LINQ

Язык интегрированных запросов в C# 2010

ДЛЯ ПРОФЕССИОНАЛОВ

LINQ_title.indd 1 24.01.2011 19:01:29

Pro LINQ

Language Integrated Query in C# 2010

Adam Freeman and
Joseph C. Rattz, Jr.

Apress®

LINQ

Язык интегрированных запросов в C# 2010

ДЛЯ ПРОФЕССИОНАЛОВ

Адам Фримен
Джозеф Раттц-мл.

ВИЛЬЯМС

Москва • Санкт-Петербург • Киев
2011

ББК 32.973.26-018.2.75
Ф88
УДК 681.3.07

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

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

Фримен, Адам, **Раттц-мл.**, Джозеф С.

Ф88 LINQ: язык интегрированных запросов в C# 2010 для профессионалов. : Пер. с англ. — М. : ООО “И.Д. Вильямс”, 2011. — 656 с. : ил. — Парал. тит. англ.

ISBN 978-5-8459-1701-0 (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, Copyright © 2010 by Adam Freeman and Joseph C. Rattz, Jr.

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 © 2011.

Научно-популярное издание
Адам Фримен, Джозеф С. Раттц-мл.

LINQ: язык интегрированных запросов в C# 2010 для профессионалов

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

Подписано в печать 24.01.2011. Формат 70×100/16.
Гарнитура Times. Печать офсетная.
Усл. печ. л. 52,89. Уч.-изд. л. 37,6.
Тираж 1500 экз. Заказ № 0000.

Отпечатано по технологии CtP
в ОАО “Печатный двор” им. А. М. Горького
197110, Санкт-Петербург, Чкаловский пр., 15.

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

ISBN 978-5-8459-1701-0 (рус.)
ISBN 978-1430-22653-6 (англ.)

© Издательский дом “Вильямс”, 2011
© by Adam Freeman and Joseph C. Rattz, Jr., 2010

Оглавление

Часть I. LINQ: язык интегрированных запросов в C# 2010 17

Глава 1. Знакомство с LINQ 18

Глава 2. Расширения языка C# для LINQ 32

Часть II. LINQ to Objects 61

Глава 3. Введение в LINQ to Objects 62

Глава 4. Отложенные операции 71

Глава 5. Не отложенные операции 135

Часть III. LINQ to XML 181

Глава 6. Введение в LINQ to XML 182

Глава 7. Интерфейс LINQ to XML API 187

Глава 8. Операции LINQ to XML 260

Глава 9. Дополнительные возможности LINQ to XML 285

Часть IV. LINQ to DataSet 319

Глава 10. Операции LINQ to DataSet 320

Глава 11. Дополнительные возможности LINQ to DataSet 350

Часть V. LINQ to SQL 357

Глава 12. Введение в LINQ to SQL 358

Глава 13. Советы и инструменты, связанные с LINQ to SQL 368

Глава 14. Операции LINQ to SQL для баз данных 389

Глава 15. Сущностные классы LINQ to SQL 428

Глава 16. Класс `DataContext` в LINQ to SQL 469

Глава 17. Конфликты параллельного доступа LINQ to SQL 522

Глава 18. Дополнительные возможности LINQ to SQL 539

Часть VI. LINQ to Entities 549

Глава 19. Введение в LINQ to Entities 550

Глава 20. Операции LINQ to Entities 559

Глава 21. Классы LINQ to Entities 593

Часть VII. Parallel LINQ 613

Глава 22. Введение в Parallel LINQ 614

Глава 23. Использование Parallel LINQ 619

Глава 24. Операции Parallel LINQ 631

Предметный указатель 647

Содержание

Об авторах 15
О техническом рецензенте 15
Благодарности 16
От издательства 16
Исходный код примеров 16

Часть I. LINQ: язык интегрированных запросов в C# 2010 17

Глава 1. Знакомство с LINQ 18

Сдвиг парадигмы 18
Запрос к XML 19
Запрос к базе данных SQL Server 20
Появление LINQ 21
LINQ поддерживает запросы данных 21
Как получить LINQ 22
LINQ предназначен не только для запросов 23
Советы начинающим 25
Используйте ключевое слово `var`, когда запутались 26
Используйте операции `Cast` или `OfType` для унаследованных коллекций 27
Отдавайте предпочтение операции `OfType` перед `Cast` 28
Не рассчитывайте на безошибочность запросов 28
Используйте преимущество отложенных запросов 29
Используйте свойство `Log` из `DataContext` 30
Используйте форум LINQ 31
Резюме 31

Глава 2. Расширения языка C# для LINQ 32

Дополнения языка C# 32
Лямбда-выражения 32
Деревья выражений 37
Ключевое слово `var`, инициализация объектов и анонимные типы 38
Расширяющие методы 43
Частичные методы 47
Выражения запросов 49
Резюме 58

Часть II. LINQ to Objects 61

Глава 3. Введение в LINQ to Objects 62

Обзор LINQ to Objects 62
Интерфейс `IEnumerable<T>`, последовательности и стандартные операции запросов 63
Возврат `IEnumerable<T>`, выдача и отложенные запросы 64
Делегаты `Func` 67
Алфавитный указатель стандартных операций запросов 68
История о двух синтаксисах 69
Резюме 70

Глава 4. Отложенные операции 71

Необходимые пространства имен 71
Необходимые сборки 71
Общие классы 71
Организация отложенных операций по назначению 73
Ограничение 73
Проекция 75
Разбиение 83
Конкатенация 90
Упорядочивание 92
Соединение 107
Группирование 111
Множества 116
Преобразование 121
Элемент 127
Генерация 131
Резюме 134

Глава 5. Не отложенные операции 135

Необходимые пространства имен 135
Общие классы 135
Организация не отложенных операций по назначению 138
Преобразование 138
Эквивалентность 149
Элемент 151
Квантификаторы 163
Агрегация 168
Резюме 180

Часть III. LINQ to XML 181

Глава 6. Введение в LINQ to XML 182

Введение 184
Обман W3C DOM XML API 184
Резюме 186

Глава 7. Интерфейс LINQ to XML API 187

Необходимые пространства имен 187
Значимые проектные усовершенствования API-интерфейса 187
Конструирование деревьев XML было упрощено с помощью функционального конструирования 188
Центральная роль элемента вместо документа 190
Имена, пространства имен и префиксы 191
Извлечение значения узла 194
Объектная модель LINQ to XML 196
Отложенное выполнение запросов, удаление узлов и “проблема Хэллоуина” 197
Создание XML 199
Создание элементов с помощью `XElement` 200
Создание атрибутов с помощью `XAttribute` 202
Создание комментариев с помощью `XComment` 203

Создание контейнеров с помощью XContainer 203
Создание объявлений с помощью XDeclaration 204
Создание типов документов с помощью XDocumentType 204
Создание документов с помощью XDocument 206
Создание имен с помощью XName 206
Создание пространств имен с помощью XNamespace 207
Создание узлов с помощью XNode 207
Создание инструкций обработки с помощью XProcessingInstruction 207
Создание потоковых элементов с помощью XStreamingElement 209
Создание текста с помощью XText 210
Создание CData с помощью XCData 211
Вывод XML 211
Сохранение с помощью XDocument.Save() 211
Сохранение с помощью XElement.Save() 212
Ввод XML 213
Загрузка с помощью XDocument.Load() 213
Загрузка с помощью XElement.Load() 215
Разбор содержимого с помощью методов XDocument.Parse() или XElement.Parse() 215
Обход XML 216
Свойства обхода 217
Обход вперед с помощью XNode.NextNode 217
Обход назад с помощью XNode.PreviousNode 218
Методы обхода 220
Модификация XML 232
Добавление узлов 233
Удаление узлов 237
Обновление узлов 239
Вызов XElement.SetElementValue() на дочерних объектах XElement 242
Атрибуты XML 244
Создание атрибута 244
Обход атрибутов 244
Модификация атрибутов 247
Аннотации XML 251
Добавление аннотаций с помощью XObject.AddAnnotation() 251
Обращение к аннотациям с помощью XObject.Annotation() или XObject.Annotations() 251
Удаление аннотаций с помощью XObject.RemoveAnnotations() 251
Пример аннотаций 251
События XML 254
XObject.Changing 255
XObject.Changed 255
Несколько примеров событий 255
Трюк, забава или неопределенность? 258
Резюме 259

Глава 8. Операции LINQ to XML 260

Введение в операции LINQ to XML 260
Ancestors 261
Прототипы 261
Примеры 261

AncestorsAndSelf 265
Прототипы 265
Примеры 265
Attributes 267
Прототипы 267
Примеры 267
DescendantNodes 269
Прототипы 269
Примеры 269
DescendantNodesAndSelf 270
Прототипы 271
Примеры 271
Descendants 272
Прототипы 272
Примеры 272
DescendantsAndSelf 274
Прототипы 274
Примеры 274
Elements 276
Прототипы 276
Примеры 277
InDocumentOrder 278
Прототипы 278
Примеры 279
Nodes 280
Прототипы 280
Примеры 280
Remove 281
Прототипы 281
Примеры 282
Резюме 284

Глава 9. Дополнительные возможности LINQ to XML 285

Необходимые пространства имен 285
Запросы 286
Отсутствие иерархического спуска 286
Сложный запрос 288
Трансформации 293
Трансформации с использованием XSLT 294
Трансформация с использованием функционального конструирования 295
Советы 298
Проверка достоверности 303
Расширяющие методы 303
Прототипы 303
Получение схемы XML 304
Примеры 306
XPath 316
Прототипы 316
Примеры 316
Резюме 317

Часть IV. LINQ to DataSet 319

Глава 10. Операции LINQ to DataSet 320

Необходимые сборки 321
Необходимые пространства имен 321
Общий код для примеров 321
Операции множеств `DataRow` 322
`Distinct` 323
`Except` 326
`Intersect` 328
`Union` 330
`SequenceEqual` 331
Операции над полями `DataRow` 333
`Field<T>` 336
`SetField<T>` 341
Операции `DataTable` 343
`AsEnumerable` 344
`CopyToDataTable<DataRow>` 344
Примеры 345
Резюме 349

Глава 11. Дополнительные возможности LINQ to DataSet 350

Необходимые пространства имен 350
Типизированные `DataSet` 350
Собираем все вместе 352
Резюме 355

Часть V. LINQ to SQL 357

Глава 12. Введение в LINQ to SQL 358

Введение в LINQ to SQL 359
`DataContext` 360
Сущностные классы 361
Ассоциации 361
Обнаружение конфликтов параллельного доступа 362
Разрешение конфликтов параллельного доступа 362
Предварительные условия для запуска примеров 363
Получение соответствующей версии базы данных Northwind 363
Генерация сущностных классов Northwind 363
Генерация XML-файла отображения Northwind 364
Использование LINQ to SQL 364
`IQueryable<T>` 365
Некоторые общие методы 365
`GetStringFromDb()` 365
`ExecuteStatementInDb()` 366
Резюме 367

Глава 13. Советы и инструменты, связанные с LINQ to SQL 368

Введение 368
Советы 368
Используйте свойство `DataContext.Log` 369

Используйте метод GetChangeSet() 370
Подумайте об использовании частичных классов или файлов отображения 370
Подумайте об использовании частичных методов 370
Инструменты 370
SQLMetal 370
Object Relational Designer 376
Совместное использование SQLMetal и Object Relational Designer 387
Резюме 388

Глава 14. Операции LINQ to SQL для баз данных 389

Предварительные условия для запуска примеров 389
Некоторые общие методы 389
Использование API-интерфейса LINQ to SQL 390
Стандартные операции для баз данных 390
Вставки 390
Запросы 393
Обновления 415
Удаления 418
Переопределение операторов модификации базы данных 421
Переопределение метода Insert 421
Переопределение метода Update 421
Переопределение метода Delete 422
Пример 422
Переопределение в Object Relational Designer 424
Соображения 424
Трансляция SQL 424
Резюме 426

Глава 15. Сущностные классы LINQ to SQL 428

Предварительные условия для запуска примеров 428
Сущностные классы 428
Создание сущностных классов 428
XML-схема внешнего файла отображения 456
Сравнение проекций на сущностные и на несущностные классы 456
Расширение сущностных классов с помощью частичных методов 461
Важные классы API-интерфейса System.Data.Linq 462
EntitySet<T> 463
EntityRef<T> 463
Table<T> 465
IExecuteResult 465
ISingleResult<T> 466
IMultipleResults 467
Резюме 468

Глава 16. Класс DataContext в LINQ to SQL 469

Предварительные условия для запуска примеров 469
Некоторые общие методы 469
Использование API-интерфейса LINQ to SQL 469
Класс [Your]DataContext 469
Класс DataContext 470
Класс DataContext реализует интерфейс IDisposable 472

Основное назначение 473
Время жизни контекста данных 478
`DataContext()` и `[Your]DataContext()` 479
`SubmitChanges()` 490
`DatabaseExists()` 496
`CreateDatabase()` 496
`DeleteDatabase()` 498
`CreateMethodCallQuery()` 498
`ExecuteQuery()` 500
`Translate()` 502
`ExecuteCommand()` 503
`ExecuteMethodCall()` 505
`GetCommand()` 511
`GetChangeSet()` 512
`GetTable()` 514
`Refresh()` 515
Резюме 521

Глава 17. Конфликты параллельного доступа LINQ to SQL 522

Предварительные условия для запуска примеров 522
Некоторые общие методы 522
Использование API-интерфейса LINQ to SQL 522
Конфликты параллелизма 522
Оптимистический параллелизм 523
Пессимистический параллелизм 533
Альтернативный подход для средних звеньев и серверов 535
Резюме 538

Глава 18. Дополнительные возможности LINQ to SQL 539

Предварительные условия для запуска примеров 539
Использование API-интерфейса LINQ to SQL 539
Использование API-интерфейса LINQ to XML 539
Представления базы данных 539
Наследование сущностных классов 541
Транзакции 546
Резюме 548

Часть VI. LINQ to Entities 549

Глава 19. Введение в LINQ to Entities 550

Введение 551
`ObjectContext` 552
Сущностные классы 552
Ассоциации 553
Предварительные условия для запуска примеров 553
Получение соответствующей версии базы данных Northwind 553
Генерация сущностной модели данных Northwind 553
Использование API-интерфейса LINQ to Entities 556
`IQueryable<T>` 556
Некоторые общие методы 556
`GetStringFromDo()` 557

ExecuteStatementInDb() 557
Резюме 558

Глава 20. Операции LINQ to Entities 559

Предварительные условия для запуска примеров 559
Некоторые общие методы 559
Стандартные операции базы данных 559
Вставки 560
Запросы 565
Обновления 578
Удаления 579
Удаление связанных объектов 581
Управление параллельным доступом 586
Включение проверок параллелизма 587
Обработка конфликтов параллелизма 588
Резюме 592

Глава 21. Классы LINQ to Entities 593

Предварительные условия для запуска примеров 593
Класс ObjectContext 593
Конструктор 594
DatabaseExists() 595
DeleteDatabase() 596
CreateDatabase() 596
SaveChanges() 596
Refresh() 597
AddObject() 598
CreateObject() 599
DeleteObject() 600
EntityObject 600
Конструктор 600
Фабричный метод 601
Примитивные свойства 603
Навигационные свойства 604
EntityReference 606
Load() 606
Value 607
EntityCollection 607
Add() 607
Remove() 609
Clear() 610
Contains() 610
Load() 611
Count 611
Резюме 612

Часть VII. Parallel LINQ 613

Глава 22. Введение в Parallel LINQ 614

Введение 614
Parallel LINQ предназначен для объектов 617

Использование API-интерфейса Parallel LINQ 618
Резюме 618

Глава 23. Использование Parallel LINQ 619

Создание запроса Parallel LINQ 619
Предохранение порядка результатов 621
Управление параллелизмом 624
Принудительное параллельное выполнение 624
Ограничение степени параллелизма 625
Обработка исключений 625
Запросы без результатов 628
Создание диапазонов и повторов 630
Резюме 630

Глава 24. Операции Parallel LINQ 631

Операции создания `ParallelQuery` 631
`AsParallel` 631
`Range` 634
`Repeat` 635
`Empty` 636
Операции управления выполнением 636
`AsOrdered` 636
`AsUnordered` 638
`AsSequential` 639
`AsEnumerable` 640
`WithDegreeOfParallelism` 641
Прототипы 641
`WithExecutionMode` 641
`WithMergeOptions` 642
Операции преобразования 644
`Cast` 644
`OfType` 645
Операция `ForAll` 646
Прототипы 646
Примеры 646
Резюме 646

Предметный указатель 647

Об авторах

Адам Фримен — профессионал в области информационных технологий, который занимал ведущие должности в ряде компаний, последняя из которых — директор по развитию технологий и главный операционной директор в глобальном банке. Он написал несколько книг по Java и .NET и давно интересуется параллельными вычислениями.

Джозеф Раттц-мл. неосознанно начал свою карьеру в разработке программного обеспечения в 1990 г., когда друг попросил его помочь в написании текстового редактора ANSI под названием ANSI Master для компьютера Commodore Amiga. Вскоре за этим последовала игра в палача (The Gallows — "Виселица"). От этих программ, написанных на компилируемом Basic, он перешел к программированию на C, в поисках более высокой скорости и мощи. После этого Джо разрабатывал приложения для *JumpDisk* — журнала Amiga на дисках, а также для журнала *Amiga World*. Из-за того, что ему пришлось работать в маленьком городе в относительной изоляции, Джо изучил все неправильные способы написания кода. Это обучение происходило при попытках усовершенствовать плохо написанные приложения, в процессе которых он осознал важность написания хорошо сопровождаемого кода. Впервые познакомившись с отладчиком уровня исходного кода, он влюбился в него с первого взгляда.

Двумя годами позже Джо получил свою первую работу в качестве разработчика программного обеспечения в Policy Management Systems Corporation, как программист начального уровня, разрабатывающий клиент-серверное приложение системы страхования для OS/2 и Presentation Manager. С годами он добавил к своему багажу знаний C++, Unix, Java, ASP, ASP.NET, C#, HTML, DHTML и XML, разрабатывая приложения для SCT, DocuCorp, IBM, Комитета по проведению олимпийских игр в Атланте, CheckFree, NCR, EDS, Delta Technology, Radiant Systems и Genuine Parts Company. Джо нравились творческие аспекты дизайна пользовательского интерфейса, и он осознал необходимость дисциплины при разработке программного обеспечения серверной стороны. Но когда у него была такая возможность, его любимым времяпрепровождением была отладка кода.

Сегодня Джо можно найти в Genuine Parts Company — родительской компании NAPA — в подразделении Automotive Parts Group Information Systems, где он трудится над своим детищем — веб-сайтом Storefront. Этот сайт обслуживает хранилища NAPA, предоставляя их счета и данные в сети систем AS/400. Связаться к Джо можно через его веб-сайт по адресу `www.linqdev.com`.

О техническом рецензенте

Фабио Клаудио Феррачати — старший консультант и аналитик-разработчик, имеющий дело с технологиями Microsoft. Он работает в компании Brain Force (`www.brainforce.com`), в ее итальянском подразделении (`www.brainforce.it`). Является сертифицированным разработчиком решений Microsoft для .NET, сертифицированным разработчиком приложений Microsoft для .NET, сертифицированным профессионалом Microsoft, а также плодовитым автором и техническим рецензентом. В течение последних десяти лет он написал множество статей для итальянских и международных изданий, а также является соавтором более десятка книг на различные компьютерные темы.

Благодарности

Мы хотели бы поблагодарить Джона Скита (Jon Skeet), Джадсона Уайта (Judson White) и всех остальных сотрудников издательства Apress за совместную работу над этой книгой. В частности, благодарим Мери Тобин (Mary Tobin) за отслеживание работ и Эвана Бакингема (Ewan Buckingham) за рудактирование книги. Мы также выражаем благодарность Ким Уимпсетт (Kim Wimpsett) и Фабио Феррачати (Fabio Ferracchiati) за их усилия, которые сделали эту книгу намного лучше, чем она была в начале.

Адам Фримен и Джозеф Раттц-мл.

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

Исходный код примеров

Исходный код примеров, рассмотренных в книге, доступен для загрузки на сайте издательства по адресу `http://www.williamspublishing.com`.

ЧАСТЬ I

LINQ: язык интегрированных запросов в C# 2010

В этой части

Глава 1. Знакомство с LINQ

Глава 2. Расширения языка C# для LINQ

ГЛАВА 1

Знакомство с LINQ

Листинг 1.1. Программа “Hello LINQ”

```
using System;
using System.Linq;
string[] greetings = {"hello world", "hello LINQ", "hello Apress"};
var items =
  from s in greetings
  where s.EndsWith("LINQ")
  select s;
foreach (var item in items)
  Console.WriteLine(item);
```

На заметку! Код из листинга 1.1 был добавлен в проект, созданный с помощью шаблона консольного приложения в Visual Studio 2010. Если это еще не сделано, добавьте директиву `using` для пространства имен `System.Linq`.

Запуск приведенного выше кода по нажатию <Ctrl+F5> выдаст следующий вывод в окно консоли:

```
hello LINQ
```

Сдвиг парадигмы

Вы почувствовали только что, как *ваш* мир сдвинулся с места? Как разработчик .NET, вы должны были это почувствовать. То, что продемонстрировано в тривиальном примере программы из листинга 1.1, похоже на запрос на языке структурированных запросов (Structured Query Language — SQL) к массиву строк[1]. Взгляните на конструкцию `where`. Она выглядит так, будто использовался метод `EndsWith` объекта `string`, потому что так оно и есть. Может возникнуть вопрос: а как насчет типа переменной `var`? Выполняет ли по-прежнему компилятор C# контроль типов? Ответ — да, он проверяет статически типы во время компиляции. Какое средство или средства C# позволяют все это? Ответ: Microsoft Language Integrated Query (язык интегрированных запросов Microsoft), иначе называемый *LINQ*.

[1] Важно отметить, что порядок следования конструкций в запросе противоположен типичному SQL. К тому же добавлена часть `s in` запроса, которая представляет ссылку на набор элементов, содержащихся в источнике, которым в данном случае является массив строк `"hello world"`, `"hello LINQ"` и `"hello Apress"`.

Запрос к XML

В то время как пример из листинга 1.1 достаточно тривиален, пример в листинге 1.2 начинает отражать потенциальную мощь, которую вручает LINQ в руки разработчика .NET. Он демонстрирует легкость, с которой можно взаимодействовать и опрашивать данные XML (Extensible Markup Language — расширяемый язык разметки) с помощью API-интерфейса LINQ to XML. Обратите внимание, как на основе данных XML конструируется объект по имени `books`, с которым впоследствии можно взаимодействовать программно.

Листинг 1.2. Простой запрос к XML-разметке с использованием LINQ to XML

```
using System;
using System.Linq;
using System.Xml.Linq;
XElement books = XElement.Parse(
  @"<books>
    <book>
      <title>Pro LINQ: Language Integrated Query in C# 2010</title>
      <author>Joe Rattz</author>
    </book>
    <book>
      <title>Pro .NET 4.0 Parallel Programming in C#</title>
      <author>Adam Freeman</author>
    </book>
    <book>
      <title>Pro VB 2010 and the .NET 4.0 Platform</title>
      <author>Andrew Troelsen</author>
    </book>
  </books>");
var titles =
  from book in books.Elements("book")
  where (string) book.Element("author") == "Joe Rattz"
  select book.Element("title");
foreach(var title in titles)
  Console.WriteLine(title.Value);
```

На заметку! Код в листинге 1.2 требует добавления к ссылкам проекта сборки `System.Xml.Linq.dll`, если это еще не сделано. Также обратите внимание, что добавлена директива `using` для пространства имен `System.Xml.Linq`.

Запуск предыдущего кода нажатием <Ctrl+F5> приводит к выводу следующих данных в окно консоли:

```
Pro LINQ: Language Integrated Query in C# 2010
```

Обратите внимание на то, как данные XML были разобраны для помещения в объект `XElement`. Объект `XmlDocument` нигде не создавался. Среди преимуществ LINQ to XML — расширения, которые он привносит в XML API. Теперь вместо того, чтобы сосредоточивать все вокруг `XmlDocument`, как того требует W3C Document Object Model (DOM) XML API, интерфейс LINQ to XML позволяет разработчику взаимодействовать на уровне элемента, используя класс `XElement`.

На заметку! В дополнение к средствам запросов, LINQ to XML предоставляет более мощный и простой способ использования интерфейса для работы с данными XML.

Здесь также применялся SQL-подобный синтаксис для опроса данных XML, как если бы это была база данных.

Запрос к базе данных SQL Server

В следующем примере демонстрирует использование LINQ to SQL для опроса таблиц базы данных. В коде из листинга 1.3 выполняется запрос к стандартной базе данных примеров Microsoft Northwind.

Листинг 1.3. Простой запрос XML с использованием LINQ to XML

```
using System.Linq;
using System.Data.Linq;
using nwind;
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
var custs =
  from c in db.Customers
  where c.City == "Rio de Janeiro"
  select c;
foreach (var cust in custs)
  Console.WriteLine("{0}", cust.CompanyName);
```

На заметку! Код в листинге 1.3 требует добавления сборки `System.Data.Linq.dll` к списку ссылок проекта, если это не сделано ранее. Также обратите внимание, что добавлена директива `using` для пространства имен `System.Xml.Linq`.

В коде была добавлена директива `using` для пространства имен `nwind`. В этом примере должна использоваться утилита командной строки SQLMetal или Object Relational Designer, чтобы сгенерировать сущностные классы для целевой базы данных, которой в данном случае является база данных примеров Microsoft Northwind. В главе 12 показано, как это делать с SQLMetal. Сгенерированные сущностные классы создаются в пространстве имен `nwind`, которое указано при их генерации. После этого в проект добавляется сгенерированный SQLMetal исходный модуль, а также директива `using` для пространства имен `nwind`.

На заметку! Для корректной установки соединения может понадобиться изменить строку соединения, переданную конструктору Northwind в листинге 1.3. В разделе "`DataContext()` и `[Your] DataContext()`" главы 16 описаны разные способы подключения к базе данных.

Запуск предыдущего кода нажатием <Ctrl+F5> приводит к выводу следующих данных в окно консоли:

```
Hanari Carnes
Que Delícia
Ricardo Adocicados
```

В этом примере демонстрируется опрос таблицы `Customers` из базы данных Northwind на предмет списка заказчиков из Рио-де-Жанейро (Rio de Janeiro). Хотя может показаться, что здесь не происходит ничего нового или особенного, чего нельзя было бы получить существующими средствами, все же есть серьезные отличия. Важнее всего то, что этот запрос интегрирован в язык, а это значит, что получается поддержка уровня языка, включающая проверку синтаксиса и средство IntelliSense. Ушли в прошлое те дни, когда запрос SQL записывался в строку, и ошибку невозможно было обнаружить вплоть до выполнения кода. Вы хотите сделать конструкцию `where` зависящей от поля

таблицы `Customers`, но не помните имени этого поля? Средство IntelliSense покажет поля таблицы. Как только вы введете `c.` в предыдущем примере, IntelliSense отобразит все поля таблицы `Customers`.

Во всех предыдущих запросах используется синтаксис *выражений запросов*. В главе 12 вы узнаете, что существуют два синтаксиса запросов LINQ, один из которых — синтаксис выражений запросов. Конечно, можно также применять привычный для C# синтаксис *стандартной точечной нотации*. Этот синтаксис предлагает нормальный шаблон вызовов `объект.метод()`.

Появление LINQ

По мере становления платформы .NET Framework и поддерживаемых ею языков C# и VB, стало ясно, что одной из наиболее проблемных областей для разработчиков остается доступ к данным из разных источников. В частности, доступ к базе данных и манипуляции XML часто в лучшем случае запутаны, а в худшем — проблематичны.

Проблемы, связанные с базами данных, многочисленны. Первая сложность в том, что нельзя программно взаимодействовать с базой данных на уровне естественного языка. Это приводит к синтаксическим ошибкам, которые не проявляются вплоть до момента запуска. Неправильные ссылки на поля базы данных тоже не обнаруживаются. Это может пагубно отразиться на программе, особенно если произойдет во время выполнения кода обработки ошибок. Нет ничего хуже, чем сбой механизма обработки ошибок из-за синтаксически неверного кода, который никогда не тестировался. Иногда это неизбежно из-за непредсказуемого поведения ошибки. Наличие кода базы данных, который не проверяется во время компиляции, определенно может привести к этой проблеме.

Вторая проблема связана с неудобством, которое вызвано различными типами данных, используемыми определенным доменом данных, например, разница между типами базы данных или типами XML и типами данных в языке, на котором написана программа. В частности, серьезные сложности могут вызывать типы времени и даты.

Разбор, итерация и манипулирование XML-разметкой могут быть достаточно утомительными. Часто фрагмент XML — это все, что нужно, но из-за требований интерфейса W3C DOM XML API объект `XmlDocument` должен быть обязательно создан, чтобы выполнять различные операции над фрагментом XML.

Вместо того чтобы просто добавить больше классов и методов для постепенного восполнения этих недостатков, в Microsoft решили пойти на один шаг дальше в абстрагировании основ запросов данных из этих конкретных доменов данных. В результате появился LINQ — технология Microsoft, предназначенная для поддержки запросов к данным всех типов на уровне языка. Эти типы включают массивы и коллекции в памяти, базы данных, документы XML и многое другое.

LINQ поддерживает запросы данных

По большей части LINQ ориентирован на запросы — будь то запросы, возвращающие набор подходящих объектов, единственный объект или подмножество полей из объекта либо набора объектов. В LINQ этот возвращенный набор объектов называется *последовательностью* (sequence). Большинство последовательностей LINQ имеют тип `IEnumerable<T>`, где `T` — тип данных объектов, находящихся в последовательности. Например, если есть последовательность целых чисел, они должны храниться в переменной типа `IEnumerable<int>`. Вы увидите, что `IEnumerable<T>` буквально господствует в LINQ. Очень многие методы LINQ возвращают `IEnumerable<T>`.

LINQ to Objects

LINQ to Objects — название, данное API-интерфейсу `IEnumerable<T>` для стандартных операций запросов (Standard Query Operators). Именно LINQ to Objects позволяет выполнять запросы к массивам и находящимся в памяти коллекциям данных. Стандартные операции запросов — это статические методы класса `System.Linq.Enumerable`, которые используются для создания запросов LINQ to Objects.

LINQ to XML

LINQ to XML — название, назначенное API-интерфейсу LINQ, который ориентирован на работу с XML. В Microsoft не только добавили необходимые библиотеки XML для работы с LINQ, но также восполнили недостатки стандартной модели XML DOM, существенно облегчив работу с XML. Прошли времена, когда нужно было создавать `XmlDocument` только для того, чтобы поработать с небольшим фрагментом XML-кода. Чтобы воспользоваться преимуществами LINQ to XML, в проект понадобится добавить ссылку на сборку `System.Xml.Linq.dll` и директиву `using` следующего вида:

```
using System.Xml.Linq;
```

LINQ to DataSet

LINQ to DataSet — название, данное API-интерфейсу LINQ, который предназначен для работы с `DataSet`. У многих разработчиков есть масса кода, полагающегося на `DataSet`. Те, кто не хотят отставать от новых веяний, но и не готовы переписывать свой код, благодаря этому интерфейсу могут воспользоваться всей мощью LINQ.

LINQ to SQL

LINQ to SQL — наименование, присвоенное API-интерфейсу `IQueryable<T>`, который позволяет запросам LINQ работать с базой данных Microsoft SQL Server. Чтобы воспользоваться преимуществами LINQ to SQL, в проект понадобится добавить ссылку на сборку `System.Data.Linq.dll`, а также следующую директиву `using`:

```
using System.Data.Linq;
```

LINQ to Entities

LINQ to Entities — альтернативный API-интерфейс LINQ, используемый для обращения к базе данных. Он отделяет сущностную объектную модель от физической базы данных, вводя логическое отображение между ними двумя. С таким отделением возрастает мощь и гибкость, но также растет и сложность. Поскольку LINQ to Entities не входит в состав ядра LINQ, в настоящей книге он не рассматривается. Если нужна более высокая гибкость, чем обеспечивается LINQ to SQL, имеет смысл рассмотреть эту альтернативу. В частности, когда необходимо ослабить связь между сущностной объектной моделью и базой данных, если сущностные объекты конструируются из нескольких таблиц или требуется большая гибкость в моделировании сущностных объектов, то в этом случае LINQ to Entities может стать оптимальным выбором.

Как получить LINQ

Формально отдельного продукта LINQ, который нужно было бы получать отдельно, не существует. LINQ полностью интегрирован в .NET Framework, начиная с версии 3.5 и Visual Studio 2008. В .NET 4.0 и Visual Studio 2010 добавлена поддержка средств Parallel LINQ, которые рассматриваются в главах 22–24.

LINQ предназначен не только для запросов

Может показаться, что LINQ — это нечто, связанное только с запросами, поскольку расшифровывается как *язык интегрированных запросов* (Language Integrated Query). Однако не думайте о нем лишь в этом контексте. Предпочтительнее воспринимать LINQ как механизм итерации данных (data iteration engine), но возможно в Microsoft не захотели обозначать эту технологию аббревиатурой DIE ("умереть").

Приходилось ли вам когда-нибудь вызывать метод, возвращающий данные в структуре, которую затем приходилось преобразовывать в еще одну структуру данных, прежде чем передать другому методу? Предположим, например, что вызывается метод A, и этот метод возвращает массив типа `string`, содержащий числовые значения в виде строк. Затем нужно вызвать метод B, но метод B требует массива целых чисел. Обычно приходится организовывать цикл для прохода по массиву строк и наполнения вновь сконструированного массива целых чисел. Давайте рассмотрим краткий пример мощи Microsoft LINQ.

Предположим, что имеется массив строк, которые приняты от метода A, как показано в листинге 1.4.

Листинг 1.4. Преобразование массива строк в массив целых

```
string[] numbers = { "0042", "010", "9", "27" };
```

В этом примере объявлен статический массив строк. Теперь перед вызовом метода B нужно преобразовать массив строк в массив целых чисел:

```
int[] nums = numbers.Select(s => Int32.Parse(s)).ToArray();
```

Вот и все. Что может быть проще? А вот код, необходимый для отображения результирующего массива целых чисел:

```
foreach(int num in nums)
  Console.WriteLine(num);
```

Вывод выглядит следующим образом:

```
42
10
9
27
```

Возможно, вы подумали, что просто в строках отброшены ведущие пробелы. Но убедит ли вас, если отсортировать результат? Если бы это были по-прежнему строки, то 9 окажется в конце, а 10 — в начале. В листинге 1.5 приведен код, который выполняет преобразование и сортирует вывод.

Листинг 1.5. Преобразование массива строк в массив целых чисел с последующей сортировкой

```
string[] numbers = { "0042", "010", "9", "27" };
int[] nums = numbers.Select(s => Int32.Parse(s)).OrderBy(s => s).ToArray();
foreach(int num in nums)
  Console.WriteLine(num);
```

И вот результат:

```
9
10
27
42
```

Не правда ли гладко? Вы можете возразить, что все это прекрасно, но пример был очень прост. Давайте рассмотрим более сложный пример.

Предположим, что есть некоторый код, содержащий класс Employee. В этом классе Employee имеется метод, возвращающий всех сотрудников. Также предположим, что есть другой код, включающий класс Contact, с определенным в нем методом, публикующим контакты. Пусть необходимо опубликовать всех сотрудников в виде контактов.

Задача выглядит достаточно простой, но здесь таится ловушка. Общий метод Employee, который извлекает всех сотрудников, возвращает их в списке ArrayList, хранящем объекты Employee, а метод Contact, публикующий контакты, требует массива объектов типа Contact. Ниже показан обычный для такого случая код:

```
namespace LINQDev.HR
{
  public class Employee
  {
    public int id;
    public string firstName;
    public string lastName;

    public static ArrayList GetEmployees()
    {
      // Конечно, реальный код здесь должен был бы выполнять запрос к базе данных.
      ArrayList al = new ArrayList();

      // Средство инициализации объектов C# превращает это в пару пустяков.
      al.Add(new Employee { id = 1, firstName = "Joe", lastName = "Rattz"} );
      al.Add(new Employee { id = 2, firstName = "William", lastName = "Gates"} );
      al.Add(new Employee { id = 3, firstName = "Anders", lastName = "Hejlsberg"} );
        return(al);
      }
  }
}

namespace LINQDev.Common
{
   public class Contact
   {
      public int Id;
      public string Name;

      public static void PublishContacts(Contact[] contacts)
      {
         // Этот метод публикации просто выводит их в окно консоли.
         foreach(Contact c in contacts)
            Console.WriteLine("Contact Id: {0} Contact: {1}", c.Id, c.Name);
      }
   }
}
```

Как видите, класс Employee и метод GetEmployees находятся в одном пространстве имен — LINQDev.HR, а метод GetEmployees возвращает ArrayList. Метод PublishContacts расположен в другом пространстве имен — LINQDev.Common, и требует передачи ему массива объектов Contact.

Ранее это всегда требовало итерации по списку ArrayList, который возвращен методом GetEmployees, и создания нового массива типа Contact для передачи методу PublishContacts. Как можно видеть в листинге 1.6, LINQ значительно облегчает решение этой задачи.

Листинг 1.6. Вызов обычного кода

```
ArrayList alEmployees = LINQDev.HR.Employee.GetEmployees();

LINQDev.Common.Contact[] contacts = alEmployees
  .Cast<LINQDev.HR.Employee>()
  .Select(e => new LINQDev.Common.Contact {
              Id = e.id,
              Name = string.Format("{0} {1}", e.firstName, e.lastName)
            })
  .ToArray<LINQDev.Common.Contact>();

LINQDev.Common.Contact.PublishContacts(contacts);
```

Чтобы преобразовать коллекцию ArrayList объектов Employee в массив объектов Contact, сначала выполняется приведение ArrayList объектов Employee к последовательности IEnumerable<Employee> с использованием стандартной операции запросов Cast. Это необходимо, потому что использован унаследованный класс коллекции ArrayList. С синтаксической точки зрения в коллекции ArrayList хранятся объекты класса System.Object, а не объекты типа класса Employee. Поэтому они должны быть приведены к объектам Employee. Если бы метод GetEmployees возвращал обобщенную коллекцию List, необходимости в этом не было бы. Однако на момент написания унаследованного кода этот тип коллекции не был доступным.

Затем на возвращенной последовательности объектов Employee вызывается операция Select, и в *лямбда-выражении* — коде, переданном внутрь вызова метода Select, — создается и инициализируется экземпляр объекта Contact с использованием для этого средства инициализации объектов C#, чтобы присвоить значения входного элемента Employee вновь сконструированному выходному элементу Contact. Лямбда-выражение (lambda expression) — это средство C#, которое является сокращенным способом указания анонимных методов и описано в главе 2. И, наконец, последовательность вновь сконструированных объектов Contact преобразуется в массив объектов Contact с применением операции ToArray, потому что этого требует метод PublishContacts. Разве не изящно? А вот результат:

```
Contact Id: 1 Contact: Joe Rattz
Contact Id: 2 Contact: William Gates
Contact Id: 3 Contact: Anders Hejlsberg
```

Как видите, LINQ может делать многое помимо запросов данных. По мере чтения глав книги задумайтесь о дополнительных применениях средств, предоставленных LINQ.

Советы начинающим

Во время работы с LINQ при написании этой книги, мы часто чувствовал себя запутанными, сбитыми с толку и несообразительными. Несмотря на то что доступно много очень полезных ресурсов для разработчиков, которые желают изучить весь потенциал LINQ, ниже приведено несколько небольших советов, которые помогут начать. В некоторых отношениях эти советы кажутся более уместными в конце книги. В конце концов, часть концепций, упомянутых в них, пока еще не рассматривалась. Но было бы не очень корректно заставить сначала прочесть всю книгу лишь для того, чтобы добраться до полезных советов в ее конце. С учетом всего сказанного, в последующих разделах содержатся советы, которые вы сочтете полезными, даже пока не полностью понимая их.

Используйте ключевое слово var, когда запутались

Ключевое слово var необходимо использовать при захвате последовательности от анонимных классов в переменную, иногда это удобный способ заставить код компилироваться, когда возникает путаница со сложными обобщенными типами. Хотя предпочтительнее подход к разработке, при котором точно известно, какого типа данные содержатся в последовательности — в том смысле, что для IEnumerable<T> должен быть известен тип T — иногда, особенно в начале работы с LINQ, это может вводить в заблуждение. Если обнаруживается, что код не компилируется из-за несоответствия типов данных, попробуйте заменить явно установленные типы переменных на указанные с применением ключевым словом var.

Например, предположим, что есть следующий код:

```
// Этот код не компилируется.
Northwind db =
  new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");

IEnumerable<?> orders = db.Customers
  .Where(c => c.Country == "USA" && c.Region == "WA")
  .SelectMany(c => c.Orders);
```

Может быть неясно, каков тип данных у последовательности IEnumerable. Вы знаете, что это IEnumerable некоторого типа T, но что собой представляет T? Удобный трюк состоит в присваивании результата запроса переменной, тип которой указан с помощью ключевого слова var, и затем получить тип текущего значения переменной, так что тип T известен. В листинге 1.7 показано, как должен выглядеть такой код.

Листинг 1.7. Пример кода, использующего ключевое слово var

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");

var orders = db.Customers
  .Where(c => c.Country == "USA" && c.Region == "WA")
  .SelectMany(c => c.Orders);

Console.WriteLine(orders.GetType());
```

Обратите в этом примере внимание на то, что теперь тип переменной orders указан с использованием ключевого слова var. Запуск этого кода даст следующий вывод:

```
System.Data.Linq.DataQuery`1[nwind.Order]
```

Здесь присутствует загадочный жаргон компилятора, но интерес представляет часть nwind.Order. Теперь вы знаете, что типом данных полученной последовательности является nwind.Order.

Если такой способ не устаивает, запустите пример в отладчике и просмотрите переменную orders в окне Locals (Локальные); для нее будет показан следующий тип данных:

```
System.Linq.IQueryable<nwind.Order> {System.Data.Linq.DataQuery<nwind.Order>}
```

Это проясняет тот факт, что вы имеете дело с последовательностью объектов nwind.Order. Формально здесь получается IQueryable<nwind.Order>, но это при желании можно присвоить IEnumerable<nwind.Order>, поскольку IQueryable<T> наследуется от IEnumerable<T>.

Таким образом, можно переписать предыдущий код и добавить перечисление результатов, как показано в листинге 1.8.

Листинг 1.8. Пример кода из листинга 1.7, но с явными типами

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
IEnumerable<Order> orders = db.Customers
  .Where(c => c.Country == "USA" && c.Region == "WA").SelectMany(c => c.Orders);
foreach(Order item in orders)
   Console.WriteLine("{0} - {1} - {2}", item.OrderDate, item.OrderID, item.ShipName);
```

На заметку! Чтобы приведенный код работал, должна быть предусмотрена директива `using` для пространства имен `System.Collectiobs.Generic` в дополнение к пространству имен `System.Linq`, которое всегда используется при работе с кодом LINQ.

Этот код должен дать следующий результат (приведенный с сокращениями):

```
3/21/1997 12:00:00 AM - 10482 - Lazy K Kountry Store
5/22/1997 12:00:00 AM - 10545 - Lazy K Kountry Store
...
4/17/1998 12:00:00 AM - 11032 - White Clover Markets
5/1/1998 12:00:00 AM - 11066 - White Clover Markets
```

Используйте операции `Cast` или `OfType` для унаследованных коллекций

Вы обнаружите, что большинство стандартных операций запросов LINQ могут быть вызваны на коллекциях, реализующих интерфейс `IEnumerable<T>`. Ни одна из унаследованных коллекций C# из пространства имен `System.Collections` не реализует `IEnumerable<T>`. Поэтому возникает вопрос: как использовать LINQ с унаследованными коллекциями?

Есть две стандартных операции запросов, специально предназначенные для этой цели — `Cast` и `OfType`. Обе они могут использоваться для преобразования унаследованных коллекций в последовательности `IEnumerable<T>`. В листинге 1.9 показан пример.

Листинг 1.9. Преобразование унаследованной коллекции в `IEnumerable<T>` с применением операции `Cast`

```
// Построим унаследованную коллекцию.
ArrayList arrayList = new ArrayList();
arrayList.Add("Adams");
arrayList.Add("Arthur");
arrayList.Add("Buchanan");
IEnumerable<string> names = arrayList.Cast<string>().Where(n => n.Length < 7);
foreach(string name in names)
   Console.WriteLine(name);
```

В листинге 1.10 представлен пример использования операции `OfType`.

Листинг 1.10. Использование операции `OfType`

```
// Построим унаследованную коллекцию.
ArrayList arrayList = new ArrayList();
arrayList.Add("Adams");
arrayList.Add("Arthur");
arrayList.Add("Buchanan");
IEnumerable<string> names = arrayList.OfType<string>().Where(n => n.Length < 7);
foreach(string name in names)
   Console.WriteLine(name);
```

Оба примера дают одинаковый результат:

```
Adams
Arthur
```

Разница между двумя операциями состоит в том, что `Cast` пытается привести все элементы в коллекции к указанному типу, помещая их в выходную последовательность. Если в коллекции есть объект типа, который не может быть приведен к указанному, генерируется исключение. Операция `OfType` пытается поместить в выходную последовательность только те элементы, которые могут быть приведены к указанному типу.

Отдавайте предпочтение операции `OfType` перед `Cast`

Одной из наиболее важных причин добавления обобщений в C# была необходимость предоставить языку возможность создавать коллекции со статическим контролем типов. До появления обобщений приходилось создавать собственные специфические типы коллекций для каждого типа данных, которые нужно было в них хранить — отсутствовал способ гарантировать, что каждый элемент, помещаемый в унаследованную коллекцию, имеет один и тот же корректный тип. Ничто не могло помешать коду добавить объект `TextBox` в `ArrayList`, предназначенный для хранения только объектов `Label`.

С появлением обобщений в версии C# 2.0 разработчики получили в свои руки способ явно устанавливать, что коллекция может содержать только элементы заданного типа. Хотя операции `OfType` и `Cast` могут работать с унаследованными коллекциями, `Cast` требует, чтобы каждый объект в коллекции относился к правильному типу, что было фундаментальным недостатком унаследованных коллекций, из-за которого появились обобщения. Когда используется операция `Cast` и любой из объектов в коллекции не может быть приведен к указанному типу данных, генерируется исключение. С другой стороны, с помощью операции `OfType` в выходной последовательности `IEnumerable<T>` будут сохранены только объекты указанного типа, и никаких исключений генерироваться не будет. При лучшем сценарии все объекты относятся к правильному типу, поэтому все попадают в выходную последовательность. В худшем сценарии некоторые элементы будут пропущены, но в случае применения операции `Cast` они привели бы к исключению.

Не рассчитывайте на безошибочность запросов

В главе 3 речь пойдет о том, что запросы LINQ часто являются отложенными и не выполняются сразу при вызове. Например, рассмотрим следующий фрагмент кода из листинга 1.1:

```
var items =
  from s in greetings
  where s.EndsWith("LINQ")
  select s;

foreach (var item in items)
  Console.WriteLine(item);
```

Хотя может показаться, что запрос выполняется при инициализации переменной `items`, на самом деле это не так. Поскольку операции `Where` и `Select` являются отложенными, запрос на самом деле не выполняется в этой точке. Запрос просто вызывается, объявляется или определяется, но не выполняется. Все начинает происходить тогда, когда из него извлекается первый результат. Это обычно происходит при перечислении переменной с результатами запроса. В этом примере результат запроса не востребован до тех пор, пока не запустится оператор `foreach`. Такое поведение запроса позволяет называть его *отложенным*.

Очень легко забыть о том, что многие операции запросов являются отложенными и не выполняются до тех пор, пока не начнется перечисление результатов. Это значит, что можно иметь неправильно написанный запрос, который сгенерирует исключение только тогда, когда начнется перечисление его результатов. Такое перечисление может начаться намного позже, так что можно легко забыть, что причиной неприятностей стал неправильный запрос.

Рассмотрим код в листинге 1.11.

Листинг 1.11. Запрос с преднамеренным исключением, отложенным до перечисления

```
string[] strings = { "one", "two", null, "three" };
Console.WriteLine("Before Where() is called.");
IEnumerable<string> ieStrings = strings.Where(s => s.Length == 3);
Console.WriteLine("After Where() is called.");

foreach(string s in ieStrings)
{
  Console.WriteLine("Processing " + s);
}
```

Известно, что третий элемент в массиве строк — `null`, и нельзя вызвать `null.Length` без генерации исключения. Выполнение кода благополучно пройдет строку, где вызывается запрос. Все будет хорошо до тех пор, пока не начнется перечисление последовательности `ieStrings`, и не дойдет до третьего элемента, где возникнет исключение. Ниже показан результат выполнения этого кода:

```
Before Where() is called.
After Where() is called.
Processing one
Processing two

Unhandled Exception: System.NullReferenceException: Object reference not set to
an instance of an object.

Необработанное исключение: System.NullReferenceException: Объектная ссылка не
установлена в экземпляр объекта.
...
```

Как видите, вызов операции `Where` прошел без исключения. Оно не появилось до тех пор, пока при перечислении не была произведена попытка обратиться к третьему элементу последовательности. Теперь представьте, что последовательность `ieStrings` передана функции, которая дальше выполняет перечисление последовательности — возможно, чтобы наполнить раскрывающийся список или какой-то другой элемент управления. Легко подумать, что исключение вызвано сбоем в этой функции, а не самим запросом LINQ.

Используйте преимущество отложенных запросов

В главе 3 тема отложенных запросов будет раскрыта более глубоко. Однако здесь следует отметить, что отложенный запрос, который в конечном итоге возвращает `IEnumerable<T>`, может перечисляться снова и снова, получая последние данные из источника. В этом случае не нужно ни вызывать, ни, как отмечалось ранее, объявлять запрос заново.

В большинстве примеров кода этой книги вы увидите вызов запроса и возврат `IEnumerable<T>` для некоторого типа `T`, сохраняемый в переменной. Затем обычно запускается оператор `foreach` на последовательности `IEnumerable<T>`. Это реализовано для демонстрационных целей. Если код выполняется много раз, повторный вызов за-

проса — лишняя работа. Более оправданным может быть наличие метода инициализации запроса, который вызывается однажды в жизненном цикле контекста, и в котором конструируются все запросы. Затем можно выполнить перечисление конкретной последовательности, чтобы получить последнюю версию результатов из запроса.

Используйте свойство Log из DataContext

При работе с LINQ to SQL не забывайте, что класс базы данных, генерируемый SQLMetal, унаследован от System.Data.Linq.DataContext. Это значит, что сгенерированный класс DataContext имеет некоторую полезную встроенную функциональность, такую как свойство Log типа TextWriter.

Одна из полезных возможностей объекта Log состоит в том, что он выводит эквивалентный SQL-оператор запроса IQueryable<T> до подстановки параметров. Случалось ли вам сталкиваться с отказом кода в рабочей среде, который, как вам кажется, вызван данными? Не правда ли, было бы хорошо запустить запрос на базе данных, вводя его в SQL Enterprise Manager или Query Analyzer, чтобы увидеть в точности, какие данные он возвращает? Свойство Log класса DataContext выводит запрос SQL. Соответствующий пример показан в листинге 1.12.

Листинг 1.12. Пример использования объекта DataContext.Log

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
db.Log = Console.Out;

IQueryable<Order> orders = from c in db.Customers
                           from o in c.Orders
                           where c.Country == "USA" && c.Region == "WA"
                           select o;

foreach(Order item in orders)
  Console.WriteLine("{0} - {1} - {2}", item.OrderDate, item.OrderID, item.ShipName);
```

Этот код производит следующий вывод:

```
SELECT [t1].[OrderID], [t1].[CustomerID], [t1].[EmployeeID], [t1].[OrderDate],
 [t1].[RequiredDate], [t1].[ShippedDate], [t1].[ShipVia], [t1].[Freight],
 [t1].[ShipName], [t1].[ShipAddress], [t1].[ShipCity], [t1].[ShipRegion],
 [t1].[ShipPostalCode], [t1].[ShipCountry]
FROM [dbo].[Customers] AS [t0], [dbo].[Orders] AS [t1]
WHERE ([t0].[Country] = @p0) AND ([t0].[Region] = @p1) AND ([t1].[CustomerID] =
 [t0].[CustomerID])
-- @p0: Input String (Size = 3; Prec = 0; Scale = 0) [USA]
-- @p1: Input String (Size = 2; Prec = 0; Scale = 0) [WA]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.20706.1

3/21/1997 12:00:00 AM - 10482 - Lazy K Kountry Store
5/22/1997 12:00:00 AM - 10545 - Lazy K Kountry Store
6/19/1997 12:00:00 AM - 10574 - Trail's Head Gourmet Provisioners
6/23/1997 12:00:00 AM - 10577 - Trail's Head Gourmet Provisioners
1/8/1998 12:00:00 AM - 10822 - Trail's Head Gourmet Provisioners
7/31/1996 12:00:00 AM - 10269 - White Clover Markets
11/1/1996 12:00:00 AM - 10344 - White Clover Markets
3/10/1997 12:00:00 AM - 10469 - White Clover Markets
3/24/1997 12:00:00 AM - 10483 - White Clover Markets
4/11/1997 12:00:00 AM - 10504 - White Clover Markets
7/11/1997 12:00:00 AM - 10596 - White Clover Markets
10/6/1997 12:00:00 AM - 10693 - White Clover Markets
10/8/1997 12:00:00 AM - 10696 - White Clover Markets
```

```
10/30/1997 12:00:00 AM - 10723 - White Clover Markets
11/13/1997 12:00:00 AM - 10740 - White Clover Markets
1/30/1998 12:00:00 AM - 10861 - White Clover Markets
2/24/1998 12:00:00 AM - 10904 - White Clover Markets
4/17/1998 12:00:00 AM - 11032 - White Clover Markets
5/1/1998 12:00:00 AM - 11066 - White Clover Markets
```

Используйте форум LINQ

Несмотря на приведенные полезные советы, все же более чем вероятно то, что вы периодически будете оказываться в тупике. Не забывайте, что на MSDN.com существует отдельный форум, посвященный LINQ (`http://www.linqdev.com`). Этот форум отслеживается разработчиками Microsoft, и на нем можно почерпнуть массу полезных сведений.

Резюме

Вероятно, что вам не терпится перейти к следующей главе, но сначала давайте вспомним несколько важных моментов.

Самое главное — LINQ изменяет способ, которым разработчики .NET выполняют запросы данных. Следует иметь в виду, что LINQ — это не просто новая библиотека для добавления к проекту. Это общий подход к опросу данных, объединяющий несколько компонентов в зависимости от типа опрашиваемого хранилища данных. В настоящее время LINQ можно применять для запросов к следующим источникам данных: коллекциям, находящимся в памяти, используя LINQ to Objects; к XML, используя LINQ to XML; DataSet — с помощью LINQ to DataSet; к базам данных SQL Server с применением LINQ to SQL.

Также помните, что LINQ предназначен не только для запросов. Мы сочли LINQ очень полезным также для форматирования, проверки достоверности и даже извлечения данных с последующим их преобразованием в формат, подходящий для использования в элементах управления Windows Forms и WPF.

И последнее (по порядку, но не по важности): мы надеемся, что вы не будете пренебрегать советами, приведенными в этой главе. Это не проблема, если некоторые из них пока еще не понятны. Все прояснится в процессе чтения книги. Просто вспоминайте о них, когда попадаете в сложную ситуацию.

В следующей главе будут описаны расширения, добавленные Microsoft в язык C#, которые сделали возможным синтаксис и работу LINQ.

ГЛАВА 2

Расширения языка C# для LINQ

В предыдущей главе было представлено введение в LINQ. Рассматривались некоторые простые примеры и давались несколько предварительных советов. Возможно, используемый в них синтаксис показался непривычным. Причина в том, что язык C# был расширен новыми средствами, специфичными для LINQ. Эти средства появились в версии C# 3.0 и получили дальнейшее развитие в .NET 4.0. В настоящей главе вы ознакомитесь с мощными дополнениями языка C#.

Дополнения языка C#

Чтобы обеспечить гладкую интеграцию LINQ с C#, в язык C# понадобилось внести существенные усовершенствования. Хотя все эти средства ценны и сами по себе, на самом деле они являются частями общего вклада в LINQ, который делает расширения C# столь замечательными.

Чтобы действительно понять большую часть синтаксиса LINQ, необходимо сначала разобраться в некоторых новых средствах языка C# и только затем приступать к работе с компонентами LINQ. В этой главе будут раскрыты следующие дополнения языка:

- лямбда-выражения;
- деревья выражений;
- ключевое слово `var`, инициализация объектов и коллекций и анонимные типы;
- расширяющие методы;
- частичные методы;
- выражения запросов.

В примерах этой главы добавляемые сборки и пространства имен в директивах `using`, которые упоминались в главе 1, явно не указываются. Однако будет указан ряд новых сборок и пространств имен, но только в тех примерах, где они встречаются впервые.

Лямбда-выражения

Начиная с версии 3, язык C# поддерживает *лямбда-выражения* (lambda expressions). Лямбда-выражения использовались в языках программирования вроде LISP с давних времен, а впервые их концепция была сформулирована в 1936 г. американским математиком Алонзо Черчем (Alonzo Church). Эти выражения представляют сокращенный синтаксис для определения алгоритма.

Но, прежде чем обратиться непосредственно к лямбда-выражениям, давайте взглянем на эволюцию способов указания алгоритма как аргумента метода, поскольку именно в этом и состоит назначение лямбда-выражений.

Использование именованных методов

Ранее, когда метод или переменная была типизирована так, что требовала делегата (`delegate`), разработчик должен был создавать именованный метод и передавать его имя туда, где требовался делегат.

В качестве примера рассмотрим следующую ситуацию. Предположим, что есть два разработчика, один из которых занимается кодом общего назначения, а другой — прикладным. Не обязательно, чтобы это были два разных разработчика, нам просто нужно разграничить две разные роли. Разработчик общего кода хочет создавать код общего назначения, который может быть использован многократно во всем проекте. Прикладной разработчик будет потреблять код общего назначения, чтобы создавать приложение. В этом случае разработчик общего кода желает создать метод для фильтрации массивов целых чисел, но с возможностью указания алгоритма, применяемого для фильтрации. Для начала должен быть объявлен делегат. Этот делегат будет прототипирован для приема параметра `int` и возврата значения `true`, если данный `int` должен быть включен в отфильтрованный массив.

Итак, он создает служебный класс и добавляет делегат и метод фильтрации. Вот этот код общего назначения:

```
public class Common
{
   public delegate bool IntFilter(int i);
   public static int[] FilterArrayOfInts(int[] ints, IntFilter filter)
   {
      ArrayList aList = new ArrayList();
      foreach (int i in ints)
      {
         if (filter(i))
         {
            aList.Add(i);
         }
      }
      return ((int[])aList.ToArray(typeof(int)));
   }
}
```

Разработчик общего кода поместит и объявление делегата, и `FilterArrayOfInts` в общую библиотечную сборку — динамически подключаемую библиотеку (DLL), — чтобы его можно было использовать во многих приложениях.

Приведенный выше метод `FilterArrayOfInts` позволяет прикладному разработчику передавать массив целых чисел и делегат его метода фильтрации, получая обратно отфильтрованный массив.

Теперь предположим, что прикладной разработчик желает отфильтровать только нечетные числа. Вот его метод фильтрации, который объявлен в прикладном коде:

```
public class Application
{
   public static bool IsOdd(int i)
   {
      return ((i & 1) == 1);
   }
}
```

На основе кода метода `FilterArrayOfInts` этот метод будет вызван для каждого значения `int` в массиве, переданном ему. Фильтр вернет `true`, если переданное значение `int` является нечетным. В листинге 2.1 показан пример использования метода `FilterArrayOfInts`, за которым представлен результат.

Листинг 2.1. Вызов метода фильтрации из общей библиотеки

```
using System.Collections;
int[] nums = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int[] oddNums = Common.FilterArrayOfInts(nums, Application.IsOdd);
foreach (int i in oddNums)
  Console.WriteLine(i);
```

Вот результат:

```
1
3
5
7
9
```

Обратите внимание, что для передачи делегата во втором параметре `FilterArrayOfInts` прикладной разработчик передает имя метода. Просто создав другой фильтр, он может фильтровать числа иначе. Он может иметь фильтр для четных чисел, простых чисел или отобранных в соответствии с любым нужным критерием. Делегаты позволяют создавать в высшей степени многократно используемый код.

Использование анонимных методов

Так или иначе, но написание всех этих методов фильтрации или любых других методов `delegate` может оказаться довольно утомительным. Многие из этих методов будут использованы лишь однократно, и скучно создавать именованные методы для таких случаев. Начиная с версии C# 2.0, у разработчиков появилась возможность создавать экземпляр делегата за счет предоставления встроенного кода как анонимного метода. Анонимные методы позволяют разработчику указывать код практически везде, где обычно должен передаваться делегат. Вместо создания метода `IsOdd` он может написать код фильтрации прямо в точке, где обычно передается делегат. В листинге 2.2 показан тот же код, что в листинге 2.1, но с использованием анонимного метода.

Листинг 2.2. Вызов метода фильтрации из общей библиотеки

```
using System.Collections;
int[] nums = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int[] oddNums = Common.FilterArrayOfInts(nums, delegate(int i)
  { return ((i & 1) == 1); });
foreach (int i in oddNums)
  Console.WriteLine(i);
```

Совсем неплохо. Прикладной разработчик более не обязан где-то объявлять метод. Это замечательно для кода логики фильтрации, вероятность многократного использования которого не высока. Как и ожидалось, вывод программы не отличается от предыдущего:

```
1
3
5
7
9
```

С применением анонимных методов связан один недостаток. Получаемый в результате код довольно громоздкий и трудно читаемый. Должен существовать более удобный способ написания кода метода.

Использование лямбда-выражений

Лямбда-выражения определяются как разделенный запятыми список параметров, за которым следует лямбда-операция, а за ней — выражение или блок операторов. Если параметров более одного, входные параметры помещаются в скобки. В C# лямбда-операция записывается как =>. Таким образом, лямбда-выражение в C# выглядит подобно следующему:

```
(параметр1, параметр2, ..., параметрN) => выражение
```

Или, когда требуется более высокая сложность, может применяться блок операторов:

```
(параметр1, параметр2, ..., параметрN) =>
{
  оператор1;
  оператор2;
  ...
  операторN;
  return(тип_возврата_лямбда_выражения);
}
```

В этом примере возвращаемый тип данных в конце блока операторов должен соответствовать типу возврата, указанному делегатом. Вот пример лямбда-выражения:

```
x => x
```

Это лямбда-выражение может быть прочитано, как “`x` идет к `x`”, или, возможно, “ввод `x` возвращает `x`”. Это значит, что при входной переменной `x` выражение вернет `x`. Это выражение просто возвращает то, что оно получило. Поскольку здесь только единственный параметр `x`, нет необходимости заключать его в скобки. Важно знать, что этот делегат диктует тип входного параметра `x` и тип возврата. Например, если делегат определен как принимающий `string`, но возвращающий `bool`, тогда выражение `x => x` не может использоваться, потому что если входной `x` будет иметь тип `string`, то возвращаемый `x` также должен относиться к типу `string`, но делегат определен как возвращающий `bool`. Поэтому с `delegate`, определенным таким образом, часть выражения справа от лямбда-операции (=>) должна вычисляться для возврата `bool`, как показано в следующем примере:

```
x => x.Length > 0
```

Это лямбда-выражение может быть прочитано как “`x` идет в `x.Length > 0`”, или, возможно, “ввод `x` возвращает `x.Length > 0`”. Поскольку правая часть выражения вычисляется как `bool`, делегат должен указывать, что метод возвращает `bool`, иначе компилятор сообщит об ошибке.

Следующее лямбда-выражение пытается вернуть длину входного аргумента. Значит, делегат должен определять `int` в качестве типа возврата:

```
s => s.Length
```

Если лямбда-выражению передается несколько параметров, отделяйте их запятыми и помещайте в скобки, как показано ниже:

```
(x, y) => x == y
```

Сложные лямбда-выражения могут даже включать блок операторов:

```
(x, y) =>
{
  if (x > y)
    return (x);
  else
    return (y);
}
```

Важно помнить, что делегат определяет, какими должны быть типы входных параметров и каким — тип возврата. Поэтому удостоверьтесь, что лямбда-выражение соответствует определению делегата.

Внимание! Удостоверьтесь, что лямбда-выражение рассчитано на прием входных типов, указанных в определении делегата, и возвращает тип, определенный как возвращаемый тип делегата.

Чтобы освежить память, ниже приведено объявление делегата, определенное разработчиком общего кода:

```
delegate bool IntFilter(int i);
```

Лямбда-выражение разработчика прикладного кода должно поддерживать `int`, переданный в параметре, и возвращать `bool`. Это может быть выведено из вызываемого им метода и назначения метода фильтрации, но важно помнить, что это диктуется делегатом.

Предыдущий пример, в котором на этот раз используется лямбда-выражение, должен выглядеть, как показано в листинге 2.3.

Листинг 2.3. Вызов метода фильтрации с лямбда-выражением

```
int[] nums = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int[] oddNums = Common.FilterArrayOfInts(nums, i => ((i & 1) == 1));
foreach (int i in oddNums)
  Console.WriteLine(i);
```

Вот такой удобный код получился. Поначалу забавным это может показаться, но как только вы привыкнете, то оцените читабельность и сопровождаемость такого кода. Как и можно было ожидать, результат будет тем же, что у предыдущих примеров:

```
1
3
5
7
9
```

С целью сравнения ниже показаны важнейшие строки кода в каждом из подходов:

```
int[] oddNums = // использование именованного метода
  Common.FilterArrayOfInts(nums, Application.IsOdd);
int[] oddNums = // использование анонимного метода
  Common.FilterArrayOfInts(nums, delegate(int i){return((i & 1) == 1);});
int[] oddNums = // использование лямбда-выражения
  Common.FilterArrayOfInts(nums, i => ((i & 1) == 1));
```

Конечно, первая строка действительно короче, но не забудьте, что там еще есть именованный метод, объявленный где-то в другом месте, в котором определено то, что он делает. Разумеется, если логика фильтрации должна использоваться в нескольких местах, или, возможно, алгоритм чересчур сложен и должен быть поручен специальному разработчику, может быть, имеет больше смысла создать именованный метод, который будут применять другие разработчики.

Совет. Сложные или многократно используемые алгоритмы лучше реализовать как именованные методы, чтобы их можно было использовать повторно любым разработчиком, не требуя от него полного понимания функционирования алгоритма.

Использовать именованные методы, анонимные методы или лямбда-выражения — выбор за разработчиком. Применяйте то, что имеет смысл в каждой конкретной ситуации.

Преимуществами лямбда-выражений часто пользуются при передаче их как аргументов в вызовы операций запросов LINQ. Поскольку каждый запрос LINQ, скорее всего, будет иметь уникальное или очень ограничено используемое лямбда-выражение, это обеспечивает гибкость указания логики операции без необходимости постоянного создания методов почти для каждого запроса.

Деревья выражений

Дерево выражения (expression tree) — эффективное представление в древовидной форме данных лямбда-выражения операции запроса. Эти представления деревьев выражений могут быть вычислены все сразу, так что единственный запрос может быть построен и выполнен на одном источнике данных, таком как база данных.

В большинстве примеров, рассмотренных до сих пор, операции выражений выполнялись в линейной манере. Рассмотрим следующий код:

```
int[] nums = new int[] { 6, 2, 7, 1, 9, 3 };
IEnumerable<int> numsLessThanFour = nums
  .Where(i => i < 4)
  .OrderBy(i => i);
```

Этот запрос содержит две операции, `Where` и `OrderBy`, которые ожидают делегаты в качестве своих аргументов. В результате его компиляции генерируется код на промежуточном языке .NET (Intermediate Language — IL), который идентичен IL-коду анонимного метода для каждого лямбда-выражения операции запроса.

Когда выполняется этот запрос, сначала вызывается операция `Where`, за ней — операция `OrderBy`. Такое линейное выполнение операций кажется оправданным для данного примера, но давайте подумаем о запросе к очень большому источнику, такому как база данных. Имеет ли смысл для SQL-запроса сначала обратиться к базе данных только с конструкцией `Where`, чтобы изменить порядок последующих вызовов? Естественно, это не реально для запросов к базе данных, как потенциально и для других типов запросов. Именно здесь приходят на помощь деревья выражений. Поскольку дерево выражений допускает параллельное вычисление и выполнение всех операций в запросе, может быть произведен единственный общий запрос вместо отдельных запросов для каждой операции.

Итак, теперь есть две разные вещи, которые может генерировать компилятор для лямбда-выражения операции — IL-код и дерево выражения. Что определяет, будет лямбда-выражение операции компилироваться в IL-код или в дерево выражения? Какое из этих двух действий предпримет компилятор определяется прототипом операции. Если операция объявлена для приема делегата метода, будет сгенерирован IL-код. Если же операция объявлена для приема *выражения* делегата, будет создано дерево выражения.

В качестве примера давайте рассмотрим две разных реализации операции `Where`. Первая — стандартная операция запроса, присутствующая в API- интерфейсе LINQ to Objects и определенная в классе `System.Linq.Enumerable`:

```
public static IEnumerable<T> Where<T>(
  this IEnumerable<T> source,
  Func<T, bool> predicate);
```

Вторая реализация операции Where находится в API-интерфейсе LINQ to SQL и принадлежит классу System.Linq.Queryable:

```
public static IQueryable<T> Where<T>(
  this IQueryable<T> source,
  System.Linq.Expressions.Expression<Func<int, bool>> predicate);
```

Как видите, первая операция Where объявлена как принимающая делегат, на что указывает делегат Func, и компилятор для этого лямбда-выражения операции сгенерирует IL-код. Делегат Func рассматривается в главе 3. А пока просто имейте в виду, что он определяет сигнатуру делегата, передаваемого в качестве аргумента предиката. Вторая операция Where объявлена для приема дерева выражения (Expression), поэтому здесь компилятор сгенерирует древовидное представление лямбда-выражения.

На заметку! Расширяющие методы на последовательностях IEnumerable<T> имеют IL-код, сгенерированный компилятором. Расширяющие методы на IQueryable<T> имеют сгенерированные компилятором деревья выражений.

Ключевое слово var, инициализация объектов и анонимные типы

Одно важное предупреждение: почти невозможно говорить о ключевом слове var и выведении неявных типов без демонстрации инициализации объектов или анонимных типов. Точно так же практически невозможно обсуждать инициализацию объектов или анонимные типы, не упоминая ключевого слова var. Все эти три расширения языка C# тесно связаны между собой.

Прежде чем описать в деталях каждое из этих средств языка — поскольку каждое из них описывает себя в терминах других — давайте взглянет на них всех вместе. Рассмотрим следующий оператор:

```
var[1] mySpouse = new {[2] FirstName = "Vickey"[3], LastName = "Rattz"[3] };
```

В этом примере объявляется переменная по имени mySpouse с использованием ключевого слова var. Ей присваивается значение анонимного типа, инициализированного с помощью новых средств инициализации объектов. Эта единственная строка кода пользуется преимуществами ключевого слова var, анонимных типов и инициализации объектов.

Короче говоря, ключевое слово var позволяет вывести тип данных локальной переменной на основе типа данных, которыми он инициализирован. Анонимные типы позволяют во время компиляции создавать новые типы данных — классы. В соответствии со словом *анонимные*, эти новые типы данных не имеют имен. Нельзя обычным образом создать анонимный тип данных, поскольку неизвестно, какие переменные-члены он содержит, и невозможно узнать, какие члены он содержит, если неизвестны их типы. И, наконец, типы этих членов могут быть ясны только после инициализации. Средство инициализации объектов обрабатывает все эти тонкости.

На основе этой строки кода компилятор создаст новый анонимный класс, содержащий два общедоступных члена типа string; первый получит имя FirstName, а второй — LastName.

[1] В этой строке кода легко обнаружить ключевое слово var, потому что оно указано явно.

[2] Можно определить, что здесь используется анонимный тип, так как в операции new не указан именованный класс.

[3] Применяется явная инициализация анонимного объекта с использованием нового средства инициализации объектов.

Ключевое слово `var` неявно типизированной локальной переменной

С появлением в C# анонимных типов возникла новая проблема. Если создается переменная анонимного типа, то переменной какого типа ее можно присваивать? Рассмотрим в качестве примера следующий код:

```
// Этот код не компилируется.
??? unnamedTypeVar = new {firstArg = 1, secondArg = "Joe" };
```

Какой тип переменной должен быть объявлен для `unnamedTypeVar`? Это проблема. В Microsoft решили ее, создав ключевое слово `var`. Это новое ключевое слово информирует компилятор о том, что он должен неявно вывести тип переменной из инициализатора переменной. Это значит, что переменная, объявленная с ключевым словом `var`, *должна* иметь инициализатор.

Пропуск инициализатора приводит к возникновению ошибки компиляции. В листинге 2.4 показан код, в котором переменная объявляется с ключевым словом `var`, но не инициализируется.

Листинг 2.4. Неправильное объявление переменной с использованием ключевого слова `var`

```
var name;
```

Здесь компилятор сообщит об ошибке:

```
Implicitly-typed local variables must be initialized
Неявно типизированная локальная переменная должна быть инициализирована
```

Поскольку эти переменные подвергаются статической проверке типов во время компиляции, инициализатор необходим, чтобы компилятор мог неявно вывести тип из него. Попытка присвоить значение другого типа где-нибудь в коде вызовет ошибку компиляции. Например, рассмотрим код в листинге 2.5.

Листинг 2.5. Неправильное присваивание переменной, объявленной с ключевым словом `var`

```
var name = "Joe";   // Пока все хорошо.
name = 1;           // Ошибка!
Console.WriteLine(name);
```

Этот код компилироваться не будет, потому что тип переменной `name` неявно выводится как `string`; однако производится попытка присвоить ей целочисленное значение `1`. Компилятор сообщит о следующей ошибке:

