом.

Листинг 9.19. Использование именованного запроса с параметрами

```
@Entity
@Table(name = "contact")
@NamedQueries({
@NamedQuery(name="Contact.findById",
query="select distinct c from Contact c left join fetch c.contactTelDetails t
left join fetch c.hobbies h where c.id = :id"),
@NamedQuery(name="Contact.findAllWithDetail",
query="select distinct c from Contact c left join fetch c.contactTelDetails t
left join fetch c.hobbies h")
})
public class Contact implements Serializable {
    // Остальной код не показан.
}
```

В именованном запросе `Contact.findById` мы объявляем именованный параметр `:id`. В листинге 9.20 приведена реализация `findById()` в `ContactDaoImpl`.

Листинг 9.20. Реализация метода `findById`

```
package com.apress.prospring3.ch9.dao.hibernate;

// Операторы импорта опущены.
@Repository("contactDao")
@Transactional
public class ContactDaoImpl implements ContactDao {

    // Остальной код не показан.
```

```
    @Transactional(readOnly=true)
    public Contact findById(Long id) {
        return (Contact) sessionFactory.getCurrentSession().
            getNamedQuery("Contact.findById").
            setParameter("id", id).uniqueResult();
    }
}
```

В листинге 9.20 мы используем тот же самый метод `Session.getNameQuery()`. Кроме того, мы вызываем метод `setParameter()`, передавая ему именованный параметр и значение для него. В случае нескольких параметров можно применять метод `setParameterList()` или `setParameters()` интерфейса `Query`.

Существуют также и более сложные методы запросов, такие как собственный запрос или запрос с критерием, которые мы обсудим в следующей главе при рассмотрении JPA.

Чтобы протестировать метод `findById`, добавьте в класс `SpringHibernateSample` фрагмент кода, показанный в листинге 9.21.

Листинг 9.21. Тестирование метода `findById`

```
    Contact contact;

    // Найти контакт по идентификатору.
    contact = contactDao.findById(1l);
    System.out.println("");
    System.out.println("Contact with id 1:" + contact);
    System.out.println("");
```

Запуск тестовой программы дает следующий вывод (показанный не полностью):

```
Contact with id 1:Contact - Id: 1, First name: Clarence, Last name: Ho,
Birthday: 1980-07-30
```

Вставка данных

Вставка данных в Hibernate осуществляется очень просто. Единственным нестандартным действием является извлечение первичного ключа, сгенерированного базой данных. Как было показано в предыдущей главе, в JDBC нужно было явно заявлять, что мы хотим извлечь сгенерированный ключ, передавая экземпляр `KeyHolder` и получая из него ключ после выполнения оператора вставки. В Hibernate все эти действия не требуются. Инфраструктура Hibernate самостоятельно извлечет сгенерированный ключ и заполнит объект предметной области после вставки. Реализация метода `save()` представлена в листинге 9.22.

Листинг 9.22. Реализация метода `save`

```
package com.apress.prospring3.ch9.dao.hibernate;

// Операторы импорта опущены.
@Repository("contactDao")
@Transactional
public class ContactDaoImpl implements ContactDao {

    // Остальной код не показан.
    public Contact save(Contact contact) {
        sessionFactory.getCurrentSession().saveOrUpdate(contact);
```

```
        log.info("Contact saved with id: " + contact.getId());
        return contact;
    }
}
```

Мы должны лишь вызвать метод `Session.saveOrUpdate()`, который может использоваться для операций вставки и обновления. Мы также выводим в журнал идентификатор сохраненного объекта контакта, который будет заполняться Hibernate после помещения объекта в базу данных. Давайте протестируем операцию вставки.

В листинге 9.23 показан фрагмент кода для вставки новой записи контакта. Добавьте этот фрагмент кода в класс `SpringHibernateSample`.

Листинг 9.23. Тестирование операции вставки

```
// Добавить новый контакт.
contact = new Contact();
contact.setFirstName("Michael");
contact.setLastName("Jackson");
contact.setBirthDate(new Date());
ContactTelDetail contactTelDetail =
   new ContactTelDetail("Home", "1111111111");
contact.addContactTelDetail(contactTelDetail);
contactTelDetail = new ContactTelDetail("Mobile", "2222222222");
contact.addContactTelDetail(contactTelDetail);
contactDao.save(contact);
contacts = contactDao.findAllWithDetail();
listContactsWithDetail(contacts);
```

В листинге 9.23 мы создаем новый объект контакта, добавляем сведения о двух телефонах и сохраняем этот объект. Затем мы выводим список всех контактов. Запуск этой программы дает следующий вывод:

```
Hibernate: insert into contact_tel_detail (ID, CONTACT_ID, TEL_NUMBER,
TEL_TYPE, VERSION) values (null, ?, ?, ?, ?)
2011-10-13 15:25:52,405 INFO [com.apress.prospring3.ch9.dao.hibernate.
ContactDaoImpl] - <Contact saved with id: 4>

Listing contacts with details:
Contact - Id: 1, First name: Clarence, Last name: Ho, Birthday: 1980-07-30
Contact Tel Detail - Id: 1, Contact id: 1, Type: Mobile, Number: 1234567890
Contact Tel Detail - Id: 2, Contact id: 1, Type: Home, Number: 1234567890
Hobby :Swimming
Hobby :Movies

Contact - Id: 4, First name: Michael, Last name: Jackson, Birthday: 2011-10-13
Contact Tel Detail - Id: 4, Contact id: 4, Type: Mobile, Number: 2222222222
Contact Tel Detail - Id: 5, Contact id: 4, Type: Home, Number: 1111111111

Contact - Id: 3, First name: John, Last name: Smith, Birthday: 1964-02-28

Contact - Id: 2, First name: Scott, Last name: Tiger, Birthday: 1990-11-02
Contact Tel Detail - Id: 3, Contact id: 2, Type: Home, Number: 1234567890
Hobby :Swimming
```

В записи журнала видно, что идентификатор вновь сохраненного контакта заполнен корректно. Инфраструктура Hibernate также выводит все SQL-операторы, отправляемые базе данных, поэтому вы получаете хорошее представление о том, что происходит “за кулисами”.

Обновление данных

Обновление контакта производится так же просто, как его вставка. Рассмотрим пример. Предположим, что для контакта с идентификатором 1 необходимо обновить имя и удалить запись с домашним телефоном. Чтобы протестировать операцию обновления, добавьте в класс `SpringHibernateSample` фрагмент кода, показанный в листинге 9.24.

Листинг 9.24. Тестирование операции обновления

```
// Обновить контакт.
contact = contactDao.findById(1l);
contact.setFirstName("Kim Fung");
Set<ContactTelDetail> contactTels = contact.getContactTelDetails();
ContactTelDetail toDeleteContactTel = null;
for (ContactTelDetail contactTel: contactTels) {
   if (contactTel.getTelType().equals("Home")) {
      toDeleteContactTel = contactTel;
   }
}

contact.removeContactTelDetail(toDeleteContactTel);
contactDao.save(contact);
contacts = contactDao.findAllWithDetail();
listContactsWithDetail(contacts);
```

В листинге 9.24 видно, что мы сначала извлекаем запись с идентификатором 1, а затем изменяем имя. После этого мы проходим в цикле по объектам с деталями телефонов, извлекаем объект с типом `"Home"` и удаляем его из контакта. Наконец, мы вызываем метод `ContactDao.save()`. Запуск программы дает в результате следующий вывод (часть вывода не показана):

```
Contact - Id: 1, First name: Kim Fung, Last name: Ho, Birthday: 1980-07-30
Contact Tel Detail - Id: 1, Contact id: 1, Type: Mobile, Number: 1234567890
Hobby :Movies
Hobby :Swimming
```

Как видите, имя контакта было обновлено, а сведения о домашнем телефоне удалены. Запись о телефоне может быть удалена, поскольку мы передаем ассоциации “один ко многим” атрибут `orphanRemoval=true`, который инструктирует Hibernate о том, что все висячие (т.е. не связанные с контактами) записи в базе данных должны быть удалены:

```
@OneToMany(mappedBy = "contact", cascade=CascadeType.ALL, orphanRemoval=true)
```

Удаление данных

Удаление данных также выполняется просто. Нужно лишь вызвать метод `Session.delete()` и передать ему объект контакта. В листинге 9.25 показан необходимый фрагмент кода.

Листинг 9.25. Реализация метода `delete`

```
package com.apress.prospring3.ch9.dao.hibernate;

// Операторы импорта опущены.
@Repository("contactDao")
@Transactional
```

```
public class ContactDaoImpl implements ContactDao {
    // Остальной код не показан.
    public void delete(Contact contact) {
        sessionFactory.getCurrentSession().delete(contact);
        log.info("Contact deleted with id: " + contact.getId());
    }
}
```

Операция удаления удалит запись контакта, а также всю связанную с ней информацию, включая детали о телефонах и хобби, т.к. в отображении было определено `cascade=CascadeType.ALL`. В листинге 9.26 приведен фрагмент кода для тестирования метода удаления; добавьте этот фрагмент в класс `SpringHibernateSample`.

Листинг 9.26. Тестирование операции удаления

```
// Удалить контакт.
contact = contactDao.findById(1l);
contactDao.delete(contact);
contacts = contactDao.findAllWithDetail();
listContactsWithDetail(contacts);
```

В показанном выше листинге мы извлекаем контакт с идентификатором 1 и затем вызываем метод `delete()` для удаления информации о контакте. Запуск этой программы даст следующий вывод (часть вывода не показана):

```
Listing contacts with details:
Contact - Id: 4, First name: Michael, Last name: Jackson, Birthday: 2011-10-13
Contact Tel Detail - Id: 4, Contact id: 4, Type: Home, Number: 1111111111
Contact Tel Detail - Id: 5, Contact id: 4, Type: Mobile, Number: 2222222222

Contact - Id: 3, First name: John, Last name: Smith, Birthday: 1964-02-28

Contact - Id: 2, First name: Scott, Last name: Tiger, Birthday: 1990-11-02
Contact Tel Detail - Id: 3, Contact id: 2, Type: Home, Number: 1234567890
Hobby :Swimming
```

Как видите, контакт с идентификатором 1 был успешно удален.

Соображения по поводу использования Hibernate

Примеры, рассмотренные в этой главе, продемонстрировали, что при правильном определении объектно-реляционного отображения, ассоциаций и запросов инфраструктура Hibernate может обеспечить среду, которая позволит сосредоточить внимание на программировании с использованием объектной модели, а не на построении SQL-операторов для каждой операции. В последние несколько лет Hibernate быстро развивалась и была принята широким сообществом Java-разработчиков в качестве библиотеки для уровня доступа к данным.

Тем не менее, необходимо помнить о нескольких важных моментах. Поскольку вы не имеете никакого контроля над генерируемыми SQL-операторами, то должны очень тщательно определять отображения, особенно ассоциации и связанную с ними стратегию выборки. Затем понаблюдайте за SQL-операторами, генерируемыми Hibernate, чтобы удостовериться в корректном их поведении.

Понимание внутреннего механизма, согласно которому Hibernate управляет своим сеансом, также играет очень важную роль, особенно в пакетных операциях. Hibernate хранит управляемые объекты внутри сеанса и регулярно сбрасывает их и очищает.

Плохо спроектированная логика доступа к данным может привести к тому, что Hibernate будет сбрасывать сеанс слишком часто, из-за чего существенно снизится производительность. Если нужен полный контроль над запросом, можно воспользоваться собственным запросом, который рассматривается в следующей главе.

Наконец, важную роль в настройке производительности Hibernate играют и параметры (размер пакета, размер выборки и т.д.). Вы должны определить их в фабрике сеансов и корректировать во время нагрузочного тестирования приложения для нахождения оптимальных значений.

В конечном счете, инфраструктура Hibernate и ее великолепная поддержка JPA, которую мы обсудим в следующей главе, представляют собой естественное решение для Java-разработчиков, нуждающихся в объектно-ориентированном подходе к реализации логики доступа к данным.

Резюме

В этой главе мы рассмотрели базовые концепции Hibernate и показали, как они конфигурируются в рамках приложения Spring. Мы также представили общие приемы для определения отображений ORM, раскрыли понятие ассоциаций и продемонстрировали использование класса `HibernateTemplate` для выполнения разнообразных операций базы данных.

В настоящей главе мы смогли охватить лишь небольшую часть функциональности и возможностей Hibernate. Мы настоятельно рекомендуем изучить стандартную документацию по Hibernate, если вы планируете применять Hibernate вместе с платформой Spring. Кроме того, есть немало книг, посвященных этой теме, в частности, *Beginning Hibernate*, 2-е изд., и *Pro JPA 2: Mastering the Java Persistence API* (обе издательства Apress).

В следующей главе мы посмотрим на интерфейс Java Persistence API (JPA) и на то, как он используется в Spring. Инфраструктура Hibernate предлагает великолепную поддержку JPA, поэтому мы продолжим применять Hibernate в качестве поставщика службы постоянства в примерах следующей главы. Для операций выборки и обновления JPA действует подобно Hibernate. В следующей главе мы рассмотрим несколько дополнительных тем, включая собственный запрос и запрос с критерием, а также использование Hibernate вместе с поддержкой JPA в примере приложения.

ГЛАВА 10

Доступ к данным в Spring с использованием JPA 2

В предыдущей главе было показано, как использовать Hibernate в приложениях Spring при реализации логики доступа к данным с помощью подхода ORM. Мы продемонстрировали возможности настройки фабрики сеансов Hibernate в конфигурации Spring и объяснили, как применять интерфейс `Session` из Hibernate для выполнения разнообразных операций доступа к данным.

Однако, как обсуждалось в предыдущей главе, это лишь один аспект, связанный с использованием Hibernate. Еще один способ эксплуатации Hibernate в Spring-приложении предусматривает использование Hibernate в качестве поставщика службы постоянства по стандарту JCP — Java Persistence API (JPA).

Ранее уже упоминалось, что базовое отображение POJO в Hibernate и мощный язык запросов HQL добились большого успеха и оказали влияние на разработку стандартов для технологий доступа к данным в мире Java. После Hibernate в рамках JCP был подготовлен стандарт JDO (Java Data Objects — объекты данных Java) и затем JPA.

На момент написания этой книги была доступна версия 2.0 интерфейса JPA (который является частью стека технологий JEE 6); это гораздо более зрелая технология и в настоящее время она широко применяется разработчиками как стандартный API-интерфейс доступа к данным. Причина в том, что JPA стандартизирует программную модель ORM через такие концепции, как `PersistenceContext`, `EntityManager` и JPQL (Java Persistence Query Language — язык запросов постоянства Java). Эта стандартизация предоставляет разработчикам способ переключения между поставщиками постоянства JPA, к которым относятся Hibernate, EclipseLink, Oracle TopLink, OpenJPA и т.д. В результате большинство новых JEE-приложений теперь используют JPA в качестве уровня доступа к данным.

В Spring также предлагается интенсивная поддержка JPA. Например, имеется несколько бинов `EntityManagerFactoryBean` для начальной загрузки диспетчера сущностей JPA в Spring-приложении, с поддержкой всех упомянутых ранее поставщиков JPA. В рамках проекта Spring Data также ведется подпроект Spring Data JPA, который ориентирован на расширенную поддержку использования JPA в Spring-приложениях. В число основных возможностей проекта Spring Data JPA входят концепции `Repository` и `Specification`, поддержка QueryDSL (Query Domain Specific Language — язык запросов, специфичный для предметной области) и т.п.

В этой главе мы обсудим применение JPA (в частности, версии JPA 2) в Spring с использованием Hibernate в качестве лежащего в основе поставщика службы постоянства. Вы узнаете, каким образом реализовать разнообразные операции базы данных, используя JPA-интерфейс `EntityManager` и JPQL (язык, подобный HQL). Затем мы покажем, как проект Spring Data JPA может дополнительно упростить разработку JPA.

Наконец, мы рассмотрим ряд более сложных тем, относящихся к ORM, включая собственные запросы и запросы с критериями.

Итак, в настоящей главе будут обсуждаться следующие темы.

- *Ключевые концепции JPA.* Мы раскроем некоторые главные концепции JPA.
- *Создание простого служебного проекта Spring JPA.* Мы пройдем через все шаги по созданию проекта Spring JPA с использованием STS. Хотя среда STS создает проект с обязательными зависимостями, мы все равно должны производить тонкую настройку конфигурации проекта для поддержки всех функциональных возможностей, обсуждаемых в данной главе. По мере перехода к более сложным темам потребуются дополнительные библиотеки третьих сторон, поэтому мы покажем, как их добавлять в STS.
- *Конфигурирование диспетчера сущностей JPA.* Мы обсудим типы `EntityManagerFactory`, поддерживаемые Spring, и способы конфигурирования наиболее часто используемого из них — класса `LocalContainerEntityManagerFactoryBean` — в XML-конфигурации Spring.
- *Операции с данными.* Мы покажем, каким образом реализовать элементарные операции базы данных в JPA. Концепции очень близки к таковым из Hibernate. Мы также обсудим устранение уровня DAO в JPA-приложениях.
- *Расширенные операции запросов.* Мы посмотрим, как использовать собственные запросы в JPA и строго типизированный API-интерфейс критериев для реализации более гибких операций запросов.
- *Введение в Spring Data JPA.* Мы обсудим проект Spring Data JPA и покажем, каким образом он позволяет упростить процесс разработки логики доступа к данным.
- *Отслеживание изменений в сущностях и аудит.* Общим требованием в операциях обновления базы данных является отслеживание дат создания и обновления сущностей, а также того, кто производил изменение. Кроме того, всегда требуется такая критически важная информация, как пользователь и таблица хронологии, хранящая все версии сущности. Мы покажем, как Spring Data JPA и Hibernate Envers (Hibernate Entity Versioning System) могут упростить разработку логики подобного рода.
- *Использование JPA с Hibernate в примере приложения.* В примере приложения в целях реализации JPA мы решили избавиться от уровня DAO и внедрить диспетчер сущностей прямо в уровень обслуживания для выполнения бизнес-логики. Мы обсудим применение JPA в примере приложения.

На заметку! Подобно Hibernate, JPA поддерживает определение отображений либо в XML, либо в Java-аннотациях. В этой главе мы сконцентрируем внимание на отображении с помощью аннотаций, поскольку такой стиль более популярен, нежели вариант с XML.

Введение в JPA 2

Как и со многими другими запросами спецификации Java (JSR), цель спецификации JPA 2.0 (JSR-317) заключается в стандартизации программной модели ORM в средах JSE и JEE. В этой спецификации определен общий набор концепций, аннотаций, интерфейсов и других служб, которые поставщик постоянства JPA должен реализовать (все это находится в пакете `javax.persistence`). При программировании в стандарте JPA разработчики имеют возможность переключения лежащего в основе поставщика по своему желанию, что очень похоже на переключение на другой JEE-совместимый сервер приложений для приложений, построенных в соответствии со стандартами JEE.

В основе JPA лежит интерфейс `EntityManager`, который поступает из фабрик типа `EntityManagerFactory`. Главной задачей `EntityManager` является поддержка контекста постоянства, внутри которого будут храниться все управляемые экземпляры сущностей. Конфигурация `EntityManager` определена как единица постоянства, и в приложении может существовать более одной такой единицы. Если вы используете Hibernate, то можете считать контекст постоянства тем же самым, что и интерфейс `Session`, а `EntityManagerFactory` — тем же, что и `SessionFactory`. В Hibernate управляемые сущности сохраняются в сеансе, с которым можно взаимодействовать напрямую через Hibernate-интерфейс `SessionFactory` или `Session`. Однако в JPA непосредственно взаимодействовать с контекстом постоянства нельзя. Вместо этого для выполнения всей необходимой работы следует полагаться на `EntityManager`.

Язык JPQL очень похож на HQL, поэтому если вы ранее работали с HQL, то перейти на JPQL будет нетрудно. Тем не менее, в JPA 2 появился строго типизированный API-интерфейс критериев, который при конструировании запроса полагается на метаданные отображенных сущностей, так что любые ошибки будут обнаруживаться во время компиляции, а не во время выполнения.

За более детальными сведениями о спецификации JPA 2 рекомендуем обратиться к книге *Pro JPA 2: Mastering the Java Persistence API* (Apress, 2009 г.).

В этом разделе мы рассмотрим базовые концепции JPA, пример модели данных, который будет использоваться в главе, и способы конфигурирования `ApplicationContext` для поддержки JPA.

Создание простого служебного проекта Spring JPA в STS

Давайте для начала создадим проект JPA в STS. Среда STS предоставляет шаблонный проект под названием Simple Spring JPA Utility Project (Простой служебный проект Spring JPA). Чтобы создать такой проект в STS, выберите пункт меню New Project⇨Spring Template Project (Новый проект⇨Шаблонный проект Spring), укажите вариант Simple Spring JPA Utility Project (Простой служебный проект Spring JPA) и введите детальные сведения о проекте. На рис. 10.1 показано диалоговое окно New Template Project (Новый шаблонный проект), а на рис. 10.2 — настройки проекта.

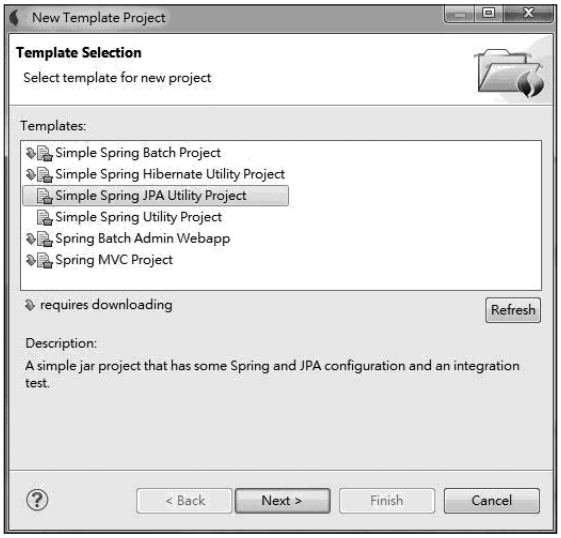


Рис. 10.1. Создание простого служебного проекта Spring JPA в среде STS

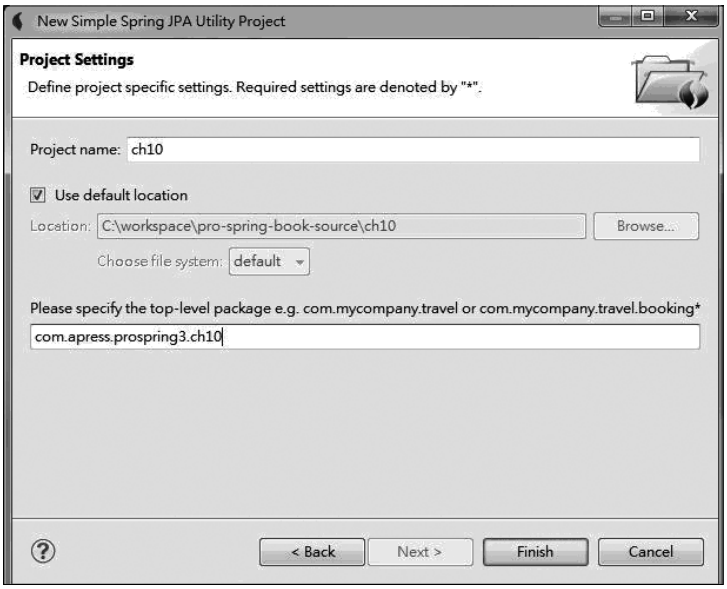


Рис. 10.2. Настройки простого служебного проекта Spring JPA

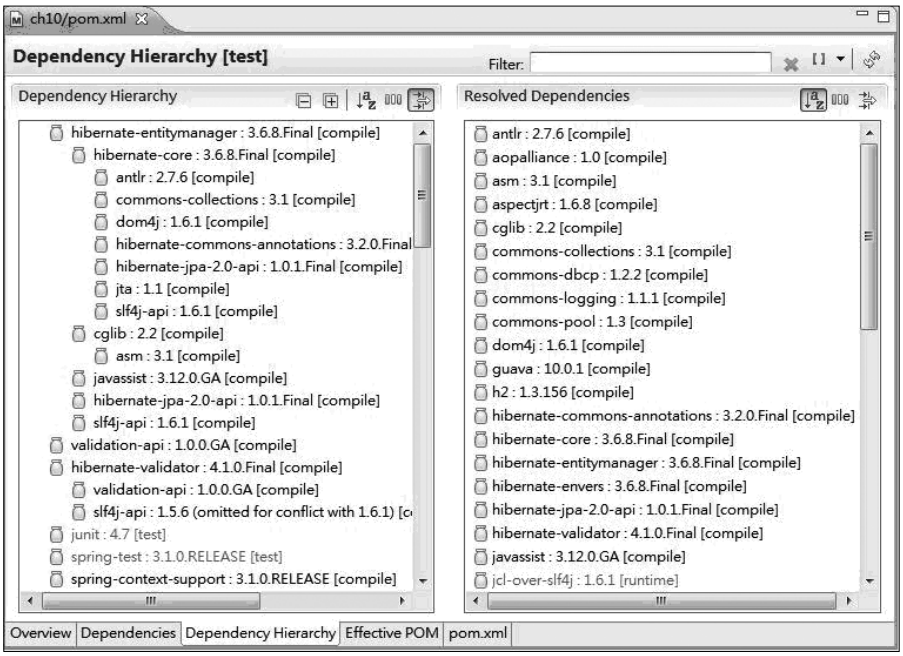


Рис. 10.3. Зависимости Maven для простого служебного проекта Spring JPA

После этого STS создаст основанный на Maven проект с обязательными зависимостями. Открыв файл pom.xml проекта, вы увидите зависимости, определенные по умолчанию. На рис. 10.3 показан неполный список этих зависимостей в STS.

На рис. 10.3 видно, что по умолчанию Spring использует Hibernate в качестве поставщика JPA для шаблонного проекта JPA. Следующим шагом, который потребуется предпринять, будет модификация содержимого файла pom.xml для использования JDK 6 как целевой исполняющей среды и обновление конфигурации проекта. Удостоверьтесь, что проект использует Spring 3.1.

На время написания этой книги шаблонный проект JPA добавлял в качестве зависимости версию Hibernate 3.6.0.Final. Измените это на версию 3.6.8.Final в файле pom.xml, т.к. в этой более новой версии исправлены ошибки в примерах, которые обсуждаются в данной главе.

Модель данных для кода примеров

В этой главе мы будем пользоваться той же самой моделью данных, что и в главе 9. Однако при обсуждении реализации средств аудита для целей демонстрации мы добавим несколько столбцов и таблицу хронологии. Таким образом, мы начнем с тех же сценариев создания базы данных, какие приводились в предыдущей главе.

Если вы по каким-то причинам пропустили главу 9, взгляните на модель данных, представленную в разделе “Модель данных для кода примера”; это поможет понять пример кода в настоящей главе.

Конфигурирование EntityManagerFactory в JPA

Как упоминалось ранее в этой главе, для применения JPA в Spring необходимо сконфигурировать EntityManagerFactory в Spring, как это делалось для SessionFactory в Hibernate. В Spring поддерживаются три типа конфигурации EntityManagerFactory.

При первом типе конфигурации используется класс LocalEntityManagerFactoryBean. Это простейший случай, который требует только указания имени единицы постоянства. Однако поскольку он не поддерживает внедрение источника данных и, следовательно, не имеет возможности участвовать в глобальных транзакциях, такой тип конфигурации подходит только для простого развертывания.

Второй тип конфигурации применяется для совместимого с JEE 6 контейнера, в который сервер приложений производит начальную загрузку единицы постоянства JPA на основе информации в дескрипторах развертывания, так что Spring будет иметь возможность найти диспетчер сущностей с помощью поиска JNDI. В листинге 10.1 приведен фрагмент кода для поиска диспетчера сущностей через JNDI.

Листинг 10.1. Поиск диспетчера сущностей через JNDI

```
<beans>
   <jee:jndi-lookup id="prospring3Emf"
      jndi-name="persistence/prospring3PersistenceUnit"/>
</beans>
```

В спецификации JPA единица постоянства должна быть определена в конфигурационном файле META-INF/persistence.xml. Тем не менее, Spring 3.1 предлагает новую возможность, которая устраняет эту необходимость. Мы рассмотрим упомянутую возможность в текущем разделе.

Третий тип конфигурации, который является наиболее распространенным, и он же будет использоваться в этой главе и в примере приложения — класс LocalContainerEntityManagerFactoryBean. Он поддерживает внедрение источника данных и может прини-

мать участие как в локальных, так и в глобальных транзакциях. В листинге 10.2 показано содержимое соответствующего XML-файла конфигурации (`app-context.xml`).

Листинг 10.2. Конфигурация Spring для `LocalContainerEntityManagerFactoryBean`

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
   xmlns:context="http://www.springframework.org/schema/context"
   xmlns:jdbc=http://www.springframework.org/schema/jdbc
   xmlns:tx="http://www.springframework.org/schema/tx"
   xsi:schemaLocation="http://www.springframework.org/schema/jdbc
      http://www.springframework.org/schema/jdbc/spring-jdbc-3.1.xsd
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
      http://www.springframework.org/schema/tx
      http://www.springframework.org/schema/tx/spring-tx-3.1.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context-3.1.xsd">

   <description>Example configuration to get you started.</description>

   <jdbc:embedded-database id="dataSource" type="H2">
      <jdbc:script location="classpath:schema.sql"/>
      <jdbc:script location="classpath:test-data.sql"/>
   </jdbc:embedded-database>

   <bean id="transactionManager"
      class="org.springframework.orm.jpa.JpaTransactionManager">
      <property name="entityManagerFactory" ref="emf"/>
   </bean>

<tx:annotation-driven transaction-manager="transactionManager" />

   <bean id="emf"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
      <property name="dataSource" ref="dataSource" />
      <property name="jpaVendorAdapter">
       <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter" />
      </property>
      <property name="packagesToScan"
         value="com.apress.prospring3.ch10.domain"/>
      <property name="jpaProperties">
         <props>
            <prop key="hibernate.dialect">
               org.hibernate.dialect.H2Dialect
            </prop>
            <prop key="hibernate.max_fetch_depth">3</prop>
            <prop key="hibernate.jdbc.fetch_size">50</prop>
            <prop key="hibernate.jdbc.batch_size">10</prop>
            <prop key="hibernate.show_sql">true</prop>
         </props>
      </property>
   </bean>

   <context:annotation-config/>

   <context:component-scan
      base-package="com.apress.prospring3.ch10.service.jpa" />
</beans>
```

В приведенной выше конфигурации объявлено несколько бинов для поддержки конфигурации `LocalContainerEntityManagerFactoryBean` с Hibernate в качестве поставщика постоянства. Ниже описаны основные элементы конфигурации.

- *Бин `dataSource`*. Мы объявляем источник данных со встроенной базой данных типа H2. Поскольку это встроенная база данных, имя для нее не требуется.
- *Бин `transactionManager`*. Фабрика диспетчера сущностей требует диспетчера транзакций для транзакционного доступа к данным. В Spring предусмотрен диспетчер транзакций специально для JPA (`org.springframework.orm.jpa.JpaTransactionManager`). Этот бин объявлен с идентификатором `transactionManager`. Транзакции более подробно рассматриваются в главе 13. Дескриптор `<tx:annotation-driven>` предназначен для поддержки объявления требований к установлению границ транзакций с использованием аннотаций.
- *Дескриптор `component-scan`*. Этот дескриптор уже должен быть знаком. Мы указываем Spring на необходимость сканирования компонентов в пакете `com.apress.prospring3.ch10.service.jpa`.
- *Бин `EntityManagerFactory` из JPA (`emf`)*. Бин `emf` является наиболее важным элементом конфигурации. Первым делом, мы объявляем, что этот бин использует класс `LocalContainerEntityManagerFactoryBean`. Внутри бина предоставляются различные свойства. Во-первых, как и можно было ожидать, необходимо внедрить бин источника данных. Во-вторых, мы указываем в свойстве `jpaVendorAdapter` класс `HibernateJpaVendorAdapter`, т.к. используется Hibernate. В-третьих, мы инструктируем фабрику сущностей о поиске объектов предметной области с аннотациями ORM в пакете `com.apress.prospring3.ch10.domain` (указанном с помощью дескриптора `<property name="packagesToScan">`). Обратите внимание, что эта возможность доступна только начиная с версии Spring 3.1, и за счет поддержки сканирования классов предметной области можно опустить определение единицы постоянства в файле `META-INF/persistence.xml`. Наконец, в-четвертых, свойство `jpaProperties` предоставляет детали конфигурации для поставщика постоянства, т.е. Hibernate. Вы увидите, что здесь применяется та же самая конфигурация, которая использовалась в главе 9, поэтому не приводим ее пояснения снова.

Отображение ORM с использованием аннотаций JPA

Как уже упоминалось, инфраструктура Hibernate оказала большое влияние на проектное решение, положенное в основу JPA. Аннотации отображения в JPA очень близки к тем аннотациям, которые применялись в главе 9 для отображения объектов предметной области на базу данных.

Если вы взглянете на исходный код классов предметной области в главе 9, то заметите, что все аннотации отображения находятся в пакете `javax.persistence`, а это означает их совместимость с JPA. Таким образом, мы не будем повторять этот код в данной главе, поэтому ищите соответствующие пояснения в главе 9.

Устранение уровня DAO

В главах 8 и 9 было показано, что в случае применения JDBC и Hibernate для программирования логики доступа к данным мы используем шаблон DAO. Этот шаблон предназначен для выделения различных реализаций логики доступа к данным в собственный уровень, так что детали реализации полностью скрываются от уровня обслуживания, которому ничего не известно о том, какие средства применяются для доступа к данным — JDBC или Hibernate.

Однако после появления JPA обоснование существования уровня для логики доступа к данным становится сомнительным. Поскольку спецификация JPA была спроектирована как стандарт, в рамках которого можно легко переключаться между разными поставщиками постоянства, просто нет веских причин поддерживать логику доступа к данным на отдельном уровне DAO. Таким образом, в настоящее время многие разработчики JEE, придерживающиеся стандарта JPA в качестве уровня доступа к данным, решили избавиться от уровня DAO и внедрять диспетчер сущностей напрямую в уровень обслуживания.

По поводу обоснования причин существования уровня DAO в JEE-приложениях по-прежнему ведутся горячие споры, но дело в том, что устранение уровня DAO значительно упрощает архитектуру, и это является одним из главных преимуществ. В настоящей главе и в примере приложения мы решили избавиться от уровня DAO в реализации JPA, вместо этого предоставив реализацию уровня обслуживания с прямым внедрением диспетчера сущностей в классы уровня обслуживания.

Тем не менее, не будет большой разницы, если вы или ваша команда разработчиков придерживаются мнения, что уровень DAO должен существовать. Вы можете по-прежнему программировать всю логику JPA в виде отдельного уровня DAO и внедрять его в уровень обслуживания.

Внедрение диспетчера сущностей в классы уровня обслуживания

Для поддержки JDBC и Hibernate в Spring предусмотрены соответствующие шаблонные классы `JdbcTemplate` и `HibernateTemplate` (хотя, как обсуждалось в главе 9, они явно объявлены устаревшими и вместо них рекомендуется напрямую применять интерфейс `Session` из Hibernate), которые существенно упрощают разрабатываемый код. Что касается JPA, то в Spring предлагается шаблонный класс `JpaTemplate`. Однако с обретением зрелости JPA 2 в команде разработки Spring обнаружили, что предоставлять такой шаблон не обязательно. Заглянув в документацию по Spring Framework 3.1, вы увидите, что класс `JpaTemplate` объявлен устаревшим, а для JPA 2 рекомендуется работать с диспетчером сущностей напрямую.

После того как `EntityManagerFactory` соответствующим образом сконфигурирован, внедрение его в классы уровня обслуживания осуществляется очень просто. В листинге 10.3 приведен фрагмент кода класса `ContactServiceImpl`, который будет использоваться в качестве примера для выполнения операций базы данных с применением JPA.

Листинг 10.3. Внедрение диспетчера сущностей

```
package com.apress.prospring3.ch10.service.jpa;

// Операторы импорта опущены.
@Service("jpaContactService")
@Repository
@Transactional
public class ContactServiceImpl implements ContactService {

    private Log log = LogFactory.getLog(ContactServiceImpl.class);

    @PersistenceContext
    private EntityManager em;

    // Остальной код не показан.
}
```

Как показано в листинге 10.3, к классу применено множество аннотаций. Аннотация `@Service` предназначена для идентификации класса как компонента Spring, который предоставляет бизнес-службы другому уровню; этот бин Spring получает имя `jpaContactService`. Аннотация `@Repository` указывает, что класс содержит логику доступа к данным и заставляет Spring транслировать исключения, специфичные для поставщика, в иерархию `DataAccessException`, определенную Spring. В действительности можно использовать `@Repository("jpaContactService")` и избавиться от аннотации `@Service`; результат будет тем же самым. Здесь аннотация `@Service` применяется просто для того, чтобы отразить принадлежность компонента к уровню обслуживания (подобно тому, как `@Controller` указывает, что компонент относится к веб-уровню), т.к. она более дружественна к разработчику. Как вам уже должно быть известно, аннотация `@Transactional` предназначена для определения требований к транзакции.

Для внедрения `EntityManager` мы используем аннотацию `@PersistenceContext`, которая представляет собой стандартную аннотацию JPA для внедрения диспетчера сущностей. Вас может удивить, почему для внедрения диспетчера сущностей применяется другое имя (`@PersistenceContext`), но если принять во внимание, что контекст постоянства управляется `EntityManager`, то эта аннотация имеет большой смысл. При наличии в приложении множества единиц постоянства можно также добавить к аннотации атрибут `unitName` для указания единицы постоянства, подлежащей внедрению. Обычно единиц постоянства представляет отдельный источник данных.

После внедрения `EntityManager` все готово для выполнения операций базы данных, которые обсуждаются в следующем разделе.

Операции базы данных в JPA

В этом разделе мы покажем, как выполнять операции базы данных в JPA.

В листинге 10.4 приведен интерфейс `ContactService`, отражающий службы информации о контактах, которые мы собираемся предоставлять.

Листинг 10.4. Интерфейс `ContactService`

```
package com.apress.prospring3.ch10.service;

import java.util.List;

import com.apress.prospring3.ch10.domain.Contact;

public interface ContactService {

    // Найти все контакты.
    public List<Contact> findAll();

    // Найти все контакты с заданным телефоном и хобби.
    public List<Contact> findAllWithDetail();

    // Найти контакт со всеми деталями по идентификатору.
    public Contact findById(Long id);

    // Вставить или обновить контакт.
    public Contact save(Contact contact);

    // Удалить контакт.
    public void delete(Contact contact);
}
```

Интерфейс очень прост; он имеет три метода поиска, один метод сохранения и один метод удаления. Метод `save()` будет выполнять операции вставки и обновления.

Запрашивание данных с использованием языка запросов постоянства Java

Языки JPQL и HQL имеют очень похожий синтаксис и на самом деле все запросы HQL, которые применялись в главе 9, можно использовать повторно для реализации трех методов поиска в интерфейсе `ContactService`.

Давайте вспомним именованные запросы, определенные для сущностного класса `Contact`. Соответствующий фрагмент кода показан в листинге 10.5.

Листинг 10.5. Именованные запросы для объекта предметной области `Contact`

```
package com.apress.prospring3.ch10.domain;

import java.io.Serializable;
import javax.persistence.*;
@Entity
@Table(name = "contact")
@NamedQueries({
   @NamedQuery(name="Contact.findAll", query="select c from Contact c"),
   @NamedQuery(name="Contact.findById",
      query="select distinct c from Contact c left join fetch
c.contactTelDetails t left join fetch c.hobbies h where c.id = :id"),
   @NamedQuery(name="Contact.findAllWithDetail",
      query="select distinct c from Contact c left join fetch
c.contactTelDetails t left join fetch c.hobbies h")
})
public class Contact implements Serializable {
   // Остальной код не показан.
}
```

Если сравнить запросы в листинге 10.5 с запросами, которые применялись в главе 9, то никаких отличий не обнаружится. Таким образом, в случае использования Hibernate миграция на JPA осуществляется относительно легко.

В классе реализации, `com.apress.prospring3.ch10.service.jpa.ContactServiceImpl`, мы начинаем с создания класса с пустой реализацией всех методов интерфейса `ContactService`. В STS это делается очень просто. За подробными сведениями обращайтесь в раздел “Настройка DAO-классов JDBC с использованием аннотаций” главы 8.

Начнем с метода `findAll()`, который просто извлекает все контакты из базы данных. В листинге 10.6 приведен фрагмент кода.

Листинг 10.6. Реализация метода `findAll()`

```
package com.apress.prospring3.ch10.service.jpa;
// Операторы импорта опущены.
@Service("jpaContactService")
@Repository
@Transactional
public class ContactServiceImpl implements ContactService {
   // Остальной код не показан.
   @Transactional(readOnly=true)
   public List<Contact> findAll() {
      List<Contact> contacts = em.createNamedQuery("Contact.findAll",
         Contact.class).getResultList();
      return contacts;
   }
}
```

В этом листинге видно, что мы используем метод `EntityManager.createNamedQuery()`, передавая ему имя запроса и ожидаемый тип возврата. В данном случае `EntityManager` вернет интерфейс `TypedQuery<X>`. Затем вызывается метод `TypedQuery.getResultList()` для извлечения контактов.

В листинге 10.7 приведен код простой программы для тестирования `ContactServiceImpl`.

Листинг 10.7. Тестирование класса `ContactServiceImpl`

```
package com.apress.prospring3.ch10;

import java.util.List;

import org.springframework.context.support.GenericXmlApplicationContext;

import com.apress.prospring3.ch10.domain.Contact;
import com.apress.prospring3.ch10.domain.ContactTelDetail;
import com.apress.prospring3.ch10.domain.Hobby;
import com.apress.prospring3.ch10.service.ContactService;

public class JpaSample {

   public static void main(String[] args) {

      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:app-context.xml");
      ctx.refresh();

      ContactService contactService = ctx.getBean(
         "jpaContactService", ContactService.class);

      // Вывести список контактов без подробных сведений.
      List<Contact> contacts = contactService.findAll();
      listContacts(contacts);
   }

   private static void listContacts(List<Contact> contacts) {
      System.out.println("");
      System.out.println("Listing contacts without details:");
      for (Contact contact: contacts) {
         System.out.println(contact);
         System.out.println();
      }
   }
}
```

Запуск предыдущего кода на выполнение дает следующий вывод:

```
Listing contacts without details:
Contact - Id: 1, First name: Clarence, Last name: Ho, Birthday: 1980-07-30

Contact - Id: 2, First name: Scott, Last name: Tiger, Birthday: 1990-11-02

Contact - Id: 3, First name: John, Last name: Smith, Birthday: 1964-02-28
```

На заметку! Для ассоциаций в спецификации JPA утверждается, что по умолчанию поставщики постоянства должны выбирать ассоциацию незамедлительно. Однако в реализации JPA, предлагаемой Hibernate, по умолчанию по-прежнему используется стратегия отложенной выборки. Поэтому в таком случае явное определение ассоциации с отложенной выборкой не требуется. Стандартная стратегия выборки Hibernate отличается от описанной в спецификации JPA.

Теперь перейдем к методу `findAllWithDetail()`, который будет выбирать все связанные телефоны и хобби. Реализация этого метода показана в листинге 10.8.

Листинг 10.8. Реализация метода `FindAllWithDetail()`

```
package com.apress.prospring3.ch10.service.jpa;
// Операторы импорта опущены.
@Service("jpaContactService")
@Repository
@Transactional
public class ContactServiceImpl implements ContactService {
   // Остальной код не показан.
   @Transactional(readOnly=true)
   public List<Contact> findAllWithDetail() {
      List<Contact> contacts = em.createNamedQuery(
         "Contact.findAllWithDetail", Contact.class).getResultList();
      return contacts;
   }
}
```

Этот метод похож на метод `findAll()`, но только он использует другой именованный запрос с включенной конструкцией `left join fetch`. В листинге 10.9 приведена модифицированная тестовая программа для вывода деталей, связанных с контактами.

Листинг 10.9. Тестирование класса `ContactServiceImpl`

```
package com.apress.prospring3.ch10;
// Операторы импорта опущены.
public class JpaSample {
   public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:app-context.xml");
      ctx.refresh();
      ContactService contactService = ctx.getBean(
         "jpaContactService", ContactService.class);
      List<Contact> contacts = contactService.findAllWithDetail();
      listContactsWithDetail (contacts);
   }
   private static void listContactsWithDetail(List<Contact> contacts) {
      System.out.println("");
      System.out.println("Listing contacts with details:");
      for (Contact contact: contacts) {
         System.out.println(contact);
         if (contact.getContactTelDetails() != null) {
            for (ContactTelDetail contactTelDetail:
                contact.getContactTelDetails()) {
               System.out.println(contactTelDetail);
            }
         }
         if (contact.getHobbies() != null) {
            for (Hobby hobby: contact.getHobbies()) {
               System.out.println(hobby);
            }
         }
         System.out.println();
      }
   }
}
```

Запустив эту программу, вы получите следующий вывод:

```
Listing contacts with details:
Contact - Id: 1, First name: Clarence, Last name: Ho, Birthday: 1980-07-30
Contact Tel Detail - Id: 2, Contact id: 1, Type: Home, Number: 1234567890
Contact Tel Detail - Id: 1, Contact id: 1, Type: Mobile, Number: 1234567890
Hobby :Swimming
Hobby :Movies

Contact - Id: 2, First name: Scott, Last name: Tiger, Birthday: 1990-11-02
Contact Tel Detail - Id: 3, Contact id: 2, Type: Home, Number: 1234567890
Hobby :Swimming

Contact - Id: 3, First name: John, Last name: Smith, Birthday: 1964-02-28
```

Давайте следующим рассмотрим метод `findById()`, который демонстрирует использование именованного запроса с именованными параметрами в JPA. Кроме того, будет произведена выборка ассоциаций. Реализация этого метода показана в листинге 10.10.

Листинг 10.10. Реализация метода `findById()`

```
package com.apress.prospring3.ch10.service.jpa;

// Операторы импорта опущены.
@Service("jpaContactService")
@Repository
@Transactional
public class ContactServiceImpl implements ContactService {
    // Остальной код не показан.
    @Transactional(readOnly=true)
    public Contact findById(Long id) {
        TypedQuery<Contact> query = em.createNamedQuery(
            "Contact.findById", Contact.class);
        query.setParameter("id", id);
        return query.getSingleResult();
    }
}
```

В листинге 10.10 легко заметить, что метод `EntityManager.createNamedQuery(java.lang.String name, java.lang.Class<T> resultClass)` вызывается для получения экземпляра интерфейса `TypedQuery<T>`, который гарантирует, что результат запроса должен относиться к типу `Contact`. Затем с помощью метода `TypedQuery<T>.setParameter()` устанавливаются значения именованных параметров внутри запроса, после чего вызывается метод `getSingleResult()`, поскольку результат должен содержать только одиночный объект `Contact` с указанным идентификатором. Тестирование этого метода мы оставляем в качестве упражнения для самостоятельного выполнения.

Запрос с нетипизированными результатами

Во многих случаях возникает необходимость отправить запрос базе данных и манипулировать результатами по собственному усмотрению, а не сохранять их в отображенном сущностном классе. Типичным примером может быть веб-отчет, содержащий список заданного количества столбцов из множества таблиц.

Предположим, что имеется веб-страница, которая отображает итоговую информацию по всем контактам. Итоговая информация содержит для каждого контакта имя, фамилию и домашний телефон. Контакты, не имеющие домашнего телефона, не отображаются. В этом случае мы можем реализовать функцию с запросом и вручную манипулировать результирующим набором.

Давайте назовем метод `displayAllContactSummary()`. В листинге 10.11 приведена типовая реализация этого метода.

Листинг 10.11. Реализация метода `displayAllContactSummary()`

```
package com.apress.prospring3.ch10.service.jpa;

// Операторы импорта опущены.
@Service("contactSummaryUntype")
@Repository
@Transactional
public class ContactSummaryUntypeImpl {
    @PersistenceContext
    private EntityManager em;
    @Transactional(readOnly=true)
    public void displayAllContactSummary() {
      List result = em
            .createQuery("select c.firstName, c.lastName, t.telNumber "
               + "from Contact c left join c.contactTelDetails t "
               + " where t.telType='Home'").getResultList();
      int count = 0;
      for (Iterator i = result.iterator(); i.hasNext();) {
         Object[] values = (Object[]) i.next();
         System.out.println(++count + ": " + values[0] + ", "
           + values[1] + ", " + values[2]);
      }
   }
}
```

Как показано в листинге 10.11, мы используем метод `EntityManager.createQuery()` для создания запроса, передавая ему оператор JPQL, и затем получаем результирующий список.

В случае явного указания выбираемых столбцов в JPQL-операторе JPA возвращает итератор, а каждый элемент внутри этого итератора представляет собой массив объектов. После этого организуется цикл по итератору и для каждого элемента массива отображается значение. Каждый массив объектов соответствует записи в результирующем наборе. В листинге 10.12 приведена тестовая программа.

Листинг 10.12. Тестирование метода `displayAllContactSummary()`

```
package com.apress.prospring3.ch10;

import org.springframework.context.support.GenericXmlApplicationContext;

import com.apress.prospring3.ch10.service.jpa.ContactSummaryUntypeImpl;
public class ContactSummarySample {
   public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:app-context.xml");
      ctx.refresh();
      // Итоговая информация по контактам с нетипизированным результатом.
      ContactSummaryUntypeImpl contactSummaryUntype =
         ctx.getBean("contactSummaryUntype",
            ContactSummaryUntypeImpl.class);
      contactSummaryUntype.displayAllContactSummary();
   }
}
```

Запуск этой программы дает следующий вывод:

```
1: Clarence, Ho, 1234567890
2: Scott, Tiger, 1234567890
```

В JPA имеется более элегантное решение, нежели обработка массива объектов, возвращаемого запросом; это решение обсуждается в следующем разделе.

Запрос со специальным типом результата и конструирующим выражением

При запрашивании специального результата, подобного рассмотренному в предшествующем разделе, можно заставить JPA напрямую конструировать POJO из каждой записи. Продолжая пример из предыдущего раздела, давайте создадим POJO по имени `ContactSummary`, который хранит результаты запроса итоговой информации о контакте. Код класса `ContactSummary` приведен в листинге 10.13.

Листинг 10.13. Класс `ContactSummary`

```
package com.apress.prospring3.ch10.domain;

import java.io.Serializable;

public class ContactSummary implements Serializable {

   private String firstName;

   private String lastName;

   private String homeTelNumber;

   public ContactSummary(String firstName, String lastName,
         String homeTelNumber) {
      this.firstName = firstName;
      this.lastName = lastName;
      this.homeTelNumber = homeTelNumber;
   }

   // Методы извлечения/установки не показаны.

   public String toString() {
      return "First name: " + firstName + " Last Name: " + lastName
         + " Home Phone: " + homeTelNumber;
   }
}
```

Класс `ContactSummary` имеет свойства для итоговой информации о контакте, а также метод конструктора, который принимает все эти свойства.

Имея класс `ContactSummary`, можно модифицировать метод и воспользоваться конструирующим выражением внутри запроса, чтобы инструктировать поставщик JPA о необходимости отображения результирующего набора на класс `ContactSummary`. Сначала создадим интерфейс `ContactSummary`. Его код представлен в листинге 10.14.

Листинг 10.14. Интерфейс `ContactSummaryService`

```
package com.apress.prospring3.ch10.service;

import java.util.List;

import com.apress.prospring3.ch10.domain.ContactSummary;

public interface ContactSummaryService {
   public List<ContactSummary> findAll();
}
```

В листинге 10.15 приведена реализация метода `ContactSummaryService.findAll()`, в которой используется конструирующее выражение для отображения результирующего набора.

Листинг 10.15. Реализация метода `findAll()` с использованием конструирующего выражения

```
package com.apress.prospring3.ch10.service.jpa;

// Операторы импорта опущены.
@Service("contactSummaryService")
@Repository
@Transactional
public class ContactSummaryServiceImpl implements ContactSummaryService {

   @PersistenceContext
   private EntityManager em;

   @Transactional(readOnly=true)
   public List<ContactSummary> findAll() {

   List<ContactSummary> result = em.createQuery(
      "select new com.apress.prospring3.ch10.domain.ContactSummary("
      + "c.firstName, c.lastName, t.telNumber) "
      + "from Contact c left join c.contactTelDetails t "
      + "where t.telType='Home'",
      ContactSummary.class).getResultList();
      return result;
   }
}
```

В листинге 10.15 видно, что в операторе JPQL указано ключевое слово `new` вместе с полностью определенным именем класса POJO, который будет сохранять результаты, и передаются выбираемые атрибуты как аргумент конструктора каждого класса `ContactSummary`. Наконец, класс `ContactSummary` передается методу `createQuery()` для указания типа результата.

Чтобы протестировать метод `findAll()`, добавьте фрагмент кода из листинга 10.16 в метод `main()` класса `ContactSummarySample`.

Листинг 10.16. Тестирование метода `findAll()`, использующего конструирующее выражение

```
      // Получение итоговой информации о контакте
      // с помощью конструирующего выражения.
      ContactSummaryService contactSummaryService =
         ctx.getBean("contactSummaryService",ContactSummaryService.class);
      List<ContactSummary> contacts = contactSummaryService.findAll();

      for (ContactSummary contactSummary: contacts) {
         System.out.println(contactSummary);
      }
```

Запуск на выполнение класса `ContactSummarySample` сгенерирует вывод для каждого объекта `ContactSummary` внутри списка, как показано ниже (прочий вывод опущен):

```
First name: Clarence Last Name: Ho Home Phone: 1234567890
First name: Scott Last Name: Tiger Home Phone: 1234567890
```

Как видите, конструирующее выражение очень удобно для отображения результата специального запроса на объекты POJO для дальнейшей обработки.

Вставка данных

Вставка данных в JPA осуществляется очень просто. Подобно Hibernate, JPA также поддерживает извлечение первичного ключа, генерируемого базой данных.

В листинге 10.17 показана реализация метода `save()`.

Листинг 10.17. Реализация метода `save()`

```
package com.apress.prospring3.ch10.service.jpa;

// Операторы импорта опущены.
@Service("jpaContactService")
@Repository
@Transactional
public class ContactServiceImpl implements ContactService {

    // Остальной код не показан.
    public Contact save(Contact contact) {
        if (contact.getId() == null) {      // вставить контакт
            log.info("Inserting new contact");
            em.persist(contact);
        } else {                            // обновить контакт
            em.merge(contact);
            log.info("Updating existing contact");
        }
        log.info("Contact saved with id: " + contact.getId());
        return contact;
    }
}
```

В листинге 10.17 видно, что метод `save()` сначала проверяет, является ли объект новым экземпляром сущности, по значению `id`. Если `id` равно `null` (т.е. идентификатор еще не назначен), то это новый экземпляр сущности, и будет вызван метод `EntityManager.persist()`. Когда вызывается метод `persist()`, диспетчер сущностей (`EntityManager`) сохранит сущность и сделает ее управляемым экземпляром в рамках контекста постоянства. Если значение `id` существует, значит, производится обновление, и тогда будет вызван метод `EntityManager.merge()`. При вызове метода `merge()` диспетчер сущностей объединит состояние сущности с текущим контекстом постоянства.

Давайте протестируем операцию вставки. В листинге 10.18 приведен фрагмент кода для вставки новой записи контакта.

Листинг 10.18. Тестирование операции вставки

```
// Остальной код не показан.
public class JpaSample {

    public static void main(String[] args) {

        GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
        ctx.load("classpath:app-context.xml");
        ctx.refresh();

        ContactService contactService = ctx.getBean(
            "jpaContactService", ContactService.class);
```

```
        // Добавить новый контакт.
        Contact contact = new Contact();
        contact.setFirstName("Michael");
        contact.setLastName("Jackson");
        contact.setBirthDate(new Date());
        ContactTelDetail contactTelDetail =
           new ContactTelDetail("Home", "1111111111");
        contact.addContactTelDetail(contactTelDetail);
        contactTelDetail = new ContactTelDetail("Mobile", "2222222222");
        contact.addContactTelDetail(contactTelDetail);
        contactService.save(contact);
        List<Contact> contacts = contactService.findAllWithDetail();
        listContactsWithDetail(contacts);
    }

    private static void listContactsWithDetail(List<Contact> contacts) {
        System.out.println("");
        System.out.println("Listing contacts with details:");
        for (Contact contact: contacts) {
           System.out.println(contact);
           if (contact.getContactTelDetails() != null) {
              for (ContactTelDetail contactTelDetail:
                 contact.getContactTelDetails()) {
                 System.out.println(contactTelDetail);
              }
           }
           if (contact.getHobbies() != null) {
              for (Hobby hobby: contact.getHobbies()) {
                 System.out.println(hobby);
              }
           }
           System.out.println();
        }
    }
}
```

Как показано в листинге 10.18, мы создаем новый контакт, добавляем к нему два телефона и сохраняем объект. После этого мы снова выводим список всех контактов. Запуск программы дает следующий вывод (вывод существующих контактов не показан):

```
2011-10-19 19:35:54,968 INFO [com.apress.prospring3.ch10.service.jpa.
ContactServiceImpl] <Inserting new contact>
Hibernate: insert into contact (ID, BIRTH_DATE, FIRST_NAME, LAST_NAME,
VERSION) values (null, ?, ?, ?, ?)
Hibernate: insert into contact_tel_detail (ID, CONTACT_ID, TEL_NUMBER,
TEL_TYPE, VERSION) values(null, ?, ?, ?, ?)
Hibernate: insert into contact_tel_detail (ID, CONTACT_ID, TEL_NUMBER,
TEL_TYPE, VERSION) values (null, ?, ?, ?, ?)
2011-10-19 19:35:54,992 INFO [com.apress.prospring3.ch10.service.jpa.
ContactServiceImpl] <Contact saved with id: 4>

Listing contacts with details:
...
Contact - Id: 4, First name: Michael, Last name: Jackson, Birthday: 2011-10-19
Contact Tel Detail - Id: 5, Contact id: 4, Type: Mobile, Number: 2222222222
Contact Tel Detail - Id: 4, Contact id: 4, Type: Home, Number: 1111111111
```

В журнальной записи INFO видно, что значение id вновь вставленного контакта установлено корректно. Инфраструктура Hibernate также выводит SQL-оператор, запускаемый в базе данных.

Обновление данных

Обновление контактов реализуется так же просто, как вставка данных. Рассмотрим пример. Предположим, что для контакта с идентификатором 1 необходимо обновить имя и удалить запись для домашнего телефона. Чтобы протестировать операцию обновления, добавьте фрагмент кода из листинга 10.19 в метод main() класса JpaSample.

Листинг 10.19. Тестирование операции обновления

```
// Остальной код не показан.

// Найти контакт по идентификатору.
contact = contactService.findById(1l);
System.out.println("");
System.out.println("Contact with id 1:" + contact);
System.out.println("");

// Обновить контакт.
contact.setFirstName("Kim Fung");
Set<ContactTelDetail> contactTels = contact.getContactTelDetails();
ContactTelDetail toDeleteContactTel = null;
for (ContactTelDetail contactTel: contactTels) {
   if (contactTel.getTelType().equals("Home")) {
      toDeleteContactTel = contactTel;
   }
}
contactTels.remove(toDeleteContactTel);
contactService.save(contact);
contacts = contactService.findAllWithDetail();
listContactsWithDetail(contacts);
```

Как показано в листинге 10.19, сначала мы извлекаем запись с идентификатором 1. После этого мы изменяем имя. Затем мы проходим в цикле по объектам телефонов, извлекаем объект с типом "Home" и удаляем его из свойства сведений о телефоне для контакта. Наконец, мы вызываем метод ContactService.save() снова. В результате запуска этой программы получается следующий вывод (показан не полностью):

```
Contact - Id: 1, First name: Kim Fung, Last name: Ho, Birthday: 1980-07-30
Contact Tel Detail - Id: 1, Contact id: 1, Type: Mobile, Number: 1234567890
Hobby :Movies
Hobby :Swimming
```

Видно, что имя было обновлено, а домашний телефон удален. Телефон может быть удален по причине определения в ассоциации “один ко многим” атрибута orphanRemoval=true, который указывает поставщику JPA (т.е. Hibernate) на необходимость удаления всех висячих записей, существующих в базе данных, но не принадлежащих какому-либо сохраненному объекту.

```
@OneToMany(mappedBy = "contact",
           cascade=CascadeType.ALL, orphanRemoval=true)
```

Удаление данных

Удаление данных осуществляется просто. Нужно лишь вызвать метод `EntityManager.remove()` и передать ему объект контакта. Соответствующий фрагмент кода представлен в листинге 10.20.

Листинг 10.20. Реализация метода `delete()`

```
package com.apress.prospring3.ch10.service.jpa;
// Операторы импорта опущены.
@Service("jpaContactService")
@Repository
@Transactional
public class ContactServiceImpl implements ContactService {
   private Log log = LogFactory.getLog(ContactServiceImpl.class);
   // Остальной код не показан.
   public void delete(Contact contact) {
      Contact mergedContact = em.merge(contact);
      em.remove(mergedContact);
      log.info("Contact with id: " + contact.getId()
         + " deleted successfully");
   }
}
```

В листинге 10.20 видно, что сначала вызывается метод `EntityManager.merge()` для объединения состояния сущности с текущим контекстом постоянства. Метод `merge()` вернет управляемый экземпляр сущности. Затем вызывается метод `EntityManager.remove()`, которому передается управляемый экземпляр сущности контакта.

Метод `remove()` удалит запись контакта, а также всю связанную информацию, включая телефоны и хобби, поскольку в отображении было определено `cascade=CascadeType.ALL`. Для тестирования операции удаления добавьте фрагмент кода, приведенный в листинге 10.21, в метод `main()` класса `JpaSample`.

Листинг 10.21. Тестирование операции удаления

```
      // Удалить контакт.
      contact = contactService.findById(1l);
      contactService.delete(contact);
      contacts = contactService.findAllWithDetail();
      listContactsWithDetail(contacts);
```

В предыдущем листинге извлекается контакт с идентификатором 1, после чего вызывается метод `delete()` для удаления информации о контакте. Запуск программы дает следующий вывод (показан не полностью):

```
Listing contacts with details:
Contact - Id: 4, First name: Michael, Last name: Jackson, Birthday: 2011-10-13
Contact Tel Detail - Id: 4, Contact id: 4, Type: Home, Number: 1111111111
Contact Tel Detail - Id: 5, Contact id: 4, Type: Mobile, Number: 2222222222

Contact - Id: 3, First name: John, Last name: Smith, Birthday: 1964-02-28

Contact - Id: 2, First name: Scott, Last name: Tiger, Birthday: 1990-11-02
Contact Tel Detail - Id: 3, Contact id: 2, Type: Home, Number: 1234567890
Hobby :Swimming
```

В выводе видно, что контакт с идентификатором 1 был удален.

Собственный запрос

После обсуждения элементарных операций базы данных с применением JPA давайте перейдем к некоторым более сложным темам. Временами требуется полный контроль над запросом, который будет отправлен базе данных. Примером может служить иерархический запрос в базе данных Oracle. Такая разновидность запроса является специфичной для базы данных и называется *собственным запросом* (native query).

В JPA поддерживается выполнение собственных запросов; `EntityManager` отправит такой запрос базе данных в том виде, как он есть, не производя никакого отображения или трансформации. Вас может удивить, почему мы не пользуемся непосредственно JDBC, ведь JDBC поддерживает прямую отправку запросов базе данных. Но Spring также предлагает хорошую поддержку программирования логики доступа JDBC и выполнения отображения строк на объекты POJO.

Главным преимуществом применения собственных запросов JPA является отображение результирующего набора на сущностные классы ORM. В следующих двух разделах будет показано, как использовать собственный запрос для извлечения всех контактов и прямого отображения результирующего набора на объекты `Contact`.

Простой собственный запрос

Для демонстрации работы с собственным запросом мы реализуем новый метод, который будет извлекать все контакты из базы данных с помощью собственного запроса. Этот новый метод в интерфейсе `ContactService` представлен в листинге 10.22.

Листинг 10.22. Метод `findAllByNativeQuery()`

```
package com.apress.prospring3.ch10.service;

// Остальной код не показан.
public interface ContactService {
   // Остальной код не показан.

   // Найти все контакты с помощью собственного запроса.
   public List<Contact> findAllByNativeQuery();
}
```

В листинге 10.23 приведена реализация метода `findAllByNativeQuery()`.

Листинг 10.23. Реализация метода `findAllByNativeQuery()`

```
package com.apress.prospring3.ch10.service.jpa;

// Операторы импорта опущены.
@Service("jpaContactService")
@Repository
@Transactional
public class ContactServiceImpl implements ContactService {
   private Log log = LogFactory.getLog(ContactServiceImpl.class);
   // Остальной код не показан.
   final static String ALL_CONTACT_NATIVE_QUERY =
      "select id, first_name, last_name, birth_date, version from contact";
   @Transactional(readOnly=true)
   public List<Contact> findAllByNativeQuery() {
      return em.createNativeQuery(ALL_CONTACT_NATIVE_QUERY,
         Contact.class).getResultList();
   }
}
```

Как показано в листинге 10.23, собственный запрос можно рассматривать как простой SQL-оператор для извлечения всех столбцов из таблицы CONTACT. Чтобы создать и запустить запрос, сначала вызывается метод EntityManager.createNativeQuery(), которому передается строка запроса и тип результата. Типом результата должен быть отображенный сущностный класс (в этом случае класс Contact).

Метод createNativeQuery() возвращает интерфейс Query, который предоставляет операцию getResultList() для получения результирующего списка. Поставщик JPA выполнит запрос и преобразует результирующий набор в экземпляры сущностей, основываясь на отображениях JPA, которые были определены в сущностном классе. Запуск приведенного выше метода даст тот же самый результат, что и запуск метода findAll().

Собственный запрос с отображением результирующего набора SQL

В предшествующем разделе методу EntityManager.createNativeQuery() кроме отображенного объекта предметной области также передавалась строка, указывающая имя отображения результирующего набора SQL. Отображение результирующего набора SQL определяется на уровне сущностного класса с использованием аннотации @SqlResultSetMapping. Отображение результирующего набора SQL может иметь один или более отображений сущностей и столбцов.

Давайте определим простое отображение результирующего набора SQL в сущностном классе Contact (листинг 10.24).

Листинг 10.24. Использование отображения результирующего набора SQL

```
package com.apress.prospring3.ch10.domain;
// Остальной код не показан.
@SqlResultSetMapping(
    name="contactResult",
    entities=@EntityResult(entityClass=Contact.class)
)
public class Contact implements Serializable {
...
```

Здесь для сущностного класса было определено отображение результирующего набора SQL по имени contactResult с указанием в атрибуте entityClass самого класса Contact. В JPA поддерживается более сложное отображение для множества сущностей, а также отображение вплоть до уровня отдельных столбцов.

После определения отображения результирующего набора SQL метод findAllBy NativeQuery() может вызываться с применением имени этого отображения. Соответствующий фрагмент кода показан в листинге 10.25.

Листинг 10.25. Реализация метода findAllByNativeQuery() с использованием отображения результирующего набора SQL

```
package com.apress.prospring3.ch10.service.jpa;
// Остальной код не показан.
public class ContactServiceImpl implements ContactService {
   // Остальной код не показан.
   @Transactional(readOnly=true)
   public List<Contact> findAllByNativeQuery() {
      return em.createNativeQuery(ALL_CONTACT_NATIVE_QUERY,
         "contactResult").getResultList();
   }
}
```

Как видите, JPA также поддерживает запуск собственных запросов с гибкими возможностями отображения результирующих наборов SQL.

Запрос с критерием, использующий API-интерфейс критериев JPA 2

Большинство приложений будут предоставлять пользовательские интерфейсы для поиска информации. Скорее всего, будет отображаться большое количество поисковых полей, но пользователи будут вводить что-нибудь только в некоторых из них и затем запускать поиск. Очень трудно подготовить запросы для каждой комбинации параметров, которые пользователи могут вводить, ввиду слишком большого их числа. В этой ситуации на помощь приходит средство запросов с API-интерфейсом критериев.

В JPA 2 крупным нововведением стала функциональность запросов, использующих строго типизированный API-интерфейс критериев. В этом новом API-интерфейсе критериев передаваемый в запрос критерий основывается на метамодели отображенных сущностных классов. В результате каждый указанный критерий является строго типизированным, и ошибки обнаруживаются на этапе компиляции, а не во время выполнения.

В API-интерфейсе критериев JPA метамодель сущностного класса представляется именем сущностного класса с суффиксом в виде символа подчеркивания (_). Например, класс метамодели для сущностного класса `Contact` имеет имя `Contact_`. Код класса `Contact_` приведен в листинге 10.26.

Листинг 10.26. Строго типизированный API-интерфейс критериев JPA 2: метамодель

```
package com.apress.prospring3.ch10.domain;

import java.util.Date;
import javax.persistence.metamodel.SetAttribute;
import javax.persistence.metamodel.SingularAttribute;
import javax.persistence.metamodel.StaticMetamodel;

@StaticMetamodel(Contact.class)
public abstract class Contact_ {

public static volatile SingularAttribute<Contact, Long> id;
   public static volatile SetAttribute<Contact, ContactTelDetail>
      contactTelDetails;
   public static volatile SingularAttribute<Contact, String> lastName;
   public static volatile SingularAttribute<Contact, Date> birthDate;
   public static volatile SetAttribute<Contact, Hobby> hobbies;
   public static volatile SingularAttribute<Contact, String> firstName;
   public static volatile SingularAttribute<Contact, Integer> version;
}
```

В листинге 10.26 видно, что класс метамодели аннотирован с помощью `@StaticMetamodel` с указанием в атрибуте отображенного сущностного класса. Внутри класса присутствуют объявления всех атрибутов и связанных с ними типов.

Кодирование и сопровождение таких классов метамоделей были бы довольно утомительными. К счастью, доступны инструменты, которые генерируют классы метамоделей автоматически на основе отображений JPA в рамках сущностных классов. Инфраструктура Hibernate предлагает такой инструмент, который называется Hibernate Metamodel Generator (генератор метамоделей Hibernate; `www.hibernate.org/subprojects/jpamodelgen.html`). С помощью этого инструмента в некоторых конфигурациях STS можно автоматически генерировать и обновлять классы метамоделей при каждой сборке проекта.

Давайте рассмотрим процедуру включения автоматической генерации классов метамоделей для сущностных классов JPA. Во-первых, нам необходимы JAR-файлы, описанные в табл. 10.1. Оба JAR-файла могут быть найдены в загружаемом пакете на ранее упомянутом веб-сайте. Мы пользовались версией 1.1.1.

Таблица 10.1. JAR-файлы для генерации классов метамоделей

JAR-файл	Описание
`hibernate-jpamodelgen-1.1.1.Final.jar`	Главная библиотека для генерации классов метамоделей. После распаковки загруженного пакета вы найдете файл в `hibernate-jpamodelgen-1.1.1.Final`
`hibernate-jpa-2.0-api-1.0.0.Final.jar`	Стандартная библиотека JPA, предлагаемая Hibernate. После распаковки загруженного пакета вы найдете файл в `hibernate-jpamodelgen-1.1.1.Final\lib`

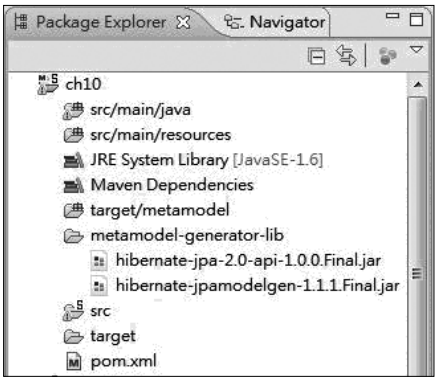


Рис. 10.4. Библиотека генератора метамоделей Hibernate в STS

Во-вторых, скопируйте полученные файлы в папку проекта. Например, JAR-файлы могут быть помещены в папку по имени `metamodel-generator-lib` в корневой папке проекта, как показано на рис. 10.4.

После помещения JAR-файлов в нужное место щелкните правой кнопкой мыши на имени проекта и выберите в контекстном меню пункт Properties (Свойства). Последовательно выберите элементы Java Compiler (Компилятор Java) и Annotation Processing (Обработка аннотаций), а затем отметьте флажки Enable project specific settings (Разрешить настройки, специфичные для проекта) и Enable annotation processing (Разрешить обработку аннотаций). В поле Generated source directory (Каталог для сгенерированных исходных текстов) введите **`target/metamodel`**. Окно свойств проекта показано на рис. 10.5.

После этого щелкните на элементе Factory Path (Путь фабрики), расположенном слева, отметьте флажок Enable project specific settings (Разрешить настройки, специфичные для проекта), щелкните на кнопке Add JARs (Добавить JAR-файлы) и в открывшемся диалоговом окне выберите упомянутые ранее два JAR-файла для добавления в путь фабрики. На рис. 10.6 показаны выбранные файлы.

По завершении щелкните на кнопке OK, и STS запросит, нужно ли пересобрать проект. Щелчок на кнопке Yes (Да) приведет к повторной сборке проекта, и средство обработки аннотаций сгенерирует классы метамоделей. Сгенерированные классы метамоделей можно видеть на рис. 10.7.

После генерации классов метамоделей можно реализовать более гибкие запросы, используя строго типизированный API-интерфейс критериев JPA 2. Давайте определим запрос, который принимает имя и фамилию для поиска контактов.

В листинге 10.27 приведено определение нового метода `findByCriteriaQuery()` в интерфейсе `ContactService`.

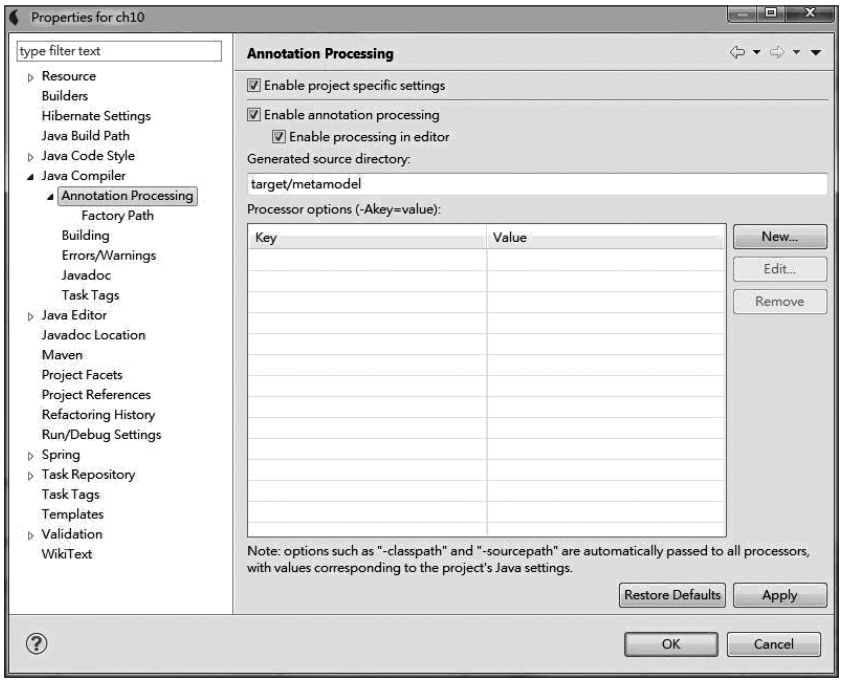


Рис. 10.5. Включение обработки аннотаций в STS

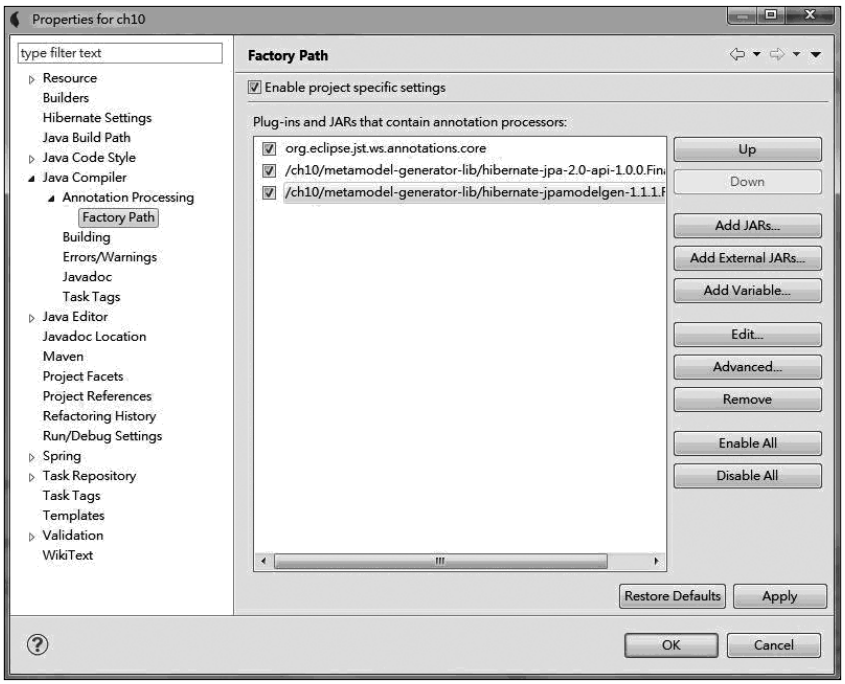


Рис. 10.6. Настройка пути фабрики для обработки аннотаций в STS

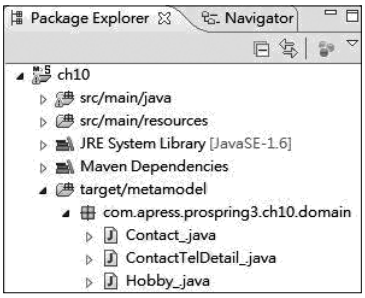


Рис. 10.7. Сгенерированные классы метамоделей

Листинг 10.27. Метод `findByCriteriaQuery()`

```
package com.apress.prospring3.ch10.service;

// Остальной код не показан.
public interface ContactService {

   // Найти контакты с помощью запроса с критерием.
   public List<Contact> findByCriteriaQuery(
      String firstName, String lastName);
}
```

В листинге 10.28 показана реализация метода `findByCriteriaQuery()`, использующая запрос с критерием JPA 2.

Листинг 10.28. Реализация метода `findByCriteriaQuery()`

```
package com.apress.prospring3.ch10.service.jpa;

// Остальной код не показан.
public class ContactServiceImpl implements ContactService {

   @Transactional(readOnly=true)
   public List<Contact> findByCriteriaQuery(
      String firstName, String lastName) {

      log.info("Finding contact for firstName: " + firstName
              + " and lastName: " + lastName);

      CriteriaBuilder cb = em.getCriteriaBuilder();
      CriteriaQuery<Contact> criteriaQuery = cb.createQuery(Contact.class);
      Root<Contact> contactRoot = criteriaQuery.from(Contact.class);
      contactRoot.fetch(Contact_.contactTelDetails, JoinType.LEFT);
      contactRoot.fetch(Contact_.hobbies, JoinType.LEFT);

      criteriaQuery.select(contactRoot).distinct(true);

      Predicate criteria = cb.conjunction();

      // Имя.
      if (firstName != null) {
         Predicate p = cb.equal(contactRoot.get(Contact_.firstName),
            firstName);
          criteria = cb.and(criteria, p);
      }

      // Фамилия.
```

```
        if (lastName != null) {
            Predicate p = cb.equal(contactRoot.get(Contact_.lastName),
                lastName);
            criteria = cb.and(criteria, p);
        }
        criteriaQuery.where(criteria);
        List<Contact> result = em.createQuery(criteriaQuery).getResultList();
        return result;
    }
}
```

Рассмотрим основные моменты, связанные с листингом 10.28.

- Метод `EntityManager.getCriteriaBuilder()` вызывается для извлечения экземпляра `CriteriaBuilder`.
- С использованием метода `CriteriaBuilder.createQuery()`, которому передается `Contact` в качестве результирующего типа, создается типизированный запрос.
- Вызывается метод `CriteriaQuery.from()` с передачей ему сущностного класса. Результатом будет объект корня запроса (т.е. интерфейс `Root<Contact>`), соответствующий указанной сущности. Объект корня запроса формирует основу для путевых выражений внутри запроса.
- Два вызова метода `Root.fetch()` обеспечивают незамедлительную выборку ассоциаций, относящихся к телефонам и хобби. Аргумент `JoinType.LEFT` задает внешнее соединение. Вызов метода `Root.fetch()` со значением `JoinType.LEFT` во втором аргументе эквивалентно указанию операции соединения `left join fetch` в JPQL.
- Вызывается метод `CriteriaQuery.select()` с передачей ему объекта корня запроса в качестве результирующего типа. Вызов метода `distinct()` со значением `true` означает, что дублированные записи должны быть устранены.
- С помощью вызова метода `CriteriaBuilder.conjunction()` получается экземпляр `Predicate`; этот вызов означает, что было сделано объединение одного или более ограничений. Экземпляр `Predicate` может быть как простым, так и сложным предикатом; здесь предикат — это ограничение, которое указывает критерий выборки, определенный выражением.
- Производится проверка аргументов имени и фамилии. Если аргумент не равен `null`, создается новый экземпляр `Predicate` с применением метода интерфейса `CriteriaBuilder` (т.е. метода `CriteriaBuilder.and()`). Метод `equal()` предназначен для указания ограничения равенства; внутри него вызывается `Root.get()` с передачей соответствующего атрибута метамодели сущностного класса, к которому будет применяться ограничение. Сконструированный предикат затем объединяется с существующим предикатом (хранящимся в переменной `criteria`) с помощью вызова метода `CriteriaBuilder.and()`.
- Экземпляр `Predicate` создается со всеми критериями и ограничениями, после чего передается в виде конструкции `where` запросу посредством вызова метода `CriteriaQuery.where()`.
- Наконец, `CriteriaQuery` передается в `EntityManager`. Затем `EntityManager` конструирует запрос на основе переданного `CriteriaQuery`, выполняет его и возвращает результат.

Чтобы протестировать запрос с критерием, добавьте фрагмент кода, показанный в листинге 10.29, в метод `main()` класса `JpaSample`.

Листинг 10.29. Тестирование метода findByCriteriaQuery()

```
// Найти контакты с помощью запроса с критерием.
contacts = contactService.findByCriteriaQuery("John", "Smith");
listContactsWithDetail(contacts);
```

Запуск этой программы дает следующий вывод (показан не полностью):

```
2011-10-20 18:42:34,411 INFO [com.apress.prospring3.ch10.service.jpa.
ContactServiceImpl] - <Finding contact for firstName: John and lastName: Smith>
...
Listing contacts with details:
Contact - Id: 3, First name: John, Last name: Smith, Birthday: 1964-02-28
```

Попробуйте различные комбинации аргументов, а также значения `null`, и понаблюдайте за выводом.

Введение в проект Spring Data JPA

Проект Spring Data JPA является подпроектом в рамках проекта Spring Data. Главное назначение проекта Spring Data JPA состоит в предоставлении дополнительных возможностей для упрощения разработки приложений с использованием JPA.

Проект Spring Data JPA предлагает множество основных средств. В этом разделе мы обсудим два главных средства. Первое — это абстракция `Repository`, а второе — слушатель сущностей, предназначенный для отслеживания базовой информации аудита по сущностным классам.

Добавление библиотечных зависимостей Spring Data JPA

Для использования Spring Data JPA понадобится добавить к проекту несколько зависимостей. Обязательные зависимости Maven описаны в табл. 10.2.

Таблица 10.2. Зависимости Maven для Spring Data JPA

Идентификатор группы	Идентификатор артефакта	Версия	Описание
`org.springframework.data`	`spring-data-jpa`	1.0.1.RELEASE	Библиотека Spring Data JPA
`org.springframework`	`spring-aspects`	3.1.0.RELEASE	Расширенные функциональные средства АОП в Spring, от которых зависит Spring Data JPA
`joda-time`	`joda-time`	2.0	API-интерфейс для работы с датой и временем под названием Joda-Time (`http://joda-time.sourceforge.net/`), который используется в Spring Data JPA
`joda-time`	`joda-time-hibernate`	1.3	Библиотека Joda-Time для интеграции с Hibernate, обеспечивающая сохранение типов данных даты и времени
`com.google.guava`	`guava`	10.0.1	Полезные вспомогательные классы

В STS для простого добавления этих зависимостей можно воспользоваться редактором POM. Дважды щелкните на файле `pom.xml` в проводнике пакетов. В окне редактора POM щелкните на вкладке Dependencies (Зависимости) и введите дополнительные зависимости. На рис. 10.8 показано добавление `spring-data-jpa`.

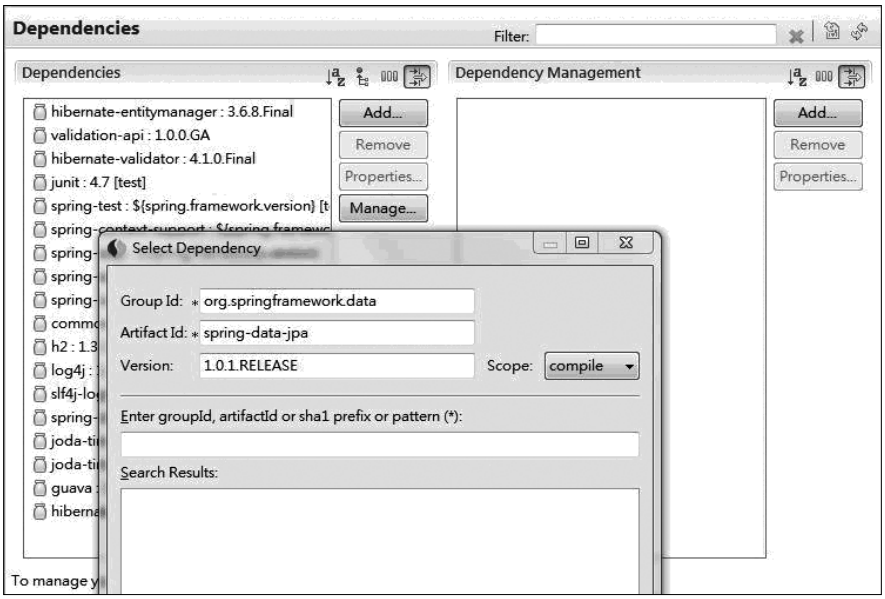


Рис. 10.8. Добавление зависимостей к проекту Spring

На рис. 10.9 представлен окончательный список зависимостей в STS.

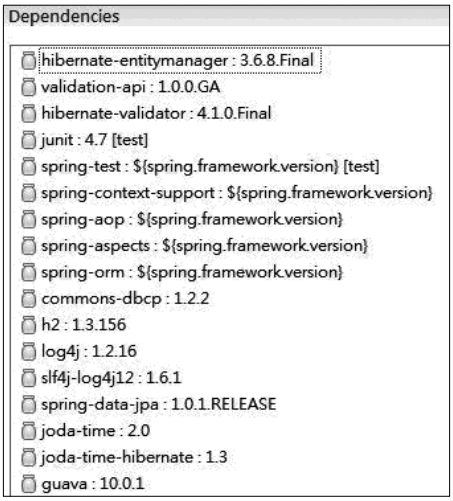


Рис. 10.9. Зависимости Maven для проекта Spring Data JPA

После добавления обязательных зависимостей можно приступать к исследованию возможностей Spring Data JPA.

Операции базы данных, использующие абстракцию `Repository` из проекта Spring Data JPA

Одной из главных концепций проекта Spring Data и всех его подпроектов является абстракция `Repository` (репозиторий), которая принадлежит проекту Spring Data Commons (`www.springsource.org/spring-data/commons`). На время написания этой книги актуальной была версия 1.1.0.RELEASE. В Spring Data JPA абстракция репозитория представляет собой оболочку вокруг JPA-интерфейса `EntityManager` и предлагает более простой интерфейс для доступа к данным на основе JPA. Центральным интерфейсом в рамках Spring Data является `org.springframework.data.repository.Repository<T,ID extends Serializable>`, который представляет собой маркерный интерфейс, принадлежащий дистрибутиву Spring Data Commons. В проекте Spring Data предоставляются различные расширения интерфейса `Repository`; одно из них — это интерфейс `org.springframework.data.repository.CrudRepository` (также принадлежащий проекту Spring Data Commons), который мы обсудим в настоящем разделе.

Интерфейс `CrudRepository` предоставляет несколько часто используемых методов. В листинге 10.30 показано объявление этого интерфейса, взятое из исходного кода проекта Spring Data Commons.

Листинг 10.30. Интерфейс `CrudRepository`

```
package org.springframework.data.repository;

import java.io.Serializable;

@NoRepositoryBean
public interface CrudRepository<T, ID extends Serializable> extends
Repository<T, ID> {

   T save(T entity);

   Iterable<T> save(Iterable<? extends T> entities);

   T findOne(ID id);

   boolean exists(ID id);

   Iterable<T> findAll();

   long count();

   void delete(ID id);

   void delete(T entity);

   void delete(Iterable<? extends T> entities);

   void deleteAll();
}
```

Имена методов вполне очевидны. Удобнее всего рассматривать работу абстракции `Repository` на простом примере. Давайте немного переделаем интерфейс `ContactService`, в частности, его три метода поиска. Модифицированный интерфейс `ContactService` показан в листинге 10.31. После такой модификации интерфейса `ContactService` классы `com.apress.prospring3.ch10.JpaSample` и `com.apress.prospring3.ch10.service.jpa.ContactServiceImpl` будут содержать ошибки. Просто удалите эти два класса и продолжайте реализацию интерфейса с использованием Spring Data JPA.

Листинг 10.31. Модифицированный интерфейс ContactService

```
package com.apress.prospring3.ch10.service;

import java.util.List;
import com.apress.prospring3.ch10.domain.Contact;

public interface ContactService {

    // Найти все контакты.
    public List<Contact> findAll();

    // Найти контакты по имени.
    public List<Contact> findByFirstName(String firstName);

    // Найти контакты по имени и фамилии.
    public List<Contact> findByFirstNameAndLastName(
        String firstName, String lastName);
}
```

Следующий шаг заключается в подготовке интерфейса ContactRepository, который расширяет интерфейс CrudRepository. Интерфейс ContactRepository представлен в листинге 10.32.

Листинг 10.32. Интерфейс ContactRepository

```
package com.apress.prospring3.ch10.repository;

import java.util.List;

import org.springframework.data.repository.CrudRepository;

import com.apress.prospring3.ch10.domain.Contact;

public interface ContactRepository extends CrudRepository<Contact, Long> {

    public List<Contact> findByFirstName(String firstName);

    public List<Contact> findByFirstNameAndLastName(
        String firstName, String lastName);

}
```

В этом интерфейсе нам необходимо просто объявить два метода, поскольку метод findAll() уже предоставлен как CrudRepository.findAll(). В листинге 10.32 видно, что интерфейс ContactRepository расширяет интерфейс CrudRepository, передавая ему сущностный класс (Contact) и тип идентификатора (Long). Один причудливый аспект абстракции Repository из Spring Data заключается в том, что если соблюдать общее соглашение об именовании, такое как findByFirstName и findByFirstNameAndLastName, то не понадобится предоставлять Spring Data JPA именованный запрос. Вместо этого Spring Data JPA самостоятельно “выведет” и сконструирует запрос на основе имени метода.

Например, для метода findByFirstName() будет автоматически подготовлен запрос select c from Contact c where c.firstName = :firstName и установлен именованный параметр firstName из аргумента.

Чтобы пользоваться абстракцией Repository, необходимо определить ее в конфигурации Spring. В листинге 10.33 приведен конфигурационный файл (spring-data-app-context.xml).

Листинг 10.33. Конфигурация репозитория JPA в Spring

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:context="http://www.springframework.org/schema/context"
   xmlns:jdbc="http://www.springframework.org/schema/jdbc"
   xmlns:jpa="http://www.springframework.org/schema/data/jpa"
   xmlns:tx="http://www.springframework.org/schema/tx"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context-3.1.xsd
      http://www.springframework.org/schema/jdbc
      http://www.springframework.org/schema/jdbc/spring-jdbc-3.1.xsd
      http://www.springframework.org/schema/data/jpa
      http://www.springframework.org/schema/data/jpa/spring-jpa-1.0.xsd
      http://www.springframework.org/schema/tx
      http://www.springframework.org/schema/tx/spring-tx-3.1.xsd">
   <jdbc:embedded-database id="dataSource" type="H2">
      <jdbc:script location="classpath:schema.sql"/>
      <jdbc:script location="classpath:test-data.sql"/>
   </jdbc:embedded-database>
   <bean id="transactionManager"
      class="org.springframework.orm.jpa.JpaTransactionManager">
      <property name="entityManagerFactory" ref="emf"/>
   </bean>
   <tx:annotation-driven transaction-manager="transactionManager" />
   <bean id="emf"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
      <property name="dataSource" ref="dataSource" />
      <property name="jpaVendorAdapter">
       <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter" />
      </property>
      <property name="packagesToScan"
          value="com.apress.prospring3.ch10.domain"/>
      <property name="jpaProperties">
         <props>
            <prop key="hibernate.dialect">
               org.hibernate.dialect.H2Dialect
            </prop>
            <prop key="hibernate.max_fetch_depth">3</prop>
            <prop key="hibernate.jdbc.fetch_size">50</prop>
            <prop key="hibernate.jdbc.batch_size">10</prop>
            <prop key="hibernate.show_sql">true</prop>
         </props>
      </property>
   </bean>
   <context:annotation-config/>
   <context:component-scan
        base-package="com.apress.prospring3.ch10.service.springjpa"/>
   <jpa:repositories base-package="com.apress.prospring3.ch10.repository"
                  entity-manager-factory-ref="emf"
                  transaction-manager-ref="transactionManager"/>
</beans>
```

Сначала нужно добавить в конфигурационный файл пространство имен jpa. Затем с помощью дескриптора <jpa:repositories> конфигурируется абстракция Repository из Spring Data JPA. Мы указываем Spring на необходимость сканирования пакета com.apress.prospring3.ch10.repository на предмет интерфейсов репозитория, а также передачи EntityManagerFactory и диспетчера транзакций соответственно.

Теперь можно приступать к реализации. В листинге 10.34 показана реализация трех методов поиска из интерфейса ContactService.

Листинг 10.34. Класс ContactServiceImpl

```
package com.apress.prospring3.ch10.service.springjpa;

// Операторы импорта опущены.
@Service("springJpaContactService")
@Repository
@Transactional
public class ContactServiceImpl implements ContactService {

   @Autowired
   private ContactRepository contactRepository;

   @Transactional(readOnly=true)
   public List<Contact> findAll() {
      return Lists.newArrayList(contactRepository.findAll());
   }

   @Transactional(readOnly=true)
   public List<Contact> findByFirstName(String firstName) {
      return contactRepository.findByFirstName(firstName);
   }

   @Transactional(readOnly=true)
   public List<Contact> findByFirstNameAndLastName(
      String firstName, String lastName) {
      return contactRepository.findByFirstNameAndLastName(
         firstName, lastName);
   }
}
```

Как видите, вместо EntityManager в класс службы мы должны внедрить интерфейс ContactRepository, и Spring Data JPA самостоятельно сделает всю черновую работу.

В листинге 10.35 приведена тестовая программа.

Листинг 10.35. Класс SpringJpaSample

```
package com.apress.prospring3.ch10;

// Операторы импорта опущены.
public class SpringJpaSample {

   public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:spring-data-app-context.xml");
      ctx.refresh();
      ContactService contactService = ctx.getBean(
         "springJpaContactService", ContactService.class);

      // Найти все контакты.
      List<Contact> contacts = contactService.findAll();
      listContacts(contacts);
```

```
        // Найти контакты по имени.
        contacts = contactService.findByFirstName("Clarence");
        listContacts(contacts);

        // Найти контакты по имени и фамилии.
        contacts = contactService.findByFirstNameAndLastName("Clarence", "Ho");
        listContacts(contacts);
    }

    private static void listContacts(List<Contact> contacts) {
        System.out.println("");
        System.out.println("Listing contacts without details:");
        for (Contact contact: contacts) {
            System.out.println(contact);
            System.out.println();
        }
    }
}
```

В этом классе нет ничего особенного. Запуск программы, как и ожидалось, приведет к выводу списка контактов. Мы оставляем тестирование на самостоятельную проработку.

Итак, вы увидели, как Spring Data JPA может упростить разработку. Мы не должны создавать именованный запрос, вызывать метод `EntityManager.createQuery()` и т.д.

Ранее были рассмотрены только простейшие примеры. Абстракция `Repository` поддерживает множество функциональных возможностей, включая определение специальных запросов с помощью аннотации `@Query`, средство `Specification` (принадлежащее версии 1.0.1.RELEASE проекта Spring Data JPA) и т.п. За дополнительными сведениями обращайтесь в документацию SpringSource:

- Проект Spring Data Commons: `http://static.springsource.org/spring-data/data-commons/docs/current/reference/html/`
- Проект Spring Data JPA: `http://static.springsource.org/spring-data/data-jpa/docs/current/reference/html/`

Отслеживание изменений в сущностном классе

В большинстве приложений требуется отслеживать основную информацию аудита для бизнес-данных, поддерживаемых пользователями. Информация аудита обычно включает имя пользователя, создавшего данные, дату создания, дату последней модификации, а также имя пользователя, который провел эту последнюю модификацию.

Проект Spring Data JPA предлагает функцию в форме слушателя сущностей JPA, который помогает отслеживать такую информацию аудита автоматически. Чтобы пользоваться упомянутым средством, сущностный класс должен реализовать интерфейс `org.springframework.data.domain.Auditable<U,ID extends Serializable>` (принадлежащий Spring Data Commons) или расширять любой класс, который реализует этот интерфейс. В листинге 10.36 показан интерфейс `Auditable`, который был взят из документации по Spring Data.

Листинг 10.36. Интерфейс `Auditable`

```
package org.springframework.data.domain;

import java.io.Serializable;

import org.joda.time.DateTime;
```

```
public interface Auditable<U, ID extends Serializable>
   extends Persistable<ID> {

   U getCreatedBy();
   void setCreatedBy(final U createdBy);

   DateTime getCreatedDate();
   void setCreated(final DateTime creationDate);

   U getLastModifiedBy();
   void setLastModifiedBy(final U lastModifiedBy);

   DateTime getLastModifiedDate();
   void setLastModified(final DateTime lastModifiedDate);
}
```

Чтобы посмотреть, как это работает, создадим в схеме базы данных новую таблицу по имени CONTACT_AUDIT, которая основана на таблице CONTACT, с добавленными четырьмя столбцами, связанными с аудитом. В листинге 10.37 приведен сценарий создания таблицы (schema.sql).

Листинг 10.37. Таблица CONTACT _ AUDIT

```
CREATE TABLE CONTACT_AUDIT (
     ID INT NOT NULL AUTO_INCREMENT
   , FIRST_NAME VARCHAR(60) NOT NULL
   , LAST_NAME VARCHAR(40) NOT NULL
   , BIRTH_DATE DATE
   , VERSION INT NOT NULL DEFAULT 0
   , CREATED_BY VARCHAR(20)
   , CREATED_DATE TIMESTAMP
   , LAST_MODIFIED_BY VARCHAR(20)
   , LAST_MODIFIED_DATE TIMESTAMP
   , UNIQUE UQ_CONTACT_AUDIT_1 (FIRST_NAME, LAST_NAME)
   , PRIMARY KEY (ID)
);
```

Четыре столбца, связанные с аудитом, в листинге 10.37 выделены полужирным. Следующий шаг состоит в создании сущностного класса по имени ContactAudit. В основном он такой же, как сущностный класс Contact; мы просто добавляем отображение для четырех столбцов аудита. Кроме того, этот класс реализует интерфейс Auditable. В листинге 10.38 показан фрагмент кода, который добавляется в начало класса Contact.

Листинг 10.38. Класс ContactAudit

```
package com.apress.prospring3.ch10.domain;

// Операторы импорта опущены.
@Entity
@Table(name = "contact_audit")
public class ContactAudit implements Auditable<String, Long>, Serializable {

   // Объявления других свойств не показаны.

   // Поля аудита.
   private String createdBy;
   private DateTime createdDate;
   private String lastModifiedBy;
   private DateTime lastModifiedDate;
```

```
    @Column(name="CREATED_BY")
    public String getCreatedBy() {
        return createdBy;
    }

    public void setCreatedBy(String createdBy) {
        this.createdBy = createdBy;
    }

    @Column(name="CREATED_DATE")
    @Type(type="org.joda.time.contrib.hibernate.PersistentDateTime")
    public DateTime getCreatedDate() {
        return createdDate;
    }

    public void setCreatedDate(DateTime createdDate) {
        this.createdDate = createdDate;
    }

    @Column(name="LAST_MODIFIED_BY")
    public String getLastModifiedBy() {
        return lastModifiedBy;
    }

    public void setLastModifiedBy(String lastModifiedBy) {
        this.lastModifiedBy = lastModifiedBy;
    }

    @Column(name="LAST_MODIFIED_DATE")
    @Type(type="org.joda.time.contrib.hibernate.PersistentDateTime")
    public DateTime getLastModifiedDate() {
        return lastModifiedDate;
    }

    public void setLastModifiedDate(DateTime lastModifiedDate) {
        this.lastModifiedDate = lastModifiedDate;
    }

    @Transient
    public boolean isNew() {
        if (id == null) {
            return true;
        } else {
            return false;
        }
    }

    public String toString() {
        return "Contact - Id: " + id + ", First name: " + firstName
            + ", Last name: " + lastName + ", Birthday: " + birthDate
            + ", Create by: " + createdBy + ", Create date: " + createdDate
            + ", Modified by: " + lastModifiedBy + ", Modified date: "
            + lastModifiedDate;
    }
}
```

В листинге 10.38 сущностный класс `ContactAudit` реализует интерфейс `Auditable` и реализует методы за счет отображения четырех столбцов аудита. Аннотации `@Column` применяются для отображения на действительные столбцы в таблице. Для двух атрибутов даты (`createdDate`, `lastModifiedDate`) применяется аннотация специально-

го пользовательского типа Hibernate по имени `@Type` с указанием класса реализации `org.joda.time.contrib.hibernate.PersistentDateTime`. Подобно тому, как интерфейс `Auditable` из Spring Data JPA использует тип `DateTime` из библиотеки Joda-Time, библиотека `joda-time-hibernate` предоставляет этот специальный пользовательский тип для применения с Hibernate при сохранении атрибута в столбце `TIMESTAMP` базы данных. Метод `isNew()` интерфейса `Auditable` (унаследованного от интерфейса `org.springframework.data.domain.Persistable<ID extends Serializable>`) также реализован. Аннотация `@Transient` означает, что поле не должно сохраняться. Эта функция используется Spring Data JPA для идентификации, является ли это новой сущностью, при определении необходимости установки атрибутов `createdBy` и `createdDate`. В реализации мы просто проверяем идентификатор, и если он равен `null`, мы возвращаем `true`, обозначая новый экземпляр сущности.

В листинге 10.39 представлен интерфейс `ContactAuditService`, в котором мы определяем лишь несколько методов для демонстрации возможностей аудита.

Листинг 10.39. Интерфейс `ContactAuditService`

```
package com.apress.prospring3.ch10.service;

import java.util.List;

import com.apress.prospring3.ch10.domain.ContactAudit;

public interface ContactAuditService {

   public List<ContactAudit> findAll();

   public ContactAudit findById(Long id);

   public ContactAudit save(ContactAudit contact);

}
```

Следующий шаг заключается в создании интерфейса `ContactAuditRepository`, который показан в листинге 10.40.

Листинг 10.40. Интерфейс `ContactAuditRepository`

```
package com.apress.prospring3.ch10.repository;

import org.springframework.data.repository.CrudRepository;

import com.apress.prospring3.ch10.domain.ContactAudit;

public interface ContactAuditRepository extends
    CrudRepository<ContactAudit, Long> {

}
```

Интерфейс `ContactAuditRepository` расширяет `CrudRepository`, где уже реализованы все методы, которые мы собираемся реализовать для `ContactAuditService`. Метод `findById()` реализован посредством метода `CrudRepository.findOne()`.

В листинге 10.41 представлен класс реализации `ContactAuditServiceImpl`.

Листинг 10.41. Класс `ContactAuditServiceImpl`

```
package com.apress.prospring3.ch10.service.springjpa;

// Операторы импорта опущены.
@Service("contactAuditService")
@Repository
```

```
@Transactional
public class ContactAuditServiceImpl implements ContactAuditService {

   @Autowired
   private ContactAuditRepository contactAuditRepository;

   @Transactional(readOnly=true)
   public List<ContactAudit> findAll() {
      return Lists.newArrayList(contactAuditRepository.findAll());
   }

   public ContactAudit findById(Long id) {
      return contactAuditRepository.findOne(id);
   }

   public ContactAudit save(ContactAudit contact) {
      return contactAuditRepository.save(contact);
   }

}
```

Этот код может быть уже знаком. Интерфейс `ContactAuditRepository` является автосвязываемым и все методы в интерфейсе `ContactAuditService` реализованы.

Нам также необходимо добавить некоторую конфигурацию. Первое, что следует добавить — это объявление `AuditingEntityListener<T>`, который представляет собой слушатель сущностей JPA, предоставляющий службу аудита. Чтобы объявить слушатель, создайте файл по имени `/src/main/resources/META-INF/orm.xml` (использовать такое имя файла обязательно, и это указано в спецификации JPA) в корневой папке проекта и объявите слушатель, как показано в листинге 10.42.

Листинг 10.42. Объявление слушателя сущностей

```
<?xml version="1.0" encoding="UTF-8" ?>
<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
      http://java.sun.com/xml/ns/persistence/orm_2_0.xsd"
      version="2.0">
   <description>JPA</description>

   <persistence-unit-metadata>
      <persistence-unit-defaults>
         <entity-listeners>
            <entity-listener
class="org.springframework.data.jpa.domain.support.AuditingEntityListener" />
      </entity-listeners>
      </persistence-unit-defaults>
   </persistence-unit-metadata>

</entity-mappings>
```

Слушатель будет выбран поставщиком JPA во время выполнения действий, связанных с постоянством (т.е. события сохранения и обновления), для обработки полей аудита.

Мы также должны определить слушатель в конфигурации Spring. Соответствующий фрагмент кода показан в листинге 10.43 (из файла `spring-data-app-context.xml`).

Листинг 10.43. Объявление слушателя сущностей в Spring

```
<!-- Другие определения не показаны. -->
<jpa:auditing auditor-aware-ref="auditorAwareBean"/>

<bean id="auditorAwareBean"
  class="com.apress.prospring3.ch10.springjpa.auditor.AuditorAwareBean"/>
```

Дескриптор `<jpa:auditing>` предназначен для включения средства аудита Spring Data JPA, а `auditorAwareBean` — это бин, предоставляющий пользовательскую информацию. В листинге 10.44 приведен класс `AuditorAwareBean`.

Листинг 10.44. Класс `AuditorAwareBean`

```
package com.apress.prospring3.ch10.springjpa.auditor;

import org.springframework.data.domain.AuditorAware;

public class AuditorAwareBean implements AuditorAware<String> {

   public String getCurrentAuditor() {
      return "prospring3";
   }

}
```

Класс `AuditorAwareBean` реализует интерфейс `AuditorAware<T>`, передавая ему тип `String`. В реальных ситуациях это должен быть экземпляр пользовательской информации, например, класс `User`, который представляет зарегистрированного пользователя, выполнившего действие обновления данных. Мы используем здесь `String` только ради простоты.

В классе `AuditorAwareBean` реализован метод `getCurrentAuditor()`, и значение жестко закодировано как `prospring3`. В реальности пользователь должен быть получен из лежащей в основе инфраструктуры безопасности. Например, в Spring Security информация о пользователе может быть извлечена из класса `SecurityContextHolder`.

На этом вся работа по реализации завершена. В листинге 10.45 показана тестовая программа.

Листинг 10.45. Тестирование средств аудита Spring Data JPA

```
package com.apress.prospring3.ch10;

// Операторы импорта опущены.
public class SpringJpaAuditSample {

   public static void main(String[] args) {

      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:spring-data-app-context.xml");
      ctx.refresh();

      ContactAuditService contactService = ctx.getBean(
         "contactAuditService", ContactAuditService.class);

      List<ContactAudit> contacts = contactService.findAll();
      listContacts(contacts);

      // Добавить новый контакт.
      System.out.println("Add new contact");
      ContactAudit contact = new ContactAudit();
      contact.setFirstName("Michael");
```

```
        contact.setLastName("Jackson");
        contact.setBirthDate(new Date());
        contactService.save(contact);
        contacts = contactService.findAll();
        listContacts(contacts);

        // Найти по идентификатору.
        contact = contactService.findById(1l);
        System.out.println("");
        System.out.println("Contact with id 1:" + contact);
        System.out.println("");

        // Обновить контакт.
        System.out.println("Update contact");
        contact.setFirstName("Tom");
        contactService.save(contact);
        contacts = contactService.findAll();
        listContacts(contacts);
    }

    private static void listContacts(List<ContactAudit> contacts) {
        System.out.println("");
        System.out.println("Listing contacts without details:");
        for (ContactAudit contact: contacts) {
            System.out.println(contact);
            System.out.println();
        }
    }
}
```

В листинге 10.45 мы выводим информацию аудита контактов после вставки нового контакта и после его обновления. Запуск этой программы дает следующий вывод:

```
Add new contact
Listing contacts without details:
Contact - Id: 1, First name: Michael, Last name: Jackson, Birthday:
2011-10-21, Create by: prospring3,
Create date: 2011-10-21T10:45:52.483+08:00, Modified by: prospring3,
Modified date: 2011-10-21T10:45:52.483+08:00

Update contact
Listing contacts without details:
Contact - Id: 1, First name: Tom, Last name: Jackson, Birthday: 2011-10-21,
Create by: prospring3,
Create date: 2011-10-21T10:45:52.483+08:00, Modified by: prospring3,
Modified date: 2011-10-21T10:45:52.596+08:00
```

В приведенном выше выводе видно, что после создания нового контакта даты создания и последней модификации совпадают. Тем не менее, после обновления дата последней модификации изменяется. Аудит — это еще одно удобное средство, предоставляемое Spring Data JPA, так что самостоятельно реализовывать его логику не понадобится.

Итак, обсуждение Spring Data JPA завершено. Давайте теперь перейдем к рассмотрению следующей полезной функции, предлагаемой Hibernate для отслеживания версий сущностей.

Отслеживание версий сущностей с использованием Hibernate Envers

В корпоративном приложении для критически важных бизнес-данных всегда существует требование хранить "версии" каждой сущности. Например, в системе управления взаимосвязями с клиентами (customer relationship management — CRM) каждый раз, когда запись клиента вставляется, обновляется или удаляется, ее предыдущая версия должна быть сохранена в таблице хронологии или аудита, чтобы удовлетворять требованиям аудита или другим стандартам, принятым на предприятии.

Для достижения указанных целей существуют два общих варианта. Первый — создание триггеров базы данных, которые будут создавать копию записи в состоянии до обновления в таблице хронологии перед любой операцией обновления, а второй — разработка соответствующей логики на уровне доступа к данным (например, с применением АОП). Тем не менее, обоим вариантам присущи недостатки. Подход с триггерами привязан к платформе базы данных, тогда как реализация логики вручную довольно неуклюжа и чревата ошибками.

Hibernate Envers (Entity Versioning System — система поддержки версий сущностей) — это модуль Hibernate, специально спроектированный для автоматизации поддержки версий сущностей. В этом разделе мы обсудим использование Hibernate Envers для реализации поддержки версий сущности `ContactAudit`.

На заметку! Hibernate Envers не является функциональным средством JPA. Мы упоминаем здесь этот модуль, поскольку считаем, что более уместно рассмотреть его после обсуждения ряда базовых средств аудита, которые доступны в Spring Data JPA. На самом деле поддержка записей хронологии для критически важных бизнес-данных (например, клиентов, транзакций и т.п.) представляет собой базовую возможность практически любого корпоративного приложения.

В Hibernate Envers поддерживаются две разных стратегии аудита, которые описаны в табл. 10.3.

Таблица 10.3. Стратегии аудита Hibernate Envers

Стратегия аудита	Описание
По умолчанию	Hibernate Envers будет поддерживать столбец для номера версии записи. Каждый раз, когда запись вставляется или обновляется, в таблицу хронологии вставляется новая запись с номером версии, извлеченным из последовательности базы данных или таблицы
Аудит достоверности	При такой стратегии сохраняются начальная и конечная версии каждой записи хронологии. Каждый раз, когда запись вставляется или обновляется, в таблицу хронологии вставляется новая запись с номером начальной версии. В то же самое время предыдущая запись обновляется номером конечной версии. Также возможно сконфигурировать Hibernate Envers на запись метки времени, когда конечная версия было обновлена, в предыдущей записи хронологии

В этом разделе мы продемонстрируем работу стратегии аудита достоверности. Хотя она приводит к запуску большего числа обновлений базы данных, извлечение записей хронологии становится намного быстрее. Поскольку метка времени конечной версии также записывается в записи хронологии, при запросе данных упрощается идентификация образа записи в специфический момент времени.

Добавление зависимостей Hibernate Envers

В проект понадобится добавить зависимость Maven, описанную в табл. 10.4.

Таблица 10.4. Зависимость Hibernate Envers

Идентификатор группы	Идентификатор артефакта	Версия	Описание
org.hibernate	hibernate-envers	3.6.8.Final	Библиотека модуля Hibernate Envers

Добавление таблиц для отслеживания версий сущностей

Для поддержки отслеживания версий сущностей необходимо добавить несколько таблиц. Первым делом, для каждой таблицы, для которой будут отслеживаться версии сущностей (в этом случае — сущностный класс ContactAudit), нужно создать соответствующую таблицу хронологии. Для отслеживания версий записей в таблице CONTACT_AUDIT создадим таблицу по имени CONTACT_AUDIT_H. В листинге 10.46 приведен сценарий создания этой таблицы (schema.sql).

Листинг 10.46. Таблица CONTACT_AUDIT_H

```
CREATE TABLE CONTACT_AUDIT_H (
     ID INT NOT NULL
   , FIRST_NAME VARCHAR(60) NOT NULL
   , LAST_NAME VARCHAR(40) NOT NULL
   , BIRTH_DATE DATE
   , VERSION INT NOT NULL DEFAULT 0
   , CREATED_BY VARCHAR(20)
   , CREATED_DATE TIMESTAMP
   , LAST_MODIFIED_BY VARCHAR(20)
   , LAST_MODIFIED_DATE TIMESTAMP
   , AUDIT_REVISION INT NOT NULL
   , ACTION_TYPE INT
   , AUDIT_REVISION_END INT
   , AUDIT_REVISION_END_TS TIMESTAMP
   , UNIQUE UQ_CONTACT_AUDIT_H_1 (FIRST_NAME, LAST_NAME)
   , PRIMARY KEY (ID, AUDIT_REVISION)
);
```

Для поддержки стратегии аудита достоверности в каждую таблицу хронологии должны быть добавлены четыре столбца (в листинге 10.46 они выделены полужирным). Эти столбцы кратко описаны в табл. 10.5.

Таблица 10.5. Столбцы, требуемые для таблицы хронологии

Имя столбца	Тип данных	Описание
AUDIT_REVISION	INT	Начальная версия записи хронологии
AUDIT_TYPE	INT	Тип действия. Возможные значения: 0 — добавление, 1 — модификация, 2 — удаление
AUDIT_REVISION_END	INT	Конечная версия записи хронологии
AUDIT_REVISION_END_TS	TIMESTAMP	Метка времени, когда была обновлена конечная версия

Hibernate Envers требует еще одну таблицу для отслеживания номеров версий и меток времени, когда каждая версия была создана. Эта таблица должна называться REVINFO. Сценарий создания этой таблицы представлен в листинге 10.47 (schema.sql).

Листинг 10.47. Таблица REVINFO

```
CREATE TABLE REVINFO (
     REVTSTMP BIGINT NOT NULL
   , REV INT NOT NULL AUTO_INCREMENT
   , PRIMARY KEY (REVTSTMP, REV)
);
```

Столбец REV предназначен для сохранения каждого номера версии, который будет автоматически инкрементироваться, когда создается новая запись хронологии. Столбец REVTSTMP хранит метку времени (в числовом формате), соответствующую созданию версии.

Конфигурирование EntityManagerFactory для отслеживания версий сущностей

Модуль Hibernate Envers реализован в форме слушателей EJB. Эти слушатели могут быть сконфигурированы в бине LocalContainerEntityManagerFactory.

В листинге 10.48 показана пересмотренная конфигурация бина (файл spring-data-app-context.xml).

Листинг 10.48. Конфигурирование Hibernate Envers в Spring

```
   <bean id="emf"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
      <!--  Другие параметры не показаны.  -->
      <property name="jpaProperties">
         <props>
            <!-- Другие свойства не показаны. -->
            <!-- Слушатели для Hibernate Envers -->
            <prop key="hibernate.ejb.event.post-insert">
               org.hibernate.ejb.event.EJB3PostInsertEventListener,
               org.hibernate.envers.event.AuditEventListener
            </prop>
            <prop key="hibernate.ejb.event.post-update">
               org.hibernate.ejb.event.EJB3PostUpdateEventListener,
               org.hibernate.envers.event.AuditEventListener
            </prop>
            <prop key="hibernate.ejb.event.post-delete">
               org.hibernate.ejb.event.EJB3PostDeleteEventListener,
               org.hibernate.envers.event.AuditEventListener
            </prop>
            <prop key="hibernate.ejb.event.pre-collection-update">
               org.hibernate.envers.event.AuditEventListener
            </prop>
            <prop key="hibernate.ejb.event.pre-collection-remove">
               org.hibernate.envers.event.AuditEventListener
            </prop>
            <prop key="hibernate.ejb.event.post-collection-recreate">
               org.hibernate.envers.event.AuditEventListener
            </prop>
```

```
            <!-- Свойства для Hibernate Envers -->
            <prop key="org.hibernate.envers.audit_table_suffix">_H</prop>
            <prop key="org.hibernate.envers.revision_field_name">
               AUDIT_REVISION
            </prop>
            <prop key="org.hibernate.envers.revision_type_field_name">
               ACTION_TYPE
            </prop>
            <prop key="org.hibernate.envers.audit_strategy">
              org.hibernate.envers.strategy.ValidityAuditStrategy
            </prop>
            <prop
key="org.hibernate.envers.audit_strategy_validity_end_rev_field_name">
               AUDIT_REVISION_END
            </prop>
            <prop
key="org.hibernate.envers.audit_strategy_validity_store_revend_timestamp">
               True
            </prop>
            <prop
key="org.hibernate.envers.audit_strategy_validity_revend_timestamp_field_name">
               AUDIT_REVISION_END_TS
            </prop>
         </props>
      </property>
   </bean>
```

В листинге 10.48 видно, что слушатель событий аудита Hibernate Envers (org.hibernate.envers.event.AuditEventListener) присоединяется к различным событиям постоянства. Слушатель перехватывает события после вставки, после обновления либо после удаления и клонирует образ сущностного класса перед обновлением в таблице хронологии. Слушатель также присоединен к событиям обновления ассоциации pre-collection-update, pre-collection-remove и pre-collection-recreate для обработки операций обновления ассоциаций сущностного класса. Hibernate Envers имеет возможность отслеживать хронологию сущностей внутри ассоциации (например, “один ко многим”, “многие ко многим” и т.д.).

Для Hibernate Envers также определено несколько свойств, которые кратко описаны в табл. 10.6 (для ясности префикс свойств, org.hibernate.envers, опущен).

Таблица 10.6. Свойства для Hibernate Envers

Имя свойства	Описание
audit_table_suffix	Суффикс имени таблицы для сущности, для которой отслеживаются версии. Например, для сущностного класса ContactAudit, который отображается на таблицу CONTACT_AUDIT, Hibernate Envers будет сохранять хронологию в таблице CONTACT_AUDIT_H, поскольку свойство audit_table_suffix установлено в _H
revision_field_name	Столбец таблицы хронологии для сохранения номера версии для каждой записи хронологии
revision_type_field_name	Столбец таблицы хронологии для сохранения типа действия обновления

Окончание табл. 10.6

Имя свойства	Описание
audit_strategy	Стратегия аудита, используемая для отслеживания версий сущностей
audit_strategy_validity_end_rev_field_name	Столбец таблицы хронологии для сохранения номера конечной версии для каждой записи хронологии. Требуется только в случае применения стратегии аудита достоверности
audit_strategy_validity_store_revend_timestamp	Следует ли сохранять метки времени при обновлении номера конечной версии для каждой записи хронологии. Требуется только в случае применения стратегии аудита достоверности
audit_strategy_validity_revend_timestamp_field_name	Столбец таблицы хронологии для сохранения метки времени, когда обновляется номер конечной версии для каждой записи хронологии. Требуется только в случае применения стратегии аудита достоверности и при условии, что предыдущее свойство установлено в true

Кодирование изменений для отслеживания версий сущностей и извлечения хронологии

Чтобы активизировать отслеживание версий для некоторой сущности, необходимо аннотировать сущностный класс с помощью @Audited. В листинге 10.49 приведен сущностный класс ContactAudit с примененной аннотацией @Audited.

Листинг 10.49. Пересмотренный класс ContactAudit

```
package com.apress.prospring3.ch10.domain;

// Операторы импорта опущены.

import org.hibernate.envers.Audited;
import org.hibernate.envers.NotAudited;

@Entity
@Audited
@Table(name = "contact_audit")
public class ContactAudit implements Auditable<String, Long>, Serializable {

   // Остальной код не показан.
   @ManyToMany
   @NotAudited
   // Остальные аннотации отображения не показаны.
   public Set<Hobby> getHobbies() {
      return this.hobbies;
   }

   @OneToMany(mappedBy = "contact", cascade=CascadeType.ALL,
      orphanRemoval=true)
 @NotAudited
   public Set<ContactTelDetail> getContactTelDetails() {
      return this.contactTelDetails;
   }
}
```

Как показано в листинге 10.49, сущностный класс аннотирован посредством аннотации `@Audited`, которую слушатели Hibernate Envers обнаруживают, после чего будут отслеживать версии обновляемых сущностей. По умолчанию модуль Hibernate Envers также будет пытаться отслеживать хронологию ассоциаций. Таким образом, он попытается сохранить хронологию сведений о телефонах и хобби, связанных с контактом. В случае если отслеживание версий для сущностей ассоциации не требуется, их можно аннотировать с помощью `@NotAudited`.

Для извлечения записей хронологии в Hibernate Envers предоставляется интерфейс `org.hibernate.envers.AuditReader`, который можно получить из класса `AuditReaderFactory`. Давайте добавим в интерфейс `ContactAuditService` новый метод по имени `findAuditByRevision()`, предназначенный для извлечения записи хронологии `ContactAudit` по номеру версии. Необходимый фрагмент кода показан в листинге 10.50.

Листинг 10.50. Метод `findAuditByRevision()`

```
package com.apress.prospring3.ch10.service;

public interface ContactAuditService {
   public ContactAudit findAuditByRevision(Long id, int revision);
   ...
}
```

Один из вариантов извлечения записи хронологии предполагает передачу идентификатора сущности и номера версии. Hibernate Envers также предлагает другие методы для извлечения записей хронологии (за детальной информацией обращайтесь по адресу `docs.jboss.org/hibernate/envers/3.6/reference/en-US/html/queries.html`).

В листинге 10.51 представлена реализация метода `findAuditByRevision()`.

Листинг 10.51. Реализация метода `findAuditByRevision()`

```
package com.apress.prospring3.ch10.service.springjpa;

// Операторы импорта опущены.

import org.hibernate.envers.AuditReader;
import org.hibernate.envers.AuditReaderFactory;

@Service("contactAuditService")
@Repository
@Transactional
public class ContactAuditServiceImpl implements ContactAuditService {

   // Остальной код не показан.
   @PersistenceContext
   private EntityManager entityManager;
   @Transactional(readOnly=true)
   public ContactAudit findAuditByRevision(Long id, int revision) {
      AuditReader auditReader = AuditReaderFactory.get(entityManager);
      return auditReader.find(ContactAudit.class, id, revision);
   }
}
```

Как показано в листинге 10.51, `EntityManager` внедряется в класс и затем передается `AuditReaderFactory` для извлечения экземпляра `AuditReader`. После этого можно вызвать метод `AuditReader.find()`, чтобы извлечь экземпляр сущности `ContactAudit` с заданной версией.

Тестирование отслеживания версий сущностей

Давайте посмотрим, как работает отслеживание версий сущностей. В листинге 10.52 приведен фрагмент кода, предназначенный для тестирования. Код для начальной загрузки `ApplicationContext` и для функции `listContacts()` в листинге 10.52 совпадает с соответствующим кодом из класса `SpringJpaSample`.

Листинг 10.52. Тестирование отслеживания версий сущностей

```
package com.apress.prospring3.ch10;

// Операторы импорта опущены.

public class SpringJpaAuditSample {
   public static void main(String[] args) {
      // Остальной код не показан.
      // Добавить новый контакт.
      System.out.println("Add new contact");
      ContactAudit contact = new ContactAudit();
      contact.setFirstName("Michael");
      contact.setLastName("Jackson");
      contact.setBirthDate(new Date());
      contactService.save(contact);
      contacts = contactService.findAll();
      listContacts(contacts);

      // Обновить контакт.
      System.out.println("Update contact");
      contact.setFirstName("Tom");
      contactService.save(contact);
      contacts = contactService.findAll();
      listContacts(contacts);

      // Найти запись аудита по номеру версии.
      ContactAudit oldContact = contactService.findAuditByRevision(1l, 1);
      System.out.println("");
      System.out.println("Old Contact with id 1 and rev 1:" + oldContact);
      System.out.println("");
      oldContact = contactService.findAuditByRevision(1l, 2);
      System.out.println("");
      System.out.println("Old Contact with id 1 and rev 2:" + oldContact);
      System.out.println("");
   }
}
```

В листинге 10.52 мы сначала создаем новый контакт и затем обновляем его. После этого мы извлекаем сущности `ContactAudit` с версиями 1 и 2 соответственно. Запуск этого кода дает следующий вывод:

```
Listing contacts without details:
Contact - Id: 1, First name: Tom, Last name: Jackson, Birthday: 2011-10-21,
Create by: prospring3, Create date: 2011-10-21T13:54:24.242+08:00, Modified
by: prospring3, Modified date: 2011-10-21T13:54:24.360+08:00

Old Contact with id 1 and rev 1:Contact - Id: 1, First name: Michael, Last
name: Jackson, Birthday: 2011-10-21, Create by: prospring3, Create date:
2011-10-21T13:54:24.242+08:00, Modified by: prospring3, Modified date:
2011-10-21T13:54:24.242+08:00
```

```
Old Contact with id 1 and rev 2:Contact - Id: 1, First name: Tom, Last
name: Jackson, Birthday: 2011-10-21, Create by: prospring3, Create date:
2011-10-21T13:54:24.242+08:00, Modified by: prospring3, Modified date:
2011-10-21T13:54:24.360+08:00
```

В показанном выводе видно, что после операции обновления имя `ContactAudit` было изменено на `Tom`. Однако при просмотре хронологии для версии 1 именем является `Michael`. В версии 2 именем становится `Tom`. Также обратите внимание, что дата последней модификации для версии 2 корректно отражает обновленное значение даты-времени.

Соображения по поводу того, когда использовать JPA

Несмотря на относительно большой объем этой главы, в ней была рассмотрена только малая часть JPA. Например, не обсуждалось применение JPA для вызова хранимых процедур базы данных. JPA представляет собой завершенный и мощный стандарт доступа к данным ORM, и благодаря набору библиотек третьих сторон, таких как Spring Data JPA и Hibernate Envers, можно относительно просто реализовать разнообразную сквозную функциональность.

JPA — это стандарт JEE, который поддерживается большинством крупных сообществ открытого кода, а также коммерческими поставщиками (например, JBoss, GlassFish, WebSphere, WebLogic и т.д.). Таким образом, использование JPA в качестве стандарта доступа к данным является вполне обоснованным выбором. Если требуется абсолютный контроль над запросом, вместо работы с JDBC напрямую можно применять средство собственных запросов JPA.

В заключение отметим, что для разработки JEE-приложений с помощью Spring мы рекомендуем использовать JPA для реализации уровня доступа к данным. При желании для реализации некоторых специальных потребностей доступа к данным JPA можно смешивать с JDBC. Всегда помните, что Spring позволяет легко сочетать различные технологии доступа к данным с прозрачным управлением транзакциями.

Использование JPA в примере приложения

В этом разделе мы обсудим отношения между темами, рассмотренными в главе, и примером приложения, которое мы будем разрабатывать. Темы включают базу данных серверной части и реализацию JPA для различных операций базы данных.

Мы также покажем, как использовать средства Spring Data JPA для упрощения логики доступа к данным и отслеживания базовой информации аудита.

Наконец, мы посмотрим, как реализовать в примере приложения отслеживание версий сущностей с применением модуля Hibernate Envers.

База данных серверной части

В качестве базы данных серверной части будет использоваться встроенная база данных JDBC типа H2. Тем не менее, сценарии (для создания база данных и для заполнения ее начальными данными) будут разрабатываться так, чтобы быть совместимыми также и с MySQL. Это позволит при желании запускать пример приложения с базой данных MySQL.

В конфигурации `ApplicationContext` мы просто применим дескриптор `<jdbc:embedded-database>` для объявления встроенной базы данных. Будут написаны два SQL-сценария. Один из них (по имени `schema.sql`) предназначен для создания

базы данных, а другой (по имени `initial-data.sql`) — для наполнения начальными данными (начальными пользователями, категориями и подкатегориями, записями блога и т.д.).

Использование JPA для реализации уровня постоянства

Как упоминалось в главе 3, для уровня постоянства будут предоставлены две разных реализации: одна будет использовать JPA, а другая — MyBatis.

В реализации JPA мы будем применять JPA 2 с Hibernate в качестве поставщика постоянства. Все сущностные классы будут помещены в пакет `com.apress.prospring3.springblog.domain`, а свойства объектно-реляционного отображения будут определены с помощью стандартных аннотаций JPA 2 (из пакета `javax.persistence`).

Вдобавок мы будем использовать Spring Data JPA для упрощения реализации разнообразных операций базы данных. Все интерфейсы репозитория (например, интерфейс `EntryRepository` для поддержки сущностного класса `Entry`) будут помещены в пакет `com.apress.prospring3.springblog.repository`. Более того, для поддержки просмотра записей блога в рамках пользовательского интерфейса веб-приложения будет реализовано средство разбиения на страницы.

Что касается операций базы данных, то наиболее сложной частью является выполнение запросов. В пользовательском интерфейсе можно фильтровать записи по датам создания, категориям и подкатегориям и т.п. Для этого будут применяться запросы, использующие строго типизированный API-интерфейс критериев JPA 2.

Аудит и отслеживание версий сущностей

В примере приложения для записей блога и комментариев к ним будут включены средства аудита.

С помощью функциональных возможностей аудита Spring Data JPA мы будем отслеживать базовую информацию аудита (пользователь, который создал; дата создания; пользователь, который внес последнюю модификацию; дата последней модификации) для записей блога и комментариев.

Когда запись блога или комментарий обновляется, модель Hibernate Envers отслеживает версии каждой записи. Для хранения записей хронологии создаются таблицы хронологии, связанные с таблицами записей блога и комментариев.

Дополнительные детали можно найти в главе 21.

Резюме

В этой главе мы раскрыли базовые концепции JPA и показали, как конфигурировать `EntityManagerFactory` из JPA в Spring, используя Hibernate в качестве поставщика службы постоянства.

Затем мы обсудили применение JPA для выполнения элементарных операций базы данных. В качестве дополнительных тем рассматривались собственные запросы и строго типизированный API-интерфейс критериев JPA 2.

Кроме того, мы показали, каким образом абстракция `Repository` в Spring Data JPA может упростить разработку приложений JPA, и как использовать слушатели сущностей Spring Data JPA для отслеживания базовой информации аудита для сущностных классов. Также рассматривалось применение модуля Hibernate Envers для полного отслеживания версий сущностных классов.

В следующей главе мы обсудим еще одну популярную библиотеку доступа к данным: MyBatis (ранее известную под названием iBATIS).

ГЛАВА 11

Использование MyBatis в Spring

В предшествующих трех главах было показано, что Spring поддерживает бесшовную интеграцию с различными библиотеками и технологиями для реализации логики доступа к данным, от традиционного подхода JDBC до решений ORM, включая Hibernate и JEE-стандарт JPA. В этой главе мы рассмотрим еще одну популярную библиотеку доступа к данным — MyBatis (`www.mybatis.org`).

Ранее известная под названием iBATIS (которая поддерживалась Apache Software Foundation и затем была изъята из обращения), MyBatis является Java-библиотекой (доступна также ее версия для .NET), которая предоставляет инфраструктуру для отображения реляционной структуры базы данных на объектно-ориентированную модель Java. Однако вместо сосредоточения на программировании для упомянутой объектно-ориентированной модели, как это сделано в ORM, библиотека MyBatis больше направлена в сторону SQL. О MyBatis можно думать как о смешанном подходе между JDBC и ORM. Команда разработчиков MyBatis позиционирует MyBatis как основанное на SQL решение по отображению данных для разработки объектно-ориентированного программного обеспечения.

В этой главе мы собираемся сконцентрировать внимание на реализации уровня доступа к данным Spring-приложения с использованием MyBatis. В частности, мы рассмотрим следующие темы.

- *Конфигурация.* Мы покажем, как конфигурировать MyBatis с помощью Spring, включая обязательные зависимости Maven и конфигурацию экземпляра `SqlSessionFactory` из MyBatis в `ApplicationContext` в Spring.
- *SQL-отображение MyBatis.* Мы рассмотрим, каким образом определять SQL-отображения с помощью MyBatis для трансформирования результирующего набора SQL-запросов в свойства соответствующих Java-объектов предметной области.
- *Операции CRUD.* Вы узнаете, как реализовать операции выборки, обновления и удаления, используя MyBatis. Вы также научитесь строить операции выборки, которые представляют основные типы отношений между данными.
- *Использование MyBatis в примере приложения.* В примере приложения для реализации MyBatis мы устраним уровень DAO и внедрим средства отображения MyBatis напрямую в уровень обслуживания с целью выполнения бизнес-логики. Мы также обсудим применение MyBatis в примере приложения.

Начало работы с MyBatis в Spring

В этом разделе мы объясним, как настроить MyBatis для работы с платформой Spring. Сначала мы предложим краткое введение в MyBatis, после чего покажем, каким образом создать проект в STS для работы с MyBatis. Кроме того, мы представим модель данных, которая будет использоваться для реализации примеров в этой главе. Наконец, мы обсудим конфигурацию Spring, требуемую для работы с MyBatis.

Введение в MyBatis

Библиотека iBATIS — старое название MyBatis — была начата Клинтоном Бегином (Clinton Begin) в 2001 г., а в 2004 г. передана Apache Software Foundation (ASF) как код проекта. Основанная на SQL-отображении, простой инфраструктуре и несложных для понимания определений отображения между SQL-запросами и объектно-ориентированной моделью Java, библиотека iBATIS быстро обрела популярность и стала одной из наиболее часто используемых библиотек доступа к данным в сообществе разработчиков на языке Java.

После шестилетнего нахождения в среде ASF разработчики iBATIS поняли, что технологии доступа к данным в мире открытого кода радикально изменились. В результате разработчики приняли решение внести ряд значимых изменений в свою библиотеку. Начиная с версии 3, проект также поменял свое название на MyBatis, в нем отказались от ASF, и он превратился в независимую инфраструктуру с открытым кодом. Проект iBATIS на ASF прекратил свое существование. На время написания этой книги текущей версией MyBatis была 3.0.6.

До выхода версии Spring 3.0 в Spring была встроенная поддержка iBATIS версии 2. Однако из-за трудностей в синхронизации крупных изменений между iBATIS 2 и MyBatis 3 с платформой Spring разработчики решили отказаться от встроенной поддержки MyBatis.

Чтобы справиться с возникшей ситуацией, команда MyBatis начала проект интеграции MyBatis и Spring (MyBatis Spring Integration Project) под названием `mybatis-spring`. На время написания этой книги текущей версией `mybatis-spring` была 1.0.2.

Создание простого служебного проекта с поддержкой MyBatis в STS

В STS отсутствует шаблонный проект, обеспечивающий использование Spring вместе с MyBatis, поэтому мы должны создать простой служебный проект Spring и затем вручную добавить обязательные зависимости. Для начала создайте простой служебный проект Spring в STS (соответствующие детали приведены в приложении А). Удостоверьтесь, что в проекте используются JSE 6 и Spring 3.1.

После создания проекта откройте файл `pom.xml` в редакторе POM и добавьте зависимости, описанные в табл. 11.1.

На рис. 11.1 можно видеть вновь добавленные зависимости Maven в STS.

Чтобы заставить MyBatis отображать SQL-запросы, запускаемые в базе данных, понадобится включить уровень `DEBUG` ведения журнала. Давайте поместим файл свойств `log4j` в папку `src/main/resources` с корневым уровнем ведения журнала, установленным в `DEBUG`. В листинге 11.1 показано содержимое файла `log4j.properties`.

На заметку! В среде STS после создания шаблонного проекта Spring файл `log4j.properties` сгенерируется в папке `src/test/resources`. Вы можете просто переместить его в папку `src/main/resources` и соответствующим образом модифицировать или же удалить этот файл в `src/test/resources` и затем создать новый файл `log4j.properties` в папке `/src/main/resources`.

Таблица 11.1. Зависимости Maven для Spring с MyBatis

Идентификатор группы	Идентификатор артефакта	Версия	Описание
org.springframework	spring-jdbc	3.1.0.RELEASE	Библиотека Spring для доступа JDBC, которая требуется для всех приложений, содержащих логику доступа к данным
org.springframework	spring-tx	3.1.0.RELEASE	Библиотека Spring для поддержки транзакций
com.h2database	h2	1.3.160	Библиотека для поддержки встроенной базы данных H2
org.mybatis	mybatis	3.0.6	Основная библиотека MyBatis
org.mybatis	mybatis-spring	1.0.2	Библиотека интеграции MyBatis и Spring

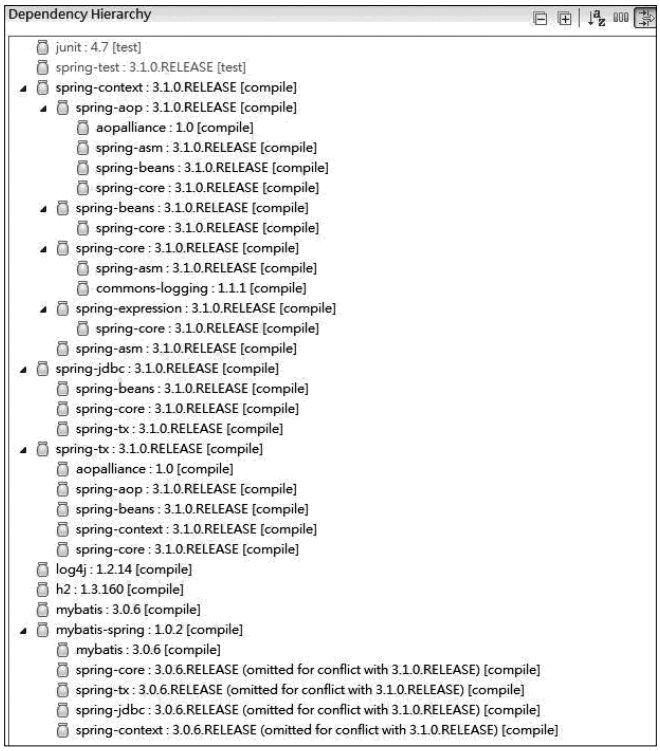


Рис. 11.1. Зависимости Maven для Spring с MyBatis

Листинг 11.1. Файл `log4j.properties`

```
log4j.rootCategory=DEBUG, stdout
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE} %5p %40.40c:%4L - %m%n
```

Модель данных для кода примеров

В примерах этой главы будет использоваться та же самая модель данных, что и в главе 9, когда обсуждалось применение Hibernate в Spring, но с небольшими изменениями. Эта модель данных изображена на рис. 11.2.

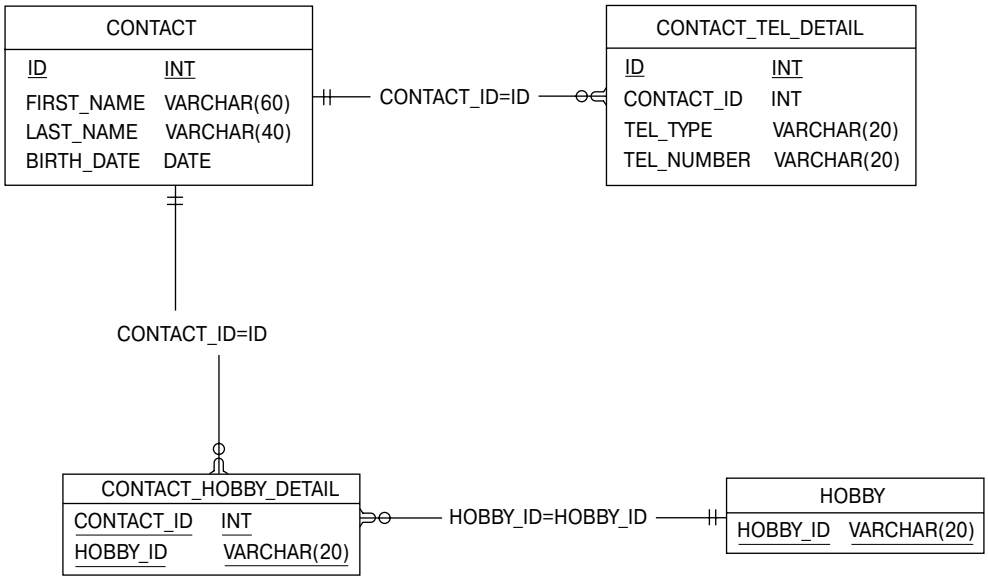


Рис. 11.2. Модель данных для MyBatis

Главное отличие между этой моделью и моделью из главы 9 заключается в том, что столбец `VERSION` в таблицах `CONTACT` и `CONTACT_TEL_DETAIL` удален. Основная причина этого связана с тем, что MyBatis не предоставляет поддержку оптимистической блокировки для столбца `VERSION`, как это делает Hibernate. Тем не менее, в примере приложения будет показано, как реализовать оптимистическую блокировку, используя MyBatis.

В листингах 11.2 и 11.3 приведены сценарии, предназначенные для создания схемы (`schema.sql`) и наполнения начальными данными (`test-data.sql`).

Листинг 11.2. Сценарий для создания модели данных

```
CREATE TABLE CONTACT (
     ID INT NOT NULL AUTO_INCREMENT
   , FIRST_NAME VARCHAR(60) NOT NULL
   , LAST_NAME VARCHAR(40) NOT NULL
   , BIRTH_DATE DATE
   , UNIQUE UQ_CONTACT_1 (FIRST_NAME, LAST_NAME)
   , PRIMARY KEY (ID)
);

CREATE TABLE HOBBY (
     HOBBY_ID VARCHAR(20) NOT NULL
   , PRIMARY KEY (HOBBY_ID)
);

CREATE TABLE CONTACT_TEL_DETAIL (
```

```
     ID INT NOT NULL AUTO_INCREMENT
   , CONTACT_ID INT NOT NULL
   , TEL_TYPE VARCHAR(20) NOT NULL
   , TEL_NUMBER VARCHAR(20) NOT NULL
   , UNIQUE UQ_CONTACT_TEL_DETAIL_1 (CONTACT_ID, TEL_TYPE)
   , PRIMARY KEY (ID)
   , CONSTRAINT FK_CONTACT_TEL_DETAIL_1 FOREIGN KEY (CONTACT_ID)
     REFERENCES CONTACT (ID)
);

CREATE TABLE CONTACT_HOBBY_DETAIL (
     CONTACT_ID INT NOT NULL
   , HOBBY_ID VARCHAR(20) NOT NULL
   , PRIMARY KEY (CONTACT_ID, HOBBY_ID)
   , CONSTRAINT FK_CONTACT_HOBBY_DETAIL_1 FOREIGN KEY (CONTACT_ID)
     REFERENCES CONTACT (ID) ON DELETE CASCADE
   , CONSTRAINT FK_CONTACT_HOBBY_DETAIL_2 FOREIGN KEY (HOBBY_ID)
     REFERENCES HOBBY (HOBBY_ID)
);
```

Листинг 11.3. Сценарий для наполнения начальными данными

```
insert into contact (first_name, last_name, birth_date)
   values ('Clarence', 'Ho', '1980-07-30');
insert into contact (first_name, last_name, birth_date)
   values ('Scott', 'Tiger', '1990-11-02');
insert into contact (first_name, last_name, birth_date)
   values ('John', 'Smith', '1964-02-28');

insert into contact_tel_detail (contact_id, tel_type, tel_number)
   values (1, 'Mobile', '1234567890');
insert into contact_tel_detail (contact_id, tel_type, tel_number)
   values (1, 'Home', '1234567890');
insert into contact_tel_detail (contact_id, tel_type, tel_number)
   values (2, 'Home', '1234567890');

insert into hobby (hobby_id) values ('Swimming');
insert into hobby (hobby_id) values ('Jogging');
insert into hobby (hobby_id) values ('Programming');
insert into hobby (hobby_id) values ('Movies');
insert into hobby (hobby_id) values ('Reading');

insert into contact_hobby_detail(contact_id, hobby_id) values (1, 'Swimming');
insert into contact_hobby_detail(contact_id, hobby_id) values (1, 'Movies');
insert into contact_hobby_detail(contact_id, hobby_id) values (2, 'Swimming');
```

Конфигурирование `SqlSessionFactory` и `MapperScannerConfigurer` в MyBatis

Ключевая концепция MyBatis вращается вокруг интерфейса `SqlSession` (из пакета `org.apache.ibatis.session`), который получен из интерфейса `SqlSessionFactory` (из пакета `org.apache.ibatis.session`).

Оба интерфейса, `SqlSession` и `SqlSessionFactory`, принадлежат библиотеке `mybatis-3.0.6`. Для конфигурирования фабрики в Spring мы используем класс `SqlSessionFactoryBean` (из пакета `org.mybatis.spring`), который относится к модулю `mybatis-spring` (библиотека `mybatis-spring-1.0.2`).

Другой важной концепцией в MyBatis являются интерфейсы отображателей (mapper), которые представляют собой просто Java-интерфейсы, которые будут обрабатываться MyBatis для построения конфигурации отображения между SQL-запросами и свойствами объектов предметной области. Отображение может быть определено либо в XML-файле (имеющем такое же имя, как у интерфейса), либо с помощью аннотаций внутри интерфейса отображателя. Модуль `mybatis-spring` предоставляет еще один класс по имени `MapperScannerConfigurer` (в пакете `org.mybatis.spring.mapper`), который поддерживает удобный способ уведомления MyBatis о необходимости поиска интерфейсов отображателей и регистрации их как `MapperFactoryBean<T>` (из пакета `org.mybatis.spring.mapper`, принадлежащего модулю `mybatis-spring`); это делает возможным внедрение интерфейсов отображателей MyBatis в другие бины Spring.

Интерфейсы отображателей будут детально рассматриваться далее в главе, и вы увидите, что они могут упростить разработку операций базы данных без необходимости в прямом взаимодействии с интерфейсом `SqlSession`.

В листинге 11.4 приведена конфигурация контекста приложения Spring (`app-context.xml`).

Листинг 11.4. Конфигурация Spring для MyBatis

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:context="http://www.springframework.org/schema/context"
   xmlns:jdbc="http://www.springframework.org/schema/jdbc"
   xmlns:tx="http://www.springframework.org/schema/tx"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context-3.1.xsd
      http://www.springframework.org/schema/jdbc
      http://www.springframework.org/schema/jdbc/spring-jdbc-3.1.xsd
      http://www.springframework.org/schema/tx
      http://www.springframework.org/schema/tx/spring-tx-3.1.xsd">

   <jdbc:embedded-database id="dataSource" type="H2">
      <jdbc:script location="classpath:schema.sql" />
      <jdbc:script location="classpath:test-data.sql" />
   </jdbc:embedded-database>

   <bean id="transactionManager"
 class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
      <property name="dataSource" ref="dataSource" />
   </bean>

   <tx:annotation-driven />

   <context:component-scan
      base-package="com.apress.prospring3.ch11.service.mybatis" />

   <!-- Определить SqlSessionFactory -->
   <bean id="sqlSessionFactory"
      class="org.mybatis.spring.SqlSessionFactoryBean">
      <property name="dataSource" ref="dataSource" />
      <property name="typeAliasesPackage"
         value="com.apress.prospring3.ch11.domain" />
   </bean>
```

```
    <!-- Сканировать отображатели и разрешить для них автосвязывание -->
    <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
       <property name="basePackage"
          value="com.apress.prospring3.ch11.persistence" />
    </bean>
</beans>
```

Большинство бинов должны быть хорошо знакомы; мы же сосредоточим внимание на бинах `sqlSessionFactory` и `org.mybatis.spring.mapper.MapperScannerConfigurer`.

Бин `sqlSessionFactory` реализован с помощью класса `org.mybatis.spring.SqlSessionFactoryBean`, предоставленного `mybatis-spring`. Как и можно было ожидать, фабрика требует бина источника данных.

Кроме того, свойство `typeAliasesPackage` определяет пакеты, хранящие объекты предметной области, которые библиотека MyBatis должна просматривать при выполнении отображения результирующего набора SQL на объекты POJO. Аналогично работают отображенные сущностные классы в Hibernate и JPA.

Обратите внимание, что в конфигурации один и тот же экземпляр бина `dataSource` внедряется в Spring-бин `transactionManager` и в MyBatis-бин `sqlSessionFactory`. Причина в том, что класс `SqlSessionFactoryBean` спроектирован специально для использования существующего класса `DataSourceTransactionManager` в Spring. После конфигурирования диспетчера транзакций Spring можно настраивать транзакции обычным образом. Поддерживаются как аннотации `@Transactional`, так и конфигурация в стиле АОП. После настройки модуль `mybatis-spring` будет прозрачно управлять транзакциями. Необходимо лишь удостовериться, что в бины `transactionManager` и `SqlSessionFactory` внедрен один и тот же экземпляр `dataSource`.

Модуль `mybatis-spring` также предоставляет класс `org.mybatis.spring.mapper.MapperScannerConfigurer`. Этот класс принимает имя пакета, в котором MyBatis будет искать интерфейсы отображателей. Кроме того, бин `MapperScannerConfigurer` также требует внедрения экземпляра `SqlSessionFactory`.

Если в `ApplicationContext` имеется только один бин `SqlSessionFactory`, он будет автосвязываться с бином `MapperScannerConfigurer`. Однако если приложение содержит множество бинов `SqlSessionFactory` (например, в приложении с несколькими источниками данных может быть определен бин `sqlSessionFactoryA` для источника данных A и бин `sqlSessionFactoryB` для источника данных B), понадобится явно передавать ссылку на соответствующий бин `SqlSessionFactory` в каждый бин `MapperScannerConfigurer`.

Подготовив указанную выше конфигурацию, можно переходить к следующему шагу, который заключается в определении SQL-отображения MyBatis между запросами и моделью POJO.

SQL-отображение в MyBatis

При реализации логики доступа к данным с использованием MyBatis наиболее важным и отнимающим львиную долю времени шагом является определение отображений между запросами для разнообразных операций базы данных и соответствующими объектами предметной области. Отображение тесно связано с тем, какие операции требуются, и лежащими в их основе запросами. Итак, для службы информации о контакте определим интерфейс `ContactService`, который будет реализован первым. Этот интерфейс представлен в листинге 11.5.

Листинг 11.5. Интерфейс ContactService

```
package com.apress.prospring3.ch11.service;

import java.util.List;
import com.apress.prospring3.ch11.domain.Contact;

public interface ContactService {
    // Найти все контакты — без деталей.
    public List<Contact> findAll();

    // Найти все контакты — с телефонами и хобби.
    public List<Contact> findAllWithDetail();

    // Найти контакт по идентификатору - с телефонами и хобби.
    public Contact findById(Long id);

    // Создать новый или сохранить существующий контакт.
    public Contact save(Contact contact);

    // Удалить контакт.
    public void delete(Contact contact);

    // Найти контакт по имени и фамилии.
    public List<Contact> findByFirstNameAndLastName(String firstName,
        String lastName);
}
```

Интерфейс ContactService должен выглядеть вполне понятным. Он содержит несколько методов поиска для извлечения общих или детальных сведений о контакте, часть из которых принимают параметры для критериев поиска, а также методы для операций вставки, обновления и удаления. Позже мы реализуем эти методы в классе ContactServiceImpl.

Интерфейсы отображателей и файлы SQL-отображения

Интерфейсы отображателей и файлы отображения лежат в основе функционирования MyBatis. Они работают вместе с объектами предметной области для отображения результатов запроса на объекты предметной области и наоборот. При отображении каждого объекта предметной области используются три главных файла. В табл. 11.2 приведено описание этих файлов вместе с примерами.

Таблица 11.2. Интерфейс отображателя и файл отображения MyBatis

Тип файла	Назначение	Пример
Объект предметной области	Объект POJO, на который отображаются запросы	Класс Contact (в пакете com.apress.prospring3.ch11.domain), который хранит сведения о контакте
Интерфейс отображателя	Java-интерфейсы, которые MyBatis будет искать и регистрировать как бины MapperFactoryBean. Здесь будут определены все операции базы данных, поддерживаемые для связанного объекта предметной области	Интерфейс ContactMapper (в пакете com.apress.prospring3.ch11.persistence) для объекта предметной области Contact
Файл конфигурации SQL-отображения	XML-файл конфигурации, хранящий детали отображения между SQL-запросами и объектами предметной области. Этот файл должен находиться в пути классов проекта и иметь такое же имя, как у интерфейса отображателя	Файл ContactMapper.xml (в папке src/main/resources с именем пакета com.apress.prospring3.ch11.persistence)

Лучше понять эту структуру поможет диаграмма. Например, чтобы реализовать метод `findAll()` интерфейса `ContactService`, файлы и их содержимое будут выглядеть так, как показано на рис. 11.3.

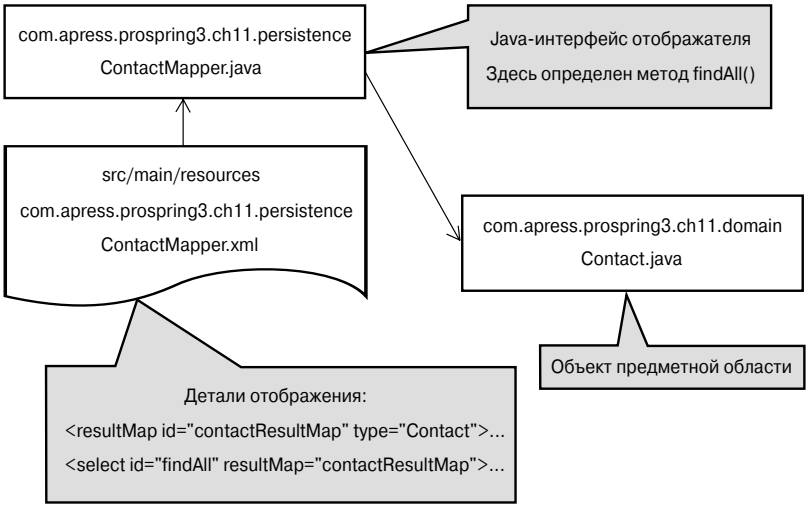


Рис. 11.3. Структура файлов отображения MyBatis

На рис. 11.3 видно, что интерфейс отображателя имеет соответствующий XML-файл с таким же именем (`ContactMapper.java` и `ContactMapper.xml`), поддерживающий операции для объекта предметной области `Contact.java`.

По завершении построения всех отображений структура проекта должна выглядеть примерно так, как показано на рис. 11.4.

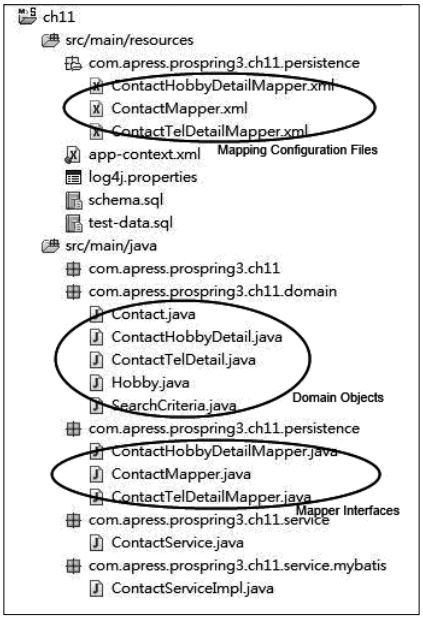


Рис. 11.4. Структура файлов отображения MyBatis в STS

XML-конфигурация SQL-отображения

В XML-файлах конфигурации отображателей MyBatis предоставляет множество дескрипторов для выполнения операций отображения. Наиболее часто используемые дескрипторы описаны в табл. 11.3.

Таблица 11.3. Основные дескрипторы XML-файла отображения MyBatis

Дескриптор	Описание
`resultMap`	Отображает результирующий набор операций выборки на объекты предметной области. Кроме того, предусмотрено множество поддескрипторов (например, `<id>`, `<result>`, `<association>`, `<collection>` и т.д.), предназначенных для определения типов данных, ассоциаций отображения и т.п.
`select`	Определяет операцию выборки. Здесь определяются запрос для отправки, именованные параметры и отображение результатов
`insert`	Определяет операцию вставки. Свойства объекта предметной области будут отображаться на именованные параметры оператора `insert`. Сгенерированный базой данных ключ может быть извлечен MyBatis после выполнения оператора вставки
`update`	Определяет операцию обновления
`delete`	Определяет операцию удаления

Операции базы данных в MyBatis

В этом разделе мы посмотрим, как выполнять разнообразные операции CRUD с помощью MyBatis в Spring.

Для запрашивания данных мы обсудим, каким образом осуществляется отображение SQL на POJO, а также как моделируются отношения между объектами предметной области. Вдобавок будут раскрыты темы именованных параметров и поддержки динамического SQL в MyBatis.

Наконец, мы также покажем, как применять MyBatis для реализации операций вставки, обновления и удаления.

Запрашивание данных

Библиотека MyBatis предлагает интенсивную поддержку запросов данных из базы. В сущности, для каждой операции выборки мы будем определять SQL-оператор, параметры, результирующий тип (Java-тип) и необходимое отображение результатов. Можно легко отображать различные отношения (“один к одному”, “один ко многим” и “многие ко многим”). Еще одним мощным средством в MyBatis является поддержка динамического SQL, которая будет обсуждаться в последующих разделах.

Простые выборки

Давайте начнем с простейшего метода `ContactService` — `findAll()`. Этот метод просто извлекает все контакты из базы данных без деталей о телефонах и хобби.

Первым делом мы реализуем объект предметной области `Contact`, не имеющий ассоциаций. Код класса `Contact` показан в листинге 11.6.

Листинг 11.6. Объект предметной области `Contact`

```
package com.apress.prospring3.ch11.domain;

import java.io.Serializable;
import java.util.Date;

public class Contact implements Serializable {

   private Long id;
   private String firstName;
   private String lastName;
   private Date birthDate;

   // Методы извлечения/установки не показаны.

   public String toString() {
      return "Contact - Id: " + id + ", First name: " + firstName
         + ", Last name: " + lastName + ", Birthday: " + birthDate;
   }
}
```

Класс — это простой POJO, не требующий дополнительных пояснений. Следующий шаг заключается в определении интерфейса отображателя для контакта (и метода `findAll()`), как показано в листинге 11.7.

Листинг 11.7. Интерфейс `ContactMapper`

```
package com.apress.prospring3.ch11.persistence;

import java.util.List;

import com.apress.prospring3.ch11.domain.Contact;

public interface ContactMapper {

   public List<Contact> findAll();

}
```

В этом интерфейсе также нет ничего особенного. Нужно просто определить операции базы данных, поддерживаемые отображателем, которые будут найдены классом `MapperScannerConfigurer` и могут быть автосвязаны с бинами Spring.

Следующий шаг является наиболее важным — создание XML-файла отображения. В листинге 11.8 показано его содержимое для отображения операции `findAll` (`ContactMapper.xml`).

Листинг 11.8. Отображение для операции `findAll`

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper namespace="com.apress.prospring3.ch11.persistence.ContactMapper">

   <resultMap id="contactResultMap" type="Contact">
      <id property="id" column="ID" />
      <result property="firstName" column="FIRST_NAME" />
      <result property="lastName" column="LAST_NAME" />
      <result property="birthDate" column="BIRTH_DATE" />
   </resultMap>
```

```
    <select id="findAll" resultMap="contactResultMap">
        SELECT
        ID,
        FIRST_NAME,
        LAST_NAME,
        BIRTH_DATE
        FROM CONTACT
    </select>
</mapper>
```

Ниже перечислены основные моменты, относящиеся к конфигурации отображения.

- Файл начинается с дескриптора `<mapper>` с установленным атрибутом пространства имен (`namespace`). Атрибут `namespace` (`com.apress.prospring3.ch11.persistence.ContactMapper`) указывает имя файла и интерфейса (т.е. `ContactMapper.xml` и `ContactMapper.java`), которые должны совпадать (в этом случае `com.apress.prospring3.ch11.persistence.ContactMapper`).
- Дескриптор `<resultMap>` определяет уникальное отображение результата операции выборки. В зависимости от потребностей приложения, можно определять множество отображений результатов. Как показано в листинге 11.8, мы указываем, что отображаться должен объект предметной области `Contact`. Здесь нет необходимости предоставлять имя пакета для объекта предметной области, потому что оно уже было предоставлено в свойстве `typeAliasesPackage` бина `sqlSessionFactory` внутри `ApplicationContext` в Spring (см. файл `app-context.xml` в листинге 11.4). Затем с помощью дескриптора `<id>` определяется свойство для поля `id` и соответствующее имя столбца в запросе. Далее применяется множество дескрипторов `<result>` для отображения свойства объекта предметной области на выбранные столбцы.
- Дескриптор `<select>` определяет операцию запроса. Обратите внимание, что атрибут `id` должен соответствовать имени метода, определенного в интерфейсе `ContactMapper`. Затем с помощью атрибута `resultMap` указывается отображение, используемое для трансформации результирующего набора в объект предметной области `Contact`. В теле этого дескриптора задается применяемый SQL-оператор.

После определения отображения можно приступать к реализации. Класс реализации, `com.apress.prospring3.ch11.service.mybatis.ContactServiceImpl`, начинается с создания класса с пустыми реализациями всех методов интерфейса `ContactService`. В STS это делается очень просто. За подробной информацией обращайтесь в главу 8.

В листинге 11.9 приведена реализация метода `findAll()` интерфейса `ContactService`.

Листинг 11.9. Реализация метода `findAll()`

```
package com.apress.prospring3.ch11.service.mybatis;

// Операторы импорта опущены.

@Service("contactService")
@Repository
@Transactional
public class ContactServiceImpl implements ContactService {

    private Log log = LogFactory.getLog(ContactServiceImpl.class);

    @Autowired
```

```
    private ContactMapper contactMapper;

    @Transactional(readOnly=true)
    public List<Contact> findAll() {
        List<Contact> contacts = contactMapper.findAll();
        return contacts;
    }

    // Остальной код не показан.
}
```

В листинге 11.9 видно, что интерфейс отображателя внедряется в класс службы, а метод `findAll()` просто вызывает соответствующий метод интерфейса `ContactMapper`. Всю остальную работу выполнит MyBatis.

На этом реализация метода `findAll()` завершена. В листинге 11.10 представлена тестовая программа.

Листинг 11.10. Тестирование метода `findAll()`

```
package com.apress.prospring3.ch11;

import java.util.List;

import org.springframework.context.support.GenericXmlApplicationContext;

import com.apress.prospring3.ch11.domain.Contact;
import com.apress.prospring3.ch11.service.ContactService;

public class MyBatisSample {

    public static void main(String[] args) {

        GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
        ctx.load("classpath:app-context.xml");
        ctx.refresh();

        ContactService contactService = ctx.getBean("contactService",
            ContactService.class);

        List<Contact> contacts;

        // Найти все контакты.
        contacts = contactService.findAll();
        listContacts(contacts);

    }

    private static void listContacts(List<Contact> contacts) {
        System.out.println("");
        System.out.println("Listing contacts without details:");
        for (Contact contact: contacts) {
            System.out.println(contact);
            System.out.println();
        }
    }

}
```

Запуск этой программы дает в результате вывод, показанный на рис. 11.5.

Как видно на рис. 11.5, был активизирован уровень ведения журнала `DEBUG`, поэтому кроме сведений о контакте также отображается оператор выборки, отправленный библиотекой MyBatis базе данных. Это очень полезное средство отладки при разработке.

```
11:22:22,733 DEBUG       org.mybatis.spring.SqlSessionUtils:  27 - Registering transaction synchronization for SqlSession [org.apache.ibatis.
11:22:22,784 DEBUG               java.sql.PreparedStatement:  27 - ==> Executing: SELECT ID, FIRST_NAME, LAST_NAME, BIRTH_DATE FROM CONTACT
11:22:22,785 DEBUG               java.sql.PreparedStatement:  27 - ==> Parameters:
11:22:22,802 DEBUG                       java.sql.ResultSet:  27 - <==    Columns: ID, FIRST_NAME, LAST_NAME, BIRTH_DATE
11:22:22,802 DEBUG                       java.sql.ResultSet:  27 - <==        Row: 1, Clarence, Ho, 1980-07-30
11:22:22,803 DEBUG                       java.sql.ResultSet:  27 - <==        Row: 2, Scott, Tiger, 1990-11-02
11:22:22,804 DEBUG                       java.sql.ResultSet:  27 - <==        Row: 3, John, Smith, 1964-02-28
11:22:22,807 DEBUG       org.mybatis.spring.SqlSessionUtils:  27 - Releasing transactional SqlSession [org.apache.ibatis.session.defaults.Def
11:22:22,807 DEBUG       org.mybatis.spring.SqlSessionUtils:  27 - Transaction synchronization committing SqlSession [org.apache.ibatis.sessi
11:22:22,807 DEBUG .datasource.DataSourceTransactionManager: 752 - Initiating transaction commit
11:22:22,807 DEBUG .datasource.DataSourceTransactionManager: 264 - Committing JDBC transaction on Connection [conn2: url=jdbc:h2:mem:dataSour
11:22:22,808 DEBUG       org.mybatis.spring.SqlSessionUtils:  27 - Transaction synchronization closing SqlSession [org.apache.ibatis.session.
11:22:22,808 DEBUG .datasource.DataSourceTransactionManager: 322 - Releasing JDBC Connection [conn2: url=jdbc:h2:mem:dataSource user=SA] afte
11:22:22,809 DEBUG ramework.jdbc.datasource.DataSourceUtils: 332 - Returning JDBC Connection to DataSource

Listing contacts without details:
Contact - Id: 1, First name: Clarence, Last name: Ho, Birthday: Wed Jul 30 00:00:00 CST 1980

Contact - Id: 2, First name: Scott, Last name: Tiger, Birthday: Fri Nov 02 00:00:00 CST 1990

Contact - Id: 3, First name: John, Last name: Smith, Birthday: Fri Feb 28 00:00:00 CST 1964
```

Рис. 11.5. Вывод программы для тестирования метода `findAll()` в STS

Определение отображения с использованием аннотаций MyBatis

В дополнение к XML-конфигурации MyBatis, как и многие другие инструменты, также поддерживает определения отображений с помощью Java-аннотаций.

Аннотации должны применяться к интерфейсу отображателя. В листинге 11.11 показан интерфейс `ContactMapper` с аннотациями, используемыми для определения отображений.

Листинг 11.11. Отображение с помощью аннотаций MyBatis

```
package com.apress.prospring3.ch11.persistence;

import java.util.List;

import org.apache.ibatis.annotations.*;

import com.apress.prospring3.ch11.domain.Contact;

public interface ContactMapperAnnotation {

    @Select(value="SELECT ID,FIRST_NAME,LAST_NAME,BIRTH_DATE FROM CONTACT")
    @Results(value={
        @Result(javaType=Contact.class),
        @Result(property="id", column="ID"),
        @Result(property="firstName", column="FIRST_NAME"),
        @Result(property="lastName", column="LAST_NAME"),
        @Result(property="birthDate", column="BIRTH_DATE")
    })
    List<Contact> findAll();
}
```

В листинге 11.11 видно, что метод `findAll()` аннотирован посредством аннотации `@Select`, которая указывает запрос, аннотации `@Results` и вложенных аннотаций `@Result`, определяющих детали отображений. Структура аннотаций довольно похожа на XML-конфигурацию отображения в листинге 11.8. (В случае XML-конфигурации имя метода `findAll` указывается в атрибуте `id` дескриптора `<select>`.)

По сути, MyBatis предлагает эквиваленты в виде аннотаций для всех элементов XML-конфигурации. Таким образом, можно избрать подход с аннотациями и полностью избавиться от XML-файлов отображателей. Однако для сложных запросов и отображений в интерфейс понадобится внедрить большой объем кода.

В настоящей главе мы уделяем внимание определению отображений в MyBatis с помощью XML-стиля.

Выборки “один ко многим” и “многие ко многим” в MyBatis

Продолжим рассмотрение отображения ассоциаций в MyBatis исследованием отношений внутри объекта предметной области `Contact`. Он имеет ассоциацию “один ко многим” с объектом предметной области `ContactTelDetail` и ассоциацию “многие ко многим” с объектом предметной области `Hobby`.

Мы начнем с ассоциации “один ко многим”. В листингах 11.12 и 11.13 показано свойство `contactTelDetails`, добавленное в класс `Contact`, и класс `ContactTelDetail`.

Листинг 11.12. Свойство `contactTelDetails` в классе `Contact`

```
package com.apress.prospring3.ch11.domain;

import java.io.Serializable;
import java.util.*;

public class Contact implements Serializable {

   // Остальной код не показан.

   private Set<ContactTelDetail> contactTelDetails =
      new HashSet<ContactTelDetail>();
}
```

Листинг 11.13. Класс `ContactTelDetail`

```
package com.apress.prospring3.ch11.domain;

import java.io.Serializable;

public class ContactTelDetail implements Serializable {

   private Long id;
   private Contact contact;
   private String telType;
   private String telNumber;

   // Методы извлечения/установки не показаны.

   public String toString() {
      return "Contact Tel Detail - Id: " + id + ", Contact id: "
         + contact.getId() + ", Type: " + telType
         + ", Number: " + telNumber;
   }
}
```

В MyBatis для отображения “один ко многим” (и также “многие ко многим”) типа коллекции мы используем дескриптор `<collection>` (для отношения “один к одному” в MyBatis предусмотрен дескриптор `<association>`). Для ассоциаций отображения MyBatis предлагает два варианта: *вложенные выборки* и *вложенные результаты*. Давайте взглянем на оба варианта и посмотрим, чем они отличаются.

Ассоциации с вложенными выборками

Как и можно было предположить, подход с вложенными выборками означает запуск для каждого контакта дополнительных запросов, которые извлекут сведения о телефонах. Таким образом, если запрос контакта выбрал пять контактов, то для выборки сведений о телефонах в базу данных будут отправлены еще пять запросов, что дает в результате 5 + 1 = 6 запросов. Это хорошо известное условие запроса N+1.

Давайте посмотрим на вложенные выборки в действии, реализовав метод `findAllWithDetail()` в интерфейсе `ContactService`. В листинге 11.14 показано объявление метода `findAllWithDetail()` в интерфейсе `ContactMapper`.

Листинг 11.14. Метод `findAllWithDetail()`

```
package com.apress.prospring3.ch11.persistence;

import java.util.List;

import com.apress.prospring3.ch11.domain.Contact;

public interface ContactMapper {
   public List<Contact> findAll();
   public List<Contact> findAllWithDetail();
}
```

В листинге 11.15 приведено содержимое файла `ContactMapper.xml` с ассоциацией "один ко многим" между объектами `Contact` и `ContactTelDetail`, определенной с помощью вложенной выборки.

Листинг 11.15. Вложенная выборка в XML-определении отображения MyBatis

```
<mapper namespace="com.apress.prospring3.ch11.persistence.ContactMapper">

   <!-- Остальной код не показан. -->

   <resultMap id="contactResultDetailMap" type="Contact">
      <id property="id" column="ID" />
      <result property="firstName" column="FIRST_NAME" />
      <result property="lastName" column="LAST_NAME" />
      <result property="birthDate" column="BIRTH_DATE" />
      <collection property="contactTelDetails" ofType="ContactTelDetail"
         column="id" select="selectTelDetailsForContact">
      </collection>
   </resultMap>

   <resultMap id="contactTelDetailResultMap" type="ContactTelDetail" >
      <id property="id" column="ID" />
      <result property="telType" column="TEL_TYPE"/>
      <result property="telNumber" column="TEL_NUMBER"/>
   </resultMap>

   <select id="findAllWithDetail" resultMap="contactResultDetailMap">
      SELECT
      C.ID,
      C.FIRST_NAME,
      C.LAST_NAME,
      C.BIRTH_DATE
      FROM CONTACT C
   </select>

   <select id="selectTelDetailsForContact" parameterType="long"
      resultType="ContactTelDetail" resultMap="contactTelDetailResultMap">
      SELECT
      ID,
      TEL_TYPE,
      TEL_NUMBER
      FROM CONTACT_TEL_DETAIL WHERE CONTACT_ID = #{id}
   </select>
</mapper>
```

Ниже перечислены основные моменты, касающиеся конфигурации отображения в листинге 11.15.

- Определено новое отображение `<resultMap>` с идентификатором `contactResultDetailMap`. В это отображение результата добавлен дескриптор `<collection>`, отображенный на свойство `contactTelDetails` класса `Contact`. Типом является `ContactTelDetail`, соответствующим оператором для выборки телефона — `selectTelDetailsForContact`, а значением, передаваемым оператору выборки — идентификатор столбца в таблице `CONTACT`.
- Добавлено новое отображение `<resultMap>` с идентификатором `contactTelDetailResultMap`, которое отображает результат операции `selectTelDetailsForContact` на объекты `ContactTelDetail`.
- Добавлена новая операция `<select>` по имени `selectTelDetailsForContact`, которая будет использоваться операцией `findAllWithDetail`. Указанный в `<select>` SQL-оператор получает параметр, который указывает идентификатор каждой записи контакта.

В листинге 11.16 приведена реализация метода `findAllWithDetail()` в классе `ContactServiceImpl`.

Листинг 11.16. Реализация метода `findAllWithDetail()`

```
package com.apress.prospring3.ch11.service.mybatis;

// Операторы импорта опущены.

@Service("contactService")
@Repository
@Transactional
public class ContactServiceImpl implements ContactService {

   // Остальной код не показан.

   @Transactional(readOnly=true)
   public List<Contact> findAllWithDetail() {
      List<Contact> contacts = contactMapper.findAllWithDetail();
      for (Contact contact: contacts) {
         populateContactTelDetail(contact);
      }
      return contacts;
   }

   private void populateContactTelDetail(Contact contact) {
      if (contact.getContactTelDetails() != null) {
         for (ContactTelDetail contactTelDetail:
            contact.getContactTelDetails()) {
            contactTelDetail.setContact(contact);
         }
      }
   }
}
```

Метод `findAllWithDetail()` будет вызывать метод с таким же именем из интерфейса `ContactMapper`. Затем вызывается закрытый метод `populateContactTelDetail()`, который заполнит объектные ссылки `Contact` для каждого объекта `ContactTelDetail`, полученного для каждого контакта.

Чтобы протестировать метод `findAllWithDetail()`, добавьте фрагмент кода из листинга 11.17 в метод `main()` класса `MyBatisSample`.

Листинг 11.17. Тестирование вложенной выборки MyBatis

```
// Найти все контакты вместе с детальной информацией.
contacts = contactService.findAllWithDetail();
listContactsWithDetail(contacts);
```

После этого добавьте в класс `MyBatisSample` метод `listContactsWithDetail()`, показанный в листинге 11.18.

Листинг 11.18. Метод `listContactsWithDetail()`

```
private static void listContactsWithDetail(List<Contact> contacts) {
   System.out.println("");
   System.out.println("Listing contacts with details:");
   for (Contact contact: contacts) {
      System.out.println(contact);
      if (contact.getContactTelDetails() != null) {
         for (ContactTelDetail contactTelDetail:
            contact.getContactTelDetails()) {
            System.out.println(contactTelDetail);
         }
      }
      System.out.println();
   }
}
```

Запуск программы `MyBatisSample` даст следующий вывод (показан не полностью):

```
==>  Executing: SELECT C.ID, C.FIRST_NAME, C.LAST_NAME, C.BIRTH_DATE
                FROM CONTACT C
==> Parameters:
<==    Columns: ID, FIRST_NAME, LAST_NAME, BIRTH_DATE
<==        Row: 1, Clarence, Ho, 1980-07-30
==>  Executing: SELECT ID, TEL_TYPE, TEL_NUMBER
                FROM CONTACT_TEL_DETAIL WHERE CONTACT_ID = ?
==> Parameters: 1(Long)
<==    Columns: ID, TEL_TYPE, TEL_NUMBER
<==        Row: 2, Home, 1234567890
<==        Row: 1, Mobile, 1234567890
<==        Row: 2, Scott, Tiger, 1990-11-02
==>  Executing: SELECT ID, TEL_TYPE, TEL_NUMBER
                FROM CONTACT_TEL_DETAIL WHERE CONTACT_ID = ?
==> Parameters: 2(Long)
<==    Columns: ID, TEL_TYPE, TEL_NUMBER
<==        Row: 3, Home, 1234567890
<==        Row: 3, John, Smith, 1964-02-28
==>  Executing: SELECT ID, TEL_TYPE, TEL_NUMBER
                FROM CONTACT_TEL_DETAIL WHERE CONTACT_ID = ?
==> Parameters: 3(Long)

Listing contacts with details:
Contact - Id: 1, First name: Clarence, Last name: Ho,
          Birthday: Wed Jul 30 00:00:00 CST 1980
Contact Tel Detail - Id: 1, Contact id: 1, Type: Mobile, Number: 1234567890
```

```
Contact Tel Detail - Id: 2, Contact id: 1, Type: Home, Number: 1234567890

Contact - Id: 2, First name: Scott, Last name: Tiger,
          Birthday: Fri Nov 02 00:00:00 CST 1990
Contact Tel Detail - Id: 3, Contact id: 2, Type: Home, Number: 1234567890

Contact - Id: 3, First name: John, Last name: Smith,
          Birthday: Fri Feb 28 00:00:00 CST 1964
```

В этом выводе (другие сообщения DEBUG удалены) видно, что были запущены четыре оператора выборки: один для контакта и по одному для каждого из трех выбранных контактов. Если результирующий набор 1000 записей, то по отдельности будут запускаться 1001 оператор, что породит массовую загрузку данных на сервере базы данных и высокий трафик между контейнером приложения и базой данных.

Другой более элегантный способ предусматривает использование вложенных результатов, и мы рассмотрим его в следующем разделе.

Ассоциации с вложенными результатами

Еще один подход к модулированию ассоциаций в MyBatis заключается в применении вложенных результатов. В сущности, вместо отправки отдельных запросов для каждого родительского объекта основной запрос переписывается с целью соединения таблиц. После этого в отображении результатов свойства вложенных объектов соответствующим образом отображаются, и MyBatis корректно наполнит граф объектов.

Давайте смоделируем отношение “многие ко многим” между объектами Contact и Hobby. В листингах 11.19 и 11.20 показаны пересмотренные объекты предметной области Contact и Hobby.

Листинг 11.19. Свойство hobbies в классе Contact

```
package com.apress.prospring3.ch11.domain;

import java.io.Serializable;
import java.util.*;

public class Contact implements Serializable {

    // Остальной код не показан.
    private Set<Hobby> hobbies = new HashSet<Hobby>();
}
```

Листинг 11.20. Класс Hobby

```
package com.apress.prospring3.ch11.domain;

import java.io.Serializable;

public class Hobby implements Serializable {

    private String hobbyId;

    // Методы извлечения/установки не показаны.

    public String toString() {
        return "Hobby :" + getHobbyId();
    }
}
```

В листинге 11.21 представлено модифицированное содержимое файла ContactMapper.xml с ассоциацией “один ко многим” между объектами Contact и ContactTelDetail, а также ассоциацией “многие ко многим” с объектом Hobby, определенной посредством вложенных результатов.

Листинг 11.21. Вложенные результаты в XML-определении отображения MyBatis

```
<mapper namespace="com.apress.prospring3.ch11.persistence.ContactMapper">
   <!-- Остальной код не показан. -->
   <resultMap id="contactResultDetailMap" type="Contact">
      <id property="id" column="ID" />
      <result property="firstName" column="FIRST_NAME" />
      <result property="lastName" column="LAST_NAME" />
      <result property="birthDate" column="BIRTH_DATE" />
      <collection property="contactTelDetails" ofType="ContactTelDetail">
         <id property="id" column="CONTACT_TEL_ID"/>
         <result property="telType" column="TEL_TYPE"/>
         <result property="telNumber" column="TEL_NUMBER"/>
      </collection>
      <collection property="hobbies" ofType="Hobby">
         <result property="hobbyId" column="HOBBY_ID"/>
      </collection>
   </resultMap>
   <select id="findAllWithDetail" resultMap="contactResultDetailMap">
      SELECT
      C.ID,
      C.FIRST_NAME,
      C.LAST_NAME,
      C.BIRTH_DATE,
      T.ID AS CONTACT_TEL_ID,
      T.TEL_TYPE,
      T.TEL_NUMBER,
      H.HOBBY_ID
      FROM CONTACT C
      LEFT OUTER JOIN CONTACT_TEL_DETAIL T ON C.ID = T.CONTACT_ID
      LEFT OUTER JOIN CONTACT_HOBBY_DETAIL H ON C.ID = H.CONTACT_ID
   </select>
<mapper>
```

Как показано в листинге 11.21, запрос выборки для `findAllWithDetail` был переделан за счет добавления внешнего соединения таблиц `CONTACT_TEL_DETAIL` и `CONTACT_HOBBY_DETAIL`, чтобы извлечь детальную информацию, связанную с контактом.

Отображение результатов `contactResultDetailMap` также было переделано, чтобы напрямую отображать столбцы результирующего набора на свойства вложенных объектов внутри ассоциаций.

Например, для свойства `contactTelDetails` каждое свойство соответствующего объекта `ContactTelDetail` отображается на подходящий столбец внутри оператора выборки. Аналогично, отображение “многие ко многим” для свойства `hobbies` точно такое же. Однако запрос выполняет соединение для таблицы `CONTACT_HOBBY_DETAIL` (которая хранит отображение между таблицами `CONTACT` и `HOBBY`), а результаты отображаются на лежащий в основе объект `Hobby` свойства `hobbies`.

Чтобы протестировать модифицированный метод `findAllWithDetail()`, понадобится изменить код метода `listContactsWithDetail()` с целью отображения также и атрибута `hobbies`. Код этого метода класса `MyBatisSample` показан в листинге 11.22.

Листинг 11.22. Переделанный метод `listContactsWithDetail()`

```
private static void listContactsWithDetail(List<Contact> contacts) {
   System.out.println("");
   System.out.println("Listing contacts with details:");
   for (Contact contact: contacts) {
      System.out.println(contact);
      if (contact.getContactTelDetails() != null) {
         for (ContactTelDetail contactTelDetail:
            contact.getContactTelDetails()) {
            System.out.println(contactTelDetail);
         }
      }
      if (contact.getHobbies() != null) {
         for (Hobby hobby: contact.getHobbies()) {
            System.out.println(hobby);
         }
      }
      System.out.println();
   }
}
```

Отличия с предыдущей версией выделены полужирным. Запуск программы `MyBatisSample` теперь даст следующий вывод (показан не полностью):

```
==>  Executing: SELECT C.ID, C.FIRST_NAME, C.LAST_NAME, C.BIRTH_DATE,
T.ID AS CONTACT_TEL_ID, T.TEL_TYPE, T.TEL_NUMBER, H.HOBBY_ID
FROM CONTACT C LEFT OUTER JOIN CONTACT_TEL_DETAIL T ON C.ID = T.CONTACT_ID
LEFT OUTER JOIN CONTACT_HOBBY_DETAIL H ON C.ID = H.CONTACT_ID
==> Parameters:
<==    Columns: ID, FIRST_NAME, LAST_NAME, BIRTH_DATE, ID, TEL_TYPE,
                TEL_NUMBER, HOBBY_ID
<==       Row: 1, Clarence, Ho, 1980-07-30, 2, Home, 1234567890, Movies
<==       Row: 1, Clarence, Ho, 1980-07-30, 2, Home, 1234567890, Swimming
<==       Row: 1, Clarence, Ho, 1980-07-30, 1, Mobile, 1234567890, Movies
<==       Row: 1, Clarence, Ho, 1980-07-30, 1, Mobile, 1234567890, Swimming
<==       Row: 2, Scott, Tiger, 1990-11-02, 3, Home, 1234567890, Swimming
<==       Row: 3, John, Smith, 1964-02-28, null, null, null, null

Listing contacts with details:
Contact - Id: 1, First name: Clarence, Last name: Ho,
          Birthday: Wed Jul 30 00:00:00 CST 1980
Contact Tel Detail - Id: 1, Contact id: 1, Type: Mobile, Number: 1234567890
Contact Tel Detail - Id: 2, Contact id: 1, Type: Home, Number: 1234567890
Hobby :Movies
Hobby :Swimming

Contact - Id: 2, First name: Scott, Last name: Tiger,
          Birthday: Fri Nov 02 00:00:00 CST 1990
Contact Tel Detail - Id: 3, Contact id: 2, Type: Home, Number: 1234567890
Hobby :Swimming

Contact - Id: 3, First name: John, Last name: Smith,
          Birthday: Fri Feb 28 00:00:00 CST 1964
```

В этом выводе видно, что запускался только один запрос, вернувший шесть записей; библиотека MyBatis соответствующим образом выполнила отображение для графа объектов, а сведения о контакте вместе со всеми деталями были корректно выведены.

Выборки в MyBatis с именованными параметрами

Давайте рассмотрим еще один пример выборки с именованным параметром, которая реализована в методе `findById()`. Этот метод принимает именованный параметр `id` и должен передать его запросу в MyBatis.

В листинге 11.23 представлен метод `findById()`, добавленный к интерфейсу `ContactMapper`.

Листинг 11.23. Метод `findById()`

```
package com.apress.prospring3.ch11.persistence;
import java.util.List;
import com.apress.prospring3.ch11.domain.Contact;
public interface ContactMapper {
   public List<Contact> findAll();
   public List<Contact> findAllWithDetail();
   public Contact findById(Long id);
}
```

Этот метод принимает идентификатор и возвращает найденную запись контакта. В листинге 11.24 показано отображение, определенное в файле `ContactMapper.xml`.

Листинг 11.24. Отображение для операции выборки `findById`

```
   <!-- Остальной код не показан. -->
   <select id="findById" parameterType="long"
      resultMap="contactResultDetailMap">
      SELECT
      C.ID,
      C.FIRST_NAME,
      C.LAST_NAME,
      C.BIRTH_DATE,
      T.ID AS CONTACT_TEL_ID,
      T.TEL_TYPE,
      T.TEL_NUMBER,
      H.HOBBY_ID
      FROM CONTACT C
      LEFT OUTER JOIN CONTACT_TEL_DETAIL T ON C.ID = T.CONTACT_ID
      LEFT OUTER JOIN CONTACT_HOBBY_DETAIL H ON C.ID = H.CONTACT_ID
      WHERE C.ID = #{id}
   </select>
```

Операция выборки использует то же самое отображение результатов, что и предыдущая операция `findAllWithDetail`, которая заполняет граф объектов `Contact`. Запрос также не менялся; к нему просто был добавлен именованный параметр `id`. При вызове метода `ContactMapper.findById()` с передачей ему `id` библиотека MyBatis передаст значение именованного параметра операции выборки.

В листинге 11.25 приведена реализация метода `findById()` в классе `ContactServiceImpl`.

Листинг 11.25. Реализация метода findById()

```
package com.apress.prospring3.ch11.service.mybatis;

// Операторы импорта опущены.

@Service("contactService")
@Repository
@Transactional
public class ContactServiceImpl implements ContactService {

   // Остальной код не показан.
   @Transactional(readOnly=true)
   public Contact findById(Long id) {
      Contact contact = contactMapper.findById(id);
      populateContactTelDetail(contact);
      return contact;
   }

   // Остальной код не показан.
}
```

Для тестирования метода findById() добавьте фрагмент кода из листинга 11.26 в метод main() класса MyBatisSample.

Листинг 11.26. Тестирование метода findById()

```
      // Найти контакт по идентификатору.
      contacts = new ArrayList<Contact>();
      System.out.println("Finding contact with id 1");
      Contact contact = contactService.findById(1l);
      contacts.add(contact);
      listContactsWithDetail(contacts);
```

Запуск программы даст в результате контакт с идентификатором 1.

Выборки в MyBatis с использованием динамического SQL

Мощным средством MyBatis для программирования операций выборки является динамический SQL, который значительно упрощает разработку сложных запросов. Для демонстрации его использования добавим в интерфейс ContactMapper метод findByFirstNameAndLastName(), как показано в листинге 11.27.

Листинг 11.27. Метод findByFirstNameAndLastName()

```
package com.apress.prospring3.ch11.persistence;

import java.util.List;

import com.apress.prospring3.ch11.domain.*;

public interface ContactMapper {

   public List<Contact> findAll();

   public List<Contact> findAllWithDetail();

   public Contact findById(Long id);

   public List<Contact> findByFirstNameAndLastName(SearchCriteria criteria);
}
```

В листинге 11.27 вместо приема двух параметров (имени и фамилии) мы определили Java-класс для хранения критерия поиска. Этот класс `SearchCriteria` представлен в листинге 11.28.

Листинг 11.28. Класс `SearchCriteria`

```
package com.apress.prospring3.ch11.domain;

public class SearchCriteria {
   private String firstName;
   private String lastName;
   // Методы извлечения/установки не показаны.
}
```

Этот класс просто хранит возможный критерий поиска в своих атрибутах. Давайте посмотрим, каким образом критерий поиска заменяется динамическим SQL в MyBatis. В листинге 11.29 приведено отображение для операции `findByFirstNameAndLastName`.

Листинг 11.29. Отображение для операции `findByFirstNameAndLastName`

```
<!-- Остальной код не показан. -->
<select id="findByFirstNameAndLastName" parameterType="SearchCriteria"
   resultMap="contactResultDetailMap">
   SELECT
   C.ID,
   C.FIRST_NAME,
   C.LAST_NAME,
   C.BIRTH_DATE,
   T.ID AS CONTACT_TEL_ID,
   T.TEL_TYPE,
   T.TEL_NUMBER,
   H.HOBBY_ID
   FROM CONTACT C
   LEFT OUTER JOIN CONTACT_TEL_DETAIL T ON C.ID = T.CONTACT_ID
   LEFT OUTER JOIN CONTACT_HOBBY_DETAIL H ON C.ID = H.CONTACT_ID
   <where>
      <if test="firstName != null">
         first_name = #{firstName}
      </if>
      <if test="lastName != null">
         AND last_name = #{lastName}
      </if>
   </where>
</select>
```

В листинге 11.29 вся магия скрыта в дескрипторе `<where>`, который будет обрабатываться MyBatis для интеллектуального формирования динамического SQL. Например, в нем сначала проверяется, равен ли `null` передаваемый атрибут `firstName` в `SearchCriteria`. Если нет, к запросу добавляется конструкция `WHERE FIRST_NAME =` и этот атрибут. То же самое действие предпринимается и для атрибута `lastName`. Один интересный момент заключается в том, что если не предоставить имя, то MyBatis интеллектуальным образом удалит операцию `AND` из конструкции `WHERE`, в которой используется фамилия. Если оба упомянутых атрибута равны `null`, MyBatis просто пол-

ностью отбросит конструкцию WHERE. В листинге 11.30 приведена реализация метода findByFirstNameAndLastName() в классе ContactServiceImpl.

Листинг 11.30. Реализация метода findByFirstNameAndLastName()

```
package com.apress.prospring3.ch11.service.mybatis;

// Операторы импорта опущены.

@Service("contactService")
@Repository
@Transactional
public class ContactServiceImpl implements ContactService {

   // Остальной код не показан.
   @Transactional(readOnly=true)
   public List<Contact> findByFirstNameAndLastName(String firstName,
      String lastName) {
      log.info("Finding contact with first name: " + firstName
         + " and last name: " + lastName);
      Contact contact = new Contact();
      contact.setFirstName(firstName);
      contact.setLastName(lastName);
      SearchCriteria criteria = new SearchCriteria();
      criteria.setFirstName(firstName);
      criteria.setLastName(lastName);

      List<Contact> contacts = contactMapper.findByFirstNameAndLastName(criteria);
      for (Contact contactTemp: contacts) {
         populateContactTelDetail(contactTemp);
      }

      return contacts;
   }
}
```

Как показано в листинге 11.30, метод по-прежнему принимает в качестве параметров имя и фамилию. Тем не менее, он будет конструировать экземпляр SearchCriteria и передавать его ContactMapper для извлечения данных. Чтобы протестировать метод findByFirstNameAndLastName(), добавьте фрагмент кода из листинга 11.31 в метод main() класса MyBatisSample.

Листинг 11.31. Тестирование метода findByFirstNameAndLastName()

```
// Найти контакт по имени и фамилии.
contacts = contactService.findByFirstNameAndLastName("Clarence", "Ho");
listContactsWithDetail(contacts);
```

Запуск этого кода приводит к выводу соответствующей информации о контакте. Можете попробовать разные комбинации и понаблюдать за SQL-оператором, генерируемым MyBatis.

Кроме <where> в MyBatis предлагается множество различных дескрипторов для построения очень гибких операторов динамического SQL. Кроме того, для конструирования динамического SQL в Java-коде MyBatis предоставляет такие классы, как SqlBuilder и SelectBuilder. За более подробными сведениями обращайтесь к официальной документации на веб-сайте MyBatis.

Вставка данных

А теперь мы посмотрим, как вставлять новую информацию о контакте в базу данных, используя MyBatis. Для операций выборки мы определили только интерфейс отображателя `ContactMapper`. Тем не менее, для поддержки операции вставки ассоциаций (т.е. `ContactTelDetail` и `Hobby`) нам также нужно определить интерфейс отображателя и XML-файлы конфигурации. В листингах 11.32 и 11.33 показан новый метод `insertContact()` для интерфейса `ContactMapper` и соответствующее отображение запроса в файле `ContactMapper.xml`.

Листинг 11.32. Операция `insertContact` для `ContactMapper`

```
package com.apress.prospring3.ch11.persistence;

import java.util.List;

import com.apress.prospring3.ch11.domain.*;
public interface ContactMapper {

   public List<Contact> findAll();

   public List<Contact> findAllWithDetail();

   public Contact findById(Long id);

   public List<Contact> findByFirstNameAndLastName(SearchCriteria criteria);

   public void insertContact(Contact contact);

}
```

Листинг 11.33. Операция `insertContact` в файле `ContactMapper.xml`

```
   <insert id="insertContact" parameterType="Contact" useGeneratedKeys="true"
      keyProperty="id">
      INSERT INTO CONTACT (FIRST_NAME, LAST_NAME, BIRTH_DATE)
      VALUES (#{firstName}, #{lastName}, #{birthDate})
   </insert>
```

В листинге 11.33 видно, что с помощью дескриптора `<insert>` мы определяем отображение для операции вставки. Атрибутом `parameterType`, естественно, является объект `Contact`. Кроме того, атрибут `useGeneratedKeys` инструктирует MyBatis о необходимости извлечения ключа записи, сгенерированного базой данных во время вставки, и свойство для значения ключа должно быть сохранено в атрибуте `id` объекта `Contact`.

В листингах 11.34 и 11.35 представлен код интерфейса `ContactTelDetailMapper` (с методом `insertContactTelDetail()`) и содержимое файла отображения `ContactTelDetailMapper.xml`.

Листинг 11.34. Интерфейс `ContactTelDetailMapper`

```
package com.apress.prospring3.ch11.persistence;

import com.apress.prospring3.ch11.domain.ContactTelDetail;

public interface ContactTelDetailMapper {

   public void insertContactTelDetail(ContactTelDetail contactTelDetail);

}
```

Листинг 11.35. Файл отображения ContactTelDetailMapper.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<!-- Отображатель для телефонов контакта -->
<mapper namespace="com.apress.prospring3.ch11.persistence.ContactTelDetailMapper">
   <insert id="insertContactTelDetail" parameterType="ContactTelDetail"
      useGeneratedKeys="true" keyProperty="id">
      INSERT INTO CONTACT_TEL_DETAIL (CONTACT_ID, TEL_TYPE, TEL_NUMBER)
      VALUES (#{contact.id}, #{telType}, #{telNumber})
   </insert>
</mapper>
```

Ассоциация “многие ко многим” между объектами `Contact` и `Hobby` полагается на таблицу отображения `CONTACT_HOBBY_DETAIL`, поэтому для нее мы также должны определить объект предметной области, интерфейс отображателя и XML-конфигурацию отображения. Объект `ContactHobbyDetail` показан в листинге 11.36.

Листинг 11.36. Объект предметной области ContactHobbyDetail

```
package com.apress.prospring3.ch11.domain;

public class ContactHobbyDetail {
   private Long contactId;
   private String hobbyId;
   // Методы извлечения/установки не показаны.
}
```

В листингах 11.37 и 11.38 показан код интерфейса `ContactHobbyDetailMapper` (с методом `insertContactHobbyDetail()`) и содержимое файла отображения `ContactHobbyDetailMapper.xml`.

Листинг 11.37. Интерфейс ContactHobbyDetailMapper

```
package com.apress.prospring3.ch11.persistence;

import com.apress.prospring3.ch11.domain.ContactHobbyDetail;

public interface ContactHobbyDetailMapper {
   void insertContactHobbyDetail(ContactHobbyDetail contactHobbyDetail);
}
```

Листинг 11.38. Файл отображения ContactHobbyDetailMapper.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<!-- Отображатель для хобби контакта -->
<mapper namespace="com.apress.prospring3.ch11.persistence.ContactHobbyDetailMapper">
   <insert id="insertContactHobbyDetail" parameterType="ContactHobbyDetail">
      INSERT INTO CONTACT_HOBBY_DETAIL (CONTACT_ID, HOBBY_ID)
      VALUES (#{contactId}, #{hobbyId})
   </insert>
</mapper>
```

Теперь мы можем реализовать метод `save()` в классе `ContactServiceImpl`. Фрагмент кода приведен в листинге 11.39.

Листинг 11.39. Реализация операции вставки

```
package com.apress.prospring3.ch11.service.mybatis;

// Операторы импорта опущены.

@Service("contactService")
@Repository
@Transactional
public class ContactServiceImpl implements ContactService {
   // Остальной код не показан.
   @Autowired
   private ContactTelDetailMapper contactTelDetailMapper;
   @Autowired
   private ContactHobbyDetailMapper contactHobbyDetailMapper;
   public Contact save(Contact contact) {
      if (contact.getId() == null) {
         insert(contact);
      }
      return contact;
   }
   private Contact insert(Contact contact) {
      contactMapper.insertContact(contact);
      Long contactId = contact.getId();
      ContactHobbyDetail contactHobbyDetail;
      if (contact.getContactTelDetails() != null) {
         for (ContactTelDetail contactTelDetail:
            contact.getContactTelDetails()) {
            contactTelDetail.setContact(contact);
            contactTelDetailMapper.insertContactTelDetail(contactTelDetail);
         }
      }
      if (contact.getHobbies() != null) {
         for (Hobby hobby: contact.getHobbies()) {
            contactHobbyDetail = new ContactHobbyDetail();
            contactHobbyDetail.setContactId(contactId);
            contactHobbyDetail.setHobbyId(hobby.getHobbyId());
            contactHobbyDetailMapper.insertContactHobbyDetail(
               contactHobbyDetail);
         }
      }
      return contact;
   }
}
```

Как показано в листинге 11.39, отображатель для ассоциаций автосвязан с классом службы. Вдобавок метод `save()` проверяет существование атрибута `id` в экземпляре `Contact`. Если он не существует, то это операция вставки, и будет вызван закрытый метод `insert()`. В методе `insert()` сначала сохраняется запись контакта и затем из объекта `Contact` извлекается идентификатор. Этот идентификатор используется в операциях вставки соответствующих деталей о телефонах и хобби. Чтобы протестировать операцию вставки, добавьте фрагмент кода из листинга 11.40 в метод `main()` класса `MyBatisSample`.

Листинг 11.40. Тестирование реализации метода `save()`

```
// Добавить новый контакт.
System.out.println("Add new contact");
contact = new Contact();
contact.setFirstName("Michael");
contact.setLastName("Jackson");
contact.setBirthDate(new Date());
Set<ContactTelDetail> contactTelDetails =
   new HashSet<ContactTelDetail>();
ContactTelDetail contactTelDetail = new ContactTelDetail();
contactTelDetail.setTelType("Home");
contactTelDetail.setTelNumber("111111");
contactTelDetails.add(contactTelDetail);
contactTelDetail = new ContactTelDetail();
contactTelDetail.setTelType("Mobile");
contactTelDetail.setTelNumber("222222");
contactTelDetails.add(contactTelDetail);
contact.setContactTelDetails(contactTelDetails);
Set<Hobby> hobbies = new HashSet<Hobby>();
Hobby hobby = new Hobby();
hobby.setHobbyId("Jogging");
hobbies.add(hobby);
contact.setHobbies(hobbies);
contactService.save(contact);
contacts = contactService.findAllWithDetail();
listContactsWithDetail(contacts);
```

В листинге 11.40 конструируется новый контакт, а затем создаются две записи телефонов и одна запись хобби. После этого вызывается метод `save()` интерфейса `ContactService`. Запуск этой программы дает следующий вывод (показан не полностью):

```
Contact - Id: 4, First name: Michael, Last name: Jackson,
          Birthday: Wed Oct 26 00:00:00 CST 2011
Contact Tel Detail - Id: 4, Contact id: 4, Type: Home, Number: 111111
Contact Tel Detail - Id: 5, Contact id: 4, Type: Mobile, Number: 222222
Hobby :Jogging
```

Здесь видно, что был создан новый контакт с идентификатором 4 и соответствующими данными для телефонов и хобби.

Обновление данных

Обновление данных контакта немного сложнее, в основном, из-за ассоциаций “один ко многим” для телефонов и “многие ко многим” для хобби. Давайте реализуем метод обновления со следующей последовательностью операций.

- Обновить запись контакта.
- Перед обновлением деталей о телефонах извлечь существующую информацию о телефонах для контакта и сохранить список идентификаторов для удаления висячих записей.
- Для каждого телефона проверить, существует ли идентификатор. Если он не существует, вставить новую запись. В противном случае обновить существующую запись. Кроме того, удалить этот идентификатор из списка висячих записей.

После обновления всех деталей о телефонах проверить список идентификаторов для любых висячих записей телефонов. Если такие записи существуют, удалить их.

- Для хобби сначала удалить все существующие отображения из таблицы CONTACT_HOBBY_DETAIL и затем вставить новые отображения из обновленного объекта Contact.

Эта последовательность представляет собой типичный механизм обработки; его можно подстроить под специфические нужды. Сначала понадобится добавить операцию обновления в интерфейс ContactMapper и XML-файл. Соответствующие фрагменты кода приведены в листингах 11.41 и 11.42.

Листинг 11.41. Операция updateContact для ContactMapper

```
package com.apress.prospring3.ch11.persistence;
import java.util.List;
import com.apress.prospring3.ch11.domain.*;
public interface ContactMapper {
   public List<Contact> findAll();
   public List<Contact> findAllWithDetail();
   public Contact findById(Long id);
   public List<Contact> findByFirstNameAndLastName(SearchCriteria criteria);

   public void insertContact(Contact contact);
   public void updateContact(Contact contact);
}
```

Листинг 11.42. Операция updateContact в файле ContactMapper.xml

```
   <update id="updateContact" parameterType="Contact">
      UPDATE CONTACT SET
      FIRST_NAME = #{firstName},
      LAST_NAME = #{lastName},
      BIRTH_DATE = #{birthDate}
      WHERE ID = #{id}
   </update>
```

В листинге 11.42 следует отметить только один новый момент: использование дескриптора <update> для операции обновления.

В интерфейс ContactTelDetailMapper и связанный с ним XML-файл необходимо добавить три операции. Первая операция предназначена для извлечения существующих телефонов по идентификатору контакта. Вторая операция отвечает за обновление существующих деталей о телефонах, а третья — за удаление висячих записей телефонов через список идентификаторов. В листингах 11.43 и 11.44 показаны фрагменты кода для интерфейса ContactTelDetailMapper (с новыми методами selectTelDetailForContact(), updateContactTelDetail() и deleteOrphanContactTelDetail()) и файла ContactTelDetailMapper.xml.

Листинг 11.43. Интерфейс ContactTelDetailMapper

```
package com.apress.prospring3.ch11.persistence;
import java.util.List;
import com.apress.prospring3.ch11.domain.ContactTelDetail;
public interface ContactTelDetailMapper {
   public void insertContactTelDetail(ContactTelDetail contactTelDetail);
   public List<ContactTelDetail> selectTelDetailForContact(Long contactId);
   public void updateContactTelDetail(ContactTelDetail contactTelDetail);
   public void deleteOrphanContactTelDetail(List<Long> ids);
}
```

Листинг 11.44. Файл отображения ContactTelDetailMapper.xml

```
<!-- Отображатель для телефонов контакта -->
<mapper namespace="com.apress.prospring3.ch11.persistence.
ContactTelDetailMapper">
   <!--  Другие отображения не показаны  -->
   <resultMap id="contactTelDetailResultMap" type="ContactTelDetail">
      <id property="id" column="id" />
      <result property="telType" column="tel_type" />
      <result property="telNumber" column="tel_number" />
   </resultMap>
   <select id="selectTelDetailForContact" parameterType="long"
      resultMap="contactTelDetailResultMap">
      SELECT
      ID,
      TEL_TYPE,
      TEL_NUMBER
      FROM CONTACT_TEL_DETAIL WHERE CONTACT_ID = #{contactId}
   </select>
   <update id="updateContactTelDetail" parameterType="ContactTelDetail">
      UPDATE CONTACT_TEL_DETAIL SET
      TEL_TYPE = #{telType},
      TEL_NUMBER = #{telNumber}
      WHERE ID = #{id}
   </update>
   <delete id="deleteOrphanContactTelDetail">
      DELETE FROM CONTACT_TEL_DETAIL
      WHERE ID IN
      <foreach item="item" index="index" collection="list"
         open="(" separator="," close=")">
         #{item}
      </foreach>
   </delete>
</mapper>
```

В листинге 11.44 видно, что операция `selectTelDetailForContact` выбирает детали о телефонах по идентификатору контакта и возвращает список объектов

ContactTelDetail. Операция updateContactTelDetail предназначена для обновления существующей записи телефона, а операция deleteOrphanContactTelDetail — для удаления висячих записей телефонов. Для удаления применяется дескриптор <foreach>, предоставляемый MyBatis, которому передается список идентификаторов. MyBatis самостоятельно построит оператор удаления DELETE FROM CONTACT_TEL_DETAIL WHERE ID IN (...). Это еще один полезный дескриптор.

В интерфейс ContactHobbyDetailMapper и XML-файл понадобится добавить еще одну операцию — метод deleteHobbyDetailForContact() для удаления всех отображений, связанных с контактом.

В листинге 11.45 и 11.46 приведен код интерфейса ContactHobbyDetailMapper (с новым методом deleteHobbyDetailForContact()) и содержимое файла ContactHobbyDetailMapper.xml для операции обновления.

Листинг 11.45. Интерфейс ContactHobbyDetailMapper

```
package com.apress.prospring3.ch11.persistence;

import com.apress.prospring3.ch11.domain.ContactHobbyDetail;

public interface ContactHobbyDetailMapper {
   public void insertContactHobbyDetail(ContactHobbyDetail contactHobbyDetail);
   public void deleteHobbyDetailForContact(Long contactId);
}
```

Листинг 11.46. Файл отображения ContactHobbyDetailMapper.xml

```
<!-- Остальной код не показан. -->

<delete id="deleteHobbyDetailForContact" parameterType="long">
   DELETE FROM CONTACT_HOBBY_DETAIL WHERE CONTACT_ID = #{contactId}
</delete>
```

Теперь можно заняться реализацией метода update() в классе ContactServiceImpl. Соответствующий фрагмент кода представлен в листинге 11.47.

Листинг 11.47. Реализация операции обновления

```
// Остальной код не показан.

public Contact save(Contact contact) {
   if (contact.getId() == null) {
      insert(contact);
   } else {
      update(contact);
   }
   return contact;
}

private Contact update(Contact contact) {
   contactMapper.updateContact(contact);
   Long contactId = contact.getId();
   ContactHobbyDetail contactHobbyDetail;

   // Список для хранения идентификаторов висячих записей телефонов.
   List<Long> ids = new ArrayList<Long>();
   // Извлечь существующие телефоны для контакта.
   List<ContactTelDetail> oldContactTelDetails =
      contactTelDetailMapper.selectTelDetailForContact(contactId);
```

```
    for (ContactTelDetail contactTelDetail: oldContactTelDetails) {
       ids.add(contactTelDetail.getId());
    }
    // Обновить детали о телефонах.
    if (contact.getContactTelDetails() != null) {
      for (ContactTelDetail contactTelDetail:
          contact.getContactTelDetails()) {
          if (contactTelDetail.getId() == null) {
            contactTelDetailMapper.insertContactTelDetail(contactTelDetail);
          } else {
            contactTelDetailMapper.updateContactTelDetail(contactTelDetail);
            ids.remove(contactTelDetail.getId());
          }
        }
        if (ids.size() > 0) {
          contactTelDetailMapper.deleteOrphanContactTelDetail(ids);
        }
    }
   // Обновить детали о хобби.
   contactHobbyDetailMapper.deleteHobbyDetailForContact(contact.getId());
   if (contact.getHobbies() != null) {
     for (Hobby hobby: contact.getHobbies()) {
       contactHobbyDetail = new ContactHobbyDetail();
       contactHobbyDetail.setContactId(contactId);
       contactHobbyDetail.setHobbyId(hobby.getHobbyId());
       contactHobbyDetailMapper.insertContactHobbyDetail(contactHobbyDetail);
     }
   }
   return contact;
}
```

В листинге 11.47 логика обновления реализована согласно последовательности, описанной в начале этого раздела. Чтобы протестировать операцию обновления, добавьте фрагмент кода из листинга 11.48 в метод `main()` класса `MyBatisSample`.

Листинг 11.48. Тестирование реализации метода `save()` для операции обновления

```
// Обновить контакт.
System.out.println("Update contact with id 1");
contact = contactService.findById(1l);
contact.setFirstName("Kim Fung");
Set<ContactTelDetail> contactTels = contact.getContactTelDetails();
ContactTelDetail toDeleteContactTel = null;
for (ContactTelDetail contactTel: contactTels) {
   if (contactTel.getTelType().equals("Home")) {
      toDeleteContactTel = contactTel;
   }
}
contactTels.remove(toDeleteContactTel);
hobby = new Hobby();
hobby.setHobbyId("Jogging");
contact.getHobbies().add(hobby);
contactService.save(contact);
contacts = contactService.findAllWithDetail();
listContactsWithDetail(contacts);
```

В листинге 11.48 сначала извлекается контакт с идентификатором 1. Затем обновляется имя, удаляется запись домашнего телефона и добавляется новая запись хобби. После этого вызывается метод `save()` для сохранения контакта. Запуск программы дает следующий вывод (показан не полностью):

```
Contact - Id: 1, First name: Kim Fung, Last name: Ho,
          Birthday: Wed Jul 30 00:00:00 CST 1980
Contact Tel Detail - Id: 1, Contact id: 1, Type: Mobile, Number: 1234567890
Hobby :Movies
Hobby :Swimming
Hobby :Jogging
```

Как видите, контакт был успешно обновлен.

Удаление данных

Удалять данные намного проще. Необходимо только реализовать операции для удаления контакта в таблице `CONTACT`, а также связанных с ним записей в таблицах `CONTACT_TEL_DETAIL` и `CONTACT_HOBBY_DETAIL`. В листингах 11.49 и 11.50 показан новый метод `deleteContact()` для интерфейса `ContactMapper` и соответствующее отображение для запроса из файла `ContactMapper.xml`.

Листинг 11.49. Операция `deleteContact` для `ContactMapper`

```
package com.apress.prospring3.ch11.persistence;
import java.util.List;
import com.apress.prospring3.ch11.domain.*;
public interface ContactMapper {
   public List<Contact> findAll();
   public List<Contact> findAllWithDetail();
   public Contact findById(Long id);
   public List<Contact> findByFirstNameAndLastName(SearchCriteria criteria);
   public void insertContact(Contact contact);
   public void updateContact(Contact contact);
   void deleteContact(Long id);
}
```

Листинг 11.50. Операция `deleteContact` в файле `ContactMapper.xml`

```
   <delete id="deleteContact" parameterType="long">
      DELETE FROM CONTACT WHERE ID = #{id}
   </delete>
```

В листинге 11.50 присутствует только один новый момент: использование дескриптора `<delete>` для указания операции удаления.

В листингах 11.51 и 11.52 представлен код интерфейса `ContactTelDetailMapper` (с новым методом `deleteTelDetailForContact()`) и содержимое файла отображения `ContactTelDetailMapper.xml`.

Листинг 11.51. Интерфейс ContactTelDetailMapper

```
package com.apress.prospring3.ch11.persistence;
import java.util.List;
import com.apress.prospring3.ch11.domain.ContactTelDetail;
public interface ContactTelDetailMapper {
   public List<ContactTelDetail> selectTelDetailForContact(Long contactId);
   public void insertContactTelDetail(ContactTelDetail contactTelDetail);
   public void updateContactTelDetail(ContactTelDetail contactTelDetail);
   public void deleteOrphanContactTelDetail(List<Long> ids);
   void deleteTelDetailForContact(Long contactId);
}
```

Листинг 11.52. Файл отображения ContactTelDetailMapper.xml

```
   <delete id="deleteTelDetailForContact" parameterType="long">
      DELETE FROM CONTACT_TEL_DETAIL WHERE CONTACT_ID = #{contactId}
   </delete>
```

В интерфейсе `ContactHobbyDetailMapper` добавлять новую операцию не нужно, поскольку операция для удаления отображений хобби, связанных с контактом, уже реализована в рамках операции обновления.

Теперь можно реализовать метод `delete()` в классе `ContactServiceImpl`. Фрагмент кода приведен в листинге 11.53.

Листинг 11.53. Реализация операции удаления

```
   // Остальной код не показан.
   public void delete(Contact contact) {
      Long contactId = contact.getId();
      contactTelDetailMapper.deleteTelDetailForContact(contactId);
      contactHobbyDetailMapper.deleteHobbyDetailForContact(contactId);
      contactMapper.deleteContact(contactId);
   }
```

Метод удаления очень прост. Сначала удаляются дочерние записи, т.е. детали о телефонах и хобби. Затем удаляется запись о контакте.

Чтобы протестировать операцию удаления, добавьте фрагмент кода из листинга 11.54 в метод `main()` класса `MyBatisSample`.

Листинг 11.54. Тестирование реализации метода delete()

```
      // Удалить контакт.
      System.out.println("Delete contact with id 1");
      contact = contactService.findById(1l);
      contactService.delete(contact);
      contacts = contactService.findAllWithDetail();
      listContactsWithDetail(contacts);
```

Запуск этой программы приводит к удалению контакта с идентификатором 1.

Соображения по поводу того, когда использовать MyBatis

Как должно быть ясно из примеров, представленных в этой главе, MyBatis является великолепной библиотекой доступа к данным, ориентированной на SQL-отображении. При разработке приложений с применением MyBatis большую часть времени вы будете сосредоточены на проектировании и отображении результирующих наборов и различных операций выборки, вставки, обновления и удаления.

В терминах отображения MyBatis предоставляет широкий набор дескрипторов, которые помогают строить сложные динамические запросы и без труда отображать их на соответствующие объекты предметной области. В примерах было показано, что взаимодействовать напрямую с интерфейсом `SqlSession` не требуется. В большинстве случаев интерфейсы отображателей способны удовлетворить все существующие потребности.

Тем не менее, при желании можно взаимодействовать с интерфейсом `SqlSession` и конструировать запросы программным путем.

В завершение отметим, что если вы предпочитаете иметь больший контроль над SQL-операторами, отправляемыми базе данных, и хотите заполучить гибкий и простой способ определения отображений результирующих наборов на объекты предметной области, то MyBatis предлагает намного более элегантное решение, чем JDBC.

Однако при использовании MyBatis следует принимать во внимание ряд соображений. Так, понадобится реализовать механизм блокирования записей (например, столбец `VERSION` для оптимистической блокировки) и вручную сохранять версии записей хронологии. Кроме того, если модель данных часто изменяется, нужно будет пересматривать все запросы, на которые это изменение оказывает влияние. В смысле сопровождения усилия, затрачиваемые при использовании MyBatis, будут большими, нежели в случае применения таких библиотек ORM, как Hibernate и JPA.

Использование MyBatis в примере приложения

В этом разделе мы обсудим отношения между темами, рассмотренными в главе, и примером приложения, которое мы будем разрабатывать. Темы включают базу данных и реализацию MyBatis для различных операций базы данных.

Мы также покажем, как использовать средства MyBatis для упрощения логики доступа к данным.

Наконец, мы посмотрим, как реализовать в примере приложения отслеживание базовой информации аудита и версий сущностей, несмотря на то, что MyBatis не предоставляет готовой поддержки этих возможностей.

База данных серверной части

В качестве базы данных серверной части будет использоваться встроенная база данных JDBC типа H2. Тем не менее, сценарии (для создания база данных и для заполнения ее начальными данными) будут разрабатываться так, чтобы быть совместимыми также и с MySQL. Это позволит при желании запускать пример приложения с базой данных MySQL.

В конфигурации `ApplicationContext` мы просто применим дескриптор `<jdbc:embedded-database>` для объявления встроенной базы данных. Будут написаны два SQL-сценария. Один из них (по имени `schema.sql`) предназначен для создания базы данных, а другой (по имени `initial-data.sql`) — для наполнения начальными данными (начальными пользователями, категориями и подкатегориями, записями блога и т.д.).

Использование MyBatis для реализации уровня постоянства

Как упоминалось в главе 3, для уровня постоянства будут предоставлены две разных реализации: одна будет использовать JPA, а другая — MyBatis.

В реализации MyBatis мы будем применять MyBatis 3. В частности, будут использоваться интерфейсы отображателей MyBatis и XML-конфигурации отображения. Все сущностные классы будут помещены в пакет `com.apress.prospring3.springblog.domain` (на самом деле для отображения будет применяться один и тот же набор объектов предметной области как в JPA, так и в MyBatis).

Вдобавок все интерфейсы отображателей MyBatis (например, интерфейс `EntryMapper` для поддержки сущностного класса `Entry`) будут помещены в пакет `com.apress.prospring3.springblog.persistence`. Более того, для поддержки просмотра записей блога в рамках пользовательского интерфейса веб-приложения будет реализовано средство разбиения на страницы.

Что касается операций базы данных, то наиболее сложной частью является выполнение запросов. В пользовательском интерфейсе можно фильтровать записи по датам создания, категориям и подкатегориям и т.п. Для этого будет применяться средство динамического SQL, предлагаемое MyBatis.

Аудит и отслеживание версий сущностей

В примере приложения для записей блога и комментариев к ним будут включены средства аудита.

Для отслеживания базовой информации аудита (пользователь, который создал; дата создания; пользователь, который внес последнюю модификацию; дата последней модификации) для записей блога и комментариев, а также сохранения версий каждой записи мы будем пользоваться механизмом подключаемого модуля MyBatis, перехватывая вызовы операции обновления и соответствующим образом формируя базовую информацию аудита. С другой стороны, в таблице хронологии будет сохраняться копия записи в состоянии перед обновлением.

Дополнительные детали можно найти в главе 21.

Резюме

В этой главе было показано, как Spring интегрируется с библиотекой MyBatis (ранее называемой iBATIS), предназначенной для разработки логики доступа к данным. Подробно рассматривались ключевые концепции, конфигурация, определение отображений и т.д.

Кроме того, мы обсудили выполнение разнообразных операций базы данных с использованием MyBatis. Мы продемонстрировали различные приемы извлечения данных с разными ассоциациями и показали, как применять динамический SQL.

В главе была рассмотрена только небольшая часть средств, доступных в MyBatis для реализации наиболее общих операций базы данных. За более подробными сведениями о MyBatis и документацией по этой библиотеке обращайтесь на веб-сайт проекта по адресу `www.mybatis.org`.

ГЛАВА 12

Проектирование и реализация Spring-приложений

В предшествующих главах мы рассматривали различные темы, связанные с разработкой приложений с помощью Spring, в том числе конфигурация `ApplicationContext`, разнообразные механизмы внедрения зависимостей, поддержка АОП для сквозной функциональности, а также интеграция Spring с другими технологиями доступа к данным для взаимодействия с реляционными базами данных.

Во время обсуждения реализаций логики доступа к данным вы узнали, как использовать различные шаблоны реализации. Например, в главе 8, посвященной доступу к данным с помощью JDBC, логика доступа к данным была инкапсулирована в объекты DAO. В главах 10 и 11, в которых рассматривались JPA и MyBatis, мы избавились от уровня DAO и внедрили `EntityManager` из JPA и интерфейсы отображателей из MyBatis напрямую в уровень обслуживания. В этих главах мы также упомянули о проектировании объектов предметной области (или сущностный классов в терминологии JPA) и показали, каким образом моделировать и отображать их атрибуты и отношения на структуру базы данных.

Перед тем, как продолжить, давайте ненадолго оставим в стороне тему программирования и поговорим о некоторых основных принципах проектирования и реализации Spring-приложений. Мы считаем, что ознакомление с приемами проектирования JEE-приложений поможет лучше ориентироваться в темах, связанных с общей архитектурой приложений.

Проектирование приложений — крупная тема, и мы не намерены раскрывать здесь все ее аспекты. В этой главе мы рассмотрим некоторые опробованные и протестированные приемы ООП, обеспечивающие получение в результате приложений, которые имеют четко определенные ответственности компонентов и характеризуются простотой тестирования и сопровождения. В главе мы расскажем о влиянии Spring на проектирование приложений, обращая особое внимание на шаблоны и приемы, которые вы найдете простыми для применения при построении собственных приложений с помощью Spring. Большая часть материала этой главы сфокусирована на проектных решениях, принятых во время построения приложения SpringBlog, и мы будем использовать эту главу как основу для примеров и обсуждений. В дополнение к принципам проектирования приложений мы покажем, как применять многие технологии Spring, рассмотренные до сих пор, для реализации уровней доступа к данным и служб в приложении SpringBlog.

В частности, в этой главе рассматриваются следующие темы.

- *Проектирование, управляемое интерфейсами.* Проектирование, управляемое интерфейсами, представляет собой наилучшую практику традиционного ООП. При таком подходе главные компоненты приложения определяются в терминах интерфейсов, а не конкретных классов. В Java предлагается первоклассная поддержка такого вида проектирования с отдельными нотациями для интерфейсов и классов. В этом разделе главы мы обсудим общие аспекты проектирования, управляемого интерфейсами, и причины его применения в приложениях.
- *Построение объектной модели предметной области.* В этом разделе главы мы рассмотрим понятие объектной модели предметной области (Domain Object Model — DOM) — коллекции объектов, которые предоставляют абстрактную модель данных в предметной области приложения. Создавая DOM для приложения, вы получаете набор объектов для моделирования данных и поведения приложения, что соответствует некоторым абстрактным идеям предметной области.
- *Уровень доступа к данным.* Большинству современных приложений требуется доступ к какому-то постоянному хранилищу данных, которым обычно является реляционная база данных. В главах с 8 по 11 обсуждалась поддержка Spring для разнообразных методов доступа к данным. В этом разделе главы мы рассмотрим вопросы проектирования, связанные с построением уровня доступа к данным для обслуживания остальной части приложения.
- *Построение уровня обслуживания.* На уровне обслуживания приложения инкапсулирована вся бизнес-логика, образующая собственно приложение. В этом разделе мы посмотрим, как уровень обслуживания взаимодействует с DOM и уровнем доступа к данным с целью предоставления унифицированного интерфейса для доступа к функциональности приложения. Мы также опишем бизнес-требования для приложения SpringBlog и объясним, как они транслируются в проектирование и реализацию этого интерфейса.

Следует отметить, что в этой главе не раскрываются вопросы проектирования или реализации веб-уровня приложения; все это подробно рассматривается в главах 17 и 18. В частности, мы не показываем ни каким образом пользовательский веб-интерфейс взаимодействует с уровнем обслуживания SpringBlog, ни как Spring использует логику проверки достоверности для поддержки проверки данных и сообщения об ошибках в пользовательском интерфейсе. Этим темам посвящены главы 17 и 18.

Проектирование интерфейсов

Проектирование интерфейсов — это обычная практика при построении архитектуры и реализации приложения, не зависящая от применяемой платформы или стандарта (Spring, JBoss Seam, Guice, EJB и т.п.). Одной из главных проектных целей Spring является дальнейшее упрощение разработки приложений, которые спроектированы и реализованы на основе набора хорошо определенных интерфейсов. Прежде чем мы сможем детально рассматривать проектирование Spring-приложения, мы должны хорошо разобраться с тем, почему проектирование интерфейсов представляет собой такую важную задачу и как Spring упрощает ее решение.

Причины выбора подхода с проектированием интерфейсов

Существует множество причин, по которым приложения должны проектироваться и кодироваться с применением интерфейсов, а не иерархии конкретных классов, и,

пожалуй, самой важной причиной является снижение связности. Если компоненты в приложении взаимодействуют друг с другом в терминах интерфейсов, а не конкретных типов, становится очень легко заменить какой-либо компонент, когда с ним начинают возникать проблемы. Эта возможность позволяет переключаться от одной реализации к другой, не затрагивая остальной код приложения; и действительно, приложение может работать с множеством разных реализаций одного и того же интерфейса, даже не зная, что он делает. Например, при использовании поддержки профилей в Spring можно предоставить различные конкретные реализации того же самого интерфейса, не оказывая влияния на остальные части приложения.

Помните также, что в Java класс поддерживает только одиночное конкретное наследование, но может реализовывать любое необходимое количество интерфейсов. Определяя компоненты приложения в терминах интерфейсов, а не классов, вы не ограничиваете класс реализации каким-то заданным базовым классом, не являющимся обязательным. Еще одно преимущество, как было показано в главе 7, связано с тем, что использование интерфейсов делает возможной поддержку введений АОП для "смешивания" индивидуальных интерфейсов с целевыми бинами при построении сквозной функциональности, не внося дополнительной сложности в исходный код целевых компонентов.

С точки зрения тестирования одним из главных преимуществ такой слабой связности является повышение тестируемости. Как руководители сильно занятых команд разработчиков, мы постоянно ищем новые способы по улучшению степени покрытия тестами в создаваемых приложениях, поскольку это прямой путь к увеличению качества конечных продуктов, поставляемых заказчикам. За счет проектирования интерфейсов мы можем заменять какую-то реализацию имитированной реализацией, что позволяет проводить тестирование более гибким образом. Например, при выполнении модульного тестирования веб-контроллера мы определенно хотим сосредоточиться на выяснении, корректно ли себя ведет этот контроллер, предполагая, что служебные классы, от которых он зависит, работают правильно. В таком случае очень полезно предоставить "имитированную" версию служебного класса, которая во время тестирования всегда возвращает веб-контроллеру ожидаемые результаты.

Шаблон "Фабрика"

Одной из ключевых проблем, с которыми вы столкнетесь при реализации приложения, где все компоненты определены в терминах интерфейсов, является нахождение приложением экземпляров классов, реализующих эти интерфейсы. Традиционное решение проблемы — использование шаблона "Фабрика" (Factory Pattern). Шаблон "Фабрика" определяет класс, отвечающий за предоставление компонентам приложения реализаций других компонентов приложения; в этом случае доступные компоненты определяются в терминах интерфейсов, а не конкретных реализаций. Представим себе систему, в которой имеется бизнес-интерфейс по имени `OrderService`. Другие компоненты в системе желают получать реализации этого интерфейса, не зная заранее моменты времени, когда они понадобятся. Для реализации такой системы можно построить класс `BasicFactory`, подобный показанному в листинге 12.1. Обратите внимание, что интерфейс `OrderService` и класс реализации `DefaultOrderServiceImpl` (реализующий интерфейс `OrderService`) являются пустыми, не имея ни одного определенного метода.

Листинг 12.1. Базовый класс реализации фабрики

```
package com.apress.prospring3.ch12.factory;

import com.apress.prospring3.ch12.service.OrderService;
import com.apress.prospring3.ch12.service.impl.DefaultOrderServiceImpl;
```

```
public class BasicFactory {
   private static final BasicFactory instance;
   private OrderService orderService;

   static {
      instance = new BasicFactory();
   }

   public static BasicFactory getlnstance() {
      return instance;
   }

   public BasicFactory() {
      this.orderService = new DefaultOrderServiceImpl();
   }

   public OrderService getOrderService() {
      return this.orderService;
   }
}
```

Это упрощенная реализация класса фабрики, но она иллюстрирует подход базового шаблона “Фабрика”. Теперь любое приложение, которому нужен доступ к реализации `OrderService`, просто использует метод `getOrderService()` класса `BasicFactory`.

Недостатки базового шаблона “Фабрика”

В своей базовой форме шаблон “Фабрика” имеет три главных недостатка.

- Отсутствует возможность изменить класс реализации без повторной компиляции.
- Отсутствует возможность сделать прозрачно доступными множество реализаций разным компонентам. Это часть более крупной проблемы, заключающейся в том, что класс фабрики требует, чтобы каждый компонент был осведомлен как о самом классе, так и о методе, который необходимо вызывать.
- Отсутствует возможность простого переключения между моделями создания экземпляров. В листинге 12.1 поддерживается одиночный экземпляр `DefaultOrderServiceImpl`, но если необходимо возвращать множество экземпляров, класс `BasicFactory` потребуется перекомпилировать.

Эти недостатки более подробно обсуждаются в следующих трех разделах.

Внешне конфигурируемые фабрики

В примере, приведенном в листинге 12.1, видно, что изменение класса реализации означает изменение класса `BasicFactory` и повторную компиляцию. Одно из достоинств проектирования на основе интерфейсов состоит в том, что можно очень просто заменять старые реализации новыми. Тем не менее, необходимость в перекомпиляции класса фабрики в какой-то мере уменьшает простоту, с которой это может быть сделано. Во многих проектах в прошлом, до появления Spring, мы создавали фабрики, которые позволяли указывать класс реализации для интерфейсов во внешнем файле конфигурации (как было показано в главе 2 при реализации класса `MessageSupportFactory`). Это решает первоначальную проблему, но добавляет к проекту дополнительные затраты на разработку, к тому же никак не помогает в решении оставшихся двух проблем.

Прозрачная поддержка множества реализаций

Невозможность прозрачной поддержки множества реализаций, пожалуй, является наибольшим недостатком традиционного шаблона “Фабрика”. Базовый метод класса `BasicFactory`, `getOrderService()`, может возвращать только одну конкретную реализацию (или случайно выбранную реализацию), но он не позволяет выбирать реализацию для возврата на основе вызывающего класса. Естественно, это приводит к реализации, такой как показанная в листинге 12.2. Обратите внимание, что `SuperOrderServiceImpl` также представляет собой пустой класс, который реализует интерфейс `OrderService`.

Листинг 12.2. Базовая поддержка множества реализаций

```
package com.apress.prospring3.ch12.factory;

import com.apress.prospring3.ch12.service.OrderService;
import com.apress.prospring3.ch12.service.impl.DefaultOrderServiceImpl;
import com.apress.prospring3.ch12.service.impl.SuperOrderServiceImpl;

public class MultiFactory {
   private static final BasicFactory instance;
   private OrderService orderService;
   private OrderService superOrderService;
   static {
      instance = new BasicFactory();
   }

   public static BasicFactory getInstance() {
      return instance;
   }

   public MultiFactory() {
      this.orderService = new DefaultOrderServiceImpl();
      this.superOrderService = new SuperOrderServiceImpl();
   }

   public OrderService getOrderService() {
      return this.orderService;
   }

   public OrderService getSuperOrderService() {
      return this.superOrderService;
   }
}
```

Благодаря такой реализации, компоненты, нуждающиеся в доступе к классу реализации `SuperOrderServiceImpl`, могут вызывать метод `getSuperOrderService()`. Однако такой подход попросту нивелирует преимущества фабрики. Хотя компоненты не связаны с конкретной реализацией класса, они связаны с методом, который вызывается из фабрики. Еще один недостаток этого подхода состоит в том, что каждая новая реализация требует изменения кода фабрики и компонента, которому нужна новая реализация. Необходимость добавления нового метода для каждой новой реализации существенно усложняет возможность внешнего конфигурирования для данного подхода.

Другая реализация, которая пытается решить проблему прозрачной поддержки множества реализаций, требует, чтобы компоненты, вызывающие метод `getOrderService()`, передавали ему свои классы, на основе чего фабрика может решить, какую реализацию возвращать. Такая реализация страдает от многочисленных проблем, не последней из которых является то, что она работает только с классами, а

это значит, что два экземпляра одного и того же класса не могут получить разные реализации `OrderService`. Вы также обнаружите, что реализация метода `getOrderService()` очень быстро становится запутанной, если есть много компонентов, которым необходима реализация `OrderService`.

Несколько более элегантный подход к решению этой проблемы предусматривает использование поиска, заставляя каждый компонент искать нужную реализацию по ключу. Таким образом, вместо вызова `getOrderService()` компонент вызывает `getOrderService("someKey")`. Недостаток этого подхода в том, что для обеспечения высокой гибкости каждый компонент должен применять отдельный ключ, поэтому его реализация может изменяться отдельно от других. Когда зависимому объекту требуется другая реализация той же самой бизнес-службы, он должен изменить код для использования ключа соответствующей реализации.

Корень этой проблемы произрастает из того факта, что компонент активно должен запрашивать класс реализации, и чтобы достичь высокой гибкости и обеспечить каждому компоненту возможность получения подходящего класса реализации, запрос должен быть полностью уникальным. Эта проблема не может быть решена с применением традиционного шаблона “Фабрика”.

Поддержка множества режимов создания экземпляров

Еще одна проблема связана с поддержкой для разных компонентов множества режимов создания экземпляров класса реализации. Эта проблема объясняется многими моментами, которые обсуждались в предшествующем разделе, и она, опять-таки, вызвана тем, что компонент активно должен запрашивать класс реализации. Данная проблема также не может быть решена с применением традиционного шаблона “Фабрика”. К счастью, в Spring все упомянутые проблемы устранены; об этом пойдет речь в следующем разделе.

Влияние Spring на проектирование, основанное на интерфейсах

Платформа Spring оказывает большое влияние на приложения, проектируемые с использованием интерфейсов. Поскольку Spring заботится о связывании всех компонентов вместе, вам больше не понадобится переживать по поводу создания классов фабрик, учитывающих каждую возможную ситуацию.

На первый взгляд, при построении приложений, основанных на интерфейсах, самым большим преимуществом применения Spring является сокращение объема связующего кода, который приходится писать. Это преимущество усиливается благодаря великолепной встроенной поддержке внешней конфигурации для зависимостей между компонентами. Однако наибольшее преимущество обеспечивается использованием внедрения зависимостей Spring. Так как Spring снимает ответственность за размещение зависимостей с самих компонент и просто запрашивает, разрешают ли компоненты предоставлять их в зависимостях, эта платформа способна решить последние две из трех ранее обсуждаемых проблем.

Внедрение зависимостей означает, что Spring может предоставить любой экземпляр любого класса реализации любому экземпляру любого компонента приложения, не требуя написания никакого специального кода внутри компонента. Это связано с тем, что Spring может свободно управлять жизненным циклом любого экземпляра любой зависимости, поддерживаемой для компонента приложения.

В целом это означает, что Spring обладает всеми функциональными возможностями, необходимыми для проектирования приложений, основанных на интерфейсах, и во время реализации нам не нужно заботиться о том, каким образом компоненты будут связываться друг с другом.

Построение объектной модели предметной области

Объектная модель предметной области (Domain Object Model — DOM) — это набор классов, которые моделируют концепции предметной области. Тема DOM довольно обширна, и в этом разделе мы рассмотрим только основные концепции DOM. За более полным описанием шаблона DOM рекомендуем обратиться к книгам *Шаблоны корпоративных приложений* Мартина Фаулера (ИД "Вильямс", 2012 г.) и *Предметно-ориентированное проектирование: структуризация сложных программных систем* Эрика Эванса (ИД "Вильямс", 2011 г.). Хотя мы не будем вдаваться в особые подробности этого шаблона, мы все же объясним, почему решили создать DOM для приложения SpringBlog, и покажем, как строить собственную DOM-модель.

Spring и объектная модель предметной области

Учитывая, что эта книга посвящена Spring, может показаться странным выделение определенного объема на рассмотрение темы, которая не имеет непосредственного отношения к Spring. В приложениях, построенных с помощью Spring, объектами предметной области являются только те объекты, которые не управляются Spring. (Даже несмотря на то, что допускается иметь управляемые Spring объекты предметной области за счет применения аннотации `@Component` к классам и назначения им области действия на уровне прототипа, большую часть времени мы будем выбирать вариант управления объектами предметной области внутри приложения.) Причина в том, что на практике платформа Spring не должна иметь дело с объектами предметной области. В общем случае мы создаем множество экземпляров объектов предметной области, используя операцию `new()`, и выполняем обработку либо на уровне обслуживания, либо на уровне доступа к данным. Хотя Spring также поддерживает внедрение нового экземпляра объекта предметной области при каждом его запросе (за счет указания `prototype` для области действия бина), обычно разработчики не применяют такой подход. Это объясняется тем, что, как правило, объекты предметной области не используют внедрение зависимостей по причине наличия у них нескольких зависимостей за пределами самой DOM-модели, и они не требуют особого конфигурирования.

Может также возникнуть вопрос: почему DOM уделяется так много внимания? Ответ на этот вопрос довольно прост. Модель DOM является наиболее критичной областью, которая оказывает влияние на такое большое количество других частей приложения — частей, управляемых Spring, — что ее корректность очень важна для правильной работы приложения в целом.

DOM — это не то же самое, что объект значения

Относительно шаблона DOM важно понимать, что он не является тем же самым, что и шаблон "Объект значения" (Value Object Pattern), часто называемый шаблоном "Объект передачи данных" (Data Transfer Object Pattern). Шаблон "Объект значения" был создан с целью преодоления недостатка в исходной спецификации EJB, который заключался в том, что все вызовы EJB были удаленными. Конфигурирование состояния EJB обычно означает наличие множества вызовов, а все они являются удаленными. За счет использования объекта значения состояние EJB передается целиком в пакете с помощью единственного удаленного вызова, таким образом сглаживая выбросы производительности, которые возникают при множестве удаленных вызовов.

На заметку! Формально шаблон “Объект значения” — это не то же самое, что шаблон “Объект передачи данных”. Мартин Фаулер определяет объект значения как небольшой простой объект, подобный денежной сумме или диапазону дат, равенство которого не основано на идентичности. Дополнительные сведения можно найти по адресу `http://martinfowler.com/bliki/ValueObject.html`. Однако из-за каталога ключевых шаблонов J2EE (Core J2EE Patterns) возникает путаница, т.к. в нем термин *объект значения* используется для многих примеров шаблона “Объект передачи данных”. (За подробной информацией обращайтесь к документу “Core J2EE Patterns: Transfer Object” (Ключевые шаблоны J2EE: объект передачи) и диаграмме классов для объекта передачи в онлайновом каталоге по адресу `http://java.sun.com/blueprints/corej2eepatterns/Patterns/TransferObject.html`.) В этом разделе мы применяем термины *объект значения* и *объект передачи данных* взаимозаменяемо, но имеем в виду шаблон “Объект передачи данных”.

Модель DOM — это основанное на объектах представление предметной области приложения, предназначение которого состоит в том, чтобы позволить программисту проводить кодирование в терминах объектов, существующих в конкретной предметной области. Хотя объект значения заключает в себе чистое состояние, он хорошо подходит для хранения состояния и поведения объекта предметной области (несмотря на то, что поведение в объектах предметной области можно не инкапсулировать).

Другим важным отличием между объектами предметной области и объектами значений является то, что структура объекта значения обусловлена необходимостью в удаленной передаче данных, тогда как объект предметной области моделируется для представления концепции реального мира и не управляется какой-то потребностью инфраструктуры приложения. Как мы будем обсуждать позже, мы считаем, что никаких жестких и быстрых правил для моделирования объектов предметной области не существует; вы должны самостоятельно выбрать уровень детализации, соответствующий вашему приложению, и функции, которые будут выполняться.

В приложении могут присутствовать как объекты предметной области, так и объекты значений. При таком подходе объекты значений используются уровнем обслуживания для взаимодействия с другими уровнями, такими как уровень презентаций и уровень доступа к данным. Эти объекты значений затем соответствующим образом преобразуются в объекты предметной области и передаются уровню презентаций для визуализации. В общем случае применять этот подход не рекомендуется. Одна из причин заключается в усложнении сопровождения, потому что изменение объектов предметной области означает также изменение связанных объектов значений. Другая причина состоит в том, что предлагаемая Spring инфраструктура доступа к данным и веб является настолько мощной, что лучше просто отображать данные непосредственно на объекты предметной области, как для обработки, так и для представления.

Причины создания объектной модели предметной области

Создание DOM-модели требует затрат некоторых предварительных усилий по идентификации объектов предметной области и построения кодового представления этих объектов. Тем не менее, эти предварительные усилия с лихвой компенсируются временем, которое вы сэкономите, и ошибками, которых вы избежите, когда дело дойдет реализации бизнес-логики, манипулирующей объектами предметной области. Мы считаем, что использование качественной DOM-модели намного упрощает написание кода для решения бизнес-задач, т.к. появляется возможность кодировать в терминах задачи, а не машины, на которой будет выполняться приложение. Качественная DOM-модель упрощает разработчикам трансформацию требований приложения в функциональные средства приложения.

Моделирование объектов предметной области

Для моделирования объектов предметной области существует очень много разных методик и подходов. Некоторые разработчики выступают в защиту того, чтобы позволить лежащему в основе хранилищу данных управлять объектной моделью, тогда как другие считают, что объектной моделью должна управлять сама предметная область. На практике мы обнаружили, что, придерживаясь золотой середины между этими двумя подходами, можно получить в результате DOM-модель, которая одновременно является простой в работе и хорошо выполненной.

Для простых приложений с пятью или шестью таблицами базы данных часто проще создавать по одному объекту предметной области, соответствующему каждой таблице базы данных. Хотя эти объекты не являются строго объектами предметной области — в том, что их созданием управляет не предметная область, а структура данных — они достаточно близко соответствуют целям такого простого приложения. Действительно, во многих простых приложениях результатом широкого моделирования предметной области является объектная модель, которая соответствует полной структуре базы данных.

Для более крупных приложений реальной предметной области и лежащему в основе хранилищу данных должно уделяться намного больше внимания. При построении DOM-модели для приложения мы обычно фокусируемся на трех главных моментах:

- как структурирована предметная область;
- как будут использоваться объекты предметной области;
- как конструируется лежащее в основе хранилище данных.

Мы ищем DOM-модель, которая была бы максимально возможно близка к идеальной, не оказывала слишком большого воздействия на производительность хранилища данных и не сильно влияла на код, использующий объекты предметной области.

Обычно DOM-модель является довольно детальной, и для одиночной логической концепции может быть получено более одного класса. Например, рассмотрим концепцию заказа в закупочной системе. Обычно заказ моделируется как одиночный объект `Order` с одним и более объектами `OrderLine`, представляющими каждую строку заказа. Попытка смоделировать заказ с помощью одного объекта приводит к получению излишне грубой и громоздкой объектной модели, которую трудно реализовать и сопровождать. Всегда следует искать возможности увеличения степени детализации объектов предметной области, если это упрощает работу с DOM.

Вы также обнаружите, что DOM-модель содержит объекты, которые в хранилище данных не существуют. Например, в типичной закупочной системе имеется понятие товарной тележки, возможно представленной объектами `Cart` и `CartItem`. Если только не требуется сохранять содержимое между сеансами, скорее всего, эти объекты предметной области не будут иметь соответствующих таблиц в хранилище данных. Помните, что вы не просто строите объектно-ориентированное представление базы данных — вы моделируете предметную область. Этот момент трудно переоценить. Нам приходилось видеть немало проектов, в которых создавались псевдомодели DOM, напрямую порожденные из хранилищ данных; эти проекты неизбежно страдали от недостатка абстракции, которую можно было бы получить из хорошо определенной DOM-модели.

Мы выяснили, что цельная DOM-модель получается в случае, если потратить должное время на исследование предметной области, идентификацию объектов в предметной области и выяснение, удовлетворяет ли естественная детализация этих объектов требованиям приложения. Хотя мы принимаем во внимание как использование объектов предметной области, так и лежащее в основе хранилище, мы не хотим позволить им чрезмерно влиять на DOM-модель.

Важно не забывать, что целью построения DOM-модели является создание набора классов, которые помогут вам и другим разработчикам построить приложение на таком уровне абстракции, который близок к предметной области приложения. В общем случае при создании DOM-модели мы считаем все остальные вопросы вторичными. Например, если вы обнаруживаете, что производительность страдает из-за проектного решения, положенного в основу DOM, проведите тонкую настройку модели. Однако удостоверьтесь, что проблема не связана с проектированием DOM (скажем, имеется одиночный объект предметной области, содержащий большое количество длинных текстовых полей вроде CLOB или TEXT, которые негативно влияют на производительность извлечения данных). Вы не должны преуменьшать преимущества DOM-модели, ошибочно полагая, что она выполняется плохо.

Моделирование базы данных и моделирование объектов предметной области

Хотя моделирование базы данных и моделирование объектов предметной области очень похожи, получаемые результаты редко оказываются одинаковыми, и действительно, редко когда требуется, чтобы они были таковыми. При моделировании базы данных вы ищете структуру, которая позволит сохранять и извлекать данные наиболее эффективным и согласованным образом. При построении DOM-модели производительность очевидно важна, но именно так строится приложение поверх объектно-ориентированной модели, которая является простой в работе и облегчает сборку бизнес-логики. В общем, мы обнаружили, что базу данных лучше всего моделировать способом, наилучшим для базы данных, а DOM-модель — по крайней мере, в начале — способом, который лучше подходит для DOM-модели. Позже, когда будут идентифицированы узкие места производительности, можно внести любые изменения.

Моделирование отношений между объектами предметной области

Наиболее распространенной ошибкой, которую мы видим в DOM-моделях, особенно когда DOM-модель управляется проектным решением базы данных, является создание объектов предметной области для представления отношений между табличными сущностями. Это произрастает из того факта, что отношение “многие ко многим” между двумя таблицами в базе данных должно иметь конструирующую его таблицу соединения. Отношения в DOM должны моделироваться в намного более объектно-ориентированном стиле, с объектами предметной области, поддерживающими ссылки на другие объекты предметной области или списки таких объектов.

Общая ошибка при заполнении данными объектов предметной области из базы, вроде того, что делается на уровне доступа к данным в приложении, заключается в предположении, что все связанные объекты предметной области (например, когда данные из таблицы `ORDER` извлекаются в объект предметной области `Order`, вместе с ними из таблицы `ORDER_ITEM` должен быть извлечен список объектов `Item` внутри `Order`) должны загружаться также из базы данных; это далеко не так. Более детальное обсуждение этой проблемы приведено в разделе “Отношения между объектами предметной области” далее в главе.

Инкапсулировать поведение или нет?

Вы не обязаны инкапсулировать поведение в объекты предметной области; на самом деле эти объекты могут представлять только состояние предметной области. В большинстве случаев мы считаем, что лучше выносить максимальный объем бизнес-логики в набор служебных объектов, работающих с объектами предметной области, чем ин-

капсулировать эту логику в объектах предметной области. Обычно мы помещаем всю логику, которая взаимодействует с компонентами за пределами DOM-модели, в служебные объекты. Подобным образом мы снижаем степень связности между DOM-моделью и компонентами, задействованными в логике приложения. Это позволяет применять DOM-модель в более широком спектре сценариев, и часто обнаруживается, что DOM-модель можно повторно использовать в других приложениях, которые решают задачи из той же самой предметной области.

Когда желательно инкапсулировать поведение в DOM-модели — это ситуации, при которых логика реализуется чисто для взаимодействия между объектами предметной области. Приложение JPetStore, включенное в Spring, предоставляет хороший пример этого, и данный пример можно отобразить на рассматриваемый пример закупочной системы. При таком сценарии пользователь имеет тележку для покупок, представленную объектом `Cart`, и список объектов `CartItem`. Когда пользователь готов к покупке товаров в тележке и оформлению заказа, приложение создает объект `Order` вместе со списком объектов `OrderLine`, которые соответствуют данным, моделируемым с помощью объектов `Cart` и `CartItem`. Это великолепный пример, когда поведение должно быть инкапсулировано в DOM-модели. Преобразование из `Cart` в `Order` кодируется исключительно в терминах объектов предметной области, без каких-либо зависимостей от других компонентов в приложении.

В приложении JPetStore класс `Order` имеет метод `initOrder()`, принимающий два аргумента, `Account` и `Cart`. В этом методе находится вся логика, требуемая для создания объекта `Order` на основе объекта `Cart` для пользователя, представленного объектом `Account`.

Как и в большинстве аспектов, связанных с моделированием, нет никаких жестких и быстрых правил о том, когда помещать логику внутрь объекта предметной области, а когда выносить ее в служебный объект. Помещения логики в объекты предметной области следует избегать, если это приводит к образованию зависимостей объектов предметной области от других компонентов, находящихся за пределами DOM-модели. Подобным образом мы обеспечиваем максимально возможную степень повторного использования DOM-модели. В противоположность этому, логика, имеющая дело только с объектами предметной области, идеальна для помещения в DOM-модели, и это дает возможность применять ее везде, где используется сама DOM-модель.

Объектная модель предметной области для примера приложения SpringBlog

Давайте взглянем на DOM-модель для примера приложения. На рис. 12.1 показана DOM-модель SpringBlog для объектов предметной области, относящихся к сообщениям блога, а на рис. 12.2 — DOM-модель для пользователей блога и назначенных им ролей.

Хотя DOM-модель не очень сложна, она подчеркивает некоторые аспекты, о которых мы должны упомянуть особо. Все они обсуждаются в следующих трех разделах.

Наследование в DOM-модели SpringBlog

Концепция сообщения является центральной в приложении SpringBlog. Сообщения бывают двух типов: записи, представляющие собой высокоуровневые сообщения в блоге, и комментарии, которые относятся к определенной записи блога. Мы решили определить общие характеристики сообщения в интерфейсе `BlogPosting`, показанному в листинге 12.3, и классы `Entry` и `Comment` реализуют этот интерфейс.

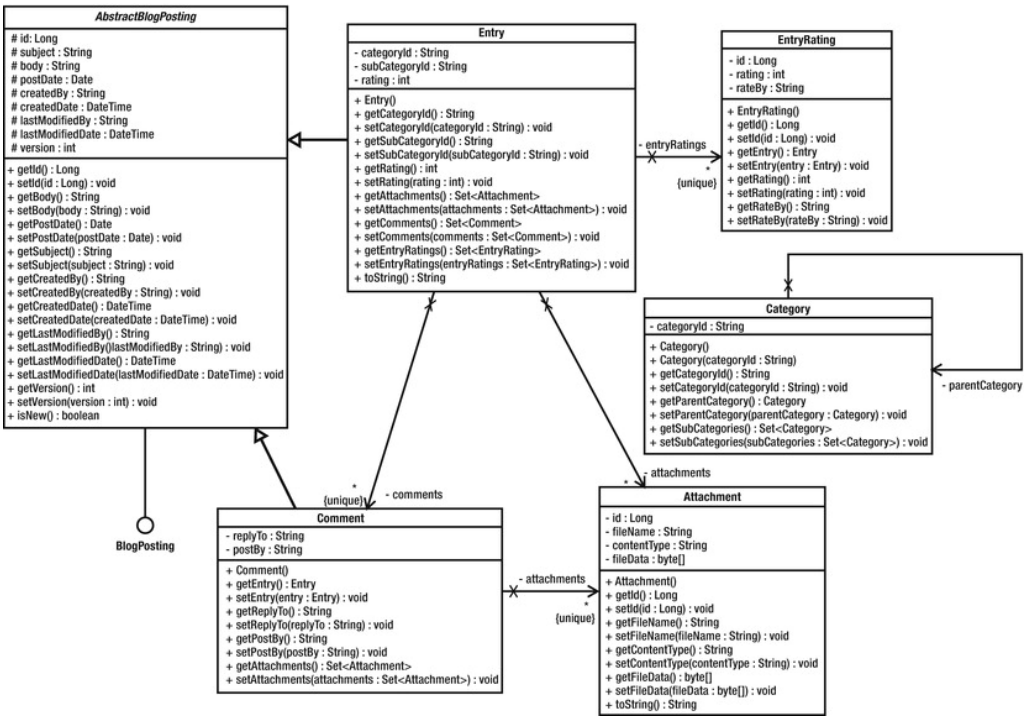


Рис. 12.1. DOM-модель SpringBlog для сообщений блога

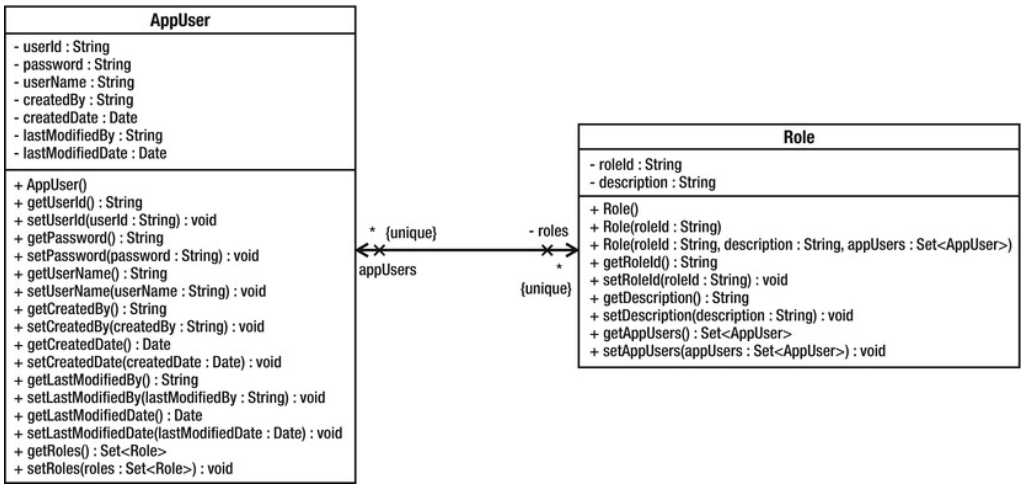


Рис. 12.2. DOM-модель SpringBlog для пользователей блога и ролей

Листинг 12.3. Интерфейс BlogPosting

```
package com.apress.prospring3.springblog.domain;
import java.util.Date;
import java.util.Set;
import com.apress.prospring3.springblog.domain.Attachment;

public interface BlogPosting {
   public String getBody();
   public void setBody(String body);
   public Date getPostDate();
   public void setPostDate(Date postDate);
   public String getSubject();
   public void setSubject(String subject);
}
```

Однако это приводит в результате к излишнему дублированию кода, причем классы Entry и Comment имеют собственные реализации интерфейса BlogPosting. Чтобы обойти это, мы вводим класс AbstractBlogPosting и заставляем Entry и Comment расширять этот класс. Класс AbstractBlogPosting показан в листинге 12.4.

Листинг 12.4. Класс AbstractBlogPosting

```
package com.apress.prospring3.springblog.domain;

// Операторы импорта опущены.

public abstract class AbstractBlogPosting implements BlogPosting {
   protected Long id;
   protected String subject;
   protected String body;
   protected Date postDate;
   protected String createdBy;
   protected DateTime createdDate;
   protected String lastModifiedBy;
   protected DateTime lastModifiedDate;
   protected int version;
   public String getBody() {
      return body;
   }
   public void setBody(String body) {
      this.body = body;
   }
   public Date getPostDate() {
      return postDate;
   }
   public void setPostDate(Date postDate) {
      this.postDate = postDate;
   }
   public String getSubject() {
      return subject;
   }
   public void setSubject(String subject) {
      this.subject = subject;
   }

   // Методы извлечения/установки не показаны.
}
```

За счет расширения этого базового класса мы перемещаем все детали реализации `BlogPosting` из `Entry` и `Comment`, сокращая дублирование кода. В качестве примера в листинге 12.5 показан код класса `Entry`.

Листинг 12.5. Класс `Entry`

```
package com.apress.prospring3.springblog.domain;

// Операторы импорта опущены.

public class Entry extends AbstractBlogPosting {

   private static final int MAX_BODY_LENGTH = 80;
   private static final String THREE_DOTS = "...";

   private String categoryId;
   private String subCategoryId;
   private Set<EntryAttachment> attachments = new HashSet<EntryAttachment>();
   private Set<Comment> comments = new HashSet<Comment>();

   public Entry() {
   }

   public String getShortBody() {
      if (body.length() <= MAX_BODY_LENGTH)
         return body;
      StringBuffer result = new StringBuffer(MAX_BODY_LENGTH + 3);
      result.append(body.substring(0, MAX_BODY_LENGTH));
      result.append(THREE_DOTS);
      return result.toString();
   }

   public String getCategoryId() {
      return this.categoryId;
   }

   public void setCategoryId(String categoryId) {
      this.categoryId = categoryId;
   }

   public String getSubCategoryId() {
      return this.subCategoryId;
   }

   public void setSubCategoryId(String subCategoryId) {
      this.subCategoryId = subCategoryId;
   }
   public Set<EntryAttachment> getAttachments() {
      return this.attachments;
   }

   public void setAttachments(Set<EntryAttachment> attachments) {
      this.attachments = attachments;
   }

   public Set<Comment> getComments() {
      return this.comments;
   }

   public void setComments(Set<Comment> comments) {
      this.comments = comments;
   }
}
```

Этот шаблон широко используется в Spring и по всему приложению SpringBlog. Общая функциональность определена в интерфейсах, а не в абстрактных классах, но мы предоставляем стандартную реализацию интерфейса в виде абстрактного класса. Причина применения такого подхода в том, что где это возможно, мы можем использовать данный абстрактный базовый класс с `Entry` и `Comment`, устраняя необходимость в реализации интерфейса `BlogPosting` напрямую. Однако если для класса `Entry` возникает требование расширить класс `Foo`, мы может просто реализовать интерфейс `BlogPosting` прямо в `Entry`. Основной момент, который здесь следует запомнить: вы не определяете общую функциональность в терминах абстрактных классов, поскольку это ограничит вас набором иерархий наследования. Вместо этого общая функциональность определяется в терминах интерфейсов с предоставлением стандартных реализаций этих интерфейсов в виде абстрактных базовых классов. При таком подходе можно пользоваться унаследованной реализацией, где это возможно, но не ограничивать искусственно иерархию наследования.

Важно отметить, что мы не отражаем это дерево наследования в базе данных. Другими словами, мы не создаем таблицу `BLOG_POSTING` для хранения совместно используемых данных и еще две таблицы, `ENTRY` и `COMMENT`, для хранения данных, специфичных для сущностей. Основной причиной такого подхода является то, что мы не считаем, что приложение размера SpringBlog оправдывает сложность этой структуры; кроме того, рассмотренный пример подчеркивает нашу точку зрения о наличии DOM-модели, которая отличается по своей структуре от базы данных. Главная причина определения такой иерархии наследования, помимо того, что это удачное проектное решение, связана с необходимостью позволить приложению SpringBlog работать с общими данными в объектах `Entry` и `Comment`, не проводя различия между ними. Хорошим примером может служить фильтр нежелательного контента, который был введен в главе 7 и детально рассматривается в главе 21.

Поведение предметной области в SpringBlog

Несмотря на то что модель предметной области SpringBlog является упрощенной, мы еще должны инкапсулировать в нее некоторую логику. Поскольку тело сообщения блога потенциально может быть очень большим, нам необходим механизм получения фрагмента тела для использования при отображении списка сообщений блога. Для этой цели мы создадим метод `Entry.getShortBody()` (обратите внимание, что данный метод не был определен ни в интерфейсе `BlogPosting`, ни в классе `AbstractBlogPosting`), код которого показан в листинге 12.6.

Листинг 12.6. Поведение в классе `Entry`

```
package com.apress.prospring3.springblog.domain;
// Операторы импорта опущены.
public class Entry extends AbstractBlogPosting {
   private static final int MAX_BODY_LENGTH = 80;
   private static final String THREE_DOTS = "...";
   public String getShortBody() {
      if (body.length() <= MAX_BODY_LENGTH)
         return body;
      StringBuffer result = new StringBuffer(MAX_BODY_LENGTH + 3);
      result.append(body.substring(0, MAX_BODY_LENGTH));
      result.append(THREE_DOTS);
      return result.toString();
   }
   // Остальной код не показан.
}
```

В коде видно, что для построения сокращенного тела мы берем первые 80 символов тела и добавляем три точки в конце. Хотя это упрощенная реализация, она отражает типовой сценарий инкапсуляции логики в DOM-модели.

Отношения между объектами предметной области

В DOM-модели, представленной на рис. 12.1, обратите внимание на определение ассоциации между `Entry` и `Attachment`, а также между `Comment` и `Attachment`. В качестве части требований приложения SpringBlog мы хотим иметь возможность загрузки и сохранения файлов с обоими типами сообщений. В базе данных мы имеем таблицу по имени `ATTACHMENT` для хранения вложений. Для ассоциирования вложений с записью или комментарием предусмотрены две других таблицы: `ENTRY_ATTACHMENT_DETAIL` и `COMMENT_ATTACHMENT_DETAIL`. Общая ошибка, которую мы нередко наблюдаем, состоит в том, что для моделирования этих отношений некоторые разработчики создают объекты предметной области, а не используют стандартные средства Java для связывания объектов вместе. Отношение "один к одному" в базе данных можно моделировать в DOM с помощью одного объекта, поддерживающего ссылку на другой. Для отношений "один ко многим" и "многие ко многим" использование Java-коллекций упрощает представление этих сложных отношений и работу с ними в коде. Фрагмент кода из класса `Entry`, приведенный в листинге 12.7, показывает, как применять набор `Set` для сохранения объектов `Attachment` в каждом сообщении.

Листинг 12.7. Использование Set для представления отношений между объектами предметной области

```
package com.apress.prospring3.springblog.domain;
// Операторы импорта опущены.
public class Entry extends AbstractBlogPosting {
   private Set<Attachment> attachments = new HashSet<Attachment>(0);
   public Set<Attachment> getAttachments() {
      return this.attachments;
   }
   public void setAttachments(Set<Attachment> attachments) {
      this.attachments = attachments;
   }
   // Остальной код не показан.
}
```

Вместо использования для моделирования отношений дополнительных объектов мы применяем простой набор `Set`, чтобы смоделировать отношение "один ко многим". Помимо сокращения объема кода, этот метод предотвращает засорение DOM-модели ненужными классами и позволяет использовать хорошо знакомые концепции Java, такие как `Iterator` для перемещения по отношениям.

Итоги по объектной модели предметной области

В этом разделе мы рассмотрели DOM-модель для приложения SpringBlog и уделили некоторое время обсуждению основ моделирования и реализации объектов предметной области. Вне всяких сомнений, область данной темы значительно шире, чем мы раскрыли здесь. В действительности доступен целый ряд книг, посвященных этим вопросам (в их число входит книга *Предметно-ориентированное проектирование: структуризация сложных программных систем* Эрика Эванса (ИД "Вильямс", 2011 г.)). Здесь мы лишь слегка коснулись поверхности, а основное внимание было уделено причинам построения DOM-модели, аспектам, на которых следует сосредоточиться при ее создании, и на общих вопросах, связанных с приложением SpringBlog.

Хотя вполне допускается строить приложения без определения DOM-модели, наш опыт позволяет утверждать, что выделение времени на создание DOM-модели приводит к снижению сложности, уменьшению затрат на сопровождение и сокращению количества ошибок.

Проектирование и построение уровня доступа к данным

После обсуждения DOM давайте перейдем к рассмотрению уровня, который выделен для выполнения всего взаимодействия с базой данных и трансформации извлекаемых данных для DOM-модели с целью обработки уровнем обслуживания.

Никто не ставит под сомнение необходимость в наличии уровня доступа к данным, т.к. логика доступа к данным всегда является сложной и критичной в плане производительности. Например, в некоторых случаях, когда производительность доступа к данным является очень важной, уровень доступа к данным должен иметь возможность запускать настроенные собственные запросы в базе данных, увеличивая производительности извлечения данных. Кроме того, на уровне доступа к данным должны решаться и такие вопросы, как управление параллелизмом (к примеру, оптимистическая/пессимистическая блокировка), кеширование, аудит данных и удовлетворение требований безопасности.

Как правило, при реализации уровня доступа к данным в JEE-приложениях широко применяется шаблон DAO, в котором ответственность DAO состоит в инкапсуляции лежащих в основе технологий постоянства (JDBC, Hibernate и т.д.). Однако поскольку в наши дни инфраструктуры постоянства обрели зрелость и в связи с ростом использования стандарта JPA и его `EntityManager`, которые уже скрывают поставщик постоянства от разработчика, обычной практикой становится полное устранение уровня DAO. Вместо этого поставщик постоянства Service Provider Interface (SPI) внедряется напрямую в уровень обслуживания при извлечении данных. На рис. 12.3 показана диаграмма, в которой используется DAO. (Например, в случае JDBC логика доступа к данным по-прежнему предусматривает написание кода множества запросов, и этот код лучше поместить в класс DAO для сокрытия уровня обслуживания от SQL-операторов.)

На рис. 12.4 приведена другая диаграмма, в рамках которой поставщики постоянства (например, `EntityManager` в JPA, интерфейсы отображателей в MyBatis) внедряются напрямую в уровень обслуживания.

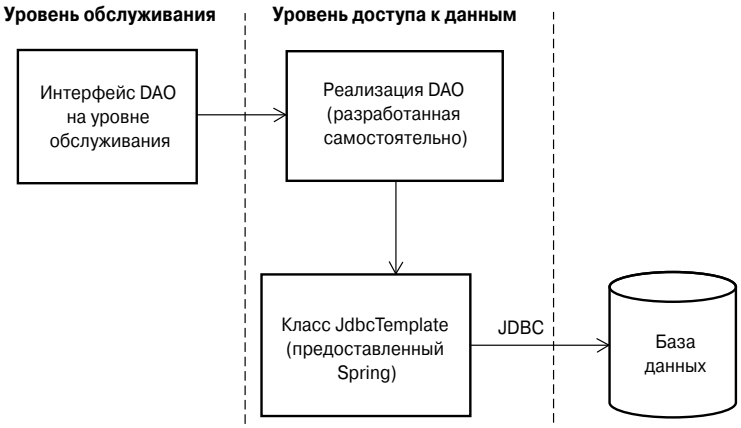


Рис. 12.3. Проектирование приложений с использованием шаблона DAO

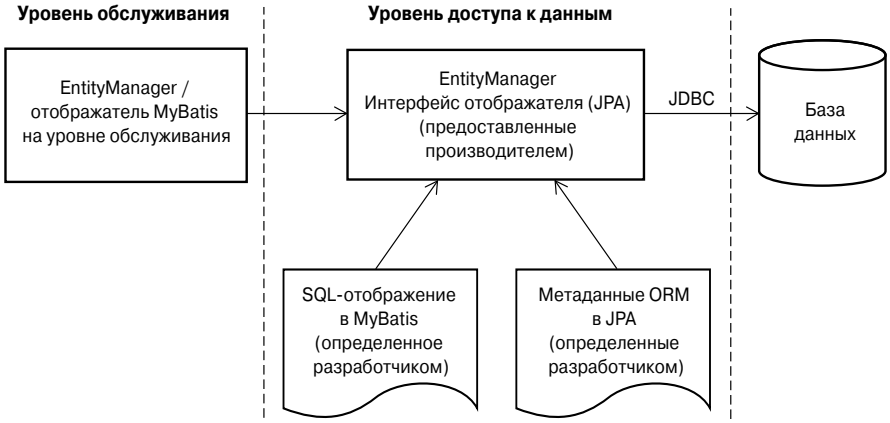


Рис. 12.4. Проектирование приложений с использованием поставщиков постоянства

Практические соображения по поводу проектирования

Благодаря эволюции и становлению различных технологий постоянства и гибкой интеграции Spring с этими технологиями, задача проектирования и построения уровня доступа к данным в наши дни стала намного проще. На практике самой сложной и отнимающей время задачей является определение отображений между структурой лежащей в основе базы данных и DOM-моделью в приложении. Однако при создании уровня доступа к данным вы должны принимать во внимание несколько практических соображений, которые помогут строить более простую в использовании и расширении логику доступа к данным.

Объекты предметной области или объекты передачи данных?

В большинстве ситуаций полагаться на объекты передачи данных для перемещения данных между различными уровнями не нужно. Как было указано ранее в главе, в наши дни все популярные технологии постоянства и сама платформа JEE поддерживают применение POJO в качестве лежащей в основе DOM-модели, делая возможным взаимодействие с уровнем обслуживания, уровнем презентаций и уровнем доступа к данным с использованием объектов предметной области напрямую.

Поскольку все объекты предметной области теперь являются POJO, они могут легко поддерживаться инфраструктурами представлений пользовательского интерфейса (среди которых Spring MVC, JSF, Adobe Flex и многие другие) для непосредственного взаимодействия с уровнем обслуживания через объекты предметной области.

Тем не менее, в некоторых случаях шаблон “Объект передачи данных” по-прежнему стоит принимать во внимание. Например, пусть объект предметной области содержит множество атрибутов в виде крупных текстовых данных, которые не обязательно должны передаваться пользовательскому интерфейсу в каждом запросе. В такой ситуации стоит подумать о создании объекта значения, который будет хранить только обязательные атрибуты для соответствующего пользовательского интерфейса, чтобы минимизировать передачу данных между уровнем презентаций и уровнем обслуживания.

Интерфейсы DAO

Возьмите себе за правило всегда определять объекты DAO (если вы решили придерживаться шаблона DAO) в терминах интерфейсов, а не классов. При написании в служебных объектах кода, работающего с объектами DAO, кодируйте с применением

интерфейсов, а не классов реализации. Помните, что при использовании Spring передавать экземпляр соответствующей реализации DAO на уровень обслуживания очень легко, поэтому применение интерфейсов для объектов DAO требует очень небольшого дополнительного кодирования.

Уровень детализации DAO

Принимая решение о структурировании интерфейсов DAO, однозначно избегайте создания одного DAO на объект предметной области и одного DAO на таблицу. Иногда такие структуры появляются естественным образом в результате продуманного проектирования, но не следует полагать, что использование одной из них обязательно даст лучшее решение.

Крупная проблема, которую мы часто видим в проектах — это проблема "один DAO на таблицу". Определяя подобного рода структуру, вы останетесь с объектами DAO, представляющими таблицы соединения, которые служат ни для каких других целей, кроме соединения двух таблиц в отношении "многие ко многим". Вдобавок вы будете часто обнаруживать, что передаете одиночный объект предметной области множеству различных объектов DAO для сохранения данных.

Это классический пример разрешения базе данных управлять проектированием уровня DAO. Мы считаем его неудачным практическим подходом. Назначение DAO состоит в том, чтобы отображать объекты предметной области на базу данных и наоборот. Поскольку основная часть приложения взаимодействует с объектами предметной области, а не с базой данных, имеет смысл позволить DOM управлять проектированием базы данных. Дайте возможность объектам DAO скрыть сложность отображения данных в объектах предметной области на базу данных; это их работа. Вы пытаетесь избежать ситуации, при которой сохранение объекта предметной области требует взаимодействия со многими различными объектами DAO. Такие ситуации возникают естественным образом, например, когда объект предметной области имеет ссылку на другой объект предметной области отличающегося типа, и оба они модифицированы, потому должны быть сохранены. В этом случае такую логику можно инкапсулировать в уровне обслуживания, однако вы не должны порождать эту проблему искусственно.

Таким образом, может возникнуть вопрос: позволять ли DOM-модели управлять проектированием объектов DAO? Ответов на этот вопрос два: да и нет. Да, потому что объект DAO предназначен для получения данных DOM из постоянного хранилища данных, поэтому имеет смысл позволить DOM-модели действовать в качестве драйвера. И нет, т.к. необдуманное создание объекта DAO для каждого объекта предметной области приводит к ситуации, когда сохранение логической единицы данных требует обращения ко многим разным объектам DAO. Вспомните ранее приводившийся пример с объектами `Order` и `OrderLine`, которые созданы из объектов `Cart` и `CartItem`. Поскольку маловероятно, что вы захотите сохранять или извлекать объекты `OrderLine`, не делая то же самое для объекта `Order`, имеет смысл инкапсулировать логику постоянства для обоих объектов предметной области в единственный объект `OrderDao`, нежели создавать объекты `OrderDao` и `OrderLineDao`.

Итоги по уровню доступа к данным

Создание уровня доступа к данным для приложения предоставляет остальным компонентам приложения стандартный механизм сохранения и извлечения данных. Вы заметите, что без этого уровня код доступа к данным распространится по всему приложению, часто вызывая появление дублированных фрагментов, которые сложно сопровождать. В долгосрочной перспективе этот плохо управляемый код неизбежно приведет к возникновению ошибок и головной боли у разработчиков.

В зависимости от технологии реализации уровня доступа к данным, в большинстве случаев использовать шаблон DAO больше не понадобится. Многие популярные технологии и стандарты постоянства довольно хорошо скрывают все детали доступа к базам данных. Таким образом, рекомендуется внедрять эти поставщики постоянства напрямую в уровень обслуживания. Это упростит архитектуру приложения, а также сделает код более легким для отладки и сопровождения.

В случае если вы решили применять DAO либо это является стандартом, принятым в команде или компании, вы должны всегда определять объекты DAO как интерфейсы и затем реализовать их, используя выбранную технологию доступа к данным. В Spring работа с интерфейсами тривиальна, и вы можете легко предоставить конкретные реализации интерфейсов DAO другим компонентам внутри приложения.

В этом разделе мы рассмотрели некоторые из главных вопросов проектирования, возникающих при построении уровня доступа к данным для приложения. В реальности большая часть сложности на уровне доступа к данным привносится реализацией, а не проектированием. Дополнительные сведения о доступе к данным были приведены в главах 8–11.

Проектирование уровня обслуживания

На этот момент в обсуждении проектирования приложения у нас есть способ представления данных в предметной области, так что мы можем манипулировать ими в коде, в также способ сохранения этих данных в базе с последующим их извлечением. Тем не менее, мы пока что не делаем что-то особенное с этим данными. Если только приложение не упрощено преднамеренно, скорее всего, какая-нибудь логика будет реализована. Ранее мы обсуждали случаи, когда нужно было инкапсулировать логику внутри объектов предметной области. В этом разделе мы рассмотрим предоставление уровня служебных объектов для обеспечения стандартного интерфейса остальной части логики приложения.

Для чего нужен уровень обслуживания?

Как и с вопросом, зачем требуется уровень доступа к данным, ответ на этот вопрос прост и станет ясным, как только вы реализуете несколько приложений без уровня обслуживания. Если не собрать всю бизнес-логику в одном месте, она рассредоточится по коду представления, как правило, порождая множество дублированного кода, не говоря уже о получении кода, который лишен четко определенных границ ответственности. Оставив в стороне проблему дублированного кода, невозможность определить четкие границы между кодом с разной ответственностью часто приводит к получению кода, который трудно отлаживать и сопровождать, т.к. в нем нелегко найти местоположение той или иной функции.

Хорошо определенный уровень обслуживания действует как своего рода шлюз в приложение, предоставляя коду представления простой унифицированный способ получения бизнес-логики. Качественный уровень обслуживания также позволяет определить, что в действительности делает приложение, и какая логика может быть представлена пользователю.

Существенный недостаток отсутствия уровня обслуживания проявляется в ситуации, когда возникает необходимость иметь два вида пользовательского интерфейса для одной и той же логики. Возможно, вы построили веб-приложение, но теперь хотите предложить его настольный вариант пользователям, которые часто с ним работают. Если код бизнес-логики встроен в код веб-представления, понадобится либо вынести этот код на уровень презентаций, что потребует значительных усилий по переделке и

тестированию, либо просто воспроизвести код бизнес-логики заново, на этот раз встроив его в код полнофункционального клиента (например, Swing, RCP и т.п.).

В наши дни кроме пользовательских интерфейсов другим системам могут быть доступны также и бизнес-службы. Например, приложение может предоставлять информацию о биржевых котировках в реальном времени своим бизнес-партнерам. Скорее всего, понадобится поддерживать доступ партнерами к данным реального времени через веб-службы (простые веб-службы, RESTful-WS и т.п.). Кроме того, при обработке данных в пакетах (например, из файла) логика манипулирования данными не должна сильно отличаться от логики, обслуживающей ввод данных пользователем через веб-интерфейс. Централизованный уровень обслуживания как входная точка для данных, поступающих со всех возможных источников, становится критически важной частью при сопровождении приложения.

Проектирование бизнес-интерфейсов

Как и для большинства компонентов в приложении, вы должны начать с определения набора интерфейсов для служебных объектов приложения. Любой код, взаимодействующий с уровнем обслуживания, должен делать это через упомянутые интерфейсы. Для компонентов, управляемых Spring, реализации можно поставлять с использованием внедрения зависимостей. Если нужно поддерживать компоненты, не управляемые Spring, может понадобиться создать простой класс фабрики, чтобы обеспечить возможность нахождения реализаций.

Зависимости уровня обслуживания

Как и со всеми интерфейсами, о которых шла речь ранее, следует избегать определения зависимостей в интерфейсах служебных объектов. Служебный объект должен быть полностью независимым от реализации. Хорошо определенный интерфейс служебного объекта имеет только методы, которые обслуживают бизнес-функции. Избегайте делать видимыми типы из уровня доступа к данным через служебные объекты. Служебные объекты должны полностью изолировать код представления от уровня доступа к данным. Например, один из служебных объектов может иметь доступ к своим данным с использованием веб-служб или очереди JMS. Хороший способ удостовериться, что интерфейс служебного объекта максимально возможно доступен, заключается в проверке того, что типы возврата и аргументов не связывают код представления ни с чем, кроме DOM-модели. Объекты предметной области должны передаваться через все уровни приложения, но другие компоненты, такие как объекты DAO, должны оставаться на собственном уровне.

Уровень детализации служебных объектов

При проектировании объектов, требуемых на уровне обслуживания, постарайтесь связать их с возможными сценариями использования в бизнес-требованиях.

Общепринятая практика предполагает группирование логики, которая манипулирует тесно связанными объектами предметной области, в один и тот же интерфейс служебного объекта. Например, в приложении SpringBlog уровень обслуживания состоит из интерфейсов, перечисленных в табл. 12.1.

Имейте в виду, что при проектировании приложений не существует однозначно правильного или однозначно неправильного подхода. Тем не менее, различают лучшие и худшие проектные решения. Хорошее проектное решение должно быть простым в документировании и понятным как техническому персоналу, так и бизнес-аналитику. Вдобавок, всегда думайте о гибкости, сопровождении и отслеживании. Наконец, удостоверьтесь, что уровень обслуживания не привязан к какому-то специальному типу потребителя.

Таблица 12.1. Интерфейсы уровня обслуживания в приложении SpringBlog

Служебный интерфейс	Назначение	Связанные объекты предметной области
`EntryService`	Предоставляет службы для записей блога: • разнообразные функции поиска; • извлечение отдельной записи; • операции обновления; • операции, связанные с вложениями.	`Entry` `Attachment`
`CommentService`	Предоставляет службы для комментариев блога: • извлечение всех вложений для отдельной записи; • операции обновления; • операции, связанные с вложениями.	`Comment` `Attachment`
`UserService`	Предоставляет службы для средств безопасности: • управление пользователями; • назначение ролей.	`AppUser` `Role`
`HousekeepingService`	Предоставляет вспомогательные службы: • удаление старых записей хронологии для сообщений и комментариев блога.	Хронологии для объектов `Entry` и `Comment`

Итоги по уровню обслуживания

В этом разделе мы обсудили вопросы, которые должны приниматься во внимание при проектировании и построении уровня обслуживания для Spring-приложения. Затем для уровня обслуживания приложения SpringBlog мы продемонстрировали, как проектировать интерфейсы внутри этого уровня, группируя операции для тесно связанных объектов предметной области в один и тот же служебный интерфейс.

Резюме

В этой главе мы взглянули на множество проблем, связанных с традиционными приложениями, и показали, как Spring может помочь в решении этих проблем. Мы обсудили целый ряд проблем, которые возникают во время проектирования приложений, и посмотрели на практические способы эффективного их решения. На протяжении всей главы мы показывали, каким образом обсуждаемые здесь приемы использовались при проектировании приложения SpringBlog.

В этой главе мы лишь слегка коснулись поверхности процесса проектирования приложений в целом, но вместе с тем рассмотрели ряд основных соображений, специфичных для Spring-приложений, и проблемы, которые можно легко решить при использовании Spring. Более полное обсуждение проектного решения для приложения SpringBlog можно найти в главе 23.

В следующей главе мы представим детальное изложение поддержки транзакций, включая полноценные примеры локальных и распределенных транзакций и полезные советы по тестированию транзакционных методов.

ГЛАВА 13

Управление транзакциями

Транзакции являются одной из самых важных частей при построении надежного корпоративного приложения. Наиболее общим типом транзакции является операция базы данных. В рамках типовой операции обновления базы данных начинается транзакция базы данных, данные обновляются, а затем транзакция фиксируется или откатывается в зависимости от результата выполнения операции базы данных. Однако во многих случаях, в зависимости от требований приложения и ресурсов серверной части, с которыми должно взаимодействовать приложение (СУРБД, промежуточное ПО для обработки сообщений, система ERP и т.д.), управление транзакциями может оказаться намного более сложным.

В ранние дни разработки Java-приложений (после появления JDBC, но перед тем, как стал доступным стандарт JEE или платформа для создания приложений, подобная Spring), разработчики программно контролировали и управляли транзакциями внутри кода приложения. Когда стал доступным стандарт JEE, точнее — стандарт EJB, разработчики получили возможность использовать транзакции, управляемые контейнером (container-managed transaction — CMT), для управления транзакциями декларативным путем. Но сложное объявление транзакций в дескрипторе развертывания EJB было трудно поддерживать, и оно привносило ненужную сложность в обработку транзакций. Некоторые разработчики предпочитают иметь больший контроль над транзакцией и работают с транзакциями, управляемыми бином (bean-managed transaction — BMT), чтобы управлять ими программным путем. Тем не менее, сложность программирования с использованием API-интерфейса транзакций Java (Java Transaction API — JTA) также препятствует высокой продуктивности разработчиков.

Как обсуждалось в главах 6 и 7, посвященным АОП, управление транзакциями является сквозной функциональностью и должно кодироваться внутри бизнес-логики. Самый подходящий способ реализации управления транзакциями — дать возможность разработчикам определить требования к транзакциям декларативным путем и позволить такой инфраструктуре, как Spring, JEE или АОП, самостоятельно связать логику управления транзакциями. В этой главе мы посмотрим, как Spring может упростить реализацию логики обработки транзакций. Платформа Spring предлагает поддержку как декларативного, так и программного способа управления транзакциями.

В Spring имеется великолепная поддержка декларативных транзакций, а это значит, что загромождать бизнес-логику кодом управления транзакциями не придется. Все, что потребуется сделать — это объявить методы (внутри классов или на нужных уровнях), которые должны принимать участие в транзакции, вместе с конфигурационными дета-

лями транзакции, а Spring позаботится о поддержке управления транзакциями. В частности, в этой главе рассматриваются следующие темы.

- *Уровень абстракции транзакций Spring.* Мы обсудим базовые классы абстракции транзакций Spring и объясним, как использовать эти классы для управления свойствами транзакций.
- *Декларативное управление транзакциями.* Мы покажем, как использовать Spring для реализации декларативного управления транзакциями с помощью только простых Java-объектов. Мы предложим примеры декларативного управления транзакциями, в которых применяются XML-файлы конфигурации, а также Java-аннотации.
- *Программное управление транзакциями.* Несмотря на то что программное управление транзакциями используется очень редко, мы покажем, как работать с предоставляемым Spring классом `TransactionTemplate`, который дает полный контроль над кодом управления транзакциями.
- *Глобальные транзакции с помощью JTA.* Для глобальных транзакций, которые должны охватывать несколько ресурсов серверной части, мы представим пример конфигурирования и реализации таких транзакций в Spring с использованием JTA.

Исследование уровня абстракции транзакций Spring

При разработке приложения, независимо от того, используется Spring или нет, вы должны сделать фундаментальный выбор, какие транзакции использовать — локальные или глобальные. Локальные транзакции специфичны для отдельного транзакционного ресурса (JDBC-подключение, к примеру), тогда как глобальные транзакции управляются контейнером и могут охватывать множество транзакционных ресурсов.

Типы транзакций

Локальные транзакции просты в управлении, и если все операции в приложении должны взаимодействовать с только одним транзакционным ресурсом (таким как транзакция JDBC), то использования локальных транзакций будет вполне достаточно. Однако если не применяется какая-нибудь платформа для разработки приложений, такая как Spring, понадобится написать много кода управления транзакциями, и в случае, когда в будущем область действия транзакции потребует расширения, придется отбросить код управления локальными транзакциями и переписать его для использования глобальных транзакций.

В мире Java глобальные транзакции были реализованы с помощью API-интерфейса транзакций Java (Java Transaction API — JTA). При таком сценарии совместимый с JTA диспетчер транзакций подключается к множеству транзакционных ресурсов через соответствующие диспетчеры ресурсов, которые способны взаимодействовать с диспетчером транзакций по протоколу XA (открытый стандарт, определяющий распределенные транзакции), а механизм двухфазной фиксации (2 Phase Commit — 2PC) обеспечивает корректное обновление или откат всех участвующих источников данных серверной части. Если любой из ресурсов серверной части дает сбой, будет выполнен откат всей транзакции, и, следовательно, откат будет совершен и для обновлений других ресурсов.

На рис. 13.1 показано высокоуровневое представление глобальных транзакций, реализуемых с помощью JTA.

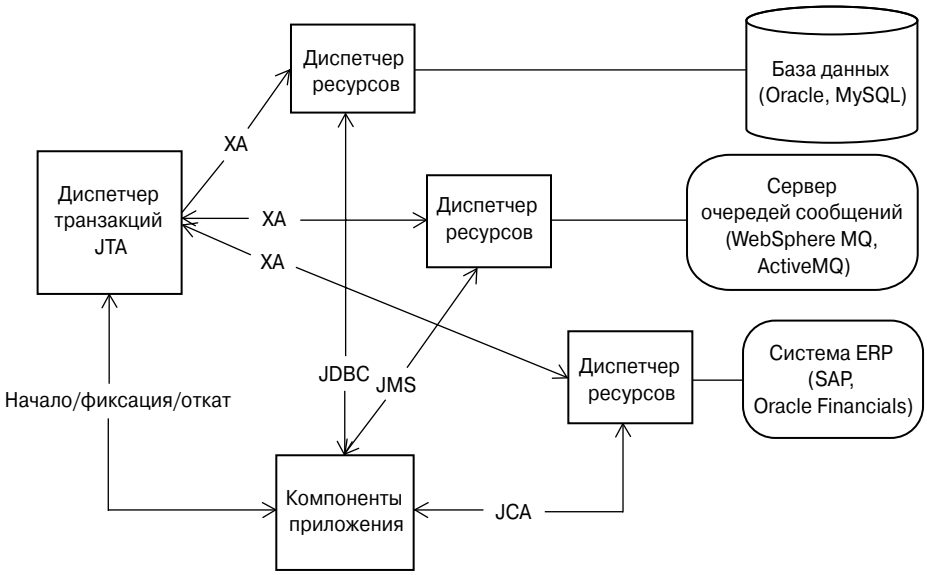


Рис. 13.1. Обзор глобальных транзакций, реализуемых с помощью JTA

На рис. 13.1 видно, что в глобальной транзакции (также в общем случае называемой *распределенной транзакцией*) принимают участие четыре основных части. Первый участник — это ресурс серверной части, такой как СУРБД, промежуточное ПО для обмена сообщениями, система планирования ресурсов предприятия (enterprise resource planning — ERP) и т.д.

Вторым участником является диспетчер ресурсов, который в общем случае предоставляется производителем ресурса серверной части и отвечает за взаимодействие с этим ресурсом. Например, при подключении к базе данных MySQL мы должны взаимодействовать с классом `MysqlXADataSource`, предоставленным Java-коннектором MySQL. Другие ресурсы серверной части (MQ, ERP и т.д.) также предоставляют свои диспетчеры ресурсов.

Третий участник — это диспетчер транзакций JTA, который отвечает за управление, координацию и синхронизацию состояния транзакции со всеми диспетчерами ресурсов, принимающих участие в транзакции. При этом будет использоваться протокол XA, который представляет собой открытый стандарт, широко используемый для обработки распределенных транзакций. Диспетчер транзакций JTA также поддерживает 2PC, поэтому все изменения будут фиксироваться вместе, и если обновление любого ресурса дает сбой, производится откат всей транзакции, в результате чего ни один из ресурсов обновляться не будет. Весь механизм целиком описан в спецификации Java Transaction Service (JTS).

Последним, четвертым, участником является приложение. Либо само приложение, либо контейнер или инфраструктура Spring, в рамках которой запускается приложение, будет управлять транзакцией (начинать, фиксировать, открывать транзакцию и т.п.). В то же самое время приложение будет взаимодействовать с ресурсами серверной части через различные стандарты, определенные JEE. Как показано на рис. 13.1, приложение подключается к СУРБД через JDBC, к MQ через JMS и к системе ERP через JCA (Java Connector Architecture — архитектура Java-коннекторов).

JTA поддерживается всеми полномасштабными серверами приложений, совместимыми с JEE (например, JBoss, WebSphere, WebLogic, GlassFish и т.д.), внутри которых

транзакция доступна через поиск JNDI. Как и для автономных приложений или веб-контейнеров (Tomcat, Jetty и т.д.), существуют также коммерческие решения и решения с открытым кодом, предлагающие поддержку JTA/XA в упомянутых средах (например, Atomikos, JOTM, Bitronix и др.).

Реализации интерфейса `PlatformTransactionManager`

В Spring интерфейс `PlatformTransactionManager` использует интерфейсы `TransactionDefinition` и `TransactionStatus` для создания и управления транзакциями. Действительная реализация этих интерфейсов требует детальных знаний диспетчера транзакций. На рис. 13.2 показаны реализации `PlatformTransactionManager` в версии Spring 3.1.

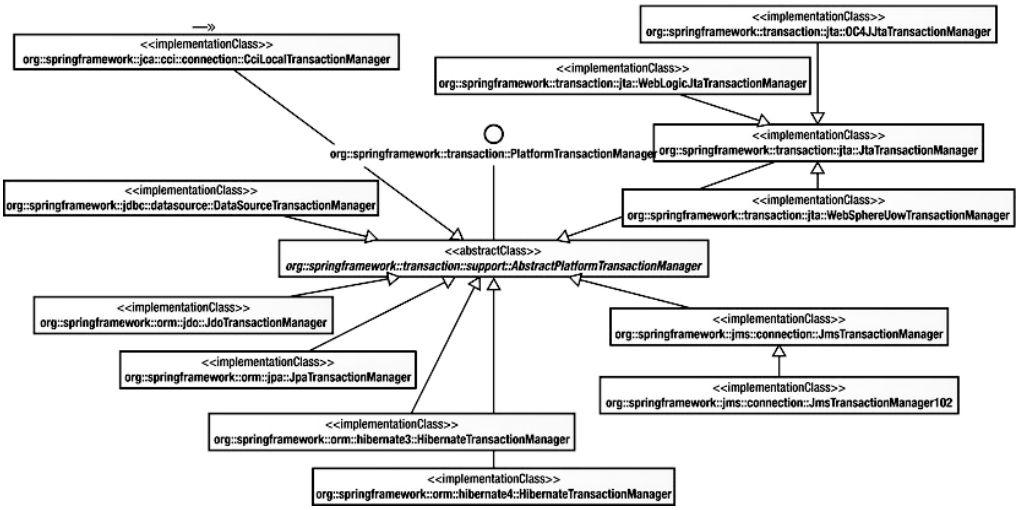


Рис. 13.2. Реализации `PlatformTransactionManager` в Spring 3.1

Spring предлагает широкий набор реализаций интерфейса `PlatformTransactionManager`. Класс `CciLocalTransactionManager` поддерживает JEE JCA и CCI (Common Client Interface — интерфейс общего клиента). Класс `DataSourceTransactionManager` предназначен для обобщенных JDBC-подключений.

Для объектно-реляционного отображения предусмотрен ряд реализаций, включая JDO (класс `JdoTransactionManager`), JPA (класс `JpaTransactionManager`), а также Hibernate 3 и Hibernate 4 (класс `HibernateTransactionManager` с разными именами пакетов). Для JMS доступны реализации для JMS 1.1 и более новых версий (класс `JmsTransactionManager`) и JMS 1.0.2 (класс `JmsTransactionManager102`). Для JTA имеется обобщенный класс реализации `JtaTransactionManager`.

В Spring также предоставляется множество классов диспетчеров транзакций JTA, специфичных для конкретных серверов приложений. Эти классы обеспечивают встроенную поддержку для WebSphere (класс `WebSphereUowTransactionManager`), WebLogic (класс `WebLogicJtaTransactionManager`) и Oracle OC4J (класс `OC4JJtaTransaction Manager`).

Анализ свойств транзакций

В этом разделе мы обсудим свойства транзакций, поддерживаемые Spring, уделяя основное внимание взаимодействию с СУРБД в качестве ресурса серверной части.

Транзакции обладают четырьмя хорошо известными свойствами ACID (atomicity, consistency, isolation, durability — атомарность, согласованность, изолированность, долговечность), и за поддержку этих аспектов транзакции отвечают транзакционные ресурсы. Управлять атомарностью, согласованностью и долговечностью транзакции нет возможности, но можно управлять ее распространением и тайм-аутом, а также конфигурировать, должна ли транзакция предназначаться только для чтения, и задавать уровень изоляции.

Все эти параметры инкапсулированы Spring в интерфейсе `TransactionDefinition`. Этот интерфейс используется в ключевом интерфейсе поддержки транзакций — `PlatformTransactionManager`, реализации которого выполняют управление транзакциями на конкретной платформе, такой как JDBC или JTA.

Основной метод, `PlatformTransactionManager.getTransaction()`, получает интерфейс `TransactionDefinition` в качестве аргумента и возвращает интерфейс `TransactionStatus`. Интерфейс `TransactionStatus` применяется для управления выполнением транзакции, конкретнее — для установки результата транзакции и проверки, завершена ли транзакция, или является ли транзакция новой.

Интерфейс `TransactionDefinition`

Как упоминалось ранее, интерфейс `TransactionDefinition` управляет свойствами транзакции. Давайте рассмотрим этот интерфейс более детально (листинг 13.1) и опишем его методы.

Листинг 13.1. Интерфейс `TransactionDefinition`

```
package org.springframework.transaction;

import java.sql.Connection;

public interface TransactionDefinition {

    // Объявления переменных не показаны.

    int getPropagationBehavior();
    int getIsolationLevel();
    int getTimeout();

    boolean isReadOnly();
    String getName();
}
```

Простыми и очевидными методами этого интерфейса являются `getTimeout()`, возвращающий промежуток времени (в секундах), в течение которого транзакция должна быть завершена, и `isReadOnly()`, указывающий, предназначена ли транзакция только для чтения. Реализация диспетчера транзакций может использовать эти значения для оптимизации выполнения и проверки, что транзакция выполняет только операции чтения. Метод `getName()` возвращает имя транзакции.

Оставшиеся два метода, `getPropagationBehavior()` и `getIsolationLevel()`, заслуживают более подробного рассмотрения. Мы начнем с метода `getIsolationLevel()`, который управляет тем, какие изменения в данных видны другим транзакциям.

В табл. 13.1 перечислены уровни изоляции транзакции вместе с описаниями, какие изменения, сделанные текущей транзакцией, доступны другим транзакциям.

Таблица 13.1. Уровни изоляции транзакции

Уровень изоляции	Описание
TransactionDefinition. ISOLATION_DEFAULT	Стандартный уровень изоляции лежащего в основе хранилища данных
TransactionDefinition. ISOLATION_READ_UNCOMMITTED	Самый низкий уровень изоляции; это вообще трудно назвать транзакцией, поскольку при таком уровне разрешено видеть данные, модифицированные другими незафиксированными транзакциями
TransactionDefinition. ISOLATION_READ_COMMITTED	Стандартный уровень изоляции в большинстве баз данных; он гарантирует, что другие транзакции не имеют возможности читать данные, которые не были зафиксированы текущей транзакцией. Однако данные, прочитанные одной транзакцией, могут быть обновлены другими транзакциями
TransactionDefinition. ISOLATION_REPEATABLE_READ	Более строгий уровень, чем ISOLATION_READ_COMMITED; он гарантирует, что после выборки данных можно произвести выборку, по крайней мере, того же набора данных снова. Если другие транзакции вставили новые данные, эти новые данные могут быть извлечены
TransactionDefinition. ISOLATION_SERIALIZABLE	Наиболее дорогостоящий и надежный уровень изоляции; все транзакции трактуются как выполняемые последовательно, друг за другом

Выбор подходящего уровня изоляции очень важен для согласованности данных, но оказывает сильное влияние на производительность. Самый высокий уровень изоляции, TransactionDefinition.ISOLATION_SERIALIZABLE, особенно дорогостоящ в обслуживании. Метод getPropagationBehavior() определяет, что произойдет с транзакционным вызовом в зависимости от того, существует ли активная транзакция. Значения, с которыми работает этот метод, описаны в табл. 13.2.

Таблица 13.2. Значения, описывающие поведение распространения

Поведение распространения	Описание
TransactionDefinition. PROPAGATION_REQUIRED	Поддерживает транзакцию, если она уже существует. Если транзакций нет, начинается новая транзакция
TransactionDefinition. PROPAGATION_SUPPORTS	Поддерживает транзакцию, если она уже существует. Если транзакций нет, выполнение происходит без транзакций
TransactionDefinition. PROPAGATION_MANDATORY	Поддерживает транзакцию, если она уже существует. Если нет активной транзакции, генерируется исключение
TransactionDefinition. PROPAGATION_REQUIRES_NEW	Всегда начинает новую транзакцию. Если активная транзакция уже существует, она приостанавливается
TransactionDefinition. PROPAGATION_NOT_SUPPORTED	Не поддерживает выполнение с активной транзакцией. Всегда выполняется без транзакций и приостанавливает любые существующие транзакции
TransactionDefinition. PROPAGATION_NEVER	Всегда выполняется без транзакций, даже если существует активная транзакция. Если активная транзакция существует, генерируется исключение
TransactionDefinition. PROPAGATION_NESTED	Выполняется во вложенной транзакции, если существует активная транзакция. Если активной транзакции нет, поведение соответствует TransactionDefinition. PROPAGATION_REQUIRED

Интерфейс TransactionStatus

Интерфейс TransactionStatus, показанный в листинге 13.2, позволяет диспетчеру транзакций управлять выполнением транзакции. Код может проверить, является ли транзакция новой или предназначенной только для чтения, и может инициировать откат.

Листинг 13.2. Объявление TransactionStatus

```
package org.springframework.transaction;
public interface TransactionStatus extends SavepointManager {
    boolean isNewTransaction();
    boolean hasSavepoint();
    void setRollbackOnly();
    boolean isRollbackOnly();
    void flush();
    boolean isCompleted();
}
```

Методы интерфейса TransactionStatus довольно очевидны; наиболее примечательным из них является setRollbackOnly(), который вызывает откат и завершает активную транзакцию.

Метод hasSavePoint() возвращает признак наличия точки сохранения внутри транзакции (т.е. эта транзакция была создана как вложенная на основе точки сохранения). Метод flush() сбрасывает сеанс в хранилище данных, если это применимо (например, когда используется Hibernate). Метод isCompleted() возвращает признак завершения транзакции (т.е. фиксация или откат).

Модель данных и инфраструктура для кода примеров

В этом разделе представлен обзор модели данных и инфраструктуры, которые будут использоваться в примерах управления транзакциями.

В этой главе мы будем применять JPA с Hibernate в качестве уровня постоянства для реализации логики доступа к данным. Вдобавок, для упрощения разработки основных операций базы данных будет также использоваться проект Spring Data JPA и его абстракция репозитория.

Создание простого служебного проекта Spring JPA с зависимостями

Давайте начнем с создания проекта. Поскольку используется JPA, создайте новый шаблонный проект Spring, указав вариант Simple Spring JPA Utility Project (Простой служебный проект Spring JPA). Введите детальные сведения о проекте, как показано на рис. 13.3.

После создания к проекту необходимо добавить обязательные зависимости, требуемые для примеров этой главы. Все эти дополнительные зависимости перечислены в табл. 13.3.

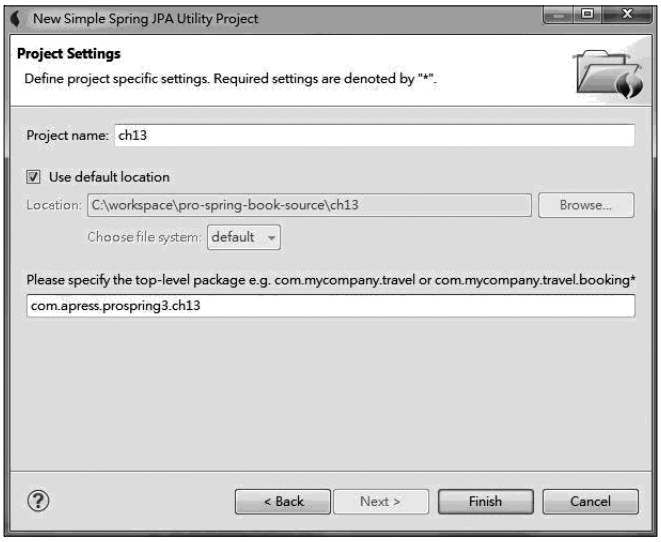


Рис. 13.3. Создание простого служебного проекта Spring JPA в STS

Таблица 13.3. Зависимости Maven для Spring Data JPA

Идентификатор группы	Идентификатор артефакта	Версия	Описание
`org.springframework.data`	`spring-data-jpa`	1.0.1.RELEASE	Библиотека Spring Data JPA
`org.springframework`	`spring-aspects`	3.1.0.RELEASE	Расширенные функциональные средства АОП в Spring, от которых зависит Spring Data JPA
`joda-time`	`joda-time`	2.0	API-интерфейс для работы с датой и временем под названием Joda-Time (`http://joda-time.sourceforge.net/`), который используется в Spring Data JPA
`joda-time`	`joda-time-hibernate`	1.3	Библиотека Joda-Time для интеграции с Hibernate, обеспечивающая сохранение типов данных даты и времени
`com.google.guava`	`guava`	10.0.1	Полезные вспомогательные классы
`org.aspectj`	`aspectjrt`	1.6.11	Библиотека AspectJ. Требуется в примерах использования пространства имен `aop` для объявления транзакций
`org.aspectj`	`aspectjweaver`	1.6.11	Библиотека связывания AspectJ
`org.slf4j`	`slf4j-log4j12`	1.6.1	Библиотека, которая соединяет ведение журнала SLF4J с библиотекой log4j

Удостоверьтесь, что используется Spring 3.1 с JDK 6. За дополнительной информацией обращайтесь в приложение А.

Чтобы исследовать поведение кода примера при изменении атрибутов транзакции, мы установим уровень ведения журнала log4j в `DEBUG`. В листинге 13.3 приведено содержимое файла `log4j.properties`.

Листинг 13.3. Включение уровня ведения журнала `DEBUG` в `log4j.properties`

```
log4j.rootCategory=DEBUG, stdout
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d %p [%c] - <%m>%n
log4j.category.org.apache.activemq=ERROR
log4j.category.org.springframework.batch=DEBUG
log4j.category.org.springframework.transaction=INFO
log4j.category.org.hibernate.SQL=DEBUG
```

Модель данных и общие классы

Чтобы максимально упростить примеры, мы будем использовать всего одну таблицу по имени `CONTACT`, которая применялась в главах, посвященных доступу к данным. В листингах 13.4 и 13.5 показаны сценарии создания этой таблицы (`schema.sql`) и наполнения ее тестовыми данными (`test-data.sql`).

Листинг 13.4. Сценарий для создания таблицы

```
DROP TABLE IF EXISTS CONTACT;
CREATE TABLE CONTACT (
     ID INT NOT NULL AUTO_INCREMENT
   , FIRST_NAME VARCHAR(60) NOT NULL
   , LAST_NAME VARCHAR(40) NOT NULL
   , BIRTH_DATE DATE
   , VERSION INT NOT NULL DEFAULT 0
   , UNIQUE UQ_CONTACT_1 (FIRST_NAME, LAST_NAME)
   , PRIMARY KEY (ID)
);
```

Листинг 13.5. Сценарий для наполнения таблицы тестовыми данными

```
insert into contact (first_name, last_name, birth_date)
   values ('Clarence', 'Ho', '1980-07-30');
insert into contact (first_name, last_name, birth_date)
   values ('Scott', 'Tiger', '1990-11-02');
insert into contact (first_name, last_name, birth_date)
   values ('John', 'Smith', '1964-02-28');
```

Сущностный класс также прост; код класса `Contact` приведен в листинге 13.6.

Листинг 13.6. Класс `Contact`

```
package com.apress.prospring3.ch13.domain;

// Операторы импорта опущены.
@Entity
@Table(name = "contact")
```

```
@NamedQueries({
@NamedQuery(name="Contact.findAll", query="select c from Contact c"),
@NamedQuery(name="Contact.countAll", query="select count(c) from Contact c")
})
public class Contact implements Serializable {
   private Long id;
   private int version;
   private String firstName;
   private String lastName;
   private Date birthDate;
   public Contact() {
   }
   @Id
   @GeneratedValue(strategy = IDENTITY)
   @Column(name = "ID")
   public Long getId() {
      return this.id;
   }
   public void setId(Long id) {
      this.id = id;
   }
   @Version
   @Column(name = "VERSION")
   public int getVersion() {
      return this.version;
   }
   public void setVersion(int version) {
      this.version = version;
   }
   @Column(name = "FIRST_NAME")
   public String getFirstName() {
      return this.firstName;
   }
   public void setFirstName(String firstName) {
      this.firstName = firstName;
   }
   @Column(name = "LAST_NAME")
   public String getLastName() {
      return this.lastName;
   }
   public void setLastName(String lastName) {
      this.lastName = lastName;
   }
   @Temporal(TemporalType.DATE)
   @Column(name = "BIRTH_DATE")
   public Date getBirthDate() {
      return this.birthDate;
   }
   public void setBirthDate(Date birthDate) {
      this.birthDate = birthDate;
   }
   public String toString() {
      return "Contact - Id: " + id + ", First name: " + firstName
         + ", Last name: " + lastName + ", Birthday: " + birthDate;
   }
}
```

Для использования абстракции репозитория Spring Data JPA мы также должны определить интерфейс ContactRepository, который расширяет интерфейс CrudRepository<T,ID extends Serializable> из проекта Spring Data Common.

Интерфейс ContactRepository представлен в листинге 13.7.

Листинг 13.7. Интерфейс ContactRepository

```
package com.apress.prospring3.ch13.repository;
import org.springframework.data.repository.CrudRepository;
import com.apress.prospring3.ch13.domain.Contact;
public interface ContactRepository extends CrudRepository<Contact, Long> {
}
```

Как показано в листинге 13.7, никаких дополнительных методов не требуется, поскольку методов, предоставляемых интерфейсом CrudRepositoy, уже достаточно для примеров, рассматриваемых в этой главе.

Наконец, давайте взглянем на интерфейс ContactService, который определяет всю бизнес-логику, относящуюся к сущностному классу Contact.

Интерфейс ContactService приведен в листинге 13.8.

Листинг 13.8. Интерфейс ContactService

```
package com.apress.prospring3.ch13.service;
import java.util.List;
import com.apress.prospring3.ch13.domain.Contact;
public interface ContactService {
   public List<Contact> findAll();
   public Contact findById(Long id);
   public Contact save(Contact contact);
   public long countAll();
}
```

Методы этого интерфейса не требуют дополнительных пояснений. В следующем разделе мы обсудим, как реализовать управление транзакциями различными путями, реализуя интерфейс ContactService.

Декларативные и программные транзакции в Spring

В Spring существуют три варианта управления транзакциями. Два из них предназначены для декларативного управления транзакциями, причем в одном варианте используются Java-аннотации, а в другом — XML-конфигурация. Третий вариант предусматривает программное управление транзакциями. Все три варианта рассматриваются в последующих разделах.

Использование аннотаций для управления транзакциями

В настоящее время использование аннотаций является общепринятым способом определения требований к транзакциям в Spring. Главное достоинство такого подхода состоит в том, что требования к транзакциям вместе с их свойствами (тайм-аут, уровень изоляции, поведение распространения и т.д.) определены внутри кода, упрощая отладку и сопровождение приложения.

Чтобы включить поддержку аннотаций для управления транзакциями в Spring, необходимо добавить дескриптор `<tx:annotation-driven>` в XML-файл конфигурации. Содержимое этого файла (`tx-annotation-app-context.xml`) показано в листинге 13.9.

Листинг 13.9. Конфигурация Spring для поддержки аннотаций при управлении транзакциями

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:context="http://www.springframework.org/schema/context"
   xmlns:jdbc="http://www.springframework.org/schema/jdbc"
   xmlns:tx="http://www.springframework.org/schema/tx"
   xmlns:jpa="http://www.springframework.org/schema/data/jpa"
   xsi:schemaLocation="http://www.springframework.org/schema/jdbc
      http://www.springframework.org/schema/jdbc/spring-jdbc-3.1.xsd
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
      http://www.springframework.org/schema/tx
      http://www.springframework.org/schema/tx/spring-tx-3.1.xsd
      http://www.springframework.org/schema/data/jpa
      http://www.springframework.org/schema/data/jpa/spring-jpa-1.0.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context-3.1.xsd">

   <jdbc:embedded-database id="dataSource" type="H2">
      <jdbc:script location="classpath:schema.sql"/>
      <jdbc:script location="classpath:test-data.sql"/>
   </jdbc:embedded-database>

   <bean id="transactionManager"
         class="org.springframework.orm.jpa.JpaTransactionManager">
      <property name="entityManagerFactory" ref="emf"/>
   </bean>

   <tx:annotation-driven/>

   <bean id="emf"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
      <property name="dataSource" ref="dataSource" />
      <property name="jpaVendorAdapter">
       <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter" />
      </property>
      <property name="packagesToScan" value="com.apress.prospring3.ch13.domain"/>
      <property name="jpaProperties">
         <props>
            <prop key="hibernate.dialect">
               org.hibernate.dialect.H2Dialect
            </prop>
            <prop key="hibernate.max_fetch_depth">3</prop>
            <prop key="hibernate.jdbc.fetch_size">50</prop>
```

```
            <prop key="hibernate.jdbc.batch_size">10</prop>
            <prop key="hibernate.show_sql">true</prop>
        </props>
    </property>
</bean>

<context:component-scan
    base-package="com.apress.prospring3.ch13.service.annotation" />

<jpa:repositories base-package="com.apress.prospring3.ch13.repository"
    entity-manager-factory-ref="emf"
    transaction-manager-ref="transactionManager"/>

</beans>
```

Если вы читали главу 10, то приведенная выше конфигурация должна быть знакомой. Сначала определяется встроенная база данных H2 с указанием сценариев для ее создания и наполнения тестовыми данными. Затем, поскольку используется JPA, определяется бин `JpaTransactionManager`. Дескриптор `<tx:annotation-driven>` указывает, что для управления транзакциями применяются аннотации. После этого определяется бин `EntityManagerFactory`, а за ним задается дескриптор `<context:component-scan>`, предназначенный для сканирования классов уровня обслуживания. Наконец, с помощью дескриптора `<jpa:repositories>` включается абстракция репозитория Spring Data JPA.

Для класса реализации интерфейса `ContactService`, `com.apress.prospring3.ch13.service.annotation.ContactServiceImpl`, мы начинаем с пустых реализаций всех методов интерфейса `ContactService`. В STS это делается очень просто. За дополнительными деталями обращайтесь в главу 8.

Реализуем сначала метод `ContactService.findAll()`. В листинге 13.10 приведен класс `ContactServiceImpl` с реализованным методом `findAll()`.

Листинг 13.10. Класс `ContactServiceImpl` с реализованным методом `findAll()`

```
package com.apress.prospring3.ch13.service.annotation;

// Операторы импорта опущены.

@Service("contactService")
@Repository
@Transactional
public class ContactServiceImpl implements ContactService {
    @Autowired
    private ContactRepository contactRepository;

    @Transactional(readOnly=true)
    public List<Contact> findAll() {
        return Lists.newArrayList(contactRepository.findAll());
    }
}
```

При использовании управления транзакциями на основе аннотаций нам придется иметь дело с единственной аннотацией — `@Transactional`. В листинге 13.10 аннотация `@Transactional` применялась на уровне класса, а это значит, что по умолчанию Spring будет гарантировать существование транзакции перед выполнением каждого метода этого класса. Аннотация `@Transactional` поддерживает несколько атрибутов, с помощью которых можно переопределить ее стандартное поведение. В табл. 13.4 описаны доступные атрибуты вместе с их возможными и стандартными значениями.

Таблица 13.4. Атрибуты для аннотации `@Transactional`

Имя атрибута	Значение по умолчанию	Возможные значения
propagation	Propagation.REQUIRED	Propagation.REQUIRED Propagation.SUPPORTS Propagation.MANDATORY Propagation.REQUIRES_NEW Propagation.NOT_SUPPORTED Propagation.NEVER Propagation.NESTED
isolation	Isolation.DEFAULT (стандартный уровень изоляции лежащего в основе ресурса)	Isolation.DEFAULT Isolation.READ_UNCOMMITTED Isolation.READ_COMMITTED Isolation.REPEATABLE_READ Isolation.SERIALIZABLE
timeout	TransactionDefinition.TIMEOUT_DEFAULT (стандартный тайм-аут транзакции в секундах лежащего в основе ресурса)	Целочисленное значение больше нуля; задает число секунд для тайм-аута транзакции
readOnly	false	true false
rollbackFor	Классы исключений, для которых будет осуществлен откат транзакции	–
rollbackForClassName	Имена классов исключений, для которых будет осуществлен откат транзакции	–
noRollbackFor	Классы исключений, для которых не будет осуществлен откат транзакции	–
noRollbackForClassName	Имена классов исключений, для которых не будет осуществлен откат транзакции	–
value	"" (значение квалификатора для указанной транзакции)	–

Согласно табл. 13.4, аннотация `@Transactional` без атрибутов означает, что распространение транзакции является обязательным, уровень изоляции — стандартным, тайм-аут имеет значение по умолчанию, и установлен режим чтения-записи.

Метод `findAll()` в листинге 13.10 аннотирован посредством `@Transactional(readOnly=true)`. Это переопределяет аннотацию по умолчанию, примененную на уровне класса, оставляя неизмененными все атрибуты, кроме режима только для чтения.

В листинге 13.11 приведена программа для тестирования метода `findAll()`.

Листинг 13.11. Тестирование метода `findAll()`

```
package com.apress.prospring3.ch13;

import java.util.List;
```

```
import org.springframework.context.support.GenericXmlApplicationContext;

import com.apress.prospring3.ch13.domain.Contact;
import com.apress.prospring3.ch13.service.ContactService;

public class TxAnnotationSample {

   public static void main(String[] args) {

      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:tx-annotation-app-context.xml");
      ctx.refresh();

      ContactService contactService = ctx.getBean("contactService",
         ContactService.class);

      List<Contact> contacts = contactService.findAll();

      for (Contact contactTemp: contacts) {
         System.out.println(contactTemp);
      }
   }
}
```

Запуск этой программы дает следующий вывод:

```
DEBUG [org.springframework.orm.jpa.JpaTransactionManager] -
<Creating new transaction with name [com.apress.prospring3.ch13.service.
annotation.ContactServiceImpl.findAll]: PROPAGATION_REQUIRED,
ISOLATION_DEFAULT,readOnly; ''>
DEBUG [org.hibernate.transaction.JDBCTransaction] - <begin>
Hibernate: select contact0_.ID as ID0_, contact0_.BIRTH_DATE as
BIRTH2_0_, contact0_.FIRST_NAME as FIRST3_0_, contact0_.LAST_NAME as
LAST4_0_, contact0_.VERSION as VERSION0_ from contact contact0_
DEBUG [org.springframework.orm.jpa.JpaTransactionManager] -
<Initiating transaction commit>
DEBUG [org.springframework.orm.jpa.JpaTransactionManager] -
<Committing JPA transaction on EntityManager
[org.hibernate.ejb.EntityManagerImpl@5b62f030]>
DEBUG [org.hibernate.transaction.JDBCTransaction] - <commit>
DEBUG [org.hibernate.transaction.JDBCTransaction] - <re-enabling
autocommit>
DEBUG [org.hibernate.transaction.JDBCTransaction] - <committed JDBC
Connection>
DEBUG [org.hibernate.jdbc.ConnectionManager] - <aggressively releasing
JDBC connection>
DEBUG [org.hibernate.jdbc.ConnectionManager] - <releasing JDBC connection
[ (open PreparedStatements: 0, globally: 0) (open ResultSets: 0, globally: 0)]>
DEBUG [org.springframework.orm.jpa.JpaTransactionManager] -
<Closing JPA EntityManager [org.hibernate.ejb.EntityManagerImpl@5b62f030]
after transaction>
Contact - Id: 1, First name: Clarence, Last name: Ho, Birthday: 1980-07-30
Contact - Id: 2, First name: Scott, Last name: Tiger, Birthday: 1990-11-02
Contact - Id: 3, First name: John, Last name: Smith, Birthday: 1964-02-28
```

Для ясности в показанном выше выводе операторы, не относящиеся к делу, не показаны. Важные журнальные сообщения, имеющие отношение к обработке транзакций, выделены полужирным. В первой строке, перед запуском метода `findAll()`, экземпляр `JpaTransactionManager` из Spring создает новую транзакцию (имя соответствует полностью определенному имени класса с именем метода) со стандартными атрибу-

тами, но транзакция предназначена только для чтения, как определено в аннотации `@Transactional` на уровне метода. Затем запрос отправляется и после его завершения без ошибок транзакция фиксируется. Операции создания и фиксации обрабатываются `JpaTransactionManager`.

А теперь перейдем к реализации операции обновления. Необходимо реализовать методы `findById()` и `save()` в интерфейсе `ContactService`. Их реализация приведена в листинге 13.12.

Листинг 13.12. Класс `ContactServiceImpl` с реализованными методами `findById()` и `save()`

```
package com.apress.prospring3.ch13.service.annotation;

// Операторы импорта опущены.