```
Cannot implicitly convert type 'int' to 'string'
Не удается неявно преобразовать тип 'int' в 'string'
```

Как видите, компилятор обязывает переменную сохранять первоначальный тип. Если вернуться к исходному примеру кода с назначением анонимного типа с использованием ключевого слова `var`, то код с дополнительной строкой для вывода значения переменной будет выглядеть так, как показано в листинге 2.6.

Листинг 2.6. Назначение анонимного типа переменной, объявленной с ключевым словом `var`

```
var unnamedTypeVar = new {firstArg = 1, secondArg = "Joe" };
Console.WriteLine(unnamedTypeVar.firstArg + ". " + unnamedTypeVar.secondArg);
```

Вот результат работы этого кода:

```
1. Joe
```

Как видите, применение ключевого слова `var` обеспечивает статический контроль типа, а также гибкость поддержки анонимных типов. Это станет очень важно, когда речь пойдет об операциях проекции типов в остальной части книги.

В примерах, приведенных до сих пор, использование ключевого слова `var` было обязательным, поскольку альтернативы не существовало. Если переменной необходимо присвоить объект анонимного типа, то такая переменная должна обязательно быть объявлена с ключевым словом `var`. Однако допускается указывать `var` при каждом объявлении переменной, если только она правильно инициализируется. Из соображений сопровождаемости рекомендуется этим приемом не злоупотреблять. Понятно, что разработчики всегда должны знать тип данных, с которым они работают, но если вы помните действительный тип данных сейчас, то будет ли это так, когда вы вернетесь к этому коду полгода спустя? Как насчет других разработчиков, которые впоследствии могут заняться сопровождением?

Совет. В целях сопровождаемости кода избегайте использования ключевого слова `var` лишь потому, что это удобно. Применяйте его только там, где это необходимо, например, в случае присваивания переменной объекта анонимного типа.

Выражения инициализации объектов и коллекций

Из-за потребности в динамических типах данных, которые позволяют создавать анонимные типы, возникает необходимость изменения способа инициализации объектов и коллекций. Поскольку выражения теперь представлены в виде лямбда-выражений или деревьев выражений, инициализация объектов и коллекций упростилась.

Инициализация объектов

Инициализация объектов позволяет указывать значения инициализации для общедоступных полей и свойств класса во время создания экземпляра. Для примера рассмотрим следующий код:

```
public class Address
{
   public string address;
   public string city;
   public string state;
   public string postalCode;
}
```

До появления средства инициализации объектов в C#, при отсутствии специализированного конструктора, объект типа `Address` нужно было инициализировать так, как показано в листинге 2.7.

Листинг 2.7. Создание экземпляра и инициализация класса старым способом

```
Address address = new Address();
address.address = "105 Elm Street";
address.city = "Atlanta";
address.state = "GA";
address.postalCode = "30339";
```

Такой подход был бы очень неудобным в лямбда-выражениях. Предположим, что нужно запросить значения из источника данных и с помощью операции `Select` спроектировать определенные члены на объект `Address`:

```
// Этот код не скомпилируется.
IEnumerable<Address> addresses = somedatasource
  .Where(a => a.State = "GA")
  .Select(a => new Address(???)???);
```

Просто отсутствовал бы удобный способ инициализировать члены вновь сконструированного объекта `Address`. Но не беспокойтесь! На помощь пришла инициализация объектов. Вы можете возразить, что проблема решается простым созданием конструктора, который принимал бы все эти инициализирующие значения при конструировании экземпляра объекта. Да, это так — иногда. Но это сопряжено со сложностями. И как быть в случае анонимного типа? Не проще ли создавать экземпляры объектов, как показано в листинге 2.8?

Листинг 2.8. Создание экземпляра и инициализация класса новым способом

```
Address address = new Address {
                   address = "105 Elm Street",
                   city = "Atlanta",
                   state = "GA",
                   postalCode = "30339"
                  };
```

Это *может* решить проблему в лямбда-выражении. Кроме того, помните, что возможности инициализации объектов могут применяться где угодно, а не только в запросах LINQ.

При использовании инициализации объекта компилятор создает экземпляр объекта с помощью конструктора класса без параметров, после чего инициализирует именованные члены указанными значениями. Любые члены, которые не указаны, получают значения по умолчанию, принятые для своих типов данных.

Инициализация коллекций

Поскольку одной только инициализации объектов не достаточно, кто-то в Microsoft должен был сказать: “А как насчет коллекций?”. *Инициализация коллекций* позволяет указывать инициализирующие значения для коллекции — как это делается для объектов. Пример инициализации коллекции приведен в листинге 2.9.

Листинг 2.9. Пример инициализации коллекции

```
using System.Collections.Generic;
List<string> presidents = new List<string> { "Adams", "Arthur", "Buchanan" };
foreach(string president in presidents)
{
  Console.WriteLine(president);
}
```

После запуска этого примера нажатием <Ctrl+F5> получается следующий результат:

```
Adams
Arthur
Buchman
```

В дополнение к использованию инициализации коллекций с LINQ, инициализированные коллекции часто удобно создавать в коде, где не присутствуют запросы LINQ.

Анонимные типы

Создание API-интерфейса на уровне языка для обобщенных запросов данных было затруднено в C# из-за нехватки возможностей создания новых типов данных во время компиляции. Если необходимо, чтобы запросы данных извлекали первоклассные элементы на уровне языка, для этого язык должен обладать возможностью создания первоклассных элементов данных уровня языка, которыми в C# являются классы. Поэтому спецификация языка C# теперь включает возможность динамического создания новых безымянных классов и объектов этих классов. Классы такого рода известны как *анонимные типы*.

Анонимный тип не имеет имени и генерируется компилятором на основе инициализации создаваемого экземпляра объекта. Поскольку класс не имеет имени типа, любые переменные, присваиваемые объекту анонимного типа, должны иметь какой-то способ объявления. Именно в этом состоит предназначение ключевого слова `var` в C#.

Анонимный тип незаменим при проектировании (projecting) новых типов данных с использованием операций `Select` или `SelectMany`. Без анонимных типов при вызове этих операций должны были всегда существовать предопределенные именованные классы. Было бы очень неудобно создавать именованные классы для каждого запроса.

В разделе “Инициализация объектов” ранее в главе обсуждался следующий код создания экземпляра и инициализации объекта:

```
Address address = new Address {
                      address = "105 Elm Street",
                      city = "Atlanta",
                      state = "GA",
                      postalCode = "30339"
                   };
```

Если бы вместо использования именованного класса `Address` возникло желание применить анонимный тип, нужно было бы просто опустить имя класса. Однако тогда не получится сохранить вновь созданный объект в переменной типа `Address`, потому что он уже не будет переменной типа `Address`. В таком случае он относился бы к сгенерированному типу, который известен только компилятору. Поэтому пришлось бы изменить также и тип переменной `address`. Здесь снова на помощь приходит ключевое слово `var`, как продемонстрировано в листинге 2.10.

Листинг 2.10. Создание и инициализация экземпляра анонимного типа с использованием инициализации объектов

```
var address = new {
              address = "105 Elm Street",
              city = "Atlanta",
              state = "GA",
              postalCode = "30339"
            };
Console.WriteLine("address = {0} : city = {1} : state = {2} : zip = {3}",
  address.address, address.city, address.state, address.postalCode);
Console.WriteLine("{0}", address.GetType().ToString());
```

Последний вызов `Console.WriteLine` был добавлен для того, чтобы можно было увидеть внутреннее, сгенерированное компилятором имя анонимного класса. Вот результат выполнения:

```
address = 105 Elm Street : city = Atlanta : state = GA : zip = 30339
<>f__AnonymousType5`4[System.String,System.String,System.String,System.String]
```

Этот тип анонимного класса определенно выглядит как сгенерированный компилятором. Естественно, имя класса, сгенерированное вашим компилятором, может отличаться.

Расширяющие методы

Расширяющий метод (extension method) — это статический метод статического класса, который можно вызывать, как если бы он был методом экземпляра другого класса. Например, можно создать расширяющий метод по имени `ToDouble`, который является статическим методом созданного вами статического класса по имени `StringConversion`, но будет вызываться так, как будто бы он представляет собой метод объекта типа `String`.

Прежде чем перейти к подробному рассмотрению расширяющих методов, давайте обсудим проблему, которая привела к их созданию, ознакомившись с разницей между статическими методами (класса) и методами уровня экземпляра (объекта). Методы уровня экземпляра могут вызываться только на *экземплярах* класса, иначе известных как *объекты*. Метод уровня экземпляра нельзя вызывать в отношении самого класса. Аналогично, статические методы должны вызываться на классе, а не на его экземплярах.

Сравнение методов экземпляра (объекта) и статических методов (класса)

Метод `ToUpper` класса `string` является примером метода уровня экземпляра. Вызывать `ToUpper` на самом классе `string` нельзя, это должно делаться на объекте `string`.

В листинге 2.11 это демонстрируется вызовом метода `ToUpper` на объекте по имени `name`.

Листинг 2.11. Вызов метода экземпляра на объекте

```
// Этот код компилируется.
string name = "Joe";
Console.WriteLine(name.ToUpper());
```

Этот код успешно компилируется и при запуске выдает следующий вывод:

```
JOE
```

Однако если попробовать вызвать метод `ToUpper` на самом классе `string`, то возникнет ошибка компиляции, потому что `ToUpper` — это метод уровня экземпляра. В листинге 2.12 показан пример такой попытки и сообщение об ошибке, выданное компилятором.

Листинг 2.12. Попытка вызвать метод экземпляра на классе

```
// Этот код не компилируется.
string.ToUpper();
```

Попытка скомпилировать этот код приведет к следующей ошибке компиляции:

```
An object reference is required for the nonstatic field, method, or property
'string.ToUpper()'
```

Объектная ссылка требуется для нестатического поля, метода или свойства 'string.ToUpper()'

Этот пример кажется несколько надуманным, поскольку он никак не может работать, т.к. здесь даже не задано строковое значение для преобразования к верхнему

регистру. Любая попытка сделать это эквивалентна попытке вызвать несуществующий вариант метода ToUpper, потому что нет прототипа для ToUpper, сигнатура которого бы содержала тип string.

Противоположностью методу ToUpper может быть метод Format класса string. Он определен как static. Это требует вызова метода Format на самом классе string, а не на объекте типа string. Сначала попытаемся вызвать его на объекте в листинге 2.13.

Листинг 2.13. Попытка вызвать метод класса на объекте

```
string firstName = "Joe";
string lastName = "Rattz";
string name = firstName.Format("{0} {1}", firstName, lastName);
Console.WriteLine(name);
```

Этот код приводит к выдаче компилятором следующего сообщения об ошибке:

```
Member 'string.Format(string, object, object)' cannot be accessed with an
instance reference; qualify it with a type name instead
```

Член 'string.Format(string, object, object)' не может быть доступным через ссылку экземпляра; квалифицируйте его с помощью имени типа

Однако если вызвать метод Format на самом классе string, код компилируется и работает, как ожидалось (листинг 2.14).

Листинг 2.14. Попытка вызвать метод класса на объекте

```
string firstName = "Joe";
string lastName = "Rattz";
string name = string.Format("{0} {1}", firstName, lastName);
Console.WriteLine(name);
```

Этот код выдаст следующий результат:

```
Joe Rattz
```

Иногда и без ключевого слова static из частей сигнатуры легко понять, что метод должен быть методом уровня экземпляра. Например, рассмотрим метод ToUpper. Он не принимает никаких аргументов, за исключением одной его перегруженной версии, ожидающей ссылку на объект CultureInfo. Если бы он не полагался на внутренние данные экземпляра string, то какую строку тогда бы преобразовывал к верхнему регистру?

Решение проблемы расширяющими методами

Так в чем же проблема? Для целей обсуждения предположим, что вы — разработчик, отвечающий за проектирование нового способа выдачи запросов к множеству объектов. Скажем, вы решили создать метод Where, чтобы помочь в обработке конструкций where. Как это сделать?

Должна ли операция Where быть сделана методом экземпляра? Если да, то в какой класс будет добавлен метод Where? Вы хотите, чтобы он работал для запроса любой коллекции объектов. Подходящий класс, куда логично было бы добавить метод Where, отсутствует. Для получения универсального средства опроса данных при таком подходе пришлось бы модифицировать массу разных классов.

Таким образом, ясно, что метод должен быть статическим — и что дальше? Подумайте о типовом запросе (SQL) и о том, насколько много конструкций where в нем часто бывает. Также учтите соединения, группирование и упорядочивание.

Пусть создана концепция нового типа данных — последовательность обобщенных объектов данных, которая будет называться Enumerable. Вполне уместно предположить,

что метод Where понадобится для оперирования последовательностью Enumerable (данных) и возврата другой отфильтрованной последовательности Enumerable. Вдобавок метод Where должен будет принимать аргумент, который позволит разработчику указывать точную логику, используемую для фильтрации записей данных из и в Enumerable. Этот аргумент, названный *предикатом* (predicate), может указываться в виде именованного метода, анонимного метода или лямбда-выражения.

Внимание! Следующие три примера кода в этом разделе являются гипотетическими и компилироваться не будут.

Поскольку метод Where требует входной последовательности Enumerable для фильтрации, и метод объявлен как static, эта входная Enumerable должна быть указана как аргумент в методе Where. Выглядеть это должно примерно так:

```
static Enumerable Enumerable.Where(Enumerable input, LambdaExpression predicate) {
...
}
```

Игнорируя пока что семантику лямбда-выражения, вызов метода Where должен выглядеть подобно показанному ниже:

```
Enumerable enumerable = {"one", "two", "three"};
Enumerable filteredEnumerable = Enumerable.Where(enumerable, lambdaExpression);
```

Этот код не особо сложный. Но что будет, если понадобится несколько конструкций where? Поскольку Enumerable, которым оперирует метод Where, должен быть аргументом метода, то в результате получается, что связывание методов в цепочку требует внедрения их друг в друга. Всего три конструкции where изменят код следующим образом:

```
Enumerable enumerable = {"one", "two", "three"};
Enumerable finalEnumerable =
  Enumerable.Where(Enumerable.Where(Enumerable.Where(enumerable, lX1), lX2), lX3);
```

Этот оператор должен читаться, начиная изнутри и наружу. Такое нелегко прочитать быстро. Можете ли вы представить, как будет выглядеть действительно сложный запрос? Должен существовать какой-то другой, лучший путь.

Решение

Замечательным решением была бы возможность вызывать статический метод Where на каждом объекте Enumerable, а не на классе. Тогда не пришлось бы передавать каждый Enumerable в метод Where, потому что объект Enumerable имел бы доступ к собственному внутреннему содержимому Enumerable. Это изменило бы синтаксис запроса, представленного выше, примерно так:

```
Enumerable enumerable = {"one", "two", "three"};
Enumerable finalEnumerable = enumerable.Where(lX1).Where(lX2).Where(lX3);
```

Внимание! Предыдущий и следующий примеры кода являются гипотетическими и компилироваться не будут.

Его можно даже переписать следующим образом:

```
Enumerable enumerable = {"one", "two", "three"};
Enumerable finalEnumerable = enumerable
  .Where(lX1)
  .Where(lX2)
  .Where(lX3);
```

Такой код читать намного легче. Оператор можно прочесть слева направо, сверху вниз. Как видите, этому синтаксису легко следовать, если только знать, что он делает. Именно потому в документации по LINQ и в настоящей книге часто встречаются запросы LINQ, записанные в таком формате.

В конечном итоге все, что нужно — это возможность иметь статический метод, который можно вызывать на экземпляре класса. Именно для этого предназначены расширяющие методы, и именно это они позволяют делать. Расширяющие методы были добавлены в C#, чтобы предоставить синтаксически элегантный способ вызова статических методов, без передачи им первого аргумента. Это позволяет вызывать расширяющие методы так, как будто они являются методами первого аргумента, что делает гораздо более читабельными цепочки вызовов расширяющих методов, чем в случае обычной передачи им первого аргумента. Расширяющие методы помогают LINQ, позволяя вызывать стандартные операции запросов на интерфейсе `IEnumerable<T>`.

На заметку! Расширяющие методы — это такие методы, которые, будучи статическими, могут быть вызваны на экземпляре (объекте) класса, а не на самом классе.

Объявление и вызов расширяющих методов

Указание первого аргумента метода с модификатором в виде ключевого слова `this` превращает его в расширяющий.

Расширяющий метод выглядит как метод экземпляра любого объекта с тем же типом, что и у его первого аргумента. Например, если первый аргумент расширяющего метода имеет тип `string`, то расширяющий метод будет выглядеть как метод экземпляра `string` и может быть вызван на любом экземпляре `string`.

Также имейте в виду, что расширяющие методы могут быть объявлены только в статических классах.

Ниже приведены два примера расширяющих методов.

```
namespace Netsplore.Utilities
{
   public static class StringConversions
   {
      public static double ToDouble(this string s) {
         return Double.Parse(s);
      }
      public static bool ToBool(this string s) {
         return Boolean.Parse(s);
      }
   }
}
```

Обратите внимание, что и класс, и каждый метод в нем объявлены как `static`. Теперь можно воспользоваться преимуществом этих расширяющих методов, вызывая статические методы на экземплярах объектов, как показано в листинге 2.15. Поскольку метод `ToDouble` статический, и его первый аргумент указан с ключевым словом `this`, метод `ToDouble` является расширяющим.

Листинг 2.15. Вызов расширяющего метода

```
using Netsplore.Utilities;
double pi = "3.1415926535".ToDouble();
Console.WriteLine(pi);
```

Этот код дает следующий вывод:

```
3.1415926535
```

Важно указать директиву `using` для пространства имен `Netsplore.Utilities`, иначе компилятор не найдет расширяющие методы и возникнет ошибка компиляции:

```
'string' does not contain a definition for 'ToDouble' and no extension method
'ToDouble' accepting a first argument of type 'string' could be found
(are you missing a using directive or an assembly reference?)
```

'string' не содержит определения для 'ToDouble' и не найдено ни одного расширяющего метода 'ToDouble', который принимает первый аргумент типа 'string' (возможно, пропущена директива using или ссылка на сборку)

Как уже упоминалось ранее, объявление расширяющего метода внутри нестатического класса не допускается. Если попробовать сделать это, компилятор выдаст следующее сообщение об ошибке:

```
Extension methods must be defined in a non-generic static class
```

Расширяющие методы должны быть определены в необобщенном статическом классе

Приоритет расширяющих методов

Обычные методы экземпляров объектов имеют преимущество перед расширяющими методами, когда их сигнатура соответствует сигнатуре вызова.

Расширяющие методы выглядят действительно полезной концепцией, особенно когда необходимо иметь возможность расширить класс, который расширять нельзя, такой как запечатанный (`sealed`) класс либо класс, исходный код которого недоступен. В показанных ранее примерах расширяющих методов добавлялись методы в класс `string`. Без механизма расширяющих методов сделать бы это не получилось, поскольку класс `string` запечатан.

Частичные методы

Появившиеся в версии C# 3.0 *частичные методы* (partial methods) добавляют в C# легковесный механизм обработки событий. Выбросьте из головы предположения о частичных методах, которые могли появиться на основе только их названия. Частичные методы имеют только один общий аспект с частичными классами — они могут существовать только в частичном классе. В действительности это правило номер 1 для частичных методов.

Прежде чем приступить к ознакомлению со всеми правилами, которым подчиняются частичные методы, рассмотрим, что они собой представляют. Частичный метод — это метод, прототип или определение которого указано в объявлении частичного класса, а реализация метода не предоставлена в том же самом объявлении частичного класса. Фактически они могут не иметь вообще *никакой* реализации *ни в одном* из объявлений одного и того же частичного класса. И если реализации метода нет ни в одном из объявлений частей этого класса, то компилятор не будет генерировать IL-код ни для объявления метода, ни для его вызова, ни для оценки аргументов, переданных методу. То есть все выглядит так, будто этот метод вообще не существует.

Некоторым не нравится название “частичные методы” из-за их несоответствия поведению частичных классов. Возможно, модификатор такого метода имело бы смысл назвать `ghost` (привидение), а не `partial`.

Пример частичного метода

Давайте взглянем на код частичного класса, который содержит определение частичного метода, в файле по имени `MyWidget.cs`:

Файл класса `MyWidget`

```
public partial class MyWidget
{
   partial void MyWidgetStart(int count);
   partial void MyWidgetEnd(int count);
   public MyWidget()
   {
      int count = 0;
      MyWidgetStart(++count);
      Console.WriteLine("In the constructor of MyWidget.");
      MyWidgetEnd(++count);
      Console.WriteLine("count = " + count);
   }
}
```

В приведенном объявлении класса `MyWidget` содержится частичный класс по имени `MyWidget`. Первые две строки кода являются определениями частичных методов. Здесь определены частичные методы с именами `MyWidgetStart` и `MyWidgetEnd`, каждый из которых принимает входной параметр типа `int` и возвращает `void`. Это еще одно правило: частичные методы должны возвращать `void`.

Следующая часть кода в классе `MyWidget` — это конструктор. Сначала объявляется переменная типа `int` по имени `count`, которая инициализируется `0`. Затем вызывается метод `MyWidgetStart`, выводится сообщение на консоль, вызывается метод `MyWidgetEnd` и, наконец, на консоль выводится значение переменной `count`. Обратите внимание, что при каждой передаче частичному методу значение `count` увеличивается на `1`. Это делается для того, чтобы доказать, что если реализация частичного метода не предоставлена, то его аргументы обрабатываться не будут.

В листинге 2.16 создается экземпляр объекта `MyWidget`.

Листинг 2.16. Создание экземпляра `MyWidget`

```
MyWidget myWidget = new MyWidget();
```

После нажатия <Ctrl+F5> получается следующий вывод:

```
In the constructor of MyWidget.
count = 0
```

Как видите, даже после того, как в конструкторе `MyWidget` значение переменной `count` было два раза увеличено, вывод в конце конструктора показывает, что оно по-прежнему равно `0`. Это объясняется тем, что компилятор не генерирует код вычисления аргументов для частичных методов, не имеющих реализации. Ни для одного из вызовов этих двух частичных методов код IL не создавался.

Теперь давайте добавим реализацию этих двух частичных методов:

Другое объявление `MyWidget`, содержащее реализации частичных методов

```
public partial class MyWidget
{
   partial void MyWidgetStart(int count)
   {
      Console.WriteLine("In MyWidgetStart(count is {0})", count);
   }
   partial void MyWidgetEnd(int count)
   {
      Console.WriteLine("In MyWidgetEnd(count is {0})", count);
   }
}
```

Снова запустите код из листинга 2.16. Результат будет выглядеть следующим образом:

```
In MyWidgetStart(count is 1)
In the constructor of MyWidget.
In MyWidgetEnd(count is 2)
count = 2
```

Как видите, теперь не только вызваны реализации частичных методов, но и вычислены переданные им аргументы. Это доказывает новое значение `count` в конце вывода.

В чем смысл частичных методов?

У одних может возникнуть вопрос: в чем смысл всего этого? Другие скажут, что это похоже на использование наследования и виртуальных методов. Зачем засорять язык чем-то похожим? Ответ в том, что частичные методы более эффективны, когда в коде планируется иметь множество потенциально нереализованных обработчиков. Механизм частичных методов позволяет писать код с учетом того, что кто-то другой расширит его через парадигму частичных классов, не нарушив работоспособности, если этого не случится.

Возможно, причиной появления частичных методов послужило то, что они позволяют генерировать код инструментами для сущностных классов LINQ to SQL. Чтобы сделать сгенерированные сущностные классы более полезными, к ним были добавлены частичные методы. Например, каждое отображенное свойство сгенерированного сущностного класса имеет частичный метод, который вызывается перед изменением свойства, а другой частичный метод вызывается после изменения свойства. Можно добавить другой модуль, который объявляет тот же самый сущностный класс, реализовать эти частичные методы и получать уведомления всякий раз, когда свойство собирается измениться, а также сразу после того, как оно изменится. Причем если не сделать этого, код не станет длиннее или медленнее. Не правда ли, эффективное решение?

Правила

К сожалению, существуют ряд правил, которым подчиняются частичные методы. Ниже приведен их список.

- Частичные методы должны быть определены только в частичных классах.
- В частичных методах должен быть указан модификатор `partial`.
- Частичные методы являются приватными, но в них не должен указываться модификатор `private`, иначе компилятор выдаст ошибку.
- Частичные методы должны возвращать `void`.
- Частичные методы могут не иметь реализации.
- Частичные методы должны быть статическими.
- Частичные методы могут иметь аргументы.

Все эти правила не так плохи. Учитывая выигрыш в гибкости генерируемых сущностных классов, плюс то, что можно делать с ними самостоятельно, язык C# обзавелся отличным средством.

Выражения запросов

Одним из удобных средств, предлагаемых языком C# ,является оператор `foreach`. Этот оператор транслируется компилятором в цикл с вызовами таких методов, как `GetEnumerator` и `MoveNext`. Простота оператора `foreach`, используемого для перечисления элементов массивов и коллекций, обеспечила ему широкую популярность и частое применение.

Одним из привлекательных для разработчиков средств LINQ является SQL-подобный синтаксис, доступный в LINQ-запросах. Этот синтаксис использовался в нескольких примерах LINQ, приведенных в начале первой главы. Синтаксис предоставлен через расширение языка C#, которое называется *выражения запросов*. Выражения запросов позволяют запросам LINQ принимать форму, подобную SQL, всего лишь с рядом небольших отличий.

Для выполнения запроса LINQ выражения запросов не обязательны. Альтернативой является использование стандартной точечной нотации C# с вызовом методов на объектах и классах. Во многих случаях применение стандартной точечной нотации оказывается более предпочтительным, поскольку она более наглядно демонстрирует, что в действительности происходит и когда. При записи запроса в стандартной точечной нотации не происходит никакой трансформации при компиляции. Именно поэтому во многих примерах этой книги не используется синтаксис выражений запросов, а предпочтение отдается стандартному синтаксису точечной нотации. Однако привлекательность синтаксиса выражений запросов бесспорна. Впечатление чего-то знакомого, которое он производит, когда вы формулируете свой первый запрос, может оказаться весьма привлекательным.

Получить представление о различиях между этими двумя синтаксисами лучше всего на примере. В листинге 2.17 показан запрос в стандартном точечном синтаксисе.

Листинг 2.17. Запрос, использующий стандартный синтаксис точечной нотации

```
string[] names = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
IEnumerable<string> sequence = names
  .Where(n => n.Length < 6)
  .Select(n => n);
foreach (string name in sequence)
{
   Console.WriteLine("{0}", name);
}
```

В листинге 2.18 представлен эквивалентный запрос, использующий синтаксис выражения запроса.

Листинг 2.18. Эквивалентный запрос, использующий синтаксис выражений запросов

```
string[] names = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
IEnumerable<string> sequence = from n in names
                          where n.Length < 6
                          select n;
foreach (string name in sequence)
{
   Console.WriteLine("{0}", name);
}
```

Первое, что можно заметить в примере с выражением запроса — это то, что в отличие от SQL, операция `from` предшествует операции `select`. Одной из причин этого была необходимость сужения контекста для средства IntelliSense. Без такой инверсии операций ввод в текстовом редакторе Visual Studio слова “select”, за которым следует пробел, не позволит средству IntelliSense определить, какие переменные должны отображаться в его раскрывающемся списке. Контекст допустимых переменных в этой точке ничем не ограничен. Если же сначала указать, откуда поступают данные, то IntelliSense получает контекст и может предоставить список переменных для выбора. Оба показанных примера дают один и тот же результат:

```
Adams
Bush
Ford
Grant
Hayes
Nixon
Obama
Polk
Taft
Tyler
```

Важно отметить, что синтаксис выражений запросов поддерживается только для наиболее распространенных операций запросов: `Where`, `Select`, `SelectMany`, `Join`, `GroupJoin`, `GroupBy`, `OrderBy`, `ThenBy`, `OrderByDescending` и `ThenByDescending`.

Грамматика выражений запросов

Выражения запросов *должны* подчиняться перечисленным ниже правилам.

1. Выражение должно начинаться с конструкции `from`.
2. Остальная часть выражения может содержать ноль или более конструкций `from`, `let` или `where`. Конструкция `from` — это генератор, который объявляет одну или более переменных диапазона, перечисляющих последовательность или соединение нескольких последовательностей. Конструкция `let` представляет переменную диапазона и присваивает ей значение. Конструкция `where` фильтрует элементы из входной последовательности или соединения несколько входных последовательностей в выходную последовательность.
3. Остальная часть выражения запроса может затем включать конструкцию `orderby`, содержащую одно или более полей сортировки с необязательным направлением упорядочивания. Направлением может быть `ascending` (по возрастанию) или `descending` (по убыванию).
4. Затем в оставшейся части выражения может идти конструкция `select` или `group`.
5. Наконец, в оставшейся части выражения может следовать необязательная конструкция продолжения. Такой конструкцией может быть либо `into`, ноль или более конструкций `join`, или же другая повторяющаяся последовательность перечисленных элементов, начиная с конструкций из правила 2. Конструкция `into` направляет результаты запроса в воображаемую выходную последовательность, которая служит конструкцией `from` для последующих выражения запросов, начиная с конструкций из правила 2.

Трансляция выражений запросов

Предположим, что создано синтаксически корректное выражение запроса. Следующая сложность связана с тем, как компилятор транслирует это выражение запроса в код C#.

Он должен транслировать выражение в стандартную точечную нотацию C#, о которой говорилось в разделе, посвященном выражениям запросов. Но как это делается?

Чтобы транслировать выражение запроса, компилятор ищет в нем шаблоны кода (code patterns). Компилятор выполняет несколько шагов трансляции в определенном порядке, чтобы превратить выражение запроса в стандартную нотацию C#. На каждом из этих шагов ищется один или более взаимосвязанных шаблонов кода. Прежде чем перейти к следующему шагу, компилятор должен многократно транслировать все вхождения шаблонов кода для данного шага трансляции. На каждом шаге он исходит из предположения, что все шаблоны кода для всех предыдущих шагов уже транслированы.

Прозрачные идентификаторы

Некоторые трансляции вставляют переменные перечислений с прозрачными идентификаторами. На шаге трансляции, описанном в следующем разделе, прозрачный идентификатор обозначен звездочкой (*). Его не нужно путать с символом * — шаблоном полей для выбора в SQL. При трансляции выражения запроса иногда компилятором генерируются дополнительные перечисления, и прозрачные идентификаторы используются для прохода по ним. Прозрачные идентификаторы существуют только во время процесса трансляции, и как только выражение запроса полностью транслировано, никаких прозрачных идентификаторов в запросе не остается.

Шаги трансляции

Теперь рассмотрим шаги трансляции. При этом для представления определенных частей запроса используются буквы переменных, описанные в табл. 2.1.

Таблица 2.1. Переменные шагов трансляции

Переменная	Описание	Пример
c	Сгенерированная компилятором временная переменная	—
e	Переменная диапазона	`from` **`e`** `in s`
f	Выбранный элемент-поле или новый анонимный тип	`from e in s select` **`f`**
g	Групповой элемент	`from e in s group` **`g`** `by k`
i	Воображаемая последовательность `into`	`from e in s select f into` **`i`**
k	Групповой или объединенный ключевой элемент	`from e in s group g by` **`k`**
l	Переменная, представленная `let`	`from e in s let` **`l`** `= v`
o	Упорядочивающий элемент	`from e in s orderby` **`o`**
s	Входная последовательность	`from e in` **`s`**
v	Значение, присвоенное переменной `let`	`from e in s let l =` **`v`**
w	Конструкция `where`	`from e in s where` **`w`**

Здесь необходимо высказать одно предостережение. Описание шагов трансляции может показаться довольно сложным. Пусть это вас не расстраивает. Для написания запросов LINQ полностью понимать все шаги трансляции не обязательно. Они приводятся здесь лишь для того, чтобы предоставить дополнительную информацию о трансляции на тот случай, если вдруг она понадобится, что случается редко, а может быть и никогда.

Шаги трансляции документированы в виде `шаблон кода ➤ трансляция`. Как ни странно, несмотря на то, что шаги трансляции представляются в порядке их выполнения компилятором, понять этот процесс проще, если изучать его в обратном порядке. Причина в том, что на первом шаге трансляции осуществляется только трансляция

первого шаблона кода, и остается масса не транслированных шаблонов кода, которые нужно разобрать. Это оставляет место для неясностей. Поскольку каждый шаг трансляции требует, чтобы шаблон кода предыдущего шага уже был транслирован, на момент получения финального шага трансляции не остается никаких неясностей. И потому понять последний шаг легче, чем первый. К тому же, проход по шагам трансляции в обратном порядке — наилучший способ понять весь процесс.

Итак, ниже описаны шаги трансляции в порядке их выполнения компилятором.

Конструкции `select` и `group` с конструкцией продолжения `into`

Если выражение запроса содержит конструкцию продолжения `into`, выполняется следующая трансляция:

```
from ...1 into ...2                    from i in
                                     ➤ from ...1
                                       ...2
```

Вот пример:

```
from c in customers                    from g in
group c by c.Country into g            from c in customers
select new                             group c by c.Country
  { Country = g.Key,                 ➤ select new
    CustCount = g.Count() }              { Country = g.Key,
                                           CustCount = g.Count() }
```

В результате последующих шагов трансляции это в конечном итоге транслируется в

```
customers.GroupBy(c => c.Country)
  .Select(g =>  new {Country = g.Key, CustCount = g.Count() })
```

Явные типы переменных перечислений

Если выражение запроса содержит конструкцию `from`, которая явно указывает тип переменной перечисления, выполнится следующая трансляция:

```
from T e in s                        ➤ from e in s.Cast<t>()
```

Вот пример:

```
from Customer c in customers         ➤ from c in customers.Cast<Customer>()
select c
```

В результате последующих шагов трансляции это в конечном итоге транслируется в

```
customers.Cast<Customer>()
```

Если выражение запроса содержит конструкцию `join`, которая явно указывает тип переменной перечисления, то будет выполнена следующая трансляция:

```
from T e in s                        ➤ join e in s.Cast<T>()
on k1 equals k2                          on k1 equals k2
```

Вот пример:

```
from c in customers                    from c in customers
join Order o in orders                 join o in orders.Cast<Order>()
on c,CustomerID equals o.CustomersID ➤ on c,CustomerID equals o.CustomersID
select new { c.Name,                   select new
             o.OrderDate,                { c.Name, o.OrderDate, o.Total }
             o.Total }
```

В результате последующих шагов трансляции это в конечном итоге транслируется в

```
customers
.Join(orders.Cast<Order>(),
     c => c.CustomerID,
     o => o.CustomerID,
     (c, o) => new { c.Name, o.OrderDate, o.Total })
```

Совет. Явно типизированные переменные перечислений необходимы, когда перечисляемая коллекция данных является одной из унаследованных коллекций данных, например `ArrayList`. Приведение, выполняемое при явной типизации переменной перечисления, преобразует унаследованную коллекцию в последовательность, реализующую `IEnumerable<T>` — так, что могут выполняться другие операции запросов.

Конструкции `join`

Если выражение запроса содержит конструкцию `from`, за которой следует конструкция `join` *без* конструкции продолжения `into` и вслед за ней — `select`, то имеет место следующая трансляция (`t` — временная переменная, сгенерированная компилятором):

```
from e1 in s1
join e2 in s2
on k1 equals k2
select f
```

➤

```
from t in s1
.Join(s2,
      e1 => k1,
      e2 => k2,
      (e1, e2) => f)
select t
```

Вот пример:

```
from c in customers
join o in orders
on c.CustomerID equals o.CustomerID
select new { c.Name,
             o.OrderDate,
             o.Total }
```

➤

```
from t in customers
.Join(orders,
      c => c.CustomerID,
      o => o.CustomerID,
      (c, o) => new
          { c.Name,
            o.OrderDate,
            o.Total })
```

В результате последующих шагов трансляции это в конечном итоге транслируется в

```
customers
.Join(orders,
     c => c.CustomerID,
     o => o.CustomerID,
     (c, o) => new { c.Name, o.OrderDate, o.Total })
```

Если выражение запроса содержит конструкцию `from`, за которой следует `join` с конструкцией продолжения `into` и вслед за ней — `select`, имеет место следующая трансляция (`t` — временная переменная, сгенерированная компилятором):

```
from e1 in s1
join e2 in s2
on k1 equals k2
into i
select f
```

➤

```
from t in s1
.GroupJoin(s2,
      e1 => k1,
      e2 => k2,
      (e1, e2) => f)
select t
```

Вот пример:

```
from c in customers
join o in orders
on c.CustomerID equals o.CustomerID
into co
select new
 { c.Name, Sum = co.Sum(0 => o.Total) }
```

➤

```
from t in customers
.GroupJoin(orders,
         c => c.CustomerID,
         o => o.CustomerID
         (c, o) => new
           { c.Name,
             Sum = co.Sum(
              o => co.Total)
select t
```

В результате последующих шагов трансляции это в конечном итоге транслируется в

```
customers
.GroupJoin(orders,
     c => c.CustomerID,
     o => o.CustomerID,
     (c, o) => new { c.Name, Sum = co.Sum(o = o.Total) })
```

Если выражение запроса содержит конструкцию `from`, за которой следует конструкция `join` без конструкции продолжения `into` и вслед за ней — что-то, отличное от `select`, то имеет место следующая трансляция (* — прозрачный идентификатор):

```
from e1 in s1                          from * in
join e2 in s2                          from e1 in s1
on k1 equals k2                      ➤ join e2 in s2
...                                    on k1 equals k2
                                       select new {e1, e2}
```

Обратите внимание, что в это время уже есть шаблон кода, соответствующий первому шаблону кода на этом шаге трансляции. Точнее, имеется выражение запроса, содержащее конструкцию `from`, за которой следует конструкция `join`, без конструкции продолжения `into`, за которой следует `select`. Поэтому компилятор повторит этот шаг трансляции. Если выражение запроса содержит конструкцию `from`, за которой следует конструкция `join`, с конструкцией продолжения `into`, за которой следует нечто, отличное от `select`, то имеет место следующая трансляция (* — прозрачный идентификатор):

```
from e1 in s1                          From * in
join e2 in s2                          from e1 in s1
on k1 equals k2                      ➤ join  e2 in s2
into i                                 on k1 equals k2
...                                    into i
                                       select new { e1, i }
```

На этот раз обратите внимание на присутствие шаблона кода, соответствующего второму шаблона кода на этом шаге трансляции. Точнее, есть выражение запроса, содержащее конструкцию `from`, за которой следует конструкция `join`, с конструкцией продолжения `into`, за которой идет `select`. Поэтому компилятор повторит этот шаг трансляции.

Конструкции `let` и `where`

Если выражение запроса содержит конструкцию `from`, за которой немедленно следует `let`, то имеет место следующая трансляция (* — прозрачный идентификатор):

```
from e in s                            from * in
let l = v                            ➤ from e1 in s1
                                       select new { e, l = v }
```

Вот пример (`t` — сгенерированный компилятором идентификатор, который невидим и недоступен в любом написанном коде):

```
from c on customers                    from * in
let cityStateZip =                     from c in customers
  c.City + ", " + c.State + " " + c.Zip select new {
select new { c.Name, cityStateZip }  ➤  c,
                                        cityStateZip =
                                          c.City + ", " + c.State + " " +
                                          c.Zip
                                       select new { c.Name, cityStateZip }
```

В результате последующих шагов трансляции это в конечном итоге транслируется в

```
customers
.Select( c => new { c,  cityStateZip = c.City + ", " + c.State + " " + c.Zip })
.Select( t => new { t.c.Name, t. cityStateZip })
```

Если выражение запроса содержит конструкцию `from`, за которой немедленно следует конструкция `where`, имеет место следующая трансляция:

```
from e in s                          from e in s
where w                            ➤ .Where(e => w)
```

Вот пример:

```
from c in customers                  from c in customers
where c.Country == "USA"           ➤ .Where(c => c.Country == "USA")
select new { c/Name, c.Country }     select new { c.Name, c.Country }
```

В результате последующих шагов трансляции это в конечном итоге транслируется в

```
customers
.Where(c => c.Country == "USA")
.Select(c => new { c.Name, c.Country })
```

Множественные генерирующие конструкции `from`

Если выражение запроса содержит две конструкции `from`, за которыми следует `select`, происходит следующая трансляция:

```
from e1 in s1                        from c in s1
from e2 in s2                        .SelectMany(e1 => from e2 in s2
select f                           ➤           select f)
                                     select c
```

Вот пример (`t` — временная переменная, сгенерированная компилятором):

```
from c in customers                  from t in customers
from o in c.Orders                   .SelectMany(c => from o in c.Orders
select new                         ➤          select new {
  { c.Name, o.OrderID, o.OrderDate }                  c.Name,
                                                      o.OrderID,
                                                      o.OrderDate
                                                    })
                                     select t
```

В результате последующих шагов трансляции это в конечном итоге транслируется в

```
customers
.SelectMany(c => c.Orders.Select(o => new { c.Name, o.OrderID, o.OrderDate }))
```

Если выражение запроса содержит две конструкции `from`, за которыми следует что-то отличное от `select`, то происходит следующая трансляция (`*` — прозрачный идентификатор):

```
from e1 in s1                        from * in
from e2 in s2                        from e1 in s1
...                                ➤ from e2 in s2
                                     select new { e1, e2 }
                                     ...
```

Вот пример (`*` — прозрачный идентификатор):

```
from c in customers                  from * in
from o in c.Orders                   from c in customers
orderby o.OrderDate descending       from o in c.Orders
select new                         ➤ select new { c, o }
  { c.Name, o.OrderID, o.OrderDate } orderby o.OrderDate descending
                                     select new
                                      { c.New, o.OrderID, o.OrderDate }
```

В дополнение к последующим шагам трансляции предыдущий транслированный код должен вызывать этот шаг трансляции снова, потому что после предыдущего пер-

вого шага имеется конструкция `from`, за которой следует `from`, а за ней — конструкция `select`, что является первым шаблоном кода, который данный шаг ищет для трансляции. Это пример шагов трансляции, которые иногда должны вызываться несколько раз, чтобы полностью транслировать все шаблоны кода, которые ищет любой конкретный шаг.

```
customers
.SelectMany(c => c.Orders.Select(o => new { c, o }))
.OrderByDescending( t => t.o.OrderDate)
.Select(t => new { t.c.Name, t.o.OrderID, t.o.OrderDate})
```

Конструкции `orderby`

Если в качестве направления упорядочивания выбрано `ascending`, происходит следующая трансляция:

```
from e in s                                  from e in s
orderby o1, o2                            ➤  .OrderBy(e => o2).ThenBy(e => o2)
```

Вот пример:

```
from c in customers                          from c in customers
orderby c.Country, c.Name                 ➤  .OrderBy(e => c.Country)
select new { c.Country, c.Name }             .ThenBy(c.Name)
                                             select new { c.Country, c.Name }
```

В результате последующих шагов трансляции это в конечном итоге транслируется в

```
customers
.OrderBy(c => c.Country)
.ThenBy(c.Name)
.Select(c => new { c.Country, c.Name }
```

Если в качестве направления упорядочивания выбрано `descending`, происходит трансляция в операции `OrderByDescending` или `ThenByDescending`. Следующий пример в основном совпадает с предыдущим, за исключением того, что имена запрашиваются в убывающем порядке:

```
from c in customers                          from c in customers
orderby c.Country, c.Name descending      ➤  .OrderBy(e => c.Country)
select new { c.Country, c.Name }             .ThenByDescending(c.Name)
                                             select new { c.Country, c.Name }
```

В результате последующих шагов трансляции это в конечном итоге транслируется в

```
customers
.OrderBy(c => c.Country)
.ThenByDescending (c.Name)
.Select(c => new { c.Country, c.Name }
```

Конструкции `select`

Если выбранный элемент в выражении запроса — тот же идентификатор, что и переменная-перечислитель последовательности, в том смысле, что выбирается целый элемент, хранящийся в последовательности, то происходит следующая трансляция:

```
from e in s                               ➤  s
select f
```

Вот пример:

```
from c in customers                       ➤  customers
select c
```

Если выбираемый элемент — не тот же идентификатор, что и переменная-перечислитель последовательности, в том смысле, что выбирается что-то отличное от целого элемента, хранящегося в последовательности, например, член элемента или анонимный тип, сконструированный из нескольких членов элемента, то происходит следующая трансляция:

```
from e in s                          ➤ s.Select(e => f)
select c
```

Вот пример:

```
from c in customers                  ➤ customers.Select(c => c.Name)
select c.Name
```

Конструкции group

Если группируемый элемент в выражении запроса — это тот же идентификатор, что и перечислитель последовательности, в том смысле, что группируется весь элемент, хранящийся в последовательности, то происходит такая трансляция:

```
from e in s                          ➤ s.GroupBy(e => k)
group g by k
```

Вот пример:

```
from c in customers                  ➤ customers.GroupBy(c => c.Country)
group c by c.Country
```

Если групповой элемент — не тот же идентификатор, что и перечислитель последовательности, в том смысле, что группируется нечто отличное от всего элемента, хранящегося в последовательности, то происходит следующая трансляция:

```
from e in s                          ➤ s.GroupBy(e => k, e=> g)
group g by k
```

Ниже показан пример:

```
from c in customers                    customers
group new { c.Country, c.Name }        .GroupBy(c => c.Country,
by c.Country                                    C => new {
                                     ➤                c.Country,
                                                     c.Name
                                                  })
```

В этой точке все шаги трансляции завершены, и выражение запроса должно быть полностью транслировано в синтаксис стандартной точечной нотации.

Резюме

Как видите, команда разработчиков C# в Microsoft хорошо добавила множество расширений в язык C#. Все расширения C#, которые рассматривались в этой главе, были предназначены специально для LINQ. Но даже без LINQ новые средства C# предоставляют массу преимуществ.

Новые выражения инициализации объектов и коллекций — просто подарок. Установка статических данных, выборок или тестовых данных стала намного легче, чем раньше, существенно сократив количество строк кода, необходимых для генерации данных. Это средство, в сочетании с новым ключевым словом `var` и анонимными типами, существенно упростило создание данных и типов данных “на лету”.

Расширяющие методы обеспечили возможность добавления функциональности к объектам, таким как запечатанные классы или классы, с недоступным исходным кодом, что было совершенно невозможно ранее.

Лямбда-выражения позволили сократить спецификации функциональности. Устранив потребность в анонимных методах, они добавили целый арсенал способов указывать простую функциональность с применением лаконичного синтаксиса. Хотя поначалу они могут показаться непривычными, со временем и накоплением опыта отношение к ним изменится.

Деревья выражений предоставляют независимым поставщикам, которые хотят добавить поддержку LINQ для собственных хранилищ данных, возможность обеспечить первоклассную производительность.

Частичные методы предлагают очень легковесный механизм обработки событий. Microsoft использует их в своих инструментах генерации сущностных классов для LINQ to SQL, так что можно легко вмешиваться в работу сущностных классов в нужные моменты времени.

И, наконец, выражения запросов вызывают желание немедленно ими воспользоваться, когда вы впервые видите запрос LINQ. Ничто так не облегчает задачу разработчику, изучающему новую технологию, как ее сходство с хорошо знакомой и зарекомендовавшей себя технологией. Придав запросам LINQ сходство с запросами SQL, Microsoft существенно облегчила их изучение.

Хотя все эти расширения языка являются замечательными средствами и по отдельности, все вместе они формируют фундамент LINQ. Теперь, когда вы знаете, что собой представляет LINQ, какой синтаксис и новые средства C# ему необходимы, пришло время заняться практикой. Следующим этапом будет изучение особенностей выполнения запросов LINQ к находящимся в памяти коллекциям, таким как массивы, `ArrayList` и все классы обобщенных коллекций .NET. Во второй части книги вы найдете комплект функций, дополняющих запросы. Это подмножество LINQ известно под названием LINQ to Objects.

ЧАСТЬ II

LINQ to Objects

В этой части

Глава 3. Введение в LINQ to Objects
Глава 4. Отложенные операции
Глава 5. Не отложенные операции

ГЛАВА 3

Введение в LINQ to Objects

Листинг 3.1. Простой запрос LINQ to Objects

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
string president = presidents.Where(p => p.StartsWith("Lin")).First();
Console.WriteLine(president);
```

На заметку! Этот код должен быть добавлен в консольное приложение Visual Studio 2010.

В листинге 3.1 демонстрируется практически все, что нужно знать о LINQ to SQL — выполнение SQL-подобных запросов к коллекциям и массивам, находящимся в памяти. Запустив пример нажатием <Ctrl+F5>, можно получить следующий результат:

```
Lincoln
```

Обзор LINQ to Objects

Отчасти то, что делает LINQ настолько мощным и удобным в применении, заключается в его тесной интеграции с языком C#. Вместо того, чтобы иметь дело с полностью новым набором средств в форме классов, можно применять все те же самые привычные коллекции[1] и массивы, с существующими классами. Это значит, что все преимущества запросов LINQ можно получить с минимальными модификациями существующего кода или же вовсе без них. Функциональность LINQ to Objects обеспечивается интерфейсом `IEnumerable<T>`, последовательностями и стандартными операциями запросов.

Например, если нужно отсортировать массив целых чисел, можно выполнить запрос LINQ для упорядочивания результатов — почти так же, как если бы это был запрос SQL. Может существовать список `ArrayList` объектов `Customer`, в котором требуется найти определенный объект `Customer`. В этом случае LINQ to Object будет наилучшим выбором.

[1] Коллекция должна реализовывать интерфейс `IEnumerable<T>` или `IEnumerable`, чтобы ее можно было опрашивать с помощью LINQ.

Возможно, многие будут склонны использовать главы этой части в качестве справочника. Хотя были предприняты существенные усилия, чтобы сделать их полезными в этом отношении, вы получите большую отдачу, прочитав их полностью от начала до конца. Многие концепции, применимые к одной операции, также применимы и к другим. Каждый раздел, посвященный отдельной операции, был сделан максимально независимым и самодостаточным, но при прочтении всего от начала до конца сформируется контекст, который будет утерян, если читать об одной операции и пропускать другие.

Интерфейс IEnumerable<T>, последовательности и стандартные операции запросов

IEnumerable<T> — это интерфейс, реализуемый всеми классами обобщенных коллекций C#, как это делают массивы. Этот интерфейс позволяет выполнять перечисление элементов коллекций.

Последовательность — это термин для обозначения коллекции, реализующей интерфейс IEnumerable<T>. Если есть переменная типа IEnumerable<T>, то можно сказать, что имеется последовательность элементов типа *T*. Например, IEnumerable<string> означает последовательность строк.

На заметку! Любая переменная, объявленная как IEnumerable<T> для типа T, рассматривается как последовательность типа T.

Большинство стандартных операций запросов представляют собой расширяющие методы в статическом классе System.Linq.Enumerable и прототипированы с IEnumerable<T> в качестве первого аргумента. Поскольку они являются расширяющими методами, предпочтительно вызывать их на переменной типа IEnumerable<T>, что позволяет синтаксис расширяющих методов, а не передавать переменную типа IEnumerable<T> в первом аргументе.

Методы стандартных операций запросов класса System.Linq.Enumerable, не являющиеся расширяющими методами — это просто статические методы, которые должны быть вызваны на классе System.Linq.Enumerable. Комбинация этих методов стандартных операций запросов дает возможность выполнять сложные запросы данных на последовательности IEnumerable<T>.

Унаследованные коллекции, не являющиеся обобщенными, которые существовали до C# 2.0, поддерживают интерфейс IEnumerable, а не IEnumerable<T>. Это значит, что нельзя *непосредственно* вызывать эти расширяющие методы с первым аргументом типа IEnumerable<T> на унаследованных коллекциях. Однако можно выполнять запросы LINQ на унаследованных коллекциях, вызывая стандартную операцию запроса Cast или OfType на унаследованной коллекции, чтобы произвести последовательность, реализующую IEnumerable<T>, а это откроет доступ к полному арсеналу стандартных операций запросов.

На заметку! Для выполнения запросов LINQ на унаследованных, необобщенных коллекциях C# используйте операции Cast или OfType.

Чтобы получить доступ к стандартным операциям запросов, добавьте в код директиву using System.Linq;, если ее еще там нет. Добавлять ссылку на сборку не понадобится, потому что необходимый код содержится в сборке System.Core.dll, которая автоматически добавляется к проекту средой Visual Studio 2010.

Возврат IEnumerable<T>, выдача и отложенные запросы

Важно помнить, что хотя многие из стандартных операций запросов прототипированы на возврат IEnumerable<T>, и IEnumerable<T> воспринимается как последовательность, на самом деле операции не возвращают последовательность в момент их вызова. Вместо этого операции возвращают объект, который при перечислении *выдает* (yield) очередной элемент последовательности. Во время перечисления возвращенного объекта запрос выполняется, и выданный элемент помещается в выходную последовательность. Таким образом, выполнение запроса откладывается.

Понятие *выдачи* (yield) связано с ключевым словом yield, которое было добавлено к языку C# для облегчения написания перечислителей.

Например, рассмотрим код, показанный в листинге 3.2.

Листинг 3.2. Тривиальный пример запроса

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
IEnumerable<string> items = presidents.Where(p => p.StartsWith("A"));
foreach(string item in items)
  Console.WriteLine(item);
```

Запрос, использующий операцию Where, на самом деле не запускается, когда выполняется строка, содержащая запрос. Вместо этого возвращается объект. И только во время перечисления элементов возвращенного объекта этот запрос Where выполняется. Это значит, что ошибка, возникающая в самом запросе, может быть не обнаружена до тех пор, пока не начнется перечисление.

На заметку! Ошибки запроса могут не обнаруживаться до тех пор, пока не начнется перечисление выходной последовательности.

Результат предыдущего запроса выглядит следующим образом:

```
Adams
Arthur
```

Запрос выполнился, как и ожидалось. Теперь внесем в него ошибку. В следующем коде (листинг 3.3) предпринимается попытка проиндексировать пятый символ каждого имени президента. Когда перечисление достигает элемента, длина которого меньше 5 символов, возникает исключение. Однако помните, что исключение не произойдет до тех пор, пока не начнется перечисление выходной последовательности.

Листинг 3.3. Тривиальный пример запроса с внутренней ошибкой

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
```

```
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
IEnumerable<string> items = presidents.Where(s => Char.IsLower(s[4]));
Console.WriteLine("After the query.");
foreach (string item in items)
  Console.WriteLine(item);
```

Этот код успешно компилируется, но при запуске дает такой результат:

```
After the query.
Adams
Arthur
Buchanan
Unhandled Exception: System.IndexOutOfRangeException: Index was outside the
bounds of the array.
Необработанное исключение: System.IndexOutOfRangeException: Индекс вышел за
границы массива.
...
```

Обратите внимание на вывод `After the query` (после запроса). Он не появляется до тех пор, пока в перечислении не дойдет очередь до четвертого элемента — `Bush`, где и возникает исключение. Урок, который можно отсюда извлечь, состоит в том, факт успешной компиляции запроса и кажущееся отсутствие проблем при его выполнении еще не говорит о том, что он свободен от ошибок.

Вдобавок, поскольку такого рода запросы, возвращающие `IEnumerable<T>`, являются отложенными, код определения запроса может быть вызван однажды и затем использован его многократно, с перечислением его результатов несколько раз. В случае изменения данных при каждом перечислении результатов будут выдаваться разные результаты. В листинге 3.4 показан пример отложенного запроса, где результат не кэшируется и может изменяться от одного перечисления к другому.

Листинг 3.4. Пример, демонстрирующий изменение результатов запроса между перечислениями

```
// Создать массив целых чисел.
int[] intArray = new int[] { 1,2,3 };
IEnumerable<int> ints = intArray.Select(i => i);

// Отобразить результаты.
foreach(int i in ints)
  Console.WriteLine(i);

// Изменить элемент в источнике данных.
intArray[0] = 5;
Console.WriteLine("---------");

// Снова отобразить результат.
foreach(int i in ints)
  Console.WriteLine(i);
```

Давайте более подробно рассмотрим, что здесь происходит. Когда вызывается операция `Select`, возвращается объект, хранящийся в переменной `ints` типа, реализующего интерфейс `IEnumerable<int>`. В этой точке запрос в действительности еще не выполняется, но хранится в объекте по имени `ints`. Другими словами, поскольку запрос еще не выполнен, последовательность целых чисел пока не существует, но этот объект `ints` знает, как получить последовательность, выполнив присвоенный ему запрос, которым в этом случае является операция `Select`.

Когда оператор `foreach` выполняется на `ints` в первый раз, объект `ints` производит запрос и получает последовательность по одному элементу за раз.

После этого в исходном массиве целых чисел изменяется один элемент. Затем снова запускается оператор foreach. Это заставляет ints снова выполнить запрос. Поскольку элемент в исходном массиве был изменен, а запрос выполнен снова, т.к. заново запущено перечисление ints, на этот раз возвращается измененный элемент.

Вызванный запрос вернул объект, реализующий IEnumerable<int>. Однако в большинстве случаев при обсуждении LINQ в этой книге, как и в других дискуссиях за рамками книги, обычно говорится, что запрос вернул последовательность целых чисел. Логически это верно, и в конечном счете так оно и есть. Но важно понимать, что происходит в действительности.

Ниже показан результат запуска этого кода:

```
1
2
3
---------
5
2
3
```

Обратите внимание, что несмотря на однократный вызов запроса, результаты двух перечислений отличаются. Это еще одно доказательство того, что запрос является отложенным. Если бы это было не так, то результаты двух перечислений совпали бы. Это может рассматриваться как преимущество, так и недостаток. Если не хотите, чтобы в таких ситуациях результаты отличались, воспользуйтесь одной из операций преобразования, которые не возвращают IEnumerable<T>, так что запрос получается не отложенным, а возвращают ToArray, ToList, ToDictionary или ToLookup, создавая различные структуры данных с кэшированными результатами, не изменяющимися с изменением источника данных.

В листинге 3.5 показан тот же код, что и в предыдущем примере, но запрос возвращает не IEnumerable<T>, а List<int> — за счет вызова операции ToList.

Листинг 3.5. Возврат List, так что запрос выполняется немедленно, а результаты кэшируются

```
// Создать массив целых чисел.
int[] intArray = new int[] { 1, 2, 3 };
List<int> ints = intArray.Select(i => i).ToList();

// Отобразить результаты.
foreach(int i in ints)
  Console.WriteLine(i);

// Изменить элемент в исходном массиве.
intArray[0] = 5;
Console.WriteLine("---------");

// Снова отобразить результаты.
foreach(int i in ints)
  Console.WriteLine(i);
```

Вот результат выполнения примера:

```
1
2
3
---------
1
2
3
```

Обратите внимание, что результаты, полученные от двух перечислений, одинаковы. Причина в том, что метод ToList не является отложенным, и запрос на самом деле выполняется в тот момент, когда он был вызван.

Возвращаясь к дискуссии об отличиях между этим примером и листингом 3.4, где операция Select отложена, следует отметить, что операция ToList из листинга 3.5 отложенной не является. Когда ToList вызывается в операторе запроса, она немедленно перечисляет объект, возвращенный оператором Select, в результате чего весь запрос перестает быть отложенным.

Делегаты Func

Некоторые стандартные операции запросов прототипированы на прием делегата Func в качестве аргумента. Это предотвращает явное объявление типов делегатов. Ниже приведены объявления делегата Func:

```
public delegate TR Func<TR>();
public delegate TR Func<T0, TR>(T0 a0);
public delegate TR Func<T0, T1, TR>(T0 a0, T1 a1);
public delegate TR Func<T0, T1, T2, TR>(T0 a0, T1 a1, T2 a2);
public delegate TR Func<T0, T1, T2, T3, TR>(T0 a0, T1 a1, T2 a2, T3 a3);
```

В каждом объявлении TR ссылается на возвращаемый тип данных. Обратите внимание, что тип возвращаемого аргумента TR находится в конце шаблона типов параметров для каждой перегрузки делегата Func. Другие параметры типа — T0, T1, T2 и T3 — ссылаются на входные параметры, переданные методу. Существует множество объявлений, потому что некоторые стандартные операции запросов имеют аргументы-делегаты, требующие больше параметров, чем другие. Взглянув на объявления, можно заметить, что ни одна из стандартных операций запросов не имеет аргумента-делегата, требующего более четырех входных параметров.

Давайте взглянем на один из прототипов операции Where:

```
public static IEnumerable<T> Where<T>(
  this IEnumerable<T> source,
  Func<T, bool> predicate);
```

Аргумент-предикат указан как Func<T, bool>. Отсюда видно, что метод-предикат или лямбда-выражение должны принимать один аргумент — параметр T и возвращать bool. Вы знаете это потому, что известен тип возврата, указанный в конце списка параметров шаблона.

Листинг 3.6. Пример использования одного из объявлений делегата Func

```
// Создать массив целых чисел.
int[] ints = new int[] { 1,2,3,4,5,6 };

// Объявление делегата.
Func<int, bool> GreaterThanTwo = i => i > 2;

// Выполнить запрос... Не забывайте об отложенных запросах!
IEnumerable<int> intsGreaterThanTwo = ints.Where(GreaterThanTwo);

// Отобразить результаты.
foreach(int i in intsGreaterThanTwo)
  Console.WriteLine(i);
```

Этот код вернет следующие результаты:

```
3
4
5
6
```

Алфавитный указатель стандартных операций запросов

В табл. 3.1 приведен упорядоченный по алфавиту список стандартных операций запросов. Поскольку эти операции будут разнесены по главам в соответствии с тем, являются они отложенными или нет, данная таблица поможет найти каждую операцию в остальных главах этой части.

Таблица 3.1. Алфавитный указатель стандартных операций запросов

Операция	Назначение	Отложенная?
`Aggregate`	Агрегация	
`All`	Квантификатор	
`Any`	Квантификатор	
`AsEnumerable`	Преобразование	Да
`Average`	Агрегация	
`Cast`	Преобразование	Да
`Concat`	Конкатенация	Да
`Contains`	Квантификатор	
`Count`	Агрегация	
`DefaultIfEmpty`	Элемент	Да
`Distinct`	Множество	Да
`ElementAt`	Элемент	
`ElementAtOrDefault`	Элемент	
`Empty`	Генерация	Да
`Except`	Множество	Да
`First`	Элемент	
`FirstOrDefault`	Элемент	
`GroupBy`	Группирование	Да
`GroupJoin`	Соединение	Да
`Interest`	Множество	Да
`Join`	Соединение	Да
`Last`	Элемент	
`LastOrDefault`	Элемент	
`LongCount`	Агрегация	
`Max`	Агрегация	
`Min`	Агрегация	
`OfType`	Преобразование	Да
`OrderBy`	Упорядочивание	Да

Окончание табл. 3.1

Операция	Назначение	Отложенная?
OrderByDescending	Упорядочивание	Да
Range	Генерация	Да
Repeat	Генерация	Да
Reverse	Упорядочивание	Да
Select	Проекция	Да
SelectMany	Проекция	Да
SequenceEqual	Эквивалентность	
Single	Элемент	
SingleOrDefault	Элемент	
Skip	Разбиение	Да
SkipWhile	Разбиение	Да
Sum	Агрегация	
Take	Разбиение	Да
TakeWhile	Разбиение	Да
ThenBy	Упорядочивание	Да
ThenByDescending	Упорядочивание	Да
ToArray	Преобразование	
ToDictionary	Преобразование	
ToList	Преобразование	
ToLookup	Преобразование	
Union	Множество	Да
Where	Ограничение	Да

История о двух синтаксисах

Поскольку записывать запросы LINQ можно с использованием либо синтаксиса выражений запросов, либо стандартного синтаксиса с точечной нотацией, может возникнуть вопрос о том, какой из них следует выбрать. В большинстве случаев это вопрос персональных предпочтений, если только необходимые стандартные операции запроса поддерживаются синтаксисом выражений запросов. Не все операции поддерживаются им, поэтому когда применяются неподдерживаемые операции, следует обратиться к стандартному синтаксису с вызовом методов через точечную нотацию.

Тем не менее, можно применять и смесь двух синтаксисов, заключая выражение запроса в скобки и добавляя вызов неподдерживаемой операции, как показано ниже:

```
IEnumerable<int> oddNumbers = (from n in nums
                               where n % 2 == 1
                               select n).Reverse();
```

Резюме

В этой главе было представлено понятие *последовательности* и ее формальный тип данных — `IEnumerable<T>`. Если пока вы чувствуете себя не комфортно с этой терминологией, со временем она станет вашей второй натурой. Просто воспринимайте `IEnumerable<T>` как последовательность объектов, на которых будут вызываться методы для выполнения с ними каких-то действий.

Однако наиболее существенным аспектом, о котором шла речь в этой главе, является важность отложенного выполнения запросов. Отложенное выполнение может работать как на вас, так и против вас. Глубокое понимание этого механизма и его важности — ключ к успеху. Неспроста стандартные операции запросов разнесены по отдельным главам на основе именно этой характеристики. Отложенные операции рассматриваются в главе 4, а не отложенные — в главе 5.

В следующей главе мы приступим к углубленному рассмотрению отложенных операций.

ГЛАВА 4

Отложенные операции

В предыдущей главе рассказывалось о том, что такое последовательности, о типах данных, представляющих их, а также об их влиянии на отложенное выполнение запросов. Из-за важности понимания отложенных запросов описания отложенных и не отложенных операций разнесены по разным главам, чтобы подчеркнуть, какие из стандартных операций запросов являются отложенными. В этой главе речь пойдет об отложенных операциях.

Отложенную операцию легко выделить, поскольку она должна возвращать тип `IEnumerable<T>` или `IOrderedEnumerable<T>`. Каждая из этих отложенных операций будет отнесена к определенной категории по своему назначению.

Чтобы кодировать и выполнять примеры этой главы, следует помнить об использовании директивы `using` со всеми необходимыми пространствами имен, ссылках на все нужные сборки, а также на общий код, разделяемый примерами.

Необходимые пространства имен

В примерах этой главы будут использоваться пространства имен `System.Linq`, `System.Collections`, `System.Collections.Generic` и `System.Data.Linq`. Поэтому потребуется добавить в код следующие директивы `using`, если их еще там нет:

```
using System.Linq;
using System.Collections;
using System.Collections.Generic;
using System.Data.Linq;
```

Загрузив сопровождающий код, вы увидите, что в дополнение к этим пространствам имен также добавлена директива `using` для пространства имен `System.Diagnostics`. Это не является необходимым для примеров этой главы. Это понадобится только для сопровождающего кода — из-за некоторого добавленного рабочего кода.

Необходимые сборки

В дополнение к типичным сборкам потребуется установить ссылку на сборку `System.Data.Linq.dll`.

Общие классы

В нескольких примерах этой главы понадобятся классы — для всесторонней демонстрации поведения той или иной операции. Ниже приведен список классов, которые будут использоваться в более чем одном примере.

Класс `Employee` представляет сотрудника. Для удобства он содержит статические методы, возвращающие `ArrayList` или массив сотрудников.

Разделяемый класс Employee

```
public class Employee
{
  public int id;
  public string firstName;
  public string lastName;

  public static ArrayList GetEmployeesArrayList()
  {
    ArrayList al = new ArrayList();
    al.Add(new Employee { id = 1, firstName = "Joe", lastName = "Rattz" });
    al.Add(new Employee { id = 2, firstName = "William", lastName = "Gates" });
    al.Add(new Employee { id = 3, firstName = "Anders", lastName = "Hejlsberg" });
    al.Add(new Employee { id = 4, firstName = "David", lastName = "Lightman" });
    al.Add(new Employee { id = 101, firstName = "Kevin", lastName = "Flynn" });
    return (al);
  }
  public static Employee[] GetEmployeesArray()
  {
    return ((Employee[])GetEmployeesArrayList().ToArray());
  }
}
```

Класс EmployeeOptionEntry представляет присуждение опционов на акции определенному сотруднику. Для удобства он содержит статический метод, возвращающий список назначенных опционов.

Разделяемый класс EmployeeOptionEntry

```
public class EmployeeOptionEntry
{
   public int id;
   public long optionsCount;
   public DateTime dateAwarded;
   public static EmployeeOptionEntry[] GetEmployeeOptionEntries()
   {
      EmployeeOptionEntry[] empOptions = new EmployeeOptionEntry[] {
         new EmployeeOptionEntry {
           id = 1,
           optionsCount = 2,
           dateAwarded = DateTime.Parse("1999/12/31") },
         new EmployeeOptionEntry {
           id = 2,
           optionsCount = 10000,
           dateAwarded = DateTime.Parse("1992/06/30") },
         new EmployeeOptionEntry {
           id = 2,
           optionsCount = 10000,
           dateAwarded = DateTime.Parse("1994/01/01") },
         new EmployeeOptionEntry {
           id = 3,
           optionsCount = 5000,
           dateAwarded = DateTime.Parse("1997/09/30") },
         new EmployeeOptionEntry {
           id = 2,
           optionsCount = 10000,
           dateAwarded = DateTime.Parse("2003/04/01") },
         new EmployeeOptionEntry {
           id = 3,
```

```
            optionsCount = 7500,
            dateAwarded = DateTime.Parse("1998/09/30") },
          new EmployeeOptionEntry {
            id = 3,
            optionsCount = 7500,
            dateAwarded = DateTime.Parse("1998/09/30") },
          new EmployeeOptionEntry {
            id = 4,
            optionsCount = 1500,
            dateAwarded = DateTime.Parse("1997/12/31") },
          new EmployeeOptionEntry {
            id = 101,
            optionsCount = 2,
            dateAwarded = DateTime.Parse("1998/12/31") }
        };
        return (empOptions);
    }
}
```

Организация отложенных операций по назначению

В настоящем разделе отложенные стандартные операции запросов организованы по их предназначению.

Ограничение

Операции ограничения (restriction) используются для включения или исключения элементов из входной последовательности.

Where

Операция `Where` используется для фильтрации элементов в последовательность.

Прототипы

Операция `Where` имеет два прототипа, которые описаны ниже.

Первый прототип `Where`

```
public static IEnumerable<T> Where<T>(
  this IEnumerable<T> source,
  Func<T, bool> predicate);
```

Этот прототип `Where` принимает входную последовательность и делегат метода-предиката, а возвращает объект, который при перечислении проходит по входной последовательности, выдавая элементы, для которых делегат метода-предиката возвращает `true`.

Поскольку это расширяющий метод, входная последовательность в действительности не передается, поскольку операция `Where` вызывается с использованием синтаксиса метода экземпляра.

На заметку! Благодаря расширяющим методам нет необходимости передавать первый аргумент в стандартную операцию запроса, первый аргумент которой помечен модификатором — ключевым словом `this`, при условии, что операция вызывается на объекте того же типа, что у первого аргумента.

При вызове `Where` передается делегат метода-предиката. Этот метод-предикат должен принимать тип `T` в качестве входного, где `T` — тип элементов, содержащихся во

входной последовательности, и возвращать bool. Операция Where вызовет метод-предикат для каждого элемента входной последовательности и передаст ему этот элемент. Если метод-предикат вернет true, то Where выдаст этот элемент в выходную последовательность Where. Если метод-предикат вернет false, то Where этого не сделает.

Второй прототип Where

```
public static IEnumerable<T> Where<T>(
  this IEnumerable<T> source,
  Func<T, int, bool> predicate);
```

Второй прототип Where идентичен первому, но с тем отличием, что он указывает на то, что делегат метода-предиката принимает дополнительный целочисленный аргумент. Этот аргумент будет индексом элемента во входной последовательности.

Нумерация индекса начинается с нуля, поэтому индексом первого элемента будет 0. Последний элемент имеет номер, соответствующий количеству элементов в последовательности минус 1.

На заметку! Помните, что нумерация индексов начинается с нуля.

Исключения

Если любой из аргументов равен null, генерируется исключение ArgumentNullException.

Примеры

В листинге 4.1 показан пример вызова первого прототипа.

Листинг 4.1. Пример вызова первого прототипа Where

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
IEnumerable<string> sequence = presidents.Where(p => p.StartsWith("J"));
foreach (string s in sequence)
  Console.WriteLine("{0}", s);
```

В приведенном примере ограничение последовательности с использованием первого прототипа операции Where осуществляется просто вызовом метода Where на последовательности, с передачей ему лямбда-выражения, возвращающего значение bool, которое указывает, должен ли элемент быть включен в выходную последовательность. В этом примере возвращаются только элементы, начинающиеся со строки "J". После нажатия <Ctrl+F5> код даст следующий результат:

```
Jackson
Jefferson
Johnson
```

Обратите внимание, что метод-предикат был передан через лямбда-выражение.

В листинге 4.2 показан код, вызывающий второй прототип операции Where. Обратите внимание, что в этой версии даже не используется сам элемент p, а только индекс i. Это обеспечит включение в выходную последовательность элементов с нечетными индексами.

Листинг 4.2. Пример вызова второго прототипа `Where`

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
IEnumerable<string> sequence = presidents.Where((p, i) => (i & 1) == 1);
foreach (string s in sequence)
  Console.WriteLine("{0}", s);
```

Нажатие <Ctrl+F5> выдаст следующий результат:

```
Arthur
Bush
Cleveland
Coolidge
Fillmore
Garfield
Harding
Hayes
Jackson
Johnson
Lincoln
McKinley
Nixon
Pierce
Reagan
Taft
Truman
Van Buren
Wilson
```

Проекция

Операции проекции возвращают выходную последовательность элементов, которая сгенерирована за счет выбора элементов или путем создания совершенно новых элементов, содержащих части элементов из входной последовательности. Тип данных элементов выходной последовательности может отличаться от типа элементов входной последовательности.

Select

Операция `Select` используется для создания выходной последовательности одного типа элементов из входной последовательности элементов другого типа. Эти типы не обязательно должны совпадать.

Прототипы

Существуют два прототипа этой операции, которые описаны ниже.

Первый прототип `Select`

```
public static IEnumerable<S> Select<T, S>(
  this IEnumerable<T> source,
  Func<T, S> selector);
```

Этот прототип `Select` принимает входную последовательность и делегат метода-селектора в качестве входных параметров, а возвращает объект, который при перечислении проходит по входной последовательности и выдает последовательность элементов типа `S`. Как упоминалось ранее, `T` и `S` могут быть как одного, так и разных типов.

При вызове `Select` делегат метода-селектора передается в аргументе `selector`. Метод-селектор должен принимать тип `T` в качестве входного, где `T` — тип элементов, содержащихся во входной последовательности, и возвращать элемент типа `S`. Операция `Select` вызовет метод-селектор для каждого элемента входной последовательности, передав ему этот элемент. Метод-селектор выберет интересующую часть входного элемента, создаст новый элемент — возможно, другого типа (даже анонимного) — и вернет его.

Второй прототип `Select`

```
public static IEnumerable<S> Select<T, S>(
  this IEnumerable<T> source,
  Func<T, int, S> selector);
```

В этом прототипе операции `Select` методу-селектору передается дополнительный целочисленный параметр. Это индекс, начинающийся с нуля, входного элемента во входной последовательности.

Исключения

Если любой из аргументов равен `null`, генерируется исключение `ArgumentNullException`.

Примеры

Пример вызова первого прототипа показан в листинге 4.3.

Листинг 4.3. Пример вызова первого прототипа `Select`

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
IEnumerable<string> sequence = presidents.Select(p => p.Length);
foreach (int item in nameLengths)
  Console.WriteLine(item);
```

Обратите внимание, что метод-селектор передается через лямбда-выражение. В данном случае лямбда-выражение вернет длину каждого элемента из входной последовательности. Также отметьте, что хотя тип входных элементов — строка, тип выходных элементов — `int`.

Нажатие <Ctrl+F5> выдаст следующий результат:

```
5
6
8
4
6
9
7
8
10
8
```

```
4
8
5
7
8
5
6
7
9
7
7
7
7
8
6
5
5
6
4
6
9
4
6
6
5
9
10
6
```

Это простой пример, потому что никакого класса не генерируется. В листинге 4.4 приведена более интересная демонстрация использования первого прототипа.

Листинг 4.4. Другой пример вызова первого прототипа `Select`

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
var nameObjs = presidents.Select(p => new { p, p.Length });
foreach (var item in nameObjs)
  Console.WriteLine(item);
```

Обратите внимание, что лямбда-выражение создает экземпляр нового анонимного типа. Компилятор динамически сгенерирует анонимный тип, который будет содержать `string p` и `int p.Length`, и метод-селектор вернет этот вновь созданный объект. Поскольку тип возвращаемого элемента является анонимным, сослаться на него нет возможности. Поэтому присвоить выходную последовательность `Select` экземпляру `IEnumerable` какого-то известного типа нельзя, как это делалось в первом примере, где она присваивалась переменной типа `IEnumerable<int>`, представляющей выходную последовательность. Поэтому выходная последовательность присваивается переменной, указанной с помощью ключевого слова `var`.

На заметку! Операции проекции, методы-селекторы которых создают анонимные типы для возврата, должны присваивать свою выходную последовательность переменной, тип которой указан с помощью ключевого слова `var`.

При запуске нажатием <Ctrl+F5> этот код производит следующий вывод:

```
{ p = Adams, Length = 5 }
{ p = Arthur, Length = 6 }
{ p = Buchanan, Length = 8 }
{ p = Bush, Length = 4 }
{ p = Carter, Length = 6 }
{ p = Cleveland, Length = 9 }
{ p = Clinton, Length = 7 }
{ p = Coolidge, Length = 8 }
{ p = Eisenhower, Length = 10 }
{ p = Fillmore, Length = 8 }
{ p = Ford, Length = 4 }
{ p = Garfield, Length = 8 }
{ p = Grant, Length = 5 }
{ p = Harding, Length = 7 }
{ p = Harrison, Length = 8 }
{ p = Hayes, Length = 5 }
{ p = Hoover, Length = 6 }
{ p = Jackson, Length = 7 }
{ p = Jefferson, Length = 9 }
{ p = Johnson, Length = 7 }
{ p = Kennedy, Length = 7 }
{ p = Lincoln, Length = 7 }
{ p = Madison, Length = 7 }
{ p = McKinley, Length = 8 }
{ p = Monroe, Length = 6 }
{ p = Nixon, Length = 5 }
{ p = Obama, Length = 5 }
{ p = Pierce, Length = 6 }
{ p = Polk, Length = 4 }
{ p = Reagan, Length = 6 }
{ p = Roosevelt, Length = 9 }
{ p = Taft, Length = 4 }
{ p = Taylor, Length = 6 }
{ p = Truman, Length = 6 }
{ p = Tyler, Length = 5 }
{ p = Van Buren, Length = 9 }
{ p = Washington, Length = 10 }
{ p = Wilson, Length = 6 }
```

С этим кодом связана одна проблема: управлять именами членов динамически сгенерированного анонимного класса нельзя. Однако, благодаря средству инициализации объектов C#, можно написать лямбда-выражение и задать имена членов анонимного класса, как показано в листинге 4.5.

Листинг 4.5. Третий пример вызова первого прототипа `Select`

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
```

```
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
var nameObjs = presidents.Select(p => new { LastName = p, Length = p.Length });
foreach (var item in nameObjs)
  Console.WriteLine("{0} имеет длину {1} символов.", item.LastName, item.Length);
```

Обратите внимание на указание имени для каждого члена в лямбда-выражении и обращение к каждому члену по имени в вызове метода `Console.WriteLine`. Вот результат запуска этого кода:

```
Adams имеет длину 5 символов.
Arthur имеет длину 6 символов.
Buchanan имеет длину 8 символов.
Bush имеет длину 4 символов.
Carter имеет длину 6 символов.
Cleveland имеет длину 9 символов.
Clinton имеет длину 7 символов.
Coolidge имеет длину 8 символов.
Eisenhower имеет длину 10 символов.
Fillmore имеет длину 8 символов.
Ford имеет длину 4 символов.
Garfield имеет длину 8 символов.
Grant имеет длину 5 символов.
Harding имеет длину 7 символов.
Harrison имеет длину 8 символов.
Hayes имеет длину 5 символов.
Hoover имеет длину 6 символов.
Jackson имеет длину 7 символов.
Jefferson имеет длину 9 символов.
Johnson имеет длину 7 символов.
Kennedy имеет длину 7 символов.
Lincoln имеет длину 7 символов.
Madison имеет длину 7 символов.
McKinley имеет длину 8 символов.
Monroe имеет длину 6 символов.
Nixon имеет длину 5 символов.
Obama имеет длину 5 символов.
Pierce имеет длину 6 символов.
Polk имеет длину 4 символов.
Reagan имеет длину 6 символов.
Roosevelt имеет длину 9 символов.
Taft имеет длину 4 символов.
Taylor имеет длину 6 символов.
Truman имеет длину 6 символов.
Tyler имеет длину 5 символов.
Van Buren имеет длину 9 символов.
Washington имеет длину 10 символов.
Wilson имеет длину 6 символов.
```

Для примера второго прототипа будет добавлен индекс, который передается методу-селектору, в тип элемента выходной последовательности, как показано в листинге 4.6.

Листинг 4.6. Пример вызова второго прототипа `Select`

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
```

```
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
var nameObjs = presidents.Select((p, i) => new { Index = I, LastName = p });
foreach (var item in nameObjs)
  Console.WriteLine("{0}. {1}", item.Index + 1, item.LastName);
```

Этот пример выводит номер индекса плюс единица, за которым следует имя. Код производит следующий результат (приведенный с сокращениями):

```
1. Adams
2. Arthur
3. Buchanan
4. Bush
5. Carter
...
35. Tyler
36. Van Buren
37. Washington
38. Wilson
```

SelectMany

Операция `SelectMany` используется для создания выходной последовательности с проекцией “один ко многим” из входной последовательности. В то время как операция `Select` возвращает один выходной элемент для каждого входного элемента, `SelectMany` вернет ноль или более выходных элементов для каждого входного.

Прототипы

Существуют два прототипа этой операции, которые рассматриваются ниже.

Первый прототип SelectMany

```
public static IEnumerable<S> SelectMany<T, S>(
  this IEnumerable<T> source,
  Func<T, IEnumerable<S>> selector);
```

Этот прототип операции получает входную последовательность элементов типа `T` и делегат метода-селектора, а возвращает объект, который при перечислении проходит по входной последовательности, получая каждый элемент индивидуально из входной последовательности и передавая его в метод-селектор. Последний затем возвращает объект, который во время перечисления выдает ноль или более элементов типа `S` в промежуточную выходную последовательность. Операция `SelectMany` вернет конкатенированную выходную последовательность при каждом вызове метода-селектора.

Второй прототип SelectMany

```
public static IEnumerable<S> SelectMany<T, S>(
  this IEnumerable<T> source,
  Func<T, int, IEnumerable<S>> selector);
```

Этот прототип ведет себя так же, как и первый, за исключением того, что методу-селектору дополнительно передается индекс, начинающийся с нуля, каждого элемента входной последовательности.

Исключения

Если любой из аргументов равен `null`, генерируется исключение `ArgumentNullException`.

Примеры

В листинге 4.7 показан пример вызова первого прототипа.

Листинг 4.7. Пример вызова первого прототипа `SelectMany`

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
IEnumerable<char> chars = presidents.SelectMany(p => p.ToArray());
foreach (char ch in chars)
  Console.WriteLine(ch);
```

В предыдущем примере метод-селектор принимает на входе строку и, вызывая метод `ToArray` на этой строке, возвращает массив символов, который становится выходной последовательностью типа `char`.

Таким образом, для одного элемента входной последовательности, которым в данном случае является экземпляр `string`, метод-селектор возвращает последовательность символов. Для каждой входной строки выводится последовательность `char`. Операция `SelectMany` соединяет все эти последовательности символов в единую последовательность, которую и возвращает. Вывод предыдущего кода выглядит следующим образом:

```
u
r
B
u
c
h
a
n
a
n
B
u
s
h
...
W
a
s
h
i
n
g
t
o
n
W
i
l
s
o
n
```

Это достаточно простой запрос, но не слишком демонстративный в смысле типового применения. В следующем примере будут использоваться общие классы `Employee` и `EmployeeOptionEntry`.

На массиве элементов Employee будет вызвана операция SelectMany, и для каждого элемента Employee в массиве делегат метода-селектора вернет ноль или более элементов созданного анонимного класса, содержащего id и optionsCount из массива элементов EmployeeOptionEntry для объекта Employee. Необходимый код приведен в листинге 4.8.

Листинг 4.8. Более сложный пример вызова первого прототипа SelectMany

```
Employee[] employees = Employee.GetEmployeesArray();
EmployeeOptionEntry[] empOptions = EmployeeOptionEntry.GetEmployeeOptionEntries();
var employeeOptions = employees
  .SelectMany(e => empOptions
                 .Where(eo => eo.id == e.id)
                 .Select(eo => new {
                             id = eo.id,
                             optionsCount = eo.optionsCount }));
foreach (var item in employeeOptions)
  Console.WriteLine(item);
```

В этом примере каждый сотрудник в массиве Employee передается в лямбда-выражение, в свою очередь, переданное операции SelectMany. Это лямбда-выражение затем извлечет каждый элемент EmployeeOptionEntry, чей id соответствует id текущего сотрудника, переданного ему посредством операции Where. Это эффективно связывает массив Employee с массивом EmployeeOptionEntry по их членам id. Операция Select лямбда-выражения затем создает анонимный объект, содержащий члены id и optionsCount для каждой соответствующей записи в массиве EmployeeOptionEntry. Это значит, что лямбда-выражение возвращает последовательность из нуля или более анонимных объектов для каждого переданного сотрудника. Это дает в результате последовательность или последовательности, которые операция SelectMany может соединить вместе.

Предыдущий код даст такой вывод:

```
{ id = 1, optionsCount = 2 }
{ id = 2, optionsCount = 10000 }
{ id = 2, optionsCount = 10000 }
{ id = 2, optionsCount = 10000 }
{ id = 3, optionsCount = 5000 }
{ id = 3, optionsCount = 7500 }
{ id = 3, optionsCount = 7500 }
{ id = 4, optionsCount = 1500 }
{ id = 101, optionsCount = 2 }
```

Пример в листинге 4.9, хотя и несколько надуман, демонстрирует вызов второго прототипа SelectMany.

Листинг 4.9. Пример вызова второго прототипа SelectMany

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
IEnumerable<char> chars = presidents
  .SelectMany((p, i) => i < 5 ? p.ToArray() : new char[] { });
foreach (char ch in chars)
  Console.WriteLine(ch);
```

Представленное здесь лямбда-выражение проверяет входной индекс и выводит массив символов из входной строки только в том случае, если значение индекса меньше 5. Это значит, что должны быть получены символы первых пяти входных строк, что и подтверждает следующий вывод:

```
A
d
a
m
s
A
r
t
h
A
d
a
m
s
A
r
t
h
u
r
B
u
c
h
a
n
a
n
B
u
s
h
C
a
r
t
e
r
```

Имейте в виду, что это лямбда-выражение не особо эффективно, если входных элементов много. Дело в том, что лямбда-выражение вызывается для *каждого* входного элемента. После обработки первых пяти входных элементов просто возвращается пустой массив. Для повышения производительности необходимо отдать предпочтение операции `Take`, которая будет описана в следующем разделе этой главы.

Операция `SelectMany` также полезна для соединения множества последовательностей в одну. Читайте об этом в разделе, посвященном операции `Concat`, далее в этой главе.

Разбиение

Операции разбиения (partitioning) позволяют вернуть выходную последовательность, которая является подмножеством входной последовательности.

Take

Операция Take возвращает указанное количество элементов из входной последовательности, начиная с ее начала.

Прототипы

Операция Take имеет один прототип, описанный ниже.

Прототип Take

```
public static IEnumerable<T> Take<T>(
  this IEnumerable<T> source,
  int count);
```

Этот прототип указывает, что Take принимает входную последовательность и целое число count, задающее количество элементов, которые нужно вернуть, и возвращает объект, который при перечислении выдает первые count элементов из входной последовательности.

Если значение count больше количества элементов во входной последовательности, тогда каждый элемент из нее попадает в выходную последовательность.

Исключения

Если аргумент source равен null, генерируется исключение ArgumentNullException.

Примеры

В листинге 4.10 показан пример вызова первого прототипа.

Листинг 4.10. Пример вызова единственного прототипа Take

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
IEnumerable<string> items = presidents.Take(5);
foreach (string item in items)
  Console.WriteLine(item);
```

Этот код вернет первые пять элементов из массива presidents. Полученный результат выглядит следующим образом:

```
Adams
Arthur
Buchanan
Bush
Carter
```

В листинге 4.9 приведен код, о котором говорилось, что он был бы более эффективным при условии использования операции Take вместо индекса, передаваемого лямбда-выражению. В листинге 4.11 показан эквивалентный код, в котором применяется операция Take. Он дает точно такой же результат, что и код в листинге 4.9, но при этом намного эффективнее.

Листинг 4.11. Другой пример вызова прототипа `Take`

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
IEnumerable<string> items = presidents.Take(5).SelectMany(s => s.ToArray());
foreach (char ch in chars)
  Console.WriteLine(ch);
```

Подобно примеру `SelectMany`, использующему второй прототип и показанному в листинге 4.9, предыдущий код возвращает следующие результаты:

```
A
d
a
m
s
A
r
t
h
u
r
B
u
c
h
a
n
a
n
B
u
s
h
C
a
r
t
e
r
```

Разница между этим примером кода и примером из листинга 4.9 состоит в том, что этот принимает только первые пять элементов из входной последовательности, и только они передаются в качестве входной последовательности в `SelectMany`. В примере кода из листинга 4.9 передаются все элементы `SelectMany`; там просто возвращается пустой массив для всех входных элементов кроме первых пяти.

TakeWhile

Операция `TakeWhile` возвращает элементы из входной последовательности, пока истинно некоторое условие, начиная с начала последовательности. Остальные входные элементы пропускаются.

Прототипы

Операция `TakeWhile` имеет два прототипа, описанные ниже.

Первый прототип `TakeWhile`

```
public static IEnumerable<T> TakeWhile<T>(
  this IEnumerable<T> source,
  Func<T, bool> predicate);
```

Эта операция `TakeWhile` принимает входную последовательность и делегат метода-предиката, а возвращает объект, перечисление по которому выдает элементы до тех пор, пока метод-предикат не вернет `false`. Метод-предикат принимает элементы по одному из входной последовательности и возвращает признак того, должен элемент включаться в выходную последовательность или нет. Если да, обработка входных элементов продолжается. Как только метод-предикат вернет `false`, никакие последующие входные элементы не обрабатываются.

Второй прототип `TakeWhile`

```
public static IEnumerable<T> TakeWhile<T>(
  this IEnumerable<T> source,
  Func<T, int, bool> predicate);
```

Этот прототип подобен первому, за исключением того, что метод-предикат получает вдобавок индекс элемента из входной последовательности, начинающийся с нуля.

Исключения

Если любой из аргументов равен `null`, генерируется исключение `ArgumentNullException`.

Примеры

В листинге 4.12 приведен пример вызова первого прототипа.

Листинг 4.12. Пример вызова первого прототипа `TakeWhile`

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
IEnumerable<string> items = presidents.TakeWhile(s => s.Length < 10);
foreach (string item in items)
  Console.WriteLine(item);
```

В приведенном коде входные элементы извлекаются до тех пор, пока их длина не превышает 10 символов. Ниже показан результат:

```
Adams
Arthur
Buchanan
Bush
Carter
Cleveland
Clinton
Coolidge
```

Элемент, который заставил операцию TakeWhile прекратить обработку входной последовательности — Eisenhower. Рассмотрим пример второго прототипа операции TakeWhile, приведенный в листинге 4.13.

Листинг 4.13. Пример вызова второго прототипа TakeWhile

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
IEnumerable<string> items = presidents
  .TakeWhile((s, i) => s.Length < 10 && i < 5);
foreach (string item in items)
  Console.WriteLine(item);
```

Код в этом примере прекращает выполнение, когда входной элемент превысит 9 символов в длину или когда будет достигнут шестой элемент — в зависимости от того, что произойдет раньше. Вот результат:

```
Adams
Arthur
Buchanan
Bush
Carter
```

В этом случае обработка остановилась по достижении шестого элемента.

Skip

Операция Skip пропускает указанное количество элементов из входной последовательности, начиная с ее начала, и выводит остальные.

Прототипы

Операция Skip имеет один прототип, описанный ниже.

Прототип Skip

```
public static IEnumerable<T> Skip<T>(
  this IEnumerable<T> source,
  int count);
```

Операция Skip получает входную последовательность и целое число count, задающее количество входных элементов, которое должно быть пропущено, и возвращает объект, который при перечислении пропускает первые count элементов и выводит все последующие элементы.

Исключения

Если аргумент source равен null, генерируется исключение ArgumentNullException.

Примеры

В листинге 4.14 приведен пример вызова операции Skip.

Листинг 4.14. Пример вызова единственного прототипа `Skip`

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
IEnumerable<string> items = presidents.Skip(1);
foreach (string item in items)
  Console.WriteLine(item);
```

В данном примере пропускается первый элемент. Обратите внимание, что в следующем выводе действительно пропущен первый элемент входной последовательности (`Adams`):

```
Arthur
Buchanan
Bush
...
Van Buren
Washington
Wilson
```

SkipWhile

Операция `SkipWhile` обрабатывает входную последовательность, пропуская элементы до тех пор, пока условие истинно, а затем выводит остальные в выходную последовательность.

Прототипы

У операции `SkipWhile` есть два прототипа, описанные ниже.

Первый прототип `SkipWhile`

```
public static IEnumerable<T> SkipWhile<T>(
  this IEnumerable<T> source,
  Func<T, bool> predicate);
```

Операция `SkipWhile` принимает входную последовательность и делегат метода-предиката, а возвращает объект, который при перечислении пропускает элементы до тех пор, пока метод-предикат возвращает `true`. Как только метод-предикат вернет `false`, операция `SkipWhile` начинает вывод всех прочих элементов. Метод-предикат принимает элементы входной последовательности по одному и возвращает признак того, должен ли элемент быть пропущен из входной последовательности.

Ниже описан второй прототип операции `SkipWhile`.

Второй прототип `SkipWhile`

```
public static IEnumerable<T> SkipWhile<T>(
  this IEnumerable<T> source,
  Func<T, int, bool> predicate);
```

Этот прототип подобен первому во всем, за исключением дополнительного параметра — индекса элемента из входной последовательности, начинающегося с нуля.

Исключения

Если любой из аргументов равен `null`, генерируется исключение `ArgumentNullException`.

Примеры

В листинге 4.15 показан пример вызова первого прототипа `SkipWhile`.

Листинг 4.15. Пример вызова первого прототипа `SkipWhile`

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
IEnumerable<string> items = presidents.SkipWhile(s => s.StartsWith("A"));
foreach (string item in items)
  Console.WriteLine(item);
```

В этом примере метод `SkipWhile` должен пропускать элементы до тех пор, пока они начинаются с буквы "A". Все остальные элементы выдаются в выходную последовательность. Результат предыдущего запроса выглядит так:

```
Buchanan
Bush
Carter
...
Van Buren
Washington
Wilson
```

Теперь рассмотрим пример использования второго прототипа `SkipWhile`, показанный в листинге 4.16.

Листинг 4.16. Пример вызова второго прототипа `SkipWhile`

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
IEnumerable<string> items = presidents
  .SkipWhile((s, i) => s.Length > 4 && i < 10);
foreach (string item in items)
  Console.WriteLine(item);
```

В данном примере входные элементы пропускаются до тех пор, пока их длина не превышает 4 символа или пока не будет достигнут десятый элемент. Остальные элементы выдаются в выходную последовательность. Вот как выглядит результат:

```
Bush
Carter
Cleveland
...
Van Buren
Washington
Wilson
```

Пропуск элементов был прекращен, как только встретился элемент `Bush`, поскольку его длина не превышает 4 символа, хотя его индексом является `3`.

Конкатенация

Операция конкатенации позволяет объединить несколько однотипных входных последовательностей в одну выходную.

Concat

Операция Concat соединяет две входные последовательности и выдает одну выходную последовательность.

Прототипы

Операция Concat имеет один прототип, который описан ниже.

Первый прототип Concat

```
public static IEnumerable<T> SkipWhile<T>(
  this IEnumerable<T> first,
  IEnumerable<T> second);
```

В этом прототипе две последовательности одного типа T — first и second — являются входными. Возвращается объект, который при перечислении проходит по первой последовательности, выдавая каждый ее элемент в выходную последовательность, за которым начинается перечисление второй входной последовательности с выдачей каждого ее элемента в ту же выходную последовательность.

Исключения

Если любой из аргументов равен null, генерируется исключение ArgumentNullException.

Примеры

В листинге 4.17 приведен пример использования операции Concat, а также операций Take и Skip.

Листинг 4.17. Пример вызова единственного прототипа Concat

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
IEnumerable<string> items = presidents.Take(5).Concat(presidents.Skip(5));
foreach (string item in items)
  Console.WriteLine(item);
```

Этот код берет пять первых членов из входной последовательности presidents и соединяет со всеми, *кроме* первых пяти входных элементов из последовательности presidents. В результате должна получиться последовательность, по содержимому идентичная последовательности presidents, следующего вида:

```
Adams
Arthur
Buchanan
Bush
Carter
Cleveland
Clinton
Coolidge
```

```
Eisenhower
Fillmore
Ford
Garfield
Grant
Harding
Harrison
Hayes
Hoover
Jackson
Jefferson
Johnson
Kennedy
Lincoln
Madison
McKinley
Monroe
Nixon
Obama
Pierce
Polk
Reagan
Roosevelt
Taft
Taylor
Truman
Tyler
Van Buren
Washington
Wilson
```

Альтернативный подход к соединению предусматривает вызов операции `SelectMany` на массиве последовательностей, как показано в листинге 4.18.

Листинг 4.18. Пример выполнения конкатенации с альтернативой применению операции `Concat`

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
IEnumerable<string> items = new[] {
                        presidents.Take(5),
                        presidents.Skip(5)
                       }
                       .SelectMany(s => s);
foreach (string item in items)
  Console.WriteLine(item);
```

В данном примере создается экземпляр массива, состоящего из двух последовательностей: одной, созданной вызовом операции `Take` на входной последовательности, и другой, созданной вызовом операции `Skip` на входной последовательности. Обратите внимание, что это подобно предыдущему примеру во всем, за исключением того, что на массиве последовательностей вызывается операция `SelectMany`. С учетом того, что операция `Concat` позволяет объединять только две последовательности, при наличии массива последовательностей продемонстрированный прием может оказаться удобнее.

Совет. Когда нужно объединить вместе более двух последовательностей, рассмотрите возможность использования подхода на основе `SelectMany`.

Результат получается тем же самым, что и при использовании операции `Concat`:

```
Adams
Arthur
Buchanan
Bush
Carter
Cleveland
Clinton
Coolidge
Eisenhower
Fillmore
Ford
Garfield
Grant
Harding
Harrison
Hayes
Hoover
Jackson
Jefferson
Johnson
Kennedy
Lincoln
Madison
McKinley
Monroe
Nixon
Obama
Pierce
Polk
Reagan
Roosevelt
Taft
Taylor
Truman
Tyler
Van Buren
Washington
Wilson
```

Упорядочивание

Операции упорядочивания позволяют выстраивать входные последовательности в определенном порядке. Важно отметить, что и `OrderBy`, и `OrderByDescending` требуют входной последовательности типа `IEnumerable<T>` и возвращают последовательность типа `IOrderedEnumerable<T>`. Передавать операциям `OrderBy` и `OrderByDescending` в качестве входной последовательности `IOrderedEnumerable<T>` нельзя. Причина в том, что последующие вызовы операций `OrderBy` и `OrderByDescending` не принимают во внимание порядок, созданный предыдущими вызовами `OrderBy` и `OrderByDescending`. Это значит, что передавать последовательность, возвращенную из `OrderBy` либо `OrderByDescending`, в последующий вызов операции `OrderBy` или `OrderByDescending` не имеет смысла.

Если требуется большая степень упорядочивания, чем возможно достичь с помощью одиночного вызова операции OrderBy или OrderByDescending, необходимо последовательно вызывать операции ThenBy или ThenByDescending. Вызовы ThenBy и ThenByDescending могут соединяться в цепочку, т.к. они принимают в качестве входной последовательности IOrderedEnumerable<T> и возвращают в качестве выходной последовательности тоже IOrderedEnumerable<T>.

Например, следующая последовательность вызовов не разрешена:

```
inputSequence.OrderBy(s => s.LastName).OrderBy(s => s.FirstName)...
```

Вместо нее должна использоваться такая цепочка:

```
inputSequence.OrderBy(s => s.LastName).ThenBy(s => s.FirstName)...
```

OrderBy

Операция OrderBy позволяет упорядочить входную последовательность на основе метода keySelector, который возвращает значение ключа для каждого входного элемента. Упорядоченная выходная последовательность IOrderedEnumerable<T> выдается в порядке возрастания на основе значений возвращенных ключей.

Сортировка, выполненная операцией OrderBy, определена как *неустойчивая*. Это значит, что она не сохраняет входной порядок элементов. Если два входных элемента поступают в операцию OrderBy в определенном порядке, и значения ключей этих двух элементов совпадают, их расположение в выходной последовательности может остаться прежним или поменяться, причем ни то, ни другое не гарантируется. Даже если все выглядит нормально, поскольку порядок определен как неустойчивый, всегда следует исходить из этого. Это значит, что никогда нельзя полагаться на порядок элементов, поступающих из операций OrderBy или OrderByDescending, для любого поля кроме указанного в вызове метода. Сохранение любого порядка, который существует в последовательности, передаваемой любой из этих операций, не может гарантироваться.

Прототипы

Операция OrderBy имеет два прототипа, описанные ниже.

Первый прототип OrderBy

```
public static IOrderedEnumerable<T> OrderBy<T, K>(
  this IEnumerable<T> source,
  Func<T, K> keySelector)
where
  K : IComparable<K>;
```

В этом прототипе операции OrderBy передается входная последовательность source и делегат метода keySelector, а возвращается объект, который при перечислении проходит входную коллекцию source, собирая все элементы и передавая каждый из них методу keySelector, таким образом, извлекая каждый ключ и упорядочивая последовательность на основе этих ключей.

Методу keySelector получает входной элемент типа T и возвращает поле внутри элемента, которое используется в качестве значения ключа типа K для этого входного элемента. Типы K и T могут быть одинаковыми или разными. Тип значения, возвращенного методом keySelector, должен реализовывать интерфейс IComparable.

Операция OrderBy имеет второй прототип, который описан ниже.

Второй прототип OrderBy

```
public static IOrderedEnumerable<T> OrderBy<T, K>(
  this IEnumerable<T> source,
  Func<T, K> keySelector,
  IComparer<K> comparer);
```

Этот прототип такой же, как первый, за исключением того, что он позволяет передавать объект-компаратор. Если используется эта версия операции `OrderBy`, то нет необходимости в том, чтобы тип `K` реализовывал интерфейс `IComparable`.

Исключения

Если любой из аргументов равен `null`, генерируется исключение `ArgumentNullException`.

Примеры

В листинге 4.19 показан пример вызова первого прототипа.

Листинг 4.19. Пример вызова первого прототипа `OrderBy`

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
IEnumerable<string> items = presidents.OrderBy(s => s.Length);
foreach (string item in items)
  Console.WriteLine(item);
```

Код в этом примере упорядочивает список президентов по длине их имен. Результат выглядит следующим образом:

```
Bush
Ford
Polk
Taft
Adams
Grant
Hayes
Nixon
Obama
Tyler
Arthur
Carter
Hoover
Monroe
Pierce
Reagan
Taylor
Truman
Wilson
Clinton
Harding
Jackson
Johnson
Kennedy
Lincoln
Madison
Buchanan
Coolidge
Fillmore
Garfield
Harrison
```

```
McKinley
Cleveland
Jefferson
Roosevelt
Van Buren
Eisenhower
Washington
```

Теперь испытаем пример второго прототипа с использованием собственного компаратора. Прежде чем продемонстрировать код, стоит взглянуть на интерфейс `IComparer`.

Интерфейс `IComparer<T>`

```
interface IComparer<T> {
  int Compare(T x, T y);
}
```

Интерфейс `IComparer` требует реализации единственного метода по имени `Compare`. Этот метод принимает два аргумента одного и того же типа `T` и возвращает значение `int` меньше нуля, если первый аргумент меньше второго, ноль — если аргументы эквивалентны, и значение больше нуля — если второй аргумент больше первого. Обратите внимание, насколько в этом интерфейсе и прототипе полезны обобщения C#.

В этом примере для ясности какой-либо компаратор по умолчанию не используется. Просто создается класс, реализующий интерфейс `IComparer`, который упорядочивает элементы на основе соотношения гласных и согласных.

Специальная реализация интерфейса `IComparer` для примера вызова второго прототипа `OrderBy`

```
public class MyVowelToConsonantRatioComparer : IComparer<string>
{
   public int Compare(string s1, string s2)
   {
      int vCount1 = 0;
      int cCount1 = 0;
      int vCount2 = 0;
      int cCount2 = 0;
      GetVowelConsonantCount(s1, ref vCount1, ref cCount1);
      GetVowelConsonantCount(s2, ref vCount2, ref cCount2);
      double dRatio1 = (double)vCount1/(double)cCount1;
      double dRatio2 = (double)vCount2/(double)cCount2;
      if(dRatio1 < dRatio2)
         return(-1);
      else if (dRatio1 > dRatio2)
         return(1);
      else
         return(0);
   }
   // Это общедоступный метод, так что код, использующий
   // данный компаратор, может получить нужные значения.
   public void GetVowelConsonantCount(string s,
                              ref int vowelCount,
                              ref int consonantCount)
   {
      // ВНИМАНИЕ: этот код предназначен только для демонстрационных целей.
      // Код всегда трактует 'y' и 'Y' как гласные, что с
      // лингвистической точки зрения, возможно, неверно.
      string vowels = "AEIOUY";

      // Инициализация счетчиков.
```

```
            vowelCount = 0;
            consonantCount = 0;
            // Преобразование в верхний регистр, чтобы избежать зависимости от регистра.
            string sUpper = s.ToUpper();
            foreach(char ch in sUpper)
            {
               if(vowels.IndexOf(ch) < 0)
                  consonantCount++;
               else
                  vowelCount++;
            }
            return;
         }
      }
```

Этот класс содержит два метода — `Compare` и `GetVowelConstantCount`. Метод `Compare` требуется интерфейсом `IComparer`. Метод `GetVowelConstantCount` необходим для внутреннего использования в `Compare` и позволяет получить количество гласных и согласных во входной строке. Также нужна возможность вызывать ту же логику за пределами метода `Compare`, чтобы можно было получать значения для отображения при проходе циклом по упорядоченной последовательности.

Что конкретно делает компаратор — не столь важно. Весьма маловероятно, что когда-либо понадобится определять соотношение гласных и согласных в строке, и еще менее вероятно, что придется сравнивать две строки на основе этого соотношения. Более важно то, как был создан класс, реализующий интерфейс `IComparer` за счет реализации метода `Compare`. Здесь можно наблюдать обычную реализацию метода `Compare` в блоке `if/else` в конце метода. Как видите, в этом блоке кода возвращается `-1`, `1` или `0`, что требует контракт интерфейса `IComparer`.

Теперь можно использовать этот код, как показано в листинге 4.20.

Листинг 4.20. Пример вызова второго прототипа `OrderBy`

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};

MyVowelToConsonantRatioComparer myComp = new MyVowelToConsonantRatioComparer();
IEnumerable<string> namesByVToCRatio = presidents
  .OrderBy((s => s), myComp);

foreach (string item in namesByVToCRatio)
{
   int vCount = 0;
   int cCount = 0;
   myComp.GetVowelConsonantCount(item, ref vCount, ref cCount);
   double dRatio = (double)vCount / (double)cCount;
   Console.WriteLine(item + " - " + dRatio + " - " + vCount + ":" + cCount);
}
```

Перед вызовом операции `OrderBy` создается экземпляр компаратора. Его экземпляр можно было бы создать в самом вызове метода `OrderBy`, но тогда не получилось бы ссылаться на него в цикле `foreach`.

Ниже показаны результаты запуска примера:

```
Grant - 0.25 - 1:4
Bush - 0.333333333333333 - 1:3
Ford - 0.333333333333333 - 1:3
Polk - 0.333333333333333 - 1:3
Taft - 0.333333333333333 - 1:3
Clinton - 0.4 - 2:5
Harding - 0.4 - 2:5
Jackson - 0.4 - 2:5
Johnson - 0.4 - 2:5
Lincoln - 0.4 - 2:5
Washington - 0.428571428571429 - 3:7
Arthur - 0.5 - 2:4
Carter - 0.5 - 2:4
Cleveland - 0.5 - 3:6
Jefferson - 0.5 - 3:6
Truman - 0.5 - 2:4
Van Buren - 0.5 - 3:6
Wilson - 0.5 - 2:4
Buchanan - 0.6 - 3:5
Fillmore - 0.6 - 3:5
Garfield - 0.6 - 3:5
Harrison - 0.6 - 3:5
McKinley - 0.6 - 3:5
Adams - 0.666666666666667 - 2:3
Nixon - 0.666666666666667 - 2:3
Tyler - 0.666666666666667 - 2:3
Kennedy - 0.75 - 3:4
Madison - 0.75 - 3:4
Roosevelt - 0.8 - 4:5
Coolidge - 1 - 4:4
Eisenhower - 1 - 5:5
Hoover - 1 - 3:3
Monroe - 1 - 3:3
Pierce - 1 - 3:3
Reagan - 1 - 3:3
Taylor - 1 - 3:3
Hayes - 1.5 — 3:2
Obama - 1.5 - 3:2
```

Как видите, элементы с меньшим соотношением гласных и согласных, выводятся первыми.

OrderByDescending

Эта операции прототипирована и ведет себя подобно операции `OrderBy`, но с тем отличием, что упорядочивает по убыванию.

Прототипы

Эта операция имеет два прототипа, которые описаны ниже.

Первый прототип `OrderByDescending`

```
public static IOrderedEnumerable<T> OrderByDescending<T, K>(
  this IEnumerable<T> source,
  Func<T, K> keySelector)
where
  K : IComparable<K>;
```

Этот прототип операции `OrderByDescending` ведет себя подобно своему эквиваленту `OrderBy`, за исключением того, что выполняет упорядочивание по убыванию.

Внимание! Сортировка, выполняемая `OrderBy` и `OrderByDescending`, является неустойчивой.

Операция `OrderByDescending` имеет второй прототип, описанный ниже.

Второй прототип `OrderByDescending`

```
public static IOrderedEnumerable<T> OrderByDescending<T, K>(
  this IEnumerable<T> source,
  Func<T, K> keySelector,
  IComparer<K> comparer);
```

Этот прототип похож на первый, но вдобавок позволяет передавать объект-компаратор. В случае использования этой версии операции `OrderByDescending` типу `K` не обязательно реализовывать интерфейс `IComparable`.

Исключения

Если любой из аргументов равен `null`, генерируется исключение `ArgumentNullException`.

Примеры

В примере применения первого прототипа, который приведен в листинге 4.21, список `presidents` упорядочивается по убыванию.

Листинг 4.21. Пример вызова первого прототипа `OrderByDescending`

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
IEnumerable<string> items = presidents.OrderByDescending(s => s);
foreach (string item in items)
  Console.WriteLine(item);
```

Как видите, имена президентов располагаются в порядке по убыванию:

```
Wilson
Washington
Van Buren
Tyler
Truman
Taylor
Taft
Roosevelt
Reagan
Polk
Pierce
Obama
Nixon
Monroe
McKinley
Madison
Lincoln
Kennedy
Johnson
Jefferson
```

```
Jackson
Hoover
Hayes
Harrison
Harding
Grant
Garfield
Ford
Fillmore
Eisenhower
Coolidge
Clinton
Cleveland
Carter
Bush
Buchanan
Arthur
Adams
```

Теперь рассмотрим пример использования второго прототипа `OredrByDescending`. За основу будет взят тот же самый пример, что и для второго прототипа операции `OrderBy`, за исключением того, что вместо вызова `OrderBy` будет выполнен вызов `OredrByDescending`. При этом применяется тот же самый компаратор `MyVowelToConsonantRatioComparer`. Код показан в листинге 4.22.

Листинг 4.22. Пример вызова второго прототипа `OrderByDescending`

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
MyVowelToConsonantRatioComparer myComp = new MyVowelToConsonantRatioComparer();
IEnumerable<string> namesByVToCRatio = presidents
  .OrderByDescending((s => s), myComp);
foreach (string item in namesByVToCRatio)
{
  int vCount = 0;
  int cCount = 0;
  myComp.GetVowelConsonantCount(item, ref vCount, ref cCount);
  double dRatio = (double)vCount / (double)cCount;
  Console.WriteLine(item + " - " + dRatio + " - " + vCount + ":" + cCount);
}
```

Результаты выглядят следующим образом:

```
Hayes - 1.5 - 3:2
Obama - 1.5 - 3:2
Coolidge - 1 - 4:4
Eisenhower - 1 - 5:5
Hoover - 1 - 3:3
Monroe - 1 - 3:3
Pierce - 1 - 3:3
Reagan - 1 - 3:3
Taylor - 1 - 3:3
Roosevelt - 0.8 - 4:5
Kennedy - 0.75 - 3:4
```

```
Madison - 0.75 - 3:4
Adams - 0.666666666666667 - 2:3
Nixon - 0.666666666666667 - 2:3
Tyler - 0.666666666666667 - 2:3
Buchanan - 0.6 - 3:5
Fillmore - 0.6 - 3:5
Garfield - 0.6 - 3:5
Harrison - 0.6 - 3:5
McKinley - 0.6 - 3:5
Arthur - 0.5 - 2:4
Carter - 0.5 - 2:4
Cleveland - 0.5 - 3:6
Jefferson - 0.5 - 3:6
Truman - 0.5 - 2:4
Van Buren - 0.5 - 3:6
Wilson - 0.5 - 2:4
Washington - 0.428571428571429 - 3:7
Clinton - 0.4 - 2:5
Harding - 0.4 - 2:5
Jackson - 0.4 - 2:5
Johnson - 0.4 - 2:5
Lincoln - 0.4 - 2:5
Bush - 0.333333333333333 - 1:3
Ford - 0.333333333333333 - 1:3
Polk - 0.333333333333333 - 1:3
Taft - 0.333333333333333 - 1:3
Grant - 0.25 - 1:4
```

Результаты те же, что и у эквивалентного примера `OrderBy`, только представлены в обратном порядке. Теперь элементы перечислены в порядке убывания соотношения гласных и согласных.

ThenBy

Операция `ThenBy` позволяет упорядочивать входную последовательность типа `IOrderedEnumerable<T>` на основе метода `keySelector`, который возвращает значение ключа. В результате выдается упорядоченная последовательность типа `IOrderedEnumerable<T>`.

На заметку! В отличие от большинства операций отложенных запросов LINQ to Objects, операции `ThenBy` и `ThenByDescending` принимают другой тип входных последовательностей — `IOrderedEnumerable<T>`. Это значит, что сначала должна быть вызвана операция `OrderBy` или `OrderByDescending` для создания последовательности `IOrderedEnumerable`, на которой можно затем вызывать операции `ThenBy` и `ThenByDescending`.

Сортировка, выполняемая операцией `ThenBy`, является *устойчивой*. Другими словами, она сохраняет входной порядок элементов с эквивалентными ключами. Если два входных элемента поступили в операцию `ThenBy` в определенном порядке, и ключевое значение обоих элементов одинаково, то порядок тех же выходных элементов гарантированно сохранится.

На заметку! В отличие от `OrderBy` и `OrderByDescending`, операции `ThenBy` и `ThenByDescending` выполняют устойчивую сортировку.

Прототипы

Операция `ThenBy` имеет два прототипа, которые описаны ниже.

Первый прототип ThenBy

```
public static IOrderedEnumerable<T> ThenBy<T, K>(
  this IOrderedEnumerable<T> source,
  Func<T, K> keySelector)
where
  K : IComparable<K>;
```

В этом прототипе операции ThenBy упорядоченная входная последовательность типа IOrderedEnumerable<T> передается операции ThenBy наряду с делегатом метода keySelector. Метод keySelector принимает элемент типа T и возвращает поле внутри элемента, которое используется в качестве значения ключа с типом K для входного элемента. Типы T и K могут быть как одинаковыми, так и различными. Значение, возвращенное методом keySelector, должно реализовывать интерфейс IComparable. Операция ThenBy упорядочит входную последовательность по возрастанию на основе возвращенных ключей.

Есть и второй прототип.

Второй прототип ThenBy

```
public static IOrderedEnumerable<T> ThenBy<T, K>(
  this IOrderedEnumerable<T> source,
  Func<T, K> keySelector,
  IComparer<K> comparer);
```

Этот прототип подобен первому, за исключением того, что принимает объект-компаратор. Если используется эта версия операции ThenBy, то типу K не обязательно реализовывать интерфейс IComparable.

Исключения

Если любой из аргументов равен null, генерируется исключение ArgumentNullException.

Примеры

В листинге 4.23 показан пример использования первого прототипа.

Листинг 4.23. Пример вызова первого прототипа ThenBy

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
IEnumerable<string> items = presidents.OrderBy(s => s.Length).ThenBy(s => s);
foreach (string item in items)
  Console.WriteLine(item);
```

Этот код сначала упорядочивает элементы по их длине, в данном случае — длине имени президента. Затем упорядочивает по самому элементу. В результате получается список имен, отсортированный по длине от меньшей к большей (по возрастанию), а затем — по имени в алфавитном порядке:

```
Bush
Ford
Polk
Taft
```

```
Adams
Grant
Hayes
Nixon
Obama
Tyler
Arthur
Carter
Hoover
Monroe
Pierce
Reagan
Taylor
Truman
Wilson
Clinton
Harding
Jackson
Johnson
Kennedy
Lincoln
Madison
Buchanan
Coolidge
Fillmore
Garfield
Harrison
McKinley
Cleveland
Jefferson
Roosevelt
Van Buren
Eisenhower
Washington
```

В примере применения второго прототипа `ThenBy` снова будет использоваться ранее показанный объект-компаратор `MyVowelToConsonantRatioComparer`. Однако перед вызовом `ThenBy` сначала потребуется вызвать либо `OrderBy`, либо `OrderByDescending`. В данном примере будет вызван `OrderBy`, чтобы упорядочить список по количеству символов в имени. Таким образом, имена будут упорядочены по возрастанию количества букв в них, а затем — внутри каждой группы с одинаковой длиной — по соотношению гласных и согласных. Код примера приведен в листинге 4.24.

Листинг 4.24. Пример вызова второго прототипа `ThenBy`

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
MyVowelToConsonantRatioComparer myComp = new MyVowelToConsonantRatioComparer();
IEnumerable<string> namesByVToCRatio = presidents
  .OrderBy(n => n.Length)
  .ThenBy((s => s), myComp);
foreach (string item in namesByVToCRatio)
{
```

```
    int vCount = 0;
    int cCount = 0;
    myComp.GetVowelConsonantCount(item, ref vCount, ref cCount);
    double dRatio = (double)vCount / (double)cCount;
    Console.WriteLine(item + " - " + dRatio + " - " + vCount + ":" + cCount);
}
```

Этот код выдаст следующий результат:

```
Bush - 0.333333333333333 - 1:3
Ford - 0.333333333333333 - 1:3
Polk - 0.333333333333333 - 1:3
Taft - 0.333333333333333 - 1:3
Grant - 0.25 - 1:4
Adams - 0.666666666666667 - 2:3
Nixon - 0.666666666666667 - 2:3
Tyler - 0.666666666666667 - 2:3
Hayes - 1.5 - 3:2
Obama - 1.5 - 3:2
Arthur - 0.5 - 2:4
Carter - 0.5 - 2:4
Truman - 0.5 - 2:4
Wilson - 0.5 - 2:4
Hoover - 1 - 3:3
Monroe - 1 - 3:3
Pierce - 1 - 3:3
Reagan - 1 - 3:3
Taylor - 1 - 3:3
Clinton - 0.4 - 2:5
Harding - 0.4 - 2:5
Jackson - 0.4 - 2:5
Johnson - 0.4 - 2:5
Lincoln - 0.4 - 2:5
Kennedy - 0.75 - 3:4
Madison - 0.75 - 3:4
Buchanan - 0.6 - 3:5
Fillmore - 0.6 - 3:5
Garfield - 0.6 - 3:5
Harrison - 0.6 - 3:5
McKinley - 0.6 - 3:5
Coolidge - 1 - 4:4
Cleveland - 0.5 - 3:6
Jefferson - 0.5 - 3:6
Van Buren - 0.5 - 3:6
Roosevelt - 0.8 - 4:5
Washington - 0.428571428571429 - 3:7
Eisenhower - 1 - 5:5
```

Как и следовало ожидать, имена сначала упорядочены по длине, а затем — по соотношению гласных и согласных.

ThenByDescending

Эта операция прототипирована и ведет себя подобно операции `ThenBy`, за исключением того, что упорядочивает элементы в по убыванию.

Прототипы

Операция имеет два прототипа, которые описаны ниже.

Первый прототип ThenByDescending

```
public static IOrderedEnumerable<T> ThenByDescending<T, K>(
  this IOrderedEnumerable<T> source,
  Func<T, K> keySelector)
where
  K : IComparable<K>;
```

Этот прототип операции ведет себя точно так же, как и первый прототип операции ThenBy, за исключением того, что упорядочивает по убыванию.

Операция ThenByDescending имеет и второй прототип.

Второй прототип ThenByDescending

```
public static IOrderedEnumerable<T> ThenByDescending<T, K>(
  this IOrderedEnumerable<T> source,
  Func<T, K> keySelector,
  IComparer<K> comparer);
```

Этот прототип похож на первый, но вдобавок принимает объект-компаратор. Если используется эта версия ThenByDescending, то нет необходимости в реализации интерфейса IComparable типов K.

Исключения

Если любой из аргументов равен null, генерируется исключение ArgumentNullException.

Примеры

В примере применения первого прототипа операции ThenByDescending используется тот же базовый подход, что и в примере вызова первого прототипа операции ThenBy, за исключением того, что вместо ThenBy будет вызываться ThenByDescending. Код показан в листинге 4.25.

Листинг 4.25. Пример вызова первого прототипа ThenByDescending

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
IEnumerable<string> items =
  presidents.OrderBy(s => s.Length).ThenByDescending(s => s);
foreach (string item in items)
  Console.WriteLine(item);
```

Этот код порождает вывод, где имена в пределах группы с одной и той же длиной сортируются по алфавиту в обратном порядке тому, который обеспечивает операция ThenBy:

```
Taft
Polk
Ford
Bush
Tyler
Obama
Nixon
Hayes
```

```
  Grant
  Adams
  Wilson
  Truman
  Taylor
  Reagan
  Pierce
  Monroe
  Hoover
  Carter
  Arthur
  Madison
  Lincoln
  Kennedy
  Johnson
  Jackson
  Harding
  Clinton
  McKinley
  Harrison
  Garfield
  Fillmore
  Coolidge
  Buchanan
  Van Buren
  Roosevelt
  Jefferson
  Cleveland
  Washington
  Eisenhower
```

Основу примера использования второго прототипа `ThenByDescending`, который показан в листинге 4.26, составляет пример применения второго прототипа операции `ThenBy`, но вместо `ThenBy` вызывается `ThenByDescending`.

Листинг 4.26. Пример вызова второго прототипа `ThenByDescending`

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
MyVowelToConsonantRatioComparer myComp = new MyVowelToConsonantRatioComparer();
IEnumerable<string> namesByVToCRatio = presidents
  .OrderBy(n => n.Length)
  .ThenByDescending((s => s), myComp);
foreach (string item in namesByVToCRatio)
{
  int vCount = 0;
  int cCount = 0;
  myComp.GetVowelConsonantCount(item, ref vCount, ref cCount);
  double dRatio = (double)vCount / (double)cCount;

  Console.WriteLine(item + " - " + dRatio + " - " + vCount + ":" + cCount);
}
```

Этот код порождает следующие результаты:

```
Bush - 0.333333333333333 - 1:3
Ford - 0.333333333333333 - 1:3
Polk - 0.333333333333333 - 1:3
Taft - 0.333333333333333 - 1:3
Hayes - 1.5 - 3:2
Obama - 1.5 - 3:2
Adams - 0.666666666666667 - 2:3
Nixon - 0.666666666666667 - 2:3
Tyler - 0.666666666666667 - 2:3
Grant - 0.25 - 1:4
Hoover - 1 - 3:3
Monroe - 1 - 3:3
Pierce - 1 - 3:3
Reagan - 1 - 3:3
Taylor - 1 - 3:3
Arthur - 0.5 - 2:4
Carter - 0.5 - 2:4
Truman - 0.5 - 2:4
Wilson - 0.5 - 2:4
Kennedy - 0.75 - 3:4
Madison - 0.75 - 3:4
Clinton - 0.4 - 2:5
Harding - 0.4 - 2:5
Jackson - 0.4 - 2:5
Johnson - 0.4 - 2:5
Lincoln - 0.4 - 2:5
Coolidge - 1 - 4:4
Buchanan - 0.6 - 3:5
Fillmore - 0.6 - 3:5
Garfield - 0.6 - 3:5
Harrison - 0.6 - 3:5
McKinley - 0.6 - 3:5
Roosevelt - 0.8 - 4:5
Cleveland - 0.5 - 3:6
Jefferson - 0.5 - 3:6
Van Buren - 0.5 - 3:6
Eisenhower - 1 - 5:5
Washington - 0.428571428571429 - 3:7
```

Как видите, имена упорядочены сначала по длине, а потом — в обратном порядке по соотношению гласных и согласных.

Reverse

Эта операция выводит последовательность того же типа, что и входная, но в обратном порядке.

Прототипы

Единственный прототип операции описан ниже.

Прототип Reverse

```
public static IEnumerable<T> Reverse<T>(
  this IEnumerable<T> source);
```

Эта операция возвращает объект, который при перечислении перебирает элементы входной последовательности и выдает их в выходную последовательность в обратном порядке.

Исключения

Если аргумент `source` равен `null`, генерируется исключение `ArgumentNullException`.

Примеры

Пример вызова прототипа операции `Reverse` приведен в листинге 4.27.

Листинг 4.27. Пример вызова прототипа `Reverse`

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
IEnumerable<string> items = presidents.Reverse();
foreach (string item in items)
  Console.WriteLine(item);
```

Код должен вывести элементы в порядке, обратном массиву `presidents`:

```
Wilson
Washington
Van Buren
...
Bush
Buchanan
Arthur
Adams
```

Соединение

Операции соединения (join) связывают вместе несколько последовательностей.

Join

Операция `Join` выполняет внутреннее соединение по эквивалентности двух последовательностей на основе ключей, извлеченных из каждого элемента этих последовательностей.

Прототипы

Операция `Join` имеет один прототип, описанный ниже.

Прототип `Join`

```
public static IEnumerable<V> Join<T, U, K, V>(
  this IEnumerable<T> outer,
  IEnumerable<U> inner,
  Func<T, K> outerKeySelector,
  Func<U, K> innerKeySelector,
  Func<T, U, V> resultSelector);
```

Обратите внимание, что первый аргумент метода имеет имя `outer`. Поскольку это расширяющий метод, последовательность, на которой вызывается операция `Join`, будет называться *внешней последовательностью*.

Операция `Join` возвращает объект, который при перечислении сначала проходит последовательность `inner` элементов типа `U`, вызывая метод `innerKeySelector` по одному разу для каждого элемента и сохраняя элемент, на который ссылается его ключ, в хеш-

таблице. Затем он проходит последовательность outer элементов типа T. По мере того, как возвращаемый объект перечисляет каждый объект последовательности outer, он вызывает метод outerKeySelector для получения ключа и извлекает соответствующий элемент последовательности inner из хеш-таблицы, используя этот ключ. Для каждой соответствующей пары элементов из последовательности outer и inner возвращаемый объект вызывает метод resultSelector, передавая ему и элемент outer, и соответствующий ему элемент inner. Метод resultSelector вернет экземпляр объекта типа V, который возвращаемый объект поместит в выходную последовательность типа V.

Порядок элементов последовательности outer сохраняется, равно как и порядок элементов inner в пределах каждого элемента outer.

Исключения

Если любой из аргументов равен null, генерируется исключение ArgumentNullException.

Примеры

Для того чтобы продемонстрировать пример применения этой операции, вместо массива presidents, используемого в большинстве примеров, будут использоваться два общих класса, определенные в начале этой главы — Employee и EmployeeOptionEntry.

Ниже приведен пример вызова операции Join с использованием этих классов. Чтобы повысить читабельность каждого аргумента Join, код в листинге 4.28 построен немного иначе, чем обычно.

Листинг 4.28. Пример вызова операции Join

```
Employee[] employees = Employee.GetEmployeesArray();
EmployeeOptionEntry[] empOptions = EmployeeOptionEntry.GetEmployeeOptionEntries();
var employeeOptions = employees
  .Join(
   empOptions,          // последовательность inner
   e => e.id,           // outerKeySelector
   o => o.id,           // innerKeySelector
   (e, o) => new        // resultSelector
           {
             id = e.id,
             name = string.Format("{0} {1}", e.firstName, e.lastName),
             options = o.optionsCount
           });
foreach (var item in employeeOptions)
  Console.WriteLine(item);
```

В коде сначала получается пара массивов для соединения с использованием двух общих классов. Поскольку операция Join вызывается на массиве employees, он становится внешней (outer) последовательностью, а empOptions — внутренней (inner) последовательностью. Ниже показаны результаты вызова операции Join:

```
{ id = 1, name = Joe Rattz, options = 2 }
{ id = 2, name = William Gates, options = 10000 }
{ id = 2, name = William Gates, options = 10000 }
{ id = 2, name = William Gates, options = 10000 }
{ id = 3, name = Anders Hejlsberg, options = 5000 }
{ id = 3, name = Anders Hejlsberg, options = 7500 }
{ id = 3, name = Anders Hejlsberg, options = 7500 }
{ id = 4, name = David Lightman, options = 1500 }
{ id = 101, name = Kevin Flynn, options = 2 }
```

Обратите внимание, что resultSelector создает анонимный класс в качестве типа элемента для результирующей выходной последовательности. То, что это анонимный класс, можно понять по вызову new без указания имени класса. Поскольку тип анонимный, выходную последовательность необходимо сохранить в переменной, тип которой указан с использованием ключевого слова var. Его нельзя указать как IEnumerable<>, поскольку нет именованного типа, который послужил бы параметром для объявления IEnumerable.

Совет. Когда последняя вызванная операция возвращает анонимный тип, для сохранения последовательности должно использоваться ключевое слово var.

GroupJoin

Операция GroupJoin выполняет групповое соединение двух последовательностей на основе ключей, извлеченных из каждого элемента последовательностей.

Операция GroupJoin работает очень похоже на Join, за исключением того, что Join передает один элемент внешней последовательности с одним соответствующим элементом внутренней последовательности методу resultSelector. Это значит, что множество элементов внутренней последовательности, соответствующих одному элементу внешней последовательности, приведут в результате к множеству вызовов resultSelector для этого элемента внешней последовательности. В случае операции GroupJoin все соответствующие элементы внутренней последовательности для определенного элемента внешней последовательности передаются в resultSelector как последовательность этого типа элемента, в результате чего метод resultSelector вызывается только по одному разу для каждого элемента внешней последовательности.

Прототипы

Эта операция имеет один прототип, описанный ниже.

Прототип GroupJoin

```
public static IEnumerable<V> GroupJoin<T, U, K, V>(
  this IEnumerable<T> outer,
  IEnumerable<U> inner,
  Func<T, K> outerKeySelector,
  Func<U, K> innerKeySelector,
  Func<T, IEnumerable<U>, V> resultSelector);
```

Обратите внимание, что первый аргумент метода имеет имя outer. Поскольку это расширяющий метод, последовательность, на которой вызывается операция Join, будет называться *внешней последовательностью*.

Операция GroupJoin возвращает объект, который при перечислении сначала проходит по последовательности inner элементов типа U, вызывая метод innerKeySelector по одному разу для каждого элемента и сохраняя элемент, на который ссылается ключ, в хеш-таблице. Затем возвращенный объект выполняет перечисление последовательности outer элементов типа T. По мере того, как возвращаемый объект перечисляет каждый элемент последовательности outer, он вызывает метод outerKeySelector для получения его ключа и извлекает соответствующий элемент последовательности inner из хеш-таблицы по этому ключу. Для каждого элемента последовательности outer возвращаемый объект вызывает метод resultSelector, передавая ему элемент outer и *последовательность* соответствующих элементов inner, так что resultSelector может вернуть экземпляр объекта типа V, куда возвращаемый объект поместит выходную последовательность типа V.

Порядок элементов последовательности `outer` сохраняется, равно как и порядок элементов `inner` в пределах каждого элемента `outer`.

Исключения

Если любой из аргументов равен `null`, генерируется исключение `ArgumentNullException`.

Примеры

В примере использования операции `GroupJoin` применяются те же классы `Employee` и `EmployeeOptionEntry`, что и в примере с `Join`. Код, приведенный в листинге 4.29, соединяет сотрудников с опционами и вычисляет сумму опционов для каждого сотрудника с помощью операции `GroupJoin`.

Листинг 4.29. Пример вызова операции `GroupJoin`

```
Employee[] employees = Employee.GetEmployeesArray();
EmployeeOptionEntry[] empOptions = EmployeeOptionEntry.GetEmployeeOptionEntries();

var employeeOptions = employees
  .GroupJoin(
    empOptions,
    e => e.id,
    o => o.id,
    (e, os) => new
             {
               id = e.id,
               name = string.Format("{0} {1}", e.firstName, e.lastName),
               options = os.Sum(o => o.optionsCount)
             });

foreach (var item in employeeOptions)
  Console.WriteLine(item);
```

Приведенный код почти идентичен примеру с операцией `Join`. Однако здесь второй аргумент лямбда-выражения, переданного в качестве метода `resultSelector`, назван `os`, в отличие от `o` в примере с `Join`. Причина в том, что в примере с `Join` в этом аргументе передавался единственный объект опциона — `o`, а в примере с `GroupJoin` передается последовательность объектов опционов сотрудника — `os`. Затем последнему члену экземпляра анонимного объекта, `optionsCount`, присваивается сумма последовательности из объектов опционов сотрудника с использованием операции `Sum`, которая будет описана в следующей главе (поскольку это не отложенная операция запроса). Пока достаточно знать, что операция `Sum` позволяет вычислить сумму каждого элемента или члена каждого элемента из входной последовательности.

Этот код выдаст следующие результаты:

```
{ id = 1, name = Joe Rattz, options = 2 }
{ id = 2, name = William Gates, options = 30000 }
{ id = 3, name = Anders Hejlsberg, options = 20000 }
{ id = 4, name = David Lightman, options = 1500 }
{ id = 101, name = Kevin Flynn, options = 2 }
```

Обратите внимание, что в этих результатах одна запись для каждого сотрудника содержит сумму всех записей о его опционах. Отличие от примера применения операции `Join` состоит в том, что там были отдельные записи для каждой записи опциона сотрудника.

Группирование

Операции группирования помогают объединять вместе элементы последовательности по общему ключу.

GroupBy

Операция GroupBy используется для группирования элементов входной последовательности.

Прототипы

Все прототипы операции GroupBy возвращают последовательность элементов IGrouping<K, T>. Здесь IGrouping<K, T> — интерфейс, который определен, как показано ниже.

Интерфейс IGrouping<K, T>

```
public interface IGrouping<K, T> : IEnumerable<T>
{
   K Key { get; }
}
```

Таким образом, IGrouping — это последовательность элементов типа T с ключами типа K. Существуют четыре прототипа, которые описаны ниже.

Первый прототип GroupBy

```
public static IEnumerable<IGrouping<K, T>> GroupBy<T, K>(
  this IEnumerable<T> source,
  Func<T, K> keySelector);
```

Этот прототип операции GroupBy возвращает объект, который при перечислении перебирает входную последовательность source, вызывает метод keySelector, собирает каждый элемент с его ключом и выдает последовательность экземпляров IGrouping<K, E>, где каждый элемент IGrouping<K, E> представляет собой последовательность элементов с одинаковым значением ключа. Значения ключа сравниваются с использованием компаратора эквивалентности по умолчанию — EqualityComparerDefault. Говоря иначе, возвращаемое значение метода GroupBy — это последовательность объектов IGrouping, каждый из которых содержит ключ и последовательность элементов из входной последовательности, имеющих тот же ключ.

Порядок экземпляров IGruoping будет тем же, что и вхождения ключей в последовательности source, и каждый элемент в последовательности IGruoping будет расположен в том порядке, в котором элементы находились в последовательности source.

Второй прототип GroupBy

```
public static IEnumerable<IGrouping<K, T>> GroupBy<T, K>(
  this IEnumerable<T> source,
  Func<T, K> keySelector,
  IEqualityComparer<K> comparer);
```

Этот прототип операции GroupBy похож на первый, за исключением того, что вместо использования компаратора эквивалентности по умолчанию, EqualityComparerDefault, указывается собственный компаратор.

Третий прототип GroupBy

```
public static IEnumerable<IGrouping<K, T>> GroupBy<T, K>(
  this IEnumerable<T> source,
  Func<T, K> keySelector,
  Func<T, E> elementSelector);
```

Этот прототип операции GroupBy подобен первому, за исключением того, что вместо помещения в выходную последовательность IGrouping всего исходного элемента целиком, можно указать, какая часть входного элемента должна попасть на выход, используя для этого elementSelector.

Четвертый прототип GroupBy

```
public static IEnumerable<IGrouping<K, T>> GroupBy<T, K>(
  this IEnumerable<T> source,
  Func<T, K> keySelector,
  Func<T, E> elementSelector,
  IEqualityComparer<K> comparer);
```

Этот прототип операции GroupBy является комбинацией второго и третьего, так что можно указывать компаратор в аргументе comparer и выводить элементы типа, отличного от входных элементов, используя аргумент elementSelector.

Исключения

Если любой из аргументов, кроме comparer, равен null, генерируется исключение ArgumentNullException.

Примеры

В примере применения первого прототипа GroupBy будет использоваться общий класс EmployeeOptionEntry. В коде, приведенном в листинге 4.30, записи EmployeeOptionEntry группируются по id и затем отображаются.

Листинг 4.30. Пример вызова первого прототипа GroupBy

```
EmployeeOptionEntry[] empOptions = EmployeeOptionEntry.GetEmployeeOptionEntries();
IEnumerable<IGrouping<int, EmployeeOptionEntry>> outerSequence =
  empOptions.GroupBy(o => o.id);

// Сначала перечисление по внешней последовательности IGrouping.
foreach (IGrouping<int, EmployeeOptionEntry> keyGroupSequence in outerSequence)
{
  Console.WriteLine("Записи опционов для сотрудника: " + keyGroupSequence.Key);

  // Теперь перечисление по сгруппированной последовательности
  // элементов EmployeeOptionEntry.
  foreach (EmployeeOptionEntry element in keyGroupSequence)
    Console.WriteLine("id={0} : optionsCount={1} : dateAwarded={2:d}",
      element.id, element.optionsCount, element.dateAwarded);
}
```

Обратите внимание, что перечисление выполняется по внешней последовательности с именем outerSequence, где каждый элемент — это объект, реализующий IGrouping, который содержит ключ, и последовательность элементов EmployeeOptionEntry, имеющих одинаковые ключи.

Ниже показаны результаты:

```
Записи опционов для сотрудника: 1
id=1 : optionsCount=2 : dateAwarded=12/31/1999
Записи опционов для сотрудника: 2
id=2 : optionsCount=10000 : dateAwarded=6/30/1992
id=2 : optionsCount=10000 : dateAwarded=1/1/1994
id=2 : optionsCount=10000 : dateAwarded=4/1/2003
Записи опционов для сотрудника: 3
id=3 : optionsCount=5000 : dateAwarded=9/30/1997
id=3 : optionsCount=7500 : dateAwarded=9/30/1998
```

```
id=3 : optionsCount=7500 : dateAwarded=9/30/1998
Записи опционов для сотрудника: 4
id=4 : optionsCount=1500 : dateAwarded=12/31/1997
Записи опционов для сотрудника: 101
id=101 : optionsCount=2 : dateAwarded=12/31/1998
```

Для примера использования второго прототипа `GroupBy` давайте предположим, что каждый сотрудник, `id` которого меньше `100`, является одним из основателей компании. Те, у кого `id` равен `100` и больше, основателями не являются. Задача состоит в том, чтобы вывести все записи опционов, сгруппированные по статусу сотрудника. Все опционы основателей будут сгруппированы вместе, а отдельно от них — опционы сотрудников, не являющихся основателями.

Теперь необходим компаратор эквивалентности, который сможет выполнить это ключевое сравнение. Этот компаратор должен реализовывать интерфейс `IEqualityComparer`. Прежде чем рассматривать сам компаратор, давайте сначала взглянем на интерфейс.

Интерфейс `IEqualityComparer<T>`

```
interface IEqualityComparer<T> {
  bool Equals(T x, T y);
  int GetHashCode(T x);
}
```

Этот интерфейс требует реализации двух методов — `Equals` и `GetHashCode`. Метод `Equals` принимает два объекта одного типа `T` и возвращает `true`, если два объекта считаются эквивалентными, и `false` — в противном случае. Метод `GetHashCode` принимает единственный объект и возвращает хеш-код типа `int` для этого объекта.

Хеш-код — это числовое значение, обычно вычисляемое математически на основе некоторой части данных объекта, известной как ключ, в целях уникальной идентификации объекта. Функция вычисляемого хеш-кода состоит в том, чтобы служить индексом в некоторой структуре данных для хранения объекта и последующего его нахождения. Поскольку допускается, что множество ключей производят один и тот же хеш-код, что делает его не уникальным, также есть необходимость в определении эквивалентности двух ключей. В этом предназначение метода `Equals`.

Ниже приведен код класса, реализующего интерфейс `IEqualityComparer`.

Класс, реализующий интерфейс `IEqualityComparer` для второго примера применения `GroupBy`

```
public class MyFounderNumberComparer : IEqualityComparer<int>
{
   public bool Equals(int x, int y)
   {
      return(isFounder(x) == isFounder(y));
   }

   public int GetHashCode(int i)
   {
      int f = 1;
      int nf = 100;
      return (isFounder(i) ? f.GetHashCode() : nf.GetHashCode());
   }

   public bool isFounder(int id)
   {
      return(id < 100);
   }
}
```

В дополнение к методам, которые требует интерфейс, был добавлен метод isFounder для определения того, является ли сотрудник основателем компании на базе приведенного выше определения. Это немного прояснит код. Этот метод сделан общедоступным, чтобы его можно было вызывать извне интерфейса, что и будет осуществляться в примере.

Компаратор эквивалентности рассматривает любой целочисленный идентификатор сотрудника, который меньше 100, как признак основателя компании, и если два идентификатора указывают на то, что оба являются основателями или оба основателями не являются, они считаются эквивалентными. Для основателей возвращается хеш-код 1, а для не основателей — 100, так что все основатели попадают в одну группу, а прочие — в другую.

Пример использования GroupBy приведен в листинге 4.31.

Листинг 4.31. Пример вызова второго прототипа GroupBy

```
MyFounderNumberComparer comp = new MyFounderNumberComparer();
EmployeeOptionEntry[] empOptions = EmployeeOptionEntry.GetEmployeeOptionEntries();

IEnumerable<IGrouping<int, EmployeeOptionEntry>> opts = empOptions
  .GroupBy(o => o.id, comp);

// Сначала перечисление по последовательности IGrouping.
foreach (IGrouping<int, EmployeeOptionEntry> keyGroup in opts)
{
   Console.WriteLine("Записи опционов для: " +
     (comp.isFounder(keyGroup.Key) ? "основатель" : "не основатель"));

   // Теперь перечисление по сгруппированной последовательности
   // элементов EmployeeOptionEntry.
   foreach (EmployeeOptionEntry element in keyGroup)
     Console.WriteLine("id={0} : optionsCount={1} : dateAwarded={2:d}",
       element.id, element.optionsCount, element.dateAwarded);
}
```

В этом примере экземпляр компаратора эквивалентности создается заблаговременно — в противоположность тому, чтобы делать это в вызове метода GroupBy — так что его можно использовать для вызова метода isFounder в цикле foreach.

Ниже показан результат работы этого кода:

```
Записи опционов для: основатель
id=1 : optionsCount=2 : dateAwarded=12/31/1999
id=2 : optionsCount=10000 : dateAwarded=6/30/1992
id=2 : optionsCount=10000 : dateAwarded=1/1/1994
id=3 : optionsCount=5000 : dateAwarded=9/30/1997
id=2 : optionsCount=10000 : dateAwarded=4/1/2003
id=3 : optionsCount=7500 : dateAwarded=9/30/1998
id=3 : optionsCount=7500 : dateAwarded=9/30/1998
id=4 : optionsCount=1500 : dateAwarded=12/31/1997
Записи опционов для: не основатель
id=101 : optionsCount=2 : dateAwarded=12/31/1998
```

Как видите, все записи опционов сотрудников, id которых меньше 100, попадают в группу основателей. В противном случае они попадают в группу не основателей.

Для примера применения третьего прототипа GroupBy предположим, что нас интересуют даты назначения опционов для каждого сотрудника. Этот код будет очень похож на код примера первого прототипа.

Итак, в листинге 4.32 вместо возврата последовательности сгруппированных объектов EmployeeOptionEntry будут группироваться даты.

Листинг 4.32. Пример вызова третьего прототипа `GroupBy`

```
EmployeeOptionEntry[] empOptions = EmployeeOptionEntry.GetEmployeeOptionEntries();
IEnumerable<IGrouping<int, DateTime>> opts = empOptions
  .GroupBy(o => o.id, e => e.dateAwarded);

// Сначала перечисление по последовательности IGrouping.
foreach (IGrouping<int, DateTime> keyGroup in opts)
{
   Console.WriteLine("Записи опционов для сотрудника: " + keyGroup.Key);
   // Теперь перечисление по сгруппированной последовательности элементов DateTime.
   foreach (DateTime date in keyGroup)
      Console.WriteLine(date.ToShortDateString());
}
```

Обратите внимание, что в вызове операции `GroupBy` второй аргумент `elementSelector` просто возвращает член `dateAwarded`. Поскольку возвращается `DateTime`, интерфейс `IGrouping` теперь служит для типа `DateTime` вместо `EmployeeOptionEntry`.

Как и можно было ожидать, выводятся даты назначений опционов, сгруппированные по сотрудникам:

```
Записи опционов для сотрудника: 1
12/31/1999
Записи опционов для сотрудника: 2
6/30/1992
1/1/1994
4/1/2003
Записи опционов для сотрудника: 3
9/30/1997
9/30/1998
9/30/1998
Записи опционов для сотрудника: 4
12/31/1997
Записи опционов для сотрудника: 101
12/31/1998
```

Для того чтобы продемонстрировать четвертый и последний прототип, необходимо использовать метод `elementSelector` и объект `comparer`, поэтому будет произведена комбинация примеров для второго и третьего прототипов. Задача состоит в группировании дат назначения опционов с тем, были они назначены основателям или нет, где основателями считаются сотрудники с `id` меньше `100`. Код показан в листинге 4.33.

Листинг 4.33. Пример вызова четвертого прототипа `GroupBy`

```
MyFounderNumberComparer comp = new MyFounderNumberComparer();
EmployeeOptionEntry[] empOptions = EmployeeOptionEntry.GetEmployeeOptionEntries();

IEnumerable<IGrouping<int, DateTime>> opts = empOptions
  .GroupBy(o => o.id, o => o.dateAwarded, comp);

// Сначала перечисление по последовательности IGrouping.
foreach (IGrouping<int, DateTime> keyGroup in opts)
{
   Console.WriteLine("Записи опционов для: " +
     (comp.isFounder(keyGroup.Key) ? "основатель" : "не основатель"));
   // Теперь перечисление по сгруппированной последовательности
   // элементов EmployeeOptionEntry.
   foreach (DateTime date in keyGroup)
     Console.WriteLine(date.ToShortDateString());
}
```

В выводе должны присутствовать просто даты, сгруппированные по основателям и не основателям:

```
Записи опционов для: основатель
12/31/1999
6/30/1992
1/1/1994
9/30/1997
4/1/2003
9/30/1998
9/30/1998
12/31/1997
Записи опционов для: не основатель
12/31/1998
```

Множества

Операции множеств используются для выполнения математических операций с множествами на последовательностях.