@Service("contactService")
@Repository
@Transactional
public class ContactServiceImpl implements ContactService {

    // Остальной код не показан.

    @Transactional(readOnly=true)
    public Contact findById(Long id) {
        return contactRepository.findOne(id);
    }

    public Contact save(Contact contact) {
        return contactRepository.save(contact);
    }
}
```

Как показано в листинге 13.12, метод `findById()` также аннотирован с помощью `@Transactional(readOnly=true)`. В общем случае атрибут `readOnly=true` должен применяться ко всем методам поиска. Главная причина такого подхода связана с тем, что большинство поставщиков постоянства будут проводить определенную оптимизацию транзакций, предназначенных только для чтения. Например, Hibernate не будет поддерживать моментальные снимки управляемых экземпляров, извлекаемых из базы данных с включенным режимом только для чтения.

В методе `save()` мы просто вызываем метод `CrudRepository.save()` и не предоставляем никаких аннотаций. Это значит, что будет использоваться аннотация уровня класса, определяющая транзакцию только для чтения.

Давайте модифицируем класс `TxAnnotationSample` для тестирования метода `save()`, как показано в листинге 13.13.

Листинг 13.13. Тестирование метода `save()`

```
package com.apress.prospring3.ch13;

// Операторы импорта опущены.

public class TxAnnotationSample {

    public static void main(String[] args) {
        GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
        ctx.load("classpath:tx-annotation-app-context.xml");
        ctx.refresh();
        ContactService contactService = ctx.getBean("contactService",
            ContactService.class);
```

```
        Contact contact = contactService.findById(1l);
        contact.setFirstName("Peter");
        contactService.save(contact);
        System.out.println("Contact saved successfully");
    }
}
```

В листинге 13.13 видно, что был извлечен объект `Contact` с идентификатором 1, его имя было изменено, после чего объект был сохранен в базе данных. Запуск этого кода дает следующий вывод:

```
DEBUG [org.springframework.orm.jpa.JpaTransactionManager] -
<Creating new transaction with name [com.apress.prospring3.ch13.service.
annotation.ContactServiceImpl.findById] :PROPAGATION_REQUIRED,
ISOLATION_DEFAULT,readOnly; ''>
DEBUG [org.hibernate.transaction.JDBCTransaction] - <begin>
Hibernate: select contact0_.ID as ID0_0_, contact0_.BIRTH_DATE as
BIRTH2_0_0_, contact0_.FIRST_NAME as FIRST3_0_0_, contact0_.LAST_NAME as
LAST4_0_0_, contact0_.VERSION as VERSION0_0_ from contact contact0_
where contact0_.ID=?
DEBUG [org.springframework.orm.jpa.JpaTransactionManager] — <Creating new
transaction with name [com.apress.prospring3.ch13.service.annotation.
ContactServiceImpl.save]: PROPAGATION_REQUIRED,ISOLATION_DEFAULT; ''>
DEBUG [org.hibernate.transaction.JDBCTransaction] - <begin>
DEBUG [org.springframework.orm.jpa.JpaTransactionManager] — <Initiating
transaction commit>
DEBUG [org.springframework.orm.jpa.JpaTransactionManager] -
<Committing JPA transaction on EntityManager [org.hibernate.ejb.
EntityManagerImpl@396c75ed]>
DEBUG [org.hibernate.transaction.JDBCTransaction] - <commit>
DEBUG [org.hibernate.event.def.AbstractFlushingEventListener] -
<processing flush-time cascades>
DEBUG [org.hibernate.event.def.AbstractFlushingEventListener] -
<dirty checking collections>
DEBUG [org.hibernate.event.def.AbstractFlushingEventListener] -
<Flushed: 0 insertions, 1 updates, 0 deletions to 1 objects>
DEBUG [org.hibernate.event.def.AbstractFlushingEventListener] -
<Flushed: 0 (re)creations, 0 updates, 0 removals to 0 collections>
Hibernate: update contact set BIRTH_DATE=?, FIRST_NAME=?, LAST_NAME=?,
VERSION=? where ID=? and VERSION=?
DEBUG [org.hibernate.transaction.JDBCTransaction] - <re-enabling autocommit>
DEBUG [org.hibernate.transaction.JDBCTransaction] - <committed JDBC Connection>
DEBUG [org.hibernate.jdbc.ConnectionManager] - <aggressively releasing
JDBC connection>
DEBUG [org.hibernate.jdbc.ConnectionManager] - <releasing JDBC connection
[ (open PreparedStatements: 0, globally: 0) (open ResultSets: 0, globally: 0)]>
DEBUG [org.springframework.orm.jpa.JpaTransactionManager] -
<Closing JPA EntityManager [org.hibernate.ejb.EntityManagerImpl@396c75ed]
after transaction>
DEBUG [org.springframework.orm.jpa.EntityManagerFactoryUtils] -
<Closing JPA EntityManager>
Contact saved successfully
```

Важные журнальные сообщения выделены полужирным. Здесь видно, что в транзакции для метода `save()` стандартные атрибуты наследуются из аннотации `@Transactional` уровня класса. После завершения операции обновления `JpaTransactionManager` запускает фиксацию транзакции, что заставляет Hibernate сбросить контекст постоянства и закрыть лежащее в основе JDBC-подключение к базе данных.

Наконец, рассмотрим метод `countAll()`. В этом методе мы исследуем две разных конфигурации для транзакции. Хотя метод `CrudRepository.count()` вполне способен решить задачу, мы не будем его использовать. Взамен для демонстрационных целей мы реализуем другой метод. В основном это объясняется тем, что методы, определенные интерфейсом `CrudRepository` в Spring Data, уже помечены соответствующими атрибутами транзакций.

В листинге 13.14 показан новый метод `countAllContacts()`, определенный в интерфейсе `ContactRepository`.

Листинг 13.14. Интерфейс `ContactRepository` с методом `countAllContacts()`

```
package com.apress.prospring3.ch13.repository;

import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.CrudRepository;

import com.apress.prospring3.ch13.domain.Contact;

public interface ContactRepository extends CrudRepository<Contact, Long> {

    @Query("select count(c) from Contact c")
    public Long countAllContacts();
}
```

Как видно в листинге 13.14, к новому методу `countAllContacts()` применена аннотация `@Query` со значением, задающим JPQL-оператор, который подчитывает количество контактов. Реализация метода `countAll()` в классе `ContactServiceImpl` представлена в листинге 13.15.

Листинг 13.15. Класс `ContactServiceImpl` с реализованным методом `countAll()`

```
package com.apress.prospring3.ch13.service.annotation;

// Операторы импорта опущены.

@Service("contactService")
@Repository
@Transactional
public class ContactServiceImpl implements ContactService {

    // Остальной код не показан.

    @Transactional(readOnly=true)
    public long countAll() {
        return contactRepository.countAllContacts();
    }
}
```

Аннотация здесь та же самая, что и в других методах поиска. В листинге 13.16 приведен фрагмент кода для тестирования нового метода.

Листинг 13.16. Тестирование метода `countAll()`

```
package com.apress.prospring3.ch13;
public class TxAnnotationSample {
   public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:tx-annotation-app-context.xml");
      ctx.refresh();
      ContactService contactService = ctx.getBean("contactService",
         ContactService.class);
      // Тестирование метода countAll().
      System.out.println("Contact count: " + contactService.countAll());
   }
}
```

Запуск этой программы дает следующий вывод:

```
DEBUG [org.springframework.orm.jpa.JpaTransactionManager] -
<Creating new transaction with name
[com.apress.prospring3.ch13.service.annotation.ContactServiceImpl.countAll]:
PROPAGATION_REQUIRED,ISOLATION_DEFAULT,readOnly; ''>
DEBUG [org.hibernate.transaction.JDBCTransaction] - <begin>
Hibernate: select count(*) as col_0_0_ from contact contact0_ limit ?
DEBUG [org.hibernate.transaction.JDBCTransaction] - <commit>
DEBUG [org.hibernate.transaction.JDBCTransaction] - <re-enabling autocommit>
DEBUG [org.hibernate.transaction.JDBCTransaction] - <committed JDBC Connection>
DEBUG [org.springframework.orm.jpa.JpaTransactionManager] - <Closing
JPA EntityManager [org.hibernate.ejb.EntityManagerImpl@4a005364] after
transaction>
DEBUG [org.springframework.orm.jpa.EntityManagerFactoryUtils] -
<Closing JPA EntityManager>
Contact count: 3
```

В выводе видно, что для метода `countAll()` была создана транзакция, предназначенная только для чтения, чего и следовало ожидать.

Но мы не хотим, чтобы в транзакции участвовала функция `countAll()`. Причина в том, что ее результат не должен управляться диспетчером сущностей JPA. Вместо этого нужно просто получить счетчик и забыть о нем. В таком случае можно переопределить поведение распространения транзакции на `Propagation.NEVER`. В листинге 13.17 показан переделанный метод `countAll()`.

Листинг 13.17. Класс `ContactServiceImpl` с переделанной реализацией метода `countAll()`

```
package com.apress.prospring3.ch13.service.annotation;
// Операторы импорта опущены.
@Service("contactService")
@Repository
@Transactional
public class ContactServiceImpl implements ContactService {
   // Остальной код не показан.
   @Transactional(propagation=Propagation.NEVER)
   public long countAll() {
      return contactRepository.count();
   }
}
```

Запустив тестовый код из листинга 13.16 снова, вы увидите, что для метода `countAll()` транзакция создаваться не будет.

В этом разделе рассматривались основные конфигурации, с которыми вам придется иметь дело при повседневной обработке транзакций. В специальных случаях может понадобиться определять тайм-аут, уровень изоляции, выполнять ли откат (или нет) для специфических исключений и т.п.

На заметку! Класс `JpaTransactionManager` в Spring не поддерживает специальный уровень изоляции. Вместо этого он всегда использует стандартный уровень изоляции для лежащего в основе хранилища данных. В случае применения Hibernate в качестве поставщика постоянства для поддержки специального уровня изоляции существует обходной путь, заключающийся в расширении класса `HibernateJpaDialect` (за деталями обращайтесь в статью по адресу `http://amitstechblog.wordpress.com/2011/05/31/supporting-custom-isolation-levels-with-jpa`).

Использование XML-конфигурации для управления транзакциями

Другой общий подход к декларативному управлению транзакциями предусматривает применение поддержки АОП в Spring. До выхода версии Spring 2 для определения требований к транзакциям с бинами Spring нужно было использовать класс `TransactionProxyFactoryBean`. Однако, начиная с версии Spring 2, появился более простой способ определения таких требований за счет введения пространства имен `aop` и применения общих приемов конфигурирования АОП.

В этом разделе будет использоваться тот же самый пример, который приводился при объяснении подхода с аннотациями. Мы просто приведем его к стилю XML-конфигурации. В листинге 13.18 показано содержимое XML-файла конфигурации для управления транзакциями (`tx-declarative-app-context.xml`).

Листинг 13.18. XML-конфигурация для управления транзакциями

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:context="http://www.springframework.org/schema/context"
   xmlns:jdbc="http://www.springframework.org/schema/jdbc"
   xmlns:tx="http://www.springframework.org/schema/tx"
   xmlns:jpa="http://www.springframework.org/schema/data/jpa"
   xmlns:aop="http://www.springframework.org/schema/aop"
   xsi:schemaLocation="http://www.springframework.org/schema/jdbc
      http://www.springframework.org/schema/jdbc/spring-jdbc-3.1.xsd
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
      http://www.springframework.org/schema/tx
      http://www.springframework.org/schema/tx/spring-tx-3.1.xsd
      http://www.springframework.org/schema/data/jpa
      http://www.springframework.org/schema/data/jpa/spring-jpa-1.0.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context-3.1.xsd
      http://www.springframework.org/schema/aop
      http://www.springframework.org/schema/aop/spring-aop-3.1.xsd">

   <aop:config>
      <aop:pointcut id="serviceOperation" expression=
"execution(* com.apress.prospring3.ch13.service.declarative.*ServiceImpl.*(..))"/>
```

```
      <aop:advisor pointcut-ref="serviceOperation" advice-ref="txAdvice"/>
   </aop:config>

   <tx:advice id="txAdvice">
      <tx:attributes>
         <tx:method name="find*" read-only="true"/>
         <tx:method name="count*" propagation="NEVER"/>
         <tx:method name="*"/>
      </tx:attributes>
   </tx:advice>

   <jdbc:embedded-database id="dataSource" type="H2">
      <jdbc:script location="classpath:schema.sql"/>
      <jdbc:script location="classpath:test-data.sql"/>
   </jdbc:embedded-database>

   <bean id="transactionManager"
         class="org.springframework.orm.jpa.JpaTransactionManager">
      <property name="entityManagerFactory" ref="emf"/>
   </bean>

   <bean id="emf"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
     <property name="dataSource" ref="dataSource" />
     <property name="jpaVendorAdapter">
       <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter" />
     </property>
     <property name="packagesToScan" value="com.apress.prospring3.ch13.domain"/>
     <property name="jpaProperties">
        <props>
           <prop key="hibernate.dialect">
                 org.hibernate.dialect.H2Dialect
           </prop>
           <prop key="hibernate.max_fetch_depth">3</prop>
           <prop key="hibernate.jdbc.fetch_size">50</prop>
           <prop key="hibernate.jdbc.batch_size">10</prop>
           <prop key="hibernate.show_sql">true</prop>
        </props>
     </property>
   </bean>

   <context:component-scan
      base-package="com.apress.prospring3.ch13.service.declarative" />

   <jpa:repositories base-package="com.apress.prospring3.ch13.repository"
      entity-manager-factory-ref="emf"
      transaction-manager-ref="transactionManager"/>
</beans>
```

Конфигурация очень похожа на используемую в случае аннотаций, которая была представлена в листинге 13.9 (отличия выделены полужирным). В сущности, дескриптор `<tx:annotation-driven>` был удален, а в дескрипторе `<context:component-scan>` изменено имя пакета, применяемого для декларативного управления транзакциями. Наиболее важными дескрипторами являются `<aop:config>` и `<tx:advice>`.

В дескрипторе `<aop:config>` был определен срез для всех операций внутри уровня обслуживания (т.е. все классы реализации из пакета `com.apress.prospring3.ch13.service.declarative`). Совет ссылается на бин с идентификатором `txAdvice`, который определен дескриптором `<tx:advice>`. В дескрипторе `<tx:advice>` мы конфигурируем атрибуты транзакции для различных методов, которые должны участвовать в транзакции. Как видно в этом дескрипторе, мы указываем, что все методы поиска (методы с префиксом `find`) будут предназначены только для чтения, а методы подсчета (методы с префиксом `count`) не будут принимать участие в транзакции. К остальным методам будет применяться стандартное поведение транзакции. Это та же самая конфигурация, которая использовалась в примере с аннотациями. В листинге 13.19 показан класс реализации для декларативного управления транзакциями в XML-стиле.

Листинг 13.19. Класс `ContactServiceImpl` для XML-конфигурации транзакций

```
package com.apress.prospring3.ch13.service.declarative;

// Операторы импорта опущены.

@Service("contactService")
@Repository
public class ContactServiceImpl implements ContactService {

    @Autowired
    private ContactRepository contactRepository;

    public List<Contact> findAll() {
        return Lists.newArrayList(contactRepository.findAll());
    }

    public Contact findById(Long id) {
        return contactRepository.findOne(id);
    }

    public Contact save(Contact contact) {
        return contactRepository.save(contact);
    }

    public long countAll() {
        return contactRepository.countAllContacts ();
    }
}
```

Код в листинге 13.19 в основном тот же, что и в примере с аннотациями, только лишь удалены аннотации `@Transactional`, поскольку теперь связывание транзакций осуществляется с помощью АОП в Spring на основе XML-конфигурации.

В листинге 13.20 приведена тестовая программа.

Листинг 13.20. Тестирование XML-конфигурации транзакций

```
package com.apress.prospring3.ch13;

import java.util.List;

import org.springframework.context.support.GenericXmlApplicationContext;

import com.apress.prospring3.ch13.domain.Contact;
import com.apress.prospring3.ch13.service.ContactService;

public class TxDeclarativeSample {
```

```
    public static void main(String[] args) {
        GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
        ctx.load("classpath:tx-declarative-app-context.xml");
        ctx.refresh();

        ContactService contactService = ctx.getBean("contactService",
            ContactService.class);

        // Тестирование метода findAll().
        List<Contact> contacts = contactService.findAll();

        for (Contact contact: contacts) {
            System.out.println(contact);
        }

        // Тестирование метода save().
        Contact contact = contactService.findById(1l);
        contact.setFirstName("Peter");
        contactService.save(contact);
        System.out.println("Contact saved successfully");

        // Тестирование метода countAll().
        System.out.println("Contact count: " + contactService.countAll());
    }
}
```

Мы оставляем запуск тестовой программы и наблюдение за выводом операций, связанных с транзакциями, которые выполняются Spring и Hibernate, в качестве самостоятельной работы. В основном все должно выглядеть почти так же, как в примере с аннотациями.

Использование программных транзакций

Третий вариант предусматривает управление поведением транзакций программным образом. В этом случае доступны две возможности. Первая — внедрение экземпляра `PlatformTransactionManager` в бин и взаимодействие с диспетчером транзакций напрямую. Вторая возможность заключается в использовании класса `TransactionTemplate`, предоставляемого Spring, который значительно упрощает работу. В этом разделе мы продемонстрируем работу с классом `TransactionTemplate`. Чтобы упростить задачу, мы сосредоточим внимание только на реализации метода `ContactService.countAll()`.

В листинге 13.21 приведена XML-конфигурация для использования программных транзакций (`tx-programmatic-app-context.xml`).

Листинг 13.21. Конфигурация Spring для программного управления транзакциями

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:jdbc="http://www.springframework.org/schema/jdbc"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xmlns:jpa="http://www.springframework.org/schema/data/jpa"
    xsi:schemaLocation="http://www.springframework.org/schema/jdbc
        http://www.springframework.org/schema/jdbc/spring-jdbc-3.1.xsd
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
        http://www.springframework.org/schema/tx
```

```
      http://www.springframework.org/schema/tx/spring-tx-3.1.xsd
      http://www.springframework.org/schema/data/jpa
      http://www.springframework.org/schema/data/jpa/spring-jpa-1.0.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context-3.1.xsd">

   <jdbc:embedded-database id="dataSource" type="H2">
      <jdbc:script location="classpath:schema.sql"/>
      <jdbc:script location="classpath:test-data.sql"/>
   </jdbc:embedded-database>

   <bean id="transactionTemplate"
      class="org.springframework.transaction.support.TransactionTemplate">
      <property name="propagationBehaviorName" value="PROPAGATION_NEVER"/>
      <property name="timeout" value="30"/>
      <property name="transactionManager" ref="transactionManager"/>
   </bean>

   <bean id="transactionManager"
         class="org.springframework.orm.jpa.JpaTransactionManager">
      <property name="entityManagerFactory" ref="emf"/>
   </bean>

   <bean id="emf"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
      <property name="dataSource" ref="dataSource" />
      <property name="jpaVendorAdapter">
<bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter" />
      </property>
      <property name="packagesToScan"
         value="com.apress.prospring3.ch13.domain"/>
      <property name="jpaProperties">
         <props>
            <prop key="hibernate.dialect">
               org.hibernate.dialect.H2Dialect
            </prop>
            <prop key="hibernate.max_fetch_depth">3</prop>
            <prop key="hibernate.jdbc.fetch_size">50</prop>
            <prop key="hibernate.jdbc.batch_size">10</prop>
            <prop key="hibernate.show_sql">true</prop>
         </props>
      </property>
   </bean>

   <context:component-scan
      base-package="com.apress.prospring3.ch13.service.programmatic" />

   <jpa:repositories base-package="com.apress.prospring3.ch13.repository"
      entity-manager-factory-ref="emf"
      transaction-manager-ref="transactionManager"/>

</beans>
```

Как показано в листинге 13.21, совет АОП для транзакции был удален. Вдобавок был определен бин `transactionTemplate` с использованием класса `org.springframework.transaction.support.TransactionTemplate`, атрибуты транзакции для которого определены в разделе свойств. Давайте рассмотрим реализацию метода `countAll()`, которая показана в листинге 13.22.

Листинг 13.22. Реализация программного управления транзакциями

```
package com.apress.prospring3.ch13.service.programmatic;

// Операторы импорта опущены.

@Service("contactService")
@Repository
public class ContactServiceImpl implements ContactService {

   @Autowired
   private TransactionTemplate transactionTemplate;

   @PersistenceContext
   private EntityManager em;

   // Другие методы не реализованы.

   public long countAll() {
      return transactionTemplate.execute(new TransactionCallback<Long>() {
         public Long doInTransaction(TransactionStatus transactionStatus) {
            return em.createNamedQuery("Contact.countAll",
               Long.class).getSingleResult();
         }

      });
   }
}
```

В листинге 13.22 класс `TransactionTemplate` был внедрен Spring. Затем в методе `countAll()` был вызван метод `TransactionTemplate.execute()` с передачей ему объявления внутреннего класса, реализующего интерфейс `TransactionCallback<T>`. Метод `doInTransaction()` был переопределен с применением желаемой логики. Эта логика будет работать с атрибутами, как определено бином `transactionTemplate`. Тестовая программа представлена в листинге 13.23.

Листинг 13.23. Тестирование программных транзакций

```
package com.apress.prospring3.ch13;

import org.springframework.context.support.GenericXmlApplicationContext;

import com.apress.prospring3.ch13.service.ContactService;

public class TxProgrammaticSample {

   public static void main(String[] args) {

      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:tx-programmatic-app-context.xml");
      ctx.refresh();

      ContactService contactService = ctx.getBean("contactService",
         ContactService.class);

      System.out.println("Contact count: " + contactService.countAll());

   }
}
```

Запуск тестовой программы и наблюдение за результатами мы оставляем в качестве самостоятельной работы. Попробуйте настроить атрибуты транзакции и посмотрите, что случиться при обработке транзакции для метода `countAll()`.

Соображения по поводу управления транзакциями

Итак, обсудив различные пути для реализации управления транзакциями, осталось выяснить, какой из них должен использоваться. Декларативный подход рекомендуется применять во всех случаях, и следует по возможности избегать реализации управления транзакциями в коде. Когда возникает необходимость в написании логики управления транзакциями внутри приложения, чаще всего это объясняется неудачным проектным решением, и в такой ситуации нужно вынести логику в управляемые фрагменты и определить с их помощью требования к транзакциям декларативно.

При декларативном подходе стили с XML и аннотациями имеют свои за и против. Одни разработчики предпочитают не объявлять требования к транзакциям в коде, в то время как другие отдают предпочтение аннотациям за их простое сопровождение, поскольку в коде можно видеть все объявление требований к транзакциям. Здесь снова следует позволить требованиям к приложению управлять принимаемым решением, а если в команде или компании существует стандартный подход, то придерживаться соответствующего ему стиля конфигурации.

Глобальные транзакции в Spring

Многим корпоративным Java-приложениям необходим доступ к множеству ресурсов серверной части. Например, порция информации о заказчике, полученная от внешнего бизнес-партнера, будет требовать обновления баз данных сразу в нескольких системах (CRM, ERP и т.д.). В ряде случаев может понадобиться формирование сообщения и его отправка серверу MQ через JMS для всех остальных приложений в компании, которые заинтересованы в информации о заказчике. Транзакции, которые охватывают множество ресурсов серверной части, называются *глобальными* (или распределенными).

Главной характеристикой глобальной транзакции является гарантия атомарности, а это значит, что задействованные ресурсы будут либо все обновлены, либо ни один из них обновляться не будет. Диспетчер транзакций должен поддерживать сложную логику координации и синхронизации. В мире Java стандартом де-факто для реализации глобальных транзакций является JTA.

Платформа Spring поддерживает транзакции JTA так же хорошо, как и локальные транзакции, и скрывает эту логику от бизнес-кода. В этом разделе мы покажем, как реализовать глобальные транзакции, используя JTA и Spring.

Инфраструктура для реализации примера применения JTA

Мы будем использовать ту же таблицу, что и в предыдущих примерах этой главы. Однако встроенная база данных H2 не полностью поддерживает XA (во всяком случае, на момент написания этой книги), поэтому в настоящем примере в качестве базы данных серверной части будет применяться MySQL.

Мы также хотим показать, каким образом реализовать глобальные транзакции с помощью JTA в автономном приложении или в среде веб-контейнера. Таким образом, в этом примере мы будем использовать продукт Atomikos (`www.atomikos.com/Main/TransactionsEssentials`), который представляет собой широко применяемый диспетчер транзакций JTA с открытым кодом, предназначенный для сред, отличных от JEE.

Чтобы посмотреть, как работают глобальные транзакции, необходимо иметь, по крайней мере, два разных ресурса серверной части. Для простоты мы будем использовать одну базу данных MySQL, но два диспетчера сущностей JPA. Требование наличия нескольких ресурсов будет достигнуто, т.к. имеется несколько единиц постоянства JPA для различения баз данных серверной части. При разработке примера применялась версия MySQL 5.1.58 Community Server.

В базе данных MySQL мы создадим две схемы, описанные в табл. 13.5.

Таблица 13.5. Схемы базы данных MySQL

Имя схемы	Информация подключения	Сценарий для наполнения данными
prospring3_ch13a	Имя пользователя: prospring3_ch13a Пароль: prospring3_ch13a	schema.sql test-data.sql
prospring3_ch13b	Имя пользователя: prospring3_ch13b Пароль: prospring3_ch13b	schema.sql test-data.sql

Для установки схемы и сценариев использовался инструмент phpMyAdmin (www.phpmyadmin.net). Тем не менее, для этого можно применять любой инструмент, предназначенный для настройки схем и пользователей.

После этого к проекту потребуется добавить обязательные зависимости для MySQL и Atomikos. Эти зависимости перечислены в табл. 13.6.

Таблица 13.6. Зависимости Maven для MySQL и Atomikos

Идентификатор группы	Идентификатор артефакта	Версия	Описание
mysql	mysql-connector-java	5.1.18	Java-библиотека для MySQL 5
com.atomikos	transactions-jdbc	3.7.0	Библиотека транзакций JTA для Atomikos
com.atomikos	transactions-hibernate3	3.7.0	Библиотека для интеграции Atomikos с Hibernate 3

После завершения всех настроек можно приступать к конфигурированию и реализации.

Реализация глобальных транзакций с помощью JTA

Сначала давайте посмотрим на конфигурацию Spring. В листинге 13.24 показано содержимое файла tx-jta-app-context.xml.

Листинг 13.24. Конфигурация Spring для JTA

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:context="http://www.springframework.org/schema/context"
   xmlns:jdbc="http://www.springframework.org/schema/jdbc"
   xmlns:tx="http://www.springframework.org/schema/tx"
   xsi:schemaLocation="http://www.springframework.org/schema/jdbc
      http://www.springframework.org/schema/jdbc/spring-jdbc-3.1.xsd
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
      http://www.springframework.org/schema/tx
      http://www.springframework.org/schema/tx/spring-tx-3.1.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context-3.1.xsd">

   <bean id="dataSourceA"
```

```
      class="com.atomikos.jdbc.AtomikosDataSourceBean"
      init-method="init" destroy-method="close">
      <property name="uniqueResourceName"><value>XADBMSA</value></property>
      <property name="xaDataSourceClassName">
         <value>com.mysql.jdbc.jdbc2.optional.MysqlXADataSource</value>
      </property>
      <property name="xaProperties">
         <props>
            <prop key="databaseName">prospring3_ch13a</prop>
            <prop key="user">prospring3_ch13a</prop>
           <prop key="password">prospring3_ch13a</prop>
         </props>
      </property>
      <property name="poolSize"><value>1</value></property>
   </bean>

   <bean id="dataSourceB"
      class="com.atomikos.jdbc.AtomikosDataSourceBean"
      init-method="init" destroy-method="close">
      <property name="uniqueResourceName"><value>XADBMSB</value></property>
      <property name="xaDataSourceClassName">
         <value>com.mysql.jdbc.jdbc2.optional.MysqlXADataSource</value>
      </property>
      <property name="xaProperties">
         <props>
            <prop key="databaseName">prospring3_ch13b</prop>
            <prop key="user">prospring3_ch13b</prop>
            <prop key="password">prospring3_ch13b</prop>
         </props>
      </property>
      <property name="poolSize"><value>1</value></property>
   </bean>

   <!-- Сконструировать Atomikos UserTransactionManager, необходимый для
конфигурирования Spring -->
   <bean id="atomikosTransactionManager"
      class="com.atomikos.icatch.jta.UserTransactionManager"
      init-method="init" destroy-method="close">
      <!--  Должны ли завершаться транзакции при вызове close?  -->
      <property name="forceShutdown">
         <value>true</value>
      </property>
   </bean>

   <!-- Также использовать Atomikos UserTransactionImp, необходимый для
конфигурирования Spring  -->
   <bean id="atomikosUserTransaction"
      class="com.atomikos.icatch.jta.UserTransactionImp">
      <property name="transactionTimeout">><value>300</value></property>
   </bean>

   <!-- Конфигурировать Spring для использования транзакций JTA из Atomikos -->
   <bean id="transactionManager"
      class="org.springframework.transaction.jta.JtaTransactionManager">
      <property name="transactionManager">
         <ref bean="atomikosTransactionManager"/>
      </property>
      <property name="userTransaction">
```

```
         <ref bean="atomikosUserTransaction"/>
      </property>
   </bean>

   <tx:annotation-driven transaction-manager="transactionManager" />

   <bean id="emfBase"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean"
abstract="true">
      <property name="jpaVendorAdapter">
         <bean
class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter" />
      </property>
      <property name="packagesToScan"
         value="com.apress.prospring3.ch13.domain"/>
      <property name="jpaProperties">
         <props>
            <prop key="hibernate.transaction.factory_class">
com.atomikos.icatch.jta.hibernate3.AtomikosJTATransactionFactory
            </prop>
            <prop key="hibernate.transaction.manager_lookup_class">
com.atomikos.icatch.jta.hibernate3.TransactionManagerLookup</prop>
            <prop key="hibernate.dialect">
               org.hibernate.dialect.MySQL5Dialect
            </prop>
            <prop key="hibernate.max_fetch_depth">3</prop>
            <prop key="hibernate.jdbc.fetch_size">50</prop>
            <prop key="hibernate.jdbc.batch_size">10</prop>
            <prop key="hibernate.show_sql">true</prop>
         </props>
      </property>
   </bean>

   <bean id="emfA" parent="emfBase">
      <property name="dataSource" ref="dataSourceA" />
   </bean>

   <bean id="emfB" parent="emfBase">
      <property name="dataSource" ref="dataSourceB" />
   </bean>

   <context:component-scan
      base-package="com.apress.prospring3.ch13.service.jta" />
</beans>
```

Конфигурация довольно обширна, но не особо сложна. Первым делом, определены два бина для источников данных, указывающие два разных ресурса баз данных. Бины имеют имена `dataSourceA` и `dataSourceB` и подключаются к схемам `prospring3_ch13a` и `prospring3_ch13b`. Оба бина для источников данных используют класс `com.atomikos.jdbc.AtomikosDataSourceBean`, который поддерживает совместимый с XA источник данных, и внутри определений этих бинов определен класс реализации источника данных XA типа MySQL (`com.mysql.jdbc.jdbc2.optional.MysqlXADataSource`); он представляет собой диспетчер ресурсов для MySQL. Затем предоставляется информация подключения к базе данных. Обратите внимание, что атрибут `poolSize` задает количество подключений в пуле подключений, который должен поддерживаться Atomikos. Это не является обязательным. Если этот атрибут не указан, Atomikos будет использовать значение по умолчанию, равное 1.

В части, связанной с Atomikos, определены два бина, atomikosTransactionManager и atomikosUserTransaction. Классы реализации предоставляются Atomikos и реализуют интерфейсы TransactionManager и UserTransaction из стандарта JEE. Указанные бины предлагают службы по координации и синхронизации транзакций, требуемые JTA, и взаимодействуют с диспетчерами ресурсов по протоколу XA при поддержке 2PC. После этого определяется Spring-бин transactionManager (с классом реализации JtaTransactionManager) и внедряются два бина транзакций, предоставляемые Atomikos. Это информирует Spring о необходимости использования Atomikos JTA для управления транзакциями.

Далее определяются три бина EntityManagerFactory с именами emfBase, emfA и emfB. Бин emfBase — это абстрактный родительский бин, который является оболочкой для общих свойств JPA. Бин emfBase реализован с помощью класса LocalContainerEntityManagerFactoryBean из Spring.

Бины emfA и emfB наследуют конфигурацию от родительского бина emfBase и отличаются только тем, что в них внедряются разные источники данных (т.е. dataSourceA внедряется в emfA, а dataSourceB — в emfB). Следовательно, emfA будет подключаться к MySQL-схеме prospring3_ch13a через бин dataSourceA, а emfB — подключаться к схеме prospring3_ch13b через бин dataSourceB.

Взгляните на свойства hibernate.transaction.factory_class и hibernate.transaction.manager_lookup_class в бине emfBase. Эти два свойства очень важны, потому что они используются Hibernate для поиска интерфейсов UserTransaction и TransactionManager, чтобы участвовать в контексте постоянства, который управляется в глобальной транзакции. Указанные классы предоставляются модулем transactions-hibernate3 в Atomikos.

В листинге 13.25 показан класс ContactServiceImpl для JTA. Обратите внимание, что для простоты реализован только метод save().

Листинг 13.25. Класс реализации ContactService для JTA

```
package com.apress.prospring3.ch13.service.jta;
import java.util.List;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.PersistenceException;
import org.springframework.orm.jpa.JpaSystemException;
import org.springframework.stereotype.Repository;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import com.apress.prospring3.ch13.domain.Contact;
import com.apress.prospring3.ch13.service.ContactService;
@Service("contactService")
@Repository
@Transactional
public class ContactServiceImpl implements ContactService {

    @PersistenceContext(unitName="emfA")
    private EntityManager emA;

    @PersistenceContext(unitName="emfB")
    private EntityManager emB;

    @Transactional(readOnly=true)
    public List<Contact> findAll() {
        return null;
    }
```

```
    @Transactional(readOnly=true)
    public Contact findById(Long id) {
        return null;
    }

    public Contact save(Contact contact) {
        Contact contactB = new Contact();
        contactB.setFirstName(contact.getFirstName());
        contactB.setLastName(contact.getLastName());
        if (contact.getId() == null) {
            emA.persist(contact);
            emB.persist(contactB);
            //throw new JpaSystemException(new PersistenceException());
        } else {
            emA.merge(contact);
            emB.merge(contact);
        }

        return contact;
    }

    public long countAll() {
        return 0;
    }
}
```

В листинге 13.25 определены два диспетчера сущностей, которые внедрены в класс `ContactServiceImpl`. В методе `save()` мы сохраняем объект контакта в две разных схемы. Сейчас пока проигнорируйте оператор `throw` для исключения; мы будем использовать его позже для проверки, произошел ли откат транзакции в случае сбоя сохранения в схему `prospring3_ch13b`. В листинге 13.26 приведен код тестовой программы.

Листинг 13.26. Тестирование транзакции JTA

```
package com.apress.prospring3.ch13;

import org.springframework.context.support.GenericXmlApplicationContext;

import com.apress.prospring3.ch13.domain.Contact;
import com.apress.prospring3.ch13.service.ContactService;

public class TxJtaSample {

    public static void main(String[] args) {

        GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
        ctx.load("classpath:tx-jta-app-context.xml");
        ctx.refresh();

        ContactService contactService = ctx.getBean("contactService",
            ContactService.class);

        Contact contact = new Contact();
        contact.setFirstName("Jta");
        contact.setLastName("Manager");
        contactService.save(contact);
        System.out.println("Contact saved successfully");

    }
}
```

Эта программа создает новый объект контакта и вызывает метод `ContactService.save()`. Реализация попытается сохранить один и тот же объект в двух базах данных. Запуск этой программы дает следующий вывод (показан не полностью):

```
INFO [atomikos] - <createCompositeTransaction ( 300000 ): created new ROOT
transaction with id 192.168.11.8.tm0000100001>
DEBUG [org.springframework.orm.jpa.EntityManagerFactoryUtils] -
<Opening JPA EntityManager>
DEBUG [org.hibernate.impl.SessionImpl] - <opened session at timestamp:
13208330451>
DEBUG [atomikos] - <getCompositeTransaction()  returning instance with id
192.168.11.8.tm0000100001>
INFO [atomikos] - <registerSynchronization ( com.atomikos.icatch.jta.
Sync2Sync@380199d8 ) for transaction 192.168.11.8.tm0000100001>
DEBUG [org.hibernate.jdbc.JDBCContext] -
<successfully registered Synchronization>
DEBUG [org.springframework.orm.jpa.EntityManagerFactoryUtils] -
<Registering transaction synchronization for JPA EntityManager>
DEBUG [org.hibernate.jdbc.ConnectionManager] - <opening JDBC connection>
Hibernate: insert into contact (BIRTH_DATE, FIRST_NAME, LAST_NAME, VERSION)
values (?, ?, ?, ?)
INFO [atomikos] - <addParticipant ( XAResourceTransaction: 3139322E3136382
E31312E382E746D30303030313030303031:3139322E3136382E31312E382E746D31 ) for
transaction 192.168.11.8.tm0000100001>
INFO [atomikos] - <XAResource.start ( 3139322E3136382E31312E382E746D3030
3030313030303031:3139322E3136382E31312E382E746D31 , XAResource.TMNOFLAGS
) on resource XADBMSA represented by XAResource instance com.mysql.jdbc.
jdbc2.optional.JDBC4MysqlXAConnection@49b29f80>
INFO [atomikos] - <XAResource.end ( 3139322E3136382E31312E382E746D30303030
313030303031:3139322E3136382E31312E382E746D31 , XAResource.TMSUCCESS ) on
resource XADBMSA represented by XAResource instance com.mysql.jdbc.jdbc2.
optional.JDBC4MysqlXAConnection@49b29f80>
DEBUG [org.springframework.orm.jpa.EntityManagerFactoryUtils] - <Opening
JPA EntityManager>
DEBUG [org.hibernate.impl.SessionImpl] - <opened session at timestamp:
13208330452>
DEBUG [atomikos] - <getCompositeTransaction()  returning instance with id
192.168.11.8.tm0000100001>
DEBUG [org.hibernate.jdbc.ConnectionManager] - <opening JDBC connection>
DEBUG [atomikos] - <getCompositeTransaction()  returning instance with id
192.168.11.8.tm0000100001>
Hibernate: insert into contact (BIRTH_DATE, FIRST_NAME, LAST_NAME, VERSION)
values (?, ?, ?, ?)
DEBUG [atomikos] - <getCompositeTransaction()  returning instance with id
192.168.11.8.tm0000100001>
DEBUG [atomikos] - <getCompositeTransaction()  returning instance with id
192.168.11.8.tm0000100001>
INFO [atomikos] - <registerSynchronization ( com.atomikos.icatch.jta.
Sync2Sync@45b2b450 ) for transaction 192.168.11.8.tm0000100001>
DEBUG [org.hibernate.jdbc.JDBCContext] - <successfully registered Synchronization>
DEBUG [org.springframework.orm.jpa.EntityManagerFactoryUtils] -
<Registering transaction synchronization for JPA EntityManager>
INFO [atomikos] - <addParticipant ( XAResourceTransaction: 3139322E3136382
E31312E382E746D30303030313030303031:3139322E3136382E31312E382E746D32 ) for
transaction 192.168.11.8.tm0000100001>
INFO [atomikos] - <XAResource.start ( 3139322E3136382E31312E382E746D303030
```

```
30313030303031:3139322E3136382E31312E382E746D32 , XAResource.TMNOFLAGS )
on resource XADBMSB represented by XAResource instance com.mysql.jdbc.
jdbc2.optional.JDBC4MysqlXAConnection@3f7aa2a8>
INFO [atomikos] - <XAResource.end ( 3139322E3136382E31312E382E746D30303030
313030303031:3139322E3136382E31312E382E746D32 , XAResource.TMSUCCESS ) on
resource XADBMSB represented by XAResource instance com.mysql.jdbc.jdbc2.
optional.JDBC4MysqlXAConnection@3f7aa2a8>
DEBUG [org.springframework.orm.jpa.EntityManagerFactoryUtils] -
<Closing JPA EntityManager>
DEBUG [org.springframework.orm.jpa.EntityManagerFactoryUtils] -
<Closing JPA EntityManager>
INFO [atomikos] - <commit() done (by application) of transaction
192.168.11.8.tm0000100001>
DEBUG [atomikos] - <About to call prepare on XAResource instance:
com.mysql.jdbc.jdbc2.optional.JDBC4MysqlXAConnection@49b29f80>
INFO [atomikos] - <XAResource.prepare ( 3139322E3136382E31312E382E746D3
030303031303030303031:3139322E3136382E31312E382E746D32 ) returning OK on
resource XADBMSB represented by XAResource instance com.mysql.jdbc.jdbc2.
optional.JDBC4MysqlXAConnection@3f7aa2a8>
INFO [atomikos] - <XAResource.commit ( 3139322E3136382E31312E382E746D303
03030313030303031:3139322E3136382E31312E382E746D31 , false) on resource
XADBMSA represented by XAResource instance com.mysql.jdbc.jdbc2.optional.
JDBC4MysqlXAConnection@49b29f80>
Contact saved successfully
```

В этом выводе видно, что Atomikos создает составную транзакцию, взаимодействует с источником данных XA (в рассматриваемом случае MySQL), выполняет синхронизацию, фиксирует транзакцию и т.д.

Новый контакт сохраняется в обеих схемах базы данных. А теперь посмотрим, как работает откат.

В листинге 13.25 вместо вызова `emB.persist()` мы просто сгенерируем исключение. Соответствующий фрагмент кода показан в листинге 13.27.

Листинг 13.27. Тестирование отката транзакции JTA

```
//emB.persist(contactB);
throw new JpaSystemException(new PersistenceException());
```

Чтобы протестировать сценарий с откатом, сначала удалите новые записи, вставленные предыдущим примером, из двух баз данных MySQL (т.е. запись контакта с полем `FIRST_NAME`, равным `Jta`, и полем `LAST_NAME`, равным `Manager`). Запуск программы теперь даст следующие результаты:

```
Hibernate: insert into contact (BIRTH_DATE, FIRST_NAME, LAST_NAME, VERSION)
values (?, ?, ?, ?)
INFO [atomikos] - <afterCompletion ( STATUS_ROLLEDBACK ) called  on
Synchronization: org.hibernate.transaction.synchronization.HibernateSynch
ronizationImpl@45c81ac0>
INFO [atomikos] - <rollback() done of transaction 192.168.11.8.tm0000100002>
Exception in thread "main" org.springframework.orm.jpa.JpaSystemException:
nested exception is javax.persistence.PersistenceException
...
Caused by: javax.persistence.PersistenceException
```

В показанном выводе видно, что первый контакт был сохранен (обратите внимание на оператор `insert`). Однако при сохранении во втором источнике данных генериру-

ется исключение и Atomikos производит откат всей транзакции. Можете просмотреть схему prospring3_ch13a и удостовериться, что новый контакт не был вставлен.

Соображения по поводу использования диспетчера транзакций JTA

По поводу того, использовать или нет JTA для управления глобальными транзакциями, ведутся бурные обсуждения. Например, команда разработчиков Spring вообще не рекомендует применять JTA для глобальных транзакций, и доктор Дэвид Сайер (Dr. David Syer) из SpringSource разместил статью, описывающую семь способов реализации распределенных транзакций, в четырех из которых JTA не используется (www.javaworld.com/javaworld/jw-01-2009/jw-01-spring-transactions.html).

Как правило, когда приложение разворачивается на полномасштабном сервере приложений JEE, нет смысла не применять JTA, поскольку все производители популярных серверов приложений JEE оптимизируют собственные реализации JTA для своих платформ. Это одно из главных функциональных средств, за которые вы платите.

В случае автономной версии или развертывания в веб-контейнере позвольте требованиям приложения управлять принятием решения и проводите нагрузочное тестирование как можно раньше, удостоверяясь, что использование JTA не оказывает большого влияния на производительность.

Хорошая новость заключается в том, что Spring гладко работает с локальными и глобальными транзакциями в большинстве основных веб- и JEE-контейнеров, поэтому при переходе от одной стратегии управления транзакциями к другой обычно модификация кода не требуется.

В случае если вы решите применять JTA в своем приложении, используйте JtaTransactionManager из Spring. Имеется хорошая статья от команды Spring, посвященная применению Spring с JTA (http://blog.springsource.org/2011/08/15/configuring-spring-and-jta-without-full-java-ee).

Резюме

Управление транзакциями является ключевым аспектом обеспечения целостности данных в почти любом типе приложения. В этой главе мы обсудили, как использовать Spring для управления транзакциями, практически не влияя на исходный код приложений. Было также показано, как работать с локальными и глобальными транзакциями.

Мы рассмотрели множество примеров реализации транзакций, в том числе декларативные способы применения XML-конфигурации и аннотаций, а также программный подход.

Локальные транзакции поддерживаются внутри или за пределами сервера приложений JEE, а для включения поддержки локальных транзакций в Spring требуется лишь простое конфигурирование. Тем не менее, настройка среды для глобальной транзакции предусматривает больше работы и сильно зависит от того, с каким поставщиком JTA и соответствующими ресурсами серверной части должно взаимодействовать приложение.

ГЛАВА 14

Проверка достоверности с преобразованием типов и форматированием

В корпоративном приложении проверка достоверности критически важна. Проверка достоверности предназначена для верификации того, что обрабатываемые данные удовлетворяют всем заранее определенным бизнес-требованиям, а также для обеспечения целостности и пригодности данных на других уровнях приложения.

При разработке приложений проверка достоверности данных всегда упоминается вместе с преобразованием и форматированием. Причина в том, что, скорее всего, формат источника данных отличается от формата, используемого на сервере приложений. Например, в веб-приложении пользователь вводит информацию с помощью пользовательского интерфейса веб-браузера. Когда пользователь сохраняет данные, эти данные отправляются серверу (после завершения локальной проверки достоверности). На стороне сервера будет выполнен процесс привязки данных, в рамках которого данные извлекаются из HTTP-запроса, преобразуются и привязываются к соответствующим объектам предметной области (например, пользователи могут вводить контактную информацию в HTML-форме, которая будет привязываться к объекту `Contact` на сервере) на основе правил форматирования, определенных для каждого атрибута (к примеру, шаблон формата даты может выглядеть как `yyyy-MM-dd`). После завершения привязки данных к объекту предметной области применяются правила проверки достоверности для поиска любого нарушения ограничений. Если все проходит нормально, данные сохраняются, а пользователю отображается сообщение об успехе операции. В противном случае пользователю отображаются сообщения об ошибках проверки достоверности.

В последней части этой главы вы ознакомитесь о развитой поддержке Spring преобразования типов, форматирования полей и проверки достоверности. В частности, в главе рассматриваются следующие темы.

- *Система преобразования типов Spring и интерфейс поставщика службы форматировщиков (service provider interface — SPI).* Мы обсудим новую обобщенную систему преобразования типов и SPI-интерфейс форматировщиков (Formatter SPI), введенные в Spring 3. Мы покажем, как эти новые службы можно использовать для замены предыдущей поддержки редакторов свойств, и каким образом они выполняют преобразования между любыми Java-типами.
- *Проверка достоверности в Spring.* Мы посмотрим, как Spring поддерживает проверку достоверности объектов предметной области. Сначала мы предложим крат-

кое введение в собственный интерфейс Spring по имени `Validator`. После этого мы сосредоточим внимание на поддержке JSR-303: Bean Validation (JSR-303: Проверка достоверности бинов), ставшей доступной в версии Spring 3.

Создание проекта для примеров в STS

Начнем с создания проекта для примеров, рассматриваемых в этой главе. Создайте новый шаблонный проект Spring в STS, выбрав вариант Simple Spring Utility Project (Простой служебный проект Spring).

После создания в проект нужно добавить обязательные зависимости, описанные в табл. 14.1. Удостоверьтесь, что используется Spring 3.1 с JDK 6.

Таблица 14.1. Зависимости Maven для проверки достоверности

Идентификатор группы	Идентификатор артефакта	Версия	Описание
`javax.validation`	`validation-api`	1.0.0.GA	Библиотека API-интерфейса JSR-303
`org.hibernate`	`hibernate-validator`	4.2.0.Final	Библиотека Hibernate Validator, которая поддерживает JSR-303: Bean Validation
`joda-time`	`joda-time`	2.0	API-интерфейс для работы с датой и временем под названием Joda-Time (`http://joda-time.sourceforge.net/`), который используется в Spring Data JPA. В этой главе мы будем применять его в объектах предметной области
`org.slf4j`	`slf4j-log4j12`	1.6.1	Библиотека ведения журналов (`www.slf4j.org`), которая будет использоваться в примерах этой главы. Данная библиотека соединяет ведение журнала SLF4J с библиотекой log4j

Система преобразования типов Spring

В версии Spring 3 появилась новая система преобразования типов, предоставляющая мощный способ для выполнения преобразований между любыми Java-типами в рамках Spring-приложений. В этом разделе мы посмотрим, как эта новая служба реализует ту же функциональность, что и ранее использовавшиеся редакторы свойств, и ознакомимся с ее поддержкой преобразований между любыми Java-типами. Мы также продемонстрируем реализацию специального преобразователя типов с применением Converter SPI.

Преобразование из String с использованием редакторов свойств

В главе 5 было показано, что Spring обрабатывает преобразование из значений `String` в файлах свойств в свойства объектов POJO за счет поддержки редакторов свойств (`PropertyEditor`). Мы приведем здесь краткий обзор, а затем рассмотрим интерфейс Converter SPI (доступный, начиная с версии Spring 3.0), предлагающий более мощную альтернативу.

Пусть имеется класс `Contact` с несколькими атрибутами, показанный в листинге 14.1.

Листинг 14.1. Класс `Contact`

```
package com.apress.prospring3.ch14.domain;
import java.net.URL;
import org.joda.time.DateTime;
public class Contact {
   private String firstName;
   private String lastName;
   private DateTime birthDate;
   private URL personalSite;

   // Методы извлечения/установки не показаны.

   public String toString() {
      return "First name: " + getFirstName()
         + " - Last name: " + getLastName()
         + " - Birth date: " + getBirthDate()
         + " - Personal site: " + getPersonalSite();
   }
}
```

Как показано в листинге 14.1, для атрибута дня рождения (`birthDate`) используется класс `DateTime` из Joda-Time. Вдобавок имеется поле типа `URL`, указывающее персональный веб-сайт для контакта, если это применимо.

Теперь предположим, что необходимо конструировать объекты `Contact` в `ApplicationContext` с использованием значений, хранящихся либо в конфигурационном файле Spring, либо в файле свойств. Содержимое XML-файла конфигурации Spring (`prop-editor-app-context.xml`) приведено в листинге 14.2.

Листинг 14.2. Конфигурация Spring для редактора свойств

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:context="http://www.springframework.org/schema/context"
   xmlns:p="http://www.springframework.org/schema/p"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context-3.1.xsd">
   <context:annotation-config/>
   <context:property-placeholder
      location="classpath:application.properties"/>
   <bean id="dateTimeEditor"
      class="com.apress.prospring3.ch14.pe.editor.DateTimeEditor">
      <constructor-arg value="${date.format.pattern}"/>
   </bean>

   <bean class="org.springframework.beans.factory.config.CustomEditorConfigurer">
      <property name="customEditors">
         <map>
            <entry key="org.joda.time.DateTime">
               <ref local="dateTimeEditor" />
            </entry>
         </map>
      </property>
   </bean>
```

```
    <bean id="clarence" class="com.apress.prospring3.ch14.domain.Contact"
        p:firstName="Clarence"
        p:lastName="Ho"
        p:birthDate="1970-12-31"
        p:personalSite="http://www.clarence.com"
    />
    <bean id="myContact" class="com.apress.prospring3.ch14.domain.Contact"
        p:firstName="${myContact.firstName}"
        p:lastName="${myContact.lastName}"
        p:birthDate="${myContact.birthDate}"
        p:personalSite="${myContact.personalSite}"
    />
</beans>
```

В листинге 14.2 видно, что мы конструируем два разных бина из класса `Contact`. Бин `clarence` создан со значениями, предоставленными в конфигурационном файле, а в случае бина `myContact` атрибуты вынесены во внешний файл свойств. Кроме того, определен специальный редактор, предназначенный для преобразования значений `String` в тип `DateTime` из Joda-Time, а шаблон формата даты-времени также вынесен во внешний файл свойств. Содержимое файла свойств (`application.properties`) показано в листинге 14.3.

Листинг 14.3. Файл `application.properties`

```
date.format.pattern=yyyy-MM-dd

myContact.firstName=Scott
myContact.lastName=Tiger
myContact.birthDate=1984-6-30
myContact.personalSite=http://www.somedomain.com
```

В листинге 14.4 представлен специальный редактор для преобразования значений `String` в тип `DateTime` из Joda-Time.

Листинг 14.4. Специальный редактор для типа `DateTime` из Joda-Time

```
package com.apress.prospring3.ch14.pe.editor;

import java.beans.PropertyEditorSupport;

import org.joda.time.DateTime;
import org.joda.time.format.DateTimeFormat;
import org.joda.time.format.DateTimeFormatter;

public class DateTimeEditor extends PropertyEditorSupport {

    private DateTimeFormatter dateTimeFormatter;

    public DateTimeEditor(String dateFormatPattern) {
        dateTimeFormatter = DateTimeFormat.forPattern(dateFormatPattern);
    }

    public void setAsText(String text) throws IllegalArgumentException {
        setValue(DateTime.parse(text, dateTimeFormatter));
    }
}
```

Код в листинге 14.4 должен быть самоочевидным. Давайте его протестируем. Тестовая программа приведена в листинге 14.5.

Листинг 14.5. Тестирование редактора свойств

```
package com.apress.prospring3.ch14.pe;

import org.springframework.context.support.GenericXmlApplicationContext;

import com.apress.prospring3.ch14.domain.Contact;

public class PropEditorExample {

   public static void main(String[] args) {

      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:prop-editor-app-context.xml");
      ctx.refresh();

      Contact clarence = ctx.getBean("clarence", Contact.class);
      System.out.println("Clarence info: " + clarence);

      Contact myContact = ctx.getBean("myContact", Contact.class);
      System.out.println("My contact info: " + myContact);
   }
}
```

Как показано в листинге 14.5, два разных бина `Contact` извлекаются из `ApplicationContext` и затем отображаются. Запуск этой программы дает следующий вывод:

```
Clarence info: First name: Clarence - Last name: Ho - Birth date:
1970-12-31T00:00:00.000+08:00 — Personal site: http://www.clarence.com
My contact info: First name: Scott - Last name: Tiger - Birth date:
1984-06-30T00:00:00.000+08:00 — Personal site: http://www.somedomain.com
```

В выводе видно, что свойства были преобразованы и применены к бинам `Contact`. На рис. 14.1 показано логическое представление того, как осуществлялось преобразование редакторами свойств.

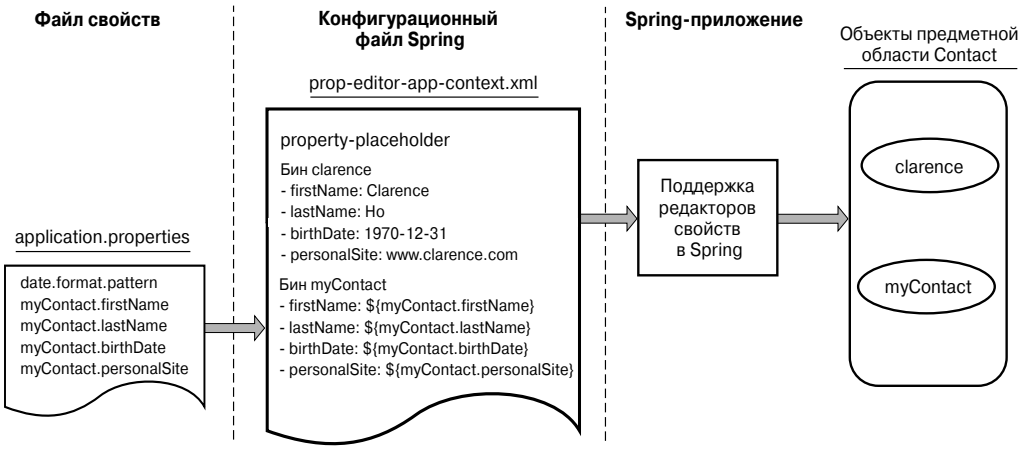


Рис. 14.1. Поддержка редакторов свойств в Spring

Появление в Spring 3 системы преобразования типов

В версии Spring 3.0 появилась обобщенная система преобразования типов, которая находится в пакете `org.springframework.core.convert`. В дополнение к предоставлению альтернативы поддержке редакторов свойств, система преобразования типов может быть также сконфигурирована на выполнение преобразования между Java-типами и объектами POJO (тогда как редакторы свойств ориентированы на преобразование в Java-типы строковых представлений из файла свойств).

Реализация специального преобразователя

Чтобы посмотреть на систему преобразования типов в действии, давайте вернемся к предыдущему примеру и воспользуемся тем же самым классом `Contact`. На этот раз предположим, что нужно использовать систему преобразования типов для преобразования формата `String` в свойство `birthDate` класса `Contact`, которое имеет тип `DateTime`, определенный Joda-Time. Для поддержки преобразования вместо специального редактора свойства мы создаем специальный преобразователь, реализуя интерфейс `org.springframework.core.convert.converter.Converter<S,T>`. Код этого специального преобразователя приведен в листинге 14.6.

Листинг 14.6. Специальный преобразователь `DateTime`

```
package com.apress.prospring3.ch14.converter;

import javax.annotation.PostConstruct;

import org.joda.time.DateTime;
import org.joda.time.format.DateTimeFormat;
import org.joda.time.format.DateTimeFormatter;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.core.convert.converter.Converter;

public class StringToDateTimeConverter
   implements Converter<String, DateTime> {

   private static final String DEFAULT_DATE_PATTERN = "yyyy-MM-dd";

   private DateTimeFormatter dateFormat;

   private String datePattern = DEFAULT_DATE_PATTERN;

   public String getDatePattern() {
      return datePattern;
   }

   @Autowired(required=false)
   public void setDatePattern(String datePattern) {
      this.datePattern = datePattern;
   }

   @PostConstruct
   public void init() {
      dateFormat = DateTimeFormat.forPattern(datePattern);
   }

   public DateTime convert(String dateString) {
      return dateFormat.parseDateTime(dateString);
   }
}
```

Как показано в листинге 14.6, мы реализуем интерфейс `Converter<String, DateTime>`, а это значит, что преобразователь отвечает за преобразование типа `String` (исходный тип `S`) в тип `DateTime` (целевой тип `T`).

Внедрение шаблона даты-времени является необязательным за счет его аннотирования с помощью `@Autowired(required=false)`. Если внедрение шаблона не производилось, используется шаблон по умолчанию `yyyy-MM-dd`. После этого в методе инициализации (метод `init()`, аннотированный посредством `@PostConstruct`) конструируется экземпляр класса `DateTimeFormat` из Joda-Time, который выполнит преобразование на основе указанного шаблона. Наконец, реализован метод `convert()`, содержащий логику преобразования.

Конфигурирование `ConversionService`

Для использования вместо редактора свойства службы преобразования необходимо сконфигурировать экземпляр интерфейса `org.springframework.core.convert.ConversionService` в `ApplicationContext`. Содержимое конфигурационного файла (`conv-service-app-context.xml`) приведено в листинге 14.7.

Листинг 14.7. Конфигурация службы преобразования

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:context="http://www.springframework.org/schema/context"
   xmlns:p="http://www.springframework.org/schema/p"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context-3.1.xsd">
   <context:annotation-config/>
   <bean id="conversionService"
  class="org.springframework.context.support.ConversionServiceFactoryBean">
      <property name="converters">
         <list>
            <bean
   class="com.apress.prospring3.ch14.converter.StringToDateTimeConverter"/>
         </list>
      </property>
   </bean>
   <bean id="clarence" class="com.apress.prospring3.ch14.domain.Contact"
      p:firstName="Clarence"
      p:lastName="Ho"
      p:birthDate="1978-08-09"
      p:personalSite="http://www.clarence.com"
   />
</beans>
```

В листинге 14.7 мы указываем Spring на использование системы преобразования типов за счет объявления бина `conversionService` с классом `ConversionServiceFactoryBean`. Если ни одного бина службы преобразования не определено, Spring будет работать с системой на основе редакторов свойств.

По умолчанию система преобразования типов поддерживает преобразование между общими типами, такими как строки, числа, перечисления, коллекции, карты и т.д. Кроме того, поддерживается преобразование значений `String` в Java-типы внутри системы, основанной на редакторах свойств.

В бине conversionService сконфигурирован специальный преобразователь из String в DateTime. В листинге 14.8 показана тестовая программа.

Листинг 14.8. Тестирование службы преобразования

```
package com.apress.prospring3.ch14.convserv;

import org.springframework.context.support.GenericXmlApplicationContext;

import com.apress.prospring3.ch14.domain.Contact;

public class ConvServExample {

   public static void main(String[] args) {

      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:conv-service-app-context.xml");
      ctx.refresh();

      Contact clarence = ctx.getBean("clarence", Contact.class);

      System.out.println("Contact info: " + clarence);

   }
}
```

Запуск этой программы дает следующий вывод:

```
Contact info: First name: Clarence - Last name: Ho - Birth date:
1978-08-09T00:00:00.000+08:00 — Personal site: http://www.clarence.com
```

Как видите, результат преобразования свойства бина clarence такой же, как в случае применения редакторов свойств.

Преобразование между произвольными типами

Реальная мощь системы преобразования типов заключается в ее возможности выполнять преобразования между произвольными типами. Чтобы увидеть это в действии, создадим еще один класс по имени AnotherContact, который совпадает с классом Contact. Его код представлен в листинге 14.9.

Листинг 14.9. Класс AnotherContact

```
package com.apress.prospring3.ch14.domain;

import java.net.URL;

import org.joda.time.DateTime;

public class AnotherContact {

   private String firstName;

   private String lastName;

   private DateTime birthDate;

   private URL personalSite;

   // Остальной код не показан.
}
```

Нам нужна возможность преобразования любого экземпляра класса Contact в экземпляр класса AnotherContact, но атрибуты firstName и lastName из Contact должны стать атрибутами lastName и firstName в AnotherContact. Давайте реализуем для этого специальный преобразователь. Его код показан в листинге 14.10.

Листинг 14.10. Класс ContactToAnotherContactConverter

```
package com.apress.prospring3.ch14.converter;

import org.springframework.core.convert.converter.Converter;

import com.apress.prospring3.ch14.domain.AnotherContact;
import com.apress.prospring3.ch14.domain.Contact;

public class ContactToAnotherContactConverter
   implements Converter<Contact, AnotherContact> {

   public AnotherContact convert(Contact contact) {
      AnotherContact anotherContact = new AnotherContact();
      anotherContact.setFirstName(contact.getLastName());
      anotherContact.setLastName(contact.getFirstName());
      anotherContact.setBirthDate(contact.getBirthDate());
      anotherContact.setPersonalSite(contact.getPersonalSite());
      return anotherContact;
   }
}
```

Класс ContactToAnotherContactConverter очень прост; он лишь меняет местами значения атрибутов firstName и lastName в классах Contact и AnotherContact. Чтобы зарегистрировать специальный преобразователь в ApplicationContext, замените определение бина conversionService в файле conv-service-app-context.xml фрагментом кода, приведенным в листинге 14.11.

Листинг 14.11. Добавление специального преобразователя к службе преобразования

```
   <bean id="conversionService" class="org.springframework.context.support.
ConversionServiceFactoryBean">
      <property name="converters">
         <list>
            <bean
class="com.apress.prospring3.ch14.converter.StringToDateTimeConverter"/>
            <bean
class="com.apress.prospring3.ch14.converter.ContactToAnotherContactConverter"/>
         </list>
      </property>
   </bean>
```

Порядок следования бинов в свойстве converters не важен.

Для тестирования преобразования мы воспользуемся тестовой программой из предыдущего примера, т.е. классом ConvServExample. В листинге 14.12 показан модифицированный метод main().

Листинг 14.12. Тестирование службы преобразования

```
package com.apress.prospring3.ch14.convserv;

// Операторы импорта опущены.

public class ConvServExample {

   public static void main(String[] args) {

      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:conv-service-app-context.xml");
      ctx.refresh();
```

```
        Contact clarence = ctx.getBean("clarence", Contact.class);

        System.out.println("Contact info: " + clarence);

        ConversionService conversionService = ctx.getBean(ConversionService.class);

        // Преобразование из Contact в AnotherContact.
        AnotherContact anotherContact =
           conversionService.convert(clarence, AnotherContact.class);
        System.out.println("Another contact info: " + anotherContact);

        // Преобразование из String в Array.
        String[] stringArray = conversionService.convert(
           "a,b,c", String[].class);
        System.out.println("String array: " + stringArray[0] +
           stringArray[1] + stringArray[2]);

        // Преобразование из List в Set.
        List<String> listString = new ArrayList<String>();
        listString.add("a");
        listString.add("b");
        listString.add("c");
        Set<String> setString = conversionService.convert(
           listString, HashSet.class);
        for (String string: setString)
        System.out.println("Set: " + string);
    }
}
```

В листинге 14.12 обратите внимание на выделенную полужирным строку кода, в которой получается обработчик интерфейса `ConversionService` из `ApplicationContext`. Поскольку мы уже зарегистрировали `ConversionService` в `ApplicationContext` со специальным преобразователем, его можно использовать для преобразования объекта `Contact`, а также для преобразований между другими типами, которые поддерживает служба преобразования. Для демонстрационных целей в код были добавлены примеры преобразования из `String` (разделенных запятыми) в `Array` и из `List` в `Set`.

Запуск этой программы дает следующий вывод:

```
Contact info: First name: Clarence - Last name: Ho - Birth date:
1978-08-09T00:00:00.000+08:00 — Personal site: http://www.clarence.com
Another contact info: First name: Ho - Last name: Clarence - Birth date:
1978-08-09T00:00:00.000+08:00 — Personal site: http://www.clarence.com
String array: abc
Set: b
Set: c
Set: a
```

В этом выводе видно, что преобразования между `Contact` и `AnotherContact` выполнены корректно, и то же самое можно сказать о преобразованиях `String` в `Array` и `List` в `Set`. Для службы преобразования типов Spring можно легко создавать специальные преобразователи и осуществлять преобразования на любом уровне приложения. Возможный сценарий использования предусматривает наличие двух разных систем с одной и той же информацией о контакте, которую необходимо обновить. Однако структуры баз данных в этих системах отличаются (например, фамилия в одной системе означает имя в другой и т.п.). Службу преобразования типов можно применять для преобразования объектов перед их сохранением в каждой отдельной системе.

Начиная с версии Spring 3.0, инфраструктура Spring MVC широко использует службу преобразования (а также интерфейс Formatter SPI, который обсуждается в сле-

дующем разделе). В конфигурации контекста веб-приложения объявление дескриптора `<mvc:annotation-driven/>` автоматически регистрирует все стандартные преобразователи (к примеру, `StringToArrayConverter`, `StringToBooleanConverter` и `StringToLocaleConverter` из пакета `org.springframework.core.convert.support`) и форматировщики (например, `CurrencyFormatter`, `DateFormatter` и `NumberFormatter` из различных подпакетов внутри пакета `org.springframework.format`).

Дополнительные сведения по этому поводу будут представлены в главах 17 и 18, когда будет обсуждаться разработка веб-приложений в Spring.

Форматирование полей в Spring 3

Помимо системы преобразования типов, разработчикам Spring-приложений доступно также другое важное средство — Formatter SPI. Как и можно было ожидать, этот SPI-интерфейс помогает настраивать все аспекты, связанные с форматированием полей.

Главным интерфейсом в Formatter SPI, предназначенным для реализации форматировщиков, является `org.springframework.format.Formatter<T>`. В Spring предлагается несколько его реализаций для часто используемых типов, среди которых `CurrencyFormatter`, `DateFormatter`, `NumberFormatter` и `PercentFormatter`.

Реализация специального форматировщика

Реализация специального форматировщика довольно проста. Мы снова воспользуемся классом `Contact` и реализуем специальный форматировщик для преобразования типа `DateTime` атрибута `birthDate` в и из `String`.

Однако в этот раз мы применим другой подход — мы расширим класс `org.springframework.format.support.FormattingConversionServiceFactoryBean` из Spring и предоставим специальный форматировщик.

Класс `FormattingConversionServiceFactoryBean` — это фабричный класс, обеспечивающий удобный доступ к лежащему в основе классу `FormattingConversionService`, который поддерживает систему преобразования типов, а также форматирование полей в соответствии с правилами форматирования, определенными для каждого типа поля.

В листинге 14.13 представлен специальный класс, который расширяет класс `FormattingConversionServiceFactoryBean`, со специальным форматировщиком, определенным для форматирования типа `DateTime` из Joda-Time.

Листинг 14.13. Класс `ApplicationConversionServiceFactoryBean`

```
package com.apress.prospring3.ch14.convserv.factory;

import java.text.ParseException;
import java.util.HashSet;
import java.util.Locale;
import java.util.Set;

import javax.annotation.PostConstruct;

import org.joda.time.DateTime;
import org.joda.time.format.DateTimeFormat;
import org.joda.time.format.DateTimeFormatter;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.format.Formatter;
import org.springframework.format.support.FormattingConversionServiceFactory
Bean;
```

```
public class ApplicationConversionServiceFactoryBean extends
   FormattingConversionServiceFactoryBean {

   final Logger logger =
     LoggerFactory.getLogger(ApplicationConversionServiceFactoryBean.class);

   private static final String DEFAULT_DATE_PATTERN = "yyyy-MM-dd";

   private DateTimeFormatter dateFormat;

   private String datePattern = DEFAULT_DATE_PATTERN;

   private Set<Formatter<?>> formatters = new HashSet<Formatter<?>>();

   public String getDatePattern() {
      return datePattern;
   }

   @Autowired(required=false)
   public void setDatePattern(String datePattern) {
      this.datePattern = datePattern;
   }

   @PostConstruct
   public void init() {
      dateFormat = DateTimeFormat.forPattern(datePattern);
      formatters.add(getDateTimeFormatter());
      setFormatters(formatters);
   }

   public Formatter<DateTime> getDateTimeFormatter() {
      return new Formatter<DateTime>() {
         public DateTime parse(String dateTimeString, Locale locale)
              throws ParseException {
            logger.info("Parsing date string: " + dateTimeString);
            return dateFormat.parseDateTime(dateTimeString);
         }
         public String print(DateTime dateTime, Locale locale) {
            logger.info("Formatting datetime: " + dateTime);
            return dateFormat.print(dateTime);
         }
      };
   }
}
```

Код специального форматировщика в листинге 14.13 выделен полужирным. Он реализует интерфейс `Formatter<DateTime>` и два метода, определенные этим интерфейсом. Метод `parse()` разбирает формат `String` и строит тип `DateTime` (для поддержки локализации также передается нужная локаль), в метод `print()` форматирует экземпляр `DateTime` в `String`. Шаблон даты может быть внедрен в бин (или же будет использоваться шаблон по умолчанию вида `yyyy-MM-dd`). Кроме того, в методе `init()` производится регистрация специального форматировщика с помощью вызова метода `setFormatters()`. Можно добавлять столько форматировщиков, сколько требуется в приложении.

Конфигурирование `ConversionServiceFactoryBean`

Чтобы сконфигурировать `ApplicationConversionServiceFactoryBean` в `ApplicationContext`, нужно всего лишь объявить бин с этим классом в качестве поставщика. Содержимое конфигурационного файла `conv-format-service-app-context.xml` показано в листинге 14.14.

Листинг 14.14. Файл conv-format-service-app-context.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:context="http://www.springframework.org/schema/context"
   xmlns:p="http://www.springframework.org/schema/p"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context-3.1.xsd">

   <context:annotation-config/>

   <bean id="conversionService"
class="com.apress.prospring3.ch14.convserv.factory.ApplicationConversion
ServiceFactoryBean"/>

   <bean id="clarence" class="com.apress.prospring3.ch14.domain.Contact"
      p:firstName="Clarence"
      p:lastName="Ho"
      p:birthDate="1978-08-09"
      p:personalSite="http://www.clarence.com"
   />
</beans>
```

В листинге 14.15 приведена тестовая программа.

Листинг 14.15. Тестирование специального форматировщика

```
package com.apress.prospring3.ch14.convserv;

import org.springframework.context.support.GenericXmlApplicationContext;
import org.springframework.core.convert.ConversionService;

import com.apress.prospring3.ch14.domain.Contact;

public class ConvFormatServExample {

   public static void main(String[] args) {

      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:conv-format-service-app-context.xml");
      ctx.refresh();

      Contact clarence = ctx.getBean("clarence", Contact.class);

      System.out.println("Contact info: " + clarence);

      ConversionService conversionService = ctx.getBean(
         "conversionService", ConversionService.class);
      System.out.println("Birthdate of contact is : " +
      conversionService.convert(clarence.getBirthDate(), String.class));
   }
}
```

Запуск этой программы дает следующий вывод:

```
INFO [com.apress.prospring3.ch14.convserv.factory.ApplicationConversion
ServiceFactoryBean] — <Parsing date string: 1978-08-09>
Contact info: First name: Clarence - Last name: Ho - Birth date:
1978-08-09T00:00:00.000+08:00 — Personal site: http://www.clarence.com
```

```
INFO [com.apress.prospring3.ch14.convserv.factory.ApplicationConversion
ServiceFactoryBean] - <Formatting datetime: 1978-08-09T00:00:00.000+08:00>
Birthdate of contact is : 1978-08-09
```

В выводе видно, что Spring применяет метод `parse()` специального форматировщика для преобразования свойства из `String` в тип `DateTime` атрибута `birthDate`. При вызове метода `ConversionService.convert()` с передачей ему атрибута `birthDate` будет вызван метод `print()` для форматирования вывода.

Проверка достоверности в Spring

Проверка достоверности является критически важной частью любого приложения. Правила проверки достоверности, применяемые к объектам предметной области, гарантируют, что все бизнес-данные хорошо структурированы и удовлетворяют всем бизнес-требованиям. В идеальном случае все правила проверки достоверности поддерживаются в централизованном местоположении, причем один и тот же набор правил применяется к одному и тому же типу данных вне зависимости от того, из какого источника данные поступили (например, через пользовательский ввод в веб-приложении, из удаленного приложения через веб-службы, из сообщения JMS, из файла и т.д.).

Когда речь идет о проверке достоверности, функции преобразования и форматирования также важны, потому что перед тем, как порция данных может быть проверена, она должна быть преобразована в желаемый объект POJO согласно правилам форматирования, определенным для каждого типа.

Предположим, что пользователь вводит некоторую контактную информацию в веб-приложении внутри браузера и затем отправляет данные на сервер. На стороне сервера, если веб-приложение было разработано с применением инфраструктуры Spring MVC, то Spring извлечет данные из HTTP-запроса и выполнит преобразование `String` в желаемый тип на основе правила форматирования (например, строка `String`, представляющая дату, будет преобразована в поле `Date`, с помощью правила форматирования `yyyy-MM-dd`). Этот процесс называется *привязкой данных*. Когда привязка данных завершена, а объект предметной области сконструирован, объект подвергается проверке достоверности, и возникшие ошибки возвращаются и отображаются пользователю. Если проверка достоверности прошла успешно, объект сохраняется в базе данных.

Этот процесс проиллюстрирован на рис. 14.2.

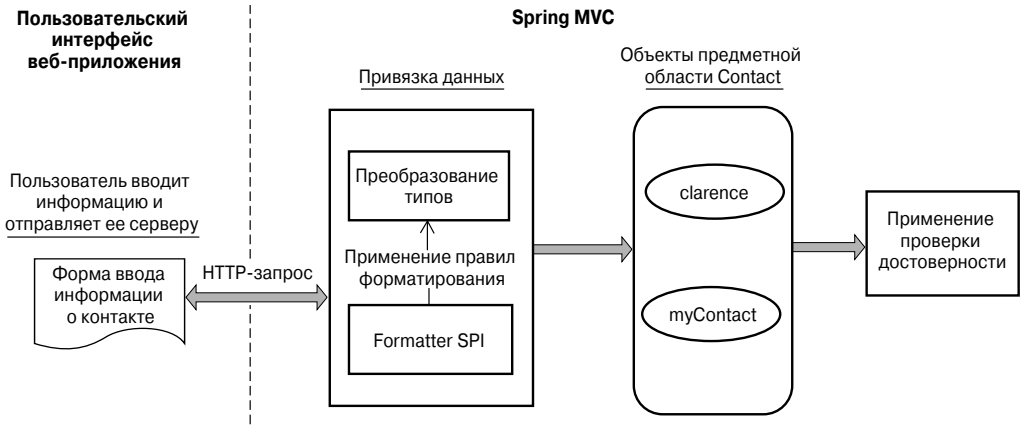


Рис. 14.2. Отношения между проверкой достоверности, преобразованием и форматированием

В Spring поддерживаются два основных вида проверки достоверности. Один из них предоставляется самой платформой Spring и предусматривает создание специальных валидаторов за счет реализации интерфейса `org.springframework.validation.Validator`. Другой вид проверки достоверности использует поддержку JSR-303, API-интерфейса проверки достоверности бинов, встроенную в Spring. В последующих разделах мы рассмотрим обе разновидности.

Использование интерфейса `Validator` в Spring

Используя интерфейс `Validator` в Spring, можно разрабатывать логику проверки достоверности, создавая класс для реализации этого интерфейса.

Давайте посмотрим, как это работает. Предположим, что в классе `Contact`, с которым мы имели дело до сих пор, имя не должно быть пустым. Чтобы проверить объекты `Contact` на предмет соблюдения этого правила, можно создать специальный валидатор. Класс такого валидатора показан в листинге 14.16.

Листинг 14.16. Класс `ContactValidator`

```
package com.apress.prospring3.ch14.validator;

import org.springframework.stereotype.Component;
import org.springframework.validation.Errors;
import org.springframework.validation.ValidationUtils;
import org.springframework.validation.Validator;

import com.apress.prospring3.ch14.domain.Contact;

@Component("contactValidator")
public class ContactValidator implements Validator {
   public boolean supports(Class<?> clazz) {
      return Contact.class.equals(clazz);
   }
   public void validate(Object obj, Errors e) {
      ValidationUtils.rejectIfEmpty(e, "firstName", "firstName.empty");
   }
}
```

В листинге 14.16 класс валидатора реализует интерфейс `Validator` и два его метода. Метод `supports()` указывает, поддерживает ли валидатор проверку достоверности для передаваемого ему класса. Метод `validate()` выполняет проверку достоверности для передаваемого ему объекта.

Результат будет сохранен в экземпляре `org.springframework.validation.Errors`. В методе `validate()` мы выполняем проверку только атрибута `firstName` и с помощью удобного метода `ValidationUtils.rejectIfEmpty()` гарантируем, что имя контакта не является пустым. Последний аргумент — это код ошибки, который может использоваться для поиска в пакетах ресурсов сообщений, связанных с проверкой достоверности, и последующего отображения локализованных сообщений об ошибках.

В листинге 14.17 приведено содержимое конфигурационного файла Spring (`spring-validator-app-context.xml`).

Листинг 14.17. Конфигурация для проверки достоверности в Spring

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:context="http://www.springframework.org/schema/context"
```

```
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context-3.1.xsd">
   <context:annotation-config/>
   <context:component-scan base-package="com.apress.prospring3.ch14.
validator"/>
</beans>
```

В листинге 14.18 показана программа для тестирования класса валидатора.

Листинг 14.18. Тестирование класса валидатора

```
package com.apress.prospring3.ch14.validator;

import java.util.List;

import org.springframework.context.support.GenericXmlApplicationContext;
import org.springframework.validation.BeanPropertyBindingResult;
import org.springframework.validation.ObjectError;
import org.springframework.validation.ValidationUtils;
import org.springframework.validation.Validator;

import com.apress.prospring3.ch14.domain.Contact;
public class SpringValidatorSample {
   public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:spring-validator-app-context.xml");
      ctx.refresh();

      Contact contact = new Contact();
      contact.setFirstName(null);
      contact.setLastName("Ho");

      Validator contactValidator = ctx.getBean(
         "contactValidator", Validator.class);
      BeanPropertyBindingResult result =
         new BeanPropertyBindingResult(contact, "Clarence");

      ValidationUtils.invokeValidator(contactValidator, contact, result);

      List<ObjectError> errors = result.getAllErrors();
      System.out.println("No of validation errors: " + errors.size());

      for (ObjectError error: errors) {
         System.out.println(error.getCode());
      }
   }
}
```

В листинге 14.18 конструируется объект `Contact` с именем, установленным в `null`. Затем из `ApplicationContext` извлекается валидатор. Для сохранения результата проверки достоверности создается экземпляр класса `BeanPropertyBindingResult`. Для выполнения проверки достоверности вызывается метод `ValidationUtils.invokeValidator()`. После этого осуществляется проверка на предмет возникновения ошибок.

Запуск этой программы дает следующий вывод:

```
No of validation errors: 1
firstName.empty
```

В процессе проверки достоверности произошла одна ошибка, код которой был корректно отображен.

Использование JSR-303: Bean Validation API

В версии Spring 3 появилась полноценная поддержка спецификации JSR-303, описывающей API-интерфейс проверки достоверности бинов (Bean Validation API). Этот интерфейс определяет набор ограничений в форме Java-аннотаций (например, `@NotNull`) в пакете `javax.validation.constraints`, которые могут быть применены к объектам предметной области. Вдобавок могут быть разработаны специальные валидаторы (скажем, валидаторы на уровне класса) и также применены с использованием аннотаций.

Применение Bean Validation API освобождает вас от привязки к специфическому поставщику службы проверки достоверности. Благодаря Bean Validation API, для реализации логики проверки достоверности в объектах предметной области вы можете использовать стандартные аннотации и API-интерфейс, не зная лежащего в основе поставщика службы проверки достоверности. Например, популярным таким поставщиком, совместимым с JSR-303, является Hibernate Validator (`http://hibernate.org/subprojects/validator`). В случае создания простого служебного проекта Spring JPA, как было описано в главе 10, зависимость для Hibernate Validator добавляется автоматически.

В Spring предлагается гладкая поддержка Bean Validation API. Основные функциональные возможности включают поддержку стандартных аннотаций JSR-303 для определения ограничений проверки достоверности, специальные валидаторы и средства конфигурирования проверки достоверности JSR-303 в `ApplicationContext`. В следующих разделах все эти возможности рассматриваются по очереди.

Определение ограничений проверки достоверности для свойств объектов

Начнем с применения ограничений проверки достоверности к свойствам объектов предметной области. В листинге 14.19 показан класс `Customer` с ограничениями проверки достоверности, примененными к атрибутам `firstName` и `customerType`.

Листинг 14.19. Класс `Customer`

```
package com.apress.prospring3.ch14.domain;

import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

public class Customer {

    @NotNull
    @Size(min=2, max=60)
    private String firstName;

    private String lastName;

    @NotNull
    private CustomerType customerType;

    private Gender gender;

    // Методы извлечения/установки не показаны.

    public boolean isIndividualCustomer() {
        return this.customerType.equals(CustomerType.INDIVIDUAL);
    }
}
```

Примененные ограничения проверки достоверности в листинге 14.19 выделены полужирным. К атрибуту `firstName` применены два ограничения. Первое из них управляется аннотацией `@NotNull`, которая указывает, что значение не должно быть равно `null`. Кроме того, аннотация `@Size` управляет длиной атрибута `firstName`. Ограничение `@NotNull` также применяется и к атрибуту `customerType`. В листингах 14.20 и 14.21 представлены классы `CustomerType` и `Gender`.

Листинг 14.20. Класс `CustomerType`

```
package com.apress.prospring3.ch14.domain;

public enum CustomerType {

   INDIVIDUAL("I"), CORPORATE("C");

   private String code;

   private CustomerType(String code) {
      this.code = code;
   }

   public String toString() {
      return this.code;
   }
}
```

Листинг 14.21. Класс `Gender`

```
package com.apress.prospring3.ch14.domain;

public enum Gender {

   MALE("M"), FEMALE("F");

   private String code;

   private Gender(String code) {
      this.code = code;
   }

   public String toString() {
      return this.code;
   }
}
```

Тип заказчика (класс `CustomerType`) указывает, является ли заказчик физическим лицом или компанией, а пол (класс `Gender`) должен применяться только к заказчикам — физическим лицам. В случае компаний пол должен быть `null`.

Конфигурирование поддержки проверки достоверности бинов в Spring

Чтобы сконфигурировать поддержку Bean Validation API в `ApplicationContext`, мы определяем экземпляр класса `org.springframework.validation.beanvalidation.LocalValidatorFactoryBean` в конфигурации Spring. Содержимое конфигурационного файла `jsr303-app-context.xml` показано в листинге 14.22.

Листинг 14.22. Конфигурирование Bean Validation API в Spring

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
   xmlns:context="http://www.springframework.org/schema/context"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context-3.1.xsd">

   <context:annotation-config/>

   <context:component-scan
      base-package="com.apress.prospring3.ch14.jsr303.service"/>

   <bean id="validator" class="org.springframework.validation.
beanvalidation.LocalValidatorFactoryBean"/>

</beans>
```

В листинге 14.22 видно, что требуется лишь указать объявление бина с классом `LocalValidatorFactoryBean`. По умолчанию Spring будет искать библиотеку Hibernate Validator в пути классов. А теперь создадим служебный класс для обеспечения проверки достоверности в классе `Customer`. Этот служебный класс по имени `MyBeanValidationService` представлен в листинге 14.23.

Листинг 14.23. Класс `MyBeanValidationService`

```
package com.apress.prospring3.ch14.jsr303.service;

import java.util.Set;

import javax.validation.ConstraintViolation;
import javax.validation.Validator;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.apress.prospring3.ch14.domain.Customer;

@Service("myBeanValidationService")
public class MyBeanValidationService {

   @Autowired
   private Validator validator;

   public Set<ConstraintViolation<Customer>> validateCustomer(Customer
customer) {
      return validator.validate(customer);
   }
}
```

В листинге 14.23 внедряется экземпляр `javax.validation.Validator` (обратите внимание на его отличие от интерфейса `Validator`, предоставляемого Spring, которым является `org.springframework.validation.Validator`). После определения бина `LocalValidatorFactoryBean` можно создавать обработчик для интерфейса `Validator` в любом месте приложения. Для осуществления проверки достоверности объекта POJO вызывается метод `Validator.validate()`. Результаты проверки достоверности возвращаются в виде списка (`List`) интерфейсов `ConstraintViolation<T>`.

Тестовая программа приведена в листинге 14.24.

Листинг 14.24. Класс Jsr303Sample

```
package com.apress.prospring3.ch14.jsr303;

import java.util.HashSet;
import java.util.Set;

import javax.validation.ConstraintViolation;

import org.springframework.context.support.GenericXmlApplicationContext;

import com.apress.prospring3.ch14.domain.Customer;
import com.apress.prospring3.ch14.domain.CustomerType;
import com.apress.prospring3.ch14.jsr303.service.MyBeanValidationService;

public class Jsr303Sample {

   public static void main(String[] args) {

      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:jsr303-app-context.xml");
      ctx.refresh();

      MyBeanValidationService myBeanValidationService =
         ctx.getBean("myBeanValidationService",
            MyBeanValidationService.class);

      Customer customer = new Customer();

      // Проверить базовые ограничения.
      customer.setFirstName("C");
      customer.setLastName("Ho");
      customer.setCustomerType(null);
      customer.setGender(null);

      validateCustomer(customer, myBeanValidationService);

   }

   private static void validateCustomer(Customer customer,
      MyBeanValidationService myBeanValidationService) {

      Set<ConstraintViolation<Customer>> violations =
         new HashSet<ConstraintViolation<Customer>>();
      violations = myBeanValidationService.validateCustomer(customer);

      listViolations(violations);
   }

   private static void listViolations(Set<ConstraintViolation<Customer>> violations) {

      System.out.println("No. of violations: " + violations.size());
      for (ConstraintViolation<Customer> violation: violations) {
         System.out.println("Validation error for property: " +
            violation.getPropertyPath()
            + " with value: " + violation.getInvalidValue()
            + " with error message: " + violation.getMessage());
      }
   }
}
```

Как показано в листинге 14.24, объект `Customer` конструируется со значениями `firstName` и `customerType`, нарушающими ограничения. В методе `validateCustomer()` вызывается метод `MyBeanValidationService.validateCustomer()`, который, в свою очередь, обращается к JSR-303 Bean Valiadation API.

Запуск этой программы дает следующий вывод:

```
No. of violations: 2
Validation error for property: firstName with value: C with error message:
size must be between 2 and 60
Validation error for property: customerType with value: null with error
message: may not be null
```

В этом выводе видно, что были обнаружены два нарушения, для которых отображены сообщения. Вдобавок здесь видно, что Hibernate Validator на основе аннотации сконструировал стандартные сообщения об ошибках проверки достоверности. Можно также предоставить собственные сообщения об ошибках, как будет показано в следующем разделе.

Создание специального валидатора

Кроме проверки достоверности на уровне атрибутов, можно применять проверку на уровне классов. Например, в случае класса `Customer` мы хотим гарантировать для отдельных заказчиков, что атрибуты `lastName` и `gender` не равны `null`. В такой ситуации для выполнения упомянутой проверки мы можем разработать специальный валидатор. В рамках Bean Validation API разработка специального валидатора является двухшаговым процессом. Первый шаг — создание типа аннотации для валидатора, как показано в листинге 14.25. Второй шаг — написание класса, который реализует логику проверки достоверности.

Листинг 14.25. Аннотация `CheckIndividualCustomer`

```
package com.apress.prospring3.ch14.jsr303.validator;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import javax.validation.Constraint;
import javax.validation.Payload;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Constraint(validatedBy=IndividualCustomerValidator.class)
@Documented
public @interface CheckIndividualCustomer {

   String message()
     default "Individual customer should have gender and last name defined";
   Class<?>[] groups() default {};
   Class<? extends Payload>[] payload() default {};

}
```

В листинге 14.25 с помощью `@Target(ElementType.TYPE)` указывается, что аннотация должна применяться только на уровне классов. Аннотация `@Constraint` определяет, что это валидатор, а в атрибуте `validatedBy` задается класс, предоставляющий логику проверки достоверности.

Внутри тела определены три атрибута (в форме методов), которые описаны ниже.

- Атрибут `message` определяет сообщение (или код ошибки) для возврата, когда ограничение нарушено. В аннотации также предоставляется сообщение по умолчанию.
- Атрибут `groups` указывает группу проверки достоверности, если это применимо. Валидаторы могут быть назначены в группы и выполнять проверку в конкретной группе.
- Атрибут `payload` задает дополнительные объекты полезной нагрузки (класса, реализующего интерфейс `javax.validation.Payload`). Этот атрибут позволяет присоединять дополнительную информацию к ограничению (например, объект полезной нагрузки может указывать серьезность нарушения ограничения).

В листинге 14.26 представлен класс `IndividualCustomerValidator`, который предоставляет логику проверки достоверности.

Листинг 14.26. Класс `IndividualCustomerValidator`

```
package com.apress.prospring3.ch14.jsr303.validator;

import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;

import com.apress.prospring3.ch14.domain.Customer;

public class IndividualCustomerValidator implements
   ConstraintValidator<CheckIndividualCustomer, Customer> {

   public void initialize(CheckIndividualCustomer constraintAnnotation) {
   }

   public boolean isValid(Customer customer,
      ConstraintValidatorContext context) {

      boolean result = true;

      if (customer.getCustomerType() != null &&
         (customer.isIndividualCustomer()
         && (customer.getLastName() == null ||
            customer.getGender() == null))
      ) {
         result = false;
      }

      return result;
   }

}
```

В листинге 14.26 класс `IndividualCustomerValidator` реализует интерфейс `ConstraintValidator<CheckIndividualCustomer, Customer>`, а это значит, что валидатор проверяет аннотацию `CheckIndividualCustomer` в классах `Customer`. Метод `isValid()` реализован, и лежащий в основе поставщик службы проверки достоверности (к примеру, Hibernate Validator) будет передавать этому методу проверяемый экземпляр. В коде метода мы удостоверяемся, что пользователь является физическим лицом, и затем в том, что свойства `lastName` и `gender` не равны `null`. Результатом является булевское значение, указывающее, как прошла проверка.

Чтобы включить проверку достоверности, примените аннотацию `@CheckIndividualCustomer` к классу `Customer`, как показано в листинге 14.27.

Листинг 14.27. Применение специальной проверки достоверности к классу `Customer`

```
package com.apress.prospring3.ch14.domain;

// Операторы импорта опущены.

@CheckIndividualCustomer
public class Customer {

   // Остальной код не показан.
}
```

Для тестирования специальной проверки достоверности добавьте код, приведенный в листинге 14.28, в метод `main()` класса `Jsr303Sample` в листинге 14.24.

Листинг 14.28. Тестирование применения специальной проверки достоверности к классу `Customer`

```
      // Проверить специальные ограничения.
      customer.setFirstName("Clarence");
      customer.setLastName("Ho");
      customer.setCustomerType(CustomerType.INDIVIDUAL);
      customer.setGender(null);

      validateCustomer(customer, myBeanValidationService);
```

Запуск этой программы дает следующий вывод (показан не полностью):

```
No. of violations: 1
Validation error for property:  with value: com.apress.prospring3.ch14.
domain.Customer@d3f136e with error message: Individual customer should have
gender and last name defined
```

В этом выводе видно, что проверяемое значение (являющееся объектом `Customer`) нарушает правило проверки достоверности для заказчиков — физических лиц, т.к. атрибут `gender` равен `null`. Обратите также внимание, что путь свойства пуст, поскольку это ошибка проверки достоверности на уровне класса.

Использование аннотации `@AssertTrue` для специальной проверки достоверности

Кроме реализации специального валидатора применить специальную проверку достоверности в Bean Validation API можно также с использованием аннотации `@AssertTrue`. Давайте посмотрим, как это работает.

В классе `Customer` удалите аннотацию `@CheckIndividualCustomer` и добавьте фрагмент кода из листинга 14.29.

Листинг 14.29. Применение аннотации `@AssertTrue` к классу `Customer`

```
package com.apress.prospring3.ch14.domain;

// Операторы импорта опущены.

public class Customer {

   // Остальной код не показан.
   @AssertTrue(message="Individual customer should have gender and last name
defined")
   private boolean isValidIndividualCustomer() {

      boolean result = true;
```

```
        if (getCustomerType() != null &&
            (isIndividualCustomer() && (gender == null || lastName == null)))
            result = false;

        return result;
    }
}
```

Как показано в листинге 14.29, в класс `Customer` добавлен метод `isValidIndividualCustomer()`, который аннотирован с помощью `@AssertTrue` (из пакета `javax.validation.constraints`). При запуске проверки достоверности поставщик будет инициировать проверку и проверять результат на предмет равенства `true`. Спецификация JSR-303 также предоставляет аннотацию `@AssertFalse` для проверки условий, которые должны быть `false`. Запустив тестовую программу (`Jsr303Sample`) еще раз, вы получите тот же самый вывод, что и в случае использования специального валидатора.

Соображения по поводу специальной проверки достоверности

Итак, какой подход должен использоваться для реализации специальной проверки достоверности в JSR-303: специальный валидатор или аннотация `@AssertTrue`? Как правило, метод с `@AssertTrue` проще в реализации, и правила проверки можно видеть прямо в коде объектов предметной области.

Тем не менее, при более сложной логике проверки достоверности (например, нужно внедрить служебный класс, получить доступ в базу данных и выполнить проверку с рядом допустимых значений) следует избрать подход со специальными валидаторами, поскольку совершенно не желательно внедрять объекты уровня обслуживания в объекты предметной области. Кроме того, специальные валидаторы могут быть многократно использованы во множестве объектов предметной области.

Какой API-интерфейс проверки достоверности использовать?

После обсуждения собственного интерфейса `Validator` в Spring и Bean Validation API осталось выяснить, какой из них должен использоваться в приложении? Безусловно, необходимо избрать JSR-303 по следующим причинам.

- JSR-303 — это стандарт JEE, который широко поддерживается многими платформами для интерфейсной и серверной частей (например, Spring, JPA 2, Spring MVC, GWT и т.д.).
- JSR-303 предоставляет стандартный API-интерфейс проверки достоверности и скрывает лежащего в основе поставщика, поэтому вы не привязываетесь к конкретному поставщику.
- Платформа Spring тесно интегрирована с JSR-303, начиная со своей версии 3. Например, в веб-контроллере Spring MVC можно аннотировать аргумент метода с помощью аннотации `@Valid` (из пакета `javax.validation`), и Spring запустит проверку достоверности JSR-303 во время процесса привязки данных. Более того, в конфигурации веб-приложения Spring MVC простой дескриптор `<mvc:annotation-driven/>` сконфигурирует Spring на автоматическое включение системы преобразования типов и форматирования полей Spring 3, а также поддержки JSR-303: Bean Validation.
- Если вы используете JPA 2, поставщик автоматически выполнит проверку достоверности JSR-303 для сущности перед ее сохранением, предоставляя еще один уровень защиты.

За более подробными сведениями по использованию JSR-303: Bean Validation с Hibernate Validator в качестве поставщика реализации обращайтесь в документацию Hibernate Validator по адресу `http://docs.jboss.org/hibernate/validator/4.2/reference/en-US/html`.

Проверка достоверности с преобразованиями типов и форматированием в примере приложения

В приложении SpringBlog будет использоваться новая система преобразования типов и форматирования Spring 3. Для проверки достоверности будет применяться JSR-303: Bean Validation API с Hibernate Validator в качестве поставщика службы проверки.

Чтобы посмотреть, как все эти технологии используются, давайте возьмем для примера основной сущностный класс `AbstractBlogPosting`, который является базовым классом для записей блога и комментариев к ним. Этот класс хранит такие общие свойства, как тема, тело, дата отправки и ряд других.

Ниже описаны главные правила проверки достоверности и форматирования свойств внутри сущностного класса `AbstractBlogPosting`.

- Поле `subject` является обязательным, и количество символов в нем должно быть от 10 до 50.
- Поле `body` является обязательным, и количество символов в нем должно быть от 10 до 200.
- Поле `postDate` будет автоматически заполняться приложением, и при его отображении в пользовательском интерфейсе будет использоваться формат `yyyy-MM-dd` (стандарный шаблон даты, определенный ISO).
- Дата создания и дата последней модификации будут автоматически заполняться приложением, и при их отображении в пользовательском интерфейсе будет использоваться формат `yyyy-MM-dd'T'HH:mm:ss.SSSZZ` (стандарный шаблон даты-времени, определенный ISO).

Для определения правила проверки достоверности и форматирования в классе `AbstractBlogPosting` к соответствующим свойствам будут применяться стандартные аннотации JSR-303. В листинге 14.30 приведен фрагмент кода из класса `AbstractBlogPosting` с примененными аннотациями.

Листинг 14.30. Класс `AbstractBlogPosting`

```
package com.apress.prospring3.springblog.domain;

import static javax.persistence.GenerationType.IDENTITY;

import java.io.Serializable;

import javax.persistence.*;
import javax.validation.constraints.*;

import org.hibernate.annotations.Type;
import org.joda.time.DateTime;
import org.springframework.data.domain.Auditable;
import org.springframework.format.annotation.DateTimeFormat;
import org.springframework.format.annotation.DateTimeFormat.ISO;

@MappedSuperclass
public abstract class AbstractBlogPosting
   implements BlogPosting, Auditable<String, Long>, Serializable {
```

```
    // Остальной код не показан.

    @NotNull
    @Size(min=10, max=2000, message="{validation.posting.body.Size.message}")
    @Column(name = "BODY")
    public String getBody() {
        return body;
    }

    @Column(name = "POST_DATE")
    @Type(type="org.joda.time.contrib.hibernate.PersistentDateTime")
    @DateTimeFormat(iso=ISO.DATE)
    public DateTime getPostDate() {
        return postDate;
    }

    @NotNull
    @Size(min=10, max=50, message="{validation.posting.subject.Size.message}")
    @Column(name = "SUBJECT")
    public String getSubject() {
        return subject;
    }

    @Column(name = "CREATED_DATE")
    @Type(type="org.joda.time.contrib.hibernate.PersistentDateTime")
    @DateTimeFormat(iso=ISO.DATE_TIME)
    public DateTime getCreatedDate() {
        return this.createdDate;
    }

    @Column(name = "LAST_MODIFIED_DATE")
    @Type(type="org.joda.time.contrib.hibernate.PersistentDateTime")
    @DateTimeFormat(iso=ISO.DATE_TIME)
    public DateTime getLastModifiedDate() {
        return this.lastModifiedDate;
    }
}
```

В листинге 14.30 видно, что к свойствам `body` и `subject` применены аннотации `@NotNull` и `@Size`, определенные JSR-303. В аннотации `@Size` вместо жесткого кодирования текста сообщения в атрибуте `message` мы предоставили ключ сообщения, и Hibernate Validator извлечет текст шаблона сообщения из пакета ресурсов сообщений (именем файла свойств пакета сообщений по умолчанию является `ValidationMessages.properties`), поддерживая интернационализацию. В листинге 14.31 показаны два сообщения, определенные в листинге 14.30.

Листинг 14.31. Файл `ValidationMessages.properties`

```
validation.posting.body.Size.message=body must be between {min} and {max}
characters
validation.posting.subject.Size.message=subject must be between {min} and
{max} characters
```

В шаблонах сообщений из листинга 14.31 определены переменные `min` и `max`, которые будут автоматически заменяться соответствующими атрибутами внутри аннотации `@Size`. Определяя сообщения подобным образом, легко реализовать интернационализацию. Например, для поддержки традиционного китайского языка,

используемого в Гонконге (с кодом локали zh_HK), понадобится лишь подготовить файл ValidationMessages_zh_HK.properties с сообщениями на этом языке. Кроме того, как показано в листинге 14.30, к свойствам postDate, createdDate и lastModifiedDate применяется аннотация @DateTimeFormat. Указанные выше аннотации принадлежат системе преобразования типов и форматирования Spring 3. Для свойства postDate определен шаблон ISO.DATE, которому соответствует формат yyyy-MM-dd. Для свойств createdDate и LastModifiedDate определен шаблон ISO.DATE_TIME, которому соответствует формат yyyy-MM-dd'T'HH:mm:ss.SSSZZ.

Включение поддержки проверки достоверности, преобразования типов и форматирования в Spring 3 осуществляется легко.

В примере приложения мы будем применять Spring MVC на уровне презентаций, а именно — в контексте WebApplicationContext сервлета диспетчера, поэтому нам нужно только определить дескриптор <mvc:annotation-driven>, а Spring сделает всю остальную работу. Дополнительные сведения об этом будут даны в главе 17, когда речь пойдет о проверке достоверности данных при разработке приложений.

Более подробно об использовании проверки достоверности, преобразования типов и форматирования в примере приложения рассказывается в главе 21.

Резюме

В этой главе была рассмотрена система преобразования типов Spring 3, а также SPI-интерфейс форматировщиков полей. Вы узнали, как использовать новую систему преобразования типов для преобразований между произвольными типами в дополнение к поддержке редакторов свойств.

Кроме того, мы обсудили поддержку проверки достоверности в Spring, интерфейс Validator из Spring и рекомендуемую поддержку JSR-303: Bean Validation в версии Spring 3.

ГЛАВА 15

Планирование задач в Spring

Планирование задач является общей функциональной возможностью в корпоративных приложениях. Планирование задач в основном состоит из трех частей: задачи (представляющей собой порцию бизнес-логики, которая должна запускаться в определенное время или на регулярной основе), триггера (указывающего условие, при удовлетворении которого задача должна выполняться) и планировщика (который запускает задачу на основе информации, полученной от триггера).

В частности, в этой главе будут рассматриваться следующие темы.

- *Планирование задач в Spring.* Мы обсудим поддержку планирования задач в Spring, уделяя особое внимание абстракции `TaskScheduler`, которая появилась в версии Spring 3. Мы также опишем сценарии планирования, такие как планирование с фиксированным интервалом и с помощью выражения `cron`.
- *Асинхронное выполнение задач.* Мы покажем, как использовать новую аннотацию `@Async` в Spring 3 для выполнения задач асинхронным образом.

Создание проекта для примеров в STS

Начнем с создания проекта для примеров, рассматриваемых в этой главе. Создайте новый шаблонный проект Spring в STS, выбрав вариант Simple Spring JPA Utility Project (Простой служебный проект Spring JPA). Причина выбора этого шаблона проекта заключается в том, что мы намерены разработать пример задания, которое будет обновлять данные в СУРБД серверной части.

После создания в проект нужно добавить обязательные зависимости, описанные в табл. 15.1. Удостоверьтесь, что используется Spring 3.1 с JDK 6.

Таблица 15.1. Зависимости Maven для планирования задач

Идентификатор группы	Идентификатор артефакта	Версия	Описание
`org.springframework.data`	`spring-data-jpa`	1.0.1.RELEASE	Библиотека Spring Data JPA
`joda-time`	`joda-time`	2.0	API-интерфейс для работы с датой и временем под названием Joda-Time (`http://joda-time.sourceforge.net/`), который используется в Spring Data JPA. В этой главе мы будем применять его в объектах предметной области

Окончание табл. 15.1

Идентификатор группы	Идентификатор артефакта	Версия	Описание
joda-time	joda-time-hibernate	1.3	Библиотека Joda-Time для интеграции с Hibernate, обеспечивающая сохранение типов данных даты и времени
com.google.guava	guava	10.0.1	Полезные вспомогательные классы
org.slf4j	slf4j-log4j12	1.6.1	Библиотека ведения журналов (www.slf4j.org), которая будет использоваться в примерах этой главы. Данная библиотека соединяет ведение журнала SLF4J с библиотекой log4j

Реализация планирования задач в Spring

Корпоративные приложения часто нуждаются в планировании задач. Во многих приложениях разнообразные задачи (вроде отправки заказчикам уведомлений по электронной почте, запуска заданий в конце дня, выполнение обслуживания данных, обновление данных в пакетах и т.п.) должны планироваться к запуску на регулярной основе, либо через фиксированные интервалы (например, каждый час), либо по заданному расписанию (скажем, каждый вечер в 20:00 с понедельника по пятницу). Как упоминалось ранее, планирование задач состоит из трех частей: определение расписания (триггер), выполнение задачи (планировщик) и сама задача.

Для инициирования запуска задачи в Spring-приложении имеется множество разных способов. Один из них предусматривает запуск задания внешне из системы планирования, которая уже существует в среде развертывания приложений. Например, на многих предприятиях для планирования задач используются коммерческие системы, такие как Ctrl-M или CA Autosys. Если приложение выполняется на платформе Linux/Unix, можно пользоваться планировщиком crontab. Запуск заданий можно осуществлять отправкой запроса RESTful-WS к Spring-приложению, в результате чего контроллер Spring MVC инициирует выполнение задания.

Другой способ заключается в применении поддержки планирования задач, предлагаемой Spring. Для планирования задач в Spring доступны три варианта.

- *Поддержка JDK-объекта* `Timer`. При планировании задач в Spring поддерживается объект `Timer` из JDK.
- *Интеграция с Quartz*. Платформа Spring интегрирована с Quartz Scheduler (www.quartz-scheduler.org) — популярной библиотекой планирования с открытым кодом.
- *Абстракция* `TaskScheduler`, *встроенная в Spring*. В версии Spring 3 появилась абстракция `TaskScheduler`, которая предлагает простой способ планирования задач и поддерживает наиболее типичные требования.

В этом разделе мы сосредоточим внимание на использовании абстракции `TaskScheduler` для планирования задач.

Введение в абстракцию `TaskScheduler`

В Spring-абстракции `TaskScheduler` имеются три главных участника.

- *Интерфейс `Trigger`*. Интерфейс `org.springframework.scheduling.Trigger` предоставляет поддержку для определения механизма запуска. В Spring доступны две реализации `Trigger`. Класс `CronTrigger` поддерживает запуск на основе выражения `cron`, а класс `PeriodicTrigger` — запуск на основе начальной задержки и затем фиксированного интервала.
- *Задача*. Задача — это порция бизнес-логики, запуск которой необходимо запланировать. В Spring задача может быть указана как метод внутри любого бина Spring.
- *Интерфейс `TaskScheduler`*. Интерфейс `org.springframework.scheduling.TaskScheduler` предоставляет поддержку для планирования задач. В Spring доступны три класса реализации интерфейса `TaskScheduler`. Класс `TimerManagerTaskScheduler` (из пакета `org.springframework.scheduling.commonj`) является оболочкой для интерфейса `commonj.timers.TimerManager` из CommonJ, который обычно используется в коммерческих серверах приложений JEE, таких как WebSphere, WebLogic и т.д. Классы `ConcurrentTaskScheduler` и `ThreadPoolTaskScheduler` (оба из пакета `org.springframework.scheduling.concurrent`) представляют собой оболочки для класса `java.util.concurrent.ScheduledThreadPoolExecutor`. Оба класса поддерживают запуск задач из разделяемого пула потоков.

На рис. 15.1 показаны отношения между интерфейсом `Trigger`, интерфейсом `TaskScheduler` и задачей (которая реализует интерфейс `java.lang.Runnable`).

Планировать задачи с применением абстракции `TaskScheduler` из Spring можно двумя способами. Один из них предусматривает использование пространства имен `task` в XML-конфигурации Spring, а другой — аннотаций. Давайте рассмотрим оба эти способа.

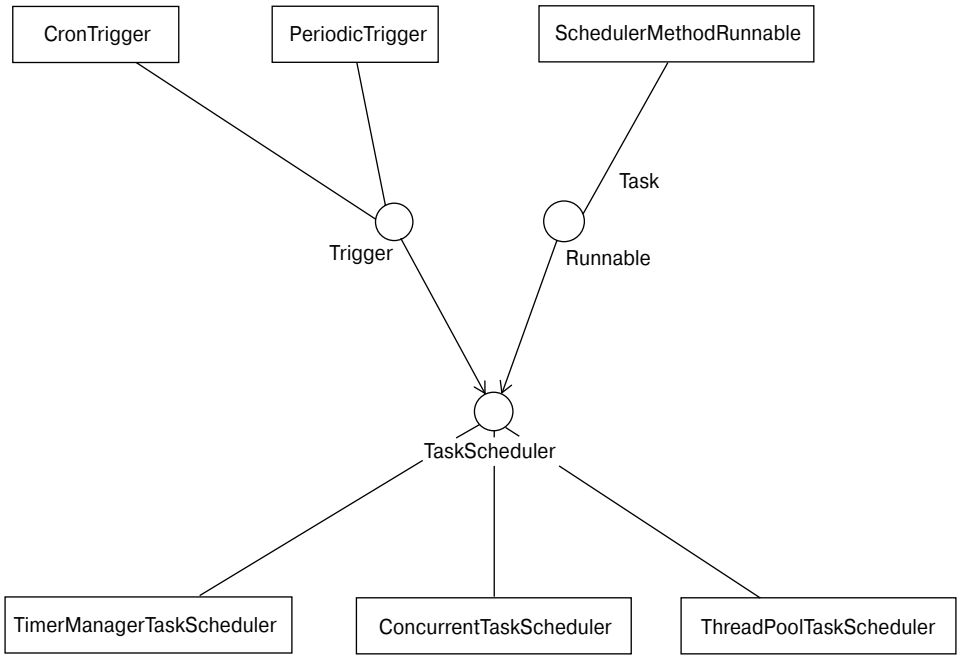


Рис. 15.1. Отношения между триггером, задачей и планировщиком

Пример задачи

Для демонстрации планирования задач в Spring реализуем сначала простое задание, а именно — приложение, обслуживающее базу данных с информацией об автомобилях. В листинге 15.1 показан класс `Car`, который реализован как сущностный класс JPA.

Листинг 15.1. Класс `Car`

```
package com.apress.prospring3.ch15.domain;

import static javax.persistence.GenerationType.IDENTITY;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Table;
import javax.persistence.Version;

import org.hibernate.annotations.Type;
import org.joda.time.DateTime;

@Entity
@Table(name="car")
public class Car {

   private Long id;

   private String licensePlate;

   private String manufacturer;

   private DateTime manufactureDate;

   private int age;

   private int version;

   @Id
   @GeneratedValue(strategy = IDENTITY)
   @Column(name = "ID")
   public Long getId() {
      return id;
   }

   @Column(name="LICENSE_PLATE")
   public String getLicensePlate() {
      return licensePlate;
   }

   @Column(name="MANUFACTURER")
   public String getManufacturer() {
      return manufacturer;
   }

   @Column(name="MANUFACTURE_DATE")
   @Type(type="org.joda.time.contrib.hibernate.PersistentDateTime")
   public DateTime getManufactureDate() {
      return manufactureDate;
   }

   @Column(name="AGE")
   public int getAge() {
      return age;
   }
```

```
    @Version
    public int getVersion() {
        return version;
    }

    // Методы установки не показаны.

    public String toString() {
        return "License: " + licensePlate + " - Manufacturer: " + manufacturer
            + " - Manufacture Date: " + manufactureDate + " - Age: " + age;
    }
}
```

В листингах 15.2 и 15.3 приведен код сценариев создания таблицы (`schema.sql`) и заполнения ее тестовыми данными (`test-data.sql`) для сущностного класса `Car`.

Листинг 15.2. Сценарий для создания таблицы

```
DROP TABLE IF EXISTS CONTACT;

CREATE TABLE CAR (
     ID INT NOT NULL AUTO_INCREMENT
   , LICENSE_PLATE VARCHAR(20) NOT NULL
   , MANUFACTURER VARCHAR(20) NOT NULL
   , MANUFACTURE_DATE DATE NOT NULL
   , AGE INT NOT NULL DEFAULT 0
   , VERSION INT NOT NULL DEFAULT 0
   , UNIQUE UQ_CAR_1 (LICENSE_PLATE)
   , PRIMARY KEY (ID)
);
```

Листинг 15.3. Сценарий для заполнения таблицы тестовыми данными

```
insert into car (license_plate, manufacturer, manufacture_date)
  values ('LICENSE-1001', 'Ford', '1980-07-30');
insert into car (license_plate, manufacturer, manufacture_date)
  values ('LICENSE-1002', 'Toyota', '1992-12-30');
insert into car (license_plate, manufacturer, manufacture_date)
  values ('LICENSE-1003', 'BMW', '2003-1-6');
```

А теперь определим уровень обслуживания для сущности `Car`. Мы будем использовать проект Spring Data JPA и его поддержку абстракции репозитория. В листинге 15.4 представлен интерфейс `CarRepository`.

Листинг 15.4. Интерфейс `CarRepository`

```
package com.apress.prospring3.ch15.repository;

import org.springframework.data.repository.CrudRepository;

import com.apress.prospring3.ch15.domain.Car;

public interface CarRepository extends CrudRepository<Car, Long> {
}
```

Здесь нет ничего особенного; мы просто реализовали интерфейс `CrudRepository <Car,Long>`. В листингах 15.5 и 15.6 показан интерфейс `CarService` его класс реализации `CarServiceImpl`.

Листинг 15.5. Интерфейс `CarService`

```
package com.apress.prospring3.ch15.service;

import java.util.List;
import com.apress.prospring3.ch15.domain.Car;

public interface CarService {
   public List<Car> findAll();
   public Car save(Car car);
   public void updateCarAgeJob();
}
```

Листинг 15.6. Класс `CarServiceImpl`

```
package com.apress.prospring3.ch15.service.jpa;

// Операторы импорта опущены.
@Service("carService")
@Repository
@Transactional
public class CarServiceImpl implements CarService {
   final Logger logger = LoggerFactory.getLogger(CarServiceImpl.class);

   @Autowired
   CarRepository carRepository;

   @Transactional(readOnly=true)
   public List<Car> findAll() {
      return Lists.newArrayList(carRepository.findAll());
   }

   public Car save(Car car) {
      return carRepository.save(car);
   }

   public void updateCarAgeJob() {
      // Обновить возраст автомобиля.
      List<Car> cars = findAll();
      DateTime currentDate = DateTime.now();
      int age;
      logger.info("");
      logger.info("Car age update job started");
      for (Car car: cars) {
         age = new Period(car.getManufactureDate(), currentDate,
                 PeriodType.years()).getYears();
         car.setAge(age);
         save(car);
         logger.info("Car age update--- " + car);
      }
      logger.info("Car age update job completed successfully");
      logger.info("");
   }
}
```

В коде определены два метода; один из них извлекает информацию обо всех автомобилях, а другой сохраняет обновленный объект `Car`. Третий метод, `updateCarAgeJob()`, представляет собой задание, которое должно запускаться на регулярной основе и обновлять возраст автомобиля, имея дату его изготовления и текущую дату.

В листинге 15.7 представлена конфигурация Spring для поддержки рассматриваемого примера приложения с автомобилями (car-job-app-context.xml).

Листинг 15.7. Файл car-job-app-context.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns=http://www.springframework.org/schema/beans
   xmlns:context="http://www.springframework.org/schema/context"
   xmlns:jdbc="http://www.springframework.org/schema/jdbc"
   xmlns:tx="http://www.springframework.org/schema/tx"
   xmlns:jpa="http://www.springframework.org/schema/data/jpa"
   xmlns:repository="http://www.springframework.org/schema/data/repository"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context-3.1.xsd
      http://www.springframework.org/schema/jdbc
      http://www.springframework.org/schema/jdbc/spring-jdbc-3.1.xsd
      http://www.springframework.org/schema/tx
      http://www.springframework.org/schema/tx/spring-tx-3.1.xsd
      http://www.springframework.org/schema/data/jpa
      http://www.springframework.org/schema/data/jpa/spring-jpa-1.0.xsd
      http://www.springframework.org/schema/data/repository
      http://www.springframework.org/schema/data/repository/
spring-repository-1.0.xsd">

   <jdbc:embedded-database id="dataSource" type="H2">
      <jdbc:script location="classpath:schema.sql"/>
      <jdbc:script location="classpath:test-data.sql"/>
   </jdbc:embedded-database>
   <bean id="transactionManager"
      class="org.springframework.orm.jpa.JpaTransactionManager">
      <property name="entityManagerFactory" ref="emf"/>
   </bean>

   <tx:annotation-driven transaction-manager="transactionManager" />

   <bean id="emf"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
      <property name="dataSource" ref="dataSource" />
      <property name="jpaVendorAdapter">
        <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter" />
      </property>
      <property name="packagesToScan"
         value="com.apress.prospring3.ch15.domain"/>
      <property name="jpaProperties">
         <props>
            <prop key="hibernate.dialect">
               org.hibernate.dialect.H2Dialect
            </prop>
            <prop key="hibernate.max_fetch_depth">3</prop>
            <prop key="hibernate.jdbc.fetch_size">50</prop>
            <prop key="hibernate.jdbc.batch_size">10</prop>
            <prop key="hibernate.show_sql">true</prop>
         </props>
      </property>
   </bean>
```

```
   <context:annotation-config/>

   <jpa:repositories base-package="com.apress.prospring3.ch15.repository"
      entity-manager-factory-ref="emf"
      transaction-manager-ref="transactionManager"/>

   <context:component-scan
      base-package="com.apress.prospring3.ch15.service.jpa" />
</beans>
```

Эта конфигурация должна выглядеть знакомой. А теперь займемся планированием задания по обновлению возраста автомобилей в Spring.

Планирование задач с использованием пространства имен `task`

Подобно другим пространствами имен в Spring, пространство имен `task` предоставляет упрощенную конфигурацию для планируемых задач с применением абстракции `TaskScheduler`, доступной в Spring.

Пространство имен `task` для планирования задач используется очень просто. В листинге 15.8 приведено содержимое конфигурационного файла `task-namespace-app-context.xml`.

Листинг 15.8. Конфигурация Spring, использующая пространство имен `task`

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns=http://www.springframework.org/schema/beans
   xmlns:task="http://www.springframework.org/schema/task"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
      http://www.springframework.org/schema/task
      http://www.springframework.org/schema/task/spring-task-3.1.xsd">

   <import resource="car-job-app-context.xml"/>

   <task:scheduler id="myScheduler" pool-size="10"/>

   <task:scheduled-tasks scheduler="myScheduler">
      <task:scheduled ref="carService" method="updateCarAgeJob"
         fixed-delay="10000"/>
   </task:scheduled-tasks>

</beans>
```

Как показано в листинге 15.8, был импортирован контекст для приложения работы с автомобилями. Натолкнувшись на дескриптор `<task:scheduler>`, Spring создаст экземпляр класса `ThreadPoolTaskScheduler`, при этом атрибут `pool-size` задает размер пула потоков, который планировщик может использовать. Внутри дескриптора `<task:scheduled-tasks>` модет быть запланирована одна или большее число задач. В дескрипторе `<task:scheduled>` задача может ссылаться на бин Spring (в данном случае это бин `carService`) и специфический метод этого бина (в рассматриваемом примере это метод `updateCarAgeJob()`). Атрибут `fixed-delay` указывает Spring на необходимость создания `PeriodicTrigger` как реализации `Trigger` для `TaskScheduler`.

В листинге 15.9 приведен код программы для тестирования планирования задач.

Листинг 15.9. Тестирование планирования задач в Spring

```
package com.apress.prospring3.ch15.schedule;

import org.springframework.context.support.GenericXmlApplicationContext;

public class ScheduleTaskSample {

   public static void main(String[] args) {

      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:task-namespace-app-context.xml");
      ctx.refresh();

      while (true) {
      }
   }
}
```

Класс `ScheduleTaskSample` довольно прост; он лишь загружает `ApplicationContext` и организует бесконечный цикл. Если приложение развернуто в среде сервера приложений, планировщик продолжит выполняться.

Запуск этой программы даст следующий вывод, получаемый каждые десять секунд:

```
INFO [com.apress.prospring3.ch15.service.jpa.CarServiceImpl] - <>
INFO [com.apress.prospring3.ch15.service.jpa.CarServiceImpl] -
<Car age update job started>
INFO [com.apress.prospring3.ch15.service.jpa.CarServiceImpl] -
<Car age update--- License: LICENSE-1001 — Manufacturer: Ford -
Manufacture Date: 1980-07-30T00:00:00.000+08:00 - Age: 31>
INFO [com.apress.prospring3.ch15.service.jpa.CarServiceImpl] -
<Car age update--- License: LICENSE-1002 — Manufacturer: Toyota -
Manufacture Date: 1992-12-30T00:00:00.000+08:00 - Age: 18>
INFO [com.apress.prospring3.ch15.service.jpa.CarServiceImpl] -
<Car age update--- License: LICENSE-1003 — Manufacturer: BMW -
Manufacture Date: 2003-01-06T00:00:00.000+08:00 - Age: 8>
INFO [com.apress.prospring3.ch15.service.jpa.CarServiceImpl] -
<Car age update job completed successfully>
INFO [com.apress.prospring3.ch15.service.jpa.CarServiceImpl] - <>
```

В этом выводе видно, что атрибуты `age` автомобилей были успешно обновлены.

Кроме фиксированного интервала доступен более гибкий механизм планирования, предусматривающий использование выражения `cron`. Измените следующую строку в листинге 15.8:

```
<task:scheduled ref="carService" method="updateCarAgeJob"
fixed-delay="10000"/>
```

такой строкой:

```
<task:scheduled ref="carService" method="updateCarAgeJob" cron="0 * * * * *"/>
```

После такого изменения снова запустите тестовую программу с классом `ScheduleTaskSample` и вы увидите, что задание будет выполняться каждую минуту. На странице руководства по `CronTrigger` из библиотеки Quartz Scheduler (`www.quartz-scheduler.org/documentation/quartz-2.1.x/tutorials/crontrigger`) приведено детальное описание структуры и показаны примеры выражений `cron`.

Планирование задач с использованием аннотаций

Еще один способ для планирования задач с применением абстракции `TaskScheduler` из Spring предполагает использование аннотаций. Для этой цели в Spring предусмотрена аннотация `@Scheduled`.

Чтобы включить поддержку аннотаций для планирования задач, необходимо предоставить дескриптор `<task:annotation-driven>` в XML-конфигурации Spring. Содержимое конфигурационного файла `task-annotation-app-context.xml` показано в листинге 15.10.

Листинг 15.10. Конфигурация Spring для планирования на основе аннотаций

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns=http://www.springframework.org/schema/beans
   xmlns:task="http://www.springframework.org/schema/task"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
      http://www.springframework.org/schema/task
      http://www.springframework.org/schema/task/spring-task-3.1.xsd">

   <import resource="car-job-app-context.xml"/>

   <task:scheduler id="myScheduler" pool-size="10"/>

   <task:annotation-driven scheduler="myScheduler"/>

</beans>
```

В листинге 15.10 с помощью дескриптора `<task:annotation-driven>` включается поддержка планирования на основе аннотаций, с указанием в атрибуте `scheduler` ссылки на бин `myScheduler`.

Чтобы запланировать выполнение конкретного метода в бине Spring, просто аннотируйте метод посредством `@Scheduled` и передайте требования к планированию. В листинге 15.11 приведен фрагмент кода из переделанного класса `CarServiceImpl`.

Листинг 15.11. Переделанный класс `CarServiceImpl`

```
package com.apress.prospring3.ch15.service.jpa;

import org.springframework.scheduling.annotation.Scheduled;

// Остальной код не показан.

public class CarServiceImpl implements CarService {

   @Scheduled(fixedDelay=10000)
   //@Scheduled(fixedRate=10000)
   //@Scheduled(cron="0 * * * * *")
   public void updateCarAgeJob() {

      // Остальной код не показан.
   }
}
```

Тестовая программа представлена в листинге 15.12.

Листинг 15.12. Тестирование планирования на основе аннотаций

```
package com.apress.prospring3.ch15.schedule;

import org.springframework.context.support.GenericXmlApplicationContext;

public class ScheduleTaskAnnotationSample {

   public static void main(String[] args) {

      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:task-annotation-app-context.xml");
      ctx.refresh();

      while (true) {
      }
   }
}
```

Выполнение этой программы даст тот же самый вывод, что и пример использования пространства имен `task`. Меняя атрибут в аннотации `@Scheduled`, можно опробовать разные механизмы запуска (`fixedDelay`, `fixedRate`, `cron`). Протестируйте это самостоятельно.

Асинхронное выполнение задач в Spring

В версии Spring 3.0 также поддерживается использование аннотаций для выполнения задачи асинхронным образом. Необходимо просто аннотировать метод с помощью `@Async`.

Чтобы увидеть это в действии, рассмотрим небольшой пример. В листингах 15.13 и Listing 15.14 показан интерфейс `AsyncService` и класс реализации `AsyncServiceImpl`.

Листинг 15.13. Интерфейс `AsyncService`

```
package com.apress.prospring3.ch15.service;
import java.util.concurrent.Future;
public interface AsyncService {
   public void asyncTask();
   public Future<String> asyncWithReturn(String name);
}
```

Листинг 15.14. Класс `AsyncServiceImpl`

```
package com.apress.prospring3.ch15.service.async;

import java.util.concurrent.Future;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.scheduling.annotation.Async;
import org.springframework.scheduling.annotation.AsyncResult;
import org.springframework.stereotype.Service;

import com.apress.prospring3.ch15.service.AsyncService;

@Service("asyncService")
public class AsyncServiceImpl implements AsyncService {

   final Logger logger = LoggerFactory.getLogger(AsyncServiceImpl.class);
```

```
    @Async
    public void asyncTask() {
        logger.info("Start execution of async. task");
        try {
            Thread.sleep(10000);
        } catch (Exception ex) {
            ex.printStackTrace();
        }
        logger.info("Complete execution of async. task");
    }
    @Async
    public Future<String> asyncWithReturn(String name) {
        logger.info("Start execution of async. task with return");
        try {
            Thread.sleep(5000);
        } catch (Exception ex) {
            ex.printStackTrace();
        }
        logger.info("Complete execution of async. task with return");
        return new AsyncResult<String>("Hello: " + name);
    }
}
```

В интерфейсе `AsyncService` определены два метода. Метод `asyncTask()` — это простая задача, которая записывает информацию в журнал.

Метод `asyncWithReturn()` принимает аргумент типа `String` и возвращает экземпляр интерфейса `java.util.concurrent.Future<V>`. Обратите внимание на код, выделенный полужирным; после завершения `asyncWithReturn()` результат хранится в экземпляре класса `org.springframework.scheduling.annotation.AsyncResult<V>`, который реализует интерфейс `Future<V>` и может использоваться для извлечения результата выполнения в более позднее время.

В листинге 15.15 показано содержимое файла конфигурации Spring (`async-appcontext.xml`).

Листинг 15.15. Конфигурация Spring для асинхронной задачи

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:task="http://www.springframework.org/schema/task"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-3.1.xsd
        http://www.springframework.org/schema/task
        http://www.springframework.org/schema/task/spring-task-3.1.xsd">
    <context:annotation-config/>
    <context:component-scan
        base-package="com.apress.prospring3.ch15.service.async"/>
    <task:annotation-driven />
</beans>
```

Дескриптор `<task:annotation-driven />` в листинге 15.15 необходим для поддержки аннотации `@Async`. Тестовая программа представлена в листинге 15.16.

Листинг 15.16. Тестирование асинхронной задачи

```
package com.apress.prospring3.ch15.schedule;

import java.util.concurrent.Future;

import org.springframework.context.support.GenericXmlApplicationContext;

import com.apress.prospring3.ch15.service.AsyncService;

public class AsyncTaskSample {

   public static void main(String[] args) {

      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:async-app-context.xml");
      ctx.refresh();

      AsyncService asyncService = ctx.getBean(
         "asyncService", AsyncService.class);

      for (int i = 0; i < 5; i++)
         asyncService.asyncTask();

      Future<String> result1 = asyncService.asyncWithReturn("Clarence");
      Future<String> result2 = asyncService.asyncWithReturn("John");
      Future<String> result3 = asyncService.asyncWithReturn("Robert");

      try {
         Thread.sleep(6000);

         System.out.println("Result1: " + result1.get());
         System.out.println("Result2: " + result2.get());
         System.out.println("Result3: " + result3.get());
      } catch (Exception ex) {
         ex.printStackTrace();
      }
   }
}
```

В листинге 15.16 мы вызываем метод `asyncTask()` пять раз и метод `asyncWithReturn()` три раза с разными аргументами, а затем извлекаем результат после паузы в шесть секунд.

Запуск этой программы дает следующий вывод:

```
2011-11-23 17:46:16,276 INFO [com.apress.prospring3.ch15.service.async.
AsyncServiceImpl] — <Start execution of async. task>
2011-11-23 17:46:16,277 INFO [com.apress.prospring3.ch15.service.async.
AsyncServiceImpl] — <Start execution of async. task>
2011-11-23 17:46:16,277 INFO [com.apress.prospring3.ch15.service.async.
AsyncServiceImpl] — <Start execution of async. task>
2011-11-23 17:46:16,277 INFO [com.apress.prospring3.ch15.service.async.
AsyncServiceImpl] — <Start execution of async. task>
2011-11-23 17:46:16,277 INFO [com.apress.prospring3.ch15.service.async.
AsyncServiceImpl] — <Start execution of async. task>
2011-11-23 17:46:16,277 INFO [com.apress.prospring3.ch15.service.async.
AsyncServiceImpl] — <Start execution of async. task with return>
2011-11-23 17:46:16,277 INFO [com.apress.prospring3.ch15.service.async.
AsyncServiceImpl] — <Start execution of async. task with return>
```

```
2011-11-23 17:46:16,277 INFO [com.apress.prospring3.ch15.service.async.
AsyncServiceImpl] — <Start execution of async. task with return>
2011-11-23 17:46:21,278 INFO [com.apress.prospring3.ch15.service.async.
AsyncServiceImpl] — <Complete execution of async. task with return>
2011-11-23 17:46:21,278 INFO [com.apress.prospring3.ch15.service.async.
AsyncServiceImpl] — <Complete execution of async. task with return>
2011-11-23 17:46:21,278 INFO [com.apress.prospring3.ch15.service.async.
AsyncServiceImpl] — <Complete execution of async. task with return>
Result1: Hello: Clarence
Result2: Hello: John
Result3: Hello: Robert
2011-11-23 17:46:26,277 INFO [com.apress.prospring3.ch15.service.async.
AsyncServiceImpl] — <Complete execution of async. task>
2011-11-23 17:46:26,278 INFO [com.apress.prospring3.ch15.service.async.
AsyncServiceImpl] — <Complete execution of async. task>
2011-11-23 17:46:26,278 INFO [com.apress.prospring3.ch15.service.async.
AsyncServiceImpl] — <Complete execution of async. task>
2011-11-23 17:46:26,278 INFO [com.apress.prospring3.ch15.service.async.
AsyncServiceImpl] — <Complete execution of async. task>
2011-11-23 17:46:26,278 INFO [com.apress.prospring3.ch15.service.async.
AsyncServiceImpl] — <Complete execution of async. task>
```

В этом выводе видно, что все вызовы начались в одно и то же время. Три вызова со значениями возврата завершились первыми и отобразились в консольном выводе. Наконец, были завершены пять вызовов метода `asyncTask()`.

Планирование задач в примере приложения

В приложении SpringBlog мы будем хранить записи хронологии для записей блога и связанных с ними комментариев в целях аудита. Тем не менее, для экономии пространства в базе данных мы также решили хранить только записи хронологии для 30 дней. Чтобы удовлетворить этому требованию, мы воспользуемся поддержкой абстракции `TaskScheduler` в Spring 3 и реализуем запланированное задание для удаления записей аудита, которые старее 30 дней. Это задание будет запускаться ежесуточно в полночь.

Для планирования задач мы будем применять стиль аннотаций. Первым делом мы определим интерфейс для задания по обслуживанию, который показан в листинге 15.17.

Листинг 15.17. Интерфейс `HousekeepingService`

```
package com.apress.prospring3.springblog.service;

public interface HousekeepingService {

    /**
     * Запланированное задание для удаления записей аудита.
     */
    public void auditPurgeJob();

}
```

В интерфейсе `HousekeepingService` определен метод `auditPurgeJob()` для задачи обслуживания записей аудита. В листинге 15.18 представлен класс реализации с примененной аннотацией `@Scheduled`.

Листинг 15.18. Класс HousekeepingServiceImpl

```
package com.apress.prospring3.springblog.service.jpa;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Repository;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import com.apress.prospring3.springblog.service.HousekeepingService;

@Service("housekeepingService")
@Repository
@Transactional
public class HousekeepingServiceImpl implements HousekeepingService {

   @Value("${audit.record.history.days}")
   private int auditHistoryDays;

   @Scheduled(cron="0 0 0 * * ?")
   public void auditPurgeJob() {
      // Логика удаления записей аудита.
   }
}
```

Как показано в листинге 15.18, к методу `auditPurgeJob()` была применена аннотация `@Scheduled` с выражением `cron`. Это выражение `cron` означает, что задание должно запускаться ежесуточно в полночь. Для простоты сопровождения количество дней, в течение которых будут храниться записи аудита, вынесено во внешний файл свойств. Чтобы включить планирование задач в стиле аннотаций, мы определим дескриптор `<task:annotation-driven>` в корневом `WebApplicationContext`.

За дополнительными деталями обращайтесь в главу 21.

Резюме

В этой главе была рассмотрена поддержка планирования задач в Spring. Мы сосредоточили внимание на встроенной в Spring абстракции `TaskScheduler` и продемонстрировали ее использование для удовлетворения потребностей планирования задач на примере с обновлением данных. Мы также показали, как в Spring 3 поддерживается аннотация для асинхронного выполнения задач.

ГЛАВА 16

Использование удаленной обработки в Spring

Корпоративное приложение обычно нуждается во взаимодействии с другими приложениями. Возьмем, к примеру, компанию, продающую товары; когда пользователь размещает заказ, система обработки заказов обрабатывает его и генерирует транзакцию. Во время обработки заказа производится запрос к складской системе для проверки, доступен ли нужный товар на складе. После подтверждения заказа системе исполнения заказов отправляется уведомление о необходимости доставки товара заказчику. Наконец, соответствующая информация посылается системе бухгалтерского учета; генерируется счет-фактура и обрабатывается платеж.

Большую часть времени этот бизнес-процесс не может быть реализован каким-то одним приложением, поэтому для удовлетворения всех его требований вместе работают несколько приложений. Некоторые из таких приложений могут быть разработаны самостоятельно, а другие — приобретены у независимых поставщиков. Кроме того, приложения могут выполняться на разных машинах, расположенных в различных местах, и быть реализованы с помощью разных технологий и языков программирования (например, Java, .NET, C++ и т.д.). Подтверждение установления связи между приложениями для построения эффективного бизнес-процесса является критически важной задачей при построении архитектуры и реализации приложения. В результате приложению требуется поддержка удаленной обработки через разнообразные протоколы и технологии, чтобы оно могло принимать активное участие в корпоративной среде.

В мире Java поддержка удаленной обработки существует с момента его зарождения. В первые дни (Java 1.*x*) большинство требований удаленной обработки были реализованы с применением традиционных сокетов TCP или протокола RMI (Remote Method Invocation — удаленный вызов методов). С появлением J2EE обычным выбором для взаимодействия между приложениями на сервере стали EJB и JMS. Быстрое развитие XML и Интернета привело к возникновению удаленной поддержки с использованием XML через HTTP, включая Java API для протокола RPC на основе XML (JAX-RPC), Java API для веб-служб XML (JAX-RPC) и технологий, основанных на HTTP (например, Hessian, Burlap и т.п.). В Spring также предлагается собственная поддержка удаленной обработки на основе HTTP, которая называется HTTP-активатором (invoker) Spring. Чтобы справиться с бурным ростом Интернета и требованиями большей отзывчивости веб-приложений (например, с помощью средств Ajax) в последние годы, критически важной для успеха предприятия становится более облегченная и эффективная удаленная поддержка при-

ложений. В результате был создан интерфейс Java API для веб-служб REST (JAX-RS), который быстро обрел популярность. Немалую часть разработчиков привлекают и другие протоколы, такие как Comet и HTML5 WebSocket. Нечего и говорить, что технологии удаленной обработки продолжают развиваться быстрыми темпами.

Как уже было указано, в Spring предоставляется собственная поддержка удаленной обработки (к примеру, HTTP-активатор Spring), а также поддержка множества технологий, упоминавшихся ранее (RMI, EJB, JMS, Hessian, Burlap, JAX-RPC, JAX-WS, JAX-RS и т.д.). Описать всех их в одной главе невозможно, поэтому мы сосредоточим внимание здесь только на самых часто используемых технологиях. В частности, в этой главе будут рассмотрены следующие темы.

- *HTTP-активатор Spring.* Если приложения, которые должны взаимодействовать, основаны на Spring, то HTTP-активатор Spring предоставляет простой и эффективный способ для обращения к службам, предлагаемым другими приложениями. Мы покажем, как использовать HTTP-активатор Spring для отображения службы на уровне обслуживания и как вызывать службы, предоставляемые удаленным приложением.
- *Использование JMS в Spring.* Служба обмена сообщениями Java (Java Messaging Service — JMS) предлагает еще один асинхронный и слабо связанный способ обмена сообщениями между приложениями. Мы объясним, как Spring упрощает разработку приложений с помощью JMS.
- *Использование веб-служб REST в Spring.* Спроектированные специально для протокола HTTP, веб-службы REST являются наиболее часто применяемой технологией для предоставления удаленной поддержки приложениям, а также для поддержки интерактивных пользовательских интерфейсов веб-приложений, в которых используется Ajax. Мы продемонстрируем комплексную поддержку отображения служб с применением JAX-RS в инфраструктуре Spring 3 MVC и объясним, как обращаться к службам с помощью класса `RestTemplate`. Мы также обсудим вопросы защиты служб от несанкционированного доступа.

Создание проекта для примеров в STS

Начнем с создания проекта для примеров, рассматриваемых в этой главе. Поскольку мы будем отображать разработанные службы через HTTP, необходимо создать проект веб-приложения с помощью Spring MVC. Создайте новый шаблонный проект Spring в STS, выбрав вариант Spring MVC Project (Проект Spring MVC), как показано на рис. 16.1.

На следующем экране введите информацию, касающуюся проекта, как показано на рис. 16.2.

По завершении STS создаст проект веб-приложения с обязательными зависимостями, включая модуль Spring MVC.

Реализация уровня обслуживания для примеров

Мы собираемся продемонстрировать более полезные примеры, нежели простое приложение типа “Hello World”. Мы реализуем простую службу для выдачи информации о контакте (подобную тем, что рассматривались в предыдущих главах), используя JPA, и затем отобразим ее удаленным клиентам. В этом разделе рассматриваются вопросы реализации уровня обслуживания.

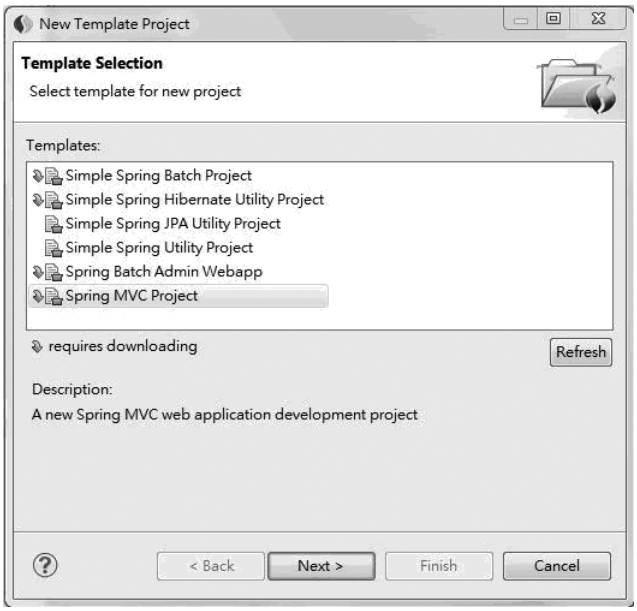


Рис. 16.1. Создание проекта Spring MVC в STS

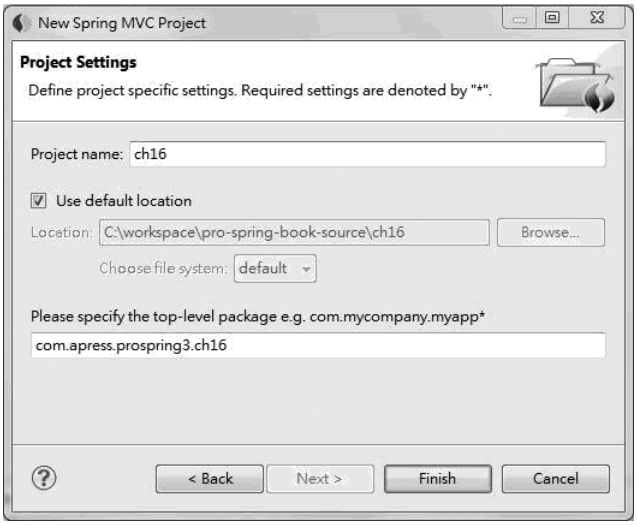


Рис. 16.2. Ввод деталей о проекте Spring MVC

Добавление обязательных зависимостей для серверной части JPA

В проект должны быть добавлены обязательные зависимости. В табл. 16.1 описаны зависимости, требуемые для реализации уровня обслуживания с применением JPA 2 и Hibernate в качестве поставщика постоянства. Кроме того, будет использоваться Spring Data JPA. Удостоверьтесь, что в проекте применяется Spring 3.1.

Таблица 16.1. Зависимости Maven для уровня обслуживания

Идентификатор группы	Идентификатор артефакта	Версия	Описание
org.springframework	spring-jdbc	3.1.0.RELEASE	Модуль Spring JDBC
org.springframework	spring-orm	3.1.0.RELEASE	Модуль Spring ORM
org.springframework	spring-tx	3.1.0.RELEASE	Модуль поддержки транзакций Spring
org.springframework.data	spring-data-jpa	1.0.1.RELEASE	Библиотека Spring Data JPA
org.hibernate	hibernate-entitymanager	3.6.8.Final	Диспетчер сущностей Hibernate с поддержкой JPA 2
javax.validation	validation-api	1.0.0.GA	Библиотека API-интерфейса JSR-303
org.hibernate	hibernate-validator	4.2.0.Final	Реализация JSR-303 в Hibernate
com.h2database	h2	1.3.160	Встроенная база данных JDBC типа H2
joda-time	joda-time	2.0	API-интерфейс для работы с датой и временем под названием Joda-Time (http://joda-time.sourceforge.net/), который используется в Spring Data JPA. В этой главе мы будем применять его в объектах предметной области
joda-time	joda-time-hibernate	1.3	Библиотека Joda-Time для интеграции с Hibernate, обеспечивающая сохранение типов данных даты и времени
com.google.guava	guava	10.0.1	Полезные вспомогательные классы

На рис. 16.3 показаны зависимости в STS после их добавления.

Проверка работоспособности проекта

Давайте перед тем, как продолжить, проверим, работает ли проект. В представлении Servers (Серверы) в STS удостоверьтесь в существовании экземпляра tc Server (специализированная версия Tomcat, предоставленная VMware SpringSource), как показано на рис. 16.4.

Если вы не видите этот экземпляр, его понадобится создать. Соответствующие детали ищите в приложении А.

Теперь можно сконфигурировать проект для запуска на сервере, развернуть и протестировать его. Щелкните правой кнопкой мыши на имени проекта и выберите в контекстном меню пункт Run As⇨Run on Server (Запустить как⇨Запустить на сервере), как показано на рис. 16.5. Выберите вариант tc Server (если он не единственный в списке), как показано на рис. 16.6.

На следующем экране среда STS должна добавить проект на сервер tc Server (рис. 16.7).

Щелкните на кнопке Finish (Готово). Среда STS выполнит сборку проекта, развернет его на сервере и затем отобразит страницу приветствия, которая автоматически создается шаблонным проектом, как показано на рис. 16.8.

Это простая страница JSP, которая показывает сообщение “Hello World” и время на сервере. Не переживайте по поводу символов вопросительного знака; их наличие объясняется настройкой локали на машине разработки (zh_HK для Гонконга). Если на вашей машине используется другая локаль, то вы должны получить корректный формат времени.

Страница приветствия указывает на то, что проект создан нормально, поэтому мы можем продолжить реализацию уровня обслуживания и сделать его видимым для удаленного доступа.

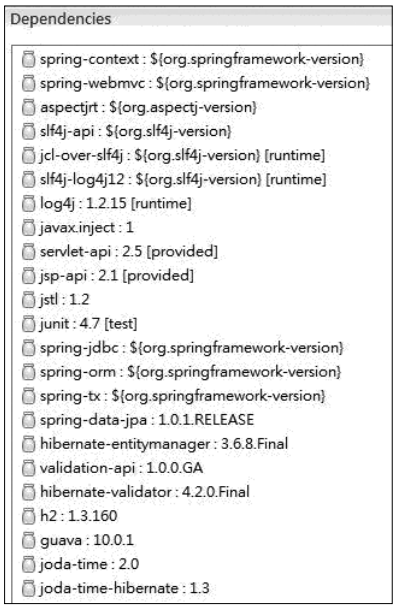


Рис. 16.3. Зависимости Maven для проекта Spring MVC с серверной частью JPA

Модель данных для примеров

В примерах этой главы мы будем использовать очень простую модель данных, которая содержит единственную таблицу `CONTACT` для хранения информации о контактах.

В листинге 16.1 показан сценарий для создания схемы базы данных (файл `schema.sql` в папке `/src/main/resources`).

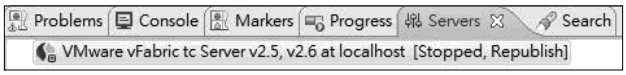


Рис. 16.4. Экземпляр tc Server в STS

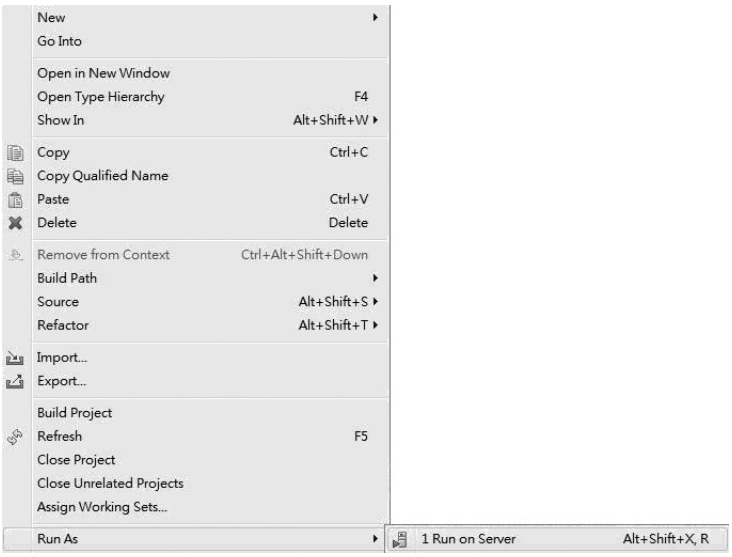


Рис. 16.5. Запуск на сервере в STS

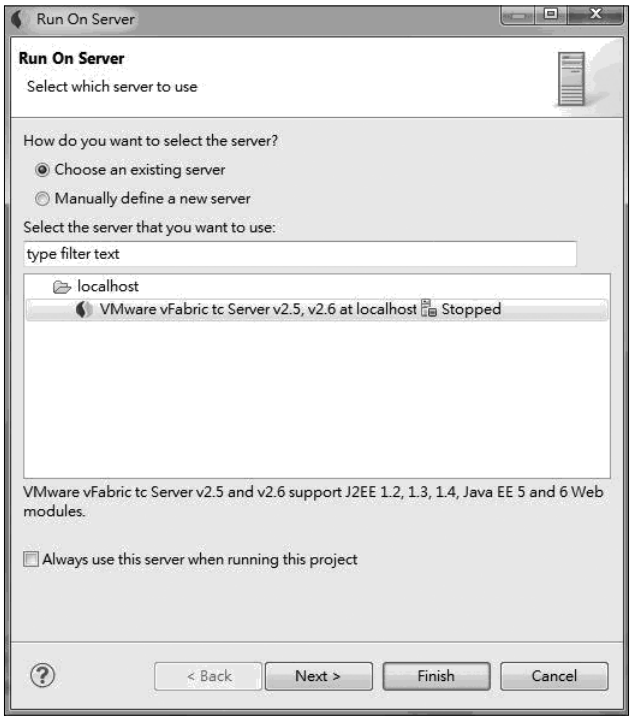


Рис. 16.6. Выбор tc Server в STS

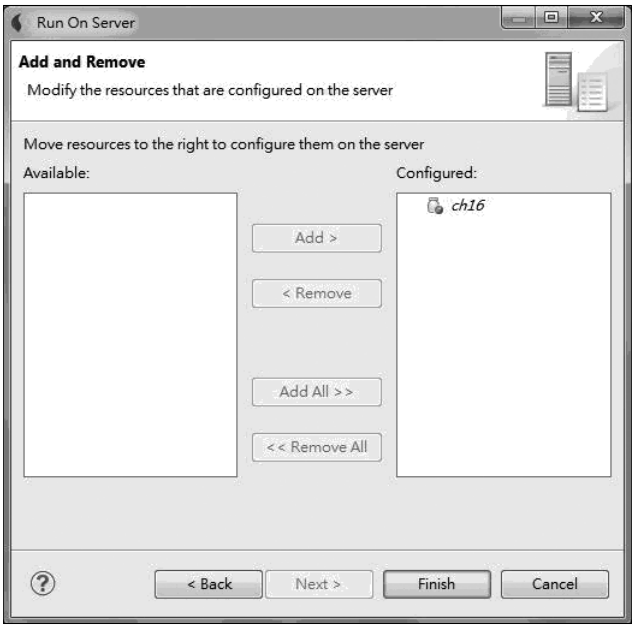


Рис. 16.7. Проект добавлен на сервер tc Server

Рис. 16.8. Стандартная страница приветствия проекта Spring MVC

Листинг 16.1. Сценарий для создания схемы базы данных

```
DROP TABLE IF EXISTS CONTACT;

CREATE TABLE CONTACT (
     ID INT NOT NULL AUTO_INCREMENT
   , FIRST_NAME VARCHAR(60) NOT NULL
   , LAST_NAME VARCHAR(40) NOT NULL
   , BIRTH_DATE DATE
   , VERSION INT NOT NULL DEFAULT 0
   , UNIQUE UQ_CONTACT_1 (FIRST_NAME, LAST_NAME)
   , PRIMARY KEY (ID)
);
```

Как видите, таблица `CONTACT` содержит только несколько базовых полей информации о контакте. В листинге 16.2 приведен код сценария для наполнения таблицы тестовыми данными (`test-data.sql`).

Листинг 16.2. Сценарий для наполнения тестовыми данными

```
insert into contact (first_name, last_name, birth_date) values
   ('Clarence', 'Ho', '1980-07-30');
insert into contact (first_name, last_name, birth_date) values
   ('Scott', 'Tiger', '1990-11-02');
insert into contact (first_name, last_name, birth_date) values
   ('John', 'Smith', '1964-02-28');
```

Реализация и конфигурирование интерфейса `ContactService`

После создания шаблонного проекта, модели данных и сценариев можно приступать к реализации и конфигурированию уровня обслуживания для примеров этой главы.

В последующих разделах мы обсудим реализацию интерфейса `ContactService` с использованием JPA 2, Spring Data JPA и Hibernate в качестве поставщика постоянства. Затем мы покажем, как сконфигурировать уровень обслуживания в проекте Spring.

Реализация интерфейса `ContactService`

В примерах этой главы мы будем отображать удаленным клиентам службы для разнообразных операций над контактной информацией. Первым делом необходимо создать сущностный класс `Contact`, который представлен в листинге 16.3.

Листинг 16.3. Сущностный класс `Contact`

```
package com.apress.prospring3.ch16.domain;

import static javax.persistence.GenerationType.IDENTITY;

import java.io.Serializable;

import javax.persistence.*;

import org.hibernate.annotations.Type;
import org.joda.time.DateTime;

@Entity
@Table(name = "contact")
public class Contact implements Serializable {

   private Long id;
   private int version;
   private String firstName;
   private String lastName;
   private DateTime birthDate;

   @Id
   @GeneratedValue(strategy = IDENTITY)
   @Column(name = "ID")
   public Long getId() {
      return id;
   }

   @Version
   @Column(name = "VERSION")
   public int getVersion() {
      return version;
   }

   @Column(name = "FIRST_NAME")
   public String getFirstName() {
      return firstName;
   }

   @Column(name = "LAST_NAME")
   public String getLastName() {
      return lastName;
   }

   @Column(name = "BIRTH_DATE")
   @Type(type="org.joda.time.contrib.hibernate.PersistentDateTime")
   public DateTime getBirthDate() {
      return birthDate;
   }

   // Методы установки не показаны.

   public String toString() {
      return "Contact - Id: " + id + ", First name: " + firstName
         + ", Last name: " + lastName + ", Birthday: " + birthDate;
   }
}
```

Как показано в листинге 16.3, в классе применяются стандартные аннотации JPA. Мы также используем класс `DateTime` из Joda-Time для атрибута `birthDate`.

Продолжим реализацию уровня обслуживания; в листинге 16.4 приведен интерфейс `ContactService` со службами, которые должны быть отображены.

Листинг 16.4. Интерфейс `ContactService`

```
package com.apress.prospring3.ch16.service;

import java.util.List;

import com.apress.prospring3.ch16.domain.Contact;

public interface ContactService {

   public List<Contact> findAll();

   public List<Contact> findByFirstName(String firstName);

   public Contact findById(Long id);

   public Contact save(Contact contact);

   public void delete(Contact contact);
}
```

Методы этого интерфейса самоочевидны. Поскольку мы будем пользоваться поддержкой репозитория Spring Data JPA, реализуем интерфейс `ContactRepository` (листинг 16.5).

Листинг 16.5. Интерфейс `ContactRepository`

```
package com.apress.prospring3.ch16.repository;

import java.util.List;

import org.springframework.data.repository.CrudRepository;

import com.apress.prospring3.ch16.domain.Contact;

public interface ContactRepository extends CrudRepository<Contact, Long> {

   public List<Contact> findByFirstName(String firstName);
}
```

Как обсуждалось в главе 10, за счет расширения интерфейса `CrudRepository<T,ID extends Serializable>` в `ContactService` нужно явно объявить только метод `findByFirstName()`.

В листинге 16.6 приведен класс реализации интерфейса `ContactService`.

Листинг 16.6. Класс `ContactServiceImpl`

```
package com.apress.prospring3.ch16.service.jpa;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import com.apress.prospring3.ch16.domain.Contact;
import com.apress.prospring3.ch16.repository.ContactRepository;
import com.apress.prospring3.ch16.service.ContactService;
import com.google.common.collect.Lists;

@Service("contactService")
```

```
@Repository
@Transactional
public class ContactServiceImpl implements ContactService {
   @Autowired
   private ContactRepository contactRepository;
   @Transactional(readOnly=true)
   public List<Contact> findAll() {
      return Lists.newArrayList(contactRepository.findAll());
   }
   @Transactional(readOnly=true)
   public List<Contact> findByFirstName(String firstName) {
      return contactRepository.findByFirstName(firstName);
   }
   @Transactional(readOnly=true)
   public Contact findById(Long id) {
      return contactRepository.findOne(id);
   }
   public Contact save(Contact contact) {
      return contactRepository.save(contact);
   }
   public void delete(Contact contact) {
      contactRepository.delete(contact);
   }
}
```

При наличии проблем с пониманием кода в листинге 16.6, обратитесь в главу 10 за детальным описанием реализации уровня обслуживания, где в качестве поставщика постоянства используется JPA. Теперь реализация в основном завершена, и следующий шаг заключается в конфигурировании службы в `ApplicationContext` проекта веб-приложения, что будет темой следующего раздела.

Конфигурирование уровня обслуживания

Для настройки уровня обслуживания в проекте Spring MVC мы создаем отдельный конфигурационный файл по имени `datasource-tx-jpa.xml` в папке `/src/main/resource`. Содержимое этого файла показано в листинге 16.7.

Листинг 16.7. Конфигурационный файл `datasource-tx-jpa.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:context="http://www.springframework.org/schema/context"
   xmlns:jdbc="http://www.springframework.org/schema/jdbc"
   xmlns:jpa="http://www.springframework.org/schema/data/jpa"
   xmlns:tx="http://www.springframework.org/schema/tx"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context-3.1.xsd
      http://www.springframework.org/schema/jdbc
      http://www.springframework.org/schema/jdbc/spring-jdbc-3.1.xsd
      http://www.springframework.org/schema/data/jpa
      http://www.springframework.org/schema/data/jpa/spring-jpa-1.0.xsd
      http://www.springframework.org/schema/tx
      http://www.springframework.org/schema/tx/spring-tx-3.1.xsd">
```

```
   <jdbc:embedded-database id="dataSource" type="H2">
      <jdbc:script location="classpath:schema.sql"/>
      <jdbc:script location="classpath:test-data.sql"/>
   </jdbc:embedded-database>

   <bean id="transactionManager"
         class="org.springframework.orm.jpa.JpaTransactionManager">
      <property name="entityManagerFactory" ref="emf"/>
   </bean>

   <tx:annotation-driven transaction-manager="transactionManager" />

   <bean id="emf"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
      <property name="dataSource" ref="dataSource" />
      <property name="jpaVendorAdapter">
       <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter" />
      </property>
      <property name="packagesToScan" value="com.apress.prospring3.ch16.domain"/>
      <property name="jpaProperties">
         <props>
            <prop key="hibernate.dialect">
                  org.hibernate.dialect.H2Dialect
            </prop>
            <prop key="hibernate.max_fetch_depth">3</prop>
            <prop key="hibernate.jdbc.fetch_size">50</prop>
            <prop key="hibernate.jdbc.batch_size">10</prop>
            <prop key="hibernate.show_sql">true</prop>
         </props>
      </property>
   </bean>
   <context:annotation-config/>
   <jpa:repositories base-package="com.apress.prospring3.ch16.repository"
                  entity-manager-factory-ref="emf"
                  transaction-manager-ref="transactionManager"/>
</beans>
```

Подробные сведения об определениях бинов ищите в главе 10. Теперь необходимо импортировать конфигурацию в корневой контекст `WebApplicationContext`. Для шаблонного проекта Spring MVC соответствующий файл расположен в `/src/main/webapp/WEB-INF/spring/root-context.xml`. Модифицированное содержимое этого файла приведено в листинге 16.8.