Совет. Свойства операций множеств, описанные в этой главе, не работают правильно с `DataSet`. В случае `DataSet` используйте прототипы, описанные в главе 10.

Distinct

Операция `Distinct` удаляет дублированные элементы из входной последовательности.

Прототипы

У операции `Distinct` есть один прототип, описанный ниже.

Прототип `Distinct`

```
public static IEnumerable<T> Distinct<T>(
  this IEnumerable<T> source);
```

Эта операция возвращает объект, перечисляющий элементы входной последовательности `source` и выдающий последовательность, в которой каждый элемент не эквивалентен предыдущим выданным. Эквивалентность элементов определяется методами `GetHashCode` и `Equals`.

Исключения

Если аргумент `source` равен `null`, генерируется исключение `ArgumentNullException`.

Примеры

В рассматриваемом здесь примере сначала отображается количество элементов массива `presidents`, затем массив `presidents` соединяется сам с собой, отображается количество элементов результирующей объединенной последовательности, вызывается операция `Distinct` на этой объединенной последовательности и, наконец, отображается количество элементов последовательности, полученной от `Distinct`, которая должна совпасть с начальным массивом `presidents`.

Для определения длины двух сгенерированных последовательностей используется стандартная операция запроса `Count`. Поскольку это не отложенная операция, она рассматривается в следующей главе. А пока просто знайте, что она возвращает количество элементов в последовательности, на которой вызвана.

Код примера приведен в листинге 4.34.

Листинг 4.34. Пример вызова операции `Distinct`

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};

// Отобразить количество элементов в массиве presidents.
Console.WriteLine("Количество элементов в presidents: " + presidents.Count());

// Соединить presidents с самим собой. Теперь каждый элемент
// представлен в последовательности дважды.
IEnumerable<string> presidentsWithDupes = presidents.Concat(presidents);

// Отобразить количество элементов в объединенной последовательности.
Console.WriteLine("Количество элементов в presidentsWithDupes: "
  + presidentsWithDupes.Count());

// Исключить дубликаты и отобразить количество элементов.
IEnumerable<string> presidentsDistinct = presidentsWithDupes.Distinct();
Console.WriteLine("Количество элементов в presidentsDistinct: "
  + presidentsDistinct.Count());
```

Количество элементов в последовательности `presidentsDistinct` должно совпадать с количеством элементов в последовательности `presidents`. Результат подтверждает это:

```
Количество элементов в presidents: 38
Количество элементов в presidentsWithDupes: 76
Количество элементов в presidentsDistinct: 38
```

Union

Операция `Union` возвращает объединение множеств из двух исходных последовательностей.

Прототипы

У этой операции имеется один прототип, описанный ниже.

Прототип `Union`

```
public static IEnumerable<T> Union<T>(
  this IEnumerable<T> first,
  IEnumerable<T> second);
```

Эта операция возвращает объект, который сначала перечисляет элементы последовательности по имени `first`, выдавая последовательность, в которой каждый элемент не эквивалентен предыдущим выданным, затем перечисляет вторую входную последовательность `second`, опять-таки, выдавая последовательность без повторений. Эквивалентность элементов определяется методами `GetHashCode` и `Equals`.

Исключения

Если любой из аргументов равен `null`, генерируется исключение `ArgumentNullException`.

Примеры

Чтобы продемонстрировать разницу между операцией `Union` и описанной выше операцией `Concat`, в примере, представленном в листинге 4.35, создаются последовательности `first` и `second` из массива `presidents`, что приведет к дублированию пятого элемента в обеих последовательностях. Затем отображается количество элементов в массиве `presidents`, а также в последовательностях `first` и `second`, наряду с количеством элементов в конкатенированной и объединенной последовательностях.

Листинг 4.35. Пример вызова операции `Union`

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
IEnumerable<string> first = presidents.Take(5);
IEnumerable<string> second = presidents.Skip(4);

// Поскольку пропущены 4 элемента, пятый элемент
// должен присутствовать в обеих последовательностях.
IEnumerable<string> concat = first.Concat<string>(second);
IEnumerable<string> union = first.Union<string>(second);
Console.WriteLine("Количество элементов в массиве presidents: "
  + presidents.Count());
Console.WriteLine("Количество элементов в последовательности first: "
  + first.Count());
Console.WriteLine("Количество элементов в последовательности second: "
  + second.Count());
Console.WriteLine("Количество элементов в последовательности concat: "
  + concat.Count());
Console.WriteLine("Количество элементов в последовательности union: "
  + union.Count());
```

В конечном итоге последовательность `concat` должна иметь на один элемент больше, чем массив `presidents`. Последовательность `union` должна содержать то же количество элементов, что и массив `presidents`. Это доказывают результаты выполнения кода:

```
Количество элементов в массиве presidents: 38
Количество элементов в последовательности first: 5
Количество элементов в последовательности second: 34
Количество элементов в последовательности concat: 39
Количество элементов в последовательности union: 38
```

Intersect

Операция `Intersect` возвращает пересечение множеств из двух исходных последовательностей.

Прототипы

Операция `Intersect` имеет один прототип, описанный ниже.

Прототип `Intersect`

```
public static IEnumerable<T> Intersect<T>(
  this IEnumerable<T> first,
  IEnumerable<T> second);
```

Эта операция возвращает объект, который сначала перечисляет элементы последовательности по имени `first`, выбирая оттуда каждый элемент, который не эквивалентен предыдущему выбранному элементу. Затем он перечисляет вторую входную последовательность, помечая любой элемент, имеющийся в обеих последовательностях, для включения в выходную последовательность. Затем осуществляется проход по помеченным элементам, с помещением их в выходную последовательность в том порядке, в котором они были собраны. Эквивалентность элементов определяется с помощью методов `GetHashCode` и `Equals`.

Исключения

Если любой из аргументов равен `null`, генерируется исключение `ArgumentNullException`.

Примеры

Чтобы продемонстрировать применение операции `Intersect`, в листинге 4.36 используются операции `Take` и `Skip` для генерации двух последовательностей и получения некоторого их перекрытия, как в примере с `Union`, где был намеренно дублирован пятый элемент. После вызова операции `Intersect` на этих двух сгенерированных последовательностях в возвращаемой последовательности `intersect` должен оказаться только дублированный пятый элемент. Выводится количество элементов в массиве `presidents` и всех последовательностях. И, наконец, производится проход по последовательности `intersect` с отображением каждого элемента на консоль. В ней должен оказаться только один пятый элемент массива `presidents`.

Листинг 4.36. Пример вызова операции `Intersect`

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
IEnumerable<string> first = presidents.Take(5);
IEnumerable<string> second = presidents.Skip(4);

// Поскольку пропущены 4 элемента, пятый элемент
// должен присутствовать в обеих последовательностях.
IEnumerable<string> intersect = first.Intersect(second);
Console.WriteLine("Количество элементов в массиве presidents: "
  + presidents.Count());
Console.WriteLine("Количество элементов в последовательности first: "
  + first.Count());
Console.WriteLine("Количество элементов в последовательности second: "
  + second.Count());
Console.WriteLine("Количество элементов в последовательности intersect: "
  + intersect.Count());

// Просто ради интереса выводится содержимое последовательности intersection,
// в которой должен присутствовать только один пятый элемент из presidents.
foreach (string name in intersect)
  Console.WriteLine(name);
```

В конечном итоге должна быть получена последовательность `intersect`, которая состоит только из одного элемента, содержащего дублированный пятый элемент массива `presidents` — `Carter`:

```
Количество элементов в массиве presidents: 38
Количество элементов в последовательности first: 5
Количество элементов в последовательности second: 34
Количество элементов в последовательности intersect: 1
Carter
```

Итак, LINQ работает. Насколько часто вам приходилось ранее выполнять операции над множествами элементов из двух коллекций? Было ли это трудно? Благодаря LINQ, все сложности в прошлом.

Except

Операция `Except` возвращает последовательность, содержащую все элементы первой последовательности, которых нет во второй последовательности.

Прототипы

Эта операция имеет один прототип, описанный ниже.

Прототип `Except`

```
public static IEnumerable<T> Except<T>(
  this IEnumerable<T> first,
  IEnumerable<T> second);
```

Эта операция возвращает объект, который при перечислении перебирает элементы входной последовательности по имени `second`, собирая все элементы, которые не эквивалентны ранее собранным. Затем происходит перечисление входной последовательности `first`, с выдачей каждого ее элемента, которого нет в коллекции из второй последовательности. Эквивалентность одного элемента другому определяется с использованием их методов `GetHashCode` и `Equals`.

Исключения

Если любой из аргументов равен `null`, генерируется исключение `ArgumentNullException`.

Примеры

В этом примере используется массив `presidents`, который применялся в большинстве ранее показанных примеров. Представим сценарий, когда есть первичный источник данных — массив `presidents`, с вхождениями, которые должны быть подвергнуты какой-то обработке. По мере завершения обработки каждого вхождения, необходимо добавлять его в коллекцию обработанных — так, чтобы в ситуации, когда обработка начинается заново, можно было бы воспользоваться операцией `Except` и произвести последовательность исключаемых из первичной последовательности элементов данных, чтобы не обрабатывать их повторно.

Для этого примера, представленного в листинге 4.37, было принято, что первые четыре вхождений уже обработаны. Чтобы получить последовательность, содержащую первые четыре элемента массива `presidents`, на нем просто выполняется операция `Take`.

Листинг 4.37. Пример вызова операции `Intersect`

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
```

```
// Сначала сгенерировать последовательность обработанных элементов.
IEnumerable<string> processed = presidents.Take(4);
IEnumerable<string> exceptions = presidents.Except(processed);
foreach (string name in exceptions)
  Console.WriteLine(name);
```

В этом примере полученный результат будет содержать имена из массива `presidents`, расположенные после четвертого элемента — `Bush`:

```
Carter
Cleveland
Clinton
Coolidge
Eisenhower
Fillmore
Ford
Garfield
Grant
Harding
Harrison
Hayes
Hoover
Jackson
Jefferson
Johnson
Kennedy
Lincoln
Madison
McKinley
Monroe
Nixon
Obama
Pierce
Polk
Reagan
Roosevelt
Taft
Taylor
Truman
Tyler
Van Buren
Washington
Wilson
```

Судя по выводу, все работает, как следовало ожидать.

Преобразование

Операции преобразования предоставляют простой и удобный способ преобразования последовательностей в другие типы коллекций.

Cast

Операция `Cast` используется для приведения каждого элемента входной последовательности в выходную последовательность указанного типа.

Прототипы

Операция `Cast` имеет один прототип, описанный ниже.

Прототип Cast

```
public static IEnumerable<T> Cast<T>(
  this IEnumerable source);
```

Первое, что следует отметить в операции Cast — это то, что ее первый аргумент по имени source имеет тип IEnumerable, а не IEnumerable<T>, тогда как большинство отложенных стандартных операций запросов принимают первый аргумент типа IEnumerable<T>. Это связано с тем, что операция Cast предназначена для вызова на классах, реализующих интерфейс IEnumerable, а не IEnumerable<T>. В частности, речь идет об унаследованных коллекциях, разработанных до появления версии C# 2.0 и обобщений.

Операцию Cast можете вызывать на унаследованных коллекциях C# до тех пор, пока они реализуют IEnumerable, и при этом будет создана выходная последовательность IEnumerable<T>. Поскольку большинство стандартных операций запросов работает на последовательностях типа IEnumerable<T>, для преобразования к ним унаследованной коллекции понадобится вызвать метод вроде этого или, возможно, операцию OfType, которая рассматривается ниже.

Эта операция вернет объект, который при перечислении проходит по исходной коллекции данных, преобразуя каждый элемент к типу T. Если элемент не может быть приведен к типу T, генерируется исключение. Из-за этого данная операция должна вызываться, только когда точно известно, что каждый элемент входной последовательности может быть преобразован в тип T.

Совет. Пытаясь выполнять запросы LINQ на унаследованных коллекциях, не забудьте вызывать на них операцию Cast или OfType, чтобы создать последовательность IEnumerable<T>, на которой затем можно вызывать стандартные операции запросов.

Исключения

Если аргумент source равен null, генерируется исключение ArgumentNullException. Если элемент входной коллекции не может быть приведен к типу T, генерируется исключение InvalidCastException.

Примеры

В приведенном ниже примере с помощью метода GetEmployeesArrayList общего класса Employee будет возвращена унаследованная, необобщенная коллекция ArrayList.

В листинге 4.38 содержится код, демонстрирующий, как тип данных элементов ArrayList может быть приведен к элементам последовательности IEnumerable<T>.

Листинг 4.38. Код, преобразующий коллекцию ArrayList в последовательность IEnumerable<T>, которая может использоваться со стандартными операциями запросов

```
ArrayList employees = Employee.GetEmployeesArrayList();
Console.WriteLine("Тип данных employees: " + employees.GetType());

var seq = employees.Cast<Employee>();
Console.WriteLine("Тип данных seq: " + seq.GetType());

var emps = seq.OrderBy(e => e.lastName);
foreach (Employee emp in emps)
  Console.WriteLine("{0} {1}", emp.firstName, emp.lastName);
```

Сначала производится вызов метода GetEmployeesArrayList, который вернет коллекцию ArrayList объектов Employee и затем отображается тип данных переменной employees. Далее этот ArrayList преобразуется в последовательность IEnumerable<T>, вызывая для этого операцию Cast, после чего отображается тип данных возвращенной последовательности. И, наконец, для демонстрации, что все работает, элементы возвращенной последовательности выводятся на консоль.

Ниже показан результат выполнения этого кода:

```
Тип данных employees: System.Collections.ArrayList
Тип данных seq: System.Linq.Enumerable+<CastIterator>d__b0`1[LINQChapter4.Employee]
Kevin Flynn
William Gates
Anders Hejlsberg
David Lightman
Joe Rattz
```

Как видите, типом данных переменной employees является ArrayList. Немного сложнее выяснить тип данных seq. Определенно можно сказать, что он отличается и выглядит как последовательность. Кроме того, в наименовании его типа присутствует слово CastIterator. Вы должны помнить, что отложенные операции не возвращают немедленно выходную последовательность, а возвращают объект, перечисление по которому будет выдавать элементы в выходную последовательность. Тип данных переменной seq в предыдущем примере указывает именно на объект такого рода. Тем не менее, это деталь реализации, которая в будущем может измениться.

Внимание! Операция Cast попытается привести каждый элемент входной последовательности к указанному типу. Если любой из этих элементов не может быть приведен к указанному типу, будет сгенерировано исключение InvalidCastException. Если существует вероятность присутствия разнотипных элементов в исходной коллекции, применяйте вместо Cast операцию OfType.

OfType

Операция OfType используется для построения выходной последовательности, содержащей только те элементы, которые могут быть успешно преобразованы к указанному типу.

Прототипы

Эта операция имеет один прототип, описанный ниже.

Прототип OfType

```
public static IEnumerable<T> OfType<T>(
  this IEnumerable source);
```

Первое, что следует отметить в операции OfType — это то, что ее первый аргумент по имени source, как и у операции Cast, имеет тип IEnumerable, а не IEnumerable<T>. Большинство первых аргументов отложенных стандартных операций запросов имеет тип IEnumerable<T>. Это объясняется тем, что операция OfType предназначена для вызова на классах, реализующих интерфейс IEnumerable, а не IEnumerable<T>. В частности, речь идет обо всех унаследованных коллекциях, разработанных до появления версии C# 2.0 и обобщений.

Таким образом, вызывать операцию OfType на унаследованной коллекции можно до тех пор, пока она реализует IEnumerable, и при этом будет создана выходная последовательность IEnumerable<T>. Поскольку большинство стандартных операций запросов работает на последовательностях типа IEnumerable<T>, для преобразования к

ним унаследованной коллекции понадобится вызвать метод вроде этого или, возможно, операцию OfType. Это важно, когда вы пытаетесь использовать стандартные операции запросов на унаследованных коллекциях.

Операция OfType вернет объект, который при перечислении проходит по входной последовательности, выдавая только элементы, тип которых преобразуется к указанному типу T.

Операция OfType отличается от Cast тем, что Cast пытается привести каждый элемент входной последовательности к типу T и выдать его в выходную последовательность. Если приведение не удается, генерируется исключение. Операция OfType пытается выдать входной элемент только в том случае, если он *может* быть приведен к типу T. Формально чтобы элемент e попал в выходную последовательность, для него должно быть истинно выражение e is T.

Исключения

Если аргумент source равен null, генерируется исключение ArgumentNullException.

Примеры

Для примера в листинге 4.39 создается коллекция ArrayList, содержащая объекты двух общих классов — Employee и EmployeeOptionEntry. После наполнения ArrayList объектами обоих классов сначала вызывается операция Cast, чтобы продемонстрировать, что она терпит неудачу в данной ситуации. Затем выполняется вызов операции OfType.

Листинг 4.39. Пример вызова операций Cast и OfType

```
ArrayList al = new ArrayList();
al.Add(new Employee { id = 1, firstName = "Joe", lastName = "Rattz" });
al.Add(new Employee { id = 2, firstName = "William", lastName = "Gates" });
al.Add(new EmployeeOptionEntry { id = 1, optionsCount = 0 });
al.Add(new EmployeeOptionEntry { id = 2, optionsCount = 99999999999 });
al.Add(new Employee { id = 3, firstName = "Anders", lastName = "Hejlsberg" });
al.Add(new EmployeeOptionEntry { id = 3, optionsCount = 848475745 });

var items = al.Cast<Employee>();

Console.WriteLine("Попытка использования операции Cast...");
try
{
  foreach (Employee item in items)
    Console.WriteLine("{0} {1} {2}", item.id, item.firstName, item.lastName);
}
catch (Exception ex)
{
  Console.WriteLine("{0}{1}", ex.Message, System.Environment.NewLine);
}

Console.WriteLine("Попытка использования операции OfType...");
var items2 = al.OfType<Employee>();
foreach (Employee item in items2)
  Console.WriteLine("{0} {1} {2}", item.id, item.firstName, item.lastName);
```

После создания и наполнения коллекции ArrayList элементами вызывается операция Cast. Следующий шаг — попытка перечисления результатов. Этот шаг необходим, поскольку операция Cast является отложенной. Если никогда не перечислять результаты этого запроса, он никогда не будет выполнен, и проблема не обнаружится. Обратите

внимание, что цикл foreach, который перечисляет результаты запроса, заключен в блок try/catch. В данном случае это необходимо, поскольку известно, что возникнет исключение по причине наличия объектов двух совершенно разных типов. Затем вызывается операция OfType и выполняется перечисление с отображением его результатов. Обратите внимание, что этот цикл foreach не помещен в блок try/catch. Разумеется, в реальном рабочем коде не следует игнорировать защиту, которую предоставляет блок try/catch.

Ниже показан результат выполнения этого запроса:

```
Попытка использования операции Cast...
1 Joe Rattz
2 William Gates
Unable to cast object of type 'LINQChapter4.EmployeeOptionEntry' to type
'LINQChapter4.Employee'.
```

Не удается привести объект типа 'LINQChapter4.EmployeeOptionEntry' к типу 'LINQChapter4.Employee'.

```
Попытка использования операции OfType...
1 Joe Rattz
2 William Gates
3 Anders Hejlsberg
```

Обратите внимание, что полностью выполнить перечисление результата запроса операции Cast без генерации исключения не удалось. Но все прошло нормально при перечислении результата операции OfType; в этом случае в выходную последовательность были включены только элементы типа Employee.

Мораль этой истории в том, что если входная последовательность может содержать элементы более чем одного типа данных, следует отдать предпочтение операции OfType перед Cast.

Совет. При попытке преобразовать необобщенную коллекцию, такую как один из унаследованных классов коллекций, в тип IEnumerable<T>, который может использоваться со стандартными операциями запросов, применяйте операцию OfType вместо Cast, если существует возможность того, что входная коллекция содержит объекты разных типов.

AsEnumerable

Операция AsEnumerable возвращает входную последовательность типа IEnumerable<T> как тип IEnumerable<T>.

Прототипы

Операция AsEnumerable имеет один прототип, который описан ниже.

Прототип AsEnumerable

```
public static IEnumerable<T> AsEnumerable<T>(
  this IEnumerable<T> source);
```

Приведенный выше прототип объявляет, что операция AsEnumerable работает над входной последовательностью типа IEnumerable<T> по имени source и возвращает ту же последовательность типа IEnumerable<T>. Он служит ни чему иному, кроме как изменению типа выходной последовательности во время компиляции.

Это может показаться излишним, поскольку операция уже вызывается на типе IEnumerable<T>. Может возникнуть вопрос, зачем нужно преобразовывать последовательность IEnumerable<T> в последовательность типа IEnumerable<T>? Ответ будет дан ниже.

Стандартные операции запросов объявлены для работы с нормальными последовательностями LINQ to Objects — коллекциями, реализующими интерфейс IEnumerable<T>. Однако другие специфичные для предметной области коллекции, такие как обращающиеся к базе данных, могут реализовывать собственные типы последовательностей со своими операциями. Обычно при вызове операции запроса на коллекции одного из таких типов должны вызываться операции, специфичные для данного типа. Операция AsEnumerated позволяет привести входную коллекцию к нормальному типу последовательности IEnumerable<T>, позволяя вызывать на ней методы стандартных операций запросов.

Например, когда речь пойдет о LINQ to SQL в остальной части книги, вы увидите, что LINQ to SQL в действительности использует собственный тип последовательности — IQueryable<T> — и реализует собственные операции. Вызов метода Where на последовательности типа IQueryable<T> — это на самом деле вызов метода Where из LINQ to SQL, а не стандартная операция запроса Where из LINQ to Objects. Попытка вызова одной из стандартных операций запросов приведет к исключению, если только не окажется одноименной операции LINQ to SQL. С помощью операции AsEnumerbale можно выполнить приведение последовательности IQueryable<T> к последовательности IEnumerable<T>, что позволит вызывать на ней стандартные операции запросов. Это бывает очень удобно, когда нужно контролировать, в каком API-интерфейсе вызывается операция.

Исключения

Исключений нет.

Пример

Чтобы лучше понять эту операцию, необходима ситуация, когда реализована операция, специфичная для предметной области. Для этого понадобится пример LINQ to SQL. Вспомним первый пример LINQ to SQL, приведенный в главе 1, код которого показан ниже.

Повтор листинга 1.3

```
using System;
using System.Linq;
using System.Data.Linq;
using nwind;
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
var custs =
  from c in db.Customers
  where c.City == "Rio de Janeiro"
  select c;
foreach (var cust in custs)
  Console.WriteLine("{0}", cust.CompanyName);
```

Вот результаты выполнения этого примера:

```
Hanari Carnes
Que Delícia
Ricardo Adocicados
```

Чтобы пример работал, в проект необходимо добавить ссылку на сборку System.Data.Linq.dll, директиву using для пространства имен nwind, а также сгенерированные классы, о которых будет сказано в главах, посвященных LINQ to SQL. Вдобавок может понадобиться подкорректировать строку соединения.

Предположим, что по какой-то причине нужно изменить порядок следования записей, поступающих из базы данных. Ранее в главе была описана операция Reverse.

В листинге 4.40 показан модифицированный предыдущий пример, в котором вызывается операция Reverse.

Листинг 4.40. Вызов операции Reverse

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
var custs =
  (from c in db.Customers
   where c.City == "Rio de Janeiro"
   select c)
  .Reverse();
foreach (var cust in custs)
  Console.WriteLine("{0}", cust.CompanyName);
```

Выглядит довольно просто. Как видите, единственное изменение связано с добавлением вызова метода Reverse. Код нормально компилируется. А вот результат его запуска:

```
Unhandled Exception: System.NotSupportedException: The query operator 'Reverse'
is not supported.
```

Необработанное исключение: System.NotSupportedException: Операция запроса 'Reverse' не поддерживается.

```
...
```

Все казалось так просто, что же случилось? А случилось то, что в интерфейсе IQueryable<T> отсуствует метод Reverse, потому и было сгенерировано исключение. Необходимо воспользоваться методом AsEnumerable для преобразования последовательности типа IQueryable<T> в последовательность типа IEnumerable<T>, чтобы при вызове метода Reverse был вызван метод IEnumerable<T>.Reverse(). Соответствующим образом модифицированный код представлен в листинге 4.41.

Листинг 4.41. Вызов операции AsEnumerable перед вызовом Reverse

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial
Catalog=Northwind");
var custs =
 (from c in db.Customers
  where c.City == "Rio de Janeiro"
  select c)
 .AsEnumerable()
 .Reverse();
foreach (var cust in custs)
  Console.WriteLine("{0}", cust.CompanyName);
```

Здесь сначала вызывается метод AsEnumerable, а затем операция Reverse, так что будет вызвана операция Reverse из LINQ to Objects. И вот результат:

```
Ricardo Adocicados
Que Delнcia
Hanari Carnes
```

Результирующий список представляет элементы из начального примера в обратном порядке, значит, все работает, как надо.

Элемент

Операции элементов позволяют извлекать элементы из входной последовательности.

DefaultIfEmpty

Операция DefaultIfEmpty возвращает последовательность, содержащую элемент по умолчанию, если входная последовательность пуста.

Прототипы

У операции DefaultIfEmpty есть два прототипа, которые описаны ниже.

Первый прототип DefaultIfEmpty

```
public static IEnumerable<T> DefaultIfEmpty<T>(
  this IEnumerable<T> source);
```

Этот прототип операции DefaultIfEmpty возвращает объект, который при перечислении входной последовательности source выдает каждый ее элемент, если только последовательность не окажется пустой — тогда возвращается последовательность из одного элемента default(T). Для ссылочных и допускающих null типов значением по умолчанию является null.

Обратите внимание, что в отличие от всех прочих операций типа элементов, DefaultIfEmpty возвращает последовательность типа IEnumerable<T> вместо самого типа T. Существуют еще дополнительные операции типа элементов, но они не включены в эту главу, поскольку не являются отложенными.

Второй прототип позволяет указать значение по умолчанию.

Второй прототип DefaultIfEmpty

```
public static IEnumerable<T> DefaultIfEmpty<T>(
  this IEnumerable<T> source,
  T defaultValue);
```

Эта операция полезна для всех других операций, которые генерируют исключения в случае пустой входной последовательности. Вдобавок эта операция удобна в сочетании с операцией GroupJoin для производства левосторонних внешних соединений (left outer joins).

Исключения

Если аргумент source равен null, генерируется исключение ArgumentNullException.

Примеры

В листинге 4.42 показан пример прототипа DefaultIfEmpty с пустой последовательностью. В данном примере не применяется операция DefaultIfEmpty, чтобы посмотреть, что происходит. Производится поиск в массиве presidents имени Jones, возвращается первый элемент, и если он не будет равен null, выводится сообщение.

Листинг 4.42. Первый пример, DefaultIfEmpty не используется

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
string jones = presidents.Where(n => n.Equals("Jones")).First();
if (jones != null)
  Console.WriteLine("Jones was found");
else
  Console.WriteLine("Jones was not found");
```

Вот результат:

```
Unhandled Exception: System.InvalidOperationException: Sequence contains no
elements
Необработанное исключение: System.InvalidOperationException: Последовательность
не содержит элементов
...
```

В приведенном коде запрос не нашел элемента, эквивалентного "Jones", поэтому операции First была передана пустая последовательность. Операции First не нравятся пустые последовательности, поэтому она генерирует исключение.

В примере из листинга 4.43 используется тот же самый код, но на этот раз между операциями Where и First вставлена операция DefaultIfEmpty. Таким образом, вместо пустой последовательности операции First будет передана последовательность, содержащая элемент null.

Листинг 4.43. Второй пример, DefaultIfEmpty используется

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
string jones = presidents.Where(n => n.Equals("Jones")).DefaultIfEmpty().First();
if (jones != null)
  Console.WriteLine("Элемент Jones найден.");
else
  Console.WriteLine("Элемент Jones не найден.");
```

Теперь результат выглядит так:

```
Элемент Jones не найден.
```

В примере применения второго прототипа DefaultIfEmpty, приведенном в листинге 4.44, разрешено указывать значение по умолчанию для пустой последовательности.

Листинг 4.44. Пример вызова второго прототипа DefaultIfEmpty

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
string name =
  presidents.Where(n => n.Equals("Jones")).DefaultIfEmpty("Missing").First();
Console.WriteLine(name);
```

Результат будет таким:

```
Missing
```

В последнем наборе примеров выполняется левое внешнее соединение с использованием как операции GroupJoin, так и DefaultIfEmpty. Также задействованы два общих класса — Employee и EmployeeOptionEntry. В листинге 4.45 представлен пример без применения операции DefaultIfEmpty.

Листинг 4.45. Пример без использования операции `DefaultIfEmpty`

```
ArrayList employeesAL = Employee.GetEmployeesArrayList();
// Добавить нового сотрудника, не имеющего записей EmployeeOptionEntry.
employeesAL.Add(new Employee {
               id = 102,
               firstName = "Michael",
               lastName = "Bolton" });
Employee[] employees = employeesAL.Cast<Employee>().ToArray();
EmployeeOptionEntry[] empOptions = EmployeeOptionEntry.GetEmployeeOptionEntries();
var employeeOptions = employees
  .GroupJoin(
   empOptions,
   e => e.id,
   o => o.id,
   (e, os) => os
     .Select(o => new
                 {
                   id = e.id,
                   name = string.Format("{0} {1}", e.firstName, e.lastName),
                   options = o != null ? o.optionsCount : 0
                 }))
  .SelectMany(r => r);
foreach (var item in employeeOptions)
  Console.WriteLine(item);
```

В этом примере следует отметить три момента. Во-первых, он очень похож на пример, который был представлен во время обсуждения операции `GroupJoin`. Во-вторых, поскольку общий класс `EmployeeOptionEntry` уже имел соответствующий объект для каждого сотрудника в общем классе `Employee`, можно получить коллекцию `ArrayList` сотрудников и добавить в нее нового сотрудника по имени Michael Bolton, так что она будет иметь одного сотрудника, для которого нет соответствующих объектов `EmployeeOptionEntry`. В-третьих, в этом примере не вызывается операция `DefaultIfEmpty`.

Ниже показаны результаты запроса:

```
{ id = 1, name = Joe Rattz, options = 2 }
{ id = 2, name = William Gates, options = 10000 }
{ id = 2, name = William Gates, options = 10000 }
{ id = 2, name = William Gates, options = 10000 }
{ id = 3, name = Anders Hejlsberg, options = 5000 }
{ id = 3, name = Anders Hejlsberg, options = 7500 }
{ id = 3, name = Anders Hejlsberg, options = 7500 }
{ id = 4, name = David Lightman, options = 1500 }
{ id = 101, name = Kevin Flynn, options = 2 }
```

Обратите внимание, что поскольку для сотрудника по имени Michael Bolton не предусмотрено соответствующих объектов в массиве `EmployeeOptionEntry`, можно предоставить соответствующую запись по умолчанию, как показано в листинге 4.46.

Листинг 4.46. Пример с использованием операции `DefaultIfEmpty`

```
ArrayList employeesAL = Employee.GetEmployeesArrayList();
// Добавить нового сотрудника, не имеющего записей EmployeeOptionEntry.
employeesAL.Add(new Employee {
               id = 102,
               firstName = "Michael",
               lastName = "Bolton" });
```

```
Employee[] employees = employeesAL.Cast<Employee>().ToArray();
EmployeeOptionEntry[] empOptions = EmployeeOptionEntry.GetEmployeeOptionEntries();
var employeeOptions = employees
  .GroupJoin(
    empOptions,
    e => e.id,
    o => o.id,
    (e, os) => os
      .DefaultIfEmpty()
      .Select(o => new
                  {
                    id = e.id,
                    name = string.Format("{0} {1}", e.firstName, e.lastName),
                    options = o != null ? o.optionsCount : 0
                  }))
  .SelectMany(r => r);
foreach (var item in employeeOptions)
  Console.WriteLine(item);
```

В приведенном примере также добавляется объект сотрудника Michael Bolton без связанных с ним объектов `EmployeeOptionEntry`. Но теперь вызывается операция `DefaultIfEmpty`. Вот как выглядит результат левого внешнего соединения:

```
{ id = 1, name = Joe Rattz, options = 2 }
{ id = 2, name = William Gates, options = 10000 }
{ id = 2, name = William Gates, options = 10000 }
{ id = 2, name = William Gates, options = 10000 }
{ id = 3, name = Anders Hejlsberg, options = 5000 }
{ id = 3, name = Anders Hejlsberg, options = 7500 }
{ id = 3, name = Anders Hejlsberg, options = 7500 }
{ id = 4, name = David Lightman, options = 1500 }
{ id = 101, name = Kevin Flynn, options = 2 }
{ id = 102, name = Michael Bolton, options = 0 }
```

Теперь запись Michael Bolton видна, несмотря на отсутствие соответствующих объектов `EmployeeOptionEntry`. Из результатов можно сделать вывод, что сотрудник Michael Bolton не получил никаких опционов.

Генерация

Операции генерации помогают в генерации последовательностей.

Range

Операция `Range` генерирует последовательность целых чисел.

Прототипы

У операции `Range` есть один прототип, описанный ниже.

Прототип Range

```
public static IEnumerable<int> Range(
  int start,
  int count);
```

Последовательность целых чисел генерируется, начиная со значения, переданного в `start`, протяженностью до `count`.

Обратите внимание, что это не расширяющий метод, а одна из немногих стандартных операций запросов, которые не расширяют `IEnumerable<T>`.

На заметку! `Range` — не расширяющий метод. Это статический метод класса `System.Linq.Enumerable`.

Исключения

Если `count` меньше `0` или сумма `start` и `count` минус `1` больше `int.MaxValue`, генерируется исключение `ArgumentOutOfRangeException`.

Примеры

Листинг 4.47. Пример вызова операции `Range`

```
IEnumerable<int> ints = Enumerable.Range(1, 10);
foreach(int i in ints)
  Console.WriteLine(i);
```

Опять-таки, следует подчеркнуть, что операция `Range` не вызывается на последовательности. Это статический метод класса `System.Linq.Enumerable`. Вывод доказывает отсутствие каких-либо сюрпризов:

```
1
2
3
4
5
6
7
8
9
10
```

Repeat

Операция `Repeat` генерирует последовательность, повторяя указанный элемент заданное количество раз.

Прототипы

У операции `Repeat` есть один прототип, описанный ниже.

Прототип `Repeat`

```
public static IEnumerable<int> Repeat(
  T element,
  int count);
```

Этот прототип возвращает объект, который при перечислении выдаст `count` экземпляров элемента `T`.

Обратите внимание, что он является не расширяющим методом, а одной из немногих стандартных операций запросов, которые не расширяют `IEnumerable<T>`.

На заметку! `Repeat` — не расширяющий метод. Это статический метод класса `System.Linq.Enumerable`.

Исключения

Если значение аргумента `count` меньше нуля, генерируется исключение `ArgumentOutOfRangeException`.

Примеры

В листинге 4.48 генерируется последовательность из десяти элементов, каждый из которых является числом 2.

Листинг 4.48. Возврат последовательности из десяти целых чисел, каждое из которых равно 2

```
IEnumerable<int> ints = Enumerable.Repeat(2, 10);
foreach(int i in ints)
  Console.WriteLine(i);
```

Вот результат работы этого примера:

```
2
2
2
2
2
2
2
2
2
2
```

Empty

Операция Empty генерирует пустую последовательность заданного типа.

Прототипы

Операция Empty имеет один прототип, представленный ниже.

Прототип Empty

```
public static IEnumerable<T> Empty<T>();
```

Этот прототип возвращает объект, при перечислении выдающий последовательность из нуля элементов типа T.

Обратите внимание, что это не расширяющий метод, а одна из немногих стандартных операций запросов, которая не расширяет IEnumerable<T>.

На заметку! Empty — не расширяющий метод. Это статический метод класса System.Linq.Enumerable.

Исключения

Исключений нет.

Примеры

В листинге 4.49 генерируется пустая последовательность типа string с использованием операции Empty и выводится значение Count для сгенерированной последовательности, которое должно быть равно нулю, поскольку последовательность пуста.

Листинг 4.49. Пример возврата пустой последовательность строк

```
IEnumerable<string> strings = Enumerable.Empty<string>();
foreach(string s in strings)
  Console.WriteLine(s);
Console.WriteLine(strings.Count());
```

Ниже показан вывод этого кода:

```
0
```

Поскольку последовательность пуста, в цикле `foreach` отображать нечего, поэтому был предусмотрен вывод количества элементов в последовательности.

Резюме

В этой главе был дан очень быстрый тур по отложенным стандартным операциям запросов. Были представлены примеры применения почти каждого прототипа каждой отложенной операции, причем далеко не все их можно назвать простейшими.

Разделение стандартных операций запросов на отложенные и не отложенные должно дополнительно подчеркивать важность этого обстоятельства для запросов.

В следующей главе рассмотрение LINQ to Objects дополняется описанием не отложенных стандартных операций запросов.

ГЛАВА 5

Не отложенные операции

В предыдущей главе рассматривались отложенные стандартные операции запросов. Их легко отличить по тому, что они возвращают либо IEnumerable<T>, либо OrderesSequence<T>. Однако отложенные операции — это только половина из всех стандартных операциях запросов. Для полноты картины нужно также описать не отложенные (nondeferred) операции. Их легко отличить по типу возврата, которым не является IEnumerable<T> или OrderesSequence<T>. Каждая из операций будет отнесена к определенной категории по своему назначению.

Чтобы кодировать и выполнять примеры этой главы, следует помнить об использовании директивы using со всеми необходимыми пространствами имен, ссылках на все нужные сборки, а также на общий код, разделяемый примерами.

Необходимые пространства имен

В примерах этой главы будут использоваться пространства имен System.Linq, System.Collections и System.Collections.Generic. Таким образом, потребуется добавить в код следующие директивы using, если их еще там нет:

```
using System.Linq;
using System.Collections;
using System.Collections.Generic;
```

Загрузив сопровождающий код, вы увидите, что в дополнение к этим пространствам имен также добавлена директива using для пространства имен System.Diagnostics. Это не является необходимым для примеров этой главы. Это понадобится только для сопровождающего кода — из-за некоторого добавленного рабочего кода.

Общие классы

В нескольких примерах этой главы понадобятся классы — для всесторонней демонстрации поведения той или иной операции. Ниже описаны четыре класса, которые будут использоваться в более чем одном примере, начиная с Employee.

Класс Employee представляет сотрудника. Для удобства он содержит статические методы, возвращающие ArrayList или массив сотрудников.

Разделяемый класс Employee

```
public class Employee
{
  public int id;
  public string firstName;
  public string lastName;
```

```
  public static ArrayList GetEmployeesArrayList()
  {
    ArrayList al = new ArrayList();
    al.Add(new Employee { id = 1, firstName = "Joe", lastName = "Rattz" });
    al.Add(new Employee { id = 2, firstName = "William", lastName = "Gates" });
    al.Add(new Employee { id = 3, firstName = "Anders", lastName = "Hejlsberg" });
    al.Add(new Employee { id = 4, firstName = "David", lastName = "Lightman" });
    al.Add(new Employee { id = 101, firstName = "Kevin", lastName = "Flynn" });
    return (al);
  }
  public static Employee[] GetEmployeesArray()
  {
    return ((Employee[])GetEmployeesArrayList().ToArray());
  }
}
```

Класс `EmployeeOptionEntry` представляет присуждение опционов на акции определенному сотруднику. Для удобства он содержит статический метод, возвращающий список назначенных опционов.

Разделяемый класс `EmployeeOptionEntry`

```
public class EmployeeOptionEntry
{
   public int id;
   public long optionsCount;
   public DateTime dateAwarded;
   public static EmployeeOptionEntry[] GetEmployeeOptionEntries()
   {
      EmployeeOptionEntry[] empOptions = new EmployeeOptionEntry[] {
         new EmployeeOptionEntry {
            id = 1,
            optionsCount = 2,
            dateAwarded = DateTime.Parse("1999/12/31") },
         new EmployeeOptionEntry {
            id = 2,
            optionsCount = 10000,
            dateAwarded = DateTime.Parse("1992/06/30") },
         new EmployeeOptionEntry {
            id = 2,
            optionsCount = 10000,
            dateAwarded = DateTime.Parse("1994/01/01") },
         new EmployeeOptionEntry {
            id = 3,
            optionsCount = 5000,
            dateAwarded = DateTime.Parse("1997/09/30") },
         new EmployeeOptionEntry {
            id = 2,
            optionsCount = 10000,
            dateAwarded = DateTime.Parse("2003/04/01") },
         new EmployeeOptionEntry {
            id = 3,
            optionsCount = 7500,
            dateAwarded = DateTime.Parse("1998/09/30") },
         new EmployeeOptionEntry {
            id = 3,
            optionsCount = 7500,
            dateAwarded = DateTime.Parse("1998/09/30") },
         new EmployeeOptionEntry {
            id = 4,
```

```
            optionsCount = 1500,
            dateAwarded = DateTime.Parse("1997/12/31") },
        new EmployeeOptionEntry {
            id = 101,
            optionsCount = 2,
            dateAwarded = DateTime.Parse("1998/12/31") }
    };
    return (empOptions);
  }
}
```

Несколько операций будут принимать классы, которые реализуют интерфейс IEqualityComparer<T> в целях сравнения элементов для определения их эквивалентности. Это полезно для тех случаев, когда два значения не в точности эквивалентны, но их нужно трактовать как эквивалентные. Например, при сравнении двух строк может понадобиться игнорировать регистр. Однако для такой ситуации в .NET Framework уже предусмотрен соответствующий класс сравнения эквивалентности.

Поскольку интерфейс IEqualityComparer<T> был детально описан в предыдущей главе, здесь он не повторяется.

В примерах понадобится класс сравнения эквивалентности, который знает, как сравнивать числа в строковом формате. Так, например, строки "17" и "00017" должны трактоваться как эквивалентные. Ниже показан класс MyStringifiedNumberComparer, который делает это.

Разделяемый класс MyStringifiedNumberComparer

```
public class MyStringifiedNumberComparer : IEqualityComparer<string>
{
   public bool Equals(string x, string y)
   {
      return(Int32.Parse(x) == Int32.Parse(y));
   }
   public int GetHashCode(string obj)
   {
      return Int32.Parse(obj).ToString().GetHashCode();
   }
}
```

Обратите внимание, что эта реализация интерфейса IEqualityComparer будет работать только с переменными типа string, но для примеров этого достаточно. В основном, во всех сравнениях значения string просто преобразуются в Int32. Таким образом, строка "002" превращается в целочисленное значение 2, так что ведущие нули никак не влияют на ключевое значение.

Для реализации некоторых примеров понадобится класс, который сможет хранить записи с не уникальными ключами. Для этой цели создан класс Actor, приведенный ниже. Его член birthYear используется в качестве ключа.

Разделяемый класс Actor

```
public class Actor
{
   public int birthYear;
   public string firstName;
   public string lastName;
   public static Actor[] GetActors()
   {
     Actor[] actors = new Actor[] {
       new Actor { birthYear = 1964, firstName = "Keanu", lastName = "Reeves" },
```

```
            new Actor { birthYear = 1968, firstName = "Owen", lastName = "Wilson" },
            new Actor { birthYear = 1960, firstName = "James", lastName = "Spader" },
            new Actor { birthYear = 1964, firstName = "Sandra", lastName = "Bullock" },
          };
          return (actors);
        }
    }
```

Организация не отложенных операций по назначению

В настоящем разделе не отложенные стандартные операции запросов организованы по их предназначению.

Преобразование

Следующие операции преобразования предоставляют простой и удобный способ преобразования последовательностей в другие типы коллекций.

ToArray

Операция `ToArray` создает массив типа `T` из входной последовательности типа `T`.

Прототипы

Эта операция имеет один прототип, описанный ниже.

Прототип ToArray

```
public static T[] ToArray<T>(
  this IEnumerable<T> source);
```

Эта операция берет входную последовательность `source` с элементами типа `T` и возвращает массив элементов типа `T`.

Исключения

Если аргумент `source` равен `null`, генерируется исключение `ArgumentNullException`.

Примеры

Для примера, демонстрирующего работу операции `ToArray`, понадобится последовательность типа `IEnumerable<T>`. Последовательность этого типа создается вызовом на массиве операции `OfType`, которая была описана в предыдущей главе. Имея такую последовательность, можно вызывать операцию `ToArray` для создания массива, как показано в листинге 5.1.

Листинг 5.1. Пример вызова операции ToArray

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
string[] names = presidents.OfType<string>().ToArray();
foreach (string name in names)
  Console.WriteLine(name);
```

Сначала с помощью операции OfType массив presidents преобразуется в последовательность типа IEnumerable<string>. Затем эта последовательность преобразуется в массив с помощью операции ToArray. Поскольку ToArray — не отложенная операция, запрос выполняется немедленно, еще до его перечисления.

Запуск приведенного кода даст такие результаты:

```
Adams
Arthur
Buchanan
Bush
Carter
Cleveland
Clinton
Coolidge
Eisenhower
Fillmore
Ford
Garfield
Grant
Harding
Harrison
Hayes
Hoover
Jackson
Jefferson
Johnson
Kennedy
Lincoln
Madison
McKinley
Monroe
Nixon
Obama
Pierce
Polk
Reagan
Roosevelt
Taft
Taylor
Truman
Tyler
Van Buren
Washington
Wilson
```

Формально код этого примера несколько избыточен. Массив presidents — это уже последовательность, поскольку в C# массивы реализуют интерфейс IEnumerable<T>. Поэтому вызов операции OfType можно было опустить и просто вызвать операцию ToArray на массиве presidents. Однако преобразование массива в массив выглядело бы не очень впечатляюще.

Эта операция часто бывает удобной для кэширования последовательности, чтобы она не могла измениться до начала ее перечисления. К тому же, поскольку эта операция не является отложенной и выполняется немедленно, множество перечислений, созданных на одном массиве, всегда будут видеть одни и те же данные.

ToList

Операция ToList создает List типа T из входной последовательности типа T.

Прототипы

Эта операция имеет один прототип, описанный ниже.

Прототип ToList

```
public static List<T> ToList<T>(
  this IEnumerable<T> source);
```

Данная операция принимает последовательность по имени source элементов типа T и возвращает список List элементов типа T.

Исключения

Если аргумент source равен null, генерируется исключение ArgumentNullException.

Примеры

В листинге 5.2 демонстрируется применение операции ToList.

Листинг 5.2. Пример вызова операции ToList

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
List<string> names = presidents.ToList();
foreach (string name in names)
  Console.WriteLine(name);
```

В приведенном коде используется массив из предыдущего примера. Но в отличие от предыдущего примера операция OfType для создания промежуточной последовательности IEnumerable<T> не вызывается, поскольку вполне достаточно преобразовать массив presidents в список List<string>.

Вот результат:

```
Adams
Arthur
Buchanan
Bush
Carter
Cleveland
Clinton
Coolidge
Eisenhower
Fillmore
Ford
Garfield
Grant
Harding
Harrison
Hayes
Hoover
Jackson
Jefferson
Johnson
Kennedy
Lincoln
```

```
Madison
McKinley
Monroe
Nixon
Obama
Pierce
Polk
Reagan
Roosevelt
Taft
Taylor
Truman
Tyler
Van Buren
Washington
Wilson
```

Эта операция часто полезна для кэширования последовательности, чтобы она не могла измениться перед ее перечислением. Также, поскольку эта операция не является отложенной и выполняется немедленно, множество перечислений на созданном списке `List<T>` всегда видят одинаковые данные.

ToDictionary

Операция `ToDictionary` создает `Dictionary` типа `<K,T>`, или, возможно, `<K,E>`, если прототип имеет аргумент `elementSelector`, из входной последовательности типа `T`, где `K` — тип ключа, а `T` — тип хранимых значений. Или же, если `Dictionary` имеет тип `<K,E>`, то типом хранимых значений будет `E`, отличающийся от типа элементов в последовательности — `T`.

На заметку! Если вы не знакомы с классом `Dictionary` коллекций C#, то знайте, что он позволяет хранить элементы, которые можно извлекать по ключу. Каждый ключ должен быть уникальным, и только один элемент может быть сохранен для одного ключа. Элементы в `Dictionary` индексируются по ключу для последующего их извлечения по соответствующим ключам.

Прототипы

Эта операция имеет четыре прототипа, описанные ниже.

Первый прототип операции `ToDictionary`

```
public static Dictionary<K, T> ToDictionary<T, K>(
  this IEnumerable<T> source,
  Func<T, K> keySelector);
```

В этом прототипе создается словарь `Dictionary` типа `<K, T>`, который возвращается при перечислении входной последовательности по имени `source`. Делегат метода `keySelector` вызывается для извлечения значения ключа из каждого входного элемента, и этот ключ становится ключом элемента в `Dictionary`. Эта версия операции дает в результате элементы в `Dictionary` того же типа, что и элементы входной последовательности.

Поскольку данный прототип не предусматривает указание объекта `IEqualityComparer<K>`, эта версия `ToDictionary` по умолчанию использует объект проверки эквивалентности `EqualityComparer<K>.Default`.

Второй прототип `ToDictionary` подобен первому, за исключением того, что он позволяет указывать объект проверки эквивалентности `IEqualityComparer<K>`. Ниже показан второй прототип.

Второй прототип операции ToDictionary

```
public static Dictionary<K, T> ToDictionary<T, K>(
  this IEnumerable<T> source,
  Func<T, K> keySelector,
  vIEqualityComparer<K> comparer);
```

Этот прототип предоставляет возможность указать объект проверки эквивалентности IEqualityComparer<K>. Данный объект используется для сравнения элементов по значению ключа. Поэтому при добавлении или обращении к элементу в Dictionary он использует этот comparer для сравнения указанного ключа с ключами, содержащимися в Dictionary, чтобы определить их соответствие.

Реализация по умолчанию интерфейса IEqualityComparer<K> предоставляется EqualityComparer.Default. Однако если вы собираетесь использовать класс проверки эквивалентности по умолчанию, то нет причин указывать параметр comparer, потому что предыдущий прототип, где comparer не указан, использует эту установку по умолчанию. Класс StringComparer реализует в себе несколько классов проверки эквивалентности, включая класс, который игнорирует регистр символов. Таким образом, ключи "Joe" и "joe" оцениваются как эквивалентные.

Третий прототип ToDictionary подобен первому, за исключением того, что позволяет указывать селектор элемента, поэтому тип данных элементов, хранимых в Dictionary, может отличаться от типа элементов входной последовательности.

Третий прототип ToDictionary

```
public static Dictionary<K, E> ToDictionary<T, K, E>(
  this IEnumerable<T> source,
  Func<T, K> keySelector,
  Func<T, E> elementSelector);
```

С помощью аргумента elementSelector можно задать делегат метода, возвращающий часть входного элемента — или вновь созданный объект совершенно другого типа, — который необходимо сохранить в Dictionary.

Четвертый прототип операции ToDictionary предлагает лучшее из всех предыдущих. Он представляет собой комбинацию второго и третьего прототипов, а это означает, что можно указывать объекты elementSelector и comparer.

Четвертый прототип ToDictionary

```
public static Dictionary<K, E> ToDictionary<T, K, E>(
  this IEnumerable<T> source,
  Func<T, K> keySelector,
  Func<T, E> elementSelector,
  IEqualityComparer<K> comparer);
```

Этот прототип позволяет указывать объекты elementSelector и comparer.

Исключения

Если аргумент source, keySelector или elementSelector равен null, либо ключ, возвращенный keySelector, равен null, генерируется исключение ArgumentNullException.

Если keySelector возвращает одинаковый ключ для двух элементов, генерируется исключение ArgumentException.

Примеры

В этом примере вместо типичного массива presidents, применяемого выше, используется общий класс Employee. Будет создан словарь типа Dictionary<int, Employee>,

где ключ типа int — это член id класса Employee, и сам объект Employee представляет хранящийся элемент.

В листинге 5.3 приведен пример вызова операции ToDictionary с использованием класса Employee.

Листинг 5.3. Пример вызова первого прототипа ToDictionary

```
Dictionary<int, Employee> eDictionary =
  Employee.GetEmployeesArray().ToDictionary(k => k.id);
Employee e = eDictionary[2];
Console.WriteLine("Сотрудником с id == 2 является {0} {1}", e.firstName, e.lastName);
```

Словарь Dictionary объявляется с целочисленным типом ключа, т.к. для ключа выбрано поле Employee.id. Поскольку прототип операции ToDictionary позволяет хранить только входной элемент целиком, которым является объект Employee, типом элемента Dictionary также является Employee. Словарь Dictionary<int, Employee> затем позволяет искать сотрудников по их идентификатору id, предлагая производительность и удобство класса Dictionary. Вот результат работы приведенного выше кода:

```
Сотрудником с id == 2 является William Gates
```

Так как предназначение второго прототипа — позволить указывать объект, проверяющий эквивалентность типа IEqualityComparer<T>, для написания примера понадобится ситуация, в которой будет полезен класс, проверяющий эквивалентность. Это ситуация, когда ключи, которые литерально могут быть не эквивалентны, должны трактоваться таковыми посредством специального класса проверки эквивалентности. Для этой цели в качестве ключа используется числовое значение в строковом формате, такое как "1". Поскольку иногда числовые значения в строковом формате имеют ведущие нули, также может оказаться, что ключ для одного и того же элемента данных имеет вид "1", "01" или даже "00001". Поскольку эти строковые значения не эквивалентны, понадобится класс проверки эквивалентности, который будет знать, что все эти значения следует трактовать как эквивалентные.

Однако сначала понадобится класс с ключом типа string. Для этого в используемый выше общий класс Employee вносятся небольшие изменения. Показанный ниже класс Employee2 во всем идентичен Employee, за исключением того, что типом члена id будет string вместо int.

Класс для примера применения второго прототипа операции ToDictionary

```
public class Employee2
{
  public string id;
  public string firstName;
  public string lastName;
  public static ArrayList GetEmployeesArrayList()
  {
    ArrayList al = new ArrayList();
    al.Add(new Employee2 { id = "1", firstName = "Joe", lastName = "Rattz" });
    al.Add(new Employee2 { id = "2", firstName = "William", lastName = "Gates" });
    al.Add(new Employee2 { id = "3", firstName = "Anders",
                           lastName = "Hejlsberg" });
    al.Add(new Employee2 { id = "4", firstName = "David", lastName = "Lightman" });
    al.Add(new Employee2 { id = "101", firstName = "Kevin", lastName = "Flynn" });
    return (al);
  }
```

```
    public static Employee2[] GetEmployeesArray()
    {
        return ((Employee2[])GetEmployeesArrayList().ToArray(typeof(Employee2)));
    }
  }
```

Тип ключа изменен на string, чтобы продемонстрировать, как можно использован класс, проверяющий эквивалентность ключей, даже в случае, когда они литерально не эквивалентны. Поскольку ключи теперь string, в этом примере задействован общий класс MyStringifiedNumberComparer, которому известно, что ключ "02" эквивалентен ключу "2".

Теперь рассмотрим код, в котором применяется класс Employee2 и реализация IEqualityComparer, представленная в листинге 5.4.

Листинг 5.4. Пример вызова второго прототипа ToDictionary

```
Dictionary<string, Employee2> eDictionary = Employee2.GetEmployeesArray()
  .ToDictionary(k => k.id, new MyStringifiedNumberComparer());
Employee2 e = eDictionary["2"];
Console.WriteLine("Сотрудник с id == \"2\" : {0} {1}",
  e.firstName, e.lastName);
e = eDictionary["000002"];
Console.WriteLine("Сотрудник с id == \"000002\" : {0} {1}",
  e.firstName, e.lastName);
```

В примере производится попытка обратиться к элементам в Dictionary со значениями ключей "2" и "000002". Если класс, проверяющий эквивалентность, работает правильно, в обоих случаях из Dictionary должен быть получен один и тот же элемент. Вот результат:

```
Сотрудник с id == "2" : William Gates
Сотрудник с id == "000002" : William Gates
```

Как видите, из Dictionary получен один и тот же элемент, независимо от того, какой строковый ключ применялся для доступа. Это справедливо до тех пор, пока каждое переданное строковое значение представляет одно и то же целое число.

Третий прототип позволяет сохранять в словаре элемент, тип которого отличается от типа элементов входной последовательности. Для примера третьего прототипа используется тот же самый класс Employee, который применялся в коде примера первого прототипа ToDictionary. В листинге 5.5 содержится код примера вызова третьего прототипа ToDictionary.

Листинг 5.5. Пример вызова третьего прототипа ToDictionary

```
Dictionary<int, string> eDictionary = Employee.GetEmployeesArray()
  .ToDictionary(k => k.id,
                i => string.Format("{0} {1}",          // elementSelector
                     i.firstName, i.lastName));
string name = eDictionary[2];
Console.WriteLine("Сотрудником с id == 2 является {0}", name);
```

В данном коде написано лямбда-выражение, соединяющее firstName и lastName в одну строку. Эта объединенная строка становится значением, хранимым в Dictionary. Таким образом, хотя типом входного элемента является Employee, тип данных элемента, хранимого в словаре — string. Вот результаты этого запроса:

```
Сотрудником с id == 2 является William Gates
```

Для демонстрации прототипа ToDictionary применяются классы Employee2 и MyStringifiedNumberComparer. Код примера представлен в листинге 5.6.

Листинг 5.6. Пример вызова четвертого прототипа ToDictionary

```
Dictionary<string, string> eDictionary = Employee2.GetEmployeesArray()
  .ToDictionary(k => k.id,                        // keySelector
                i => string.Format("{0} {1}",     // elementSelector
                     i.firstName, i.lastName),
             new MyStringifiedNumberComparer()); // comparer
string name = eDictionary["2"];
Console.WriteLine("Сотрудник с id == \"2\" : {0}", name);
name = eDictionary["000002"];
Console.WriteLine("Сотрудник с id == \"000002\" : {0}", name);
```

В коде создается elementSelector, который указывает единственную строку в качестве значения, сохраняемого в Dictionary, а также пользовательский объект проверки эквивалентности. В результате для извлечения элемента из Dictionary можно использовать "2" или "000002", благодаря специальному классу проверки эквивалентности. То, что теперь получается из Dictionary — это просто строка, в которой содержатся имя и фамилия сотрудника, соединенные вместе. Вот результат:

```
Сотрудник с id == "2" : William Gates
Сотрудник с id == "000002" : William Gates
```

Как видите, обращение по индексу к Dictionary со значением ключа "2" или "000002" извлекает один и тот же элемент.

ToLookup

Операция ToLookup создает объект Lookup типа <K, T> или, возможно, <K, E> из входной последовательности типа T, где K — тип ключа, а T — тип хранимых значений. Либо же, если Lookup имеет тип <K, T>, то типом хранимых значений может быть E, который отличается от типа элементов входной последовательности T.

Хотя все прототипы операции ToLookup создают Lookup, возвращают они объект, реализующий интерфейс ILookup. В этом разделе объект, реализующий интерфейс ILookup, обычно будет называться просто Lookup.

На заметку! Если вы незнакомы с классом Lookup коллекций C#, то знайте, что он позволяет хранить элементы, которые могут быть извлечены по ключу. Каждый ключ должен быть уникальным, и под одним ключом может быть сохранено *множество* элементов. Обращение по индексу к Lookup с применением ключа извлекает *последовательность* сохраненных с этим ключом элементов.

Прототипы

Операция ToLookup имеет четыре прототипа, описанные ниже.

Первый прототип операции ToLookup

```
public static ILookup<K, T> ToLookup<T, K>(
  this IEnumerable<T> source,
  Func<T, K> keySelector);
```

В этом прототипе создается Lookup типа <K, T> и возвращается за счет перечисления входной последовательности source. Делегат метода keySelector вызывается для извлечения ключевого значения из каждого входного элемента, и этот ключ является

ключом элемента в Lookup. Эта версия операции сохраняет в Lookup значения того же типа, что и элементы входной последовательности.

Поскольку прототип предотвращает указание объекта проверки эквивалентности IEqualityComparer<K>, данная версия Lookup по умолчанию использует в качестве такого объекта экземпляр класса EqualityComparer<K>.Default.

Второй прототип ToLookup подобен первому, за исключением того, что предоставляет возможность указывать объект проверки эквивалентности IEqualityComparer<K>. Он рассматривается ниже.

Второй прототип операции ToLookup

```
public static ILookup<K, T> ToLookup<T, K>(
  this IEnumerable<T> source,
  Func<T, K> keySelector,
  IEqualityComparer<K> comparer);
```

Этот прототип дает возможность указывать объект comparer типа IEquality Comparer<K> для проверки эквивалентности ключей. Поэтому при добавлении или обращении к элементу в Lookup он использует этот объект comparer для сравнения указанного ключа с ключами, уже находящимися в Lookup, определяя их соответствие.

Реализация по умолчанию интерфейса IEqualityComparer<K> предоставляется EqualityComparer.Default. Однако если вы собираетесь использовать класс проверки эквивалентности по умолчанию, то указывать объект, проверяющий эквивалентность, незачем, потому что предыдущий прототип применяет его в любом случае. Класс StringComparer реализует некоторые классы проверки эквивалентности, такие как, например, игнорирующий различия в регистре. Таким образом, ключи "Joe" и "joe" трактуются как один и тот же ключ.

Третий прототип ToLookup подобен первому, за исключением того, что позволяет указывать селектор элемента, чтобы тип данных значения, сохраненного в Lookup, мог отличаться от типа элементов входной последовательности. Третий прототип представлен ниже.

Третий прототип операции ToLookup

```
public static ILookup<K, E> ToLookup<T, K, E>(
  this IEnumerable<T> source,
  Func<T, K> keySelector,
  Func<T, E> elementSelector);
```

В аргументе elementSelector можно указывать делегат метода, возвращающий часть входного элемента, либо вновь созданный объект совершенно другого типа, который должен быть сохранен в Lookup.

Четвертый прототип операции ToLookup концентрирует в себе лучшее из предыдущих. Он представляет собой комбинацию второго и третьего прототипов, а это означает, что можно указывать elementSelector и объект проверки эквивалентности comparer. Четвертый прототип описан ниже.

Четвертый прототип операции ToLookup

```
public static ILookup<K, E> ToLookup<T, K, E>(
  this IEnumerable<T> source,
  Func<T, K> keySelector,
  Func<T, E> elementSelector,
  IEqualityComparer<K> comparer);
```

Этот прототип позволяет указывать и elementSelector, и comparer.

Исключения

Если аргумент source, keySelector или elementSelector равен null, либо ключ, возвращенный keySelector, равен null, генерируется исключение ArgumentNullException.

Примеры

В этом примере прототипа ToLookup вместо типичного массива presidents, который постоянно применялся, необходим класс с элементами, содержащими члены, которые могут быть использованы в качестве ключей, но не являются уникальными. Для этой цели задействован общий класс Actor.

В листинге 5.7 содержится пример вызова операции ToLookup с использованием класса Actor.

Листинг 5.7. Пример вызова первого прототипа ToLookup

```
ILookup<int, Actor> lookup = Actor.GetActors().ToLookup(k => k.birthYear);

// Посмотрим, можно ли найти кого-то, рожденного в 1964 г.
IEnumerable<Actor> actors = lookup[1964];

foreach (var actor in actors)
  Console.WriteLine("{0} {1}", actor.firstName, actor.lastName);
```

Сначала создается Lookup с указанием члена Actor.birthYear в качестве ключа к Lookup. Затем производится доступ к нему по индексу с ключом 1964. Затем возвращенные значения перечисляются. Ниже показан результат:

```
Keanu Reeves
Sandra Bullock
```

Неожиданно был получен множественный результат. Хорошо, что входная последовательность была преобразована в Lookup вместо Dictionary, т.к. имеется несколько элементов с одинаковым ключом.

Для построения примера, демонстрирующего работу второго прототипа ToLookup, в общий класс Actor вносится небольшая модификация. Будет создан класс Actor2, во всем идентичный Actor, за исключением того, что член birthYear теперь имеет тип string вместо int.

Класс для примера применения второго прототипа операции ToLookup

```
public class Actor2
{
  public string birthYear;
  public string firstName;
  public string lastName;

  public static Actor2[] GetActors()
  {
    Actor2[] actors = new Actor2[] {
      new Actor2 { birthYear = "1964", firstName = "Keanu", lastName = "Reeves" },
      new Actor2 { birthYear = "1968", firstName = "Owen", lastName = "Wilson" },
      new Actor2 { birthYear = "1960", firstName = "James", lastName = "Spader" },

      // Пример даты с ведущим нулем
      new Actor2 { birthYear = "01964", firstName = "Sandra", lastName = "Bullock" },
    };
    return(actors);
  }
}
```

Обратите внимание, что в этом классе тип члена birthYear изменен на string. Теперь операция ToLookup будет вызываться, как показано в листинге 5.8.

Листинг 5.8. Пример вызова второго прототипа ToLookup

```
ILookup<string, Actor2> lookup = Actor2.GetActors()
  .ToLookup(k => k.birthYear, new MyStringifiedNumberComparer());

// Посмотрим, можно ли найти кого-то, рожденного в 1964 г.
IEnumerable<Actor2> actors = lookup["0001964"];
foreach (var actor in actors)
  Console.WriteLine("{0} {1}", actor.firstName, actor.lastName);
```

Здесь используется тот же самый объект проверки эквивалентности, что и в примере Dictionary. В этом случае входная последовательность преобразуется в Lookup, с предоставлением объекта проверки эквивалентности, поскольку известно, что ключ, хранимый в виде string, может иногда содержать ведущие нули. Этот объект знает, как с этим справиться. Ниже показан результат:

```
Keanu Reeves
Sandra Bullock
```

Обратите внимание на попытку извлечь все элементы со значением ключа "0001964". В результате получаются элементы с ключами "1964" и "01964". Значит, объект проверки эквивалентности работает.

В примере с третьим прототипом операции ToLookup применяется тот же самый класс Actor, что и в примере кода первого прототипа для ToLookup. В листинге 5.9 показан необходимый код.

Листинг 5.9. Пример вызова третьего прототипа ToLookup

```
ILookup<int, string> lookup = Actor.GetActors()
  .ToLookup(k => k.birthYear,
            a => string.Format("{0} {1}", a.firstName, a.lastName));

// Посмотрим, можно ли найти кого-то, рожденного в 1964 г.
IEnumerable<string> actors = lookup[1964];
foreach (var actor in actors)
  Console.WriteLine("{0}", actor);
```

В elementSelector просто соединяются в одну строку члены firstName и lastName. И вот результат:

```
Keanu Reeves
Sandra Bullock
```

Использование варианта операции ToLookup с elementSelector позволяет сохранять в Lookup данные другого типа, отличного от типа данных элемента входной последовательности.

При написании примера для четвертого прототипа ToLookup используется класс Actor2 и общий класс MyStringifiedNumberComparer. Код примера показан в листинге 5.10.

Листинг 5.10. Пример вызова четвертого прототипа ToLookup

```
ILookup<string, string> lookup = Actor2.GetActors()
  .ToLookup(k => k.birthYear,
            a => string.Format("{0} {1}", a.firstName, a.lastName),
                 new MyStringifiedNumberComparer());
```

```
// Посмотрим, можно ли найти кого-то, рожденного в 1964 г.
IEnumerable<string> actors = lookup["0001964"];
foreach (var actor in actors)
  Console.WriteLine("{0}", actor);
```

Вот результат:

```
Keanu Reeves
Sandra Bullock
```

Как видите, здесь производится обращение к `Lookup` по индексу с указанием значения ключа, отличного от любого из значений ключей извлеченных значений, поэтому можно сделать вывод, что объект проверки эквивалентности работает. Вместо сохранения всего объекта `Actor2` сохраняется просто интересующая строка.

Эквивалентность

Следующие операции эквивалентности используются для проверки эквивалентности последовательностей.

SequenceEqual

Операция `SequenceEqual` определяет, эквивалентны ли две входные последовательности.

Прототипы

Эта операция имеет два прототипа, описанные ниже.

Первый прототип SequenceEqual

```
public static bool SequenceEqual<T>(
  this IEnumerable<T> first,
  IEnumerable<T> second);
```

Эта операция перечисляет каждую входную последовательность параллельно, сравнивая элементы с помощью метода `System.Object.Equals`. Если элементы эквивалентны, и последовательности содержат одинаковое количество элементов, операция возвращает `true`. Иначе она возвращает `false`.

Второй прототип операции работает так же, как и первый, за исключением того, что принимает объект `IEqualityComparer<T>`, который может быть использован для проверки эквивалентности.

Второй прототип SequenceEqual

```
public static bool SequenceEqual<T>(
  this IEnumerable<T> first,
  IEnumerable<T> second,
  IEqualityComparer<T> comparer);
```

Исключения

Если любой из аргументов равен `null`, генерируется исключение `ArgumentNullException`.

Примеры

Пример приведен в листинге 5.11.

Листинг 5.11. Пример вызова первого прототипа операции `SequenceEqual`

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};

bool eq = presidents.SequenceEqual(presidents);
Console.WriteLine(eq);
```

Результат выглядит следующим образом:

```
True
```

Не особо впечатляет. В листинге 5.12 пример будет немного усложнен.

Листинг 5.12. Другой пример вызова первого прототипа операции `SequenceEqual`

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};

bool eq = presidents.SequenceEqual(presidents.Take(presidents.Count()));
Console.WriteLine(eq);
```

В приведенном коде с помощью операции `Take` были выбраны только первые *N* элементов из массива `presidents`, и полученная выходная последовательность сравнивалась с исходным массивом `presidents`. Итак, если в приведенном выше коде взять все элементы массива `presidents`, указав количество элементов через `presidents.Count()`, то будет получена вся выходная последовательность целиком. Как и следовало ожидать, вот результат:

```
True
```

Все работает, как и должно. Теперь возьмем все элементы кроме последнего, вычтя единицу из `presidents.Count()`, как показано в листинге 5.13.

Листинг 5.13. Еще один пример вызова первого прототипа операции `SequenceEqual`

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};

bool eq = presidents.SequenceEqual(presidents.Take(presidents.Count() - 1));
Console.WriteLine(eq);
```

Теперь результат должен быть `false`, потому что две последовательности имеют разную длину. Во второй из них недостает последнего элемента:

```
False
```

Все идет хорошо. Но просто из любопытства давайте продолжим. Если вспомнить описание операций `Take` и `Skip` из предыдущей главы, то там было сказано, что соединенные вместе надлежащим образом, они должны дать исходную последовательность. Попробуем это реализовать. В листинге 5.14 приведен код, в котором с помощью операций `Take`, `Skip`, `Concat` и `SequenceEqual` доказывает это утверждение.

Листинг 5.14. Более сложный пример вызова первого прототипа операции `SequenceEqual`

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
bool eq =
  presidents.SequenceEqual(presidents.Take(5).Concat(presidents.Skip(5)));
Console.WriteLine(eq);
```

В данном примере с помощью операции `Take` из исходной последовательности извлекаются первые пять элементов. Затем осуществляется конкатенация с входной последовательностью, начиная с шестого элемента, с использованием операций `Skip` и `Concat`. И, наконец, определяется, эквивалентна ли эта объединенная последовательность и исходная последовательность, вызывая для этого операцию `SequenceEqual`. Давайте посмотрим на результат:

```
True
```

Как видите, все работает. В примере использования второго прототипа создаются два массива типа `string`, где каждый элемент представляет собой число в строковой форме. Элементы двух массивов будут такими, что при преобразовании в числа окажутся эквивалентными. В этом примере (листинг 5.15) задействован общий класс `MyStringifiedNumberComparer`.

Листинг 5.15. Пример вызова второго прототипа операции `SequenceEqual`

```
string[] stringifiedNums1 = {
  "001", "49", "017", "0080", "00027", "2" };
string[] stringifiedNums2 = {
  "1", "0049", "17", "080", "27", "02" };
bool eq = stringifiedNums1.SequenceEqual(stringifiedNums2,
                                 new MyStringifiedNumberComparer());
Console.WriteLine(eq);
```

Если преобразовать каждый элемент обоих массивов в целое число, а затем сравнить соответствующие числа, то эти два массива должны быть считаться эквивалентными. Это подтверждает результат:

```
True
```

Элемент

Следующие операции элементов позволяют извлекать отдельные элементы из входной последовательности.

First

Операция First возвращает первый элемент последовательности или первый элемент последовательности, соответствующий предикату — в зависимости от использованного прототипа.

Прототипы

Эта операция имеет два прототипа, описанные ниже.

Первый прототип First

```
public static T First<T>(
  this IEnumerable<T> source);
```

При использовании этого прототипа операции First выполняется перечисление входной последовательности source и возвращается ее первый элемент.

Второй прототип операции First позволяет передать ему предикат.

Второй прототип First

```
public static T First<T>(
  this IEnumerable<T> source,
  Func<T, bool> predicate);
```

Эта версия операции First возвращает первый найденный ею элемент, для которого predicate дал true. Если ни один из элементов не заставил predicate вернуть true, то операция First генерирует исключение InvalidOperationException.

Исключения

Если любой из аргументов равен null, генерируется исключение ArgumentNullException.

Если последовательность source пуста либо predicate ни разу не вернул true, генерируется исключение InvalidOperationException.

Примеры

Пример первого прототипа First приведен в листинге 5.16.

Листинг 5.16. Пример кода, вызывающего первый прототип First

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
string name = presidents.First();
Console.WriteLine(name);
```

Вот результат:

```
Adams
```

Может возникнуть вопрос: чем отличается эта операция от вызова операции Take с параметром 1? Отличие в том, что Take возвращает *последовательность* элементов, даже если она состоит всего из одного элемента. Операция First всегда возвращает в точности один *элемент* либо генерирует исключение, если возвращать нечего.

В листинге 5.17 приведен пример использования второго прототипа операции First.

Листинг 5.17. Пример кода, вызывающего второй прототип `First`

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
string name = presidents.First(p => p.StartsWith("H"));
Console.WriteLine(name);
```

Он должен вернуть первый элемент из входной последовательности, начинающийся со строки "H". Вот результат:

```
Harding
```

Вспомните, что если любой из прототипов операции First не находит элемент, который нужно возвратить, генерируется исключение InvalidOperationException. Чтобы избежать этого, используйте операцию FirstOrDefault.

FirstOrDefault

Операция FirstOrDefault подобна First во всем, кроме поведения, когда элемент не найден.

Прототипы

Эта операция имеет два прототипа, описанные ниже.

Первый прототип `FirstOrDefault`

```
public static T FirstOrDefault<T>(
  this IEnumerable<T> source);
```

Эта версия прототипа FirstOrDefault возвращает первый элемент, найденный во входной последовательности. Если последовательность пуста, возвращается default(T). Для ссылочных и допускающих null типов значением по умолчанию является null.

Второй прототип операции FirstOrDefault позволяет передать predicate, определяющий, какой элемент следует возвратить.

Второй прототип `FirstOrDefault`

```
public static T FirstOrDefault<T>(
  this IEnumerable<T> source,
  Func<T, bool> predicate);
```

Исключения

Если любой из аргументов равен null, генерируется исключение ArgumentNull Exception.

Примеры

В листинге 5.18 содержит пример использования первого прототипа FirstOrDefault, где элемент не найден. Для этого понадобилась пустая последовательность, которая была создана с помощью Take(0).

Листинг 5.18. Вызов первого прототипа `FirstOfDefault`; элемент не найден

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
```

```
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
string name = presidents.Take(0).FirstOrDefault();
Console.WriteLine(name == null ? "NULL" : name);
```

Вот результат:

```
NULL
```

В листинге 5.19 представлен тот же пример без вызова `Take(0)`, так что элемент будет найден.

Листинг 5.19. Вызов первого прототипа `FirstOfDefault`; элемент найден

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
string name = presidents.FirstOrDefault();
Console.WriteLine(name == null ? "NULL" : name);
```

И, наконец, ниже показан результат работы кода, когда элемент найден:

```
Adams
```

Для второго прототипа `FirstOrDefault` указывается, что нужен первый элемент, который начинается со строки `"B"` (листинг 5.20).

Листинг 5.20. Вызов второго прототипа `FirstOfDefault`; элемент найден

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
string name = presidents.FirstOrDefault(p => p.StartsWith("B"));
Console.WriteLine(name == null ? "NULL" : name);
```

И вот результат:

```
Buchanan
```

Теперь попробуем это предикатом с `predicate`, который не найдет соответствия (листинг 5.21).

Листинг 5.21. Вызов второго прототипа `FirstOfDefault`; элемент не найден

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
```

```
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
string name = presidents.FirstOrDefault(p => p.StartsWith("Z"));
Console.WriteLine(name == null ? "NULL" : name);
```

Поскольку в массиве `presidents` нет имени, начинающегося с `"Z"`, результат будет выглядеть следующим образом:

```
NULL
```

Last

Операция `Last` возвращает последний элемент последовательности или последний элемент, соответствующий предикату — в зависимости от используемого прототипа.

Прототипы

Эта операция имеет два прототипа, которые описаны ниже.

Первый прототип Last

```
public static T Last<T>(
  this IEnumerable<T> source);
```

В случае этого прототипа операция `Last` перечисляет входную последовательность по имени `source` и возвращает ее последний элемент.

Второй прототип `Last` позволяет передать `predicate` и рассматривается ниже.

Второй прототип Last

```
public static T Last<T>(
  this IEnumerable<T> source,
  Func<T, bool> predicate);
```

Эта версия операции `Last` возвращает последний из найденных элементов, для которых `predicate` вернет `true`.

Примеры

В листинге 5.22 содержится пример применения первого прототипа `Last`.

Листинг 5.22. Пример вызова первого прототипа Last

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
string name = presidents.Last();
Console.WriteLine(name);
```

Вот результат:

```
Wilson
```

Операция `Last` всегда возвращает в точности один элемент или же генерирует исключение, если возвращать нечего.

В листинге 5.23 представлен код примера применения второго прототипа `Last`.

Листинг 5.23. Вызов второго прототипа `Last`

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
string name = presidents.Last(p => p.StartsWith("H"));
Console.WriteLine(name);
```

Это должно вернуть последний элемент входной последовательности, начинающийся со строки "H". Вот результат запуска этого примера:

```
Hoover
```

Вспомните, что если любому из двух прототипов операции `Last` возвращать нечего, генерируется исключение `InvalidOperationException`. Чтобы избежать этого, используйте операцию `LastOrDefault`.

LastOrDefault

Операция `LastOrDefault` подобна `Last` во всем, за исключением поведения в случае, когда элемент не найден.

Прототипы

Эта операция имеет два прототипа, которые описаны ниже.

Первый прототип `LastOrDefault`

```
public static T LastOrDefault<T>(
  this IEnumerable<T> source);
```

Эта версия прототипа `LastOrDefault` возвращает последний элемент, найденный во входной последовательности. Если последовательность пуста, возвращается `default(T)`. Для ссылочных и допускающих `null` типов значением по умолчанию является `null`.

Второй прототип операции `LastOrDefault` позволяет передать ему `predicate`, определяющий, какой элемент должен быть возвращен.

Второй прототип `LastOrDefault`

```
public static T LastOrDefault<T>(
  this IEnumerable<T> source,
  Func<T, bool> predicate);
```

Исключения

Если любой из аргументов равен `null`, генерируется исключение `ArgumentNullException`.

Примеры

В листинге 5.24 содержится пример применения первого прототипа `LastOrDefault`, при этом элемент не найден. Для этого понадобилась пустая последовательность, которая была получена с помощью `Take(0)`.

Листинг 5.24. Вызов прототипа `LastOrDefault`; элемент не найден

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
```

```
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
string name = presidents.Take(0).LastOrDefault();
Console.WriteLine(name == null ? "NULL" : name);
```

Вот результат:

```
NULL
```

В листинге 5.25 приведен тот же пример, но без `Take(0)`, поэтому элемент будет найден.

Листинг 5.25. Вызов прототипа `LastOrDefault`; элемент найден

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
string name = presidents. LastOrDefault();
Console.WriteLine(name == null ? "NULL" : name);
```

Ниже показан результат:

```
Wilson
```

При использовании второго прототипа операции `LastOrDefault` в листинге 5.26 указано, что нужен последний элемент, начинающийся со строки `"B"`.

Листинг 5.26. Вызов второго прототипа `LastOrDefault`; элемент найден

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
string name = presidents.LastOrDefault(p => p.StartsWith("B"));
Console.WriteLine(name == null ? "NULL" : name);
```

Результат будет таким:

```
Bush
```

Теперь попробуем указать предикат `predicate`, который не найдет соответствия (листинг 5.27).

Листинг 5.27. Вызов второго прототипа `LastOrDefault`; элемент не найден

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
```

```
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
string name = presidents.LastOrDefault(p => p.StartsWith("Z"));
Console.WriteLine(name == null ? "NULL" : name);
```

Поскольку ни одно имя в массиве `presidents` не начинается с `"Z"`, результат вполне ожидаемый:

```
NULL
```

Single

Операция `Single` возвращает единственный элемент последовательности или единственный элемент последовательности, соответствующий предикату — в зависимости от используемого прототипа.

Прототипы

Эта операция имеет два прототипа, описанные ниже.

Первый прототип Single

```
public static T Single<T> (
  this IEnumerable<T> source);
```

В случае использования этого прототипа операция `Single` перечисляет входную последовательность по имени `source` и возвращает единственный ее элемент.

Второй прототип `Single` принимает `predicate` и выглядит следующим образом.

Второй прототип Single

```
public static T Single<T> (
  this IEnumerable<T> source,
  Func<T, bool> predicate);
```

Эта версия операции `Single` возвращает единственный найденный элемент, для которого `predicate` дает `true`. Если ни один из элементов не заставляет `predicate` вернуть `true`, то операция `Single` генерирует исключение `InvalidOperationException`.

Исключения

Если любой из аргументов равен `null`, генерируется исключение `ArgumentNullException`.

Если последовательность `source` пуста, либо `predicate` ни разу не вернул `true` или нашел более одного элемента, для которых вернул `true`, генерируется исключение `InvalidOperationException`.

Примеры

В листинге 5.28 содержится пример применения первого прототипа `Single`, использующий общий класс `Employee`.

Листинг 5.28. Пример вызова первого прототипа Single

```
Employee emp = Employee.GetEmployeesArray()
  .Where(e => e.id == 3).Single();
Console.WriteLine("{0} {1}", emp.firstName, emp.lastName);
```

В данном примере вместо ожидания получения последовательности от запроса мне нужна просто ссылка на определенного сотрудника. Операция `Single` очень полезна в

такой ситуации — до тех пор, пока вы можете гарантировать, что в переданной ему последовательности есть только один элемент. В данном случае, поскольку я вызвал операцию Where, указав уникальный ключ, я в безопасности. И вот результат:

```
Anders Hejlsberg
```

Листинг 5.29 содержит пример кода, использующего второй прототип операции Single.

Листинг 5.29. Пример вызова второго прототипа Single

```
Employee emp = Employee.GetEmployeesArray()
  .Single(e => e.id == 3);
Console.WriteLine("{0} {1}", emp.firstName, emp.lastName);
```

Данный код функционально эквивалентен предыдущему примеру. Вместо вызова операции Where для того, чтобы гарантировать *наличие* единственного элемента в последовательности, можно выполнить ту же операцию фильтрации последовательности в операции Single. Она должна вернуть только один элемент из входной последовательности, id которого равен 3. Ниже показан результат:

```
Anders Hejlsberg
```

Вспомните: если любой прототип операции Single не находит элемента для возврата, генерируется исключение InvalidOperationException. Избежать этого позволяет операция SingleOrDefault.

SingleOrDefault

Операция SingleOrDefault подобна Single, но отличается поведением в случае, когда элемент не найден.

Прототипы

Эта операция имеет два прототипа, описанные ниже.

Первый прототип SingleOrDefault

```
public static T SingleOrDefault<T>(
  this IEnumerable<T> source);
```

Эта версия SingleOrDefault возвращает единственный элемент, найденный во входной последовательности. Если последовательность пуста, возвращается default(T). Для ссылочных и допускающих null типов значением по умолчанию является null. Если найдено более одного элемента, генерируется исключение InvalidOperationException.

Второй прототип операции SingleOrDefault позволяет передать predicate, который определяет, какой элемент должен возвращаться.

Второй прототип SingleOrDefault

```
public static T SingleOrDefault<T>(
  this IEnumerable<T> source,
  Func<T, bool> predicate);
```

Исключения

Если любой из аргументов равен null, генерируется исключение ArgumentNull Exception.

Если predicate встречает более одного элемента, для которого возвращает true, генерируется исключение InvalidOperationException.

Примеры

В листинге 5.30 представлен пример вызова первого прототипа `SingleOrDefault`, когда элемент не найден. Для этого понадобилась пустая последовательность. Будет использована операция `Where` с указанием несуществующего значения ключа в качестве условия фильтра.

Листинг 5.30. Вызов первого прототипа `SingleOrDefault`; элемент не найден

```
Employee emp = Employee.GetEmployeesArray()
  .Where(e => e.id == 5).SingleOrDefault();

Console.WriteLine(emp == null ? "NULL" :
  string.Format("{0} {1}", emp.firstName, emp.lastName));
```

В коде запрашивается сотрудник с `id` равным `5`. Поскольку известно, что такого нет, возвращается пустая последовательность. В отличие от операции `Single`, `SingleOrDefault` нормально обрабатывает пустые последовательности. Вот результат работы примера:

```
NULL
```

В листинге 5.31 демонстрируется тот же пример, но с найденным единственным элементjv. Для получения последовательности, содержащей единственный элемент, используется операция `Where`.

Листинг 5.31. Вызов первого прототипа `SingleOrDefault`; элемент найден

```
Employee emp = Employee.GetEmployeesArray()
  .Where(e => e.id == 4).SingleOrDefault();

Console.WriteLine(emp == null ? "NULL" :
  string.Format("{0} {1}", emp.firstName, emp.lastName));
```

На этот раз указан существующий `id`. Результат работы кода, когда элемент найден, выглядит следующим образом:

```
David Lightman
```

Как видите, сотрудник найден. При использовании второго прототипа `SingleOrDefault` в листинге 5.32 указывается `id`, который точно существует, но вместо применения операции `Where` фильтр встраивается в сам вызов операции `SingleOrDefault`.

Листинг 5.32. Вызов второго прототипа `SingleOrDefault`; элемент найден

```
Employee emp = Employee.GetEmployeesArray()
  .SingleOrDefault(e => e.id == 4);

Console.WriteLine(emp == null ? "NULL" :
  string.Format("{0} {1}", emp.firstName, emp.lastName));
```

Этот пример функционально эквивалентен предыдущему, но с тем отличием, что вместо фильтрации элементов посредством операции `Where` это делается передачей предиката операции `SingleOrDefault`. Вот результат:

```
David Lightman
```

Давайте попробуем сделать то же самое, но с предикатом, который не найдет соответствия (листинг 5.33).

Листинг 5.33. Вызов второго прототипа SingleOrDefault; элемент не найден

```
Employee emp = Employee.GetEmployeesArray()
  .SingleOrDefault(e => e.id == 5);
Console.WriteLine(emp == null ? "NULL" :
  string.Format("{0} {1}", emp.firstName, emp.lastName));
```

Поскольку нет элемента с id равным 5, никакого элемента не найдено. Об этом свидетельствует результат:

```
NULL
```

Когда элемент не обнаруживается во входной последовательности, операция SingleOrDefault изящно обрабатывает эту ситуацию вместо генерации исключений.

ElementAt

Операция ElementAt возвращает элемент из исходной последовательности по указанному индексу.

Прототипы

Эта операция имеет один прототип, описанный ниже.

Прототип ElementAt

```
public static T ElementAt<T>(
  this IEnumerable<T> source,
  int index);
```

Если последовательность реализует IList<T>, то интерфейс IList используется для извлечения индексированного элемента непосредственно. Если же последовательность не реализует IList<T>, то последовательность перечисляется до тех пор, пока не будет достигнут указанный индексом элемент. Исключение ArgumentOutOfRangeException генерируется, если индекс меньше нуля либо больше или равен количеству элементов в последовательности.

На заметку! В C# индексы начинаются с нуля. Это значит, что индекс первого элемента равен нулю. Индекс последнего элемента равен количеству элементов последовательности минус 1.

Исключения

Если аргумент source равен null, генерируется исключение ArgumentNullException.

Если значение index меньше нуля или больше или равно количеству элементов в последовательности, генерируется исключение ArgumentOutOfRangeException.

Примеры

В листинге 5.34 приведен пример вызова единственного прототипа операции ElementAt.

Листинг 5.34. Вызов операции ElementAt

```
Employee emp = Employee.GetEmployeesArray()
  .ElementAt(3);
Console.WriteLine("{0} {1}", emp.firstName, emp.lastName);
```

Здесь указано, что требуется элемент, индекс которого равен 3, т.е. четвертый по счету. И вот результат этого запроса:

```
David Lightman
```

ElementAtOrDefault

Операция ElementAtOrDefault возвращает из исходной последовательности элемент, имеющий указанный индекс местоположения.

Прототипы

Эта операция имеет один прототип, описанный ниже.

Прототип ElementAtOrDefault

```
public static T ElementAtOrDefault<T>(
  this IEnumerable<T> source,
  int index);
```

Если последовательность реализует IList<T>, то интерфейс IList используется для извлечения индексированного элемента непосредственно. Если же последовательность не реализует IList<T>, то последовательность перечисляется до тех пор, пока не будет достигнутый указанный индексом элемент. Если заданный индекс меньше нуля либо больше или равен количеству элементов в последовательности, возвращается default(T). Для ссылочных и допускающих null типов значение по умолчанию равно null. Это поведение отличает его от операции ElementAt.

Исключения

Если аргумент source равен null, генерируется исключение ArgumentNullException.

Примеры

В листинге 5.35 содержится пример вызова операции ElementAtOrDefault с допустимым индексом.

Листинг 5.35. Вызов операции ElementAtOrDefault с допустимым индексом

```
Employee emp = Employee.GetEmployeesArray()
  .ElementAtOrDefault(3);

Console.WriteLine(emp == null ? "NULL" :
  string.Format("{0} {1}", emp.firstName, emp.lastName));
```

Вот результат запроса:

```
David Lightman
```

Как и ожидалось, извлечен элемент с индексом 3. Теперь запустим запрос с недопустимым индексом, используя код, приведенный в листинге 5.36.

Листинг 5.36. Вызов операции ElementAtOrDefault с недопустимым индексом

```
Employee emp = Employee.GetEmployeesArray()
  .ElementAtOrDefault(5);

Console.WriteLine(emp == null ? "NULL" :
  string.Format("{0} {1}", emp.firstName, emp.lastName));
```

Поскольку в последовательности нет элемента с индексом 5, результат будет таким:

```
NULL
```

Квантификаторы

Следующие операции-квантификаторы позволяют выполнять операции квантификации над входной последовательностью.

Any

Операция Any возвращает true, если любой из элементов входной последовательности отвечает условию.

Прототипы

Эта операция имеет два прототипа, которые описаны ниже.

Первый прототип Any

```
public static bool Any<T>(
  this IEnumerable<T> source);
```

Этот прототип операции Any вернет true, если входная последовательность source содержит любые элементы. Второй прототип операции Any перечисляет входную последовательность и возвращает true, если хотя бы для одного элемента из входной последовательности вызов делегата метода predicate возвращает true. Перечисление входной последовательности source прекращается, как только predicate вернет true.

Второй прототип Any

```
public static bool Any<T>(
  this IEnumerable<T> source,
  Func<T, bool> predicate);
```

Исключения

Если любой из аргументов равен null, генерируется исключение ArgumentNullException.

Примеры

Для начала воспроизведем ситуацию с пустой последовательностью, как показано в листинге 5.37. В коде используется операция Empty, описанная в предыдущей главе.

Листинг 5.37. Первый прототип Any при отсутствии элементов во входной последовательности

```
bool any = Enumerable.Empty<string>().Any();
Console.WriteLine(any);
```

Вот результат выполнения этого кода:

```
False
```

Теперь вызовем этот же прототип с непустой входной последовательностью, как показано в листинге 5.38.

Листинг 5.38. Первый прототип Any при наличии элементов во входной последовательности

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
```

```
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
bool any = presidents.Any();
Console.WriteLine(any);
```

Вот результат запуска этого кода:

```
True
```

В следующем примере используется второй прототип — сначала без элементов, соответствующих `predicate`, как показано в листинге 5.39.

Листинг 5.39. Второй прототип `Any`, когда нет элементов, для которых предикат возвращает `true`

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
bool any = presidents.Any(s => s.StartsWith("Z"));
Console.WriteLine(any);
```

Здесь указано, что нужно извлечь президентов с именами, начинающимися со строки `"Z"`. Поскольку таких нет, будет возвращена пустая последовательность, что заставит операцию `Any` вернуть `false`:

```
False
```

И, наконец, ниже представлен пример использования второго прототипа с `predicate`, который должен вернуть `true`, по крайней мере, для одного элемента (листинг 5.40).

Листинг 5.40. Второй прототип `Any`, когда предикат возвращает `true` минимум для одного элемента

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
bool any = presidents.Any(s => s.StartsWith("A"));
Console.WriteLine(any);
```

Вот результат:

```
True
```

All

Операция `All` возвращает `true`, если каждый элемент входной последовательности отвечает условию.

Прототипы

Операция `All` имеет один прототип, описанный ниже.

Прототип All

```
public static bool All<T>(
  this IEnumerable<T> source,
  Func<T, bool> predicate);
```

Операция All перечисляет входную последовательность source и возвращает true, только если predicate возвращает true для каждого элемента последовательности. После возврата из predicate значения false перечисление прекращается.

Исключения

Если любой из аргументов равен null, генерируется исключение ArgumentNullException.

Примеры

В листинге 5.41 используется predicate, о котором известно, что для некоторых элементов он вернет false.

Листинг 5.41. Прототип All, когда не все элементы заставляют предикат вернуть true

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};

bool all = presidents.All(s => s.Length > 5);
Console.WriteLine(all);
```

Поскольку известно, что не все имена президентов в массиве имеют длину более пяти символов, известно, что predicate вернет false для некоторых элементов. Вот результат:

```
False
```

Теперь воспроизведем случай, когда каждый элемент последовательности заставит predicate вернуть true, как показано в листинге 5.42.

Листинг 5.42. Прототип All, когда все элементы заставляют предикат вернуть true

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};

bool all = presidents.All(s => s.Length > 3);
Console.WriteLine(all);
```

Поскольку известно, что каждое имя президента состоит из не менее чем четырех символов, операция All должна вернуть true. Что и можно видеть в выводе:

```
True
```

Contains

Операция Contains возвращает true, если любой элемент входной последовательности соответствует указанному значению.

Прототипы

Эта операция имеет два прототипа, которые описаны ниже.

Первый прототип Contains

```
public static bool Contains<T>(
  this IEnumerable<T> source,
  T value);
```

Этот прототип операции Contains сначала проверяет входную последовательность на предмет реализации ею интерфейса ICollection<T>, и если она его реализует, то вызывается метод Contains реализации последовательности. Если последовательность не реализует интерфейс ICollection<T>, входная последовательность source перечисляется с проверкой соответствия каждого элемента указанному значению. Как только найден элемент, который ему не соответствует, перечисление прекращается.

Указанное значение сравнивается с каждым элементом с использованием класса проверки эквивалентности по умолчанию EqualityComparer<K>.Default.

Второй прототип подобен первому, за исключением возможности указания объекта IEqualityComparer<T>. Если используется этот прототип, каждый элемент в последовательности сравнивается с переданным значением с применением переданного объекта проверки эквивалентности.

Второй прототип Contains

```
public static bool Contains<T>(
  this IEnumerable<T> source,
  T value,
  IEqualityComparer<T> comparer);
```

Исключения

Если аргумент source равен null, генерируется исключение ArgumentNullException.

Примеры

Для демонстрации работы первого прототипа начнем со значения, о котором известно, что оно отсутствует во входной последовательности (листинг 5.43).

Листинг 5.43. Прототип Contains, когда ни один элемент не соответствует указанному значению

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
bool contains = presidents.Contains("Rattz");
Console.WriteLine(contains);
```

Поскольку в массиве нет элемента со значением "Rattz", переменная contains должна быть равна false. Вот вывод:

```
False
```

В листинге 5.44 известно, что элемент соответствует указанному значению.

Листинг 5.44. Прототип `Contains`, когда элемент соответствует указанному значению

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
bool contains = presidents.Contains("Hayes");
Console.WriteLine(contains);
```

Поскольку элемент со значением `"Hayes"` существует, переменная `contains` должна быть равна `true`. И вот результат:

```
True
```

При демонстрации примера использования второго прототипа `Contains` задействован общий класс `MyStringifiedNumberComparer`. Массив чисел в строковом формате проверяется на предмет наличия числа, формально не эквивалентного ни одному элементу массива, но поскольку применяется собственный класс проверки эквивалентности, соответствующий элемент будет найден. Код приведен в листинге 5.45.

Листинг 5.45. Второй прототип `Contains`, когда элемент соответствует указанному значению

```
string[] stringifiedNums = {
  "001", "49", "017", "0080", "00027", "2" };
bool contains = stringifiedNums.Contains("0000002",
                                 new MyStringifiedNumberComparer());
Console.WriteLine(contains);
```

Поскольку ищется элемент со значением `"0000002"` и используется объект проверки эквивалентности, который перед сравнением преобразует это строковое значение наряду со всеми элементами последовательности в целое число, и т.к. последовательность содержит элемент `"2"`, переменная `contains` должна быть равна `true`. Взглянем на результат:

```
True
```

Теперь попробуем тот же пример, но на этот раз запросим элемент, который, как известно, не существует. Код показан в листинге 5.46.

Листинг 5.46. Второй прототип `Contains`, когда элемент не соответствует указанному значению

```
string[] stringifiedNums = {
  "001", "49", "017", "0080", "00027", "2" };
bool contains = stringifiedNums.Contains("0000271",
                                 new MyStringifiedNumberComparer());
Console.WriteLine(contains);
```

Поскольку ни один из элементов не может быть преобразован в целое число, равное `271`, в массиве производится поиск значения `"000271"`. И вот результат:

```
False
```

Агрегация

Описанные далее операции выполняют агрегатные операции над элементами входной последовательности.

Count

Операция Count возвращает количество элементов во входной последовательности.

Прототипы

Эта операция имеет два прототипа, описанные ниже.

Первый прототип Count

```
public static int Count<T>(
  this IEnumerable<T> source);
```

Этот прототип операции Count возвращает общее количество элементов во входной последовательности, проверяя сначала, реализует ли она интерфейс ICollection<T>, и если да, то получает счетчик последовательности через реализацию этого интерфейса. Если же входная последовательность source не реализует интерфейс ICollection<T>, операция Count перечисляет всю эту последовательность, подсчитывая количество элементов.

Второй прототип операции Count перечисляет входную последовательность source и подсчитывает все элементы, которые заставляют делегат метода predicate вернуть true.

Второй прототип Count

```
public static int Count<T>(
  this IEnumerable<T> source,
  Func<T, bool> predicate);
```

Исключения

Если любой из аргументов равен null, генерируется исключение ArgumentNullException.

Если значение count превышает Int32.MaxValue, генерируется исключение OverflowException.

Примеры

Код в листинге 5.47 начинается с использования первого прототипа. Сколько элементов содержится в последовательности presidents?

Листинг 5.47. Первый прототип Count

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};

int count = presidents.Count();
Console.WriteLine(count);
```

Вот результат:

```
38
```

Теперь попробуем второй прототип, как показано в листинге 5.48. На этот раз подсчитывается количество президентов с фамилией, начинающейся с "J".

Листинг 5.48. Второй прототип Count

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
int count = presidents.Count(s => s.StartsWith("J"));
Console.WriteLine(count);
```

Результат работы этого кода:

```
3
```

А как быть, если количество элементов превышает значение Int32.MaxValue? Для таких ситуаций предусмотрена операция LongCount.

LongCount

Операция LongCount возвращает количество элементов входной последовательности как значение типа long.

Прототипы

Эта операция имеет два прототипа, описанные ниже.

Первый прототип LongCount

```
public static long LongCount<T>(
  this IEnumerable<T> source);
```

Этот прототип операции LongCount возвращает общее количество элементов во входной последовательности, перечисляя всю эту последовательность и подсчитывая число элементов.

Второй прототип операции LongCount перечисляет входную последовательность source и подсчитывает все элементы, которые заставляют делегат метода predicate вернуть true.

Второй прототип LongCount

```
public static int LongCount<T>(
  this IEnumerable<T> source,
  Func<T, bool> predicate);
```

Исключения

Если любой из аргументов равен null, генерируется исключение ArgumentNullException.

Примеры

Начнем с примера применения первого прототипа, показанного в листинге 5.49. Можно было бы просто повторить те же два примера, которые сопровождали описание операции Count, изменив соответствующие места на тип long, но это не слишком наглядно. Поскольку нереально описать достаточно длинную статическую последовательность, для которой понадобилась бы операция LongCount, она генерируется с помощью стандартной операции запроса. К сожалению, операции генерации, описанные в пре-

дыдущей главе, позволяют указать количество элементов только как значение `int`. Для получения достаточного количества элементов придется соединить вместе пару таких сгенерированных последовательностей, чтобы потребовалась операция `LongCount`.

Листинг 5.49. Первый прототип `LongCount`

```
long count = Enumerable.Range(0, int.MaxValue).
  Concat(Enumerable.Range(0, int.MaxValue)).LongCount();
Console.WriteLine(count);
```

Как видите, были сгенерированы две последовательности с использованием операции `Range`, описанной в предыдущей главе, и соединены вместе с помощью операции `Concat`, также описанной в предыдущей главе.

Внимание! Выполнение этого примера займет немало времени. Например, на машине с четырехядерным процессором и 4 Гбайт памяти для этого потребовалось около минуты.

Перед запуском примера наберитесь терпения — он выполняется долго, возможно, даже несколько минут. В конце концов, будут сгенерированы две последовательности, каждая по 2 147 483 647 элементов. Ниже показан результат:

```
4294967294
```

Если вы запустите тот же пример с использованием операции `Count`, возникнет исключение. Теперь давайте попробуем вызвать второй прототип. В этом примере используется тот же базовый подход, что и в предыдущем, за исключением того, что указывается `predicate`, который возвратит `true` только для целых чисел больше 1 и меньше 4. То есть — для элементов 2 и 3. Поскольку есть две последовательности с одинаковыми значениями, в результате должно быть получено значение количества элементов, равное 4. Код приведен в листинге 5.50.

Листинг 5.50. Пример вызова второго прототипа `LongCount`

```
long count = Enumerable.Range(0, int.MaxValue).
  Concat(Enumerable.Range(0, int.MaxValue)).LongCount(n => n > 1 && n < 4);
Console.WriteLine(count);
```

Этот код похож на код предыдущего примера, за исключением того, что дополнительно указан предикат `predicate`. Его выполнение займет даже больше времени, чем предыдущего примера.

Результат работы этого кода выглядит так:

```
4
```

Sum

Операция `Sum` возвращает сумму числовых значений, содержащихся в элементах последовательности.

Прототипы

Эта операция имеет два прототипа, описанные ниже.

Первый прототип `Sum`

```
public static Numeric Sum(
  this IEnumerable<Numeric> source);
```

Тип *Numeric* должен быть одним из int, long, double или decimal, либо одним из их допускающих null эквивалентов: int?, long?, double? или decimal?.

Первый прототип операции Sum возвращает сумму всех элементов входной последовательности source.

Пустая последовательность приведет к возврату нуля. Операция Sum не включает значения null в результат для числовых типов, допускающих null.

Второй прототип Sum ведет себя так же, как и первый, но суммирует только те значения входной последовательности, которые выбраны делегатом метода selector.

Второй прототип Sum

```
public static Numeric Sum<T>(
  this IEnumerable<T> source,
  Func<T, Numeric> selector);
```

Исключения

Если любой из аргументов равен null, генерируется исключение ArgumentNull Exception.

Если сумма оказывается слишком большой, чтобы уместиться в тип Numeric, и если тип Numeric отличается от decimal или decimal?, генерируется исключение OverflowException. Если же типом Numeric является decimal или decimal?, возвращается положительная или отрицательная бесконечность.

Примеры

Начнем с примера применения первого прототипа, который показан в листинге 5.51. Сначала генерируется последовательность целых чисел с помощью операции Range, а затем используется операция Sum для их суммирования.

Листинг 5.51. Пример вызова первого прототипа Sum

```
IEnumerable<int> ints = Enumerable.Range(1, 10);
foreach (int i in ints)
  Console.WriteLine(i);

Console.WriteLine("--");

int sum = ints.Sum();
  Console.WriteLine(sum);
```

Результат выглядит следующим образом:

```
1
2
3
4
5
6
7
8
9
10
--
55
```

Теперь попробуем второй прототип, как показано в листинге 5.52. В этом примере задействован общий класс EmployeeOptionEntry, и будут суммироваться опционы для всех сотрудников.

Листинг 5.52. Пример вызова второго прототипа `Sum`

```
IEnumerable<EmployeeOptionEntry> options =
  EmployeeOptionEntry.GetEmployeeOptionEntries();

long optionsSum = options.Sum(o => o.optionsCount);
Console.WriteLine("Сумма опционов сотрудников: {0}", optionsSum);
```

Вместо того чтобы пытаться суммировать весь элемент, что не имело бы смысла в данном примере, поскольку речь идет об объекте сотрудника, можно использовать элемент `selector` второго прототипа, чтобы извлечь только интересующие члены — в данном случае `optionsCount`. Результат выполнения этого кода показан ниже:

```
Сумма опционов сотрудников: 51504
```

Min

Операция `Min` возвращает минимальное значение входной последовательности.

Прототипы

Эта операция имеет четыре прототипа, которые описаны ниже.

Первый прототип `Min`

```
public static Numeric Min(
  this IEnumerable<Numeric> source);
```

Тип *Numeric* должен быть одним из `int`, `long`, `double` или `decimal`, либо одним из их допускающих `null` эквивалентов: `int?`, `long?`, `double?` или `decimal?`.

Первый прототип операции `Min` возвращает элемент с минимальным числовым значением из входной последовательности `source`. Если тип элемента реализует интерфейс `IComparable<T>`, этот интерфейс применяется для сравнения элементов. Если же элемент не реализует интерфейс `IComparable<T>`, будет использован необобщенный интерфейс `IComparable`.

Пустая последовательность либо последовательность, состоящая только из значений `null`, вернет `null`.

Второй прототип операции `Min` ведет себя подобно первому, за исключением того, что он предназначен для нечисловых типов.

Второй прототип `Min`

```
public static T Min<T>(
  this IEnumerable<T> source);
```

Третий прототип предназначен для типов *Numeric* и подобен первому, но с возможностью указания делегата метода `selector`, который позволяет сравнивать члены каждого элемента входной последовательности в процессе поиска минимального значения и возвращать это минимальное значение.

Третий прототип `Min`

```
public static T Min<T>(
  this IEnumerable<T> source,
  Func<T, Numeric> selector);
```

Четвертый прототип предназначен для нечисловых типов и подобен второму, но с возможностью указания делегата метода `selector`, который позволяет сравнивать члены каждого элемента входной последовательности в процессе поиска минимального значения и возвращать это минимальное значение.

Четвертый прототип Min

```
public static S Min<T, S>(
  this IEnumerable<T> source,
  Func<T, S> selector);
```

Исключения

Если любой из аргументов равен null, генерируется исключение ArgumentNullException.

Если последовательность source пуста для Numeric-версий прототипов и если тип T не допускает значений null, такой как int, long, double или decimal, генерируется исключение InvalidOperationException. Если типы допускают null, подобно int?, long?, double? или decimal?, то вместо этого операция возвращает null.

Примеры

В примере использования прототипа Min, показанном в листинге 5.53, объявляется массив целых чисел и затем возвращается минимальное из них.

Листинг 5.53. Пример вызова первого прототипа Min

```
int[] myInts = new int[] { 974, 2, 7, 1374, 27, 54 };
int minInt = myInts.Min();
Console.WriteLine(minInt);
```

Это очень тривиальный пример, и его результат выглядит так:

```
2
```

В примере использования второго прототипа, показанного в листинге 5.54, операцию Min просто вызывается на стандартном массиве presidents. Она должна вернуть элемент с минимальным значением в алфавитном порядке.

Листинг 5.54. Пример вызова второго прототипа Min

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};

string minName = presidents.Min();
Console.WriteLine(minName);
```

Этот пример выдает следующий результат:

```
Adams
```

Получился тот же вывод, что и при вызове операции First. Такое совпадение объясняется лишь тем, что массив presidents упорядочен по алфавиту. Если бы его элементы были неупорядочены либо располагались в каком-то другом порядке, то результат все равно был бы Adams.

В примере применения третьего прототипа операции Min используется общий класс Actor для поиска самой ранней даты рождения актера посредством вызова Min на годах рождения.

В листинге 5.55 содержится соответствующий код с вызовом операции Min.

Листинг 5.55. Пример вызова третьего прототипа Min

```
int oldestActorAge = Actor.GetActors().Min(a => a.birthYear);
Console.WriteLine(oldestActorAge);
```

В результате запуска этого примера выводится год рождения самого старшего актера:

```
1960
```

В примере использования четвертого прототипа Min, показанном в листинге 5.56, снова задействован класс Actor. Здесь извлекается фамилия актера, которая находится первой в алфавитном порядке.

Листинг 5.56. Пример вызова четвертого прототипа Min

```
string firstAlphabetically = Actor.GetActors().Min(a => a.lastName);
Console.WriteLine(firstAlphabetically);
```

Вот результат:

```
Bullock
```

Max

Операция Max возвращает максимальное значение из входной последовательности.

Прототипы

Эта операция имеет четыре прототипа, описанные ниже.

Первый прототип Max

```
public static Numeric Max(
  this IEnumerable<Numeric> source);
```

Тип *Numeric* должен быть одним из int, long, double или decimal, либо одним из их допускающих null эквивалентов: int?, long?, double? или decimal?.

Первый прототип операции Max возвращает элемент с максимальным числовым значением из входной последовательности source. Если тип элемента реализует интерфейс IComparable<T>, этот интерфейс применяется для сравнения элементов. Если же элемент не реализует интерфейс IComparable<T>, будет использован необобщенный интерфейс IComparable.

Пустая последовательность либо последовательность, состоящая только из значений null, вернет null.

Второй прототип операции Max ведет себя подобно первому, за исключением того, что он предназначен для нечисловых типов.

Второй прототип Max

```
public static T Max<T>(
  this IEnumerable<T> source);
```

Третий прототип предназначен для значений *Numeric* и подобен первому, но с возможностью указания делегата метода selector, который позволяет сравнивать члены каждого элемента входной последовательности в процессе поиска максимального значения и возвращать это максимальное значение.

Третий прототип Max

```
public static T Max<T>(
  this IEnumerable<T> source,
  Func<T, Numeric> selector);
```

Четвертый прототип предназначен для нечисловых типов и подобен второму, но с возможностью указания делегата метода `selector`, который позволяет сравнивать члены каждого элемента входной последовательности в процессе поиска максимального значения и возвращать это максимальное значение.

Четвертый прототип `Max`

```
public static S Max<T, S>(
  this IEnumerable<T> source,
  Func<T, S> selector);
```

Исключения

Если любой из аргументов равен `null`, генерируется исключение `ArgumentNullException`.

Если последовательность `source` пуста для `Numeric`-версий прототипов и если тип `T` не допускает значений `null`, такой как `int`, `long`, `double` или `decimal`, генерируется исключение `InvalidOperationException`. Если типы допускают `null`, подобно `int?`, `long?`, `double?` или `decimal?`, то вместо этого операция возвращает `null`.

Примеры

В примере применения первого прототипа `Max`, показанном в листинге 5.57, объявляется массив целых чисел и возвращается максимальное из них.

Листинг 5.57. Пример вызова первого прототипа `Min`

```
int[] myInts = new int[] { 974, 2, 7, 1374, 27, 54 };
int maxInt = myInts.Max();
Console.WriteLine(maxInt);
```

Результат выглядит следующим образом:

```
1374
```

В примере использования второго прототипа, показанном в листинге 5.58, операция `Max` просто вызывается на стандартном массиве `presidents`.

Листинг 5.58. Пример вызова второго прототипа `Max`

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
  "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};

string maxName = presidents.Max();
Console.WriteLine(maxName);
```

Это даст следующий результат:

```
Wilson
```

Как я уже упоминалось при описании эквивалентного примера операции `Min`, хотя данный пример выдает тот же результат, что выдала бы операция `Last`, это только потому, что массив `presidents` упорядочен по алфавиту.

В примере применения третьего прототипа операции `Max` используется общий класс `Actor` для поиска самого младшего актера, для чего операция `Max` вызывается на годах рождения. Код приведен в листинге 5.59.

Листинг 5.59. Пример вызова третьего прототипа Max

```
int oldestActorAge = Actor.GetActors().Max(a => a.birthYear);
Console.WriteLine(oldestActorAge);
```

В результате запуска примера выводится год рождения самого молодого актера:

```
1968
```

В примере использования четвертого прототипа Max, показанном в листинге 5.60, с применением того же класса Actor извлекается фамилия актера, которая идет последней в алфавитном порядке.

Листинг 5.60. Пример вызова четвертого прототипа Max

```
string firstAlphabetically = Actor.GetActors().Max(a => a.lastName);
Console.WriteLine(firstAlphabetically);
```

Результат будет таким:

```
Wilson
```

Average

Операция Average возвращает среднее арифметическое числовых значений элементов входной последовательности.

Прототипы

Эта операция имеет два прототипа, описанные ниже.

Первый прототип Average

```
public static Result Average(
  this IEnumerable<Numeric> source);
```

Тип *Numeric* должен быть одним из int, long, double или decimal, либо одним из их допускающих null эквивалентов: int?, long?, double? или decimal?. Если тип *Numeric* — int или long, то типом Result будет double. Если же тип *Numeric* — int? или long?, то типом Result будет double?. В противном случае тип Result совпадет с типом *Numeric*. Первый прототип операции Average перечисляет входную последовательность source элементов типа Numeric, вычисляя их среднее значение.

Второй прототип операции Average перечисляет входную последовательность source элементов и определяет среднее значение членов элементов, возвращаемых функцией selector для каждого элемента входной последовательности.

Второй прототип Average

```
public static Result Average<T>(
  this IEnumerable<T> source,
  Func<T, Numeric> selector);
```

Исключения

Если любой из аргументов равен null, генерируется исключение ArgumentNullException. Если сумма усредняемых значений превышает емкость long для типов *Numeric* — int, int?, long и long?, генерируется исключение OverflowException.

Примеры

Начнем с примера применения первого прототипа, который показан в листинге 5.61. В этом примере используется операция Range, чтобы создать последовательность целых чисел, а затем подсчитать их среднее значение.

Листинг 5.61. Пример вызова первого прототипа `Average`

```
IEnumerable<int> intSequence = Enumerable.Range(1, 10);
Console.WriteLine("Последовательность целых чисел:");

foreach (int i in intSequence)
  Console.WriteLine(i);

double average = intSequence.Average();
Console.WriteLine("Среднее значение: {0}", average);
```

Результат этого примера:

```
Последовательность целых чисел:
1
2
3
4
5
6
7
8
9
10
Среднее значение: 5.5
```

Теперь попробуем воспользоваться вторым прототипом, который обращается к членам элементов. В примере, приведенном в листинге 5.62, используется общий класс `EmployeeOptionEntry`.

Листинг 5.62. Пример вызова второго прототипа `Average`

```
IEnumerable<EmployeeOptionEntry> options =
EmployeeOptionEntry.GetEmployeeOptionEntries();
Console.WriteLine("Список идентификаторов сотрудников и опционов:");

foreach (EmployeeOptionEntry eo in options)
  Console.WriteLine("Идентификатор сотрудника: {0}, Опционы: {1}",
                    eo.id, eo.optionsCount);

// Теперь получим среднее значений опций.
double optionAverage = options.Average(o => o.optionsCount);
Console.WriteLine("Среднее значение опционов, присужденных сотрудникам: {0}",
                  optionAverage);
```

Сначала извлекаются объекты `EmployeeOptionEntry`. Затем выполняется перечисление последовательности объектов с отображением каждого из них. В конце вычисляется и отображается среднее значение опционов. Результат работы этого кода выглядит следующим образом:

```
Список идентификаторов сотрудников и опционов:
Идентификатор сотрудника: 1, Опционы: 2
Идентификатор сотрудника: 2, Опционы: 10000
Идентификатор сотрудника: 2, Опционы: 10000
Идентификатор сотрудника: 3, Опционы: 5000
Идентификатор сотрудника: 2, Опционы: 10000
Идентификатор сотрудника: 3, Опционы: 7500
Идентификатор сотрудника: 3, Опционы: 7500
Идентификатор сотрудника: 4, Опционы: 1500
Идентификатор сотрудника: 101, Опционы: 2
Среднее значение опционов, присужденных сотрудникам: 5722.66666666667
```

Aggregate

Операция Aggregate выполняет указанную пользователем функцию на каждом элементе входной последовательности, передавая значение, возвращенное этой функцией для предыдущего элемента, и возвращая ее значение для последнего элемента.

Прототипы

Эта операция имеет два прототипа, которые описаны ниже.

Первый прототип Aggregate

```
public static T Aggregate<T>(
  this IEnumerable<T> source,
  Func<T, T, T> func);
```

В этой версии прототипа операция Aggregate перечисляет каждый элемент входной последовательности source, вызывая делегат метода func на каждом из них, передавая значение, возвращенное им же на предыдущем элементе, и, наконец, помещая значение, возвращенное func, во внутренний аккумулятор, который затем передается на обработку следующего элемента. Первый элемент сам передается в качестве входного значения делегату метода func.

Второй прототип операции Aggregate ведет себя так же, как и первая версия, за исключением того, что принимает также начальное значение, служащее входным при первоначальном вызове делегата метода func, вместо самого первого элемента.

Второй прототип Aggregate

```
public static U Aggregate<T, U>(
  this IEnumerable<T> source,
  U seed,
  Func<U, T, U> func);
```

Исключения

Если любой из аргументов равен null, генерируется исключение ArgumentNull Exception.

Если последовательность source пуста, генерируется исключение InvalidOperation Exception, но только для первого прототипа Aggregate, когда не указано никакого начального значения.

Примеры

Начнем с примера применения первого прототипа, который приведен в листинге 5.63. В этом примере вычисляется факториал числа 5. *Факториал* — это произведение всех положительных целых чисел, меньших или равных заданному. Итак, 5! будет равно 1*2*3*4*5. Похоже, что для этого вычисления можно использовать операции Range и Aggregate.

Листинг 5.63. Пример вызова первого прототипа Aggregate

```
int N = 5;
IEnumerable<int> intSequence = Enumerable.Range(1, N);

// Вывод последовательности на консоль.
foreach (int item in intSequence)
  Console.WriteLine(item);

// Вычисление и вывод значения факториала.
// av == aggregated value, e == element
int agg = intSequence.Aggregate((av, e) => av * e);
Console.WriteLine("{0}! = {1}", N, agg);
```

В этом коде генерируется последовательность, содержащая целые числа от 1 до 5, для чего используется операция `Range`. После вывода на консоль всех элементов сгенерированной последовательности вызывается операция `Aggregate`, которой передается лямбда-выражение, умножающее переданное агрегатное значение на сам переданный элемент. Результат выглядит следующим образом:

```
1
2
3
4
5
5! = 120
```

Внимание! Применяя эту версию операции `Aggregate`, следует проявлять осторожность, чтобы первый элемент не был обработан дважды, поскольку он передается в качестве входного для первого элемента. В предыдущем примере первый вызов лямбда-выражения `func` должен получить 1 и 1. Поскольку два этих значения только перемножаются, и оба они равны 1, никакого вредного побочного эффекта нет. Но если бы нужно было складывать два значения, то в итоговую сумму первый элемент был бы включен дважды.

Для примера использования второго прототипа, показанного в листинге 5.64, написана собственная версия операции `Sum`.

Листинг 5.64. Пример вызова второго прототипа `Aggregate`

```
IEnumerable<int> intSequence = Enumerable.Range(1, 10);

// Вывод последовательности на консоль.
foreach (int item in intSequence)
  Console.WriteLine(item);

Console.WriteLine("--");

// Вычисление и вывод суммы.
int sum = intSequence.Aggregate(0, (s, i) => s + i);
Console.WriteLine(sum);
```

Обратите внимание, что в качестве начального значения при вызове операции `Aggregate` был передан `0`. Ниже показан результат:

```
1
2
3
4
5
6
7
8
9
10
--
55
```

Как видите, получен в точности тот же результат, что и при вызове `Sum` в листинге 5.51.

Резюме

Большая часть этой и предыдущей глав получилась немного суховатой, но здесь содержатся сущности, составляющие LINQ. Была раскрыта каждая операция, которая может понадобиться. Большая часть эффективности LINQ зависит от знания операций и понимания того, что они делают. Даже если вы не запомните все вариации каждой операции, просто знание об их существовании и о том, что они делают, может здорово помочь.

Представленный обзор LINQ to Objects и стандартных операций запросов позволил удостовериться в мощи и удобстве LINQ, когда речь идет об опросе данных любого типа, которые хранятся коллекциях, расположенных в памяти.

Предлагая около 50 операций, LINQ to Objects без сомнения позволит сделать код опроса данных более согласованным, надежным и легким в написании.

Следует еще раз подчеркнуть, что большинство стандартных операций запросов работают на коллекциях, реализующих интерфейс `IEnumerable<T>`, что исключает унаследованные коллекции C#. Если унаследованный код с `ArrayList` не поддается обработке запросами нового типа, прочтите еще раз об операциях `Cast` и `OfType`.

Теперь, когда появилось понимание LINQ to Objects и того, что LINQ вообще может делать, самое время приступить к рассмотрению LINQ для запросов и генерации XML. Эта функциональность называется LINQ to XML и ей посвящена следующая часть книги.

ЧАСТЬ III

LINQ to XML

В этой части

Глава 6. Введение в LINQ to XML

Глава 7. Интерфейс LINQ to XML API

Глава 8. Операции LINQ to XML

Глава 9. Дополнительные возможности LINQ to XML

ГЛАВА 6

Введение в LINQ to XML

Итак, вы хотите стать “героем XML”? Готовы ли вы к тому, чтобы выдержать шквал камней и стрел? В листинге 6.1 показан код, который в конечном итоге создает тривиальную иерархию XML, используя исходный API-интерфейс Microsoft объектной модели документов — XML Document Object Model (DOM), основанный на API-интерфейсе W3C DOM XML, который демонстрирует, насколько сложной в применении может оказаться данная модель.

Листинг 6.1. Простой пример XML

```
using System.Xml;

// Объявление переменных, которые будут использоваться повторно.
XmlElement xmlBookParticipant;
XmlAttribute xmlParticipantType;
XmlElement xmlFirstName;
XmlElement xmlLastName;

// Сначала необходимо построить документ XML.
XmlDocument xmlDoc = new XmlDocument();

// Создать корневой элемент и добавить его в документ.
XmlElement xmlBookParticipants = xmlDoc.CreateElement("BookParticipants");
xmlDoc.AppendChild(xmlBookParticipants);

// Создать список участников и добавить в список несколько участников подготовки книги.
xmlBookParticipant = xmlDoc.CreateElement("BookParticipant");
xmlParticipantType = xmlDoc.CreateAttribute("type");
xmlParticipantType.InnerText = "Author";
xmlBookParticipant.Attributes.Append(xmlParticipantType);
xmlFirstName = xmlDoc.CreateElement("FirstName");
xmlFirstName.InnerText = "Joe";
xmlBookParticipant.AppendChild(xmlFirstName);
xmlLastName = xmlDoc.CreateElement("LastName");
xmlLastName.InnerText = "Rattz";
xmlBookParticipant.AppendChild(xmlLastName);
xmlBookParticipants.AppendChild(xmlBookParticipant);

// Создать еще одного участника и добавить в список участников подготовки книги.
xmlBookParticipant = xmlDoc.CreateElement("BookParticipant");
xmlParticipantType = xmlDoc.CreateAttribute("type");
xmlParticipantType.InnerText = "Editor";
xmlBookParticipant.Attributes.Append(xmlParticipantType);
xmlFirstName = xmlDoc.CreateElement("FirstName");
xmlFirstName.InnerText = "Ewan";
xmlBookParticipant.AppendChild(xmlFirstName);
xmlLastName = xmlDoc.CreateElement("LastName");
xmlLastName.InnerText = "Buckingham";
```

```
xmlBookParticipant.AppendChild(xmlLastName);
xmlBookParticipants.AppendChild(xmlBookParticipant);

// Найти авторов и отобразить их имена и фамилии.
XmlNodeList authorsList =
  xmlDoc.SelectNodes("BookParticipants/BookParticipant[@type=\"Author\"]");
foreach (XmlNode node in authorsList)
{
  XmlNode firstName = node.SelectSingleNode("FirstName");
  XmlNode lastName = node.SelectSingleNode("LastName");
  Console.WriteLine("{0} {1}", firstName, lastName);
}
```

Последняя строка кода с вызовом метода `WriteLine` выделена полужирным, потому что она будет изменена. Весь этот код строит следующую иерархию XML и пытается отобразить имя каждого участника книги:

Желаемая структура XML

```
<BookParticipants>
  <BookParticipant type="Author">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
  <BookParticipant type="Editor">
    <FirstName>Ewan</FirstName>
    <LastName>Buckingham</LastName>
  </BookParticipant>
</BookParticipants>
```

Писать, разбирать и сопровождать такой код — сущий кошмар. Он чрезвычайно многословен. Просто взглянув на него, невозможно получить представление о том, как должна выглядеть результирующая структура XML. Отчасти то, что делает его таким громоздким — тот факт, что нельзя создать элемент, инициализировать его и присоединить к иерархии в одном операторе. Вместо этого каждый элемент сначала необходимо создать, затем вызовом его внутреннего метода `InnerTest` установить нужное значение и, наконец, добавить к некоторому существующему узлу в документе XML. Это должно быть сделано с каждым элементом и атрибутом. Все это порождает огромный объем кода. Вдобавок XML-документ должен сначала быть создан, потому что без этого не получится даже создать элемент. Очень часто действительный документ XML не требуется, а нужен только небольшой его фрагмент вроде приведенного выше. И, наконец, просто посмотрите, насколько много строк кода понадобилось, чтобы сгенерировать XML такого небольшого объема.

Нажмите <Ctrl+F5> и взгляните на вывод:

```
System.Xml.XmlElement System.Xml.XmlElement
```

Похоже, что в этом цикле `foreach` действительный текст узлов `FirstName` и `LastName` не получен. Модифицируем этот метод `Console.WriteLine`, чтобы получить эти данные:

```
Console.WriteLine("{0} {1}", firstName.ToString(), lastName.ToString());
```

Посмотрим, что получилось теперь:

```
System.Xml.XmlElement System.Xml.XmlElement
```

Если у вас есть какой-либо опыт использования XML, то наверняка есть, что рассказать о разочарованиях, путанице и различных схемах “промышленных стандартов”, разрабатываемых каждой компанией в каждой отрасли промышленности.

Независимо от нашего сопротивления, XML без сомнения стал стандартом обмена данными:

```
<PuttyInYourHands>True</PuttyInYourHands>
```

Введение

В Microsoft могли бы ограничиться предоставлением API-интерфейса LINQ XML, который позволял бы лишь выполнять запросы LINQ. К счастью для разработчиков XML, они пошли немного дальше. В дополнение к предоставлению поддержки XML запросов LINQ, в Microsoft восполнили многие недостатки стандартного API-интерфейса DOM XML. После нескольких лет мучений с API-интерфейсом W3C DOM XML большинству разработчиков стало ясно, что многие задачи, оказывается, не так просты, как должны были быть. Имея дело с небольшими фрагментами XML, использование W3C DOM требует создавать целый документ XML, даже когда нужно получить всего несколько элементов. Наверняка вам приходилось строить строки, выглядящие как XML, вместо применения API-интерфейса W3C DOM XML, просто потому что это было проще.

Несколько ключевых недостатков API-интерфейса W3C DOM XML были преодолены. Была создана новая модель документов. В результате появился намного более простой и элегантный метод создания деревьев XML. Непомерно раздутый код, вроде приведенного в листинге 6.1, с появлением LINQ стал достоянием прошлого. Создание полного дерева XML с помощью единственного оператора стало реальностью, благодаря *функциональному конструированию*. Функциональное конструирование — термин, используемый для описания возможности создания полной иерархии XML в единственном операторе. Уже одно это превращает LINQ to XML в практически бесценный инструмент.

Конечно, это не стало бы частью LINQ, если бы новый API-интерфейс XML не поддерживал запросы LINQ. Именно для этого было добавлено несколько специфичных для XML операций запросов, реализованных в виде расширяющих методов. Комбинация этих новых XML-специфичных операций со стандартными операциями запросов LINQ to Objects, которые обсуждались во второй части книги, создает мощное и элегантное решение для нахождения любых нужных данных в дереве XML.

LINQ не только поддерживает все это, но за счет комбинирования запроса с функциональным конструированием можно получить трансформации XML. LINQ to XML чрезвычайно гибок.

Обман W3C DOM XML API

Предположим, вы работаете со своим проектом и знаете, что какие-то конкретные данные должны быть сохранены в виде XML. Пусть разрабатывается класс общего назначения для протоколирования, который должен отслеживать все, что делает пользователь внутри приложения ASP.NET. Класс для протоколирования разработан по двум причинам. Во-первых, он разработан в качестве средства для обнаружения злоупотреблений системой, если таковые случатся. Во-вторых, и это более важно, когда веб-приложение уведомляет по электронной почте о возникновении исключения, пользователи, у которых возникло исключение, никогда не помнят, что они делали непосредственно перед этим. Они не могут вспомнить подробных действий, которые привели к ошибке.

Поэтому нужно что-то такое, что отслеживало бы все действия пользователей, по крайней мере, на стороне сервера. Любое действие пользователя, такое как запуск запроса или отправка заказа, должно рассматриваться как *событие*. В базе данных предусмотрены поля, которые фиксируют пользователя, дату, время и тип события, а также все прочие необходимые поля общего назначения. Однако недостаточно просто знать,

что они запросили, скажем, счет-фактуру; нужно также знать, каковы были параметры поиска. Если отправлен заказ, необходимо знать идентификатор партии, и сколько единиц товара заказано. По сути, нужны все данные, которые позволят выполнить точно ту же операцию, чтобы воспроизвести условия, повлекшие за собой исключения. Каждый тип события имеет свои данные параметров. Заводить отдельную таблицу для каждого типа событий неэффективно, и код просмотра событий не должен обращаться к излишне большому числу таблиц, чтобы воспроизвести действия пользователя. Необходима одна таблица, способная вместить всю необходимую информацию, чтобы при ее просмотре можно было увидеть каждое действие (событие), выполненное пользователем. Таким образом, все, что нужно — это строка данных XML, хранящая в базе информацию о параметрах событий.

Может не существовать схемы, определяющей то, как выглядит этот XML, потому что это зависит от нужд конкретного события. Если событием является запрос счетов-фактур для некоторого диапазона дат, оно может выглядеть так:

```
<StartDate>10/2/2006</StartDate>
<EndDate>10/9/2006</EndDate>
<IncludePaid>False</IncludePaid>
```

Если же это событие отправки заказа, оно может выглядеть иначе:

```
<PartId>4754611903</PartId>
<Quantity>12</Quantity>
<DistributionCenter>Atlanta<DistributionCenter>
<ShippingCode>USPS First Class<ShippingCode>
```

Итак, зафиксированы поля, которые понадобятся для того, чтобы воспроизвести событие. Поскольку данные варьируются в зависимости от типа события, это исключает проверку достоверности XML, так что уже есть преимущество перед использованием XML DOM API.

Этот инструмент отслеживания событий стал первоклассным средством поддержки, значительно упростившим идентификацию и разрешение ошибок. В качестве побочного эффекта было довольно забавно звонить пользователю на следующий день и говорить, что ошибка, которую он получил, пытаясь получить счет-фактуру номер 3847329 накануне, уже исправлена. Паранойя, которую вселяло в пользователя знание того, что всегда в точности известно, что он делал, часто сама по себе оправдывала применение следящего кода.

Те, кто уже знаком с XML, могут посмотреть на эти схемы и сказать: “А ведь они неправильно сформированы! Здесь нет корневого узла”. Это правда, и представляет собой проблему, если используется W3C DOM API. Однако для производства XML применяется не W3C DOM API, а другой API-интерфейс XML. Возможно, вы также будете им пользоваться. Он называется `String.Format` XML API, и его применение выглядит примерно так:

```
string xmlData =
  string.Format(
    "<StartDate>{0}</StartDate><EndDate>{1}</EndDate><IncPaid>{2}</IncPaid>",
    Date.ToShortDateString(),
    endDate.ToShortDateString(),
    includePaid.ToString());
```

Да, это плохой способ создания XML-данных. К тому же он подвержен ошибкам. Очень легко допустить опечатку или изменить регистр (`EndDate` вместо `endDate`, например), некорректно закрыв дескриптор. Был также создан метод, принимающий список параметров — имен элементов и их данных. Таким образом, код выглядит примерно так:

```
string xmlData =
  XMLHelper(
    "StartDate", startDate.ToShortDateString(),
    "EndDate", endDate.ToShortDateString(),
    "IncPaid", includePaid.ToString());
```

Метод `XMLHelper` также создает корневой узел. Но это не намного лучше. В этом вызове нет ничего для кодирования данных.

Хотя применение метода `String.Format` или любого другого приема, отличного от XML DOM API — плохая замена DOM, все же существующий API-интерфейс часто привносит с собой слишком много проблем, когда приходится иметь дело с небольшим фрагментом XML.

Если вы думаете, что такой подход к созданию XML в чем-то уникален, следует отметить, что иногда строки XML строятся даже с использованием обычной конкатенации.

Резюме

Первое впечатление, которое возникает у большинства разработчиков в отношении LINQ — это нечто, предназначенное для выполнения запросов данных. Более того, при этом свойственно исключать из области внимания другие источники информации, помимо баз данных. LINQ to XML напоминает, что LINQ предназначен также и для обработки XML, а не только для выдачи запросов к XML.

В этой главе демонстрировались неудобства обращения с XML, свойственные W3C DOM XML API, а также некоторые традиционные способы избежать этих неудобств. В следующей главе рассматривается API-интерфейс LINQ to XML. Будет показано, как с помощью этого API-интерфейса создавать иерархии XML, написав лишь небольшой объем кода. Например, в следующей главе будет создана та же иерархия XML, что и в листинге 6.1, посредством LINQ to XML, для чего вместо 29 понадобится всего 10 строк кода.

Благодаря следующим двум главам, вы согласитесь с тем, что LINQ произвел такую же революцию в манипуляциях XML, как и в запросах баз данных.

ГЛАВА 7

Интерфейс LINQ to XML API

В предыдущей главе демонстрировалось создание документа XML с использованием API-интерфейса W3C DOM XML, и можно было увидеть, насколько он может быть неуклюжим. Также были показаны некоторые приемы, которые призваны облегчить вызванные им неудобства.

Также вы узнали маленький секрет LINQ, который состоит в том, что LINQ касается не только запросов данных, но также и XML. Появился новый API-интерфейс XML — LINQ to XML.

Итак, существует лучший или, по крайней мере, более простой способ конструировать, обходить, манипулировать и опрашивать XML, который называется LINQ to XML. В этой главе будет показано, как конструировать, обходить, манипулировать и опрашивать документы XML, используя LINQ to XML, а также каким образом выполнять поиск в объекте XML.

Для примеров этой главы создается консольное приложение. Однако прежде чем можно будет пользоваться этим новым API-интерфейсом, потребуется добавить в проект ссылку на сборку `System.Xml.Linq`, если это еще не сделано.

Необходимые пространства имен

В примерах этой главы используются пространства имен `System.Linq`, `System.Xml.Linq` и `System.Collections.Generic`. Поэтому в код необходимо добавить директивы `using` для следующих пространств имен, если это не было сделано раньше:

```
using System.Linq;
using System.Xml.Linq;
using System.Collections.Generic;
```

Загрузив сопровождающий код, вы увидите, что в дополнение к этим пространствам имен также добавлена директива `using` для пространства имен `System.Diagnostics`. Это не является необходимым для примеров этой главы. Это понадобится только для сопровождающего кода — из-за некоторого добавленного рабочего кода.

Значимые проектные усовершенствования API-интерфейса

После нескольких лет существования с W3C XML DOM API от Microsoft специалистами этой корпорации было выделено несколько ключевых областей, в которых проявля-

ется неудобство, сложность или слабость исходного API-интерфейса. Чтобы побороть эти проблемы, были выделены следующие точки приложения усилий:

- конструирование деревьев XML;
- центральная роль документа;
- пространства имен и префиксы;
- извлечение значений узлов.

Каждая из этих проблемных областей представляла собой препятствие в работе с XML. Они не только вызывали "разбухание" кода работы с XML, часто непреднамеренно затеняя этот код. Их необходимо было преодолеть, чтобы обеспечить гладкую работу LINQ с XML. Например, когда нужно использовать проекцию для возврата XML из запроса LINQ, проблемой была невозможность создания экземпляра элемента операцией `new`. Такое ограничение существующего API-интерфейса XML должно было быть преодолено, чтобы обеспечить практическую работу LINQ с XML. Давайте рассмотрим каждую из перечисленных проблемных областей, а также способ их разрешения в новом API-интерфейсе LINQ to XML.

Конструирование деревьев XML было упрощено с помощью функционального конструирования

При чтении кода первого примера в предыдущей главе (листинг 6.1) сразу становится ясно, что, просто глядя на этот код создания дерева XML, очень сложно определить схему XML. После создания документа XML понадобится создать некоторого типа узел XML, такой как элемент, установить его значение и добавить к родительскому элементу. Однако каждый из этих трех шагов должен быть выполнен отдельно с использованием W3C DOM API. Это приводит к неясной схеме и объемному коду. В этом API-интерфейсе не предусмотрена поддержка создания элемента или любого типа узла в определенном месте дерева XML по отношению к его родителю и его инициализация за одну операцию.

API-интерфейс LINQ to XML не только предоставляет ту же возможность создания дерева XML, что и W3C DOM API, но также предлагает новую технику, называемую *функциональным конструированием*, для создания дерева XML. Функциональное конструирование позволяет схеме диктовать то, как конструируются объекты XML и инициализируются их значения, и все это — одновременно, в единственном операторе. API-интерфейс достигает этого за счет предоставления конструкторов новых XML-объектов, которые принимают в качестве параметров как отдельные объекты, так и их множества, указывая их значения. Тип добавляемого объекта или объектов определяет то, где именно в схеме они располагаются. Общий шаблон выглядит следующим образом:

```
XMLOBJECT o =
 new XMLOBJECT(OBJECTNAME,
            XMLOBJECT1,
            XMLOBJECT2,
            ...
            XMLOBJECTN);
```

На заметку! Приведенный фрагмент является просто псевдокодом, предназначенным для иллюстрации шаблона. Ни один из классов, присутствующих в нем, не существует на самом деле; они лишь представляют некоторый концептуальный абстрактный класс XML.

При добавлении к элементу, реализованному классом `XElement`, атрибута XML, который реализован классом `XAttribute` из LINQ to XML, этот атрибут становится атрибутом

данного элемента. Например, если в предыдущем псевдокоде XMLOBJECT1 добавляется к вновь созданному XMLOBJECT по имени o, и o является XElement, а XMLOBJECT1 — XAttribute, то XMLOBJECT1 становится атрибутом XElement по имени o.

Если XElement добавляется к XElement, то добавляемый XElement становится дочерним элементом того, к которому он добавлен. Поэтому, например, если XMLOBJECT1 и o являются элементами, то XMLOBJECT1 становится дочерним элементом o.

При создании экземпляра объекта XMLOBJECT, как показано в приведенном псевдокоде, можно задавать его содержимое, указывая от 1 до N объектов XMLOBJECT. Как будет показано в разделе "Создание текста с помощью XText" далее в главе, можно даже указывать его содержимое с помощью строки, поскольку строка автоматически преобразуется в XMLOBJECT.

Все это совершенно логично и составляет суть функционального конструирования. В листинге 7.1 приведен пример.

Листинг 7.1. Использование функционального конструирования для создания схемы XML

```
XElement xBookParticipant =
  new XElement("BookParticipant",
    new XElement("FirstName", "Joe"),
    new XElement("LastName", "Rattz"));

Console.WriteLine(xBookParticipant.ToString());
```

Обратите внимание, что при конструировании элемента по имени BookParticipant в качестве его значения передаются два объекта XElement, каждый из которых становится его дочерним элементом. Также обратите внимание, что при конструировании элементов FirstName и LastName вместо указания дочерних объектов, как это было сделано при конструировании элемента BookParticipant, указаны просто текстовые значения элементов. Вот результат работы этого кода:

```
<BookParticipant>
  <FirstName>Joe</FirstName>
  <LastName>Rattz</LastName>
</BookParticipant>
```

Обратите внимание, насколько проще теперь инициализировать в коде схему XML. Также отметьте, насколько менее многословным стал код по сравнению с кодом в листинге 6.1 предыдущей главы. Код LINQ to XML полностью заменяет код из листинга 6.1, создающий дерево XML, будучи при этом существенно короче, как доказывает листинг 7.2.

Листинг 7.2. Создание того же дерева XML, что и в листинге 6.1, с помощью намного более короткого кода

```
XElement xBookParticipants =
  new XElement("BookParticipants",
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham")));

Console.WriteLine(xBookParticipants.ToString());
```

Как видите, здесь пришлось создать намного меньше кода, что впоследствии упростит сопровождение. К тому же схему намного легче понять, просто прочитав этот код. Ниже показан вывод:

```
<BookParticipants>
  <BookParticipant type="Author">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
  <BookParticipant type="Editor">
    <FirstName>Ewan</FirstName>
    <LastName>Buckingham</LastName>
  </BookParticipant>
</BookParticipants>
```

Есть еще одно преимущество нового API-интерфейса, которое наглядно проявляется в результатах этого примера. Обратите внимание, что вывод сформатирован так, что он выглядит как дерево XML. Если же вывести дерево XML, созданное в листинге 6.1, оно будет выглядеть примерно так:

```
<BookParticipants><BookParticipant type="Author"><FirstName>Joe</FirstName>...
```

Очевидно, что легче читать первый вариант. В следующей главе, когда речь пойдет о выполнении запросов, производящих вывод XML, вы убедитесь в незаменимости функционального конструирования.

Центральная роль элемента вместо документа

В исходном W3C DOM API нельзя просто создать XML-элемент — `XmlElement`; нужно было иметь XML-документ — `XmlDocument`, из которого создавать его. Попытка создать экземпляр `XmlElement` вроде следующего:

```
XmlElement xmlBookParticipant = new XmlElement("BookParticipant");
```

привела бы к ошибке компиляции:

```
'System.Xml.XmlElement.XmlElement(string, string, string, System.Xml.XmlDocument)'
is inaccessible due to its protection level
```

```
'System.Xml.XmlElement.XmlElement(string, string, string, System.Xml.XmlDocument)'
не доступен из-за своего уровня защиты
```

С помощью W3C DOM API создавать `XmlElement` можно только вызовом метода `XmlDocument` по имени `CreateElement`, как показано ниже:

```
XmlDocument xmlDoc = new XmlDocument();
XmlElement xmlBookParticipant = xmlDoc.CreateElement("BookParticipant");
```

Этот код компилируется успешно. Но часто обязательное создание документа XML причиняет неудобства, особенно когда нужно просто создать элемент XML. Новый LINQ-оснащенный XML API позволяет создавать экземпляр элемента без создания документа XML:

```
XElement xeBookParticipant = new XElement("BookParticipant");
```

XML-элементы — не единственный тип узлов, подчиняющихся этому ограничению W3C DOM. Атрибуты, комментарии, разделы CData, инструкции обработки и ссылки на сущности — все это должно было создаваться из документа XML. К счастью, LINQ to XML дает возможность непосредственно создавать каждый их этих объектов “на лету”.

Конечно, ничто не мешает создавать XML-документы с помощью нового API-интерфейса. Например, можно создать XML-документ и добавить в него элемент `BookParticipants`, а в последний — `BookParticipant`, как показано в листинге 7.3.

Листинг 7.3. Использование LINQ to XML для создания документа XML и добавления к нему некоторой структуры

```
XDocument xDocument =
  new XDocument(
    new XElement("BookParticipants",
      new XElement("BookParticipant",
        new XAttribute("type", "Author"),
        new XElement("FirstName", "Joe"),
        new XElement("LastName", "Rattz"))));

Console.WriteLine(xDocument.ToString());
```

Нажатие <Ctrl+F5> приведет к следующему результату:

```
<BookParticipants>
  <BookParticipant type="Author">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
</BookParticipants>
```

Разметка XML, произведенная предыдущим кодом, очень похожа на XML-разметку, которая была создана в листинге 6.1, за исключением того, что был добавлен один элемент `BookParticipant` вместо двух. Однако благодаря новым средствам функционального конструирования, этот код намного более читабелен. К тому же, глядя на этот код, можно легко проследить схему. Теперь, когда документы XML не обязательны, можно опустить создание документ XML и получить тот же результат, как показано в листинге 7.4.

Листинг 7.4. Пример, аналогичный предыдущему, но без документа XML

```
XElement xElement =
  new XElement("BookParticipants",
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")));

Console.WriteLine(xElement.ToString());
```

Запуск этого кода даст в точности тот же результат, что и предыдущий пример:

```
<BookParticipants>
  <BookParticipant type="Author">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
</BookParticipants>
```

В дополнение к созданию деревьев XML без документов XML можно делать большинство тех же вещей, которые требуют документов, вроде чтения XML-разметки из файла и сохранение ее в файл.

Имена, пространства имен и префиксы

Чтобы исключить путаницу, связанную с именами, пространствами имен и префиксами пространств имен, последние изъяты из API-интерфейса. С помощью LINQ to XML префиксы пространств имен разворачиваются на вводе и возвращаются в выводе. Внутри они не существуют.

Пространство имен используется в XML с целью уникальной идентификации схемы XML для некоторой части дерева XML. В качестве пространств имен XML используются URI, потому что они уже уникальны для каждой организации. В некоторых примерах кода дерево XML создается следующим образом:

```
<BookParticipants>
  <BookParticipant type="Author">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
</BookParticipants>
```

Любой код, обрабатывающий эти данные XML, будет написан в предположении, что узел `BookParticipants` может содержать множество узлов `BookParticipant`, каждый из которых имеет атрибут `type` и узлы `FirstName` и `LastName`. Но что, если коду также придется обрабатывать XML из другого источника, в котором окажется узел `BookParticipants`, но с отличающейся внутренней схемой? Пространство имен известит код о том, как должна выглядеть схема, позволяя ему соответствующим образом обработать XML. В XML каждый элемент должен иметь имя. Когда элемент создается, если его имя указано в конструкторе, оно неявно преобразуется из `string` в объект `XName`. Объект `XName` состоит из пространства имен — объекта `XNamespace` — и своего локального имени, того, которое было указано. Поэтому, например, можно создать элемент `BookParticipants` следующим образом:

```
XElement xBookParticipants = new XElement("BookParticipants");
```

При создании элемента объект `XName` получает пустое пространство имен и локальное имя `BookParticipants`. Если во время отладки этой строки кода просмотреть переменную `xBookParticipants` в окне слежения, то увидите, что ее член `Name` установлен в `{BookParticipants}`. Развернув член `Name`, вы увидите, что он содержит член `LocalName`, установленный в `BookParticipants`, и член по имени `Namespace`, который будет пустым — `{}`. В данном случае пространство имен отсутствует.

Чтобы указать пространство имен, понадобится просто создать объект `XNamespace` и предварить им локальное имя следующим образом:

```
XNamespace nameSpace = "http://www.linqdev.com";
XElement xBookParticipants = new XElement(nameSpace + "BookParticipants");
```

После этого при просмотре элемента `xBookParticipants` в окне слежения отладчика можно увидеть, что `Name` установлено в `http://www.linqdev.com/BookParticipants`.