Листинг 16.8. Файл `root-context.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:context="http://www.springframework.org/schema/context"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context-3.1.xsd">
   <import resource="classpath:datasource-tx-jpa.xml" />
   <context:component-scan
      base-package="com.apress.prospring3.ch16.service.jpa" />
</beans>
```

Сначала в этот конфигурационный файл было добавлено пространство имен `context`. Затем файл `datasource-tx-jpa.xml` был импортирован в `WebApplicationContext` и, наконец, платформе Spring было указано сканировать заданный пакет в поисках бинов Spring. Теперь уровень обслуживания завершен и готов к отображению и использованию удаленными клиентами.

Использование HTTP-активатора Spring

Если приложение, с которым планируется взаимодействовать, также построено с помощью Spring, то удобным вариантом будет применение HTTP-активатора Spring. Он предлагает исключительно простой способ отображения служб внутри `WebApplicationContext` удаленным клиентам. Процедуры отображения и доступа к службе описаны в следующих разделах.

Отображение службы

Чтобы отобразить службу, добавьте в конфигурационный файл `root-context.xml` определение бина, представленное в листинге 16.9.

Листинг 16.9. Бин для отображения службы выдачи информации о контакте, использующей HTTP-активатор Spring

```
   <bean name="contactExporter"
class="org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter">
      <property name="service" ref="contactService" />
      <property name="serviceInterface"
         value="com.apress.prospring3.ch16.service.ContactService" />
   </bean>
```

Как показано в листинге 16.9, бин `contactExporter` определен с помощью класса `HttpInvokerServiceExporter`, который предназначен для экспорта любого бина Spring в виде службы через HTTP-активатор. Внутри бина определены два свойства. Первое из них, свойство `service`, отражает бин, предоставляющий службу. В этом свойстве внедрен бин `contactService`. Второе свойство, `serviceInterface`, задает тип отображаемого интерфейса, которым является `com.apress.prospring3.ch16.service.ContactService`.

Далее внутри дескриптора развертывания веб-приложения (`/src/main/webapp/WEB-INF/web.xml`) мы должны определить сервлет для службы. В листинге 16.10 приведен фрагмент кода для добавления в файл `web.xml`.

Листинг 16.10. Определение сервлета для HTTP-активатора

```
   <!-- Удаленная обработка в Spring с помощью HTTP-активатора -->
   <servlet>
      <servlet-name>contactExporter</servlet-name>
      <servlet-class>
         org.springframework.web.context.support.HttpRequestHandlerServlet
      </servlet-class>
   </servlet>

   <servlet-mapping>
      <servlet-name>contactExporter</servlet-name>
      <url-pattern>/remoting/ContactService</url-pattern>
   </servlet-mapping>
```

В этом фрагменте определен сервлет с классом HttpRequestHandlerServlet, который используется для отображения экспортируемого бина Spring, определенного в WebApplicationContext. Обратите внимание, что имя сервлета (contactExporter) должно совпадать с именем экспортируемого бина (см. листинг 16.9; там имя бина тоже выглядит как contactExporter). Затем сервлету ставится в соответствие URL /remoting/ContactService в рамках веб-контекста (т.е. http://localhost:8080/ch16) приложения.

К этому моменту, при условии, что вы оставили настройки проекта STS в состоянии по умолчанию, проект должен быть пересобран и развернут на экземпляре tc Server. Теперь мы можем перейти к разработке клиента, вызывающего нашу службу.

Вызов службы

Вызов службы через HTTP-активатор Spring осуществляется очень просто. Сначала нужно сконфигурировать ApplicationContext, как показано в листинге 16.11 (файл http-invoker-app-context.xml).

Листинг 16.11. Конфигурация ApplicationContext для клиента HTTP-активатора

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd">

   <bean id="remoteContactService"      class="org.springframework.remoting.
httpinvoker.HttpInvokerProxyFactoryBean">
      <property name="serviceUrl"
         value="http://localhost:8080/ch16/remoting/ContactService" />
      <property name="serviceInterface"
         value="com.apress.prospring3.ch16.service.ContactService" />
   </bean>

</beans>
```

В листинге 16.11 для клиентской стороны объявляется бин типа HttpInvokerProxyFactoryBean. Устанавливаются два свойства.

- Свойство serviceUrl указывает местоположение удаленной службы, которым является http://localhost:8080/ch16/remoting/ContactService.
- Свойство serviceInterface задает интерфейс службы (т.е. ContactService). Если для клиента разрабатывается отдельный проект, интерфейс ContactService и сущностный класс Contact должны присутствовать в пути классов этого клиентского приложения.

В листинге 16.12 представлен основной класс для вызова удаленной службы.

Листинг 16.12. Класс HttpInvokerClientSample

```
package com.apress.prospring3.ch16.remoting;

import java.util.List;

import org.springframework.context.support.GenericXmlApplicationContext;

import com.apress.prospring3.ch16.domain.Contact;
import com.apress.prospring3.ch16.service.ContactService;
```

```
public class HttpInvokerClientSample {

   public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:http-invoker-app-context.xml");
      ctx.refresh();

      ContactService contactService =
         ctx.getBean("remoteContactService", ContactService.class);

      // Найти все контакты.
      System.out.println("Finding all contacts");
      List<Contact> contacts = contactService.findAll();
      listContacts(contacts);

      // Найти контакты по имени.
      System.out.println("Finding contact with first name equals Clarence");
      contacts = contactService.findByFirstName("Clarence");
      listContacts(contacts);

   }

   private static void listContacts(List<Contact> contacts) {

      for (Contact contact: contacts) {
         System.out.println(contact);
      }
      System.out.println("");
   }
}
```

Как показано в листинге 16.12, эта программа очень похожа на любое другое автономное приложение Spring. Сначала инициализируется `ApplicationContext`, после чего извлекается бин `contactService`. Затем просто вызываются его методы, как в локальном приложении. Запуск программы дает следующий вывод:

```
Finding all contacts
Contact - Id: 1, First name: Clarence, Last name: Ho, Birthday:
1980-07-30T00:00:00.000+08:00
Contact - Id: 2, First name: Scott, Last name: Tiger, Birthday:
1990-11-02T00:00:00.000+08:00
Contact - Id: 3, First name: John, Last name: Smith, Birthday:
1964-02-28T00:00:00.000+08:00

Finding contact with first name equals Clarence
Contact - Id: 1, First name: Clarence, Last name: Ho, Birthday:
1980-07-30T00:00:00.000+08:00
```

В этом выводе видно, что методы `findAll()` и `findByFirstName()` были успешно вызваны и результаты возвращены.

Использование JMS в Spring

Использование промежуточного программного обеспечения, ориентированного на сообщения (которое в общем случае называют *сервером MQ*) — это еще один популярный способ поддержки взаимодействия между приложениями. Основное преимущество сервера очереди сообщений (message queue — MQ) заключается в том, что он предлагает асинхронный и слабо связный путь для интеграции приложений. В мире Java стандартом для подключения к серверу MQ с целью отправки или получения сообщений является JMS.

Сервер MQ поддерживает список очередей и тем, для которых приложения могут подключаться, посылать и получать сообщения. Ниже приведено краткое описание отличий между очередью и темой.

- *Очередь*. Очередь используется для поддержки модели обмена сообщениями "точка-точка". Когда генератор отправляет сообщение в очередь, сервер MQ сохраняет это сообщение в очереди и доставляет его одному и только одному потребителю при следующем его подключении.
- *Тема*. Тема используется для поддержки модели обмена сообщениями "публикация-подписка". На сообщение внутри темы может подписываться любое количество клиентов. Когда сообщение поступает в заданную тему, сервер MQ доставляет его всем клиентам, которые на него подписаны. Эта модель особенно удобна в ситуации, когда имеется множество приложений, заинтересованных в получении одной и той же порции информации (например, лента новостей).

В стандарте JMS генератор подключается к серверу MQ и отправляет сообщение в очередь или в тему. Потребитель также подключается к серверу MQ и прослушивает очередь или темы на предмет наличия интересующих его сообщений. В JMS 1.1 был унифицирован API-интерфейс, так что генератор и потребитель не должны были иметь дело с разными API-интерфейсами при взаимодействии с очередями и темами. В этом разделе мы сосредоточим внимание на модели "точка-точка" работы с очередями, которая представляет собой более часто применяемый шаблон в корпоративной среде.

Для разработки и тестирования приложения JMS требуется сервер MQ. В данном разделе мы будем пользоваться сервером Apache ActiveMQ (`activemq.apache.org`) — очень популярным сервером MQ с открытым кодом.

Пример требует нескольких новых зависимостей Maven, которые описаны в табл. 16.2. Добавьте их в проект.

Таблица 16.2. Зависимости Maven для JMS и ActiveMQ

Идентификатор группы	Идентификатор артефакта	Версия	Описание
`org.springframework`	`spring-jms`	3.1.0.RELEASE	Модуль Spring JMS
`org.apache.activemq`	`activemq-core`	5.5.1	Java-библиотека ActiveMQ
`javax.jms`	`jms-api`	1.1-rev-1	API-интерфейс JMS 1.1

Установка сервера ActiveMQ и применение JMS в Spring рассматриваются в следующих разделах.

Установка сервера ActiveMQ

Установка сервера ActiveMQ для использования при разработке осуществляется легко. Для начала загрузите последний выпуск из веб-сайта ActiveMQ (`activemq.apache.org/download.html`) и извлеките файлы из архива в выбранную папку на своем компьютере. На момент написания этой главы последним выпуском был 5.5.1.

После извлечения перейдите в папку `bin` и запустите команду `activemq` (для Windows это `activemq.bat`, а для Unix/Linux — `activemq`). Сервер запустится, а вы увидите вывод, подобный показанному на рис. 16.9.

В выводе видно, что сервер ActiveMQ прослушивает порт 61616 на предмет подключений JMS. Кроме того, был также запущен встроенный веб-сервер для целей администрирования, `localhost:8161/admin`.

```
C:\apache-activemq-5.5.1\bin>activemq
Java Runtime: Sun Microsystems Inc. 1.6.0_29 C:\Program Files\Java\jdk1.6.0_29\jre
  Heap sizes: current=125056k  free=122439k  max=466048k
    JVM args: -Dcom.sun.management.jmxremote -Xmx512M -Dorg.apache.activemq.UseDedicatedTaskRunner=true -Djava.util.logg
ing.config.file=logging.properties -Dactivemq.classpath=C:\apache-activemq-5.5.1\bin\../conf;C:\apache-activemq-5.5.1\bi
n\../conf; -Dactivemq.home=C:\apache-activemq-5.5.1\bin\.. -Dactivemq.base=C:\apache-activemq-5.5.1\bin\..
ACTIVEMQ_HOME: C:\apache-activemq-5.5.1\bin\..
ACTIVEMQ_BASE: C:\apache-activemq-5.5.1\bin\..
Loading message broker from: xbean:activemq.xml
 INFO | Refreshing org.apache.activemq.xbean.XBeanBrokerFactory$1@6c4fc156: startup date [Wed Nov 30 14:48:35 CST 2011];
 root of context hierarchy
 WARN | destroyApplicationContextOnStop parameter is deprecated, please use shutdown hooks instead
 INFO | PListStore:C:\apache-activemq-5.5.1\bin\..\data\localhost\tmp_storage started
 INFO | Using Persistence Adapter: KahaDBPersistenceAdapter[C:\apache-activemq-5.5.1\bin\..\data\kahadb]
 INFO | KahaDB is version 3
 INFO | Recovering from the journal ...
 INFO | Recovery replayed 1 operations from the journal in 0.015 seconds.
 INFO | ActiveMQ 5.5.1 JMS Message Broker (localhost) is starting
 INFO | For help or more information please see: http://activemq.apache.org/
 INFO | Listening for connections at: tcp://Clarence-PC:61616
 INFO | Connector openwire Started
 INFO | ActiveMQ JMS Message Broker (localhost, ID:Clarence-PC-63489-1322635717052-0:1) started
 INFO | jetty-7.1.6.v20100715
 INFO | ActiveMQ WebConsole initialized.
 INFO | Initializing Spring FrameworkServlet 'dispatcher'
 INFO | ActiveMQ Console at http://0.0.0.0:8161/admin
 INFO | ActiveMQ Web Demos at http://0.0.0.0:8161/demo
 INFO | RESTful file access application at http://0.0.0.0:8161/fileserver
 INFO | Started SelectChannelConnector@0.0.0.0:8161
```

Рис. 16.9. Запуск сервера ActiveMQ

Откройте веб-браузер и зайдите на административный веб-сайт. Первая страница, которую вы увидите, будет похожа на показанную на рис. 16.10.

Рис. 16.10. Административный веб-сайт сервера ActiveMQ

Давайте создадим очередь для целей тестирования. Щелкните на пункте Queues (Очереди) в верхней полосе меню, введите `prospring3` в поле Queue Name (Имя очереди) и щелкните на кнопке Create (Создать). Очередь будет создана, как показано на рис. 16.11.

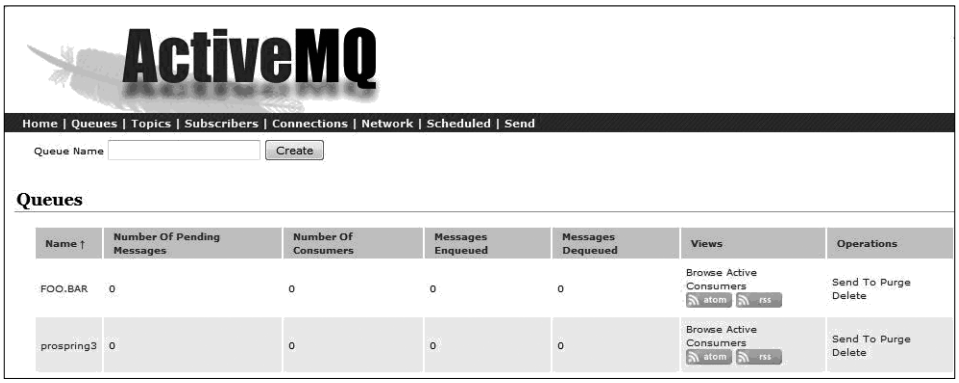


Рис. 16.11. Создание очереди в ActiveMQ

На этом установка сервера ActiveMQ завершена, поэтому можно приступать к разработке клиента JMS, использующего Spring для подключения к серверу и получения сообщений.

Реализация слушателя JMS в Spring

Для разработки слушателя сообщений нам нужно создать класс, который реализует интерфейс `javax.jms.MessageListener` и его метод `onMessage()`. Этот класс показан в листинге 16.13.

Листинг 16.13. Класс слушателя сообщений JMS

```
package com.apress.prospring3.ch16.jms.listener;

import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.TextMessage;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class SimpleMessageListener implements MessageListener {

    private static final Logger logger =
    LoggerFactory.getLogger(SimpleMessageListener.class);

    public void onMessage(Message message) {
        TextMessage textMessage = (TextMessage) message;
        try {
            logger.info("Message received: " + textMessage.getText());
        } catch (JMSException ex) {
            logger.error("JMS error", ex);
        }
    }
}
```

При поступлении сообщения методу `onMessage()` передается экземпляр интерфейса `javax.jms.Message`. Внутри этого метода сообщение приводится к экземпляру интерфейса `javax.jms.TextMessage`, и с помощью метода `TextMessage.getText()` извлекается тело сообщения. Список возможных форматов сообщения приведен в онлайновой документации JEE (`http://docs.oracle.com/javaee/6/api/javax/jms/Message.html`).

После построения слушателя сообщений следующий шаг заключается в определении конфигурации ApplicationContext. В листинге 16.14 представлено содержимое файла jms-listener-app-context.xml.

Листинг 16.14. Конфигурация слушателя сообщений JMS

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:jms="http://www.springframework.org/schema/jms"
   xmlns:p="http://www.springframework.org/schema/p"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
      http://www.springframework.org/schema/jms
      http://www.springframework.org/schema/jms/spring-jms-3.1.xsd">

   <bean id="connectionFactory"
      class="org.apache.activemq.ActiveMQConnectionFactory"
      p:brokerURL="tcp://localhost:61616" />

   <bean id="simpleMessageListener"
     class="com.apress.prospring3.ch16.jms.listener.SimpleMessageListener"/>

   <jms:listener-container container-type="default"
      connection-factory="connectionFactory" acknowledge="auto">
      <jms:listener destination="prospring3" ref="simpleMessageListener"
         method="onMessage" />
   </jms:listener-container>

</beans>
```

Как показано в листинге 16.14, мы сначала объявляем интерфейс javax.jms.ConnectionFactory, предоставленный Java-библиотекой ActiveMQ (класс ActiveMQConnectionFactory). Обратите внимание, что свойство brokerURL должно иметь то же значение, что и при запуске ActiveMQ (см. рис. 16.9). Затем объявляется бин типа SimpleMessageListener. Наконец, мы используем удобный дескриптор <jms:listener-container>, предоставляемый пространством имен jms в Spring, для объявления слушателя сообщений с указанием получателя (т.е. очереди prospring3), ссылки на бин и метода, вызываемого при поступлении сообщения.

В листинге 16.15 приведена программа, позволяющая протестировать получение сообщения.

Листинг 16.15. Тестирование слушателя сообщений JMS

```
package com.apress.prospring3.ch16.jms;

import org.springframework.context.support.GenericXmlApplicationContext;

public class JmsListenerSample {

   public static void main(String[] args) {

      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:jms-listener-app-context.xml");
      ctx.refresh();

      while (true) {
      }
   }
}
```

В этой программе нет ничего необычного. Она просто инициализирует `Application Context` и начинает бесконечный цикл. Просто запустите программу; она подключится к серверу ActiveMQ, после чего будет ожидать поступления сообщений в очередь `prospring3`. Теперь давайте попробуем отправить сообщение в очередь `prospring3`. На административном веб-сайте ActiveMQ щелкните на пункте Send (Отправить) в верхней полосе меню, в поле Destination (Получатель) введите имя очереди, а в поле Message body (Тело сообщения) — текст сообщения и щелкните на кнопке Send (Отправить), как показано на рис. 16.12.

Рис. 16.12. Отправка сообщения в очередь ActiveMQ

После щелчка на кнопке Send в консоли STS вы увидите следующий вывод из слушателя сообщений:

```
INFO : com.apress.prospring3.ch16.jms.listener.SimpleMessageListener -
Message received: Testing message for Spring JMS Listener
```

Как видите, настроить в Spring слушатель JMS для удаленного взаимодействия довольно просто. В среде сервера приложений JEE понадобится лишь сконфигурировать фабрику подключений, используя поиск JNDI, внутри конфигурационного файла Spring. Никаких других изменений в код вносить не придется.

Отправка сообщений JMS в Spring

А теперь посмотрим, как отправлять сообщения JMS в Spring. Для этой цели мы будем применять удобный класс `org.springframework.jms.core.JmsTemplate`. Сначала мы разработаем интерфейс `MessageSender` и его класс реализации `SimpleMessageSender`. Соответствующий код приведен в листингах 16.16 и 16.17.

Листинг 16.16. Интерфейс `MessageSender`

```
package com.apress.prospring3.ch16.jms.sender;
public interface MessageSender {
   public void sendMessage(String message);
}
```

Листинг 16.17. Класс `SimpleMessageSender`

```
package com.apress.prospring3.ch16.jms.sender;

import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.Session;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.jms.core.MessageCreator;
import org.springframework.stereotype.Component;
@Component("messageSender")
public class SimpleMessageSender implements MessageSender {
   @Autowired
   private JmsTemplate jmsTemplate;
   public void sendMessage(final String message) {
      this.jmsTemplate.send(new MessageCreator() {
         public Message createMessage(Session session)
            throws JMSException {
            return session.createTextMessage(message);
         }
      });
   }
}
```

В листинге 16.17 видно, что был внедрен экземпляр `JmsTemplate`.

В методе `sendMessage()` мы вызываем метод `JmsTemplate.send()` с конструированием на месте экземпляра интерфейса `org.springframework.jms.core.MessageCreator`. В экземпляре `MessageCreator` реализован метод `createMessage()`, создающий новый экземпляр `TextMessage`, который будет отправлен ActiveMQ.

В листинге 16.18 показана конфигурация Spring для отправки сообщений JMS (файл `jms-sender-app-context.xml`).

Листинг 16.18. Конфигурация Spring для отправки сообщений

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:context="http://www.springframework.org/schema/context"
   xmlns:jms="http://www.springframework.org/schema/jms"
   xmlns:p="http://www.springframework.org/schema/p"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context-3.1.xsd
      http://www.springframework.org/schema/jms
      http://www.springframework.org/schema/jms/spring-jms-3.1.xsd">
   <context:annotation-config/>
```

```
    <bean id="connectionFactory"
        class="org.apache.activemq.ActiveMQConnectionFactory"
        p:brokerURL="tcp://localhost:61616" />

    <bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
        <constructor-arg name="connectionFactory" ref="connectionFactory"/>
        <property name="defaultDestinationName" value="prospring3"/>
    </bean>

    <context:component-scan
        base-package="com.apress.prospring3.ch16.jms.sender"/>

</beans>
```

В листинге 16.18, как обычно, определяется бин `connectionFactory`. Кроме того, объявляется экземпляр `JmsTemplate` с аргументом конструктора `connectionFactory` и свойством `defaultDestinationName`, установленным в очередь `prospring3`.

В листинге 16.19 представлена программа для тестирования отправки сообщений.

Листинг 16.19. Тестирование отправки сообщений

```
package com.apress.prospring3.ch16.jms;

import org.springframework.context.support.GenericXmlApplicationContext;
import com.apress.prospring3.ch16.jms.sender.MessageSender;

public class JmsSenderSample {
    public static void main(String[] args) {
        GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
        ctx.load("classpath:jms-sender-app-context.xml");
        ctx.refresh();
        // Отправить сообщение.
        MessageSender messageSender = ctx.getBean("messageSender",
            MessageSender.class);
        messageSender.sendMessage("Clarence testing JMS message");
    }
}
```

Программа очень проста. Запуск этой программы приведет к отправке сообщения в очередь. Если программа с классом `JmsListenerSample` все еще выполняется (или можете сейчас запустить ее), вы получите следующий вывод из нее:

```
INFO : com.apress.prospring3.ch16.jms.listener.SimpleMessageListener -
Message received: Clarence testing JMS message
```

В реальности сообщение, скорее всего, будет иметь формат XML, представляя порцию бизнес-информации (например, онлайновый заказ, транзакцию, запрос на резервирование и т.п.).

В этом разделе были показаны только базовые сценарии использования JMS. За дополнительными сведениями обращайтесь в онлайновое руководство JEE по адресу:

`http://docs.oracle.com/javaee/5/tutorial/doc/bncdq.html`

Использование веб-служб REST в Spring

В наши дни веб-службы REST (RESTful-WS), вероятно, являются самой широко используемой технологией для удаленного доступа. Веб-службы REST интенсивно приме-

няются везде, от вызова удаленных служб через HTTP до поддержки интерактивного пользовательского интерфейса в стиле Ajax.

Популярность веб-служб REST объясняется несколькими причинами.

- *Простота понимания.* Веб-службы REST спроектированы на основе протокола HTTP. Адрес URL вместе с HTTP-методом определяют намерение запроса. Например, URL вида `http://somedomain.com/restful/customer/1` с HTTP-методом `GET` означает, что клиент желает извлечь информацию о заказчике, идентификатор которого равен 1.
- *Облегченность.* По сравнению с веб-службами на основе SOAP веб-службы REST являются более легковесными; веб-службы SOAP требуют большого объема метаданных, описывающих то, какую службу клиент должен вызывать. Для запроса и ответа REST достаточно просто запроса и ответа HTTP, как в любом другом веб-приложении.
- *Дружественность к брандмауэру.* Поскольку веб-службы REST предназначены для доступа по протоколу HTTP (или HTTPS), приложение становится более дружественным к брандмауэру и легко доступным для удаленных клиентов.

В этом разделе мы обсудим базовые концепции веб-служб REST и их поддержку в Spring посредством модуля Spring MVC.

Введение в веб-службы REST

Аббревиатура REST означает REpresentational State Transfer (передача состояния представления). Технология REST определяет набор архитектурных ограничений, которые вместе описывают "унифицированный интерфейс" для доступа к ресурсам.

Основные концепции "унифицированного интерфейса" включает идентификацию ресурсов и манипулирование ресурсами через представления.

Для идентификации ресурсов порция информации должна быть доступна через URI (Uniform Resource Identifier — унифицированный идентификатор ресурса). Например, URL вида `http://www.somedomain.com/api/contact/1` — это URI, представляющий ресурс, который является порцией информации о контакте с идентификатором 1. Если контакт с идентификатором 1 не существует, клиент получит HTTP-ошибку 404, как и в случае, когда какая-то страница не найдена на веб-сайте. Или вот еще один пример: `http://www.somedomain.com/api/contacts` — URI, представляющий ресурс, который является списком информации о контактах.

Такими идентифицируемыми ресурсами можно управлять посредством различных представлений, которые описаны в табл. 16.3.

Таблица 16.3. Представления для манипулирования ресурсами

Представление	Описание
`GET`	`GET` извлекает представление ресурса
`HEAD`	Идентично `GET`, но без тела ответа. Обычно используется для получения заголовка
`POST`	`POST` создает новый ресурс
`PUT`	`PUT` обновляет ресурс
`DELETE`	`DELETE` удаляет ресурс
`OPTIONS`	`OPTIONS` извлекает разрешенные HTTP-методы

Детальное описание веб-служб REST можно найти в книге *Ajax and REST Recipes, A Problem-Solution Approach* (Apress, 2006 г.).

Добавление обязательных зависимостей для примеров

Для разработки примеров этого раздела требуется ряд зависимостей, которые описаны в табл. 16.4. Добавьте их в свой проект.

Таблица 16.4. Зависимости Maven для веб-служб REST

Идентификатор группы	Идентификатор артефакта	Версия	Описание
`org.springframework`	`spring-oxm`	3.1.0.RELEASE	Модуль отображения объектов на XML для Spring
`org.codehaus.jackson`	`jackson-mapper-lgpl`	1.9.2	Процессор Jackson JSON для поддержки данных в формате JSON
`org.codehaus.castor`	`castor-xml`	1.3.2	Библиотека Castor XML, которая будет использоваться для маршализации и демаршализации XML-данных
`org.springframework.security`	`spring-security-core`	3.1.0.RELEASE	Модуль ядра Spring Security
`org.springframework.security`	`spring-security-web`	3.1.0.RELEASE	Веб-модуль Spring Security для защиты веб-служб REST
`org.springframework.security`	`spring-security-config`	3.1.0.RELEASE	Модуль конфигурации Spring Security
`org.apache.httpcomponents`	`httpclient`	4.1.2	Проект Apache HTTP Components. Библиотека клиента HTTP будет использоваться для вызова веб-служб REST

Веб-службы REST для контактной информации

Первый шаг при разработке приложения RESTful-WS заключается в проектировании структуры служб, которая включает поддерживаемые HTTP-методы и целевые URL для различных операций.

В веб-службах REST для контактов мы хотим поддерживать операции запроса, создания, обновления и удаления. В рамках операции запроса нам нужно поддерживать извлечение всех контактов и одиночного контакта по идентификатору.

Службы будут реализованы в виде контроллера Spring MVC. Именем класса будет `ContactController` из пакета `com.apress.prospring3.ch16.web.restful.controller`. Шаблон URL, HTTP-метод, описание и соответствующие методы контроллера показаны в табл. 16.5. Все URL используют префикс `http://localhost:8080/ch16/restful`. В качестве форматов данных будут поддерживаться XML и JSON. Соответствующий формат будет предоставляться согласно атрибуту `Accept` в заголовке клиентского HTTP-запроса.

Таблица 16.5. Проектное решение для веб-служб REST контактов

URL	HTTP-метод	Описание	Метод контроллера
/contact/listdata	GET	Извлечение всех контактов	listData(...)
/contact/{id}	GET	Извлечение одиночного контакта по заданному идентификатору	findContactById(...)
/contact	POST	Создание нового контакта	create(...)
/contact/{id}	PUT	Обновление существующего контакта с указанным идентификатором	update(...)
/contact	DELETE	Удаление контакта с указанным идентификатором	delete(...)

Использование Spring MVC для отображения веб-служб REST

В этом разделе мы покажем, как использовать Spring MVC для отображения служб, связанных с контактами, в виде веб-служб REST согласно проектному решению, представленному в предыдущем разделе.

Для начала мы создадим еще один объект предметной области — класс Contacts. Его код приведен в листинге 16.20.

Листинг 16.20. Класс Contacts

```
package com.apress.prospring3.ch16.domain;

import java.io.Serializable;
import java.util.List;

public class Contacts implements Serializable {

   private List<Contact> contacts;

   public Contacts() {
   }

   public Contacts(List<Contact> contacts) {
      this.contacts = contacts;
   }

   // Методы извлечения/установки не показаны.
}
```

Класс Contacts имеет единственное свойство, представляющее собой список объектов Contact. Это предназначено для поддержки преобразования списка контактов (возвращаемого методом listData() класса ContactController) в формат XML или JSON.

Конфигурирование библиотеки Castor XML

Для преобразования возвращаемой информации о контактах в формат XML мы будем пользоваться библиотекой Castor XML (http://castor.codehaus.org). Эта библиотека поддерживает множество режимов преобразования между POJO и XML, и в рассматриваемом примере для определения отображений мы будем применять XML-файл. Содержимое файла отображений (/src/main/resources/oxm-mapping.xml) приведено в листинге 16.21.

Листинг 16.21. Определение отображений для Castor XML

```
<mapping>
   <class name="com.apress.prospring3.ch16.domain.Contacts">
     <field name="contacts" type="com.apress.prospring3.ch16.domain.Contact"
            collection="arraylist">
          <bind-xml name="contact"/>
      </field>
   </class>
   <class name="com.apress.prospring3.ch16.domain.Contact" identity="id">
      <map-to xml="contact" />
      <field name="id" type="long">
         <bind-xml name="id" node="element"/>
      </field>
      <field name="firstName" type="string">
         <bind-xml name="firstName" node="element" />
      </field>
      <field name="lastName" type="string">
         <bind-xml name="lastName" node="element" />
      </field>
      <field name="birthDate" type="string" handler="dateHandler">
         <bind-xml name="birthDate" node="element" />
      </field>
      <field name="version" type="integer">
         <bind-xml name="version" node="element" />
      </field>
   </class>
   <field-handler name="dateHandler"
        class="com.apress.prospring3.ch16.xml.handler.DateTimeFieldHandler">
      <param name="date-format" value="yyyy-MM-dd"/>
   </field-handler>
</mapping>
```

Как показано в листинге 16.21, определены два отображения. Первый дескриптор `<class>` отображает класс `Contacts`, а внутри него свойство `contacts` (список объектов `Contact`) отображается с использованием дескриптора `<bind-xml name="contact"/>` (выделен полужирным). Затем отображается объект `Contact` (с помощью дескриптора `<map-to xml="contact" />` внутри второго дескриптора `<class>`). Кроме того, для поддержки преобразования из типа `DateTime`, определенного в Joda-Time (для атрибута `birthDate` в `Contact`), мы реализуем специальный обработчик полей Castor XML. Этот обработчик полей представлен в листинге 16.22.

Листинг 16.22. Специальный обработчик полей для типа `DateTime` в Castor XML

```
package com.apress.prospring3.ch16.xml.handler;

import java.util.Properties;

import org.exolab.castor.mapping.GeneralizedFieldHandler;
import org.exolab.castor.mapping.ValidityException;
import org.joda.time.DateTime;
import org.joda.time.format.DateTimeFormat;
import org.joda.time.format.DateTimeFormatter;
```

```
public class DateTimeFieldHandler extends GeneralizedFieldHandler {
   private static String dateFormatPattern;
   public void setConfiguration(Properties config) throws ValidityException {
      dateFormatPattern = config.getProperty("date-format");
   }
   public Object convertUponGet(Object value) {
      DateTime dateTime = (DateTime) value;
      return format(dateTime);
   }
   public Object convertUponSet(Object value) {
      String dateTimeString = (String) value;
      return parse(dateTimeString);
   }
   public Class<DateTime> getFieldType() {
      return DateTime.class;
   }
   protected static String format(final DateTime dateTime) {
      String dateTimeString = "";
      if (dateTime != null) {
         DateTimeFormatter dateTimeFormatter =
            DateTimeFormat.forPattern(dateFormatPattern);
         dateTimeString = dateTimeFormatter.print(dateTime);
      }
      return dateTimeString;
   }
   protected static DateTime parse(final String dateTimeString) {
      DateTime dateTime = new DateTime();
      if (dateTimeString != null) {
         DateTimeFormatter dateTimeFormatter =
         DateTimeFormat.forPattern(dateFormatPattern);
         dateTime = dateTimeFormatter.parseDateTime(dateTimeString);
      }
      return dateTime;
   }
}
```

Как показано в листинге 16.22, мы расширяем класс `org.exolab.castor.mapping.GeneralizedFieldHandler` из Castor XML и реализуем методы `convertUponGet()`, `convertUponSet()` и `getFieldType()`. Внутри этих методов мы реализуем логику преобразования между `DateTime` и `String`, предназначенную для Castor XML.

Вдобавок мы также определяем файл свойств для использования с Castor XML. Содержимое этого файла (`src/main/resources/castor.properties`) приведено в листинге 16.23.

Листинг 16.23. Файл `castor.properties`

```
org.exolab.castor.indent=true
```

В файле свойств библиотеке Castor XML указывается на необходимость генерации XML-кода с отступами, что намного упрощает его чтение во время тестирования.

Реализация класса ContactController

Следующий шаг состоит в реализации класса контроллера, ContactController. В листинге 16.24 показан класс ContactController, в котором реализованы все методы из табл. 16.5.

Листинг 16.24. Класс ContactController

```
package com.apress.prospring3.ch16.web.restful.controller;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.ResponseBody;

import com.apress.prospring3.ch16.domain.Contact;
import com.apress.prospring3.ch16.domain.Contacts;
import com.apress.prospring3.ch16.service.ContactService;

@Controller
@RequestMapping(value="/contact")
public class ContactController {

   final Logger logger = LoggerFactory.getLogger(ContactController.class);

   @Autowired
   private ContactService contactService;

   @RequestMapping(value = "/listdata", method = RequestMethod.GET)
   @ResponseBody
   public Contacts listData() {
      return new Contacts(contactService.findAll());
   }

   @RequestMapping(value="/{id}", method=RequestMethod.GET)
   @ResponseBody
   public Contact findContactById(@PathVariable Long id) {
      return contactService.findById(id);
   }

   @RequestMapping(value="/", method=RequestMethod.POST)
   @ResponseBody
   public Contact create(@RequestBody Contact contact) {
      logger.info("Creating contact: " + contact);
      contactService.save(contact);
      logger.info("Contact created successfully with info: " + contact);
      return contact;
   }

   @RequestMapping(value="/{id}", method=RequestMethod.PUT)
   @ResponseBody
   public void update(@RequestBody Contact contact,
      @PathVariable Long id) {
```

```
        logger.info("Updating contact: " + contact);
        contactService.save(contact);
        logger.info("Contact updated successfully with info: " + contact);
    }

    @RequestMapping(value="/{id}", method=RequestMethod.DELETE)
    @ResponseBody
    public void delete(@PathVariable Long id) {
        logger.info("Deleting contact with id: " + id);
        Contact contact = contactService.findById(id);
        contactService.delete(contact);
        logger.info("Contact deleted successfully");
    }
}
```

Ниже перечислены основные моменты, связанные с листингом 16.24.

- Класс аннотирован с помощью аннотации `@Controller`, указывающей на то, что он является контроллером Spring MVC.
- Аннотация уровня класса `@RequestMapping(value="/contact")` определяет, что этот контроллер будет отображаться на все URL в главном веб-контексте. В рассматриваемом примере данный контроллер будет обрабатывать все URL, начинающиеся с `http://localhost:8080/ch16/contact`.
- Реализованный ранее класс `ContactService` на уровне обслуживания автосвязан с контроллером.
- С помощью аннотаций `@RequestMapping` в каждом методе класса `ContactController` указываются шаблон URL и соответствующий HTTP-метод, на который будет осуществляться отображение. Например, метод `listData()` будет отображаться на URL вида `http://localhost:8080/ch16/contact/listdata` с HTTP-методом `GET`. Метод `update()` будет отображаться на URL вида `http://localhost:8080/ch16/contact/{id}` с HTTP-методом `PUT`.
- Аннотация `@ResponseBody` применяется ко всем методам. Она обеспечивает запись всех возвращаемых методами значений напрямую в поток HTTP-ответа.
- В методах, принимающих переменные пути (например, `findContactById()`), эта переменная пути аннотирована посредством `@PathVariable`. Это заставляет Spring MVC связывать переменную пути внутри URL (к примеру, `http://localhost:8080/ch16/contact/1`) с аргументом `id` метода `findContactById()`. Обратите внимание, что аргумент `id` имеет тип `Long`, но система преобразования типов Spring 3 автоматически выполнит проеобразование из `String` в `Long`.
- В методах `create()` и `update()` аргумент `Contact` аннотирован с помощью `@RequestBody`. Это заставляет Spring автоматически связывать содержимое тела HTTP-запроса с объектом предметной области `Contact`. Преобразование будет осуществляться объявленными экземплярами интерфейса `HttpMessageConverter<Object>` (из пакета `org.springframework.http.converter`) для поддержки форматов, которые обсуждаются далее в этой главе.

Конфигурирование сервлета REST

После завершения разработки контроллера его можно определить в Spring MVC. Сначала понадобится определить сервлет `DispatcherServlet` (из пакета `org.springframework.web.servlet`) для указания Spring MVC на необходимость диспетчеризации всех запросов REST к контроллеру `ContactController`.

Чтобы объявить сервлет, добавьте фрагмент кода из листинга 16.25 в файл дескриптора развертывания веб-приложений (`src/main/webapp/WEB-INF/web.xml`).

Листинг 16.25. Сервлет диспетчера для веб-служб REST

```
<!-- Сервлет REST -->
<servlet>
   <servlet-name>restful</servlet-name>
   <servlet-class>
      org.springframework.web.servlet.DispatcherServlet
   </servlet-class>
   <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>
         /WEB-INF/spring/appServlet/restful-context.xml
      </param-value>
   </init-param>
   <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
   <servlet-name>restful</servlet-name>
   <url-pattern>/restful/*</url-pattern>
</servlet-mapping>
```

Как показано в листинге 16.25, был объявлен сервлет по имени `restful`, имеющий тип `DispatcherServlet` (детали, связанные с `DispatcherServlet`, ищите в главах 17 и 18). В Spring MVC каждый сервлет `DispatchServlet` будет иметь собственный контекст `WebApplicationContext` (однако все бины уровня обслуживания, определенные в файле `root-context.xml`, который вызывается корневым `WebApplicationContext`, будут доступны также и для `WebApplicationContext` каждого сервлета).

Дескриптор `<servlet-mapping>` указывает веб-контейнеру (к примеру, Tomcat), что все URL в шаблоне `/restful/*` (скажем, `http://localhost:8080/ch16/restful/contact`) будут обрабатываться сервлетом `restful`.

Для сервлета `restful` в листинге 16.25 мы также указываем, что контекст `WebApplicationContext` для данного `DispatcherServlet` должен быть загружен из файла `/WEB-INF/spring/appServlet/restful-context.xml`. Содержимое этого конфигурационного файла представлено в листинге 16.26.

Листинг 16.26. Конфигурация `WebApplicationContext` для веб-служб REST

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:context="http://www.springframework.org/schema/context"
   xmlns:mvc="http://www.springframework.org/schema/mvc"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context-3.1.xsd
      http://www.springframework.org/schema/mvc
      http://www.springframework.org/schema/mvc/spring-mvc-3.1.xsd">
   <mvc:annotation-driven>
      <mvc:message-converters>
         <bean class="org.springframework.http.converter.json.
MappingJacksonHttpMessageConverter"/>
```

```
        <bean class="org.springframework.http.converter.xml.
MarshallingHttpMessageConverter">
            <property name="marshaller" ref="castorMarshaller"/>
            <property name="unmarshaller" ref="castorMarshaller"/>
        </bean>
    </mvc:message-converters>
  </mvc:annotation-driven>
  <context:component-scan
    base-package="com.apress.prospring3.ch16.web.restful.controller"/>

  <bean id="castorMarshaller"
    class="org.springframework.oxm.castor.CastorMarshaller">
    <property name="mappingLocation" value="classpath:oxm-mapping.xml"/>
  </bean>
</beans>
```

Ниже отмечены важные моменты, связанные с листингом 16.26.

- Дескриптор `<mvc:annotation-driven>` включает поддержку аннотаций для Spring MVC (т.е. аннотацию `@Controller`), а также регистрирует систему преобразования типов и форматирования Spring 3. Кроме того, под определением этого дескриптора также включается поддержка проверки достоверности JSR-303.
- Внутри `<mvc:annotation-driven>` дескриптор `<mvc:message-converters>` объявляет экземпляры `HttpMessageConverter`, которые будут использоваться для преобразований в поддерживаемые форматы. Обратите внимание на то, что дескриптор `<mvc:message-converters>` появился в версии Spring 3.1. Поскольку мы планируем поддерживать форматы данных JSON и XML, объявляются два преобразователя. Первый из них — это `MappingJacksonHttpMessageConverter`, предлагаемый библиотекой Jackson JSON (`http://jackson.codehaus.org`). Второй преобразователь, `MarshallingHttpMessageConverter`, предоставляется модулем `spring-oxm` и предназначен для маршализации/демаршализации XML. Внутри `MarshallingHttpMessageConverter` мы должны определить используемые маршализатор и демаршализатор, которые в данном случае предоставляются библиотекой Castor XML.
- Для бина `castorMarshaller` используется Spring-класс `org.springframework.oxm.castor.CastorMarshaller`, который интегрируется с Castor, и мы указываем местоположение файла отображения, требуемого для работы Castor.
- Дескриптор `<context:component-scan>` указывает Spring пакеты для сканирования на предмет классов контроллеров.

На этом разработка службы серверной стороны завершена. Если экземпляр сервера tc Server функционирует, проект будет собран и развернут автоматически. Если нет, выполните сборку проекта и запустите сервер.

При запуске веб-приложения вы можете обнаружить, что Spring сообщает об ошибках, возникающих из-за ранее реализованных примеров. Чтобы исправить это, выполните следующие действия.

- Перенесите класс контроллера `HomeController` из пакета `com.apress.prospring3` в новый пакет по имени `com.apress.prospring3.ch16.web.app.controller`.
- В файле `src/main/webapp/WEB-INF/spring/appServlet/servlet-context.xml` замените строку `<context:component-scan base-package="com.apress.prospring3"/>` строкой `<context:component-scan base-package="com.apress.prospring3.ch16.web.app.controller"/>`.

Использование curl для тестирования веб-служб REST

Давайте проведем быстрый тест реализованных нами веб-служб REST. Один из простых способов тестирования предполагает применение curl (http://curl.haxx.se) — инструмента командной строки для передачи данных в синтаксисе URL. Для использования этого инструмента загрузите его из соответствующего веб-сайта и распакуйте у себя на компьютере.

Например, чтобы протестировать извлечение всех контактов, откройте окно командной строки в Windows или окно терминала в Unix/Linux и запустите команду, приведенную в листинге 16.27.

Листинг 16.27. Команда curl для тестирования веб-службы REST в формате JSON

```
curl -v -H "Accept: application/json"
http://localhost:8080/ch16/restful/contact/listdata
```

Команда в листинге 16.27 отправит HTTP-запрос серверной веб-службе REST; в этом случае будет вызван метод listData() класса ContactController для извлечения и возврата всех контактов. Опция -H объявляет атрибут HTTP-заголовка, который указывает, что клиент ожидает получения данных в формате JSON. В результате выполнения этой команды будет получен вывод, показанный на рис. 16.13.

```
C:\Users\Clarence>curl -v -H "Accept: application/json" http://localhost:8080/ch16/restful/contact/listdata
* About to connect() to localhost port 8080 (#0)
* connected
* Connected to localhost () port 8080 (#0)
> GET /ch16/restful/contact/listdata HTTP/1.1
> User-Agent: curl/7.21.0 (amd64-pc-win32) libcurl/7.21.0 OpenSSL/0.9.8o zlib/1.2.3
> Host: localhost:8080
> Accept: application/json
>
< HTTP/1.1 200 OK
< Server: Apache-Coyote/1.1
< Content-Type: application/json
< Transfer-Encoding: chunked
< Date: Thu, 01 Dec 2011 03:42:24 GMT
<
{"contacts":[{"id":1,"version":0,"firstName":"Clarence","lastName":"Ho","birthDate":333734400000},{"id":2,"version":0,"f
irstName":"Scott","lastName":"Tiger","birthDate":657475200000},{"id":3,"version":0,"firstName":"John","lastName":"Smith"
,"birthDate":-184406400000}]}* Connection #0 to host localhost left intact
* Closing connection #0
```

Рис. 16.13. Тестирование веб-службы REST с использованием curl в формате JSON

На рис. 16.13 видно, что были возвращены данные в формате JSON для изначально заполненных контактов. А теперь давайте взглянем на формат XML; соответствующая команда представлена в листинге 16.28.

Листинг 16.28. Команда curl для тестирования веб-службы REST в формате XML

```
curl -v -H "Accept: application/xml"
http://localhost:8080/ch16/restful/contact/listdata
```

По сравнению с листингом 16.27 команда в листинге 16.28 имеет только одно отличие. Атрибут Accept изменен с JSON на XML. Запуск этой команды даст вывод, показанный на рис. 16.14.

На рис. 16.14 легко заметить, что данные в формате XML были корректно возвращены веб-службой. Это объясняется объявлением экземпляров HttpMessageConverter в контексте WebApplicationContext сервлета REST; инфраструктура Spring MVC будет

вызывать соответствующий преобразователь сообщений на основе атрибута `Accept`, указанного в HTTP-заголовке клиента, и записывать результат в HTTP-ответ.

```
C:\Users\Clarence>curl -v -H "Accept: application/xml" http://localhost:8080/ch16/restful/contact/listdata
* About to connect() to localhost port 8080 (#0)
* connected
* Connected to localhost () port 8080 (#0)
> GET /ch16/restful/contact/listdata HTTP/1.1
> User-Agent: curl/7.21.0 (amd64-pc-win32) libcurl/7.21.0 OpenSSL/0.9.8o zlib/1.2.3
> Host: localhost:8080
> Accept: application/xml
>
< HTTP/1.1 200 OK
< Server: Apache-Coyote/1.1
< Content-Type: application/xml
< Transfer-Encoding: chunked
< Date: Thu, 01 Dec 2011 03:42:55 GMT
<
<?xml version="1.0" encoding="UTF-8"?>
<contacts>
    <contact>
        <id>1</id>
        <firstName>Clarence</firstName>
        <lastName>Ho</lastName>
        <birthDate>1980-07-30</birthDate>
        <version>0</version>
    </contact>
    <contact>
        <id>2</id>
        <firstName>Scott</firstName>
        <lastName>Tiger</lastName>
        <birthDate>1990-11-02</birthDate>
        <version>0</version>
    </contact>
    <contact>
        <id>3</id>
        <firstName>John</firstName>
        <lastName>Smith</lastName>
        <birthDate>1964-02-28</birthDate>
        <version>0</version>
    </contact>
</contacts>
* Connection #0 to host localhost left intact
```

Рис. 16.14. Тестирование веб-службы REST с использованием `curl` в формате XML

Использование класса `RestTemplate` для доступа к веб-службам REST

Для Spring-приложений спроектирован класс `RestTemplate`, предназначенный для доступа к веб-службам REST. В этом разделе мы посмотрим, как использовать этот класс для доступа к службе контакта на сервере.

Вначале давайте взглянем на базовую конфигурацию `ApplicationContext` для Spring-класса `RestTemplate`, которая представлена в листинге 16.29 (`restful-client-app-context.xml`).

Листинг 16.29. Файл `restful-client-app-context.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd">

   <bean id="restTemplate"
      class="org.springframework.web.client.RestTemplate">
      <property name="messageConverters">
         <list>
            <bean class=
"org.springframework.http.converter.xml.MarshallingHttpMessageConverter">
```

```
                <property name="marshaller" ref="castorMarshaller"/>
                <property name="unmarshaller" ref="castorMarshaller"/>
                <property name="supportedMediaTypes">
                   <list>
                      <bean class="org.springframework.http.MediaType">
                         <constructor-arg index="0" value="application"/>
                         <constructor-arg index="1" value="xml"/>
                      </bean>
                   </list>
                </property>
             </bean>
          </list>
       </property>
    </bean>

    <bean id="castorMarshaller"
       class="org.springframework.oxm.castor.CastorMarshaller">
       <property name="mappingLocation" value="classpath:oxm-mapping.xml"/>
    </bean>
</beans>
```

В листинге 16.29 объявлен бин `restTemplate`, использующий класс `RestTemplate`. В этом классе внедрено свойство `messageConverters` с экземпляром `MarshallingHttpMessageConverter`, который работает с библиотекой Castor; это тот же самый экземпляр, что и применяемый на стороне сервера. Файл отображений будет совместно использоваться на серверной и клиентской стороне. Кроме того, в бине `restTemplate` внутри анонимного бина класса `MarshallingHttpMessageConverter` внедрено свойство `supportedMediaTypes` с объявлением анонимного бина класса `MediaType`, указывающее единственный поддерживаемый формат XML. В результате клиент всегда ожидает XML в качестве формата возвращаемых данных, и библиотека Castor XML поможет выполнять преобразования между POJO и XML.

Давайте опробуем службу для получения всех контактов. Соответствующая тестовая программа приведена в листинге 16.30.

Листинг 16.30. Тестирование `RestTemplate` для операции извлечения всех контактов

```
package com.apress.prospring3.ch16.restful;

import org.springframework.context.support.GenericXmlApplicationContext;
import org.springframework.web.client.RestTemplate;

import com.apress.prospring3.ch16.domain.Contact;
import com.apress.prospring3.ch16.domain.Contacts;

public class RestfulClientSample {

   private static final String URL_GET_ALL_CONTACTS =
      "http://localhost:8080/ch16/restful/contact/listdata";
   private static final String URL_GET_CONTACT_BY_ID =
      "http://localhost:8080/ch16/restful/contact/{id}";
   private static final String URL_CREATE_CONTACT =
      "http://localhost:8080/ch16/restful/contact/";
   private static final String URL_UPDATE_CONTACT =
      "http://localhost:8080/ch16/restful/contact/{id}";
   private static final String URL_DELETE_CONTACT =
      "http://localhost:8080/ch16/restful/contact/{id}";
```

```
    public static void main(String[] args) {

        GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
        ctx.load("classpath:restful-client-app-context.xml");
        ctx.refresh();

        Contacts contacts;
        Contact contact;

        RestTemplate restTemplate = ctx.getBean(
            "restTemplate", RestTemplate.class);

        // Протестировать извлечение всех контактов.
        System.out.println("Testing retrieve all contacts:");
        contacts = restTemplate.getForObject(
            URL_GET_ALL_CONTACTS, Contacts.class);
        listContacts(contacts);

    }

    private static void listContacts(Contacts contacts) {
        for (Contact contact: contacts.getContacts()) {
            System.out.println(contact);
        }
        System.out.println("");
    }
}
```

В листинге 16.30 объявлены URL для доступа к различным операциям, которые будут использоваться в последующих примерах. В методе `main()` извлекается экземпляр `RestTemplate` и затем вызывается метод `RestTemplate.getForObject()` (соответствующий HTTP-методу `GET`), которому передается URL и ожидаемый тип возврата — класс `Contacts`, содержащий полный список контактов.

Удостоверьтесь, что сервер tc Server функционирует. Запуск этой программы даст следующий вывод:

```
Testing retrieve all contacts:
Contact - Id: 1, First name: Clarence, Last name: Ho, Birthday:
1980-07-30T00:00:00.000+08:00
Contact - Id: 2, First name: Scott, Last name: Tiger, Birthday:
1990-11-02T00:00:00.000+08:00
Contact - Id: 3, First name: John, Last name: Smith, Birthday:
1964-02-28T00:00:00.000+08:00
```

Как видите, экземпляр `MarshallingHttpMessageConverter`, зарегистрированный внутри `RestTemplate`, автоматически преобразует сообщение в POJO.

А теперь попытаемся извлечь контакт по идентификатору. Добавьте фрагмент кода из листинга 16.31 в метод `main()` класса `RestfulClientSample`.

Листинг 16.31. Тестирования `RestTemplate` для операции извлечения контакта по идентификатору

```
        // Протестировать извлечение контакта по идентификатору.
        System.out.println("Testing retrieve a contact by id :");
        contact = restTemplate.getForObject(
            URL_GET_CONTACT_BY_ID, Contact.class, 1);
        System.out.println(contact);
        System.out.println("");
```

В листинге 16.31 используется версия метода RestTemplate.getForObject(), которой также передается идентификатор контакта для извлечения как переменная пути внутри URL (переменная пути {id} в URL_GET_CONTACT_BY_ID). Если URL содержит более одной переменной пути, для передачи этих переменных можно применять экземпляр Map<String,Object> или поддержку переменного количества аргументов. В последнем случае переменные пути должны следовать в порядке их объявления в URL. Запуск программы теперь даст такой вывод (показан не полностью):

```
Testing retrieve all contact by id :
Contact - Id: 1, First name: Clarence, Last name: Ho, Birthday:
1980-07-30T00:00:00.000+08:00
```

Как видите, был извлечен правильный контакт.

Наступила очередь обновления. Добавьте фрагмент кода из листинга 16.32 в метод main() класса RestfulClientSample.

Листинг 16.32. Тестирование RestTemplate для операции обновления контакта

```
// Протестировать обновление контакта.
contact = restTemplate.getForObject(
   URL_UPDATE_CONTACT, Contact.class, 1);
contact.setFirstName("Kim Fung");
System.out.println("Testing update contact by id :");
restTemplate.put(URL_UPDATE_CONTACT, contact, 1);
System.out.println("Contact update successfully: " + contact);
System.out.println("");
```

Как показано в листинге 16.32, сначала мы извлекаем контакт, предназначенный для обновления. После обновления объекта контакта мы используем метод RestTemplate.put(), соответствующий HTTP-методу PUT, передавая ему URL обновления, модифицированный объект контакта и идентификатор обновляемого контакта. Запуск программы дает следующий вывод (показан не полностью):

```
Testing update contact by id :
Contact update successfully: Contact - Id: 1, First name: Kim Fung,
Last name: Ho, Birthday: 1980-07-30T00:00:00.000+08:00
```

Поскольку мы указали инфраструктуре Hibernate на необходимость отображения SQL и занесения в журнал информации во время обновления, в консольном выводе сервера tc Server можно увидеть такие строки:

```
Hibernate: update contact set BIRTH_DATE=?, FIRST_NAME=?, LAST_NAME=?,
VERSION=? where ID=? and VERSION=?
INFO : com.apress.prospring3.ch16.web.restful.controller.
ContactController — Contact updated successfully with info:
Contact - Id: 1, First name: Kim Fung, Last name: Ho, Birthday:
1980-07-30T00:00:00.000+08:00
```

Далее мы протестируем операцию удаления. Добавьте фрагмент кода из листинга 16.33 в метод main() класса RestfulClientSample.

Листинг 16.33. Тестирование RestTemplate для операции удаления контакта

```
// Протестировать удаление контакта.
restTemplate.delete(URL_DELETE_CONTACT, 1);
System.out.println("Testing delete contact by id :");
contacts = restTemplate.getForObject(
   URL_GET_ALL_CONTACTS, Contacts.class);
listContacts(contacts);
```

В листинге 16.33 вызывается метод `RestTemplate.delete()`, соответствующий HTTP-методу `DELETE`, которому передается URL и идентификатор. После этого извлекаются все контакты и отображаются для подтверждения удаления. Запуск программы дает следующий вывод (показан не полностью):

```
Testing delete contact by id :
Contact - Id: 2, First name: Scott, Last name: Tiger, Birthday:
1990-11-02T00:00:00.000+08:00
Contact - Id: 3, First name: John, Last name: Smith, Birthday:
1964-02-28T00:00:00.000+08:00
```

Легко заметить, что контакт с идентификатором 1 был успешно удален.

Наконец, давайте проверим операцию вставки. Добавьте фрагмент кода из листинга 16.34 в метод `main()` класса `RestfulClientSample`.

Листинг 16.34. Тестирование `RestTemplate` для операции вставки контакта

```
// Протестировать вставку нового контакта.
System.out.println("Testing create contact :");
Contact contactNew = new Contact();
contactNew.setFirstName("James");
contactNew.setLastName("Gosling");
contactNew.setBirthDate(new DateTime());
contactNew = restTemplate.postForObject(
   URL_CREATE_CONTACT, contactNew, Contact.class);
System.out.println("Contact created successfully: " + contactNew);
```

В листинге 16.34 сначала создается новый экземпляр объекта `Contact`. Затем вызывается метод `RestTemplate.postForObject()`, соответствующий HTTP-методу `POST`, которому передается URL, созданный экземпляр `Contact` и тип класса. Выполнение тестовой программы (первым делом перезапустите сервер tc Server для повторной инициализации данных контактов) даст в результате следующий вывод:

```
Testing create contact :
Contact created successfully: Contact - Id: 4, First name: James,
Last name: Gosling, Birthday: 2011-12-01T00:00:00.000+08:00
```

Новый контакт был создан на сервере и возвращен клиенту.

Защита веб-служб REST с помощью Spring Security

Любая удаленная служба требует защиты от несанкционированного доступа для извлечения бизнес-информации или взаимодействия с ней. Веб-службы REST не являются исключением. В этом разделе мы покажем, как использовать проект Spring Security (`http://static.springsource.org/spring-security/site/index.html`) для защиты веб-служб REST на сервере. В примере, рассматриваемом в этом разделе, мы будем пользоваться версией Spring Security 3.1 (на момент написания книги последним выпуском был 3.1.0.RELEASE), которая предлагает более удобную поддержку веб-служб REST.

Применение Spring Security для защиты веб-служб REST производится в три этапа. На первом этапе в дескрипторе развертывания веб-приложения (`web.xml`) понадобится объявить фильтр; соответствующий фрагмент кода приведен в листинге 16.35.

Листинг 16.35. Объявление фильтра Spring Security в `web.xml`

```
<!-- Конфигурация Spring Security -->
<filter>
   <filter-name>springSecurityFilterChain</filter-name>
   <filter-class>
      org.springframework.web.filter.DelegatingFilterProxy
   </filter-class>
</filter>

<filter-mapping>
   <filter-name>springSecurityFilterChain</filter-name>
   <url-pattern>/restful/*</url-pattern>
</filter-mapping>
```

В листинге 16.35 объявлен фильтр, который позволяет Spring Security перехватывать HTTP-запрос для выполнения проверок, связанных с аутентификацией и авторизацией. Поскольку мы лишь хотим защитить веб-службы REST, фильтр применен только к шаблону URL вида `/restful/*` (в дескрипторе `<filter-mapping>`).

На втором этапе нужно создать конфигурацию Spring Security, которая будет располагаться в корневом `WebApplicationContext`. Содержимое файла `src/main/webapp/WEB-INF/spring/web-security.xml` показано в листинге 16.36.

Листинг 16.36. Конфигурация Spring Security

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/security"
   xmlns:beans="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
      http://www.springframework.org/schema/security
      http://www.springframework.org/schema/security/spring-security-3.1.xsd">

   <!-- Служба REST, не поддерживающая состояния
       и использующая базовую аутентификацию -->
   <http pattern="/restful/**" create-session="stateless">
      <intercept-url pattern='/**' access='ROLE_REMOTE' />
      <http-basic />
   </http>

   <authentication-manager>
      <authentication-provider>
         <user-service>
            <user name="remote" password="remote"
               authorities="ROLE_REMOTE" />
         </user-service>
      </authentication-provider>
   </authentication-manager>

</beans:beans>
```

В листинге 16.36 мы объявляем пространство имен `security` (обратите внимание на строку `xmlns="http://www.springframework.org/schema/security"`) и затем используем его как пространство имен по умолчанию в конфигурационном файле. В дескрипторе `<http>` мы указываем, что должны защищаться ресурсы под URL `/restful/**`. Атрибут `create-session`, появившийся в Spring Security 3.1.0, позволяет сконфигурировать, бу-

дет ли при аутентификации создаваться HTTP-сеанс. Так как используемые веб-службы REST не поддерживают состояния, мы устанавливаем значение этого атрибута в stateless, что указывает Spring Security не создавать HTTP-сеанс для всех запросов REST. В результате можно улучшить показатели производительности веб-служб REST.

В дескрипторе <intercept-url> мы указываем, что получать доступ к веб-службам REST могут только пользователи с назначенной ролью ROLE_REMOTE. Дескриптор <http-basic/> определяет, что для веб-служб REST поддерживается только базовая аутентификация HTTP.

Дескриптор <authentication-manager> определяет информацию для аутентификации. Здесь мы задаем простой поставщик аутентификации с жестко закодированным именем пользователя и паролем (оба установлены в remote), а также назначенной ролью ROLE_REMOTE. В корпоративной среде аутентификация, скорее всего, будет производиться либо через базу данных, либо за счет поиска LDAP.

На третьем этапе мы импортируем конфигурацию Spring Security в корневой WebApplicationContext. Добавьте в файл src/main/webapp/WEB-INF/spring/root-context.xml следующую строку:

```
<import resource="web-security.xml" />
```

На этом настройка защиты завершена. Проект потребуется повторно собрать и развернуть. Запустив приложение с классом RestfulClientSample еще раз, вы увидите следующий вывод (показан не полностью):

```
Exception in thread "main" org.springframework.web.cient.
HttpClientErrorException: 401 Unauthorized
```

Вы получите код состояния HTTP 401, который означает отсутствие авторизации для доступа к службе. Теперь давайте сконфигурируем RestTemplate клиента для предоставления учетных данных серверу.

Для начала понадобится модифицировать конфигурацию клиентского приложения REST. В листинге 16.37 показано измененное содержимое конфигурационного файла restful-client-app-context.xml.

Листинг 16.37. Модифицированный файл restful-client-app-context.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd">

   <bean id="restTemplate"
      class="org.springframework.web.client.RestTemplate">
      <constructor-arg ref="httpRequestFactory"/>
      <property name="messageConverters">
         <!-- Настройки совпадают, поэтому здесь не показаны. -->
      </property>
   </bean>

   <bean id="castorMarshaller"
      class="org.springframework.oxm.castor.CastorMarshaller">
      <property name="mappingLocation" value="classpath:oxm-mapping.xml"/>
   </bean>

   <bean id="httpRequestFactory"
class="org.springframework.http.client.HttpComponentsClientHttpRequestFactory">
      <constructor-arg>
```

```
        <bean class="org.apache.http.impl.client.DefaultHttpClient">
           <property name="credentialsProvider">
              <bean class="com.apress.prospring3.ch16.restful.support.
CustomCredentialsProvider">
                 <property name="credentials">
                    <bean class="org.apache.http.auth.
UsernamePasswordCredentials">
                       <constructor-arg name="userName" value="remote"/>
                       <constructor-arg name="password" value="remote"/>
                    </bean>
                 </property>
              </bean>
           </property>
        </bean>
     </constructor-arg>
   </bean>
</beans>
```

Отличия от предыдущей версии выделены полужирным. В бине `restTemplate` внедрен аргумент конструктора со ссылкой на бин `httpRequestFactory`. Для бина `httpRequestFactory` используется класс `HttpComponentsClientHttpRequestFactory`, который представляет собой поддержку Spring для библиотеки HttpClient проекта Apache HttpComponents; эта библиотека нужна для конструирования экземпляра `DefaultHttpClient`, хранящего учетные данные для клиента. Для поддержки внедрения учетных данных мы реализуем простой класс `CustomCredentialsProvider`, показанный в листинге 16.38.

Листинг 16.38. Класс `CustomCredentialsProvider`

```
package com.apress.prospring3.ch16.restful.support;

import org.apache.http.auth.AuthScope;
import org.apache.http.auth.Credentials;
import org.apache.http.impl.client.BasicCredentialsProvider;

public class CustomCredentialsProvider extends BasicCredentialsProvider {

   public void setCredentials(Credentials credentials) {
      this.setCredentials(AuthScope.ANY, credentials);
   }

}
```

В листинге 16.38 видно, что класс `CustomCredentialsProvider` расширяет класс `BasicCredentialsProvider` из библиотеки HttpComponents, и реализован новый метод установки для поддержки внедрения учетных данных. Вернувшись к листингу 16.37, вы увидите, что учетные данные внедрены в этот класс с использованием экземпляра класса `UsernamePasswordCredentials`. Класс `UsernamePasswordCredentials` сконструирован с именем удаленного пользователя и паролем. За счет внедрения сконструированного `httpRequestFactory` в `RestTemplate` все запросы REST, отправленные с применением этого шаблона, будут содержать учетные данные. Если теперь снова запустить приложение с классом `RestfulClientSample`, можно увидеть, что службы вызываются обычным образом.

Использование JSR-303 с веб-службами REST

Последней темой, которую мы рассмотрим в этой главе, является бесшовная интеграция Spring MVC с API-интерфейсом проверки достоверности бинов JSR-303. При использовании веб-служб REST можно применять заранее определенные правила проверки достоверности к аргументам запроса.

Давайте вернемся к сущностному классу `Contact` и определим ограничение на атрибуте `firstName`. В листинге 16.39 показан фрагмент кода переделанного метода `getFirstName()`.

Листинг 16.39. Модифицированный метод `getFirstName()` класса `Contact`

```
@NotNull
@Size(min=3, max=60)
@Column(name = "FIRST_NAME")
public String getFirstName() {
   return firstName;
}
```

После этого поместим в метод `create()` класса `ContactController` аннотацию `@Valid` (`javax.validation.Valid`) перед аргументом `contact`. В листинге 16.40 приведен фрагмент кода переделанного метода `create()` класса `ContactController`.

Листинг 16.40. Модифицированный метод `create()` класса `ContactController`

```
@RequestMapping(value="/", method=RequestMethod.POST)
@ResponseBody
public Contact create(@RequestBody @Valid Contact contact) {
   logger.info("Creating contact: " + contact);
   contactService.save(contact);
   logger.info("Contact created successfully with info: " + contact);
   return contact;
}
```

Отличия от предыдущей версии выделены полужирным. Поддержка аннотации `@Valid` для аргументов также появилась в версии Spring 3.1. При наличии такой аннотации Spring будет применять проверку достоверности JSR-303 к объекту предметной области `Contact` после привязки данных, и если обнаружатся нарушения, будут сгенерированы соответствующие исключения.

Чтобы увидеть это в действии, изменим операцию вставки в классе `RestfulClientSample`. В листинге 16.41 представлен модифицированный фрагмент кода в классе `RestfulClientSample`, предназначенный для тестирования операции вставки.

Листинг 16.41. Тестирование модифицированной операции вставки в классе `RestfulClientSample`

```
// Протестировать вставку нового контакта.
System.out.println("Testing create contact :");
Contact contactNew = new Contact();
contactNew.setFirstName("JJ");
contactNew.setLastName("Gosling");
contactNew.setBirthDate(new DateTime());
contactNew = restTemplate.postForObject(
   URL_CREATE_CONTACT, contactNew, Contact.class);
System.out.println("Contact created successfully: " + contactNew);
```

Отличия от предыдущей версии в листинге 16.41 выделены полужирным. При установке имени мы преднамеренно указываем только два символа, что нарушает правило (о минимальных трех символах), определенное для атрибута.

Чтобы увидеть также нарушения в журнале на сервере, понадобится установить уровень ведения журнала пакета `org.springframework.web` в `debug`. Для этого найдите в файле `src/main/resources/log4j.xml` соответствующий фрагмент кода и измените уровень ведения журнала с `info` на `debug`, как показано в листинге 16.42.

Листинг 16.42. Изменение уровня ведения журнала Log4j

```
<logger name="org.springframework.web">
   <level value="debug" />
</logger>
```

Пересоберите проект и разверните его на сервере tc Server. Запустив приложение с классом `RestfulClientSample` снова, вы увидите следующий вывод (показан не полностью):

```
Exception in thread "main" org.springframework.web.client.
HttpClientErrorException: 400 Bad Request
```

В случае если проверка достоверности не проходит, Spring MVC автоматически возвращает код состояния HTTP 400, который означает, что данные в теле запроса вызывают ошибку. Ниже показан неполный консольный вывод на сервере:

```
DEBUG: org.springframework.web.servlet.mvc.support.
DefaultHandlerExceptionResolver — Resolving exception from handler
[public com.apress.prospring3.ch16.domain.Contact com.apress.prospring3.
ch16.web.restful.controller.ContactController.create (com.apress.
prospring3.ch16.domain.Contact)]: org.springframework.web.method.
annotation.support.
MethodArgumentNotValidException: Validation failed for argument at
index [0] in method: public com.apress.prospring3.ch16.domain.Contact
com.apress.prospring3.ch16.web.restful.controller.ContactController.
create(com.apress.prospring3.ch16.domain.Contact), with 1 error(s):
[Field error in object 'contact' on field 'firstName': rejected
value [JJ]; codes [Size.contact.firstName,Size.firstName,Size.java.
lang.String,Size]; arguments[org.springframework.context.support.
DefaultMessageSourceResolvable: codes [contact.firstName,firstName];
arguments []; default message [firstName],60,3]; default message
[size must be between 3 and 60]]
```

В этом выводе отображается сообщение об ошибке проверки достоверности JSR-303. В реальности вы должны обеспечить, чтобы клиент знал об этой ошибке. Это достигается проектированием объекта POJO, который хранит ошибки и возвращает их клиенту, тем самым уведомляя о том, что с запросом связаны проблемы. Мы оставляем это в качестве упражнения для самостоятельной проработки.

Удаленная обработка в примере приложения

В приложении SpringBlog основная функциональность удаленной обработки связана с извлечением записей блога клиентами. Для этого мы будем использовать всеобъемлющую поддержку веб-служб REST в Spring MVC.

Ниже перечислены основные аспекты реализации службы ленты блога.

- Будет определен сервлет REST для удаленного извлечения записей блога через веб-службы REST.
- На веб-уровне будет реализован контроллер для принятия запросов (с предварительно определенным URL и HTTP-методом `GET`) и возврата последних отправленных записей блога клиенту.
- Будут поддерживаться форматы данных JSON и XML, а также применяться приемы, описанные в настоящей главе.

За более подробной информацией обращайтесь в главу 21.

Резюме

В этой главе рассматривались наиболее часто используемые технологии удаленной обработки в Spring-приложениях.

Если взаимодействующие приложения основаны на Spring, то удобным вариантом будет применение HTTP-активатора Spring. Если требуется асинхронный режим или слабая связность, то общепринятым выбором будет JMS. Наконец, было показано, как использовать веб-службы REST в Spring для отображения служб или доступа к ним с помощью класса `RestTemplate`.

В следующей главе мы обсудим применение Spring для реализации веб-уровня.

ГЛАВА 17

Разработка веб-приложений в Spring

В корпоративном приложении уровень презентаций является критически важным и оказывает значительное влияние на степень принятия приложения пользователями. Уровень презентаций — это своего рода "парадная дверь" в приложение. Он позволяет пользователям выполнять бизнес-функции, предоставляемые приложением, а также обеспечивает визуальное представление информации, которой управляет приложение. Эффективность пользовательского интерфейса вносит весьма существенный вклад в успех приложения в целом.

Из-за бурного роста Интернета (особенно благодаря облачным вычислениям и появлению различных типов устройств) разработка уровня презентаций в приложении становится довольно непростой задачей. Ниже приведены некоторые из основных соображений, которые должны приниматься во внимание во время разработки веб-приложений.

- *Производительность.* Производительность всегда была главным требованием к веб-приложению. Если пользователи после выбора какой-то функции или щелчка на ссылке вынуждены слишком долго ожидать выполнения (в мире Интернета, десяток секунд подобен столетию), то они определенно не будет удовлетворены таким приложением.
- *Дружественность к пользователю.* Приложение должно быть простым в использовании и навигации с очевидными инструкциями, не вводящими пользователя в заблуждение.
- *Интерактивность и широкие возможности.* Пользовательский интерфейс должен быть интерактивным и отзывчивым. Кроме того, уровень презентаций должен быть обогащенным в визуальном смысле, включая диаграммы, интерфейс в виде панелей и т.п.
- *Доступность.* В наши дни пользователи требуют, чтобы приложение было доступным откуда угодно и на любом устройстве. На работе они будут использовать для доступа к приложению настольные компьютеры. В дороге пользователи будут осуществлять доступ к приложению из разнообразных мобильных устройств, таких как ноутбуки, планшеты, смартфоны и т.д.

Построение веб-приложения, удовлетворяющего указанным выше требованиям, не является простой, однако данные требования рассматриваются бизнес-пользователями как обязательные. К счастью, для решения этих потребностей также были разработаны многие новые технологии и инфраструктуры. В последнее время многие плат-

формы и библиотеки для разработки веб-приложений — среди которых Spring MVC, Struts, Tapestry, Java Server Faces (JSF), Adobe Flex, Google Web Toolkit (GWT), jQuery и Dojo — предлагают инструменты и развитые библиотеки компонентов, помогающие строить высокоинтерактивные пользовательские интерфейсы для веб-сети. Кроме того, многие инфраструктуры предоставляют инструменты или соответствующие библиотеки виджетов, которые предназначены для мобильных устройств, включая смартфоны и планшеты. Вдобавок появление стандартов HTML5 и CSS3 вместе с их поддержкой большинством веб-браузеров и производителей мобильных устройств помогает упростить разработку веб-приложений, которые должны быть доступными откуда угодно и на любом устройстве.

Платформа Spring предлагает широкую и интенсивную поддержку разработки веб-приложений. Модуль Spring MVC обеспечивает надежную инфраструктуру и архитектуру MVC (Model View Controller — модель-представление-контроллер) для построения веб-приложений. Вместе с модулем Spring MVC можно использовать различные технологии представлений (например, JSP, Velocity и т.д.). Кроме того, Spring MVC интегрируется со многими известными веб-платформами и инструментами (в том числе Struts, Adobe Flex, GWT и т.п.). Доступны также другие проекты Spring, которые помогают решать специфические потребности веб-приложений. Например, модуль Spring MVC, скомбинированный с проектом Spring Web Flow и его модулем Spring Faces, предлагает развернутую поддержку разработки веб-приложений со сложными потоками, используя JSF в качестве технологии представлений. Проще говоря, для построения уровня презентаций доступно очень много вариантов.

В этой главе внимание будет сосредоточено на Spring MVC и будет показано, как использовать мощные средства Spring MVC для разработки высокопроизводительных веб-приложений. В частности, в главе рассматриваются следующие темы.

- *Spring MVC*. Мы обсудим основные концепции шаблона MVC и предоставим введение в Spring MVC. Затем мы рассмотрим ключевые концепции Spring MVC, включая иерархию `WebApplicationContext` и жизненный цикл обработки запроса.
- *Интернационализация, локаль и оформление темами*. В Spring MVC предлагается расширенная поддержка для удовлетворения общих требований к веб-приложениям, включая интернационализацию (i18n), локаль и оформление темами. Мы обсудим, как применять Spring MVC для разработки веб-приложений, поддерживающих эти требования.
- *Поддержка технологий представления и Ajax*. В Spring MVC поддерживаются многие технологии представления. В этой главе мы сосредоточимся на использовании JavaServer Pages (JSP) в качестве технологии представления для веб-приложения. Для реализации расширенных возможностей вместе с JSP будет применяться JavaScript. Существует много выдающихся и популярных JavaScript-библиотек, таких как jQuery, Dojo и т.д. В этой главе мы покажем, как работать с jQuery — подпроектом библиотеки jQuery UI, которая поддерживает построение интерактивных веб-приложений.
- *Поддержка разбиения на страницы и загрузки файлов*. При разработке примеров в этой главе мы обсудим применение Spring Data JPA и компонента пользовательского интерфейса jQuery для обеспечения поддержки разбиения на страницы во время просмотра данных в виде экранной сетки. Кроме того, будет показано, как реализовать загрузку файлов в Spring MVC. Вместо интеграции с Apache Commons File Upload мы рассмотрим, как использовать для загрузки файлов Spring MVC с поддержкой множественного содержимого, встроенной в контейнер Servlet 3.0.

- *Безопасность.* Безопасность — это крупная тема в рамках построения веб-приложений. Мы обсудим, как использовать Spring Security для защиты приложения и обработки входа и выхода из него.
- *Spring MVC и пример приложения.* Многие аспекты, рассмотренные в этой главе, будут применяться при разработке примера приложения SpringBlog. Мы дадим высокоуровневый обзор отношений между темами этой главы и примером приложения.

Создание проекта для примеров в STS

Как обычно, первый шаг заключается в создании проекта для примеров этой главы. Поскольку мы обсуждаем веб-приложения, мы должны создать проект Spring MVC. Кроме того, мы построим уровень обслуживания с использованием JPA 2 с Hibernate и Spring Data JPA в качестве поставщика постоянства. За деталями по созданию проекта обращайтесь в главу 16; там же описаны обязательные зависимости. Не забудьте изменить имя проекта и имена всех пакетов, чтобы в них применялась строка `ch17`, а не `ch16`. После создания проекта проверьте его работоспособность, о чем будет говорить отображение домашней страницы по умолчанию, созданной шаблонным проектом.

Реализация уровня обслуживания для примеров

На уровне обслуживания, предназначенном для целей этой главы, мы по-прежнему будем пользоваться приложением контактов, как это было в главе 16. Однако сам уровень обслуживания и модель данных будут немного сложнее, чем в главе 16, поскольку мы хотим продемонстрировать такие возможности веб-приложений, как поддержка загрузки файлов. В этом разделе мы обсудим модель данных и реализацию уровня обслуживания, которые будут использоваться далее в главе.

Модель данных для примеров

Модель данных для примеров этой главы очень проста и содержит единственную таблицу `CONTACT`, предназначенную для хранения информации о контактах.

В листинге 17.1 показан сценарий для создания схемы базы данных (файл `schema.sql` в папке `/src/main/resources`).

Листинг 17.1. Сценарий для создания схемы базы данных

```
DROP TABLE IF EXISTS CONTACT;

CREATE TABLE CONTACT (
      ID INT NOT NULL AUTO_INCREMENT
    , FIRST_NAME VARCHAR(60) NOT NULL
    , LAST_NAME VARCHAR(40) NOT NULL
    , BIRTH_DATE DATE
    , DESCRIPTION VARCHAR(2000)
    , PHOTO BLOB
    , VERSION INT NOT NULL DEFAULT 0
    , UNIQUE UQ_CONTACT_1 (FIRST_NAME, LAST_NAME)
    , PRIMARY KEY (ID)
);
```

Как видите, таблица `CONTACT` содержит только несколько базовых полей контактной информации. Единственным моментом, стоящим упоминания, является столбец `PHOTO`

с типом данных BLOB (binary large object — большой двоичный объект), который будет использоваться для хранения фотографии контакта с применением загрузки файлов. В листинге 17.2 приведен код сценария для наполнения таблицы тестовыми данными (/src/main/resources/test-data.sql).

Листинг 17.2. Сценарий для наполнения тестовыми данными

```
insert into contact (first_name, last_name, birth_date)
values ('Clarence', 'Ho', '1980-07-30');
insert into contact (first_name, last_name, birth_date)
values ('Scott', 'Tiger', '1990-11-02');
insert into contact (first_name, last_name, birth_date)
values ('John', 'Smith', '1964-02-28');
insert into contact (first_name, last_name, birth_date)
values ('Peter', 'Jackson', '1944-1-10');
insert into contact (first_name, last_name, birth_date)
values ('Jacky', 'Chan', '1955-10-31');
insert into contact (first_name, last_name, birth_date)
values ('Susan', 'Boyle', '1970-05-06');
insert into contact (first_name, last_name, birth_date)
values ('Tinner', 'Turner', '1967-04-30');
insert into contact (first_name, last_name, birth_date)
values ('Lotus', 'Notes', '1990-02-28');
insert into contact (first_name, last_name, birth_date)
values ('Henry', 'Dickson', '1997-06-30');
insert into contact (first_name, last_name, birth_date)
values ('Sam', 'Davis', '2001-01-31');
insert into contact (first_name, last_name, birth_date)
values ('Max', 'Beckham', '2002-02-01');
insert into contact (first_name, last_name, birth_date)
values ('Paul', 'Simon', '2002-02-28');
```

На этот раз объем тестовых данных больше, поскольку планируется демонстрация поддержки разбиения на страницы.

Реализация и конфигурирование интерфейса ContactService

После создания проекта, модели данных и сценариев можно приступать к реализации и конфигурированию уровня обслуживания для примеров этой главы.

В последующих разделах мы обсудим реализацию интерфейса ContactService с использованием JPA 2, Spring Data JPA и Hibernate в качестве поставщика постоянства. Затем мы покажем, как сконфигурировать уровень обслуживания в проекте Spring.

Реализация интерфейса ContactService

В примерах этой главы мы сделаем доступными уровню презентаций службы для разнообразных операций над контактной информацией. Первым делом необходимо создать сущностный класс Contact, который приведен в листинге 17.3.

Листинг 17.3. Сущностный класс Contact

```
package com.apress.prospring3.ch17.domain;
import static javax.persistence.GenerationType.IDENTITY;
import java.io.Serializable;
import javax.persistence.*;
import org.hibernate.annotations.Type;
```

```
import org.joda.time.DateTime;
import org.springframework.format.annotation.DateTimeFormat;
import org.springframework.format.annotation.DateTimeFormat.ISO;

@Entity
@Table(name = "contact")
public class Contact implements Serializable {
   private Long id;
   private int version;
   private String firstName;
   private String lastName;
   private DateTime birthDate;
   private String description;
   private byte[] photo;
   @Id
   @GeneratedValue(strategy = IDENTITY)
   @Column(name = "ID")
   public Long getId() {
      return id;
   }

   @Version
   @Column(name = "VERSION")
   public int getVersion() {
      return version;
   }

   @Column(name = "FIRST_NAME")
   public String getFirstName() {
      return firstName;
   }
   @Column(name = "LAST_NAME")
   public String getLastName() {
      return lastName;
   }
   @Column(name = "BIRTH_DATE")
   @Type(type="org.joda.time.contrib.hibernate.PersistentDateTime")
   @DateTimeFormat(iso=ISO.DATE)
   public DateTime getBirthDate() {
      return birthDate;
   }
   @Column(name = "DESCRIPTION")
   public String getDescription() {
      return description;
   }
   @Basic(fetch=FetchType.LAZY)
   @Lob @Column(name = "PHOTO")
   public byte[] getPhoto() {
      return photo;
   }
   @Transient
   public String getBirthDateString() {
      String birthDateString = "";
      if (birthDate != null)
         birthDateString =
org.joda.time.format.DateTimeFormat.forPattern("yyyy-MM-dd").print(birthDate);
         return birthDateString;
   }
```

```
    public String toString() {
        return "Contact - Id: " + id + ", First name: " + firstName
            + ", Last name: " + lastName + ", Birthday: " + birthDate
            + ", Description: " + description;
    }
    // Методы установки не показаны.
}
```

Как показано в листинге 17.3, в классе применяются стандартные аннотации. Мы также используем класс `DateTime` из Joda-Time для атрибута `birthDate`. Кроме того, обратите внимание на следующие моменты.

- Добавлено новое прозрачное свойство (за счет применения аннотации `@Transient` к методу извлечения) по имени `birthDateString`, которое будет использоваться для представления в пользовательском интерфейсе в последующих примерах.
- Для атрибута `photo` в качестве типа данных Java применяется байтовый массив, который соответствует типу данных `BLOB` в СУРБД. Вдобавок метод извлечения аннотирован посредством `@Lob` и `@Basic(fetch=FetchType.LAZY)`. Первая аннотация указывает поставщику JPA, что это столбец большого объекта, а вторая — что атрибут должен извлекаться ленивым образом, чтобы не оказывать влияния на производительность, когда загружается класс, не требующий фотографии.

Давайте продолжим разработку уровня обслуживания. В листинге 17.4 показан интерфейс `ContactService` со службами, которые должны быть отображены.

Листинг 17.4. Интерфейс `ContactService`

```
package com.apress.prospring3.ch17.service;

import java.util.List;

import com.apress.prospring3.ch17.domain.Contact;

public interface ContactService {

public List<Contact> findAll();

public Contact findById(Long id);

public Contact save(Contact contact);
}
```

Методы этого интерфейса самоочевидны. Поскольку мы будем пользоваться поддержкой репозитория Spring Data JPA, реализуем интерфейс `ContactRepository` (листинг 17.5).

Листинг 17.5. Интерфейс `ContactRepository`

```
package com.apress.prospring3.ch17.repository;

import org.springframework.data.repository.CrudRepository;

import com.apress.prospring3.ch17.domain.Contact;

public interface ContactRepository extends CrudRepository<Contact, Long> {
}
```

Как обсуждалось в главе 10, интерфейс `ContactRepository` расширяет интерфейс `CrudRepository<T,ID extends Serializable>` из модуля Spring Data Commons, который уже предоставляет базовые методы, требуемые в примерах.

В листинге 17.6 приведен класс реализации интерфейса `ContactService`.

Листинг 17.6. Класс `ContactServiceImpl`

```
package com.apress.prospring3.ch17.service.jpa;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import com.apress.prospring3.ch17.domain.Contact;
import com.apress.prospring3.ch17.repository.ContactRepository;
import com.apress.prospring3.ch17.service.ContactService;
import com.google.common.collect.Lists;

@Service("contactService")
@Repository
@Transactional
public class ContactServiceImpl implements ContactService {

    @Autowired
    private ContactRepository contactRepository;

    @Transactional(readOnly=true)
    public List<Contact> findAll() {
        return Lists.newArrayList(contactRepository.findAll());
    }

    @Transactional(readOnly=true)
    public Contact findById(Long id) {
        return contactRepository.findOne(id);
    }

    public Contact save(Contact contact) {
        return contactRepository.save(contact);
    }
}
```

При наличии проблем с пониманием кода в листинге 17.6, обратитесь в главу 10 за детальным описанием реализации уровня обслуживания, где в качестве поставщика постоянства используется JPA. Теперь реализация в основном завершена, и следующий шаг заключается в конфигурировании службы в `ApplicationContext` проекта веб-приложения, что будет темой следующего раздела.

Конфигурирование уровня обслуживания

Для настройки уровня обслуживания в проекте Spring MVC мы создаем отдельный конфигурационный файл по имени `datasource-tx-jpa.xml` в папке `/src/main/resources`. Содержимое этого файла показано в листинге 17.7.

Листинг 17.7. Конфигурационный файл datasource-tx-jpa.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:context="http://www.springframework.org/schema/context"
   xmlns:jdbc="http://www.springframework.org/schema/jdbc"
   xmlns:jpa="http://www.springframework.org/schema/data/jpa"
   xmlns:tx="http://www.springframework.org/schema/tx"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context-3.1.xsd
      http://www.springframework.org/schema/jdbc
      http://www.springframework.org/schema/jdbc/spring-jdbc-3.1.xsd
      http://www.springframework.org/schema/data/jpa
      http://www.springframework.org/schema/data/jpa/spring-jpa-1.0.xsd
      http://www.springframework.org/schema/tx
      http://www.springframework.org/schema/tx/spring-tx-3.1.xsd">

   <jdbc:embedded-database id="dataSource" type="H2">
      <jdbc:script location="classpath:schema.sql"/>
      <jdbc:script location="classpath:test-data.sql"/>
   </jdbc:embedded-database>

   <bean id="transactionManager"
         class="org.springframework.orm.jpa.JpaTransactionManager">
      <property name="entityManagerFactory" ref="emf"/>
   </bean>

   <tx:annotation-driven transaction-manager="transactionManager" />

   <bean id="emf"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
      <property name="dataSource" ref="dataSource" />
      <property name="jpaVendorAdapter">
       <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter" />
      </property>
      <property name="packagesToScan" value="com.apress.prospring3.ch17.domain"/>
      <property name="jpaProperties">
         <props>
            <prop key="hibernate.dialect">
                  org.hibernate.dialect.H2Dialect
            </prop>
            <prop key="hibernate.max_fetch_depth">3</prop>
            <prop key="hibernate.jdbc.fetch_size">50</prop>
            <prop key="hibernate.jdbc.batch_size">10</prop>
            <prop key="hibernate.show_sql">true</prop>
         </props>
      </property>
   </bean>

   <context:annotation-config/>

   <jpa:repositories base-package="com.apress.prospring3.ch17.repository"
                  entity-manager-factory-ref="emf"
                  transaction-manager-ref="transactionManager"/>
</beans>
```

Подробные сведения об определениях бинов ищите в главе 10. Теперь необходимо импортировать конфигурацию в корневой контекст `WebApplicationContext`. Для шаблонного проекта Spring MVC соответствующий файл расположен в `/src/main/webapp/WEB-INF/spring/root-context.xml`. Модифицированное содержимое этого файла приведено в листинге 17.8.

Листинг 17.8. Файл `root-context.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:context="http://www.springframework.org/schema/context"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context-3.1.xsd">

   <import resource="classpath:datasource-tx-jpa.xml" />

   <context:component-scan
      base-package="com.apress.prospring3.ch17.service.jpa" />
</beans>
```

Сначала в этот конфигурационный файл было добавлено пространство имен `context`. Затем файл `datasource-tx-jpa.xml` был импортирован в `WebApplicationContext` и, наконец, платформе Spring было указано сканировать заданный пакет в поисках бинов Spring. Теперь уровень обслуживания завершен и готов к отображению и использованию удаленными клиентами.

Введение в MVC и Spring MVC

Перед тем, как заняться реализацией уровня презентаций, давайте рассмотрим некоторые основные концепции MVC как шаблона проектирования веб-приложений и выясним, каким образом Spring MVC обеспечивает развитую поддержку в этой области.

В последующих разделах мы по очереди обсудим эти высокоуровневые концепции. Во-первых, мы предоставим краткое введение в MVC. Во-вторых, мы дадим высокоуровневое представление инфраструктуры Spring MVC и ее иерархии `WebApplicationContext`. В-третьих, мы рассмотрим жизненный цикл обработки запроса в Spring MVC.

Введение в MVC

MVC — это шаблон, который часто используется при реализации уровня презентаций в приложении. Главный принцип шаблона MVC состоит в определении архитектуры с четкой ответственностью для каждого компонента. В шаблоне MVC различают три участника.

- *Модель*. Модель представляет бизнес-данные, а также “состояние” приложения внутри контекста конкретного пользователя. Например, на веб-сайте электронной коммерции модель будет включать информацию профиля пользователя, данные тележки для покупок и данные заказа, если пользователи приобретают товары на этом сайте.
- *Представление*. Представление отображает данные пользователю в желаемом формате, поддерживает взаимодействие с пользователями и выполняет проверку достоверности клиентской стороны, интернационализацию, оформление стилями и т.д.

- *Контроллер*. Контроллер обрабатывает запросы к действиям, осуществляемые пользователями в пользовательском интерфейсе, взаимодействуя с уровнем обслуживания, обновляя модель и направляя пользователей на соответствующие представления в зависимости от результатов выполнения.

В связи с появлением веб-приложений на основе Ajax шаблон MVC был расширен для предоставления более отзывчивого и развитого пользовательского интерфейса. Например, когда применяется JavaScript, представление может "прослушивать" события или действия, предпринимаемые пользователем, и затем отправлять запрос `XMLHttpRequest` серверу. На стороне контроллера вместо возврата целого представления производится возврат низкоуровневых данных (скажем, в формате XML или JSON), после чего JavaScript-код осуществляет "частичное" обновление представления с применением полученных данных. Эта концепция изображена на рис. 17.1.

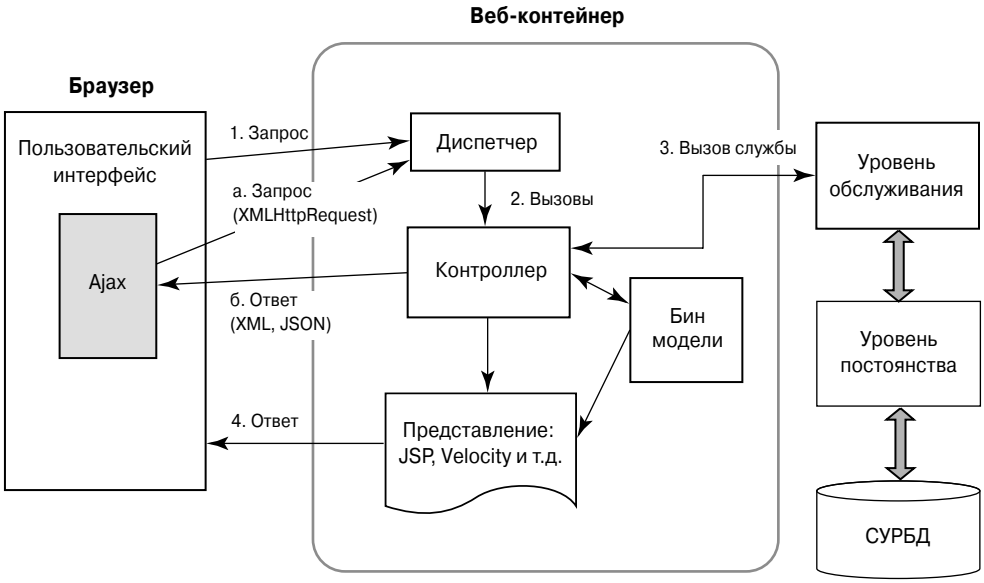


Рис. 17.1. Шаблон MVC

На рис. 17.1 показан часто используемый шаблон веб-приложения, который можно трактовать как расширение традиционного шаблона MVC. Обычный запрос представления обрабатывается следующим образом.

- *Запрос*. Запрос отправляется серверу. На стороне сервера большинство инфраструктур (например, Spring MVC, Struts и т.д.) будут иметь диспетчер (в форме сервлета) для обработки запроса.
- *Вызовы*. Диспетчер направляет запрос соответствующему контроллеру на основе информации HTTP-запроса и конфигурации веб-приложения.
- *Вызов службы*. Контроллер взаимодействует с уровнем обслуживания.
- *Ответ*. Контроллер обновляет модель и в зависимости от результата выполнения возвращает соответствующее представление пользователю.

Кроме того, внутри представления происходят вызовы Ajax. Например, предположим, что пользователь просматривает данные в экранной сетке. Когда он щелкает на

ссылке для перехода к следующей странице, вместо обновления всей страницы будет инициирован следующий поток.

- *Запрос.* Запрос `XMLHttpRequest` подготавливается и отправляется серверу. Диспетчер направит запрос соответствующему контроллеру.
- *Ответ.* Контроллер взаимодействует с уровнем обслуживания, и данные ответа будут сформатированы и отправлены браузеру. В этом случае никакие представления не задействованы. Браузер получает данные и выполняет частичное обновление существующего представления.

Введение в Spring MVC

В Spring Framework модуль Spring MVC предоставляет развернутую поддержку шаблона MVC, а также других средств (например, оформление темами, интернационализацию, проверку достоверности, преобразование типов и форматирование и т.п.), упрощающих реализацию уровня презентаций.

В последующих разделах мы обсудим основные концепции Spring MVC, в том числе иерархию `WebApplicationContext` в Spring MVC, типичный жизненный цикл обработки запроса и конфигурирование.

Иерархия `WebApplicationContext` в Spring MVC

В Spring MVC класс `DispatcherServlet` является центральным сервлетом, который получает запросы и направляет их соответствующим контроллерам. В приложении Spring MVC может существовать произвольное количество экземпляров `DispatcherServlet`, предназначенных для разных целей (например, для обработки запросов пользовательского интерфейса, запросов веб-служб REST и т.д.). Каждый экземпляр `DispatcherServlet` имеет собственную конфигурацию `WebApplicationContext`, которая определяет характеристики уровня сервлета, такие как контроллеры, поддерживающие сервлет, отображение обработчиков, распознавание представлений, интернационализация, оформление темами, проверка достоверности, преобразование типов и форматирование и т.п.

Помимо конфигураций `WebApplicationContext` уровня сервлетов, в Spring MVC также поддерживается корневая конфигурация `WebApplicationContext`, которая включает конфигурации уровня приложения, в том числе для источников данных серверной стороны, безопасности, уровней обслуживания и постоянства и т.д. Корневая конфигурация `WebApplicationContext` будет доступна всем конфигурациям `WebApplicationContext` уровня сервлетов.

Давайте рассмотрим пример. Предположим, что в приложении имеются два экземпляра `DispatcherServlet`. Один сервлет предназначен для поддержки пользовательского интерфейса (назовем его сервлетом приложения), а другой — для предоставления служб в форме веб-служб REST другим приложениям (назовем его сервлетом REST). В Spring MVC мы определим корневую конфигурацию `WebApplicationContext` и конфигурации `WebApplicationContext` для двух `DispatcherServlet`s. На рис. 17.2 показана иерархия `WebApplicationContext`, которая будет поддерживаться Spring MVC для описанного сценария.

Жизненный цикл запроса Spring MVC

А теперь посмотрим, как Spring MVC обрабатывает запрос. На рис. 17.3 показаны основные компоненты, вовлеченные в обработку запроса внутри Spring MVC. Эта диаграмма основана на диаграмме, вынесенной на обсуждение на форуме Spring Framework (`http://forum.springsource.org/showthread.php?21639-Spring-MVC-Request-Lifecycle-Diagram`), но с рядом модификаций.

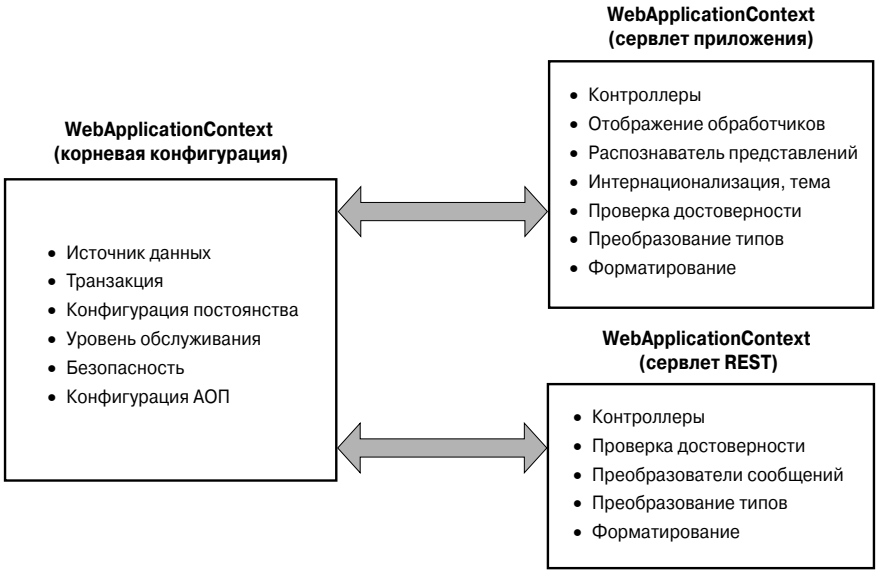


Рис. 17.2. Иерархия `WebApplicationContext` в Spring MVC

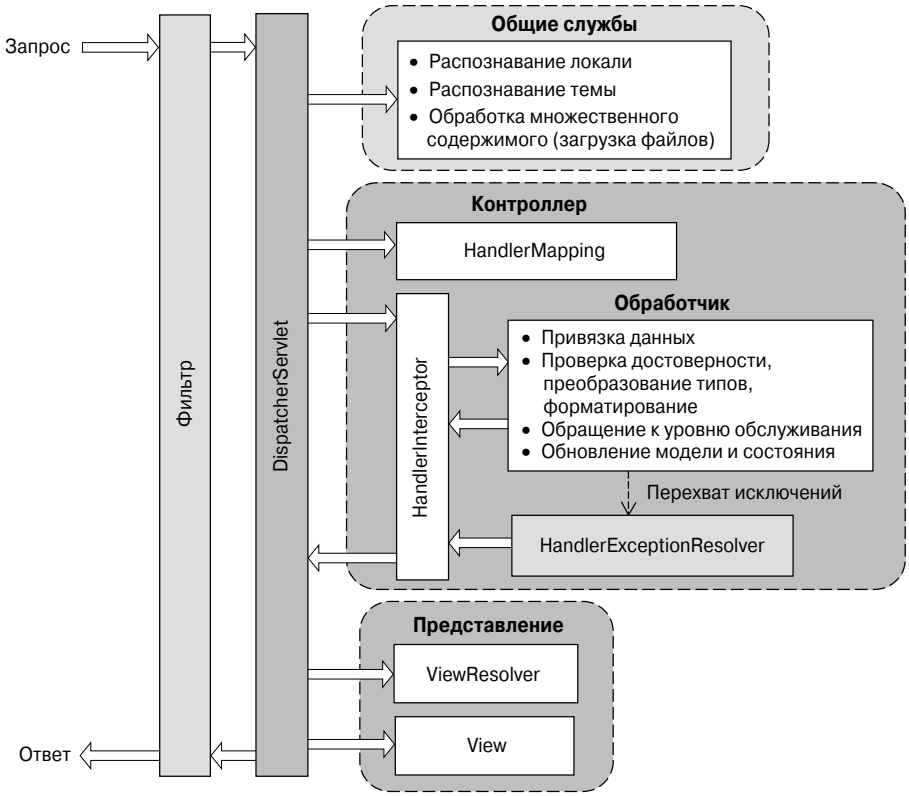


Рис. 17.3. Жизненный цикл запроса Spring MVC

Ниже перечислены основные компоненты и указано их назначение.

- *Фильтр.* Фильтр применяется к каждому запросу. Часто используемые фильтры вместе с их предназначением описаны в следующем разделе.
- *Сервлет диспетчера.* Этот сервлет анализирует запросы и направляет их соответствующему контроллеру для обработки.
- *Общие службы.* Общие службы будут применяться к каждому запросу для предоставления поддержки интернационализации, оформления темами, загрузки файлов и т.д. Их конфигурация определена в `WebApplicationContext` сервлета диспетчера.
- *Отображение обработчиков.* Отображает запрос на обработчик (метод внутри класса контроллера Spring MVC). Начиная с версии Spring 2.5, в большинстве ситуаций конфигурация отображения не требуется, поскольку Spring MVC автоматически регистрирует класс `org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping`, который отображает обработчики на основе путей HTTP, выраженных через аннотацию `@RequestMapping` на уровне типа или метода внутри классов контроллеров.
- *Перехватчик обработчиков.* В Spring MVC можно зарегистрировать перехватчики для обработчиков, чтобы реализовать общую проверку или логику. Например, перехватчик обработчиков может проверять, что обработчики вызываются только в рабочие часы.
- *Распознаватель исключений в обработчиках.* В Spring MVC интерфейс `HandlerExceptionResolver` (из пакета `org.springframework.web.servlet`) предназначен для работы с непредвиденными исключениями, возникающими во время выполнения обработчиков. По умолчанию `DispatcherServlet` регистрирует класс `DefaultHandlerExceptionResolver` (из пакета `org.springframework.web.servlet.mvc.support`). Этот распознаватель обрабатывает определенные стандартные исключения Spring MVC, устанавливая специальный код состояния ответа. Можно также реализовать собственный обработчик исключений, аннотировав метод контроллера с помощью аннотации `@ExceptionHandler` и передав ей в качестве атрибута тип исключения.
- *Распознаватель представлений.* Интерфейс `ViewResolver` в Spring MVC (из пакета `org.springframework.web.servlet`) поддерживает распознавание представлений на основе логического имени, возвращаемого контроллером. Для поддержки различных механизмов распознавания представлений предусмотрено множество классов реализации. Например, класс `UrlBasedViewResolver` поддерживает прямое преобразование логических имен в URL. Класс `ContentNegotiatingViewResolver` поддерживает динамическое распознавание представлений в зависимости от типа медиа, поддерживаемого клиентом (XML, PDF, JSON и т.д.). Существует также несколько реализаций для интеграции с различными технологиями представлений, такими как FreeMarker (`FreeMarkerViewResolver`), Velocity (`VelocityViewResolver`) и JasperReports (`JasperReportsViewResolver`).

Приведенные описания касаются только немногих часто используемых обработчиков и распознавателей. За полным описанием обращайтесь в справочную документацию Spring Framework.

Конфигурация Spring MVC

Чтобы сделать доступным модуль Spring MVC в веб-приложении, требуется некоторая начальная конфигурация, особенно для дескриптора веб-развертывания `web.xml`, находящегося в папке `/src/main/webapp/WEB-INF` (в Spring 3.1 появилась новая воз-

можность конфигурации на основе кода в веб-контейнере Servlet 3.0, которую мы обсудим в разделе “Поддержка конфигурации на основе кода для Servlet 3” далее в главе).

Для настройки поддержки Spring MVC для веб-приложений понадобится выполнить следующее конфигурирование в дескрипторе веб-развертывания:

- конфигурирование корневого `WebApplicationContext`;
- конфигурирование фильтров сервлетов, требуемых Spring MVC;
- конфигурирование сервлетов диспетчеров внутри приложения.

В шаблонном проекте Spring MVC генерируемый им файл `web.xml` поддерживает Servlet 2.5. В этой главе мы будем использовать Servlet 3.0 (сервер tc Server, поставляемый вместе с STS, построен поверх Apache Tomcat 7, который уже поддерживает Servlet 3.0), поэтому также необходимо изменить в XML-заголовке версию 2.5 на 3.0. Модифицированное содержимое файла `web.xml` приведено в листинге 17.9.

Листинг 17.9. Дескриптор веб-развертывания для Spring MVC

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
      http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">

   <!-- Определение корневого контейнера Spring,
       разделяемого всеми сервлетами и фильтрами -->
   <context-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>/WEB-INF/spring/root-context.xml</param-value>
   </context-param>
   <!-- Фильтры Spring MVC -->
   <filter>
      <filter-name>CharacterEncodingFilter</filter-name>
      <filter-class>
         org.springframework.web.filter.CharacterEncodingFilter
      </filter-class>
      <init-param>
         <param-name>encoding</param-name>
         <param-value>UTF-8</param-value>
      </init-param>
      <init-param>
         <param-name>forceEncoding</param-name>
         <param-value>true</param-value>
      </init-param>
   </filter>
   <filter>
      <filter-name>HttpMethodFilter</filter-name>
      <filter-class>
         org.springframework.web.filter.HiddenHttpMethodFilter
      </filter-class>
   </filter>
   <filter>
      <filter-name>Spring OpenEntityManagerInViewFilter</filter-name>
      <filter-class>
         org.springframework.orm.jpa.support.OpenEntityManagerInViewFilter
      </filter-class>
   </filter>
```

```
   <filter-mapping>
      <filter-name>CharacterEncodingFilter</filter-name>
      <url-pattern>/*</url-pattern>
   </filter-mapping>

   <filter-mapping>
      <filter-name>HttpMethodFilter</filter-name>
      <url-pattern>/*</url-pattern>
   </filter-mapping>

   <filter-mapping>
      <filter-name>Spring OpenEntityManagerInViewFilter</filter-name>
      <url-pattern>/*</url-pattern>
   </filter-mapping>

   <!-- Создание контейнера Spring, разделяемого всеми сервлетами
        и фильтрами -->
   <listener>
      <listener-class>
         org.springframework.web.context.ContextLoaderListener
      </listener-class>
   </listener>

   <!-- Обработка запросов приложения -->
   <servlet>
      <servlet-name>appServlet</servlet-name>
      <servlet-class>
         org.springframework.web.servlet.DispatcherServlet
      </servlet-class>
      <init-param>
         <param-name>contextConfigLocation</param-name>
         <param-value>
            /WEB-INF/spring/appServlet/servlet-context.xml
         </param-value>
      </init-param>
      <load-on-startup>1</load-on-startup>
   </servlet>

   <servlet-mapping>
      <servlet-name>appServlet</servlet-name>
      <url-pattern>/</url-pattern>
   </servlet-mapping>

</web-app>
```

Ниже отмечены основные моменты, связанные с листингом 17.9.

- В дескрипторе `<web-app>` атрибут `version` и соответствующий URL изменены для версии 3.0, чтобы отразить использование веб-приложением веб-контейнера Servlet 3.0.
- В дескрипторе `<context-param>` указан параметр `contextConfigLocation`, который определяет местоположение корневого файла конфигурации `WebApplicationContext` для Spring.
- Определено несколько фильтров сервлетов, предоставляемых Spring MVC, и все фильтры отображены на URL корневого контекста веб-приложения. Эти фильтры часто используются в веб-приложениях. В табл. 17.1 перечислены сконфигурированные фильтры и указано их назначение.

- Определен слушатель класса `org.springframework.web.context.ContextLoaderListener`. Он позволяет Spring загрузить и завершить корневой `WebApplicationContext`.
- Определен один сервлет диспетчера (по имени `appServlet`). Мы используем сервлет, сгенерированный шаблонным проектом для уровня презентаций в приложении контактов. Контекст `WebApplicationContext` для сервлета диспетчера расположен в `/src/main/webapp/WEB-INF/spring/appServlet/servlet-context.xml`.

Таблица 17.1. Часто используемые фильтры сервлетов Spring MVC

Имя класса фильтра	Описание
`org.springframework.web.filter.CharacterEncodingFilter`	Этот фильтр используется для указания кодировки символов в запросе
`org.springframework.web.filter.HiddenHttpMethodFilter`	Этот фильтр предоставляет поддержку для HTTP-методов, отличных от `GET` и `POST` (например, `PUT`)
`org.springframework.orm.jpa.support.OpenEntityManagerInViewFilter`	Этот фильтр связывает диспетчер сущностей JPA (`EntityManager`) с потоком для полной обработки запроса. Он может помочь в восстановлении того же диспетчера сущностей для последующих запросов, поступающих от того же самого пользователя, что позволит функционировать таким средствам JPA, как ленивая выборка

Создание первого представления в Spring MVC

Теперь, имея уровень обслуживания и готовую конфигурацию Spring MVC, можно приступать к реализации нашего первого представления. В этом разделе мы реализуем простое представление для отображения всех контактов, которые были изначально занесены в базу данных сценарием `test-data.sql`.

Как упоминалось ранее, для реализации представления мы будем использовать формат JSPX, который является форматом JSP с правильно построенным XML. Ниже перечислены основные преимущества JSPX перед JSP.

- JPSX требует более строгого разделения кода от уровня презентаций. Например, в документ JSPX не могут быть помещены скриптлеты Java.
- Доступны инструменты для выполнения моментальной проверки (синтаксиса XML), поэтому ошибки могут быть обнаружены на ранних этапах.
- Формат JSPX является рекомендуемым при использовании JSP с инфраструктурой Spring MVC, а также с другими проектами Spring (Spring Web Flow, Spring Roo и т.п.).

Для подготовки первого представления потребуется одна новая зависимость Maven, описанная в табл. 17.2. Добавьте ее в свой проект.

Таблица 17.2. Зависимость Maven для представления Spring MVC

Идентификатор группы	Идентификатор артефакта	Версия	Описание
`joda-time`	`joda-time-jsptags`	1.1.1	Библиотека дескрипторов JSP, которая поддерживает форматирование типов Joda-Time в представлениях

Конфигурирование сервлета диспетчера

Следующий шаг заключается в конфигурировании сервлета диспетчера. Чтобы реализовать это простое представление, необходимо внести небольшие изменения в конфигурационный файл по умолчанию, сгенерированный шаблонным проектом Spring. Модифицированная конфигурация (файл /src/main/webapp/WEB-INF/spring/appServlet/servlet-context.xml) представлена в листинге 17.10.

Листинг 17.10. Конфигурация DispatcherServlet

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/mvc"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:beans="http://www.springframework.org/schema/beans"
   xmlns:context="http://www.springframework.org/schema/context"
   xsi:schemaLocation="http://www.springframework.org/schema/mvc
      http://www.springframework.org/schema/mvc/spring-mvc-3.1.xsd
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context-3.1.xsd">

   <annotation-driven />

   <resources mapping="/resources/**" location="/resources/" />

   <beans:bean
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
      <beans:property name="prefix" value="/WEB-INF/views/" />
      <beans:property name="suffix" value=".jspx" />
   </beans:bean>

   <context:component-scan
      base-package="com.apress.prospring3.ch17.web.controller" />

</beans:beans>
```

В листинге 17.10 объявляется пространство имен mvc как используемое по умолчанию. Дескриптор <annotation-driven> включает поддержку конфигурирования посредством аннотаций для контроллеров Spring MVC, а также поддержку преобразования типов и форматирования Spring 3. Кроме того, этим же дескриптором включается поддержка проверки достоверности бинов JSR-303. Дескриптор <resources> определяет статические ресурсы (CSS, JavaScript, изображения и т.д.) вместе с их местоположениями, так что Spring MVC сможет улучшить производительность при обслуживании этих файлов. Дескриптор <context:component-scan> уже должен быть знаком.

Для интерфейса ViewResolver мы продолжим использовать класс InternalResourceViewResolver в качестве реализации. Однако мы изменим суффикс на .jspx.

Реализация класса ContactController

После конфигурирования WebApplicationContext сервлета диспетчера займемся реализацией класса контроллера.

Класс ContactController приведен в листинге 17.11.

Листинг 17.11. Класс ContactController

```
package com.apress.prospring3.ch17.web.controller;

import java.util.List;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

import com.apress.prospring3.ch17.domain.Contact;
import com.apress.prospring3.ch17.service.ContactService;

@RequestMapping("/contacts")
@Controller
public class ContactController {

    final Logger logger = LoggerFactory.getLogger(ContactController.class);

    @Autowired
    private ContactService contactService;

    @RequestMapping(method = RequestMethod.GET)
    public String list(Model uiModel) {
        logger.info("Listing contacts");

        List<Contact> contacts = contactService.findAll();
        uiModel.addAttribute("contacts", contacts);

        logger.info("No. of contacts: " + contacts.size());

        return "contacts/list";
    }
}
```

Как показано в листинге 17.11, к классу `ContactController` применена аннотация `@Controller`, указывающая, что это контроллер Spring MVC.

Аннотация `@RequestMapping` на уровне класса задает корневой URL, который будет обрабатываться контроллером. В этом случае все URL с префиксом `/ch17/contacts` будут направляться данному контроллеру. К методу `list()` также применена аннотация `@RequestMapping`, но на этот раз метод отображается на HTTP-метод `GET`. Это значит, что с помощью `list()` будет обработан URL `/ch17/contacts` с HTTP-методом `GET`. Внутри метода `list()` извлекается список контактов, который сохраняется в интерфейсе `Model`, передаваемом методу Spring MVC. Наконец, возвращается логическое имя представления `contacts/list`. В конфигурации сервлета диспетчера в качестве распознавателя представлений указан `InternalResourceViewResolver`, а файл имеет префикс `/WEB-INF/views/` и суффикс `.jspx`. В результате Spring MVC выберет для представления файл `/WEB-INF/views/contacts/list.jspx`.

Реализация представления списка контактов

Следующий шаг состоит в реализации страницы представления для отображения контактной информации, которой соответствует файл `/src/main/webapp/WEB-INF/views/contacts/list.jspx`. Содержимое этой страницы показано в листинге 17.12.

Листинг 17.12. Представление списка контактов

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<div xmlns:jsp="http://java.sun.com/JSP/Page"
    xmlns:c="http://java.sun.com/jsp/jstl/core"
    xmlns:joda="http://www.joda.org/joda/time/tags"
    version="2.0">
   <jsp:directive.page contentType="text/html;charset=UTF-8"/>
   <jsp:output omit-xml-declaration="yes"/>

   <h1>Contact Listing</h1>

   <c:if test="${not empty contacts}">
      <table>
         <thead>
            <tr>
               <th>First Name</th>
               <th>Last Name</th>
               <th>Birth Date</th>
            </tr>
         </thead>
         <tbody>
            <c:forEach items="${contacts}" var="contact">
               <tr>
                  <td>${contact.firstName}</td>
                  <td>${contact.lastName}</td>
                  <td><joda:format value="${contact.birthDate}"
                                   pattern="yyyy-MM-dd"/></td>
               </tr>
            </c:forEach>
         </tbody>
      </table>
   </c:if>

</div>
```

Если вы занимались разработкой JSP ранее, то код в листинге 17.12 должен выглядеть знакомым. Но поскольку это страница JSPX, ее содержимое встроено в дескриптор `<div>`. Кроме того, используемые библиотеки дескрипторов объявлены в виде пространство имен XML.

Во-первых, дескриптор `<jsp:directive.page>` определяет атрибуты, которые применяются ко всей странице JSPX, тогда как дескриптор `<jsp:output>` управляет свойствами вывода JSPX-документа.

Во-вторых, дескриптор `<c:if>` обнаруживает, пуст ли атрибут модели `contacts`. Так как мы уже заполнили базу данных тестовой информацией о контактах, атрибут `contacts` должен содержать данные. В результате дескриптор `<c:forEach>` визуализирует контактную информацию из таблицы на странице. Обратите внимание, что дескриптор `<joda:format>` форматирует атрибут `birthDate`, имеющий тип `DateTime` из Joda-Time.

Тестирование представления списка контактов

Теперь все готово к тестированию представления списка контактов. Для начала проверим, что корнем контекста веб-приложения является `ch17`. Чтобы сделать это, щелкните на имени проекта правой кнопкой мыши и выберите в контекстном меню пункт

Properties (Свойства). В меню слева вберите элемент Web Project Settings (Параметры веб-проекта) и удостоверьтесь, что в поле Context root (Корень контекста) указано `ch17` (рис. 17.4).

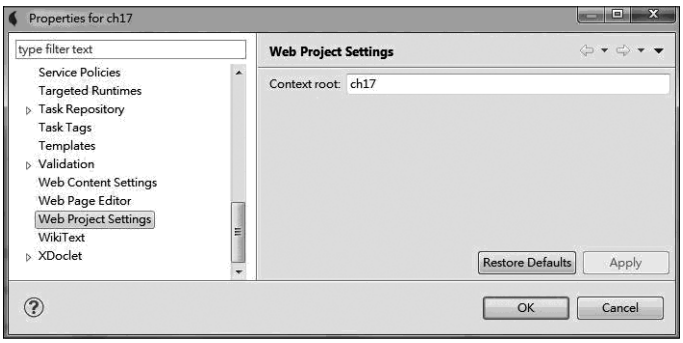


Рис. 17.4. Параметры веб-проекта в STS

Если вы используете параметры по умолчанию в рабочем пространстве STS, то проект должен быть автоматически собран и развернут на целевом сервере tc Server. Чтобы протестировать представление списка контактов, откройте веб-браузер и посетите URL `http://localhost:8080/ch17/contacts`. Вы должны увидеть страницу со списком контактов, показанную на рис. 17.5.

Итак, мы имеем первое работающее представление. В последующих разделах мы расширим приложение дополнительными представлениями и включим поддержку интернационализации, тем и т.п.

Обзор структуры проекта Spring MVC

Перед тем, как мы погрузимся в реализацию разнообразных аспектов веб-приложения, давайте посмотрим, как выглядит структура проекта в примере веб-приложения, которое разрабатывается в этой главе.

Как правило, для поддержки различных функциональных возможностей в веб-приложении требуется множество файлов. Например, имеется множество статических ресурсных файлов, таких как стилевые таблицы, файлы с кодом JavaScript, изображения, библиотеки компонентов и т.д. Вдобавок есть файлы, которые поддерживают представление пользовательского интерфейса на разных языках. И, естественно, существуют страницы представлений, которые будут проанализированы и визуализированы веб-контейнером, а также файлы компоновки и определений, применяемые шаблонной инфраструктурой (например, Apache Tiles) для обеспечения согласованного внешнего вида приложения.

Рекомендуемая практика предполагает хранение файлов, которые служат разным целям, в хорошо структурированной иерархии папок, что позволит получить четкую картину использования различных ресурсов в приложении и обеспечить простоту в сопровождении.

На рис. 17.6 показана структура папок в примере веб-приложения, разработанном в этой главе. Назначение каждой папки (внутри корневой папки `/src/main/webapp`) приведено в табл. 17.3. Обратите внимание, что представленная здесь структура не является обязательной, но она часто используется в сообществе разработчиков веб-приложений.

Рис. 17.5. Представление списка контактов

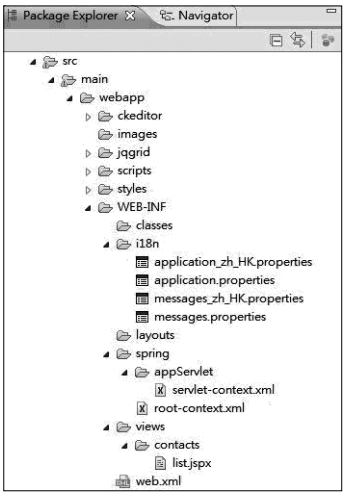


Рис. 17.6. Структура папок примера веб-проекта

В последующих разделах для поддержки реализации нам понадобятся различные файлы (файлы CSS, файлы JavaScript, файлы изображений и т.д.). Исходный код файлов CSS и JavaScript здесь приводиться не будет. Поэтому рекомендуется загрузить архив с исходным кодом для этой главы и распаковать его во временную папку для дальнейшего копирования в проект, когда возникнет необходимость.

Таблица 17.3. Описание папок в примере веб-проекта

Имя папки	Назначение	Примечание
`ckeditor`	CKEditor (`http://ckeditor.com`) — это JavaScript-библиотека компонентов, которая предоставляет редактор форматированного текста в рамках формы ввода. Мы будем использовать эту библиотеку для редактирования описания контакта	Скопируйте содержимое этой папки из исходного кода примеров в папку своего проекта
`images`	Хранит изображения для приложения	
`jqgrid`	jqGrid (`www.trirand.com/blog`) — это библиотека компонентов, построенная поверх jQuery, которая предоставляет различные основанные на сетке компоненты для представления данных. Мы будем использовать эту библиотеку для реализации сетки, отображающей контакты, а также для поддержки разбиения на страницы в стиле Ajax	Скопируйте содержимое этой папки из исходного кода примеров в папку своего проекта
`scripts`	Это папка для всех общих файлов JavaScript. В примерах этой главы будут использоваться JavaScript-библиотеки jQuery (`http://jquery.org`) и jQuery UI (`http://jqueryui.com`) для реализации обогащенного пользовательского интерфейса. Соответствующие сценарии будут помещены в этой папку. Сюда также попадут и самостоятельно разработанные JavaScript-библиотеки	Скопируйте содержимое этой папки из исходного кода примеров в папку своего проекта

Окончание табл. 17.3

Имя папки	Назначение	Примечание
styles	Хранит файлы стилевых таблиц и связанные со стилями файлы изображений	Скопируйте содержимое этой папки из исходного кода примеров в папку своего проекта
WEB-INF/i18n	Файлы для поддержки интернационализации. На рис. 17.6 файл application*.properties хранит текст, связанный с компоновкой (заголовки страниц, метки полей, заголовки меню и т.п.). Файл message*.properties хранит различные сообщения (например, сообщения об успехе и ошибках, сообщения проверки достоверности и т.д.). В рассматриваемом примере поддерживаются английский (США) и китайский (Гонконг) языки	Скопируйте содержимое этой папки из исходного кода примеров в папку своего проекта
WEB-INF/layouts	Эта папка хранит представления и определения компоновки. Файлы в ней будут использоваться шаблонной инфраструктурой Apache Tiles (http://tiles.apache.org)	
WEB-INF/spring	Эта папка хранит конфигурации WebApplicationContext для Spring MVC. Здесь находятся конфигурации контекстов и корневого уровня, и уровня сервлетов диспетчеров	
WEB-INF/views	Эта папка хранит представления (в нашем случае файлы JSPX), которые будут использоваться приложением	

Интернационализация

При разработке веб-приложений рекомендуется включать интернационализацию на самых ранних этапах. Основная работа связана с вынесением текста пользовательского интерфейса и сообщений в файлы свойств.

Даже если требования интернационализации в настоящее время отсутствуют, все равно полезно вынести строки, зависящие от языка, за пределы приложения, чтобы в будущем было проще поддерживать дополнительные языки.

В Spring MVC интернационализация включается очень просто. Для начала вынесите параметры, зависящие от языка, в различные файлы свойств внутри папки /WEB-INF/i18n, которая была описана в табл. 17.3. Поскольку мы планируем поддерживать английский (США) и китайский (Гонконг) языки, понадобятся четыре файла. Файлы application.properties и message.properties хранят параметры для локали по умолчанию, которой в данном случае является английский (США) язык, а файлы application_zh_HK.properties и message_zh_HK.properties содержат параметры для китайского (Гонконг) языка.

Конфигурирование интернационализации в сервлете диспетчера

Теперь, имея параметры языков, можно сконфигурировать WebApplicationContext сервлета диспетчера для поддержки интернационализации.

В листинге 17.13 показан модифицированный файл конфигурации с поддержкой интернационализации (servlet-context.xml).

Листинг 17.13. Модифицированная конфигурация контекста сервлета

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/mvc"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:beans="http://www.springframework.org/schema/beans"
   xmlns:p="http://www.springframework.org/schema/p"
   xmlns:context="http://www.springframework.org/schema/context"
   xsi:schemaLocation="http://www.springframework.org/schema/mvc
      http://www.springframework.org/schema/mvc/spring-mvc-3.1.xsd
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context-3.1.xsd">

   <annotation-driven />

   <resources location="/,
              classpath:/META-INF/web-resources/" mapping="/resources/**"/>

   <default-servlet-handler/>

   <beans:bean
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
      <beans:property name="prefix" value="/WEB-INF/views/" />
      <beans:property name="suffix" value=".jspx" />
   </beans:bean>

   <context:component-scan
      base-package="com.apress.prospring3.ch17.web.controller" />

   <interceptors>
      <beans:bean
        class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor"
        p:paramName="lang"/>
   </interceptors>

   <beans:bean
class="org.springframework.context.support.ReloadableResourceBundleMessageSource"
      id="messageSource"
      p:basenames="WEB-INF/i18n/messages,WEB-INF/i18n/application"
      p:fallbackToSystemLocale="false"/>

   <beans:bean class="org.springframework.web.servlet.i18n.CookieLocaleResolver"
      id="localeResolver" p:cookieName="locale"/>

</beans:beans>
```

В листинге 17.13 отличия от предыдущей версии выделены полужирным. Во-первых, добавлено пространство имен `p`, а определение ресурсов пересмотрено для отражения новой структуры папок, описанной в табл. 17.3. Дескриптор `<resources>` принадлежит пространству имен `mvc`; он был введен в Spring 3. Этот дескриптор определяет местоположения статических ресурсных файлов, которые позволяют Spring MVC обрабатывать файлы в этих папках более эффективно. Атрибут `location` внутри дескриптора определяет папки для статических ресурсов. Первый путь, /, указывает корневую папку для веб-приложения, которой является `/src/main/webapp`, а второй путь, `classpath:/META-INF/web-resources/` — ресурсные файлы для включенных библиотек. Это окажется удобным, если вы включите модуль Spring JavaScript, который имеет поддерживающие ресурсные файлы в папке `/META-INF/web-resources`. Атрибут `mapping` определяет URL для отображения на статические ресурсы; в качестве примера для URL

http://localhost:8080/ch17/resources/styles/standard.css модуль Spring MVC будет извлекать файл standard.css из папки /src/main/webapp/styles. Дескриптор <default-servlet-handler/> разрешает отображение сервлета диспетчера на URL корневого контекста веб-приложения, в то же время позволяя запросам статических ресурсов обрабатываться сервлетом, который установлен в контейнере как используемый по умолчанию.

Во-вторых, определен перехватчик Spring MVC с классом LocaleChangeInterceptor, который перехватывает все запросы к сервлету диспетчера. Перехватчик поддерживает переключение локалей с помощью конфигурируемого параметра запроса. В конфигурации перехватчика определен параметр URL с именем lang, предназначенный для изменения локали, которая применяется в приложении.

В-третьих, определен бин класса ReloadableResourceBundleMessageSource. Как объяснялось в главе 5, класс ReloadableResourceBundleMessageSource реализует интерфейс MessageSource, который загружает сообщения из определенных файлов (в данном случае это файлы messages*.properties и application*.properties в папке /WEB-INF/i18n) для поддержки интернационализации. Обратите внимание на свойство fallbackToSystemLocale. Это свойство указывает Spring MVC, нужно ли возвращаться к локали системы, в которой выполняется приложение, если специфический пакет ресурсов для клиентской локали не найден.

И, наконец, в-четвертых, определен бин класса CookieLocaleResolver. Этот класс поддерживает хранение и извлечение параметров локали из cookie-набора пользовательского браузера.

Модификация представления списка контактов для поддержки интернационализации

Теперь мы можем изменить страницу JSPX для отображения сообщений интернационализации. В листинге 17.14 показано модифицированное представление списка контактов (list.jspx).

Листинг 17.14. Модифицированное представление списка контактов для поддержки интернационализации

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<div xmlns:jsp="http://java.sun.com/JSP/Page"
   xmlns:c="http://java.sun.com/jsp/jstl/core"
   xmlns:joda="http://www.joda.org/joda/time/tags"
   xmlns:spring="http://www.springframework.org/tags"
   version="2.0">
   <jsp:directive.page contentType="text/html;charset=UTF-8"/>
   <jsp:output omit-xml-declaration="yes"/>

   <spring:message code="label_contact_list" var="labelContactList"/>
   <spring:message code="label_contact_first_name" var="labelContactFirstName"/>
   <spring:message code="label_contact_last_name" var="labelContactLastName"/>
   <spring:message code="label_contact_birth_date" var="labelContactBirthDate"/>

   <h1>${labelContactList}</h1>

   <c:if test="${not empty contacts}">
      <table>
         <thead>
            <tr>
               <th>${labelContactFirstName}</th>
               <th>${labelContactLastName}</th>
```

```
            <th>${labelContactBirthDate}</th>
         </tr>
      </thead>
      <tbody>
         <c:forEach items="${contacts}" var="contact">
            <tr>
               <td>${contact.firstName}</td>
               <td>${contact.lastName}</td>
               <td><joda:format value="${contact.birthDate}"
                                pattern="yyyy-MM-dd"/></td>
            </tr>
         </c:forEach>
      </tbody>
   </table>
</c:if>
</div>
```

В листинге 17.14 отличия с предыдущей версией выделены полужирным. Первым делом, к странице добавлено пространство имен `spring`. Затем с применением дескриптора `<spring:message>` в соответствующие переменные загружаются сообщения, требуемые представлением. Наконец, заголовок страницы и метки изменяются для использования сообщений интернационализации.

После сборки и повторного развертывания проекта откройте браузер и перейдите на URL `http://localhost:8080/ch17/contacts?lang=zh_HK`; вы увидите страницу для локали китайского языка (Гонконг), подобную показанной на рис. 17.7.

Рис. 17.7. Представление списка контактов на китайском языке (Гонконг)

Поскольку мы определили `localeResolver` в `WebApplicationContext` сервлета диспетчера, Spring MVC будет сохранять параметр локали в cookie-набор браузера (с именем `locale`) и по умолчанию cookie-набор будет предусмотрен для сеанса пользователя. Если вы хотите хранить cookie-набор для более длительного периода времени, в определении бина `localeResolver` (см. листинг 17.13) необходимо установить свойство `cookieMaxAge`, которое унаследовано от класса `org.springframework.web.util.CookieGenerator`.

Чтобы переключить локаль на английский язык (США), дополните URL в браузере строкой `?lang=en_US`, в результате чего страница переключится на английский язык (США). Хотя мы не предоставили файл свойств по имени `application_en_US.properties`, Spring MVC вернется к использованию файла `application.properties`, в котором хранятся свойства на языке, принятом по умолчанию, т.е. английском.

Оформление темами и применение шаблонов

Помимо интернационализации, веб-приложение требует соответствующего внешнего вида (например, бизнес-сайт должен выглядеть профессионально строго, в то время как социальный веб-сайт может иметь более свободный стиль), а также согласованной компоновки, чтобы пользователи не путались при работе с веб-приложением.

В веб-приложении стили должны быть вынесены во внешние стилевые таблицы, а не жестко закодированы на странице представления. Кроме того, имена стилей также должны быть согласованными, чтобы можно было подготовить различные “темы”, просто переключая файл стилевой таблицы. В Spring MVC предлагается широкая поддержка оформления темами для веб-приложений.

Вдобавок для обеспечения согласованной компоновки требуется шаблонная инфраструктура. В этом разделе для поддержки применения шаблонов к представлениям мы будем пользоваться Apache Tiles (`http://tiles.apache.org`) — популярной шаблонной инфраструктурой. Платформа Spring MVC тесно интегрирована с инфраструктурой Apache Tiles.

В последующих разделах мы покажем, как включить поддержку оформления темами в Spring MVC, а также как использовать Apache Tiles для определения компоновки страницы.

Поддержка оформления темами

В Spring MVC предлагается широкая поддержка для оформления темами, и ее включение в веб-приложениях производится очень просто. Например, в приложении контактов в этой главе мы хотим создать тему под названием `standard`. Для начала в папке `/src/main/webapp/WEB-INF/classes` создадим файл по имени `standard.properties` с содержимым, показанным в листинге 17.15.

Листинг 17.15. Файл `standard.properties`

```
styleSheet=resources/styles/standard.css
```

Приведенный в листинге 17.15 файл свойств содержит свойство по имени `styleSheet`, которое указывает стилевую таблицу для использования в стандартной (`standard`) теме. Этот файл свойств является пакетом ресурсов для темы, и к теме можно добавлять столько компонентов, сколько необходимо (скажем, местоположение изображения логотипа, местоположение фонового изображения и т.д.).

Следующий шаг состоит в конфигурировании `WebApplicationContext` сервлета диспетчера для поддержки оформления темами, для чего понадобится модифицировать файл конфигурации `servlet-context.xml`. Первым делом, в определение `<interceptors>` мы должны добавить еще один бин перехватчика, как показано в листинге 17.16.

Листинг 17.16. Конфигурирование перехватчика темы

```
<interceptors>
  <beans:bean class="org.springframework.web.servlet.theme.ThemeChangeInterceptor"/>
  <beans:bean class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor"
              p:paramName="lang"/>
</interceptors>
```

Новый перехватчик в листинге 17.16 выделен полужирным (порядок следования перехватчиков неважен). Класс `ThemeChangeInterceptor` перехватывает каждый запрос для изменения темы.

Теперь нужно добавить в файл конфигурации `servlet-context.xml` определения бинов из листинга 17.17.

Листинг 17.17. Конфигурирование поддержки темы

```
<beans:bean class="org.springframework.ui.context.support.ResourceBundleThemeSource"
          id="themeSource"/>

<beans:bean class="org.springframework.web.servlet.theme.CookieThemeResolver"
      id="themeResolver" p:cookieName="theme" p:defaultThemeName="standard"/>
```

В листинге 17.17 определены два бина. Первый бин, реализованный с помощью класса ResourceBundleThemeSource, отвечает за загрузку пакета ресурсов (ResourceBundle) активной темы. Например, если активная тема называется standard, этот бин будет искать файл standard.properties для пакета ресурсов данной темы. Второй бин, реализованный классом CookieThemeResolver, распознает активную тему для пользователей. Свойство defaultThemeName определяет тему, используемую по умолчанию, которой является тема standard. Обратите внимание, что класс CookieThemeResolver применяет cookie-наборы для сохранения темы пользователю. Существует также класс SessionThemeResolver, который сохраняет атрибут темы в пользовательском сеансе.

Итак, тема standard сконфигурирована и готова для применения в наших представлениях. В листинге 17.18 показано модифицированное представление списка контактов (/WEB-INF/views/contacts/list.jspx) с поддержкой темы.

Листинг 17.18. Представление списка контактов с поддержкой темы

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
  <!-- Остальной код не показан -->
  <spring:message code="label_contact_birth_date" var="labelContactBirthDate"/>
  <head>
       <spring:theme code="styleSheet" var="app_css" />
       <spring:url value="/${app_css}" var="app_css_url" />
       <link rel="stylesheet" type="text/css" media="screen"
             href="${app_css_url}" />
  </head>
  <h1>${labelContactList}</h1>
  <!-- Остальной код не показан -->
</div>
```

Новый код в листинге 17.18 выделен полужирным. К представлению добавлен раздел <head>, а дескриптор <spring:theme> используется для извлечения из пакета ресурсов темы свойства styleSheet, которое представляет собой файл стилевой таблицы standard.css. Наконец, к представлению добавлена ссылка на эту стилевую таблицу.

После сборки и повторного развертывания приложения на сервере откройте браузер и посетите URL представления списка контактов снова (http://localhost:8080/ch17/contacts). Вы увидите, что стиль, определенный в файле standard.css, был применен (рис. 17.8).

Используя поддержку оформления темами в Spring MVC, можно легко добавлять новые темы либо изменять существующую тему внутри приложения.

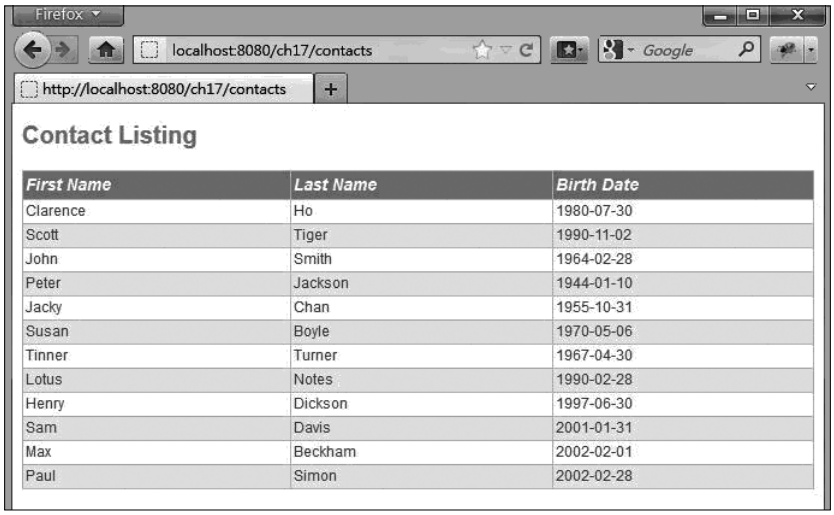


Рис. 17.8. Представление списка контактов с включенной поддержкой темы

Применение шаблонов представлений с помощью Apache Tiles

Из числа шаблонных инфраструктур, использующих технологию JSP, наиболее популярной является Apache Tiles (`http://tiles.apache.org`). Платформа Spring MVC тесно интегрирована с Apache Tiles.

Для работы с Apache Tiles понадобится добавить в проект обязательные зависимости, которые описаны в табл. 17.4.

Таблица 17.4. Зависимости Maven для Apache Tiles

Идентификатор группы	Идентификатор артефакта	Версия	Описание
`org.apache.tiles`	`tiles-core`	2.2.2	Библиотека ядра для Apache Tiles
`org.apache.tiles`	`tiles-jsp`	2.2.2	Поддержка Apache Tiles для файлов представлений JSP

В последующих разделах мы обсудим, как реализовать шаблоны страниц, включая дизайн компоновки страниц, определение и реализацию компонентов в компоновке.

Дизайн компоновки шаблона

Первым делом, мы должны определить, сколько шаблонов требуется в нашем приложении, и как должен выглядеть каждый такой шаблон.

В приложении контактов, рассматриваемом в этой главе, нам требуется только один шаблон. Как показано на рис. 17.9, шаблон довольно тривиален.

На рис. 17.9 видно, что шаблон требует следующих страничных компонентов.

- `/WEB-INF/views/header.jspx`. Эта страница предоставляет область заголовка.
- `/WEB-INF/views/menu.jspx`. Эта страница предоставляет область левого меню, а также форму входа, которая будет разработана позже в настоящей главе.
- `/WEB-INF/views/footer.jspx`. Эта страница предоставляет область нижнего колонтитула.

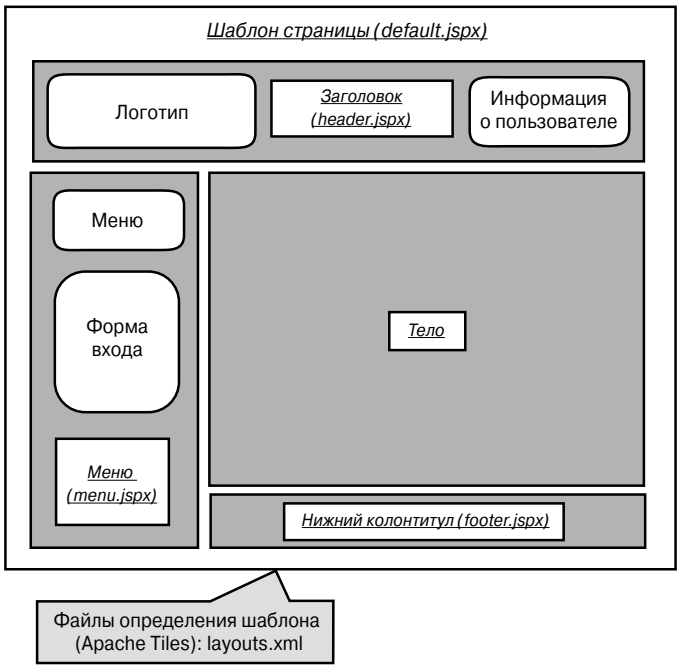


Рис. 17.9. Шаблон страницы с компонентами компоновки

Для определения шаблона мы будем использовать Apache Tiles, и нам нужно будет разработать файл шаблона страницы и файл определений компоновки, которые описаны ниже.

- `/WEB-INF/layouts/default.jspx`. Эта страница предоставляет полную компоновку для конкретного шаблона.
- `/WEB-INF/layouts/layouts.xml`. Эта страница содержит определения компоновки, требуемые Apache Tiles.

Реализация компонентов компоновки страницы

Имея определение компоновки, можно реализовать страничные компоненты. Сначала мы разработаем файл шаблона страницы и файлы определений компоновки, требуемые Apache Tiles. В листинге 17.19 приведено содержимое файла определений Apache Tiles (`src/main/webapp/WEB-INF/layouts.xml`).

Листинг 17.19. Файл определений Apache Tiles

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE tiles-definitions PUBLIC
      "-//Apache Software Foundation//DTD Tiles Configuration 2.1//EN"
      "http://tiles.apache.org/dtds/tiles-config_2_1.dtd">
<tiles-definitions>
  <definition name="default" template="/WEB-INF/layouts/default.jspx">
    <put-attribute name="header" value="/WEB-INF/views/header.jspx" />
    <put-attribute name="menu" value="/WEB-INF/views/menu.jspx" />
    <put-attribute name="footer" value="/WEB-INF/views/footer.jspx" />
  </definition>
</tiles-definitions>
```

Код в этом файле должен быть прост для понимания. Здесь имеется одно определение шаблона страницы по имени `default`. Код шаблона находится в файле `default.jspx`. Внутри страницы определены три компонента с именами `header`, `menu` и `footer`. Содержимое этих компонентов будет загружаться из файлов, указанных в атрибутах `value`. За детальным описанием определения Apache Tiles обращайтесь в документацию этого проекта (`http://tiles.apache.org/2.2/framework/tiles-core/dtddoc/index.html`).

В листинге 17.20 представлен шаблон страницы по умолчанию (`default.jspx`).

Листинг 17.20. Файл шаблона по умолчанию

```
<html xmlns:jsp="http://java.sun.com/JSP/Page"
     xmlns:c="http://java.sun.com/jsp/jstl/core"
     xmlns:fn="http://java.sun.com/jsp/jstl/functions"
     xmlns:tiles="http://tiles.apache.org/tags-tiles"
     xmlns:spring="http://www.springframework.org/tags">

  <jsp:output doctype-root-element="HTML" doctype-system="about:legacy-compat" />

  <jsp:directive.page contentType="text/html;charset=UTF-8" />
  <jsp:directive.page pageEncoding="UTF-8" />

  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=8" />

    <spring:theme code="styleSheet" var="app_css" />
    <spring:url value="/${app_css}" var="app_css_url" />
    <link rel="stylesheet" type="text/css" media="screen" href="${app_css_url}" />

    <!-- Извлечь пользовательскую локаль из контекста страницы
         (она была усановлена распознавателем локалей Spring MVC) -->
    <c:set var="userLocale">
       <c:set var="plocale">${pageContext.response.locale}</c:set>
       <c:out value="${fn:replace(plocale, '_', '-')}" default="en" />
    </c:set>

    <spring:message code="application_name"
                    var="app_name" htmlEscape="false"/>
    <title><spring:message code="welcome_h3" arguments="${app_name}" /></title>
  </head>

  <body class="tundra spring">
      <div id="headerWrapper">
         <tiles:insertAttribute name="header" ignore="true" />
      </div>
      <div id="wrapper">
         <tiles:insertAttribute name="menu" ignore="true" />
         <div id="main">
            <tiles:insertAttribute name="body"/>
            <tiles:insertAttribute name="footer" ignore="true"/>
         </div>
      </div>
  </body>
</html>
```

В основном это страница JSP. Основные моменты описаны ниже.

- В шаблон помещен дескриптор `<spring:theme>`, который поддерживает оформление темами на уровне шаблона.
- Дескриптор `<tiles:insertAttribute>` используется для указания страничных компонентов, которые должны загружаться из других файлов, как отражено в файле `layouts.xml`.

Давайте реализуем компоненты заголовка, меню и нижнего колонтитула. Код для этих компонентов приведен в листингах 17.21, 17.22 и 17.23.

Листинг 17.21. Компонент заголовка (`header.jspx`)

```
<div id="header" xmlns:jsp="http://java.sun.com/JSP/Page"
   xmlns:spring="http://www.springframework.org/tags"
   version="2.0">
   <jsp:directive.page contentType="text/html;charset=UTF-8" />
   <jsp:output omit-xml-declaration="yes" />
   <spring:message code="header_text" var="headerText"/>
   <div id="appname">
      <h1>${headerText}</h1>
   </div>
</div>
```

Листинг 17.22. Компонент меню (`menu.jspx`)

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<div id="menu" xmlns:jsp="http://java.sun.com/JSP/Page"
   xmlns:c="http://java.sun.com/jsp/jstl/core"
   xmlns:spring="http://www.springframework.org/tags"
   version="2.0">
      <jsp:directive.page contentType="text/html;charset=UTF-8" />
      <jsp:output omit-xml-declaration="yes" />
      <spring:message code="menu_header_text" var="menuHeaderText"/>
      <spring:message code="menu_add_contact" var="menuAddContact"/>
      <spring:url value="/contacts?form" var="addContactUrl"/>
   <h3>${menuHeaderText}</h3>
   <a href="${addContactUrl}"><h3>${menuAddContact}</h3></a>
</div>
```

Листинг 17.23. Компонент нижнего колонтитула (`footer.jspx`)

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<div id="footer" xmlns:jsp="http://java.sun.com/JSP/Page"
   xmlns:spring="http://www.springframework.org/tags" version="2.0">
   <jsp:directive.page contentType="text/html;charset=UTF-8" />
   <jsp:output omit-xml-declaration="yes" />

   <spring:message code="home_text" var="homeText"/>
   <spring:message code="label_en_US" var="labelEnUs"/>
   <spring:message code="label_zh_HK" var="labelZhHk"/>
   <spring:url value="/contacts" var="homeUrl"/>

   <a href="${homeUrl}">${homeText}</a> |
   <a href="${homeUrl}?lang=en_US">${labelEnUs}</a> |
   <a href="${homeUrl}?lang=zh_HK">${labelZhHk}</a>

</div>
```

Теперь можно изменить представление списка контактов с целью удовлетворения нашему шаблону. В целом понадобится только удалить раздел <head>, т.к. он уже присутствует на шаблонной странице (default.jspx). Модифицированное представление списка контактов показано в листинге 17.24.

Листинг 17.24. Модифицированное представление списка контактов (/views/contacts/list.jspx)

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<div xmlns:jsp="http://java.sun.com/JSP/Page"
    xmlns:c="http://java.sun.com/jsp/jstl/core"
    xmlns:joda="http://www.joda.org/joda/time/tags"
    xmlns:spring="http://www.springframework.org/tags"
    version="2.0">
   <jsp:directive.page contentType="text/html;charset=UTF-8"/>
   <jsp:output omit-xml-declaration="yes"/>
   <spring:message code="label_contact_list" var="labelContactList"/>
   <spring:message code="label_contact_first_name"
                   var="labelContactFirstName"/>
   <spring:message code="label_contact_last_name"
                   var="labelContactLastName"/>
   <spring:message code="label_contact_birth_date"
                   var="labelContactBirthDate"/>
   <h1>${labelContactList}</h1>
   <c:if test="${not empty contacts}">
      <table>
         <thead>
            <tr>
               <th>${labelContactFirstName}</th>
               <th>${labelContactLastName}</th>
               <th>${labelContactBirthDate}</th>
            </tr>
         </thead>
         <tbody>
            <c:forEach items="${contacts}" var="contact">
               <tr>
                  <td>${contact.firstName}</td>
                  <td>${contact.lastName}</td>
                  <td><joda:format value="${contact.birthDate}"
                            pattern="yyyy-MM-dd"/></td>
               </tr>
            </c:forEach>
         </tbody>
      </table>
   </c:if>
</div>
```

Итак, шаблон, определение и компоненты готовы; следующий шаг заключается в конфигурировании Spring MVC для интеграции с Apache Tiles.

Конфигурирование Apache Tiles в Spring MVC

Конфигурирование поддержки Apache Tiles в Spring MVC осуществляется просто. В конфигурацию сервлета диспетчера (servlet-context.xml) потребуется внести модификацию, заменив класс InternalResourceViewResolver классом

UrlBasedViewResolver. Содержимое измененного файла конфигурации приведено в листинге 17.25.

Листинг 17.25. Конфигурирование поддержки Apache Tiles в Spring MVC

```
<?xml version="1.0" encoding="UTF-8"?>
   <!-- Остальной код не показан -->
   <!-- Удалите следующий бин -->
   <beans:bean
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
      <beans:property name="prefix" value="/WEB-INF/views/" />
      <beans:property name="suffix" value=".jspx" />
   </beans:bean>
   <!-- Остальной код не показан -->
   <!-- Добавьте следующий бин -->
   <!-- Конфигурация Apache Tiles -->
   <beans:bean class="org.springframework.web.servlet.view.UrlBasedViewResolver"
      id="tilesViewResolver">
      <beans:property name="viewClass"
             value="org.springframework.web.servlet.view.tiles2.TilesView"/>
   </beans:bean>
   <beans:bean class="org.springframework.web.servlet.view.tiles2.TilesConfigurer"
              id="tilesConfigurer">
      <beans:property name="definitions">
         <beans:list>
            <beans:value>/WEB-INF/layouts/layouts.xml</beans:value>
            <!-- Scan views directory for Tiles configurations -->
            <beans:value>/WEB-INF/views/**/views.xml</beans:value>
         </beans:list>
      </beans:property>
   </beans:bean>
</beans:beans>
```

В листинге 17.25 бин, который должен быть удален, выделен курсивом, а определения новых бинов — полужирным. Сначала удаляется исходный бин ViewResolver (класса InternalResourceViewResolver). Затем определяется бин ViewResolver класса UrlBasedViewResolver со свойством viewClass, установленным в класс TilesView, который обеспечивает поддержку Apache Tiles в Spring MVC. Наконец, определяется бин tilesConfigurer, который предоставляет конфигурации компоновки, требуемые Apache Tiles.

Финальный конфигурационный файл, который понадобится подготовить — это /WEB-INF/views/contacts/views.xml, где определяются представления для приложения контактов. Содержимое этого файла показано в листинге 17.26.

Листинг 17.26. Файл views.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE tiles-definitions PUBLIC "-//Apache Software Foundation//DTD Tiles
Configuration 2.1//EN" "http://tiles.apache.org/dtds/tiles-config_2_1.dtd">
<tiles-definitions>
   <definition extends="default" name="contacts/list">
      <put-attribute name="body"
         value="/WEB-INF/views/contacts/list.jspx" />
   </definition>
</tiles-definitions>
```

В листинге 17.26 логическое имя представления отображается на соответствующий атрибут `body` представления. Как и в классе `ContactController` из листинга 17.11, метод `list()` возвращает логическое имя представления `contacts/list`, так что Apache Tiles сможет отобразить имя представления на правильный шаблон и тело представления.

Теперь можно протестировать страницу. Удостоверьтесь, что проект заново собран и развернут на сервере. После загрузки представления списка контактов снова (`http://localhost:8080/ch17/contacts`) будет отображено представление, основанное на шаблоне, подобное показанному на рис. 17.10.

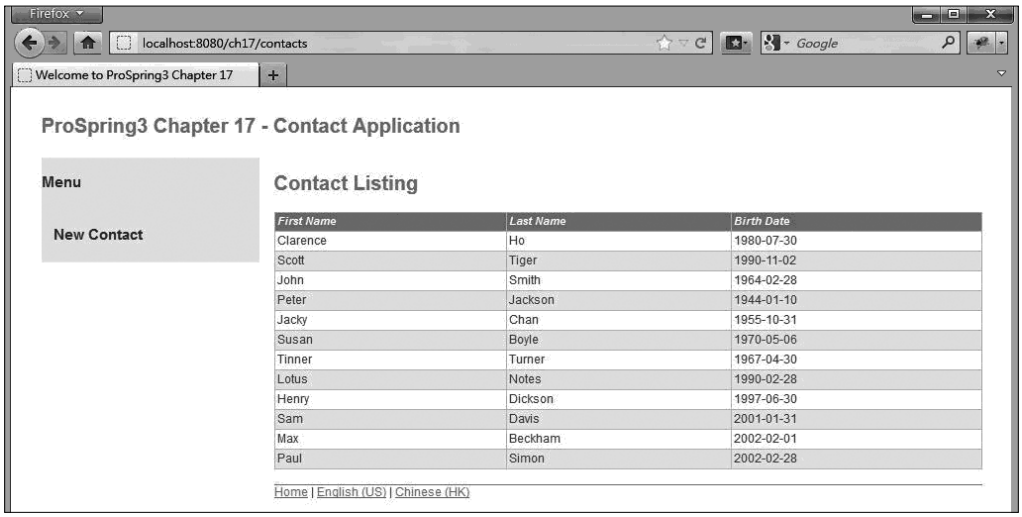


Рис. 17.10. Представление списка контактов, использующее Apache Tiles

Реализация представлений для информации о контактах

Продолжим реализацию представлений, которые позволят пользователям просматривать детальную информацию о контакте, создавать новый контакт и обновлять сведения о существующем контакте.

В последующих разделах мы обсудим отображение URL на различные представления, а также реализацию этих представлений. Мы также покажем, как включить поддержку проверки достоверности JSR-303 в Spring MVC для представления редактирования.

Отображение URL на представления

Первым делом, мы должны спроектировать, каким образом различные URL будут отображаться на соответствующие представления. Одним из рекомендуемых подходов в Spring MVC является применение для отображения представлений URL в стиле REST. В табл. 17.5 описано отображение URL на представления, а также указаны имена методов контроллера, которые будут обрабатывать соответствующие действия.

Таблица 17.5. Отображение URL на представления

URL	HTTP-метод	Метод контроллера	Описание
/contacts	GET	list()	Выводит список контактов
/contacts/{id}	GET	show()	Отображает информацию об одном контакте
/contacts/{id}?form	GET	updateForm()	Отображает форму редактирования для обновления существующего контакта
/contacts/{id}?form	POST	update()	Пользователи обновляют информацию о контакте и отправляют форму. Здесь будут обрабатываться данные
/contacts?form	GET	createForm()	Отображает форму редактирования для создания нового контакта
/contacts?form	POST	create()	Пользователи вводят информацию о контакте и отправляют форму. Здесь будут обрабатываться данные
/contacts/photo/{id}	GET	downloadPhoto()	Загружает фотографию для контакта

Реализация представления просмотра контакта

А теперь давайте реализуем представление для просмотра информации о контакте. Реализация этого представления состоит из трех шагов.

- Реализация метода контроллера.
- Реализация представления просмотра контакта (/views/contacts/show.jspx).
- Изменение файла определения представлений(/views/contacts/views.xml) для данного представления.

В листинге 17.27 приведена реализация метода show() класса ContactController, предназначенного для отображения информации о контакте.

Листинг 17.27. Метод show() класса ContactController

```
package com.apress.prospring3.ch17.web.controller;

// Операторы импорта опущены.
@RequestMapping("/contacts")
@Controller
public class ContactController {

   // Остальной код не показан.

   @RequestMapping(value = "/{id}", method = RequestMethod.GET)
   public String show(@PathVariable("id") Long id, Model uiModel) {
      Contact contact = contactService.findById(id);
      uiModel.addAttribute("contact", contact);
      return "contacts/show";
   }
}
```

В листинге 17.27 в методе `show()` аннотация `@RequestMapping`, примененная к методу, указывает, что этот метод предназначен для обработки URL `/contacts/{id}` с HTTP-методом `GET`. К аргументу `id` метода применена аннотация `@PathVariable`, которая заставляет Spring MVC извлекать `id` из URL в этот аргумент. Затем контакт извлекается и добавляется к модели, а возвращается логическое имя представления `contacts/show`. Следующий шаг заключается в реализации представления просмотра контакта (`/views/contacts/show.jspx`), которое показано в листинге 17.28.

Листинг 17.28. Представление просмотра контакта

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<div xmlns:jsp="http://java.sun.com/JSP/Page"
   xmlns:c="http://java.sun.com/jsp/jstl/core"
   xmlns:spring="http://www.springframework.org/tags"
   xmlns:form="http://www.springframework.org/tags/form"
   xmlns:joda="http://www.joda.org/joda/time/tags" version="2.0">
  <jsp:directive.page contentType="text/html;charset=UTF-8"/>
  <jsp:output omit-xml-declaration="yes"/>

  <spring:message code="label_contact_info" var="labelContactInfo"/>
  <spring:message code="label_contact_first_name" var="labelContactFirstName"/>
  <spring:message code="label_contact_last_name" var="labelContactLastName"/>
  <spring:message code="label_contact_birth_date" var="labelContactBirthDate"/>
  <spring:message code="label_contact_description" var="labelContactDescription"/>
  <spring:message code="label_contact_update" var="labelContactUpdate"/>
  <spring:message code="date_format_pattern" var="dateFormatPattern"/>

  <spring:url value="/contacts" var="editContactUrl"/>

  <h1>${labelContactInfo}</h1>

  <div id="contactInfo">
       <c:if test="${not empty message}">
          <div id="message" class="${message.type}">${message.message}</div>
       </c:if>

       <table>
          <tr>
             <td>${labelContactFirstName}</td>
             <td>${contact.firstName}</td>
          </tr>
          <tr>
             <td>${labelContactLastName}</td>
             <td>${contact.lastName}</td>
          </tr>
          <tr>
             <td>${labelContactBirthDate}</td>
             <td><joda:format value="${contact.birthDate}"
                              pattern="${dateFormatPattern}"/></td>
          </tr>
          <tr>
             <td>${labelContactDescription}</td>
             <td>${contact.description}</td>
          </tr>
       </table>
       <a href="${editContactUrl}/${contact.id}?form">${labelContactUpdate}</a>
  </div>
</div>
```

Эта страница проста; она лишь отображает атрибут `contact` модели внутри страницы. Финальный шаг связан с модификацией файла определения представлений (`/views/contacts/views.xml`) для отображения логического имени представления `contacts/show`. Просто добавьте в этот файл фрагмент кода, приведенный в листинге 17.29, поместив его в дескриптор `<tiles-definitions>`.