Разворачивание члена `Name` покажет, что `LocalName` будет по-прежнему `BookParticipants`, а член `Namespace` установлен в `http://www.linqdev.com`.

Для того чтобы указать пространство имен, не обязательно использовать объект `XNamespace`. Пространство имен можно задать в виде жестко закодированного строкового литерала:

```
XElement xBookParticipants = new XElement("{http://www.linqdev.com}" +
  "BookParticipants");
```

Обратите внимание, что пространство имен заключено в фигурные скобки. Это указывает конструктору `XElement` на тот факт, что данная часть означает пространство имен. Если вновь просмотреть член `Name` объекта `BookParticipants` в окне слежения, можно увидеть, что член `Name` и встроенные в него члены `LocalName` и `Namespace` установлены идентично значениям, установленным в предыдущем примере, где для создания элемента применялся объект `XNamespace`.

Имейте в виду, что при установке пространства имен простого указания URI компании или домена организации может быть недостаточно для того, чтобы гаранти-

ровать уникальность. Это гарантирует лишь отсутствие коллизий с любой другой существующей организацией, которая соблюдает установленные соглашения названий пространств имен. Однако внутри организации могут случиться коллизии с любым другим подразделением, если в пространстве имен не будет указано ничего помимо URI организации. И здесь весьма пригодится знание организационной структуры предприятия — его подразделений, департаментов и т.д. Лучше всего, если пространство имен будет раскрывать весь путь до определенного уровня, находящегося под вашим контролем. Например, если вы работаете в LINQDev.com, и создаете схему для отдела кадров, которая будет содержать информацию о пенсионных планах, то пространство имен может выглядеть так:

```
XNamespace nameSpace = "http://www.linqdev.com/humanresources/pension";
```

В окончательном примере использования пространств имен код из листинга 7.2 модифицируется включением пространства имен, как показано в листинге 7.5.

Листинг 7.5. Модифицированная версия листинга 7.2 с указанным пространством имен

```
XNamespace nameSpace = "http://www.linqdev.com";
XElement xBookParticipants =
  new XElement("BookParticipants",
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham")));

Console.WriteLine(xBookParticipants.ToString());
```

Нажатие <Ctrl+F5> приведет к выдаче следующего результата:

```
<BookParticipants xmlns="http://www.linqdev.com">
  <BookParticipant type="Author">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
  <BookParticipant type="Editor">
    <FirstName>Ewan</FirstName>
    <LastName>Buckingham</LastName>
  </BookParticipant>
</BookParticipants>
```

Теперь любой код может прочитать это и узнать, что схема должна соответствовать той, то предоставлена LINQDev.com.

Чтобы получить контроль над префиксами пространств имен, используйте атрибут `XAttribute` для создания префикса, как показано в листинге 7.6.

Листинг 7.6. Указание префикса пространства имен

```
XNamespace nameSpace = "http://www.linqdev.com";
XElement xBookParticipants =
  new XElement(nameSpace + "BookParticipants",
    new XAttribute(XNamespace.Xmlns + "linqdev", nameSpace),
    new XElement(nameSpace + "BookParticipant"));

Console.WriteLine(xBookParticipants.ToString());
```

В приведенном коде в качестве префикса пространства имен указывается `linqdev` и с помощью объекта `XAttribute` в схему включается спецификация префикса. Вот вывод этого кода:

```
<linqdev:BookParticipants xmlns:linqdev="http://www.linqdev.com">
  <linqdev:BookParticipant />
</linqdev:BookParticipants>
```

Извлечение значения узла

Если вы ознакомились с примером в листинге 6.1 из предыдущей главы, и посмеялись над результатом, то наверняка вам приходилось сталкиваться с той же проблемой, которая помешала сразу получить результат, т.е. действительное значение узла. Если нет достаточного опыта работы с кодом XML DOM, то неизбежно возникает ошибка подобного рода, поскольку многие забывают о необходимости выполнения дополнительного шага для получения значения узла.

В LINQ to XML эта проблема очень изящно решена. Метод `ToString` элемента выводит саму строку XML, а не тип объекта, как это делается в W3C DOM API. Это очень удобно, когда необходим определенный фрагмент XML из некоторой точки дерева, к тому же имеет намного больше смысла, чем вывод типа объекта. В листинге 7.7 приведен пример.

Листинг 7.7. Вызов метода `ToString` на элементе производит дерево XML

```
XElement name = new XElement("Name", "Joe");
Console.WriteLine(name.ToString());
```

Нажатие <Ctrl+F5> выдаст следующий результат:

```
<Name>Joe</Name>
```

Уже лучше. Однако можно добиться еще лучшего результата. Разумеется, дочерние узлы включены в вывод, и поскольку метод `WriteLine` не имеет явной перегрузки, принимающей `XElement`, он вызывает метод `ToString`, показанный в листинге 7.8.

Листинг 7.8. Неявный вызов `ToString` методом `Console.WriteLine` для производства дерева XML

```
XElement name = new XElement("Person",
  new XElement("FirstName", "Joe"),
  new XElement("LastName", "Rattz"));
Console.WriteLine(name);
```

И вот результат:

```
<Person>
  <FirstName>Joe</FirstName>
  <LastName>Rattz</LastName>
</Person>
```

Что еще более важно — если привести узел к типу данных, к которому может быть преобразовано его значение, то будет выведено само значение. В листинге 7.9 показан другой пример, при этом также выводится узел, приведенный к `string`.

Листинг 7.9. Приведение элемента к типу данных его значения выводит значение

```
XElement name = new XElement("Name", "Joe");
Console.WriteLine(name);
Console.WriteLine((string)name);
```

Вот результат этого кода:

```
<Name>Joe</Name>
Joe
```

Не правда ли, гладко? Но сколько за это придется платить? Существуют операции приведения для `string`, `int`, `int?`, `uint`, `uint?`, `long`, `long?`, `ulong`, `ulong?`, `bool`, `bool?`, `float`, `float?`, `double`, `double?`, `decimal`, `decimal?`, `TimeSpan`, `TimeSpan?`, `DateTime`, `DateTime?`, `GUID` и `GUID?`.

В листинг 7.10 приведен пример нескольких других типов узлов.

Листинг 7.10. Другие типы узлов, извлеченные через приведение к типу значения узла

```
XElement count = new XElement("Count", 12);
Console.WriteLine(count);
Console.WriteLine((int)count);
XElement smoker = new XElement("Smoker", false);
Console.WriteLine(smoker);
Console.WriteLine((bool)smoker);
XElement pi = new XElement("Pi", 3.1415926535);
Console.WriteLine(pi);
Console.WriteLine((double)pi);
```

Результаты выглядят следующим образом:

```
<Count>12</Count>
12
<Smoker>false</Smoker>
False
<Pi>3.1415926535</Pi>
3.1415926535
```

Все выглядит простым и интуитивно понятным. Похоже на то, что применение LINQ to XML вместо W3C DOM API оставит в прошлом ошибки вроде той, что проявляется в листинге 6.1 из предыдущей главы.

В то время как все эти примеры упрощают получение значения элемента, все они представляют собой случаи приведения элемента к тому типу данных, который изначально имело значение этого элемента. Это не обязательно. Все, что нужно — чтобы значение элемента могло быть преобразовано к указанному типу данных. В листинге 7.11 показан пример, где начальный тип данных — `string`, но его значение получается как `bool`.

Листинг 7.11. Приведение узла к типу данных, отличающемуся от исходного типа его значения

```
XElement smoker = new XElement("Smoker", "true");
Console.WriteLine(smoker);
Console.WriteLine((bool)smoker);
```

Поскольку значение элемента указано как `"true"`, а строка `"true"` может быть успешно преобразована в `bool`, этот код работает:

```
<Smoker>true</Smoker>
True
```

К сожалению то, как именно значения преобразуются, не задано, но на самом деле для этой цели используются методы класса `System.Xml.XmlConvert`. Код в листинге 7.12 демонстрирует, как это происходит при приведении к `bool`.

Листинг 7.12. Приведение к `bool` вызывает метод `System.Xml.XmlConvert.ToBoolean`

```
try
{
  XElement smoker = new XElement("Smoker", "Tue");
  Console.WriteLine(smoker);
  Console.WriteLine((bool)smoker);
}
catch (Exception ex)
{
  Console.WriteLine(ex);
}
```

Обратите внимание, что в слове "`True`" специально допущена опечатка, чтобы инициировать исключение, которое покажет, где именно происходило приведение. Повезет ли? Нажмем <Ctrl+F5>, чтобы проверить.

```
<Smoker>Tue</Smoker>
System.FormatException: The string 'tue' is not a valid Boolean value.
at System.Xml.XmlConvert.ToBoolean(String s)
```

```
System.FormatException: Строка 'tue' не является допустимым булевским значением.
в System.Xml.XmlConvert.ToBoolean(String s)
...
```

Как видите, исключение было сгенерировано в методе `System.Xml.XmlConvert.ToBoolean`.

Объектная модель LINQ to XML

С новым API-интерфейсом LINQ to XML пришла новая объектная модель, содержащая множество новых классов, находящихся в пространстве имен `System.Xml.Linq`. Один из них — статический класс, в котором находятся расширяющие методы LINQ to XML; два класса компараторов — `XNodeDocumentOrderComparer` и `XNodeEqualityComparer`; остальные классы используются для построения деревьев XML. Эти остальные классы показаны на диаграмме, представленной на рис. 7.1.

Рис. 7.1. Объектная модель LINQ to XML

Следует отметить некоторые интересные вещи.

- Три из этих классов, `XObject`, `XContainer` и `XNode`, являются абстрактными, так что конструировать их объекты никогда не придется.

- Атрибут `XAttribute` не наследуется от узла `XNode`. Фактически, это вообще не узел, а совершенно другого типа класс, который на самом деле представляет пару "ключ-значение".
- Потоковые элементы `XStreamingElement` не имеют отношения наследования с элементом `XElement`.
- Классы `XDocument` и `XElement` — единственные классы, имеющие узлы, унаследованные от `XNode`.

Все эти классы используются для построения деревьев XML. Самое примечательное состоит в том, что применяется класс `XElement`, поскольку, как уже упоминалось, в LINQ to XML центральную роль играет элемент, в противоположность документу в W3C XML DOM.

Отложенное выполнение запросов, удаление узлов и "проблема Хэллоуина"

Этот раздел служит предупреждением, чтобы вы опасались некоторых подводных камней. Первый из них — *отложенное выполнение запросов*. Никогда не забывайте, что многие из операций LINQ откладывают выполнение запроса до тех пор, пока это не станет абсолютно необходимо, и это может вызвать потенциальные побочные эффекты.

Другая проблема — так называемая *проблема Хэллоуина* (Halloween problem), которая получила свое название потому, что впервые всплыла в дискуссии внутри небольшой группы экспертов именно в день этого праздника. К проблемам этого типа можно отнести почти любую проблему, которая проявляется при изменении тех данных в процессе итерации, которые затрагивают эту итерацию. Впервые она была обнаружена инженерами баз данных при работе над оптимизатором базы данных. Они столкнулись с ней, когда тестировали запрос, изменяющий значение столбца базы данных, который разрабатываемый оптимизатор использовал в качестве индекса. Их тестовый запрос должен был извлечь запись на основе индекса, созданного на одном из столбцов таблицы, и этот запрос должен был изменить значение этого столбца. Поскольку затронутый столбец определяет индексацию записи, эта запись затем появлялась дальше в списке записей, извлеченная тем же запросом и подвергаясь повторной обработке. Это приводило к бесконечному циклу, потому что при всяком извлечении ее из набора записей она обновлялась и перемещалась далее в наборе записей, где вновь извлекалась и обрабатывалась, и так до бесконечности.

Вы сами наверняка сталкивались с "проблемой Хэллоуина", даже не зная, что она так называется. Приходилось ли вам когда-либо работать с какой-нибудь коллекцией, выполняя итерацию по ней и удаляя элемент, что приводило к прерыванию итерации или к ее неправильному поведению? Например, возьмем крупный комплект серверных элементов управления ASP.NET. Этот комплект включает серверный элемент `DataGrid`, и нужно выборочно удалить из него записи. При итерации по записям с начала и до конца соответствующие записи удалялись, но при этом терялась текущая точка итерации. В результате оставались записи, которые должны были быть удалены, и удалялись те, которые не должны были. Проблема решается выполнением итерации в обратном порядке.

В LINQ to XML с этой проблемой, скорее всего, придется столкнуться при удалении узлов из дерева XML, хотя это может произойти в других случаях, так что следует иметь это в виду при кодировании. Рассмотрим пример, представленный в листинге 7.13.

Листинг 7.13. Намеренный вызов “проблемы Хэллоуина”

```
XDocument xDocument = new XDocument(
  new XElement("BookParticipants",
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));

IEnumerable<XElement> elements =
  xDocument.Element("BookParticipants").Elements("BookParticipant");

foreach (XElement element in elements)
{
  Console.WriteLine("Исходный элемент: {0} : значение = {1}",
    element.Name, element.Value);
}

foreach (XElement element in elements)
{
  Console.WriteLine("Удаление {0} = {1} ...", element.Name, element.Value);
    element.Remove();
}

Console.WriteLine(xDocument);
```

В этом коде сначала строится документ XML. Затем создается последовательность элементов `BookParticipant`. По этой последовательности будет выполняться перечисление с удалением элементов. Затем отображаются элементы в последовательности, так что можно удостовериться, что в действительности есть два элемента `BookParticipant`. Далее вновь выполняется перечисление последовательности с отображением сообщения об удалении элемента и собственно удаление элемента `BookParticipant`. В конце отображается результирующий документ XML.

Если “проблема Хэллоуина” не проявится, сообщение “Удаление...” должно быть выведено дважды; и когда в конце будет отображен XML-документ, должен получиться пустой элемент `BookParticipants`. Вот результат:

```
Исходный элемент: BookParticipant : значение = JoeRattz
Исходный элемент: BookParticipant : значение = EwanBuckingham
Удаление BookParticipant = JoeRattz ...
<BookParticipants>
  <BookParticipant type="Editor">
    <FirstName>Ewan</FirstName>
    <LastName>Buckingham</LastName>
  </BookParticipant>
</BookParticipants>
```

Как и ожидалось, в последовательности есть два элемента `BookParticipant`, подлежащие удалению. Видно, что первый из них — `Joe Rattz` — был удален. Однако не похоже, чтобы второй элемент был удален, и когда отображается результирующий документ XML, второй элемент `BookParticipant` по-прежнему в нем присутствует. Перечисление ведет себя некорректно; налицо “проблема Хэллоуина”. Имейте в виду, что “проблема Хэллоуина” не всегда проявляется одинаково. Иногда перечисление может прекратиться раньше, чем должно; иногда оно генерирует исключения. Поведение варьируется в зависимости от того, что именно произошло.

Так каково же решение? Решение в данном случае заключается в кэшировании элементов и выполнении перечисления по кэшу вместо обычной техники перечисления, которая полагается на внутренние указатели, повреждаемые в процессе удаления или модификации элементов. В рассматриваемом примере последовательность элементов будет кэшироваться с использованием стандартных операций запросов, специально предназначенных для кэширования, чтобы предотвратить проблемы с отложенным выполнением запроса. Будет применяться операция `ToArray`. В листинге 7.14 показан тот же код, что и раньше, за исключением того, что вызывается операция `ToArray` и производится перечисление ее результата.

Листинг 7.14. Предотвращение “проблемы Хэллоуина”

```
XDocument xDocument = new XDocument(
  new XElement("BookParticipants",
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));
IEnumerable<XElement> elements =
  xDocument.Element("BookParticipants").Elements("BookParticipant");
foreach (XElement element in elements)
{
  Console.WriteLine("Исходный элемент: {0} : значение = {1}",
    element.Name, element.Value);
}
foreach (XElement element in elements.ToArray())
{
  Console.WriteLine("Удаление {0} = {1} ...", element.Name, element.Value);
    element.Remove();
}
Console.WriteLine(xDocument);
```

Этот код идентичен предыдущему, за исключением вызова операции `ToArray` в финальном перечислении, где удаляются элементы. Ниже показан результат:

```
Исходный элемент: BookParticipant : значение = JoeRattz
Исходный элемент: BookParticipant : значение = EwanBuckingham
Removing BookParticipant = JoeRattz ...
Removing BookParticipant = EwanBuckingham ...
<BookParticipants />
```

Обратите внимание, что на этот раз выведены два сообщения, информирующих об удалении элемента `BookParticipant`. Кроме того, когда отображается XML-документ после удаления, получается пустой элемент `BookParticipants`, потому что все его дочерние элементы удалены. “Проблема Хэллоуина” решена!

Создание XML

Как уже упоминалось, функциональное конструирование, предлагаемое LINQ to XML, позволяет очень легко создавать дерево XML, особенно если сравнить это с W3C DOM API. Рассмотрим теперь создание каждого из основных классов в LINQ to XML.

Поскольку новый API-интерфейс ориентирован на элементы, и именно они будут создаваться большей частью, сначала описано создание элементов с помощью класса `XElement`. Затем будут показаны остальные классы XML в алфавитном порядке.

Создание элементов с помощью `XElement`

Для начала следует запомнить, что в этом новом API-интерфейсе класс `XElement` является одним из наиболее часто используемых. Рассмотрим создание экземпляра `XElement`.

Класс `XElement` имеет несколько конструкторов, но ниже показаны два из них:

```
XElement.XElement(XName name, object content);
XElement.XElement(XName name, params object[] content);
```

Первый конструктор представляет простейший случай, когда элемент имеет текстовое значение и не имеет дочерних узлов. Этот случай демонстрируется в листинге 7.15.

Листинг 7.15. Создание элемента с использованием первого прототипа

```
XElement firstName = new XElement("FirstName", "Joe");
Console.WriteLine((string)firstName);
```

Первый аргумент конструктора — объект `XName`. Как упоминалось ранее, объект `XName` создается неявным преобразованием входного параметра `string` в `XName`. Второй аргумент — единственный объект, представляющий содержимое элемента. В данном случае содержимое — это `string` со значением `"Joe"`. API-интерфейс “на лету” преобразует этот строковый литерал `"Joe"` в объект `XText`. Обратите внимание, что здесь для получения значения из переменной элемента `firstName` используется преимущество новых средств извлечения значения узла. То есть элемент должен быть приведен к типу его значения, которым в данном случае является `string`. Таким образом, будет извлечено значение элемента `firstName`. И вот результат:

```
Joe
```

Тип данных единственного объекта содержимого очень гибок. Это тип данных объекта содержимого, который контролирует его отношение к элементу, к которому он добавлен. В табл. 7.1 описаны все допустимые типы данных объектов содержимого, а также то, как они обрабатываются.

Помните, что несмотря на то, что значение элемента может быть сохранено как `string`, как это возможно для *любого прочего типа* (см. табл. 7.1), такого как целое число, благодаря новым средствам извлечения значения, его можно получить в исходном типе. Поэтому, например, если при создании объекта `XElement` в качестве объекта содержимого указано `int`, то приведением узла к `int` получится значение, преобразованное в `int`. До тех пор пока выполняется приведение к одному из типов, для которых предусмотрена операция приведения, и пока значение элемента может быть преобразовано в тип, к которому выполняется приведение, такая операция представляет простой способ получения значения элемента.

Второй конструктор `XElement`, показанный выше, подобен первому, за исключением того, что можно предоставить множество объектов для содержимого. Именно это делает функциональное конструирование столь мощным. Примеры использования второго конструктора, где множество объектов содержимого передается конструктору `XElement`, приведены в листингах 7.1 и 7.2.

Таблица 7.1. Таблица поведения LINQ to XML при вставке дочернего объекта в родительский

Тип данных объекта содержимого	Способ обработки
`string`	Объект `string` или строковый литерал автоматически преобразуется в объект `XText` и с этого момента обрабатывается как `XText`
`XText`	Этот объект может иметь значение либо `string`, либо `XText`. Добавляется как дочерний узел элемента, но трактуется как текстовое содержимое элемента
`XCData`	Этот объект может иметь значение либо `string`, либо `XCData`. Добавляется как дочерний узел элемента, но трактуется как CData-содержимое элемента
`XElement`	Этот объект добавляется как дочерний элемент
`XAttribute`	Этот объект добавляется как атрибут
`XProcessingInstruction`	Этот объект добавляется как дочернее содержимое
`XComment`	Этот объект добавляется как дочернее содержимое
`IEnumerable`	Этот объект перечисляется, и обработка типов составляющих его объектов выполняется рекурсивно
`null`	Этот объект игнорируется. Вы можете удивиться, зачем вообще может понадобиться передавать `null` конструктору элемента, но оказывается, это очень полезно для трансформаций XML
Любой прочий тип	Вызывается метод `ToString`, и результирующее значение трактуется как содержимое `string`

Ранее упоминалось, что функциональное конструирование очень полезно для запросов LINQ, производящих XML. В качестве примера ниже создается стандартное XML-дерево `BookParticipants`, которое уже использовалось, но вместо жесткого кодирования значений элементов в виде строковых литералов данные будут извлекаться из источника, опрашиваемого с помощью LINQ. В данном случае таким источником будет массив.

Первых делом, необходим класс, в котором будут храниться данные. Кроме того, поскольку есть типы `BookParticipants`, имеет смысл создать перечисление `enum` для разных типов:

Перечисление `enum` и класс для следующего примера

```
enum ParticipantTypes
{
  Author = 0,
  Editor
}
class BookParticipant
{
  public string FirstName;
  public string LastName;
  public ParticipantTypes ParticipantType;
}
```

Теперь построим массив типа `BookParticipant` и сгенерируем дерево XML, применяя запрос LINQ для извлечения данных из массива, как показано в листинге 7.16.

Листинг 7.16. Генерация дерева XML запросом LINQ

```
BookParticipant[] bookParticipants = new[] {
  new BookParticipant {FirstName = "Joe", LastName = "Rattz",
                   ParticipantType = ParticipantTypes.Author},
  new BookParticipant {FirstName = "Ewan", LastName = "Buckingham",
                   ParticipantType = ParticipantTypes.Editor}
};
XElement xBookParticipants =
  new XElement("BookParticipants",
             bookParticipants.Select(p =>
               new XElement("BookParticipant",
                 new XAttribute("type", p.ParticipantType),
                 new XElement("FirstName", p.FirstName),
                 new XElement("LastName", p.LastName))));
Console.WriteLine(xBookParticipants);
```

В приведенном коде создается массив объектов `BookParticipant` по имени `bookParticipants`. Затем с помощью операции `Select` из `bookParticipants` запрашиваются значения. Для каждого значения генерируется элемент `BookParticipant`, используя члены элемента массива. Так выглядит дерево XML, сгенерированное предыдущим кодом:

```
<BookParticipants>
  <BookParticipant type="Author">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
  <BookParticipant type="Editor">
    <FirstName>Ewan</FirstName>
    <LastName>Buckingham</LastName>
  </BookParticipant>
</BookParticipants>
```

Только представьте, как это пришлось бы делать в W3C XML DOM API. Впрочем, можно просто взглянуть в листинг 6.1, потому что код создает точно такое же дерево XML.

Создание атрибутов с помощью `XAttribute`

В отличие от W3C XML DOM API, атрибуты не наследуются от узлов. Атрибут, реализованный в LINQ to XML с помощью класса `XAttribute`, является парой “имя-значение”, хранящейся в коллекции объектов `XAttribute`, которые относятся к объекту `XElement`. Используя функциональное конструирование, можно создать атрибут и “на лету” добавить его к элементу, как показано в листинге 7.17.

Листинг 7.17. Создание атрибута посредством функционального конструирования

```
XElement xBookParticipant = new XElement("BookParticipant",
                                 new XAttribute("type", "Author"));
Console.WriteLine(xBookParticipant);
```

Запуск этого кода даст следующий результат:

```
<BookParticipant type="Author" />
```

Однако иногда не удается создать атрибут одновременно с конструированием его элемента. В таком случае должен быть создан его экземпляр и затем добавлен к элементу, как показано в листинге 7.18.

Листинг 7.18. Создание атрибута и добавление его к элементу

```
XElement xBookParticipant = new XElement("BookParticipant");
XAttribute xAttribute = new XAttribute("type", "Author");
xBookParticipant.Add(xAttribute);
Console.WriteLine(xBookParticipant);
```

Результат идентичен:

```
<BookParticipant type="Author" />
```

Обратите внимание на гибкость метода `XElement.Add`. Он принимает любой объект, применяя те же правила для содержимого элемента, что и при создании экземпляра `XElement`. Замечательно!

Создание комментариев с помощью `XComment`

Создание комментариев с LINQ to XML тривиально. Комментарии XML реализуются в LINQ to XML с помощью класса `XComment`.

С использованием функционального конструирования можно создать комментарий и добавить его к элементу "на лету", как показано в листинге 7.19.

Листинг 7.19. Создание комментария посредством функционального конструирования

```
XElement xBookParticipant = new XElement("BookParticipant",
                                new XComment("This person is retired."));
Console.WriteLine(xBookParticipant);
```

Запуск этого кода даст следующий результат:

```
<BookParticipant>
  <!--This person is retired.-->
</BookParticipant>
```

Однако иногда не удается создать комментарий одновременно с конструированием его элемента. В таком случае должен быть создан его экземпляр и затем добавлен к элементу, как показано в листинге 7.20.

Листинг 7.20. Создание комментария и добавление его к элементу

```
XElement xBookParticipant = new XElement("BookParticipant");
XComment xComment = new XComment("This person is retired.");
xBookParticipant.Add(xComment);
Console.WriteLine(xBookParticipant);
```

Результат идентичен предыдущему:

```
<BookParticipant>
  <!--This person is retired.-->
</BookParticipant>
```

Создание контейнеров с помощью `XContainer`

Поскольку `XContainer` — абстрактный класс, создавать его экземпляры нельзя. Вместо этого понадобится создавать экземпляр одного из его подклассов — `XDocument` или `XElement`. Концептуально `XContainer` — это класс, унаследованный от `XNode`, который может содержать другие классы-наследники `XNode`.

Создание объявлений с помощью XDeclaration

В API-интерфейсе LINQ to XML создание объявлений также просто. Объявления XML реализованы здесь классом XDeclaration.

В отличие от большинства других классов LINQ to XML, объявления должны добавляться к XML-документу, а не к элементу. Помните, насколько гибкий конструктор у класса XElement? Любой класс, не спроектированный специально для этого, должен иметь собственный метод ToString, и выведенный им текст должен быть добавлен к элементу в виде текстового содержимого. Поэтому, используя класс XDeclaration, можно по невнимательности добавить к элементу объявление. Однако это не даст ожидаемого результата.

Внимание! В то время как объявления XML применяются к документу XML в целом и должны быть добавлены к нему, объект XElement также благополучно принимает добавляемый к нему объект XDeclaration. Тем не менее, это не дает результат, который ожидается.

С помощью функционального конструирования можно создать объявление и добавить ее к XML-документу "на лету", как показано в листинге 7.21.

Листинг 7.21. Создание объявления посредством функционального конструирования

```
XDocument xDocument = new XDocument(new XDeclaration("1.0", "UTF-8", "yes"),
                                    new XElement("BookParticipant"));
Console.WriteLine(xDocument);
```

Этот код производит следующий результат:

```
<BookParticipant />
```

Вы заметили, что в выводе нет объявления? Все правильно: метод ToString пропускает его. Однако если во время отладки кода заглянуть в документ в окне слежения, обнаружится, что объявление на месте.

Однако иногда создать объявление одновременно с конструируемым документом невозможно. В этом случае необходимо создать его экземпляр, а затем установить его в качестве значения свойства Declaration документа, как показано в листинге 7.22.

Листинг 7.22. Создание объявления и установка в свойство Declaration документа

```
XDocument xDocument = new XDocument(new XElement("BookParticipant"));
XDeclaration xDeclaration = new XDeclaration("1.0", "UTF-8", "yes");
xDocument.Declaration = xDeclaration;
Console.WriteLine(xDocument);
```

Этот код дает следующий результат:

```
<BookParticipant />
```

Опять-таки, обратите внимание, что объявление не попадает в вывод при вызове метода ToString документа. Но, как и в предыдущем примере, если просмотреть документ во время отладки кода, оно там обнаружится.

Создание типов документов с помощью XDocumentType

API-интерфейс LINQ to XML делает операцию создания типов документов совершенно безболезненной. Типы XML-документов реализованы LINQ to XML с помощью класса XDocumentType.

В отличие от большинства других классов в LINQ to XML, типы документов предназначены для добавления к XML-документам, а не к элементам. Помните, насколько гибок конструктор класса `XElement`? Любой класс, который не был специально спроектирован для обработки, должен иметь метод `ToString`, и полученный текст будет добавлен к элементу в качестве текстового содержимого. Поэтому можно нечаянно добавить тип документа `XDocumentType` к элементу, но это не даст ожидаемого результата.

Внимание! Хотя типы документов XML применимы к документу XML в целом, и должны добавляться к документу XML, объект `XElement` благополучно принимает добавляемый к нему объект `XDocumentType`. Однако это не дает результата, которого можно было ожидать.

С помощью функционального конструирования можно создавать тип документа и добавлять его к документу XML "на лету", как показано в листинге 7.23.

Листинг 7.23. Создание объявления посредством функционального конструирования

```
XDocument xDocument = new XDocument(new XDocumentType("BookParticipants",
                                          null,
                                          "BookParticipants.dtd",
                                          null),
                              new XElement("BookParticipant"));
Console.WriteLine(xDocument);
```

Этот код даст следующий результат:

```
<!DOCTYPE BookParticipants SYSTEM "BookParticipants.dtd">
<BookParticipant />
```

Однако иногда не удается создать тип документа одновременно с конструированием этого документа. Тогда понадобится сначала создать его экземпляр, а затем добавить к документу, как показано в листинге 7.24.

Листинг 7.24. Создание типа документа и добавление его к документу

```
XDocument xDocument = new XDocument();
XDocumentType documentType =
  new XDocumentType("BookParticipants", null, "BookParticipants.dtd", null);
xDocument.Add(documentType, new XElement("BookParticipants"));
Console.WriteLine(xDocument);
```

Вот результат работы этого кода:

```
<!DOCTYPE BookParticipants SYSTEM "BookParticipants.dtd">
<BookParticipants />
```

Обратите внимание, что в предыдущем коде перед добавлением типа документа не было добавлено ни одного элемента. Если добавить тип документа после добавления любого элемента, сгенерируется следующее исключение:

```
Unhandled Exception: System.InvalidOperationException: This operation would
create an incorrectly structured document.
Необработанное исключение: System.InvalidOperationException: Эта операция
создаст некорректно структурированный документ.
...
```

Поэтому, если вы собираетесь указать тип документа после создания его экземпляра, удостоверьтесь, что не указали ни одного элемента при создании экземпляра документа, используя функциональное конструирование, или не добавили ни одного элемента перед добавлением типа документа.

Создание документов с помощью XDocument

Хоть это уже повторялось много раз, но стоит это сделать еще раз: в LINQ to XML не обязательно создавать XML-документ только для того, чтобы создать дерево XML или его фрагмент. Однако при необходимости создание XML-документа с LINQ to XML столь же тривиально. Документы XML реализованы в LINQ to XML в виде класса XDocument. В листинге 7.25 приведен пример.

Листинг 7.25. Простой пример создания XML-документа с помощью XDocument

```
XDocument xDocument = new XDocument();
Console.WriteLine(xDocument);
```

Этот код не производит никакого вывода, потому что созданный XML-документ пуст. Приведенный пример может быть, чересчур тривиален, поэтому ниже создается документ с помощью всех классов LINQ to XML, которые специально предназначены для добавления к объекту XDocument (листинг 7.26).

Листинг 7.26. Несколько более сложный пример создания XML-документа с помощью XDocument

```
XDocument xDocument = new XDocument(
  new XDeclaration("1.0", "UTF-8", "yes"),
  new XDocumentType("BookParticipants", null, "BookParticipants.dtd", null),
  new XProcessingInstruction("BookCataloger", "out-of-print"),
  new XElement("BookParticipants"));
Console.WriteLine(xDocument);
```

Инструкция обработки и элемент могут добавляться также и к элементам, но нужно было получить XML-документ с некоторым содержимым. Также добавлена инструкция обработки, так что ее можно будет увидеть в действии.

Результат этого кода выглядит так:

```
<!DOCTYPE BookParticipants SYSTEM "BookParticipants.dtd">
<?BookCataloger out-of-print?>
<BookParticipants />
```

Возможно, вы заметили отсутствие объявления. Как упоминалось в связи с примером создания объявлений, метод ToString документа пропускает объявление при выводе. Однако если во время отладки кода заглянуть внутрь документа, можно увидеть, что объявление на месте.

Создание имен с помощью XName

Как упоминалось ранее в главе, в LINQ to XML нет необходимости непосредственно создавать имена с объектами XName. Фактически класс XName не имеет общедоступных конструкторов, так что нет способа создавать его экземпляры. Объект XName может быть создан из строки с необязательным пространством имен автоматически при создании объекта XName.

Объект XName состоит из LocalName — строки — и пространства имен, которое представлено в XNamespace.

Листинг 7.27 содержит код примера вызова конструктора XElement, принимающего XName в качестве аргумента.

Листинг 7.27. Пример кода, где автоматически создается объект XName

```
XElement xBookParticipant = new XElement("BookParticipant");
Console.WriteLine(xBookParticipant);
```

В приведенном примере создается экземпляр объекта XElement с передачей ему имени элемента в виде строки, так что объект XName создается с LocalName равным BookParticipant и присваивается свойству Name объекта XElement. В данном случае пространство имен не указано, поэтому объект XName его не имеет.

Нажав <Ctrl+F5>, получим следующий результат:

```
<BookParticipant />
```

В коде, показанном в листинге 7.28, пространство имен указывается.

Листинг 7.28. Пример кода, где автоматически создается объект XName и указано пространство имен

```
XNamespace ns = "http://www.linqdev.com/Books";
XElement xBookParticipant = new XElement(ns + "BookParticipant");
Console.WriteLine(xBookParticipant);
```

Этот код выдаст следующий XML:

```
<BookParticipant xmlns="http://www.linqdev.com/Books" />
```

За дополнительной информацией о создании имен с помощью LINQ to XML обращайтесь в раздел “Имена, пространства имен и префиксы” ранее в этой главе.

Создание пространств имен с помощью XNamespace

В LINQ to XML пространства имен реализованы классом XNamespace. Пример создания и использования пространства имен приведен выше в листинге 7.28. В нем демонстрируется создание пространства имен с помощью класса XNamespace.

За дополнительной информацией о создании пространств имен с помощью LINQ to XML обращайтесь в раздел “Имена, пространства имен и префиксы” ранее в этой главе.

Создание узлов с помощью XNode

Поскольку XNode — абстрактный класс, создавать его экземпляры нельзя. Вместо этого должны создаваться экземпляры одного из его подклассов: XComment, XContainer, XDocumentType, XProcessingInstruction или XText. Концептуально XNode — это любой класс, который функционирует как узел в дереве XML.

Создание инструкций обработки с помощью XProcessingInstruction

Инструкции обработки никогда ранее не было так легко создавать, как в API-интерфейсе LINQ to XML. Здесь они реализуются классом XProcessingInstruction.

Инструкции обработки можно создавать на уровне документа или элемента. В листинге 7.29 приведен пример их создания “на лету” в обоих случаях с помощью функционального конструирования.

Листинг 7.29. Создание инструкции обработки на уровне документа и элемента

```
XDocument xDocument = new XDocument(
  new XProcessingInstruction("BookCataloger", "out-of-print"),
  new XElement("BookParticipants",
    new XElement("BookParticipant",
      new XProcessingInstruction("ParticipantDeleter", "delete"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz"))));
Console.WriteLine(xDocument);
```

В этом коде добавлены инструкции обработки как к документу, так и к элементу `BookParticipant`. Прежде чем отобразить результат, обратите внимание на то, насколько хорошо здесь проходит функциональное конструирование. Очень легко создать дерево XML с двумя инструкциями обработки. По сравнению с этим самый первый пример из предыдущей главы (листинг 6.1) опять доказывает, насколько LINQ to XML упрощает код. И, наконец, вот результат:

```
<?BookCataloger out-of-print?>
<BookParticipants>
  <BookParticipant>
    <?ParticipantDeleter delete?>
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
</BookParticipants>
```

Теперь вы уже должны представлять себе код добавления инструкции обработки после конструирования, поскольку это очень похоже на добавление любого другого узла, описанного ранее. Чтобы не утомлять рутиной, в листинге 7.30 приведен существенно более сложный пример создания и добавления инструкции обработки постфактум.

Листинг 7.30. Более сложный пример добавления инструкций обработки после конструирования документа и элемента

```
XDocument xDocument =
  new XDocument(new XElement("BookParticipants",
                        new XElement("BookParticipant",
                                   new XElement("FirstName", "Joe"),
                                   new XElement("LastName", "Rattz"))));
XProcessingInstruction xPI1 = new XProcessingInstruction("BookCataloger",
                                                 "out-of-print");
xDocument.AddFirst(xPI1);
XProcessingInstruction xPI2 = new XProcessingInstruction("ParticipantDeleter",
                                                 "delete");
XElement outOfPrintParticipant = xDocument
  .Element("BookParticipants")
  .Elements("BookParticipant")
  .Where(e => ((string)((XElement)e).Element("FirstName")) == "Joe"
         && ((string)((XElement)e).Element("LastName")) == "Rattz")
  .Single<XElement>();
outOfPrintParticipant.AddFirst(xPI2);
Console.WriteLine(xDocument);
```

Относительно кода этого примера следует сделать несколько замечаний. Документ и дерево XML создано с использованием функционального конструирования. После кон-

струирования документа и дерева в документ добавлена инструкция обработки. Однако здесь применялся метод XElement.AddFirst для создания первого дочернего узла документа, в противоположность методу XElement.Add, который просто добавил бы его в конец списка дочерних узлов документа, что для любой инструкции обработки было бы слишком поздно.

Для добавления инструкции обработки к одному из элементов нужна была ссылка на него. Можно было бы просто сконструировать объект XElement и сохранить ссылку на него, но имеет смысл начать давать подсказки относительно будущих возможностей запросов. Как видите, здесь выполняется достаточно сложный запрос, где из документа получается элемент BookParticipants с использованием метода Element, который будет описан в разделе "Обход XML" далее в главе. Затем получается последовательность объектов XElement по имени BookParticipant, где элемент FirstName элемента BookParticipant равен "Joe", а LastName — "Rattz". Обратите внимание на применение новых средств извлечения узлов из LINQ to XML для получения значения узла FirstName и LastName за счет их приведения к типу string.

И, наконец, операция Where возвращает IEnumerable<T>, а нужен непосредственно объект XElement. В описании отложенных стандартных операций запросов LINQ to Object в главе 5 присутствует одна операция, которая возвращает элемент из последовательности, исходя из предположения, что он в ней единственный — операция Single. Получив от этого запроса ссылку на правильный объект XElement, совсем легко добавить к нему инструкцию обработки и отобразить результаты, которые показаны ниже:

```
<?BookCataloger out-of-print?>
<BookParticipants>
  <BookParticipant>
    <?ParticipantDeleter delete?>
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
</BookParticipants>
```

Создание потоковых элементов с помощью XStreamingElement

Во второй части книги объяснялось, что многие из стандартных операций запросов в действительности откладывают свою работу до момента, пока не начнется перечисление возвращаемых ими данных. Если вызывается операция, которая фактически откладывает свое выполнение, и нужно спроектировать вывод запроса в виде XML, возникает дилемма. С одной стороны, есть желание воспользоваться преимуществом отложенной природы операции и выполнить работу, только когда в ней возникнет необходимость. Но с другой стороны, вызов API-интерфейса LINQ to XML заставит запрос выполняться немедленно.

Обратите внимание в листинге 7.31, что даже несмотря на то, что четвертый элемент массива names был изменен при выводе значения объекта XElement, дерево XML содержит его исходное значение. Это связано с тем, что элемент xNames был полностью создан перед изменением элемента массива names.

Листинг 7.31. Немедленное выполнение конструирования дерева XML

```
string[] names = { "John", "Paul", "George", "Pete" };
XElement xNames = new XElement("Beatles",
                        from n in names
                        select new XElement("Name", n));
names[3] = "Ringo";
Console.WriteLine(xNames);
```

Перед обсуждением результатов этого кода обратите внимание, насколько он замечателен. В коде создается элемент по имени `Beatles` с содержимым, состоящим из последовательности объектов `XElement`, элементы которых названы `Name`. Этот код производит следующее дерево XML:

```
<Beatles>
  <Name>John</Name>
  <Name>Paul</Name>
  <Name>George</Name>
  <Name>Pete</Name>
</Beatles>
```

Впечатляюще. Каждый объект `XElement` из последовательности становится дочерним элементом. Чем это хорошо? Как уже упоминалось, несмотря на изменение `names[3]` в `"Ringo"` перед выводом XML, последний элемент по-прежнему содержит имя `Pete` — исходное значение. Причина в том, что последовательность `names` должна быть полностью перечислена, чтобы сконструировать объект `XElement`, а это требует немедленного выполнения запроса.

Если действительно нужно, чтобы конструирование дерева XML было отложено, необходим какой-то другой способ, и именно для этого предназначены потоковые (streaming) элементы. В LINQ to XML потоковые элементы реализованы классом `XStreamingElement`.

В листинге 7.32 показан тот же пример, но на этот раз вместо `XElement` используются объекты `XStreamingElement`.

Листинг 7.32. Демонстрация отложенного выполнения конструирования дерева XML за счет использования класса `XStreamingElement`

```
string[] names = { "John", "Paul", "George", "Pete" };
XStreamingElement xNames =
  new XStreamingElement("Beatles",
                    from n in names
                    select new XStreamingElement("Name", n));
names[3] = "Ringo";
Console.WriteLine(xNames);
```

Если это работает, как было описано выше, то теперь значение `Name` последнего узла должно быть `Ringo`, а не `Pete`. И вот доказательство:

```
<Beatles>
  <Name>John</Name>
  <Name>Paul</Name>
  <Name>George</Name>
  <Name>Ringo</Name>
</Beatles>
```

Создание текста с помощью `XText`

Создание элемента с текстовым значением — чрезвычайно простая задача. В листинге 7.33 приведен необходимый код.

Листинг 7.33. Создание элемента и присваивание строки в качестве его значения

```
XElement xFirstName = new XElement("FirstName", "Joe");
Console.WriteLine(xFirstName);
```

Все совершенно очевидно, и нет никаких сюрпризов. Запустив этот код нажатием <Ctrl+F5>, получим следующий результат:

```
<FirstName>Joe</FirstName>
```

Однако здесь скрыт тот факт, что строка "Joe" преобразуется в объект XText, и именно этот объект добавляется к объекту XElement. Фактически просмотр объекта xFirstName в отладчике покажет, что он содержит единственный узел — объект XText со значением "Joe". Поскольку все это делается автоматически, в большинстве случаев непосредственное конструирование текстового объекта не понадобится.

Тем не менее, если такая необходимость возникнет, можно создать текстовый объект, создав экземпляр объекта XText, как показано в листинге 7.34.

Листинг 7.34. Создание текстового узла и передача его в качестве значения созданному элементу

```
XText xName = new XText("Joe");
XElement xFirstName = new XElement("FirstName", xName);
Console.WriteLine(xFirstName);
```

Этот код производит в точности такой же результат, что и предыдущий пример, и если просмотреть внутреннее состояние объекта xFirstName, оно также будет идентично тому, что и в предыдущем примере:

```
<FirstName>Joe</FirstName>
```

Создание CData с помощью XCData

Создание элемента со значением CData также очень просто. В листинге 7.35 приведен пример.

Листинг 7.35. Создание узла XCData и передача его в качестве значения созданного элемента

```
XElement xErrorMessage = new XElement("HTMLMessage",
                      new XCData("<H1>Invalid user id or password.</H1>"));
Console.WriteLine(xErrorMessage);
```

Этот код дает следующий вывод:

```
<HTMLMessage><![CDATA[<H1>Invalid user id or password.</H1>]]></HTMLMessage>
```

Как видите, LINQ to XML API облегчает работу с CData.

Вывод XML

Естественно, создание, модификация и удаление данных XML ничего не стоит, если нет возможности сохранять изменения. В этом разделе будет описано несколько способов вывода XML-разметки.

Сохранение с помощью XDocument.Save()

Сохранять XML-документ можно с использованием любого из нескольких методов XDocument.Save. Вот список его прототипов:

```
void XDocument.Save(string filename);
void XDocument.Save(TextWriter textWriter);
void XDocument.Save(XmlWriter writer);
void XDocument.Save(string filename, SaveOptions options);
void XDocument.Save(TextWriter textWriter, SaveOptions options);
```

В листинге 7.36 приведен пример сохранения XML-документа в файле, находящемся в папке проекта.

Листинг 7.36. Сохранение документа методом `XDocument.Save`

```
XDocument xDocument = new XDocument(
  new XElement("BookParticipants",
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XAttribute("experience", "first-time"),
      new XAttribute("language", "English"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz"))));
xDocument.Save("bookparticipants.xml");
```

Обратите внимание, что метод `Save` вызывается на *объекте* типа `XDocument`. Причина в том, что методы `Save` являются методами экземпляра. Методы `Load`, которые рассматриваются ниже в разделе “Ввод XML”, являются статическими и должны вызываться на *классе* `XDocument` или `XElement`.

Ниже приведено содержимое сгенерированного файла `bookparticipants.xml` при просмотре его в простом текстовом редакторе:

```
<?xml version="1.0" encoding="utf-8"?>
<BookParticipants>
  <BookParticipant type="Author" experience="first-time" language="English">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
</BookParticipants>
```

Этот вывод XML-документа легко читается, поскольку вызванная версия метода `Save` форматирует вывод. То есть, если вызвать версию метода `Save`, принимающую строку имени файла и аргумент `SaveOptions`, передав в нем значение `SaveOptions.None`, то получится тот же результат, что и раньше. Но если вызвать метод `Save` следующим образом:

```
xDocument.Save("bookparticipants.xml", SaveOptions.DisableFormatting);
```

то результат в файле будет выглядеть так:

```
<?xml version="1.0" encoding="utf-8"?><BookParticipants><BookParticipant type=
"Author" experience="first-time" language="English"><FirstName>Joe</FirstName>
<LastName>Rattz</LastName></BookParticipant></BookParticipants>
```

Это одна непрерывная строка текста. Результат необходимо просматривать в текстовом редакторе, потому что браузер сформатирует его.

Разумеется, для вывода документа можно использовать любой другой доступный метод — выбор исключительно за вами.

Сохранение с помощью `XElement.Save()`

Уже много раз говорилось, что в LINQ to XML создавать XML-документ не обязательно. Также не обязательно это делать для сохранения XML-файла. Класс `XElement` также имеет несколько методов `Save`, предназначенных для этой цели:

```
void XElement.Save(string filename);
void XElement.Save(TextWriter textWriter);
void XElement.Save(XmlWriter writer);
void XElement.Save(string filename, SaveOptions options);
void XElement.Save(TextWriter textWriter, SaveOptions options);
```

В листинге 7.37 приведен пример, очень похожий на предыдущий, за исключением того, что XML-документ в нем не создается.

Листинг 7.37. Сохранение элемента методом `XElement.Save`

```
XElement bookParticipants =
  new XElement("BookParticipants",
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XAttribute("experience", "first-time"),
      new XAttribute("language", "English"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")));

bookParticipants.Save("bookparticipants.xml");
```

Сохраненный XML выглядит идентично предыдущему примеру, где в действительности существовал XML-документ:

```
<?xml version="1.0" encoding="utf-8"?>
<BookParticipants>
  <BookParticipant type="Author" experience="first-time" language="English">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
</BookParticipants>
```

Ввод XML

Создание и сохранение XML-разметки в файле не имело бы смысла, если бы не было способа загрузить его обратно в дерево XML. Ниже описаны некоторые примеры чтения XML.

Загрузка с помощью `XDocument.Load()`

Зная, как сохранять документы и фрагменты XML, наверняка интересно посмотреть, как их загружать обратно. Это делается с использованием любого из ряда доступных методов. Вот их список:

```
static XDocument XDocument.Load(string uri);
static XDocument XDocument.Load(TextReader textReader);
static XDocument XDocument.Load(XmlReader reader);
static XDocument XDocument.Load(string uri, LoadOptions options);
static XDocument XDocument.Load(TextReader textReader, LoadOptions options);
static XDocument XDocument.Load(XmlReader reader, LoadOptions options);
```

Легко заметить, что эти методы симметричны методам `XDocument.Save`. Однако есть пара отличий, на которые следует обратить внимание. Во-первых, метод `Save` должен вызываться на *объектах* типа `XDocument` или `XElement`, потому что `Save` — метод экземпляра. Однако метод `Load` — статический, поэтому он должен вызываться на самом *классе* `XDocument`. Во-вторых, методы `Save` принимают параметр типа `string`, в котором должно быть передано имя файла, в то время как методы `Load` могут принимать в параметре `string` строку URI.

Вдобавок метод `Load` допускает при загрузке документа XML указание параметра типа `LoadOptions`. Значения перечисления `LoadOptions` описаны в табл. 7.2.

Таблица 7.2. Перечисление `LoadOptions`

Опция	Описание
`LoadOptions.None`	Применяйте эту опцию, чтобы указать, что никакие опции загрузки не используются
`LoadOptions.PreserveWhitespace`	Используйте эту опцию, чтобы предохранить пробелы и пустые строки в исходном XML
`LoadOptions.SetLineInfo`	Применяйте эту опцию, чтобы иметь возможность получать строку и позицию любого объекта, унаследованного от `XObject`, посредством интерфейса `IXmlLineInfo`
`LoadOptions.SetBaseUri`	Используйте эту опцию, чтобы получать базовый URI любого объекта-наследника `XObject`

Эти опции можно комбинировать с помощью битовой операции “ИЛИ” (`|`). Однако некоторые опции не работают в ряде контекстов. Например, при создании элемента или документа передачей строки никакой информации о позициях и номерах строк недоступно, как и недоступен базовый URI. Также нет базового URI при создании документа с помощью `XmlReader`.

В листинге 7.38 показан пример загрузки XML-документа, созданного в предыдущем примере (см. листинг 7.37).

Листинг 7.38. Загрузка документа методом `XDocument.Load`

```
XDocument xDocument = XDocument.Load("bookparticipants.xml",
  LoadOptions.SetBaseUri | LoadOptions.SetLineInfo);

Console.WriteLine(xDocument);

XElement firstName = xDocument.Descendants("FirstName").First();

Console.WriteLine("FirstName Строка:{0} - Позиция:{1}",
  ((IXmlLineInfo)firstName).LineNumber,
  ((IXmlLineInfo)firstName).LinePosition);

Console.WriteLine("FirstName Базовый URI:{0}", firstName.BaseUri);
```

На заметку! Понадобится либо добавить директиву `using` для `System.Xml`, если ее еще нет, либо указать пространство имен при ссылке на интерфейс `IXmlLineInfo` в коде. В противном случае компилятор не сможет обнаружить тип `IXmlLineInfo`.

Этот код загружает файл XML, созданный в предыдущем примере. После загрузки и отображения документа получается ссылка на элемент `FirstName` и отображается строка и позиция этого элемента в исходном XML-документе. Затем отображается базовый URI элемента.

Вот результат:

```
<BookParticipants>
  <BookParticipant type="Author" experience="first-time" language="English">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
</BookParticipants>
FirstName Строка:4 - Позиция:6
FirstName Базовый URI:file:///C:/Documents and Settings/.../Projects/LINQChapter7/
LINQChapter7/bin/Debug/bookparticipants.xml
```

Этот вывод похож на ожидаемый, но с одним возможным исключением. Действительно, XML-документ выглядит хорошо. Виден номер строки и позиции элемента FirstName, но номер строки вызывает недоумение. Показано, что элемент находится в четвертой строке, но в отображенном документе на самом деле элемент FirstName расположен в третьей строке. В чем же дело? Если рассмотреть загруженный XML-документ, можно заметить, что он начинается с объявления документа, которое в выводе опущено:

```
<?xml version="1.0" encoding="utf-8"?>
```

Именно поэтому номер строки, в которой расположен элемент FirstName, отображается как четвертый.

Загрузка с помощью XElement.Load()

Точно так же, как можно сохранять XDocument и XElement, их можно и загружать. Загрузка содержимого элемента почти идентична загрузке документа. Вот доступные для этого методы:

```
static XElement XElement.Load(string uri);
static XElement XElement.LoadTextReader textReader);
static XElement XElement.Load(XmlReader reader);
static XElement XElement.Load(string uri, LoadOptions options);
static XElement XElement.Load(TextReader textReader, LoadOptions options);
static XElement XElement.Load(XmlReader reader, LoadOptions options);
```

Эти методы являются статическими, как и методы XDocument.Save, поэтому должны вызываться на классе XElement напрямую. В листинге 7.39 содержится пример загрузки XML-файла, который был сохранен с помощью метода XElement.Save в листинге 7.37.

Листинг 7.39. Загрузка элемента методом XElement.Load

```
XElement xElement = XElement.Load("bookparticipants.xml");
Console.WriteLine(xElement);
```

Как следовало ожидать, вывод выглядит следующим образом:

```
<BookParticipants>
  <BookParticipant type="Author" experience="first-time" language="English">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
</BookParticipants>
```

Как и XDocument.Load, метод XElement.Load также имеет перегрузки, принимающие параметр LoadOptions. Описание этого перечисления было дано в табл. 7.2.

Разбор содержимого с помощью методов XDocument.Parse() или XElement.Parse()

Сколько раз вам приходилось передавать в своих программах XML-содержимое в виде строки, лишь иногда нуждаясь в выполнении некоторой серьезной работы, специфичной для XML? Получение данных из переменной типа string в переменную типа XML-документа всегда связано с определенными трудностями. Отныне об этом можно не беспокоиться. В LINQ to XML доступно необходимое средство — метод Parse.

Оба класса — и XDocument, и XElement — имеют статический метод Parse для разбора строк XML. Поняв, как можно выполнять разбор строки с помощью класса XDocument, эти же знания применимы в отношении класса XElement, и наоборот. И по-

скольку весь API-интерфейс LINQ to XML построен вокруг элементов, на этот раз будет представлен только пример, ориентированный на элементы.

В разделе “Сохранение с помощью `XDocument.Save()`” ранее в этой главе был показан вывод метода `Save`, когда для его параметра `LoadOptions` задано значение `DisableFormatting`. Результат — единая строка XML. В примере из листинга 7.40 эта строка XML (после отмены внутренних кавычек) подвергается разбору в XML-элемент, который затем выводится на экран.

Листинг 7.40. Разбор строки XML в элемент

```
string xml = "<?xml version=\"1.0\" encoding=\"utf-8\"?><BookParticipants>" +
  "<BookParticipant type=\"Author\" experience=\"first-time\" language=" +
  "\"English\"><FirstName>Joe</FirstName><LastName>Rattz</LastName>" +
  "</BookParticipant></BookParticipants>";
XElement xElement = XElement.Parse(xml);
Console.WriteLine(xElement);
```

Результат выглядит следующим образом:

```
<BookParticipants>
  <BookParticipant type="Author" experience="first-time" language="English">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
</BookParticipants>
```

Здорово, не правда ли? Вспомните старый способ, когда нужно было создавать документ с использованием класса `XmlDocument` из W3C XML DOM. Благодаря исключению центральной роли документа, строку XML можно превратить в реальное дерево XML простым вызовом метода.

Обход XML

Обход XML выполняется с помощью 4 свойств и 11 методов. В данном разделе будет в основном использоваться один и тот же пример кода для каждого свойства и метода, за исключением изменения одного аргумента в единственной строке, когда это возможно. Код в листинге 7.41 строит полный XML-документ.

Листинг 7.41. Базовый пример для всех последующих примеров

```
// Это используется для хранения ссылки на один из элементов в дереве XML.
XElement firstParticipant;
XDocument xDocument = new XDocument(
  new XDeclaration("1.0", "UTF-8", "yes"),
  new XDocumentType("BookParticipants", null, "BookParticipants.dtd", null),
  new XProcessingInstruction("BookCataloger", "out-of-print"),
  // Обратите внимание, что в следующей строке сохраняется
  // ссылка на первый элемент BookParticipant.
  new XElement("BookParticipants", firstParticipant =
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));
Console.WriteLine(xDocument);
```

Для начала обратите внимание на сохранение ссылки на первый сконструированный элемент BookParticipant. Это делается для того, чтобы иметь базовый элемент, с которого можно было выполнять обход. Хотя в данном примере переменная firstParticipant не используется, она понадобится в последующих примерах. Следующее, что необходимо отметить — это аргумент метода Console.WriteLine. В данном случае выводится сам документ. В последующих примерах этот аргумент будет изменяться, демонстрируя, как обходить дерево XML. Ниже представлен вывод предыдущего примера:

```
<!DOCTYPE BookParticipants SYSTEM "BookParticipants.dtd">
<?BookCataloger out-of-print?>
<BookParticipants>
  <BookParticipant type="Author">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
  <BookParticipant type="Editor">
    <FirstName>Ewan</FirstName>
    <LastName>Buckingham</LastName>
  </BookParticipant>
</BookParticipants>
```

Свойства обхода

Рассмотрение начинается с первичных свойств обхода. Когда указаны направления (вверх, вниз и т.д.), они указываются относительно элемента, на котором вызван метод. В последующих примерах сохраняется ссылка на первый элемент BookParticipant, который и служит базовым элементом для обхода.

Обход вперед с помощью XNode.NextNode

Обход вперед по дереву XML выполняется с помощью свойства NextNode. В листинге 7.42 приведен пример.

Листинг 7.42. Обход вперед от объекта XElement через свойство NextNode

```
XElement firstParticipant;

// Полный документ со всеми мелочами.
XDocument xDocument = new XDocument(
  new XDeclaration("1.0", "UTF-8", "yes"),
  new XDocumentType("BookParticipants", null, "BookParticipants.dtd", null),
  new XProcessingInstruction("BookCataloger", "out-of-print"),

  // Обратите внимание, что в следующей строке сохраняется
  // ссылка на первый элемент BookParticipant.
  new XElement("BookParticipants", firstParticipant =
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),

    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));

Console.WriteLine(firstParticipant.NextNode);
```

Поскольку базовым служит первый элемент BookParticipant — firstParticipant, — обход вперед должен привести ко второму элементу BookParticipant. Вот результат:

```
<BookParticipant type="Editor">
  <FirstName>Ewan</FirstName>
  <LastName>Buckingham</LastName>
</BookParticipant>
```

Результат доказывает работоспособность кода. Если обратиться к свойству PreviousNode элемента, то оно вернет null, поскольку этот узел является первым в списке узлов родителя. Давайте проверим это утверждение.

Обход назад с помощью XNode.PreviousNode

Для обхода XML-дерева назад используется свойство PreviousNode. Поскольку firstParticipant не имеет предыдущего узла, предпримем небольшой трюк, сначала сделав доступ к свойству NextNode, получив второй узел BookParticipant, как это делалось в предыдущем примере, и из него получим PreviousNode. Это приведет обратно к узлу firstParticipant. Если вы когда-либо слышали выражение “шаг вперед, два шага назад”, то всего одно дополнительное обращение к свойству PreviousNode позволит сделать это. Пример представлен в листинге 7.43.

Листинг 7.43. Обход назад от объекта XElement через свойство PreviousNode

```
XElement firstParticipant;

// Полный документ со всеми мелочами.
XDocument xDocument = new XDocument(
  new XDeclaration("1.0", "UTF-8", "yes"),
  new XDocumentType("BookParticipants", null, "BookParticipants.dtd", null),
  new XProcessingInstruction("BookCataloger", "out-of-print"),

  // Обратите внимание, что в следующей строке сохраняется
  // ссылка на первый элемент BookParticipant.
  new XElement("BookParticipants", firstParticipant =
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));

Console.WriteLine(firstParticipant.NextNode.PreviousNode);
```

Если код работает, как ожидается, то должен быть получить XML-разметка первого элемента BookParticipant:

```
<BookParticipant type="Author">
  <FirstName>Joe</FirstName>
  <LastName>Rattz</LastName>
</BookParticipant>
```

LINQ to XML действительно упрощает обход дерева XML.

Вверх к документу с помощью XObject.Document

Получить XML-документ из объекта XElement можно, обратившись к свойству Document элемента. В листинге 7.44 обратите внимание на вызов метода Console.WriteLine.

Листинг 7.44. Обращение к документу XML от объекта `XElement` через свойство `Document`

```
XElement firstParticipant;

// Полный документ со всеми мелочами.
XDocument xDocument = new XDocument(
  new XDeclaration("1.0", "UTF-8", "yes"),
  new XDocumentType("BookParticipants", null, "BookParticipants.dtd", null),
  new XProcessingInstruction("BookCataloger", "out-of-print"),
  // Обратите внимание, что в следующей строке сохраняется
  // ссылка на первый элемент BookParticipant.
  new XElement("BookParticipants", firstParticipant =
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));

Console.WriteLine(firstParticipant.Document);
```

Код дает тот же документ, что и в выводе примера из листинга 7.41:

```
<!DOCTYPE BookParticipants SYSTEM "BookParticipants.dtd">
<?BookCataloger out-of-print?>
<BookParticipants>
  <BookParticipant type="Author">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
  <BookParticipant type="Editor">
    <FirstName>Ewan</FirstName>
    <LastName>Buckingham</LastName>
  </BookParticipant>
</BookParticipants>
```

Вверх с помощью `XObject.Parent`

Когда нужно подняться по дереву на один уровень выше, не удивительно, что в этом поможет свойство `Parent`. Изменение узла, переданного методу `WriteLine`, как показано в листинге 7.45, изменит вывод.

Листинг 7.45. Обход вверх от объекта `XElement` через свойство `Parent`

```
XElement firstParticipant;

// Полный документ со всеми мелочами.
XDocument xDocument = new XDocument(
  new XDeclaration("1.0", "UTF-8", "yes"),
  new XDocumentType("BookParticipants", null, "BookParticipants.dtd", null),
  new XProcessingInstruction("BookCataloger", "out-of-print"),
  // Обратите внимание, что в следующей строке сохраняется
  // ссылка на первый элемент BookParticipant.
  new XElement("BookParticipants", firstParticipant =
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
```

```
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));
Console.WriteLine(firstParticipant.Parent);
```

Вывод соответствующим образом изменится:

```
<BookParticipants>
  <BookParticipant type="Author">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
  <BookParticipant type="Editor">
    <FirstName>Ewan</FirstName>
    <LastName>Buckingham</LastName>
  </BookParticipant>
</BookParticipants>
```

Однако не давайте ввести себя в заблуждение. Это не весь документ. Здесь отсутствует описание типа документа и инструкция обработки.

Методы обхода

Чтобы продемонстрировать методы обхода, поскольку они возвращают последовательности из множества узлов, придется заменить единственный вызов метода `Console.WriteLine` циклом `foreach`, чтобы вывести потенциально многочисленные узлы. Это приведет к прежнему вызову `Console.WriteLine`, который выглядит в основном так:

```
foreach(XNode node in firstParticipant.Nodes())
{
  Console.WriteLine(node);
}
```

Единственное, что будет изменяться от примера к примеру — вызываемый метод на узле `firstParticipant` в цикле `foreach`.

Вниз с помощью XContainer.Nodes()

Спуск по дереву XML осуществляется простым вызовом метода `Nodes`, который возвращает последовательность объектов `XNode` данного объекта. Вспомните из предыдущих глав, что последовательность — это `IEnumerable<T>`, т.е. `IEnumerable` определенного типа. Пример приведен в листинге 7.46.

Листинг 7.46. Обход вниз от объекта `XElement` через свойство `Nodes`

```
XElement firstParticipant;

// Полный документ со всеми мелочами.
XDocument xDocument = new XDocument(
  new XDeclaration("1.0", "UTF-8", "yes"),
  new XDocumentType("BookParticipants", null, "BookParticipants.dtd", null),
  new XProcessingInstruction("BookCataloger", "out-of-print"),

  // Обратите внимание, что в следующей строке сохраняется
  // ссылка на первый элемент BookParticipant.
  new XElement("BookParticipants", firstParticipant =
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
```

```
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));
foreach(XNode node in firstParticipant.Nodes())
{
  Console.WriteLine(node);
}
```

Вот вывод:

```
<FirstName>Joe</FirstName>
<LastName>Rattz</LastName>
```

Не забудьте, что этот метод возвращает все дочерние узлы, а не только элементы. Поэтому любой узел из списка дочерних узлов первого участника будет включен. Сюда могут относиться комментарии (`XComment`), текст (`XText`), инструкции обработки (`XProcessingInstruction`), типы документа (`XDocumentType`) или элементы (`XElement`). Также обратите внимание, что сюда не входят атрибуты, потому что атрибут узлом не является.

В листинге 7.47 представлен лучший пример использования метода `Nodes`. Он подобен базовому примеру, но с некоторыми дополнительными узлами.

Листинг 7.47. Обход вниз от объекта `XElement` через свойство `Nodes` с дополнительными типами узлов

```
XElement firstParticipant;
// Полный документ со всеми мелочами.
XDocument xDocument = new XDocument(
  new XDeclaration("1.0", "UTF-8", "yes"),
  new XDocumentType("BookParticipants", null, "BookParticipants.dtd", null),
  new XProcessingInstruction("BookCataloger", "out-of-print"),
  // Обратите внимание, что в следующей строке сохраняется
  // ссылка на первый элемент BookParticipant.
  new XElement("BookParticipants", firstParticipant =
    new XElement("BookParticipant",
      new XComment("This is a new author."),
      new XProcessingInstruction("AuthorHandler", "new"),
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));
foreach(XNode node in firstParticipant.Nodes())
{
  Console.WriteLine(node);
}
```

Этот пример отличается от предыдущего тем, что здесь к первому элементу `BookParticipant` добавлен комментарий и инструкция обработки. Нажатие <Ctrl+F5> приведет к отображению следующего вывода:

```
<!--This is a new author.-->
<?AuthorHandler new?>
<FirstName>Joe</FirstName>
<LastName>Rattz</LastName>
```

Теперь можно видеть комментарий и инструкцию обработки. Но что, если нужны только узлы определенного типа, например, только элементы? В главе 4 была описана операция `OfType`. Ее можно использовать для возврата только тех узлов, которые относятся к определенному типу, такому как `XElement`. Используя тот же базовый код, что и в листинге 7.47, давайте обеспечим возврат только элементов, просто изменив строку `foreach`, как показано в листинге 7.48.

Листинг 7.48. Использование операции `OfType` для возврата только элементов

```
XElement firstParticipant;

// Полный документ со всеми мелочами.
XDocument xDocument = new XDocument(
  new XDeclaration("1.0", "UTF-8", "yes"),
  new XDocumentType("BookParticipants", null, "BookParticipants.dtd", null),
  new XProcessingInstruction("BookCataloger", "out-of-print"),

  // Обратите внимание, что в следующей строке сохраняется
  // ссылка на первый элемент BookParticipant.
  new XElement("BookParticipants", firstParticipant =
    new XElement("BookParticipant",
      new XComment("This is a new author."),
      new XProcessingInstruction("AuthorHandler", "new"),
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));

foreach(XNode node in firstParticipant.Nodes().OfType<XElement>())
{
  Console.WriteLine(node);
}
```

Как видите, объекты `XComment` и `XProcessingInstruction` по-прежнему создаются. Но поскольку вызвана операция `OfType`, код выдает следующий результат:

```
<FirstName>Joe</FirstName>
<LastName>Rattz</LastName>
```

Видите, насколько интеллектуально работают новые средства языка C# с LINQ? Разве не здорово, что можно использовать эту стандартную операцию запроса для подобного ограничения последовательности узлов XML? Таким образом, если необходимо получить только комментарии из первого элемента `BookParticipant`, можно воспользоваться операцией `OfType`. Код показан в листинге 7.49.

Листинг 7.49. Использование операции `OfType` для возврата только комментариев

```
XElement firstParticipant;

// Полный документ со всеми мелочами.
XDocument xDocument = new XDocument(
  new XDeclaration("1.0", "UTF-8", "yes"),
  new XDocumentType("BookParticipants", null, "BookParticipants.dtd", null),
  new XProcessingInstruction("BookCataloger", "out-of-print"),

  // Обратите внимание, что в следующей строке сохраняется
  // ссылка на первый элемент BookParticipant.
  new XElement("BookParticipants", firstParticipant =
```

```
    new XElement("BookParticipant",
      new XComment("This is a new author."),
      new XProcessingInstruction("AuthorHandler", "new"),
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));
foreach(XNode node in firstParticipant.Nodes().OfType<XComment>())
{
  Console.WriteLine(node);
}
```

Вот вывод:

```
<!--This is a new author.-->
```

А можно ли с помощью операции `OfType` извлечь только атрибуты? Нет, нельзя. Это непростой вопрос. Вспомните, что в отличие от W3C XML DOM API, в LINQ to XML атрибуты не являются узлами дерева XML. Они представляют собой последовательность пар “имя-значение”, привязанных к элементу. Чтобы получить атрибуты узла `BookParticipant`, придется изменить код, как показано в листинге 7.50.

Листинг 7.50. Использование операции `OfType` для возврата только комментариев

```
XElement firstParticipant;

// Полный документ со всеми мелочами.
XDocument xDocument = new XDocument(
  new XDeclaration("1.0", "UTF-8", "yes"),
  new XDocumentType("BookParticipants", null, "BookParticipants.dtd", null),
  new XProcessingInstruction("BookCataloger", "out-of-print"),

  // Обратите внимание, что в следующей строке сохраняется
  // ссылка на первый элемент BookParticipant.
  new XElement("BookParticipants", firstParticipant =
    new XElement("BookParticipant",
      new XComment("This is a new author."),
      new XProcessingInstruction("AuthorHandler", "new"),
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));

foreach(XAttribute attr in firstParticipant.Attributes())
{
  Console.WriteLine(attr);
}
```

Обратите внимание, что пришлось изменить не только свойство или метод первого элемента `BookParticipant`, к которому производится обращение, но также тип переменной перечисления на `XAttribute`, потому что `XAttribute` не наследуется от `XNode`. Вот результат:

```
type="Author"
```

Вниз с помощью XContainer.Elements()

Поскольку API-интерфейс LINQ to XML настолько сосредоточен на элементах, и именно с ними приходится работать больше всего, был предусмотрен быстрый способ получения только элементов из всех дочерних узлов посредством метода `Elements`. Это эквивалент вызова метода `OfType<XElement>` на последовательности, возвращенной методом `Nodes`.

В листинге 7.51 приведен пример, логически эквивалентный примеру из листинга 7.48.

Листинг 7.51. Обращение к дочерним элементам с использованием метода `Elements`

```
XElement firstParticipant;

// Полный документ со всеми мелочами.
XDocument xDocument = new XDocument(
  new XDeclaration("1.0", "UTF-8", "yes"),
  new XDocumentType("BookParticipants", null, "BookParticipants.dtd", null),
  new XProcessingInstruction("BookCataloger", "out-of-print"),
  // Обратите внимание, что в следующей строке сохраняется
  // ссылка на первый элемент BookParticipant.
  new XElement("BookParticipants", firstParticipant =
    new XElement("BookParticipant",
      new XComment("This is a new author."),
      new XProcessingInstruction("AuthorHandler", "new"),
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));
foreach(XNode node in firstParticipant.Elements())
{
  Console.WriteLine(node);
}
```

Этот код выдает в точности тот же результат, что и код в листинге 7.48:

```
<FirstName>Joe</FirstName>
<LastName>Rattz</LastName>
```

Метод `Elements` также имеет перегруженную версию, которая позволяет передавать имя искомого элемента, как показано в листинге 7.52.

Листинг 7.52. Обращение к именованным дочерним элементам с использованием метода `Elements`

```
XElement firstParticipant;

// Полный документ со всеми мелочами.
XDocument xDocument = new XDocument(
  new XDeclaration("1.0", "UTF-8", "yes"),
  new XDocumentType("BookParticipants", null, "BookParticipants.dtd", null),
  new XProcessingInstruction("BookCataloger", "out-of-print"),
  // Обратите внимание, что в следующей строке сохраняется
  // ссылка на первый элемент BookParticipant.
  new XElement("BookParticipants", firstParticipant =
    new XElement("BookParticipant",
```

```
      new XComment("This is a new author."),
      new XProcessingInstruction("AuthorHandler", "new"),
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));
foreach(XNode node in firstParticipant.Elements("FirstName"))
{
  Console.WriteLine(node);
}
```

Этот код выдаст следующий результат:

```
<FirstName>Joe</FirstName>
```

Вниз с помощью `XContainer.Element()`

Метод `Element` позволяет получить дочерний элемент, соответствующий указанному имени. Вместо возврата последовательности, требующей затем применения цикла `foreach`, возвращается единственный элемент, как показано в листинге 7.53.

Листинг 7.53. Обращение к первому дочернему элементы с указанным именем

```
XElement firstParticipant;
// Полный документ со всеми мелочами.
XDocument xDocument = new XDocument(
  new XDeclaration("1.0", "UTF-8", "yes"),
  new XDocumentType("BookParticipants", null, "BookParticipants.dtd", null),
  new XProcessingInstruction("BookCataloger", "out-of-print"),
  // Обратите внимание, что в следующей строке сохраняется
  // ссылка на первый элемент BookParticipant.
  new XElement("BookParticipants", firstParticipant =
    new XElement("BookParticipant",
      new XComment("This is a new author."),
      new XProcessingInstruction("AuthorHandler", "new"),
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));
Console.WriteLine(firstParticipant.Element("FirstName"));
```

Этот код дает следующий вывод:

```
<FirstName>Joe</FirstName>
```

Рекурсивно вверх с помощью `XNode.Ancestors()`

В то время как единственный родительский элемент можно получить из свойства узла `Parent`, для извлечения всей последовательности элементов-предков необходимо воспользоваться методом `Ancestors`. Его отличие в том, что он рекурсивно обходит вверх дерево XML, не ограничиваясь одним уровнем выше, и возвращает только элементы, а не все узлы.

Для большей наглядности добавляется ряд дочерних узлов к элементу FirstName первого участника. Кроме того, вместо перечисления предков первого элемента BookParticipant используется метод Element для спуска двумя уровнями ниже к вновь добавленному элементу NickName. Это предоставит больше предков для демонстрационных целей. Необходимый код показан в листинге 7.54.

Листинг 7.54. Обход вверх от объекта XElement методом Ancestors

```
XElement firstParticipant;

// Полный документ со всеми мелочами.
XDocument xDocument = new XDocument(
  new XDeclaration("1.0", "UTF-8", "yes"),
  new XDocumentType("BookParticipants", null, "BookParticipants.dtd", null),
  new XProcessingInstruction("BookCataloger", "out-of-print"),

  // Обратите внимание, что в следующей строке сохраняется
  // ссылка на первый элемент BookParticipant.
  new XElement("BookParticipants", firstParticipant =
    new XElement("BookParticipant",
      new XComment("This is a new author."),
      new XProcessingInstruction("AuthorHandler", "new"),
      new XAttribute("type", "Author"),
      new XElement("FirstName",
        new XText("Joe"),
        new XElement("NickName", "Joey")),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));

foreach (XElement element in firstParticipant.
  Element("FirstName").Element("NickName").Ancestors())
{
  Console.WriteLine(element.Name);
}
```

Обратите внимание на добавление некоторых дочерних узлов к элементу FirstName первого участника. Это приведет к тому, что элемент FirstName первого участника получит содержимое, включающее объект XText, эквивалентный строке "Joe", и дочерний элемент NickName. Элемент NickName элемента FirstName первого участника извлекается, чтобы получить всех его предков. Вдобавок, для перечисления последовательности, возвращенной методом Ancestors, используется переменная типа XElement, а не XNode. Это позволит добраться до свойства Name элемента. Вместо отображения XML элемента, как это делалось в предыдущих примерах, отображается только имя каждого элемента в последовательности предков. Так сделано во избежание путаницы при отображении XML каждого предка, потому что каждый из них включал бы предыдущий, что затруднило бы восприятие результата. С учетом всего сказанного, вот что получается после запуска примера:

```
FirstName
BookParticipant
BookParticipants
```

Как и ожидалось, код рекурсивно поднимается по дереву XML.

Рекурсивно вверх с помощью XElement.AncestorsAndSelf()

Этот метод работает точно так же, как и Ancestors, но с тем отличием, что в возвращенную последовательность предков он включает сам элемент, на котором вызван. В листинге 7.55 представлен тот же пример, что и предыдущий, но с вызовом метода AncestorsAndSelf.

Листинг 7.55. Обход вверх от объекта XElement методом AncestorsAndSelf

```
XElement firstParticipant;

// Полный документ со всеми мелочами.
XDocument xDocument = new XDocument(
  new XDeclaration("1.0", "UTF-8", "yes"),
  new XDocumentType("BookParticipants", null, "BookParticipants.dtd", null),
  new XProcessingInstruction("BookCataloger", "out-of-print"),

  // Обратите внимание, что в следующей строке сохраняется
  // ссылка на первый элемент BookParticipant.
  new XElement("BookParticipants", firstParticipant =
    new XElement("BookParticipant",
      new XComment("This is a new author."),
      new XProcessingInstruction("AuthorHandler", "new"),
      new XAttribute("type", "Author"),
      new XElement("FirstName",
        new XText("Joe"),
        new XElement("NickName", "Joey")),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));

foreach (XElement element in firstParticipant.
  Element("FirstName").Element("NickName"). AncestorsAndSelf())
{
  Console.WriteLine(element.Name);
}
```

Результат похож на предыдущий, но также включает имя элемента NickName в начале:

```
NickName
FirstName
BookParticipant
BookParticipants
```

Рекурсивно вниз с помощью XContainer.Descendants()

В дополнение к рекурсивному обходу вверх можно также рекурсивно обходить вниз с помощью метода Descendants. Этот метод возвращает только элементы. Существует эквивалентный метод DescendantNodes для возврата всех узлов-потомков. В листинге 7.56 представлен тот же код, что и предыдущий, но с вызовом метода Descendants на элементе первого участника.

Листинг 7.56. Обход вниз от объекта XElement методом Descendants

```
XElement firstParticipant;

// Полный документ со всеми мелочами.
XDocument xDocument = new XDocument(
```

```
  new XDeclaration("1.0", "UTF-8", "yes"),
  new XDocumentType("BookParticipants", null, "BookParticipants.dtd", null),
  new XProcessingInstruction("BookCataloger", "out-of-print"),
  // Обратите внимание, что в следующей строке сохраняется
  // ссылка на первый элемент BookParticipant.
  new XElement("BookParticipants", firstParticipant =
    new XElement("BookParticipant",
      new XComment("This is a new author."),
      new XProcessingInstruction("AuthorHandler", "new"),
      new XAttribute("type", "Author"),
      new XElement("FirstName",
        new XText("Joe"),
        new XElement("NickName", "Joey")),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));
foreach (XElement element in firstParticipant.Descendants())
{
  Console.WriteLine(element.Name);
}
```

Результат выглядит следующим образом:

```
FirstName
NickName
LastName
```

Как видите, пример выполняет обход до конца по каждой ветви дерева XML.

Рекурсивно вниз с помощью `XElement.DescendantsAndSelf()`

Точно так же, как метод `Ancestors` имеет вариацию `AncestorsAndSelf`, у метода `Descendants` есть свой подобный аналог. Метод `DescendantsAndSelf` работает похоже на `Descendants`, но с тем отличием, что включает в возвращенную последовательность сам элемент, на котором вызван. В листинге 7.57 представлен пример, подобный предыдущему, но с вызовом `DescendantsAndSelf` вместо `Descendants`.

Листинг 7.57. Обход вниз от объекта `XElement` методом `DescendantsAndSelf`

```
XElement firstParticipant;

// Полный документ со всеми мелочами.
XDocument xDocument = new XDocument(
  new XDeclaration("1.0", "UTF-8", "yes"),
  new XDocumentType("BookParticipants", null, "BookParticipants.dtd", null),
  new XProcessingInstruction("BookCataloger", "out-of-print"),

  // Обратите внимание, что в следующей строке сохраняется
  // ссылка на первый элемент BookParticipant.
  new XElement("BookParticipants", firstParticipant =
    new XElement("BookParticipant",
      new XComment("This is a new author."),
      new XProcessingInstruction("AuthorHandler", "new"),
      new XAttribute("type", "Author"),
      new XElement("FirstName",
        new XText("Joe"),
        new XElement("NickName", "Joey")),
      new XElement("LastName", "Rattz")),
```

```
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));
foreach (XElement element in firstParticipant.DescendantsAndSelf())
{
  Console.WriteLine(element.Name);
}
```

Будет ли вывод включать также имя элемента `firstParticipant`?

Давайте посмотрим:

```
BookParticipant
FirstName
NickName
LastName
```

Конечно, будет.

Вперед с помощью `XNode.NodesAfterSelf()`

В примере, показанном в листинге 7.58, помимо изменения вызова `foreach` добавлена пара комментариев к элементу `BookParticipants`, чтобы сделать более наглядной разницу между извлечением узлов и элементов, поскольку `XComment` — узел, а не элемент.

Листинг 7.58. Обход вперед от текущего узла с использованием метода `NodesAfterSelf`

```
XElement firstParticipant;

// Полный документ со всеми мелочами.
XDocument xDocument = new XDocument(
  new XDeclaration("1.0", "UTF-8", "yes"),
  new XDocumentType("BookParticipants", null, "BookParticipants.dtd", null),
  new XProcessingInstruction("BookCataloger", "out-of-print"),

  // Обратите внимание, что в следующей строке сохраняется
  // ссылка на первый элемент BookParticipant.
  new XElement("BookParticipants",
    new XComment("Begin Of List"), firstParticipant =
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham")),
    new XComment("End Of List")));

foreach (XNode node in firstParticipant.NodesAfterSelf())
{
  Console.WriteLine(node);
}
```

Обратите внимание на добавление двух комментариев по соседству с двумя элементами `BookParticipant`. Эта модификация конструируемого документа XML будет выполнена для примеров `NodesAfterSelf`, `ElementsAfterSelf`, `NodesBeforeSelf` и `ElementsBeforeSelf`.

Это вызовет перечисление всех соседних узлов первого узла BookParticipant. Вот результат:

```
<BookParticipant type="Editor">
  <FirstName>Ewan</FirstName>
  <LastName>Buckingham</LastName>
</BookParticipant>
<!--End Of List-->
```

Как видите, последний комментарий включен в вывод потому, что он является узлом. Не позволяйте этому выводу ввести вас в заблуждение. Метод NodesAfterSelf возвращает только два узла: элемент BookParticipant, чей атрибут type равен Editor, и комментарий End Of List. Остальные узлы — FirstName и LastName — отображаются просто потому, что вызывается метод ToString узла BookParticipant.

Имейте в виду, что этот метод возвращает узлы, а не элементы. Чтобы ограничить тип возвращаемых узлов, можно воспользоваться операцией TypeOf, которая демонстрировалась в предшествующих примерах. Но если интересуют элементы, то для этого предусмотрен метод ElementsAfterSelf.

Вперед с помощью XNode.ElementsAfterSelf()

В этом примере используются некоторые модификации документа XML, сделанные в листинге 7.58, которые касаются добавления двух комментариев.

Чтобы получить последовательность только соседних элементов после указанного узла, необходимо вызвать метод ElementsAfterSelf, как показано в листинге 7.59.

Листинг 7.59. Обход вперед от текущего узла с использованием метода ElementsAfterSelf

```
XElement firstParticipant;
// Полный документ со всеми мелочами.
XDocument xDocument = new XDocument(
  new XDeclaration("1.0", "UTF-8", "yes"),
  new XDocumentType("BookParticipants", null, "BookParticipants.dtd", null),
  new XProcessingInstruction("BookCataloger", "out-of-print"),
  // Обратите внимание, что в следующей строке сохраняется
  // ссылка на первый элемент BookParticipant.
  new XElement("BookParticipants",
    new XComment("Begin Of List"), firstParticipant =
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham")),
    new XComment("End Of List")));
foreach (XNode node in firstParticipant.ElementsAfterSelf())
{
  Console.WriteLine(node);
}
```

Этот код выдаст следующий результат:

```
<BookParticipant type="Editor">
  <FirstName>Ewan</FirstName>
  <LastName>Buckingham</LastName>
</BookParticipant>
```

Обратите внимание, что на этот раз комментарий исключен, поскольку не является элементом. Опять-таки, элементы FirstName и LastName отображаются лишь потому, что являются содержимым элемента BookParticipant, который был извлечен, и на этом элементе был вызван метод ToString.

Назад с помощью XNode.NodesBeforeSelf()

В этом примере используются те же модификации в XML-документе, что были проведены в листинге 7.58, касающиеся добавления двух комментариев.

Метод NodesBeforeSelf работает подобно NodesAfterSelf, но извлекает соседние узлы, расположенные перед текущим. Поскольку начальной ссылкой на документ является первый узел BookParticipant, получается ссылка на второй узел BookParticipant с использованием свойства NextNode первого узла BookParticipant, так что остаются узлы для возврата, как показано в листинге 7.60.

Листинг 7.60. Обход назад от текущего узла

```
XElement firstParticipant;
// Полный документ со всеми мелочами.
XDocument xDocument = new XDocument(
  new XDeclaration("1.0", "UTF-8", "yes"),
  new XDocumentType("BookParticipants", null, "BookParticipants.dtd", null),
  new XProcessingInstruction("BookCataloger", "out-of-print"),
  // Обратите внимание, что в следующей строке сохраняется
  // ссылка на первый элемент BookParticipant.
  new XElement("BookParticipants",
    new XComment("Begin Of List"), firstParticipant =
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham")),
    new XComment("End Of List")));
foreach (XNode node in firstParticipant.NextNode.NodesBeforeSelf())
{
  Console.WriteLine(node);
}
```

Сделанная модификация должна проявиться в возврате первого узла BookParticipant и первого комментария. Вот результат:

```
<!--Begin Of List-->
<BookParticipant type="Author">
  <FirstName>Joe</FirstName>
  <LastName>Rattz</LastName>
</BookParticipant>
```

Как интересно! Возврат двух узлов — комментария и первого BookParticipant — ожидался в обратном порядке. Казалось, что метод начнет с узла, на который произведена ссылка, и построит последовательность через свойство PreviousNode. Возможно, так и случилось, но затем была вызвана операция Reverse или InDocumentOrder. Операция InDocumentOrder рассматривается в следующей главе. Не позволяйте узлам FirstName и LastName запутать вас. Метод NodesBeforeSelf не возвращал их. Они присутствуют здесь только потому, что методом Console.WriteLine был вызван метод ToString на первом узле BookParticipant, поэтому они были отображены.

Назад с помощью XNode.ElementsBeforeSelf()

В этом примере используются те же модификации документа XML, что и в листинге 7.58, касающиеся добавления двух комментариев.

Подобно тому, как у метода NodesAfterSelf есть сопровождающий метод ElementsAfterSelf для возврата только элементов, так и у метода NodesBeforeSelf имеется дополняющий его метод ElementsBeforeSelf, который возвращает только соседние элементы, предшествующие ссылаемому узлу (листинг 7.61).

Листинг 7.61. Обход назад от текущего узла

```
XElement firstParticipant;

// Полный документ со всеми мелочами.
XDocument xDocument = new XDocument(
  new XDeclaration("1.0", "UTF-8", "yes"),
  new XDocumentType("BookParticipants", null, "BookParticipants.dtd", null),
  new XProcessingInstruction("BookCataloger", "out-of-print"),

  // Обратите внимание, что в следующей строке сохраняется
  // ссылка на первый элемент BookParticipant.
  new XElement("BookParticipants",
    new XComment("Begin Of List"), firstParticipant =
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham")),
    new XComment("End Of List")));

foreach (XNode node in firstParticipant.NextNode.ElementsBeforeSelf())
{
  Console.WriteLine(node);
}
```

Обратите внимание, что здесь опять получается ссылка на второй узел BookParticipant через свойство NextNode. Войдет ли комментарий в вывод?

```
<BookParticipant type="Author">
  <FirstName>Joe</FirstName>
  <LastName>Rattz</LastName>
</BookParticipant>
```

Конечно, нет, т.к. он не является элементом.

Модификация XML

Благодаря LINQ to XML, модификация данных XML теперь стала проще, чем когда-либо ранее. Оперируя всего лишь небольшим набором методов, можно выполнять любые желаемые модификации. Будь то добавление, изменение или удаление узлов или элементов — всегда найдется метод, который может выполнить работу.

Как уже неоднократно повторялось, в LINQ to XML большая часть времени приходится на работу с объектами XElement. Поэтому большинство приведенных примеров связано с обработкой элементов. Классы LINQ to XML, унаследованные от XNode, рассматриваются первыми, а за ними следует раздел, посвященный атрибутам.

Добавление узлов

Этот раздел, посвященный добавлению новых узлов к дереву XML, начинается с базового примера кода, представленного в листинге 7.62.

Листинг 7.62. Базовый пример с единственным участником подготовки книги

```
// Документ с одним участником подготовки книги.
XDocument xDocument = new XDocument(
  new XElement("BookParticipants",
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz"))));
Console.WriteLine(xDocument);
```

Этот код производит дерево XML с единственным участником подготовки книги. Вот его вывод:

```
<BookParticipants>
  <BookParticipant type="Author">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
</BookParticipants>
```

Описания различных методов добавления узлов будут начинаться с этого базового кода.

На заметку! Хотя во всех последующих примерах добавляются элементы, тот же прием работает со всеми классами LINQ to XML, унаследованными от `XNode`.

В дополнение к следующим способам добавления узлов не забудьте заглянуть в раздел “Вызов `XElement.SetElementValue()` на дочерних объектах `XElement`” далее в этой главе.

XContainer.Add() (AddLast)

Метод, который будет использоваться наиболее часто для добавления элементов в дерево XML — это `Add`. Он добавляет узел в конец списка дочерних узлов по отношению к указанному узлу. В листинге 7.63 приведен пример.

Листинг 7.63. Добавление узла в конец списка дочерних узлов методом `Add`

```
// Документ с одним участником подготовки книги.
XDocument xDocument = new XDocument(
  new XElement("BookParticipants",
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz"))));

xDocument.Element("BookParticipants").Add(
  new XElement("BookParticipant",
    new XAttribute("type", "Editor"),
    new XElement("FirstName", "Ewan"),
    new XElement("LastName", "Buckingham")));
Console.WriteLine(xDocument);
```

В приведенном коде видно, что базовый код дополнен кодом добавления элемента BookParticipant к элементу документа BookParticipants. Здесь с помощью метода Element документа получается элемент BookParticipants, после чего с использованием метода Add к списку его дочерних элементов добавляется новый элемент. Ниже показан результат:

```
<BookParticipants>
  <BookParticipant type="Author">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
  <BookParticipant type="Editor">
    <FirstName>Ewan</FirstName>
    <LastName>Buckingham</LastName>
  </BookParticipant>
</BookParticipants>
```

Метод Add добавляет вновь сконструированный элемент BookParticipant в конец списка дочерних узлов элемента BookParticipants. Как видите, метод Add столь же гибок, как и конструктор XElement, и следует тем же правилам относительно аргументов, что позволяет применять функциональное конструирование.

XContainer.AddFirst()

Для добавления узла в начало списка дочерних узлов служит метод AddFirst. Взяв за основу тот же самый код, что и раньше, за исключением вызова метода AddFirst, получим код, приведенный в листинге 7.64.

Листинг 7.64. Добавление узла в начало списка дочерних узлов методом AddFirst

```
// Документ с одним участником подготовки книги.
XDocument xDocument = new XDocument(
  new XElement("BookParticipants",
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz"))));
xDocument.Element("BookParticipants").AddFirst(
  new XElement("BookParticipant",
    new XAttribute("type", "Editor"),
    new XElement("FirstName", "Ewan"),
    new XElement("LastName", "Buckingham")));
Console.WriteLine(xDocument);
```

Как и можно было ожидать, вновь добавленный элемент BookParticipant появится в голове списка дочерних узлов элемента BookParticipants:

```
<BookParticipants>
  <BookParticipant type="Editor">
    <FirstName>Ewan</FirstName>
    <LastName>Buckingham</LastName>
  </BookParticipant>
  <BookParticipant type="Author">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
</BookParticipants>
```

Можно ли себе представить более простой способ манипуляций XML? Вряд ли.

XNode.AddBeforeSelf()

Чтобы вставить узел в определенное место внутри списка дочерних узлов, необходимо получить ссылку либо на предшествующий узел, либо на узел, непосредственно следующий за местом вставки, и вызвать метод AddBeforeSelf или AddAfterSelf.

В качестве отправной точки используется дерево XML, произведенное примером метода Add в листинге 7.63. В это дерево будет вставлен новый узел между двумя существующими элементами BookParticipant, как показано в листинге 7.65.

Листинг 7.65. Добавление узла в список дочерних узлов методом AddBeforeSelf

```
// Документ с одним участником подготовки книги.
XDocument xDocument = new XDocument(
  new XElement("BookParticipants",
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz"))));
xDocument.Element("BookParticipants").Add(
  new XElement("BookParticipant",
    new XAttribute("type", "Editor"),
    new XElement("FirstName", "Ewan"),
    new XElement("LastName", "Buckingham")));
xDocument.Element("BookParticipants").
  Elements("BookParticipant").
  Where(e => ((string)e.Element("FirstName")) == "Ewan").
  Single<XElement>().AddBeforeSelf(
    new XElement("BookParticipant",
      new XAttribute("type", "Technical Reviewer"),
      new XElement("FirstName", "Fabio"),
      new XElement("LastName", "Ferracchiati")));
Console.WriteLine(xDocument);
```

Чтобы освежить в памяти описание стандартных операций запросов, которое давалось во второй части книги, и для интеграции этих сведений с материалом, рассмотренным в настоящей главе, элемент BookParticiant, перед которым должен быть вставлен новый, ищется с использованием арсенала операций LINQ. С помощью метода Element производится углубление в документ с целью выбора элемента BookParticipants. Затем выбираются дочерние элементы BookParticipants с именем BookParticipant, у которых есть дочерний элемент по имени FirstName со значением "Ewan". Поскольку известно, что этому критерию отвечает только один элемент BookParticipant, и оттого, что необходим объект типа XElement, на котором можно вызвать метод AddBeforeSelf, вызывается операция Single для получения объекта BookParticipant типа XElement. Это дает ссылку на элемент BookParticipant, перед которым требуется вставить новый XElement.

Также обратите внимание, что в вызове операции Where выполняется приведение элемента FirstName к string для использования средства извлечения значения узла. Это позволяет получить значение элемента FirstName с целью его сравнения с "Ewan".

Имея ссылку на правильный элемент BookParticipant, можно просто вызвать метод AddBeforeSelf. Вот результат:

```
<BookParticipants>
  <BookParticipant type="Author">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
```

```
    <BookParticipant type="Technical Reviewer">
      <FirstName>Fabio</FirstName>
      <LastName>Ferracchiati</LastName>
    </BookParticipant>
    <BookParticipant type="Editor">
      <FirstName>Ewan</FirstName>
      <LastName>Buckingham</LastName>
    </BookParticipant>
  </BookParticipants>
```

Как и требовалось, новый элемент `BookParticipant` вставлен перед элементом `BookParticipant`, у которого значение элемента `FirstName` равно `"Ewan"`.

XNode.AddAfterSelf()

После того объема усилий, предпринятых для получения ссылки на второй элемент `BookParticipant` в предыдущем примере, код в листинге 7.66 может несколько разочаровать. Здесь с помощью метода `Element` просто получается ссылка на первый элемент `BookParticipant`, после чего вызовом метода `AddAfterSelf` сразу после него добавляется новый элемент `BookParticipant`.

Листинг 7.66. Добавление узла в определенное место списка дочерних узлов методом `AddAfterSelf`

```
// Документ с одним участником подготовки книги.
XDocument xDocument = new XDocument(
  new XElement("BookParticipants",
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz"))));
xDocument.Element("BookParticipants").Add(
  new XElement("BookParticipant",
    new XAttribute("type", "Editor"),
    new XElement("FirstName", "Ewan"),
    new XElement("LastName", "Buckingham")));
xDocument.Element("BookParticipants").
    Element("BookParticipant").AddAfterSelf(
      new XElement("BookParticipant",
        new XAttribute("type", "Technical Reviewer"),
        new XElement("FirstName", "Fabio"),
        new XElement("LastName", "Ferracchiati")));
Console.WriteLine(xDocument);
```

По сравнению с предыдущим этот пример покажется тривиальным:

```
  <BookParticipants>
    <BookParticipant type="Author">
      <FirstName>Joe</FirstName>
      <LastName>Rattz</LastName>
    </BookParticipant>
    <BookParticipant type="Technical Reviewer">
      <FirstName>Fabio</FirstName>
      <LastName>Ferracchiati</LastName>
    </BookParticipant>
    <BookParticipant type="Editor">
      <FirstName>Ewan</FirstName>
      <LastName>Buckingham</LastName>
    </BookParticipant>
  </BookParticipants>
```

Удаление узлов

Удаление узлов выполняется с помощью одного из двух методов: `Remove` или `RemoveAll`. В дополнение к способам удаления узлов, изложенным в последующих разделах, не забудьте заглянуть в раздел “Вызов `XElement.SetElementValue()` на дочерних объектах `XElement`” далее в этой главе.

XNode.Remove()

Метод `Remove` удаляет из дерева XML любой узел, а также все его дочерние узлы и атрибуты. В первом примере конструируется дерево XML и сохраняется ссылка на первый элемент, описывающий участника подготовки книги, как это делалось в некоторых предыдущих примерах. Далее сконструированное дерево XML отображается перед удалением узлов. Затем удаляется первый элемент, описывающий участника книги, и отображается полученное в результате дерево XML. В листинге 7.67 приведен код.

Листинг 7.67. Удаление определенного узла методом `Remove`

```
// Это используется для сохранения ссылки на один из элементов дерева XML.
XElement firstParticipant;
Console.WriteLine(System.Environment.NewLine + "Перед удалением узла");
XDocument xDocument = new XDocument(
  new XElement("BookParticipants", firstParticipant =
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));
Console.WriteLine(xDocument);
firstParticipant.Remove();
Console.WriteLine(System.Environment.NewLine + "После удаления узла");
Console.WriteLine(xDocument);
```

Если все верно, то должно быть получено дерево XML сначала с первым элементом, описывающим участника, а затем без него:

```
Перед удалением узла
<BookParticipants>
  <BookParticipant type="Author">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
  <BookParticipant type="Editor">
    <FirstName>Ewan</FirstName>
    <LastName>Buckingham</LastName>
  </BookParticipant>
</BookParticipants>

После удаления узла
<BookParticipants>
  <BookParticipant type="Editor">
    <FirstName>Ewan</FirstName>
    <LastName>Buckingham</LastName>
  </BookParticipant>
</BookParticipants>
```

Как видите, первый элемент BookParticipant исчез в результате удаления.

IEnumerable<T>.Remove()

В предыдущем случае метод Remove вызывался на единственном объекте XElement. Однако его можно также вызвать на последовательности (IEnumerable<T>). В листинге 7.68 представлен пример, в котором с помощью метода Descendants документа осуществляется рекурсивный обход вниз всего дерева XML с возвратом только элементов с именем FirstName, для чего используется операция Where. Затем на результирующей последовательности вызывается метод Remove.

Листинг 7.68. Удаление последовательности узлов методом Remove

```
XDocument xDocument = new XDocument(
  new XElement("BookParticipants", firstParticipant =
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));
xDocument.Descendants().Where(e => e.Name == "FirstName").Remove();
Console.WriteLine(xDocument);
```

Этот пример интересен тем, что здесь можно связать вместе все элементы LINQ. С помощью метода XDocument.Descendants извлекаются в виде последовательности все дочерние узлы, а затем вызывается стандартная операция запроса Where для фильтрации только узлов, отвечающим критерию поиска, которым в данном случае является имя элемента FirstName. Она возвращает последовательность, на которой затем вызывается метод Remove. Ниже показан результат:

```
<BookParticipants>
  <BookParticipant type="Author">
    <LastName>Rattz</LastName>
  </BookParticipant>
  <BookParticipant type="Editor">
    <LastName>Buckingham</LastName>
  </BookParticipant>
</BookParticipants>
```

Обратите внимание, что все узлы FirstName исчезли.

XElement.RemoveAll()

Иногда может понадобиться удалить содержимое элемента, но не сам элемент. Именно для этого предназначен метод RemoveAll. В листинге 7.69 приведен пример.

Листинг 7.69. Удаление содержимого узла методом RemoveAll

```
XDocument xDocument = new XDocument(
  new XElement("BookParticipants", firstParticipant =
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
```

```
    new XElement("FirstName", "Ewan"),
    new XElement("LastName", "Buckingham"))));
Console.WriteLine(System.Environment.NewLine + "Перед удалением содержимого.");
Console.WriteLine(xDocument);
xDocument.Element("BookParticipants").RemoveAll();
Console.WriteLine(System.Environment.NewLine + "После удаления содержимого.");
Console.WriteLine(xDocument);
```

В коде сначала отображается документ перед удалением содержимого узла BookParticipants. Затем содержимое узла BookParticipants удаляется, и документ отображается заново. Вот как выглядит результат:

```
Перед удалением содержимого.
<BookParticipants>
  <BookParticipant type="Author">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
    </BookParticipant>
  <BookParticipant type="Editor">
    <FirstName>Ewan</FirstName>
    <LastName>Buckingham</LastName>
  </BookParticipant>
</BookParticipants>

После удаления содержимого.
<BookParticipants />
```

Обновление узлов

Несколько подклассов XNode, такие как XElement, XText и XComment, имеют свойство Value, которое может быть обновлено непосредственно. Другие, вроде XDocumentType и XProcessingInstruction, имеют свои специфичные свойства, которые также могут обновляться. В дополнение к модификации свойства Value, значения элементов можно изменять с помощью методов XElement.SetElementValue и XContainer.ReplaceAll, которые описаны далее в главе.

XElement.Value на объектах XElement, XText.Value на объектах XText и XComment.Value на объектах XComment

Каждый из этих подклассов XNode имеет свойство Value, которое может быть установлено для обновления значения узла. Все они продемонстрированы в листинге 7.70.

Листинг 7.70. Обновление значения узла

```
// Это используется для сохранения ссылки на один из элементов дерева XML.
XElement firstParticipant;

XDocument xDocument = new XDocument(
  new XElement("BookParticipants", firstParticipant =
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz"))));

Console.WriteLine("Перед обновлением узлов:");
Console.WriteLine(xDocument);

// Теперь обновим элемент, комментарий и текстовый узел.
firstParticipant.Element("FirstName").Value = "Joey";
firstParticipant.Nodes().OfType<XComment>().Single().Value =
  "Author of Pro LINQ: Language Integrated Query in C# 2008.";
```

```
((XElement)firstParticipant.Element("FirstName").NextNode)
  .Nodes().OfType<XText>().Single().Value = "Rattz, Jr.";

Console.WriteLine("После обновления узлов:");
Console.WriteLine(xDocument);
```

В коде сначала обновляется элемент `FirstName` с использованием свойства `Value`, затем — комментарий с помощью его свойства `Value` и, наконец — элемент `LastName` за счет обращения к его значению через свойство `Value` его дочернего объекта `XText`. Обратите внимание на гибкость, которую обеспечивает LINQ to XML при получении ссылок на разные объекты, которые должны быть обновлены. Только имейте в виду, что вообще-то обращаться к значению элемента `LastName`, получая объект `XText` из его дочерних узлов, необязательно. Это сделано исключительно для целей демонстрации. В принципе можно было напрямую обратиться к его свойству `Value`. Ниже показан результат выполнения кода:

```
Перед обновлением узлов:
<BookParticipants>
  <BookParticipant type="Author">
    <!--This is a new author.-->
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
</BookParticipants>

После обновления узлов:
<BookParticipants>
  <BookParticipant type="Author">
    <!--Author of Pro LINQ: Language Integrated Query in C# 2008.-->
    <FirstName>Joey</FirstName>
    <LastName>Rattz, Jr.</LastName>
  </BookParticipant>
</BookParticipants>
```

Как видите, все значения узлов обновлены.

XDocumentType.Name, XDocumentType.PublicId, XDocumentType.SystemId и XDocumentType.InternalSubset на объектах XDocumentType

Для обновления узла типа документа класс `XDocumentType` предоставляет четыре свойства. Пример, демонстрирующий их использование, приведен в листинге 7.71.

Листинг 7.71. Обновление типа документа

```
// Это используется для хранения ссылки на DocumentType для дальнейшего обращения.
XDocumentType docType;
XDocument xDocument = new XDocument(
  docType = new XDocumentType("BookParticipants", null,
                        "BookParticipants.dtd", null),
  new XElement("BookParticipants"));

Console.WriteLine("Перед обновлением типа документа:");
Console.WriteLine(xDocument);

docType.Name = "MyBookParticipants";
docType.SystemId = "http://www.somewhere.com/DTDs/MyBookParticipants.DTD";
docType.PublicId = "-//DTDs//TEXT Book Participants//EN";

Console.WriteLine("После обновления типа документа:");
Console.WriteLine(xDocument);
```

Вот результат работы этого кода:

```
Перед обновлением типа документа:
<!DOCTYPE BookParticipants SYSTEM "BookParticipants.dtd">
<BookParticipants />

После обновления типа документа:
<!DOCTYPE MyBookParticipants PUBLIC "-//DTDs//TEXT Book Participants//EN"
"http://www.somewhere.com/DTDs/MyBookParticipants.DTD">
<BookParticipants />
```

XProcessingInstruction.Target на объектах XProcessingInstruction и XProcessingInstruction.Data на объектах XProcessingInstruction

Чтобы обновить значение инструкции обработки, просто модифицируйте свойства `Target` и `Data` объекта `XProcessingInstruction`. В листинге 7.72 приведен пример.

Листинг 7.72. Обновление инструкции обработки

```
// Это используется для хранения ссылки для дальнейшего обращения.
XProcessingInstruction procInst;
XDocument xDocument = new XDocument(
  new XElement("BookParticipants"),
  procInst = new XProcessingInstruction("BookCataloger", "out-of-print"));

Console.WriteLine("Перед обновлением инструкции обработки:");
Console.WriteLine(xDocument);

procInst.Target = "BookParticipantContactManager";
procInst.Data = "update";

Console.WriteLine("После обновления инструкции обработки:");
Console.WriteLine(xDocument);
```

Взглянем на вывод:

```
Перед обновлением инструкции обработки:
<BookParticipants />
<?BookCataloger out-of-print?>

После обновления инструкции обработки:
<BookParticipants />
<?BookParticipantContactManager update?>
```

XElement.ReplaceAll()

Метод `ReplaceAll` удобен для замены целого поддерева XML, начинающегося с элемента. Этому методу можно передавать простое значение, такое как новая строка или значение числового типа. Поскольку доступна перегруженная версия метода, принимающая множество объектов через ключевое слово `params`, можно также заменить целое поддерево. Метод `ReplaceAll` также заменяет атрибуты. В листинге 7.73 показан пример.

Листинг 7.73. Применение `ReplaceAll` для замены всего поддерева элемента

```
// Это используется для сохранения ссылки на один из элементов дерева XML.
XElement firstParticipant;
XDocument xDocument = new XDocument(
  new XElement("BookParticipants", firstParticipant =
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
```

```
    new XElement("FirstName", "Joe"),
    new XElement("LastName", "Rattz"))));

Console.WriteLine(System.Environment.NewLine + "Перед обновлением элементов:");
Console.WriteLine(xDocument);

firstParticipant.ReplaceAll(
  new XElement("FirstName", "Ewan"),
  new XElement("LastName", "Buckingham"));

Console.WriteLine(System.Environment.NewLine + "После обновления элементов:");
Console.WriteLine(xDocument);
```

Обратите внимание, что при замене содержимого с помощью метода `ReplaceAll` опущено указание атрибута. Как и можно было ожидать, содержимое заменено:

```
Перед обновлением элементов:
<BookParticipants>
  <BookParticipant type="Author">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
</BookParticipants>

После обновления элементов:
<BookParticipants>
  <BookParticipant>
    <FirstName>Ewan</FirstName>
    <LastName>Buckingham</LastName>
  </BookParticipant>
</BookParticipants>
```

Обратите внимание, что атрибут типа `BookParticipant` исчез. Интересно, что атрибуты не являются дочерними узлами элемента. Тем не менее, метод `ReplaceAll` заменяет их также.

Вызов `XElement.SetElementValue()` на дочерних объектах `XElement`

Не позволяйте этому методу с простым именем ввести вас в заблуждение. Он обладает способностью добавлять, изменять и удалять элементы. Более того, он выполняет эти операции на дочерних элементах того элемента, на котором вызывается. Другими словами, метод `SetElementValue` вызывается на родительском элементе, чтобы оказать влияние на его содержимое, т.е. на дочерние элементы.

При вызове методу `SetElementValue` передается имя дочернего элемента, значение которого должно быть установлено. Если дочерний элемент найден по имени, его значение обновляется, если только переданное значение не равно `null`. Когда передается `null`, найденный дочерний элемент удаляется. Если же элемент с указанным именем не обнаружен, он будет добавлен с переданным значением. Замечательный метод!

Также метод `SetElementValue` затрагивает только первый дочерний элемент с указанным именем, который он находит. Все последующие элементы с тем же именем не затрагиваются — будь то изменение значения или удаление элемента по причине передачи значения `null`. В листинге 7.74 демонстрируются все применения этого метода: обновление, добавление и удаление.

Листинг 7.74. Применение `SetElementValue` для обновления, добавления и удаления дочерних элементов

```
// Это используется для сохранения ссылки на один из элементов дерева XML.
XElement firstParticipant;
XDocument xDocument = new XDocument(
  new XElement("BookParticipants", firstParticipant =
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz"))));
Console.WriteLine(System.Environment.NewLine + "Перед обновлением элементов:");
Console.WriteLine(xDocument);
// Во-первых, XElement.SetElementValue применяется для обновления значения элемента.
// Поскольку элемент по имени FirstName существует, его значение будет изменено на Joseph.
firstParticipant.SetElementValue("FirstName", "Joseph");
// Во-вторых, XElement.SetElementValue применяется для добавления элемента.
// Поскольку элемента по имени MiddleInitial нет, он будет добавлен.
firstParticipant.SetElementValue("MiddleInitial", "C");
// В-третьих, XElement.SetElementValue применяется для удаления элемента.
// Установка значения элемента в null приводит к его удалению.
firstParticipant.SetElementValue("LastName", null);
Console.WriteLine(System.Environment.NewLine + "После обновления элементов:");
Console.WriteLine(xDocument);
```

Как видите, сначала метод `SetElementValue` вызывается на дочернем элементе `firstParticipant` по имени `FirstName`. Поскольку элемент с таким именем существует, его значение будет обновлено. Затем метод `SetElementValue` вызывается на дочернем элементе `firstParticipant` по имени `MiddleInitial`. Поскольку элемента с таким именем нет, он будет добавлен. И, наконец, метод `SetElementValue` вызывается на дочернем элементе `firstParticipant` по имени `LastName` с передачей значения `null`. Поскольку передано `null`, элемент `LastName` будет удален. Как видите метод `SetElementValue` обеспечивает высокую степень гибкости. А вот как выглядят результаты:

```
Перед обновлением элементов:
<BookParticipants>
  <BookParticipant type="Author">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
</BookParticipants>

После обновления элементов:
<BookParticipants>
  <BookParticipant type="Author">
    <FirstName>Joseph</FirstName>
    <MiddleInitial>C</MiddleInitial>
  </BookParticipant>
</BookParticipants>
```

Весьма неплохо. Значение элемента `FirstName` обновлено, добавлен элемент `MiddleInitial` и удален элемент `LastName`.

Внимание! Исходя из того, что вызов метода `SetElementValue` со значением `null` приводит к удалению узла, не следует думать, что установка вручную значения элемента в `null` — это то же самое, что и удаление его средствами LINQ to XML. Это просто поведение метода `SetElementValue`. Если вы попытаетесь установить значение элемента в `null`, используя свойство `Value`, будет сгенерировано исключение.

Атрибуты XML

Ранее уже упоминалось, что в API-интерфейсе LINQ to XML атрибуты реализованы классом XAttribute, и в отличие от W3C XML DOM API, они не унаследованы от узла. Поэтому они не имеют отношений наследования с элементами. Однако в LINQ to XML работать с ними так же просто, как и с элементами. Давайте посмотрим, как это делается.

Создание атрибута

Атрибуты создаются точно так же, как элементы и большинство других классов LINQ to XML. Эта тема рассматривалась в разделе “Создание атрибутов с помощью XAttribute” ранее в главе.

Обход атрибутов

Для обхода атрибутов используются свойства XElement.FirstAttribute, XElement.LastAttribute, XAttribute.NextAttribute и XAttribute.PreviousAttribute, а также методы XElement.Attribute и XElement.Attributes. Все они описаны в последующих разделах.

Вперед с помощью XElement.FirstAttribute

Получить доступ к атрибутам элемента можно, обратившись к его первому атрибуту через свойство элемента FirstAttribute. В листинге 7.75 показан пример.

Листинг 7.75. Обращение к первому атрибуту элемента через свойство FirstAttribute

```
// Это используется для хранения ссылки на один из элементов в дереве XML.
XElement firstParticipant;
XDocument xDocument = new XDocument(
  new XElement("BookParticipants", firstParticipant =
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XAttribute("experience", "first-time"),
      new XAttribute("language", "English"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz"))));
Console.WriteLine(firstParticipant.FirstAttribute);
```

Вот вывод этого кода:

```
type="Author"
```

Вперед с помощью XAttribute.NextAttribute

Чтобы выполнить обход вперед по атрибутам элемента, обратитесь к свойству атрибута NextAttribute. Пример представлен в листинге 7.76.

Листинг 7.76. Обращение к следующему атрибуту элемента через свойство NextAttribute

```
// Это используется для хранения ссылки на один из элементов в дереве XML.
XElement firstParticipant;
XDocument xDocument = new XDocument(
  new XElement("BookParticipants", firstParticipant =
```

```
   new XElement("BookParticipant",
     new XAttribute("type", "Author"),
     new XAttribute("experience", "first-time"),
     new XAttribute("language", "English"),
     new XElement("FirstName", "Joe"),
     new XElement("LastName", "Rattz"))));
Console.WriteLine(firstParticipant.FirstAttribute.NextAttribute);
```

Обратите внимание, что для получения ссылки на первый атрибут используется свойство `FirstAttribute`, а затем на ней применяется `NextAttribute`. Результат можно видеть ниже:

```
experience="first-time"
```

Если свойство атрибута `NextAttribute` равно `null`, значит, текущий атрибут у элемента является последним.

Назад с помощью `XAttribute.PreviousAttribute`

Чтобы выполнить обход атрибутов элемента в обратном направлении, обращайтесь к свойству атрибута `PreviousAttribute`. Пример представлен в листинге 7.77.

Листинг 7.77. Обращение к предыдущему атрибуту элемента через свойство `PreviousAttribute`

```
// Это используется для хранения ссылки на один из элементов в дереве XML.
XElement firstParticipant;
XDocument xDocument = new XDocument(
  new XElement("BookParticipants", firstParticipant =
   new XElement("BookParticipant",
     new XAttribute("type", "Author"),
     new XAttribute("experience", "first-time"),
     new XAttribute("language", "English"),
     new XElement("FirstName", "Joe"),
     new XElement("LastName", "Rattz"))));
Console.WriteLine(firstParticipant.FirstAttribute.NextAttribute.
PreviousAttribute);
```

Обратите внимание, что свойства `FirstAttribute` и `NextAttribute` выстроены в цепочку, чтобы получить ссылку на второй атрибут, от которого выполняется шаг назад. Это должно привести обратно к первому атрибуту. Вот результат:

```
type="Author"
```

Так и есть! Если свойство `PreviousAttribute` равно `null`, значит, данный атрибут является у элемента первым.

Назад с помощью `XElement.LastAttribute`

Для получения доступа к самому последнему атрибуту элемента, чтобы затем выполнить обратный обход атрибутов, используйте свойство `LastAttribute`, как показано в листинге 7.78.

Листинг 7.78. Обращение к последнему атрибуту элемента через свойство `LastAttribute`

```
// Это используется для хранения ссылки на один из элементов в дереве XML.
XElement firstParticipant;
XDocument xDocument = new XDocument(
```

```
  new XElement("BookParticipants", firstParticipant =
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XAttribute("experience", "first-time"),
      new XAttribute("language", "English"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz"))));
Console.WriteLine(firstParticipant.LastAttribute);
```

В результате должен быть выведен атрибут `language`:

```
language="English"
```

XElement.Attribute()

Этот метод принимает имя атрибута и возвращает *первый* атрибут с указанным именем, если таковой существует. В листинге 7.79 приведен пример.

Листинг 7.79. Обращение к атрибуту методом `Attribute`

```
// Это используется для хранения ссылки на один из элементов в дереве XML.
XElement firstParticipant;
XDocument xDocument = new XDocument(
  new XElement("BookParticipants", firstParticipant =
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz"))));
Console.WriteLine(firstParticipant.Attribute("type").Value);
```

Для получения ссылки на атрибут `type` здесь используется метод `Attribute`. Затем с помощью свойства `Value` отображается значение атрибута. Вывод должен быть таким:

```
Author
```

Однако вместо получения значения атрибута через свойство `Value` можно было бы просто привести атрибут к типу `string`.

XElement.Attributes()

Доступ сразу ко всем атрибутам элемента можно получить через метод `Attributes`. Этот метод возвращает последовательность объектов `XAttribute`. В листинге 7.80 показан пример.

Листинг 7.80. Обращение ко всем атрибутам элемента методом `Attributes`

```
// Это используется для хранения ссылки на один из элементов в дереве XML.
XElement firstParticipant;
XDocument xDocument = new XDocument(
  new XElement("BookParticipants", firstParticipant =
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XAttribute("experience", "first-time"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz"))));
foreach(XAttribute attr in firstParticipant.Attributes())
{
  Console.WriteLine(attr);
}
```

Вывод будет таким:

```
type="Author"
experience="first-time"
```

Модификация атрибутов

Для модификации атрибутов служат несколько методов и свойств, которые описаны ниже.

Добавление атрибутов

Как уже было указано, существует фундаментальное отличие в способах обработки атрибутов между интерфейсами W3C XML DOM API и LINQ to XML. В W3C API атрибут — это дочерний узел по отношению к узлу, атрибутом которого он является. В LINQ to XML атрибуты *не* являются дочерними узлами такого узла. Вместо этого атрибуты представляют собой пары "имя-значение", доступ к которым осуществляется через метод `Attributes` элемента либо через свойство `FirstAttribute`. Важно помнить об этом.

Тем не менее, работа с атрибутами очень похожа на работу с элементами. Методы и свойства атрибутов подобны методам и свойствам, предназначенным для работы с элементами. Для добавления атрибута к элементу используются следующие методы:

```
XElement.Add()
XElement.AddFirst()
XElement.AddBeforeThis()
XElement.AddAfterThis()
```

В примерах, представляющих каждый из этих методов в разделе "Добавление узлов" ранее в этой главе, атрибуты также добавлялись. Еще раз просмотрите эти примеры. Вдобавок загляните в раздел, посвященный методу `XElement.SetAttributeValue`, далее в этой главе.

Удаление атрибутов

Удаление атрибутов осуществляется либо методом `XAttribute.Remove`, либо `IEnumerable<T>.Remove` — в зависимости от того, нужно удалить единственный атрибут или последовательность атрибутов.

В дополнение к описанным ниже способам удаления атрибутов загляните в раздел "`XElement.SetAttributeValue()`" далее в этой главе.

```
XElement.Remove()
```

Подобно `XNode`, класс `XAttribute` также имеет метод `Remove`. Пример его вызова представлен в листинге 7.81.

Листинг 7.81. Удаление атрибута

```
// Это используется для хранения ссылки на один из элементов в дереве XML.
XElement firstParticipant;
XDocument xDocument = new XDocument(
  new XElement("BookParticipants", firstParticipant =
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz"))));
Console.WriteLine(System.Environment.NewLine + "Перед удалением атрибута:");
Console.WriteLine(xDocument);
firstParticipant.Attribute("type").Remove();
Console.WriteLine(System.Environment.NewLine + "После удаления атрибута:");
Console.WriteLine(xDocument);
```

Как видите, с помощью метода `Attribute` получается ссылка на атрибут, подлежащий удалению, а затем на ней вызывается метод `Remove`. Ниже показан результат работы этого примера:

```
Перед удалением атрибута:
<BookParticipants>
  <BookParticipant type="Author">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
</BookParticipants>

После удаления атрибута:
<BookParticipants>
  <BookParticipant>
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
</BookParticipants>
```

Обратите внимание, что атрибут `type` исчез.

`IEnumerable<T>.Remove()`

Точно так же, как метод `IEnumerable<T>.Remove` позволяет удалять последовательность узлов, с его помощью можно удалить и все атрибуты элемента, как показано в листинге 7.82.

Листинг 7.82. Удаление всех атрибутов элемента

```
// Это используется для хранения ссылки на один из элементов в дереве XML.
XElement firstParticipant;

XDocument xDocument = new XDocument(
  new XElement("BookParticipants", firstParticipant =
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XAttribute("experience", "first-time"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz"))));

Console.WriteLine(System.Environment.NewLine + "Перед удалением атрибутов:");
Console.WriteLine(xDocument);

firstParticipant.Attributes().Remove();

Console.WriteLine(System.Environment.NewLine + "После удаления атрибутов:");
Console.WriteLine(xDocument);
```

В приведенном примере вызывается метод `Attributes`, возвращающий последовательность атрибутов элемента, на котором вызван, а затем на этой возвращенной последовательности — метод `Remove`, чтобы удалить их все. Все выглядит простым и интуитивно понятным. Вот результат:

```
Перед удалением атрибутов:
<BookParticipants>
  <BookParticipant type="Author" experience="first-time">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
</BookParticipants>
```

```
После удаления атрибутов:
<BookParticipants>
  <BookParticipant>
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
</BookParticipants>
```

Как видите, атрибуты исчезли.

Обновление атрибутов

Для обновления значения атрибута используется свойство `XAttribute.Value`.

На заметку! В дополнение к описанию применения свойства `XAttribute.Value` для обновления атрибутов не забудьте заглянуть в раздел "`XElement.SetAttributeValue()`" далее в главе.

Обновление значения атрибута легко осуществляется через его свойство `Value`. Пример приведен в листинге 7.83.

Листинг 7.83. Изменение значения атрибута

```
// Это используется для хранения ссылки на один из элементов в дереве XML.
XElement firstParticipant;
XDocument xDocument = new XDocument(
  new XElement("BookParticipants", firstParticipant =
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XAttribute("experience", "first-time"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz"))));
Console.WriteLine(System.Environment.NewLine + "Перед изменением значения атрибута:");
Console.WriteLine(xDocument);
firstParticipant.Attribute("experience").Value = "beginner";
Console.WriteLine(System.Environment.NewLine + "После изменения значения атрибута:");
Console.WriteLine(xDocument);
```

Обратите внимание, что для получения ссылки на атрибут `experience` используется метод `Attribute`. Результат работы этого кода выглядит следующим образом:

```
Перед изменением значения атрибута:
<BookParticipants>
  <BookParticipant type="Author" experience="first-time">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
</BookParticipants>

После изменения значения атрибута:
<BookParticipants>
  <BookParticipant type="Author" experience="beginner">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
</BookParticipants>
```

Как видите, значение атрибута `experience` изменилось с "`first-time`" на "`beginner`".

XElement.SetAttributeValue()

Сохраняя симметрию с методами элементов, можно ожидать, что метод `SetAttributeValue` должен быть столь же мощным, что и метод `SetElementValue`. Так оно и есть. Метод `XElement.SetAttributeValue()` обладает способностью добавлять, удалять и обновлять атрибут.

Передача этому методу имени несуществующего атрибута приводит к его добавлению. Передача имени существующего атрибута со значением, отличным от `null`, вызывает обновление значения указанного атрибута. Передача имени существующего атрибута и значения `null` вызовет удаление соответствующего атрибута. Примеры всех трех случаев приведены в листинге 7.84.

Листинг 7.84. Применение `SetAttributeValue` для добавления, удаления и обновления атрибутов

```
// Это используется для хранения ссылки на один из элементов в дереве XML.
XElement firstParticipant;
XDocument xDocument = new XDocument(
  new XElement("BookParticipants", firstParticipant =
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XAttribute("experience", "first-time"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz"))));

Console.WriteLine(System.Environment.NewLine + "Перед изменением атрибутов:");
Console.WriteLine(xDocument);

// Этот вызов обновит значение атрибута type,
// потому что атрибут с таким именем существует.
firstParticipant.SetAttributeValue("type", "beginner");

// Этот вызов добавит атрибут, потому что атрибута с указанным именем не существует.
firstParticipant.SetAttributeValue("language", "English");

// Этот вызов удалит атрибут, потому что атрибут с указанным
// именем существует, а в качестве значения передано null.
firstParticipant.SetAttributeValue("experience", null);

Console.WriteLine(System.Environment.NewLine + "После изменения атрибутов:");
Console.WriteLine(xDocument);
```

Как видите, в этом примере сначала обновляется значение существующего атрибута, затем добавляется новый атрибут и, наконец, удаляется существующий атрибут за счет передачи значения `null`. Вот результат:

```
Перед изменением атрибутов:
<BookParticipants>
  <BookParticipant type="Author" experience="first-time">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
</BookParticipants>

После изменения атрибутов:
<BookParticipants>
  <BookParticipant type="beginner" language="English">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
</BookParticipants>
```

Аннотации XML

API-интерфейс LINQ to XML предоставляет возможность ассоциировать объект пользовательских данных с любым классом-наследником `XObject` через аннотации. Это позволяет разработчику приложений назначать данные любого типа элементу, документу или любому другому объекту, чей класс наследуется от `XObject`. Объект может быть дополнительными ключами для данных элемента, объектом, который разберет содержимого элемента и сохранит его в себе в ином виде, либо чем-то другим.

Добавление аннотаций с помощью `XObject.AddAnnotation()`

Добавление аннотаций осуществляется с помощью метода `XObject.AddAnnotation`. Вот его прототип:

```
void XObject.AddAnnotation(object annotation);
```

Обращение к аннотациям с помощью `XObject.Annotation()` или `XObject.Annotations()`

Обращение к аннотациям осуществляется с использованием методов `XObject.Annotation` или `XObject.Annotations`. Ниже показаны их прототипы:

```
object XObject.Annotation(Type type);
T XObject.Annotation<T>();
IEnumerable<object> XObject.Annotations(Type type);
IEnumerable<T> XObject.Annotations<T>();
```

Внимание! При извлечении аннотаций должен передаваться действительный тип объекта, а не базовый класс или интерфейс. В противном случае аннотация не будет найдена.

Удаление аннотаций с помощью `XObject.RemoveAnnotations()`

Удаление аннотаций осуществляется методом `XObject.RemoveAnnotations`. У него есть два прототипа:

```
void XObject.RemoveAnnotations(Type type);
void XObject.RemoveAnnotations<T>();
```

Пример аннотаций

Для демонстрации аннотаций создадим один пример, в котором аннотации будут добавляться, извлекаться и удаляться. В этом примере используется привычное XML-дерево `BookParticipants`. Необходим способ ассоциирования обработчика каждого `BookParticipant` на основе его атрибута `type`. В данном примере обработчик будет просто отображать элемент в специфичном для атрибута `type` формате: один формат для авторов и другой — для редакторов.

Сначала понадобится пара классов — по одному для авторов и редакторов:

```
public class AuthorHandler
{
   public void Display(XElement element)
   {
      Console.WriteLine("БИОГРАФИЯ АВТОРА ");
      Console.WriteLine("--------------------------");
      Console.WriteLine("ФИО: {0} {1}",
        (string)element.Element("FirstName"),
        (string)element.Element("LastName"));
```

```
            Console.WriteLine("Язык: {0}", (string)element.Attribute("language"));
            Console.WriteLine("Опыт: {0}", (string)element.Attribute("experience"));
            Console.WriteLine("=========================" + System.Environment.NewLine);
        }
    }
    public class EditorHandler
    {
        public void Display(XElement element)
        {
            Console.WriteLine("БИОГРАФИЯ РЕДАКТОРА");
            Console.WriteLine("--------------------------");
            Console.WriteLine("ФИО: {0}", (string)element.Element("FirstName"));
            Console.WriteLine(" {0}", (string)element.Element("LastName"));
            Console.WriteLine("=========================" + System.Environment.NewLine);
        }
    }
```

Здесь нет ничего особенного. Нужны два обработчика, ведущих себя по-разному. В этом случае они будут отображать данные элемента в немного отличающемся формате. Конечно, они не обязательно должны только отображать данные. Они могут делать все, что необходимо. Аннотации также могут не быть обработчиками, а просто какими-то ассоциированными данными. Но в рассматриваемом примере это обработчики.

Поскольку этот пример сложнее, чем обычно, части кода разделяются пояснениями, как показано в листинге 7.85.

Листинг 7.85. Добавление, извлечение и удаление аннотаций

```
// Это используется для хранения ссылки на один из элементов в дереве XML.
XElement firstParticipant;
XDocument xDocument = new XDocument(
  new XElement("BookParticipants", firstParticipant =
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XAttribute("experience", "first-time"),
      new XAttribute("language", "English"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));
// Отобразить документ для справочных целей.
Console.WriteLine(xDocument + System.Environment.NewLine);
```

Все, что делается в этой части — это построение типичного документа XML, который будет использован далее, а также его отображение. В следующей части кода выполняется перечисление участников книги, и для каждого создается экземпляр обработчика на основе значения атрибута `type`. Затем к элементу добавляется аннотация для соответствующего обработчика.

```
    // Добавить некоторые аннотации на основе значения атрибута type.
    foreach(XElement e in xDocument.Element("BookParticipants").Elements())
    {
        if((string)e.Attribute("type") == "Author")
        {
            AuthorHandler aHandler = new AuthorHandler();
            e.AddAnnotation(aHandler);
        }
```

```
    else if((string)e.Attribute("type") == "Editor")
    {
        EditorHandler eHandler = new EditorHandler();
        e.AddAnnotation(eHandler);
    }
}
```

Теперь каждый элемент BookParticipant имеет добавленный к нему обработчик в виде аннотации, зависящий от его атрибута type. Теперь можно выполнить перечисление элементов с вызовом обработчика, извлеченного из аннотации элемента:

```
AuthorHandler aHandler2;
EditorHandler eHandler2;
foreach(XElement e in xDocument.Element("BookParticipants").Elements())
{
    if((string)e.Attribute("type") == "Author")
    {
        aHandler2 = e.Annotation<AuthorHandler>();
        if(aHandler2 != null)
        {
            aHandler2.Display(e);
        }
    }
    else if((string)e.Attribute("type") == "Editor")
    {
        eHandler2 = e.Annotation<EditorHandler>();
        if(eHandler2 != null)
        {
            eHandler2.Display(e);
        }
    }
}
```

В этой точке для каждого элемента будет вызван обработчик отображения. Конкретный вызываемый обработчик зависит от атрибута type. Затем аннотации элементов удаляются:

```
foreach(XElement e in xDocument.Element("BookParticipants").Elements())
{
    if((string)e.Attribute("type") == "Author")
    {
        e.RemoveAnnotation<AuthorHandler>();
    }
    else if((string)e.Attribute("type") == "Editor")
    {
        e.RemoveAnnotation<EditorHandler>();
    }
}
```

Это пример состоит из четырех основных разделов. В первом разделе строится и отображается документ XML. Это уже много раз демонстрировалось. Во втором разделе выполняется перечисление элементов BookParticipants, с добавлением к каждому обработчика на основе атрибута type. В третьем разделе вновь производится перечисление элементов BookParticipants, при этом на основе атрибута type извлекается обработчик и вызывается метод Display объекта обработчика. В четвертом разделе элементы BookParticipants опять перечисляются с целью удаления аннотаций.

Обратите внимание, что при обращении к атрибутам они приводятся к string, чтобы получить значение.

Следует помнить, что эти аннотации могут быть любыми объектами данных, которые нужно ассоциировать с элементом.

И, наконец, вот результат:

```
<BookParticipants>
  <BookParticipant type="Author" experience="first-time" language="English">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
  <BookParticipant type="Editor">
    <FirstName>Ewan</FirstName>
    <LastName>Buckingham</LastName>
  </BookParticipant>
</BookParticipants>

БИОГРАФИЯ АВТОРА
--------------------------
ФИО:  Joe Rattz
Язык: English
Опыт: first-time
==========================

БИОГРАФИЯ РЕДАКТОРА
--------------------------
ФИО:  Ewan
      Buckingham
==========================
```

Относительно результатов также важно отметить, что разные обработчики вызываются в зависимости от атрибута `type` с использованием аннотаций. Естественно, объекты, которые добавляются в виде аннотаций, могут служить любой цели, а не обязательно быть какими-то обработчиками.

События XML

API-интерфейс LINQ to XML предоставляет возможность регистрировать обработчик события, чтобы можно было получать уведомления всякий раз, когда объект, унаследованный от `XObject`, изменяется либо уже изменен.

Первое, что необходимо знать при регистрации обработчика события для объекта — событие будет инициировано на объекте, когда этот объект либо любой его производный объект изменится. Это значит, что если регистрируется обработчик события на документе или корневом элементе, то всякое изменение в дереве приведет к вызову зарегистрированного метода. Поэтому не выдвигайте никаких предположений относительно типа данных объекта, который вызвал событие. Когда вызывается зарегистрированный метод, инициировавший событие объект будет передан в качестве отправителя события, и типом его данных будет `object`. Будьте очень осторожны, выполняя его приведение или обращаясь к его свойствам, либо вызывая его методы. Может оказаться, что его тип не тот, что ожидался. Эта проблема будет демонстрироваться в листинге 7.86, где `object` на самом деле оказывается объектом `XText`, тогда как ожидалось, что он относится к типу `XElement`.

И, наконец, имейте в виду, что конструирование XML не инициирует никаких событий. Да и как оно могло бы это делать? До конструирования никаких обработчиков событий еще не зарегистрировано. Только модификация или удаление уже существующей XML-разметки может инициировать возникновение события, и только в случае, если зарегистрирован обработчик события.

XObject.Changing

Это событие возникает, когда объект, унаследованный от XObject, собирается измениться, но перед действительным его изменением. Обработчик события регистрируется добавлением объекта типа EventHandler к событию Changing объекта, как показано ниже:

```
myobject.Changing += new
  EventHandler<XObjectChangeEventArgs>(MyHandler);
```

Делегат метода должен соответствовать следующей сигнатуре:

```
void MyHandler(object sender, XObjectChangeEventArgs cea)
```

sender — это объект, который собирается изменяться и который инициировал событие. Аргументы события изменения, cea, содержат свойство по имени ObjectChange типа XObjectChange, указывающее тип изменения, которое должно произойти: XObjectChange.Add, XObjectChange.Name, XObjectChange.Remove или XObjectChange.Value.

XObject.Changed

Это событие инициируется после того, как объект-наследник XObject был изменен. Обработчик события регистрируется добавлением объекта типа EventHandler к событию Changed объекта, как показано ниже:

```
myobject.Changed += new EventHandler<XObjectChangeEventArgs>(MyHandler);
```

Делегат метода должен соответствовать следующей сигнатуре:

```
void MyHandler(object sender, XObjectChangeEventArgs cea)
```

sender — это изменившийся объект, который вызвал возникновение события. Аргументы события change по имени, cea, содержат свойство по имени ObjectChange типа XObjectChange, указывающее на тип произошедшего изменения: XObjectChange.Add, XObjectChange.Name, XObjectChange.Remove или XObjectChange.Value.

Несколько примеров событий

Чтобы увидеть все, что необходимо для обработки событий XObject, понадобится пример. Однако прежде чем будет показан необходимый код, понадобятся некоторые обработчики событий, приведенные ниже.

Этот метод будет зарегистрирован для обработки события изменения элемента

```
public static void MyChangingEventHandler(object sender, XObjectChangeEventArgs cea)
{
  Console.WriteLine("Тип изменяемого объекта: {0}, Тип изменения: {1}",
    sender.GetType().Name, cea.ObjectChange);
}
```

Приведенный выше метод будет зарегистрирован в качестве обработчика события, которое происходит, когда элемент изменен.

Ранее упоминалось, что событие инициируется, когда изменяется любой из объектов-потомков зарегистрированного объекта. Чтобы лучше продемонстрировать это, предусмотрен также дополнительный метод, который зарегистрируется для вызова, когда объект изменен. Его единственное назначение — сделать более наглядным возникновение события Changed, независимо от того, изменен сам объект или его потомок, находящийся несколькими уровнями ниже. Этот метод представлен ниже.

Этот метод будет зарегистрирован для обработки события изменения XML-документа

```
public static void DocumentChangedHandler(object sender, XObjectChangeEventArgs cea)
{
  Console.WriteLine("Doc: Тип измененного объекта: {0}, Тип изменения: {1}{2}",
    sender.GetType().Name, cea.ObjectChange, System.Environment.NewLine);
}
```

Единственное существенное изменение между методом `DocumentChangedHandler` и методом `MyChangedEventHandler` заключается в том, что метод `DocumentChanged Handler` начинает экранный вывод с префикса `"Doc:"`, чтобы указать на то, что вызван метод-обработчик для события `Changed` документа, а не обработчик того же события элемента.

Теперь взглянем на пример кода, приведенный в листинге 7.86.

Листинг 7.86. Обработка события `XObject`

```
XElement firstParticipant;
XDocument xDocument = new XDocument(
  new XElement("BookParticipants", firstParticipant =
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));
Console.WriteLine("{0}{1}", xDocument, System.Environment.NewLine);
```

Пока ничего нового. Как это делалось много раз, здесь с помощью функционального конструирования создается документ XML, который затем отображается. Обратите внимание, что, как и в большинстве предыдущих примеров, сохраняется ссылка на первый элемент `BookParticipant`. Это тот элемент, для событий которого регистрируются обработчики.

```
firstParticipant.Changing += new
  EventHandler<XObjectChangeEventArgs>(MyChangingEventHandler);
firstParticipant.Changed += new
  EventHandler<XObjectChangeEventArgs>(MyChangedEventHandler);
xDocument.Changed += new
  EventHandler<XObjectChangeEventArgs>(DocumentChangedHandler);
```

В результате зарегистрировано, что первый элемент `BookParticipant` должен принимать события `Changing` и `Changed`. Вдобавок зарегистрирован обработчик для получения документом события `Changed`. Это сделано для того, чтобы продемонстрировать, что событие получается, даже когда изменяется объект-потомок, а не сам объект, для которого зарегистрирован обработчик. Теперь внесем изменение:

```
firstParticipant.Element("FirstName").Value = "Seph";
Console.WriteLine("{0}{1}", xDocument, System.Environment.NewLine);
```

Все, что сделано — модифицировано значение подэлемента `FirstName` элемента `BookParticipant`. Затем отображается результирующий XML-документ. Посмотрим на результат:

```
<BookParticipants>
  <BookParticipant type="Author">
    <FirstName>Joe</FirstName>
```

```
    <LastName>Rattz</LastName>
  </BookParticipant>
  <BookParticipant type="Editor">
    <FirstName>Ewan</FirstName>
    <LastName>Buckingham</LastName>
  </BookParticipant>
</BookParticipants>

Тип изменяемого объекта: XText, Тип изменения: Remove
Тип измененного объекта: XText, Тип изменения: Remove
Doc: Тип измененного объекта: XText, Тип изменения: Remove

Тип изменяемого объекта: XText, Тип изменения: Add
Тип измененного объекта: XText, Тип изменения: Add
Doc: Тип измененного объекта: XText, Тип изменения: Add

<BookParticipants>
  <BookParticipant type="Author">
    <FirstName>Seph</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
  <BookParticipant type="Editor">
    <FirstName>Ewan</FirstName>
    <LastName>Buckingham</LastName>
  </BookParticipant>
</BookParticipants>
```

Сравнив документ в начале и конце вывода результатов, легко заметить, что значение элемента `FirstName` изменилось, чего и следовало ожидать. Больше всего здесь интересует вывод, вызванный возникновением события, который находится между двумя выводами документа XML. Обратите внимание, что типом изменяемого объекта является `XText`. Ожидалось ли этого? Нет. Ожидался тип `XElement`. При установке значения элемента в виде строкового литерала легко забыть, что при этом автоматически создается объект типа `XText`.

Если посмотреть на вывод события, становится немного яснее, что именно происходит, когда изменяется значение элемента. Значение `XText`, подлежащее изменению, сначала должно быть удалено, и оно удаляется. Затем инициируется событие `Changed` документа. Отсюда видно, что поток событий распространяется вверх.

Затем возникает та же последовательность событий, но на этот раз объект `XText` добавляется. Таким образом, теперь стало ясно, что при изменении строкового значения элемента объект `XText` удаляется, а затем добавляется заново.

В предыдущем примере использовались именованные методы, но это вовсе не обязательно. С тем же успехом можно было бы применять анонимные методы или даже лямбда-выражения. В листинге 7.87 показан тот же пример, но на этот раз вместо регистрации уже реализованных методов обработчиков используются лямбда-выражения, чтобы “на лету” определить код, вызываемый событиями.

Листинг 7.87. Обработка события `XObject` с использованием лямбда-выражений

```
XElement firstParticipant;
XDocument xDocument = new XDocument(
  new XElement("BookParticipants", firstParticipant =
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
```

```
    new XElement("FirstName", "Ewan"),
    new XElement("LastName", "Buckingham"))));
Console.WriteLine("{0}{1}", xDocument, System.Environment.NewLine);
firstParticipant.Changing += new EventHandler<XObjectChangeEventArgs>(
  (object sender, XObjectChangeEventArgs cea) =>
    Console.WriteLine("Тип изменяемого объекта: {0}, Тип изменения: {1}",
      sender.GetType().Name, cea.ObjectChange));
firstParticipant.Changed += (object sender, XObjectChangeEventArgs cea) =>
  Console.WriteLine("Тип измененного объекта: {0}, Тип изменения: {1}",
    sender.GetType().Name, cea.ObjectChange);
xDocument.Changed += (object sender, XObjectChangeEventArgs cea) =>
  Console.WriteLine("Doc: Тип измененного объекта: {0}, Тип изменения: {1}{2}",
    sender.GetType().Name, cea.ObjectChange, System.Environment.NewLine);
xDocument.Changed += new XObjectChangeEventHandler((sender, cea) =>
  Console.WriteLine("Doc: Тип измененного объекта: {0}, Тип изменения: {1}{2}",
    sender.GetType().Name, cea.ObjectChange, System.Environment.NewLine));
firstParticipant.Element("FirstName").Value = "Seph";
Console.WriteLine("{0}{1}", xDocument, System.Environment.NewLine);
```

Теперь код совершенно самодостаточен и не зависит от ранее написанных методов-обработчиков. Взглянем на результаты:

```
<BookParticipants>
  <BookParticipant type="Author">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
  <BookParticipant type="Editor">
    <FirstName>Ewan</FirstName>
    <LastName>Buckingham</LastName>
  </BookParticipant>
</BookParticipants>

Тип изменяемого объекта: XText, Тип изменения: Remove
Тип измененного объекта: XText, Тип изменения: Remove
Doc: Тип измененного объекта: XText, Тип изменения: Remove

Тип изменяемого объекта: XText, Тип изменения: Add
Тип измененного объекта: XText, Тип изменения: Add
Doc: Тип измененного объекта: XText, Тип изменения: Add

<BookParticipants>
  <BookParticipant type="Author">
    <FirstName>Seph</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
  <BookParticipant type="Editor">
    <FirstName>Ewan</FirstName>
    <LastName>Buckingham</LastName>
  </BookParticipant>
</BookParticipants>
```

Вывод выглядит точно так же, как в предыдущем случае. Этот пример также демонстрирует изящество лямбда-выражений.

Трюк, забава или неопределенность?

Помните ли вы “проблему Хэллоуина”, о которой говорилось ранее в главе? Избегайте соблазна внутри методов-обработчиков вносить изменения в область дерева XML, со-

держащую объект, для которого инициировано текущее событие. Это приведет к непредсказуемому эффекту для дерева XML и генерируемых событий.

Резюме

В настоящей главе было показано, как использовать LINQ to XML для создания, модификации и обхода XML-документов, а также как выполнять запросы LINQ на отдельном XML-объекте. Новый API-интерфейс для создания и модификации данных XML — не роскошь, а необходимость для выполнения запросов LINQ. Хорошо проектировать данные на структуру XML не получится без возможности создания "на лету" элементов XML, инициализации их значений и помещения в дерево XML в одном операторе. Интерфейс W3C DOM XML API полностью лишен гибкости, необходимой для выполнения запроса LINQ. И это все привело к появлению совершенно нового API-интерфейса для работы с XML.

Хотя в главе демонстрировались базовые запросы LINQ на данных XML, вы не могли не заметить серьезных ограничений, присущих запросам LINQ. Все запросы, которые выполнялись здесь, имели дело с единственным объектом XML, таким как элемент. Для этого элемента запрашивались предки и потомки. А что, если нужно выполнить запрос LINQ на *последовательности* элементов, такой как все потомки или все предки элементов, которые, возможно, являются потомками одного элемента? Для этого понадобится дополнительный набор операций XML. В следующей главе рассматриваются новые операции LINQ to XML, предназначенные специально для этих целей.

ГЛАВА 8

Операции LINQ to XML

В этот момент, когда уже достаточно много известно об LINQ to XML, возможно, возник вопрос: когда же мы перейдем к запросам? Ответ: вы уже видели запросы, причем неоднократно. На протяжении всей предыдущей главы постоянно выполнялись запросы LINQ to XML — возвращали они просто все дочерние элементы одного элемента или получали всех его предков. Помните метод `XContainer.Elements` и ситуации, когда он вызывался? Именно там, помимо прочих мест, применялись запросы LINQ to XML. По причине гладкой интеграции запросов LINQ в язык иногда можно не заметить, что выполняется именно запрос.

Поскольку многие из рассмотренных до сих пор методов классов возвращают последовательность объектов классов XML, т.е. `IEnumerable<T>`, где `T` — один из классов API-интерфейса LINQ to XML, на возвращенных последовательностях можно вызывать стандартные операции запросов, обеспечивая им еще большую мощь и гибкость.

Так что способы получить последовательность XML-объектов из единственного объекта XML, таких как предки или потомки любого данного элемента, имеются. Но чего не хватает — так это приемов выполнения операций LINQ to XML на каждом объекте этих последовательностей. Например, не существует простого способа получить последовательность элементов и выполнить на каждом элементе последовательности другую XML-операцию, такую как возврат дочерних элементов. Другими словами, *можно* получить последовательность дочерних элементов, вызвав метод `Elements` элемента, но *нельзя* получить последовательность дочерних элементов для дочернего элемента. Причина в том, что метод `Elements` должен вызываться на контейнере `XContainer`, таком как `XElement` или `XDocument`, но его нельзя вызывать на *последовательности* объектов `XContainer`. И здесь на помощь приходят операции LINQ to XML.

Введение в операции LINQ to XML

API-интерфейс LINQ to XML расширяет стандартные операции запросов LINQ to Objects специфичными для XML операциями. Эти XML-операции являются расширяющими методами, которые определены в классе `System.Xml.Linq.Extensions`, представляющим собой не что иное, как класс-контейнер расширяющих методов.

Каждая из этих операций XML вызывается на *последовательности* некоторого типа данных LINQ to XML и выполняет некоторые действия на каждом элементе этой последовательности, например, возврат всех предков или потомков данного элемента.