Листинг 17.29. Отображение для представления просмотра контакта

```
<definition extends="default" name="contacts/show">
   <put-attribute name="body"
      value="/WEB-INF/views/contacts/show.jspx" />
</definition>
```

На этом представление просмотра контакта завершено. Теперь необходимо добавить ссылку на представление просмотра контакта (`/views/contacts/list.jspx`) для каждого контакта. Соответствующие изменения показаны в листинге 17.30.

Листинг 17.30. Добавление ссылки на представление просмотра контакта

```
<spring:url value="/contacts/" var="showContactUrl"/>
<!-- Остальной код не показан -->
   <td>
      <a href="${showContactUrl}/${contact.id}">${contact.firstName}</a>
   </td>
```

В листинге 17.30 с использованием дескриптора `<spring:url>` мы объявляем переменную URL и добавляем ссылку для атрибута `firstName`. Чтобы протестировать представление просмотра контакта, после повторной сборки и развертывания откройте представление списка контактов снова. Список теперь должен включать гиперссылки на представление просмотра контакта (рис. 17.11).

Щелчок на любой такой ссылке переместит на представление просмотра контакта. На рис. 17.12 показан экран после щелчка на первой ссылке.

Contact Listing

First Name	Last Name	Birth Date
Clarence	Ho	1980-07-30
Scott	Tiger	1990-11-02
John	Smith	1964-02-28
Peter	Jackson	1944-01-10
Jacky	Chan	1955-10-31
Susan	Boyle	1970-05-06
Tinner	Turner	1967-04-30
Lotus	Notes	1990-02-28
Henry	Dickson	1997-06-30
Sam	Davis	2001-01-31
Max	Beckham	2002-02-01
Paul	Simon	2002-02-28

Home | English (US) | Chinese (HK)

Рис. 17.11. Представление списка контактов со ссылками на представление просмотра контакта

Contact Information

First Name	Clarence
Last Name	Ho
Birth Date	1980-07-30
Description	

Edit Contact

Home | English (US) | Chinese (HK)

Рис. 17.12. Представление просмотра контакта

Реализация представления редактирования контакта

Давайте реализуем представление для редактирования контакта. Это то же самое представление, что и для просмотра, но сначала мы добавим методы `updateForm()` и `update()` в класс `ContactController`. В листинге 17.31 приведен фрагмент кода для этих двух методов.

Листинг 17.31. Фрагмент кода для обновления контакта

```
package com.apress.prospring3.ch17.web.controller;

// Операторы импорта опущены.
@RequestMapping("/contacts")
@Controller
public class ContactController {

 // Остальной код не показан.
   @Autowired
  MessageSource messageSource;

  @RequestMapping(value = "/{id}", params = "form", method = RequestMethod.POST)
  public String update(Contact contact, BindingResult bindingResult,
       Model uiModel, HttpServletRequest httpServletRequest,
       RedirectAttributes redirectAttributes, Locale locale) {
    logger.info("Updating contact");
    if (bindingResult.hasErrors()) {
      uiModel.addAttribute("message", new Message("error",
        messageSource.getMessage("contact_save_fail", new Object[]{}, locale)));
      uiModel.addAttribute("contact", contact);
      return "contacts/update";
    }
    uiModel.asMap().clear();
    redirectAttributes.addFlashAttribute("message", new Message("success",
      messageSource.getMessage("contact_save_success", new Object[]{}, locale)));
    contactService.save(contact);
    return "redirect:/contacts/"
        + UrlUtil.encodeUrlPathSegment(contact.getId().toString(),
        httpServletRequest);
  }

  @RequestMapping(value = "/{id}", params = "form", method = RequestMethod.GET)
  public String updateForm(@PathVariable("id") Long id, Model uiModel) {
    uiModel.addAttribute("contact", contactService.findById(id));
    return "contacts/update";
  }
}
```

Ниже перечислены основные моменты, связанные с листингом 17.31.

- Интерфейс `MessageSource` автосвязан с контроллером для извлечения сообщений с поддержкой интернационализации.
- В методе `updateForm()` контакт извлекается и сохраняется в модели, после чего возвращается логическое имя представления `contacts/update`, которое отобразит представление редактирования контакта.
- Метод `update()` будет запускаться, когда пользователь обновляет информацию о контакте и щелкает на кнопке Save (Сохранить). Этот метод требует некоторых пояснений. Сначала Spring MVC попытается связать отправленные данные с объ-

ектом предметной области Contact и автоматически выполнит преобразование типов и форматирование. В случае обнаружения ошибок привязки (например, дата рождения была введена в неверном формате) эти ошибки будут сохранены в интерфейсе BindingResult (из пакета org.springframework.validation), а сообщения об ошибках — в модели, повторно отображая представление редактирования. Если привязка прошла успешно, данные сохраняются, и возвращается логическое имя представления просмотра контакта с указанием redirect: в качестве префикса. Обратите внимание, что нам нужно отобразить сообщение после перенаправления, поэтому мы должны использовать метод RedirectAttributes.addFlashAttribute() (из интерфейса в пакете org.springframework.web.servlet.mvc.support) для отображения сообщения об успехе в представлении просмотра контакта. Flash-атрибуты в Spring MVC временно сохраняются перед перенаправлением (обычно в сеансе) и будут доступны запросу после перенаправления, а затем они немедленно удаляются.

- Класс Message — это специальный класс, который хранит сообщение, извлеченное из MessageSource, и тип сообщения (т.е. успех или ошибка) для отображения в области сообщений представления. Код класса Message приведен в листинге 17.32.
- Класс UrlUtil — это служебный класс, кодирующий URL для перенаправления. Код этого класса показан в листинге 17.33.

Листинг 17.32. Класс Message

```
package com.apress.prospring3.ch17.web.form;

public class Message {

   private String type;

   private String message;

   public Message() {
   }

   public Message(String type, String message) {
      this.type = type;
      this.message = message;
   }

   // Методы извлечения/установки не показаны.
}
```

Листинг 17.33. Класс UrlUtil

```
package com.apress.prospring3.ch17.web.util;

import java.io.UnsupportedEncodingException;

import javax.servlet.http.HttpServletRequest;

import org.springframework.web.util.UriUtils;
import org.springframework.web.util.WebUtils;

public class UrlUtil {
   public static String encodeUrlPathSegment(String pathSegment,
               HttpServletRequest httpServletRequest) {
      String enc = httpServletRequest.getCharacterEncoding();
      if (enc == null) {
         enc = WebUtils.DEFAULT_CHARACTER_ENCODING;
      }
```

```
        try {
            pathSegment = UriUtils.encodePathSegment(pathSegment, enc);
        }
        catch (UnsupportedEncodingException uee) {}
        return pathSegment;
    }
}
```

А теперь рассмотрим собственно представление редактирования контакта (`/views/contacts/edit.jspx`), которое будет использоваться как при обновлении существующего, так и при создании нового контакта. Код этого представления приведен в листинге 17.34.

Листинг 17.34. Представление редактирования контакта

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<div xmlns:jsp="http://java.sun.com/JSP/Page"
    xmlns:c="http://java.sun.com/jsp/jstl/core"
    xmlns:spring="http://www.springframework.org/tags"
    xmlns:form="http://www.springframework.org/tags/form"
    version="2.0">
   <jsp:directive.page contentType="text/html;charset=UTF-8"/>
   <jsp:output omit-xml-declaration="yes"/>
   <spring:message code="label_contact_new" var="labelContactNew"/>
   <spring:message code="label_contact_update" var="labelContactUpdate"/>
   <spring:message code="label_contact_first_name" var="labelContactFirstName"/>
   <spring:message code="label_contact_last_name" var="labelContactLastName"/>
   <spring:message code="label_contact_birth_date" var="labelContactBirthDate"/>
   <spring:message code="label_contact_description" var="labelContactDescription"/>
   <spring:message code="label_contact_photo" var="labelContactPhoto"/>
   <spring:eval expression="contact.id == null ? labelContactNew:labelContactUpdate"
      var="formTitle"/>

   <h1>${formTitle}</h1>

   <div id="contactUpdate">
   <form:form modelAttribute="contact" id="contactUpdateForm" method="post">

      <c:if test="${not empty message}">
         <div id="message" class="${message.type}">${message.message}</div>
      </c:if>

      <form:label path="firstName">
         ${labelContactFirstName}*
      </form:label>
      <form:input path="firstName" />
      <div>
         <form:errors path="firstName" cssClass="error" />
      </div>
      <p/>

      <form:label path="lastName">
         ${labelContactLastName}*
      </form:label>
      <form:input path="lastName" />
      <div>
         <form:errors path="lastName" cssClass="error" />
      </div>
      <p/>
```

```
      <form:label path="birthDate">
         ${labelContactBirthDate}
      </form:label>
      <form:input path="birthDate" id="birthDate"/>
      <div>
         <form:errors path="birthDate" cssClass="error" />
      </div>
      <p/>

      <form:label path="description">
         ${labelContactDescription}
      </form:label>
      <form:textarea cols="60" rows="8" path="description"
                     id="contactDescription"/>
      <div>
         <form:errors path="description" cssClass="error" />
      </div>
      <p/>

      <form:hidden path="version" />

      <button type="submit">Save</button>
      <button type="reset">Reset</button>

   </form:form>
   </div>
</div>
```

Ниже описаны основные аспекты, касающиеся листинга 17.34.

- Дескриптор `<spring:eval>` использует язык SpEL (Spring Expression Language — язык выражений Spring) для проверки, равен ли идентификатор контакта (`contact.id`) значению `null`. Если это так, значит, создается новый контакт, а если нет, то обновляется существующий. Будет отображен соответствующий заголовок формы.
- Внутри формы с помощью различных дескрипторов `<form>` из Spring MVC отображаются метка, поле ввода и сообщения об ошибке, если привязка завершилась сбоем при отправке формы.

Добавим отображение представления в файл определения представлений (`/views/contacts/views.xml`). Соответствующий фрагмент кода показан в листинге 17.35.

Листинг 17.35. Отображение для представления редактирования контакта

```
   <definition extends="default" name="contacts/update">
      <put-attribute name="body"
         value="/WEB-INF/views/contacts/edit.jspx" />
   </definition>
```

На этом представление редактирования контакта завершено. Повторно соберите и разверните проект. Щелкните на ссылке Edit Contact (Редактировать контакт), как показано на рис. 17.12. Отобразится представление редактирования контакта (рис. 17.13).

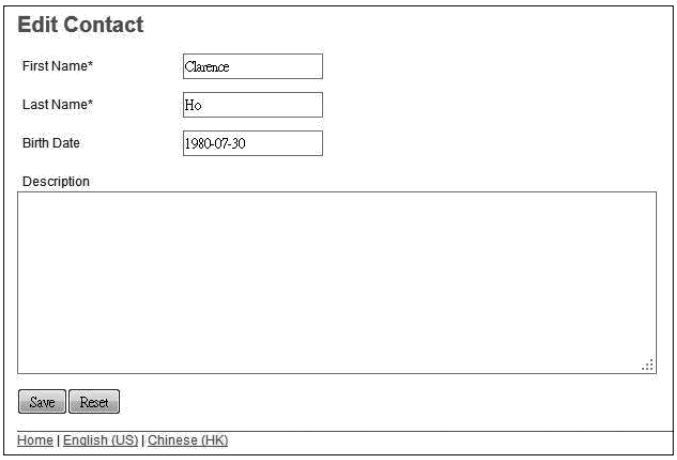


Рис. 17.13. Представление редактирования контакта

Обновите информацию о контакте и щелкните на кнопке Save (Сохранить). Если привязка прошла успешно, вы увидите сообщение Contact saved successfully (Контакт успешно сохранен) и отобразится представления просмотра контакта, как показано на рис. 17.14.

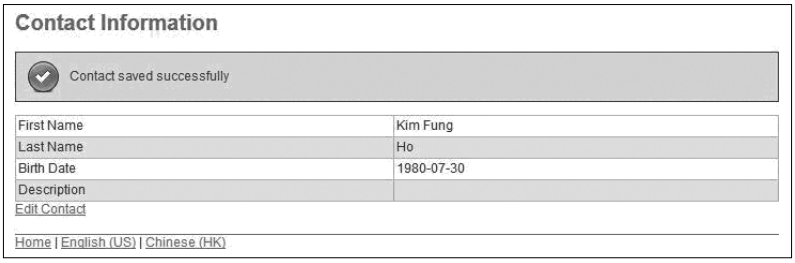


Рис. 17.14. Представление просмотра контакта (после успешного обновления контакта)

Реализация представления добавления контакта

Реализация представления добавления контакта очень похожа на реализацию представления для редактирования. Поскольку мы будем повторно использовать страницу `edit.jspx`, понадобится только добавить методы в класс `ContactController` и определение представления.

В листинге 17.36 приведен фрагмент кода для двух новых методов, обеспечивающих функциональность добавления контакта в классе `ContactController`.

Листинг 17.36. Методы для добавления контакта

```
package com.apress.prospring3.ch17.web.controller;

// Операторы импорта опущены.
@RequestMapping("/contacts")
@Controller
public class ContactController {

    // Остальной код не показан.
    @RequestMapping(params = "form", method = RequestMethod.POST)
```

```
   public String create(Contact contact, BindingResult bindingResult,
          Model uiModel, HttpServletRequest httpServletRequest,
          RedirectAttributes redirectAttributes, Locale locale) {
      logger.info("Creating contact");
      if (bindingResult.hasErrors()) {
         uiModel.addAttribute("message", new Message("error",
    messageSource.getMessage("contact_save_fail", new Object[]{}, locale)));
         uiModel.addAttribute("contact", contact);
         return "contacts/create";
      }
      uiModel.asMap().clear();
      redirectAttributes.addFlashAttribute("message", new Message("success",
 messageSource.getMessage("contact_save_success", new Object[]{}, locale)));
      logger.info("Contact id: " + contact.getId());
      contactService.save(contact);
      return "redirect:/contacts/" +
         UrlUtil.encodeUrlPathSegment(contact.getId().toString(),
                                      httpServletRequest);
   }
   @RequestMapping(params = "form", method = RequestMethod.GET)
   public String createForm(Model uiModel) {
      Contact contact = new Contact();
      uiModel.addAttribute("contact", contact);
      return "contacts/create";
   }
}
```

Далее добавим отображение представления в файл определения представлений (`/views/contacts/views.xml`). В листинге 17.37 приведен необходимый фрагмент кода.

Листинг 17.37. Отображение для представления добавления контакта

```
   <definition extends="default" name="contacts/create">
      <put-attribute name="body"
         value="/WEB-INF/views/contacts/edit.jspx" />
   </definition>
```

На этом представление добавления контакта завершено. После сборки и развертывания проекта щелкните на ссылке New Contact (Новый контакт) в области меню (см. рис. 17.10). Отобразится представление добавления контакта, показанное на рис. 17.15.

Включение проверки достоверности бинов JSR-303

Давайте сконфигурируем поддержку проверки достоверности бинов JSR-303 для действий создания и обновления контакта. Вначале примените ограничения проверки к объекту предметной области `Contact`. В рассматриваемом примере мы определяем ограничения только для атрибутов `firstName` и `lastName`. В листинге 17.38 представлен фрагмент кода с аннотациями, примененными к атрибутам `firstName` и `lastName`.

Рис. 17.15. Представление добавления контакта

Листинг 17.38. Применение ограничений к объекту предметной области `Contact`

```
package com.apress.prospring3.ch17.domain;

// Операторы импорта опущены.
@Entity
@Table(name = "contact")
public class Contact implements Serializable {
   // Остальной код не показан.
   @NotEmpty(message="{validation.firstname.NotEmpty.message}")
   @Size(min=3, max=60, message="{validation.firstname.Size.message}")
   @Column(name = "FIRST_NAME")
   public String getFirstName() {
      return firstName;
   }
   @NotEmpty(message="{validation.lastname.NotEmpty.message}")
   @Size(min=1, max=40, message="{validation.lastname.Size.message}")
   @Column(name = "LAST_NAME")
   public String getLastName() {
      return lastName;
   }
}
```

В листинге 17.38 примененные ограничения выделены полужирным. Обратите внимание, что для сообщения проверки достоверности используется код в фигурных скобках. Это приводит к извлечению таких сообщений из пакета ресурсов и обеспечивает поддержку интернационализации.

Чтобы включить проверку достоверности JSR-303 во время процесса привязки данных, нужно просто применить аннотацию `@Valid` к аргументу методов `create()` и `update()` в классе `ContactController`. Соответствующий фрагмент кода приведен в листинге 17.39.

Листинг 17.39. Включение проверки достоверности JSR-303 в контроллере

```
public String update(@Valid Contact contact, ...
public String create(@Valid Contact contact, ...
```

Также требуется, чтобы для сообщения проверки достоверности JSR-303 использовался тот же пакет ресурсов, что и в представлениях. Для этого необходимо сконфигурировать проверку в сервлете диспетчера (файл `servlet-context.xml`).

В листинге 17.40 показано, как должен быть изменен код.

Листинг 17.40. Конфигурирование поддержки JSR-303 в Spring MVC

```
<?xml version="1.0" encoding="UTF-8"?>
   <!-- Остальной код не показан -->
   <annotation-driven validator="validator"/>
   <!-- Остальной код не показан -->
<beans:bean id="validator" class="org.springframework.validation.
beanvalidation.LocalValidatorFactoryBean">
<beans:property name="validationMessageSource" ref="messageSource"/>
</beans:bean>
</beans:beans>
```

Изменения выделены полужирным. Вначале для поддержки JSR-303 определяется бин `validator` с классом `LocalValidatorFactoryBean`. Обратите внимание на установку свойства `validationMessageSource` для ссылки на определенный бин `messageSource`, который обеспечивает поиск при проверке JSR-303 сообщений по кодам из бина `messageSource`. Затем в дескрипторе `<annotation-driven>` атрибут `validator` явно ссылается на ранее определенный бин `validator`.

Теперь можно протестировать проверку достоверности. В представлении добавления контакта щелкните на кнопке Save (Сохранить). Отобразятся сообщения о возникших ошибках при проверке достоверности, как показано на рис. 17.16.

Переключитесь на китайский язык (для Гонконга) и сделайте то же самое. На этот раз сообщения будут отображаться на китайском (рис. 17.17).

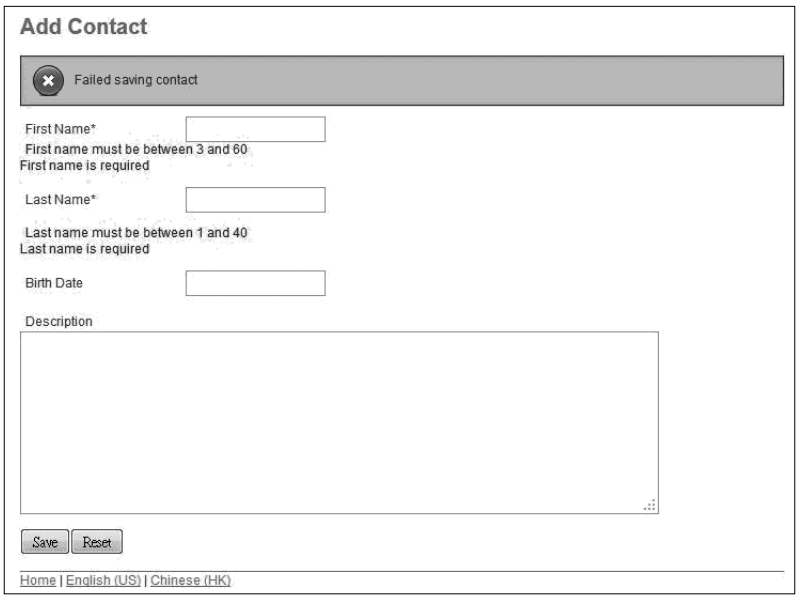


Рис. 17.16. Сообщения о возникших ошибках при проверке достоверности JSR-303

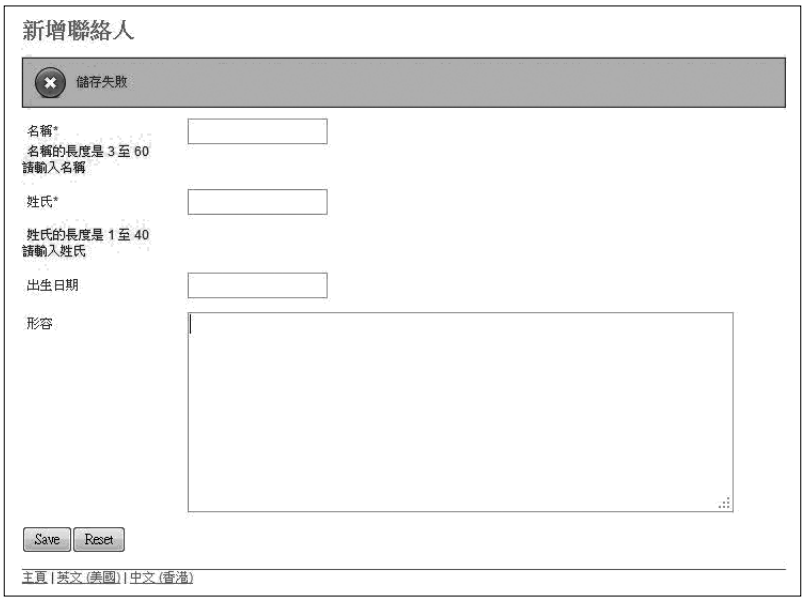


Рис. 17.17. Сообщения о возникших ошибках при проверке достоверности JSR-303 (интернационализация)

На этом представления в основном завершены, за исключением действия по удалению. Мы оставляем его реализацию в качестве упражнения для самостоятельного выполнения. А теперь приступим к расширению нашего приложения.

Использование jQuery и jQuery UI

Хотя представления в нашем приложении контактов функционируют вполне нормально, пользовательский интерфейс несколько сыроват. Например, для поля с датой рождения было бы неплохо предусмотреть средство выбора даты, а не вводить строку даты вручную.

Чтобы предоставить пользователям веб-приложения более развитый интерфейс, если только вы не имеете дело с технологиями RIA (Rich Internet Application — насыщенное Интернет-приложение), которые требуют наличия в клиентском веб-браузере специальных исполняющих сред (например, Adobe Flex требует Flash, JavaFX требует JRE, Microsoft Silverlight требует Silverlight и т.д.), придется реализовать нужные средства с помощью JavaScript.

Однако разработка пользовательских интерфейсов для веб-приложений на чистом JavaScript — задача не простая. Синтаксис языка JavaScript сильно отличается от синтаксиса Java, к тому же придется решать вопросы межбраузерной совместимости. Тем не менее, доступно множество JavaScript-библиотек с открытым кодом, которые существенно упрощают этот процесс; среди таких библиотек можно отметить jQuery, Dojo Toolkit и т.п.

В последующих разделах мы обсудим применение jQuery и jQuery UI для разработки более отзывчивых и интерактивных пользовательских интерфейсов. Мы также рассмотрим некоторые часто используемые подключаемые модули jQuery, предназначенные для специальных целей, таких как поддержка редактирования форматированного текста, и укажем ряд компонентов, основанных на сетке, применяемых для просмотра данных.

Введение в jQuery и jQuery UI

jQuery (http://jquery.org) — это одна из наиболее популярных JavaScript-библиотек, используемых при разработке пользовательских интерфейсов для веб-приложений. Библиотека jQuery предлагает широкую поддержку основных функциональных возможностей, включая надежный синтаксис "селектора" для выбора элементов DOM-модели в рамках документа, развитую модель событий, мощную поддержку Ajax и т.д.

Построенная поверх jQuery, библиотека jQuery UI (http://jqueryui.com) предоставляет широкий набор графических элементов (виджетов) и эффектов. Основные средства включают виджеты для часто применяемых компонентов пользовательского интерфейса (выбор даты, автозавершение и т.п.), перетаскивания, эффектов и анимации, оформления темами и т.д.

Существует также множество подключаемых модулей jQuery, разработанных сообществом jQuery для специальных целей, и два из них мы обсудим в этой главе.

В главе мы только слегка коснемся поверхности jQuery. За более подробными сведениями о jQuery рекомендуется обращаться к книгам *jQuery Recipes: A Problem-Solution Approach* (Apress, 2010 г.) и *jQuery in Action, Second Edition* (Manning, 2010 г.).

Активизация jQuery и jQuery UI в представлении

Чтобы можно было пользоваться компонентами jQuery и jQuery UI в представлении, необходимо включить требуемые файлы стилевых таблиц и JavaScript-кода.

Если вы проработали раздел "Обзор структуры проекта Spring MVC" ранее в этой главе, то обязательные файлы уже должны быть скопированы в проект. Ниже перечислены основные файлы, которые должны быть включены в представление.

- /src/main/webapp/scripts/jquery-1.7.1.js. Это JavaScript-библиотека ядра jQuery. В настоящей главе применяется ее версия 1.7.1. Обратите внимание, что это полная исходная версия. В производственной версии приложения должна использоваться уменьшенная версия библиотеки (т.е. jquery-1.7.1.min.js), которая оптимизирована и сжата с целью сокращения времени загрузки и улучшения производительности выполнения.
- /src/main/webapp/scripts/jquery-ui-1.8.16.custom.min.js. Это библиотека jQuery UI, упакованная со стилевой таблицей темы, которая может быть настроена и загружена со страницы Themeroller (http://jqueryui.com/themeroller). Здесь мы работаем с версией jQuery UI 1.8.16. Обратите внимание, что это уменьшенная версия JavaScript.
- /src/main/webapp/styles/custom-theme/jquery-ui-1.8.16.custom.css. Это стилевая таблица для специальной темы, которая будет использоваться jQuery UI для поддержки оформления темами.

Перечисленные выше файлы понадобится включить на странице шаблона (т.е. /layouts/default.jspx). В листинге 17.41 показан фрагмент кода, который необходимо добавить к этой странице.

Листинг 17.41. Включение jQuery в страницу шаблона

```
<html xmlns:jsp="http://java.sun.com/JSP/Page"
   <!-- Остальной код не показан -->
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
      <meta http-equiv="X-UA-Compatible" content="IE=8" />
```

```
   <spring:theme code="styleSheet" var="app_css" />
   <spring:url value="/${app_css}" var="app_css_url" />
   <link rel="stylesheet" type="text/css" media="screen" href="${app_css_url}" />
   <!-- jQuery и jQuery UI -->
   <spring:url value="/resources/scripts/jquery-1.7.1.js" var="jquery_url" />
   <spring:url value="/resources/scripts/jquery-ui-1.8.16.custom.min.js"
                var="jquery_ui_url" />
   <spring:url value="/resources/styles/custom-theme/jquery-ui-1.8.16.custom.css"
               var="jquery_ui_theme_css" />
   <link rel="stylesheet" type="text/css" media="screen"
        href="${jquery_ui_theme_css}" />
   <script src="${jquery_url}" type="text/javascript"><jsp:text/></script>
   <script src="${jquery_ui_url}" type="text/javascript"><jsp:text/></script>
   <!-- Остальной код не показан -->
</html>
```

В листинге 17.41 добавленный код выделен полужирным. Сначала с помощью дескриптора `<spring:url>` определяются URL для файлов и сохраняются в переменных. Затем в разделе `<head>` добавляется ссылка на файлы CSS и JavaScript. Обратите внимание на использование дескриптора `<jsp:text/>` внутри `<script>`. Это объясняется тем, что JSPX будет автоматически сворачивать дескрипторы, не имеющие тела. Таким образом, дескриптор `<script...></script>` в файле превратится в `<script.../>` в браузере, что приведет к неопределенному поведению страницы. Добавление `<jsp:text/>` гарантирует, что дескриптор `<script>` не будет визуализирован на странице, поскольку это позволит избежать неожиданных проблем.

Благодаря включению этих сценариев, мы можем добавить к представлению некоторые интересные вещи. Давайте в представлении редактирования контакта немного улучшим внешний вид кнопок и включим компонент выбора даты в поле ввода даты рождения. В листинге 17.42 показаны изменения, которые понадобится внести в представление (`/views/contacts/edit.jspx`).

Листинг 17.42. Декорирование кнопок и добавление компонента выбора даты в представление редактирования контакта

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
   <!-- Остальной код не показан -->
   <script type="text/javascript">
   $(function(){
      $('#birthDate').datepicker({
         dateFormat: 'yy-mm-dd',
         changeYear: true
      });
   });
   </script>
   <!-- Остальной код не показан -->
      <button type="submit"
class="ui-button ui-widget ui-state-default ui-corner-all ui-button-text-only">
         <span class="ui-button-text">Save</span>
      </button>
      <button type="reset"
class="ui-button ui-widget ui-state-default ui-corner-all ui-button-text-only">
         <span class="ui-button-text">Reset</span>
      </button>
   <!-- Остальной код не показан -->
</div>
```

В листинге 17.42 требуемые изменения выделены полужирным. Вначале с помощью синтаксиса `$(function(){})` библиотеке jQuery указывается на выполнение сценария, когда документ готов. Внутри этой функции поле ввода даты рождения (с идентификатором `birthDate`) декорируется с использованием функции `datepicker()` из jQuery UI. Затем к кнопкам добавляются классы стилей.

Перезагрузив страницу, вы увидите новый стиль кнопок, и при щелчке на поле ввода даты рождения отображается компонент выбора даты, как показано на рис. 17.18.

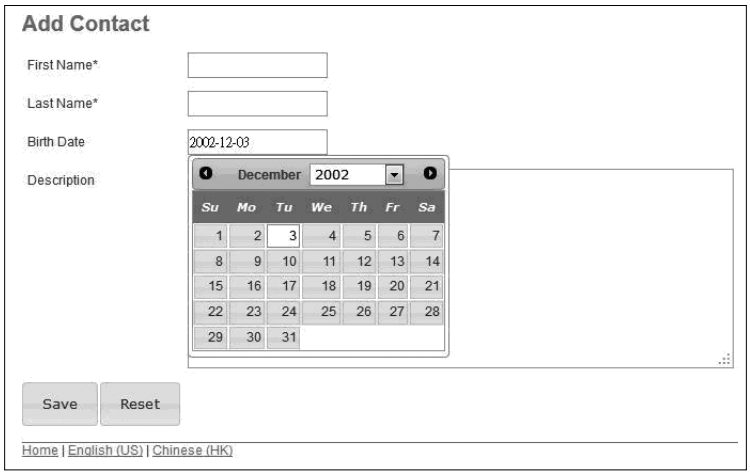


Рис. 17.18. Включение jQuery в представлении

Редактирование форматированного текста с помощью CKEditor

Для поля описания в информации о контакте мы используем дескриптор `<form:textarea>` из Spring MVC, поддерживающий многострочный ввод. Предположим, что мы хотим включить редактирование форматированного текста, которое является распространенным требованием при вводе длинных текстов, таких как пользовательские комментарии.

Для поддержки этой возможности мы будем применять библиотеку компонентов форматированного текста CKEditor (`http://ckeditor.com`), которая предлагает общий JavaScript-компонент форматированного текста и поддерживает интеграцию с jQuery UI. Файлы находятся в папке `/src/main/webapp/ckeditor` исходного кода примера.

Сначала необходимо включить обязательные JavaScript-файлы в страницу шаблона (`default.jspx`). В листинге 17.43 показан фрагмент кода, который должен быть добавлен к странице.

Листинг 17.43. Добавление CKEditor к странице шаблона

```
<html xmlns:jsp="http://java.sun.com/JSP/Page"
  <!-- Остальной код не показан -->
    <!-- jQuery и jQuery UI -->
    <spring:url value="/resources/scripts/jquery-1.7.1.js" var="jquery_url" />
    <spring:url value="/resources/scripts/jquery-ui-1.8.16.custom.min.js"
                var="jquery_ui_url" />
    <spring:url value="/resources/styles/custom-theme/jquery-ui-1.8.16.custom.css"
                var="jquery_ui_theme_css" />
```

```
   <link rel="stylesheet" type="text/css" media="screen"
         href="${jquery_ui_theme_css}" />
   <script src="${jquery_url}" type="text/javascript"><jsp:text/></script>
   <script src="${jquery_ui_url}" type="text/javascript"><jsp:text/></script>

   <!-- CKEditor -->
   <spring:url value="/resources/ckeditor/ckeditor.js" var="ckeditor_url" />
   <spring:url value="/resources/ckeditor/adapters/jquery.js"
                var="ckeditor_jquery_url" />
   <script type="text/javascript" src="${ckeditor_url}"><jsp:text/></script>
   <script type="text/javascript" src="${ckeditor_jquery_url}"><jsp:text/></script>

   <!-- Остальной код не показан -->
</html>
```

В листинге 17.43 добавленный код выделен полужирным. Мы включили два сценария — основной сценарий CKEditor и адаптер с jQuery.

Следующий шаг связан с добавлением CKEditor к представлению редактирования контакта. В листинге 17.44 приведены изменения, которые потребуется внести в код страницы (`edit.jspx`).

Листинг 17.44. Добавление CKEditor к представлению редактирования контакта

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>

   <!-- Остальной код не показан -->

   <script type="text/javascript">
   $(function(){
      $('#birthDate').datepicker({
         dateFormat: 'yy-mm-dd',
         changeYear: true
      });

      $("#contactDescription").ckeditor(
         {
            toolbar : 'Basic',
            uiColor : '#CCCCCC'
         }
      );
   });
   </script>

   <!-- Остальной код не показан -->

</div>
```

Изменения в листинге 17.44 выделены полужирным. Поле для ввода описания контакта декорировано с помощью CKEditor, которое выполняется, когда документ готов. Перезагрузив страницу добавления контакта, вы увидите, что в поле описания стала доступной поддержка редактирования форматированного текста, как показано на рис. 17.19.

За более подробными сведениями о применении и конфигурировании CKEditor обращайтесь в документацию проекта по адресу `http://docs.cksource.com/CKEditor_3.x/Developers_Guide`.

Рис. 17.19. Использование CKEditor для редактирования форматированного текста

Построение сетки данных, поддерживающей разбиение на страницы, с использованием jqGrid

Текущее представление списка контактов хорошо работает, только если в системе существует относительно немного контактов. С ростом количества записей до нескольких тысяч и более адекватная производительность становится проблемой.

Общее решение состоит в реализации компонента сетки данных с поддержкой разбиения на страницы, в котором пользователь просматривает только определенное количество записей; это уменьшает объем данных, передаваемых между браузером и веб-контейнером. В этом разделе мы продемонстрируем реализацию сетки данных с помощью jqGrid (`http://www.trirand.com/blog`) — популярного компонента сетки данных на основе JavaScript. Здесь используется версия 4.3.1 упомянутого компонента.

Кроме того, будет применяться встроенная в jqGrid поддержка разбиения на страницы Ajax, которая запускает `XMLHttpRequest` для каждой страницы и принимает формат JSON для данных на странице. Таким образом, мы должны добавить в проект зависимость от библиотеки JSON, как описано в табл. 17.6.

Таблица 17.6. Зависимости Maven для JSON

Идентификатор группы	Идентификатор артефакта	Версия	Описание
`org.codehaus.jackson`	`jackson-mapper-lgpl`	1.9.2	Процессор Jackson JSON для поддержки данных в формате JSON

В последующих разделах мы покажем, как реализовать поддержку разбиения на страницы на сторонах сервера и клиента. Сначала мы рассмотрим реализацию компонента jqGrid в представлении списка контактов. Затем мы обсудим реализацию разбиения на страницы на стороне сервера, используя поддержку такого разбиения в модуле Spring Data Commons.

Активизация jqGrid в представлении списка контактов

Чтобы сделать доступным компонент jqGrid в представлениях, понадобится включить требуемые файлы JavaScript-кода и стилевых таблиц в страницу шаблона (`default.jspx`). Необходимый фрагмент кода приведен в листинге 17.45.

Листинг 17.45. Добавление jqGrid в страницу шаблона

```
<html xmlns:jsp="http://java.sun.com/JSP/Page"
  <!-- Остальной код не показан -->
   <!-- CKEditor -->
   <spring:url value="/resources/ckeditor/ckeditor.js" var="ckeditor_url" />
   <spring:url value="/resources/ckeditor/adapters/jquery.js"
                var="ckeditor_jquery_url" />
   <script type="text/javascript" src="${ckeditor_url}"><jsp:text/></script>
   <script type="text/javascript" src="${ckeditor_jquery_url}"><jsp:text/>
   </script>
   <!-- jqGrid -->
   <spring:url value="/resources/jqgrid/css/ui.jqgrid.css" var="jqgrid_css" />
   <spring:url value="/resources/jqgrid/js/i18n/grid.locale-en.js"
                var="jqgrid_locale_url" />
   <spring:url value="/resources/jqgrid/js/jquery.jqGrid.min.js"
                var="jqgrid_url" />
   <link rel="stylesheet" type="text/css" media="screen" href="${jqgrid_css}" />
   <script type="text/javascript" src="${jqgrid_locale_url}"> <jsp:text/>
   </script>
   <script type="text/javascript" src="${jqgrid_url}"><jsp:text/></script>
  <!-- Остальной код не показан -->
</html>
```

В листинге 17.45 новый код выделен полужирным. Сначала загружается нужный CSS-файл, затем два JavaScript-файла. Первый из них — это сценарий локали (в данном случае используется английский язык), а второй — библиотека ядра jqGrid (`jquery.jqGrid.min.js`).

Следующий шаг связан с изменением представления списка контактов (`list.jspx`) с целью использования jqGrid. Модифицированный код страницы приведен в листинге 17.46.

Листинг 17.46. Страница списка контактов с jqGrid

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<div xmlns:jsp="http://java.sun.com/JSP/Page"
    xmlns:c="http://java.sun.com/jsp/jstl/core"
    xmlns:spring="http://www.springframework.org/tags"
    version="2.0">
   <jsp:directive.page contentType="text/html;charset=UTF-8"/>
   <jsp:output omit-xml-declaration="yes"/>
   <spring:message code="label_contact_list" var="labelContactList"/>
   <spring:message code="label_contact_first_name"
                   var="labelContactFirstName"/>
   <spring:message code="label_contact_last_name"
                   var="labelContactLastName"/>
   <spring:message code="label_contact_birth_date"
                   var="labelContactBirthDate"/>
```

```
    <spring:url value="/contacts/" var="showContactUrl"/>

    <script type="text/javascript">
    $(function(){
      $("#list").jqGrid({
        url:'${showContactUrl}/listgrid',
        datatype: 'json',
        mtype: 'GET',
        colNames:['${labelContactFirstName}', '${labelContactLastName}',
            '${labelContactBirthDate}'],
        colModel :[
          {name:'firstName', index:'firstName', width:150},
          {name:'lastName', index:'lastName', width:100},
          {name:'birthDateString', index:'birthDate', width:100}
        ],
        jsonReader : {
            root:"contactData",
            page: "currentPage",
            total: "totalPages",
            records: "totalRecords",
            repeatitems: false,
            id: "id"
        },
        pager: '#pager',
        rowNum:10,
        rowList:[10,20,30],
        sortname: 'firstName',
        sortorder: 'asc',
        viewrecords: true,
        gridview: true,
        height: 250,
        width: 500,
        caption: '${labelContactList}',
        onSelectRow: function(id){
            document.location.href ="${showContactUrl}/" + id;
        }
      });
    });
    </script>
    <c:if test="${not empty message}">
        <div id="message" class="${message.type}">${message.message}</div>
    </c:if>
    <h2>${labelContactList}</h2>

    <div>
    <table id="list"><tr><td/></tr></table>
    </div>
    <div id="pager"></div>
</div>
```

Как показано в листинге 17.46, для отображения данных сетки мы объявляем дескриптор `<table>` с идентификатором `list`. Под таблицей определен раздел `<div>` с идентификатором `pager`, который представляет собой часть jqGrid, отвечающую за разбиение на страницы.

В коде JavaScript, когда документ готов, мы указываем jqGrid на необходимость декорировать таблицу с идентификатором `list` как сетку, а также предоставляем конфигурационную информацию. Ниже отмечены некоторые важные аспекты сценариев.

- Атрибут `url` указывает ссылку для отправки `XMLHttpRequest`, который получает данные для текущей страницы.
- Атрибут `datatype` указывает формат данных, которым в этом случае является JSON. Компонент jqGrid также поддерживает формат XML.
- Атрибут `mtype` определяет используемый HTTP-метод, в этом случае `GET`.
- Атрибут `colNames` определяет заголовок столбца для данных, который должен отображаться в сетке, тогда как атрибут `colModel` устанавливает детали для каждой строки данных.
- Атрибут `jsonReader` определяет формат данных JSON, который будет возвращаться сервером.
- Атрибут `pager` включает поддержку разбиения на страницы.
- Атрибут `onSelectRow` определяет действие, которое должно выполняться при выборе строки. В этом случае мы направляем пользователя на представление, отображающее контакт с заданным идентификатором.

За подробным описанием конфигурирования и использования компонента jqGrid обращайтесь в документацию по проекту, которая доступна по адресу `www.trirand.com/jqgridwiki/doku.php?id=wiki:jqgriddocs`.

Включение разбиения на страницы на стороне сервера

На стороне сервера для реализации разбиения на страницы потребуется выполнить несколько шагов. Мы будем использовать поддержку разбиения на страницы из модуля Spring Data Commons. Для ее включения нужно только изменить интерфейс `ContactRepository`, чтобы он расширял интерфейс `PagingAndSortingRepository<T, ID extends Serializable>` вместо `CrudRepository<T,ID extends Serializable>`.

В листинге 17.47 показан измененный код.

Листинг 17.47. Модифицированный интерфейс `ContactRepository`

```
package com.apress.prospring3.ch17.repository;

import org.springframework.data.repository.PagingAndSortingRepository;
import com.apress.prospring3.ch17.domain.Contact;

public interface ContactRepository extends
   PagingAndSortingRepository<Contact, Long> {
}
```

Далее в интерфейс `ContactService` необходимо добавить новый метод для поддержки постраничного извлечения данных. Измененный код интерфейса приведен в листинге 17.48.

Листинг 17.48. Модифицированный интерфейс `ContactService`

```
package com.apress.prospring3.ch17.service;
import java.util.List;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import com.apress.prospring3.ch17.domain.Contact;
```

```
public interface ContactService {
   public List<Contact> findAll();
   public Contact findById(Long id);
   public Contact save(Contact contact);
   public Page<Contact> findAllByPage(Pageable pageable);
}
```

В листинге 17.48 видно, что был добавлен метод `findAllByPage()`, принимающий экземпляр интерфейса `Pageable` в качестве аргумента. В листинге 17.49 представлена реализация метода `findAllByPage()` в классе `ContactServiceImpl`. Этот метод возвращает экземпляр интерфейса `Page<T>` (который принадлежит Spring Data Commons и находится в пакете `org.springframework.data.domain`).

Листинг 17.49. Модифицированный класс `ContactServiceImpl`

```
package com.apress.prospring3.ch17.service.jpa;

// Операторы импорта опущены.

@Service("contactService")
@Repository
@Transactional
public class ContactServiceImpl implements ContactService {
   // Остальной код не показан.

   @Transactional(readOnly=true)
   public Page<Contact> findAllByPage(Pageable pageable) {
      return contactRepository.findAll(pageable);
   }
}
```

В листинге 17.49 реализация метода выделена полужирным. В этом методе мы просто вызываем метод `findAll(Pageable)`, который предоставляется интерфейсом `PagingAndSortingRepository<T,ID extends Serializable>`.

Следующий шаг связан с реализацией в классе `ContactController` метода, получающего от компонента jqGrid запрос Ajax для страничных данных. Реализация приведена в листинге 17.50.

Листинг 17.50. Модифицированный класс `ContactController`

```
package com.apress.prospring3.ch17.web.controller;

// Операторы импорта опущены.

@RequestMapping("/contacts")
@Controller
public class ContactController {

   // Остальной код не показан.

   @RequestMapping(value = "/listgrid", method = RequestMethod.GET,
      produces="application/json")
   @ResponseBody
   public ContactGrid listGrid(@RequestParam(value = "page",
                                             required = false) Integer page,
     @RequestParam(value = "rows", required = false) Integer rows,
     @RequestParam(value = "sidx", required = false) String sortBy,
     @RequestParam(value = "sord", required = false) String order) {
     logger.info("Listing contacts for grid with page: {}, rows: {}", page, rows);
```

```
    logger.info("Listing contacts for grid with sort: {}, order: {}", sortBy, order);

    // Обработать поле, по которому производится сортировка.
    Sort sort = null;
    String orderBy = sortBy;
    if (orderBy != null && orderBy.equals("birthDateString"))
       orderBy = "birthDate";
    if (orderBy != null && order != null) {
       if (order.equals("desc")) {
          sort = new Sort(Sort.Direction.DESC, orderBy);
       } else
          sort = new Sort(Sort.Direction.ASC, orderBy);
    }

    // Сконструировать страничный запрос для текущей страницы.
    // Примечание: нумерация страниц для Spring Data JPA начинается с 0,
    // тогда как в jqGrid - с 1.
    PageRequest pageRequest = null;
    if (sort != null) {
       pageRequest = new PageRequest(page - 1, rows, sort);
    } else {
       pageRequest = new PageRequest(page - 1, rows);
    }
    Page<Contact> contactPage = contactService.findAllByPage(pageRequest);

    // Сконструировать сетку, которая вернет данные в формате JSON.
    ContactGrid contactGrid = new ContactGrid();
    contactGrid.setCurrentPage(contactPage.getNumber() + 1);
    contactGrid.setTotalPages(contactPage.getTotalPages());
    contactGrid.setTotalRecords(contactPage.getTotalElements());
    contactGrid.setContactData(Lists.newArrayList(contactPage.iterator()));
    return contactGrid;
  }
}
```

В листинге 17.50 код нового метода `listGrid()` выделен полужирным. В сущности, этот метод обрабатывает запрос Ajax, читает параметры (номер страницы, количество записей на страницу, поле, по которому производится сортировка, порядок сортировки) из запроса (имена параметров в примере кода соответствуют принятым по умолчанию в jqGrid), конструирует экземпляр класса `PageRequest`, реализующий интерфейс `Pageable`, и затем вызывает метод `ContactService.findAllByPage()` для получения страничных данных. После этого создается экземпляр класса `ContactGrid`, который возвращает компонент jqGrid в формате JSON. Класс `ContactGrid` показан в листинге 17.51.

Листинг 17.51. Класс `ContactGrid`

```
package com.apress.prospring3.ch17.web.form;
import java.util.List;
import com.apress.prospring3.ch17.domain.Contact;
public class ContactGrid {
   private int totalPages;
   private int currentPage;
   private long totalRecords;
   private List<Contact> contactData;
   // Методы извлечения/установки не показаны.
}
```

Теперь все готово для тестирования нового представления списка контактов. Выполните повторную сборку и развертывание проекта и затем загрузите представление списка контактов. Вы должны увидеть представление, подобное показанному на рис. 17.20.

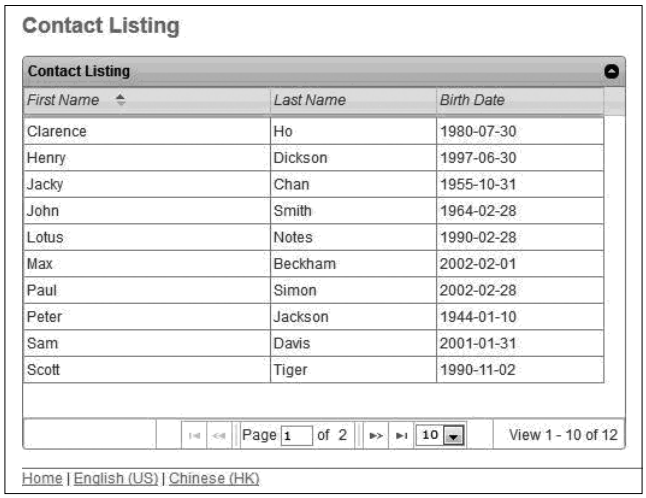


Рис. 17.20. Представление списка контактов с jgGrid

Можете поработать с сеткой, просматривая разные страницы, устанавливая разное количество записей на странице, изменяя порядок сортировки щелчками на заголовках столбцов и т.д. Интернационализация тоже поддерживается, поэтому при желании вы можете видеть сетку с метками на китайском языке.

Компонент jqGrid поддерживает фильтрацию данных. Например, данные можно фильтровать по имени, содержащем строку “clarence”, или по дате рождения, попадающей в заданный диапазон. Мы будем использовать фильтрацию в примере приложения, что будет обсуждаться в главе 21.

Обработка загрузки файлов

В рамках контактной информации имеется поле типа `BLOB`, предназначенное для хранения фотографии, которая может быть загружена на стороне клиента. В этом разделе мы покажем, как реализовать загрузку файлов в Spring MVC.

На протяжении длительного времени стандартная спецификация сервлетов не поддерживала загрузку файлов. В результате для этой цели модуль Spring MVC использовал другие библиотеки (из которых наиболее часто применялась библиотека Apache Commons FileUpload, доступная по адресу `http://commons.apache.org/fileupload`). Spring MVC имеет встроенную поддержку Commons FileUpload. Однако, начиная с версии Servlet 3.0, загрузка файлов стала встроенной функциональной возможностью веб-контейнера. Tomcat 7 поддерживает Servlet 3.0, и в Spring 3.1 также добавлена поддержка загрузки файлов Servlet 3.0.

В последующих разделах мы покажем, как реализовать загрузку файлов с использованием Spring MVC 3.1 и Servlet 3.0.

Для поддержки загрузки файлов понадобится добавить еще одну зависимость и также заменить зависимость версии Servlet 2.5 версией Servlet 3.0. Детали по удаляемой и добавляемым зависимостям приведены в табл. 17.7.

Таблица 17.7. Зависимости Maven для загрузки файлов

Идентификатор группы	Идентификатор артефакта	Версия	Описание
javax.servlet	servlet-api	2.5	Удалите эту зависимость
javax	javaee-web-api	6.0	Интерфейс JEE 6.0 Web Profile API, который содержит библиотеку для Servlet 3.0. Используйте для этой зависимости область видимости provided
commons-io	commons-io	2.1	Модуль Apache Commons IO предоставляет множество удобных функций для упрощения обработки ввода-вывода в Java

Конфигурирование поддержки загрузки файлов

В веб-контейнере, совместимом с Servlet 3.0, с Spring MVC конфигурирование поддержки загрузки файлов выполняется за два шага.

Во-первых, в дескрипторе веб-развертывания (web.xml) для определения сервлета диспетчера необходимо добавить раздел <multipart-config>. Соответствующий фрагмент кода представлен в листинге 17.52.

Листинг 17.52. Добавление поддержки загрузки файлов в web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"

   <!-- Остальной код не показан -->

   <!-- Обработка запросов приложения -->
   <servlet>
      <servlet-name>appServlet</servlet-name>
      <servlet-class>org.springframework.web.servlet.DispatcherServlet
      </servlet-class>
      <init-param>
         <param-name>contextConfigLocation</param-name>
         <param-value>/WEB-INF/spring/appServlet/servlet-context.xml
         </param-value>
      </init-param>
      <load-on-startup>1</load-on-startup>
      <multipart-config>
         <max-file-size>5000000</max-file-size>
      </multipart-config>
   </servlet>

   <!-- Остальной код не показан -->

</web-app>
```

Изменения выделены полужирным. В версии Servlet 3.0 сервлет, который поддерживает загрузку файлов, должен быть сконфигурирован с помощью дескриптора <multipart-config>. Дескриптор <max-file-size> управляет максимальным размером файла, разрешенным для загрузки, который составляет 5 Мбайт.

Во-вторых, понадобится сконфигурировать бин, реализующий интерфейс MultipartResolver, в WebApplicationContext сервлета диспетчера. В листинге 17.53 показано определения бина, которое нужно добавить в файл servlet-context.xml.

Листинг 17.53. Конфигурирование поддержки `MultipartResolver` в Spring MVC

```
   <beans:bean
class="org.springframework.web.multipart.support.StandardServletMultipartResolver"
       id="multipartResolver"/>
```

Обратите внимание на указание класса реализации `StandardServletMultipart Resolver`, который появился в Spring 3.1 для поддержки собственной загрузки файлов в контейнере Servlet 3.0.

Изменение представлений для поддержки загрузки файлов

Для добавления поддержки загрузки файлов мы должны модифицировать два представления. Первое из них — представление редактирования (`edit.jspx`) — будет поддерживать загрузку фотографии для контакта, а второе — представление просмотра (`show.jspx`) — будет выводить эту фотографию.

В листинге 17.54 показаны изменения, которые потребуется внести в представление редактирования.

Листинг 17.54. Представление редактирования контакта с поддержкой загрузки файлов

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
  <!-- Остальной код не показан -->
  <form:form modelAttribute="contact" id="contactUpdateForm" method="post"
    enctype="multipart/form-data">
    <!-- Остальной код не показан -->
    <form:label path="description">
         ${labelContactDescription}
    </form:label>
    <form:textarea cols="60" rows="8" path="description" id="contactDescription"/>
    <div>
       <form:errors path="description" cssClass="error" />
    </div>
    <p/>

    <label for="file">
       ${labelContactPhoto}
    </label>
    <input name="file" type="file"/>
    <p/>
  <!-- Остальной код не показан -->
</div>
```

В листинге 17.54 необходимые изменения выделены полужирным. В дескрипторе `<form:form>` понадобится включить поддержку загрузки файлов с множественным содержимым, указав атрибут `enctype`. Затем на форму необходимо добавить поле для загрузки файла.

Также потребуется модифицировать представление просмотра для вывода фотографии, связанной с контактом. Изменения, которые должны быть внесены в это представление (`show.jspx`), показаны в листинге 17.55.

Листинг 17.55. Представление просмотра контакта с поддержкой вывода фотографии

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
   <!-- Остальной код не показан -->
   <spring:message code="label_contact_photo" var="labelContactPhoto"/>
   <spring:url value="/contacts/photo" var="contactPhotoUrl"/>
   <!-- Остальной код не показан -->
         <tr>
            <td>${labelContactDescription}</td>
            <td>${contact.description}</td>
         </tr>
         <tr>
            <td>${labelContactPhoto}</td>
            <td><img src="${contactPhotoUrl}/${contact.id}"></img></td>
         </tr>
   <!-- Остальной код не показан -->
</div>
```

Внесенные изменения в листинге 17.55 выделены полужирным. В таблицу добавлена новая строка для отображения фотографии, в которой указывается URL загрузки фотографии, как было описано в табл. 17.5.

Изменение контроллера для поддержки загрузки файлов

Финальный шаг связан с модификацией контроллера. Потребуется внести два изменения. Первое изменение касается метода `create()`, который будет принимать загруженный файл в качестве обязательного параметра. Второе изменение — реализация нового метода для загрузки фотографии на основе указанного идентификатора контакта. Модифицированный класс `ContactController` приведен в листинге 17.56.

Листинг 17.56. Модифицированный класс контроллера с поддержкой загрузки файлов

```
package com.apress.prospring3.ch17.web.controller;

import javax.servlet.http.Part;

// Остальные операторы импорта опущены.
@RequestMapping("/contacts")
@Controller
public class ContactController {

   @RequestMapping(method = RequestMethod.POST)
   public String create(@Valid Contact contact, BindingResult bindingResult,
Model uiModel, HttpServletRequest httpServletRequest, RedirectAttributes
redirectAttributes, Locale locale, @RequestParam(value="file",
required=false) Part file) {
      logger.info("Creating contact");
      if (bindingResult.hasErrors()) {
         uiModel.addAttribute("message", new Message("error",
           messageSource.getMessage("contact_save_fail",
                                    new Object[]{}, locale)));
         uiModel.addAttribute("contact", contact);
         return "contacts/create";
      }
      uiModel.asMap().clear();
```

```
        redirectAttributes.addFlashAttribute("message", new Message("success",
            messageSource.getMessage("contact_save_success",
                                     new Object[]{}, locale)));

        logger.info("Contact id: " + contact.getId());

        // Обработка загруженного файла.
        if (file != null) {
            logger.info("File name: " + file.getName());
            logger.info("File size: " + file.getSize());
            logger.info("File content type: " + file.getContentType());
            byte[] fileContent = null;
            try {
                InputStream inputStream = file.getInputStream();
                if (inputStream == null) logger.info("File inputstream is null");
                    fileContent = IOUtils.toByteArray(inputStream);
                    contact.setPhoto(fileContent);
                } catch (IOException ex) {
                    logger.error("Error saving uploaded file");
                }
            contact.setPhoto(fileContent);
        }

        contactService.save(contact);
        return "redirect:/contacts/"
            + UrlUtil.encodeUrlPathSegment(contact.getId().toString(),
            httpServletRequest);
    }

    @RequestMapping(value = "/photo/{id}", method = RequestMethod.GET)
    @ResponseBody
    public byte[] downloadPhoto(@PathVariable("id") Long id) {

        Contact contact = contactService.findById(id);

        if (contact.getPhoto() != null) {
            logger.info("Downloading photo for id: {} with size: {}",
                        contact.getId(), contact.getPhoto().length);
        }

        return contact.getPhoto();
    }
}
```

В листинге 17.56 все изменения выделены полужирным. В метод `create()` добавлен аргумент для нового параметра запроса с типом интерфейса `javax.servlet.http.Part`, который Spring MVC предоставит на основе загруженного содержимого в запросе. Затем метод получит содержимое, сохраненное в свойстве `photo` объекта `Contact`.

Кроме того, добавлен новый метод `downloadPhoto()`, предназначенный для обработки загрузки файла. Этот метод просто извлекает поле `photo` из объекта `contact` и напрямую записывает его в поток ответа, что соответствует дескриптору `<img>` в представлении просмотра.

Чтобы протестировать функцию загрузки, откройте страницу и добавьте новый контакт с фотографией (рис. 17.21).

По завершении вы сможете увидеть эту фотографию в представлении просмотра контакта, как показано на рис. 17.22.

Рис. 17.21. Загрузка фотографии для контакта

Рис. 17.22. Представление просмотра контакта с фотографией

Еще потребуется модифицировать функцию редактирования для изменения фотографии, но мы здесь рассматривать это не будем. Можете самостоятельно просмотреть исходный код примера.

Защита веб-приложения с помощью Spring Security

Предположим, что теперь мы хотим защитить наше приложение контактов. Возможностью добавления новых контактов и обновления существующих контактов должны обладать только пользователи, которые вошли в приложение с допустимым идентификатором пользователя. Другие пользователи, называемые анонимными, могут только просматривать информацию о контактах.

Spring Security является наилучшим вариантом для защиты приложений, основанных на Spring. Несмотря на использование в основном на уровне презентаций, Spring Security может помочь защитить все уровни в приложении, включая уровень обслуживания. В последующих разделах мы продемонстрируем применение Spring Security для защиты приложения контактов.

В табл. 17.8 перечислены зависимости, требуемые для Spring Security.

Таблица 17.8. Зависимости Maven для Spring Security

Идентификатор группы	Идентификатор артефакта	Версия	Описание
org.springframework.security	spring-security-core	3.1.0.RELEASE	Модуль ядра Spring Security
org.springframework.security	spring-security-web	3.1.0.RELEASE	Веб-модуль Spring Security
org.springframework.security	spring-security-config	3.1.0.RELEASE	Модуль конфигурации Spring Security
org.springframework.security	spring-security-taglibs	3.1.0.RELEASE	Библиотека JSP-дескрипторов Spring Security

Конфигурирование Spring Security

Чтобы сконфигурировать Spring Security, сначала понадобится сконфигурировать фильтр в дескрипторе веб-развертывания (web.xml). В листинге 17.57 показан фрагмент кода, который должен быть добавлен в файл web.xml.

Листинг 17.57. Конфигурирование фильтра Spring Security

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
   <!-- Остальной код не показан -->

   <!-- Конфигурация Spring Security -->
   <filter>
      <filter-name>springSecurityFilterChain</filter-name>
      <filter-class>org.springframework.web.filter.DelegatingFilterProxy
      </filter-class>
   </filter>

   <filter-mapping>
      <filter-name>springSecurityFilterChain</filter-name>
      <url-pattern>/*</url-pattern>
   </filter-mapping>
   <!-- Остальной код не показан -->
</web-app>
```

В листинге 17.57 код фильтра для Spring Security выделен полужирным. Следующий шаг заключается в определении контекста Spring Security, который будет импортирован корневым файлом конфигурации WebApplicationContext. Содержимое конфигурационного файла /WEB-INF/spring/security-context.xml представлено в листинге 17.58.

Листинг 17.58. Конфигурация контекста Spring Security

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/security"
   xmlns:beans="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
    http://www.springframework.org/schema/security
    http://www.springframework.org/schema/security/spring-security-3.1.xsd">
   <http use-expressions="true">
      <intercept-url pattern='/*' access='permitAll' />
      <form-login login-page="/contacts"
                  authentication-failure-url="/security/loginfail"
         default-target-url="/contacts" />
      <logout logout-success-url="/contacts"/>
   </http>
   <authentication-manager>
      <authentication-provider>
         <user-service>
            <user name="user" password="user" authorities="ROLE_USER" />
         </user-service>
      </authentication-provider>
   </authentication-manager>
</beans:beans>
```

Поскольку наше требование простое, то и конфигурация не является сложной. Для начала дескриптор `<http>` определяет конфигурацию безопасности для HTTP-запросов. Атрибут `use-expressions` означает, что мы планируем использовать для выражений язык SpEL. Дескриптор `<intercept-url>` указывает, что входить в приложение разрешено всем пользователям. Мы покажем, как можно защитить функцию, скрывая опции редактирования в представлении с применением библиотеки дескрипторов Spring Security и защиты методов контроллера. В дескрипторе `<form-login>` определена поддержка формы входа. Как упоминалось при обсуждении компоновки, форма входа будет отображаться слева. Кроме того, будет предоставлена ссылка для выхода из приложения.

Дескриптор `<authentication-manager>` определяет механизм аутентификации. В примере конфигурации жестко закодирован единственный пользователь с назначенной ему ролью `ROLE_USER`. В корпоративной среде пользователи должны аутентифицироваться с помощью базы данных, LDAP или механизма SSO (single sign-on — однократная подписка). В листинге 17.59 приведено модифицированное содержимое файла `root-context.xml` для импорта файла конфигурации безопасности.

Листинг 17.59. Модифицированная конфигурация контекста Spring Security

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  <!-- Остальной код не показан -->
  <import resource="classpath:datasource-tx-jpa.xml" />
  <import resource="security-context.xml"/>
  <context:component-scan base-package="com.apress.prospring3.ch17.service.jpa"/>
</beans>
```

Оператор импорта выделен полужирным.

Добавление к приложению функций входа

Нам потребуется изменить два страничных компонента: заголовок (header.jspx) и меню (menu.jspx).

В листинге 17.60 показан модифицированный файл header.jspx, который теперь отображает информацию о пользователе, если он вошел в приложение.

Листинг 17.60. Отображение информации о вошедшем пользователе

```
<div id="header" xmlns:jsp="http://java.sun.com/JSP/Page"
      xmlns:spring="http://www.springframework.org/tags"
      xmlns:sec="http://www.springframework.org/security/tags"
      version="2.0">
      <jsp:directive.page contentType="text/html;charset=UTF-8" />
      <jsp:output omit-xml-declaration="yes" />
   <spring:message code="header_text" var="headerText"/>
   <spring:message code="label_logout" var="labelLogout"/>
   <spring:message code="label_welcome" var="labelWelcome"/>
   <spring:url var="logoutUrl" value="/j_spring_security_logout" />
   <div id="appname">
      <h1>${headerText}</h1>
   </div>
   <div id="userinfo">
      <sec:authorize access="isAuthenticated()">${labelWelcome}
         <sec:authentication property="principal.username" />
         <br/>
         <a href="${logoutUrl}">${labelLogout}</a>
      </sec:authorize>
   </div>
</div>
```

В листинге 17.60 полужирным выделены отличия от предыдущей версии. Вначале добавляется библиотека дескрипторов Spring Security с префиксом sec. Затем добавляется раздел с дескриптором <sec:authorize> для обнаружения, вошел ли пользователь в приложение. Если это так (т.е. выражение isAuthenticated() возвращает true), будет отображено имя пользователя и ссылка для выхода из приложения.

В листинге 17.61 показан модифицированный файл menu.jspx, в который добавлена форма входа; ссылка New Contact (Новый контакт) будет отображаться, только если пользователь вошел в приложение.

Листинг 17.61. Отображение формы входа

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<div id="menu" xmlns:jsp="http://java.sun.com/JSP/Page"
    xmlns:c="http://java.sun.com/jsp/jstl/core"
    xmlns:spring="http://www.springframework.org/tags"
    xmlns:sec="http://www.springframework.org/security/tags"
    version="2.0">
      <jsp:directive.page contentType="text/html;charset=UTF-8" />
      <jsp:output omit-xml-declaration="yes" />
      <spring:message code="menu_header_text" var="menuHeaderText"/>
      <spring:message code="menu_add_contact" var="menuAddContact"/>
      <spring:url value="/contacts?form" var="addContactUrl"/>
   <spring:message code="label_login" var="labelLogin"/>
   <spring:url var="loginUrl" value="/j_spring_security_check" />
   <h3>${menuHeaderText}</h3>
```

```
   <sec:authorize access="hasRole('ROLE_USER')">
      <a href="${addContactUrl}"><h3>${menuAddContact}</h3></a>
   </sec:authorize>
   <sec:authorize access="isAnonymous()">
   <div id="login">
     <form name="loginForm" action="${loginUrl}" method="post">
        <table>
           <caption align="left">Login:</caption>
           <tr>
              <td>User Name:</td>
              <td><input type="text" name="j_username"/></td>
           </tr>
           <tr>
              <td>Password:</td>
              <td><input type="password" name="j_password"/></td>
           </tr>
           <tr>
              <td colspan="2" align="center">
               <input name="submit" type="submit" value="Login"/></td>
           </tr>
        </table>
     </form>
   </div>
   </sec:authorize>
</div>
```

Отличия от предыдущей версии в листинге 17.61 выделены полужирным. Во-первых, пункт меню для добавления нового контакта будет отображаться, только если пользователь вошел и имеет назначенную ему роль `ROLE_USER` (как указано в первом дескрипторе `<sec:authorize>`). Во-вторых, если пользователь не вошел (второй дескриптор `<sec:authorize>`, когда выражение `isAnonymous()` возвращает `true`), будет отображена форма входа. Перезагрузив страницу, вы увидите, что она отображает форму входа, как показано на рис. 17.23. Обратите внимание на отсутствие ссылки New Contact.

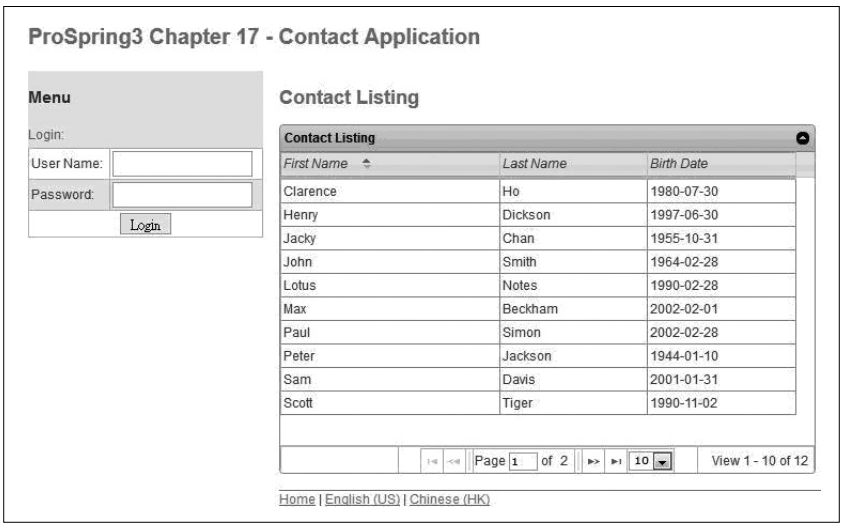


Рис. 17.23. Форма входа

Введите в полях User Name: (Имя пользователя:) и Password: (Пароль:) строку `user` и щелкните на кнопке Login (Вход). В области заголовка отобразится информация о пользователе (рис. 17.24). Кроме того, также появится ссылка New Contact.

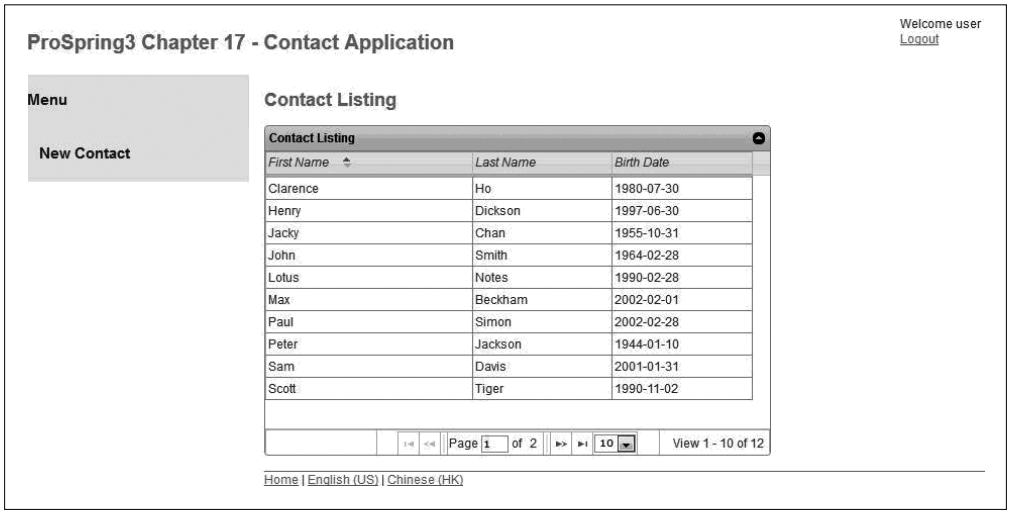


Рис. 17.24. Отображение информации о пользователе в области заголовка

Потребуется также изменить представление просмотра (`show.jspx`), чтобы отображать ссылку Edit Contact (Редактировать контакт) только для вошедших в приложение пользователей, но здесь мы это рассматривать не будем.

Как было определено в листинге 17.58, предоставление некорректной информации при входе обрабатывается с помощью URL `/security/loginfail`. Таким образом, мы должны реализовать контроллер для поддержки такого сценария. В листинге 17.62 показан класс контроллера `SecurityController`.

Листинг 17.62. Класс `SecurityController`

```
package com.apress.prospring3.ch17.web.controller;

// Операторы импорта опущены.

@RequestMapping("/security")
@Controller
public class SecurityController {

    final Logger logger = LoggerFactory.getLogger(SecurityController.class);
    @Autowired
    private MessageSource messageSource;

    @RequestMapping("/loginfail")
    public String loginFail(Model uiModel, Locale locale) {
        logger.info("Login failed detected");
        uiModel.addAttribute("message", new Message("error",
          messageSource.getMessage("message_login_fail", new Object[]{}, locale)));
        return "contacts/list";
    }
}
```

Этот класс контроллера будет обрабатывать все URL с префиксом `security`, а метод `loginFail()` — сценарий некорректного входа. В этом методе мы сохраняем сообщение о некорректном входе в модели и затем направляем пользователя на домашнюю страницу. Теперь перезагрузите страницу и введение неверную информацию о пользователе; домашняя страница отобразится повторно с сообщением о некорректном входе, как показано на рис. 17.25.

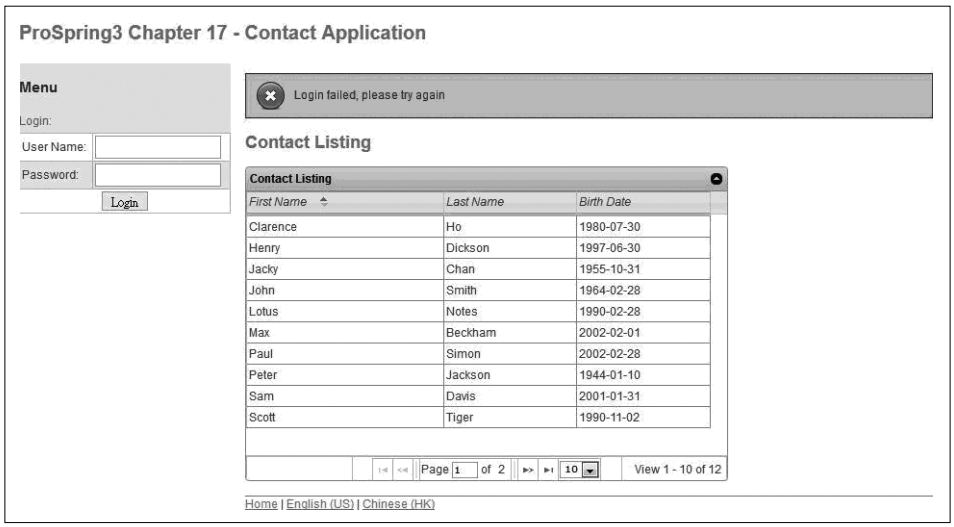


Рис. 17.25. Обработка некорректного входа

Использование аннотаций для защиты методов контроллера

Скрывания ссылки New Contact в меню недостаточно. Например, если ввести соответствующий URL (`http://localhost:8080/ch17/contacts?form`) в браузере напрямую, можно по-прежнему видеть страницу добавления контакта, несмотря на то, что вход в приложение не был совершен. Причина в том, что мы не защитили приложение на уровне URL. Один из способов защиты страницы предусматривает конфигурирование цепочки фильтров Spring Security (в файле `security-context.xml`) для перехвата URL для только аутентифицированных пользователей. Однако в этом случае остальные пользователи не смогут видеть представление списка контактов.

Альтернативный способ заключается в применении защиты на уровне методов контроллера, используя поддержку аннотаций Spring Security.

Чтобы включить защиту на уровне методов, мы должны изменить конфигурацию сервлета диспетчера (`servlet-context.xml`), как показано в листинге 17.63.

Листинг 17.63. Включение защиты на уровне методов

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/mvc"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:beans="http://www.springframework.org/schema/beans"
   xmlns:p="http://www.springframework.org/schema/p"
   xmlns:context="http://www.springframework.org/schema/context"
   xmlns:security="http://www.springframework.org/schema/security"
   xsi:schemaLocation="http://www.springframework.org/schema/mvc
```

```
      http://www.springframework.org/schema/mvc/spring-mvc-3.1.xsd
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context-3.1.xsd
      http://www.springframework.org/schema/security
      http://www.springframework.org/schema/security/spring-security-3.1.xsd">

   <!-- Контекст сервлета диспетчера: определение инфраструктуры обработки
       запросов этого сервлета -->

   <!-- Включение модели программирования @Controller в Spring MVC -->
   <annotation-driven validator="validator"/>

   <!-- Включение защиты на уровне методов -->
   <security:global-method-security pre-post-annotations="enabled"/>

   <!-- Остальной код не показан -->

</beans:beans>
```

В листинге 17.63 сначала было добавлено пространство имен `security`. Затем с помощью дескриптора `<security:global-method-security>` была разрешена защита на уровне методов Spring Security, а посредством атрибута `pre-post-annotations` включена поддержка аннотаций.

Теперь можно воспользоваться аннотацией `@PreAuthorize` для защиты необходимого метода контроллера. В листинге 17.64 приведен пример защиты метода `createForm()`.

Листинг 17.64. Применение аннотации Spring Security

```
@PreAuthorize("isAuthenticated()")
@RequestMapping(params = "form", method = RequestMethod.GET)
public String createForm(Model uiModel) {
   Contact contact = new Contact();
   uiModel.addAttribute("contact", contact);
   return "contacts/create";
}
```

Как показано в листинге 17.64, мы применяем аннотацию `@PreAuthorize` (из пакета `org.springframework.security.access.prepost`) для защиты метода `createForm()` с аргументом — выражением для требований безопасности.

После этого можно попробовать ввести URL страницы добавления нового контакта, и если вы не вошли в приложение, то Spring Security перенаправит на страницу входа, которой является представление списка контактов, как было сконфигурировано в файле `security-context.xml`.

Поддержка конфигурации на основе кода для Servlet 3

Еще одной новой функциональной возможностью Spring 3.1, связанной с веб-уровнем, является поддержка конфигурации на основе кода для Servlet 3, которая является альтернативой XML-конфигурации, требуемой в файле дескриптора веб-развертывания (`web.xml`). В этом разделе мы покажем, как использовать Java-код для загрузки `WebApplicationContext` сервлета диспетчера вместо его конфигурирования в файле `web.xml`.

Для применения конфигурации на основе кода необходимо только разработать класс, реализующий интерфейс org.springframework.web.WebApplicationInitializer. Интерфейс WebApplicationInitializer появился в версии Spring 3.1, и все классы, реализующие этот интерфейс, будут автоматически обнаруживаться классом org.springframework.web.SpringServletContainerInitializer (реализующим интерфейс javax.servlet.ServletContainerInitializer из Servlet 3), который автоматически загружается в любом контейнере Servlet 3.0.

Давайте рассмотрим простой пример использования конфигурации на основе кода для загрузки WebApplicationContext сервлета диспетчера вместо ее объявления в файле web.xml.

Для начала удалите из файла web.xml объявление сервлета и отображения для него, которые показаны в листинге 17.65.

Листинг 17.65. Удаление из файла web.xml объявления сервлета и отображения для него

```
<servlet>
   <servlet-name>appServlet</servlet-name>
   <servlet-class>org.springframework.web.servlet.DispatcherServlet
   </servlet-class>
   <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>/WEB-INF/spring/appServlet/servlet-context.xml
      </param-value>
   </init-param>
   <load-on-startup>1</load-on-startup>
   <multipart-config>
      <max-file-size>5000000</max-file-size>
   </multipart-config>
</servlet>

<servlet-mapping>
   <servlet-name>appServlet</servlet-name>
   <url-pattern>/</url-pattern>
</servlet-mapping>
```

После этого создайте класс, который реализует интерфейс WebApplicationInitializer. Этот класс называется MyWebAppInitializer, а его код приведен в листинге 17.66.

Листинг 17.66. Класс MyWebAppInitializer

```
package com.apress.prospring3.ch17.web.init;

import javax.servlet.MultipartConfigElement;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.ServletRegistration;

import org.springframework.web.WebApplicationInitializer;
import org.springframework.web.context.support.XmlWebApplicationContext;
import org.springframework.web.servlet.DispatcherServlet;

public class MyWebAppInitializer implements WebApplicationInitializer {

   @Override
   public void onStartup(ServletContext container) throws ServletException {

      XmlWebApplicationContext appContext = new XmlWebApplicationContext();
```

```
        appContext.setConfigLocation(
              "/WEB-INF/spring/appServlet/servlet-context.xml");

        ServletRegistration.Dynamic dispatcher =
          container.addServlet("appServlet", new DispatcherServlet(appContext));

        MultipartConfigElement multipartConfigElement =
           new MultipartConfigElement(null, 5000000, 5000000, 0);
        dispatcher.setMultipartConfig(multipartConfigElement);

        dispatcher.setLoadOnStartup(1);
        dispatcher.addMapping("/");
    }
}
```

Как показано в листинге 17.66, метод `WebApplicationInitializer.onStartup()` был переопределен за счет реализации логики для загрузки `WebApplicationContext` сервлета диспетчера. Вызывая метод `ServletContext.addServlet()`, мы можем добавить сервлет к веб-контейнеру. Этот метод вернет экземпляр интерфейса `javax.servlet.ServletRegistration.Dynamic`, и с помощью этого интерфейса можно сконфигурировать различные атрибуты сервлета, такие как отображение сервлета на URL, поддержка множественного содержимого при загрузке файлов и т.п. После повторной сборки и развертывания проекта `WebApplicationContext` сервлета диспетчера будет загружаться так же, как и в случае его конфигурирования в файле `web.xml`.

Используя такой подход в комбинации с конфигурированием Spring в Java-коде, можно получить чистую конфигурацию на основе Java-кода веб-приложения Spring, без необходимости в объявлении любой конфигурации Spring в `web.xml` или других XML-файлах конфигурации Spring. За более полным описанием всех возможностей обращайтесь в раздел, посвященный интерфейсу `WebApplicationInitializer`, онлайновой документации Spring Framework (`http://static.springsource.org/spring/docs/3.1.x/javadoc-api/org/springframework/web/WebApplicationInitializer.html`).

Spring MVC в примере приложения

Многие темы, рассмотренные в этой главе, будут применяться в примере приложения для реализации уровня презентаций, и в этом разделе описаны отношения с ними.

Реализация MVC для приложения SpringBlog

При разработке уровня презентаций в приложении SpringBlog модуль Spring MVC будет использоваться для реализации шаблона MVC, а в качестве технологии представлений будет выбрана JSPX. Ниже описано применение Spring MVC и связанных библиотек.

- *Интернационализация.* Для работы с различными языками в приложении SpringBlog будет использоваться встроенная в Spring MVC поддержка интернационализации. В приложении SpringBlog будет обеспечена поддержка английского (США) и китайского (Гонконг) языков.
- *Оформление темами.* Мы будем пользоваться поддержкой оформления темами в Spring MVC. Например, дескриптор `<spring:theme>` будет применяться в шаблоне страницы для загрузки активной темы.
- *Шаблоны страниц.* Для построения шаблонов страниц в приложении SpringBlog будет использоваться инфраструктура Apache Tiles. Шаблон страницы будет очень

похож на представленный в этой главе, со стандартными компонентами заголовка, меню, нижнего колонтитула, тела и т.д. Также будет применяться встроенная в Spring MVC поддержка использования Apache Tiles в качестве распознавателя представлений.

- *Проверка достоверности, преобразование типов и форматирование.* Вся проверка достоверности на уровне пользовательского интерфейса в приложении SpringBlog будет полагаться на стандарт JSR-303. Также будет применяться новая система преобразования типов и форматирования Spring 3. Все сообщения проверки достоверности JSR-303 будут поддерживать интернационализацию, как было показано в этой главе.

Насыщенный пользовательский интерфейс и Ajax

Для обеспечения многофункциональности приложения SpringBlog будут использоваться все JavaScript-библиотеки, рассмотренные в данной главе. Ниже отмечены основные аспекты.

- *Пользовательский интерфейс.* Для реализации пользовательского интерфейса приложения SpringBlog будут применяться библиотеки jQuery и jQuery UI. Например, при поиске записей блога по диапазону дат будет использоваться виджет выбора даты.
- *Редактирование форматированного текста.* При отправке в блог новых записей и комментариев пользователям будет доступно средство редактирования форматированного текста с применением CKEditor.
- *Сетка данных и разбиение на страницы.* Для просмотра записей блога в интерфейсе, основанном на сетке, будет использоваться библиотека компонентов jqGrid. Будет обеспечиваться разбиение на страницы, и пользователи смогут сортировать записи по различным атрибутам, а также изменять количество записей, отображаемых на странице. Вдобавок будет применяться поддержка фильтров jqGrid, чтобы предоставить пользователям возможность искать записи по теме, диапазону дат создания и т.д.

Поддержка безопасности

В приложении SpringBlog для защиты функций будет использоваться Spring Security. Ниже отмечены основные аспекты.

- *Пользователи и роли.* База данных SpringBlog будет иметь таблицы для хранения определенных пользователей и ролей. В целях аутентификации и авторизации для каждого пользователя будут также храниться выданные роли. Пользователи могут делать только то, что им разрешено посредством назначенных ролей. Например, изменять запись может только пользователь, который ее создал. Кроме того, просмотр хронологии аудита разрешен только пользователям с ролью администратора.
- *Пользовательский интерфейс.* Пользовательский интерфейс будет визуализироваться на основе ролей с помощью приемов, продемонстрированных в этой главе. В представлениях будет использоваться библиотека JSP-дескрипторов Spring Security для отображения информации о пользователе, а также разрешенных пользователям функций. Вдобавок на уровне методов контроллера, когда это необходимо, будет применяться защита с использованием поддержки аннотаций Spring Security.

Поддержка Servlet 3.0

В приложении SpringBlog будет также задействована поддержка MVC в Spring 3.1 для веб-контейнера, совместимого с Servlet 3.0. Основной момент описан ниже.

- *Загрузка файлов*. Одна из функциональных возможностей приложения SpringBlog заключается в том, что пользователи могут загружать вложения в свои записи или комментарии блога. Для реализации этой возможности будет применяться поддержка множественного содержимого Spring MVC 3.1.

За дополнительными сведениями обращайтесь в главу 21.

Резюме

В этой главе были раскрыты многие темы, связанные с разработкой пользовательских интерфейсов с помощью Spring MVC.

Сначала мы обсудили высокоуровневые концепции шаблона MVC. Затем мы рассмотрели архитектуру Spring MVC, включая иерархию `WebApplicationContext`, жизненный цикл обработки запросов и конфигурирование.

После этого мы разработали пример приложения контактов, используя Spring MVC и JSPX в качестве технологии представлений. Во время разработки были исследованы разнообразные области. Главные темы касались интернационализации, оформления темами и поддержки шаблонов с помощью Apache Tiles. Кроме того, для совершенствования пользовательского интерфейса мы применяли jQuery, jQuery UI и другие JavaScript-библиотеки. В число примеров входили компонент выбора даты, редактор форматированного текста, сетка данных с поддержкой разбиения на страницы и т.д. Также обсуждались вопросы защиты веб-приложений посредством Spring Security.

Кроме того, мы обсудили некоторые новые средства Spring 3.1, предназначенные для поддержки совместимых с Servlet 3.0 веб-контейнеров. Мы показали, как обрабатывать загрузку файлов с использованием Spring MVC 3.1 в среде Servlet 3.0. Также мы продемонстрировали применение конфигурации на основе кода вместо конфигурирования в файле `web.xml`.

В следующей главе мы рассмотрим дополнительные функциональные возможности, которые Spring предлагает для разработки веб-приложений.

ГЛАВА 18

Проект Spring Web Flow и JSF

В предыдущей главе мы обсуждали разработку веб-приложений с помощью Spring MVC, используя в качестве технологий представлений JSPX и JavaScript.

Для разработки веб-приложений доступно множество инструментов, в число которых входят собственные клиенты (Adobe Flex, JavaFX, Microsoft Silverlight и т.д.), платформы MVC (Spring MVC, JBoss Seam, Struts и т.п.) и технологии представлений (JSP, Velocity, JavaScript, JSF и т.д.).

Помимо Spring MVC, в Spring имеется проект под названием Spring Web Flow (`www.springsource.org/webflow`), который поддерживает разработку веб-приложений, основанных на потоках; этот проект также тесно интегрирован с такими технологиями представлений, как Dojo Toolkit и JSF. Кроме того, Spring Web Flow тесно интегрирован с модулем Spring MVC для поддержки интернационализации, оформления темами и проверки достоверности.

В этой главе мы покажем, каким образом проект Spring Web Flow может помочь в построении веб-приложения, основанного на потоках. Вдобавок мы продемонстрируем работу Spring Web Flow вместе с JSF (технология представлений на базе компонентов, являющаяся стандартной в рамках стека технологий JEE), используя модуль Spring Faces (из Spring Web Flow) и PrimeFaces (`http://primefaces.org`) — популярную библиотеку компонентов JSF. В частности, в главе будут рассматриваться следующие темы.

- *Проект Spring Web Flow*. Мы представим введение в проект Spring Web Flow и опишем основные его возможности, включая поддержку разработки потоковых приложений, более точную область действия Java-бинов и технологии представлений, с которыми этот проект интегрируется.
- *Технология JSF*. Мы дадим высокоуровневое описание JSF (точнее, JSF 2). Будут обсуждаться основные концепции этой технологии, в том числе компонентная модель, жизненный цикл обработки представления, поддержка применения шаблонов и т.п. Кроме того, мы также кратко рассмотрим некоторые часто используемые библиотеки компонентов JSF.
- *Spring Web Flow и PrimeFaces*. Мы покажем, как использовать Spring Web Flow и PrimeFaces для разработки веб-приложения, основанного на потоках.

Проект примера серверной части

Всегда полезнее построить веб-приложение с конкретной серверной частью, а не ограничиваться простыми примерами в стиле “Hello World”. В результате, мы подготовили

проект для примера серверной части, который понадобится только импортировать, и серверная часть будет готова. После этого мы сосредоточим внимание на разработке интерфейсной части. В последующих разделах мы кратко опишем серверную часть и покажем, как импортировать ее проект для разработки примеров интерфейсной части в этой главе.

Уровень обслуживания для примера серверной части

Серверная часть для примеров в этой главе представляет собой простое приложение контактов. Каждый контакт имеет базовую информацию, включающую имя, фамилию и дату рождения. Кроме того, с контактом также связано ноль или более хобби. Объектная модель предметной области показана на рис. 18.1.

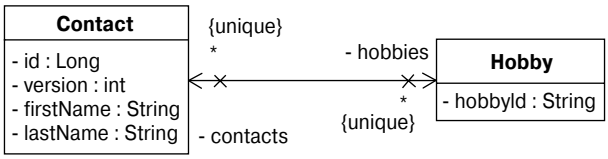


Рис. 18.1. Объектная модель предметной области для примера серверной части

Для примера серверной части мы будем использовать JPA 2 и Hibernate в качестве поставщика постоянства. Вдобавок будет применяться абстракция репозитория Spring Data JPA. Диаграмма классов для уровня обслуживания представлена на рис. 18.2.

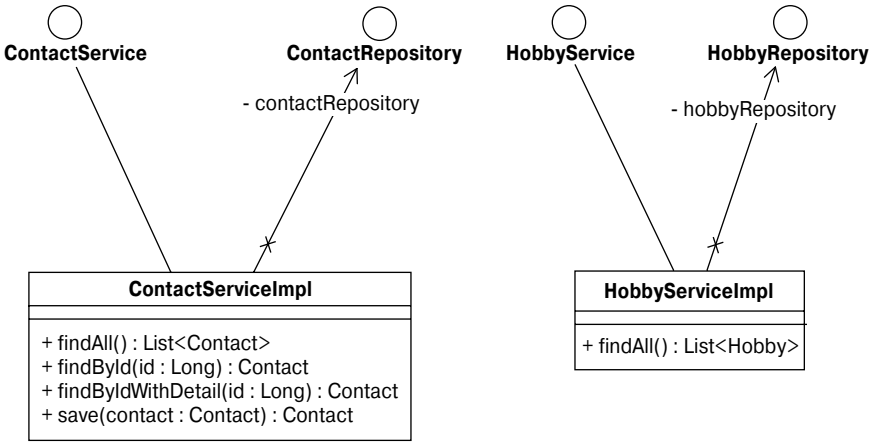


Рис. 18.2. Диаграмма классов для уровня обслуживания

Импорт проекта примера серверной части в STS

Поскольку основное внимание в этой главе будет уделяться разработке интерфейсной части, мы подготовили проект с реализованным уровнем обслуживания, так что вы можете просто импортировать его и затем продолжить заниматься реализацией интерфейсной части.

Извлеките файл `ch18-backend.zip` из кода примеров в рабочее пространство STS. После этого импортируйте проект в STS.

Чтобы импортировать этот проект, в среде STS выберите пункт меню File⇨Import (Файл⇨Импорт) и на экране Select (Выбор) укажите вариант Existing Projects into

Workspace (Существующие проекты в рабочем пространстве) в категории General (Общие), как показано на рис. 18.3; щелкните на кнопке Next (Далее).

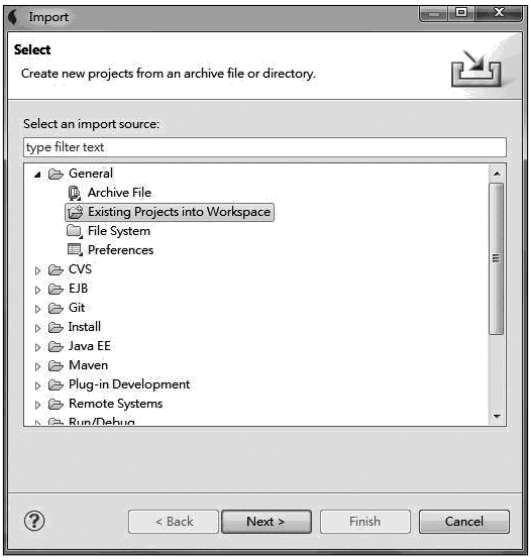


Рис. 18.3. Импорт существующего проекта в STS

После этого на экране Import Projects (Импорт проектов) выберите проект, извлеченный из кода примеров, и щелкните на кнопке Finish (Готово), как показано на рис. 18.4.

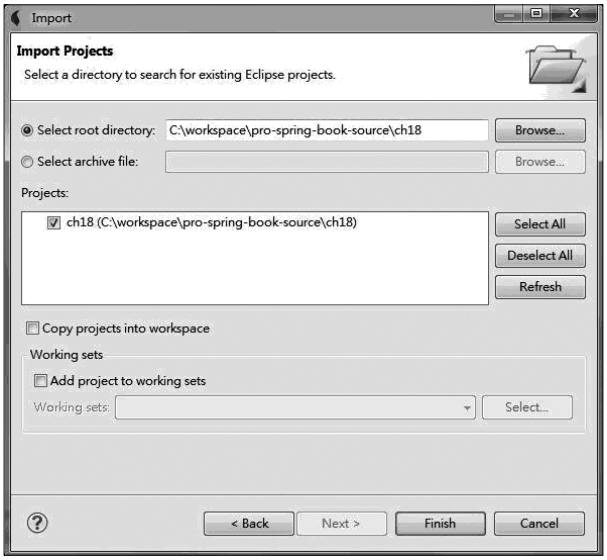


Рис. 18.4. Импорт проекта примера

На этом настройка проекта завершена и можно приступать к реализации интерфейсной части с использованием Spring Web Flow и PrimeFaces. Однако сначала давайте кратко рассмотрим высокоуровневые концепции Spring Web Flow и JSF.

Введение в Spring Web Flow

Построенный поверх инфраструктуры Spring MVC, проект Spring Web Flow предназначен для предоставления расширенной поддержки организации потоков внутри веб-приложения. Кроме того, в рамках этого проекта предлагается расширенная функциональность в различных областях, связанных с веб-приложениями.

В последующих разделах мы представим высокоуровневый обзор Spring Web Flow. Будут рассматриваться основные функциональные возможности, модули, интеграция с технологиями представлений, такими как JavaScript-библиотеки и JSF, и т.д.

Модули Spring Web Flow

В дистрибутив Spring Web Flow входит множество модулей. Например, в этой главе мы обсудим использование Spring Faces (модуль из Spring Web Flow) с библиотекой JSF-компонентов PrimeFaces для разработки потокового веб-приложения. Позже мы добавим к проекту зависимость от Spring Faces. На рис. 18.5 показана иерархия зависимостей после импорта модуля Spring Faces в проект STS.

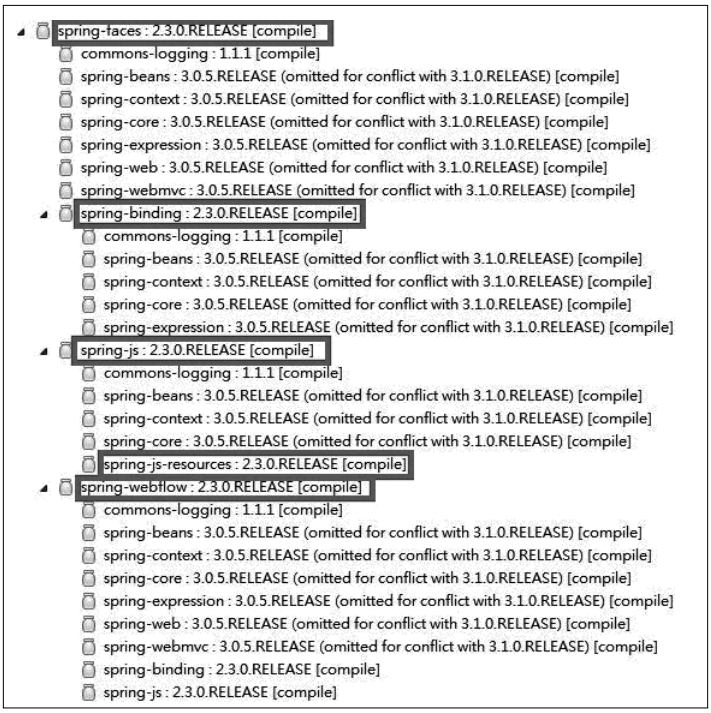


Рис. 18.5. Иерархия зависимостей Spring Web Flow

На рис. 18.5 видно, что Spring Faces зависит от других модулей Spring Web Flow наподобие Spring Web Flow, который является модулем ядра; в свою очередь, Spring Web Flow зависит от других проектов Spring, таких как Spring MVC, Spring Expression и т.д. Ниже перечислены модули из дистрибутива Spring Web Flow (версии 2.3.0.RELEASE).

- *Spring Web Flow (*`spring-webflow`*).* Это модуль ядра Spring Web Flow; он предоставляет инфраструктуру для управления потоками, а для каждого потока — его диалогом, состоянием и представлениями.

- *Spring Faces (*`spring-faces`*).* Проект Spring Web Flow предоставляет первоклассную поддержку JSF, и компоненты этой поддержки упакованы в модуль Spring Faces. Начиная с версии 2.2, в Spring Web Flow предлагается поддержка ключевых функциональных возможностей JSF 2, в том числе сохранение частичного состояния, запрос и обработка ресурсов, поддержка Ajax и т.п. Проект Spring Web Flow также тесно интегрирован с общими библиотеками компонент JSF, включая JBoss RichFaces (`www.jboss.org/richfaces`) и PrimeFaces (`www.primefaces.org`).
- *Spring JavaScript (*`spring-js`, `spring-js-resources`*).* Модуль Spring JavaScript предоставляет поддержку разработки представлений с помощью JavaScript. Например, в визуализированном HTML-представлении JavaScript может использоваться для Ajax-взаимодействия с севером и частичных обновлений представления. Кроме того, пакет Spring JavaScript Resources (`spring-js-resources.jar`) включает Dojo Toolkit (популярную JavaScript-библиотеку), а также JavaScript-библиотеку, предоставленную Spring, которая поддерживает интеграцию с Dojo Toolkit.
- *Spring Binding (*`spring-binding`*).* Модуль Spring Binding отвечает за привязку состояния представления к применяемой модели. Начиная с версии 2.1, в процессе привязки Spring Web Flow использует систему преобразования типов и форматирования Spring 3. Кроме того, в версию Spring Web Flow 2.3 также добавлена поддержка поверки достоверности бинов JSR-303.

Функциональные возможности Spring Web Flow

Spring Web Flow является расширением шаблона MVC, которое поддерживает разработку потоковых приложений и более точную область действия Java-бинов. Архитектура Spring Web Flow построена на основе следующих трех концепций.

- *Поток.* Поток — это бизнес-процесс, представляющий сценарий использования. В Spring Web Flow поток состоит из последовательности шагов, называемых *состояниями*. Каждое состояние обычно отображается пользователю внутри представления (существуют также состояния, не связанные с представлениями; например, состояние решения определяет следующее состояние потока на основе условия времени выполнения). В представлении могут возникать пользовательские события, которые обрабатываются состоянием. Эти события могут запускать переходы в другие состояния, что дает в результате навигацию по представлениям в рамках целого потока. Spring Web Flow предоставляет язык, специфичный для предметной области (domain-specific language — DSL), используемый для реализации очень сложных потоков в приложениях.
- *Представление.* Как и в шаблоне MVC, представление — это пользовательский интерфейс, который отображает состояние модели пользователю и обеспечивает взаимодействие с поддержкой Ajax и частичными обновлениями.
- *Диалог.* В веб-приложении различают три области действия бинов — запрос, сеанс и приложение. Области действия на уровне запроса и сеанса интенсивно используются на уровне представления. Однако во многих веб-приложениях, особенно основанных на потоках, бины модели должны поддерживать свое состояние на протяжении нескольких запросов, пока поток не будет завершен. В таком случае области действия на уровне запроса не достаточно. С другой стороны, применение области действия на уровне сеанса будет излишним, поскольку состояние тогда хранится в `HttpSession` пользователя в течение всего сеанса. Чтобы решить эту проблему, в Spring Web Flow вводится концепция *диалога* (conversation), кото-

рый хранит данные вплоть до завершения потока. Основная цель диалога — избежать помещения слишком большого объема данных на уровень сеанса, т.к. это может привести к нехватке памяти в случае большого количества одновременно работающих пользователей. При использовании Spring Web Flow очень важно понимать поддерживаемые области действия бинов и переменных. В табл. 18.1 перечислены доступные области действия и описано их назначение.

Таблица 18.1. Доступные области действия бинов в Spring Web Flow

Область действия	Назначение
На уровне потока	Переменные с областью действия на уровне потока создаются, когда поток запускается, и уничтожаются при его завершении. Эту область необходимо использовать, если переменная, которая обычно хранит состояние взаимодействия и модель, должна существовать на протяжении всего потока. Бин с областью действия на уровне потока должен реализовывать интерфейс `java.io.Serializable`. Переменная с областью действия на уровне потока не доступна в его подпотоках
На уровне представления	Переменные с областью действия на уровне представления создаются, когда поток заходит в представление (называемое `view-state`), и уничтожаются, когда состояние покидается. Другими словами, переменные с областью действия на уровне представления существуют только внутри представления. Если нужно отображать информацию (например, список контактов), которая требуется только в определенном представлении внутри потока, вы должны использовать область действия на уровне представления, чтобы минимизировать затраты памяти
На уровне запроса	Переменные с областью действия на уровне запроса создаются, когда поток вызывается, и удаляются при возврате из него
Временная	Переменные с временной (flash) областью действия создаются при запуске потока. Затем они будут очищаться после визуализации каждого представления и уничтожаться при завершении потока. Это удобно для переменных, которые существуют в каждом представлении, но имеют разные значения. Примером может служить сообщение, отображаемое на каждом шаге потока, которое отражает состояние потока или выдает информацию для пользователей
На уровне диалога	Область действия на уровне диалога подобна области действия на уровне потока. Единственное отличие от области действия на уровне потока заключается в том, что переменные с область действия на уровне диалога будут доступны своим потокам. Например, приложение заказов предоставляет поток для ввода информации о заказе. Внутри этого потока один из шагов связан с вызовом потока по имени *select-customer* (выбор заказчика) для выбора заказчика, размещающего заказ. Однако в подпотоке при выборе заказчика должна быть доступна информация о заказе, поскольку она содержит некоторый критерий фильтрации (к примеру, международный заказ может размещаться только заказчиками, проживающими в определенных странах). В этом случае для заказа должна выбираться область действия на уровне диалога

За более детальным описанием Spring Web Flow обращайтесь к документации по проекту (`http://static.springsource.org/spring-webflow/docs/2.3.x/reference/html/index.html`) и к книге *Pro Spring MVC with Web Flow* (Apress, 2012 г.).

Введение в JSF

Java Server Faces (JSF) — еще одна популярная технология представлений, используемая для разработки уровня презентаций в корпоративном приложении. Выражаясь

в общем, JSF — это веб-платформа MVC, управляемая запросами, которая основана на проектной модели пользовательского интерфейса, управляемой компонентами.

Однако первое поколение JSF (под названием JSF 1) было подвергнуто критике за чрезмерно крутую кривую обучения, сложность в построении специальных компонентов и отсутствие поддержки Ajax. В JSF 2 модель была значительно упрощена и, кроме того, добавлена встроенная поддержка Ajax.

JSF является стандартной технологией представлений в стеке JEE, поэтому каждый крупный производитель серверов приложений JEE (JBoss, IBM, Oracle и т.п.) поддерживает JSF и поставляет собственную библиотеку компонентов JSF. Более того, доступно множество проектов, как с открытым кодом, так и коммерческих, которые предлагают библиотеки компонентов JSF, например, JBoss RichFaces, PrimeFaces, ICEfaces и т.д.

В целом, технология JSF образована из четырех компонент: представление, взаимодействие с моделью, навигация и жизненный цикл. Все эти компоненты будут обсуждаться в последующих разделах.

Представление

В JSF, как и в шаблоне MVC, представление — это страница, отображающая данные для пользователя. Тем не менее, поскольку JSF является платформой, основанной на компонентах, представление описывается деревом компонентов пользовательского интерфейса, которые видны в представлении как дескрипторы разметки. Иерархия компонентов пользовательского интерфейса определяется вложением дескрипторов.

С выходом JSF 2 стандартной технологией для всех JSF-страниц является Facelets. Технология Facelets, которая принимает синтаксис XHTML, предоставляет встроенную поддержку шаблонов страниц, без необходимости в использовании других шаблонных инфраструктур вроде Apache Tiles.

Взаимодействие с моделью

В JSF представление хранит дерево компонентов пользовательского интерфейса со значениями и обеспечивает взаимодействие с пользователями. Когда пользователь предпринимает какое-то действие, серверу отправляется запрос на обработку. На стороне сервера значения из дерева компонентов пользовательского интерфейса будут переданы в модель для обработки. Этот процесс представляет действие между компонентами пользовательского интерфейса и моделью.

В JSF 2 для ссылки на объекты модели из компонентов пользовательского интерфейса применяется язык выражений JSF 2.0 (Expression Language — EL). Когда пользователю визуализируется ответ, компоненты пользовательского интерфейса получают свои значения из связанных с ними свойств модели. Во время этого взаимодействия автоматически происходит безопасное преобразование типов и проверка достоверности на серверной стороне.

Для поддержки высокоинтерактивных веб-страниц в JSF объекты модели обычно реализуются как “поддерживающие бины” на страничной основе. В этом случае объект модели воспроизводит состояние свойств модели, сохраненных представлением. При управлении такими бинами JSF также использует концепцию инверсии управления (IoC), и они называются бинами, управляемыми JSF (JSF Managed Beans). Эти бины могут быть определены в конфигурационном файле JSF (по имени `faces-config.xml`) или с помощью аннотаций. Тем не менее, язык EL в JSF допускает интеграцию с технологиями, выходящими за рамки JSF. Например, когда JSF применяется вместе с Spring Web Flow, стандартные бины Spring могут использоваться как бины, управляемые JSF, при помощи стандартного механизма внедрения зависимостей (DI), поддерживаемого Spring Framework.

Навигация

В JSF доступна инфраструктура для определения правил страничной навигации. Например, когда пользователь выполняет какое-то действие в представлении, серверу отправляется запрос. На стороне сервера “контроллер действий” обработает этот запрос, взаимодействуя с уровнем обслуживания, и затем, в зависимости от полученного результата и правил навигации, определенных в файле `faces-config.xml`, пользователю отобразится соответствующее представление.

Однако при использовании Spring Web Flow механизм навигации JSF не требуется. Как было показано в разделе “Функциональные возможности Spring Web Flow” ранее в этой главе, Spring Web Flow имеет встроенную поддержку определения потока, которая может заменять модель навигации, предоставляемую JSF.

Жизненный цикл приложения

Наиболее критически важный аспект работы с технологией JSF связан с пониманием ее жизненного цикла приложения. Весь жизненный цикл начинается, когда веб-приложение, использующее JSF, получает запрос, проходит через шесть фаз, а затем визуализирует и отображает ответ пользователю. Эти шесть фаз описаны ниже.

1. *Восстановление представления.* В JSF-приложении каждое представление имеет идентификатор и хранится в объекте сеанса `FacesContext` (из пакета `javax.faces.context`). В JSF представление является коллекцией компонентов со связанными текущими состояниями, которые могут быть сохранены либо на серверной, либо на клиентской стороне (это управляется параметром `javax.faces.STATE_SAVING_METHOD`) в дескрипторе веб-развертывания. Когда пользователь предпринимает какое-то действие в представлении и отправляет запрос серверу, первая фаза в JSF заключается в восстановлении представления с помощью его состояния.

2. *Применение запросов.* После восстановления дерева компонентов каждый компонент извлекает свое новое значение из параметров запроса, используя метод декодирования.

3. *Обработка проверки достоверности.* На этой фазе реализация JSF обрабатывает все валидаторы, зарегистрированные для компонентов в дереве. Она просматривает атрибуты компонентов, которые задают правила проверки достоверности, и сравнивает из значения с локальными значениями, сохраненными для компонентов. В JSF 2 доступна полная поддержка проверки достоверности бинов JSR-303.

4. *Обновление значений в модели.* После завершения проверки достоверности реализация JSF начнет проход по дереву компонентов с установкой соответствующих свойств серверных объектов в локальные значения компонентов. Будут обновляться только свойства бинов, на которые указывают атрибуты значений компонентов ввода. В этом процессе привязки также происходит преобразование типов и форматирование, и в случае сбоя преобразования жизненный цикл переходит сразу же на фазу визуализации ответа, так что страница будет визуализирована с отображением сообщений об ошибках.

5. *Вызов приложения.* На этой фазе реализация JSF вызывает приложение для обработки отправки форм. В данной точке значения компонентов будут преобразованы, проверены и применены к объектам модели, поэтому они могут использоваться для выполнения бизнес-логики приложения. Во время такого выполнения реализация JSF будет обрабатывать все события, возникающие на уровне приложения, такие как отправка формы или ссылка на другую страницу. Кроме того,

если обрабатываемое представление было реконструировано на основе информации о состоянии из предыдущего запроса, а какие-то компоненты сгенерировали события, то эти события транслируются заинтересованным слушателям.

6. *Визуализация ответа.* На этой финальной фазе результирующее представление отображается со всеми компонентами, находящимися в своих текущих состояниях.

Создав класс, реализующий интерфейс `javax.faces.event.PhaseListener`, в жизненном цикле можно установить собственный слушатель фазы, позволяющий выполнять необходимые действия перед или после любой фазы или даже всех фаз. Слушатель фазы имеет полный доступ ко всей инфраструктуре JSF, включая возможность манипулирования представлением.

Жизненный цикл показан на рис. 18.6.

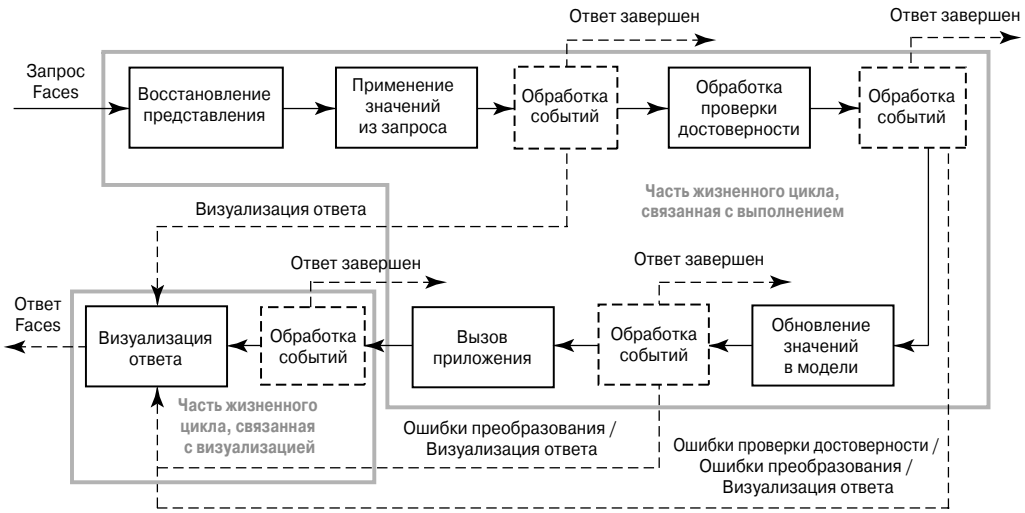


Рис. 18.6. Жизненный цикл JSF-приложения с выделенными частями, связанными с выполнением и визуализацией

Приведенное выше описание раскрывает только основы JSF. За более подробными сведениями обращайтесь в онлайновое руководство по JSF 2, доступное по адресу:

`www.coreservlets.com/JSF-Tutorial/jsf2`

Пример приложения Spring Web Flow

В этой главе для демонстрации использования комбинации Spring Web Flow и JSF в потоковых приложениях мы разработаем простой поток для реализации функции создания новых контактов.

В последующих разделах мы обсудим решение полного потока для создания новых контактов и структуру проекта для примера в этой главе.

Проектирование примера потока

Как указывалось ранее в этой главе, контакт содержит базовую информацию о пользователе (имя, фамилия, дата рождения), а также сведения о его хобби.

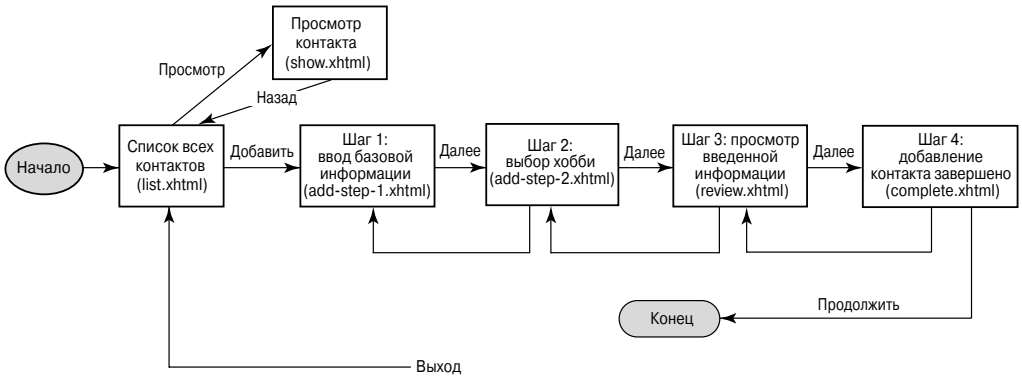


Рис. 18.7. Поток для создания новых контактов

При создании новых контактов будет применяться подход с мастером, который позволяет пользователям вводить информацию на разных страницах, просматривать всю введенную информацию и подтверждать ее, отправляя на обработку. Такой процесс очень похож на регистрацию, предлагаемую многими веб-сайтами.

На рис. 18.7 показан весь поток создания контактов и связанные с ним представления, которые понадобится реализовать. Метки на стрелках соответствуют кнопкам на страницах. Например, каждая страница имеет кнопку Back (Назад), и все страницы в процессе Add (Добавить) имеют кнопку Exit (Выход).

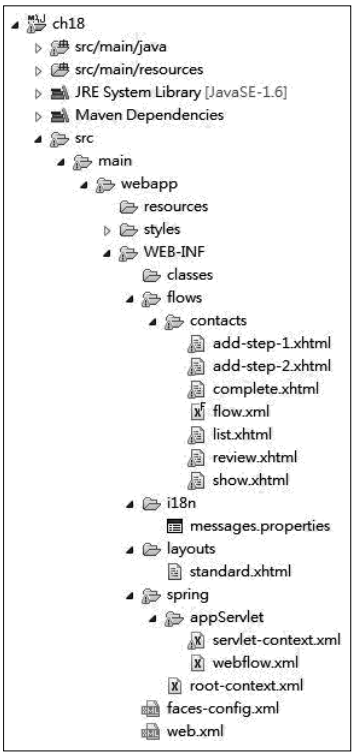


Рис. 18.8. Структура проекта Spring Web Flow

На рис. 18.7 видно, что первой страницей, которую увидят пользователи, будет список всех контактов. Когда пользователь выберет строку в списке и щелкнет на кнопке View Selected (Просмотреть выбранный), отобразится представление просмотра контакта. Щелкнув на кнопке Back (Назад), пользователь может вернуться из представления просмотра контакта обратно на страницу списка контактов.

Когда пользователь щелкает на кнопке Add Contact (Добавить контакт), запускается поток добавления контакта. Первый его шаг — это ввод базовой информации. Щелчок на кнопке Next (Далее) перемещает на представление для выбора хобби. На следующем шаге отображается страница обзора, на которой пользователь может пересмотреть всю введенную информацию. Когда пользователь подтверждает информацию, контакт сохраняется и отображается завершающее представление.

С любой страницы внутри потока пользователь может переместиться на предыдущую страницу, щелкнув на кнопке Back (Назад), или завершить поток и вернуться в представление списка контактов, щелкнув на кнопке Exit (Выход).

Структура проекта

Перед тем, как приступить к реализации потока добавления контакта, давайте рассмотрим структуру папок проекта.

На рис. 18.8 показана структура папок примера потокового веб-приложения, которое будет разработано в этой главе. Назначение каждой папки (с корнем в /src/main/webapp) описано в табл. 18.2.

Таблица 18.2. Описание структуры папок проекта Spring Web Flow

Имя папки	Назначение
styles	Хранит все обязательные стилевые таблицы. В этом примере мы будем использовать поддержку оформления темами PrimeFaces
WEB-INF/flows	Хранит все потоки приложения. Каждая папка потока содержит файлы, требуемые для поддержки данного потока. Например, папка contacts хранит файлы для потока добавления контакта. Здесь будут располагаться все файлы представлений, а также файл определения потока
WEB-INF/i18n	Хранит файлы для поддержки сообщений интернационализации
WEB-INF/layouts	Хранит шаблон компоновки, который будет использоваться Facelets
WEB-INF/spring	Хранит конфигурации WebApplicationContext для Spring MVC. Здесь находятся конфигурации как корневого уровня, так и уровня сервлетов диспетчеров. Также в этой папке хранится конфигурация Spring Web Flow

Конфигурация Spring Web Flow и JSF

В этом разделе обсуждаются конфигурации, необходимые для функционирования веб-приложения, использующего Spring Web Flow, JSF 2 и PrimeFaces.

Добавление обязательных зависимостей

Для разработки приложений с помощью Spring Web Flow, JSF 2 и PrimeFaces требуются зависимости, описанные в табл. 18.3. Добавьте их в свой проект.

Таблица 18.3. Зависимости Maven для Spring Web Flow, JSF 2 и PrimeFaces

Идентификатор группы	Идентификатор артефакта	Версия	Описание
org.springframework.webflow	spring-faces	2.3.0.RELEASE	Модуль Spring Faces из Spring Web Flow; поддерживает интеграцию с JSF 2
com.sun.faces	jsf-api	2.0.7	API-интерфейс JSF 2. Обратите внимание, что на момент написания этой книги существовала проблема с совместимостью между JSF 2.1 и Spring Web Flow 2.3
com.sun.faces	jsf-impl	2.0.7	Эталонная реализация JSF 2 от Oracle Sun
org.primefaces	primefaces	3.0.1	Библиотека компонентов PrimeFaces, которая поддерживает JSF 2

Конфигурирование JSF

Для JSF-приложения должен быть указан отдельный файл конфигурации JSF (файл faces-config.xml в папке /WEB-INF). Этот файл обычно хранит общую конфигурацию JSF, информацию по управляемым бинам (не требуется, если используется стиль с аннотациями) и правила навигации. Однако мы будем применять технологию Spring Web Flow, которая интегрируется с помощью инверсии управления Spring Framework для

внедрения зависимостей, поэтому конфигурация в данном файле минимальна. В листинге 18.1 показано содержимое файла `WEB-INF/faces-config.xml`.

Листинг 18.1. Файл конфигурации JSF

```
<?xml version='1.0' encoding='UTF-8'?>
<faces-config xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
      http://java.sun.com/xml/ns/javaee/web-facesconfig_2_0.xsd"
      version="2.0">

   <application>
      <locale-config>
         <default-locale>en</default-locale>
         <supported-locale>en</supported-locale>
      </locale-config>
      <message-bundle>MessageResources</message-bundle>
      <resource-bundle>
         <base-name>MessageResources</base-name>
         <var>msg</var>
      </resource-bundle>
   </application>

</faces-config>
```

В листинге 18.1 видно, что единственная требуемая конфигурация предназначена для интернационализации. В этом примере будет поддерживаться только английский язык, и пакет сообщений хранится в файле `MessageResources.properties` (в пути класса внутри папки `/src/main/resources`). Дескриптор `<var>` указывает, что сообщения будут доступны в EL через имя переменной `msg`, которую мы увидим в действии при разработке представлений.

Конфигурирование дескриптора веб-развертывания

Следующий шаг связан с конфигурированием файла `web.xml` для поддержки JSF. Модифицированный дескриптор веб-развертывания приведен в листинге 18.2.

Листинг 18.2. Модифицированный дескриптор веб-развертывания

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
   http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">

   <context-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>/WEB-INF/spring/root-context.xml</param-value>
   </context-param>

   <listener>
      <listener-class>
         org.springframework.web.context.ContextLoaderListener
      </listener-class>
   </listener>

   <!-- JSF 2 -->
```

```
    <!-- Использовать шаблоны представлений JSF, сохраненные как *.xhtml,
         вместе с Facelets -->
    <context-param>
       <param-name>javax.faces.DEFAULT_SUFFIX</param-name>
       <param-value>.xhtml</param-value>
    </context-param>

    <!-- Разрешить во время разработки специальный отладочный вывод Facelets -->
    <context-param>
       <param-name>javax.faces.PROJECT_STAGE</param-name>
       <param-value>Development</param-value>
    </context-param>

    <!-- Заставить Facelets обновлять шаблоны во время разработки -->
    <context-param>
       <param-name>javax.faces.FACELETS_REFRESH_PERIOD</param-name>
       <param-value>1</param-value>
    </context-param>

    <!-- Нужен для того, чтобы реализация JSF могла быть инициализирована;
         *не* используется во время выполнения -->
    <servlet>
       <servlet-name>Faces Servlet</servlet-name>
       <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
       <load-on-startup>1</load-on-startup>
    </servlet>

    <!-- Нужен для того, чтобы реализация JSF могла быть инициализирована -->
    <servlet-mapping>
       <servlet-name>Faces Servlet</servlet-name>
       <url-pattern>*.faces</url-pattern>
    </servlet-mapping>

    <servlet>
       <servlet-name>appServlet</servlet-name>
       <servlet-class>org.springframework.web.servlet.DispatcherServlet
       </servlet-class>
       <init-param>
          <param-name>contextConfigLocation</param-name>
          <param-value>/WEB-INF/spring/appServlet/servlet-context.xml
          </param-value>
       </init-param>
       <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
       <servlet-name>appServlet</servlet-name>
       <url-pattern>/app/*</url-pattern>
    </servlet-mapping>

</web-app>
```

В листинге 18.2 конфигурация, относящаяся к JSF, выделена полужирным. Во-первых, установлено множество параметров конфигурации JSF.

- Параметр `javax.faces.DEFAULT_SUFFIX` определяет суффикс для файлов JSF-представлений, который для работы с Facelets должен выглядеть как `xhtml`.
- Параметр `javax.faces.PROJECT_STAGE` определяет стадию проекта. Значение `Development` приводит к отображению большего числа сообщений в веб-браузере,

что упрощает поиск и устранение неполадок. В производственной версии должно использоваться значение `Production`.

- Параметр `javax.faces.FACELETS_REFRESH_PERIOD` определяет интервал в секундах, через который компилятор Facelets будет проверять, не вносились ли изменения в страничный шаблон. Установка для него значения `1` приводит к ежесекундной проверке, так что часто вносимые изменения в страничные шаблоны во время разработки будут немедленно обнаруживаться. В производственной версии укажите для этого параметра значение `-1`, чтобы все скомпилированные шаблоны не проверялись на предмет изменений.

Во-вторых, объявлен сервлет с классом `javax.faces.webapp.FacesServlet`. Этот сервлет служит только для начальной загрузки среды JSF.

Вы также увидите определение сервлета диспетчера (`DispatcherServlet`) для приложения контактов, которое отображено на шаблон URL вида `/app/*`. Поскольку технология Spring Web Flow тесно интегрирована с модулем Spring MVC, мы используем сервлет диспетчера для направления запроса исполняющего средству Spring Web Flow с целью обработки.

Конфигурирование Spring Web Flow и Spring MVC

Финальный шаг конфигурирования относится к Spring Web Flow и Spring MVC. Для Spring Web Flow мы подготовим отдельный файл конфигурации. При использовании Spring MVC этот файл конфигурации обычно хранится в той же папке, что и файл конфигурации сервлета диспетчера. В листинге 18.3 приведено содержимое файла конфигурации `WEB-INF/spring/appServlet/webflow.xml`.

Листинг 18.3. Файл конфигурации Spring Web Flow

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/webflow-config"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:beans="http://www.springframework.org/schema/beans"
   xmlns:faces="http://www.springframework.org/schema/faces"
   xsi:schemaLocation="http://www.springframework.org/schema/webflow-config
      http://www.springframework.org/schema/webflow-config/
spring-webflow-config-2.0.xsd
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
      http://www.springframework.org/schema/faces
      http://www.springframework.org/schema/faces/spring-faces-2.0.xsd">

   <!-- Выполняет потоки: центральная точка входа в систему Spring Web Flow -->
   <flow-executor id="flowExecutor">
      <flow-execution-listeners>
         <listener ref="facesContextListener"/>
      </flow-execution-listeners>
   </flow-executor>

   <!-- Реестр для определений исполняемых потоков -->
   <flow-registry id="flowRegistry"
      flow-builder-services="flowBuilderServices"
      base-path="/WEB-INF/flows">
      <flow-location-pattern value="/**/flow.xml" />
   </flow-registry>

   <!-- Конфигурирование интеграции Spring Web Flow и JSF -->
```

```
   <faces:flow-builder-services id="flowBuilderServices"/>

   <!-- Слушатель для создания и освобождения FacesContext -->
   <beans:bean id="facesContextListener"
class="org.springframework.faces.webflow.FlowFacesContextLifecycleListener"/>
</beans:beans>
```

В листинге 18.3 объявлено пространство имен `webflow-config-namespace` как используемое по умолчанию. Кроме того, объявлено пространство имен `faces` для конфигураций, связанных с JSF.

Во-первых, дескриптор `<flow-executor>` определяет экземпляр интерфейса `org.springframework.webflow.executor.FlowExecutor`, который является центральным фасадом и точкой входа в систему Spring Web Flow для управления выполнением определений потоков.

Во-вторых, дескриптор `<flow-registry>` определяет реестр для определений потоков внутри приложения. Как показано в листинге, определения потоков хранятся в файлах с именем `flow.xml` под родительской папкой `/WEB-INF/flows`. Например, файл `/WEB-INF/flows/contacts/flow.xml` будет хранить определение потока для приложения контактов.

В-третьих, дескриптор `<faces:flow-builder-services>` определяет экземпляр класса `org.springframework.webflow.engine.builder.support.FlowBuilderServices` с интеграцией с JSF, который требуется для Spring Web Flow при построении потоков внутри реестра потоков.

Наконец, в-четвертых, бин с идентификатором `facesContextListener` и классом реализации `FlowFacesContextLifecycleListener` определяет экземпляр интерфейса `org.springframework.webflow.execution.FlowExecutionListener`, который создает экземпляр класса `org.springframework.faces.webflow.FlowFacesContext`. Класс `FlowFacesContext` расширяет абстрактный класс `FacesContext` из JSF (пакет `javax.faces.context`), который предоставляет контекст для исполняющей среды JSF 2.

В листинге 18.4 показана конфигурация `WebApplicationContext` сервлета диспетчера Spring MVC (`/WEB-INF/spring/appServlet/servlet-context.xml`).

Листинг 18.4. Файл конфигурации Spring MVC

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/mvc"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:beans="http://www.springframework.org/schema/beans"
   xmlns:p="http://www.springframework.org/schema/p"
   xmlns:context="http://www.springframework.org/schema/context"
   xmlns:faces="http://www.springframework.org/schema/faces"
   xsi:schemaLocation="http://www.springframework.org/schema/mvc
      http://www.springframework.org/schema/mvc/spring-mvc-3.1.xsd
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context-3.1.xsd
      http://www.springframework.org/schema/faces
      http://www.springframework.org/schema/faces/spring-faces.xsd">

   <beans:import resource="webflow.xml" />

   <annotation-driven />

   <resources location="/" mapping="/resources/**"/>

   <faces:resources/>
```

```
    <context:component-scan base-package="com.apress.prospring3.ch18.web" />

    <beans:bean
          class="org.springframework.webflow.mvc.servlet.FlowHandlerMapping">
        <beans:property name="flowRegistry" ref="flowRegistry" />
    </beans:bean>
    <beans:bean class="org.springframework.web.servlet.view.UrlBasedViewResolver">
        <beans:property name="viewClass"
            value="org.springframework.faces.mvc.JsfView" />
        <beans:property name="prefix" value="/WEB-INF/views/" />
        <beans:property name="suffix" value=".xhtml" />
    </beans:bean>
    <beans:bean class="org.springframework.faces.webflow.JsfFlowHandlerAdapter">
        <beans:property name="flowExecutor" ref="flowExecutor" />
    </beans:bean>
    <beans:bean
class="org.springframework.context.support.ReloadableResourceBundleMessageSource"
        id="messageSource"
        p:basenames="WEB-INF/i18n/messages,WEB-INF/i18n/application"
        p:fallbackToSystemLocale="false"/>
</beans:beans>
```

Большинство бинов в листинге 18.4 должны быть уже знакомы. Вначале импортируется конфигурация `webflow.xml`. Дескриптор `<faces:resources/>` включает поддержку запросов к ресурсам JSF 2. Анонимный бин с классом `FlowHandlerMapping` реализует интерфейс `HandlerMapping` из Spring MVC для интеграции с модулем Spring MVC в среде Spring Web Flow. Он отображает путь URL на идентификатор зарегистрированных определений потоков.

Для распознавателя представлений мы используем тот же самый `UrlBasedViewResolver` из Spring MVC, но с суффиксом, измененным на `xhtml`. Анонимный бин с классом реализации `JsfFlowHandlerAdapter` заменяет экземпляр стандартного интерфейса `AjaxHandler` (из пакета `org.springframework.js.ajax` модуля `spring-js`) классом `JsfAjaxHandler` (из пакета `org.springframework.faces.webflow` модуля `spring-faces`), когда JSF 2 является исполняющей средой.

Наконец, бин с идентификатором `messageSource` — это тот же бин, который применялся для Spring MVC.

Реализация примера потока

На этом инфраструктура и конфигурирование завершены, и можно приступать к реализации потока добавления контакта. В последующих разделах мы пройдем через все шаги по реализации потока. Мы обсудим различные шаги, включая определение потока, шаблонное представление, разнообразные представления контактов, класс контроллера, классы поддерживающих бинов и т.д.

Создание определения потока

Начнем с определения потока добавления контактов, который был представлен на рис. 18.7. Как упоминалось в разделе, посвященном конфигурированию, файлом определения потока является `/WEB-INF/flows/contacts/flow.xml`. В STS существует удобный способ создания файла с определением веб-потока. Щелкните правой кнопкой мыши на папке `contacts` и выберите в контекстном меню пункт New⇨Spring Web Flow Definition File (Создать⇨Файл определения Spring Web Flow). Определение потока показано в листинге 18.5.

Листинг 18.5. Определение веб-потока

```
<?xml version="1.0" encoding="UTF-8"?>
<flow xmlns=http://www.springframework.org/schema/webflow
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/webflow
      http://www.springframework.org/schema/webflow/spring-webflow-2.0.xsd"
      start-state="start">

   <view-state id="start" view="list.xhtml">
      <on-entry>
         <evaluate expression="contactController.newContactListBean()"
                   result="viewScope.contactListBean"/>
      </on-entry>
      <transition on="view" to="show">
         <set name="requestScope.contactId"
              value="contactListBean.selectedContact.id"/>
      </transition>
      <transition on="add" to="add-step-1">
         <evaluate expression="contactController.newContactBean()"
                   result="flowScope.contactBean"/>
      </transition>
   </view-state>

   <view-state id="add-step-1" view="add-step-1.xhtml">
      <transition on="next" to="add-step-2"/>
   </view-state>

   <view-state id="add-step-2" view="add-step-2.xhtml">
      <transition on="next" to="review"/>
      <transition on="back" to="add-step-1"/>
   </view-state>

   <view-state id="review" view="review.xhtml">
     <transition on ="next" to="complete">
      <evaluate expression="contactController.saveContact(flowRequestContext)"/>
     </transition>
     <transition on="back" to="add-step-2"/>
   </view-state>

   <view-state id="complete" view="complete.xhtml">
      <transition on="continue" to="finish"/>
   </view-state>

   <end-state id="finish"/>

   <view-state id="show" view="show.xhtml">
     <on-entry>
      <evaluate expression="contactController.showContact(flowRequestContext)"
                result="viewScope.contact"/>
     </on-entry>
     <transition on ="back" to="start"/>
   </view-state>

   <global-transitions>
      <transition on="exit" to="start"/>
   </global-transitions>

</flow>
```

В листинге 18.5 отражен завершенный поток приложения контактов. Соответствующие представления, методы контроллера и поддерживающие бины будут разработаны позже. Поток определен с использованием языка DSL из Spring Web Flow. Как и в конфигурации `webflow.xml`, бин реестра потока сконфигурирован с базовым путем `/WEB-INF/flows`, и этот файл определения потока находится в папке `contacts`. Данный поток будет автоматически зарегистрирован в Spring Web Flow с идентификатором `contacts`, доступным через URL `/ch18/app/contacts`.

В определении потока, помимо объявлений пространств имен, предоставлен атрибут `start-state` со значением `start`, который указывает состояние потока, когда он начинается. В этом случае начальным состоянием является состояние с идентификатором `start`. В начальном состоянии присутствует атрибут `view` со значением `list.xhtml`, который указывает файл для визуализации в этом состоянии. Затем внутри дескриптора `<view-state>` в дескрипторе `<on-entry>` (который означает, что поток вошел в это состояние), будет выполнен метод `newContactListBean()` бина Spring с именем `contactController`. Он извлечен список контактов. Результирующий список контактов (хранящийся в классе `ContactListBean`, который будет реализован позже) сохраняется в переменной с областью видимости на уровне представления. Дескрипторы `<transition>` определяют возможные навигационные пути в рамках потока на основе действий, предпринимаемых пользователем. Внутри начального состояния, если пользователь выберет контакт в представлении списка контактов и щелкнет на кнопке View Selected (Просмотреть выбранный), произойдет переход на представление просмотра контакта. Если пользователь щелкнет на кнопке Add Contact (Добавить контакт), произойдет переход на первый шаг потока добавления контакта.

Другие дескрипторы `<view-state>` определены в соответствие со спроектированным потоком добавления контакта. Внутри каждого состояния пользователь может выбрать продолжение или возврат на шаг назад. Дескриптор `<global-transitions>` определяет глобальное правило переходов. В этом случае определено глобальное правило, заключающееся в том, что всегда, когда пользователь щелкает на кнопке Exit (Выход), инициируется переход выхода, а поток возвращается в начальное состояние, которым является представление списка контактов.

В состояниях представления с идентификаторами `review` и `show` обратите внимание, что соответствующим методам передается аргумент по имени `flowRequestContext`. Указанный аргумент — это неявная переменная языка EL в Spring Web Flow, которая разрешает доступ к API-интерфейсу `RequestContext`, представляющему текущий запрос потока.

Дескриптор `<end-state>` определяет конечное состояние потока. По умолчанию, когда поток завершается, будет создан новый поток, и Spring Web Flow перейдет в начальное состояние потока. Если конечное состояние появится в подпотоке, произойдет возврат в родительский поток.

В проекте Spring Web Flow поддерживается множество других функциональных возможностей, таких как набор других состояний (например, состояние принятия решения, состояние действия и т.п.) и определение подпотоков, с передачей входных и выходных параметров между потоками. За полным описанием обращайтесь к онлайновой документации Spring Web Flow, доступной по адресу `http://static.springsource.org/spring-webflow/docs/2.3.x/reference/html/index.html`.

В STS можно также просматривать поток в графическом формате. Первым делом, удостоверьтесь, что в проект добавлен файл `flow.xml` в качестве определения потока. После этого щелкните правой кнопкой мыши на файле `flow.xml` и выберите в контекстном меню пункт Open With⇨Spring Web Flow XML Editor (Открыть с помощью⇨XML-редактора Spring Web Flow), как показано на рис. 18.9.

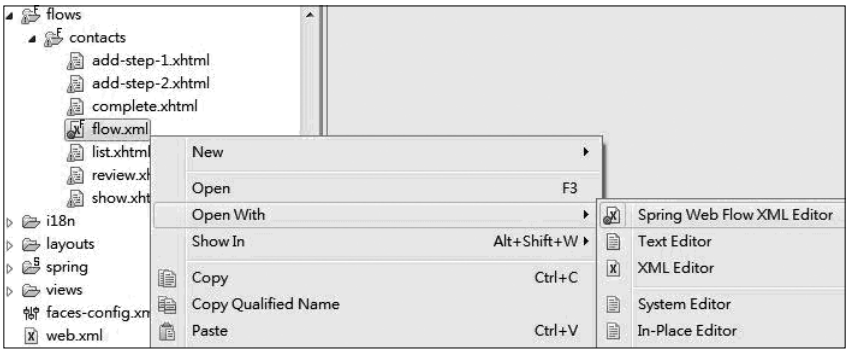


Рис. 18.9. Открытие определения потока в XML-редакторе Spring Web Flow

Затем щелкните на вкладке flow-graph (граф потока) и определение потока будет представлено в области редактирования в графическом формате, обеспечивая более дружественно к пользователю восприятие (рис. 18.10).

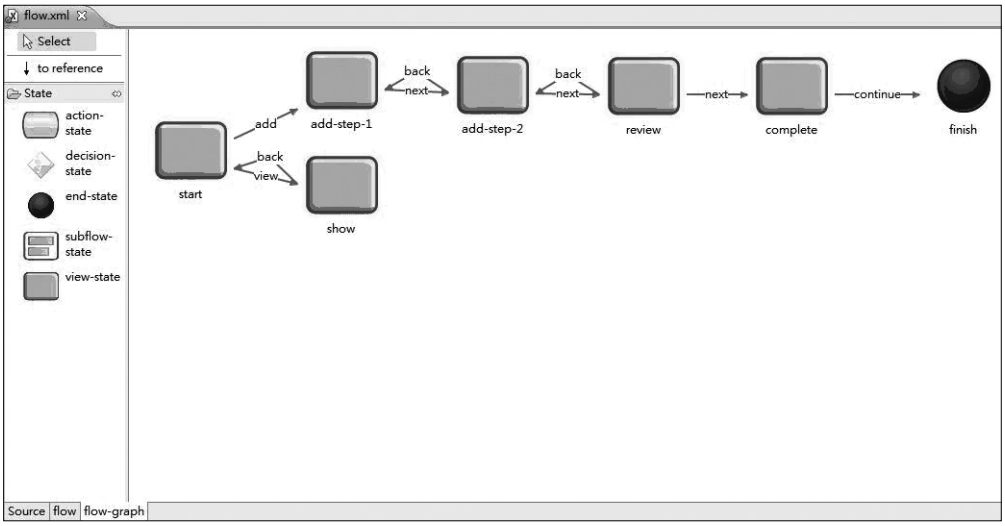


Рис. 18.10. Вкладка flow-graph в STS

Реализация шаблонной страницы

Перед реализацией индивидуальных представлений понадобится реализовать шаблонную страницу, которая будет использоваться всеми представлениями в примерах этой главы. Код шаблонной страницы (`/WEB-INF/layouts/standard.xhtml`) показан в листинге 18.6.

Листинг 18.6. Шаблонная страница

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
   xmlns:h="http://java.sun.com/jsf/html"
```

```
   xmlns:f="http://java.sun.com/jsf/core"
   xmlns:ui="http://java.sun.com/jsf/facelets">
<f:view contentType="text/html">
<h:head>
   <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
   <title>JSF 2, Spring Web Flow, and PrimeFaces</title>
   <link rel="stylesheet" href="${request.contextPath}/app/resources/styles/
blueprint/screen.css" type="text/css" media="screen, projection" />
   <link rel="stylesheet" href="${request.contextPath}/app/resources/styles/
blueprint/print.css" type="text/css" media="print" />
   <!--[if lt IE 8]>
      <link rel="stylesheet" href="${request.servletPath}/styles/blueprint/
ie.css" type="text/css" media="screen, projection" />
   <![endif]-->
</h:head>
<h:body>
   <div class="container">
      <div>
         <h1>JSF 2, PrimeFaces, and Spring Web Flow</h1>
         <h3 class="alt">
            <ui:insert name="title"/>
         </h3>
         <hr/>
      </div>
      <div>
         <ui:insert name="content"/>
      </div>
   </div>
</h:body>
</f:view>
</html>
```

Компоновка шаблонной страницы в листинге 18.6 довольно проста. В JSF существуют три основных библиотеки дескрипторов: `h-namespace`, `f-namespace` и `ui-namespace`.

- Библиотека `h-namespace` соответствует HTML-компонентам. Например, дескриптор `<h:body>` соответствует HTML-дескриптору `<BODY>`.
- Библиотека `f-namespace` соответствует средствам, специфичным для JSF. Например, дескриптор `<f:view>` определяет представление JSF. Внутри этого дескриптора исполняющая среда JSF будет сохранять все компоненты пользовательского интерфейса, вместе с их иерархией и состоянием.
- Библиотека `ui-namespace` применяется Facelets для определения элементов внутри шаблона. Например, дескриптор `<ui:insert>` указывает на необходимость вставки на его место некоторого контента. Представления, использующие этот шаблон, должны предоставлять включаемый контент.

Реализация специального преобразователя

Для атрибута даты рождения контакта мы используем Joda-Time. Однако JSF и PrimeFaces не имеют собственной поддержки отображения типов Joda-Time. В результате мы должны реализовать специальный преобразователь JSF, который будет применяться при отображении даты рождения контакта в интерфейсной части.

В JSF реализация специального преобразователя довольно проста. Все что понадобится сделать — это реализовать интерфейс `javax.faces.convert.Converter`. Класс специального преобразователя показан в листинге 18.7.

Листинг 18.7. Специальный преобразователь JSF

```
package com.apress.prospring3.ch18.web.converter;

import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.convert.Converter;
import javax.faces.convert.FacesConverter;

import org.joda.time.DateTime;
import org.joda.time.format.DateTimeFormat;

@FacesConverter("jodaDataTimeConverter")
public class JodaDateTimeConverter implements Converter {

   private static final String PATTERN = "yyyy-MM-dd";

   @Override
   public Object getAsObject(FacesContext ctx, UIComponent component, String value)
   {
      return DateTimeFormat.forPattern(PATTERN).parseDateTime(value);
   }
   @Override
   public String getAsString(FacesContext ctx, UIComponent component, Object value)
   {
      DateTime dateTime = (DateTime) value;
      return DateTimeFormat.forPattern(PATTERN).print(dateTime);
   }
}
```

В листинге 18.7 класс `JodaDateTimeConverter` реализует JSF-интерфейс `Converter` и переопределяет два его метода — `getAsObject()` (преобразует `String` в тип Joda-Time) и `getAsString()` (преобразует тип Joda-Time в `String`). Также обратите внимание, что класс аннотирован посредством аннотации `@FacesConverter`, которая указывает исполняющей среде JSF, что это преобразователь JSF. Атрибут, определенный в этой аннотации — идентификатор преобразователя. Этот преобразователь будет использоваться при реализации представления списка контактов.

Реализация контроллера и поддерживающего бина

Следующий шаг заключается в реализации класса контроллера и поддерживающего бина для представления списка контактов.

Вначале давайте реализуем поддерживающий бин — класс `ContactListBean`. Этот класс служит для хранения списка контактов, а также выбранного контакта в ситуации, когда пользователь щелкает на какой-либо строке в таблице данных. Класс `ContactListBean` показан в листинге 18.8.

Листинг 18.8. Класс `ContactListBean`

```
package com.apress.prospring3.ch18.web.view;

import java.io.Serializable;
import java.util.List;

import com.apress.prospring3.ch18.domain.Contact;
```

```
public class ContactListBean implements Serializable {
   private List<Contact> contacts;
   private Contact selectedContact;
   // Методы извлечения/установки не показаны.
}
```

В листинге 18.8 видно, что это простой бин POJO с атрибутами, предназначенными для хранения списка извлеченных контактов и контакта, который выбран пользователем в интерфейсной части. Обратите внимание, что поддерживающий бин должен реализовывать интерфейс `Serializable`.

Класс `ContactController` показан в листинге 18.9.

Листинг 18.9. Класс `ContactController`

```
package com.apress.prospring3.ch18.web.controller;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

import com.apress.prospring3.ch18.service.ContactService;
import com.apress.prospring3.ch18.web.view.ContactListBean;

@Component("contactController")
public class ContactController {

   private static final Logger logger = LoggerFactory.
getLogger(ContactController.class);

   @Autowired
   private ContactService contactService;

   public ContactListBean newContactListBean() {
      ContactListBean contactListBean = new ContactListBean();
      contactListBean.setContacts(contactService.findAll());
      return contactListBean;
   }
}
```

Этот класс также является простым POJO. Мы просто определяем его как бин Spring (используя аннотацию `@Component`), и Spring Web Flow сможет найти его и выполнить метод при входе в начальное состояние. Как видите, метод `newContactListBean()` просто вызывает метод из уровня обслуживания, сохраняет список контактов в новом экземпляре класса `ContactListBean` и затем возвращает этот экземпляр. В определении потока возвращаемый объект будет сохранен в переменной с областью видимости на уровне представления, доступной JSF во время визуализации ответа.

На заметку! В листинге 18.9 применяется аннотация `@Component`, а не `@Controller`. Причина в том, что в этой главе для управления потоком приложения мы используем Spring Web Flow, а не классы контроллеров Spring MVC. Поэтому при реализации представления списка контактов мы должны объявлять класс как обычный бин Spring.

Теперь можно реализовать собственно представление списка контактов, которое является начальным состоянием потока. В листинге 18.10 приведен код этого представления (`/WEB-INF/flows/contacts/list.xhtml`).

Листинг 18.10. Представление списка контактов

```
<!DOCTYPE composition PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://
www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
   xmlns:ui="http://java.sun.com/jsf/facelets"
   xmlns:h="http://java.sun.com/jsf/html"
   xmlns:f="http://java.sun.com/jsf/core"
   xmlns:p="http://primefaces.org/ui"
   template="/WEB-INF/layouts/standard.xhtml">
<ui:define name="title">#{msg['application_name']}</ui:define>
<ui:define name="content">
<h:form>
   <h:panelGrid columns="1">
   <p:commandButton value="#{msg['label_contact_new']}" action="add"/>
   <p:dataTable var="contact" value="#{contactListBean.contacts}"
              paginator="true" rows="10"
              paginatorTemplate="{CurrentPageReport} {FirstPageLink}
{PreviousPageLink} {PageLinks} {NextPageLink} {LastPageLink}
{RowsPerPageDropdown}"
              rowsPerPageTemplate="5,10,15"
              rowKey="#{contact.id}"
              selectionMode="single"
              selection="#{contactListBean.selectedContact}">
      <f:facet name="header">
         #{msg['label_contact_list']}
      </f:facet>
      <p:column sortBy="#{contact.firstName}">
         <f:facet name="header">
            <h:outputText value="#{msg['label_contact_first_name']}" />
         </f:facet>
         <h:outputText value="#{contact.firstName}" />
      </p:column>
      <p:column sortBy="#{contact.lastName}">
         <f:facet name="header">
            <h:outputText value="#{msg['label_contact_last_name']}" />
         </f:facet>
         <h:outputText value="#{contact.lastName}" />
      </p:column>
      <p:column sortBy="#{contact.birthDate}">
         <f:facet name="header">
            <h:outputText value="#{msg['label_contact_birth_date']}" />
         </f:facet>
         <h:outputText value="#{contact.birthDate}">
            <f:converter converterId="jodaDataTimeConverter" />
         </h:outputText>
      </p:column>
      <f:facet name="footer">
         <p:commandButton value="View Selected" icon="ui-icon-search"
               update="form:display" action="view"/>
      </f:facet>
   </p:dataTable>
   </h:panelGrid>
</h:form>
</ui:define>
</ui:composition>
```

В листинге 18.10 определено одно новое пространство имен (p), которое соответствует JSF-компонентам PrimeFaces. Дескриптор <ui:composition> из Facelets служит для определения композиции компонентов представления, используя standard.xhtml в качестве файла шаблона. Дескрипторы <ui:define> определяют контент для включения в шаблон. Как видите, были предоставлены две переменных (title и content), требуемые шаблоном (указанные с помощью дескриптора <ui:insert> в standard.xhtml).

В области content мы с помощью компонента DataTable из PrimeFaces отображаем список контактов в табличном формате. Для доступа к атрибутам модели в поддерживающих бинах, которые обслуживаются Spring Web Flow на основе определения потока, применяются выражения EL. В дескрипторе <p:dataTable> предоставлены конфигурационные параметры, такие как поддержка разбиения на страницы, количество строк на страницу, режим выбора и т.д. Внутри таблицы дескрипторы <p:column> определяют каждое отображаемое свойство контакта. Обратите внимание, что для атрибута birthDate используется дескриптор <f:converter> с атрибутом converterId, который указывает на специальный преобразователь для типа DateTime, определенного в Joda-Time. В области нижнего колонтитула таблицы (footer) размещена кнопка View Selected (Просмотреть выбранный), предназначенная для просмотра информации о выбранном контакте. На веб-сайте PrimeFaces доступно руководство пользователя, которое можно свободно загрузить (www.primefaces.org/documentation.html), а также исчерпывающая демонстрация, включающая внешний вид, описание и исходный код, для каждого компонента (www.primefaces.org/showcase-labs/ui/home.jsf).

При разработке представления с дескрипторами JSF и PrimeFaces можно также включить средство помощника по коду. В STS это делается легко. Щелкните правой кнопкой мыши на проекте и выберите в контекстном меню пункт Properties (Свойства). В списке слева выберите элемент Project Facets (Аспекты проекта). После этого справа в колонке Project Facet (Аспект проекта) отметьте флажок JavaServer Faces, как показано на рис. 18.11.

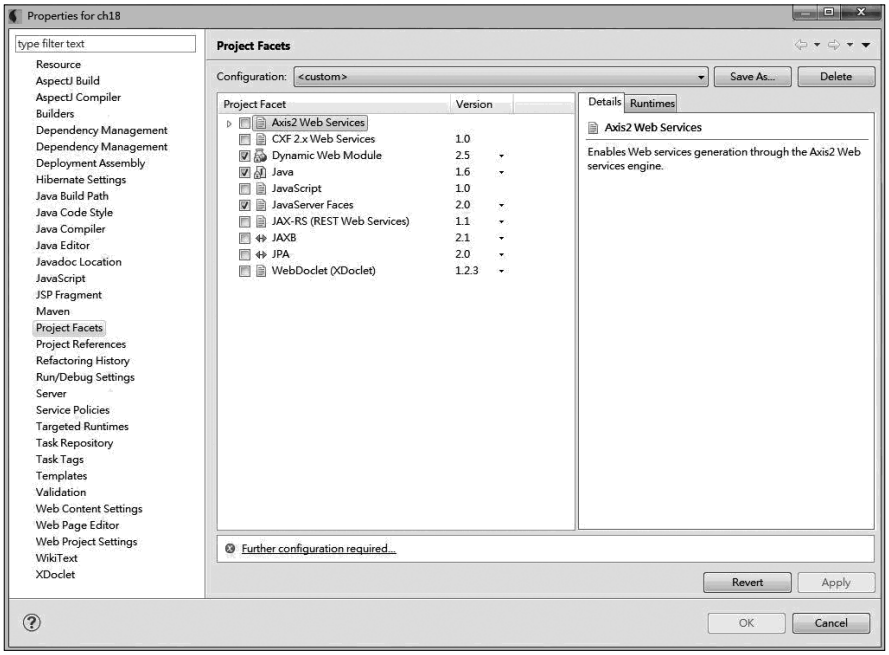


Рис. 18.11. Включение аспекта проекта JSF в STS

Внизу окна отобразится уведомление Further configuration required (Требуется дальнейшее конфигурирование) в виде ссылки. Щелчок на этой ссылке приводит к открытию диалогового окна для конфигурирования JSF. Поскольку мы уже добавили зависимости для JSF и определили файла `faces-config.xml`, здесь ничего определять не нужно. Просто снимите отметку с флажка, как показано на рис. 18.12.

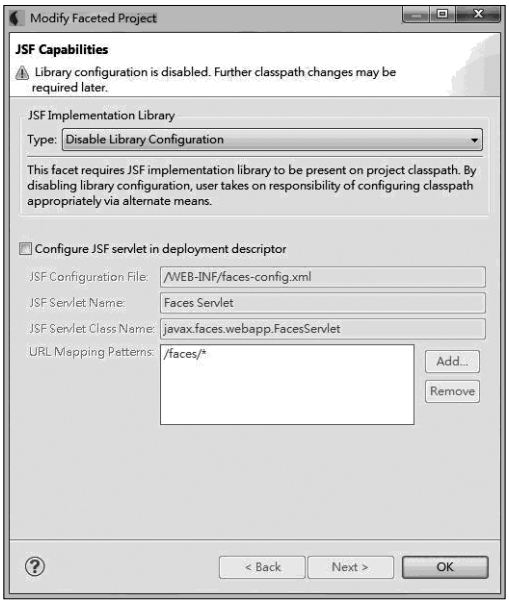


Рис. 18.12. Конфигурирование проекта JSF в STS

Теперь средство помощника по коду для JSF-представления включено. На странице редактора XHTML при нажатии комбинации клавиш <Ctrl+пробел> или при вводе префикса библиотеки дескрипторов будет отображаться диалоговое окно помощника, показанное на рис. 18.13.

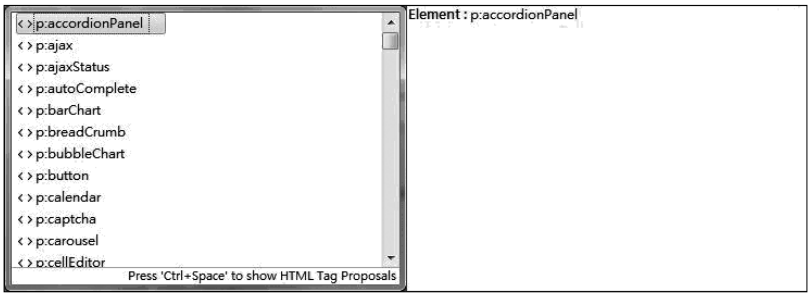


Рис. 18.13. Помощник по коду JSF в STS

Перед тем, как можно будет тестировать представление, мы также должны обновить параметры веб-проекта. Чтобы сделать это, щелкните правой кнопкой мыши на проекте и выберите в контекстном меню пункт Properties (Свойства). Затем в списке слева выберите элемент Web Project Settings (Параметры веб-проекта). В правой части диалогового окна введите `ch18` в поле Context root (Корень контекста), как показано на рис. 18.14.

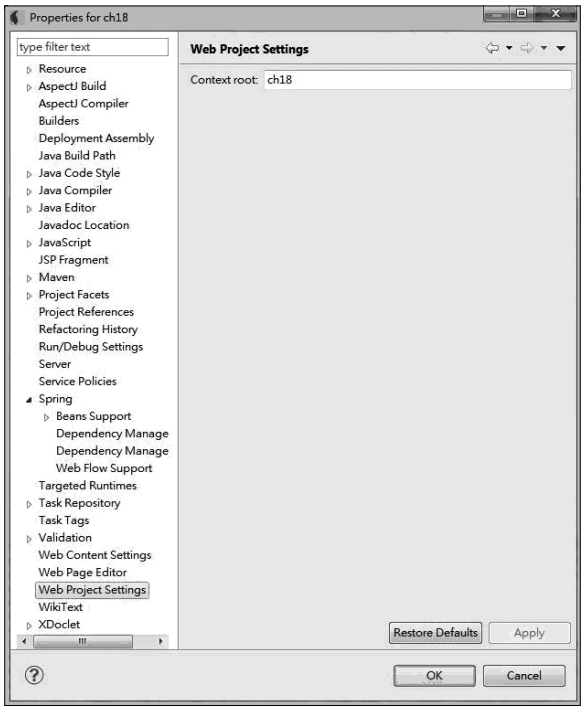


Рис. 18.14. Параметры веб-проекта в STS

Теперь мы готовы к тестированию представления списка контактов, несмотря на то, что другие компоненты потока пока еще не реализованы. После сборки и развертывания проекта откройте веб-браузер и введите URL `http://localhost:8080/ch18/app/contacts`. Отобразится представление списка контактов (являющееся начальным состоянием потока), которое можно видеть на рис. 18.15.

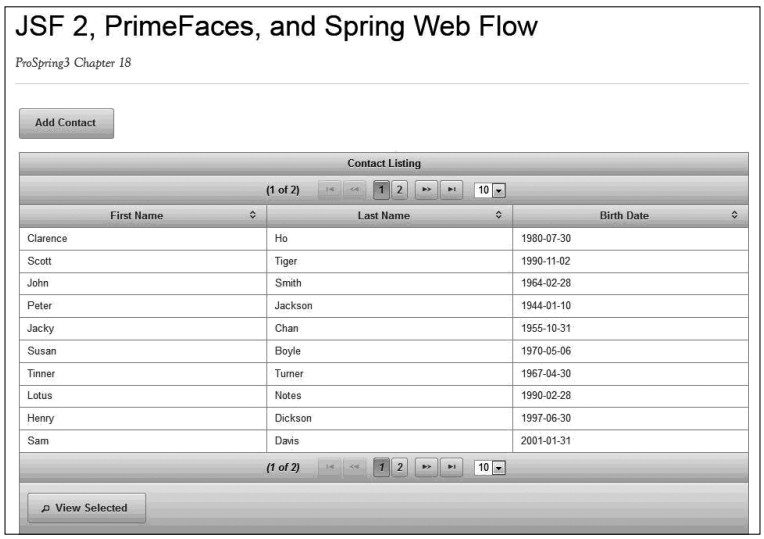


Рис. 18.15. Представление списка контактов

Реализация представления просмотра контакта

Давайте теперь реализуем представление просмотра контакта, которое будет отображаться, когда пользователь выбирает строку и щелкает на кнопке View Selected (Просмотреть выбранный).

Рассмотрим фрагмент кода, приведенный в листинге 18.11, который взят из начального состояния в определении потока, представленном в листинге 18.5 (`flow.xml`).

Листинг 18.11. Навигация из списка контактов в представление просмотра контакта

```
<view-state id="start" view="list.xhtml">
   <on-entry>
      <evaluate expression="contactController.newContactListBean()"
                result="viewScope.contactListBean"/>
   </on-entry>
   <transition on="view" to="show">
      <set name="requestScope.contactId"
           value="contactListBean.selectedContact.id"/>
   </transition>
   <transition on="add" to="add-step-1">
      <evaluate expression="contactController.newContactBean()"
                result="flowScope.contactBean"/>
   </transition>
</view-state>
...
<view-state id="show" view="show.xhtml">
  <on-entry>
    <evaluate expression="contactController.showContact(flowRequestContext)"
              result="viewScope.contact"/>
  </on-entry>
  <transition on ="back" to="start"/>
</view-state>
```

В листинге 18.11 выделен полужирным код навигации в представление просмотра контакта и самого состояния просмотра (`id="show"`). Внутри дескриптора `<transition>` в начальном состоянии (`id="start"`) устанавливается переменная по имени `contactId` с областью видимости на уровне запроса (на основе идентификатора контакта в выбранной строке) и затем происходит переход в состояние просмотра.

В состоянии просмотра вызывается метод `ContactController.showContact()`, которому передается экземпляр интерфейса `org.springframework.webflow.execution.RequestContext`; результат (объект контакта с выбранным идентификатором) сохраняется в переменной по имени `contact` с областью видимости на уровне представления, после чего представление (`show.xhtml`) визуализируется.

В листинге 18.12 показана реализация метода `showContact()`.

Листинг 18.12. Реализация метода `showContact()`

```
package com.apress.prospring3.ch18.web.controller;

// Операторы импорта опущены.
@Component("contactController")
public class ContactController {

   // Остальной код не показан.
```

```
    public Contact showContact(RequestContext context) {
        Long id = context.getRequestScope().getLong("contactId");
        logger.info("Selected contact id: {}", id);
        return contactService.findByIdWithDetail(id);
    }
}
```

В листинге 18.12 вызывается метод `RequestContext.getRequestScope().getLong()` для извлечения переменной `contactId`, после вызывается метод `ContactService.findByIdWithDetail()`, который загрузит контакт вместе с его хобби. Возвращаемый объект `Contact` сохраняется в переменной с областью видимости на уровне представления, чтобы потом использоваться при визуализации представления просмотра контакта.

В листинге 18.13 показано представление просмотра контакта (`webapp/WEB-INF/flows/contacts/show.xhtml`).

Листинг 18.13. Представление просмотра контакта

```
<!DOCTYPE composition PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:p="http://primefaces.org/ui"
    template="/WEB-INF/layouts/standard.xhtml">

<ui:define name="title">#{msg['application_name']}</ui:define>
<ui:define name="content">
<h1>Contact</h1>
<h:form >
    <h3>
        <h:outputText value="#{msg['label_contact_first_name']}: "/>
        <h:outputText value="#{contact.firstName}"/>
        <br/>
        <h:outputText value="#{msg['label_contact_last_name']}: "/>
        <h:outputText value="#{contact.lastName}"/>
        <br/>
        <h:outputText value="#{msg['label_contact_birth_date']}: "/>
        <h:outputText value="#{contact.birthDate}">
            <f:converter converterId="jodaDataTimeConverter" />
        </h:outputText>
    </h3>
    <h3>#{msg['label_contact_hobbies']}</h3>
    <p:dataList value="#{contact.hobbiesAsList}" var="hobby">
        #{hobby.hobbyId}
    </p:dataList>
    <br/>
    <p:toolbar>
        <p:toolbarGroup align="left">
            <p:commandButton value="Back" action="back"/>
        </p:toolbarGroup>
    </p:toolbar>
</h:form>
</ui:define>
</ui:composition>
```

В листинге 18.13 информация о контакте визуализируется на основе свойств, сохраненных в переменной `contact`. Обратите внимание, что для визуализации сведений о хобби вызывается метод `getHobbiesAsList()` объекта `Contact`. Причина в том, что JSF не позволяет визуализировать информацию, хранящуюся в `Set`. В связи с этим был реализован вспомогательный метод `getHobbiesAsList()`, возвращающий хобби в виде `List`.

Чтобы протестировать представление просмотра контакта, выберите какую-нибудь строку и щелкните на кнопке View Selected (Просмотреть выбранный). Отобразится информация о контакте, как показано на рис. 18.16.

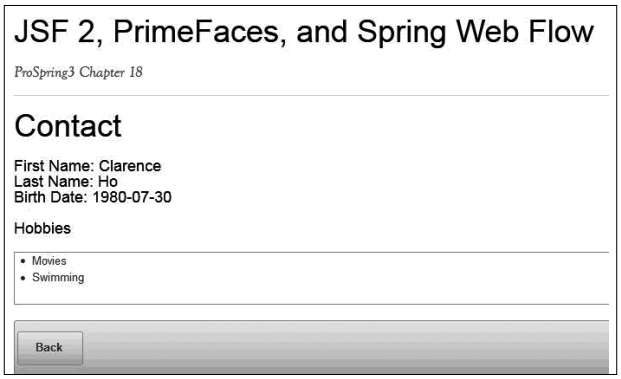


Рис. 18.16. Представление просмотра контакта

Реализация потока добавления контакта

Имея готовые представления списка контактов и просмотра контакта, мы можем переходить к реализации потока добавления контакта. В последующих разделах мы обсудим реализацию представлений в потоке добавления контакта.

Шаг 1: ввод базовой информации

Поток добавления контакта будет запускаться, когда пользователь щелкнет на кнопке Add Contact (Добавить контакт) в представлении списка контактов. Первым состоянием является шаг 1, на котором пользователь будет вводить базовую информацию контакта, включая имя, фамилию и дату рождения.

Рассмотрим фрагмент кода в листинге 18.14, который взят из начального состояния в определении потока, представленном в листинге 18.5 (`flow.xml`).

Листинг 18.14. Переход из списка контактов на шаг 1 потока добавления контакта

```
<view-state id="start" view="list.xhtml">
   <on-entry>
      <evaluate expression="contactController.newContactListBean()"
         result="viewScope.contactListBean"/>
   </on-entry>
   <transition on="view" to="show">
      <set name="requestScope.contactId"
           value="contactListBean.selectedContact.id"/>
   </transition>
   <transition on="add" to="add-step-1">
      <evaluate expression="contactController.newContactBean()"
         result="flowScope.contactBean"/>
   </transition>
```

```
    </view-state>
    <view-state id="add-step-1" view="add-step-1.xhtml">
        <transition on="next" to="add-step-2"/>
    </view-state>
```

Код перехода и состояния `add-step-1` в листинге 18.14 выделен полужирным. Перед переходом вызывается метод `ContactController.newContactBean()`, который создает новый экземпляр класса `ContactBean` и сохраняет его в переменной с областью видимости на уровне потока. Это делает переменную доступной всем шагам в потоке добавления контакта.

В листинге 18.15 приведен класс `ContactBean`, который хранит введенную информацию на протяжении потока и содержит свойства, упрощающие взаимодействие с пользователем за счет применения компонентов PrimeFaces.

Листинг 18.15. Класс `ContactBean`

```
package com.apress.prospring3.ch18.web.view;

import java.io.Serializable;

import org.primefaces.model.DualListModel;

import com.apress.prospring3.ch18.domain.Contact;
import com.apress.prospring3.ch18.domain.Hobby;

public class ContactBean implements Serializable {

    private Contact contact;

    private DualListModel<Hobby> hobbies;

    // Методы извлечения/установки не показаны.
}
```

В листинге 18.15 класс `ContactBean` содержит два свойства. Первое свойство, `contact`, хранит информацию о контакте. Второе свойство, `hobbies`, имеет тип `DualListModel` и используется PrimeFaces-компонентом `PickList` на шаге 2.

В листинге 18.16 представлена реализация метода `newContactBean()` в классе `ContactController`.

Листинг 18.16. Метод `newContactBean()`

```
package com.apress.prospring3.ch18.web.controller;

// Операторы импорта опущены.
@Component("contactController")
public class ContactController {

    // Остальной код не показан.

    @Autowired
    private HobbyService hobbyService;

    public ContactBean newContactBean() {

        logger.info("Creating new contact bean");
        ContactBean contactBean = new ContactBean();

        Contact contact = new Contact();
        contactBean.setContact(contact);

        List<Hobby> hobbiesSource = new ArrayList<Hobby>();
        List<Hobby> hobbiesTarget = new ArrayList<Hobby>();
```

```
        for (Hobby hobby: hobbyService.findAll()) {
            hobbiesSource.add(hobby);
        }
        DualListModel<Hobby> hobbies =
            new DualListModel<Hobby>(hobbiesSource, hobbiesTarget);
        contactBean.setHobbies(hobbies);
        return contactBean;
    }
}
```

В листинге 18.16 метод `newContactBean()` создает экземпляр `ContactBean`; также создается свойство `hobbies`, которое сохраняется в объекте `ContactBean` для использования на шаге 2.

В листинге 18.17 показано представление для шага 1 (`webapp/WEB-INF/flows/contacts/add-step-1.xhtml`).

Листинг 18.17. Представление для шага 1 в потоке добавления контакта

```
<!DOCTYPE composition PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:p="http://primefaces.org/ui"
    template="/WEB-INF/layouts/standard.xhtml">
<ui:define name="title">#{msg['application_name']}</ui:define>
<ui:define name="content">
<h1>#{msg['title_contact_add_step_1']}</h1>
<h:form >
    <p:messages/>
    <h:panelGrid columns="2">
        <h:outputLabel for="firstName">#{msg['label_contact_first_name']}
        </h:outputLabel>
        <p:inputText id="firstName" value="#{contactBean.contact.firstName}"/>
        <h:outputLabel for="lastName">#{msg['label_contact_last_name']}
        </h:outputLabel>
        <p:inputText id="lastName" value="#{contactBean.contact.lastName}"/>
        <h:outputLabel for="birthDate">#{msg['label_contact_birth_date']}
        </h:outputLabel>
        <p:calendar id="birthDate" value="#{contactBean.contact.birthDate}"
                    pattern="yyyy-MM-dd">
            <f:converter converterId="jodaDataTimeConverter" />
        </p:calendar>
    </h:panelGrid>
    <p:toolbar>
        <p:toolbarGroup align="left">
            <p:commandButton value="Next" action="next"
                             execute="@form" update="@form"/>
            <p:commandButton value="Exit" action="exit" immediate="true"/>
        </p:toolbarGroup>
    </p:toolbar>
</h:form>
</ui:define>
</ui:composition>
```

В листинге 18.17 поля предоставляются компонентами ввода PrimeFaces. Обратите внимание на применение дескриптора `<p:calendar>`, который использует PrimeFaces-компонент `Calendar` и поддерживает всплывающее окно для выбора даты.

После сборки и развертывания проекта протестируйте это представление, щелкнув на кнопке Add Contact (Добавить контакт) в представлении списка контактов. Представление для шага 1 будет создано (рис. 18.17).

Рис. 18.17. Представление для шага 1

Поддержка проверки достоверности бинов JSR-303

В JSF 2 имеется встроенная поддержка проверки достоверности бинов JSR-303. Класс `Contact` уже аннотирован с применением ограничений для свойств `firstName` и `lastName` properties. Файл `/src/main/resources/ValidationMessages.properties` хранит сообщения проверки достоверности. Обратите внимание на использование в листинге 18.17 PrimeFaces-компонента `CommandButton`, который запускает обновление формы. Во время процесса обновления проверка достоверности JSR-303 будет запускаться автоматически в течение жизненного цикла JSF-приложения. Чтобы увидеть это в действии, щелкните на кнопке Next (Далее) в представлении для шага 1, ничего не вводя в поля. Представление отобразится заново с отображением сообщений проверки достоверности, как показано на рис. 18.18.

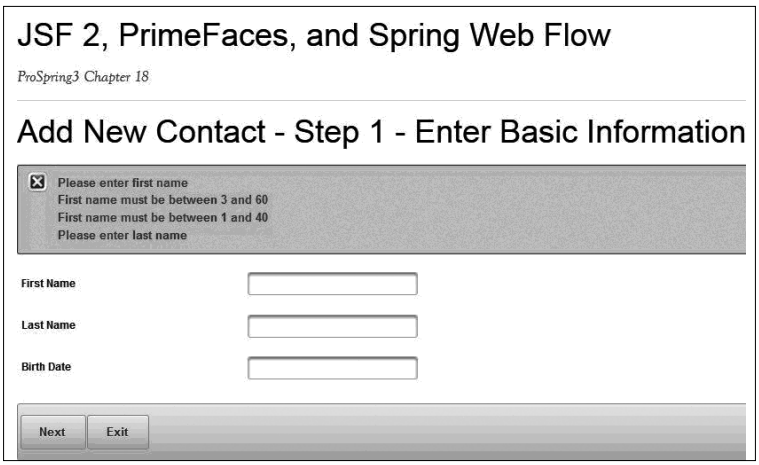


Рис. 18.18. В JSF поддерживается проверка достоверности JSR-303

Шаг 2: выбор хобби

Давайте перейдем к реализации шага 2, на котором производится выбор хобби для контакта. В листинге 18.18 показано представление для шага 2 (`webapp/WEB-INF/flows/contacts/add-step-2.xhtml`).

Листинг 18.18. Представление для шага 2 в потоке добавления контакта

```
<!DOCTYPE composition PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
   xmlns:ui="http://java.sun.com/jsf/facelets"
   xmlns:h="http://java.sun.com/jsf/html"
   xmlns:f="http://java.sun.com/jsf/core"
   xmlns:p="http://primefaces.org/ui"
   template="/WEB-INF/layouts/standard.xhtml">

<ui:define name="title">#{msg['application_name']}</ui:define>

<ui:define name="content">

<h1>#{msg['title_contact_add_step_2']}</h1>

<h:form >

   <h3>

      <h:outputText value="#{msg['label_contact_first_name']}: "/>
      <h:outputText value="#{contactBean.contact.firstName}"/>
      <br/>
      <h:outputText value="#{msg['label_contact_last_name']}: "/>
      <h:outputText value="#{contactBean.contact.lastName}"/>
      <br/>
      <h:outputText value="#{msg['label_contact_birth_date']}: "/>
      <h:outputText value="#{contactBean.contact.birthDate}">
         <f:converter converterId="jodaDataTimeConverter" />
      </h:outputText>

   </h3>

   <p:pickList value="#{contactBean.hobbies}"
            var="hobby"
            itemLabel="#{hobby.hobbyId}"
            itemValue="#{hobby.hobbyId}"
            converter="hobby"/>

   <p:toolbar>
      <p:toolbarGroup align="left">
         <p:commandButton value="Next" action="next"/>
         <p:commandButton value="Back" action="back"/>
         <p:commandButton value="Exit" action="exit" immediate="true"/>
      </p:toolbarGroup>
   </p:toolbar>
</h:form>

</ui:define>

</ui:composition>
```

В листинге 18.18 обратите внимание на использование PrimeFaces-компонента `PickList` для выбора хобби. Компонент `PickList` связан со свойством `hobbies` класса `ContactBean`, которое имеет тип класса `DualListModel`. Доступные хобби хранятся в

свойстве `source`, а выбранные хобби — в свойстве `target`. Для свойства `hobby` также понадобится реализовать специальный преобразователь для трансформаций между классами `String` и `Hobby`. Код класса преобразователя приведен в листинге 18.19.

Листинг 18.19. Класс преобразователя `Hobby`

```
package com.apress.prospring3.ch18.web.converter;

import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.convert.Converter;
import javax.faces.convert.FacesConverter;

import com.apress.prospring3.ch18.domain.Hobby;

@FacesConverter("hobby")
public class HobbyConverter implements Converter {

    @Override
    public Object getAsObject(FacesContext ctx, UIComponent component, String value)
    {
        return new Hobby(value);
    }

    @Override
    public String getAsString(FacesContext ctx, UIComponent component, Object value)
    {
        return (String) value;
    }
}
```

Код в листинге 18.19 должен быть самоочевидным. Чтобы протестировать представление, введите на шаге 1 нужную информацию и щелкните на кнопке Next (Далее). Отобразится представление для шага 2, показанное на рис. 18.19.

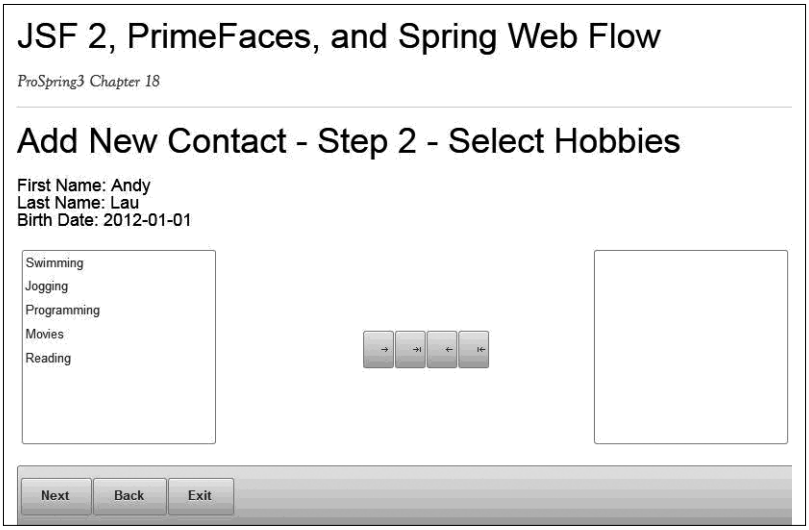


Рис. 18.19. Представление для шага 2

Шаг 3: просмотр введенной информации

Шаг 3 потока добавления контакта связан с просмотром введенной информации, чтобы пользователь мог вернуться и подкорректировать сведения или сохранить информацию о контакте. В листинге 18.20 показано представление для шага 3 (`webapp/WEB-INF/flows/contacts/review.xhtml`).

Листинг 18.20. Представление для шага 3 в потоке добавления контакта

```
<!DOCTYPE composition PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
   xmlns:ui="http://java.sun.com/jsf/facelets"
   xmlns:h="http://java.sun.com/jsf/html"
   xmlns:f="http://java.sun.com/jsf/core"
   xmlns:p="http://primefaces.org/ui"
   template="/WEB-INF/layouts/standard.xhtml">

<ui:define name="title">#{msg['application_name']}</ui:define>

<ui:define name="content">

<h1>#{msg['title_contact_add_review']}</h1>

<h:form >

   <h3>
      <h:outputText value="#{msg['label_contact_first_name']}: "/>
      <h:outputText value="#{contactBean.contact.firstName}"/>
      <br/>
      <h:outputText value="#{msg['label_contact_last_name']}: "/>
      <h:outputText value="#{contactBean.contact.lastName}"/>
      <br/>
      <h:outputText value="#{msg['label_contact_birth_date']}: "/>
      <h:outputText value="#{contactBean.contact.birthDate}">
         <f:converter converterId="jodaDataTimeConverter" />
      </h:outputText>
   </h3>
   <h3>#{msg['label_contact_hobbies']}</h3>
   <p:dataList value="#{contactBean.hobbies.target}" var="hobby">
      #{hobby.hobbyId}
   </p:dataList>
   <br/>

   <p:toolbar>
      <p:toolbarGroup align="left">
         <p:commandButton value="Save" action="next"/>
         <p:commandButton value="Back" action="back"/>
         <p:commandButton value="Exit" action="exit" immediate="true"/>
      </p:toolbarGroup>
   </p:toolbar>
</h:form>
</ui:define>
</ui:composition>
```

В листинге 18.20 отображается информация о контакте. Чтобы протестировать это представление, выберите хобби для контакта и щелкните на кнопке Next (Далее). Отобразится представление для просмотра введенной информации, показанное на рис. 18.20.

Рис. 18.20. Представление для шага 3

Шаг 4: добавление контакта завершено

Рассмотрим фрагмент кода в листинге 18.21, который взят из начального состояния в определении потока, представленном в листинге 18.5 (`flow.xml`).

Листинг 18.21. Переход от просмотра введенной информации на шаг 4 потока добавления контакта

```
<view-state id="review" view="review.xhtml">
   <transition on ="next" to="complete">
     <evaluate expression="contactController.saveContact(flowRequestContext)"/>
   </transition>
     <transition on="back" to="add-step-2"/>
</view-state>

<view-state id="complete" view="complete.xhtml">
   <transition on="continue" to="finish"/>
</view-state>

<end-state id="finish"/>
```

В листинге 18.21 состояние завершения (`id="complete"`) и конечное состояние выделены полужирным. Во время перехода вызывается метод `ContactController.saveContact()`, который сохраняет объект `Contact`, после чего отобразится представление завершения. Реализация метода `saveContact()` приведена в листинге 18.22.

Листинг 18.22. Метод `saveContact()`

```
package com.apress.prospring3.ch18.web.controller;

// Операторы импорта опущены.
@Component("contactController")
public class ContactController {

// Остальной код не показан.

public void saveContact(RequestContext context) {

   // Извлечь бин контакта.
```

```
    ContactBean contactBean =
        (ContactBean)context.getFlowScope().get("contactBean");
    Contact contact = contactBean.getContact();

    Set<Hobby> hobbies = new HashSet<Hobby>(contactBean.getHobbies().getTarget());

    contact.setHobbies(hobbies);
    contactService.save(contact);
    logger.info("Contact {} {} saved successfully", contact.getFirstName(),
       contact.getLastName());
    context.getMessageContext().addMessage(new
       MessageBuilder().info().code("message_contact_save_success").build());
    }
}
```

В методе `saveContact()` из `RequestContext` в Spring Web Flow извлекается переменная `contactBean` с областью действия на уровне потока, получается свойство `contact`, заполняется свойство `hobbies` и затем сохраняется объект `Contact`. После этого конструируется сообщение об успешном сохранении, которое помещается в `MessageContext`, и представление завершения визуализируется. В конфигурации Spring MVC, приведенной в листинге 18.4, мы определили бин `messageSource` с указанными путями файлов сообщений. Давайте создадим папку по имени `i18n` внутри папки `WEB-INF`, а в ней — файл `messages.properties` для хранения сообщения об успешном сохранении. В этот файл понадобится только добавить следующую строку:

```
message_contact_save_success=Contact saved successfully
```

Представление завершения показано в листинге 18.23 (`webapp/WEB-INF/flows/contacts/complete.xhtml`).

Листинг 18.23. Представление завершения

```
<!DOCTYPE composition PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
   xmlns:ui="http://java.sun.com/jsf/facelets"
   xmlns:h="http://java.sun.com/jsf/html"
   xmlns:f="http://java.sun.com/jsf/core"
   xmlns:p="http://primefaces.org/ui"
   template="/WEB-INF/layouts/standard.xhtml">
<ui:define name="title">#{msg['application_name']}</ui:define>
<ui:define name="content">
<h1>#{msg['title_contact_add_complete']}</h1>
<p:messages/>
<h:form >
   <h3>
      <h:outputText value="#{msg['label_contact_first_name']}: "/>
      <h:outputText value="#{contactBean.contact.firstName}"/>
      <br/>
      <h:outputText value="#{msg['label_contact_last_name']}: "/>
      <h:outputText value="#{contactBean.contact.lastName}"/>
      <br/>
      <h:outputText value="#{msg['label_contact_birth_date']}: "/>
      <h:outputText value="#{contactBean.contact.birthDate}">
         <f:converter converterId="jodaDataTimeConverter" />
      </h:outputText>
   </h3>
```

```
   <h3>#{msg['label_contact_hobbies']}</h3>
   <p:dataList value="#{contactBean.hobbies.target}" var="hobby">
      #{hobby.hobbyId}
   </p:dataList>
   <br/>

   <p:toolbar>
      <p:toolbarGroup align="left">
         <p:commandButton value="Continue" action="continue"/>
      </p:toolbarGroup>
   </p:toolbar>
</h:form>
</ui:define>
</ui:composition>
```

Итак, работа над потоком добавления контакта подошла к концу. После сборки и развертывания проекта можно протестировать весь поток. Щелкайте на кнопках Next (Далее) и Back (Назад) в любом состоянии и наблюдайте за поведением потока.

На рис. 18.21 показано, как выглядит поток после успешного сохранения контакта.

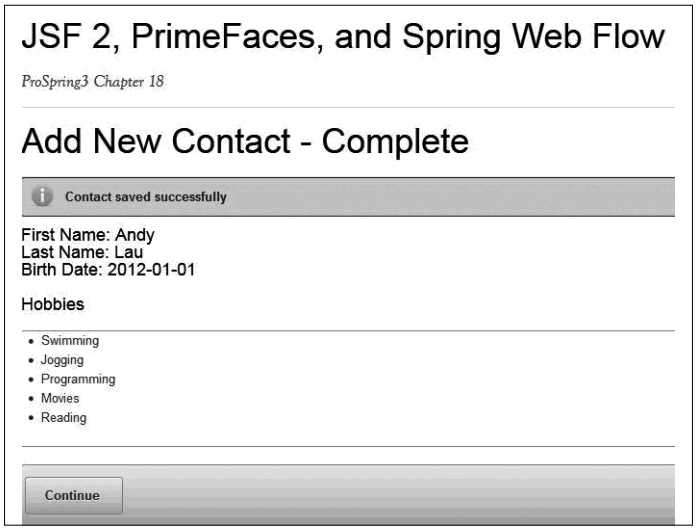


Рис. 18.21. Завершающее представление потока для добавления контакта

Резюме

В этой главе мы продемонстрировали использование Spring Web Flow и JSF 2 при реализации потокового веб-приложения.

Сначала были рассмотрены высокоуровневые концепции Spring Web Flow и JSF 2. Затем была обсуждена первоклассная поддержка JSF в Spring Web Flow, а также все обязательные зависимости и конфигурации.

После этого мы спроектировали пример потока для добавления нового контакта, в том числе переходы между состояниями, представления, переменные и т.п.

Наконец, мы прошли по шагам, необходимым для реализации потока. Эти шаги включали разработку определения потока, класса контроллера, поддерживающих бинов, специальных преобразователей, представлений JSF и компонентов PrimeFaces.

ГЛАВА 19

Тестирование в Spring

При разработке корпоративных приложений тестирование является важным способом обеспечения того, что завершенное приложение выполняется ожидаемым образом и удовлетворяет всем видам требований (архитектурным, пользовательским и т.д.).

Каждый раз, когда вносится изменение, вы должны удостовериться, что это изменение не влияет на существующую логику. Поддержка среды сборки и тестирования критически важна для обеспечения высокого качества приложений. Воспроизводимые тесты с большой степенью покрытия кода позволяют развертывать новые приложения и изменения к ним с высоким уровнем доверия.

В среде разработки корпоративных приложений существует много разновидностей тестирования, нацеленных на каждый уровень приложения, и каждый вид тестирования обладает собственными характеристиками и требованиями.

В этой главе мы обсудим базовые концепции, связанные с тестированием различных уровней приложения, особенно при тестировании Spring-приложений. Мы также рассмотрим пути, при которых Spring упрощает дл разработчиков реализацию тестовых сценариев. В частности, в главе рассматриваются следующие темы.

- *Корпоративная инфраструктура тестирования.* Мы кратко опишем корпоративную инфраструктуру тестирования. Мы обсудим различные виды тестирования и их назначение. В этой главе мы сосредоточимся на модульном тестировании, ориентированном на разные уровни приложения.
- *Модульное тестирование логики.* Самый ясный модульный тест предполагает проверку только логики методов внутри какого-то класса, а для всех остальных зависимостей "имитируется" корректное поведение. В этой главе мы обсудим реализацию модульного тестирования логики для классов контроллеров Spring MVC с применением Java-библиотеки имитации для имитирования корректного поведения зависимостей классов.
- *Модульное тестирование взаимодействия.* В корпоративной инфраструктуре тестирования проверка взаимодействия осуществляется для группы классов внутри различных уровней приложения для конкретной порции бизнес-логики. Обычно в среде тестирования взаимодействия уровень обслуживания должен проверяться вместе с уровнем постоянства, с доступной базой данных серверной части. Однако с развитием архитектуры приложения и обретения зрелости облегченных баз данных в памяти, теперь общей практикой становится "модульное тестирование" уровня обслуживания с уровнем постоянства и базой данных серверной части как единого целого. Например, в этой главе мы будем использовать JPA 2 с Hibernate и Spring Data JPA в качестве поставщика постоянства и H2 в качестве базы данных. В такой архитектуре менее важно "имитировать" Hibernate и Spring Data JPA при тестировании уровня обслуживания. В результате мы здесь будем обсуждать тестирование уровня обслуживания вместе с уровнем постоянства и

базой данных H2 в памяти. Такой вид тестирования, в общем, называется *модульным тестированием взаимодействия*, которое находится посредине между модульным тестированием и полноценным тестированием взаимодействия.

- *Модульное тестирование интерфейсной части.* Даже если мы тестируем каждый уровень приложения, после его развертывания мы по-прежнему должны удостовериться, что все приложение функционирует так, как ожидалось. В частности, в случае веб-приложения, после его развертывания в среде непрерывной сборки мы должны прогнать тестирование интерфейсной части, чтобы проверить, корректно ли работает пользовательский интерфейс. Например, для приложения контактов нужно удостовериться, что каждый шаг в рамках обычной функциональности работает правильно, а также протестировать исключительные случаи (скажем, как приложение функционирует, когда введенная пользователем информация не проходит фазу проверки достоверности). В этой главе мы обсудим применение инфраструктур с открытым кодом для автоматизации тестирования интерфейсной части веб-приложения.

Проект для примера веб-приложения

При написании тестовых сценариев в этой главе мы будем использовать разработанное в главе 17 веб-приложение контактов в качестве целевого тестируемого приложения.

Импорт проекта примера серверной части в STS

Поскольку данная глава ориентирована на разработку тестовых сценариев, мы подготовили проект примера веб-приложения, чтобы его можно было просто импортировать и заняться реализацией собственно тестовых сценариев.

Извлеките файл `ch19-notest.zip` из кода примеров в рабочее пространство STS. После этого импортируйте проект в STS.

Чтобы импортировать этот проект, в среде STS выберите пункт меню File⇨Import (Файл⇨Импорт) и на экране Select (Выбор) укажите вариант Existing Projects into Workspace (Существующие проекты в рабочем пространстве) в категории General (Общие), как показано на рис. 19.1. Щелкните на кнопке Next (Далее).

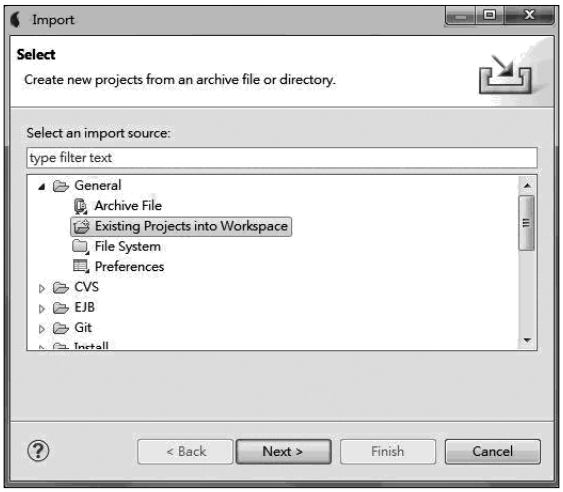


Рис. 19.1. Импорт существующего проекта в STS

После этого на экране Import Projects (Импорт проектов) выберите проект, извлеченный из кода примеров, и щелкните на кнопке Finish (Готово), как показано на рис. 19.2.

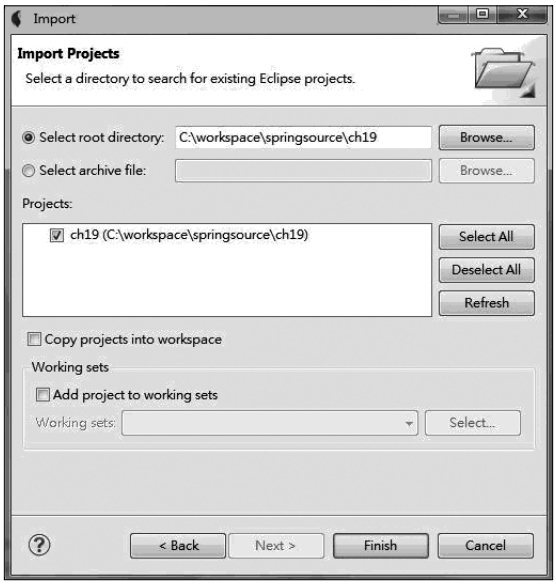


Рис. 19.2. Импорт проекта примера

По завершении импорта проекта выполните в STS сборку и развертывание проекта на сервере tc Server. После того, как проект опубликован, и сервер tc Server запущен, протестируйте пример веб-приложения контактов, посетив URL `http://localhost:8080/ch19/contacts` в веб-браузере (рис. 19.3).

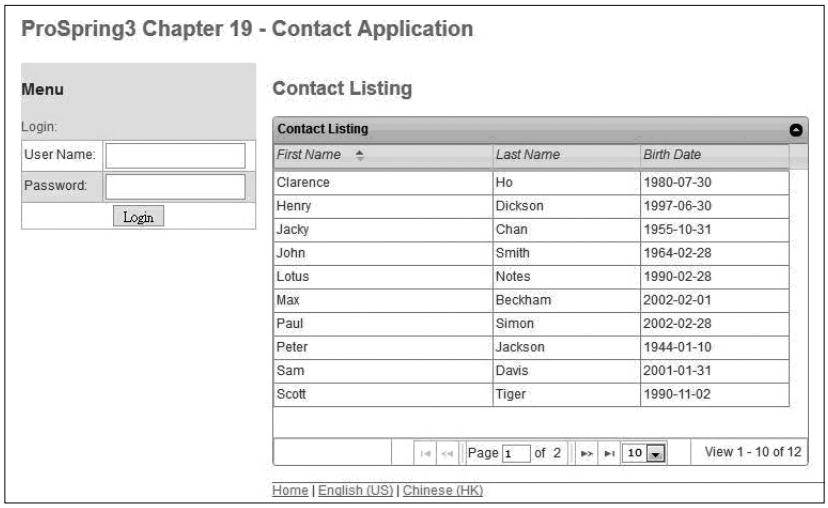


Рис. 19.3. Пример веб-приложения контактов

На этом установка примера веб-приложения завершена, и можно приступать к реализации тестовых сценариев.

Введение в корпоративную инфраструктуру тестирования

Под *корпоративной инфраструктурой тестирования* понимаются действия по тестированию в полном жизненном цикле приложения. На различных фазах будут выполняться разные действия тестирования, цель которых заключается в проверке, работает ли функциональность приложения ожидаемым образом в соответствии с определенными бизнес-правилами и техническими требованиями.

На каждой фазе будут прогоняться разные тестовые сценарии. Некоторые из них будут автоматизированы, тогда как другие — выполняться вручную. В каждом сценарии полученные результаты будут проверяться соответствующим персоналом (например, бизнес-аналитиками, пользователями приложения и т.д.).

Типичная корпоративная инфраструктура тестирования показана на рис. 19.4.

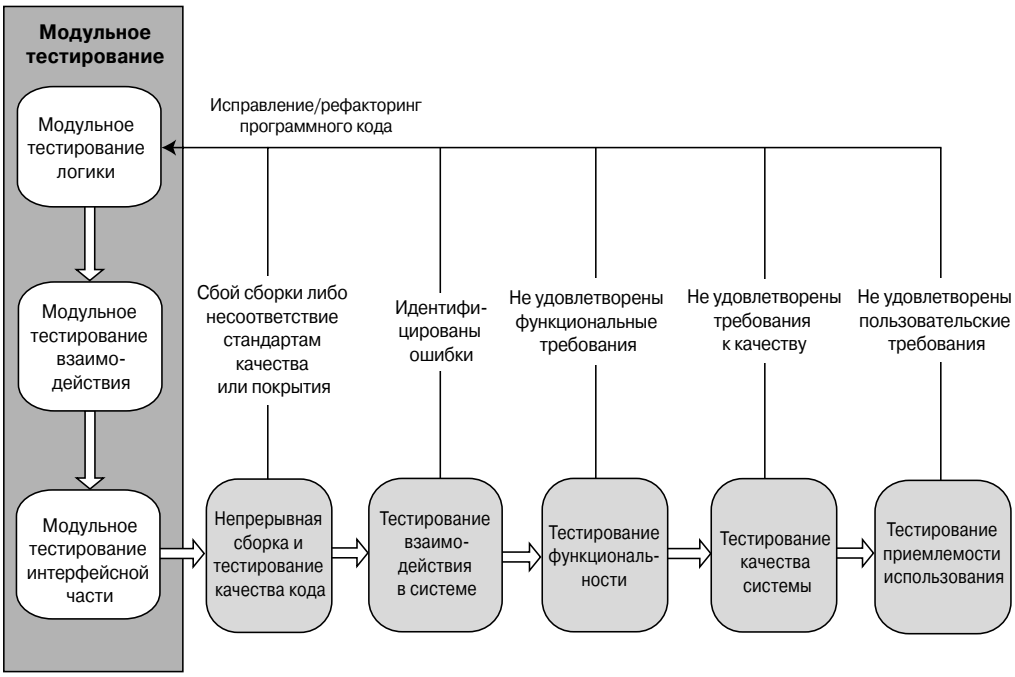


Рис. 19.4. Корпоративная инфраструктура тестирования

В табл. 19.1 описаны характеристики и цели каждого типа тестирования, а также общие инструменты и библиотеки, используемые для реализации тестовых сценариев.

В этой главе мы сосредоточим внимание на реализации трех видов модульного тестирования (модульное тестирование логики, модульное тестирование взаимодействия и модульное тестирование интерфейсной части) и покажем, как инфраструктура `TestContext` в Spring и другие поддерживающие инструменты помогают в разработке тестовых сценариев.

Вместо представления детальных сведений и списка классов, которые предлагаются в Spring Framework для области тестирования, мы обсудим применение наиболее часто используемых шаблонов, а также поддерживающих интерфейсом и классов внутри инфраструктуры `TestContext`, реализуя примеры тестовых сценариев в этой главе.

Таблица 19.1. Описание корпоративной инфраструктуры тестирования

Категория тестирования	Описание	Общие инструменты
Модульное тестирование логики	Модульное тестирование логики применяется к одиночному объекту, не принимая во внимание его роль в окружающей системе	Модульное тестирование: JUnit, TestNG Объекты-имитации: Mockito, EasyMock
Модульное тестирование взаимодействия	Модульное тестирование взаимодействия нацелено на проверку взаимодействия между компонентами в среде, “близкой к реальной”. Такие тесты будут осуществлять взаимодействия с контейнером (встроенной базой данных, веб-контейнером и т.д.)	Встроенная база данных: H2 Тестирование баз данных: DBUnit Веб-контейнер внутри памяти: Jetty
Модульное тестирование интерфейсной части	Модульное тестирование интерфейсной части нацелено на проверку пользовательского интерфейса. Его целью является обеспечение того, что каждый пользовательский интерфейс реагирует на действия пользователей и генерирует ожидаемый вывод	Selenium
Непрерывная сборка и тестирование качества	Кодовая база приложения должна собираться на регулярной основе, чтобы обеспечивать соответствие качества кода существующему стандарту (например, наличие комментариев, отсутствие пустых блоков перехвата исключений и т.п.). Кроме того, степень покрытия тестами должна быть как можно более высокой, чтобы гарантировать проверку всех строк разработанного кода	Качество кода: PMD, CheckStyle, CodePro Analytix Покрытие тестами: Cobertura Инструменты сборки: Gradle, Maven, Sonar Непрерывная сборка: Hudson, Jenkins
Тестирование взаимодействия в системе	Тестирование взаимодействия в системе проверяет правильность взаимодействия между всеми программами в новой системе и между новой системой и всеми внешними интерфейсами. Тестирование взаимодействия должно также доказать, что новая система работает в соответствии с функциональными спецификациями и эффективно функционирует в операционной среде, не оказывая отрицательного влияния на другие системы	IBM Rational Systems Tester
Тестирование функциональности	Тестирование функциональности проверяет сценарии использования и бизнес-правила. Цель таких тестов — удостовериться в том, что ввод правильно принимается, а вывод правильно генерируется, где “правильно” означает соответствие и спецификациям сценариев использования, и бизнес-правилам. Это тестирование “черного ящика”, осуществляемое путем взаимодействия с приложением через графический пользовательский интерфейс и последующего анализа результатов	IBM Rational Functional Tester, HP Quick Test Professional

Окончание табл. 19.1

Категория тестирования	Описание	Общие инструменты
Тестирование качества системы	Тестирование качества системы нацелено на проверку того, что разработанное приложение удовлетворяет ряду нефункциональных требований. По большей части при этом выполняется тестирование производительности приложения, чтобы удостовериться в удовлетворении требований относительно параллельно работающих пользователей в системе и рабочей нагрузки. Другие нефункциональные требования включают безопасность, высокую доступность и т.п.	Apache JMeter, HP LoadRunner
Тестирование приемлемости использования	Тестирование приемлемости использования эмулирует действительные рабочие условия новой системы, в том числе руководства пользователя и процедуры работы в системе. Широкое привлечение пользователей на этом этапе тестирования предоставит им неоценимый опыт работы ней. Это также позволит программистам и проектировщикам оценить эффективность новых программ. Такое совместное участие способствует более быстрому переходу пользователей и технического персонала на новую систему	IBM Rational TestManager, HP Quality Center

Реализация модульного тестирования логики

Как упоминалось ранее, модульное тестирование логики является самым ясным уровнем тестирования. Его цель заключается в проверке поведения отдельных классов, тогда как для всех зависимостей этих классов имитируется ожидаемое поведение.

В этом разделе мы продемонстрируем применение модульного тестирования логики, реализовав тестовые сценарии для класса `ContactController`, с имитацией ожидаемого поведения для уровня обслуживания.

Для имитации ожидаемого поведения уровня обслуживания мы будем использовать Mockito (`http://code.google.com/p/mockito`) — популярную инфраструктуру имитации.

Добавление обязательных зависимостей

Вначале необходимо добавить в проект зависимость для Mockito, которая описана в табл. 19.2.

Таблица 19.2. Зависимость Maven для Mockito

Идентификатор группы	Идентификатор артефакта	Версия	Описание
`org.mockito`	`mockito-core`	1.9.0	Библиотека ядра инфраструктуры имитации Mockito

Добавьте эту зависимость к проекту в STS.

Модульное тестирование контроллера Spring MVC

В веб-приложении контактов, реализованном в главе 17, в качестве уровня презентаций использовался модуль Spring MVC. На уровне презентаций классы контроллеров обеспечивают интеграцию пользовательского интерфейса и уровня обслуживания.

Методы в классах контроллеров отображаются на HTTP-запросы. Внутри такого метода запрос обрабатывается, привязывается к объектам модели и взаимодействует с уровнем обслуживания (внедренным в классы контроллеров с помощью DI из Spring) для обработки данных. В зависимости от результатов класс контроллера обновляет модуль и состояние представления (например, пользовательские сообщения и т.п.) и возвращает логическое представление (или модель вместе с представлением) модулю Spring MVC для распознавания конкретного представления с целью его отображения пользователю.

Основная цель проведения модульного тестирования для классов контроллеров заключается в проверке, что методы контроллеров правильно обновляют модель и состояния представлений и возвращают корректные представления. Кроме того, в случае возникновения ошибки проверяется факт генерации правильного исключения и сохранения в состоянии сообщения об ошибке, предназначенного для отображения пользователю. Поскольку мы хотим протестировать только поведение классов контроллеров, то должны имитировать корректное поведение для уровня обслуживания.

В случае класса `ContactController` из примера приложения мы собираемся разработать тестовые сценарии для методов `list()` и `create()`. В последующих разделах мы обсудим необходимые шаги, включая разработку ряда общих классов инфраструктуры, которые поддерживают тестирование классов контроллеров, и реализацию тестовых сценариев.

Реализация классов инфраструктуры

Для группы общих тестов (например, тестовых сценариев для классов контроллеров, классов для тестирования уровня обслуживания и т.д.) всегда лучше разрабатывать общий абстрактный родительский класс, который должным образом настраивает обязательную тестовую инфраструктуру.

В этой главе для конфигурирования `TestContext` в Spring мы будем использовать Java-классы. Вдобавок компоненты, специфичные для тестовой среды, будут конфигурироваться с применением средства профилей Spring 3.1.

Давайте реализуем Java-класс для `ApplicationContext` в Spring (класс `ControllerTestConfig`); код этого приведен в листинге 19.1. Обратите внимание, что тестовые классы и файлы ресурсов должны быть помещены в папки `/src/test/java` и `/src/test/resources` соответственно.

Листинг 19.1. Класс `ControllerTestConfig`

```
package com.apress.prospring3.ch19.test.config;

import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;

@Configuration
@Profile("test")
public class ControllerTestConfig {

}
```

В листинге 19.1 мы указываем Spring, что это класс конфигурации `ApplicationContext`, применяя к классу аннотацию `@Configuration`. Затем к классу применяется аннота-

ция `@Profile`, указывающая профиль (в данном случае профиль `test`), к которому принадлежат бины, сконфигурированные в этом классе.

В данный момент никакие бины не нужны. Тем не менее, всегда рекомендуется создавать класс конфигурации, чтобы если в будущем возникнет такая потребность (например, уровень Spring MVC должен взаимодействовать с внешним FTP-сервером), то в этом классе конфигурации можно будет определить бин имитации для FTP-сервера.

Следующий шаг связан с реализацией абстрактного базового класса для тестовых сценариев классов контроллеров (по имени `AbstractControllerTest`), который показан в листинге 19.2.

Листинг 19.2. Класс `AbstractControllerTest`

```
package com.apress.prospring3.ch19.web.controller;

import org.junit.runner.RunWith;
import org.springframework.test.context.ActiveProfiles;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import com.apress.prospring3.ch19.test.config.ControllerTestConfig;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {ControllerTestConfig.class})
@ActiveProfiles("test")
public class AbstractControllerTest {
}
```

В листинге 19.2 мы применяем к абстрактному базовому классу несколько аннотаций. Во-первых, аннотация `@RunWith`, принадлежащая библиотеке JUnit (в этой главе мы работаем с версией JUnit 4.10), задает класс запуска, используемый для выполнения тестового сценария. Внутри этой аннотации указывается `SpringJUnit4ClassRunner` — Spring-класс поддержки JUnit для запуска тестовых сценариев в среде `ApplicationContext`.

Во-вторых, аннотация `@ContextConfiguration` указывает конфигурацию для загрузки Spring JUnit. Внутри аннотации с помощью атрибута `classes` задается Java-класс, определяющий конфигурацию. Предоставляя атрибут `locations`, контекст можно загрузить из XML-файлов, однако не допускается указывать оба атрибута, `locations` и `classes`, одновременно.

Наконец, в-третьих, применяется аннотация `@ActiveProfiles`, которой в качестве атрибута передается имя профиля `test`. Это указывает Spring, что должны быть загружены бины, принадлежащие профилю `test`.

Тестирование метода `list()`

Имея классы инфраструктуры, давайте создадим первый тестовый сценарий для метода `ContactController.list()`. В этом тестовом сценарии мы хотим удостовериться, что когда этот метод вызван, после извлечения списка контактов из уровня обслуживания информация правильно сохраняется в модель и возвращается корректное логическое представление. Тестовый сценарий показан в листинге 19.3.

Листинг 19.3. Тестирование метода `list()`

```
package com.apress.prospring3.ch19.web.controller;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotNull;
import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.when;
```

```
import java.util.ArrayList;
import java.util.List;

import org.junit.Before;
import org.junit.Test;
import org.springframework.test.util.ReflectionTestUtils;
import org.springframework.ui.ExtendedModelMap;

import com.apress.prospring3.ch19.domain.Contact;
import com.apress.prospring3.ch19.service.ContactService;

public class ContactControllerTest extends AbstractControllerTest {

   private final List<Contact> contacts = new ArrayList<Contact>();

   private ContactService contactService;

   @Before
   public void initContacts() {
      // Инициализировать список контактов для имитации ContactService.
      Contact contact = new Contact();
      contact.setId(1l);
      contact.setFirstName("Clarence");
      contact.setLastName("Ho");
      contacts.add(contact);
   }

   @Test
   public void testList() throws Exception {

      contactService = mock(ContactService.class);
      when(contactService.findAll()).thenReturn(contacts);

      ContactController contactController = new ContactController();

      ReflectionTestUtils.setField(contactController, "contactService",
         contactService);

      ExtendedModelMap uiModel = new ExtendedModelMap();

      String result = contactController.list(uiModel);

      assertNotNull(result);
      assertEquals(result, "contacts/list");

      List<Contact> modelContacts = (List<Contact>) uiModel.get("contacts");

      assertEquals(1, modelContacts.size());
   }
}
```

Во-первых, в листинге 19.3 класс расширяет `AbstractControllerTest`, который наследует всю требуемую инфраструктуру для тестирования класса контроллера. Затем в тестовом сценарии метод `initContacts()` применяется с помощью аннотации `@Before`, которая указывает JUnit, что этот метод должен запускаться перед запуском каждого тестового сценария (в случае, если нужно выполнять некоторую логику перед самим классом теста, используйте аннотацию `@BeforeClass`). В методе `initContacts()` список контактов инициализируется жестко закодированными значениями.

Во-вторых, метод `testList()` применяется посредством аннотации `@Test`, которая указывает JUnit, что это тестовый сценарий, который должен быть запущен. Внутри тестового сценария закрытая переменная `contactService` (типа `ContactService`) имитируется с использованием метода `Mockito.mock()` из Mockito (обратите внимание на оператор `import static`). Инфраструктура Mockito также предоставляет метод

`when()` для имитации метода `ContactService.findAll()`, который будет применяться классом `ContactController`.

В-третьих, создается экземпляр `ContactController`, и его переменная `contactService`, внедряемая Spring в нормальных ситуациях, устанавливается в имитированный экземпляр с использованием метода `setField()` класса `ReflectionTestUtils`, предоставляемого Spring.

Класс `ReflectionTestUtils` предлагает набор служебных методов, основанных на рефлексии, которые предназначены для применения в сценариях модульного тестирования и тестирования взаимодействия. Кроме того, конструируется экземпляр класса `ExtendedModelMap` (реализующего интерфейс `org.springframework.ui.Model`).

В-четвертых, вызывается метод `ContactController.list()` с передачей ему экземпляра класса `ExtendedModelMap`. После вызова результат проверяется с помощью различных методов утверждений (предоставляемых JUnit), чтобы удостовериться в возврате правильного логического представления и в корректном сохранении списка контактов в модели для использования представлением.

Теперь можно запустить тестовый сценарий. Чтобы запустить его в STS, щелкните правой кнопкой мыши на имени класса и выберите в контекстном меню пункт Run As⇨ JUnit Test (Запустить как⇨Тест JUnit). Тестовый сценарий должен выполниться успешно, и это можно проверить на вкладке JUnit, показанной на рис. 19.5.

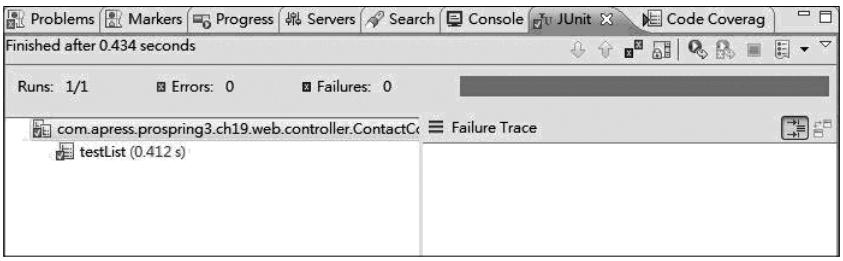


Рис. 19.5. Вкладка JUnit

Давайте перейдем к тестированию метода `create()`.

Тестирование метода `create()`

В листинге 19.4 приведен фрагмент кода, предназначенный для тестирования метода `create()`.

Листинг 19.4. Тестирование метода `create()`

```
package com.apress.prospring3.ch19.web.controller;

import org.springframework.context.MessageSource;
import org.springframework.mock.web.MockHttpServletRequest;

import org.mockito.stubbing.Answer;
import org.mockito.invocation.InvocationOnMock;

// Другие операторы импорта опущены.
public class ContactControllerTest extends AbstractControllerTest {

    // Остальной код не показан.

    private MessageSource messageSource;

    @Test
```

```
    public void testCreate() {
        final Contact newContact = new Contact();
        newContact.setId(999l);
        newContact.setFirstName("Rod");
        newContact.setFirstName("Johnson");

        contactService = mock(ContactService.class);
        when(contactService.save(newContact)).thenAnswer(new Answer<Contact>() {
          public Contact answer(InvocationOnMock invocation) throws Throwable {
               contacts.add(newContact);
               return newContact;
          }
        });

        messageSource = mock(MessageSource.class);
        when(messageSource.getMessage("contact_save_success", new Object[]{},
           Locale.ENGLISH)).thenReturn("success");
        ContactController contactController = new ContactController();
        ReflectionTestUtils.setField(contactController, "contactService",
           contactService);
        ReflectionTestUtils.setField(contactController, "messageSource",
           messageSource);

        BindingResult bindingResult = new BeanPropertyBindingResult(
           newContact, "contact");
        ExtendedModelMap uiModel = new ExtendedModelMap();
        HttpServletRequest httpServletRequest = new MockHttpServletRequest();
        RedirectAttributes redirectAttributes =
           new RedirectAttributesModelMap();
        Locale locale = Locale.ENGLISH;

        String result = contactController.create(newContact, bindingResult,
           uiModel, httpServletRequest, redirectAttributes, locale, null);

        assertNotNull(result);
        assertEquals("redirect:/contacts/999", result);
        assertEquals(2, contacts.size());
    }
}
```

В листинге 19.4 метод `ContactService.save()` имитируется для эмуляции добавления нового объекта `Contact` в список контактов. Обратите внимание на использование интерфейса `org.mockito.stubbing.Answer<T>`, который обеспечивает имитацию метода ожидаемой логикой и возвращает значение.

Кроме того, с помощью Mockito осуществляется имитация экземпляра интерфейса `MessageSource`. Затем конструируется экземпляр класса `ContactController`, и для его зависимостей `ContactService` и `MessageSource` устанавливаются имитированные реализации.

Кроме `ExtendedModelMap`, конструируются также экземпляры `RedirectAttributes` и `HttpServletRequest`. Относительно интерфейса `HttpServletRequest` обратите внимание на использование класса `MockHttpServletRequest`, который является имитированной реализацией Spring. В пакете `org.springframework.mock.web` имеется также несколько классов-имитаций, помогающих в реализации модульных тестов. Для интерфейса `BindingResult` конструируется экземпляр класса `BeanPropertyBindingResult`, который выполняет регистрацию и проверку ошибок привязки для объектов JavaBean, с использованием стандартного доступа к свойствам JavaBean.

Наконец, метод `ContactController.create()` вызывается, после чего с помощью операций утверждений проверяется результат. Запустите тестовый сценарий снова и просмотрите результат на вкладке JUnit.

Для метода `create()` мы должны создать дополнительные тестовые сценарии для проверки разнообразных ситуаций. Например, нужно протестировать случай, когда во время операции сохранения возникают ошибки доступа к данным.

Реализация модульного тестирования взаимодействия

В этом разделе мы реализуем модульное тестирование взаимодействия для уровня обслуживания. Основной службой в приложении контактов является класс `com.apress.prospring3.ch19.service.jpa.ContactServiceImpl`, который представляет собой JPA-реализацию интерфейса `com.apress.prospring3.ch19.service.ContactService`.

При выполнении модульного тестирования уровня обслуживания мы будем использовать для хранения модели и тестовой информации базу данных в памяти H2 с поставщиками JPA (Hibernate и абстракция репозитория Spring Data JPA). Цель — удостовериться, что класс `ContactServiceImpl` корректно выполняет свои бизнес-функции.

В последующих разделах мы покажем, как протестировать некоторые методы поиска и операцию сохранения в классе `ContactServiceImpl`.

Добавление обязательных зависимостей

Для реализации тестовых сценариев с базой данных нам нужна библиотека, которая поможет заполнить базу данных желаемой тестовой информацией перед запуском тестового сценария и затем легко выполнить необходимые операции базы данных. Кроме того, для упрощения подготовки тестовых данных мы будем использовать формат Microsoft Excel.

Чтобы удовлетворить указанным целям, понадобятся дополнительные библиотеки. Для стороны базы данных общей библиотекой, которая помогает реализовать тестирование, связанное с базой данных, является DBUnit (`http://dbunit.sourceforge.net`). Вдобавок будет применяться библиотека проекта Apache POI (`http://poi.apache.org`), с помощью которой будет производиться разбор тестовых данных, подготовленных в формате Microsoft Excel. В табл. 19.3 описаны обязательные зависимости.

Таблица 19.3. Зависимости Maven для модульного тестирования взаимодействия

Идентификатор группы	Идентификатор артефакта	Версия	Описание
`org.dbunit`	`dbunit`	2.4.8	Библиотека DBUnit
`org.apache.poi`	`poi`	3.2-FINAL	Библиотека Apache POI, которая поддерживает чтение и запись файлов в формате Microsoft Office

Добавьте зависимости, перечисленные в табл. 19.3, в свой проект.

Конфигурирование профиля для тестирования уровня обслуживания

Средство профилей для определений бинов, появившееся в версии Spring 3.1, очень удобно для реализации тестового сценария с соответствующей конфигурацией тестируемых компонентов. Чтобы упростить тестирование уровня обслуживания, средство профилей мы также будем использовать для конфигурации `ApplicationContext`.

Для приложения контактов нам понадобятся два профиля.

- *Профиль разработки (`dev`).* Профиль с конфигурацией для среды разработки. Например, в среде разработки для базы данных серверной части H2 будет выполняться как сценарий создания, так и сценарий наполнения начальными данными.
- *Профиль тестирования (`test`).* Профиль с конфигурацией для среды тестирования. Например, в среде тестирования для базы данных серверной части H2 будет выполняться только сценарий создания, а наполнение данными будет осуществляться тестовым сценарием.

Давайте сконфигурируем среду профилей для приложения контактов. В этом приложении конфигурация серверной части (т.е. источник данных, JPA, транзакция и т.д.) была определена в XML-файле конфигурации `/src/main/resources/datasource-tx-jpa.xml`. Мы хотели бы конфигурировать источник данных в этом файле только для профиля `dev`. Чтобы сделать это, понадобится поместить бин источника данных в конфигурацию профиля. В листинге 19.5 приведен фрагмент кода, отражающий необходимые изменения.

Листинг 19.5. Конфигурирование профиля для источника данных

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"

   <!-- Остальной код не показан -->

   <beans profile="dev">
      <jdbc:embedded-database id="dataSource" type="H2">
         <jdbc:script location="classpath:schema.sql"/>
         <jdbc:script location="classpath:test-data.sql"/>
      </jdbc:embedded-database>
   </beans>
</beans>
```

Внесенное изменение выделено полужирным. Как показано в этом фрагменте кода, бин `dataSource` помещен внутрь дескриптора `<beans>` с атрибутом `profile`, имеющим значение `dev`, которое указывает, что источник данных применим только для системы разработки.

Для загрузки веб-приложения с профилем разработки мы добавляем параметр в дескриптор веб-развертывания (другой способ предусматривает модификацию сценария запуска веб-контейнера). Соответствующий фрагмент кода представлен в листинге 19.6.

Листинг 19.6. Конфигурирование профиля в `web.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
```

```
    http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">

    <!-- Определение корневого контейнера Spring,
        разделяемого всеми сервлетами и фильтрами -->
    <context-param>
       <param-name>contextConfigLocation</param-name>
       <param-value>/WEB-INF/spring/root-context.xml</param-value>
    </context-param>
    <context-param>
       <param-name>spring.profiles.active</param-name>
       <param-value>dev</param-value>
    </context-param>

    <!-- Остальной код не показан -->

</web-app>
```

Код нового параметра выделен полужирным. Параметру `spring.profiles.active` присвоено значение `dev`, которое указывает Spring на необходимость начальной загрузки `WebApplicationContext` с бинами, определенными для профиля `dev`.

Реализация классов инфраструктуры

Перед реализацией индивидуального тестового сценария мы должны реализовать классы инфраструктуры, включая Java-класс конфигурации и базовый класс для тестирования уровня обслуживания. Кроме того, также понадобится реализовать ряд классов для поддержки наполнения тестовыми данными из файла Excel. Вдобавок, чтобы упростить разработку тестового сценария, мы хотим ввести специальную аннотацию по имени `@DataSets`, которая принимает имя файла Excel в качестве аргумента. Мы разработаем специальный слушатель выполнения теста (средство, поддерживаемое инфраструктурой тестирования Spring) для проверки существования этой аннотации и соответствующей загрузки данных.

В последующих разделах мы обсудим реализацию различных классов инфраструктуры и специального слушателя, который загружает данные из файла Excel.

Реализация специального слушателя выполнения теста

В модуле `spring-test` интерфейс `org.springframework.test.context.TestExecutionListener` определяет API-интерфейс слушателя, который может перехватывать события на различных фазах выполнения тестового сценария (например, перед и после тестового класса, перед и после тестового метода и т.д.). При тестировании уровня обслуживания мы реализуем специальный слушатель для введенной нами аннотации `@DataSets`. Цель заключается в поддержке наполнения тестовыми данными с помощью простой аннотации в тестовом сценарии. Например, для тестирования метода `ContactService.findAll()` мы хотим применять примерно такой код, как в листинге 19.7.

Листинг 19.7. Использование аннотации `@DataSets`

```
@DataSets(setUpDataSet="/com/apress/prospring3/ch19/service/jpa/
ContactServiceImplTest.xls")
@Test
public void testFindAll() throws Exception {

    List<Contact> result = customerService.findAll();
    ...
}
```

В листинге 19.7 применение аннотации `@DataSets` к тестовому сценарию указывает, что перед запуском теста тестовые данные должны быть загружены в базу из представленного в аннотации файла Excel.

Первым делом мы должны объявить специальную аннотацию, как показано в листинге 19.8.

Листинг 19.8. Специальная аннотация (`@DataSets`)

```
package com.apress.prospring3.ch19.test.annotation;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface DataSets {

   String setUpDataSet() default "";

}
```

В листинге 19.8 специальная аннотация `@DataSets` является аннотацией уровня методов. Кроме того, специальный класс слушателя теста будет разработан за счет реализации интерфейса `TestExecutionListener`, что можно видеть в листинге 19.9.

Листинг 19.9. Специальный слушатель выполнения теста

```
package com.apress.prospring3.ch19.test.listener;

import org.dbunit.IDatabaseTester;
import org.dbunit.dataset.IDataSet;
import org.dbunit.util.fileloader.XlsDataFileLoader;
import org.springframework.test.context.TestContext;
import org.springframework.test.context.TestExecutionListener;

import com.apress.prospring3.ch19.test.annotation.DataSets;

public class ServiceTestExecutionListener implements TestExecutionListener {

   private IDatabaseTester databaseTester;

   public void afterTestClass(TestContext arg0) throws Exception {
      // TODO Автоматически сгенерированная заглушка метода
   }

   public void afterTestMethod(TestContext arg0) throws Exception {
      // Очистить тестовые данные, если они существуют.
      if (databaseTester != null) {
         databaseTester.onTearDown();
      }
   }

   public void beforeTestClass(TestContext arg0) throws Exception {
      // TODO Автоматически сгенерированная заглушка метода
   }

   public void beforeTestMethod(TestContext testCtx) throws Exception {

      // Проверить существование аннотации DataSets для тестового метода.
      DataSets dataSetAnnotation =
         testCtx.getTestMethod().getAnnotation(DataSets.class);
```

```
        if ( dataSetAnnotation == null ) {
            return;
        }

        String dataSetName = dataSetAnnotation.setUpDataSet();

        // Наполнить набор данных, если имя набора данных существует.
        if ( ! dataSetName.equals("") ) {
            databaseTester = (IDatabaseTester)
                testCtx.getApplicationContext().getBean("databaseTester");
            XlsDataFileLoader xlsDataFileLoader = (XlsDataFileLoader)
                testCtx.getApplicationContext().getBean("xlsDataFileLoader");
            IDataSet dataSet = xlsDataFileLoader.load(dataSetName);

            databaseTester.setDataSet(dataSet);
            databaseTester.onSetup();
        }
    }

    public void prepareTestInstance(TestContext arg0) throws Exception {
        // TODO Автоматически сгенерированная заглушка метода
    }
}
```

После реализации интерфейса `TestExecutionListener` в листинге 19.9 осталось реализовать несколько методов. Однако в рассматриваемом случае нас интересуют только методы `beforeTestMethod()` и `afterTestMethod()`, в которых производится наполнение и очистка тестовых данных перед и после выполнения каждого тестового метода. Обратите внимание, что каждому методу Spring будет передавать экземпляр класса `TestContext`, поэтому метод может получать доступ к `ApplicationContext`, загруженному Spring Framework.

Метод `beforeTestMethod()` представляет особый интерес. Во-первых, он проверяет существование аннотации `@DataSets` для тестового метода. Если аннотация существует, тестовые данные будут загружены из указанного файла Excel. В этом случае интерфейс `IDatabaseTester` (с классом реализации `org.dbunit.DataSourceDatabaseTester`, который будет обсуждаться далее) был получен из `TestContext`. Интерфейс `IDatabaseTester` предоставлен DBUnit и поддерживает операции базы данных на основе имеющегося подключения к базе данных или источника данных.

Во-вторых, экземпляр класса `XlsDataFileLoader` был получен из `TestContext`. Класс `XlsDataFileLoader` — это поддержка DBUnit загрузки данных из файла Excel. “За кулисами” она использует библиотеку Apache POI для чтения файла в формате Microsoft Office. После этого для загрузки данных из файла вызывается метод `XlsDataFileLoader.load()`, который возвращает экземпляр интерфейса `IDataSet`, представляющий загруженный набор данных.

Наконец, в-третьих, для установки тестовых данных вызывается метод `IDatabaseTester.setDataSet()`, а для запуска наполнения данными — метод `IDatabaseTester.onSetup()`.

В методе `afterTestMethod()` для очистки данных вызывается метод `IDatabaseTester.onTearDown()`.

Реализация класса конфигурации

Теперь давайте реализуем класс конфигурации для среды тестирования. Необходимый код представлен в листинге 19.10.

Листинг 19.10. Класс ServiceTestConfig

```
package com.apress.prospring3.ch19.test.config;

import javax.sql.DataSource;

import org.dbunit.DataSourceDatabaseTester;
import org.dbunit.util.fileloader.XlsDataFileLoader;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.ImportResource;
import org.springframework.context.annotation.Profile;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseBuilder;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType;

@Configuration
@ImportResource("classpath:datasource-tx-jpa.xml")
@ComponentScan(basePackages={"com.apress.prospring3.ch19.service.jpa"})
@Profile("test")
public class ServiceTestConfig {

   @Bean
   public DataSource dataSource() {
      return new EmbeddedDatabaseBuilder().setType(EmbeddedDatabaseType.H2)
         .addScript("classpath:schema.sql").build();
   }

   @Bean(name="databaseTester")
      public DataSourceDatabaseTester dataSourceDatabaseTester() {
      DataSourceDatabaseTester databaseTester =
         new DataSourceDatabaseTester(dataSource());
      return databaseTester;
   }

   @Bean(name="xlsDataFileLoader")
   public XlsDataFileLoader xlsDataFileLoader() {
      return new XlsDataFileLoader();
   }
}
```

В листинге 19.10 класс `ServiceTestConfig` определяет `ApplicationContext` для тестирования уровня обслуживания. Во-первых, импортируется XML-файл конфигурации `datasource-tx-jpa.xml`, в котором определена транзакция и конфигурация JPA, повторно используемые для тестирования. Затем применяется аннотация `@ComponentScan`, сообщающая Spring о необходимости сканировать бины уровня обслуживания, которые мы хотим тестировать. Аннотация `@Profile` указывает, что бины, определенные в этом классе, принадлежат профилю `test`.

Во-вторых, внутри класса объявляется еще один бин `dataSource`, который выполняет только сценарий `schema.sql` в базе данных H2, не наполняя ее информацией. Бины `databaseTester` и `xlsDataFileLoader` используются специальным слушателем выполнения теста для загрузки данных из файла Excel. Обратите внимание, что бин `dataSourceDatabaseTester` сконструирован с применением бина `dataSource`, определенного для среды тестирования.

Реализация базового тестового класса

В листинге 19.11 показан абстрактный базовый класс для тестирования уровня обслуживания.

Листинг 19.11. Класс `AbstractServiceImplTest`

```
package com.apress.prospring3.ch19.service.jpa;

import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

import org.junit.runner.RunWith;
import org.springframework.test.context.ActiveProfiles;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.TestExecutionListeners;
import org.springframework.test.context.junit4.
AbstractTransactionalJUnit4SpringContextTests;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

import com.apress.prospring3.ch19.test.config.ServiceTestConfig;
import com.apress.prospring3.ch19.test.listener.ServiceTestExecutionListener;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {ServiceTestConfig.class})
@TestExecutionListeners({ServiceTestExecutionListener.class})
@ActiveProfiles("test")
public abstract class AbstractServiceImplTest
  extends AbstractTransactionalJUnit4SpringContextTests {

   @PersistenceContext
   protected EntityManager em;
}
```

Аннотация `@RunWith` в листинге 19.11 — та же, что использовалась при тестировании класса контроллера. Аннотация `@ContextConfiguration` указывает, что конфигурация `ApplicationContext` должна быть загружена из класса `ServiceTestConfig`. Аннотация `@TestExecutionListeners` определяет, что для перехвата жизненного цикла выполнения тестового сценария должен применяться класс `ServiceTestExecutionListener`. Аннотация `@ActiveProfiles` указывает используемый профиль. Таким образом, в этом случае будет загружен бин `dataSource`, определенный в классе `ServiceTestConfig`, а не в файле `datasource-tx-jpa.xml`, поскольку он принадлежит профилю `dev`.

Вдобавок класс расширяет класс `AbstractTransactionalJUnit4SpringContextTests`, который представляет собой предлагаемую Spring поддержку JUnit, вместе с внедрением зависимостей и механизмом управления транзакциями Spring. Обратите внимание, что в среде тестирования Spring будет производить откат транзакции после выполнения каждого тестового метода, поэтому все операции базы данных также подвергнутся откату. Для управления поведением отката можно использовать аннотацию `@Rollback` на уровне методов.

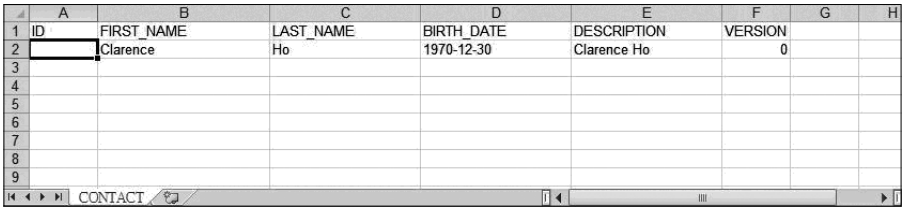
Наконец, в абстрактном базовом классе `EntityManager` автосвязан и может затем применяться внутри тестовых сценариев.

Модульное тестирование уровня обслуживания

Имея классы конфигурации и инфраструктуры, мы можем реализовать модульный тест для уровня обслуживания.

Начнем с модульного тестирования методов поиска, включая `ContactService.findAll()` и `ContactService.findByFirstNameAndLastName()`. Первым делом, необходимо подготовить тестовые данные в формате Excel. Общепринятая практика заключается в помещении файла в папку, в которой находится класс тестового сценария, и назначении ему того же самого имени. Таким образом, в рассматриваемом случае именем файла будет `/src/test/java/com/apress/prospring3/ch19/service/jpa/ContactServiceImplTest.xls`. Содержимое этого файла можно видеть на рис. 19.6.

Как показано на рис. 19.6, тестовые данные подготовлены на рабочем листе. Именем рабочего листа является имя таблицы, а в первой строке указаны имена столбцов в таблице. Столбец идентификатора `ID` присутствует, но значение для него не указано. Это объясняется тем, что идентификаторы будут заполняться самой базой данных. Мы имеем в этом файле только одну запись контакта, поэтому метод `findAll()` должен вернуть один контакт.



	A	B	C	D	E	F	G	H
1	ID	FIRST_NAME	LAST_NAME	BIRTH_DATE	DESCRIPTION	VERSION		
2		Clarence	Ho	1970-12-30	Clarence Ho	0		
3								
4								
5								
6								
7								
8								
9								

CONTACT

Рис. 19.6. Тестовые данные в формате Excel

В листинге 19.12 представлен тестовый класс с тестовыми сценариями для двух методов поиска.

Листинг 19.12. Тестирование методов поиска

```
package com.apress.prospring3.ch19.service.jpa;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotNull;
import static org.junit.Assert.assertNull;

import java.util.List;

import org.junit.Test;
import org.springframework.beans.factory.annotation.Autowired;

import com.apress.prospring3.ch19.domain.Contact;
import com.apress.prospring3.ch19.service.ContactService;
import com.apress.prospring3.ch19.test.annotation.DataSets;

public class ContactServiceImplTest extends AbstractServiceImplTest {

   @Autowired
   ContactService customerService;

   @DataSets(
setUpDataSet="/com/apress/prospring3/ch19/service/jpa/
ContactServiceImplTest.xls")
   @Test
   public void testFindAll() throws Exception {

      List<Contact> result = customerService.findAll();

      assertNotNull(result);
      assertEquals(1, result.size());
   }
```

```
    @DataSets(
setUpDataSet="/com/apress/prospring3/ch19/service/jpa/ContactServiceImplTest.xls")
    @Test
    public void testFindByFirstNameAndLastName_1() throws Exception {

        Contact result =
            customerService.findByFirstNameAndLastName("Clarence", "Ho");

        assertNotNull(result);
    }

    @DataSets(
setUpDataSet="/com/apress/prospring3/ch19/service/jpa/ContactServiceImplTest.xls")
    @Test
    public void testFindByFirstNameAndLastName_2() throws Exception {

      Contact result = customerService.findByFirstNameAndLastName("Peter", "Chan");

      assertNull(result);
    }
}
```

Тестовый сценарий в листинге 19.12 расширяет базовый тестовый класс `AbstractServiceImplTest`, который содержит всю конфигурацию, требуемую для теста. Имеется один тестовый сценарий для метода `findAll()` и два тестовых сценария для метода `testFindByFirstNameAndLastName()` (один извлекает результат, а другой нет). Ко всем методам поиска применяется аннотация `@DataSets` с указанным файлом Excel с тестовыми данными. Кроме того, служба `ContactService` автосвязана с тестовым сценарием из `ApplicationContext`. Остальной код должен быть самоочевидным. В каждом тестовом сценарии применяются различные операторы утверждений, которые проверяют, является ли результат ожидаемым.

Чтобы запустить этот тестовый класс, щелкните на нем правой кнопкой мыши и выберите в контекстном меню пункт Run As⇨JUnit Test (Запустить как⇨Тест JUnit). Вкладка JUnit в STS показана на рис. 19.7.

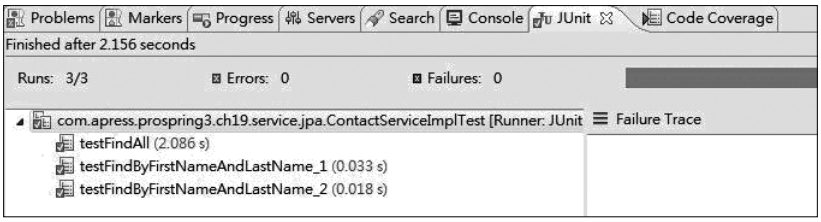


Рис. 19.7. Вкладка JUnit при тестировании уровня обслуживания

Теперь давайте протестируем операцию сохранения. Мы планируем тестировать два случая. Один из них — нормальная ситуация, при которой допустимый контакт сохраняется успешно, а другой — ситуация, когда контакт содержит ошибку, которая приводит к генерации корректного исключения. В листинге 19.13 приведен фрагмент кода для этих двух тестовых сценариев.

Листинг 19.13. Тестирование операции сохранения

```
package com.apress.prospring3.ch19.service.jpa;

import javax.validation.ConstraintViolationException;

// Другие операторы импорта опущены.
```

```
public class ContactServiceImplTest extends AbstractServiceImplTest {
    // Остальной код не показан.
    @Test
    public void testAddContact() throws Exception {
        // Очистить все существующие данные в таблице Contact.
        deleteFromTables("CONTACT");
        Contact contact = new Contact();
        contact.setFirstName("Rod");
        contact.setLastName("Johnson");
        contact = customerService.save(contact);
        em.flush();
        List<Contact> contacts = customerService.findAll();
        assertEquals(1, contacts.size());
    }
    @Test(expected=ConstraintViolationException.class)
    public void testAddContactWithJSR303Error() throws Exception {
        // Очистить все существующие данные в таблице Contact.
        deleteFromTables("CONTACT");
        Contact contact = new Contact();
        contact = customerService.save(contact);
        em.flush();
        List<Contact> contacts = customerService.findAll();
        assertEquals(0, contacts.size());
    }
}
```

Обратите внимание в листинге 19.13 на метод `testAddContact()`. Внутри этого метода для гарантии того, что в таблице `CONTACT` нет никаких данных, мы вызываем удобный метод `deleteFromTables()`, предоставляемый классом `AbstractTransactionalJUnit4SpringContextTests`, который очищает таблицу. После вызова операции сохранения мы должны явно вызвать метод `EntityManager.flush()`, чтобы заставить Hibernate сбросить контекст постоянства в базу данных, и тогда метод `findAll()` сможет корректно извлечь информацию из базы данных.

Во втором тестовом методе, `testAddContactWithJSR303Error()`, мы тестируем операцию сохранения контакта, содержащего ошибку проверки достоверности. Здесь в аннотации `@Test` передается атрибут `expected`, который указывает, что тестовый сценарий ожидает генерации исключения заданного типа, в данном случае — класса `ConstraintViolationException`.

Запустите тестовый класс снова и просмотрите результаты на вкладке JUnit.

Следует отметить, что были рассмотрены только наиболее часто используемые классы в инфраструктуре тестирования Spring. Эта инфраструктура предлагает множество поддерживающих классов и аннотаций, которые делают возможным точный контроль на протяжении жизненного цикла тестового сценария. Например, аннотации `@BeforeTransaction` и `@AfterTransaction` позволяют запускать определенную логику перед тем, как Spring инициирует транзакцию, или после того, как транзакция завершена для тестового сценария. За более подробными сведениями о разнообразных аспектах инфраструктуры тестирования Spring обращайтесь в справочную документацию Spring.

Реализация модульного тестирования интерфейсной части

Еще одной интересной областью тестирования является проверка поведения интерфейсной части как единого целого после развертывания веб-приложения в веб-контейнере, подобном Apache Tomcat.

Основная причина в том, что хотя мы тестируем каждый уровень внутри приложения, мы по-прежнему должны удостовериться, что представления ведут себя корректно при различных действиях со стороны пользователей. Автоматизация тестирования интерфейсной части очень важна для экономии времени разработчиков и пользователей при наличии в тестовом сценарии повторяющихся действий в интерфейсной части.

Однако разработка тестового сценария для интерфейсной части — задача непростая, особенно в приложениях с большим количеством интерактивных, насыщенных и основанных на Ajax компонентов.

В последующих разделах мы обсудим реализацию модульного тестирования интерфейсной части с применением Selenium (`http://seleniumhq.org`) — популярной инфраструктуры с открытым кодом для автоматизации тестирования интерфейсной части.

Добавление обязательных зависимостей

Для использования Selenium к проекту понадобится добавить зависимость, описанную в табл. 19.4.

Таблица 19.4. Зависимость Maven для Selenium

Идентификатор группы	Идентификатор артефакта	Версия	Описание
`org.seleniumhq.selenium`	`selenium-java`	2.15.0	Библиотека Selenium для Java

Введение в Selenium

Selenium представляет собой комплексный инструмент и инфраструктуру, которые предназначены для автоматизации тестирования интерфейсной части. Главная особенность заключается в том, что с помощью Selenium можно “управлять” браузерами, эмулируя взаимодействие пользователей с приложением, и контролировать состояние представлений.

Инфраструктура Selenium поддерживает такие общие браузеры, как Firefox, IE и Chrome. Что касается языков, то поддерживаются Java, C#, PHP, Perl, Ruby и Python. Инструмент Selenium также спроектирован с учетом Ajax-приложений и насыщенных Интернет-приложений (RIA), делая возможным тестирование современных веб-приложений.

В случае, когда приложение имеет множество пользовательских интерфейсов, и для него необходимо прогонять большое количество тестов интерфейсной части, модуль `selenium-server` предлагает встроенную функциональность сетки, которая поддерживает выполнение тестов интерфейсной части в группе компьютеров.

Selenium IDE — это подключаемый модуль для Firefox, который помогает “записывать” взаимодействие пользователя с веб-приложением. Он также поддерживает повторное воспроизведение и экспортирует сценарии в различных форматах, упрощая разработку тестовых сценариев.

Начиная с версии 2.0, Selenium интегрируется с WebDriver API, который снимает множество ограничений и предлагает альтернативный и более простой программный интерфейс. Результатом является комплексный объектно-ориентированный API-интерфейс, который предоставляет дополнительную поддержку для множества браузеров наряду с улучшенной поддержкой решения проблем, связанных с тестированием современных сложных веб-приложений.

Чтобы реализовать тестовые сценарии для интерфейсной части приложения контактов, мы будем использовать инструмент Selenium с его поддержкой WebDriver API.

Реализация тестового сценария для пользовательского интерфейса

В рамках тестирования интерфейсной части мы собираемся разработать тестовый сценарий для интерфейса добавления контакта. Будут реализованы нормальный сценарий (т.е. пользователь корректно вводит информацию, и контакт успешно сохраняется) и сценарий с исключением (т.е. возникает ошибка проверки достоверности с отображением соответствующего сообщения).

Реализовать простой тестовый сценарий в Selenium довольно легко. Код тестового класса приведен в листинге 19.14.

Листинг 19.14. Тестирование интерфейсной части с помощью Selenium

```
package com.apress.prospring3.ch19.test.ui;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebDriverBackedSelenium;
import org.openqa.selenium.firefox.FirefoxDriver;

import com.thoughtworks.selenium.SeleneseTestBase;

public class UIAddContactTest extends SeleneseTestBase {

   private static final String USERNAME = "user";
   private static final String PASSWORD = "user";

   @Before
   public void setUp() throws Exception {
      WebDriver driver = new FirefoxDriver();
      String baseUrl = "http://localhost:8080/";
      selenium = new WebDriverBackedSelenium(driver, baseUrl);
   }

   @Test
   public void testAddContact() {

      // Вход в приложение
      loginAs(USERNAME, PASSWORD);

      // Вход на форму добавления контакта
      selenium.open("/ch19/contacts?form");
      selenium.waitForPageToLoad("30000");

      // Заполнение информации о контакте
      selenium.type("firstName", "Andy");
      selenium.type("lastName", "Lau");
      selenium.click("name=submit");
```

```
        // Контроль
        verifyTrue(selenium.isTextPresent("Andy"));
        verifyTrue(selenium.isTextPresent("Lau"));

        // Выход из приложения
        logout();
    }
    @Test
    public void testAddContactWithEmptyForm() {
        // Вход в приложение
        loginAs(USERNAME, PASSWORD);

        // Вход на форму добавления контакта
        selenium.open("/ch19/contacts?form");
        selenium.waitForPageToLoad("30000");

        // Отправка формы
        selenium.click("name=submit");

        // Контроль
        verifyTrue(selenium.isTextPresent("Failed saving contact"));

        // Выход из приложения
        logout();
    }
    @After
    public void tearDown() throws Exception {
        selenium.stop();
    }
    private void loginAs(String userName, String password) {
        selenium.open("/ch19/contacts");
        selenium.type("j_username", userName);
        selenium.type("j_password", password);
        selenium.click("name=login");
        selenium.waitForPageToLoad("30000");
    }
    private void logout() {
        selenium.click("link=Logout");
    }
}
```

В листинге 19.14 класс расширяет класс `SeleneseTestBase`, который предоставляет много удобных методов для упрощения реализации тестового сценария. В методе `setUp()` подготовлен экземпляр `WebDriver` с помощью класса `FirefoxDriver`, который вызовет браузер Firefox, установленный на машине тестирования, для выполнения тестового сценария. Существуют также драйверы для Chrome, IE, HtmlUnit и т.д. Если веб-приложение должно поддерживать множество браузеров, тот же самый тест можно прогонять несколько раз с различными веб-драйверами, чтобы удостовериться в согласованном поведении приложения в разных браузерах. Затем конструируется экземпляр класса `WebDriverBackedSelenium`, который представляет собой предлагаемую Selenium поддержку использования API-интерфейса `WebDriver` для управления взаимодействием с браузером.

А теперь посмотрим на методы `loginAs()` и `logout()`, которые будут многократно использоваться во многих тестовых сценариях для действий входа и выхода. Например, метод `loginAs()` открывает домашнюю страницу (т.е. `http://localhost:8080/ch19/contacts`), заполняет поля `j_username` и `j_password` формы и затем щелкает на кнопке Login (Вход). Элементы в форме должны иметь соответствующие имена.

Например, взгляните на форму входа в файле представления `menu.jspx`, которая показана в листинге 19.15.

Листинг 19.15. Форма входа

```
<form name="loginForm" action="${loginUrl}" method="post">
   <table>
      <caption align="left">Login:</caption>
      <tr>
         <td>User Name:</td>
         <td><input type="text" name="j_username"/></td>
      </tr>
      <tr>
         <td>Password:</td>
         <td><input type="password" name="j_password"/></td>
      </tr>
      <tr>
         <td colspan="2" align="center"><input name="login"
             type="submit" value="Login"/></td>
      </tr>
   </table>
</form>
```

Назначенные полям формы имена выделены полужирным. Таким образом, когда выполняется вызов `selenium.type("j_username", userName)`, поле формы с именем `j_username` заполняется передаваемым именем пользователя и т.д.

Метод `testAddContact()` — это тестовый сценарий для нормальной операции добавления контакта. После входа в приложение производится обращение к URL нового контакта (с использованием метода `open()`), затем вводит информация о контакте и выполняется щелчок на кнопке Save (Сохранить). Далее вызывает метод `verifyTrue()` для контроля, что операция добавления успешно завершена с отображением информации о контакте на странице.

Метод `testAddContactWithEmptyForm()` эмулирует случай, когда пользователь щелкает на кнопке Save (Сохранить), не введя информацию о контакте. В результате операция сохранения даст сбой и метод `verifyTrue()` будет вызван для контроля за тем, что сообщение об ошибке отобразилось на странице.

Чтобы прогнать этот тест, щелкните правой кнопкой мыши на тестовом классе и выберите в контекстном меню пункт Run As⇨JUnit Test (Запустить как⇨Тест JUnit). Удостоверьтесь, что веб-приложение развернуто и сервер tc Server функционирует. Затем вы увидите, что тестовые сценарии автоматически вызывают копию браузера Firefox, и вход в приложение, ввод информации о контакте и логика контроля соответствующим образом выполняются.

Благодаря этому простому примеру, вы видите, как Selenium помогает автоматизировать взаимодействие пользователя с интерфейсной частью веб-приложения, совместимое с множеством браузеров. За дополнительными сведениями обращайтесь в онлайновую документацию по Selenium (`http://seleniumhq.org/docs`).

Контроль покрытия кода тестовыми сценариями

Тестовые сценарии, которые мы разработали, должны покрывать большую часть реализованной бизнес-логики и пользовательского интерфейса, и это называется *покрытием кода*. В идеальном случае степень покрытия кода должна быть равна 100%, что означает тестирование каждой строки написанного кода.

В среде Eclipse (под управлением которой выполняется STS) имеется множество подключаемых модулей, которые помогают визуализировать покрытие кода для разработанных тестовых сценариев. В этом разделе мы представим инструмент CodePro Analytix (`http://code.google.com/intl/en/javadevtools/codepro/doc/index.html`).

Ранее CodePro Analytix был коммерческим инструментом. Однако после приобретения Google родительской компании этот инструмент теперь распространяется бесплатно. CodePro Analytix предоставляет много средств для управления качеством кода, таких как анализ кода, генерация тестов JUnit, покрытие кода и т.п. Чтобы установить CodePro Analytix в STS, выберите пункт меню Help⇨Install New Software (Справка⇨Установить новое программное обеспечение) в STS и щелкните на кнопке Add (Добавить). Введите имя CodePro Analytix и URL `http://dl.google.com/eclipse/inst/codepro/latest/3.7`. Отобразятся компоненты, доступные для выбора, как показано на рис. 19.8.

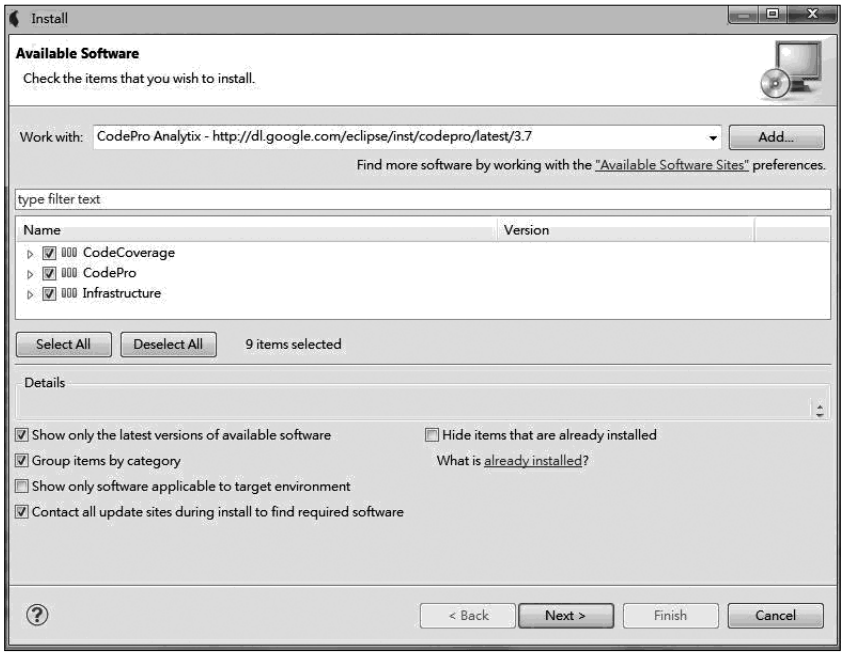


Рис. 19.8. Установка CodePro Analytix в STS

Отметьте все компоненты (хотя в этом разделе мы используем только средство Code Coverage (Покрытие кода)) и щелкните на кнопке Next (Далее) для установки подключаемого модуля.

После того, как установка завершена, а среда STS перезапущена, мы можем начать измерять покрытие кода. Первым делом, понадобится включить инструмент для проверки покрытия кода. Щелкните правой кнопкой мыши на проекте и выберите в контекстном меню пункт CodePro Tools⇨Instrument for Code Coverage (Инструменты CodePro⇨Инструмент для покрытия кода), как показано на рис. 19.9.

Затем запустите модульный тест, и по его завершении инструмент CodePro автоматически активизирует вкладку Code Coverage (Покрытие кода) с отображением результатов покрытия кода. На рис. 19.10 можно видеть часть окна STS после выполнения класса `ContactServiceImplTest`.

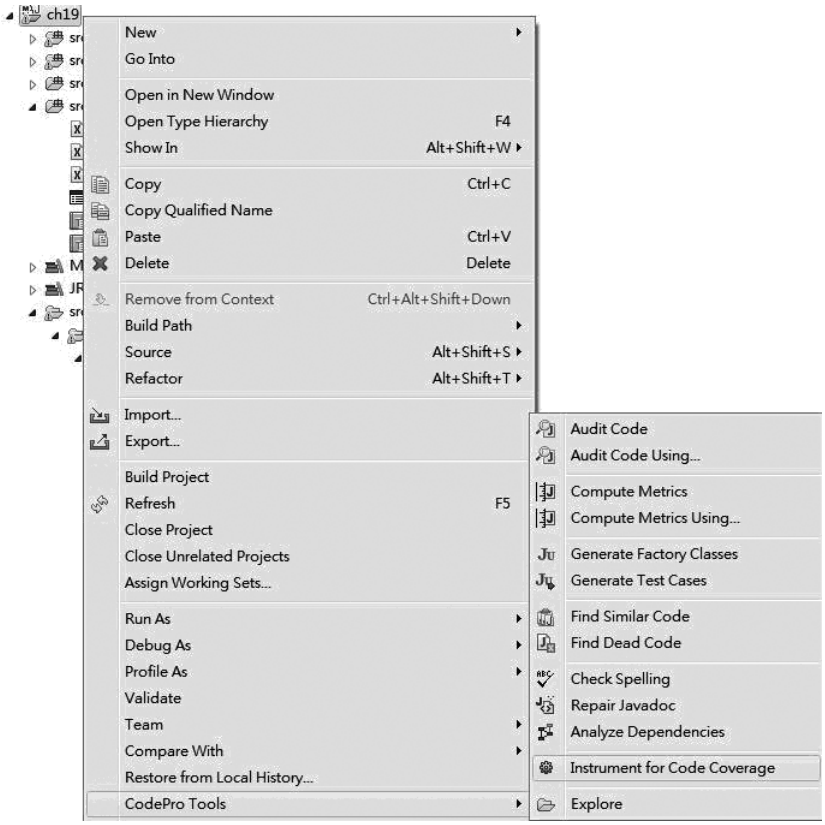


Рис. 19.9. Включение инструмента для проверки покрытия кода

В нижней части окна вкладка Code Coverage (Покрытие кода) представляет покрытие кода для всех классов. Так как тестовый класс проверял класс `ContactServiceImpl`, щелкните на этом классе, чтобы просмотреть справа детальную информацию о покрытии кода. На рис. 19.10 видно, что покрытие кода для класса `ContactServiceImpl` составляет 75%. Двойной щелчок на этом классе приведет к открытию его кода в области редактора. В этой области протестированный код помечен слева зеленым индикатором, тогда как код, который не вызывался при тестировании — красным индикатором. Следовательно, можно визуально оценить, сколько кода покрывается тестовым сценарием, и увидеть какие методы и строки кода не были протестированы. Это удобный инструмент, который поможет увеличить покрытие кода тестами.

Резюме

В этой главе мы показали, как разрабатывать различные виды модульного тестирования в Spring-приложениях с помощью часто используемых инфраструктур, библиотек и инструментов, включая JUnit, DBUnit, Mockito, Selenium и т.д.

Во-первых, мы представили высокоуровневое описание корпоративной инфраструктуры тестирования, в котором показано, какие тесты должны выполняться на каждой фазе жизненного цикла разработки приложения. Во-вторых, мы разработали три типа тестов — модульный тест логики, модульный тест взаимодействия и модульный тест интерфейсной части.

Наконец, в-третьих, мы обсудили применение инструмента CodePro Analytix для измерения покрытия кода тестовыми классами.

Тестирование корпоративного приложения — это очень крупная тема, и если вы хотите более детально изучить библиотеку JUnit, рекомендуем почитать книгу *JUnit in Action, Second Edition* (Manning, 2010 г.).

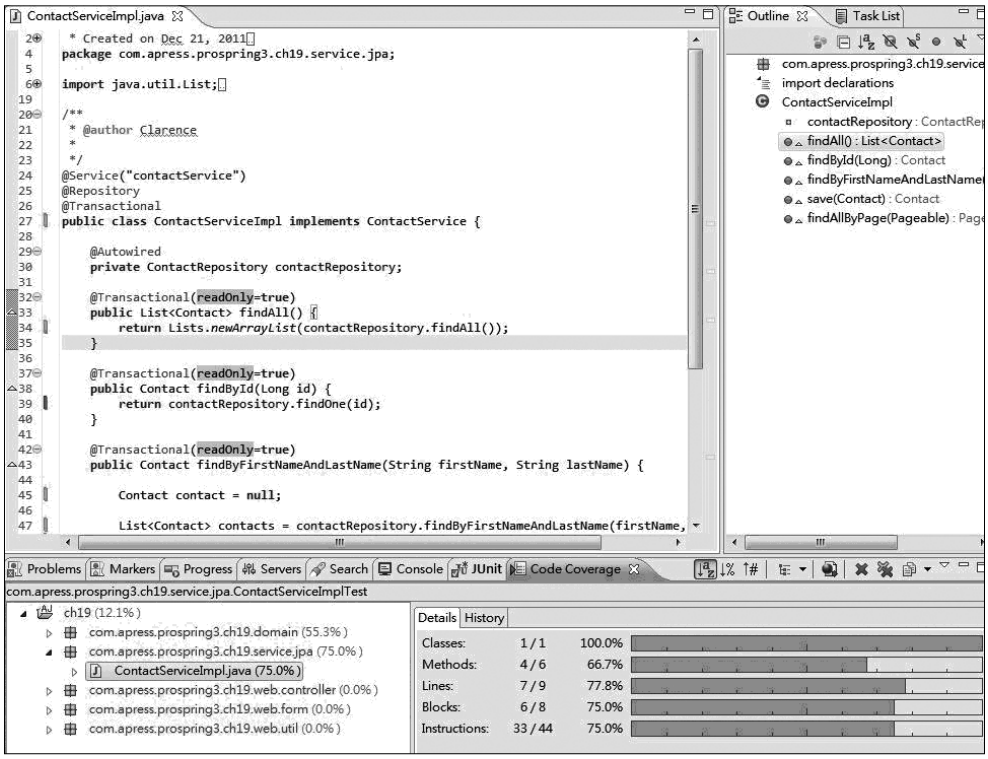


Рис. 19.10. Исследование покрытия кода

ГЛАВА 20

Проекты Spring: Spring Batch, Spring Integration и Spring Roo

В предшествующих главах мы показали, что Spring Framework предоставляет целостную инфраструктуру с внедрением зависимостей и АОП, предназначенную для разработки корпоративных Java-приложений. Мы также рассмотрели несколько других проектов Spring (Spring Data JPA, Spring Security, Spring Web Flow и т.д.), которые помогают расширять возможности Spring-приложений в специфических областях.

Очевидно, что массовое принятие Spring Framework в средах с открытым кодом и корпоративных средах привело к формированию громадной экосистемы разработки, которая создала потребность в расширениях для Spring Framework. В результате на основе Spring Framework было разработано множество других проектов Spring, нацеленных на решение более специфических аспектов приложений. Например, проект Spring Social поддерживает взаимодействие Spring-приложений с популярными социальными сетями наподобие Facebook и Twitter, а целью проекта Spring Mobile является упрощение разработки мобильных веб-приложений. Полный список проектов, сопровождаемых командой SpringSource, доступен по адресу `www.springsource.org/projects`.

Все проекты Spring были спроектированы с одним и тем же логическим обоснованием, т.е. для продвижения наилучшего опыта разработки приложений (другими словами, программирование, управляемое интерфейсами, внедрение зависимостей, АОП и т.д.) и для бесшовной интеграции со средой `ApplicationContext` из Spring Framework.

В этой главе мы представим высокоуровневое описание нескольких популярных проектов Spring, которые помогают при разработке Spring-приложений. В частности, в этой главе рассматриваются следующие темы.

- *Проект Spring Batch.* Проект Spring Batch (`static.springsource.org/spring-batch`) предоставляет комплексную инфраструктуру и стандартный каркас для разработки пакетных заданий внутри Spring-приложений. Мы дадим высокоуровневый обзор этого проекта, а также продемонстрируем его использование, разработав простое задание импорта контактов.
- *Проект Spring Integration.* Проект Spring Integration (`www.springsource.org/spring-integration`) позволяет Spring-приложениям взаимодействовать с другими приложениями и источниками данных. Он предоставляет облегченную платформу обмена сообщениями для реализации многих типовых шаблонов интеграции корпоративных приложений (enterprise integration pattern — EIP). В данной

главе мы кратко опишем этот проект и покажем, как объединить проекты Spring Integration и Spring Batch вместе для выполнения пакетных заданий согласно простому шаблону EIP.

- *Проект Spring Roo*. Проект Spring Roo (`www.springsource.org/spring-roo`) предоставляет комплексный инструмент, который помогает строить приложения с помощью простых команд. Это исключительно полезно для построения прототипов приложений, и генерируемый Spring Roo Java-код является полностью настраиваемым. В этой главе мы продемонстрируем работу Spring Roo на примере генерации простого приложения контактов.

Проект для примеров этой главы

В этой главе мы будем использовать серверную часть веб-приложения контактов, которая была реализована в главе 18, и построим пакетное задание для импорта информации о контактах в формате XML.

Импорт проекта примера серверной части в STS

Первая часть этой главы посвящена интеграции и разработке пакетных заданий, поэтому мы подготовили проект для простой серверной части, который можно импортировать и заняться непосредственно реализацией пакетного задания.

Извлеките файл `ch20-batch-backend.zip` из кода примеров в рабочее пространство STS. После этого импортируйте проект в STS.

Чтобы импортировать этот проект, в среде STS выберите пункт меню File⇨Import (Файл⇨Импорт) и на экране Select (Выбор) укажите вариант Existing Projects into Workspace (Существующие проекты в рабочем пространстве) в категории General (Общие), как показано на рис. 20.1. Щелкните на кнопке Next (Далее).

После этого на экране Import Projects (Импорт проектов) выберите проект, извлеченный из кода примеров, и щелкните на кнопке Finish (Готово), как показано на рис. 20.2.

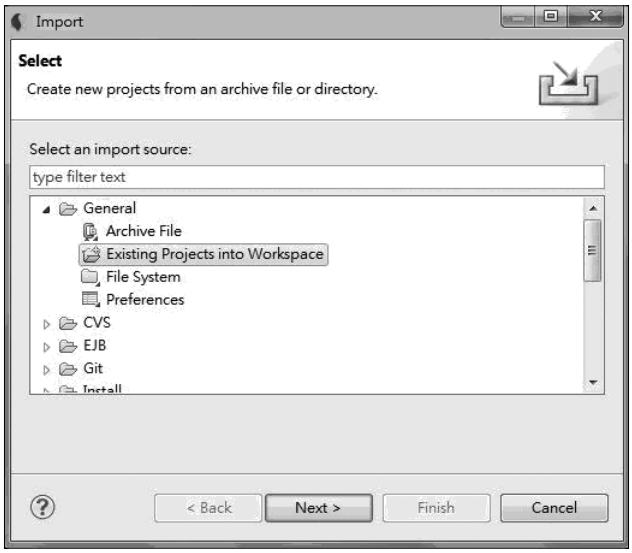


Рис. 20.1. Импорт существующего проекта в STS

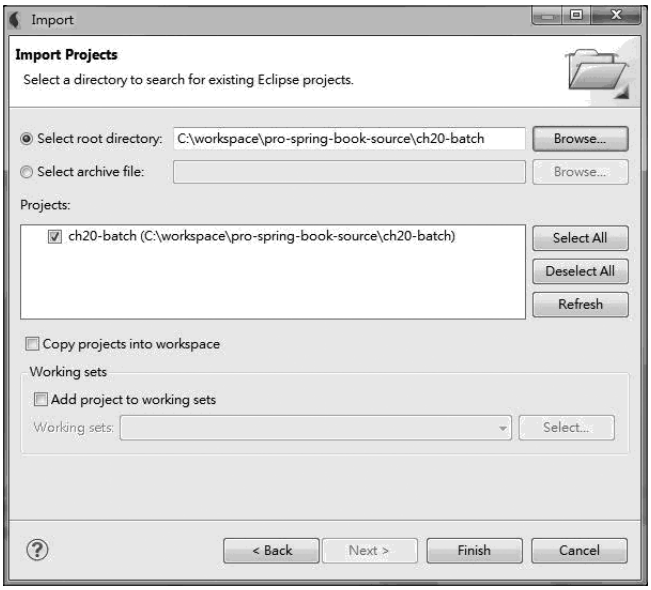


Рис. 20.2. Импорт проекта примера

Введение в Spring Batch

Начатый в 2007 г., проект Spring Batch обретал все большую и большую популярность при разработке пакетный заданий в Spring-приложениях. Это результат совместных усилий SpringSource и Accenture, и его исходной целью была разработка стандартного шаблона и инфраструктуры для реализации и выполнения различных видов пакетных заданий в корпоративной среде. В версии 2 за счет внесения изменений проект Spring Batch стал более зрелым, обеспечивая более высокую производительность и поддерживая множество разных политик (например, пропуск поврежденных записей, повтор, перезапуск, параллельное выполнения и т.п.) при выполнении пакетного задания. В последующих разделах мы рассмотрим фундаментальные концепции проекта Spring Batch, включая главный поток и обработку пакетного задания, инфраструктуру и метаданные, а также основные действия при выполнении, обеспечиваемые Spring Batch.

Поток и процесс пакетного задания

Каждое корпоративное приложение требует некоторого вида пакетной обработки, которая выполняет бизнес-логику на большом наборе информации. Ниже перечислены примеры пакетных заданий:

- импорт/экспорт информации из/во внешние системы;
- обработка, выполняемая в конце дня, в системах обработки транзакций;
- генерация внешних документов (например, ежемесячных бюллетеней для заказчиков).

Хотя каждое пакетное задание выполняет отличающуюся логику, их высокоуровневые потоки похожи.

Поток типичного пакетного задания выглядит следующим образом.

1. *Чтение*. Читает информацию из источника. Источником может быть файл, база данных, сообщение JMS, запрос веб-службы и т.д.
2. *Обработка*. Обрабатывает информацию, прочитанную из источника. Это включает проверку достоверности, выведение другой поддерживающей информации, применение бизнес-правил и т.п.
3. *Запись*. Выводит обработанную информацию в приемник. Приемником может быть файл, база данных, сообщение JMS, ответ веб-службы и т.д.

Проект Spring Batch предоставляет инфраструктуру выполнения и множество готовых классов, которые значительно упрощают разработку пакетных заданий. В этой главе мы покажем, как использовать Spring Batch для реализации пакетного задания, которое выполняет пакетный импорт информации о контактах из файла в формате XML.

Существуют два основных поколения проекта Spring Batch — версия 1 и версия 2. Версия 2 сильно отличается от версии 1, и наиболее важным изменением является переход от обработки, ориентированной на элементы, к обработке, ориентированной на порции. При обработке, ориентированной на элементы, в версии 1 каждая запись (к примеру, строка в плоском файле) будет проходить через поток чтение⇨обработка⇨запись. В случае обработки крупных объемов информации такой подход обеспечивает очень низкую производительность.

В версии 2 была введена обработка, ориентированная на порции. В этом случае шаги чтение⇨обработка повторяются несколько раз, что определено размером порции. Когда достигнут размер порции, Spring Batch отправляет пакет обработанной информации шагу записи для выполнения группового обновления. Такой подход значительно улучшает производительность при обработке крупных объемов данных.

Компоненты инфраструктуры Spring Batch

В табл. 20.1 перечислены основные компоненты инфраструктуры времени выполнения Spring Batch. Обратите внимание, что все компоненты являются интерфейсами.

Таблица 20.1. Компоненты инфраструктуры Spring Batch

Компонент	Описание
`JobRepository`	Предоставляет операции доступа к данным (CRUD) для лежащих в основе метаданных Spring Batch
`JobLauncher`	Запускает задание. Отвечает за запуск выполнения задания на основе конкретного задания и параметров
`JobOperator` (появился в версии 2)	Предоставляет операции пакетных заданий (например, останов выполнения задания, перезапуск отказавшего или остановленного задания)
`JobExplorer` (появился в версии 2)	Извлекает информацию о состоянии выполнения задания из метаданных

На рис. 20.3 показано высокоуровневое представление компонентов инфраструктуры Spring Batch, взятое из справочной документации по проекту Spring Batch.

Метаданные Spring Batch

Spring Batch хранит всю информацию о выполнении задания в собственных метаданных. Скорее всего, метаданные будут находиться в той же базе данных, что и корпоративное приложение, хотя можно держать метаданные в памяти (однако в этом случае

теряются основные преимущества Spring Batch, такие как перезапуск отказавшего задания и т.п.). Spring Batch предоставляет DDL-сценарии для создания структуры метаданных в наиболее распространенных СУРБД. На рис. 20.4 показана модель данных Spring Batch для базы данных Oracle 11g.

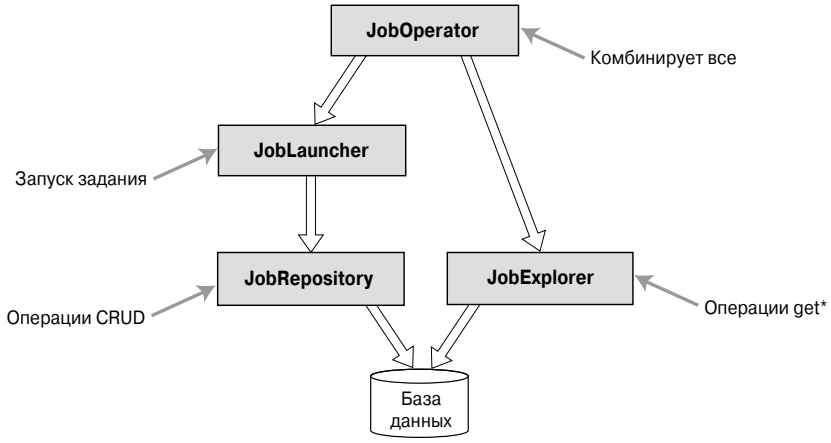


Рис. 20.3. Компоненты инфраструктуры Spring Batch

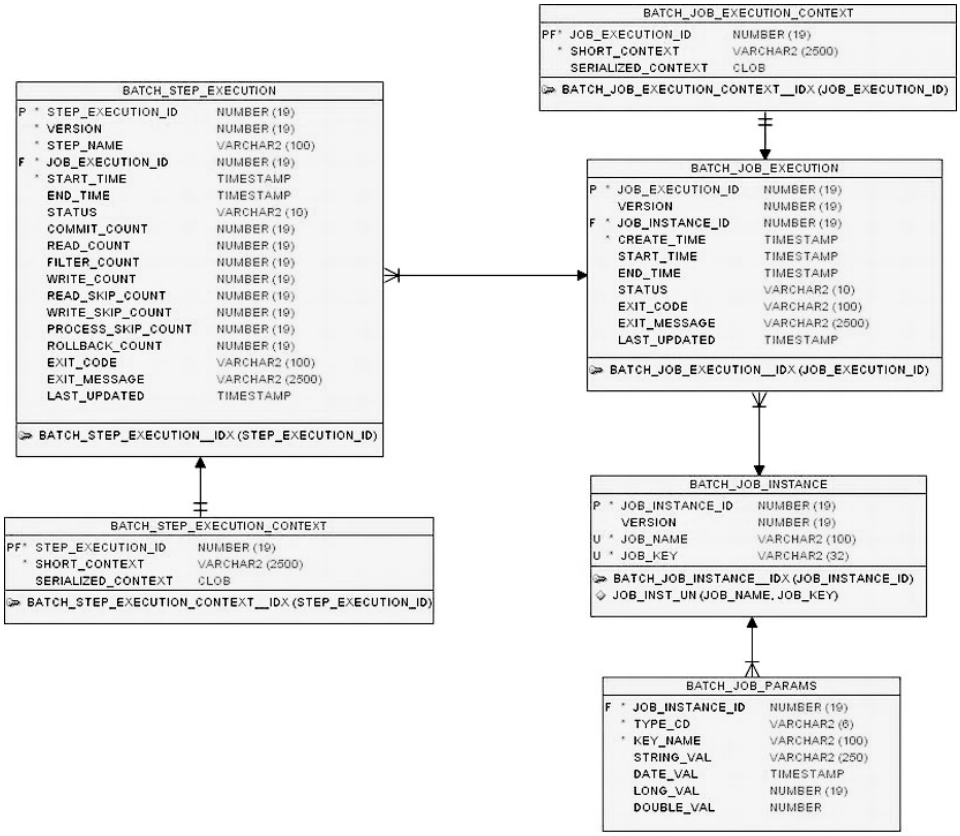


Рис. 20.4. Метаданные Spring Batch для Oracle 11g

В табл. 20.2 описаны категории и назначения каждой таблицы в метаданных Spring Batch.

Таблица 20.2. Метаданные Spring Batchz

Категория	Таблица	Описание
Экземпляр задания	BATCH_JOB_INSTANCE	Каждый раз, когда модуль запуска начинает задание, создается новый экземпляр пакетного задания, и сведения о выполнении (имя задания, параметры и т.п.) сохраняются в таблице BATCH_JOB_INSTANCE. В этой таблице имеется столбец JOB_KEY, значение которого генерируется на основе параметров данного экземпляра задания (имени файла, даты и т.д.). Важно отметить наличие в таблице BATCH_JOB_INSTANCE ограничения уникальности, состоящего из столбцов JOB_NAME и JOB_KEY.Это значит, что задание вместе со своими параметрами должно быть уникальным. Именем задания для импорта контактов является importContactJob, а параметром — имя файла. В этом случае для каждого задания импорта контактов должно указываться уникальное имя файла. Если вы попытаетесь запустить задание с дублированным именем файла, Spring Batch сообщит о то, что экземпляр задания уже существует (поскольку параметр задания повторился)
	BATCH_JOB_PARAMS	Таблица BATCH_JOB_PARAMS хранит параметры для задания
Выполнение задания	BATCH_JOB_EXECUTION	Таблица BATCH_JOB_EXECUTION хранит информацию о выполнении каждого экземпляра задания. Экземпляр пакетного задания может иметь более одного выполнения (например, первое выполнение завершилось сбоем, после чего было успешно перезапущено во втором выполнении)
	BATCH_JOB_EXECUTION_CONTEXT	Хранит метаданные для выполнения пакетного задания
Выполнение шага	BATCH_STEP_EXECUTION	Хранит информацию о выполнении для каждого шага (другими словами, чтения, обработки, записи и специальных задач) в рамках выполнения каждого задания
	BATCH_STEP_EXECUTION_CONTEXT	Хранит метаданные для выполнения шага

Политики выполнения заданий

Spring Batch поддерживает множество разных стратегий обработки ошибок и выполнения, которые удовлетворяют почти всем требованиям к пакетным заданиям корпоративного уровня. В табл. 20.3 перечислены некоторые часто используемые политики.

Таблица 20.3. Политики выполнения заданий

Политика	Описание
Пропуск поврежденных записей	Spring Batch можно указать на необходимость пропуска проблемных записей (к примеру, записей, не прошедших проверку достоверности) вместо того, чтобы отбрасывать целую порцию записей или даже файл
Повтор в случае ошибки	Spring Batch поддерживает политику повторения в случае возникновения ошибок. Например, шаг обработки элемента может повторяться несколько раз (в нем мог быть произведен вызов какой-нибудь внешней службы для проверки данных, но по причине высокой загрузки внешняя служба не ответила в течение определенного времени) перед тем, как даст сбой
Многопоточная обработка шагов	Шаг может быть сконфигурирован для запуска в многопоточной среде с целью повышения производительности
Параллельная обработка	Шаги внутри задания могут быть сконфигурированы для параллельного выполнения с целью дальнейшего повышения производительности
Разбиение шагов на части	Spring Batch предоставляет SPI-интерфейс, который поддерживает разбиение на части шага. В этом случае сам шаг называется *ведущим*, а части, на которые он разбит — *ведомыми*. Ведомые шаги могут выполняться либо локальными потоками, либо удаленно для улучшения производительности
Удаленное разбиение на порции	В Spring Batch допускается делегировать обработку порций множеству машин. Например, ведущая машина может выполнять процесс чтения, а несколько ведомых машин осуществлять запись данных на основе порций (через интерфейс `ChunkProvider<T>`). Это обеспечивает очень гибкое решение с горизонтальным масштабированием для выполнения пакетных заданий

Реализация пакетного задания

Давайте посмотрим на Spring Batch в действии, реализовав простое пакетное задание. В приложении контактов, разработанном в главе 18, каждый контакт содержит базовую информацию, такую как имя, фамилия и дата рождения. Кроме того, с каждым контактом связано ноль или более хобби. Серверная часть для базовых операций CRUD уже доступна после импорта примера проекта, упомянутого в начале главы.

В этом разделе мы реализуем пакетное задание для импорта информации о контактах из XML-файла в базу данных. Для средств маршализации и демаршализации XML будет использоваться библиотека Castor (`http://castor.codehaus.org`).

В последующих разделах мы рассмотрим все шаги реализации пакетного задания. Эти шаги включают добавление обязательных зависимостей, конфигурирование пакетного задания, установку формата файла и т.д.

Добавление обязательных зависимостей

Сначала мы должны добавить в проект обязательные зависимости для Spring Batch и Castor, которые описаны в табл. 20.4.

Таблица 20.4. Зависимости Maven для Spring Batch и Castor

Идентификатор группы	Идентификатор артефакта	Версия	Описание
org.springframework.batch	spring-batch-core	2.1.8.RELEASE	Библиотека ядра Spring Batch. В этом модуле упакованы классы, которые поддерживают конфигурацию пакетных заданий, средства запуска заданий и т.д.
org.springframework.batch	spring-batch-infrastructure	2.1.8.RELEASE	Библиотека инфраструктуры Spring. В этом модуле упакованы классы, которые поддерживают различные средства чтения и записи (например, в файл, базу данных и т.д.)
org.springframework	spring-oxm	3.1.0.RELEASE	Модуль Spring OXM для поддержки преобразования между объектами POJO и форматом XML
org.codehaus.castor	castor-xml	1.3.2	Библиотека Castor XML

Добавьте в свой проект все зависимости, перечисленные в табл. 20.4.

Конфигурирование инфраструктуры Spring Batch

Для использования Spring Batch требуется небольшая установка и конфигурирование. Во-первых, в процесс инициализации базы данных должен быть включен сценарий для создания таблиц метаданных, которые будут хранить операционные данные Spring Batch.

В приложении контактов применяется встроенная база данных H2, которая определена в файле /src/main/resources/datasource-tx-jpa.xml. В листинге 20.1 показан фрагмент кода из этого файла с необходимыми изменениями.

Листинг 20.1. Добавление сценария создания схемы для Spring Batch

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"

   <!-- Остальной код не показан -->

   <jdbc:embedded-database id="dataSource" type="H2">
      <jdbc:script location="classpath:schema.sql"/>
      <jdbc:script
        location="classpath:/org/springframework/batch/core/schema-h2.sql"/>
      <jdbc:script location="classpath:test-data.sql"/>
   </jdbc:embedded-database>

   <!-- Остальной код не показан -->

</beans>
```

В листинге 20.1 изменения выделены полужирным. В Spring Batch код DDL для создания структуры метаданных хранится в модуле `spring-batch-core` из пакета `org.springframework.batch.core`. Поскольку мы работаем с H2, нас интересует файл `schema-h2.sql`. Предусмотрены также сценарии для всех наиболее часто используемых баз данных.

Во-вторых, понадобится сконфигурировать компоненты инфраструктуры Spring Batch. Как минимум, обязательными компонентами являются бины репозитория заданий и запуска заданий. Содержимое конфигурационного файла `/src/main/resources/batch-context.xml` приведено в листинге 20.2.

Листинг 20.2. Конфигурация Spring Batch

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
   xmlns:context="http://www.springframework.org/schema/context"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
     http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
     http://www.springframework.org/schema/context
     http://www.springframework.org/schema/context/spring-context-3.1.xsd">

   <bean id="jobRepository" class="org.springframework.batch.core.repository
.support.JobRepositoryFactoryBean">
       <property name="databaseType" value="h2" />
       <property name="dataSource" ref="dataSource" />
       <property name="transactionManager" ref="transactionManager" />
       <property name="isolationLevelForCreate" value="ISOLATION_DEFAULT"/>
   </bean>

   <bean id="jobLauncher"
class="org.springframework.batch.core.launch.support.SimpleJobLauncher">
       <property name="jobRepository" ref="jobRepository" />
   </bean>
</beans>
```

В листинге 20.2 определены два бина Spring Batch. Первый из них, бин `jobRepository`, определен с классом `JobRepositoryFactoryBean`, который будет создавать экземпляр класса `org.springframework.batch.core.repository.support.SimpleJobRepository`, используя DAO-реализации Spring Batch.

Класс `SimpleJobRepository` реализует интерфейс `org.springframework.batch.core.repository.JobRepository`, который сохраняет экземпляры `JobInstance`, `JobExecution` и `StepExecution` с применением внедренных объектов DAO.

Второй бин, `jobLauncher`, определен с классом реализации `SimpleJobLauncher`, который реализует интерфейс `org.springframework.batch.core.launch.JobLauncher`. Класс `SimpleJobLauncher` для запуска заданий использует интерфейс `TaskExecutor` из Spring Core. По умолчанию для запуска заданий будет создан экземпляр синхронного средства выполнения задач (класса `SyncTaskExecutor`). Бин `jobRepository` будет внедрен в бин `jobLauncher`, поэтому детали выполнения заданий могут обновляться в хранилище метаданных Spring Batch.

Реализация задания импорта контактов

Теперь можно приступить к реализации пакетного задания. Для начала определим формат файла XML. В листинге 20.3 показан пример XML-файла с информацией о контакте (файл `contacts.xml` в корневой папке проекта).

Листинг 20.3. Файл contacts.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<contacts>
   <contact>
      <firstName>Kim Fung</firstName>
      <lastName>Ho</lastName>
      <birthDate>1980-07-30</birthDate>
      <hobby>
         <hobbyId>Swimming</hobbyId>
      </hobby>
      <hobby>
         <hobbyId>Movies</hobbyId>
      </hobby>
   </contact>
</contacts>
```

В листинге 20.3 информацией о контакте помещена в дескриптор `<contact>`. Корневым дескриптором является `<contacts>`, внутри которого может размещаться дополнительная информация о контактах, предназначенная для импорта в базу данных.

Для отображения XML на POJO мы будем применять библиотеку Castor. Файл определения отображений (`/src/main/resources/oxm-mapping.xml`) представлен в листинге 20.4.

Листинг 20.4. Файл определения отображений для Castor XML

```
<mapping>
   <class name="com.apress.prospring3.ch20.domain.Contacts">
    <field name="contacts" type="com.apress.prospring3.ch20.domain.Contact"
           collection="arraylist">
       <bind-xml name="contact"/>
    </field>
   </class>
   <class name="com.apress.prospring3.ch20.domain.Contact" identity="id">
      <map-to xml="contact" />
      <field name="id" type="long">
         <bind-xml name="id" node="element"/>
      </field>
      <field name="firstName" type="string">
         <bind-xml name="firstName" node="element" />
      </field>
      <field name="lastName" type="string">
         <bind-xml name="lastName" node="element" />
      </field>
      <field name="birthDate" type="string" handler="dateHandler">
         <bind-xml name="birthDate" node="element" />
      </field>
      <field name="version" type="integer">
         <bind-xml name="version" node="element" />
      </field>
      <field name="hobbies" type="com.apress.prospring3.ch20.domain.Hobby"
            collection="set">
         <bind-xml name="hobby"/>
      </field>
   </class>
```

```
   <class name="com.apress.prospring3.ch20.domain.Hobby" identity="hobbyId">
      <map-to xml="hobby" />
      <field name="hobbyId" type="string">
         <bind-xml name="hobbyId" node="element" />
      </field>
   </class>
   <field-handler name="dateHandler"
        class="com.apress.prospring3.ch20.xml.handler.DateTimeFieldHandler">
      <param name="date-format" value="yyyy-MM-dd"/>
   </field-handler>
</mapping>
```

Если вы читали раздел “Использование веб-служб REST в Spring” в главе 16, то приведенное определение отображений должно выглядеть знакомым. В нем отображены объекты предметной области `Contacts`, `Contact` и `Hobby`. Кроме того, отображено свойство `hobbies` класса `Contact`, которое представляет собой набор (`Set`) объектов `Hobby`.

Так как для свойства даты рождения применяется тип `DateTime`, поддерживаемый Joda-Time, мы должны определить специальный обработчик, код которого показан в листинге 20.5.

Листинг 20.5. Класс реализации специального обработчика полей Castor XML

```
package com.apress.prospring3.ch20.xml.handler;

import java.util.Properties;

import org.exolab.castor.mapping.*;
import org.joda.time.DateTime;
import org.joda.time.format.DateTimeFormat;
import org.joda.time.format.DateTimeFormatter;

public class DateTimeFieldHandler extends GeneralizedFieldHandler {

   private static String dateFormatPattern;

   public void setConfiguration(Properties config) throws ValidityException {
      dateFormatPattern = config.getProperty("date-format");
   }

   public Object convertUponGet(Object value) {
      DateTime dateTime = (DateTime) value;
      return format(dateTime);
   }

   public Object convertUponSet(Object value) {
      String dateTimeString = (String) value;
      return parse(dateTimeString);
   }

   public Class<DateTime> getFieldType() {
      return DateTime.class;
   }

   protected static String format(final DateTime dateTime) {

      String dateTimeString = "";

      if (dateTime != null) {
```

```
            DateTimeFormatter dateTimeFormatter
                = DateTimeFormat.forPattern(dateFormatPattern);
            dateTimeString = dateTimeFormatter.print(dateTime);
        }

        return dateTimeString;
    }

    protected static DateTime parse(final String dateTimeString) {

        DateTime dateTime = new DateTime();

        if (dateTimeString != null) {
            DateTimeFormatter dateTimeFormatter
              = DateTimeFormat.forPattern(dateFormatPattern);
            dateTime = dateTimeFormatter.parseDateTime(dateTimeString);
        }

        return dateTime;

    }

}
```

Специальный обработчик полей в основном такой же, как был реализован в главе 16 для поддержки взаимодействия с веб-службой REST в формате XML.

Следующим шагом является конфигурирование `ApplicationContext` для задания импорта контактов. Содержимое конфигурационного файла `/src/main/resources/app-context.xml` приведено в листинге 20.6.

Листинг 20.6. Конфигурация задания импорта контактов Spring Batch

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:context="http://www.springframework.org/schema/context"
   xmlns:batch="http://www.springframework.org/schema/batch"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
     http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
     http://www.springframework.org/schema/context
     http://www.springframework.org/schema/context/spring-context-3.1.xsd
     http://www.springframework.org/schema/batch
     http://www.springframework.org/schema/batch/spring-batch-2.1.xsd">

   <import resource="classpath:datasource-tx-jpa.xml"/>
   <import resource="classpath:batch-context.xml"/>

   <context:component-scan base-package="com.apress.prospring3.ch20.service.jpa,
           com.apress.prospring3.ch20.batch"/>

   <bean id="validator"
class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean"/>

   <!-- Бин маршализации XML -->
   <bean id="batchMarshaller"
         class="org.springframework.oxm.castor.CastorMarshaller">
      <property name="mappingLocation" value="classpath:oxm-mapping.xml"/>
   </bean>

   <!-- Пакетное задание: импорт контактов -->
   <batch:job id="importContactsJob">
      <batch:step id="readWriteStep">
```

```
            <batch:tasklet transaction-manager="transactionManager">
               <batch:chunk
                  reader="contactItemReader"
                  processor="contactItemProcessor"
                  writer="contactItemWriter"
                  commit-interval="100"/>
            </batch:tasklet>
         </batch:step>
         <batch:listeners>
            <batch:listener ref="importContactJobListener"/>
         </batch:listeners>
      </batch:job>

      <bean id="contactItemReader"
class="org.springframework.batch.item.xml.StaxEventItemReader" scope="step">
        <property name="resource" value="file:///#{jobParameters['inputFile']}"/>
        <property name="fragmentRootElementName" value="contact"/>
        <property name="unmarshaller" ref="batchMarshaller"/>
      </bean>

      <bean id="contactItemWriter"
            class="org.springframework.batch.item.adapter.ItemWriterAdapter">
         <property name="targetObject" ref="contactService" />
         <property name="targetMethod" value="save" />
      </bean>

      <bean id="batchValidator"
class="com.apress.prospring3.ch20.batch.validator.BeanValidationValidator" />

      <bean id="contactItemProcessor"
class="org.springframework.batch.item.validator.ValidatingItemProcessor">
         <property name="validator" ref="batchValidator" />
      </bean>

      <bean id="importContactJobListener"
class="com.apress.prospring3.ch20.batch.listener.ImportContactJobListener"/>

</beans>
```

Для упрощения определения задания в листинге 20.6 объявлено пространство имен `batch`. Внутри конфигурации импортированы обязательные файлы (`datasource-tx-jpa.xml`, `batch-context.xml`).

Затем определяется бин `validator`, поскольку мы будем использовать JSR-303 для проверки достоверности данных в обработчике элементов (за дополнительными сведениями обращайтесь в главу 14). Для Castor объявлен бин `batchMarshaller`, который будет выполнять маршализацию и демаршализацию между POJO и XML.

Бин `importContactsJob` определяет пакетное задание импорта контактов. В определении задания задан только один шаг, который содержит один тасклет (tasklet) и обработчик порции. Внутри обработки порции свойства `reader`, `processor` и `writer` ссылаются на, соответственно, `contactItemReader`, `contactItemProcessor` и `contactItemWriter`. В свойстве `commit-interval` указан размер порции, который в рассматриваемом случае составляет `100`.

Для бина чтения `contactItemReader` применяется класс `StaxEventItemReader<T>`, который представляет собой поддержку Spring Batch чтения данных в формате XML. Свойству `resource` передается параметр задания по имени `inputFile` (имя входного файла). В свойстве `fragmentRootElementName` определен дескриптор, соответствующий информации о каждом контакте; в данном случае это дескриптор `<contact>`.

В свойстве `unmarshaller` указан бин демаршализации XML, которым в этом случае является бин `batchMarshaller` из библиотеки Castor.

Для бина записи `contactItemWriter` используется класс `ItemWriterAdapter<T>`, который представляет собой поддержку Spring Batch для вызова метода существующего бина Spring, реализующего запись элемента. В показанной выше конфигурации применяется метод `save()` бина `contactService`. Как видите, Spring Batch может гладко интегрироваться с существующим `ApplicationContext`.

В среде STS редактор конфигурации Spring (Spring Config Editor) также предоставляет графический вид конфигурации пакетного задания. Чтобы увидеть его, откройте файл `app-context.xml` в редакторе конфигурации и перейдите на вкладку batch-graph. На рис. 20.5 показана эта вкладка для задания импорта контактов.

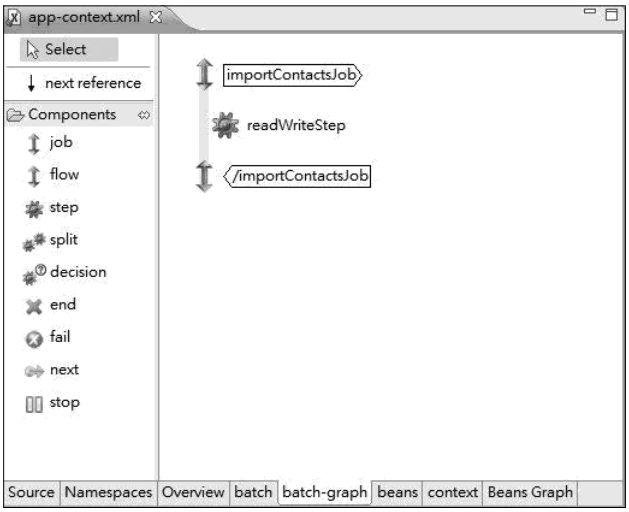


Рис. 20.5. Графический вид пакетного задания Spring в STS

Для бина обработки `contactItemProcessor` применяется класс `ValidatingItemProcessor<T>`, который представляет собой поддержку Spring Batch обработки с проверкой достоверности на уровне элементов. В рассматриваемой конфигурации внедряется бин класса `BeanValidationValidator`, содержимое которого приведено в листинге 20.7.

Листинг 20.7. Класс `BeanValidationValidator`

```
package com.apress.prospring3.ch20.batch.validator;

import java.util.HashSet;
import java.util.Set;

import javax.validation.ConstraintViolation;

import org.springframework.batch.item.validator.ValidationException;
import org.springframework.batch.item.validator.Validator;
import org.springframework.beans.factory.annotation.Autowired;

public class BeanValidationValidator<Contact> implements Validator<Contact> {

    @Autowired
    private javax.validation.Validator validator;
```

```
    public void validate(Contact value) throws ValidationException {
        Set<ConstraintViolation<Contact>> violations =
            new HashSet<ConstraintViolation<Contact>>();
        violations = validator.validate(value);
        if(!violations.isEmpty()) {
            throw new ValidationException("Validation failed for " + value
                + ": " + violationsToString(violations));
        }
    }

    private String
          violationsToString(Set<ConstraintViolation<Contact>> violations) {

        StringBuilder violationMessage = new StringBuilder();

        for (ConstraintViolation<Contact> violation: violations) {
            violationMessage.append(violation.getMessage() + "|");
        }

        return violationMessage.toString();
    }
}
```

Класс в листинге 20.7 реализует интерфейс `org.springframework.batch.item.validator.Validator<T>`, который поддерживает проверку достоверности на уровне элементов в Spring Batch. Внутри класса автосвязан бин `validator`, соответствующий стандарту JSR-303. В методе `validate()` этому бину будет передаваться элемент типа `Contact` для выполнения проверки достоверности. Затем объект `Contact` будет проверен на предмет нарушения ограничений, и если нарушения существуют, будет сгенерировано исключение типа `ValidationException`.

В листинге 20.6 обратите внимание, что в бине `importContactsJob` также определен слушатель заданий в форме бина по имени `importContactJobListener`. Слушатель заданий — это мощное средство Spring Batch, позволяющее выполнять специальную логику на каждой фазе жизненного цикла обработки пакетного задания. В этом случае слушатель заданий просто выводит список всех контактов после завершения задания, чтобы удостовериться в успешном импорте. Класс слушателя заданий показан в листинге 20.8.

Листинг 20.8. Класс `ImportContactJobListener`

```
package com.apress.prospring3.ch20.batch.listener;

import java.util.List;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.batch.core.JobExecution;
import org.springframework.batch.core.JobExecutionListener;
import org.springframework.beans.factory.annotation.Autowired;

import com.apress.prospring3.ch20.domain.Contact;
import com.apress.prospring3.ch20.service.ContactService;

public class ImportContactJobListener implements JobExecutionListener {

    private static final Logger logger = LoggerFactory.getLogger(ImportContact
JobListener.class);

    @Autowired
    private ContactService contactService;
```

```
    @Override
    public void afterJob(JobExecution arg0) {
        List<Contact> contacts = contactService.findAll();
        for (Contact contact: contacts) {
            logger.info("Contact: {}", contact);
        }
    }

    @Override
    public void beforeJob(JobExecution arg0) {
    }
}
```

Класс в листинге 20.8 реализует интерфейс `JobExecutionListener`. В методе `afterJob()` вызывается метод `ContactService.findAll()`, а затем список контактов выводится на консоль. Обратите внимание, что слушателю также передается экземпляр класса `JobExecution`, поэтому вся информация о выполнении задания доступна для обработки.

Теперь можно запустить пакетное задание. В листинге 20.9 показан класс, предназначенный для запуска задания.

Листинг 20.9. Выполнение пакетного задания

```
package com.apress.prospring3.ch20;

import java.util.HashMap;
import java.util.Map;

import org.springframework.batch.core.*;
import org.springframework.batch.core.launch.*;
import org.springframework.context.support.GenericXmlApplicationContext;

public class ContactJobTest {

  public static void main(String[] args) {

    GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
    ctx.load("classpath:app-context.xml");
    ctx.refresh();

    JobLauncher jobLauncher = ctx.getBean("jobLauncher", JobLauncher.class);

    Job contactJob = ctx.getBean("importContactsJob", Job.class);

    Map<String, JobParameter> jobParamMap = new HashMap<String, JobParameter>();
    jobParamMap.put("inputFile", new JobParameter("C:/temp/contacts.xml"));

    try {
        jobLauncher.run(contactJob, new JobParameters(jobParamMap));
    } catch (Exception ex) {
        ex.printStackTrace();
    }
    // Задание импорта контактов завершилось успешно.
    System.out.println("Contact import job completed successfully");
  }
}
```

В листинге 20.9 бин запуска заданий и бины задания импорта контактов получаются из `ApplicationContext`. Затем конструируется экземпляр `Map<String, JobParameter>` с заполнением параметра `inputFile`. Наконец, пакетное задание запускается вместе со

своими параметрами. Перед запуском класса поместите копию файла `contacts.xml` в папку `C:/temp`, как указано в параметре задания. После запуска класса на консоли появится следующий вывод (показан не полностью):

```
INFO : org.springframework.batch.core.launch.support.SimpleJobLauncher -
No TaskExecutor has been set, defaulting to synchronous executor.
INFO : org.springframework.oxm.castor.CastorMarshaller -
Configured using [class path resource [oxm-mapping.xml]]
INFO : org.springframework.batch.core.launch.support.SimpleJobLauncher - Job:
[FlowJob: [name=importContactsJob]] launched with the following parameters:
[{inputFile=C:/temp/contacts.xml}]
INFO : org.springframework.batch.core.job.SimpleStepHandler - Executing step:
[readWriteStep]
Hibernate: insert into contact (ID, BIRTH_DATE, FIRST_NAME, LAST_NAME, VERSION)
values (null, ?, ?, ?, ?)
Hibernate: insert into contact_hobby_detail (CONTACT_ID, HOBBY_ID) values (?, ?)
Hibernate: insert into contact_hobby_detail (CONTACT_ID, HOBBY_ID) values (?, ?)
INFO : com.apress.prospring3.ch20.service.jpa.ContactServiceImpl -
Finding all contacts
Hibernate: select contact0_.ID as ID0_, contact0_.BIRTH_DATE as BIRTH2_0_,
contact0_.FIRST_NAME as FIRST3_0_, contact0_.LAST_NAME as LAST4_0_,
contact0_.VERSION as VERSION0_ from contact contact0_
INFO : com.apress.prospring3.ch20.batch.listener.ImportContactJobListener -
Contact: Contact - Id: 1, First name: Clarence, Last name: Ho,
Birthday: 1980-07-30T00:00:00.000+08:00
INFO : com.apress.prospring3.ch20.batch.listener.ImportContactJobListener -
Contact: Contact - Id: 2, First name: Scott, Last name: Tiger,
Birthday: 1990-11-02T00:00:00.000+08:00
INFO : com.apress.prospring3.ch20.batch.listener.ImportContactJobListener -
Contact: Contact - Id: 3, First name: Kim Fung, Last name: Ho,
Birthday: 1980-07-30T00:00:00.000+08:00
INFO : org.springframework.batch.core.launch.support.SimpleJobLauncher - Job:
[FlowJob: [name=importContactsJob]] completed with the following parameters:
[{inputFile=C:/temp/contacts.xml}] and the following status: [COMPLETED]
Contact import job completed successfully
```

Файл тестовых данных (`test-data.sql`) содержит две записи контактов, и в выводе видно, что контакт с идентификатором `3` был успешно импортирован из XML-файла.

Spring Batch обеспечивает прочный фундамент для удовлетворения сложных требований к пакетным заданиям. За дополнительными сведениями по этой теме рекомендуем обратиться к книге *Pro Spring Batch* (Apress, 2011 г.).

Использование Spring Batch и Spring Integration

В то время как Spring Batch предоставляет сложную среду для выполнения пакетных заданий, проект Spring Integration обеспечивает великолепную среду для интеграции корпоративных приложений. Благодаря Spring Integration, обмен информацией с внешними системами становится намного более простым, независимо от того, откуда эта информация поступает или куда она должна быть отправлена. Предоставляется готовая поддержка для взаимодействия на основе файлов, электронной почты и сообщений JMS.

Даже если начальные требования к выполнению пакетного задания очень просты, все равно рекомендуется установить Spring Batch для интеграции Spring Integration, чтобы обеспечить построение достаточно гибкой среды выполнения приложений, которая удовлетворит более сложные требования к взаимодействию в будущем.

В последующих разделах мы продемонстрируем совместной использование Spring Batch и Spring Integration для реализации механизма опроса файлов, при котором в случае появления файла в заданной папке Spring Integration автоматически запустит задание импорта контактов.

Введение в Spring Integration

Ниже перечислены некоторые ключевые концепции Spring Integration.

- Канал. Spring Integration — это инфраструктура взаимодействия на основе сообщений, и канал является заполнителем для сообщений. Каналы представляют собой связующие элементы для соединения различных генераторов и потребителей сообщений внутри приложения.
- Конечная точка сообщения. В Spring Integration конечная точка сообщения принимает форму адаптеров, которые подключают компоненты приложения к различным каналам. В задании импорта контактов мы будем использовать дескриптор `<inbound-channel-adapter>` для подключения компонента опроса файлов к каналу запросов пакетных заданий.
- Преобразователь. Преобразователь подключает разные каналы и выполняет необходимое преобразование сообщений.
- Активатор службы. В Spring Integration активатор службы — это конечная точка, которая вызывает бизнес-процесс приложения, передавая соответствующее сообщение.

На рис. 20.6 показано графическое представление совместного функционирования компонентов Spring Integration и Spring Batch при запуске задания импорта контактов в рамках механизма опроса файлов.

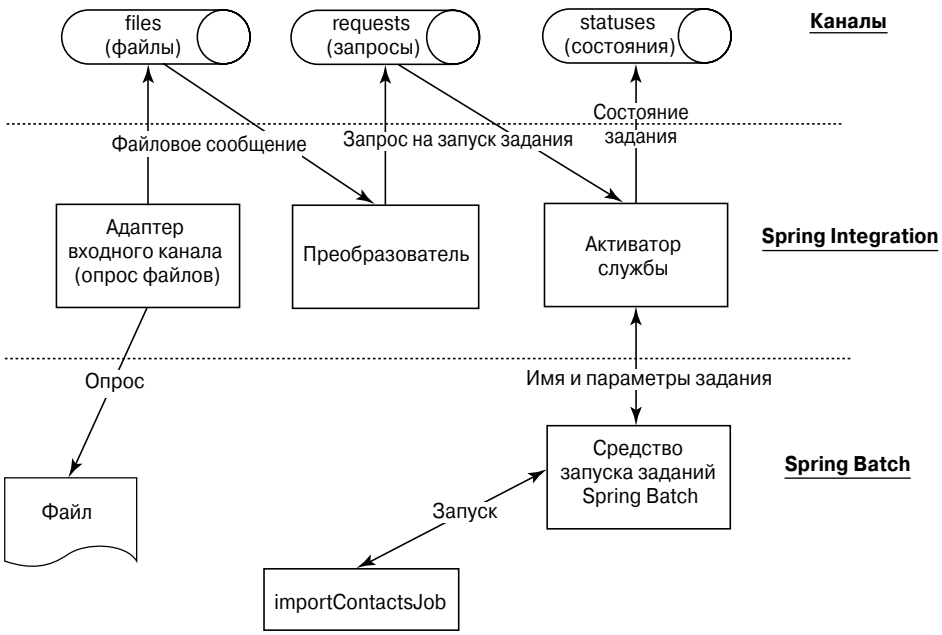


Рис. 20.6. Работа Spring Batch и Spring Integration

В верхней части рис. 20.6 можно видеть три канала, определенные для хранения сообщений, с именами `files`, `requests` и `statuses`.

Во-первых, адаптер входного канала из файловой поддержки Spring Integration используется для опроса файлов. При поступлении файла создается файловое сообщение, которое сохраняется в канале `files`.

Во-вторых, преобразователь выберет сообщение из канала `files`, выполнит трансформацию (в этом случае файловое сообщение будет преобразовано в сообщение запроса на запуск задания) и выведет трансформированное сообщение в канал `requests`. Активатор службы выберет сообщение запроса на запуск задания из канала `requests` и соответствующим образом запустит задание. Результат выполнения задания будет выведен в канал `statuses`.

Добавление обязательных зависимостей

В табл. 20.5 описаны обязательные зависимости для Spring Integration и поддержки Spring Integration в Spring Batch.

Таблица 20.5. Зависимости Maven для Spring Integration

Идентификатор группы	Идентификатор артефакта	Версия	Описание
`org.springframework.integration`	`spring-integration-core`	2.1.0.RELEASE	Библиотека ядра Spring Integration
`org.springframework.integration`	`spring-integration-file`	2.1.0.RELEASE	Поддержка Spring Integration для шаблонов взаимодействия ввода-вывода, связанных с файлами
`org.springframework.batch`	`spring-batch-integration`	1.2.1.RELEASE	Принадлежит проекту Spring Batch Admin, который предоставляет классы для работы с проектом Spring Integration. Например, в эту библиотеку упакованы классы для поддержки запуска заданий из Spring Integration

Добавьте зависимости, перечисленные в табл. 20.5, к своему проекту.

Реализация механизма опроса файлов

Реализация механизма опроса файлов в Spring Integration для запуска задания Spring Batch довольно проста.

Компоненты, показанные на рис. 20.6, понадобится сконфигурировать в `Application Context` (файл `batch-context.xml`). В листинге 20.10 показаны изменения, которые должны быть внесены в этот файл.

Листинг 20.10. Модифицированный файл `batch-context.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:integration="http://www.springframework.org/schema/integration"
   xmlns:file="http://www.springframework.org/schema/integration/file"
   xmlns:context="http://www.springframework.org/schema/context"
```

```
   xsi:schemaLocation="http://www.springframework.org/schema/beans
     http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
     http://www.springframework.org/schema/integration
     http://www.springframework.org/schema/integration/spring-integration.xsd
     http://www.springframework.org/schema/integration/file
     http://
www.springframework.org/schema/integration/file/spring-integration-file.xsd
     http://www.springframework.org/schema/context
     http://www.springframework.org/schema/context/spring-context-3.1.xsd">
   <!-- Остальной код не показан -->
   <!-- Конфигурация Spring Integration: начало -->
   <integration:channel id="files"/>
   <integration:channel id="requests"/>
   <integration:channel id="statuses">
      <integration:queue capacity="10"/>
   </integration:channel>
   <file:inbound-channel-adapter
      directory="file:C:/temp/contact"
      channel="files">
      <integration:poller id="poller" fixed-delay="5000"/>
   </file:inbound-channel-adapter>
   <integration:transformer input-channel="files" output-channel="requests">
      <bean
class="com.apress.prospring3.ch20.batch.integration.FileMessageToJobRequest">
         <property name="job" ref="importContactsJob"/>
         <property name="fileParameterName" value="inputFile"/>
      </bean>
   </integration:transformer>
   <integration:service-activator method="launch" input-channel="requests"
      output-channel="statuses">
      <bean id="messageHandler" class=
"org.springframework.batch.integration.launch.JobLaunchingMessageHandler">
         <constructor-arg ref="jobLauncher"/>
      </bean>
   </integration:service-activator>
   <!-- Конфигурация Spring Integration: конец -->
</beans>
```

Необходимые изменения в листинге 20.10 выделены полужирным. Сначала объявляются пространства имен `integration` и `file`, которые относятся к компонентам Spring Integration. Затем объявляются каналы.

После этого дескриптор `<file:inbound-channel-adapter>` используется для определения адаптера входного канала, который представляет поддержку Spring Integration взаимодействия на основе файлов. Атрибут `directory` определяет входную папку, а атрибут `channel` — канал, в который должно выводиться сообщение. Дескриптор `<integration:poller>` определяет механизм опроса с задержкой 5000 мс, которая означает, что адаптер будет опрашивать папку на предмет поступивших файлов каждые 5 секунд.

А теперь перейдем к части преобразователя. Дескриптор `<integration:transformer>` определяет преобразователь для файлового сообщения, со специальным классом реа-

лизации (FileMessageToJobRequest), который выполняет трансформацию файлового сообщения в сообщение запроса на запуск задания. Класс FileMessageToJobRequest показан в листинге 20.11.

Листинг 20.11. Класс FileMessageToJobRequest

```
package com.apress.prospring3.ch20.batch.integration;

import java.io.File;

import org.springframework.batch.core.Job;
import org.springframework.batch.core.JobParametersBuilder;
import org.springframework.batch.integration.launch.JobLaunchRequest;
import org.springframework.integration.Message;
import org.springframework.integration.annotation.Transformer;

public class FileMessageToJobRequest {

   private Job job;
   private String fileParameterName;

   public void setFileParameterName(String fileParameterName) {
      this.fileParameterName = fileParameterName;
   }

   public void setJob(Job job) {
      this.job = job;
   }

   @Transformer
   public JobLaunchRequest toRequest(Message<File> message) {
     JobParametersBuilder jobParametersBuilder = new JobParametersBuilder();
     jobParametersBuilder.addString(fileParameterName,
        message.getPayload().getAbsolutePath());
     return new JobLaunchRequest(job, jobParametersBuilder.toJobParameters());
   }
}
```

В листинге 20.11 метод toRequest() применяется с помощью аннотации @Transformer, которая указывает Spring Integration, что этот метод должен быть выполнен для трансформации сообщения. В классе FileMessageToJobRequest внедрены свойства job (пакетное задание для запуска) и fileParameterName (имя параметра задания для входного файла). Внутри метода toRequest() передаваемое сообщение типа File трансформируется в экземпляр класса JobLaunchRequest, после чего возвращается; затем оно будет выведено в канал requests.

Наконец, дескриптор <integration:service-activator> определяет активатор службы. Внутри его определения объявлен бин messageHandler с классом JobLaunchingMessageHandler, который представляет поддержку Spring Batch для запуска заданий из Spring Integration.

В среде STS редактор конфигурации Spring (Spring Config Editor) также предоставляет графический вид конфигурации взаимодействия. Чтобы увидеть его, перейдите в представлении редактора конфигурации Spring на вкладку integration-graph. Пример конфигурации взаимодействия показан на рис. 20.7.

Теперь можно протестировать механизм опроса файлов. В листинге 20.12 приведена тестовая программа.

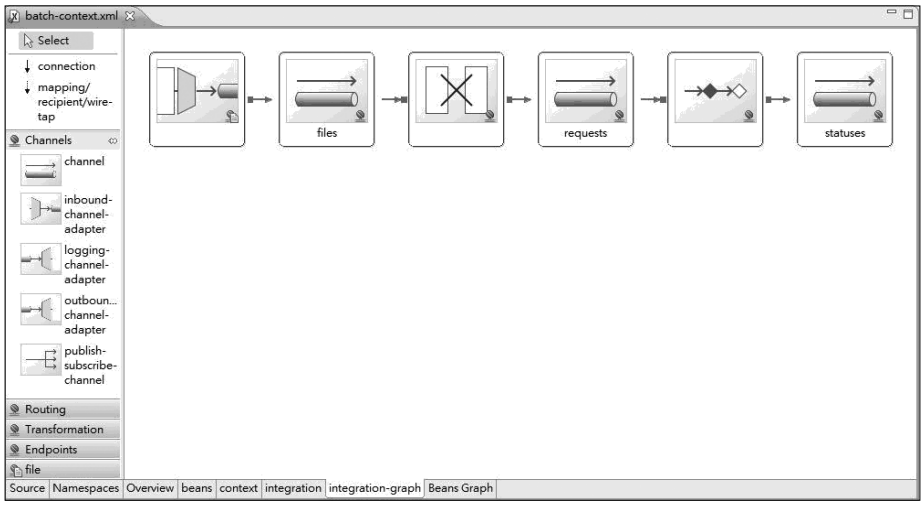


Рис. 20.7. Графический вид конфигурации Spring Integration в STS

Листинг 20.12. Тестирование механизма опроса файлов

```
package com.apress.prospring3.ch20;

import org.springframework.context.support.GenericXmlApplicationContext;

public class BatchIntegrationTest {

   public static void main(String[] args) {

      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:app-context.xml");
      ctx.refresh();

      // Контекст приложения инициализирован успешно.
      System.out.println("App context initialized successfully");

      while (true) {
      }
   }
}
```

В листинге 20.12 нужно лишь выполнить начальную загрузку `ApplicationContext` и начать бесконечный цикл. Spring Integration будет выполнять опрос файлов в папке `C:/temp/contact`, которая определена в бине адаптера входного канала. Перед тестированием поместите файл `contacts.xml` в папку `C:/temp/contact` и вы увидите в консоли STS, что задание запустится автоматически. Ниже показана часть вывода:

```
INFO : org.springframework.integration.file.FileReadingMessageSource -
Created message: [[Payload=\temp\contact\contacts.xml][Headers={timestamp=13
26274023267, id=17cc6b63-5694-4160-a2d6-a75430fa7515}]]
INFO : org.springframework.batch.core.launch.support.SimpleJobLauncher - Job:
[FlowJob: [name=importContactsJob]] launched with the following parameters:
[{inputFile=C:\temp\contact\contacts.xml}]
INFO : org.springframework.batch.core.job.SimpleStepHandler -
Executing step: [readWriteStep]
Hibernate: insert into contact (ID, BIRTH_DATE, FIRST_NAME, LAST_NAME,
VERSION) values (null, ?, ?, ?, ?)
```

```
Hibernate: insert into contact_hobby_detail (CONTACT_ID, HOBBY_ID) values (?, ?)
Hibernate: insert into contact_hobby_detail (CONTACT_ID, HOBBY_ID) values (?, ?)
INFO : com.apress.prospring3.ch20.service.jpa.ContactServiceImpl -
Finding all contacts
Hibernate: select contact0_.ID as ID0_, contact0_.BIRTH_DATE as
BIRTH2_0_, contact0_.FIRST_NAME as FIRST3_0_, contact0_.LAST_NAME as
LAST4_0_, contact0_.VERSION as VERSION0_ from contact contact0_
INFO : com.apress.prospring3.ch20.batch.listener.ImportContactJobListener
- Contact: Contact - Id: 1, First name: Clarence, Last name: Ho, Birthday:
1980-07-30T00:00:00.000+08:00
INFO : com.apress.prospring3.ch20.batch.listener.ImportContactJobListener -
Contact: Contact - Id: 2, First name: Scott, Last name: Tiger, Birthday:
1990-11-02T00:00:00.000+08:00
INFO : com.apress.prospring3.ch20.batch.listener.ImportContactJobListener -
Contact: Contact - Id: 3, First name: Kim Fung, Last name: Ho, Birthday:
1980-07-30T00:00:00.000+08:00
INFO : org.springframework.batch.core.launch.support.SimpleJobLauncher - Job:
[FlowJob: [name=importContactsJob]] completed with the following parameters:
[{inputFile=C:\temp\contact\contacts.xml}] and the following status:
[COMPLETED]
```

В выводе видно, что сообщение было создано, задание импорта контактов запущено и успешно завершено.

Сочетание Spring Batch и Spring Integration формирует мощную среду для выполнения пакетных заданий и взаимодействия корпоративных приложений, которая интегрируется с уровнем обслуживания Spring-приложения, обеспечивая согласованное выполнение бизнес-логики.

За дополнительными сведениями по Spring Integration рекомендуем обратиться к книге *Pro Spring Integration* (Apress, 2011 г.).

Введение в Spring Roo

Еще одним интересным проектом в рамках Spring является Spring Roo (`www.springsource.org/spring-roo`). Этот проект предоставляет платформу быстрой разработки приложений следующего поколения, которая значительно улучшает продуктивность разработчиков Spring-приложений.

Проект Spring Roo предлагает новый интерфейс командной строки с полезными советами, которые помогают быстро определить объектную модель предметной области (Domain Object Model — DOM), уровень постоянства, уровень обслуживания и уровень презентаций. На уровне презентаций Spring Roo обеспечивает первоклассную поддержку JPA 2, а в версии Spring Roo 1.2 была добавлена поддержка и абстракции репозитория Spring Data JPA.

Благодаря Spring Roo, появляется возможность иметь логику постоянства, встроенную в сущностные классы JPA, или же можно с помощью Roo сгенерировать уровень обслуживания для хранения бизнес-логики. В Spring Roo поддерживается JSR-303.

На уровне презентаций Roo обеспечивает первоклассную поддержку Spring MVC и может автоматически генерировать базовый пользовательский интерфейс с поддержкой шаблонов (с помощью Apache Tiles), оформления темами, интернационализацией, проверкой достоверности и т.п. Можно также указать Roo на необходимость генерации уровня презентаций с применением Spring Web Flow. Версия Spring Roo 1.2.0 в случае использования Spring MVC будет генерировать представления как JSPX-станицы. Кроме того, предоставляется поддержка JSF с PrimeFaces в качестве библиотеки компонентов.

Spring Roo также поддерживает реализацию расширений через свою архитектуру дополнений. На момент написания этой книги существовало множество дополнений (правда, некоторые из них все еще находились на ранней стадии разработки), включая дополнения для GWT, Vaadin, Adobe Flex, jQuery и т.п. Также доступны дополнения, которые поддерживают другие проекты Spring, в том числе Spring Security, Spring Integration и т.д. STS также имеет готовую поддержку Spring Roo. В STS доступна опция создания проекта Spring Roo, и также предоставляется командная оболочка Roo Shell.

Приложения, генерируемые Spring Roo, являются настраиваемыми. Можно переопределять стандартное поведение любого уровня, сгенерированного Roo, самостоятельно управлять выполняемой логикой или добавлять собственную логику в классы, генерируемые Roo; проектное решение Roo обеспечивает четкое разделение логики, генерируемой Roo, и логики, реализуемой разработчиками.

В последующих разделах мы представим высокоуровневый обзор проекта Spring Roo и посмотрим на него в действии, разработав простое приложение контактов.

Конфигурирование Spring Roo в STS

На момент написания книги набор STS 2.8.1 поставлялся в комплекте с версией Spring Roo 1.1.5. Однако в версии Spring Roo 1.2.0 появилось множество интересных возможностей, поэтому мы будем использовать ее в этом разделе. Таким образом, понадобится загрузить и установить Spring Roo 1.2.0, после чего сконфигурировать STS для работы с новой версией Spring Roo.

Загрузите дистрибутив Spring Roo 1.2.0, доступный по адресу `http://s3.amazonaws.com/dist.springframework.org/release/ROO/spring-roo-1.2.0.RELEASE.zip` (либо воспользуйтесь центром загрузки сообщества SpringSource по адресу `www.springsource.org/spring-community-download`), и распакуйте файл на своем компьютере. Выберите в STS пункт меню Window⇨Preferences (Окно⇨Параметры), а затем элемент Spring⇨Roo Support (Spring⇨Поддержка Roo). Вы увидите, что по умолчанию установлена версия 1.1.5. Для добавления версии Spring Roo 1.2.0 щелкните на кнопке Add (Добавить) и укажите папку, в которой был распакован дистрибутив Roo (рис. 20.8).

Создание проекта Spring Roo

Создать проект Spring Roo в STS исключительно легко. Выберите пункт меню File⇨New⇨Spring Roo Project (Файл⇨Создать⇨Проект Spring Roo), как показано на рис. 20.9.

По завершении удостоверьтесь, что версией по умолчанию является Spring Roo 1.2.0, и щелкните на кнопке OK. Теперь все готово к созданию проекта Spring Roo.

Введите сведения о проекте (рис. 20.10).

После этого щелкните на кнопке Next (Далее), затем на Finish (Готово), и STS создаст проект. Кроме того, STS автоматически откроет вкладку Roo Shell (Командная оболочка Roo), которая показана на рис. 20.11.

А теперь перейдем к конфигурированию DOM-модели.

Настройка уровня постоянства и сущностного класса

Для DOM-моделей в Spring Roo предлагается первоклассная поддержка сущностей JPA. Первый шаг заключается в настройке уровня постоянства и соответствующих сущностных классов, которые требуются приложению.

В этом примере мы будем настраивать уровень постоянства, используя базу данных в памяти H2 в качестве серверной части и Hibernate в качестве поставщика постоянства JPA.

Для этого введите следующую команду в оболочке Roo Shell:

```
persistence setup --database H2_IN_MEMORY --provider HIBERNATE
```

Команда `persistence setup` указывает Spring Roo на необходимость создания уровня постоянства для разрабатываемого приложения. В параметрах команды указывается база данных H2 и поставщик Hibernate. Все параметры в оболочке Roo Shell предваряются символами `--`; кроме того, всегда можно нажать клавиатурную комбинацию <Ctrl+пробел> и получить помощь по коду и подсказки. После нажатия <Enter> уровень постоянства будет сгенерирован Roo. В проводнике пакетов (Package Explorer) можно просмотреть структуру проекта, которая показана на рис. 20.12.

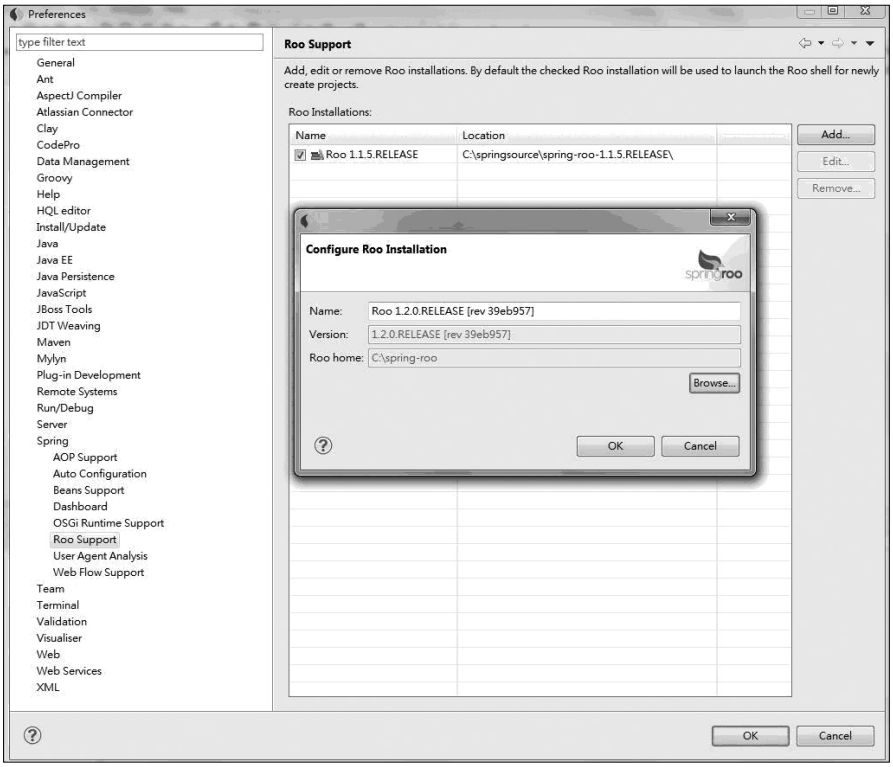


Рис. 20.8. Конфигурирование Spring Roo в STS

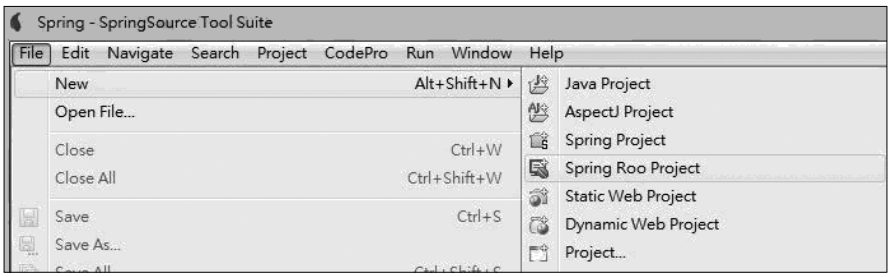


Рис. 20.9. Создание проекта Spring Roo

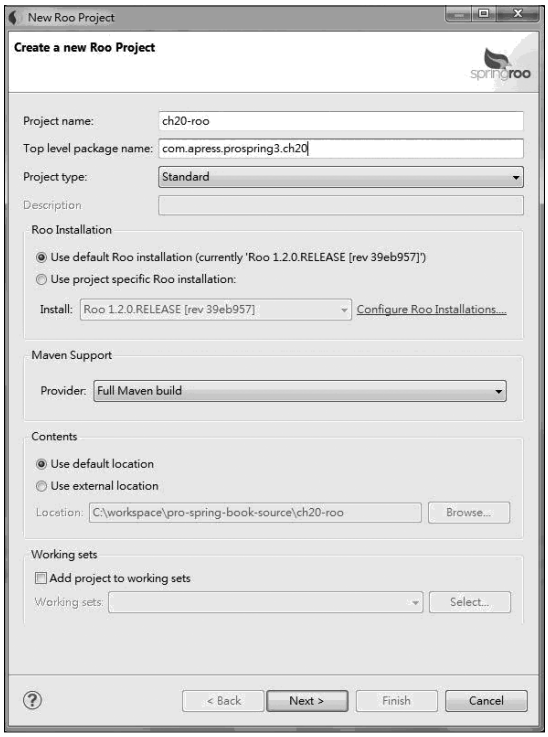


Рис. 20.10. Сведения о проекте Spring Roo

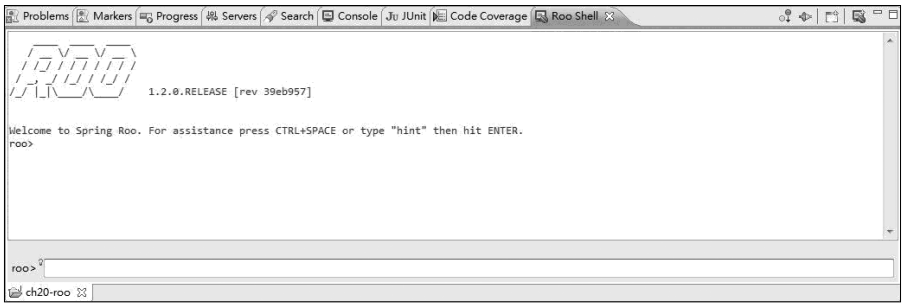


Рис. 20.11. Командная оболочка Roo в STS

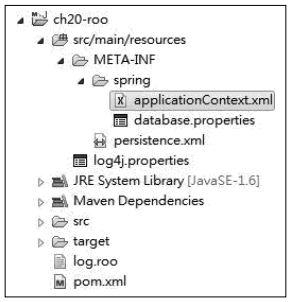


Рис. 20.12. Проект Spring Roo с уровнем постоянства

Spring Roo были добавлены обязательные зависимости Maven и сконфигурирован контекст `ApplicationContext` с требуемыми компонентами, информацией подключения к базе данных (в файле `database.properties`) и файлом `persistence.xml` для JPA.

Следующий шаг — создание сущностного класса `Contact`. Чтобы создать этот класс, введите следующую команду:

```
entity jpa --class ~.domain.Contact
```

Эта команда указывает, что мы хотим создать сущностный класс JPA с именем пакета `com.apress.prospring3.ch20` (символ ~ соответствует базовому пакету, который был определен при создании проекта), именем подпакета `domain` и именем класса `Contact`.

После этого Roo сгенерирует сущностный класс и несколько классов AspectJ. Следует отметить, что на момент написания этой книги с оболочкой Roo Shell была связана небольшая проблема: она не добавляла путь `/src/main/java` к исходной папке проекта, и это нужно было делать вручную. Чтобы добавить указанный путь, щелкните правой кнопкой мыши на проекте в STS и выберите в контекстном меню пункт Properties⇨Java Build Path (Свойства⇨Путь сборки Java). Затем щелкните на кнопке Add Folder (Добавить папку) и отметьте флажок возле папки `/src/main/java`, как показано на рис. 20.13.

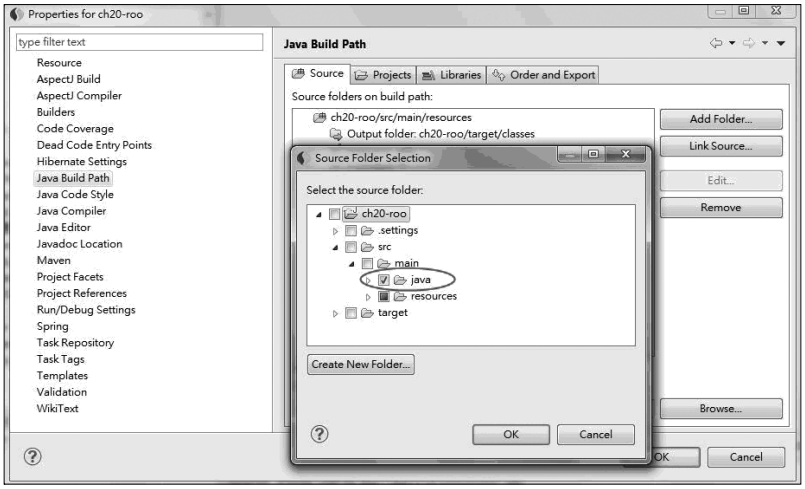


Рис. 20.13. Конфигурирование пути сборки Java для проекта Spring Roo

Если вы заглянете в проводник пакетов, то увидите только класс `Contact`, но не другие классы AspectJ, сгенерированные Roo. Причина в том, что по умолчанию представление Package Explorer (Проводник пакетов) фильтрует эти классы. Чтобы увидеть их, щелкните на меню View (Вид), представленное в виде треугольного значка и выберите пункт Filters (Фильтры), как показано на рис. 20.14.

В диалоговом окне Java Element Filters (Фильтры Java-элементов) снимите отметку с флажка Hide generated Spring Roo ITDs (Скрывать сгенерированные внутритиповые объявления Spring Roo), как показано на рис. 20.15.

После этого в проводнике пакетов можно будет видеть все классы (рис. 20.16).

Классы AspectJ (файлы `*.aj`), сгенерированные Roo, являются внутритиповыми объявлениями AspectJ (intertype declaration — ITD). Такие объявления ITD представляют собой внутреннюю “магию” Spring Roo и хранят всю логику, которая генерируется и поддерживается Roo. Например, объявления ITD будут хранить методы получения/установки, общие свойства для JPA (идентификатор, версию и т.д.), метод `toString()` и т.п.

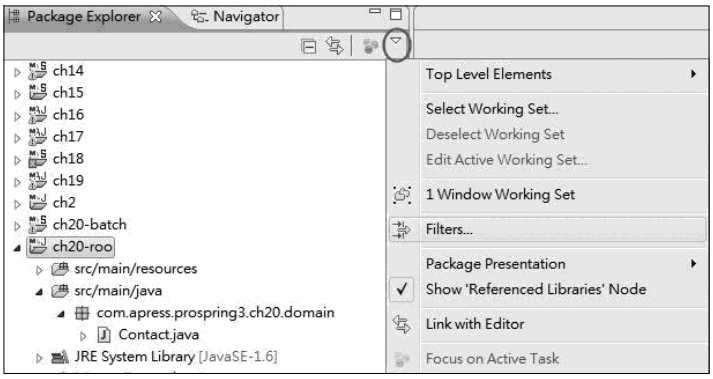


Рис. 20.14. Конфигурирование фильтров в проводнике пакетов

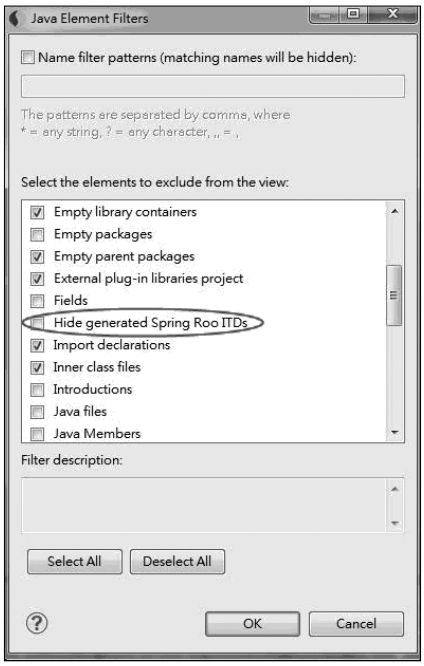


Рис. 20.15. Конфигурирование проводника пакетов для отображения классов, сгенерированных Spring Roo

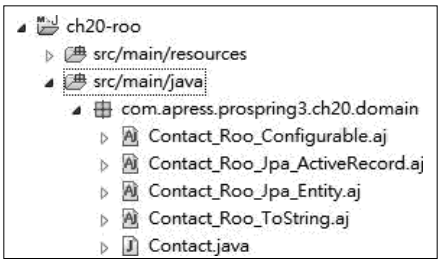


Рис. 20.16. Классы, сгенерированные Spring Roo

Если теперь заглянуть на вкладку Roo Shell, можно заметить, что оболочка “сфокусирована” на классе `Contact` (рис. 20.17).

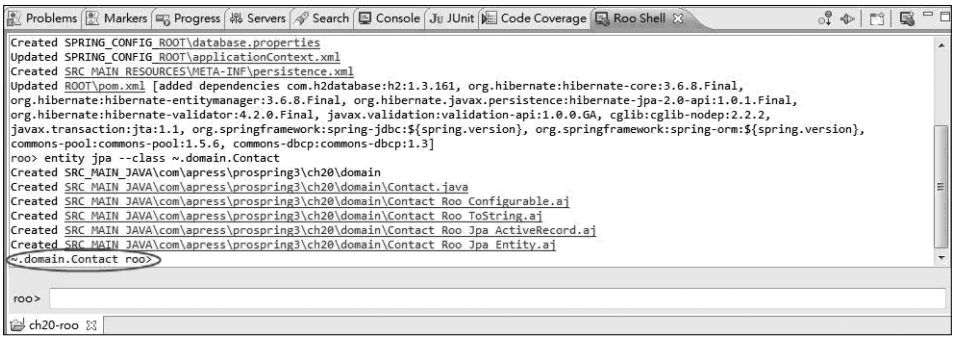


Рис. 20.17. Фокусировка Roo Shell

Фокусировка в Roo означает, что все последующие команды уровня классов будут применены к классу, который в текущий момент находится в фокусе. Следующий шаг заключается в определении свойств для класса `Contact`. Нам потребуется создать три свойства — `firstName`, `lastName` и `birthDate`. Чтобы сделать это, введите по очереди следующие команды:

```
field string --fieldName firstName --notNull --sizeMin 3 --sizeMax 60
field string --fieldName lastName --notNull --sizeMin 1 --sizeMax 40
field date --fieldName birthDate --type java.util.Date
```

Команда `field` используется для определения полей для класса `Contact`. Поскольку фокусировка уже находится на классе `Contact`, указывать имя класса не нужно. Атрибут после `field` определяет тип, затем после `--fieldName` задается имя и указываются ограничения JSR-303 для полей. После генерации свойств полей понадобится сконфигурировать абстракцию репозитория Spring Data JPA для класса `Contact`. Введите следующую команду в оболочке Roo:

```
repository jpa --interface ~.repository.ContactRepository
  --entity ~.domain.Contact
```

Команда сообщает Roo о необходимости генерации интерфейса репозитория JPA, который использует Spring Data JPA. На этом построение уровня постоянства в основном завершено.

Настройка уровня обслуживания

Начиная с версии 1.2.0, в Roo появилась поддержка генерации классов для уровня обслуживания. Чтобы увидеть это в действии, давайте создадим интерфейс `ContactService`, как мы делали в предыдущей главе. Введите следующую команду в оболочке Roo:

```
service --interface ~.service.ContactService --entity ~.domain.Contact
```

Команда `service` заставляет Roo создать интерфейс `ContactService`, который будет применяться для манипулирования сущностным классом `Contact`. По завершении вы увидите в проводнике пакетов, что кроме интерфейсного класса Roo был также сгенерирован класс `ContactServiceImpl`. Вдобавок, если вы просмотрите файл ITD по имени `ContactServiceImpl_Roo_Service.aj`, сгенерированный Roo, то заметите, что интерфейс репозитория (`ContactRepository`) автосвязан с классом службы.

К этому моменту серверная часть для сущностного класса `Contact` с базовыми операциями CRUD построена.

Настройка уровня презентаций

Генерировать веб-приложение, использующее Spring MVC, в Spring Roo очень просто. Сначала необходимо проинструктировать Roo относительно настройки Spring MVC для приложения. Введите следующую команду в оболочке Roo:

```
web mvc setup
```

Заглянув в проводник пакетов, вы увидите, что Roo была сгенерирована структура папок веб-среды, дескриптор веб-развертывания (`web.xml`), файл конфигурации `WebApplicationContext` диспетчера сервлета (`webmvc-config.xml`) и т.д.

Далее понадобится сгенерировать класс контроллера и файлы представлений для сущностного класса `Contact`. Введите следующую команду в оболочке Roo:

```
web mvc all --package ~.web.controller
```

Эта команда указывает Roo на необходимость генерации шаблонных контроллеров Spring MVC для всех сущностей в проекте, не имеющих класса контроллера. Roo сгенерирует класс контроллера для сущности `Contact`, а также JSPX-файлы представлений для поддержки информации о контакте.

Итак, базовое приложение сгенерировано, поэтому выполните его сборку и развертывание на сервере tc Server в STS, запустите сервер и введите в браузере URL вида:

```
http://localhost:8080/ch20-roo
```

Приложение отобразится, как показано на рис. 20.18.

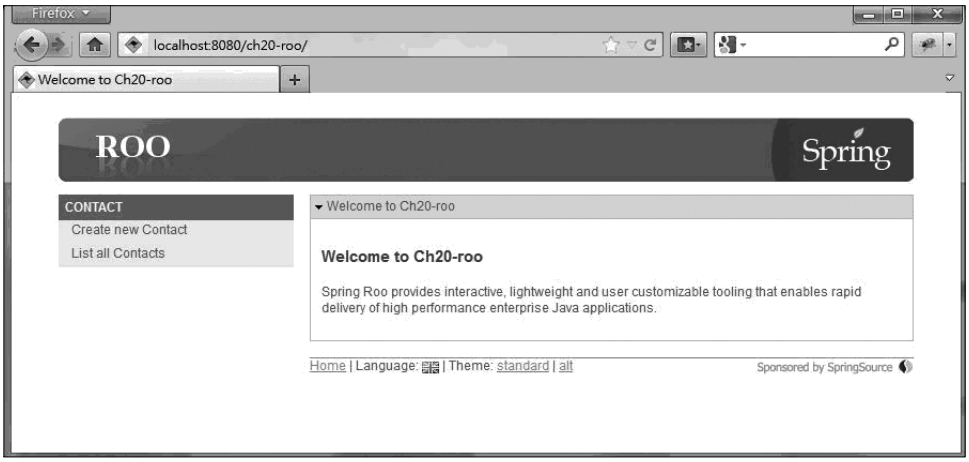


Рис. 20.18. Сгенерированное приложение Spring Roo

Щелкните на ссылке Create new Contact (Создать новый контакт), чтобы создать новый контакт, и вы увидите форму, представленную на рис. 20.19.

Обратите внимание, что поле даты уже имеет включенное окно выбора даты. Если вы просмотрите исходный код страницы, то увидите, что в сгенерированное представление включена JavaScript-библиотека Dojo Toolkit. Представление JSPX с Dojo Toolkit — это стандартный комплект, который Roo будет генерировать для интерфейсной части веб-приложения. После сохранения нового контакта щелкните на ссылке List all Contacts (Вывести список всех контактов) и вы увидите добавленный контакт (рис. 20.20).

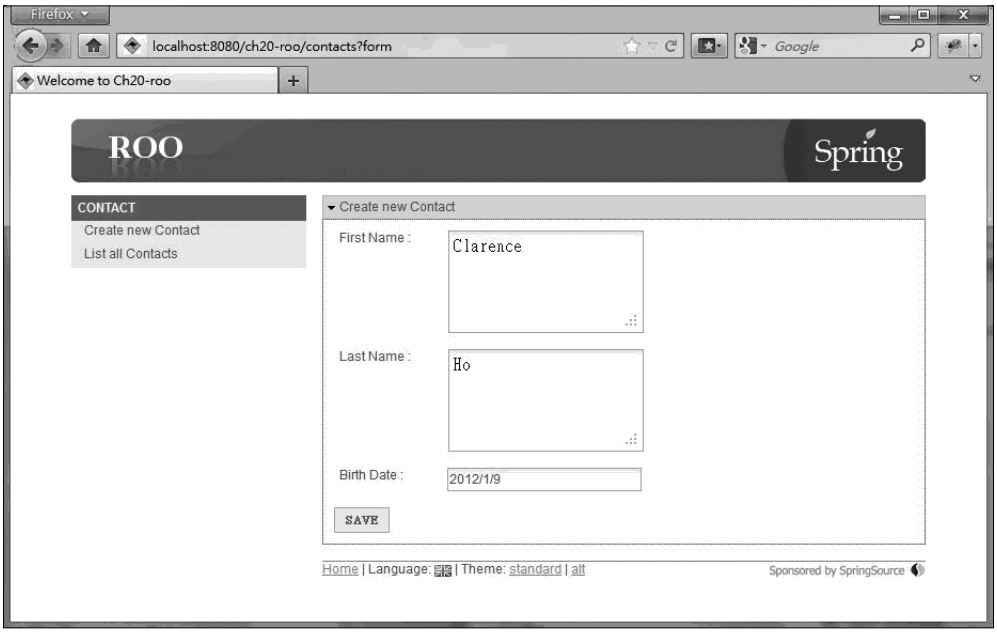


Рис. 20.19. Создание нового контакта

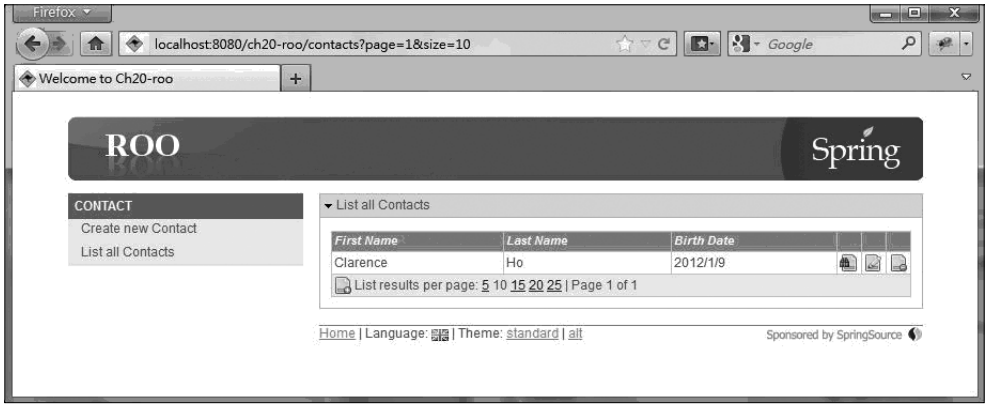


Рис. 20.20. Вывод списка всех контактов

На рис. 20.20 видно, что в таблице данных уже предусмотрены кнопки для просмотра, редактирования и удаления контакта (три кнопки справа от контакта). Кроме того, включена поддержка разбиения на страницы. В нижнем колонтитуле также видно, что присутствует поддержка оформления темами и интернационализации.

Roo позволяет управлять логикой в каждом классе, в конфигурации контекста, в файле представления и вообще где угодно. Например, предположим, что теперь необходимо поместить логику класса контроллера Spring MVC обратно в файл Java-класса, не оставляя ее под управлением Roo. Чтобы сделать это, щелкните правой кнопкой мыши на классе ITD для класса контроллера (`ContactController_Roo_Controller.aj`) и выберите в контекстном меню пункт Refactor⇨Push In (Рефакторинг⇨Поместить), как показано на рис. 20.21.

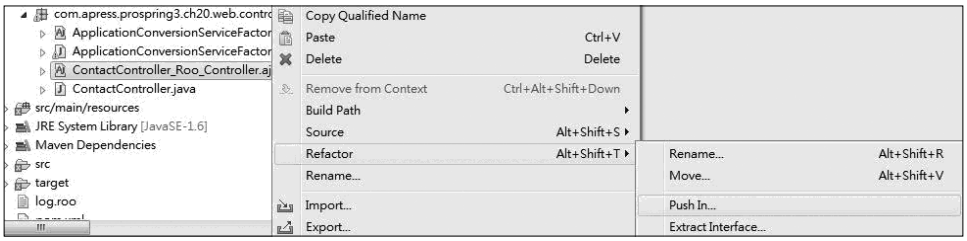


Рис. 20.21. Изъятие логики из Spring Roo

Откроется диалоговое окно Push In Intertype Declaration (Помещение внутритипового объявления) со списком всех методов, сгенерированных Roo. Здесь можно поместить в целевые типы только некоторые методы (развернув узел класса и выбрав нужные методы) или же все методы за раз (рис. 20.22).

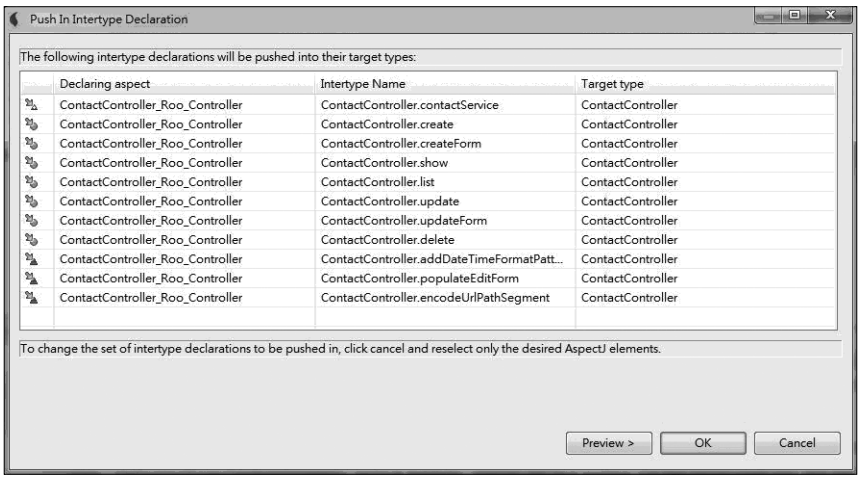


Рис. 20.22. Диалоговое окно Push In Intertype Declaration

После помещения логика будет извлечена из Roo в класс `ContactController`. Затем можно просмотреть класс `ContactController` на предмет помещенных методов.

Все введенные команды Roo будут сохраняться в файле по имени `log.roo` (расположенном в корневой папке проекта), поэтому данный файл можно использовать в качестве сценария при создании похожих проектов. Это очень удобная возможность; на рис. 20.23 показано содержимое файла `log.roo` для сгенерированного приложения контактов.

```
log.roo
1 // Spring Roo 1.2.0.RELEASE [rev 39eb957] log opened at 2012-01-12 12:47:45
2 project --topLevelPackage com.apress.prospring3.ch20 --projectName ch20-roo --java 6
3 // Spring Roo 1.2.0.RELEASE [rev 39eb957] log closed at 2012-01-12 12:47:45
4 // Spring Roo 1.2.0.RELEASE [rev 39eb957] log opened at 2012-01-12 12:47:47
5 persistence setup --database H2_IN_MEMORY --provider HIBERNATE
6 entity jpa --class ~.domain.Contact
7 field string --fieldName firstName --notNull --sizeMin 3 --sizeMax 60
8 field string --fieldName lastName --notNull --sizeMin 1 --sizeMax 40
9 field date --fieldName birthDate --type java.util.Date
10 repository jpa --interface ~.repository.ContactRepository --entity ~.domain.Contact
11 service --interface ~.service.ContactService --entity ~.domain.Contact
12 web mvc setup
13 web mvc all --package ~.web.controller
14
```

Рис. 20.23. Журнальный файл Roo

Дополнение Spring Roo

Еще одной мощной возможностью Spring Roo является архитектура дополнений, которая позволяет создавать специальные дополнения к Roo для удовлетворения специфических нужд. В настоящее время доступно очень много дополнений; некоторые из них разработаны командой Roo, другие — сообществами открытого кода, а третьи — разработчиками специфических инструментов или библиотек, которые желали сделать их доступными приложениям Roo.

В оболочке Roo Shell можно выводит список дополнений, искать их и устанавливать новые дополнения. Введите в оболочке Roo команду `addon` и нажмите клавиатурную комбинацию <Ctrl+пробел>; вы увидите команды, относящиеся к дополнениям, как показано на рис. 20.24.

Рис. 20.24. Команды, относящиеся к дополнениям в Roo

Например, команда `addon list` вводит список дополнений, доступных в настоящее время для установки и применения в приложении Roo. Вывод команды `addon list` показан на рис. 20.25.

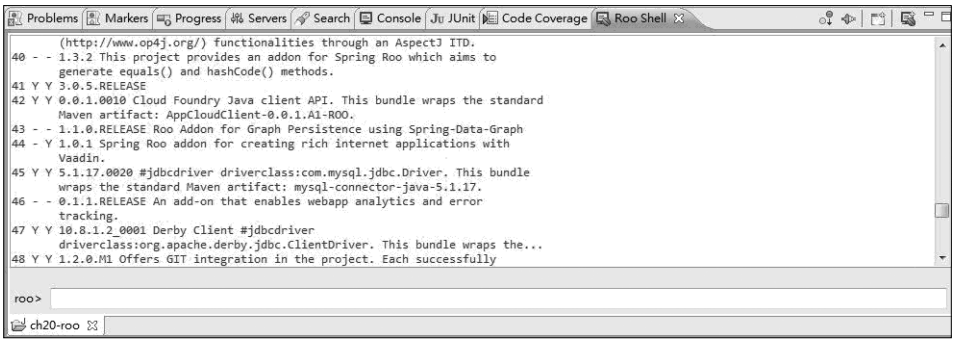


Рис. 20.25. Вывод команды `addon list`

На рис. 20.25 видны доступные дополнения вместе с их описаниями. Скажем, дополнение с идентификатором `44` позволяет разрабатывать RIA-приложения в Spring Roo с помощью Vaadin — популярной библиотеки для интерфейсной части, построенной на основе GWT.

Соображения по поводу Spring Roo

Благодаря предыдущему примеру, вы получили небольшое представление о том, как Roo помогает сформировать среду для быстрой разработки приложений. Проект Roo по-прежнему развивается быстрыми темпами, и множество дополнений находится на стадии разработки.

Проект Roo обладает большим потенциалом, поэтому стоит следить за его развитием. Даже несмотря на то что он может оказаться недостаточно мощным для генерации производственной версии JEE-приложения, он очень удобен для генерации прототипов приложения на начальном этапе проектирования. Кроме того, структура папок и классы, которые генерирует Roo, основаны на передовом опыте отрасли, при этом стиль кода, конфигурация контекста и т.п. соответствуют рекомендациям команды разработчиков SpringSource. Также полезно просматривать приложение, сгенерированное Roo, и использовать его для изучения передового опыта и стиля кодирования Spring-приложений.

Если вас интересует дальнейшее изучение Spring Roo, обратитесь к справочной документации (`http://static.springsource.org/spring-roo/reference/html/index.html`) и к книге *Spring Roo in Action* (Manning, 2012 г.).

Spring Batch и Spring Integration в примере приложения

В приложении SpringBlog мы будем использовать Spring Batch и Spring Integration для реализации задания импорта записей блога из XML-файла.

Механизм, который обсуждался в этой главе, будет применяться при разработке пакетного задания. Во-первых, будет определен формат файла XML для записей блога вместе с соответствующими определениями отображений Castor.

Во-вторых, для сканирования поступления файла, содержащего новые записи блога, будет использоваться поддержка опроса файлов Spring Integration. При появлении нового файла адаптер входного канала получит файл и в канал файла будет отправлено файловое сообщение. Тот же преобразователь, который разработан в этой главе, будет применяться для трансформации файлового сообщения в сообщение запроса на запуск задания и вывода его в канал запросов.

Для импорта записей блога будет реализовано пакетное задание, использующее тот же самый уровень обслуживания для обработки записи. При обработке элементов будет применяться проверка достоверности JSR-303. Для запуска задания будет использоваться активатор службы. За дополнительными деталями обращайтесь в главу 21.

Резюме

В этой главе мы обсудили несколько интересных и полезных проектов Spring, которые позволяют удовлетворить общие требования к приложениям.

Сначала мы рассмотрели проект Spring Batch и его архитектуру обработки на основе порций, а также увидели его в действии на примере реализации задания для импорта контактов. Затем мы продемонстрировали взаимодействие, основанное на сообщениях, с помощью проекта Spring Integration, его поддержку файловых операций и работу совместно с проектом Spring Batch для запуска заданий.

Наконец, мы описали интересный проект Spring Roo, предназначенный для быстрой разработки приложений и построения прототипов. Мы обсудили его новую поддержку командной строки для генерации кода различных уровней приложения, гибкость в возврате управления из сгенерированного кода и архитектуру дополнений, которая позволяет добавлять выбранные инструменты и библиотеки.

ГЛАВА 21

Дополнительные сведения о примере приложения

В главе 3 мы представили обзор архитектуры примера приложения, разрабатываемого в этой книге, включая многоуровневую структуру приложения, основные функциональные возможности и экранные снимки завершенного приложения. Затем в главе 12 мы обсудили несколько важных тем относительно проектирования и реализации Spring-приложений со ссылками на пример приложения. В конце каждой главы мы раскрывали основные отношения между рассмотренными в главе средствами и их применением в примере приложения.

В этой главе мы детально рассмотрим пример приложения под названием SpringBlog, которое было разработано для настоящей книги. Наша цель состоит в том, чтобы тщательно разобраться в деталях проекта приложения и объяснить реализацию основных функциональных средств, которые обсуждались в предшествующих главах, а также их совместную работу в завершенном JEE-приложении. В частности, в главе раскрываются следующие темы.

- *Настройка проекта.* Мы объясним, как настроить проект и среду STS, чтобы можно было выполнять сборку и запускать пример приложения на своей машине. Будут предоставлены инструкции по использованию средства профилей определения бинов Spring Framework для переключения СУРБД серверной части (между MySQL и H2) и реализаций уровня обслуживания (между Hibernate JPA 2 и MyBatis).
- *Проект приложения.* Мы обсудим основные проектные элементы приложения. Будут рассмотрены модель данных, объектная модель предметной области и UML-модель главных сценариев использования.
- *Детали реализации.* Мы покажем, как реализованы основные функциональные средства в примере приложения. Будут рассматриваться такие вопросы, как использование АОП для фильтрации нежелательного контента, создание уровня обслуживания с помощью Hibernate и JPA 2, реализация проверки достоверности с преобразованием и форматированием, построение интерфейсной части, применение веб-службы REST, создание пакетных заданий, планирование задания для очистки данных аудита и т.д.

Несмотря на то что все функциональные средства SpringBlog, упоминаемые в книге, включены в реализованный исходный код, некоторые из них продолжают развиваться и после выхода книги в свет (например, использование реализации MyBatis для обработки комментариев к записям блога, загрузка файлов и т.п.). Мы периодически обновляем исходный код примера приложения, который хранится в репозитории GitHub. Кроме того, по мере поступления комментариев от читателей, существующие средства могут быть расширены, а новые средства — добавлены. Таким образом, регулярно проверяйте репозиторий исходного кода на предмет внесенных изменений.

Ниже перечислены функциональные средства примера приложения, которые во время написания книги находились на стадии разработки.

- Реализации MyBatis, предназначенные для обновления записей блога, отправки комментариев к записям, загрузки файлов и поддержки версий сущностей.
- Запланированное задание для очистки старых записей хронологии, связанной с записями и комментариями в блоге.
- Управление пользователями и назначение ролей.
- Управление категориями и подкатегориями для записей блога.
- Модульное тестирование.

Настройка примера приложения

Мы начнем с получения исходного кода и настройки проекта в STS. Затем мы настроим сервер tc Server и развернем приложение SpringBlog, чтобы увидеть его в действии. Понять исходный код будет намного проще, если параллельно с чтением наблюдать за работой приложения.

В последующих разделах мы покажем, как настроить проект в STS и запустить приложение SpringBlog. Будут рассмотрены такие темы, как получение исходного кода, импорт проекта, настройка сервера, а также переключение между различными СУРБД и реализациями уровня обслуживания.

Для настройки проекта и запуска примера приложения понадобится JDK 6 или новее, SpringSource Tool Suite (версии 2.8.1 или новее) и доступное подключение к Интернету на машине разработки. Чтобы протестировать серверную часть MySQL, нужен локальный экземпляр базы данных MySQL 5.1 или более новой версии, а также элементарные знания по управлению базой данных MySQL (создание пользователя, запуск сценария и т.п.).

Настройка проекта

Для настройки проекта необходимо получить исходный код и импортировать его в STS. Доступны два способа получения исходного кода. Первый способ — извлечь файл `9781430241071_ch21.zip` из исходного кода примеров для этой книги в рабочее пространство STS. Второй способ предполагает получение последней версии исходного кода из репозитория SpringBlog в GitHub. Репозиторий для приложения SpringBlog доступен по адресу `https://github.com/prospring3/springblog`.

После того, как исходный код станет доступным на машине разработки, импортируйте проект в STS. Для этого выберите пункт меню File⇨Import (Файл⇨Импорт) и на экране Select (Выбор) укажите вариант Existing Projects into Workspace (Существующие проекты в рабочем пространстве) в категории General (Общие), как показано на рис. 21.1.

После этого на экране Import Projects (Импорт проектов) выберите проект, извлеченный из кода примеров, и щелкните на кнопке Finish (Готово), как показано на рис. 21.2.

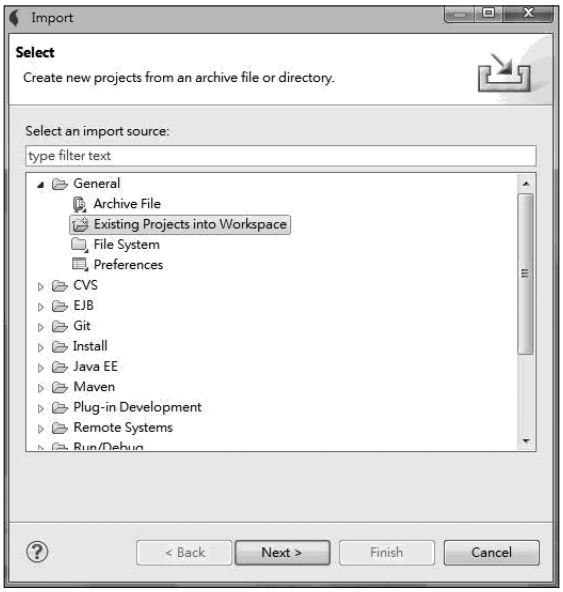


Рис. 21.1. Импорт существующего проекта в STS

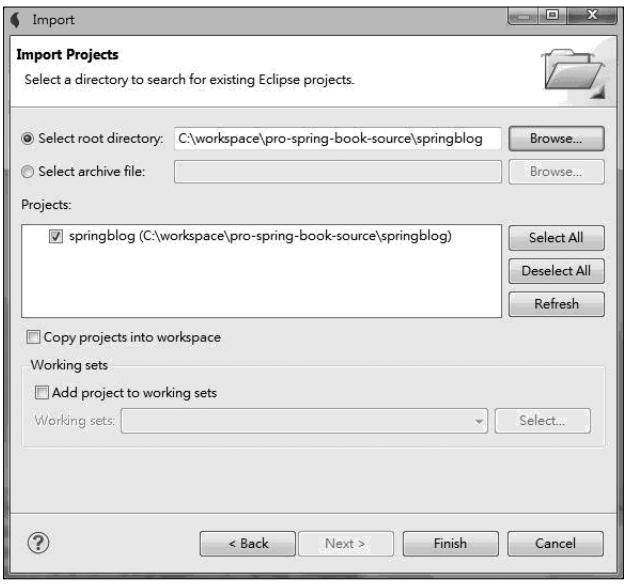


Рис. 21.2. Импорт проекта SpringBlog

Затем, возможно, придется подождать, пока подключаемый модуль Maven загрузит обязательные зависимости и выполнит сборку проекта.

По завершении импорта проекта необходимо создать экземпляр сервера tc Server для развертывания приложения SpringBlog. Развертывание можно также провести на любом существующем экземпляре tc Server в STS. Подробные шаги по созданию экземпляра tc Server приведены в приложении А. Кроме того, удостоверьтесь, что проект добавлен к

целевому экземпляру tc Server, а классы и библиотеки проекта опубликованы на сервере (другими словами, проект собран и развернут на целевом экземпляре tc Server).

На рис. 21.3 показана вкладка Servers (Серверы) в STS. В этом случае для примера приложения вполне достаточно “базового” шаблона экземпляра tc Server.

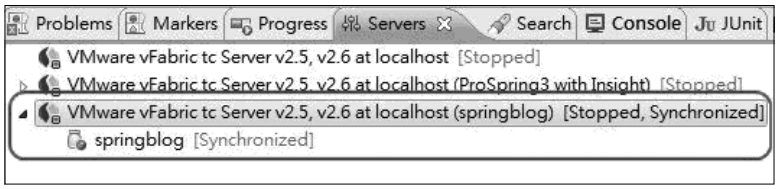


Рис. 21.3. Добавление проекта SpringBlog к экземпляру tc Server в STS

Запустите tc Server. Затем откройте браузер и введите URL `http://localhost:8080/springblog/blogs`. Вы должны увидеть начальную страницу приложения SpringBlog, подобную показанной на рис. 21.4.

Проект успешно настроен. Обратите внимание, что по умолчанию пример приложения не содержит никаких записей в блоге. Войдите в приложение (щелкните на кнопке Hint (Подсказка) под кнопкой Login (Вход), чтобы воспользоваться учетной записью по умолчанию) и отправьте несколько записей и комментариев в блог для получения полезного опыта работы с приложением.

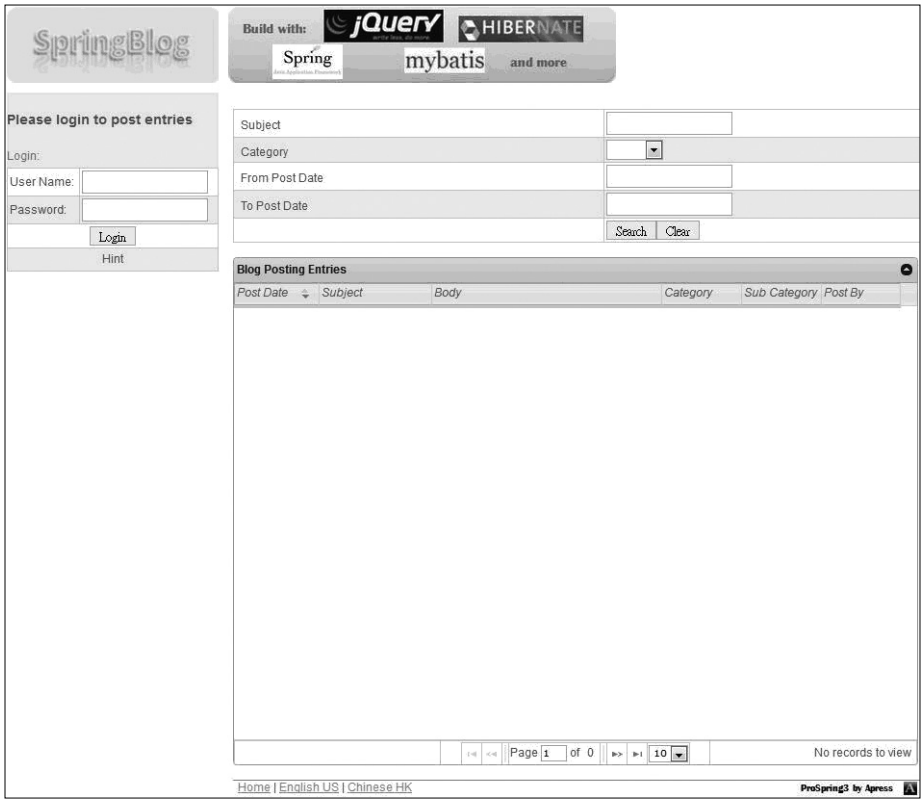


Рис. 21.4. Начальная страница приложения SpringBlog

Переключение между базами данных MySQL и H2

По умолчанию приложение SpringBlog будет инициализировать встроенную базу данных H2 и запускать соответствующие сценарии, поэтому дополнительной настройки не понадобится. Однако сценарии базы данных и приложение также могут успешно работать с MySQL. В этом разделе описана настройка приложения для использования базы данных MySQL, так что вы сможете отслеживать записи в базе данных, работая в приложении.

В листинге 21.1 приведено содержимое конфигурационного файла Spring (`/src/main/webapp/WEB-INF/spring/datasource.xml`), который хранит конфигурацию источника данных для приложения.

Листинг 21.1. Файл конфигурации источника данных

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns:jdbc="http://www.springframework.org/schema/jdbc"
     xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
        http://www.springframework.org/schema/jdbc
        http://www.springframework.org/schema/jdbc/spring-jdbc-3.1.xsd">

   <!-- Для встроенной базы данных H2 -->
   <beans profile="h2">
      <jdbc:embedded-database id="dataSource" type="H2">
         <jdbc:script location="classpath:sql/schema.sql"/>
         <jdbc:script
        location="classpath:/org/springframework/batch/core/schema-h2.sql"/>
         <jdbc:script location="classpath:sql/initial-data.sql"/>
      </jdbc:embedded-database>
   </beans>

   <!-- Для базы данных MySQL -->
   <beans profile="mysql">
    <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource"
        destroy-method="close">
        <property name="driverClass" value="com.mysql.jdbc.Driver" />
        <property name="jdbcUrl" value="jdbc:mysql://localhost:3306/springblog" />
        <property name="user" value="springblog" />
        <property name="password" value="springblog" />
     </bean>
   </beans>

</beans>
```

В листинге 21.1 объявлены два разных бина базы данных. Оба бина помещены в дескриптор `<beans>` с установленным атрибутом `profile`. Это новое средство профилей, появившееся в версии Spring Framework 3.1. В зависимости от активного профиля Spring будет инициировать нужный бин.

Когда в качестве профиля указан `h2`, дескриптор `<jdbc:embedded-database>` инициализирует встроенную базу данных H2 и выполнит сценарии, предоставленные в определении бина.

Для приложения SpringBlog должны быть выполнены три сценария (порядок их выполнения важен).

1. `schema.sql`. Модель данных для приложения SpringBlog.
2. `schema-h2.sql`. Схема для таблиц, требуемых Spring Batch.
3. `initial-data.sql`. Начальные данные (пользователи, роли, категории и т.д.) для приложения SpringBlog.

Чтобы настроить MySQL, как показано в определении бина `dataSource` для профиля `mysql`, создайте базу данных по имени `springblog`, а также пользователя с именем и паролем, установленными в `springblog` (параметры источника данных можно изменить по собственному усмотрению). Затем понадобится запустить три сценария (`schema.sql`, `schema-mysql.sql`, `initial-data.sql`) из папки `/src/main/resources/sql`. Обратите внимание, что файл `schema-mysql.sql` предназначен для создания таблиц Spring Batch.

Далее необходимо изменить активный профиль в `WebApplicationContext`, чтобы применялся бин `dataSource` для MySQL. Чтобы изменить активный профиль, нужно модифицировать соответствующий параметр в дескрипторе веб-развертывания (файл `/WEB-INF/web.xml`), как показано в листинге 21.2.

Листинг 21.2. Изменение активного профиля в файле `web.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
   <!-- Определение корневого контейнера Spring, совместно
       используемого сервлетами и фильтрами -->
   <!-- Разрешить защиту содержимого отправляемой формы -->
   <context-param>
      <param-name>defaultHtmlEscape</param-name>
      <param-value>true</param-value>
   </context-param>
   <context-param>
      <param-name>spring.profiles.active</param-name>
      <!-- Параметры профилей Spring
          Первый профиль: jpa - реализация JPA
                          mybatis - реализация MyBatis
          Второй профиль: mysql - база данных MySQL
                          h2 - база данных H2
      -->
      <param-value>jpa,h2</param-value>
   </context-param>
      <context-param>
         <param-name>contextConfigLocation</param-name>
         <param-value>
            /WEB-INF/spring/root-context.xml
            /WEB-INF/spring/datasource.xml
            /WEB-INF/spring/batch-context.xml
            /WEB-INF/spring/*-tx-config.xml
            /WEB-INF/spring/*-service-context.xml
         </param-value>
   </context-param>
   <!-- Остальной код не показан -->
</web-app>
```

В листинге 21.2 для приложения SpringBlog определены два значения профилей. Первое из них указывает используемую реализацию уровня обслуживания (`jpa` или `mybatis`), а второе (выделено полужирным) — инициализируемый бин источника данных. Для использования MySQL измените второе значение профиля с `h2` на `mysql`. После выполнения сборки и развертывания проекта на сервере tc Server приложение снова будет работать с базой данных MySQL.

Переключение между реализациями JPA и MyBatis

Как было показано в предыдущем разделе, изменение реализации для уровня обслуживания с JPA 2 на MyBatis также осуществляется просто. Нужно только изменить первое значение профиля с `jpa` на `mybatis`. Например, для параметра с именем `spring.profiles.active` значение `mybatis,mysql` указывает Spring на необходимость загрузки бинов из соответствующих профилей, чтобы использовать реализацию MyBatis уровня обслуживания и бин `dataSource` для MySQL.

В главе 5 упоминалось, что еще один метод для установки активного профиля предусматривает применение конфигурации запуска tc Server. Это устраняет необходимость в модификации файла `web.xml` и в повторной упаковке архивного файла веб-приложения.

Чтобы изменить конфигурацию запуска, на вкладке Servers (Серверы) в STS дважды щелкните на экземпляре tc Server, в котором выполняется приложение SpringBlog, и щелкните на ссылке Open launch configuration (Открыть конфигурацию запуска), как показано на рис. 21.5.

В открывшемся диалоговом окне Edit Configuration (Редактирование конфигурации) перейдите на вкладку Arguments (Аргументы) и добавьте параметр `-Dspring.profiles.active="mybatis,mysql"` в поле VM arguments (Аргументы виртуальной машины), как показано на рис. 21.6.

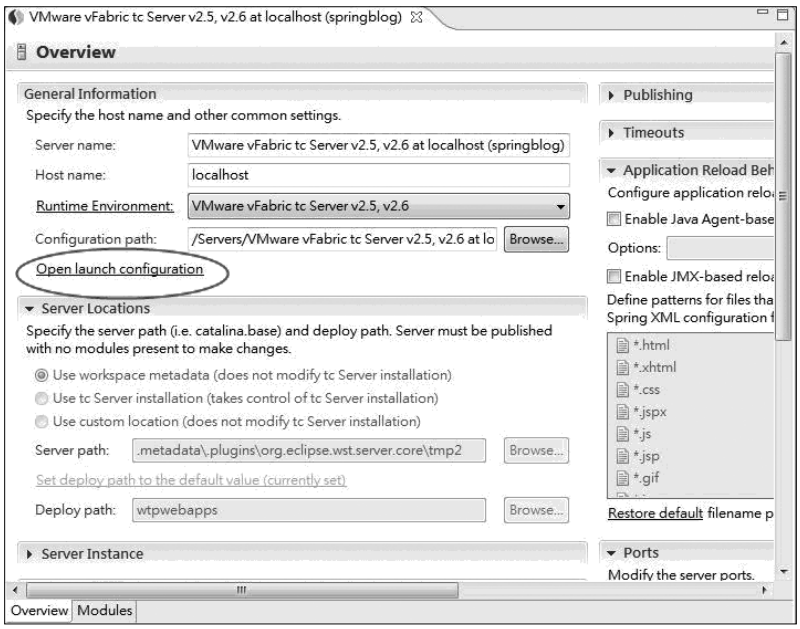


Рис. 21.5. Доступ к конфигурации запуска tc Server

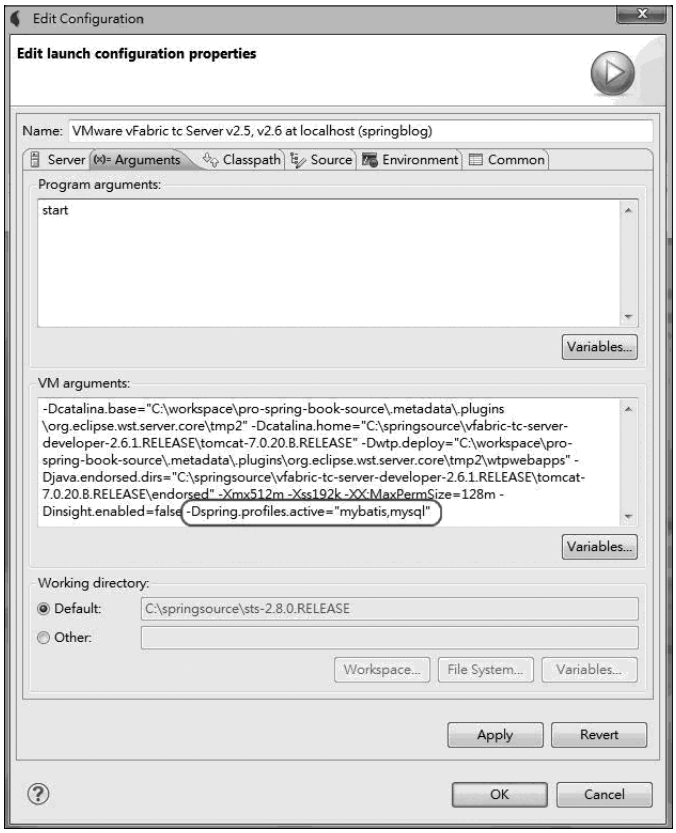


Рис. 21.6. Настройка активного профиля Spring для tc Server

После изменения аргумента виртуальной машины сервера tc Server можно удалить соответствующий параметр из файла `web.xml`. Чтобы испытать различные комбинации уровня обслуживания и СУРБД серверной части, просто изменяйте аргумент виртуальной машины в конфигурации запуска сервера tc Server.

Проект приложения

В этом разделе представлен обзор проекта приложения. В последующих разделах мы рассмотрим разнообразные аспекты проекта, в том числе модель данных, объектную модель предметной области, UML-модель для основных сценариев использования и т.п.

Модель данных

Начнем с обсуждения модели данных, которая показана в виде диаграммы “сущность-отношение” на рис. 21.7.

Обратите внимание, что схема для Spring Batch на рис. 21.7 не показана. Кроме того, таблицы, предназначенные для аудита (таблицы `ENTRY_H`, `COMMENT_H` и `REVINFO`, которые наполняются модулем Hibernate Envers), для большей ясности тоже не рассматриваются. В табл. 21.1 приведено описание таблиц, представленных на рис. 21.7.

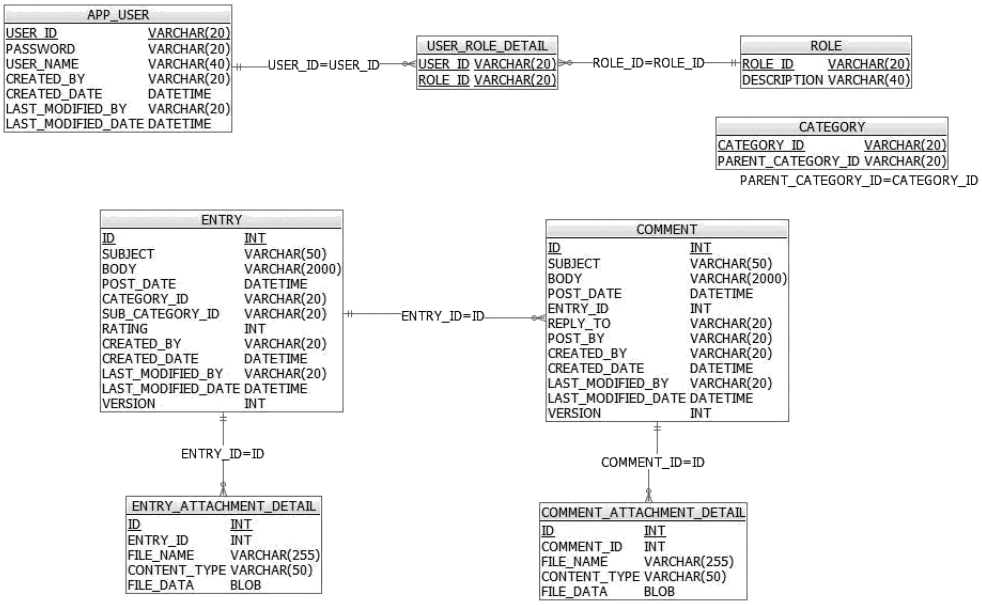


Рис. 21.7. Модель данных для приложения SpringBlog

Таблица 21.1. Таблицы для приложения SpringBlog

Имя таблицы	Назначение
APP_USER	В этой таблице хранится информация о пользователях для управления безопасностью с помощью Spring Security. Аутентификация пользователей будет производиться с применением данных из этой таблицы
ROLE	В этой таблице хранятся роли (т.е. обычный пользователь, администратор и удаленный клиент), доступные в приложении
USER_ROLE_DETAIL	Это таблица соединения, в которой хранятся роли, выданные каждому пользователю
CATEGORY	В этой таблице хранится категория для каждого сообщения блога (например, Java, JPA, Spring и т.д.). Категория может выступать подкатегорией, и в таком случае в столбце PARENT_CATEGORY_ID указывается родительская категория
ENTRY	В этой таблице хранятся сообщения блога
ENTRY_ATTACHMENT_DETAIL	В этой таблице хранятся вложения для сообщений блога
COMMENT	В этой таблице хранятся комментарии к сообщениям блога. Она связана отношением “многие к одному” с таблицей ENTRY
COMMENT_ATTACHMENT_DETAIL	В этой таблице хранятся вложения для комментариев к сообщениям блога

Объектная модель предметной области

Давайте перейдем к построению объектной модели предметной области. Как было показано в главе 12, модель DOM — это объектное представление предметной области приложения. В приложении SpringBlog предметной областью является информация,

относящаяся к сообщениям блога. Модель DOM тесно связана с моделью данных, и в JEE-приложениях, независимо от того, применяется объектно-реляционное отображение или отображение данных, будет происходить процесс “отображения” при трансформировании модели данных в DOM-модель Java. На рис. 21.8 показана DOM-модель для приложения SpringBlog.

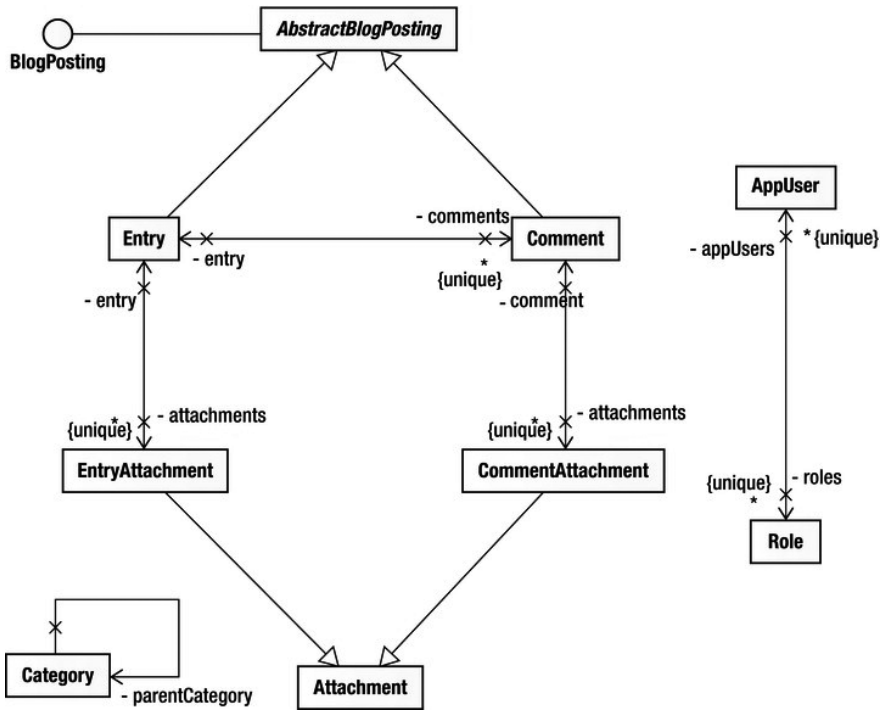


Рис. 21.8. DOM-модель для приложения SpringBlog

На рис. 21.8 можно заметить некоторые отличия между DOM-моделью и моделью данных, которая была представлена на рис. 21.7.

- Оба класса, `Entry` и `Comment`, расширяют класс `AbstractBlogPosting`, содержащий общие свойства и поведение записи, которая может быть как сообщением блога, так и комментарием. Однако вы не увидите какой-либо таблицы для этого абстрактного класса. Аналогичная ситуация и с классом `Attachment`. Кроме того, класс `AbstractBlogPosting` реализует интерфейс `BlogPosting`. Фильтр нежелательного контента, созданный с помощью АОП, будет основывать необходимость проверки и затенения нежелательного контента на том, реализует ли целевой объект упомянутый интерфейс.
- Между классами `AppUser` и `Role` установлено отношение “многие ко многим”, поскольку пользователю может быть выдано несколько ролей, а роль может быть назначена множеству пользователей. В модели данных имеется таблица соединения (`USER_ROLE_DETAIL`), которая поддерживает это отношение, в то время как в DOM-модели никакого класса для этой таблицы соединения не предусмотрено.
- Класс `Category` имеет отношение с самим собой, отражающее иерархию родительских и дочерних категорий.

Модель UML

Модель UML содержит различные диаграммы, которые отражают поведение классов внутри приложения. Различают два основных типа UML-диаграмм — статические и динамические. Статическая диаграмма отражает классы и отношения между ними с точки зрения проектирования, не указывая взаимодействие между классами. Примерами статических диаграмм могут служить диаграммы сценариев использования и диаграммы классов. Динамические диаграммы указывают, как классы совместно работают и взаимодействуют друг с другом для формирования бизнес-потока. Примеры динамических диаграмм включают диаграммы последовательностей, диаграммы действий, диаграммы совместной работы, диаграммы состояний и т.д.

В этом разделе мы сосредоточим внимание на применении диаграмм последовательностей, предназначенных для отражения потока и взаимодействия между классами на различных уровнях приложения. В последующих разделах мы обсудим диаграммы последовательностей для трех основных сценариев использования в рамках приложения SpringBlog — создание записи сообщения блога, работа с веб-службой REST и создание пакетных заданий.

Создание записи сообщения блога

Рассмотрим диаграмму последовательности для создания записи сообщения блога, которая показана на рис. 21.9.

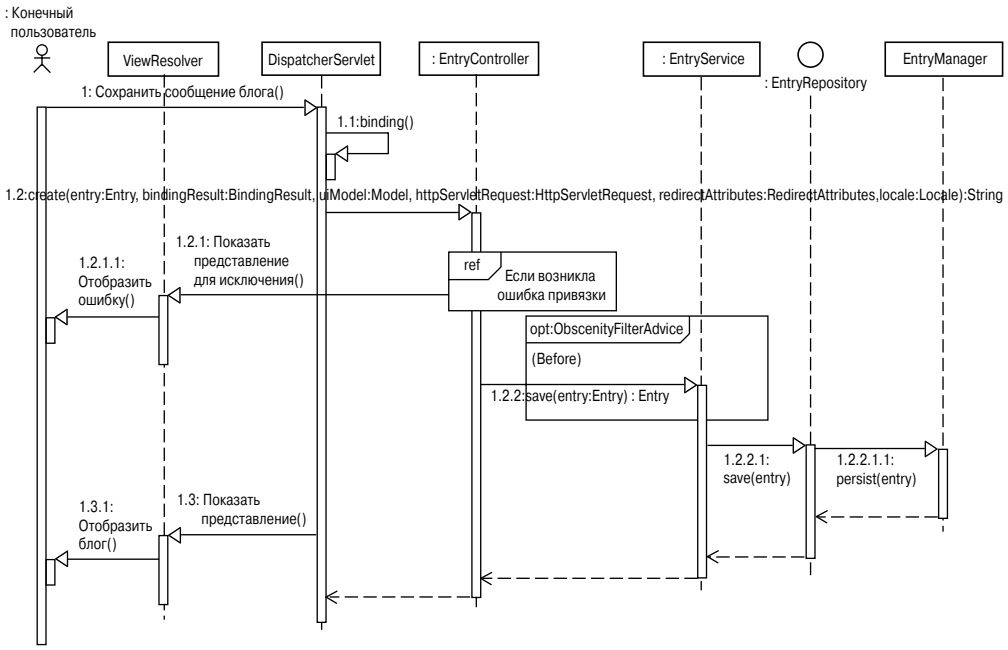


Рис. 21.9. Диаграмма последовательности SpringBlog: создание записи сообщения блога

Диаграмма последовательности на рис. 21.9 отражает поток процесса создания в блоге сообщения, а также взаимодействия между основными классами внутри каждого уровня, от уровня презентаций до уровня постоянства. В этом случае применяется реализация JPA 2, а поток выглядит следующим образом.

1. Пользователь входит в приложение SpringBlog, создает новую запись, вводит информацию и щелкает на кнопке Save (Сохранить). Браузер отправляет серверу запрос `POST`.
2. Сервлет диспетчера Spring MVC (`DispatcherServlet`) получает запрос, анализирует детали HTTP-запроса и идентифицирует метод контроллера, который должен быть вызван. В этом случае будет вызван метод `create()` класса `EntryController` (из пакета `com.apress.prospring3.springblog.web.blogapp.controller`). В то же самое время выполняется привязка данных запроса к соответствующему объекту предметной области. Полный процесс привязки включает проверку достоверности, преобразование и форматирование.
3. Внутри метода `EntryController.create()` будет проверяться аргумент `BindingResult`, чтобы выяснить, не возникли ли ошибки привязки. Если это так, в экземпляр `Model` сохраняются соответствующие сообщения, и страница редактирования отобразится повторно с сообщениями об ошибках.
4. Если процесс привязки прошел успешно, контроллер вызовет метод `save()` интерфейса `EntryService` (из пакета `com.apress.prospring3.springblog.service`), передавая ему объект `Entry` в качестве аргумента. Обратите внимание, что поскольку мы имеем дело с реализацией JPA 2, будет использоваться класс реализации `EntryServiceImpl` (из пакета `com.apress.prospring3.springblog.service.jpa`). Одновременно будет применен совет “перед” (об этом пойдет речь позже) для проверки на присутствие нежелательного контента в сообщении блога.
5. Класс `EntryServiceImpl` сохранит запись сообщения блога, вызывая поставщик постоянства. В реализации JPA 2 будет использоваться абстракция репозитория Spring Data JPA (интерфейс `EntryRepository` из пакета `com.apress.prospring3.springblog.repository`), которая, в свою очередь, вызывает диспетчер сущностей Hibernate для сохранения данных в СУРБД серверной стороны.
6. После завершения операции сохранения контроллер вернет логическое представление (в этом случае — страницу просмотра блога). Сервлет диспетчера (`DispatchServlet`) передает имя логического представления распознавателю представлений (`ViewResolver`). Распознаватель представлений распознает соответствующее представление (на основе конфигурации) для отображения интерфейсной части пользователю.

Не забывайте, что для простоты взаимодействия с некоторыми компонентами опущены. Например, `SecurityFilterChain` из Spring Security будет перехватывать каждый запрос в целях авторизации.

Веб-служба REST для RSS-канала сообщений блога

Второй сценарий использования, который мы рассмотрим — это RSS-канал, который выводит сообщения блога в формате XML или JSON для удаленных клиентов HTTP. Перед обсуждением диаграммы давайте посмотрим, каким образом работает канал. Чтобы извлечь RSS-канал в формате XML, отправьте команду `curl`, приведенную в листинге 21.3, функционирующему приложению SpringBlog.

Листинг 21.3. Команда для извлечения RSS-канала в формате XML

```
curl -v -u remote:remote -H "Accept: application/xml"
  http://localhost:8080/springblog/restful/blog/listdata
```

В листинге 21.3 указано имя пользователя для удаленного доступа к веб-службе REST приложения SpringBlog, а также URL для RSS-канала. Заголовок запроса установлен для приема XML в качестве поддерживаемого формата. На рис. 21.10 показан вывод в формате XML.

```
命令提示字元
* Connected to localhost (::1) port 8080 (#0)
* Server auth using Basic with user 'remote'
> GET /springblog/restful/blog/listdata HTTP/1.1
> Authorization: Basic cmVtb3RlOnJlbW90ZQ==
> User-Agent: curl/7.21.0 (amd64-pc-win32) libcurl/7.21.0 OpenSSL/0.9.8o zlib/1.2.3
> Host: localhost:8080
> Accept: application/xml
>
< HTTP/1.1 200 OK
< Server: Apache-Coyote/1.1
< Content-Type: application/xml;charset=UTF-8
< Transfer-Encoding: chunked
< Date: Sat, 28 Jan 2012 05:40:36 GMT
<
<?xml version="1.0" encoding="UTF-8"?>
<entries>
    <entry>
        <id>1</id>
        <subject>Welcome to SpringBlog</subject>
        <category>Spring</category>
        <subCategory></subCategory>
        <body>&lt;p&gt;&#xd;
        Welcome to SpringBlog application!&lt;/p&gt;&#xd;
</body>
        <postDate>2012-01-28</postDate>
        <postBy>clarence</postBy>
        <lastModifiedDate>2012-01-28</lastModifiedDate>
        <lastModifiedBy>clarence</lastModifiedBy>
        <version>0</version>
    </entry>
    <entry>
        <id>2</id>
        <subject>Spring Framework 3.1</subject>
        <category>Spring</category>
        <subCategory></subCategory>
        <body>&lt;p&gt;&#xd;
        Spring Framework 3.1 rocks!&lt;/p&gt;&#xd;
</body>
        <postDate>2012-01-28</postDate>
        <postBy>clarence</postBy>
        <lastModifiedDate>2012-01-28</lastModifiedDate>
        <lastModifiedBy>clarence</lastModifiedBy>
        <version>0</version>
    </entry>
</entries>
```

Рис. 21.10. RSS-канал в формате XML

В листинге 21.4 приведена команда для запроса данных в формате JSON.

Листинг 21.4. Команда для извлечения RSS-канала в формате JSON

```
curl -v -u remote:remote -H "Accept: application/json"
  http://localhost:8080/springblog/restful/blog/listdata
```

Единственное отличие между этими командами связано с заголовком запроса, в котором в данном случае указан формат JSON. На рис. 21.11 представлен ответ в формате JSON.

Теперь давайте посмотрим, как работает веб-служба REST для RSS-канала. Соответствующая диаграмма последовательности показана на рис. 21.12.

```
命令提示字元
* About to connect() to localhost port 8080 (#0)
* connected
* Connected to localhost () port 8080 (#0)
* Server auth using Basic with user 'remote'
> GET /springblog/restful/blog/listdata HTTP/1.1
> Authorization: Basic cmVtb3RlOnJlbW90ZQ==
> User-Agent: curl/7.21.0 (amd64-pc-win32) libcurl/7.21.0 OpenSSL/0.9.8o zlib/1.2.3
> Host: localhost:8080
> Accept: application/json
>
< HTTP/1.1 200 OK
< Server: Apache-Coyote/1.1
< Content-Type: application/json;charset=UTF-8
< Transfer-Encoding: chunked
< Date: Sat, 28 Jan 2012 05:41:22 GMT
<
{"entries":[{"id":1,"subject":"Welcome to SpringBlog","postDate":1327680000000,"createdBy":"clarence","createdDate":
728026664,"lastModifiedBy":"clarence","lastModifiedDate":1327728026664,"version":0,"categoryId":"Spring","subCategor
:"","shortBody":"<p>\r\n\tWelcome to SpringBlog application!</p>\r\n","new":false,"postDateString":"2012-01-28","las
ifiedDateString":"2012-01-28 13:20:26"},{"id":2,"subject":"Spring Framework 3.1","postDate":1327680000000,"createdBy
larence","createdDate":1327728053036,"lastModifiedBy":"clarence","lastModifiedDate":1327728053036,"version":0,"categ
d":"Spring","subCategoryId":"","shortBody":"<p>\r\n\tSpring Framework 3.1 rocks!</p>\r\n","new":false,"postDateStrin
2012-01-28","lastModifiedDateString":"2012-01-28 13:20:53"}]}* Connection #0 to host localhost left intact
* Closing connection #0
```

Рис. 21.11. RSS-канал в формате JSON

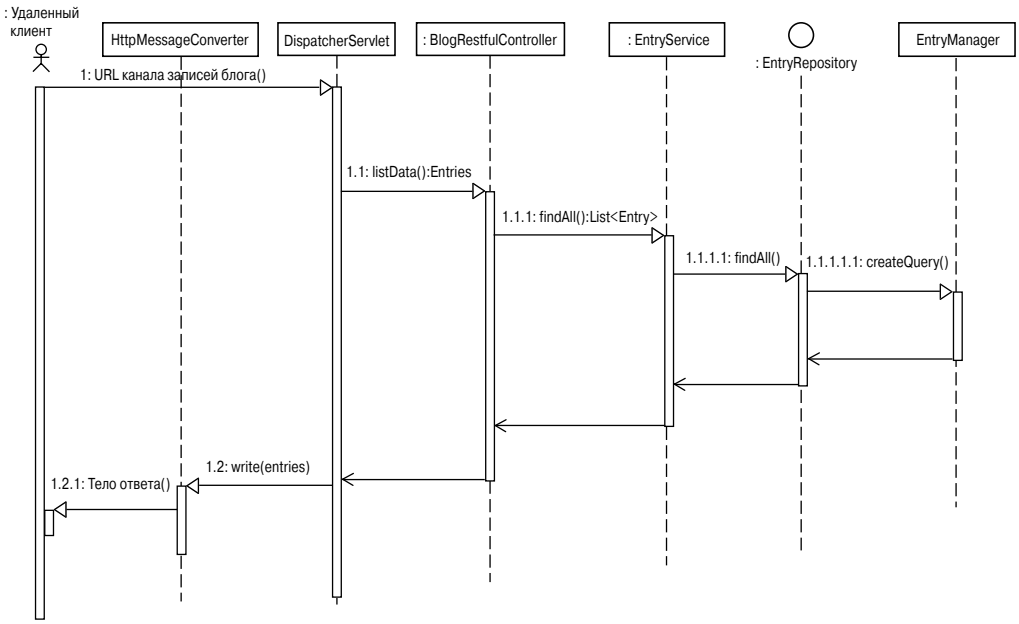


Рис. 21.12. Диаграмма последовательности для веб-службы REST, наполняющей RSS-канал

Диаграмма последовательности для взаимодействия со службой серверной части работает почти так же, как диаграмма сценария использования для создания сообщения блога, которая была показана на рис. 21.9. Основное отличие связано с уровнем презентаций. Сервлет диспетчера на основе URL запроса будет направлять запрос методу `listData()` класса `BlogRestfulController`. В свою очередь, контроллер обратится к уровню обслуживания для извлечения списка сообщений блога.

После этого контроллер возвращает результирующий класс `Entries` (который включает список объектов предметной области `Entry`).

Результат записывается напрямую в тело запроса. Как будет позже показано в конфигурации, в Spring MVC будет вызываться интерфейс `HttpMessageConverter<T>` для преобразования объекта в формат, требуемый клиентом. В зависимости от заголовка клиентского запроса будет вызван преобразователь JSON (класс `org.springframework.http.converter.json.MappingJacksonHttpMessageConverter`) или преобразователь XML (класс `org.springframework.http.converter.xml.MarshallingHttpMessageConverter`).

Пакетное задание для импорта сообщений блога из XML-файла

Еще один сценарий использования, который мы рассмотрим, касается пакетного импорта сообщений блога из XML-файлов. Давайте посмотрим на него в действии. Удостоверьтесь, что приложение SpringBlog функционирует, и затем поместите файл (по имени `entries.xml`; копия этого файла имеется в корневой папке проекта) в папку `C:\temp\entries` (как сконфигурировано в бине `<file:inbound-channel-adapter>` внутри файла `batch-context.xml`). Содержимое файла `entries.xml` (с единственным сообщением блога) приведено в листинге 21.5.

Листинг 21.5. XML-файл с сообщением блога

```
<?xml version="1.0" encoding="UTF-8"?>
<entries>
   <entry>
      <subject>SpringBlog by batch</subject>
      <category>Spring</category>
      <subCategory>Spring Batch</subCategory>
      <body>
         Testing SpringBlog posting entry by batch upload
      </body>
   </entry>
</entries>
```

По умолчанию Spring Integration опрашивает папку на предмет появления файла каждые 5 секунд (это также определено в файле `batch-context.xml`). Когда вы позже откроете пользовательский интерфейс, то увидите, что сообщение блога было импортировано в приложение SpringBlog (рис. 21.13).

Рис. 21.13. Импортированное сообщение в SpringBlog

В целом процесс импорта состоит из двух частей — запуск задания и выполнение задания. На рис. 21.14 показана диаграмма последовательности для процесса запуска задания.

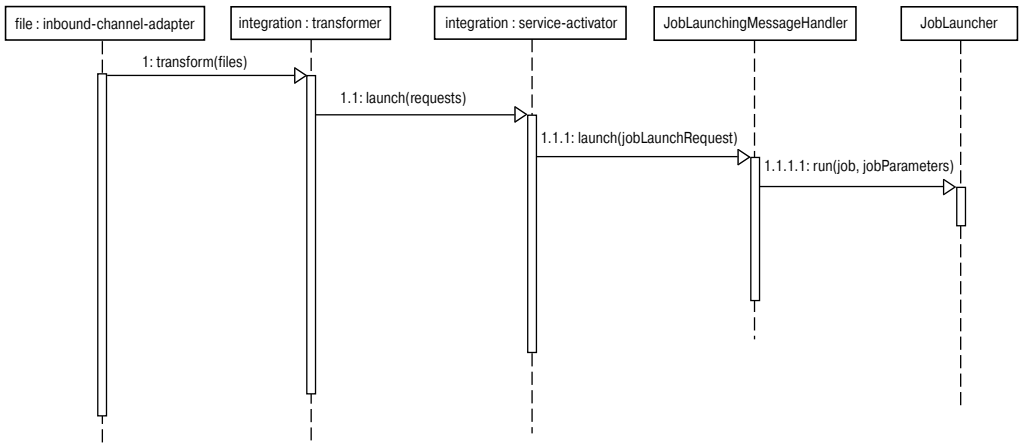


Рис. 21.14. Процесс запуска задания

Ниже описан основной поток процесса запуска задания.

1. Адаптер входного канала (`inbound-channel-adapter`) Spring Integration опрашивает папку на предмет поступления новых файлов.
2. После появления файла преобразователь (`transformer`) преобразует информацию в сообщение запроса на запуск задания.
3. Активатор службы Spring Integration (`service-activator`) получает сообщение запроса на запуск задания и передает его соответствующему обработчику сообщений. В данном случае обработчиком сообщений является класс `JobLaunchingMessageHandler` (из пакета `org.springframework.batch.integration.launch`), который принадлежит модулю Spring Batch Integration в рамках проекта Spring Batch Admin.
4. Если вы заглянете в конфигурацию бина `messageHandler`, определенного внутри бина `<integration:service-activator>` в файле `batch-context.xml`, то увидите, что аргументом конструктора обработчика сообщений является компонент запуска заданий (интерфейс `JobLauncher` из пакета `org.springframework.batch.core.launch`, который принадлежит Spring Batch). Обработчик сообщений запустит задание с преобразованными параметрами.

Следующая часть — это выполнение задания, диаграмма последовательности которого показана на рис. 21.15.

На рис. 21.15 видно, что Spring Batch будет обрабатывать каждый элемент порциями, что предполагает выполнение следующих шагов.

1. Чтение каждого элемента и демаршализация с помощью Castor XML-фрагмента в соответствующий объект `Entry`.
2. Передача объекта `Entry` обработчику элементов, который подвергнет его проверке достоверности JSR-303.

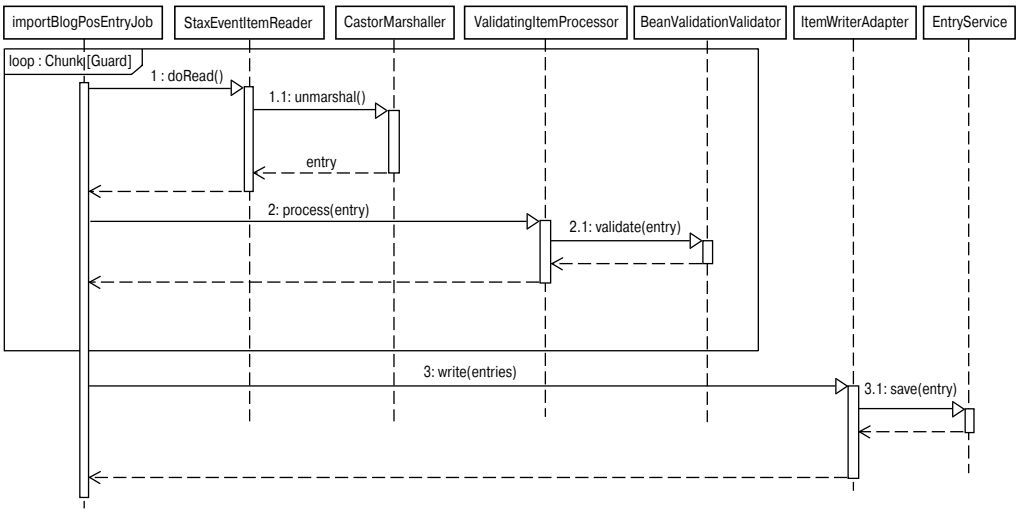


Рис. 21.15. Процесс выполнения задания

После обработки элементов в порции вызывается компонент записи для записи обработанных сообщений блога. В данном случае будет вызван класс `ItemWriterAdapter<T>` (из пакета `org.springframework.batch.item.adapter`), который инициирует процесс записи уровня обслуживания SpringBlog для сохранения каждого сообщения блога.

Подробности конфигурирования

После обсуждения деталей, связанных с проектом, давайте перейдем к рассмотрению конфигурации для приложения SpringBlog. В последующих разделах мы представим различные конфигурации приложения, от дескриптора веб-развертывания до иерархии `WebApplicationContext` в Spring.

Конфигурация дескриптора веб-развертывания

Дескриптор веб-развертывания (`web.xml`) показан в листинге 21.6.

Листинг 21.6. Файл `web.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">

   <!-- Определение корневого контейнера Spring, совместно
       используемого сервлетами и фильтрами -->
   <!-- Разрешить защиту содержимого отправляемой формы -->
   <context-param>
      <param-name>defaultHtmlEscape</param-name>
      <param-value>true</param-value>
   </context-param>
   <context-param>
      <param-name>spring.profiles.active</param-name>
      <!-- Параметры профилей Spring
```

```
        Первый профиль: jpa — реализация JPA
                        mybatis - реализация MyBatis
        Второй профиль: mysql — база данных MySQL
                        h2 - база данных H2
   -->
   <param-value>jpa,h2</param-value>
</context-param>
<context-param>
   <param-name>contextConfigLocation</param-name>
      <param-value>
         /WEB-INF/spring/root-context.xml
         /WEB-INF/spring/datasource.xml
         /WEB-INF/spring/batch-context.xml
         /WEB-INF/spring/*-tx-config.xml
         /WEB-INF/spring/*-service-context.xml
      </param-value>
   </context-param>

<!-- Конфигурация Spring Security -->
<filter>
   <filter-name>springSecurityFilterChain</filter-name>
   <filter-class>org.springframework.web.filter.DelegatingFilterProxy
   </filter-class>
</filter>
<filter-mapping>
   <filter-name>springSecurityFilterChain</filter-name>
   <url-pattern>/*</url-pattern>
</filter-mapping>
<filter>
   <filter-name>CharacterEncodingFilter</filter-name>
   <filter-class>org.springframework.web.filter.CharacterEncodingFilter
   </filter-class>
   <init-param>
      <param-name>encoding</param-name>
      <param-value>UTF-8</param-value>
   </init-param>
   <init-param>
      <param-name>forceEncoding</param-name>
      <param-value>true</param-value>
   </init-param>
</filter>

<filter>
   <filter-name>HttpMethodFilter</filter-name>
   <filter-class>org.springframework.web.filter.HiddenHttpMethodFilter
   </filter-class>
</filter>

<!-- В случае MyBatis следующий фильтр должен быть закомментирован -->
<filter>
  <filter-name>Spring OpenEntityManagerInViewFilter</filter-name>
  <filter-class>org.springframework.orm.jpa.support.OpenEntityManagerInViewFilter
  </filter-class>
</filter>
<filter-mapping>
   <filter-name>CharacterEncodingFilter</filter-name>
   <url-pattern>/*</url-pattern>
</filter-mapping>
```

```
    <filter-mapping>
       <filter-name>HttpMethodFilter</filter-name>
       <url-pattern>/*</url-pattern>
    </filter-mapping>

    <!-- В случае MyBatis следующий фильтр должен быть закомментирован -->
    <filter-mapping>
       <filter-name>Spring OpenEntityManagerInViewFilter</filter-name>
       <url-pattern>/*</url-pattern>
    </filter-mapping>

    <!-- Создание контейнера Spring, совместно используемого
        сервлетами и фильтрами -->
    <listener>
       <listener-class>org.springframework.web.context.ContextLoaderListener
       </listener-class>
    </listener>

    <!-- Сервлет диспетчера Spring MVC для приложения SpringBlog -->
    <servlet>
       <servlet-name>blogAppServlet</servlet-name>
       <servlet-class>org.springframework.web.servlet.DispatcherServlet
       </servlet-class>
       <init-param>
          <param-name>contextConfigLocation</param-name>
          <param-value>/WEB-INF/spring/blogapp-webmvc-config.xml
          </param-value>
       </init-param>
       <load-on-startup>1</load-on-startup>
       <multipart-config>
          <max-file-size>5000000</max-file-size>
       </multipart-config>
    </servlet>

    <servlet-mapping>
       <servlet-name>blogAppServlet</servlet-name>
       <url-pattern>/</url-pattern>
    </servlet-mapping>

    <!-- Сервлет диспетчера Spring MVC для веб-службы REST -->
    <servlet>
       <servlet-name>restful</servlet-name>
       <servlet-class>org.springframework.web.servlet.DispatcherServlet
       </servlet-class>
       <init-param>
          <param-name>contextConfigLocation</param-name>
          <param-value>/WEB-INF/spring/restful-context.xml</param-value>
       </init-param>
       <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
       <servlet-name>restful</servlet-name>
       <url-pattern>/restful/*</url-pattern>
    </servlet-mapping>

</web-app>
```

В листинге 21.6 сначала определено несколько параметров контекста (с помощью дескриптора `<context-param>`).

Как было указано в предыдущем разделе, параметр `spring.profiles.active` определяет активные профили для `WebApplicationContext`, которые также легко заменить с помощью конфигурации запуска tc Server (задавая активный профиль как аргумент виртуальной машины). Затем параметр `contextConfigLocation` сообщает Spring о местоположении загружаемых конфигурационных файлов. Обратите внимание на использование шаблонных символов для загрузки конфигураций из различных профилей; иерархия `WebApplicationContext` будет описана в следующем разделе.

Затем объявляются несколько фильтров. Эти фильтры применяются Spring Security и Spring MVC для обработки входящих запросов. За более подробными сведениями по этому поводу обратитесь в главу 17.

Наконец, определены два сервлета диспетчеров Spring. Сервлет `blogAppServlet` определяет конфигурацию для интерфейсной части, а сервлет `restful` — для веб-службы REST.

Иерархия `WebApplicationContext` в Spring

Все файлы конфигурации `WebApplicationContext` расположены в папке `/WEB-INF/spring`. В табл. 21.2 перечислены имена файлов и описано их назначение.

Таблица 21.2. Файлы конфигурации `WebApplicationContext` для SpringBlog

Имя файла	Назначение
`root-context.xml`	Определяет основную конфигурацию, которая должна быть доступна всему приложению SpringBlog
`security-context.xml`	Определяет конфигурацию для аутентификации и авторизации в приложении SpringBlog. Здесь определены правила для доступа ко всем защищенным ресурсам на основе ролей (включая интерфейсную часть и веб-службу REST). Этот файл импортируется файлом `root-context.xml`
`datasource.xml`	Определяет конфигурацию источника данных СУРБД серверной стороны
`batch-context.xml`	Определяет конфигурацию Spring Integration и Spring Batch. Используется пакетным заданием для импорта сообщений блога из XML-файлов
`jpa-tx-config.xml`	Определяет конфигурацию транзакции и диспетчера сущностей JPA для профиля JPA
`mybatis-tx-config.xml`	Определяет конфигурацию транзакции и фабрики сеансов MyBatis для профиля MyBatis
`jpa-service-context.xml`	Определяет конфигурацию уровня обслуживания для профиля JPA. Например, будут сканироваться только бины Spring для реализации JPA
`mybatis-service-context.xml`	Определяет конфигурацию уровня обслуживания для профиля MyBatis. Например, будут сканироваться только бины Spring для реализации MyBatis
`blogapp-webmvc-config.xml`	Определяет конфигурацию `WebApplicationContext` для сервлета `blogAppServlet`
`restful-context.xml`	Определяет конфигурацию `WebApplicationContext` для сервлета `restful`

На рис. 21.16 показано логическое представление иерархии `WebApplicationContext`.

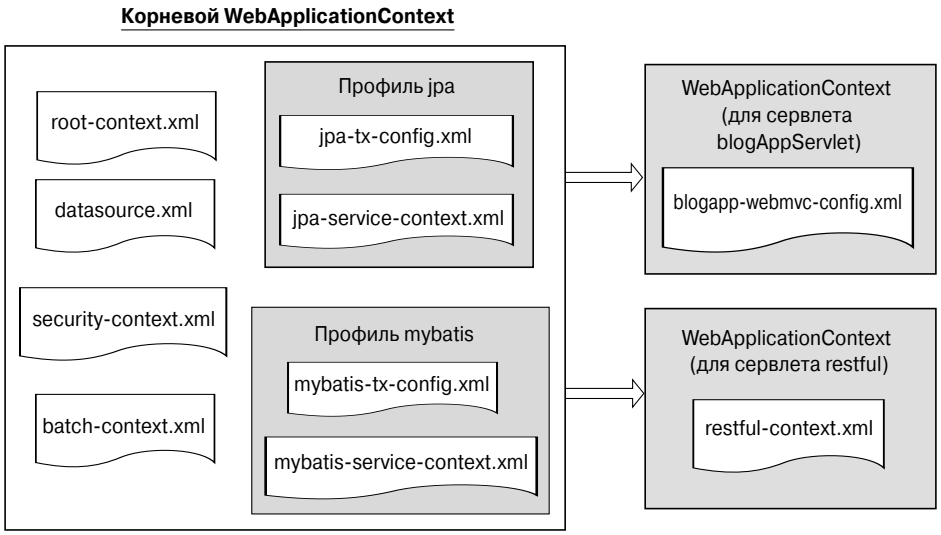


Рис. 21.16. Иерархия `WebApplicationContext` для приложения SpringBlog

На рис. 21.16 корневой `WebApplicationContext` содержит конфигурацию Spring для всех компонентов, которые будут совместно использоваться `WebApplicationContext` всех сервлетов диспетчеров. В корневом `WebApplicationContext` определена конфигурация безопасности, настройки источника данных (для профилей баз данных MySQL и H2), конфигурация пакетного задания и конфигурации уровней обслуживания и постоянства (для профилей JPA и MyBatis).

Детальные описания этих конфигурационных файлов можно найти в соответствующих главах. Например, конфигурация, связанная с JPA, рассматривалась в главе 10, конфигурация для MyBatis — в главе 11, конфигурации Spring Batch и Integration configurations — в главе 20 и т.д.

Конфигурация связывания во время загрузки AspectJ

Как упоминалось в главе 7, для фильтрации нежелательного контента мы будем использовать аннотации в стиле AspectJ (например, `@Aspect` и т.п.), реализуя совет “перед”. Чтобы разрешить эту поддержку, мы должны включить поддержку Spring Framework связывания во время загрузки (load-time weaving — LTW) для AspectJ в среде веб-контейнера. Поскольку для запуска приложения SpringBlog будет применяться tc Server, в этом разделе мы рассмотрим процедуру настройки поддержки LTW для приложения в tc Server. Следует отметить, что для Apache Tomcat процедура выглядит аналогично.

Для начала необходимо сконфигурировать поддержку LTW в `WebApplicationContext`. В приложении SpringBlog эта настройка определена в файле `root-context.xml`, содержимое которого приведено в листинге 21.7.

Листинг 21.7. Файл `root-context.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:context="http://www.springframework.org/schema/context"
```

```
   xmlns:aop="http://www.springframework.org/schema/aop"
   xmlns:util="http://www.springframework.org/schema/util"
   xmlns:task="http://www.springframework.org/schema/task"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
     http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
     http://www.springframework.org/schema/context
     http://www.springframework.org/schema/context/spring-context-3.1.xsd
     http://www.springframework.org/schema/task
     http://www.springframework.org/schema/task/spring-task-3.1.xsd
     http://www.springframework.org/schema/aop
     http://www.springframework.org/schema/aop/spring-aop-3.1.xsd
     http://www.springframework.org/schema/util
     http://www.springframework.org/schema/util/spring-util-3.1.xsd">

   <!-- Свойства приложения -->
   <context:property-placeholder location="classpath:springblog.properties"/>

   <!-- Включить связывание во время загрузки AspectJ -->
   <aop:aspectj-autoproxy/>
   <context:load-time-weaver/>

   <!-- Импортировать контекст безопасности -->
   <import resource="security-context.xml"/>

   <!-- Включить поддержку планирования задач посредством аннотаций -->
   <task:scheduler id="myScheduler" pool-size="10"/>
   <task:annotation-driven scheduler="myScheduler"/>

   <!-- Включить проверку достоверности JSR-303 -->
   <bean id="validator" class="org.springframework.validation.beanvalidation
.LocalValidatorFactoryBean"/>

   <!-- Список слов для замены при фильтрации нежелательного контента -->
   <util:list id="obscenities">
      <value>crap</value>
   </util:list>

</beans>
```

В листинге 21.7 конфигурация, связанная с LTW, выделена полужирным. Сначала дескриптор `<aop:aspectj-autoproxy>` указывает Spring на необходимость поддержки конфигурации АОП в стиле AspectJ и создания прокси АОП “за кулисами”. Затем дескриптор `<context:load-time-weaver>` включает средство LTW в Spring Framework для связывания классов с помощью специального загрузчика классов.

После этого понадобится скопировать инструментальную поддержку Spring Framework для загрузчика классов Tomcat в папку библиотек веб-контейнера. Для Tomcat файл называется `org.springframework.instrument.tomcat-3.1.0.RELEASE.jar`; его можно найти либо в загружаемом пакете Spring Framework, либо в Maven Central. Получите библиотеку и скопируйте ее в папку `lib` сервера Tomcat. Например, если Tomcat расположен в `C:\apache-tomcat-7.0.25`, то JAR-файл библиотеки должен быть помещен в папку `C:\apache-tomcat-7.0.25\lib`.

Хорошая новость для пользователей STS заключается в том, что эта библиотека для сервера tc Server будет на месте после создания нового экземпляра tc Server.

На рис. 21.17 показана папка для экземпляра tc Server по имени `springblog`, которая была создана для развертывания примера приложения.

Далее серверу Tomcat необходимо предоставить информацию о механизме LTW, который мы будем использовать в приложении SpringBlog. Это можно сделать либо в конфи-

гурации tc Server, либо путем создания связанных конфигурационных файлов в проекте (тогда модифицировать конфигурацию сервера не понадобится).

Рис. 21.17. Библиотека Spring LTW для Tomcat

В папке `/src/main/webapp/META-INF` находятся файлы `context.xml` и `aop.xml`, содержимое которых показано в листингах 21.8 и 21.9.

Листинг 21.8. Файл `context.xml`

```
<Context path="/springblog">
<Loader
   loaderClass=
"org.springframework.instrument.classloading.tomcat.TomcatInstrumentableClassLoader"
   useSystemClassLoaderAsParent="false" />
</Context>
```

Листинг 21.9. Файл `aop.xml`

```
<aspectj>
   <weaver>
      <!-- Связывать только классы в пакетах, специфичных для приложения -->
      <include within="com.apress.prospring3.springblog.*" />
   </weaver>
   <aspects>
      <aspect
         name=
"org.springframework.beans.factory.aspectj.AnnotationBeanConfigurerAspect" />
      <aspect
         name=
"org.springframework.transaction.aspectj.AnnotationTransactionAspect" />
   </aspects>
</aspectj>
```

В листинге 21.8 мы указываем серверу tc Server использовать загрузчик классов, предоставляемый Spring Framework, который поддерживает связывание во время загрузки классов в `WebApplicationContext`. В листинге 21.9 мы определяем конфигурацию LTW для AspectJ. Во-первых, мы предоставляем пакет, под которым поддержка AspectJ в Spring должна связывать классы. Во-вторых, дескриптор `<aspects>` определяет используемые аспекты. Для приложения SpringBlog их требуется только два. Аспект `AnnotationBeanConfigurerAspect` (с суффиксом `.aj`) поддерживает класс аспекта с аннотацией `@Configurable`, которая помечает класс как подпадающий под конфигурирование, управляемое Spring. Аспект `AnnotationTransactionAspect` включает поддержку аннотаций `@Transactional` на уровне обслуживания.

Реализация фильтра нежелательного контента обсуждается в следующем разделе.

Детали реализации

После обсуждения особенностей проектирования и конфигурирования давайте перейдем к деталям, связанным с реализацией. Так как кодовая база довольно велика, подробно рассмотреть все классы невозможно. Таким образом, в последующих разделах мы сосредоточим внимание на основных классах реализации и дадим краткий обзор каждого уровня. Будут рассматриваться такие темы, как реализация уровня обслуживания, реализация АОП, планирование заданий, реализация уровня презентаций и т.д.

Реализация уровня обслуживания

Как упоминалось в главе 3, для уровня обслуживания будут разработаны реализации с использованием JPA 2 (с Hibernate и Spring Data JPA) и MyBatis в качестве поставщика постоянства. Детали соответствующих реализаций будут обсуждаться в последующих разделах. В этом разделе мы рассмотрим службы, имеющие отношение к сообщению блога.

В листинге 21.10 показан интерфейс `EntryService`, который определяет службы, предоставляемые приложением SpringBlog для сообщения блога, и разработаем классы реализации с применением JPA 2 и MyBatis.

Листинг 21.10. Интерфейс `EntryService`

```
package com.apress.prospring3.springblog.service;

import java.util.List;

import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;

import com.apress.prospring3.springblog.domain.Entry;
import com.apress.prospring3.springblog.domain.SearchCriteria;

public interface EntryService {

    public List<Entry> findAll();

    public Entry findById(Long id);

    public List<Entry> findByCategoryId(String categoryId);

    public Entry save(Entry entry);

    public void delete(Entry entry);

    public Page<Entry> findAllByPage(Pageable pageable);

    public Page<Entry> findEntryByCriteria(SearchCriteria searchCriteria,
                                           Pageable pageable);

}
```

В листинге 21.10 определены различные методы поиска, некоторые с поддержкой разбиения на страницы, и методы обновления данных (включая операции вставки, обновления и удаления). С целью упрощения мы ограничимся обсуждением реализации метода `save()`, определение которого выделено полужирным.

За подробными сведениями о проектировании и реализации DOM-модели для сообщения блога обращайтесь в главу 12. Если вы пропустили эту главу, то самое время ее прочитать, чтобы понять объектную модель предметной области.

Реализация уровня обслуживания с использованием JPA

В реализации уровня обслуживания на основе JPA 2 в качестве поставщика постоянства будет использоваться Hibernate. Кроме того, для упрощения разработки логики постоянства будет применяться абстракция репозитория Spring Data JPA. Как упоминалось в разделе "Иерархия `WebApplicationContext` в Spring", конфигурация для реализации JPA предоставляется в файлах `jpa-tx-config.xml` и `jpa-service-context.xml`. Детальное объяснение конфигурации ищите в главе 10.

При написании логики постоянства будет использоваться абстракция репозитория Spring Data JPA, поэтому мы должны реализовать интерфейс репозитория для объекта `Entry`. В листинге 21.11 представлен интерфейс `EntryRepository`.

Листинг 21.11. Интерфейс `EntryRepository`

```
package com.apress.prospring3.springblog.repository;

import java.util.List;

import org.joda.time.DateTime;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.PagingAndSortingRepository;
import org.springframework.data.repository.query.Param;

import com.apress.prospring3.springblog.domain.Entry;

public interface EntryRepository extends PagingAndSortingRepository<Entry, Long>
{

   public List<Entry> findByCategoryId(String categoryId);

   @Query("select e from Entry e where e.subject like :subject and e.categoryId like
:categoryId and e.postDate between :fromPostDate and :toPostDate")
   public Page<Entry> findEntryByCriteria(
      @Param("subject") String subject,
      @Param("categoryId") String categoryId,
      @Param("fromPostDate") DateTime fromPostDate,
      @Param("toPostDate") DateTime toPostDate, Pageable pageable);
}
```

Интерфейс `EntryRespository` в листинге 21.11 расширяет интерфейс `PagingAndSortingRepository<T,ID extends Serializable>` из Spring Data Commons, чтобы предоставить базовые операции CRUD, а также поддержку функций разбиения на страницы и сортировки, требуемых на уровне обслуживания. Поскольку интерфейс `PagingAndSortingRepository<T,ID extends Serializable>` уже предоставляет метод `save()`, его не понадобится определять в интерфейсе `EntryRepository`.

В листинге 21.12 приведен фрагмент кода для реализации метода `save()`.

Листинг 21.12. Класс `EntryServiceImpl` для реализации JPA 2

```
package com.apress.prospring3.springblog.service.jpa;

// Операторы импорта опущены.

@Service("entryService")
@Repository
@Transactional
public class EntryServiceImpl implements EntryService{
```

```
    @Autowired
    private EntryRepository entryRepository;

    // Остальной код не показан.

    @Override
    public Entry save(Entry entry) {
        // Если это новая запись, установить дату отправки в текущую дату.
        if (entry.getId() == null) {
            entry.setPostDate(new DateTime());
        }
        return entryRepository.save(entry);
    }
}
```

Класс из листинга 21.12 расположен в пакете com.apress.prospring3.springblog.service.jpa, который содержит реализацию JPA 2 для всех интерфейсов уровня обслуживания. Интерфейс EntryRepository автосвязан с классом. Метод save() будет использоваться для операций вставки и обновления, поэтому сначала проверяется свойство id. Если оно равно null, это новый объект, и свойство postDate устанавливается в значение текущей даты. После этого вызывается метод CrudRepository<T,ID extends Serializable>.save() (унаследованный из интерфейса PagingAndSortingRepository<T,ID extends Serializable>), который сохраняет объект посредством Hibernate.

Реализация уровня обслуживания с использованием MyBatis

В реализации уровня обслуживания на основе MyBatis для определения отображений между SQL и DOM-моделью будет использоваться XML-конфигурация SQL-отображения. Как упоминалось в разделе “Иерархия WebApplicationContext в Spring”, конфигурация для реализации MyBatis предоставляется в файлах mybatis-tx-config.xml и mybatis-service-context.xml.

Детальное объяснение конфигурации ищите в главе 11.

В листинге 21.13 приведено содержимое файла отображений для объекта Entry (файл называется EntryMapper.xml и расположен в папке /src/main/resources/com/apress/prospring3/springblog/persistence).

Листинг 21.13. Файл EntryMapper.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper namespace="com.apress.prospring3.springblog.persistence.EntryMapper">

    <!-- Остальной код не показан -->

    <insert id="insertEntry" parameterType="Entry"
            useGeneratedKeys="true" keyProperty="id">
        INSERT INTO ENTRY (SUBJECT, BODY, POST_DATE, CATEGORY_ID, SUB_CATEGORY_ID,
            CREATED_BY, CREATED_DATE, LAST_MODIFIED_BY, LAST_MODIFIED_DATE)
        VALUES (#{subject}, #{body}, #{postDate}, #{categoryId}, #{subCategoryId},
            #{createdBy}, #{createdDate}, #{lastModifiedBy}, #{lastModifiedDate})
    </insert>

</mapper>
```

Если вы читали главу 11, то код в листинге 21.13 должен быть знаком. Отображение insertEntry конструирует SQL-оператор INSERT на основе предоставленного объекта Entry, атрибут useGeneratedKeys указывает MyBatis на необходимость использования ключа, генерируемого СУРБД серверной части (H2 или MySQL), и атрибут keyProperty задает свойство для столбца первичного ключа.

В листинге 21.14 показан фрагмент кода с реализацией метода save().

Листинг 21.14. Класс EntryServiceImpl для реализации MyBatis

```
package com.apress.prospring3.springblog.service.mybatis;

// Оператор импорта опущен.

@Service("entryService")
@Repository
@Transactional
public class EntryServiceImpl implements EntryService {

   @Autowired
   private EntryMapper entryMapper;

   // Остальной код не показан.

   @Override
   public Entry save(Entry entry) {
      if (entry.getId() == null) {
         entryMapper.insertEntry(entry);
      } else {
         entryMapper.updateEntry(entry);
      }
      return entry;
   }
}
```

В листинге 21.14 с классом автосвязан интерфейс EntryMapper, который будет создаваться автоматически на основе объявления анонимного бина с классом org.mybatis.spring.mapper.MapperScannerConfigurer в файле mybatis-tx-config.xml. Затем в методе save() проверяется свойство id для выяснения, новый ли это объект. Если это новый объект, вызывается метод insertEntry() и MyBatis будет использовать определение отображения с идентификатором insertEntry для конструирования SQL-оператора INSERT.

В главе 10 упоминалось, что в реализации JPA 2 использовалось средство аудита Spring Data JPA, с помощью которого для каждого объекта Entry заполнялась базовая информация аудита — дата создания, пользователь, который создал, дата последнего изменения и пользователь, внесший последнее изменение. В MyBatis подобный эквивалент отсутствует. В результате придется воспользоваться подключаемым модулем MyBatis и реализовать соответствующую логику, которая должна выполняться перед операцией вставки. В листинге 21.15 представлен класс подключаемого модуля MyBatis, предназначенный для обновления базовой информации аудита.

Листинг 21.15. Класс MyBatisPlugin

```
package com.apress.prospring3.springblog.mybatis.plugin;

import java.util.Properties;

import org.apache.ibatis.executor.Executor;
import org.apache.ibatis.executor.parameter.DefaultParameterHandler;
```

```
import org.apache.ibatis.mapping.MappedStatement;
import org.apache.ibatis.plugin.Interceptor;
import org.apache.ibatis.plugin.Intercepts;
import org.apache.ibatis.plugin.Invocation;
import org.apache.ibatis.plugin.Signature;
import org.joda.time.DateTime;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.Auditable;

import com.apress.prospring3.springblog.auditor.AuditorAwareBean;

@Intercepts(
   {@Signature(type = Executor.class, method = "update",
      args = {MappedStatement.class, Object.class }) }
)
public class MyBatisPlugin implements Interceptor {

   @Autowired
   private AuditorAwareBean auditorAwareBean;

   @Override
   public Object intercept(Invocation invocation) throws Throwable {
      return invocation.proceed();
   }

   @Override
   public Object plugin(Object target) {

    if (target instanceof DefaultParameterHandler) {
      DefaultParameterHandler paramHandler = (DefaultParameterHandler) target;
          Object obj = paramHandler.getParameterObject();

          if (obj != null) {

             if (obj instanceof Auditable) {

                DateTime currentTimeStamp = new DateTime();
                String currentUser = auditorAwareBean.getCurrentAuditor();

                Auditable auditable = (Auditable) obj;
                if (auditable.getCreatedDate() == null) {
                   auditable.setCreatedDate(currentTimeStamp);
                   auditable.setCreatedBy(currentUser);
                }
               auditable.setLastModifiedBy(currentUser);
               auditable.setLastModifiedDate(currentTimeStamp);

             }
          }
    }

    return target;
   }

   @Override
   public void setProperties(Properties properties) {
   }
}
```

В листинге 21.15 аннотация `@Intercepts` из MyBatis используется для указания, что этот класс будет перехватывать операции MyBatis. Вложенная аннотация `@Signature` определяет класс, метод и аргументы, которые необходимо перехватывать. В рассматриваемом случае это операция обновления (`update`).

Внутри класса реализован интерфейс `Interceptor` из MyBatis. Основная логика для обновления свойств аудита в объекте предметной области находится в методе `plugin()`. В этом методе осуществляется проверка, относится ли целевой объект к типу `DefaultParameterHandler`, который MyBatis использует для отображения свойств объекта предметной области на SQL-операторы. Если это так, производится проверка, содержит ли объект предметной области свойства, поддерживающие аудит, путем выяснения возможности присваивания класса интерфейсу `Auditable`, определенному в Spring Data Commons, поскольку этот интерфейс должны реализовывать все объекты предметной области, для которых нужен аудит свойств.

Если все условия удовлетворены, все поля аудита будут соответствующим образом обновлены перед тем, как MyBatis выполнит операцию отображения. Как видите, система подключаемого модуля MyBatis позволяет перехватывать различные операции отображения для выполнения специальной логики.

Конфигурация этого подключаемого модуля находится в определении бина `sqlSessionFactory` в файле `mybatis-tx-config.xml`.

Фильтрация нежелательного контента с использованием АОП

Фильтр нежелательного контента в приложении SpringBlog является основным средством для демонстрации использования АОП в Spring при реализации сквозной функциональности. В этом разделе мы рассмотрим детали его реализации. Конфигурирование Spring и необходимая настройка сервера tc Server были описаны в разделе “Конфигурация связывания во время загрузки AspectJ” ранее в этой главе.

В листинге 21.16 показан интерфейс `ObscenityFilter`, который определяет методы для фильтра нежелательного контента.

Листинг 21.16. Интерфейс `ObscenityFilter`

```
package com.apress.prospring3.springblog.common.aop;

public interface ObscenityFilter {

   public boolean containsObscenities(String data);

   public String obfuscateObscenities(String data);
}
```

Для фильтра нежелательного контента определены два метода. Метод `containsObscenities()` предназначен для проверки наличия внутри данных нежелательного контента, а метод `obfuscateObscenities()` — для преобразования нежелательного контента в бессмысленный набор случайных букв, используя алгоритм ROT13.

В листинге 21.17 представлен класс реализации интерфейса `ObscenityFilter`.

Листинг 21.17. Класс `ListBasedObscenityFilter`

```
package com.apress.prospring3.springblog.common.aop;

import java.util.List;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import javax.annotation.Resource;
import org.springframework.stereotype.Service;

@Service("obscenityFilter")
public class ListBasedObscenityFilter implements ObscenityFilter {

   private List<String> obscenities = null;
```

```
private Pattern obscenityPattern = null;

@Resource(name="obscenities")
public void setObscenities(List<String> obscenities) {
    this.obscenities = obscenities;
    buildRegex();
}

private void buildRegex() {
    StringBuffer sb = new StringBuffer();
    int size = obscenities.size();

    for (int x = 0; x < size; x++) {
        if (x != 0) {
            sb.append("|");
        }
        sb.append("(");
        sb.append(obscenities.get(x));
        sb.append(")");
    }

    obscenityPattern = Pattern.compile(sb.toString(), Pattern.CASE_INSENSITIVE);
}
/**
 * Возвращает true, если данные содержат нежелательный
 * контент, и false — в противном случае.
 */
public boolean containsObscenities(String data) {
    Matcher m = obscenityPattern.matcher(data);
    return m.find();
}

/**
 * Затенение нежелательного контента в данных типа String.
 */
public String obfuscateObscenities(String data) {
    Matcher m = obscenityPattern.matcher(data);
    StringBuffer out = new StringBuffer(data.length());

    while (m.find()) {
        if (m.group(0) != null) {
            m.appendReplacement(out, rot13(m.group(0)));
        }
    }

    m.appendTail(out);

    return out.toString();
}

/**
 * Кодирование ROT13 значения String.
 * @param in
 * @return Encoded string
 */
private String rot13(String in) {
    char[] chars = in.toCharArray();

    for (int x = 0; x < chars.length; x++) {
        char c = chars[x];
        if (c >= 'a' && c <= 'm')
            c += 13;
```

```
            else if (c >= 'n' && c <= 'z')
                c -= 13;
            else if (c >= 'A' && c <= 'M')
                c += 13;
            else if (c >= 'A' && c <= 'Z') c -= 13;
            chars[x] = c;
        }
        return new String(chars);
    }
}
```

Класс `ListBasedObscenityFilter` в листинге 21.17 реализует фильтр нежелательного контента с применением списка предварительно заданных нежелательных слов, который определен бином `obscenities` в файле `root-context.xml` и внедрен в класс через аннотацию `@Resource`. На основе этого списка построено регулярное выражение, которое будет использоваться методом `containsObscenities()` для проверки существования нежелательных слов. Метод `obfuscateObscenities()` также применяет это регулярное выражение для поиска нежелательных слов и затенения с помощью алгоритма ROT13.

Наконец, имеется класс совета “перед”, который будет выполнять фильтрацию для всех операций внутри уровня обслуживания с аргументом, которому может быть присвоен интерфейс `BlogPosting`. Код класса совета приведен в листинге 21.18.

Листинг 21.18. Класс `ObscenityFilterAdvice`

```
package com.apress.prospring3.springblog.common.aop;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Configurable;
import org.springframework.stereotype.Component;

import com.apress.prospring3.springblog.domain.BlogPosting;

@Component
@Configurable
@Aspect
public class ObscenityFilterAdvice {

    @Autowired
    private ObscenityFilter obscenityFilter;

    public void setObscenityFilter(ObscenityFilter obscenityFilter) {
        this.obscenityFilter = obscenityFilter;
    }

    @Before("execution(* com.apress.prospring3.springblog.service..*(..))")
    public void filterObscenities(JoinPoint joinPoint)
          throws Throwable {
        Object[] args = joinPoint.getArgs();
        for (int x = 0; x < args.length; x++) {
            if (args[x] instanceof BlogPosting) {
                BlogPosting arg = (BlogPosting) args[x];
                if (obscenityFilter.containsObscenities(arg.getBody())) {
                    arg.setBody(obscenityFilter.obfuscateObscenities(arg.getBody()));
                }
```

```
                if (obscenityFilter.containsObscenities(arg.getSubject())) {
                    arg.setSubject(obscenityFilter
                            .obfuscateObscenities(arg.getSubject()));
                }
            }
        }
    }
}
```

В листинге 21.18 видно, что на уровне класса применено несколько аннотаций. Аннотация `@Component` указывает, что это бин Spring. Аннотация `@Configurable` определяет, что этот совет должен быть сконфигурирован Spring, и Spring создаст прокси АОП для связанных классов. Аннотация `@Aspect` из AspectJ указывает Spring, что это совет АОП, и включит поддержку аннотаций в стиле AspectJ.

Внутри совета автосвязан интерфейс `ObscenityFilter`. Метод `filterObscenities()` аннотирован с помощью аннотации `@Before` вместе с выражением среза, указывающим, что это совет "перед", который должен перехватывать все методы уровня обслуживания. В этом методе аргументы будут проверяться на предмет возможности присваивания интерфейса `BlogPosting`. Если это так, аргумент должен быть объектом `Entry` или `Comment`, и логика фильтрации нежелательного контента будет применена.

Чтобы увидеть это в действии, можете отправить новую запись, которая содержит слово "crap" в поле темы или тела. После сохранения записи вы заметите, что это слово было заменено строкой "penc", которая представляет собой результат применения алгоритма ROT13. Взгляните на рис. 3.6 и 3.7 в главе 3.

Планирование задания для очистки данных аудита

Еще одной возможностью, которая демонстрируется в приложении SpringBlog, является планирование заданий, с помощью которого мы реализуем ежедневно выполняющееся задание для очистки данных хронологии аудита объектов `Entry` и `Comment` со сроком давности более 30 дней. В этом разделе мы рассмотрим детали этой реализации.

Как уже было показано в главе 15, планирование заданий в Spring реализуется очень просто, и в приложении SpringBlog при планировании задания для очистки данных аудита мы используем стиль с аннотациями. В листинге 21.19 приведен фрагмент кода из файла `root-context.xml`, который включает поддержку планирования в стиле с аннотациями.

Листинг 21.19. Конфигурирование планирования на основе аннотаций в Spring

```
<!-- Включить поддержку планирования задач посредством аннотаций -->
<task:scheduler id="myScheduler" pool-size="10"/>
<task:annotation-driven scheduler="myScheduler"/>
```

Дескриптор `<task:scheduler>` в листинге 21.19 заставляет Spring создать экземпляр `ThreadPoolTaskScheduler` с указанным размером пула потоков. Затем дескриптор `<task:annotation-driven>` разрешает определение планирования с помощью аннотаций. В листинге 21.20 показан интерфейс `HousekeepingService`, определяющий задания по обслуживанию данных, которые должны выполняться приложением SpringBlog.

Листинг 21.20. Интерфейс HousekeepingService

```
package com.apress.prospring3.springblog.service;

public interface HousekeepingService {

    /**
     * Запланированное задание для очистки записей аудита.
     */
    public void auditPurgeJob();

}
```

В интерфейсе определен только один метод `auditPurgeJob()`, который содержит логику для очистки старых данных хронологии аудита. В листинге 21.21 представлен класс реализации JPA 2.

Листинг 21.21. Класс HousekeepingServiceImpl

```
package com.apress.prospring3.springblog.service.jpa;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Repository;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import com.apress.prospring3.springblog.service.HousekeepingService;

@Service("housekeepingService")
@Repository
@Transactional
public class HousekeepingServiceImpl implements HousekeepingService {

    @Value("${audit.record.history.days}")
    private int auditHistoryDays;

    @Scheduled(cron="0 0 0 * * ?")
    public void auditPurgeJob() {
        // Логика очистки записей аудита.
    }
}
```

В листинге 21.21 количество дней для хранения хронологии аудита вынесено в файл `/src/main/resources/springblog.properties`. Метод `auditPurgeJob()` применяется с помощью аннотации `@Scheduled`, которая указывает Spring, что это запланированное задание. В атрибуте `cron` указано выражение, означающее, что это задание будет запускаться в полночь каждые сутки. Мы оставили тело метода пустым, но, как вы должны понимать, необходимую логику реализовать очень просто. Нужно лишь выполнить операцию пакетного обновления для удаления из таблиц `ENTRY_H` и `COMMENT_H` (см. файл `schema.sql`) записей с датой создания старше указанного количества дней.

Уровень презентаций

Для уровня презентаций в приложении SpringBlog применима большая часть тем, рассмотренных в главе 17. Ниже отмечены основные моменты.

- Для уровня презентаций будет реализован шаблон “модель-представление-контроллер” с использованием Spring MVC и JSPX в качестве технологии представлений.
- На уровне презентаций повсеместно будет использоваться поддержка Spring MVC для интернационализации, оформления темами и веб-служб REST (для взаимодействия в стиле Ajax с помощью jQuery).
- В качестве технологии шаблонов страниц будет применяться Apache Tiles. Кроме того, для распознавания отображаемых представлений на основе логических имен будет использоваться интеграция Spring с Apache Tiles.
- Для привязки запросов и преобразования строковых форматов в Java-типы будет применяться поддержка проверки достоверности бинов JSR-303, преобразования типов и форматирования, доступная в версии Spring 3.
- Для улучшения взаимодействия с пользователями в представлениях будут использоваться библиотеки jQuery и jQuery UI. Кроме того, для специфических целей будут применяться различные подключаемые модули jQuery. Например, jqGrid будет использоваться для отображения записей блога в виде сетки, поддерживающей разбиение на страницы указанного размера и сортировку, а CKEditor — для ввода форматированного текста.
- Для аутентификации и авторизации защищенных ресурсов будет использоваться Spring Security. Например, отправлять новые сообщения в блог и комментировать существующие записи могут только пользователи, вошедшие в приложение, а просматривать хронологию аудита записей блога — только администраторы.
- Для загрузки файловых вложений будет применяться поддержка функциональной возможности загрузки файлов Servlet 3.0, обеспечиваемая Spring Framework 3.1.

Поскольку большинство тем, рассмотренных в главе 17, применимо к уровню презентаций в приложении SpringBlog, настоятельно рекомендуется прочитать главу 17, если вы ее пропустили. В последующих разделах мы отметим некоторые основные детали реализации уровня презентаций в приложении SpringBlog. В следующем разделе рассматривается структура папок с файлами веб-ресурсов.

Структура папок с файлами веб-ресурсов

На уровне презентаций в приложении SpringBlog все ресурсные файлы (JSPX-файл представления, шаблон компоновки Apache Tiles, стилевые таблицы CSS, файлы JavaScript-кода и связанных библиотек, конфигурационные файлы `WebApplicationContext`) хранятся в папке `/src/main/webapp`.

На рис. 21.18 показана структура папок для приложения SpringBlog в STS.

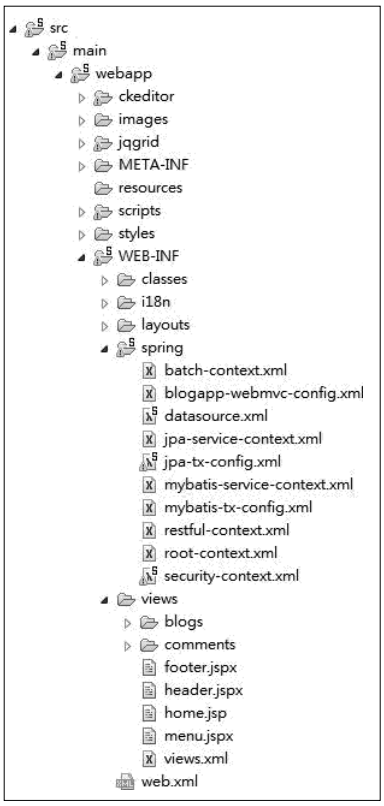


Рис. 21.18. Структура папок с файлами веб-ресурсов в приложении SpringBlog

Структура в основном та же, что и в примере приложения, которое разрабатывалось в главе 17. Ресурсы JavaScript (включая jQuery, jQuery UI, jqGrid, CKEditor и т.д.) хранятся в папках `scripts`, `jqgrid` и `ckeditor`. В папке `styles` содержатся CSS-файлы, поддерживающие оформление темами SpringBlog, а в папке `images` хранятся файлы изображений.

Папка `layouts` содержит шаблон компоновки страниц для Apache Tiles. В папке `spring` хранятся конфигурационные файлы `WebApplicationContext` для Spring. В папке `views` содержатся все JSPX-файлы представлений для интерфейсной части, в том числе компоненты шаблона, такие как заголовок, меню, нижний колонтитул и т.д. В подпапках `blogs` и `comments` располагаются файлы представлений для вывода списка, просмотра, добавления и редактирования сообщений блога и комментариев к ним.

Класс контроллера

Классы контроллеров являются центральным компонентом логики презентации интерфейсной части. Они обрабатывают запросы, получаемые сервлетом диспетчера от интерфейсной части либо от клиентов веб-служб REST. На основе запроса классы контроллеров выполняют требуемую логику обработки, взаимодействуют с уровнем обслуживания и возвращают логическое представление сервлету диспетчера (или производят запись непосредственно в тело ответа). В этом разделе мы также рассмотрим класс `EntryController`, который обрабатывает запросы к записям блога. В частности, мы обсудим создание нового сообщения блога.

В листинге 21.22 приведен фрагмент кода для методов, связанных с созданием нового сообщения блога, в классе `EntryController` из приложения SpringBlog.

Листинг 21.22. Класс `EntryController`

```
package com.apress.prospring3.springblog.web.blogapp.controller;

// Операторы импорта опущены.

@RequestMapping("/blogs")
@Controller
public class EntryController {

    final Logger logger = LoggerFactory.getLogger(EntryController.class);

    @Autowired
    private MessageSource messageSource;

    @Autowired
    private EntryService entryService;

    @Autowired
    private CategoryService categoryService;

    @RequestMapping(method = RequestMethod.POST)
    public String create(@Valid Entry entry,
           BindingResult bindingResult,
           Model uiModel,
           HttpServletRequest httpServletRequest,
           RedirectAttributes redirectAttributes,
           Locale locale) {
        logger.info("Creating entry");
        if (bindingResult.hasErrors()) {
           uiModel.addAttribute("message", new Message("error",
             messageSource.getMessage("entry_save_fail", new Object[]{}, locale)));
           uiModel.addAttribute("entry", entry);
```

```
            populateSelectBox(uiModel, entry);
            return "blogs/create";
        }
        uiModel.asMap().clear();
        redirectAttributes.addFlashAttribute("message", new Message("success",
          messageSource.getMessage("entry_save_success", new Object[]{}, locale)));
        logger.info("Entry id: " + entry.getId());
        entryService.save(entry);
        return "redirect:/blogs/"
            + UrlUtil.encodeUrlPathSegment(entry.getId().toString(),
            httpServletRequest);
    }

    @PreAuthorize("isAuthenticated()")
    @RequestMapping(params = "form", method = RequestMethod.GET)
    public String createForm(Model uiModel) {
        Entry entry = new Entry();
        uiModel.addAttribute("entry", entry);
        populateSelectBox(uiModel, entry);
        return "blogs/create";
    }

    // Остальной код не показан.
}
```

В листинге 21.22 показаны два метода, имеющие отношение к процессу создания нового сообщения блога. Метод `createForm()` отвечает за создание нового объекта предметной области `Entry`, сохранение его в интерфейсе `Model` и возврат логического представления для отображения представления обновления записи (определенного логическим представлением `blogs/create`). Обратите внимание, что этот метод снабжен аннотацией `@PreAuthorize`, которая представляет собой поддержку авторизации на уровне методов в Spring Security, т.к. создавать сообщения в блоге разрешено только пользователям, вошедшим в приложение.

Метод `create()` обрабатывает отправку формы после того, как пользователи ввели детали сообщения блога. Аннотация `@Valid` возле аргумента типа `Entry` указывает Spring MVC на необходимость выполнения проверки достоверности JSR-303 для отправленного объекта. Все нарушения ограничений будут сохраняться в `BindingResult`. Если обнаруживаются ошибки, в интерфейсе `Model` сохраняется сообщение и возвращается то же самое представление редактирования записи блога с отображенными сообщениями об ошибках. На рис. 21.19 можно видеть примеры сообщений проверки достоверности, которые отображаются, когда просто щелкнуть на кнопке Save (Сохранить), не вводя никакой информации.

В листинге 21.23 приведен фрагмент кода из класса `AbstractBlogPosting` (который расширяется классом `Entry`) с ограничениями проверки JSR-303, определенными для свойства `subject`.

Рис. 21.19. Проверка достоверности JSR-303 в приложении SpringBlog

Листинг 21.23. Класс `AbstractBlogPosting`

```
package com.apress.prospring3.springblog.domain;

// Операторы импорта опущены.

@MappedSuperclass
@Audited
public abstract class AbstractBlogPosting implements BlogPosting,
                      Auditable<String, Long>, Serializable {

   protected Long id;
   protected String subject;
   protected String body;
   protected DateTime postDate;
   protected String createdBy;
   protected DateTime createdDate;
   protected String lastModifiedBy;
   protected DateTime lastModifiedDate;
   protected int version;

   // Остальной код не показан.

   @NotEmpty(message="{validation.posting.subject.NotEmpty.message}")
   @Size(min=10, max=50, message="{validation.posting.subject.Size.message}")
   @Column(name = "SUBJECT")
   public String getSubject() {
      return subject;
   }
}
```

В листинге 21.23 аннотации `@NotEmpty` и `@Size`, примененные к методу `getSubject()`, определяют ограничения проверки JSR-303 для свойства `subject`. В `message` указывается код, а сами сообщения вынесены в файлы сообщений с целью поддержки интернационализации (за дополнительными сведениями обращайтесь в главу 17). Объявление бина `<mvc:annotation-driven validator="validator"/>` в файле `blogapp-webmvc-config.xml` включает проверку достоверности.

Преобразование типов и форматирование

Новая система преобразования типов и форматирования Spring Framework 3 в приложении SpringBlog используется для привязки, преобразования и форматирования данных между представлениями интерфейсной части и классами контроллеров. Дескриптор `<mvc:annotation-driven>` также активизирует эту новую систему преобразования типов и форматирования. После ее включения Spring будет автоматически преобразовывать общие Java-типы в строковые представления. Кроме того, предусмотрены аннотации для определения желаемого формата. В этом разделе мы обсудим применение аннотации `@DateTimeFormat` для управления форматом типа `DateTime` из Joda-Time.

В листинге 21.24 приведен еще один фрагмент кода из класса `AbstractBlogPosting` со случаями применения аннотации `@DateTimeFormat`.

Листинг 21.24. Класс `AbstractBlogPosting`

```
package com.apress.prospring3.springblog.domain;

// Операторы импорта опущены.

@MappedSuperclass
@Audited
public abstract class AbstractBlogPosting implements BlogPosting,
                  Auditable<String, Long>, Serializable {

   protected Long id;
   protected String subject;
   protected String body;
   protected DateTime postDate;
   protected String createdBy;
   protected DateTime createdDate;
   protected String lastModifiedBy;
   protected DateTime lastModifiedDate;
   protected int version;

   @Column(name = "CREATED_DATE")
   @Type(type="org.joda.time.contrib.hibernate.PersistentDateTime")
   @DateTimeFormat(iso=ISO.DATE_TIME)
   public DateTime getCreatedDate() {
      return this.createdDate;
   }

   @Column(name = "LAST_MODIFIED_DATE")
   @Type(type="org.joda.time.contrib.hibernate.PersistentDateTime")
   @DateTimeFormat(iso=ISO.DATE_TIME)
   public DateTime getLastModifiedDate() {
      return this.lastModifiedDate;
   }

   // Остальной код не показан.
}
```

Аннотации для настройки формата даты-времени в листинге 21.24 выделены полужирным. Аннотация `@DateTimeFormat` применена к свойствам `createdDate` и `lastModifiedDate` с использованием формата типа `ISO.DATE_TIME`, который соответствует формату `yyyy-MM-dd'T'hh:mm:ss.SSSZ` (например, 2012-01-30T13:53:52.324+08:00). Причина в том, что для дат создания и последнего изменения также необходимо знать и время. Обратите внимание, что Spring обладает встроенной поддержкой форматирования типа `DateTime` из Joda-Time.

На рис. 21.20 показана страница представления для записи блога в приложении SpringBlog, на которой можно видеть преобразование типов и форматирование в действии.

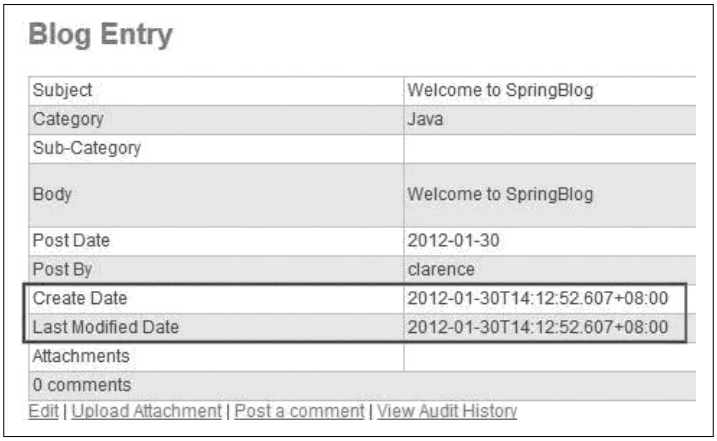


Рис. 21.20. Преобразование типов и форматирование в SpringBlog

Как видно на рис. 21.20, даты создания и последнего изменения отображаются в заданном формате.

В приложении SpringBlog реализовано и много других средств, среди которых интернационализация, оформление темами, jQuery, разбиение на страницы, Ajax для динамического обновления списком категорий (другими словами, когда родительская категория изменяется, поле со списком выбора подкатегории обновляется с помощью Ajax-вызова), загрузка файлов и т.д. Тем не менее, реализации основаны на Ajax-вызовах из jQuery и том же самом уровне Spring MVC, и они в основном такие же, как реализация разбиения на страницы в таблице данных с использованием jqGrid; таким образом, поскольку все аспекты реализации подробно рассматривались в главе 17, здесь они повторяться не будут.

Резюме

В этой главе мы обсудили детали проектировании и реализации примера приложения SpringBlog, разработанного для настоящей книги.

Во-первых, были даны инструкции по настройке проекта, а также по переключению между различными источниками данных и реализациями с помощью поддержки профилей определений бинов, появившейся в версии Spring Framework 3.1.

Во-вторых, мы тщательно разобрали проект приложения SpringBlog. Рассматриваемые темы включали модель данных, объектную модель предметной области и UML-модель для основных сценариев использования, встречающихся в рамках приложения.

В-третьих, мы объяснили конфигурацию приложения SpringBlog, в том числе конфигурацию дескриптора веб-развертывания, а также иерархию `WebApplicationContext` в Spring.

Наконец, в-четвертых, мы детально описали реализацию основных функциональных возможностей, которые демонстрируют мощные средства Spring Framework. Рассматривались такие темы, как реализация уровня обслуживания, применение АОП для фильтрации нежелательного контента, планирование задач и построение уровня презентаций.

ГЛАВА 22

Поддержка написания сценариев в Spring

В предшествующих главах вы узнали, как Spring Framework помогает Java-разработчикам создавать JEE-приложения. За счет использования механизма внедрения зависимостей Spring Framework и его интеграции с каждым уровнем (через библиотеки внутри собственных модулей Spring Framework или путем взаимодействия с библиотеками третьих сторон) можно упростить реализацию и сопровождение бизнес-логики.

Однако для всей логики, разработанной до сих пор, использовался язык Java. Хотя Java является одним из наиболее успешных языков программирования за всю историю отрасли, его по-прежнему критикуют за некоторые недостатки, такие как структура языка или отсутствие всесторонней поддержки в областях, подобных массивной параллельной обработке.

Например, одна из характеристик языка Java заключается в том, что все переменные являются статически типизированными. Другими словами, в Java-программе каждая объявленная переменная должна иметь связанный с ней статический тип (`String`, `int`, `Object`, `ArrayList` и т.д.). Тем не менее, в ряде сценариев более предпочтительной может оказаться динамическая типизация, которая поддерживается динамическими языками вроде JavaScript.

Для решения таких проблем были разработаны многочисленные языки сценариев. В число самых популярных языков сценариев входят JavaScript, Groovy, Scala, Ruby и Erlang. Почти все эти языки поддерживают динамическую типизацию и спроектированы на предоставление средств, которые не доступны в Java, а также для удовлетворения специфических целей. Например, Scala (`www.scala-lang.org`) комбинирует шаблоны функционального программирования с объектно-ориентированными шаблонами и поддерживает более развитую и масштабируемую модель параллельного программирования с концепциями акторов и передачи сообщений. Кроме того, Groovy (`http://groovy.codehaus.org`) предоставляет упрощенную модель программирования и поддерживает реализацию специфичных для предметной области языков (domain-specific language — DSL), которые делают код приложения проще в чтении и сопровождении.

Еще одной важной концепцией, которую языки сценариев предлагают Java-разработчикам, являются замыкания (мы обсудим их более подробно далее в этой главе). Говоря упрощенно, замыкание — это фрагмент (или блок) кода, помещенный в объект. Замыкание является исполняемым компонентом подобно Java-методу и может получать параметры, а также возвращать объекты и значения. Кроме того, оно также представляет собой нормальный объект, который может передаваться с помощью ссылки куда угодно в приложении, как и любой объект POJO в Java.

В этой главе мы рассмотрим некоторые основные концепции, связанные с языками сценариев, уделяя особое внимание Groovy; вы увидите, что Spring Framework может гладко работать с языком сценариев, предоставляя специфическую функциональность Spring-приложениям. В частности, в этой главе будут освещены следующие темы.

- *Поддержка написания сценариев в Java.* В рамках JCP спецификация JSR-223 (Написание сценариев для платформы Java) регламентирует поддержку языков сценариев в Java; она доступна в Java, начиная с JSE 6. Мы предоставим обзор поддержки написания сценариев в Java.
- *Язык Groovy.* Мы представим высокоуровневое введение в язык Groovy, который является одним из наиболее популярных языков сценариев, используемых вместе с Java, особенно при работе с платформой Spring Framework.
- *Использование Groovy в Spring.* Платформа Spring Framework обеспечивает комплексную поддержку для языков сценариев. В версии Spring Framework 3.1 имеется готовая поддержка языков Groovy, JRuby и BeanShell. В этой главе мы обсудим применение Groovy в Spring для реализации простого процессора правил, основанного на DSL, который поддерживается Groovy.

Эта глава не предназначена служить детальным справочником по использованию языков сценариев. Для каждого языка доступны собственные книги, посвященные работе с ним. Основная цель этой главы состоит в том, чтобы подчеркнуть главную идею поддержки языков сценариев в Spring Framework и предоставить характерный пример преимущества применения какого-либо языка сценариев в дополнение к Java при разработке Spring-приложений.

Проект для примеров этой главы

В качестве начальной точки для примеров, которые будут разработаны в этой главе, мы воспользуемся простым служебным проектом Spring. Для установки подключаемого модуля Groovy в Eclipse потребуется выполнить дополнительную процедуру; необходимые шаги описаны в последующих разделах.

Создание простого служебного проекта Spring

Первый шаг заключается в создании простого служебного проекта Spring. В STS выберите пункт меню File⇨New⇨Spring Template Project (Файл⇨Создать⇨Шаблонный проект Spring) и затем вариант Simple Spring Utility Project (Простой служебный проект Spring), как показано на рис. 22.1. После этого введите информацию о проекте (рис. 22.2).

Создав проект, удостоверьтесь, что используется Spring 3.1 и JSE 6.

Установка подключаемого модуля Groovy для Eclipse

В состав Groovy входит подключаемый модуль для Eclipse, который помогает управлять разработкой проекта, используя чистые проекты Groovy или Java с классами Groovy. Однако по умолчанию этот подключаемый модуль не привязан к STS, поэтому его необходимо установить.

Чтобы установить подключаемый модуль в STS, откройте инструментальную панель Spring (щелкнув на значке Dashboard (Инструментальная панель) панели инструментов в перспективе Spring). В представлении инструментальной панели перейдите на вкладку Extensions (Расширения) и выберите элемент Groovy Eclipse. Затем щелкните на кнопке Install (Установить), как показано на рис. 22.3.

Откроется диалоговое окно Install (Установка). Выберите все элементы и щелкните на кнопке Next (Далее) для их установки (рис. 22.4).

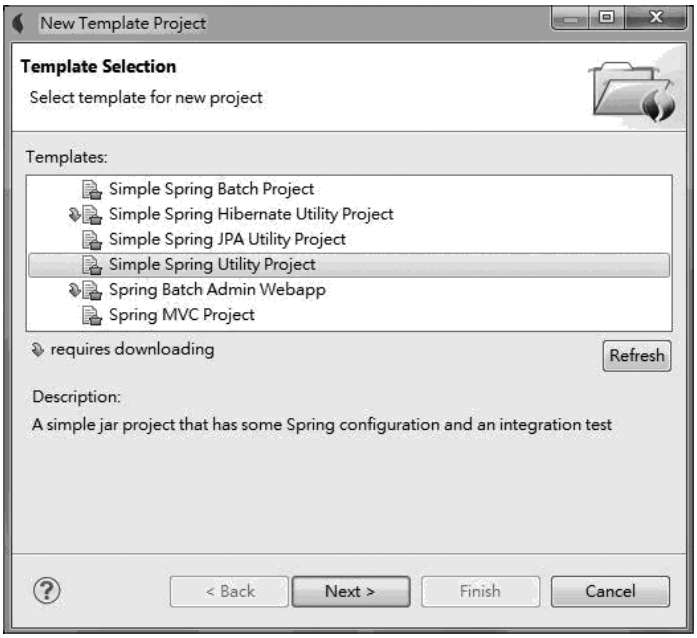


Рис. 22.1. Создание простого служебного проекта Spring в STS

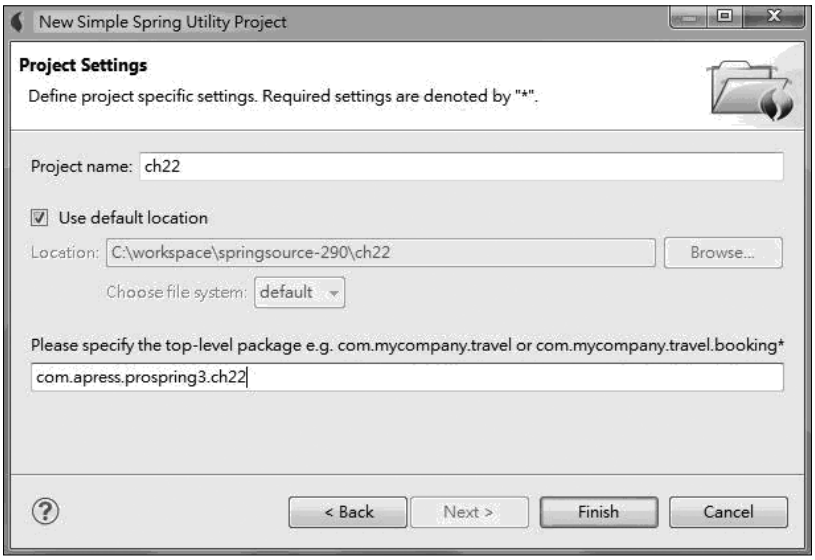


Рис. 22.2. Ввод информации о проекте

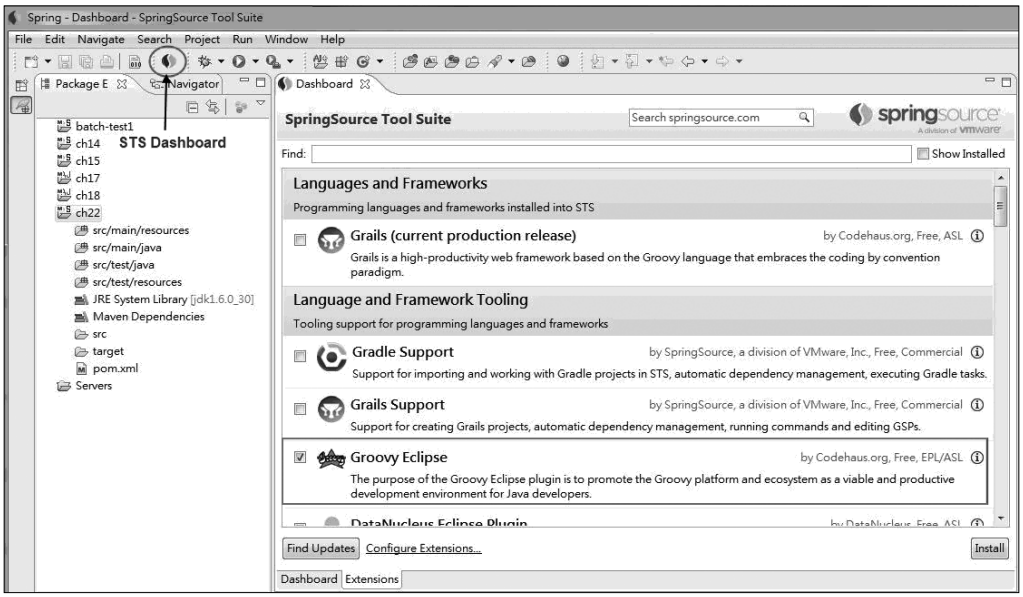


Рис. 22.3. Установка подключаемого модуля Groovy Eclipse

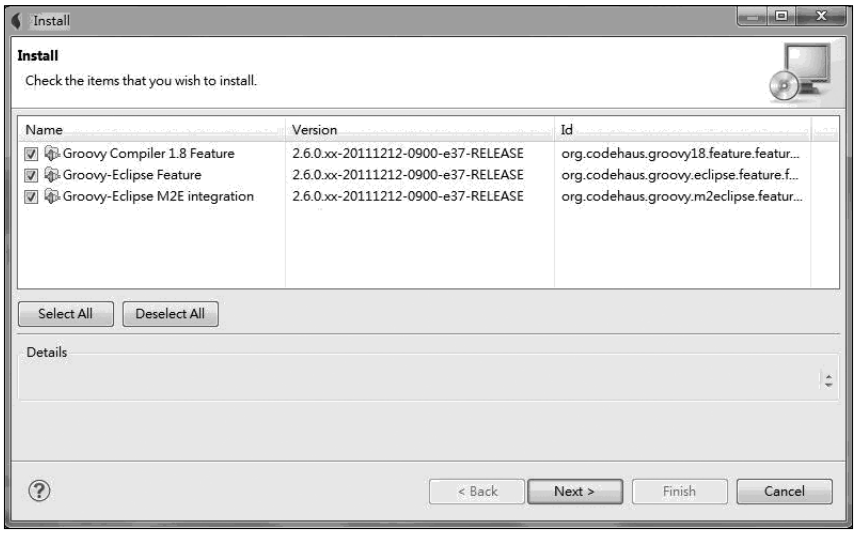


Рис. 22.4. Установка подключаемого модуля Groovy Eclipse: выбор элементов

После завершения установки и перезапуска STS щелкните правой кнопкой мыши на проекте в STS и выберите в контекстном меню пункт Configure⇨Convert to Groovy Project (Конфигурировать⇨Преобразовать в проект Groovy), чтобы сделать проект проектом Groovy, как показано на рис. 22.5.

На этом настройка проекта завершена. Тем не менее, еще нужно добавить соответствующие зависимости Maven, чтобы можно было производить сборку проекта с помощью Maven. (Позже мы сообщим, какие конкретно зависимости должны быть добавлены.)

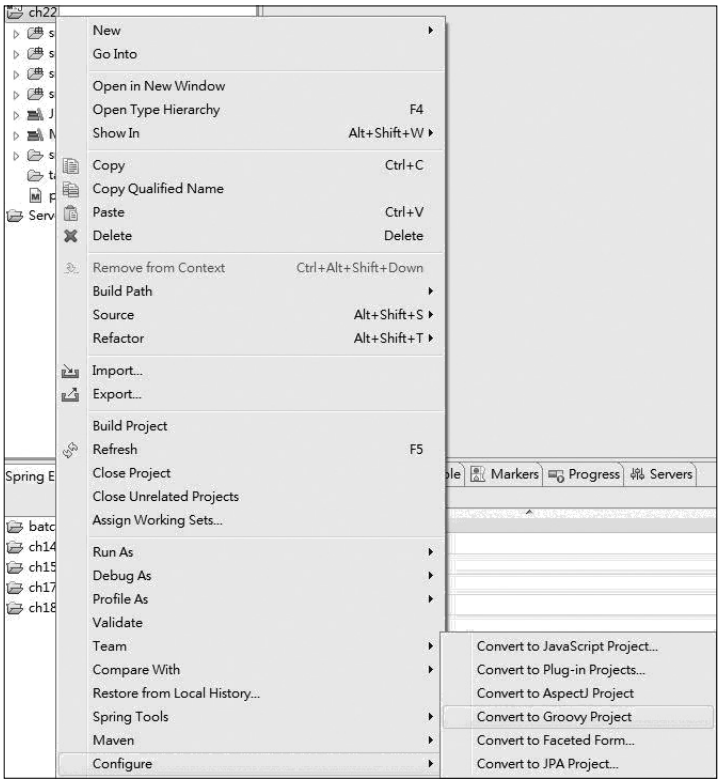


Рис. 22.5. Преобразование в проект Groovy

Поддержка написания сценариев в Java

Начиная с версии JSE 6, в JDK имеется встроенная поддержка API-интерфейса написания сценариев для платформы Java (спецификация JSR-223). Цель этого API-интерфейса заключается в том, чтобы предоставить стандартный механизм для выполнения кода, написанного на других языках сценариев, в среде JVM. В состав JDK 6 входит механизм под названием Mozilla Rhino, который способен анализировать JavaScript-программы. В этом разделе мы предоставим введение в поддержку JSR-223 внутри JDK 6.

В JDK 6 классы поддержки сценариев находятся в пакете `javax.script`. Давайте начнем с разработки простой программы для извлечения списка механизмов сценариев. Соответствующий класс показан в листинге 22.1.

Листинг 22.1. Вывод списка механизмов сценариев

```
package com.apress.prospring3.ch22.jsr223;

import javax.script.ScriptEngineFactory;
import javax.script.ScriptEngineManager;

public class ListScriptEngines {

    public static void main(String[] args) {

        ScriptEngineManager mgr = new ScriptEngineManager();
```

```
        for (ScriptEngineFactory factory : mgr.getEngineFactories()) {
            String engineName= factory.getEngineName();
            String languageName = factory.getLanguageName();
            String version = factory.getLanguageVersion();
            // Вывод имени механизма, названия языка и номера версии.
            System.out.println("Engine name: " + engineName + " Language: "
                               + languageName + " version: " + version);
        }
    }
}
```

В листинге 22.1 создается экземпляр класса `ScriptEngineManager`, который будет обнаруживать и обрабатывать список механизмов (другими словами, классов, реализующих интерфейс `javax.script.ScriptEngine`) в пути классов. Затем с помощью вызова метода `ScriptEngineManager.getEngineFactories()` извлекается список интерфейсов `ScriptEngineFactory`. Интерфейс `ScriptEngineFactory` используется для описания и создания экземпляров механизмов сценариев. Из каждого интерфейса `ScriptEngineFactory` можно извлечь информацию о поддерживаемом языке сценариев. Выполнение программы дает следующий вывод:

```
Engine name: Groovy Scripting Engine Language: Groovy version: 1.8.4
Engine name: Mozilla Rhino Language: ECMAScript version: 1.6
```

В выводе видно, что обнаружены два механизма. Первый из них — механизм Groovy, поскольку мы преобразовали проект в проект Groovy и добавили библиотеки Groovy. Второй механизм — это ECMAScript, который представляет собой JavaScript.

А теперь напишем простую программу для оценки базового выражения JavaScript. Код программы приведен в листинге 22.2.

Листинг 22.2. Оценка выражения JavaScript

```
package com.apress.prospring3.ch22.jsr223;

import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
import javax.script.ScriptException;
public class JavaScriptTest {
    public static void main(String[] args) {
        ScriptEngineManager mgr = new ScriptEngineManager();
        ScriptEngine jsEngine = mgr.getEngineByName("JavaScript");
        try {
            jsEngine.eval("print('Hello JavaScript in Java')");
        } catch (ScriptException ex) {
            ex.printStackTrace();
        }
    }
}
```

В листинге 22.2 экземпляр интерфейса `ScriptEngine` извлекается из класса `ScriptEngineManager` с использованием имени `JavaScript`. Затем вызывается метод `ScriptEngine.eval()` с передачей ему аргумента `String`, который содержит выражение JavaScript. Обратите внимание, что аргументом также может быть класс `java.io.Reader`, который может читать JavaScript-код из файла. Выполнение этой программы дает следующий вывод:

```
Hello JavaScript in Java
```

Программа должна продемонстрировать идею запуска сценариев в Java. Однако простое получение вывода, полученного из сценариев на других языках, не особенно интересно. В следующем разделе мы представим введение в Groovy — мощный и развитый язык сценариев.

Введение в Groovy

Проект Groovy был начат в 2003 г. Джеймсом Стрэченом (James Strachan). Основной целью проекта Groovy является предоставление гибкого и динамического языка для JVM, с особенностями, характерными для других популярных языков сценариев, в том числе Python, Ruby и Smalltalk. Язык Groovy построен поверх Java, расширяет Java и ликвидирует некоторые недостатки Java.

В последующих разделах мы обсудим ряд основных средств и концепций, связанных с языком Groovy, и посмотрим, как он дополняет Java с целью решения специфичных нужд в приложениях. Следует отметить, что многие упоминаемые здесь средства также доступны в других языках сценариев (например, Scala, Erlang, Python и Clojure).

Динамическая типизация

Одно из основных отличий Groovy (и многих других языков сценариев) от Java связано с поддержкой динамической типизации переменных. В Java все свойства и переменные должны быть статически типизированы. Другими словами, для них должен быть указан тип в операторе объявления. Однако Groovy поддерживает динамическую типизацию переменных. Динамически типизированные переменные в Groovy объявляются с помощью ключевого слова `def`.

Давайте посмотрим на это в действии, разработав простой сценарий Groovy. Суффиксом файла с классом или сценарием Groovy является `groovy`. Чтобы создать класс или сценарий Groovy в проекте с включенным Groovy в STS, щелкните правой кнопкой мыши на имени пакета, выберите в контекстном меню пункт New⇨Other (Создать⇨Другой) и затем укажите класс Groovy.

В листинге 22.3 показан простой сценарий Groovy с применением динамической типизации (файл `DynamicTyping.groovy`).

Листинг 22.3. Динамическая типизация в Groovy

```
package com.apress.prospring3.ch22.groovy

class Contact {
   def firstName
   def lastName
   def birthDate

   String toString() { "($firstName,$lastName,$birthDate)" }
}

Contact contact = new Contact(firstName: 'Clarence', lastName: 'Ho',
birthDate: new Date())
Contact anotherContact = new Contact(firstName: 20, lastName: 'Ho', birthDate:
new Date())

println contact
println anotherContact

println contact.firstName + 20
println anotherContact.firstName + 20
```

В листинге 22.3 приведен сценарий Groovy, который может быть запущен непосредственно в STS (щелкните на файле сценария правой кнопкой мыши и выберите в контекстном меню пункт Run As (Запустить как) и затем вариант Groovy Script (Сценарий Groovy) или Java Application (Приложение Java)). Отличия между этими вариантами заключаются в том, что сценарий Groovy может выполняться без компиляции (Groovy предоставляет инструмент командной строки под названием `groovy`, который выполняет сценарии Groovy напрямую) или же компилироваться в байт-код Java и затем запускаться подобно обычным Java-классам. Сценарии Groovy для своего выполнения не требуют метода `main()`. Кроме того, не требуется также объявлять класс с именем, совпадающим с именем файла.

В листинге 22.3 определен класс `Contact` со свойствами, динамически типизированными с помощью ключевого слова `def`. В классе объявлены три таких свойства. Затем метод `toString()` переопределен с замыканием, которое возвращает строку.

Далее конструируются два экземпляра объекта `Contact` посредством сокращенного синтаксиса, предоставляемого Groovy для определения свойств. Для первого объекта `Contact` атрибут `firstName` задается как значение `String`, в то время как для второго объекта `Contact` — в виде целого числа. Наконец, с помощью оператора `println` (его действие аналогично вызову метода `System.out.println()`) выводятся два объекта контактов. Чтобы посмотреть, как Groovy обрабатывает динамическую типизацию, предусмотрены два оператора `println` для вывода результата операции `firstName + 20`. Обратите внимание, что в Groovy при передаче аргумента методу круглые скобки не обязательны.

Выполнение этой программы дает следующий вывод:

```
(Clarence,Ho,Mon Jan 16 17:14:50 CST 2012)
(20,Ho,Mon Jan 16 17:14:50 CST 2012)
Clarence20
40
```

В этом выводе видно, что поскольку свойство определено как динамически типизированное, объект успешно конструируется при передаче значения как типа `String`, так и целочисленного. Кроме того, в двух последних операторах `println` операция сложения была корректно применена к свойству `firstName` обоих объектов. В первом случае, из-за того, что `firstName` имеет тип `String`, к значению свойства дописывается строка `20`. Во втором случае, поскольку `firstName` имеет целочисленный тип, к нему добавляется целое число `20` и результатом оказывается `40`.

Поддержка динамической типизации в Groovy обеспечивает высокую гибкость при манипулировании свойствами класса и переменными в коде приложения.

Упрощенный синтаксис

Язык Groovy также предоставляет упрощенный синтаксис, поэтому одна и та же логика при реализации на Groovy будет занимать меньше кода, чем в случае Java. Ниже перечислены некоторые особенности этого синтаксиса.

- Для завершения оператора использовать точку с запятой не обязательно.
- Ключевое слово `return` в методах является необязательным.
- Все методы и классы по умолчанию определены как `public`. Таким образом, применять ключевое слово `public` в объявлениях методов не обязательно.
- Внутри класса Groovy автоматически генерирует методы извлечения/установки для объявленных свойств. Это значит, что в классе Groovy нужно только объявить тип и имя свойства (например, `String firstName` или `def firstName`), а затем

получать доступ к нему в любом другом классе Groovy/Java, и методы извлечения/установки будут использоваться автоматически. Вдобавок к свойству можно обращаться без префикса get/set (например, contact.firstName = 'Clarence'). Groovy обработает это должным образом.

Язык Groovy также предоставляет упрощенный синтаксис и много удобных методов для работы с API-интерфейсом коллекций Java. В листинге 22.4 демонстрируется применение некоторых часто используемых операций Groovy для манипулирования списками (файл CollectionSample.groovy).

Листинг 22.4. Манипулирование списками в Groovy

```
package com.apress.prospring3.ch22.groovy

// Определить список как ArrayList.
def list = ['This', 'is', 'Clarence']
assert list.size() == 3
assert list.class == ArrayList

// Обратить список.
assert list.reverse() == ['Clarence', 'is', 'This']

// Сортировать список по размерам строк.
assert list.sort{ it.size() } == ['is', 'This', 'Clarence']

// Поучить подсписок.
assert list[0..1] == ['is', 'This']

// Использовать << для добавления элемента.
assert list << 'Но' == ['is', 'This', 'Clarence', 'Но']
```

В листинге представлена только очень малая часть возможностей, предлагаемых Groovy. За детальной информацией по этому поводу обращайтесь в онлайновую документацию по адресу http://groovy.codehaus.org/JN1015-Collections.

Замыкание

Одним из наиболее важных средств, которые Groovy добавляет к Java, является поддержка замыканий. Замыкание позволяет поместить порцию кода в объект и свободно передавать в рамках приложения. Замыкание — это очень мощное средство, которое делает возможным интеллектуальное и динамическое поведение. Добавление поддержки замыканий к языку Java запрашивалось на протяжении длительного времени. Спецификация JSR-335 (Lambda Expressions for the Java Programming Language — лямбда-выражения для языка программирования Java), которая направлена на поддержку программирования в многоядерной среде за счет добавления замыканий и связанных возможностей в язык Java, запланирована на включение в JSE 8. А пока для добавления замыканий в приложение необходимо использовать языки сценариев вроде Groovy, Python, Scala, Ruby, Clojure и т.п.

В листинге 22.5 приведен очень простой пример применения замыканий (файл SimpleClosure.groovy) в Groovy.

Листинг 22.5. Простой пример применения замыканий

```
package com.apress.prospring3.ch22.groovy.closure

def names = ['Clarence', 'Johnny', 'Mary']

names.each {println 'Hello: ' + it}
```

В листинге 22.5 объявлен список. Затем с помощью удобного метода `each()` выполняется операция над каждым элементом в списке. Аргументом метода `each()` является замыкание, которое в Groovy заключается в фигурные скобки. В результате логика из замыкания будет применена к каждому элементу списка. Внутри замыкания `it` — это специальная переменная, используемая Groovy для представления элемента, находящегося в контексте. Таким образом, замыкание обеспечит добавление к каждому элементу списка префикса в виде строки `"Hello: "` и его вывод. Выполнение сценария даст следующий вывод:

```
Hello: Clarence
Hello: Johnny
Hello: Mary
```

Как уже упоминалось, замыкание может быть объявлено как переменная, после чего использоваться, когда это необходимо. В листинге 22.6 показан еще один пример (файл `ClosureOnMap.groovy`).

Листинг 22.6. Определение замыкания как переменной

```
package com.apress.prospring3.ch22.groovy.closure
def map = ['a': 10, 'b': 50]
Closure square = {key, value -> map[key] = value * value}
map.each square
println map
```

В листинге 22.6 определена карта (`map`). Затем объявляется переменная типа `Closure`. Замыкание принимает в качестве аргументов ключ и значение элемента карты, а логика вычисляет квадрат значения, связанного с ключом. Выполнение программы дает следующий вывод:

```
[a:100, b:2500]
```

Мы предоставили лишь простое введение в замыкания. В следующем разделе мы разработаем простой процессор правил (rule engine), используя Groovy и Spring. При этом также будут применяться замыкания. За более подробными сведениями об использовании замыканий в Groovy обращайтесь в онлайновую документацию по адресу `http://groovy.codehaus.org/JN2515-Closures`.

Использование Groovy в Spring

Основное преимущество, которое Groovy и другие языки сценариев привносят в Java-приложения — это поддержка динамического поведения. За счет использования замыкания бизнес-логику можно упаковать внутрь объекта и передавать в рамках приложения подобно любой другой переменной.

Еще одной важной возможностью Groovy является поддержка разработки языков DSL с применением упрощенного синтаксиса и замыканий. Как уже упоминалось, DSL — это язык, нацеленный на конкретную предметную область с очень специфическими целями в проектировании и реализации. Задача состоит в то, чтобы построить язык, который будет понятен не только разработчикам, но также бизнес-аналитикам и пользователям. Большую часть времени предметной областью будет выступать сфера бизнеса. Например, языки DSL могут быть определены для классификации заказчиков, калькуляции налога с продаж, расчета заработной платы и т.д.

В этом разделе мы продемонстрируем использование Groovy для реализации простого процессора правил с помощью поддержки DSL в Groovy. Реализация повторяет

пример из великолепной статьи по этой теме, доступной по адресу www.pleus.net/articles/grules/grules.pdf, но с рядом изменений. Кроме того, мы покажем, как поддержка обновляемых бинов в Spring делает возможным обновление применяемых правил на лету, не требуя компиляции, упаковки и развертывания приложения.

В этом примере мы реализуем правило, используемое для классификации специфического контакта по различным категориям, основанным на возрасте, который вычисляется с применением свойства даты рождения.

Добавление обязательных зависимостей

В рассматриваемом примере используется язык Groovy библиотека Joda-Time, поэтому нам понадобится добавить в проект обязательные зависимости, описанные в табл. 22.1.

Таблица 22.1. Зависимости Maven для Groovy и Joda-Time

Идентификатор группы	Идентификатор артефакта	Версия	Описание
org.codehaus.groovy	groovy	1.8.5	Библиотека Groovy
joda-time	joda-time	2.0	Библиотека для работы с типом даты-времени Joda-Time

Предметная область, связанная с контактами

Как уже упоминалось ранее, язык DSL нацелен на специфическую предметную область, которая в большинстве случаев имеет отношение к некоторому виду бизнес-данных. Правило, которое мы собираемся реализовать, предназначено для применения к предметной области, связанной с информацией о контактах.

Таким образом, первый шаг связан с разработкой объектной модели предметной области, к которой должно применяться правило. В этом примере DOM-модель очень проста и содержит только один сущностный класс Contact, показанный в листинге 22.7. Обратите внимание, что это класс POJO, похожий на разработанные в предшествующих главах.

Листинг 22.7. Сущностный класс Contact

```
package com.apress.prospring3.ch22.domain;

import org.joda.time.DateTime;

public class Contact {
   private Long id;
   private String firstName;
   private String lastName;
   private DateTime birthDate;
   private String ageCategory;

   // Методы извлечения/установки не показаны.

   public String toString() {
      return "Contact - Id: " + id + ", First name: " + firstName
            + ", Last name: " + lastName + ", Birthday: " + birthDate
            + ", Age category: " + ageCategory;
   }
}
```

Класс Contact в листинге 22.7 — это простая информация о контакте. Для свойства ageCategory мы хотим разработать динамическое правило, которое может использоваться для проведения классификации. Это правило будет вычислять возраст на основе свойства birthDate и затем устанавливать свойство ageCategory в зависимости от результата (например, kid, youth, adult и т.п.).

Реализация процессора правил

Следующий шаг состоит в разработке простого процессора правил, предназначенного для применения правил к объекту предметной области. Для начала необходимо определить, какую информацию должно содержать правило. В листинге 22.8 показан класс Rule, который представляет собой класс Groovy (файл Rule.groovy).

Листинг 22.8. Класс Rule

```
package com.apress.prospring3.ch22.rule.domain

class Rule {

   private boolean singlehit = true
   private conditions = new ArrayList()
   private actions = new ArrayList()
   private parameters = new ArrayList()

}
```

В листинге 22.8 каждое правило имеет несколько свойств.

- Свойство conditions определяет различные условия, которые процессор правил должен проверять для обрабатываемого объекта предметной области.
- Свойство actions определяет действия, которые должны предприниматься в случае удовлетворения условия.
- Свойство parameters определяет поведение правила, которое является результатом действия для разных условий.
- Наконец, свойство singlehit определяет, должно ли правило немедленно завершать свое выполнение всякий раз, когда условие удовлетворяется.

На следующем шаге строится процессор для выполнения правил. В листинге 22.9 показан интерфейс RuleEngine (обратите внимание, что это Java-интерфейс).

Листинг 22.9. Интерфейс RuleEngine

```
package com.apress.prospring3.ch22.rule.engine;

import com.apress.prospring3.ch22.rule.domain.Rule;

public interface RuleEngine {

   public void run(Rule rule, Object object);

}
```

В этом интерфейсе определен единственный метод run(), который предназначен для применения правила к аргументу — объекту предметной области.

Мы предоставим реализацию процессора правил на языке Groovy. В листинге 22.10 приведен класс Groovy по имени RuleEngineImpl (файл RuleEngineImpl.groovy).

Листинг 22.10. Groovy-класс `RuleEngineImpl`

```
package com.apress.prospring3.ch22.rule.engine.impl

import org.springframework.stereotype.Component

import com.apress.prospring3.ch22.rule.domain.Rule
import com.apress.prospring3.ch22.rule.engine.RuleEngine

@Component("ruleEngine")
class RuleEngineImpl implements RuleEngine {

   public void run(Rule rule, Object object) {
      println "Executing rule"
      def exit=false // флаг завершения для режима singlehit
      // Проход по набору параметров.
      rule.parameters.each{ArrayList params ->
         def paramIndex=0 // указывает на текущий параметр
         def success=true
         if(!exit){
            // Проверить все условия.
            rule.conditions.each{
               println "Condition Param index: " + paramIndex
               success = success && it(object,params[paramIndex])
               println "Condition success: " + success
               paramIndex++
            }
            // Если все условия удовлетворены, выполнить действия.
            if(success && !exit){
               rule.actions.each{
                  println "Action Param index: " + paramIndex
                  it(object,params[paramIndex])
                  paramIndex++
               }
               // Если режим singlehit, завершить
               // после первого удовлетворения условия.
               if (rule.singlehit){
                  exit=true
               }
            }
         }
      }
   }
}
```

В листинге 22.10 класс `RuleEngineImpl` реализует Java-интерфейс `RuleEngine`, и к нему применяется аннотация Spring, как к любому другому объекту POJO. Внутри метода `run()` параметры, определенные в правиле, по очереди передаются замыканию на обработку. Для каждого параметра (который является списком значений) проверяется удовлетворение условий (каждое условие представляет собой замыкание) друг за другом; в проверке участвует соответствующий элемент в списке параметра и объект предметной области. Индикатор `success` становится равным `true`, только если были удовлетворены все условия. В этом случае действия (каждое действие — это также замыкание), определенные в правиле, будут применены к объекту с использованием соответствующего значения в списке параметра. Наконец, если для специфического параметра условие удовлетворено и переменная `singlehit` равна `true`, выполнение правила немедленно завершается.

Чтобы обеспечить более гибкий способ извлечения правила, определим интерфейс RuleFactory, как показано в листинге 22.11. Обратите внимание, что это Java-интерфейс.

Листинг 22.11. Интерфейс RuleFactory

```
package com.apress.prospring3.ch22.rule.factory;

import com.apress.prospring3.ch22.rule.domain.Rule;

public interface RuleFactory {

   public Rule getAgeCategoryRule();

}
```

Поскольку имеется только одно правило для классификации контактов по возрастным категориям, в интерфейсе RuleFactory определен единственный метод для извлечения правила.

Чтобы сделать процессор правил прозрачным для потребителя, разработаем простой уровень обслуживания и поместим в него процессор. В листингах 22.12 и 22.13 представлен интерфейс ContactService и класс ContactServiceImpl. Обратите внимание, что оба они являются Java-реализациями.

Листинг 22.12. Интерфейс ContactService

```
package com.apress.prospring3.ch22.service;

import com.apress.prospring3.ch22.domain.Contact;

public interface ContactService {

   public void applyRule(Contact contact);
}
```

Листинг 22.13. Класс ContactServiceImpl

```
package com.apress.prospring3.ch22.service.impl;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.ApplicationContext;
import org.springframework.stereotype.Service;

import com.apress.prospring3.ch22.domain.Contact;
import com.apress.prospring3.ch22.rule.domain.Rule;
import com.apress.prospring3.ch22.rule.engine.RuleEngine;
import com.apress.prospring3.ch22.rule.factory.RuleFactory;
import com.apress.prospring3.ch22.service.ContactService;

@Service("contactService")
public class ContactServiceImpl implements ContactService {

   @Autowired
   ApplicationContext ctx;

   @Autowired
   private RuleFactory ruleFactory;

   @Autowired
   private RuleEngine ruleEngine;

   public void applyRule(Contact contact) {
```

```
        // Применить правило ageCategory.
        ruleFactory = ctx.getBean("ruleFactory", RuleFactory.class);
        Rule ageCategoryRule = ruleFactory.getAgeCategoryRule();
        ruleEngine.run(ageCategoryRule, contact);
    }
}
```

Как показано в листинге 22.13, требуемые бины Spring автосвязаны с классом реализации службы. В методе `applyRule()` правило получается из фабрики правил и затем применяется к объекту `Contact`. В результате свойство `ageCategory` для объекта `Contact` будет выведено на основе условий, действий и параметров, определенных в правиле.

Реализация фабрики правил как обновляемого бина Spring

Теперь мы можем реализовать фабрику правил и правило для классификации по возрастным категориям. Мы хотим иметь возможность обновлять правило на лету. Среда Spring должна отслеживать изменение правила и применять самую актуальную логику. В Spring Framework предлагается великолепная поддержка для бинов Spring, написанных на языках сценариев, которые называются *обновляемыми бинами*. Мы покажем, как сконфигурировать сценарий Groovy в виде бина Spring и указать Spring на необходимость его обновления через регулярные интервалы. Сначала давайте рассмотрим реализацию фабрики правил в Groovy. Чтобы сделать возможным динамическое обновление, мы помещаем класс во внешнюю папку. Создайте в корневой папке проекта папку под названием `resources`. В эту папку будет помещен класс `RuleFactoryImpl` (который является классом Groovy и находится в файле `RuleFactoryImpl.groovy`). Код этого класса показан в листинге 22.14.

Листинг 22.14. Класс `RuleFactoryImpl`

```
import org.joda.time.DateTime
import org.joda.time.Years;

import com.apress.prospring3.ch22.rule.domain.Rule
import com.apress.prospring3.ch22.rule.factory.RuleFactory

class RuleFactoryImpl implements RuleFactory {

    Closure age = { birthDate -> return Years.yearsBetween(birthDate,
                   new DateTime()).getYears() }

    public Rule getAgeCategoryRule() {
        // Определение правила.
        Rule rule = new Rule()

        // ****************** КОНФИГУРАЦИЯ ********************
        rule.singlehit=true

        // ****************************************************
        rule.conditions=[
        // ******************** УСЛОВИЯ ***********************
        {object, param -> age(object.birthDate) >= param},
                            {object, param -> age(object.birthDate) <= param}
        // ****************************************************
        ]

        rule.actions=[
        // ******************* ДЕЙСТВИЯ ***********************
        {object, param -> object.ageCategory = param}
```

```
        // ****************************************************
        ]

        rule.parameters=[
        // *************** НАБОРЫ ПАРАМЕТРОВ *****************
        // Минимальный возраст, максимальный возраст, категория
        [0,10,'Kid'],
        [11,20,'Youth'],
        [21,40,'Adult'],
        [41,60,'Middle-aged'],
        [61,120,'Old']
        // ****************************************************
        ]

        return rule
    }
}
```

Класс в листинге 22.14 реализует интерфейс `RuleFactory`, а метод `getAgeCategoryRule()` реализован для предоставления правила. Внутри правила определено замыкание (`Closure`) по имени `age`, предназначенное для вычисления возраста на основе свойства `birthDate` (типа `DateTime` из Joda-Time) объекта `Contact`.

В рамках правила определены два условия. Первое условие проверяет, больше или равен возраст контакта предоставленному значению параметра, а второе — меньше или равен.

Затем определяется одно действие для присваивания значения, предоставлено в параметре, свойству `ageCategory` объекта `Contact`.

Параметры определяют значения для проверки условий и действия. Например, первый параметр означает, что если возраст находится между `0` и `10`, то свойству `ageCategory` объекта `Contact` будет присвоено значение `Kid`, и т.д. Таким образом, первые два значения в каждом параметре будут использоваться двумя условиями для проверки возрастного диапазона, а последнее значение будет применяться для установки свойства `ageCategory`.

Следующий шаг состоит в определении `ApplicationContext`. Содержимое конфигурационного файла `/src/main/resources/app-context.xml` приведено в листинге 22.15.

Листинг 22.15. Конфигурация Spring

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:context="http://www.springframework.org/schema/context"
   xmlns:lang="http://www.springframework.org/schema/lang"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context-3.1.xsd
      http://www.springframework.org/schema/lang
      http://www.springframework.org/schema/lang/spring-lang-3.1.xsd">

   <context:component-scan base-package="com.apress.prospring3.ch22" />

   <lang:groovy id="ruleFactory" refresh-check-delay="5000"
      script-source="file:resources/RuleFactoryImpl.groovy"/>

</beans>
```

Конфигурация довольно проста. Для определения бинов Spring на языке сценариев необходимо использовать пространство имен `lang`. Дескриптор `<lang:groovy>` предназначен для объявления бина Spring со сценарием Groovy. В атрибуте `script-source` указано местоположение сценария Groovy, который будет загружаться Spring. Для обновляемого бина должен быть предоставлен атрибут `refresh-check-delay`. В этом случае мы указываем в нем значение 5000 миллисекунд; это заставляет Spring проверять изменения в файле, если с момента последнего вызова прошло более 5 секунд. Обратите внимание, что Spring не будет проверять файл каждые 5 секунд. Вместо этого проверка будет производиться только при вызове соответствующего бина.

Тестирование правила возрастной категории

Теперь все готово для тестирования созданного правила. Тестовая программа показана в листинге 22.16 и представляет собой Java-класс.

Листинг 22.16. Тестирование процессора правил

```
package com.apress.prospring3.ch22;

import org.joda.time.format.DateTimeFormat;
import org.springframework.context.support.GenericXmlApplicationContext;

import com.apress.prospring3.ch22.domain.Contact;
import com.apress.prospring3.ch22.service.ContactService;

public class RuleEngineTest {

   public static void main(String[] args) {

      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:app-context.xml");
      ctx.refresh();

      ContactService contactService =
         ctx.getBean("contactService", ContactService.class);

      // Сконструировать объект Contact.
      Contact contact = new Contact();
      contact.setId(1l);
      contact.setFirstName("Clarence");
      contact.setLastName("Ho");
      contact.setBirthDate(
        DateTimeFormat.forPattern("yyyy-MM-dd").parseDateTime("1980-08-09"));

      // Применить правило к объекту Contact.
      contactService.applyRule(contact);
      System.out.println("Contact: " + contact);

      // Ожидать обновления правила.
      try {
         System.in.read();
      } catch (Exception ex) {
         ex.printStackTrace();
      }

      // Применить правило снова.
      contactService.applyRule(contact);
      System.out.println("Contact: " + contact);
   }
}
```

В листинге 22.16 после инициализации GenericXmlApplicationContext из Spring конструируется экземпляр объекта Contact. Затем получается экземпляр интерфейса ContactService для применения правила к объекту Contact и результат выводится на консоль. Программа остановится в ожидании пользовательского ввода, после чего правило будет применено во второй раз. В течение этой паузы можно модифицировать класс RuleFactoryImpl.groovy, чтобы среда Spring обновила бин и мы увидели измененное правило в действии.

Запуск тестовой программы дает следующий вывод:

```
Executing rule
Condition Param index: 0
Condition success: true
Condition Param index: 1
Condition success: false
Condition Param index: 0
Condition success: true
Condition Param index: 1
Condition success: false
Condition Param index: 0
Condition success: true
Condition Param index: 1
Condition success: true
Action Param index: 2
Contact: Contact - Id: 1, First name: Clarence, Last name: Ho, Birthday:
1980-08-09T00:00:00.000+08:00, Age category: Adult
```

Поскольку возраст контакта составляет 31, мы видим, что в правиле найдено совпадение для третьего параметра (т.е. [21,40,'Adult']). Это приводит к тому, что свойство ageCategory устанавливается в Adult.

Теперь, когда программа приостановлена, давайте изменим параметры в классе RuleFactoryImpl.groovy. В листинге 22.17 приведен соответствующий фрагмент кода.

Листинг 22.17. Модификация параметров правила

```
rule.parameters=[
// *************** НАБОРЫ ПАРАМЕТРОВ *****************
// Минимальный возраст, максимальный возраст, категория
 [0,10,'Kid'],
[11,20,'Youth'],
[21,30,'Adult'],
[31,60,'Middle-aged'],
[61,120,'Old']
// ****************************************************
]
```

Изменения в листинге 22.17 выделены полужирным. Внесите эти изменения и сохраните файл. Затем нажмите клавишу <Enter> в консоли, чтобы инициировать второе применение правила к тому же самому объекту. На консоли появится следующий вывод:

```
Executing rule
Condition Param index: 0
Condition success: true
Condition Param index: 1
Condition success: false
Condition Param index: 0
```

```
Condition success: true
Condition Param index: 1
Condition success: false
Condition Param index: 0
Condition success: true
Condition Param index: 1
Condition success: false
Condition Param index: 0
Condition success: true
Condition Param index: 1
Condition success: true
Action Param index: 2
Contact: Contact - Id: 1, First name: Clarence, Last name: Ho, Birthday:
1980-08-09T00:00:00.000+08:00, Age category: Middle-aged
```

Как видите, выполнение правила остановилось на четвертом параметре (т.е. `[31,60,'Middle-aged']`), в результате чего свойству `ageCategory` было присвоено значение `Middle-aged`.

Если вы просмотрите статью, на которую мы ссылались при подготовке этого примера (`www.pleus.net/articles/grules/grules.pdf`), то найдете в ней объяснения, как вынести параметр правила во внешний файл Excel, чтобы пользователи могли самостоятельно формировать и обновлять файл параметров.

Разумеется, рассмотренное выше правило было простым, однако на его примере вы должны были получить представление о том, каким образом язык сценариев вроде Groovy помогает дополнять приложения Java EE на основе Spring средствами из специфических областей, таких как процессор правил с собственным DSL.

Может возникнуть вопрос: реально ли сохранить правило в базе данных и поручить средству обновляемых бинов Spring обнаружение изменений, внесенных в базу данных? В этом случае можно было бы еще больше упростить обслуживание правила за счет предложения пользователям интерфейсной (или административной) части, позволяющей обновлять правило в базе данных на лету, а не загружать внешний файл. В действительности существует JIRA-проблема в Spring Framework, в которой это обсуждается (`https://jira.springsource.org/browse/SPR-5106`). Будьте аккуратны с описанным подходом. В то же время, предоставление пользовательского интерфейса для загрузки класса правила также является приемлемым решением. Конечно, в этом случае необходимо проявлять особую осторожность, и вводимое пользователем правило должно быть тщательно протестировано, прежде чем загружать его в производственную среду.

Резюме

В этой главе была раскрыта тема применения языков сценариев в Java-приложениях и продемонстрирована поддержка платформой Spring Framework языков сценариев, которые помогают добавлять к приложению динамическое поведение.

Первым делом, мы обсудили спецификацию JSR-223, которая была встроена в JSE 6 и поддерживает выполнение сценариев на JavaScript. Затем мы представили Groovy — популярный в сообществе Java-разработчиков язык сценариев. Мы также описали некоторые его основные возможности в сравнении с традиционным языком Java.

Наконец, мы рассмотрели поддержку языков сценариев в Spring Framework, продемонстрировав ее в действии за счет проектирования и реализации очень простого процессора правил, использующего поддержку DSL в Groovy. Мы также показали, как модифицировать правило и заставить Spring Framework автоматически обнаруживать изменения с применением средства обновляемых бинов, не требуя компиляции, упаковки и развертывания приложения.

ГЛАВА 23

Мониторинг приложений Spring

Типичное JEE-приложение содержит несколько уровней и компонентов, таких как уровень презентаций, уровень обслуживания, уровень постоянства, источник данных серверной части и т.п. Во время стадии разработки либо после развертывания приложения в среде контроля качества (quality assurance — QA) или в производственной среде нам нужно удостовериться, что приложение находится в работоспособном состоянии и не содержит никаких потенциальных ошибок или узких мест.

В Java-приложении существует много аспектов, которые могут вызвать проблемы с производительностью или перегрузкой серверных ресурсов (таких как центральный процессор, память, ввод-вывод и т.д.). В качестве примеров следует упомянуть неэффективный Java-код, утечку памяти (когда в Java-коде выделяются новые объекты, но ссылки на них не освобождаются, тем самым препятствуя JVM-машине в освобождении памяти во время процесса сборки мусора), параметры JVM, параметры пула потоков, конфигурации источников данных (например, число разрешенных одновременных подключений к базе данных), настройка базы данных, наличие длительно выполняющихся SQL-запросов и т.п.

Следовательно, существует потребность в понимании поведения приложения во время выполнения и идентификации любых потенциально узких мест или проблем. В мире Java доступно множество инструментов, которые помогают проводить мониторинг поведения JEE-приложений во время выполнения. Большинство из них построены на основе технологии Java Management Extensions (JMX).

В этой главе мы обсудим несколько общих приемов мониторинга JEE-приложений, основанных на Spring. В частности, в главе будут рассматриваться следующие темы.

- *Поддержка JMX в Spring.* Мы обсудим комплексную поддержку JMX в Spring и покажем, как сделать бины Spring открытыми для мониторинга с помощью инструментов JMX. В качестве инструмента мониторинга приложений мы будем использовать VisualVM (`http://visualvm.java.net/index.html`).
- *Отслеживание статистики Hibernate и Spring Batch.* Компоненты Hibernate и Spring Batch также предоставляют поддерживающие классы и инфраструктуру для открытия рабочего состояния и метрик производительности с использованием JMX. Мы покажем, как включить мониторинг посредством JMX для часто используемых компонентов JEE-приложений Spring.
- *Spring Insight.* Продукт Spring Insight (`www.springsource.org/insight`) — это технология, разработанная SpringSource с целью предоставления комплексной поддержки и пользовательского интерфейса, который помогает разработчикам по-

нять и визуализировать поведение любого Spring-приложения. Этот инструмент встроен в версию сервера tc Server Developer Edition в STS. Мы покажем, как использовать Spring Insight для мониторинга JEE-приложений Spring.

Важно отметить, что эта глава не предназначена служить введением в JMX, поэтому предполагается наличие у вас базового понимания инструмента JMX. За детальными сведениями по этому поводу обращайтесь к онлайновому ресурсу Oracle по адресу:

`www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html`

Проект для примеров этой главы

В этой главе в качестве приложения для мониторинга мы будем использовать веб-приложение, разработанное в главе 17. Чтобы показать, как осуществляется мониторинг заданий Spring Batch, к этому приложению было добавлено пакетное задание для импорта контактной информации в базу данных из XML-файла.

Мы подготовили проект для примера приложения, которое будет использоваться для мониторинга; вы можете просто импортировать его и заняться реализацией требуемых классов. Извлеките файл `ch23-nojmx.zip` из кода примеров в рабочее пространство STS. После этого импортируйте проект в STS.

Чтобы импортировать этот проект, в среде STS выберите пункт меню File⇨Import (Файл⇨Импорт) и на экране Select (Выбор) укажите вариант Existing Projects into Workspace (Существующие проекты в рабочем пространстве) в категории General (Общие), как показано на рис. 23.1. После этого на экране Import Projects (Импорт проектов) выберите проект, извлеченный из кода примеров, и щелкните на кнопке Finish (Готово), как показано на рис. 23.2.

По завершении импорта проекта выполните в STS сборку и развертывание проекта на сервере tc Server, а затем запустите tc Server. После того, как веб-приложение успешно запустится, откройте браузер и введите URL `http://localhost:8080/ch23/contacts`, чтобы проверить работоспособность приложения. Внешний вид примера приложения показан на рис. 23.3.

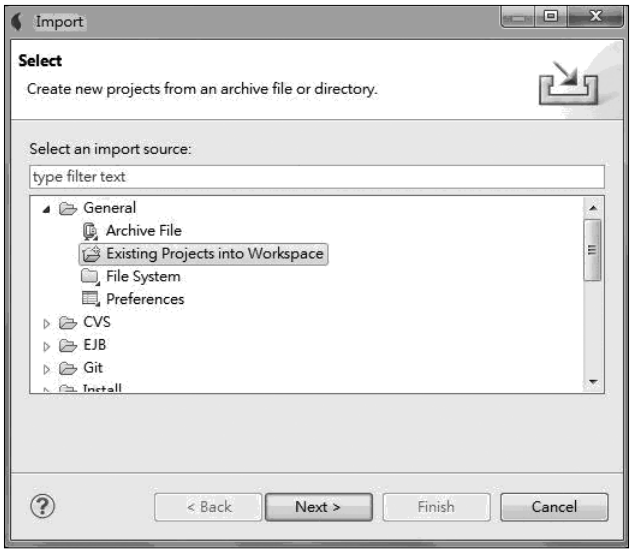


Рис. 23.1. Импорт существующего проекта в STS

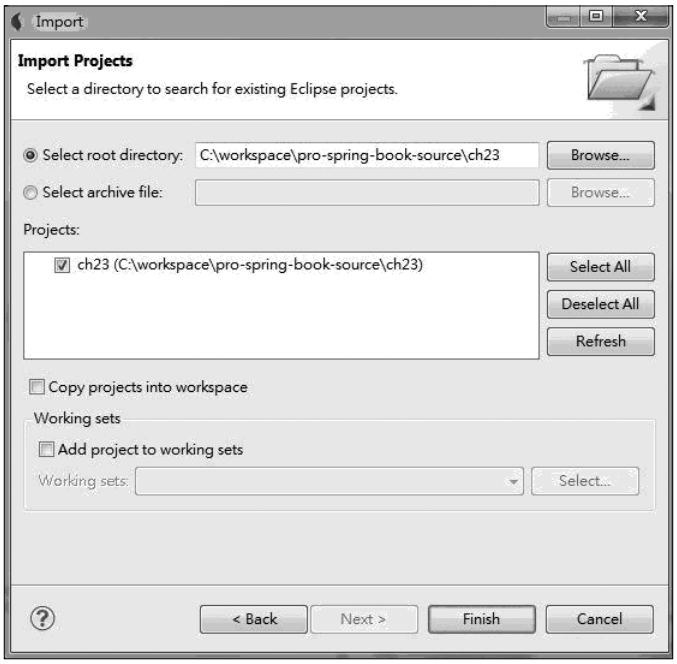


Рис. 23.2. Импорт проекта примера

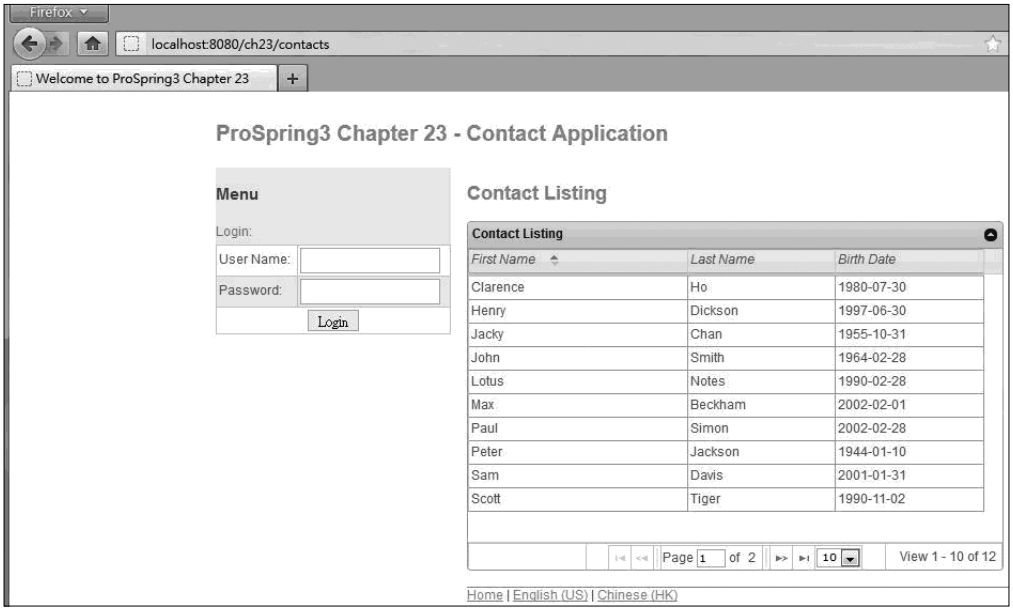


Рис. 23.3. Пример веб-приложения

Как видите, пример приложения в точности совпадает с веб-приложением, которое разрабатывалось в главе 17. Кроме того, было реализовано пакетное задание, которое опрашивает папку `C:\temp\contact\contacts.xml` на предмет появления XML-файла с контактной информацией для пакетной загрузки.

Поддержка JMX в Spring

В среде JMX классы, открытые для мониторинга и управления, называются *управляемыми бинами* (managed bean, или *MBean-компонент*). Платформа Spring Framework поддерживает несколько механизмов для открытия MBean-компонентов. Основное внимание в этой главе будет уделено открытию бинов Spring (которые разработаны как простые объекты POJO) как MBean-компонентов для мониторинга JMX.

В последующих разделах мы рассмотрим процедуру для открытия бина, содержащего связанную с приложением статистику, в качестве MBean-компонента для мониторинга JMX. Темы включают реализацию бина Spring, открытие бина Spring как MBean-компонента в `ApplicationContext` и применение инструмента VisualVM для мониторинга MBean-компонента. Вдобавок мы также покажем, как открыть для JMX некоторые метрики, касающиеся Spring Security.

Экспорт бина Spring в JMX

В примере веб-приложения, которое управляет списком контактов, мы намереваемся открыть количество контактов в базе данных в целях мониторинга JMX. Итак, давайте реализуем интерфейс и класс, как показано в листингах 23.1 и 23.2.

Листинг 23.1. Интерфейс `AppStatistics`

```
package com.apress.prospring3.ch23.jmx;

public interface AppStatistics {

   public int getTotalContactCount();
}
```

Листинг 23.2. Класс `AppStatisticsImpl`

```
package com.apress.prospring3.ch23.jmx.impl;

import org.springframework.beans.factory.annotation.Autowired;

import com.apress.prospring3.ch23.jmx.AppStatistics;
import com.apress.prospring3.ch23.service.ContactService;

public class AppStatisticsImpl implements AppStatistics {

   @Autowired
   private ContactService contactService;

   public int getTotalContactCount() {
      return contactService.findAll().size();
   }
}
```

В листинге 23.2 определен метод для извлечения общего количества записей контактов в базе данных. Чтобы открыть бин Spring для JMX, необходимо добавить конфигурацию в `ApplicationContext`. Для простоты мы будем использовать конфигурационный файл, уже существующий в примере приложения (`/src/main/resources/`

batch-context.xml), в котором хранится конфигурация для пакетных заданий. В листинге 23.3 показан фрагмент кода, который должен быть добавлен в этот файл.

Листинг 23.3. Открытие бина Spring для JMX

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"

   <!-- Остальной код не показан -->

   <!-- MBean-компонент Spring для мониторинга JMX -->
   <bean id="appStatisticsBean"
         class="com.apress.prospring3.ch23.jmx.impl.AppStatisticsImpl"/>

   <bean id="jmxExporter"
         class="org.springframework.jmx.export.MBeanExporter">
      <property name="beans">
         <map>
            <entry key="bean:name=ProSpring3ContactApp"
                   value-ref="appStatisticsBean"/>
         </map>
      </property>
   </bean>

</beans>
```

Определение бина, подлежащее добавлению, в листинге 23.3 выделено полужирным. Во-первых, объявляется бин для POJO со статистикой, которую мы хотим открыть. Во-вторых, объявляется бин jmxExporter с классом реализации MBeanExporter.

Класс MBeanExporter — это ключевой класс в рамках поддержки JMX платформой Spring Framework. Он отвечает за регистрацию бинов Spring на MBean-сервере JMX (сервер, который реализует JDK-интерфейс javax.management.MBeanServer и существует в большинстве часто используемых веб- и JEE-контейнеров, таких как Tomcat и WebSphere). При открытии бина Spring как MBean-компонента Spring будет пытаться найти на сервере выполняющийся экземпляр MBeanServer и зарегистрировать на нем этот MBean-компонент. Для сервера tc Server, который встроен в STS и основан на Tomcat, экземпляр MBeanServer будет создаваться автоматически, поэтому никакого дополнительного конфигурирования не понадобится.

Внутри бина jmxExporter свойство beans определяет бины Spring, которые необходимо открыть. Это свойство типа Map, поэтому в нем можно указать любое количество MBean-компонентов. В нашем случае нужно открыть бин appStatisticsBean, содержащий информацию о приложении контактов, которую мы хотим показать администраторам. В определении MBean-компонента ключ будет использоваться как ObjectName (класс javax.management.ObjectName в JDK) для бина Spring, на который ссылается соответствующее значение записи. В приведенной выше конфигурации MBean-компонент appStatisticsBean будет открыт с ObjectName вида bean:name=ProSpring3ContactApp. По умолчанию все public-свойства бина открываются как атрибуты, а все public-методы — как операции.

Теперь MBean-компонент доступен для мониторинга через JMX. А теперь займемся настройкой VisualVM и применением клиента JMX в целях мониторинга.

Настройка VisualVM для мониторинга JMX

VisualVM — очень полезный инструмент, помогающий проводить мониторинг разнообразных аспектов Java-приложений. Это бесплатный инструмент, встроенный в JDK 6 (файл jvisualvm.exe в папке bin внутри папки установки JDK). На веб-сайте проекта

также доступна для загрузки автономная версия VisualVM (`http://visualvm.java.net/download.html`). В этой главе мы будем пользоваться автономной версией, которой на момент написания книги была 1.3.3.

Чтобы установить VisualVM, загрузите zip-архив из веб-сайта и распакуйте его на локальной машине. После этого запустите программу `visualvm.exe` для Windows (или `visualvm` для Unix/Linux), расположенную в папке `bin`.

Для поддержки различных функций мониторинга в VisualVM используется система подключаемых модулей. Чтобы обеспечить мониторинг MBean-компонентов Java-приложений, потребуется установить подключаемый модуль VisualVM-MBeans. Для этого выберите в меню VisualVM пункт Tools⇨Plugins (Сервис⇨Подключаемые модули), в открывшемся диалоговом окне Plugins (Подключаемые модули) перейдите на вкладку Available Plugins (Доступные подключаемые модули), отметьте флажок возле подключаемого модуля VisualVM-MBeans и щелкните на кнопке Install (Установить), как показано на рис. 23.4. По завершении установки удостоверьтесь, что сервер tc Server функционирует, а пример приложения запущен. Затем в представлении Applications (Приложения) в левой части экрана VisualVM вы должны увидеть выполняющийся процесс Tomcat (рис. 23.5).

По умолчанию VisualVM будет искать Java-приложения, которые функционируют под управлением платформы JDK 6. (В JSE 6 разрешено создание клиента JMX, который использует Attach API для включения встроенного мониторинга и управления любыми приложениями, запущенными на платформе Java SE 6.) На рис. 23.5 процесс Tomcat — это сервер tc Server, выполняющийся в STS. Дважды щелкнув на его узле, вы получите экран мониторинга, показанный на рис. 23.6.

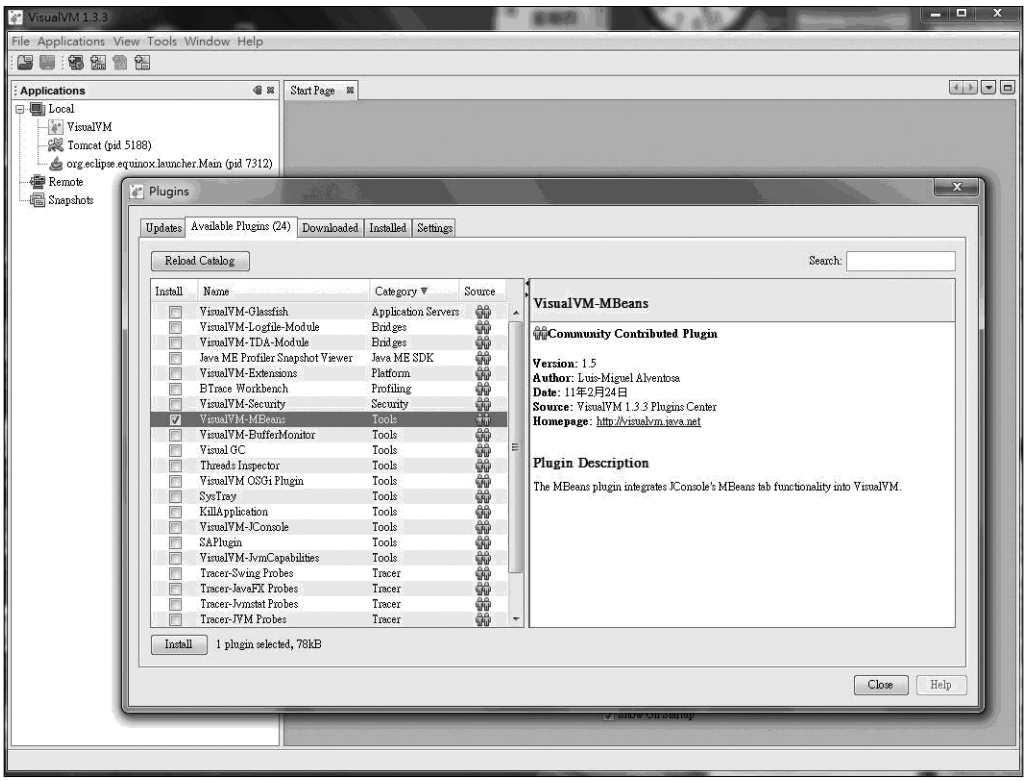


Рис. 23.4. Установка подключаемого модуля VisualVM-MBeans

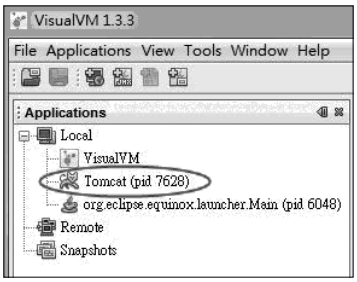


Рис. 23.5. На основном экране VisualVM показан процесс Tomcat

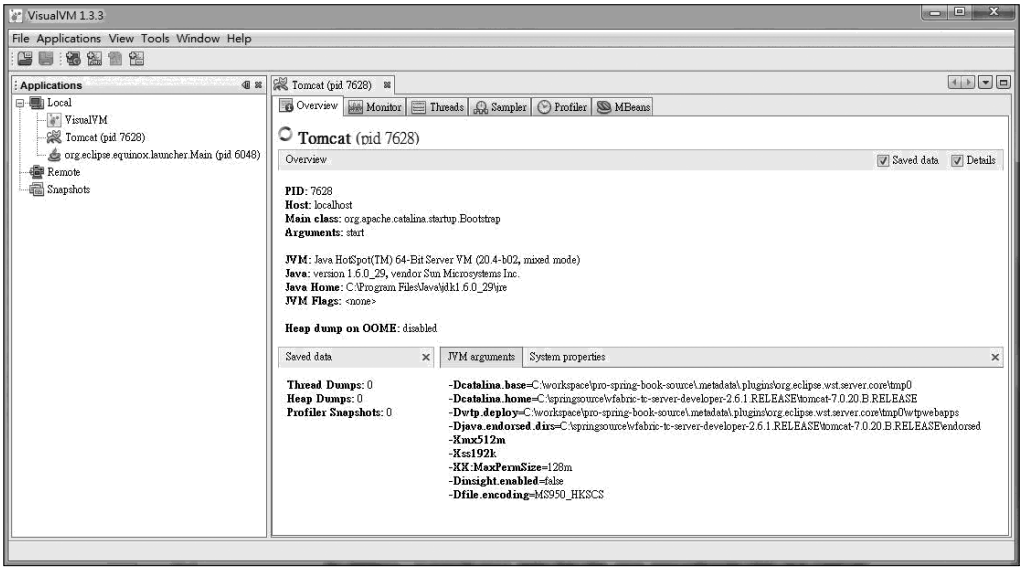


Рис. 23.6. JMX-мониторинг Tomcat в VisualVM

После установки подключаемого модуля VisualVM-MBeans появляется вкладка MBeans (MBean-компоненты). Перейдя на нее, можно увидеть доступные MBean-компоненты. Слева должен быть виден узел под названием bean (бин). Если развернуть его, станет видно MBean-компонент `ProSpring3ContactApp`, который был открыт (рис. 23.7).

В правой части показан метод, реализованный в бине, с атрибутом `TotalContactCount` (который автоматически выведен методом `getTotalContactCount()` внутри бина). Добавьте новый контакт в примере приложения и обновите представление. Значение счетчика соответствующим образом изменится.

Мониторинг вошедших пользователей

Инструмент JMX очень полезен для мониторинга состояния и метрик приложения. Один из частых сценариев использования — открытие количества пользователей, вошедших в веб-приложение. В этом разделе мы покажем, как открыть число вошедших пользователей в примере приложения для JMX.

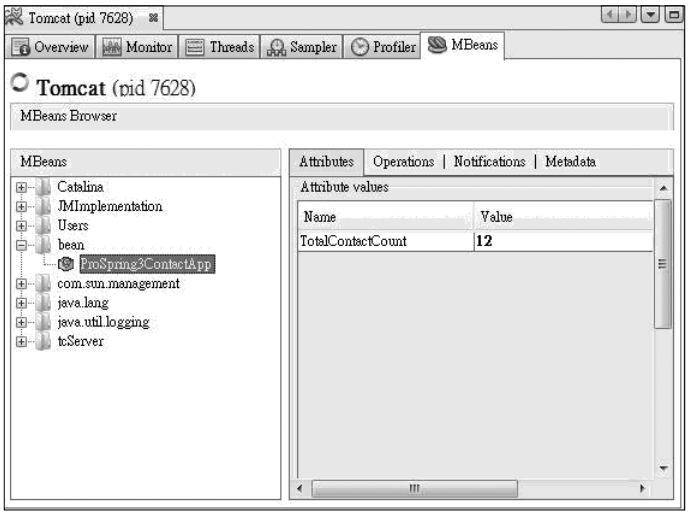


Рис. 23.7. MBean-компонент `ProSpring3ContactApp`

Как говорилось в главе 17, для управления безопасностью приложений применяется Spring Security. Инфраструктура Spring Security поддерживает конфигурацию реестра сеансов, который может быть внедрен в бин Spring, и из него можно извлечь количество вошедших пользователей. Чтобы сделать это, необходимо включить управление параллельными сеансами в Spring Security. Добавим в дескриптор веб-развертывания (файл `web.xml`) слушатель Spring Security, объявление которого показано в листинге 23.4.

Листинг 23.4. Слушатель параллельных сеансов Spring Security

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"

   <!-- Остальной код не показан -->

   <!-- Создание контейнера Spring, разделяемого всеми сервлетами и фильтрами -->
   <listener>
      <listener-class>org.springframework.web.context.ContextLoaderListener
      </listener-class>
   </listener>

   <!-- Слушатель параллельных сеансов Spring Security -->
   <listener>
      <listener-class>
         org.springframework.security.web.session.HttpSessionEventPublisher
      </listener-class>
   </listener>

   <!-- Остальной код не показан -->

</web-app>
```

Объявление слушателя в листинге 23.4 выделено полужирным. Реализация слушателя Spring Security создается для перехвата опубликованных событий, связанных с HTTP-сеансом, таких как вход пользователя и создание нового HTTP-сеанса.

После этого понадобится включить управление сеансами в конфигурации Spring Security. В листинге 23.5 представлен фрагмент кода, который должен быть добавлен в конфигурационный файл /WEB-INF/spring/security-context.xml.

Листинг 23.5. Конфигурирование управления сеансами в Spring Security

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/security"

   <!-- Остальной код не показан -->

   <http use-expressions="true">
      <intercept-url pattern='/*' access='permitAll' />
      <form-login login-page="/contacts"
                  authentication-failure-url="/security/loginfail"
         default-target-url="/contacts" />
      <logout logout-success-url="/contacts"/>
      <session-management>
         <concurrency-control max-sessions="1"
               session-registry-alias="sessionRegistry"/>
      </session-management>
   </http>

   <!-- Остальной код не показан -->

</beans:beans>
```

Дополнительная конфигурация в листинге 23.5 выделена полужирным. Внутри дескриптора <http> с помощью дескриптора <session-management> определяется конфигурация, относящаяся к управлению сеансами. Внутри него дескриптор <concurrency-control> указывает Spring Security на необходимость управления параллелизмом сеансов входа пользователей. Атрибут max-sessions задает максимальное количество одновременных сеансов, разрешенных каждому пользователю, а атрибут session-registry-alias предназначен для запроса Spring Security об открытии реестра сеансов, который управляется как бин Spring с именем sessionRegistry, для внедрения в MBean-компонент Spring позже.

Теперь можно модифицировать бин для автосвязывания реестра сеансов и открытия количества вошедших пользователей инструменту JMX. Давайте добавим два метода, чтобы отобразить информацию о вошедших пользователях. Переделанный интерфейс AppStatistics и класс AppStatisticsImpl показаны в листингах 23.6 и 23.7.

Листинг 23.6. Переделанный интерфейс AppStatistics

```
package com.apress.prospring3.ch23.jmx;

import java.util.List;

public interface AppStatistics {

   public int getTotalContactCount();

   public int getLoggedInUserCount();

   public List<Object> getLoggedInUsers();

}
```

Листинг 23.7. Переделанный класс AppStatisticsImpl

```
package com.apress.prospring3.ch23.jmx.impl;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.session.SessionRegistry;
import com.apress.prospring3.ch23.jmx.AppStatistics;
import com.apress.prospring3.ch23.service.ContactService;
public class AppStatisticsImpl implements AppStatistics {
   @Autowired
   private ContactService contactService;
   @Autowired
   private SessionRegistry sessionRegistry;
   public int getTotalContactCount() {
      return contactService.findAll().size();
   }
   public int getLoggedInUserCount() {
      return sessionRegistry.getAllPrincipals().size();
   }
   public List<Object> getLoggedInUsers() {
      return sessionRegistry.getAllPrincipals();
   }
}
```

В листингах 23.6 и 23.7 внесенные изменения выделены полужирным. В интерфейс были добавлены два метода. Метод `getLoggedInUserCount()` отображает количество вошедших пользователей, а метод `getLoggedInUsers()` возвращает список вошедших принципалов, поддерживаемый Spring Security. В классе `AppStatisticsImpl` интерфейс `SessionRegistry` из Spring Security автосвязывается, а внутри методов `getLoggedInUserCount()` и `getLoggedInUsers()` вызывается метод `SessionRegistry.getAllPrincipals()` для извлечения вошедших пользователей.

После повторной сборки и развертывания проекта и перезагрузки приложения вы увидите на экране VisualVM две новые метрики, как показано на рис. 23.8.

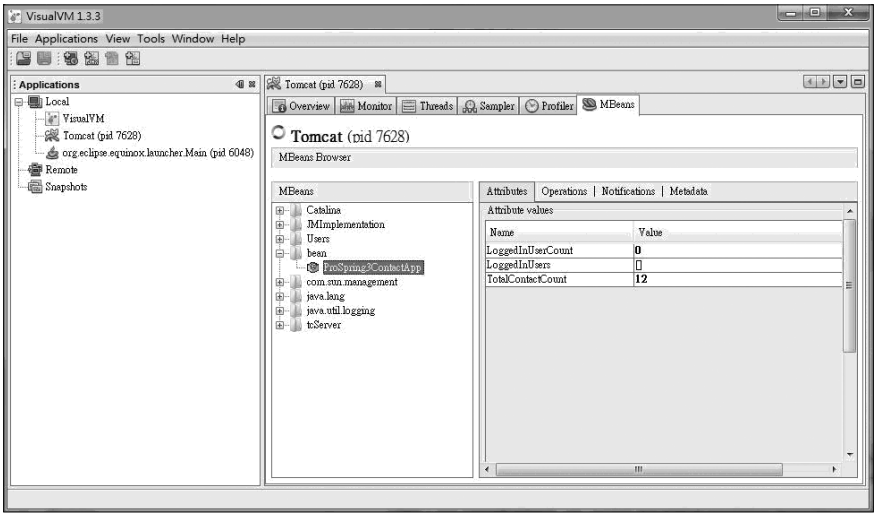


Рис. 23.8. Мониторинг вошедших пользователей

На рис. 23.8 отображены две метрики, связанные с пользователями. Метрика `LoggedInUserCount` показывает количество вошедших пользователей, в данный момент равное `0`. Метрика `LoggedInUsers` отражает список принципалов, который в текущий момент пуст.

Войдите в приложение (указав `user` для имени пользователя и пароля). Затем обновите представление в VisualVM (щелкнув на кнопке Refresh (Обновить)) и вы увидите, что количество вошедших пользователей увеличилось на единицу (рис. 23.9).

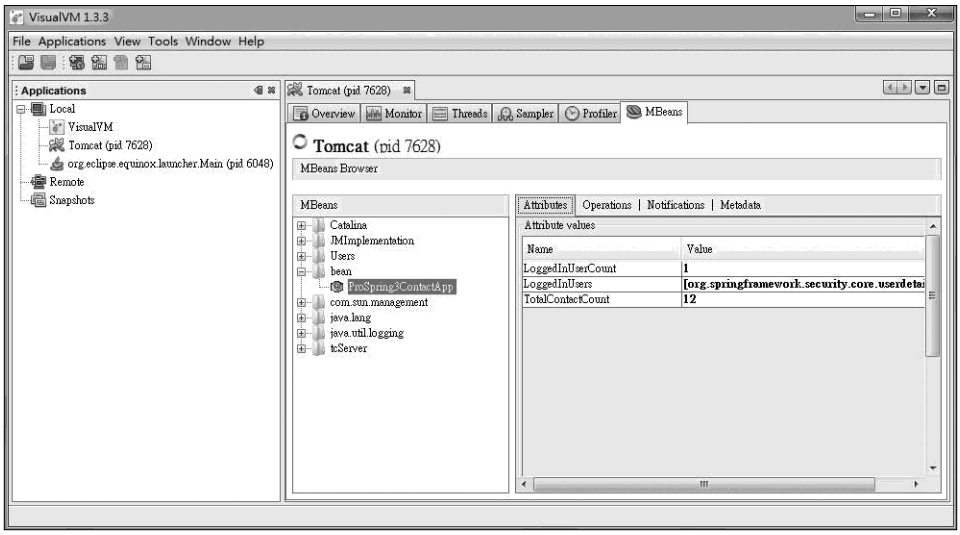


Рис. 23.9. Появление нового вошедшего пользователя

Как видите, счетчик сейчас равен `1`. Чтобы посмотреть, что за пользователь вошел в приложение, перейдите на вкладку Operations (Операции) и щелкните на операции `getLoggedInUsers`; откроется диалоговое окно с содержимым списка, который содержит имя вошедшего пользователя, как показано на рис. 23.10.

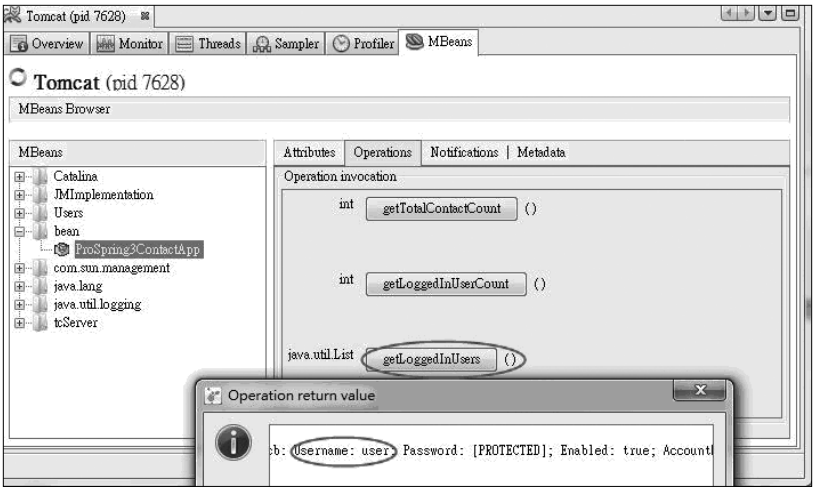


Рис. 23.10. Просмотр имени вошедшего пользователя

На рис. 23.10 отображается информация о каждом вошедшем пользователе, включая имя пользователя. Это очень удобно в ситуации, когда перед выполнением действий по обслуживанию нужно узнать, есть ли пользователи, вошедшие в приложение.

Мониторинг статистики Hibernate

Hibernate также поддерживает управление и открытие для JMX метрик, связанных с постоянством. Чтобы сделать это доступным, понадобится добавить в конфигурацию JPA (файл `/src/main/resources/datasource-tx-jpa.xml`) два дополнительных свойства Hibernate, как показано в листинге 23.8.

Листинг 23.8. Включение статистики Hibernate

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"

   <!-- Остальной код не показан -->

   <bean id="emf"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
      <property name="dataSource" ref="dataSource" />
      <property name="jpaVendorAdapter">
         <bean
class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter" />
      </property>
      <property name="packagesToScan"
               value="com.apress.prospring3.ch23.domain"/>
      <property name="jpaProperties">
         <props>
            <prop key="hibernate.dialect">org.hibernate.dialect.H2Dialect
            </prop>
            <prop key="hibernate.max_fetch_depth">3</prop>
            <prop key="hibernate.jdbc.fetch_size">50</prop>
            <prop key="hibernate.jdbc.batch_size">10</prop>
            <prop key="hibernate.show_sql">true</prop>
            <!-- Свойства статистики Hibernate -->
            <prop key="hibernate.generate_statistics">true</prop>
            <prop key="hibernate.session_factory_name">sessionFactory</prop>
         </props>
      </property>
   </bean>

   <!-- Остальной код не показан -->

</beans>
```

В листинге 23.8 новые свойства выделены полужирным. Свойство `hibernate.generate_statistics` инструктирует Hibernate о необходимости генерации статистики для его поставщика постоянства JPA, тогда как свойство `hibernate.session_factory_name` определяет имя фабрики сеансов, требуемое MBean-компонентом статистики Hibernate.

Наконец, нужно добавить этот MBean-компонент в конфигурацию `MBeanExporter` из Spring. В листинге 23.9 приведен фрагмент кода, который понадобится добавить в конфигурационный файл `src/main/resources/batch-context.xml`.

Листинг 23.9. MBean-компонент для статистики Hibernate

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"

   <!-- Остальной код не показан -->

   <!-- MBean-компонент Spring для мониторинга JMX -->
   <bean id="appStatisticsBean"
class="com.apress.prospring3.ch23.jmx.impl.AppStatisticsImpl"/>

   <bean id="jmxExporter"
         class="org.springframework.jmx.export.MBeanExporter">
      <property name="beans">
         <map>
            <entry key="bean:name=ProSpring3ContactApp"
                   value-ref="appStatisticsBean"/>
            <entry key="bean:name=ProSpring3ContactApp-hibernate"
                   value-ref="statisticsBean"/>
         </map>
      </property>
   </bean>

   <bean id="statisticsBean" class="org.hibernate.jmx.StatisticsService">
      <property name="statisticsEnabled">
         <value>true</value>
      </property>
      <property name="sessionFactoryJNDIName">
         <value>sessionFactory</value>
      </property>
   </bean>

</beans>
```

Изменения в листинге 23.9 выделены полужирным. Объявлен новый бин `statisticsBean` с классом `StatisticsService` из Hibernate в качестве реализации. С помощью этого класса Hibernate поддерживает открытие статистики для JMX. Обратите внимание на свойство `sessionFactoryJNDIName`, которое должно соответствовать свойству, определенному в листинге 23.8 (`hibernate.session_factory_name`).

Внутри бина `jmxExporter` объявлен еще один бин с `ObjectName` вида `bean:name=ProSpring3ContactApp-hibernate`, который ссылается на бин `statisticsBean`.

Теперь статистика Hibernate включена и доступна через JMX. После перезагрузки приложения и обновления VisualVM можно будет видеть MBean-компонент статистики Hibernate. Щелчок на узле `ProSpring3ContactApp-hibernate` приводит к отображению подробной статистики с правой стороны. Обратите внимание, что на полях с информацией, не относящейся к примитивным типам Java (к примеру, `List`), можно щелкать и разворачивать поле для просмотра его содержимого. На рис. 23.11 показана вкладка MBeans с развернутыми полями для отображенных сущностных классов и выполненных запросов.

На рис. 23.11 видно множество других метрик, таких как `ConnectCount`, `FlushCount`, `SecondLevelCacheHitCount` и т.д. Эти значения очень полезны для понимания поведения постоянства внутри приложения и могут помочь при поиске и устранении неполадок, а также при настройке производительности.

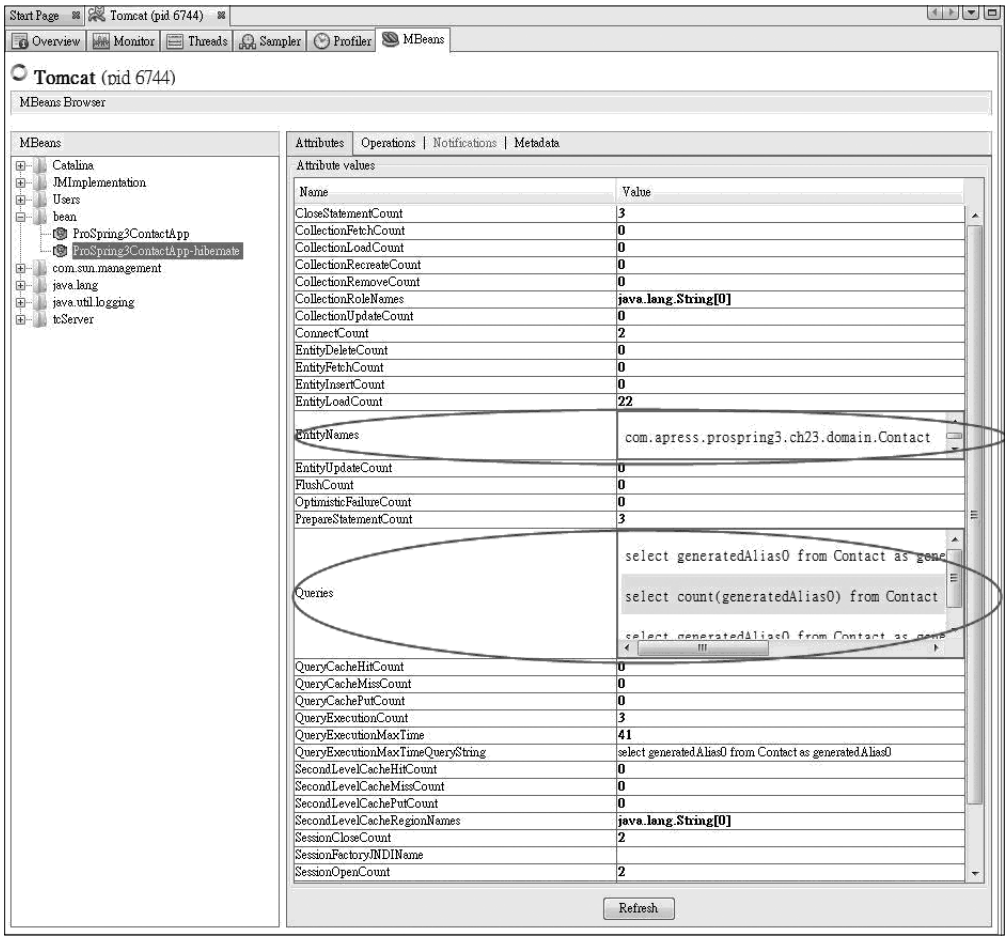


Рис. 23.11. Статистика Hibernate

Мониторинг пакетных заданий Spring

Если для запуска пакетных заданий вы применяете Spring Batch, то можете также воспользоваться поддержкой JMX в Spring Batch для мониторинга состояния пакетных заданий. Проект Spring Batch Admin является подпроектом Spring Batch и предоставляет класс и инфраструктуру, требуемую для открытия состояния выполнения задания инструменту JMX. Сначала понадобится добавить к проекту зависимость для Spring Batch Admin, которая описана в табл. 23.1.

Таблица 23.1. Зависимости Maven для Spring Batch Admin

Идентификатор группы	Идентификатор артефакта	Версия	Описание
`org.springframework.batch`	`spring-batch-admin-manager`	1.2.1.RELEASE	Модуль диспетчера Spring Batch Admin, который включает классы, требуемые для открытия состояния заданий через JMX

Следующий шаг предусматривает конфигурирование требуемых бинов для Spring Batch и Spring Batch Admin для открытия метрик через JMX. В листинге 23.10 показаны определения бинов, которые должны быть добавлены в конфигурационный файл `batch-context.xml`.

Листинг 23.10. MBean-компонент для Spring Batch

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   <!-- Остальной код не показан -->
   <!-- Spring Batch Admin для JMX -->
   <context:mbean-export default-domain="spring.application"/>
   <bean id="batchMBeanExporter"
    class="org.springframework.batch.admin.jmx.BatchMBeanExporter">
      <property name="jobService">
         <bean class="org.springframework.aop.framework.ProxyFactoryBean">
            <property name="target" ref="jobService" />
            <property name="interceptorNames" value="cacheInterceptor" />
         </bean>
      </property>
      <property name="defaultDomain" value="spring.application" />
   </bean>
   <bean id="cacheInterceptor"
    class="org.springframework.batch.admin.util.SimpleEhCacheInterceptor" />
   <bean id="jobService"
class="org.springframework.batch.admin.service.SimpleJobServiceFactoryBean">
      <property name="jobRepository" ref="jobRepository" />
      <property name="jobLauncher" ref="jobLauncher" />
      <property name="jobLocator" ref="jobRegistry" />
      <property name="dataSource" ref="dataSource" />
   </bean>
</beans>
```

Новые определения бинов в листинге 23.10 выделены полужирным. Во-первых, определен дескриптор `<context:mbean-export>`, который указывает Spring на необходимость сканирования классов с соответствующими метаданными аннотаций JMX. Это механизм, который Spring Batch Admin использует для открытия своего состояния через JMX.

Во-вторых, объявлен бин `batchMBeanExporter` с классом реализации `BatchMBeanExporter`. Этот класс расширяет класс `MBeanExporter` из Spring и ссылается на бин с классом реализации `SimpleJobServiceFactoryBean` для извлечения метрик пакетного задания.

После повторной сборки, развертывания и загрузки приложения метрики для Spring Batch будут доступны через JMX. Обновив представление в VisualVM, вы увидите узел с именем `spring.application`, который является стандартным доменом MBean-компонента, определенным в дескрипторе `<context:mbean-export>`. Развернув этот узел, вы заметите MBean-компоненты, открытые через Spring Batch Admin. Особый интерес из них представляет `JobExecution`, который показывает определенные задания и связанные с ними метрики. На рис. 23.12 изображен случай, когда приложение только что запущено, поэтому нет ни одного выполняющегося задания (в примере приложения задание имеет имя `importContactsJob`).

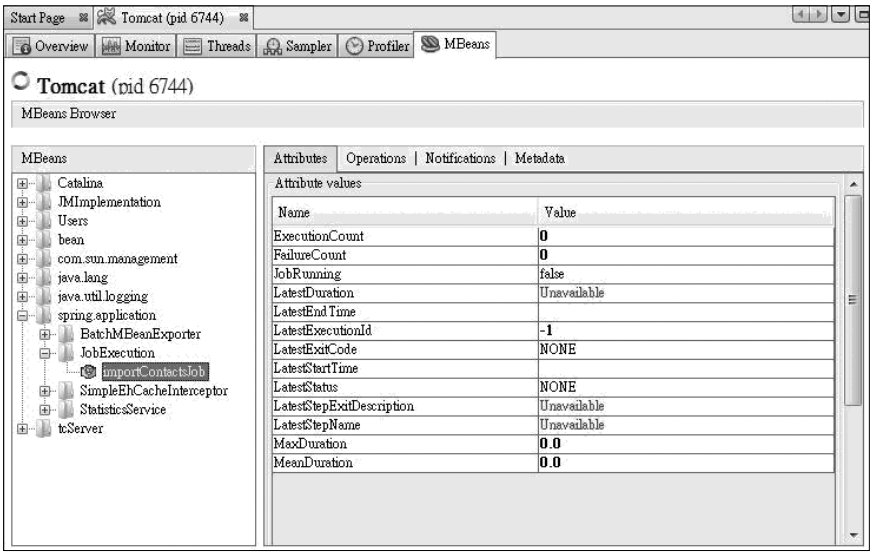


Рис. 23.12. Начальный экран Spring Batch Admin

Теперь инициируем запуск пакетного задания. Скопируйте файл `contacts.xml` из корневой папки проекта в папку `C:\temp\contact`. В выводе консоли сервера tc Server в STS вы увидите, что задание было запущено и контактная информация в файле загружена. Затем обновится экран в VisualVM, как показано на рис. 23.13.

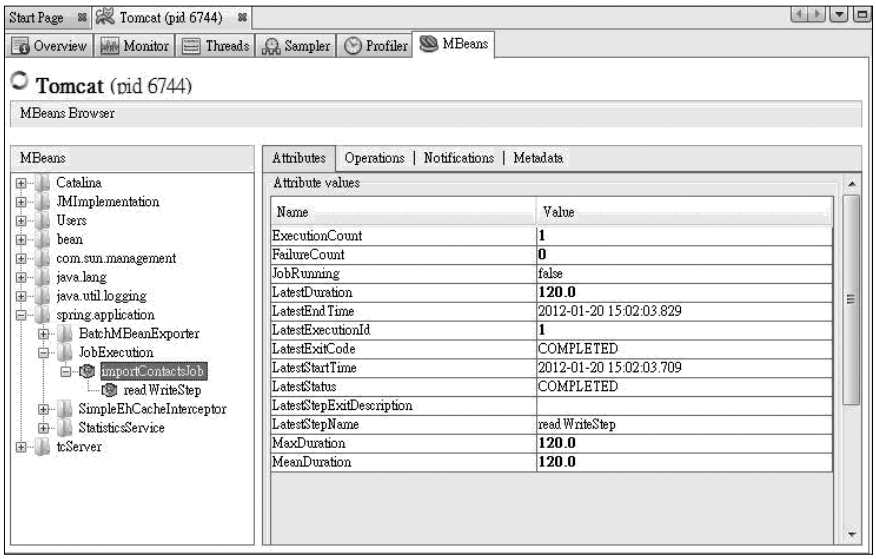


Рис. 23.13. Экран Spring Batch Admin с запущенным пакетным заданием

Здесь видно, что счетчик выполнений обновился вместе с метриками последнего выполнения задания. Вы также заметите, что появился шаг `readWriteStep` задания. Щелчок на нем приводит к отображению подробностей о метриках выполнения шага (это единственный шаг в задании), как показано на рис. 23.14.

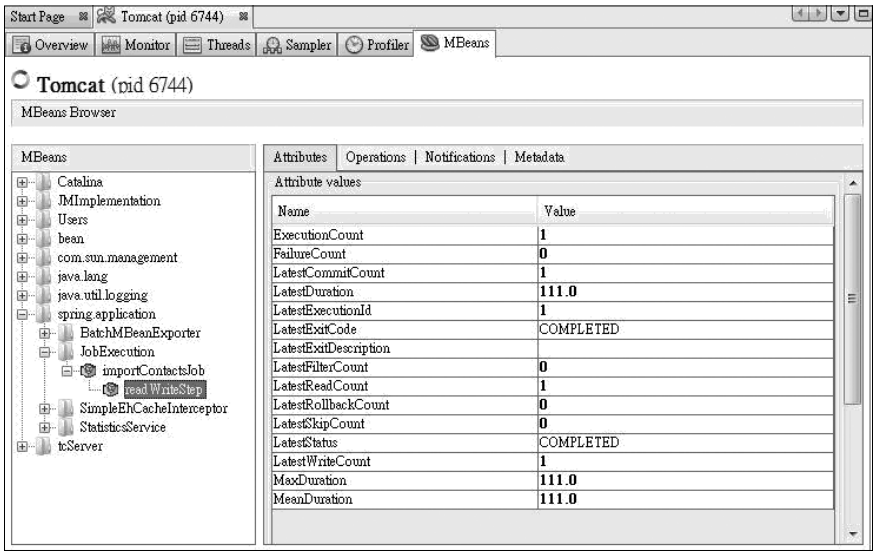


Рис. 23.14. Экран Spring Batch Admin с метриками для шага `readWriteStep`

Теперь вы имеете представление о том, каким образом поддержка JMX в Spring может помочь в мониторинге многих аспектов поведения и состояния приложения либо посредством MBean-компонентов JMX, встроенных в библиотеку третьей стороны, либо за счет реализации собственного MBean-компонента.

Мониторинг приложения с помощью Spring Insight

Инструмент Spring Insight, предлагаемый SpringSource, поддерживает комплексный мониторинг и визуализацию веб-приложений, основанных на Spring.

В этом разделе мы продемонстрируем использование Spring Insight Developer (версия Spring Insight, предназначенная для разработчиков) для мониторинга примера веб-приложения.

В последующих разделах мы обсудим требуемые процедуры, в том числе настройку экземпляра сервера tc Server с включенным инструментом Spring Insight и применение его для мониторинга основных аспектов JEE-приложения на основе Spring.

Введение в Spring Insight

Инструмент Spring Insight поддерживает несколько функциональных возможностей. Наиболее важным средством для разработчиков приложений является возможность трассировки и сбора статистических данных и показателей производительности по всем уровням веб-приложения. Ниже перечислены основные данные, которые можно получить с помощью Spring Insight.

- Время отклика различных страниц внутри веб-приложения на протяжении указанного периода времени.
- Подробное описание каждого запроса, его параметров и заголовков.
- Список вызовов методов компонентов и их параметров.
- Список всех запросов к базе данных и их производительность.
- Взаимодействие между бинами Spring и его производительность.

Внутренне Spring Insight использует AspectJ для перехвата операций в целевом веб-приложении. Целевые веб-приложения загружаются с помощью специального загрузчика классов, который динамически производит измерения для веб-приложений во время выполнения.

Конфигурирование Spring Insight

Простейший способ понять Spring Insight — посмотреть на него в действии. В этом разделе мы покажем, как настроить среду с инструментом Spring Insight Developer для мониторинга нашего примера приложения.

С помощью STS настройка Spring Insight Developer осуществляется легко. Версия сервера tc Server Developer Edition, встроенная в STS, уже содержит Spring Insight Developer. Для ее использования нужно лишь сконфигурировать экземпляр tc Server с включенным Spring Insight.

Чтобы сконфигурировать tc Server с включенным инструментом Spring Insight, на вкладке Servers (Серверы) в STS щелкните правой кнопкой мыши на пустой области и выберите в контекстном меню пункт New⇨Server (Новый⇨Сервер). Откроется диалоговое окно мастера New Server (Новый сервер). Введите имя сервера, как показано на рис. 23.15.

Щелкните на кнопке Next (Далее), после чего отобразится экран tc Server Configuration (Конфигурация tc Server) мастера создания нового сервера. Выберите переключатель Create new instance (Создать новый экземпляр) и щелкните на кнопке Next (рис. 23.16).

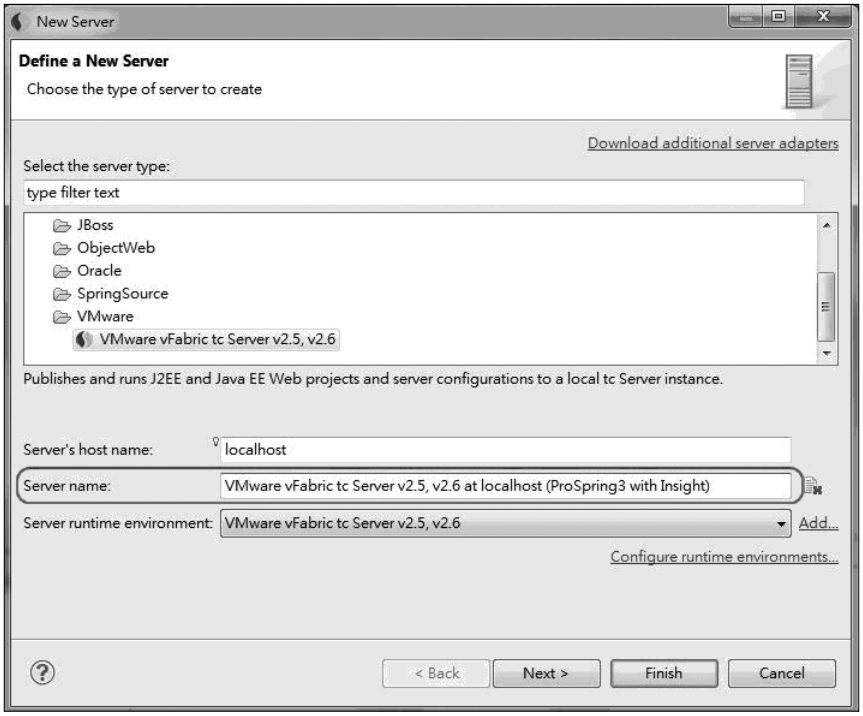


Рис. 23.15. Создание нового сервера в STS

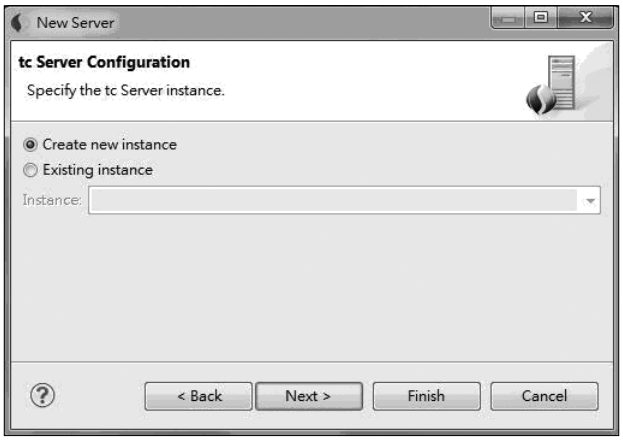


Рис. 23.16. Экран tc Server Configuration

На открывшемся экране Create tc Server Instance (Создание экземпляра tc Server) мастера создания нового сервера введите имя для экземпляра tc Server (например, `prospring3-ch23-insight`) и отметьте флажок insight в списке Templates (Шаблоны). Это приведет к созданию экземпляра tc Server с применением шаблона Spring Insight (рис. 23.17).

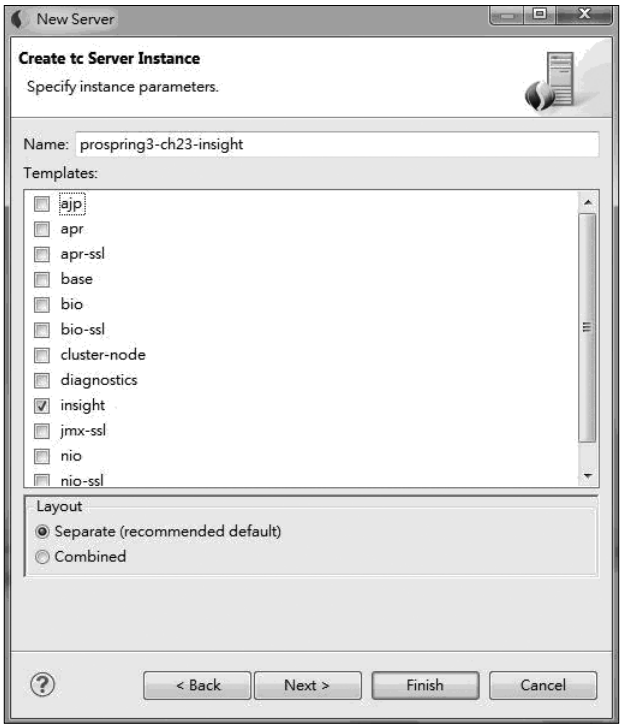


Рис. 23.17. Создание экземпляра tc Server с применением шаблона Spring Insight

Щелкните на кнопке Next, чтобы перейти к добавлению проекта на сервер (рис. 23.18).

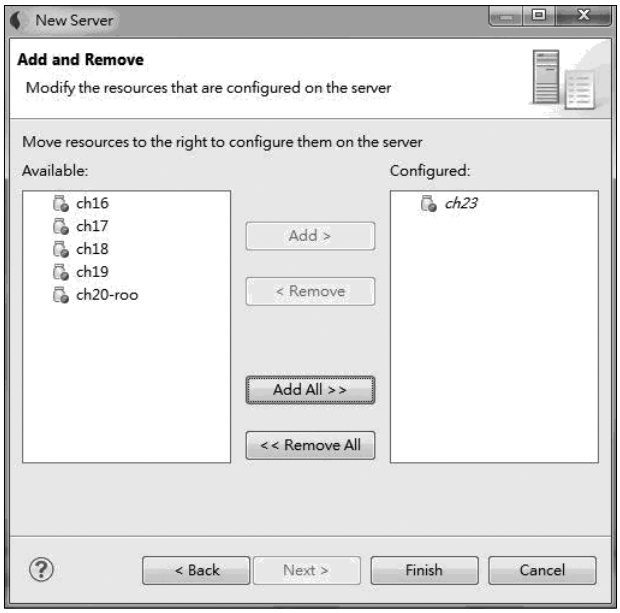


Рис. 23.18. Добавление проекта к экземпляру сервера tc Server с включенным Spring Insight

Щелкните на кнопке Finish (Готово) и подождите, пока создание экземпляра сервера завершится. Затем вы увидите новый сервер с добавленным проектом, как показано на рис. 23.19.

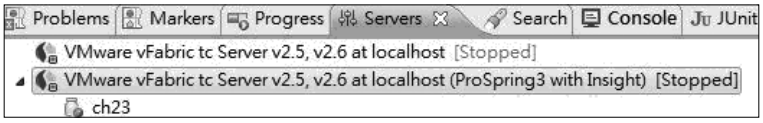


Рис. 23.19. Новый экземпляр tc Server на вкладке Servers в STS

Теперь опубликуйте проект и запустите сервер. После того, как он запустится, мы можем приступать к исследованию поведения веб-приложения с помощью инструмента Spring Insight.

Использование Spring Insight

В состав Spring Insight входит веб-приложение для визуализации показателей производительности приложений, развернутых на данном экземпляре сервера. Для доступа к Spring Insight на сервере tc Server введите в своем браузере URL `http://localhost:8080/insight`. Первая страница этого инструмента должна выглядеть, как показано на рис. 23.20.

На рис. 23.20 видно, что одним из приложений является пример приложения, разработанный в этой главе; ему соответствует URL `/ch23`. Поработайте с этим приложением (на другой вкладке браузера или в другом окне) и затем щелкните на элементе `/ch23`. Отобразятся конечные точки с высокоуровневыми показателями пропускной способности (рис. 23.21).

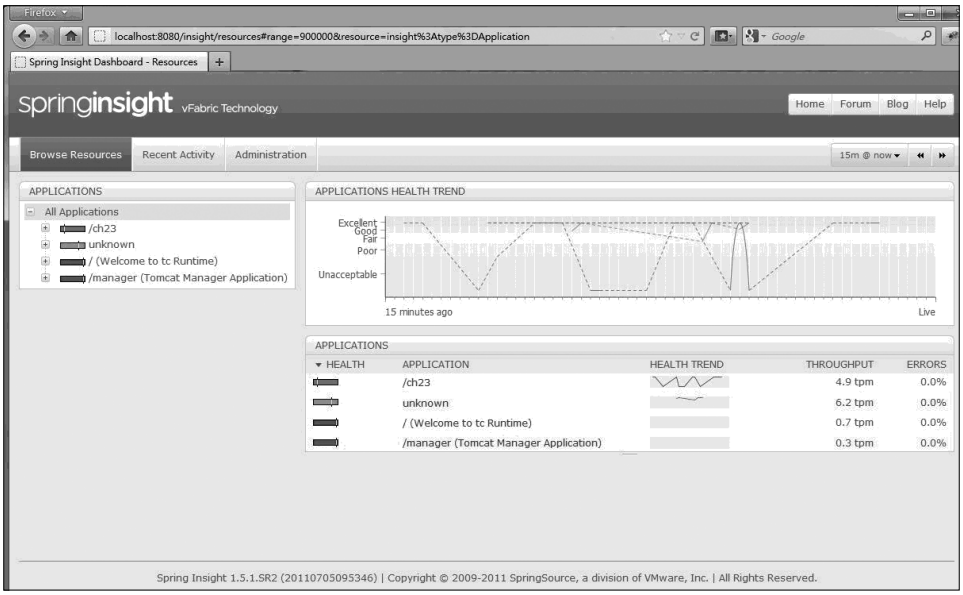


Рис. 23.20. Домашняя страница Spring Insight

Рис. 23.21. Spring Insight: мониторинг конечных точек

Чтобы просмотреть детали какой-то конечной точки, щелкните на ней в левой части, и тогда в правой части отобразятся подробные сведения, как показано на рис. 23.22.

На рис. 23.22 представлены вызовы для конечной точки `ContactController#listGrid` (соответствующей методу `listGrid()` в классе `ContactController`, который будет вызываться сеткой jQuery для извлечения страницы контактной информации с целью отображения). Справа видна гистограмма времени отклика, которая показывает различные вызовы в течение отслеживаемого периода. Щелчок на любом вызове приводит к отображению под гистограммой трассировочной информации.

Пример приведен на рис. 23.23.

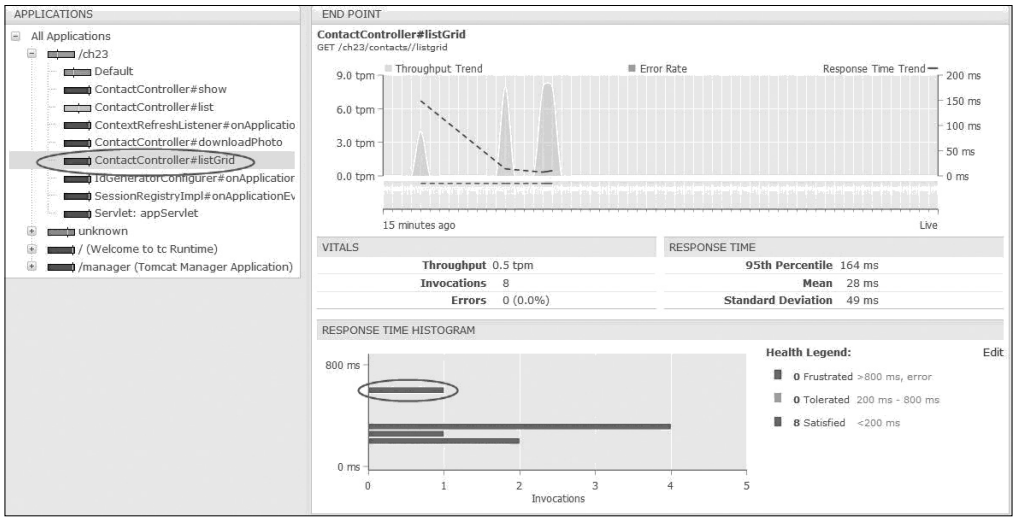


Рис. 23.22. Spring Insight: подробные сведения о конечной точке

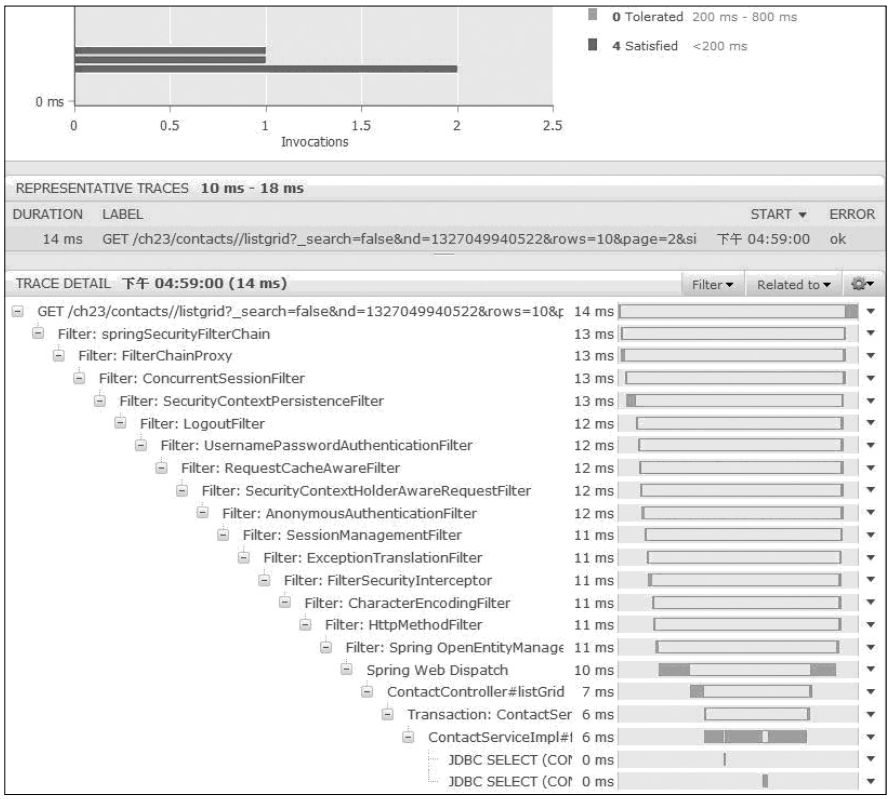


Рис. 23.23. Spring Insight: подробные сведения о вызове конечной точки

На рис. 23.23 можно видеть подробное дерево вызовов, затраченное время и запросы, которые были запущены в базе данных. Таким образом, вы имеете полную картину взаимодействия различных бинов Spring внутри приложения, а также время выполнения запросов, отправленных базе данных. С помощью этих сведений вы сможете быстро идентифицировать узкое место приложения и предпринять соответствующие действия по его устранению.

На этом введение в Spring Insight завершено. Spring Insight — очень полезный инструмент, который помогает анализировать много разных аспектов приложения. Можно даже воспользоваться комплектом Spring Insight Developer Kit для разработки подключаемых модулей, которые будут расширять Spring Insight для сбора специфичных метрик, связанных с приложением. Примером может служить анализ трассировочной информации из Spring Insight и построение новых типов конечных точек, таких как очередь сообщений JMS.

За подробными сведениями о применении Spring Insight Developer обращайтесь в онлайновую документацию SpringSource по адресу `http://static.springsource.com/projects/tc-server/2.5/devedition/htmlsingle/devedition.html`.

Резюме

В этой главе были рассмотрены различные вопросы, касающиеся мониторинга JEE-приложений, основанных на Spring.

Во-первых, мы обсудили поддержку JMX в Spring (JMX — это стандарт для мониторинга Java-приложений). Рассматриваемые темы включали реализацию специальных MBean-компонентов для открытия информации, связанной с приложением, а также открытие статистических данных по таким общим компонентам, как Hibernate и Spring Batch.

Во-вторых, мы продемонстрировали использование Spring Insight для визуализации показателей производительности приложения, выполняемого на сервере tc Server. Рассматриваемые темы включали настройку экземпляра tc Server с включенным инструментом Spring Insight, а также краткое описание ряда основных метрик, которые можно просматривать через веб-интерфейс Spring Insight.

ПРИЛОЖЕНИЕ A

SpringSource Tool Suite (STS)

В главах этой книги были предоставлены конкретные инструкции по использованию STS при создании проектов и реализации кода примеров, рассматриваемых в каждой главе. Если вы внимательно проработали все примеры, то должны довольно неплохо знать STS.

Цель этого приложения заключается в описании других аспектов работы с STS, которые не были раскрыты в главах книги. Приложение предназначено для разработчиков, которые не знакомы с STS. Тем не менее, предполагается наличие базовых знаний по применению Eclipse для разработки Java-приложений.

Если же вы являетесь опытным пользователем Eclipse либо хорошо знакомы с STS (или Spring IDE), можете спокойно пропустить это приложение и обращаться к нему за справкой только по мере необходимости.

В этом приложении мы обсудим некоторые базовые вопросы, связанные с STS. В частности, будут рассматриваться следующие темы.

- *Установка.* Мы покажем, как установить STS. Хотя STS поддерживает также операционные системы Mac OS X и Linux, мы сконцентрируем внимание на установке и использовании STS на платформе Windows.
- *Настройка проекта и управление зависимостями.* Мы объясним, как создавать проекты Spring в STS и управлять обязательными зависимостями внутри проекта. Мы будем ориентироваться на управление зависимостями Maven с применением подключаемого модуля m2e (официальный подключаемый модуль Maven для Eclipse IDE).
- *Использование STS.* Мы обсудим установку расширений, управление сервером tc Server и т.п.

Введение в STS

STS — это инструмент IDE-среды, предлагаемый SpringSource разработчикам Spring-приложений. В его состав входит набор подключаемых модулей, которые предоставляют поддержку для разработки приложений на базе Spring. Кроме того, STS может быть интегрирован с другими подключаемыми модулями Eclipse (такими как m2e, AspectJ Development Tool и т.д.). Ниже отмечены некоторые важные функциональные возможности STS.

- Он поддерживает создание и редактирование XML-файлов конфигурации `ApplicationContext` для Spring с помощником и проверкой правильности кода.
- Он предоставляет графические представления конфигурации Spring, такие как граф бинов, диаграмма веб-потоков, диаграммы Spring Batch и Spring Integration и т.д.
- Он поддерживает установку расширений для других средств (например, для разработки классов и сценариев Groovy, Grails и т.д.).
- Он поддерживает управление сервером tc Server Developer Edition для локальной разработки и тестирования веб-приложений.

Установка STS

Инструмент STS может быть установлен двумя путями. Во-первых, в SpringSource подготовили версию, встроенную с Eclipse, и требуемый подключаемый модуль. Во-вторых, если среда разработки Eclipse уже имеется, можно просто расширить ее с помощью подключаемых модулей, предоставляемых STS. В любом случае удостоверьтесь в наличии установленного комплекта JDK (версии 6 или новее).

Установка автономной версии STS

Простейший способ установки STS предусматривает загрузку из веб-сайта STS пакета, с которым связан Eclipse. Итак, вам понадобится загрузить подходящую версию для машины разработки из веб-сайта инструментов (`www.springsource.org/downloads/sts`). На рис. A.1 показана страница загрузки на веб-сайте с версиями, доступными для платформы Windows (обратите внимание, что имеются также версии для платформ Mac OS X и Linux).

Выберите версию, подходящую для вашего компьютера. В случае Windows доступны версии для 32- и 64-разрядной платформы. Кроме того, существуют две формы дистрибутива. Одна из них имеет вид файла в формате установщика Windows, на котором можно просто щелкнуть для запуска процесса установки. Другая форма — zip-архив, который понадобится предварительно распаковать и затем уже применять для установки STS.

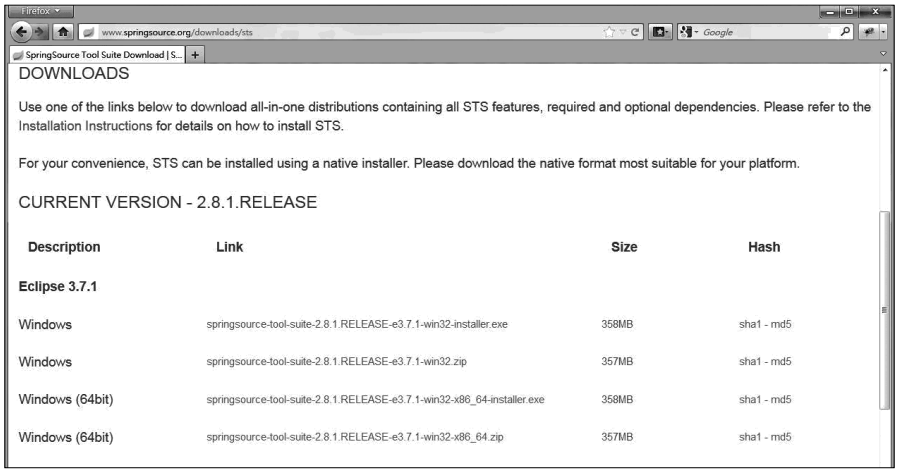


Рис. A.1. Загрузка автономной версии STS

В демонстрационных целях мы будем использовать файл в формате установщика для 64-разрядной версии Windows (на время написания книги номером последней версии был 2.8.1). Установка осуществляется очень просто. После завершения загрузки щелкните на исполняемом файле для запуска процесса установки. Установка проходит за несколько шагов; на рис. A.2 показан шаг 1.

Щелкните на кнопке Next (Далее) и перейдите к шагу 2, который представлен на рис. A.3.

Ознакомьтесь и примите условия лицензионного соглашения, затем щелкните на кнопке Next, чтобы перейти к следующему шагу, как показано на рис. A.4.

Рис. A.2. Установка STS: шаг 1

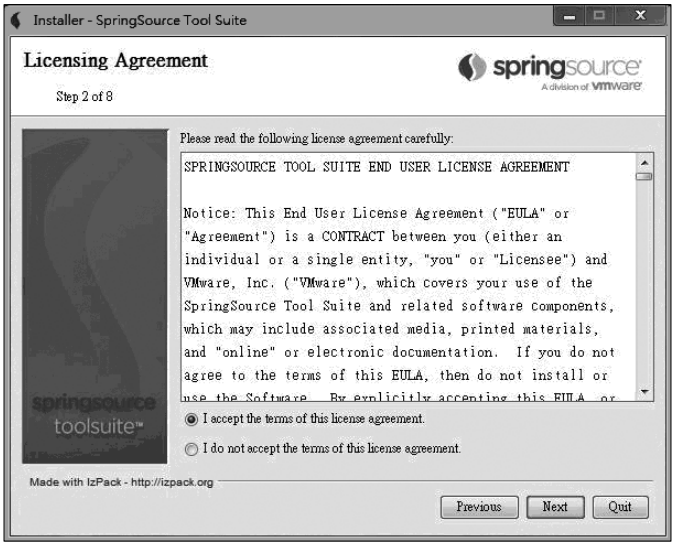


Рис. A.3. Установка STS: шаг 2

Рис. A.4. Установка STS: шаг 3

Выберите папку, в которую необходимо установить STS. Затем щелкните на кнопке Next, чтобы перейти к следующему шагу, как показано на рис. A.5.

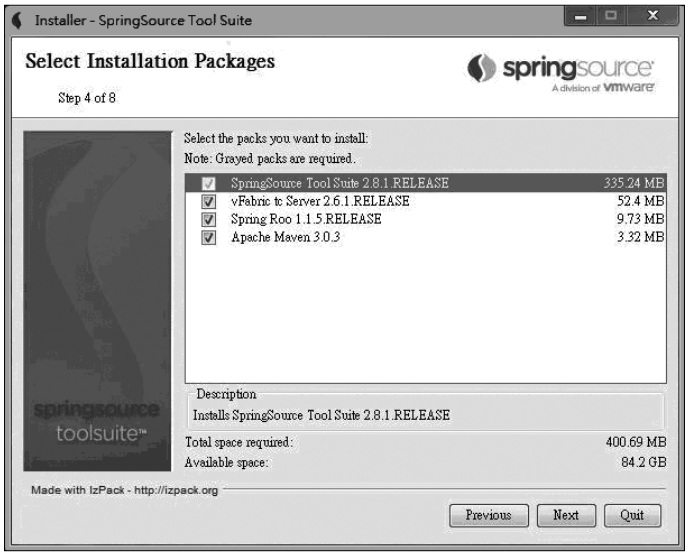


Рис. A.5. Установка STS: шаг 4

На шаге 4 отобразятся доступные необязательные элементы, которые также можно выбрать. В их число входят tc Server Developer Edition, Spring Roo и автономная версия Apache Maven (основанная на версии 3). Удостоверьтесь, что отмечены флажки возле всех элементов, и щелкните на кнопке Next, чтобы перейти к следующему шагу, как показано на рис. A.6.

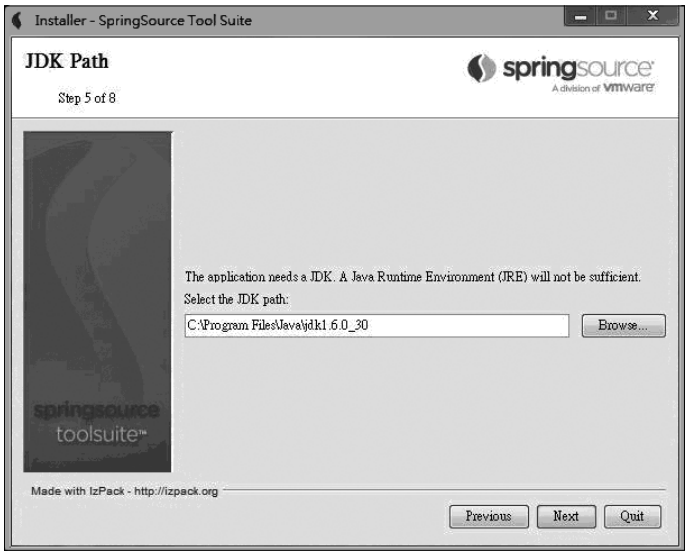


Рис. A.6. Установка STS: шаг 5

Выберите папку с установленным комплектом JDK на своем компьютере, и щелкните на кнопке Next, чтобы перейти к следующему шагу, как показано на рис. A.7.

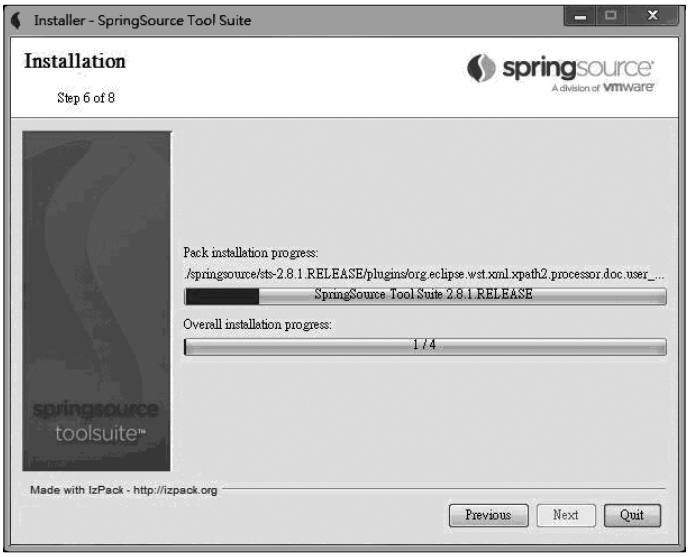


Рис. A.7. Установка STS: шаг 6

Дождитесь завершения установки и отметьте флажок Create shortcuts in the Start-Menu (Создать ярлыки в меню “Пуск”), как показано на рис. A.8.

Щелкните на кнопке Next, чтобы завершить установку, как показано на рис. A.9.

К этому моменту установка автономной версии STS завершена. Щелкните на кнопке Finish (Готово).

Рис. A.8. Установка STS: шаг 7

Рис. A.9. Установка STS: шаг 8

Установка STS в существующей среде Eclipse

Другой способ установки STS предусматривает расширение существующей версии Eclipse. Ниже мы продемонстрируем установку STS в существующей среде Eclipse (версии 3.7.1). Eclipse поступает с различными пакетами. В рассматриваемом примере используется версия Eclipse IDE for Java EE Developers (IDE-среда Eclipse для разработчиков Java EE); список доступных пакетов можно посмотреть по адресу `www.eclipse.org/downloads`. Перед тем, как можно будет установить STS, понадобится установить ряд подключаемых модулей Eclipse, требуемых STS.

Чтобы установить подключаемый модуль Maven для Eclipse, выберите в Eclipse пункт меню Help⇨Install New Software (Справка⇨Установить новое программное обеспечение). В открывшемся диалоговом окне Install (Установка) щелкните на кнопке Add (Добавить), введите имя (`m2e`) и URL сайта обновления `http://download.eclipse.org/technology/m2e/releases`, после чего щелкните на кнопке OK. Затем выберите компонент для установки, как показано на рис. A.10.

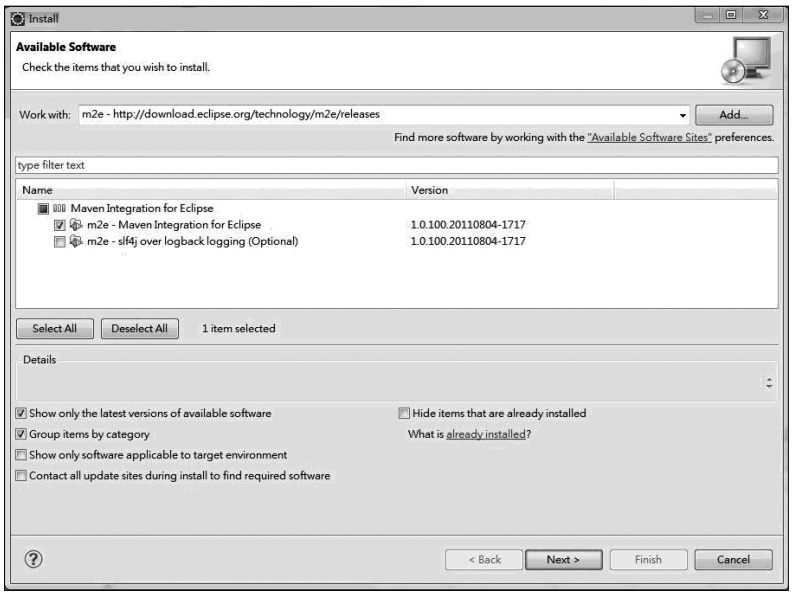


Рис. A.10. Установка подключаемого модуля Maven для Eclipse

Следующий подключаемый модуль, который нужно установить — это исполняющая среда Jetty, которая включает обновление подключаемого модуля JSP, требуемого STS. Щелкните на кнопке Add в диалоговом окне Install, чтобы открыть окно Add Repository (Добавить репозиторий), и введите имя (`Jetty Runtime`), а также URL сайта обновления `http://download.eclipse.org/jetty/updates/3.7`. После добавления репозитория в диалоговом окне Install отобразятся доступные компоненты. Выберите компоненты для установки и щелкните на кнопке Next, как показано на рис. A.11.

После того, как все это завершено, можно приступать к установке подключаемых модулей STS. Подготовьте файл по имени `bookmarks.xml` с содержимым, приведенным в листинге A.1.

Листинг A.1. Файл `bookmarks.xml` для STS

```
<?xml version="1.0" encoding="UTF-8"?>
<bookmarks>
  <site url="http://dist.springsource.com/release/TOOLS/update/e3.7" selected="true"
        name="SpringSource Update Site for Eclipse 3.7"/>
  <site url="http://dist.springsource.com/release/TOOLS/composite/e3.7"
        selected="true"
        name="SpringSource Update Site for Eclipse 3.7 (Dependencies)"/>
  <site url="http://download.eclipse.org/releases/indigo"
        selected="true" name="Indigo"/>
</bookmarks>
```

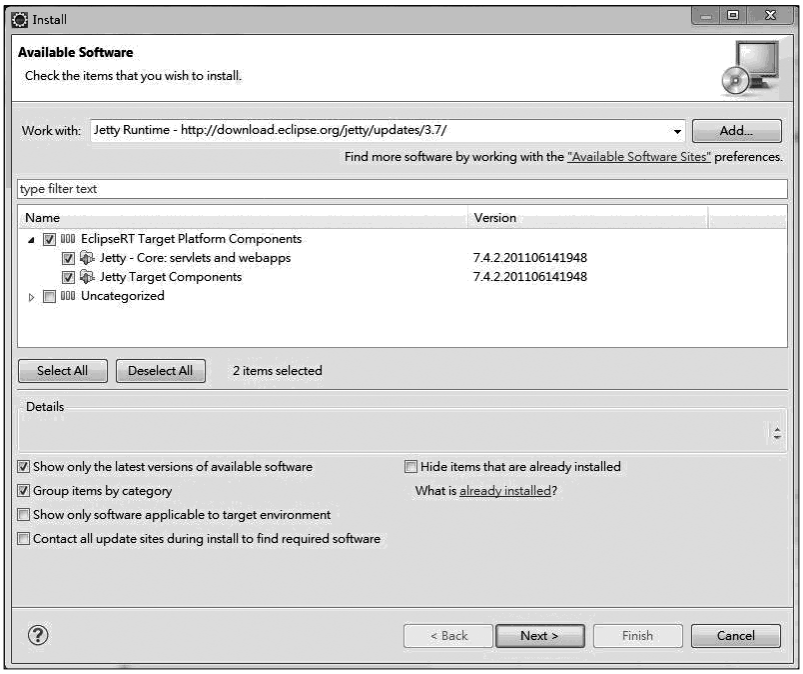


Рис. A.11. Установка исполняющей среды Jetty для Eclipse

Этот файл содержит все сайты обновлений, требуемые для установки STS. Откройте диалоговое окно Install (выбрав пункт Help⇨Install New Software в верхнем меню Eclipse), которое было показано на рис. A.11. Щелкните на ссылке Available Software Sites (Доступные сайты программного обеспечения), затем щелкните на кнопке Import (Импортировать) и выберите подготовленный ранее файл `bookmarks.xml` (рис. A.12).

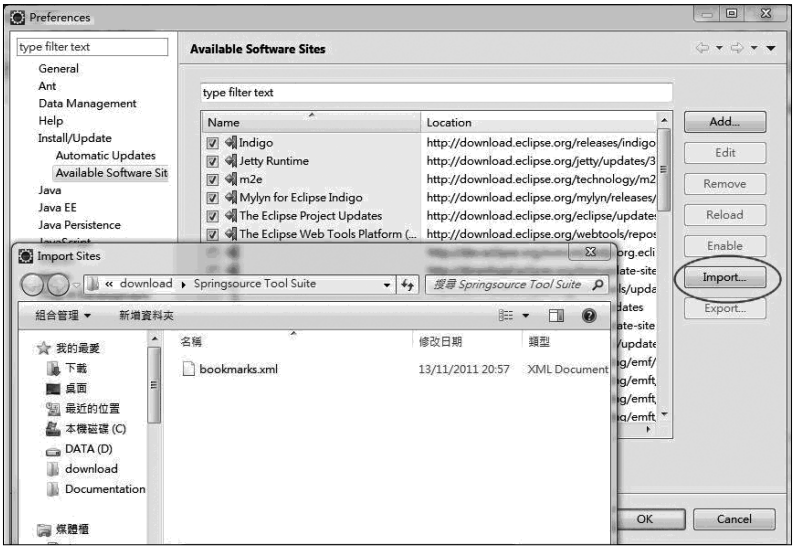


Рис. A.12. Импорт сайтов обновления STS в Eclipse

По завершении вы увидите, что сайты обновления STS были добавлены, как показано на рис. A.13.

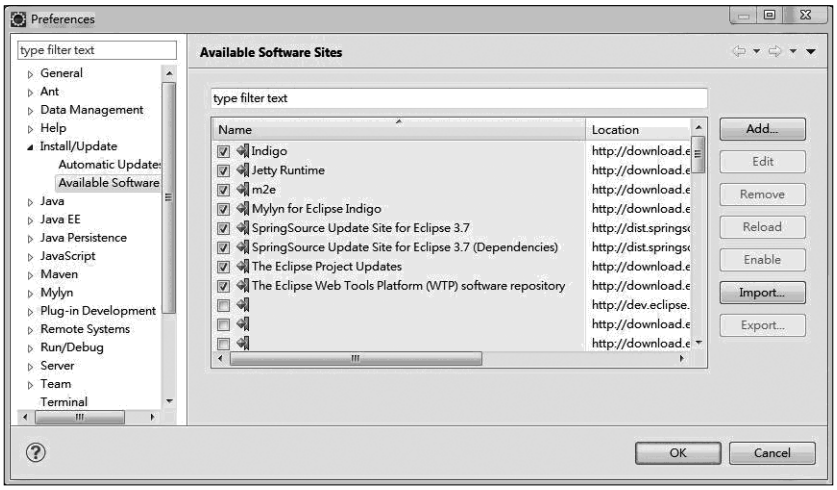


Рис. A.13. Сайты обновления STS в Eclipse

Вернитесь в диалоговое окно Install и выберите компоненты для установки (рис. A.14).

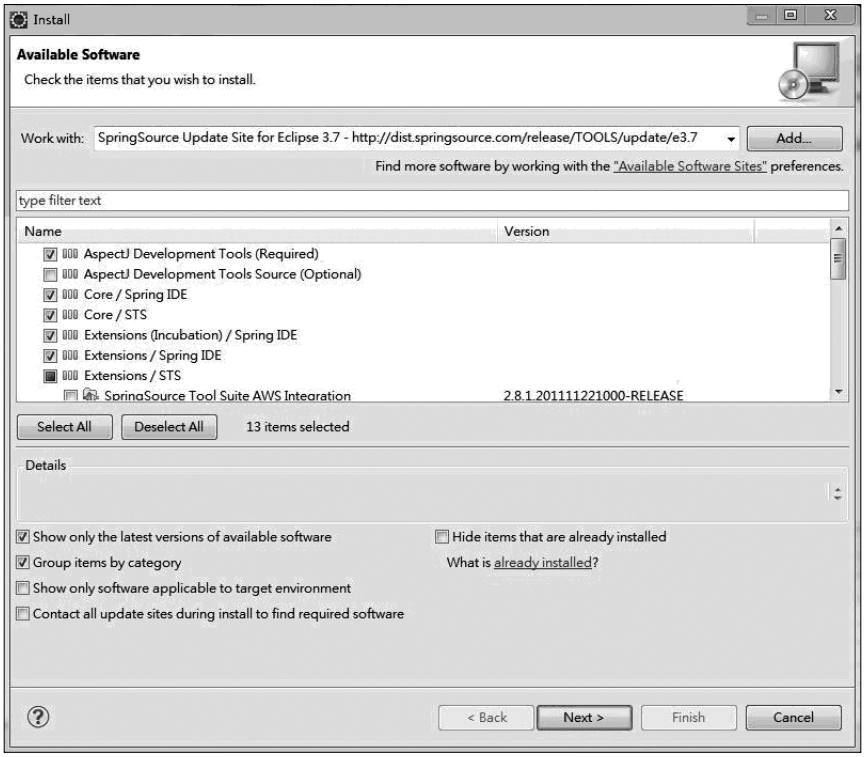


Рис. A.14. Выбор компонентов STS

Нам не нужны абсолютно все компоненты. Вдобавок некоторые компоненты требуют предварительной установки дополнительных подключаемых модулей. Выберите компоненты, подходящие для вашей среды; на рис. A.15 приведен пример выбора.

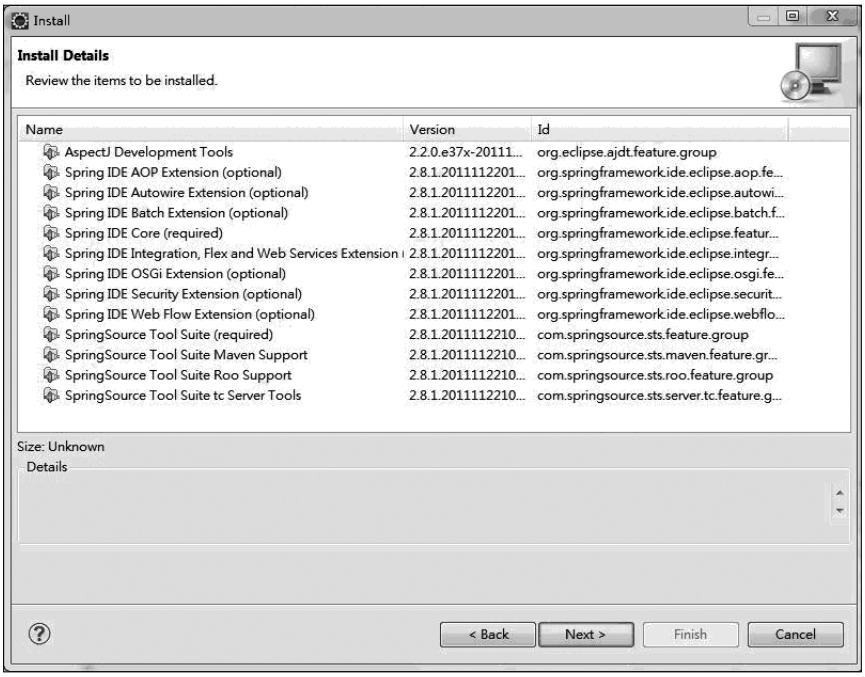


Рис. A.15. Выбранные компоненты STS

Щелкните на кнопке Next и следуйте инструкциям по запуску процесса установки. Как только установка завершится, функции, связанные с STS, станут доступными.

Настройка проекта и управление зависимостями

После установки STS можно создать проект и приступить к разработке. STS предоставляет несколько шаблонных проектов с предварительно сконфигурированными настройками и зависимостями. Выберите пункт File⇨New⇨Spring Template Project (Файл⇨Создать⇨Шаблонный проект Spring) в меню STS и на выбор будет представлен список доступных шаблонных проектов. На рис. A.16 показаны доступные шаблонные проекты для версии 2.8.1. Щелчок на названии шаблонного проекта позволяет просмотреть его краткое описание.

Создание простого служебного проекта Spring

Давайте создадим простой проект. Выберите шаблон Simple Spring Utility Project (Простой служебный проект Spring) и щелкните на кнопке Next (Далее). После этого введите информацию о проекте, как показано на рис. A.17.

STS создаст проект. Все шаблонные проекты STS сгенерированы в соответствии со стандартной структурой папок проекта, принятой в Maven, и все они используют Maven для управления зависимостями. Таким образом, проекту придаются все свойства, характерные для Spring и Maven (рис. A.18).

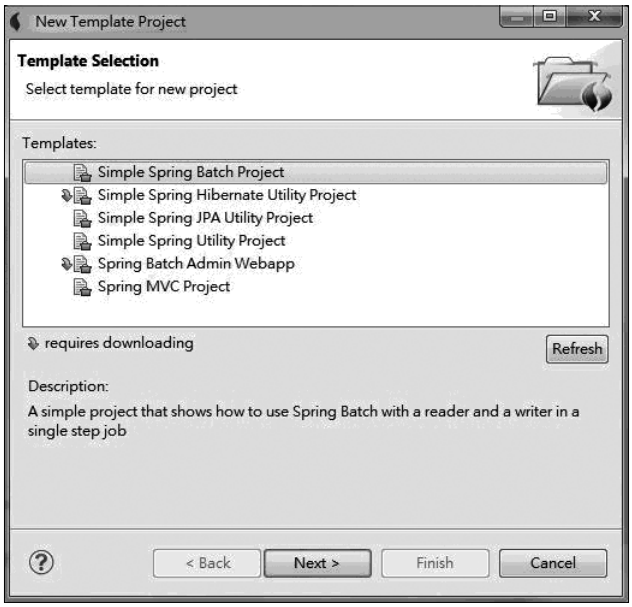


Рис. A.16. Выбор шаблонного проекта Spring

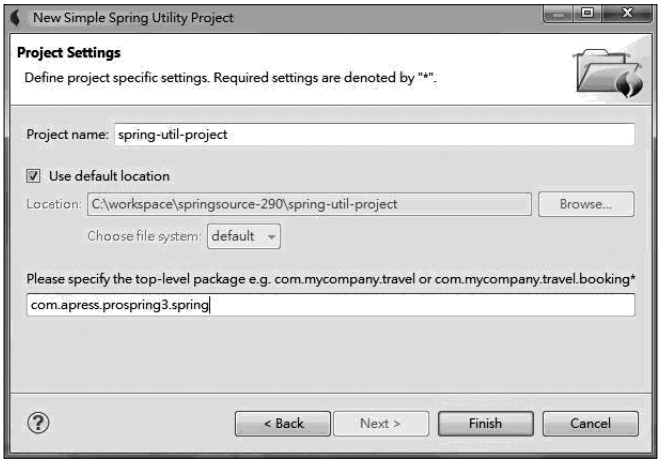


Рис. A.17. Создание простого служебного проекта Spring

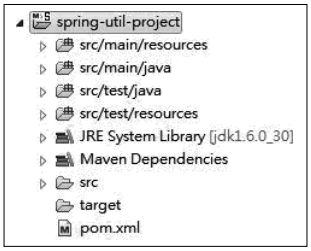


Рис. A.18. Простой служебный проект Spring

После создания проекта понадобится обновить конфигурацию, сгенерированную STS. Например, на время написания этой книги сгенерированный проект по умолчанию использовал Spring Framework 3.0.6.RELEASE и JDK 5. Давайте изменим конфигурацию, чтобы проект работал с версиями Spring Framework 3.1.0.RELEASE и JDK 6. Дважды щелкните на файле `pom.xml`, в открывшемся редакторе перейдите на вкладку `pom.xml` и модифицируйте конфигурацию, как показано на рис. A.19.

Тем не менее, одного лишь изменения файла `pom.xml` недостаточно. Например, если вы щелкнете правой кнопкой мыши на проекте и выберите в контекстном меню пункт Properties (Свойства), а затем элемент Java Compiler (Компилятор Java), то увидите, что проект по-прежнему использует JDK 5 (рис. A.20).

```
*spring-util-project/pom.xml
 6      <groupId>org.springframework.samples.spring</groupId>
 7      <artifactId>spring-utility</artifactId>
 8      <version>1.0.0.CI-SNAPSHOT</version>
 9      <packaging>jar</packaging>
10      <name>Spring Utility</name>
11      <url>http://www.springframework.org</url>
12      <description>
13          <![CDATA[
14        This project is a minimal jar utility with Spring configuration.
15      ]]>
16      </description>
17      <properties>
18          <maven.test.failure.ignore>true</maven.test.failure.ignore>
19          <spring.framework.version>3.1.0.RELEASE</spring.framework.version>
20      </properties>   <dependencies>
21          <dependency>
22              <groupId>junit</groupId>
23              <artifactId>junit</artifactId>
24              <version>4.7</version>
25              <scope>test</scope>
26          </dependency>
27          <dependency>
28              <groupId>org.springframework</groupId>
29              <artifactId>spring-test</artifactId>
30              <version>${spring.framework.version}</version>
31              <scope>test</scope>
32          </dependency>
33          <dependency>
34              <groupId>org.springframework</groupId>
35              <artifactId>spring-context</artifactId>
36              <version>${spring.framework.version}</version>
37          </dependency>
38          <dependency>
39              <groupId>log4j</groupId>
40              <artifactId>log4j</artifactId>
41              <version>1.2.14</version>
42          </dependency>
43      </dependencies>
44
45      <build>
46          <plugins>
47              <plugin>
48                  <groupId>org.apache.maven.plugins</groupId>
49                  <artifactId>maven-compiler-plugin</artifactId>
50                  <configuration>
51                      <source>1.6</source>
52                      <target>1.6</target>
53                  </configuration>
54              </plugin>
55
56          </plugins>
57      </build>
58  </project>
59
Overview | Dependencies | Dependency Hierarchy | Effective POM | pom.xml
```

Рис. A.19. Изменение конфигурации Maven

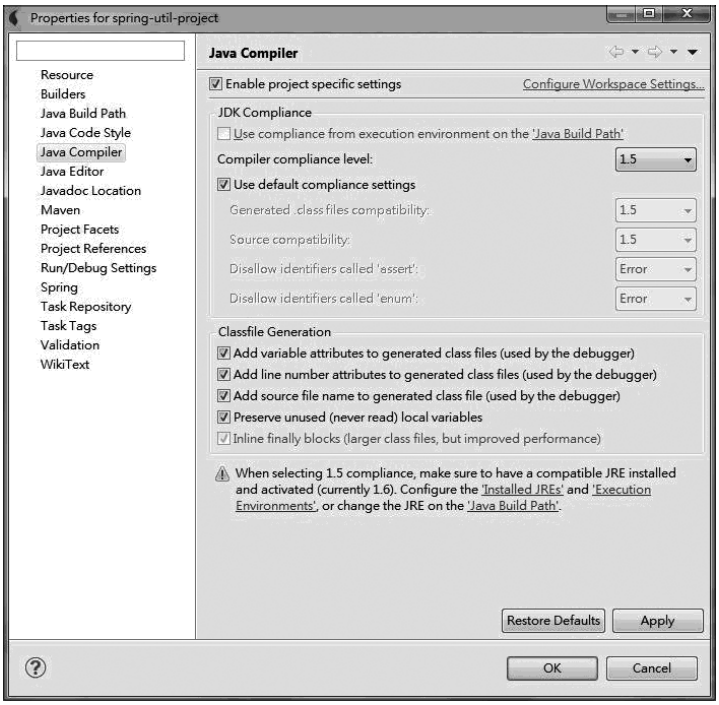


Рис. A.20. Свойства проекта, связанные с компилятором Java

Причина в том, что несмотря на внесение изменений в файл `pom.xml`, подключаемый модуль Maven не обновляет конфигурацию проекта автоматически. Обновить конфигурацию проекта можно, либо напрямую изменив свойства, представленные на рис. A.20, либо выполнив щелчок правой кнопкой мыши на проекте и выбрав в контекстном меню пункт Maven⇨Update Project Configuration (Maven⇨Обновить конфигурацию проекта), как показано на рис. A.21.

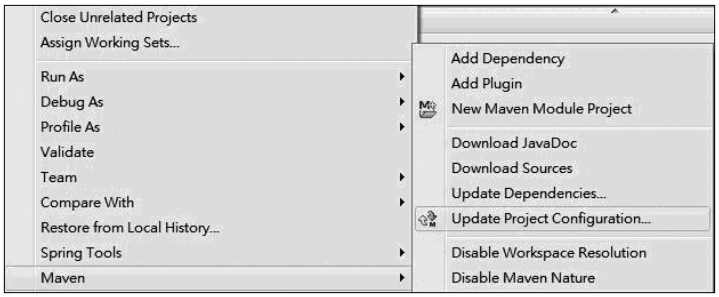


Рис. A.21. Обновление конфигурации проекта

Однако в версии STS 2.8.1 с установленным подключаемым модулем m2e проявляется странное поведение: после обновления конфигурации проекта для Maven пропадают характерные свойства Spring. В этом случае их нужно добавить обратно, выполнив щелчок правой кнопкой мыши на проекте и выбрав в контекстном меню пункт Spring Tools⇨Add Spring Project Nature (Инструменты Spring⇨Добавить характерные свойства Spring).

Управление зависимостями для проекта

Благодаря подключаемому модулю Maven, управление библиотечными зависимостями для проекта значительно упрощается. Например, в представлении редактора файла `pom.xml` щелкните на вкладке Dependencies (Зависимости), как показано на рис. A.22.

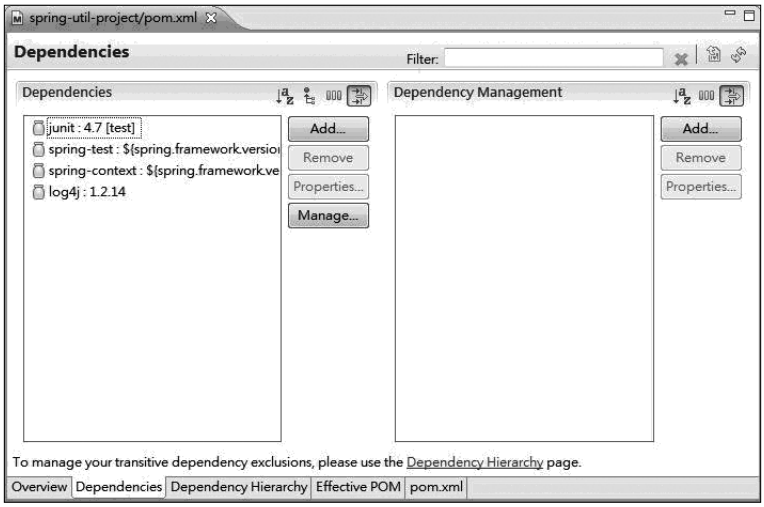


Рис. A.22. Управление зависимостями Maven в проекте

Чтобы добавить зависимость в проект, щелкните на кнопке Add (Добавить), расположенной слева; откроется диалоговое окно Select Dependency (Выбор зависимости). Введите в нем информацию о зависимости. На рис. A.23 можно видеть диалоговое окно Select Dependency с введенной информацией для добавления зависимости Hibernate Entity Manager (Диспетчер сущностей Hibernate), которая включает библиотеку Hibernate и ее поддержку JPA (они будут отражены в иерархии на рис. A.24 после добавления зависимости).

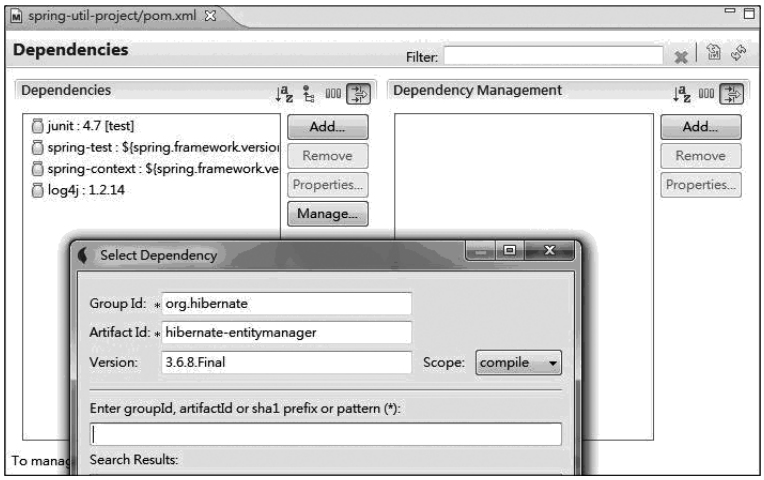


Рис. A.23. Добавление зависимости Maven в проект

После того, как зависимость добавлена, сохраните файл. Зависимости Maven для проекта будут обновлены автоматически. На вкладке Dependency Hierarchy (Иерархия зависимостей) можно просмотреть полную иерархию зависимостей, как показано на рис. A.24.

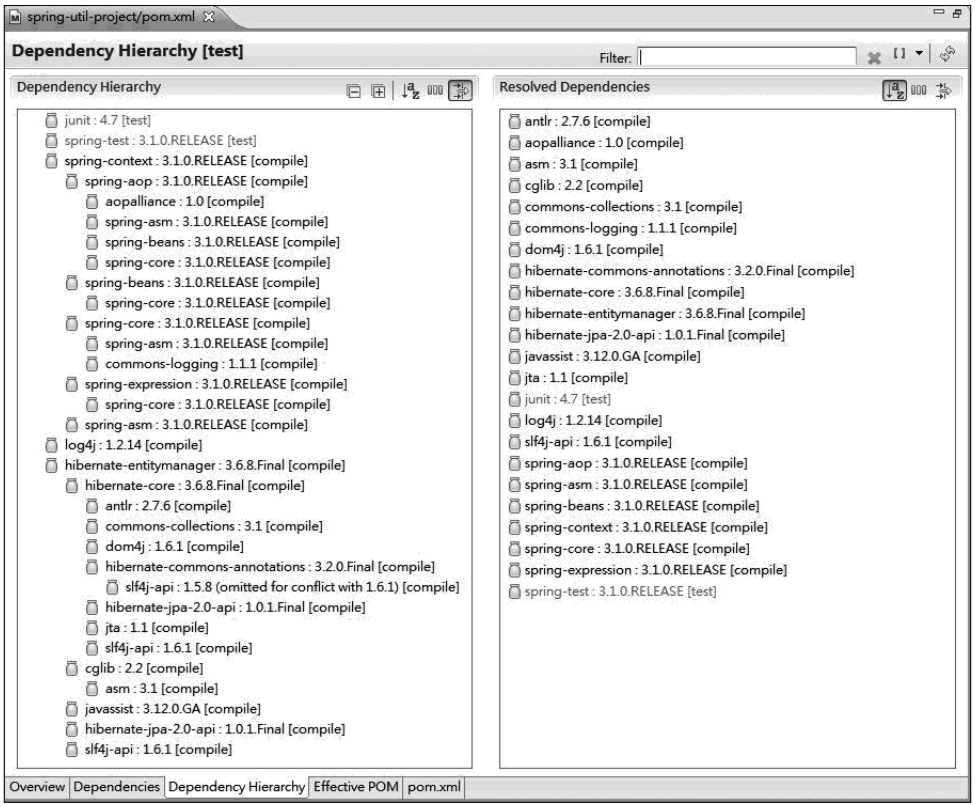


Рис. A.24. Иерархия зависимостей Maven для проекта

На рис. A.24 вы заметите, что после добавления зависимости Hibernate Entity Manager также добавляются другие обязательные зависимости (например, для библиотеки Hibernate Core). Это средство называется *управлением транзитивными зависимостями* в Maven.

В разных проектах требуются разные зависимости. Во всех случаях при управлении зависимостями можно следовать описанной здесь процедуре.

Использование STS

Теперь вам в основном понятно, что конкретно полезного может делать STS. В этом разделе мы обсудим другие области, связанные с STS, такие как установка расширений и управление сервером tc Server.

Установка расширений STS

STS поставляется в базовым набором подключаемых модулей, в состав которых входят модули для поддержки АОП, Spring Beans, Spring Security, Spring Web Flow и т.д.

Кроме этих подключаемых модулей, STS также предоставляет множество расширений, которые поддерживают разработку Spring-приложений либо интеграцию с другими популярными инструментами и инфраструктурами.

Чтобы просмотреть доступные расширения и установить нужные из них, откройте представление Dashboard (Инструментальная панель). Быстрее всего открыть это представление можно, щелкнув на значке Dashboard (Инструментальная панель) панели инструментов (рис. A.25).

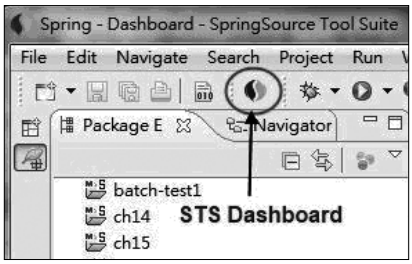


Рис. A.25. Значок для открытия представления Dashboard в STS

Находясь в представлении Dashboard, щелкните на вкладке Extensions (Расширения) и отобразится список доступных расширений, как показано на рис. A.26.

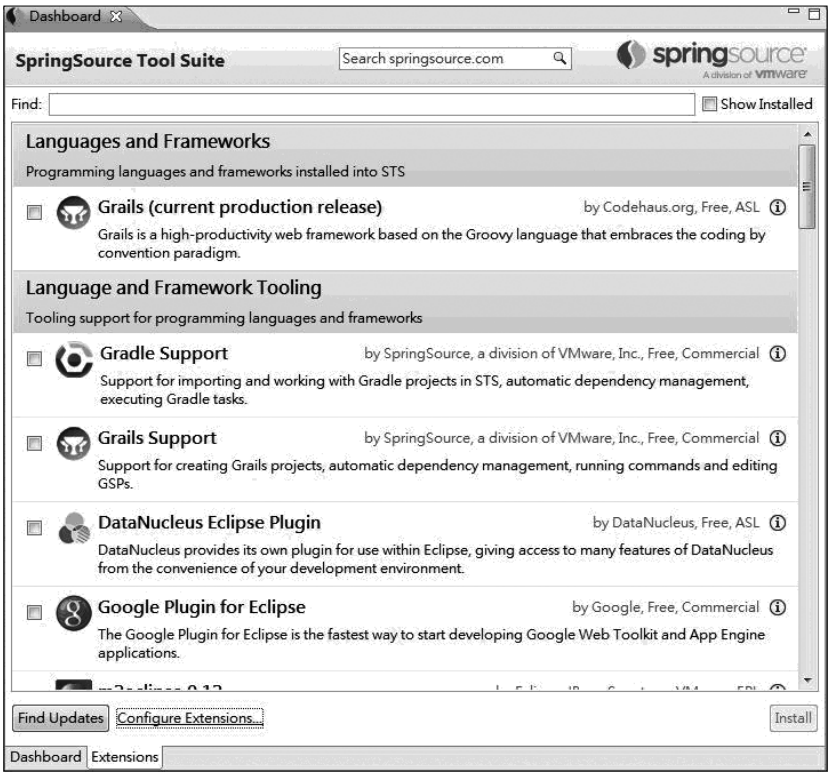


Рис. A.26. Список доступных расширений в STS

На рис. A.26 видно множество расширений, среди которых поддержка Grails, подключаемого модуля Google для Eclipse и т.д. Отметьте флажки возле желаемого расширения и щелкните на кнопке Install (Установка) для его установки.

Конфигурирование VMware tc Server в STS

STS поставляется вместе с tc Server Developer Edition, встроенным сервером Apache Tomcat 7, который очень удобен при локальной разработке веб-приложений. Сервер tc Server поддерживает создание экземпляров с разнообразными возможностями. Чтобы создать новый экземпляр сервера, щелкните правой кнопкой мыши на пустой области вкладки Servers (Серверы) (для доступа к которой необходимо выбрать пункт Window⇨Show View⇨Servers (Окна⇨Показать представление⇨Серверы) в верхнем меню STS) и выберите в контекстном меню пункт New⇨Server (Новый⇨Сервер) для открытия диалогового окна мастера создания нового сервера (рис. A.27).

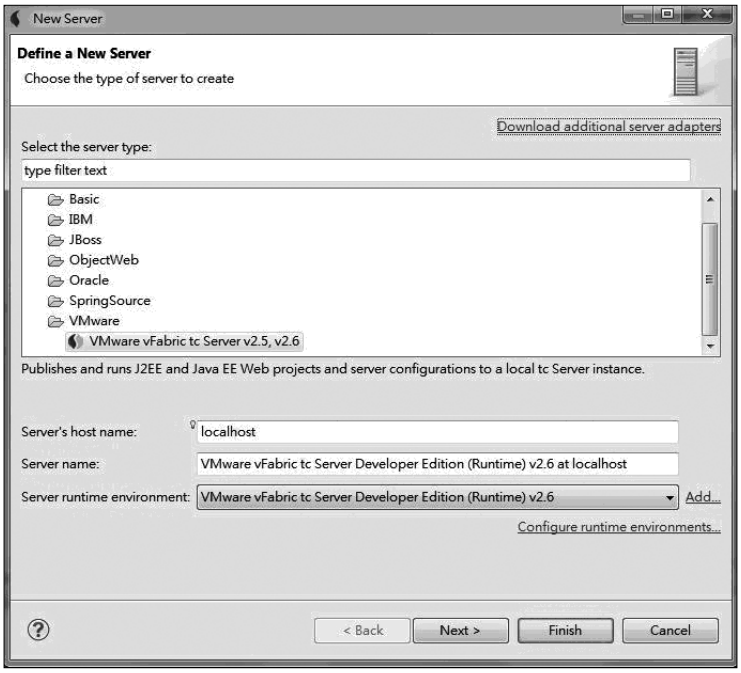


Рис. A.27. Определение нового сервера

Щелкните на кнопке Next (Далее) и выберите на следующем экране мастера переключатель Create new instance (Создать новый экземпляр). На открывшемся экране Create tc Server Instance (Создание экземпляра tc Server) мастера создания нового сервера введите имя для экземпляра сервера, после чего укажите шаблон, который должен использоваться для нового экземпляра (рис. A.28).

Как показано на рис. A.28, выбранный шаблон base (базовый) заставляет STS создать экземпляр tc Server на базе Tomcat 7. За подробным описанием всех шаблонов, а также применения tc Server во время разработки обращайтесь в онлайновую документацию по адресу:

```
http://static.springsource.com/projects/tc-server/2.5/getting-started/
    htmlsingle/getting-started.html.
```

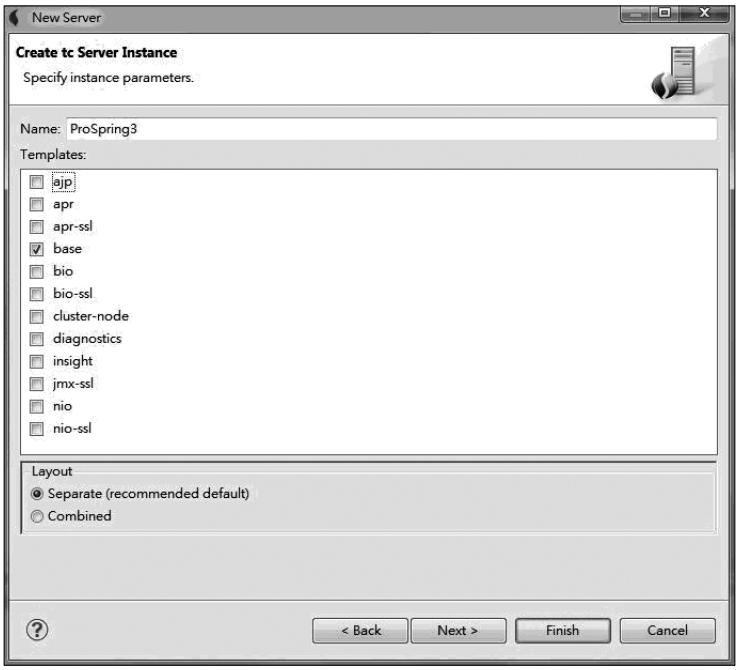


Рис. A.28. Создание нового экземпляра tc Server

Резюме

В этом приложении было показано, как использовать STS. Рассматриваемые темы включали установку STS, применение шаблонных проектов Spring, управление зависимостями Maven, обновление конфигурации проекта и управление сервером tc Server.

Предметный указатель

A

ACID (atomicity, consistency, isolation, durability), 479
ACL (Access Control List), 31
Ajax, 27; 666
Apache Tiles, 622
AspectJ, 206; 285
Aspect-Oriented Programming (AOP), 203

B

Bean-managed transaction (BMT), 475
Bean Validation API, 525

C

Constructor Injection, 78
Container-managed transaction (CMT), 475
Contextualized Dependency Lookup (CDL), 74
Customer relationship management (CRM), 405

D

Data access object (DAO), 21; 69
Dependency Injection (DI), 20; 73; 203
Dependency Pull, 74; 75
Domain Object Model (DOM), 454
Domain-specific language (DSL), 809

E

Eclipse, 858
Eclipse AspectJ Development Tools (AJDT), 286
Enterprise Integration Pattern (EIP), 31; 69
Enterprise JavaBeans (EJB), 21

G

Google Guice, 31
Google Web Toolkit (GWT), 596

H

Hibernate, 339; 363; 840
Hibernate Envers (Entity Versioning System), 405
Hibernate Query Language (HQL), 355
HTTP-активатор Spring, 554; 564

I

Inversion of Control (IoC), 20; 71; 73

J

JavaBean, 20
  редакторы свойств для компонентов JavaBean, 163
  встроенные, 164
  конфигурирование, 166
Java Connector Architecture (JCA), 477
Java Community Process (JCP), 21
Java Data Objects (JDO), 67; 339; 365
Java Enterprise Edition версии 6 (JEE 6), 21
Java Messaging Service (JMS), 554
Java Persistence API (JPA), 365; 412
Java Server Faces (JSF), 70; 674
Java Server Pages (JSP), 70
Java Transaction API (JTA), 475; 476
Java Transaction Service (JTS), 477
Java-класс конфигурации, 185
Java-классы, 181
JDBC Extensions, 337
JDBC (Java Database Connectivity), 25; 297
  пакеты JDBC в Spring, 303
JPetStore, 44
JPQL (Java Persistence Query Language), 365
JQuery, 640
JQuery UI, 640
JSON (JavaScript Object Notation), 27
JTA Transaction Manager, 26

L

Lookup Method Injection, 112

M

Maven, 38
Message queue (MQ), 566
Method Injection, 112
MVC (Model View Controller), 23; 603
MyBatis, 415; 450

O

Object-relational mapping (ORM), 66; 339
Object to XML Mapping (OXM), 25; 70
Oracle Database, 337

P

Petclinic, 44
Petclinic Groovy, 44
Plain Old Java Object (POJO), 20

Q

Quality assurance (QA), 829
QueryDSL (Query Domain Specific Language), 337; 365

R

REpresentational State Transfer (REST), 574
RESTful Web Services (RESTful-WS), 27
Rich Internet Application (RIA), 27
Remote Method Invocation (RMI), 553
RSS-лента, 63

S

Seam Framework, 31
Secure Copy Protocol (SCP), 110
Selenium, 728
Selenium IDE, 728
Service Provider Interface (SPI), 509
Setter Injection, 78
Singleton, 123
Spring, 33
Spring Batch, 69; 735; 737; 751
Spring Configuration Basic, 45
Spring Data, 337
Spring Expression Language (SpEL), 99
Spring Framework, 20; 34
Spring Insight, 845
Spring Integration, 69; 735; 751; 752
Spring MVC, 26; 576; 596; 665
Spring MVC Ajax, 45
Spring MVC Basic, 45
Spring MVC Showcase, 45
Spring Petcare, 46
Spring Roo, 735; 757
Spring Security, 30; 588; 656
SpringSource Tool Suite (STS), 30; 37; 64; 286; 853
Spring Task and Scheduling, 45
Spring Web Flow, 27; 46; 669; 672; 677

V

VisualVM, 833

А

Абстракция
  Environment, 193
  PropertySource, 193
Аннотация
  в стиле @AspectJ, 280
Аспектно-ориентированное программирование (АОП)
  конфигурирование
    декларативное, 284
    с использованием пространства имен aop, 280
Архитектура
  Enterprise JavaBeans, 21
  многоуровневая, 66
Аспект (aspect), 205; 211
Аудит, 413; 451
  Hibernate Envers, 405
Аутентификация, 58

Б

База данных, 294
  поддержка встроенной базы данных, 307
  серверной части, 450
Безопасность, 58; 597; 666
  реализация уровня безопасности, 70
Библиотека
  Castor, 25
  Castor XML, 576
  CKEditor, 643
  iBATIS, 416
  Java API for XML Binding (JAXB), 25
  jQuery, 27
  MyBatis, 415
    когда использовать MyBatis, 450
    конфигурация Spring для MyBatis, 420
  ORM, 339
  Quartz Scheduler, 538
  Spring Framework, 35
  XMLBeans, 25
  XStream, 25
  генератора метамоделей Hibernate в STS, 388
Бин (bean), 20; 82
  dataSource, 304; 345; 371
  EntityManagerFactory, 371
  sessionFactory, 345
  transactionManager, 345; 371
  автосвязывание бина, 128
  внедрение бина, 102
    конфигурирование, 103
    с использованием псевдонимов бина, 103
  доступ к бинам с применением псевдонимов, 122
  жизненный цикл бинов Spring, 139
  именование бинов, 121
  конфигурация неодиночного бина, 124
  наследование бинов, 130
    конфигурирование, 131
  области действия бинов, 126
  объявление бинов Spring
    стиль XML, 89
    стиль аннотаций, 89
    тестирование, 89
  проверка достоверности бинов JSR-303, 637
  уничтожение одиночного экземпляра бина, 147
  управляемый (managed bean), 832
  фабрики бинов, 84; 157
    доступ к фабрике бинов напрямую, 161
Блог
  загрузка блога из XML-файла, 64

В

Введение (introduction), 205; 261; 269
  на основе экземпляров, 263
  обнаружение модификации объекта с помощью введений, 264
  структура интерфейсов для введений, 262

Веб-приложение
  защита веб-приложения с помощью Spring Security, 656
Веб-службы REST, 554; 573; 574
  для контактной информации, 575
  защита веб-служб REST с помощью Spring Security, 588
Внедрение
  бина, 102
    конфигурирование, 103
    с использованием псевдонимов бина, 103
  зависимостей
    с использованием SpEL (стиль XML), 99; 100
    с использованием SpEL (стиль аннотаций), 101
    с помощью Spring, 82
    через конструктор, 77
  коллекции (стиль XML), 107
    конфигурирование, 108; 111
  через метод (Method Injection), 112

Д

Данные
  вставка в JPA, 381
  обновление, 383; 443
  привязка, 522
  удаление, 384; 448
Диаграмма
  UML-, 779
Диспетчер
  сущностей (EntityManager), 381
  транзакций JTA, 26
Дистрибутив Spring, 33
Документация Spring, 33; 46

Ж

Жизненный цикл приложения, 676

З

Зависимости
  Maven для Apache Tiles, 622
  Maven для AspectJ, 254
  Maven для Groovy и Joda-Time, 819
  Maven для JMS и ActiveMQ, 567
  Maven для JSON, 645
  Maven для Mockito, 712
  Maven для MySQL и Atomikos, 501
  Maven для Selenium, 728
  Maven для Spring Batch Admin, 842
  Maven для Spring Batch и Castor, 742
  Maven для Spring Data JPA, 392; 482
  Maven для Spring Integration, 753
  Maven для Spring MVC, 610
  Maven для Spring Security, 657
  Maven для Spring Web Flow, JSF 2 и PrimeFaces, 679
  Maven для веб-служб REST, 575
  Maven для загрузки файлов, 652
  Maven для модульного тестирования взаимодействия, 718
  Maven для планирования задач, 537
  Maven для проверки достоверности, 510
  Maven для уровня обслуживания, 556
  внедрение зависимостей
    с помощью Spring, 82
    через конструктор, 77
    через метод установки, 77
  для CGLIB, 209; 256
  для Hibernate, 342
  для Java-конфигурации, 184
  для MySQL, 302
  для spring-jdbc, 310
  для базы данных H2, 309
  для внедрения через метод, 116
  для поддержки JSR-330, 198
  для поддержки транзакций в Spring, 185
  извлечение зависимостей (Dependency Pull), 74
  контекстуализированный поиск зависимостей (CDL), 74
  получение зависимостей в CDL, 76
  разрешение зависимостей, 127
  ручное определение зависимостей, 127
Задача, 539
  асинхронное выполнение задач, 537; 547
    тестирование, 549
  планирование задач в Spring, 537; 538
    с использованием аннотаций, 546
    с использованием пространства имен task, 544
Запрос, 604
  собственный (native query), 385
Защита веб-приложения с помощью Spring Security, 656

И

Инверсия
  поиска (Dependency Lookup), 74
  управления (Dependency Injection), 74; 81
Инструмент
  Maven, 38
  Selenium, 728
  SpringSource Tool Suite (STS), 853
Интернационализация, 665
Интерфейс
  ApplicationContext, 53; 85; 171
  ApplicationContextAware, 155
  AppStatistics, 832; 837
  ArtworkSender, 110
  AsyncService, 547

Auditable, 398
BeanNameAware, 154
BlogPosting, 291; 465
CarRepository, 541
CarService, 542
ClassFilter, 229
CommentService, 474
ContactAuditRepository, 401
ContactAuditService, 401
ContactDao, 297; 308; 319; 354
ContactHobbyDetailMapper, 446
ContactMapper, 425
ContactRepository, 395; 485; 492; 561; 600; 648
ContactService, 373; 395; 422; 485; 559; 561; 598; 600; 822
ContactSfDao, 333
ContactSummaryService, 379
ContactTelDetailMapper, 445; 449
Container, 76
CrudRepository, 394
DisposableBean, 149
EntryRepository, 793
EntryService, 474; 792
FactoryBean, 135
FoodProviderService, 188
HousekeepingService, 474; 550; 801
InitializingBean, 143
ISimpleBean, 246
IsModified, 264
JNDI, 74
MessageProvider, 48
MessageRenderer, 48
MessageSender, 572
MessageSource, 172
MessageSourceResolvable, 176
MethodMatcher, 230
MethodReplacer, 118
NewsletterSender, 80
ObscenityFilter, 291; 797
Oracle, 79; 83
PlatformTransactionManager, 478
Pointcut, 229; 260
PropertyEditor, 136; 163
ResultSetExtractor, 316
RowMapper<T>, 314
RuleEngine, 820
RuleFactory, 822
Session, 354
SfFirstNameById, 333
SQLExceptionTranslator, 310
TaskScheduler, 539
TransactionDefinition, 479
TransactionStatus, 481
Trigger, 539
UserService, 474
Validator, 523
проверки достоверности бинов (Bean Validation API), 525
проектирование интерфейсов, 454
структура интерфейсов для введений, 262
Интерфейсы, предназначенные для типов советов Spring, 214
Исключения
обработка исключений, 309

К

Класс
AbstractBlogPosting, 465; 533; 805
AbstractControllerTest, 714
AbstractLookupDemoBean, 114
AbstractServiceImplTest, 724
AnotherContact, 516
AopContext, 245
AopNamespaceExample, 277
ApplicationConversionServiceFactoryBean, 519
AppStatisticsImpl, 832
AspectjexpBean, 241
AsyncServiceImpl, 547
AuditorAwareBean, 403
BatchSqlUpdate, 318
BeanOne, 233
BeanValidationValidator, 748
BookwormOracle, 103; 111
Car, 540
CarServiceImpl, 542
CciLocalTransactionManager, 478
ComposablePointcut, 257
ConfigurableMessageProvider, 92; 198
Contact, 296; 347; 483; 511; 560; 598
ContactAudit, 399; 409
ContactAuditServiceImpl, 401
ContactController, 579; 611; 649; 690
ContactDaoImpl, 356
ContactGrid, 650
ContactListBean, 689
Contacts, 576
ContactService, 504
ContactServiceImpl, 375; 376; 397; 487; 490; 492; 493; 496; 561; 601; 822
ContactSummary, 379
ContactTelDetail, 296; 349; 352; 429
ContactToAnotherContactConverter, 517
ContactValidator, 523
ContextLoaderListener, 83
ControlFlowPointcut, 255
ControllerTestConfig, 713
CustomCredentialsProvider, 591
Customer, 525
CustomerType, 526
DynamicPointcutExample, 236
Entry, 466
EntryController, 803

EntryServiceImpl, 793; 795
ErrorBean, 226
FileMessageToJobRequest, 755
Food, 187
FormattingConversionServiceFactoryBean, 519
FtpArtworkSender, 110
Gender, 526
GenericXmlApplicationContext, 90
GetAgeMethodMatcher, 259
GetterMethodMatcher, 259
HelloWorldMessageProvider, 48
Hobby, 350; 433
HousekeepingServiceImpl, 551; 801
HttpInvokerClientSample, 565
ImportContactJobListener, 749
IndividualCustomerValidator, 530
InjectRef, 103
InjectSimpleConfig, 101
InsertContact, 328
InsertContactTelDetail, 330
IsModifiedMixin, 265
Java-классы, 181
JdbcContactDao, 309
JdbcContactSfDao, 334
JdbcTemplate, 311
JdbcUserDao, 308
Jsr303Sample, 528
KeyGenerator, 221
ListBasedObscenityFilter, 797
LocalContainerEntityManagerFactoryBean, 369
LocalEntityManagerFactoryBean, 369
LoggingBeanExample, 155
LookupDemo, 115
MappingSqlQuery, 318; 321
MBeanExporter, 833
Message, 633
MessageDigester, 159
MessageDigestFactory, 162
MessageDigestFactoryBean, 158
MessageEventListener, 177
MessageSupportFactory, 50
MessageWriter, 208; 286
MyBatisPlugin, 795
MyBeanValidationService, 527
Name, 169
NameBean, 238
NamePropertyEditor, 170
NoOpBeforeAdvice, 247
ObscenityFilterAdvice, 799
ProfilingExample, 225
ProfilingInterceptor, 224
PropertiesBeanDefinitionReader, 83
Proxy, 245
ProxyFactory, 211
ProxyFactoryBean, 269; 270; 273; 275
ProxyFactoryBeanExample, 273
RegexpBean, 239
ReplacementTarget, 117
Rule, 820
RuleFactoryImpl, 823
SampleBean, 235; 258
ScheduleTaskSample, 545
SearchCriteria, 438
SecureBean, 215
SecurityAdvice, 217
SecurityController, 661
SecurityExample, 218
SecurityManager, 216
SelectAllContacts, 322
SelectContactByFirstName, 324
ServiceTestConfig, 723
SetterMethodMatcher, 259
ShutdownHookBean, 156
SimpleAdvice, 234
SimpleAfterReturningAdvice, 220
SimpleBean, 131; 140; 246
SimpleBeforeAdvice, 214; 254
SimpleDynamicPointcut, 235
SimpleMessageSender, 572
SimpleStaticPointcut, 233
SimpleTarget, 105
SimpleThrowsAdvice, 226
SpringJpaSample, 397
SQLExceptionTranslator, 310
SqlFunction, 318; 333
SqlUpdate, 318; 325
StandardLookupDemoBean, 114
StandardOutMessageRenderer, 49; 199
StaticPointcutExample, 234
StopWatch, 116
TargetBean, 267
TestBean, 255
TestPointcut, 247
UpdateContact, 325
UrlUtil, 633
UserInfo, 216
WeakKeyCheckAdvice, 222
WorkerBean, 223
моделирующий операции JDBC, 318
настройка DAO-классов JDBC
с использованием аннотаций, 318
сущностный (entity class), 340
Контроллер, 604
Конфигурация
Spring
для JTA, 501
для MyBatis, 420
для асинхронной задачи, 548
для отправки сообщений, 572
для планирования на основе аннотаций, 546
для редактора свойств, 511
использующая пространство имен task, 544

Spring Insight, 846
Spring MVC, 607
Spring Roo в STS, 758
Spring Security, 589; 657
Spring Web Flow и JSF, 679
VMware tc Server в STS, 869
дескриптора веб-развертывания, 785
инфраструктуры Spring Batch, 742
сервлета REST, 580

М

Механизм
Dependency Pull, 75
Модель, 603
DOM, 777
UML, 779
Модули Spring, 35
aop, 35
asm, 35
aspects, 35
beans, 35
context, 35
context.support, 36
core, 36
expression, 36
instrument, 36
instrument.tomcat, 36
jdbc, 36
jms, 36
orm, 36
oxm, 36
test, 36
transaction, 37
web, 37
web.portlet, 37
web.servlet, 37
web.struts, 36
Мозаика (tile), 70
Мониторинг
пакетных заданий Spring, 842
приложения с помощью Spring Insight, 845
статистики Hibernate, 840

Н

Навигация, 676

О

Обработка исключений, 309

П

Пакеты JDBC в Spring, 303
Параметры конфигурации, 80
Планирование задач в Spring, 537; 538
Платформа
Google Guice, 31
Seam Framework, 31
Spring, 19; 33
Платформа Spring
XML-конфигурация Spring, 199
Подключаемый модуль для Firefox
Selenium IDE, 728
Поток, 673
Представление, 603
Привязка данных, 522
Приложение
jPetStore, 44
Petclinic, 44
Petclinic Groovy, 44
SpringBlog, 58; 777
Spring Configuration Basic, 45
Spring MVC Ajax, 45
Spring MVC Basic, 45
Spring MVC Showcase, 45
Spring Petcare, 46
Spring Task and Scheduling, 45
Spring Web Flow, 677
жизненный цикл приложения, 676
защита с помощью Spring Security, 656
Проверка достоверности, 24
Программирование
аспектно-ориентированное (АОП), 23; 203
архитектура АОП в Spring, 210
динамическое, 206
статическое, 206
Проект
Spring Data, 337
Spring Security, 30
Прокси, 204; 245
CGLIB, 246; 250
JDK, 250
динамические прокси JDK, 245
тестирование производительности, 247
Пространство имен aop, 275
Профили, 187
пример Java-конфигурации, 192
пример XML-конфигурации, 190

Р

Редакторы свойств
встроенные, 168
для компонентов JavaBean, 163
встроенные, 164
конфигурирование, 166
создание, 169
Рефакторинг с помощью Spring, 53

С

Свойства, 193
Связывание (weaving), 205
Сервер
ActiveMQ, 566
установка, 567

Сервлет
REST
конфигурирование, 580
диспетчера для веб-служб REST, 581
Служба
RESTful-WS, 27
вызов службы, 565
отображение службы, 564
Событие, 177
публикация события, 178
создание класса события, 177
Совет (advice), 205; 228
создание совета в Spring, 212
стандартный, 262
типа введения, 262
Совместимость, 103
Спецификация
JSR-303, 24; 525
JSR-330, 197
Срез (pointcut), 205; 228
использование регулярных выражений для срезов, 240
компонуемые срезы, 257
потока управления, 254
расширенное использование срезов, 254
создание срезов, 260
с помощью выражения для срезов AspectJ, 241
Стандарт EJB, 475
Сценарии в Spring, 809

Т

Тестирование, 707
асинхронной задачи, 549
корпоративная инфраструктура тестирования, 710
модульное, 707; 718
отправки сообщений, 573
слушателя сообщений JMS, 570
Технология
Ajax, 27
JSF, 669
Точка соединения (joinpoint), 205; 211
Транзакция
глобальные транзакции в Spring, 500
реализация глобальных транзакций с помощью JTA, 501
декларативная, 485
программная, 485
управление транзакциями, 26

У

Упаковка Spring, 33; 35
Управление транзакциями, 26

Ф

Фабрика бинов, 84; 157
доступ к фабрике бинов напрямую, 161
Файл
aop.xml, 791
application.properties, 512
bookmarks.xml, 859
contacts.xml, 744
ContactTelDetailMapper.xml, 445
context.xml, 791
datasource-tx-jpa.xml, 562
JAR-файлы для генерации классов метамоделей, 388
jdbc.properties, 304
log4j.properties, 297; 417
root-context.xml, 563; 789
standard.properties, 620
ValidationMessages.properties, 534
web.xml, 785
Формат
JSON, 27

Ц

Цель (target), 205

Ш

Шаблон
EIP, 31
MVC, 604
проектирования Singleton, 123
страницы, 665

Э

Электронная почта
отправка, 27

Я

Язык
Expression Language (EL), 24
Groovy, 810; 815
QueryDSL, 365
Spring Expression Language (SpEL), 24; 99
Unified Expression Language, 24
запросов Hibernate (HQL), 355
запросов постоянства Java (JPQL), 365

АЛГОРИТМЫ НА JAVA
4-Е ИЗДАНИЕ

Роберт Седжвик,
Кевин Уэйн

www.williamspublishing.com

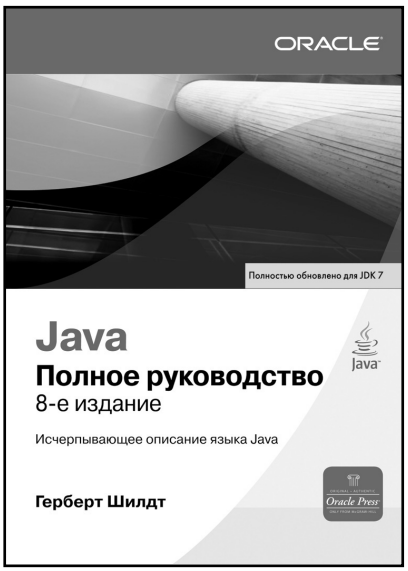
Последнее издание из серии бестселлеров Седжвика, содержащее самый важный объем знаний, наработанных за последние несколько десятилетий. Содержит полное описание структур данных и алгоритмов для сортировки, поиска, обработки графов и строк, включая пятьдесят алгоритмов, которые должен знать каждый программист. В книге представлены новые реализации, написанные на языке Java в доступном стиле модульного программирования — весь код доступен пользователю и готов к применению. Алгоритмы изучаются в контексте важных научных, технических и коммерческих приложений. Клиентские программы и алгоритмы записаны в реальном коде, а не на псевдокоде, как во многих других книгах. Дается вывод точных оценок производительности на основе соответствующих математических моделей и эмпирических тестов, подтверждающих эти модели.

ISBN 978-5-8459-1781-2

в продаже

JAVA
ПОЛНОЕ РУКОВОДСТВО
8-е издание

Герберт Шилдт



www.williamspublishing.com

Эта книга предназначена для всех программистов — как для новичков, так и для профессионалов. Начинающий программист найдет в ней подробные пошаговые описания и множество чрезвычайно полезных примеров. А углубленное рассмотрение более сложных функций и библиотек Java должно удовлетворить ожидания профессиональных программистов. Для обеих категорий читателей в книге указаны действующие ресурсы и полезные ссылки.

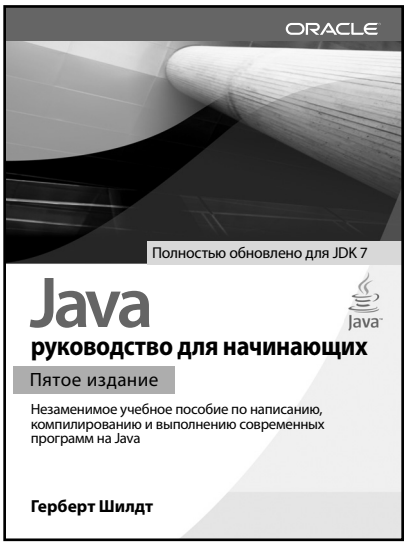
В этой книге автор бестселлеров по программированию Герберт Шилдт знакомит вас со всем необходимым для разработки, компиляции, отладки и запуска Java-программ. Полностью обновленное для платформы Java Platform, Standard Edition 7 (Java SE 7), это исчерпывающее издание рассматривает язык Java в целом, включая его синтаксис, ключевые слова и фундаментальные принципы программирования. Здесь вы найдете информацию о ключевых элементах библиотеки Java API, рассмотрите JavaBeans, сервлеты, аплеты и Swing и ознакомитесь с работой Java в реальных ситуациях. Кроме того, в этой книге подробно обсуждаются такие новые средства Java SE 7, как оператор `try`-с-ресурсами, строки в операторе `switch`, выведение типов с оператором <>, NIO.2 и Fork/Join Framework.

ISBN 978-5-8459-1759-1

в продаже

JAVA
РУКОВОДСТВО ДЛЯ НАЧИНАЮЩИХ
ПЯТОЕ ИЗДАНИЕ

Герберт Шилдт



www.williamspublishing.com

В этом учебном пособии для начинающих программировать на Java подробно рассмотрены все основные средства данного языка программирования: типы данных, операторы, циклы, классы, интерфейсы, методы, исключения, обобщения, пакеты, основные библиотеки классов, средства многопоточного программирования, потоки ввода-вывода, перечисления, апплеты и документирующие комментарии. Применение всех этих языковых средств Java на практике наглядно демонстрируется в небольших проектах для самостоятельного опробования. Книга снабжена массой полезных советов авторитетного автора и множеством примеров программ с подробными комментариями, благодаря которым они становятся понятными любому читателю независимо от уровня его подготовки. А для проверки прочности приобретенных знаний и навыков в конце каждой главы приводятся контрольные вопросы и задания. Книга рассчитана на широкий круг читателей, интересующихся программированием на Java.

ISBN 978-5-8459-1770-6

в продаже