Почти каждая операция XML, рассмотренная в этой главе, имеет эквивалентный метод, описанный в предыдущей главе. Отличие в том, что метод из предыдущей главы вызывается на единственном объекте, а операция в настоящей главе — на *последовательности* объектов. Например, в предыдущей главе был показан метод `XContainer.Elements`. Его прототип выглядит следующим образом:

```
IEnumerable<XElement> XContainer.Elements()
```

В этой главе речь пойдет об операции Extensions.Elements, прототип которой выглядит так:

```
IEnumerable<XElement> Elements<T> (this IEnumerable<T> source) where T : XContainer
```

Между этими двумя методами есть существенная разница. Первый прототип вызывается на единственном объекте, унаследованном от XContainer, в то время как второй прототип вызывается на последовательности объектов, каждый из которых должен наследоваться от XContainer. Не забывайте об этом различии.

Чтобы отличить методы, описанные в предыдущей главе, и расширяющие методы, описанные в этой, расширяющие методы обычно будут называться *операциями*.

Все операции подробно рассматриваются в последующих разделах.

Ancestors

Операция Ancestors может быть вызвана на последовательности узлов и возвращает последовательность, содержащую элементы-предки каждого исходного узла.

Прототипы

Операция Ancestors имеет два прототипа, описанные ниже.

Первый прототип Ancestors

```
public static IEnumerable<XElement> Ancestors<T> (
  this IEnumerable<T> source
) where T : XNode
```

Эта версия операции может быть вызвана на последовательности узлов или объектов, унаследованных от XNode. Она возвращает последовательность элементов, содержащую элементы-предки каждого узла исходной последовательности source.

Второй прототип Ancestors

```
public static IEnumerable<XElement> Ancestors<T> (
  this IEnumerable<T> source,
  XName name
) where T : XNode
```

Эта версия подобна первой, за исключением того, что ей передается имя, и в выходной последовательности возвращаются только элементы-предки, соответствующие указанному имени.

Примеры

В листинге 8.1 представлен пример вызова первого прототипа Ancestors.

Листинг 8.1. Пример вызова первого прототипа Ancestors

```
XDocument xDocument = new XDocument(
  new XElement("BookParticipants",
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));
IEnumerable<XElement> elements =
```

```
  xDocument.Element("BookParticipants").Descendants("FirstName");
// Сначала отобразим исходные элементы.
foreach (XElement element in elements)
{
  Console.WriteLine("Исходный элемент: {0} : значение = {1}",
  element.Name, element.Value);
}
// Теперь отобразим элементы-предки для каждого исходного элемента.
foreach (XElement element in elements.Ancestors())
{
  Console.WriteLine("Элемент-предок: {0}", element.Name);
}
```

В приведенном примере сначала создается документ XML. Затем генерируется последовательность элементов `FirstName`. Вспомните, что этот метод `Ancestors` вызывается на последовательности узлов, а не отдельном узле, поэтому нужна последовательность, на которой можно будет вызвать его. Поскольку требуется возможность отобразить имена узлов в целях идентификации, в действительности строится последовательность элементов, потому что элементы имеют имена, а узлы — нет. Затем проводится перечисление по последовательности с отображением исходных элементов — просто чтобы можно было увидеть исходную последовательность. После этого перечисляются элементы, возвращенные методом `Ancestors`, с отображением каждого из них. Результат выглядит так:

```
Исходный элемент: FirstName : значение = Joe
Исходный элемент: FirstName : значение = Ewan
Элемент-предок: BookParticipant
Элемент-предок: BookParticipants
Элемент-предок: BookParticipant
Элемент-предок: BookParticipants
```

Как видите, в выводе присутствуют два элемента исходной последовательности — два элемента `FirstName`. Затем представлены предки каждого из этих двух элементов.

Таким образом, с помощью операции `Ancestors` можно извлекать все элементы-предки для каждого узла в последовательности. В данном случае это последовательность элементов, что вполне нормально, поскольку элемент наследуется от узла. Вспомните, что не следует путать операцию `Ancestors`, вызванную на последовательности узлов, которая была продемонстрирована здесь, с методом `Ancestors`, описанным в предыдущей главе.

В этом виде пример не столь выразителен, как мог бы быть, поскольку код расширен в демонстрационных целях. Например, нужно было захватить последовательность элементов `FirstName`, чтобы отобразить их наглядно в выводе. То есть оператор, содержащий вызов метода `Descendants`, и последующий блок `foreach` предназначены для этой цели. Затем во втором цикле `foreach` вызывается операция `Ancestors` и отображается каждый элемент-предок. В действительности, во втором цикле `foreach` можно было бы вызвать на каждом элементе последовательности элементов `FirstName` метод `Ancestors` из предыдущей главы, не прибегая к методу `Ancestors`, который демонстрируется здесь. В листинге 8.2 представлен пример получения того же результата без использования операции `Ancestors`.

Листинг 8.2. Получение того же результата, что и в листинге 8.1, но без вызова операции `Ancestors`

```
XDocument xDocument = new XDocument(
  new XElement("BookParticipants",
```

```
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));

IEnumerable<XElement> elements =
  xDocument.Element("BookParticipants").Descendants("FirstName");

// Сначала отобразим исходные элементы.
foreach (XElement element in elements)
{
  Console.WriteLine("Исходный элемент: {0} : значение = {1}",
    element.Name, element.Value);
}

foreach (XElement element in elements)
{
  // Вызов метода Ancestors на каждом элементе.
  foreach(XElement e in element.Ancestors())
    // Теперь отобразим элементы-предки для каждого исходного элемента.
    Console.WriteLine("Элемент-предок: {0}", e.Name);
}
```

Разница между этим примером и предыдущим состоит в том, что вместо вызова *операции* `Ancestors` на последовательности `elements` в цикле `foreach` просто осуществляется проход в цикле по каждому элементу в последовательности с вызовом на нем *метода* `Ancestors`. Этот код дает точно такой же вывод:

```
Исходный элемент: FirstName : значение = Joe
Исходный элемент: FirstName : значение = Ewan
Элемент-предок: BookParticipant
Элемент-предок: BookParticipants
Элемент-предок: BookParticipant
Элемент-предок: BookParticipants
```

Однако, благодаря операции `Ancestor` и лаконичности LINQ, этот запрос можно комбинировать в единственный более краткий оператор, как показано в листинге 8.3.

Листинг 8.3. Более краткий пример вызова первого прототипа `Ancestors`

```
XDocument xDocument = new XDocument(
  new XElement("BookParticipants",
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));

foreach (XElement element in
  xDocument.Element("BookParticipants").Descendants("FirstName").Ancestors())
{
  Console.WriteLine("Элемент-предок: {0}", element.Name);
}
```

В этом примере сразу вызывается операция Ancestors на последовательности элементов, возвращенных методом Descendants. Поэтому метод Descendants возвращает последовательность элементов, а операция Ancestors вернет последовательность элементов, содержащих всех предков каждого элемента в последовательности, на которой она вызвана.

Поскольку этот код призван быть более кратким, он не отображает элементов FirstName, как это делалось в предыдущих двух примерах. Тем не менее, элементы-предки должны быть теми же самыми. Давайте в этом убедимся:

```
Элемент-предок: BookParticipant
Элемент-предок: BookParticipants
Элемент-предок: BookParticipant
Элемент-предок: BookParticipants
```

Так и есть! В коде реального приложения, скорее всего, будет выбран более компактный запрос, вроде только что продемонстрированного. Однако в этой главе для наглядности примеры будут более многословными, как в листинге 8.1.

Для демонстрации использования второго прототипа Ancestors применяется тот же базовый код, что и в листинге 8.1, но вызов операции Ancestors изменяется так, что она будет включать параметр BookParticipant, чтобы можно было получить только элементы с соответствующим именем. Код представлен в листинге 8.4.

Листинг 8.4. Вызов второго прототипа Ancestors

```
XDocument xDocument = new XDocument(
  new XElement("BookParticipants",
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));
IEnumerable<XElement> elements =
  xDocument.Element("BookParticipants").Descendants("FirstName");
// Сначала отобразим исходные элементы.
foreach (XElement element in elements)
{
  Console.WriteLine("Исходный элемент: {0} : значение = {1}",
    element.Name, element.Value);
}
// Теперь отобразим родительские элементы каждого исходного элемента.
foreach (XElement element in elements.Ancestors("BookParticipant"))
{
  Console.WriteLine("Элемент-предок: {0}", element.Name);
}
```

Теперь результаты должны включать только элементы BookParticipant и, конечно же, исходные элементы, но два элемента BookParticipants, которые были показаны в примере с первым прототипом, теперь должны исчезнуть:

```
Исходный элемент: FirstName : значение = Joe
Исходный элемент: FirstName : значение = Ewan
Элемент-предок: BookParticipant
Элемент-предок: BookParticipant
```

Так и произошло.

AncestorsAndSelf

Операция AncestorsAndSelf может быть вызвана на последовательности элементов и возвращает последовательность, которая включает элементы-предки каждого исходного элемента, а также его самого. Эта операция подобна операции Ancestors за исключением того, что она может быть вызвана только на элементах, а не на узлах, и также включает исходные элементы в возвращенную последовательность элементов-предков.

Прототипы

Операция AncestorsAndSelf имеет два прототипа, описанные ниже.

Первый прототип AncestorsAndSelf

```
public static IEnumerable<XElement> AncestorsAndSelf (
  this IEnumerable<XElement> source
)
```

Эта версия операции может быть вызвана на последовательности элементов и возвращает последовательность элементов, включающих каждый исходный элемент и всех его предков.

Второй прототип AncestorsAndSelf

```
public static IEnumerable<XElement> AncestorsAndSelf (
  this IEnumerable<XElement> source,
  XName name
)
```

Эта версия подобна первой, за исключением дополнительного параметра — имени, и возвращает в выходной последовательности только те элементы и его предков, которые соответствуют указанному имени.

Примеры

Для демонстрации первого прототипа AncestorsAndSelf используется тот же базовый пример, что применялся для первого прототипа Ancestors, только на этот раз вместо Ancestors вызывается операция AncestorsAndSelf, как показано в листинге 8.5.

Листинг 8.5. Пример вызова первого прототипа AncestorsAndSelf

```
XDocument xDocument = new XDocument(
  new XElement("BookParticipants",
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));
IEnumerable<XElement> elements =
  xDocument.Element("BookParticipants").Descendants("FirstName");

// Сначала отобразим исходные элементы.
foreach (XElement element in elements)
{
  Console.WriteLine("Исходный элемент: {0} : значение = {1}",
    element.Name, element.Value);
}
```

```
// Теперь отобразим элементы-предки для каждого исходного элемента.
foreach (XElement element in elements.AncestorsAndSelf())
{
  Console.WriteLine("Элемент-предок: {0}", element.Name);
}
```

Как и в первом прототипе `Ancestors`, сначала создается документ XML. Затем генерируется последовательность элементов `FirstName`. Вспомните, что этот метод `AncestorsAndSelf` вызывается на последовательности элементов, а не на отдельном элементе, так что понадобилась последовательность, на которой его вызвать. Затем эта последовательность перечисляется с отображением элементов. После этого перечисляются элементы, возвращенные методом `AncestorsAndSelf`, и также отображаются.

Если все работает, как ожидалось, то результат должен быть таким же, как и в примере прототипа `Ancestors`, за исключением того, что элементы `FirstName` также будут включены в вывод. И вот результат:

```
Исходный элемент: FirstName : значение = Joe
Исходный элемент: FirstName : значение = Ewan
Элемент-предок: FirstName
Элемент-предок: BookParticipant
Элемент-предок: BookParticipants
Элемент-предок: FirstName
Элемент-предок: BookParticipant
Элемент-предок: BookParticipants
```

Для демонстрации второго прототипа `AncestorsAndSelf` используется тот же базовый пример, что и при демонстрации второго прототипа `Ancestors`, разумеется, с заменой вызова `Ancestors` на `AncestorsAndSelf`, как показано в листинге 8.6.

Листинг 8.6. Вызов второго прототипа `AncestorsAndSelf`

```
XDocument xDocument = new XDocument(
  new XElement("BookParticipants",
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));

IEnumerable<XElement> elements =
  xDocument.Element("BookParticipants").Descendants("FirstName");

// Сначала отобразим исходные элементы.
foreach (XElement element in elements)
{
  Console.WriteLine("Исходный элемент: {0} : значение = {1}",
    element.Name, element.Value);
}

// Теперь отобразим родительские элементы каждого исходного элемента.
foreach (XElement element in elements.AncestorsAndSelf("BookParticipant"))
{
  Console.WriteLine("Элемент-предок: {0}", element.Name);
}
```

Теперь должны быть получены только элементы по имени BookParticipant. Вот результат:

```
Исходный элемент: FirstName : значение = Joe
Исходный элемент: FirstName : значение = Ewan
Элемент-предок: BookParticipant
Элемент-предок: BookParticipant
```

Обратите внимание, что в выводе метода AncestorsAndSelf теперь содержатся только элементы BookParticipant, потому что только они соответствуют переданному имени. В этом случае даже не получены исходные элементы, поскольку их имена не соответствуют. Так что функция работает должным образом.

Этот прототип операции кажется довольно бесполезным. Сколько уровней одноименных элементов планируется иметь в дереве XML? Если вы не ответите *минимум два*, то каким образом этот метод сможет вернуть сами исходные элементы вместе с предками? Это кажется маловероятным.

Attributes

Операция Attributes может быть вызвана на последовательности элементов и возвращает последовательность, содержащую атрибуты каждого исходного элемента.

Прототипы

Операция Attributes имеет два прототипа, описанные ниже.

Первый прототип Attributes

```
public static IEnumerable<XAttribute> Attributes (
  this IEnumerable<XElement> source
)
```

Эта версия операции может быть вызвана на последовательности элементов и возвращает последовательность атрибутов, содержащую все атрибуты каждого из исходных элементов.

Второй прототип Attributes

```
public static IEnumerable<XAttribute> Attributes (
  this IEnumerable<XElement> source,
  XName name
)
```

Эта версия операции подобна первой, за исключением того, что в выходной последовательности возвращаются только атрибуты, соответствующие указанному имени.

Примеры

Для демонстрации первого прототипа Attributes будет построено то же самое дерево XML, что и в предыдущих примерах. Однако генерируемая последовательность исходных элементов будет немного отличаться, потому что нужна последовательность элементов с атрибутами. Поэтому последовательность элементов BookParticipant генерируется и обрабатывается, как показано в листинге 8.7.

Листинг 8.7. Вызов первого прототипа Attributes

```
XDocument xDocument = new XDocument(
  new XElement("BookParticipants",
    new XElement("BookParticipant",
```

```
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));

IEnumerable<XElement> elements =
  xDocument.Element("BookParticipants").Elements("BookParticipant");

// Сначала отобразим исходные элементы.
foreach (XElement element in elements)
{
  Console.WriteLine("Исходный элемент: {0} : значение = {1}",
    element.Name, element.Value);
}

// Теперь отобразим атрибуты каждого элемента.
foreach (XAttribute attribute in elements.Attributes())
{
  Console.WriteLine("Атрибут: {0} : значение = {1}",
    attribute.Name, attribute.Value);
}
```

После получения последовательности элементов `BookParticipant` отображаются все ее элементы. Затем на исходной последовательности вызывается операция `Attributes` и отображаются атрибуты в последовательности, возвращенной операцией `Attributes`. Вот результат:

```
Исходный элемент: BookParticipant : значение = JoeRattz
Исходный элемент: BookParticipant : значение = EwanBuckingham
Атрибут: type : значение = Author
Атрибут: type : значение = Editor
```

Как видите, атрибуты извлечены. Для демонстрации второго прототипа `Attributes` применяется тот же базовый пример, но на этот раз указывается имя, которому должны соответствовать атрибуты, возвращенные операцией `Attributes` (листинг 8.8).

Листинг 8.8. Вызов первого прототипа `Attributes`

```
XDocument xDocument = new XDocument(
  new XElement("BookParticipants",
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));

IEnumerable<XElement> elements =
  xDocument.Element("BookParticipants").Elements("BookParticipant");

// Сначала отобразим исходные элементы.
foreach (XElement element in elements)
{
  Console.WriteLine("Исходный элемент: {0} : значение = {1}",
    element.Name, element.Value);
}
```

```
// Теперь отобразим атрибуты каждого элемента.
foreach (XAttribute attribute in elements.Attributes("type"))
{
  Console.WriteLine("Атрибут: {0} : значение = {1}",
    attribute.Name, attribute.Value);
}
```

В приведенном коде указано, что атрибуты должны соответствовать имени `type`. Поэтому данный пример должен выдать тот же вывод, что и предыдущий. По нажатию <Ctrl+F5> увидим следующее:

```
Исходный элемент: BookParticipant : значение = JoeRattz
Исходный элемент: BookParticipant : значение = EwanBuckingham
Атрибут: type : значение = Author
Атрибут: type : значение = Editor
```

Результат ожидаемый. Если бы было указано имя `Type` с заглавной первой буквой, то эти два атрибута не были бы отображены, потому что операция `Attributes` не вернула бы их из входной последовательности. Это демонстрирует случай, когда соответствия имени нет, а также тот факт, что сопоставление имени чувствительно к регистру, что не удивительно, учитывая зависимость от регистра языка XML в целом.

DescendantNodes

Операция `DescendantNodes` может быть вызвана на последовательности элементов и возвращает последовательность, содержащую узлы-наследники каждого элемента в документе.

Прототипы

Операция `DescendantNodes` имеет один прототип, который описан ниже.

Единственный прототип DescendantNodes

```
public static IEnumerable<XNode> DescendantNodes<T> (
  this IEnumerable<T> source
) where T : XContainer
```

Эта версия может быть вызвана на последовательности элементов или документов и возвращает последовательность, содержащую каждый из узлов-потомков исходного элемента или документа.

Отличие от метода `XContainer.DescendantNodes` заключается в том, что этот метод вызывается на последовательности элементов или документов, а не на одном элементе или документе.

Примеры

Для примера построим то же самое дерево XML, что и в предыдущем примере, за исключением того, что также будет добавлен комментарий к первому элементу `BookParticipant`. Это нужно для того, чтобы как минимум один возвращенный узел не был элементом. При построении исходной последовательности элементов требуется, чтобы некоторые из них имели потомков, поэтому она строится из элементов `BookParticipant`, т.к. у них есть некоторые потомки (листинг 8.9).

Листинг 8.9. Вызов единственного прототипа `DescendantNodes`

```
XDocument xDocument = new XDocument(
  new XElement("BookParticipants",
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));

IEnumerable<XElement> elements =
  xDocument.Element("BookParticipants").Elements("BookParticipant");

// Сначала отобразим исходные элементы.
foreach (XElement element in elements)
{
  Console.WriteLine("Исходный элемент: {0} : значение = {1}",
    element.Name, element.Value);
}

// Теперь отобразим узлы-потомки каждого исходного элемента.
foreach (XNode node in elements.DescendantNodes())
{
  Console.WriteLine("Узел-потомок: {0}", node);
}
```

Как это принято в примерах настоящего раздела, сначала строится дерево XML и исходная последовательность элементов. В данном случае исходная последовательность содержит элементы `BookParticipant`. Затем на исходной последовательности вызывается операция `DescendantNodes` и отображаются результаты:

```
Исходный элемент: BookParticipant : значение = JoeRattz
Исходный элемент: BookParticipant : значение = EwanBuckingham
Узел-потомок: <!--This is a new author.-->
Узел-потомок: <FirstName>Joe</FirstName>
Узел-потомок: Joe
Узел-потомок: <LastName>Rattz</LastName>
Узел-потомок: Rattz
Узел-потомок: <FirstName>Ewan</FirstName>
Узел-потомок: Ewan
Узел-потомок: <LastName>Buckingham</LastName>
Узел-потомок: Buckingham
```

Обратите внимание, что в результате получаются не только элементы-потомки, но также и комментарий. Также обратите внимание, что для каждого элемента в документе XML получаются по два узла. Например, есть узел, значение которого выглядит как "`<FirstName>Joe</FirstName>`", а также узел со значением "`Joe`". Первый узел в паре — элемент `FirstName`. Второй узел — `XText` для этого элемента. Не забывайте об этих автоматически создаваемых объектах `XText`.

DescendantNodesAndSelf

Операция `DescendantNodesAndSelf` может быть вызвана на последовательности элементов и возвращает последовательность, содержащую собственно каждый исходный элемент и его узлы-потомки.

Прототипы

Операция DescendantNodesAndSelf имеет один прототип, описанный ниже.

Единственный прототип DescendantNodesAndSelf

```
public static IEnumerable<XNode> DescendantNodesAndSelf (
  this IEnumerable<XElement> source
)
```

Эта версия вызывается на последовательности элементов и возвращает последовательность узлов, содержащую собственно каждый исходный элемент и все его узлы-потомки.

Примеры

Для демонстрации работы этой операции применяется тот же пример, что использовался для демонстрации DescendantNodes, просто с заменой ее на DescendantNodesAndSelf, как показано в листинге 8.10.

Листинг 8.10. Вызов единственного прототипа DescendantNodesAndSelf

```
XDocument xDocument = new XDocument(
  new XElement("BookParticipants",
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));

IEnumerable<XElement> elements =
  xDocument.Element("BookParticipants").Elements("BookParticipant");

// Сначала отобразим исходные элементы.
foreach (XElement element in elements)
{
  Console.WriteLine("Исходный элемент: {0} : значение = {1}",
    element.Name, element.Value);
}

// Теперь отобразим узлы-потомки каждого исходного элемента.
foreach (XNode node in elements.DescendantNodesAndSelf())
{
  Console.WriteLine("Узел-потомок: {0}", node);
}
```

Вопрос в том, будет ли вывод таким же, как и у примера DescendantNodes, только с включением исходных элементов? Можете убедиться в этом:

```
Исходный элемент: BookParticipant : значение = JoeRattz
Исходный элемент: BookParticipant : значение = EwanBuckingham
Узел-потомок: <BookParticipant type="Author">
  <!--This is a new author.-->
  <FirstName>Joe</FirstName>
  <LastName>Rattz</LastName>
</BookParticipant>
```

```
Узел-потомок: <!--This is a new author.-->
Узел-потомок: <FirstName>Joe</FirstName>
Узел-потомок: Joe
Узел-потомок: <LastName>Rattz</LastName>
Узел-потомок: Rattz
Узел-потомок: <BookParticipant type="Editor">
  <FirstName>Ewan</FirstName>
  <LastName>Buckingham</LastName>
</BookParticipant>
Узел-потомок: <FirstName>Ewan</FirstName>
Узел-потомок: Ewan
Узел-потомок: <LastName>Buckingham</LastName>
Узел-потомок: Buckingham
```

В выводе присутствуют не только сами элементы `BookParticipant` и их потомки, но также один узел, не являющийся элементом — комментарий. В этом состоит отличие между операциями `DescendantNodesAndSelf` и описанной ниже `DescendantNodes`, которая пропускает узлы, не являющиеся элементами.

Descendants

Операция `Descendants` может быть вызвана на последовательности элементов или документов и возвращает последовательность элементов, содержащую все элементы-потомки каждого исходного элемента или документа.

Прототипы

Операция `Descendants` имеет два прототипа, описанные ниже.

Первый прототип Descendants

```
public static IEnumerable<XElement> Descendants<T> (
  this IEnumerable<T> source
) where T : XContainer
```

Эта версия вызывается на последовательности элементов или документов и возвращает последовательность документов, содержащую потомков каждого исходного элемента или документа.

Отличие этой операции от метода `XContainer.Descendants` в том, что этот метод вызывается на последовательности элементов или документов, а не на отдельном элементе или документе.

Второй прототип Descendants

```
public static IEnumerable<XElement> Descendants<T> (
  this IEnumerable<T> source,
  XName name
) where T : XContainer
```

Эта версия во всем подобна первой, за исключением того, что в выходной последовательности возвращаются только те элементы, которые соответствуют указанному имени.

Примеры

Для демонстрации первого прототипа используется в основном тот же пример, что и для операции `DescendantNodes`, вызывая вместо нее операцию `Descendants`. Вывод должен быть таким же, только без узлов, не являющихся элементами. То есть комментарии в выводе не появятся. Код представлен в листинге 8.11.

Листинг 8.11. Вызов первого прототипа `Descendants`

```
XDocument xDocument = new XDocument(
  new XElement("BookParticipants",
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));

IEnumerable<XElement> elements =
  xDocument.Element("BookParticipants").Elements("BookParticipant");

// Сначала отобразим исходные элементы.
foreach (XElement element in elements)
{
  Console.WriteLine("Исходный элемент: {0} : значение = {1}",
    element.Name, element.Value);
}

// Теперь отобразим узлы-потомки каждого исходного элемента.
foreach (XElement element in elements.Descendants())
{
  Console.WriteLine("Элемент-потомок: {0}", element);
}
```

Этот пример такой же, как предыдущий, но с тем отличием, что здесь выводятся только элементы-потомки двух элементов `BookParticipant`. Результат работы этого примера выглядит следующим образом:

```
Исходный элемент: BookParticipant : значение = JoeRattz
Исходный элемент: BookParticipant : значение = EwanBuckingham
Элемент-потомок: <FirstName>Joe</FirstName>
Элемент-потомок: <LastName>Rattz</LastName>
Элемент-потомок: <FirstName>Ewan</FirstName>
Элемент-потомок: <LastName>Buckingham</LastName>
```

Сравнивая эти результаты с примером операции `DescendantNodes`, можно заметить некоторые неожиданные отличия. Конечно, потомки помечены как элементы, а не узлы, и комментарий отсутствует, но при этом отсутствуют также узлы-потомки, такие как `Joe` и `Rattz`. Разумеется, эти узлы также не являются элементами. Это объекты `XText`. API-интерфейс LINQ to XML обрабатывает текстовые узлы настолько гладко, что о них легко забыть.

Для демонстрации второго прототипа используется тот же код, что и в первом примере, за исключением дополнительного указания имени, которому должны соответствовать элементы-потомки, чтобы второй прототип операции `Descendant` вернул их (листинг 8.12).

Листинг 8.12. Вызов второго прототипа `Descendants`

```
XDocument xDocument = new XDocument(
  new XElement("BookParticipants",
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
```

```
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));
IEnumerable<XElement> elements =
  xDocument.Element("BookParticipants").Elements("BookParticipant");
// Сначала отобразим исходные элементы.
foreach (XElement element in elements)
{
  Console.WriteLine("Исходный элемент: {0} : значение = {1}",
    element.Name, element.Value);
}
// Теперь отобразим элементы-потомки каждого исходного элемента.
foreach (XElement element in elements.Descendants("LastName"))
{
  Console.WriteLine("Элемент-потомок: {0}", element);
}
```

Результат этого примера выглядит следующим образом:

```
Исходный элемент: BookParticipant : значение = JoeRattz
Исходный элемент: BookParticipant : значение = EwanBuckingham
Элемент-потомок: <LastName>Rattz</LastName>
Элемент-потомок: <LastName>Buckingham</LastName>
```

Как и можно было ожидать, возвращены только элементы `LastName`.

DescendantsAndSelf

Операция `DescendantsAndSelf` может быть вызвана на последовательности элементов и возвращает последовательность, содержащую каждый исходный элемент и его потомков.

Прототипы

Операция `DescendantsAndSelf` имеет два прототипа, которые описаны ниже.

Первый прототип DescendantsAndSelf

```
public static IEnumerable<XElement> DescendantsAndSelf (
  this IEnumerable<XElement> source
)
```

Эта версия вызывается на последовательности элементов и возвращает последовательность элементов, содержащую каждый исходный элемент и его потомков.

Второй прототип DescendantsAndSelf

```
public static IEnumerable<XElement> DescendantsAndSelf (
  this IEnumerable<XElement> source,
  XName name
)
```

Эта версия подобна первой, но принимает дополнительный параметр, указывающий имя, которому должны соответствовать возвращаемые элементы.

Примеры

Для демонстрации работы первой версии `DescendantsAndSelf` используется тот же код, что и в примере применения первого прототипа операции `Descendants`, с заменой ее на операцию `DescendantsAndSelf`, как показано в листинге 8.13.

Листинг 8.13. Вызов первого прототипа `DescendantsAndSelf`

```
XDocument xDocument = new XDocument(
  new XElement("BookParticipants",
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));

IEnumerable<XElement> elements =
  xDocument.Element("BookParticipants").Elements("BookParticipant");

// Сначала отобразим исходные элементы.
foreach (XElement element in elements)
{
  Console.WriteLine("Исходный элемент: {0} : значение = {1}",
    element.Name, element.Value);
}

// Теперь отобразим элементы-потомки каждого исходного элемента.
foreach (XElement element in elements.DescendantsAndSelf())
{
  Console.WriteLine("Элемент-потомок: {0}", element);
}
```

После запуска этого примера должны отобразиться все исходные элементы и их элементы-потомки. Результаты показаны ниже:

```
Исходный элемент: BookParticipant : значение = JoeRattz
Исходный элемент: BookParticipant : значение = EwanBuckingham
Элемент-потомок: <BookParticipant type="Author">
  <!--This is a new author.-->
  <FirstName>Joe</FirstName>
  <LastName>Rattz</LastName>
</BookParticipant>
Элемент-потомок: <FirstName>Joe</FirstName>
Элемент-потомок: <LastName>Rattz</LastName>
Элемент-потомок: <BookParticipant type="Editor">
  <FirstName>Ewan</FirstName>
  <LastName>Buckingham</LastName>
</BookParticipant>
Элемент-потомок: <FirstName>Ewan</FirstName>
Элемент-потомок: <LastName>Buckingham</LastName>
```

Таким образом, вывод тот же, что и у операции `Descendants`, за исключением того, что он также включает сами исходные элементы — элементы `BookParticipant`. Пусть не вводит в заблуждение наличие комментария в результатах. Это не потому, что он был возвращен операцией `DescendantsAndSelf`, а потому, что отображается элемент `BookParticipant`, который был возвращен операцией.

Для демонстрации второго прототипа `DescendantsAndSelf` применяется тот же пример, что и для первого прототипа, но с добавлением имени, которому должны соответствовать возвращаемые элементы (листинг 8.14).

Листинг 8.14. Вызов второго прототипа DescendantsAndSelf

```
XDocument xDocument = new XDocument(
  new XElement("BookParticipants",
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));

IEnumerable<XElement> elements =
  xDocument.Element("BookParticipants").Elements("BookParticipant");

// Сначала отобразим исходные элементы.
foreach (XElement element in elements)
{
  Console.WriteLine("Исходный элемент: {0} : значение = {1}",
    element.Name, element.Value);
}

// Теперь отобразим элементы-потомки каждого исходного элемента.
foreach (XElement element in elements.DescendantsAndSelf("LastName"))
{
  Console.WriteLine("Элемент-потомок: {0}", element);
}
```

Результат этого примера выглядит так:

```
Исходный элемент: BookParticipant : значение = JoeRattz
Исходный элемент: BookParticipant : значение = EwanBuckingham
Элемент-потомок: <LastName>Rattz</LastName>
Элемент-потомок: <LastName>Buckingham</LastName>
```

Результат включает только элементы-потомки, соответствующие указанному имени. Здесь не слишком очевидно, что была вызвана операция DescendantsAndSelf, а не Descendants, поскольку исходные элементы не возвращены по причине несоответствия их имени указанному.

Опять-таки, как и со всеми операциями, которые возвращают элементы из множества уровней дерева XML, принимающих аргумент — имя, которому должны соответствовать возвращаемые элементы, вряд ли понадобятся AndSelf-версии операций, поскольку маловероятно, что будет много уровней одноименных операций.

Elements

Операция Elements может быть вызвана на последовательности элементов или документов и возвращает последовательность, содержащую дочерние элементы каждого исходного элемента или документа.

Эта операция отличается от Descendants, поскольку операция Elements возвращает только непосредственные дочерние элементы каждого из элементов входной последовательности, в то время как операция Descendants рекурсивно возвращает все дочерние элементы до достижения конца каждого дерева.

Прототипы

Операция Elements имеет два прототипа, описанные ниже.

Первый прототип `Elements`

```
public static IEnumerable<XElement> Elements<T> (
  this IEnumerable<T> source
) where T : XContainer
```

Эта версия вызывается на последовательности элементов или документов и возвращает последовательность, содержащую дочерние элементы каждого исходного элемента.

Операция отличается от метода `XContainer.Elements` в том, что этот метод вызывается на последовательности элементов или документов, а не на единственном элементе или документе.

Второй прототип `Elements`

```
public static IEnumerable<XElement> Elements<T> (
  this IEnumerable<T> source,
  XName name
) where T : XContainer
```

Эта версия подобна первой, за исключением дополнительного параметра, заставляющего операцию извлекать в выходную последовательность только те элементы, имя которых соответствует значению этого параметра.

Примеры

Дальнейшее можно предположить. Для демонстрации первого прототипа используется тот же базовый пример, что и для операции `DescendantsAndSelf`, за исключением того, что вместо нее вызывается операция `Elements`, как показано в листинге 8.15.

Листинг 8.15. Вызов первого прототипа `Elements`

```
XDocument xDocument = new XDocument(
  new XElement("BookParticipants",
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));
IEnumerable<XElement> elements =
  xDocument.Element("BookParticipants").Elements("BookParticipant");
// Сначала отобразим исходные элементы.
foreach (XElement element in elements)
{
  Console.WriteLine("Исходный элемент: {0} : значение = {1}",
    element.Name, element.Value);
}
// Теперь отобразим элементы-потомки каждого исходного элемента.
foreach (XElement element in elements.Elements())
{
  Console.WriteLine("Дочерний элемент: {0}", element);
}
```

Как и в предыдущих примерах, строится дерево XML, получается последовательность исходных элементов, отображается каждый элемент, для него извлекается последовательность дочерних элементов, которая затем отображается:

```
Исходный элемент: BookParticipant : значение = JoeRattz
Исходный элемент: BookParticipant : значение = EwanBuckingham
Дочерний элемент: <FirstName>Joe</FirstName>
Дочерний элемент: <LastName>Rattz</LastName>
Дочерний элемент: <FirstName>Ewan</FirstName>
Дочерний элемент: <LastName>Buckingham</LastName>
```

В этом примере возвращаются все дочерние элементы. Чтобы извлечь лишь элементы, соответствующие указанному имени, используется второй прототип операции `Elements` (листинг 8.16).

Листинг 8.16. Вызов второго прототипа `Elements`

```
XDocument xDocument = new XDocument(
  new XElement("BookParticipants",
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));

IEnumerable<XElement> elements =
  xDocument.Element("BookParticipants").Elements("BookParticipant");

// Сначала отобразим исходные элементы.
foreach (XElement element in elements)
{
  Console.WriteLine("Исходный элемент: {0} : значение = {1}",
    element.Name, element.Value);
}

// Теперь отобразим элементы-потомки каждого исходного элемента.
foreach (XElement element in elements.Elements("LastName"))
{
  Console.WriteLine("Дочерний элемент: {0}", element);
}
```

Теперь должны быть получены только элементы, соответствующие имени `LastName`:

```
Исходный элемент: BookParticipant : значение = JoeRattz
Исходный элемент: BookParticipant : значение = EwanBuckingham
Дочерний элемент: <LastName>Rattz</LastName>
Дочерний элемент: <LastName>Buckingham</LastName>
```

Все работает, как и следовало ожидать.

InDocumentOrder

Операция `InDocumentOrder` может быть вызвана на последовательности узлов и возвращает последовательность, содержащую дочерние узлы каждого исходного узла в порядке их следования в документе.

Прототипы

Операция `InDocumentOrder` имеет один прототип, который описан ниже.

Единственный прототип `InDocumentOrder`

```
public static IEnumerable<T> InDocumentOrder<T> (
  this IEnumerable<T> source
) where T : XNode
```

Эта версия вызывается на последовательности указанного типа, элементы которой должны быть узлами или типами-наследниками узлов, и возвращает последовательность того же типа, содержащую дочерние узлы исходных узлов в порядке их следования в документе.

Примеры

Эта довольно странная операция. В данном примере понадобится последовательность узлов. Поскольку в дополнение к элементам нужно видеть некоторые узлы, которые не являются элементами, будет построена последовательность узлов, дочерних по отношению к элементам `BookParticipant`. Это необходимо потому, что один из них имеет комментарий, который является узлом, но не элементом. Исходный код представлен в листинге 8.17.

Листинг 8.17. Вызов единственного прототипа `InDocumentOrder`

```
XDocument xDocument = new XDocument(
  new XElement("BookParticipants",
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));

IEnumerable<XElement> elements =
  xDocument.Element("BookParticipants").Elements("BookParticipant").
    Nodes().Reverse();

// Сначала отобразим все исходные узлы.
foreach (XNode node in nodes)
{
  Console.WriteLine("Исходный узел: {0}", node);
}

// Теперь отобразим дочерние узлы каждого исходного узла.
foreach (XNode node in nodes.InDocumentOrder())
{
  Console.WriteLine("Упорядоченный узел: {0}", node);
}
```

В коде строится дерево XML. Для построения исходной последовательности с помощью операции `Nodes` извлекаются дочерние узлы элемента `BookParticipant`, а затем вызывается стандартная операция запроса `Reverse`. Как говорилось во второй части этой книги, операция `Reverse` возвращает последовательность входных элементов, выстроенную в обратном порядке. Теперь имеется последовательность узлов, расположенных не в исходном порядке. Этот дополнительный шаг по изменению порядка предпринимается для того, чтобы при вызове операции `InDocumentOrder` было обнаружено отличие. Затем отображаются исходные узлы в неупорядоченном виде, вызывается операция `InDocumentOrder` и получается следующий результат:

```
Исходный узел: <LastName>Buckingham</LastName>
Исходный узел: <FirstName>Ewan</FirstName>
Исходный узел: <LastName>Rattz</LastName>
Исходный узел: <FirstName>Joe</FirstName>
Исходный узел: <!--This is a new author.-->
Упорядоченный узел: <!--This is a new author.-->
Упорядоченный узел: <FirstName>Joe</FirstName>
Упорядоченный узел: <LastName>Rattz</LastName>
Упорядоченный узел: <FirstName>Ewan</FirstName>
Упорядоченный узел: <LastName>Buckingham</LastName>
```

Как видите, исходные узлы расположены в порядке, противоположном тому, в котором они были построены в коде, а упорядоченные узлы оказываются опять в своем исходном порядке. Отлично, хотя и странно.

Nodes

Операция `Nodes` может быть вызвана на последовательности элементов или документов и возвращает последовательность узлов, содержащих дочерние узлы каждого исходного элемента или документа.

Эта операция отличается от операции `DescendantNodes` тем, что `Nodes` возвращает только непосредственных потомков — дочерние элементы каждого элемента входной последовательности, в то время как операция `DescendantNodes` рекурсивно возвращает все дочерние узлы, пока не достигнет конца дерева.

Прототипы

Операция `Nodes` имеет один прототип, описанный ниже.

Единственный прототип Nodes

```
public static IEnumerable<XNode> Nodes<T> (
  this IEnumerable<T> source
) where T : XContainer
```

Эта версия вызывается на последовательности элементов или документов и возвращает последовательность узлов, содержащих дочерние узлы всех исходных элементов.

В отличие от метода `XContainer.Nodes` этот метод вызывается на последовательности элементов или документов, а не на отдельном элементе или документе.

Примеры

В этом примере, приведенном в листинге 8.18, строится типичное дерево XML и исходная последовательность элементов `BookParticipant`, которые сразу отображаются. После чего извлекаются и отображаются дочерние узлы каждого исходного элемента.

Листинг 8.18. Вызов единственного прототипа Nodes

```
XDocument xDocument = new XDocument(
  new XElement("BookParticipants",
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));
```

```
IEnumerable<XElement> elements =
  xDocument.Element("BookParticipants").Elements("BookParticipant");

// Сначала отобразим все исходные элементы.
foreach (XElement element in elements)
{
  Console.WriteLine("Исходный элемент: {0} : значение = {1}",
    element.Name, element.Value);
}

// Теперь отобразим дочерние узлы каждого исходного элемента.
foreach (XNode node in elements.Nodes())
{
  Console.WriteLine("Дочерний узел: {0}", node);
}
```

Поскольку эта операция возвращает дочерние узлы, а не элементы, выходная последовательность будет включать в результате комментарий из первого элемента `BookParticipant`:

```
Исходный элемент: BookParticipant : значение = JoeRattz
Исходный элемент: BookParticipant : значение = EwanBuckingham
Дочерний узел: <!--This is a new author.-->
Дочерний узел: <FirstName>Joe</FirstName>
Дочерний узел: <LastName>Rattz</LastName>
Дочерний узел: <FirstName>Ewan</FirstName>
Дочерний узел: <LastName>Buckingham</LastName>
```

В результатах присутствуют все дочерние узлы каждого исходного элемента. Обратите внимание, что поскольку извлекаются только непосредственные дочерние узлы, узлы `XText`, являющиеся дочерними по отношению к каждому элементу `FirstName` и `LastName`, не были получены, как это было в примере операции `DescendantNodes`.

Remove

Операция `Remove` может быть вызвана на последовательности узлов или атрибутов для их удаления. Этот метод кэширует копию узлов или атрибутов в `List` для того, чтобы избежать “проблемы Хэллоуина”, описанной в предыдущей главе.

Прототипы

Операция `Remove` имеет два прототипа, описанные ниже.

Первый прототип Remove

```
public static void Remove (
  this IEnumerable<XAttribute> source
)
```

Эта версия вызывается на последовательности атрибутов и удаляет все атрибуты в исходной последовательности.

Второй прототип Remove

```
public static void Remove<T> (
  this IEnumerable<T> source
) where T : XNode
```

Эта версия вызывается на последовательности указанного типа, элементы которой должны быть узлами или типом, производным от узлов, и удаляет все узлы входной последовательности.

Примеры

Поскольку первый прототип предназначен для удаления атрибутов, понадобилась последовательность атрибутов. Поэтому будет построено стандартное дерево XML и извлечена последовательность атрибутов элементов `BookParticipant`. Сначала последовательность исходных атрибутов отображается, после чего на ней вызывается операция `Remove`. Затем для доказательства работоспособности отображается весь XML-документ с удаленными атрибутами. Код приведен в листинге 8.19.

Листинг 8.19. Вызов первого прототипа `Remove`

```
XDocument xDocument = new XDocument(
  new XElement("BookParticipants",
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));

IEnumerable<XElement> attributes =
  xDocument.Element("BookParticipants").Elements("BookParticipant").Attributes();

// Сначала отобразим все исходные атрибуты.
foreach (XAttribute attribute in attributes)
{
  Console.WriteLine("Исходный атрибут: {0} : значение = {1}",
    attribute.Name, attribute.Value);
}
attributes.Remove();

// Теперь отобразим XML-документ.
Console.WriteLine(xDocument);
```

Будет ли это работать? Давайте посмотрим:

```
Исходный атрибут: type : значение = Author
Исходный атрибут: type : значение = Editor
<BookParticipants>
  <BookParticipant>
    <!--This is a new author.-->
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
  <BookParticipant>
    <FirstName>Ewan</FirstName>
    <LastName>Buckingham</LastName>
  </BookParticipant>
</BookParticipants>
```

Пока все хорошо. Теперь попробуем применить второй прототип. Для этого примера вместо получения последовательности узлов и удаления их будет продемонстрировано нечто более интересное. Будет построена последовательность комментариев определенных элементов, и только эти комментарии будут удалены. Код показан в листинге 8.20.

Листинг 8.20. Вызов второго прототипа `Remove`

```
XDocument xDocument = new XDocument(
  new XElement("BookParticipants",
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),

    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));

IEnumerable<XElement> comments =
  xDocument.Element("BookParticipants").Elements("BookParticipant").
    Nodes().OfType<XComment>();

// Сначала отобразим все исходные комментарии.
foreach (XComment comment in comments)
{
  Console.WriteLine("Исходный комментарий: {0}", comment);
}

comments.Remove();

// Теперь отобразим XML-документ.
Console.WriteLine(xDocument);
```

В данном примере при построении исходной последовательности извлекаются дочерние узлы каждого элемента `BookParticipant`. Можно было бы просто вызвать операцию `Remove` на этой последовательности, и тогда все дочерние узлы каждого элемента `BookParticipant` исчезли бы. Но вместо этого вызывается стандартная операция запроса `OfType`. Из второй части книги вы должны помнить, что упомянутая операция вернет только те объекты входной последовательности, которые соответствуют указанному типу. Вызывая операцию `OfType` и указывая тип `XComment`, получается последовательность, состоящая только из комментариев — `comments`. Затем на `comments` вызывается метод `Remove`. В результате исходный документ будет очищен от всех комментариев:

```
Исходный комментарий: <!--This is a new author.-->
<BookParticipants>
  <BookParticipant type="Author">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>

  <BookParticipant type="Editor">
    <FirstName>Ewan</FirstName>
    <LastName>Buckingham</LastName>
  </BookParticipant>
</BookParticipants>
```

Код работает весьма неплохо. Оцените, насколько удобна операция `OfType`, и как она интегрируется в запрос LINQ to XML. Похоже, она может оказаться весьма полезной во многих ситуациях.

Резюме

В предыдущей главе был описан API-интерфейс LINQ to XML, позволяющий создавать, модифицировать, сохранять и загружать деревья XML. Обратите внимание — *деревья*, а не *документы*, потому что в LINQ to XML документы не обязательны. Там же было показано, как запросить один узел или элемент на предмет узлов и элементов, иерархически связанных с ним. В настоящей главе было продемонстрировано, каким образом то же самое делается с последовательностями узлов или элементов с применением операций LINQ to XML. Теперь способы выполнения элементарных запросов по деревьям XML, используя LINQ to XML, должны быть достаточно ясны. Без сомнений, этот новый API-интерфейс для работы с XML докажет свою способность удобно запрашивать данные XML. В частности, возможность комбинирования стандартных операций запросов с операциями LINQ to XML позволяет строить элегантные и мощные запросы.

К данному моменту было описано почти все, что нужно знать о построении блоков, необходимых для выполнения запросов LINQ to XML. В следующей главе будут представлены несколько более сложные запросы и рассказано об остальных предметах первой необходимости XML, таких как проверка достоверности и трансформация.

ГЛАВА 9

Дополнительные возможности LINQ to XML

В предыдущих двух главах было показано, как создавать, модифицировать и обходить данные XML с использованием API-интерфейса LINQ to XML. Также были описаны строительные блоки, из которых можно создавать мощные запросы XML. В итоге LINQ to XML покрывает 90% ваших потребностей, но как насчет оставшихся 10%? Посмотрим, можно ли еще повысить этот процент. Если в Microsoft добавили возможности проверки достоверности схемы, трансформации и запросы `XPath`, то какой процент реальных случаев теперь можно покрыть?

Несмотря на то что был описан новый API-интерфейс LINQ to XML, а также способы выполнения самых основных запросов с его помощью, все же осталось продемонстрировать несколько более сложные запросы, потребность в которых может возникнуть на практике. В этой главе будет приведен ряд примеров, которые, как можно надеяться, сделают задачу опроса данных XML с LINQ to XML тривиальной, включая применение синтаксиса выражений запросов для тех, кто отдает ему предпочтение.

Вдобавок описание нового API-интерфейса LINQ to XML будет неполным без нескольких дополнительных возможностей, таких как трансформация и проверка достоверности. В настоящей главе рассматриваются эти дополнения к LINQ to XML, наряду с другой полезной информацией.

В частности, будет показано, как выполнять трансформации с помощью языка XSLT и без него, а также как проверить документ XML на предмет соответствия схеме. Вдобавок будет даже представлен пример выполнения запроса в стиле XPath.

Необходимые пространства имен

В примерах этой главы будут использоваться пространства имен `System.Xml`, `System.Xml.Schema`, `System.Xml.Xsl` и `System.Xml.XPath`, в дополнение к обычным пространствам имен LINQ to XML — `System.Linq` и `System.Xml.Linq`. Поэтому понадобится добавить следующие директивы `using`, если это не было сделано раньше:

```
using System.Linq;
using System.Xml;
using System.Xml.Linq;
using System.Xml.Schema;
using System.Xml.XPath;
using System.Xml.Xsl;
```

Запросы

В предыдущих главах, посвященных LINQ to XML, были продемонстрированы основные принципы, необходимые для выполнения запросов XML с использованием LINQ to XML. Однако большая часть примеров была специально спроектирована для демонстрации операции или свойства. В настоящем разделе представлены примеры, которые более ориентированы на решение определенной задачи.

Отсутствие иерархического спуска

Во многих примерах, представленных в предыдущих главах, для получения ссылки на определенный элемент осуществлялся спуск вниз по иерархии посредством рекурсивного вызова операций `Element` и `Elements` до тех пор, пока не будет достигнут нужный элемент.

Многие примеры содержали следующие строки кода:

```
IEnumerable<XElement> elements =
  xDocument.Element("BookParticipants").Elements("BookParticipant");
```

Здесь спуск начинается с уровня документа, затем получается его дочерний элемент по имени `BookParticipants`, после чего — его дочерний элемент по имени `BookParticipant`. Однако вовсе не обязательно спускаться по иерархии вниз подобным способом. Вместо этого можно было бы просто написать код, подобный показанному в листинге 9.1.

Листинг 9.1. Получение элементов без иерархического спуска

```
XDocument xDocument = new XDocument(
  new XElement("BookParticipants",
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));
IEnumerable<XElement> elements = xDocument.Descendants("BookParticipant");
foreach (XElement element in elements)
{
  Console.WriteLine("Элемент: {0} : значение = {1}",
    element.Name, element.Value);
}
```

В этом примере извлекаются все элементы-потомки документа по имени `BookParticipant`. Поскольку спуск по определенной ветви дерева XML не производится, необходимо знать схему, поскольку есть вероятность получить элементы не из той ветви, которая нужна. Однако во многих случаях, включая данный, этот подход работает хорошо. Вот результат:

```
Элемент: BookParticipant : значение = JoeRattz
Элемент: BookParticipant : значение = EwanBuckingham
```

Но предположим, что интересуют не все элементы `BookParticipant`, т.е. требуется как-то ограничить возвращаемые элементы. В листинге 9.2 представлен пример возврата только элементов, у которых значение дочернего элемента `FirstName` равно `"Ewan"`.

Листинг 9.2. Получение ограниченных элементов без иерархического спуска

```
XDocument xDocument = new XDocument(
  new XElement("BookParticipants",
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));

IEnumerable<XElement> elements = xDocument
  .Descendants("BookParticipant")
  .Where(e => ((string)e.Element("FirstName")) == "Ewan");

foreach (XElement element in elements)
{
  Console.WriteLine("Элемент: {0} : значение = {1}",
    element.Name, element.Value);
}
```

На этот раз добавлен вызов операции `Where`. Обратите внимание на приведение элемента `FirstName` к типу `string`. Это нужно, чтобы получить его значение для сравнения с `"Ewan"`. Результат показан ниже:

```
Элемент: BookParticipant : значение = EwanBuckingham
```

Конечно, иногда необходимо контролировать порядок. На этот раз, чтобы обеспечить возврат более одного элемента, когда порядок будет иметь значение, изменим лямбда-выражение операции `Where` для возврата обоих элементов. Чтобы сделать пример интереснее, запрос выполняется по атрибуту `type`, и будет переписан с использованием синтаксиса выражений запросов, как показано в листинге 9.3.

Листинг 9.3. Получение ограниченных элементов без иерархического спуска с упорядочиванием и использованием синтаксиса выражений запросов

```
XDocument xDocument = new XDocument(
  new XElement("BookParticipants",
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));

IEnumerable<XElement> elements =
  from e in xDocument.Descendants("BookParticipant")
  where ((string)e.Attribute("type")) != "Illustrator"
  orderby ((string)e.Element("LastName"))
  select e;

foreach (XElement element in elements)
{
  Console.WriteLine("Элемент: {0} : значение = {1}",
    element.Name, element.Value);
}
```

В этом примере по-прежнему запрашиваются элементы BookParticipant документа, но только те, чей атрибут type отличается от Illustrator. В данном случае это будут все элементы BookParticipant. Затем они упорядочиваются по значению элемента LastName. Обратите внимание, что для получения значений атрибута type и элемента LastName выполняется их приведение к string. Вот результат:

```
Элемент: BookParticipant : значение = EwanBuckingham
Элемент: BookParticipant : значение = JoeRattz
```

Сложный запрос

До сих пор все примеры запросов были тривиальными, поэтому прежде чем закрыть тему запросов, следует рассмотреть пример более сложного запроса. В этом примере используются те же данные, которые рекомендованы W3C специально для тестирования прецедентов использования запросов XML.

Пример в листинге 9.4 содержит данные из трех разных XML-документов. Каждый документ создается за счет разбора текстового представления каждого из рекомендованных W3C документов XML. Поскольку пример сложен, он сопровождается объяснениями.

Первый шаг состоит в создании документов из XML.

Листинг 9.4. Сложный запрос, объединяющий три документа с помощью синтаксиса выражений запросов

```
XDocument users = XDocument.Parse(
  @"<users>
    <user_tuple>
      <userid>U01</userid>
      <name>Tom Jones</name>
      <rating>B</rating>
    </user_tuple>
    <user_tuple>
      <userid>U02</userid>
      <name>Mary Doe</name>
      <rating>A</rating>
    </user_tuple>
    <user_tuple>
      <userid>U03</userid>
      <name>Dee Linquent</name>
      <rating>D</rating>
    </user_tuple>
    <user_tuple>
      <userid>U04</userid>
      <name>Roger Smith</name>
      <rating>C</rating>
    </user_tuple>
    <user_tuple>
      <userid>U05</userid>
      <name>Jack Sprat</name>
      <rating>B</rating>
    </user_tuple>
    <user_tuple>
      <userid>U06</userid>
      <name>Rip Van Winkle</name>
      <rating>B</rating>
    </user_tuple>
    </users>");
```

```
XDocument items = XDocument.Parse(
  @"<items>
    <item_tuple>
      <itemno>1001</itemno>
      <description>Red Bicycle</description>
      <offered_by>U01</offered_by>
      <start_date>1999-01-05</start_date>
      <end_date>1999-01-20</end_date>
      <reserve_price>40</reserve_price>
    </item_tuple>
    <item_tuple>
      <itemno>1002</itemno>
      <description>Motorcycle</description>
      <offered_by>U02</offered_by>
      <start_date>1999-02-11</start_date>
      <end_date>1999-03-15</end_date>
      <reserve_price>500</reserve_price>
    </item_tuple>
    <item_tuple>
      <itemno>1003</itemno>
      <description>Old Bicycle</description>
      <offered_by>U02</offered_by>
      <start_date>1999-01-10</start_date>
      <end_date>1999-02-20</end_date>
      <reserve_price>25</reserve_price>
    </item_tuple>
    <item_tuple>
      <itemno>1004</itemno>
      <description>Tricycle</description>
      <offered_by>U01</offered_by>
      <start_date>1999-02-25</start_date>
      <end_date>1999-03-08</end_date>
      <reserve_price>15</reserve_price>
    </item_tuple>
    <item_tuple>
      <itemno>1005</itemno>
      <description>Tennis Racket</description>
      <offered_by>U03</offered_by>
      <start_date>1999-03-19</start_date>
      <end_date>1999-04-30</end_date>
      <reserve_price>20</reserve_price>
    </item_tuple>
    <item_tuple>
      <itemno>1006</itemno>
      <description>Helicopter</description>
      <offered_by>U03</offered_by>
      <start_date>1999-05-05</start_date>
      <end_date>1999-05-25</end_date>
      <reserve_price>50000</reserve_price>
    </item_tuple>
    <item_tuple>
      <itemno>1007</itemno>
      <description>Racing Bicycle</description>
      <offered_by>U04</offered_by>
      <start_date>1999-01-20</start_date>
      <end_date>1999-02-20</end_date>
      <reserve_price>200</reserve_price>
    </item_tuple>
```

```
    <item_tuple>
      <itemno>1008</itemno>
      <description>Broken Bicycle</description>
      <offered_by>U01</offered_by>
      <start_date>1999-02-05</start_date>
      <end_date>1999-03-06</end_date>
      <reserve_price>25</reserve_price>
    </item_tuple>
    </items>");
XDocument bids = XDocument.Parse(
  @"<bids>
    <bid_tuple>
      <userid>U02</userid>
      <itemno>1001</itemno>
      <bid>35</bid>
      <bid_date>1999-01-07</bid_date>
    </bid_tuple>
    <bid_tuple>
      <userid>U04</userid>
      <itemno>1001</itemno>
      <bid>40</bid>
      <bid_date>1999-01-08</bid_date>
    </bid_tuple>
    <bid_tuple>
      <userid>U02</userid>
      <itemno>1001</itemno>
      <bid>45</bid>
      <bid_date>1999-01-11</bid_date>
    </bid_tuple>
    <bid_tuple>
      <userid>U04</userid>
      <itemno>1001</itemno>
      <bid>50</bid>
      <bid_date>1999-01-13</bid_date>
    </bid_tuple>
    <bid_tuple>
      <userid>U02</userid>
      <itemno>1001</itemno>
      <bid>55</bid>
      <bid_date>1999-01-15</bid_date>
    </bid_tuple>
    <bid_tuple>
      <userid>U01</userid>
      <itemno>1002</itemno>
      <bid>400</bid>
      <bid_date>1999-02-14</bid_date>
    </bid_tuple>
    <bid_tuple>
      <userid>U02</userid>
      <itemno>1002</itemno>
      <bid>600</bid>
      <bid_date>1999-02-16</bid_date>
    </bid_tuple>
    <bid_tuple>
      <userid>U03</userid>
      <itemno>1002</itemno>
      <bid>800</bid>
      <bid_date>1999-02-17</bid_date>
    </bid_tuple>
```

```
  <bid_tuple>
    <userid>U04</userid>
    <itemno>1002</itemno>
    <bid>1000</bid>
    <bid_date>1999-02-25</bid_date>
  </bid_tuple>
  <bid_tuple>
    <userid>U02</userid>
    <itemno>1002</itemno>
    <bid>1200</bid>
    <bid_date>1999-03-02</bid_date>
  </bid_tuple>
  <bid_tuple>
    <userid>U04</userid>
    <itemno>1003</itemno>
    <bid>15</bid>
    <bid_date>1999-01-22</bid_date>
  </bid_tuple>
  <bid_tuple>
    <userid>U05</userid>
    <itemno>1003</itemno>
    <bid>20</bid>
    <bid_date>1999-02-03</bid_date>
  </bid_tuple>
  <bid_tuple>
    <userid>U01</userid>
    <itemno>1004</itemno>
    <bid>40</bid>
    <bid_date>1999-03-05</bid_date>
  </bid_tuple>
  <bid_tuple>
    <userid>U03</userid>
    <itemno>1007</itemno>
    <bid>175</bid>
    <bid_date>1999-01-25</bid_date>
  </bid_tuple>
  <bid_tuple>
    <userid>U05</userid>
    <itemno>1007</itemno>
    <bid>200</bid>
    <bid_date>1999-02-08</bid_date>
  </bid_tuple>
  <bid_tuple>
    <userid>U04</userid>
    <itemno>1007</itemno>
    <bid>225</bid>
    <bid_date>1999-02-12</bid_date>
  </bid_tuple>
 </bids>");
```

Этот пример данных в основном предназначен для представления на сайте Интернет-аукциона. Здесь просто были созданы три документа XML вызовом метода `XDocument.Parse` на строковом представлении данных XML. Документы предназначены для пользователей, товаров и предложенных цен.

Запрос должен производить список всех цен, превышающих $50. В результатах должна присутствовать дата и цена, наряду с пользователем, предложившим ее, а также наименование товара и его описание.

Вот как выглядит запрос:

```
var biddata = from b in bids.Descendants("bid_tuple")
            where ((double)b.Element("bid")) > 50
            join u in users.Descendants("user_tuple")
            on ((string)b.Element("userid")) equals
              ((string)u.Element("userid"))
            join i in items.Descendants("item_tuple")
            on ((string)b.Element("itemno")) equals
              ((string)i.Element("itemno"))
            select new { Item = ((string)b.Element("itemno")),
                         Description = ((string)i.Element("description")),
                         User = ((string)u.Element("name")),
                         Date = ((string)b.Element("bid_date")),
                         Price = ((double)b.Element("bid"))};
```

Как видите, это сложный запрос. Первый шаг состоит в запрашивании потомков по имени `bid_tuple` в документе `bids`, используя метод `Descendants`. Затем выполняется конструкция `where` для элементов, имеющих дочерние элементы по имени `bid`, чье значение превышает `50`. Таким образом, извлекаются предложения цены, превышающие $50. Такое скорое выполнение `where` в запросе может показаться несколько необычным. В действительности это можно было сделать в запросе позжже — непосредственно перед вызовом конструкции `select`. Однако это означало бы необходимость извлечения и соединения всех записей из XML-документов пользователей и товаров, для которых предложения цены не превышают $50, что явно излишне. За счет как можно более ранней фильтрации результирующего набора сокращается рабочая нагрузка на остальную часть запроса, что повышает производительность.

Отфильтровав рабочий набор по предложениям цены, превышающим $50, выполняется соединение (join) этих предложений с XML-документом пользователей через одноименный в двух наборах элемент `userid`, чтобы получить имя пользователя. В этой точке есть предложения и связанные с ними пользователи, предложившие цену свыше $50.

Затем результат соединяется с XML-документом наименований товара через одноименный элемент `itemno`, чтобы получить описание товара. Таким образом, появляется соединение предложенной цены, пользователей и товаров.

Снова обратите внимание на необходимость выполнять приведение всех элементов к интересующему типу данных, чтобы получить стоимость товара. Особенно интересен тот факт, что предложенная цена получается за счет приведения элемента `bid` к типу `double`. Несмотря на то что входное значение предложенной цены является строкой, поскольку оно может быть успешно преобразовано в `double`, то есть возможность выполнить приведение к `double` и получить значение цены в числовом виде двойной точности. Здорово, не правда ли?

Следующий шаг состоит в выборе анонимного класса, содержащего информацию из дочерних элементов соединенных элементов.

Сначала отобразим заголовок:

```
Console.WriteLine("{0,-12} {1,-12} {2,-6} {3,-14} {4,10}",
  "Date",
  "User",
  "Item",
  "Description",
  "Price");
Console.WriteLine("==========================================================");
```

Здесь нет ничего особенного. Все, что осталось — выполнить перечисление результирующей последовательности и отобразить каждое предложение цены:

```
foreach (var bd in biddata)
{
 Console.WriteLine("{0,-12} {1,-12} {2,-6} {3,-14} {4,10:C}",
   bd.Date,
   bd.User,
   bd.Item,
   bd.Description,
   bd.Price);
}
```

Эта часть тривиальна. В действительности здесь все тривиально, кроме самого запроса. Давайте посмотрим на результат:

```
Date           User            Item     Description          Price
=================================================================
1999-01-15     Mary Doe        1001     Red Bicycle         $55.00
1999-02-14     Tom Jones       1002     Motorcycle         $400.00
1999-02-16     Mary Doe        1002     Motorcycle         $600.00
1999-02-17     Dee Linquent    1002     Motorcycle         $800.00
1999-02-25     Roger Smith     1002     Motorcycle       $1,000.00
1999-03-02     Mary Doe        1002     Motorcycle       $1,200.00
1999-01-25     Dee Linquent    1007     Racing Bicycle     $175.00
1999-02-08     Jack Sprat      1007     Racing Bicycle     $200.00
1999-02-12     Roger Smith     1007     Racing Bicycle     $225.00
```

Как видите, три XML-документа были соединены в одном запросе.

Определенно, теперь вы увидели мощь LINQ to XML. Однако это еще не все.

Трансформации

С помощью LINQ to XML можно выполнять трансформации XML, используя для этого два совершенно разных подхода. Первый подход заключается в применении языка XSLT (eXtensible Stylesheet Language Transformations) через классы-мосты — `XmlReader` и `XmlWriter`. Второй подход предусматривает использование для трансформаций самого API-интерфейса LINQ to XML за счет функционального конструирования целевого документа XML и встраивания запроса LINQ to XML в некоторый документ XML.

Применение XSLT обладает тем преимуществом, что это — стандартная технология XML. Существует инструментарий, который помогает в написании, отладке и тестировании трансформаций XSLT. Вдобавок, поскольку он уже существует, доступны готовые XSLT-документы, которые можно использовать в новом коде с применением LINQ to XML. Существует множество доступных XSLT-документов, из которых можно выбрать нужные. К тому же, использование XSLT для трансформаций наиболее динамично. В отличие от применения подхода на основе функционального конструирования LINQ to XML, повторно компилировать код для изменения трансформации не понадобится. Простое изменение документа XSLT позволяет модифицировать трансформации во время выполнения. И, наконец, XSLT — широко известная технология, и есть немало знающих ее разработчиков, которые могут оказать помощь. Что касается подхода на основе функционального конструирования, то пока, на начальном этапе существования LINQ, это еще не так.

Применение подхода с функциональным конструированием не потребует больших затрат. Оно позволит выполнять трансформации XML, не зная ничего помимо LINQ to XML. Поэтому если вы не знакомы с XSLT, а потребности в трансформации скромны, этот подход может оказаться более подходящим. К тому же, хотя функциональное конструирование менее удобно, чем простая модификация документа XSLT, необходимость перекомпиляции кода для модификации трансформаций может добавить безопасности.

Никто не сможет извне вмешаться и модифицировать трансформацию. Таким образом, для тех случаев, когда вы думаете, что нарушаете ограничения, используя закон Сарбейнса-Оксли[1] в качестве оправдания своего бездействия, упирайте на тот факт, что невозможно просто изменить трансформацию без изменения кода. Или же, если вы работаете в области здравоохранения и не считаете, что можете избежать ответственности за еще одно нарушение HIPAA (закон США о преемственности страхования и отчетности в области здравоохранения), то трансформация функциональным конструированием может послужить оправданием, если вас обвинят в нерасторопности при удовлетворении потребностей заказчика.

Трансформации с использованием XSLT

Чтобы выполнить трансформацию с использованием XSLT, необходимо обратиться к классам-мостам `XmlWriter` и `XmlReader`, экземпляры которых получаются из методов `CreateWriter` и `CreateReader` классов `XDocument`.

Пример, показанный в листинге 9.5, требует некоторых пояснений, которые будут приводиться по мере продвижения по коду. Для начала определяется таблица стилей трансформации.

Листинг 9.5. Трансформация документа XML с использованием XSLT

```
string xsl =
  @"<xsl:stylesheet version='1.0' xmlns:xsl='http://www.w3.org/1999/XSL/Transform'>
  <xsl:template match='//BookParticipants'>
    <html>
      <body>
      <h1>Book Participants</h1>
      <table>
        <tr align='left'>
          <th>Role</th>
          <th>First Name</th>
          <th>Last Name</th>
        </tr>
        <xsl:apply-templates></xsl:apply-templates>
      </table>
      </body>
    </html>
  </xsl:template>
  <xsl:template match='BookParticipant'>
    <tr>
      <td><xsl:value-of select='@type'/></td>
      <td><xsl:value-of select='FirstName'/></td>
      <td><xsl:value-of select='LastName'/></td>
    </tr>
  </xsl:template>
</xsl:stylesheet>";
```

Здесь нет ничего особо выдающегося. Просто указывается XSL-разметка для создания некоторой HTML-разметки, предназначенной для отображения типичного XML-

[1] Закон Сарбейнса-Оксли, принятый в 2002 г. (Sarbanes-Oxley Act of 2002) в США, касается дополнений в закон “О фондовых биржах” 1934 г. В соответствие с ним, в обязанность компаний, акции которых котируются на фондовых биржах, регулируемых Комиссией по ценным бумагам и биржам, входит ведение финансовой отчетности согласно общепринятым принципам бухгалтерского учета.

документа с участниками издания книги в виде HTML-таблицы. Далее создается XML-документ с участниками книги:

```
XDocument xDocument = new XDocument(
  new XElement("BookParticipants",
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));
```

Это просто обычный XML-документ. А теперь о том, где начинается “магия”. Необходимо создать новый `XDocument` для трансформированной версии. Затем из этого документа будет построен `XmlWriter`, создан экземпляр объекта `XslCompiledTransform`, загружен объект трансформации с помощью таблицы стилей трансформации и выполнена трансформация входного XML-документа в выходной `XmlWriter`:

```
XDocument transformedDoc = new XDocument();
using (XmlWriter writer = transformedDoc.CreateWriter())
{
  XslCompiledTransform transform = new XslCompiledTransform();
  transform.Load(XmlReader.Create(new StringReader(xsl)));
  transform.Transform(xDocument.CreateReader(), writer);
}
Console.WriteLine(transformedDoc);
```

Разумеется, после всего этого трансформированная версия документа отображается. Как видите, для выполнения трансформации используются оба класса-моста — `XmlWriter` и `XmlReader`. Ниже показан результат:

```
<html>
  <body>
    <h1>Book Participants</h1>
    <table>
      <tr align="left">
        <th>Role</th>
        <th>First Name</th>
        <th>Last Name</th>
      </tr>
      <tr>
        <td>Author</td>
        <td>Joe</td>
        <td>Rattz</td>
      </tr>
      <tr>
        <td>Editor</td>
        <td>Ewan</td>
        <td>Buckingham</td>
      </tr>
    </table>
  </body>
</html>
```

Трансформация с использованием функционального конструирования

Хотя LINQ to XML поддерживает трансформацию XSLT, существуют очень эффективные способы выполнения трансформаций на основе самого API-интерфейса LINQ to XML.

Логически рассуждая, трансформация может быть сведена к комбинации функционально сконструированного дерева XML с встроенным запросом XML.

Совет. Для выполнения трансформации комбинируйте функциональное конструирование со встроенным запросом XML LINQ.

XML-трансформации лучше всего объяснить на примере. Во многих примерах, которые приводились в главах, посвященных LINQ to XML, работа осуществлялась со следующим деревом XML:

```
<BookParticipants>
  <BookParticipant type="Author">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
  <BookParticipant type="Editor">
    <FirstName>Ewan</FirstName>
    <LastName>Buckingham</LastName>
  </BookParticipant>
</BookParticipants>
```

Предположим, что это дерево XML нужно трансформировать в следующее:

```
<MediaParticipants type="book">
  <Participant Role="Author" Name="Joe Rattz" />
  <Participant Role="Editor" Name="Ewan Buckingham" / >
</MediaParticipants>
```

Для проведения этой трансформации будет использоваться функциональное конструирование со встроенным запросом. С помощью этого подхода будет функционально сконструирован новый документ, соответствующий желаемой выходной древовидной структуре XML, с получением необходимых для этого данных из исходного документа XML с помощью запроса LINQ to XML. Функциональное конструирование и логика запроса определяются желаемой выходной древовидной структурой.

Поскольку эта задача несколько сложнее многих предыдущих примеров LINQ to XML, ее решение будет объясняться шаг за шагом. Код представлен в листинге 9.6.

Листинг 9.6. Трансформация документа XML

```
XDocument xDocument = new XDocument(
  new XElement("BookParticipants",
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));
Console.WriteLine("Исходный XML-документ:");
Console.WriteLine("{0}{1}{1}", xDocument, System.Environment.NewLine);
```

Приведенный код просто создает исходный XML-документ, который будет трансформироваться и отображаться. Затем нужно построить новый документ и корневой элемент:

```
XDocument xTransDocument = new XDocument(
  new XElement("MediaParticipants",
```

Вспомните, что функциональное конструирование определяется желаемой структурой выходного XML-дерева. В этой точке имеется документ и корневой элемент — MediaParticipants. Теперь понадобится добавить атрибут type к корневому элементу:

```
new XAttribute("type", "book"),
```

Атрибут type и его значение в исходном документе XML не существует. Он должен быть жестко закодирован или наоборот — сконфигурирован, в программной логике, что безопасно, поскольку уже известно, что этот код предназначен для описания книг; в противном случае он не должен вызываться.

Теперь есть атрибут type элемента MediaParticipants. Затем необходимо сгенерировать элемент Participant для каждого элемента BookParticipant исходного XML. Чтобы сделать это, в исходном XML-документе запрашиваются элементы BookParticipant:

```
xDocument.Element("BookParticipants")
  .Elements("BookParticipant")
```

Теперь есть возвращенная последовательность элементов BookParticipant. Далее потребуется сгенерировать элемент Participant для каждого элемента BookParticipant и заполнить его атрибуты. Для этого используется проекция через операцию Select, чтобы сконструировать элементы Participant:

```
.Select(e => new XElement("Participant",
```

Далее конструируются два атрибута — Role и Name — для элемента Participant, с получением их значений из элемента BookParticipant:

```
new XAttribute("Role", (string)e.Attribute("type")),
new XAttribute("Name", (string)e.Element("FirstName") + " " +
  (string)e.Element("LastName"))))));
```

И, наконец, трансформированный XML-документ отображается:

```
Console.WriteLine("Трансформированный XML-документ:");
Console.WriteLine(xTransDocument);
```

Давайте посмотрим, получено ли в результате то, что было нужно:

```
Исходный XML-документ:
<BookParticipants>
  <BookParticipant type="Author">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
  <BookParticipant type="Editor">
    <FirstName>Ewan</FirstName>
    <LastName>Buckingham</LastName>
  </BookParticipant>
</BookParticipants>

Трансформированный XML-документ:
<MediaParticipants type="book">
  <Participant Role="Author" Name="Joe Rattz" />
  <Participant Role="Editor" Name="Ewan Buckingham" />
</MediaParticipants>
```

Все прошло блестяще! Вывод соответствует ожиданиям. Неплохо, если учесть, что не использовалось ничего помимо LINQ to XML.

Советы

Ниже дано несколько советов относительно выполнения трансформаций XML посредством LINQ to XML. Возможно, вы в них не нуждаетесь, но нет причин им не следовать.

Упрощайте решение сложных задач с помощью вспомогательных методов

Не существует требования, чтобы каждый фрагмент кода, необходимый для проведения трансформации или запроса, находился в самом коде трансформации. Можно создавать вспомогательные методы, которые возьмут на себя выполнение наиболее сложной части рутинной работы.

Ниже приведен пример кода, демонстрирующего создание вспомогательного метода для разбиения на части более сложной задачи.

Вспомогательный метод для трансформации документа XML

```
static IEnumerable<XElement> Helper()
{
  XElement[] elements = new XElement[] {
    new XElement("Element", "A"),
    new XElement("Element", "B")};
  return(elements);
}
```

Код в листинге 9.7 начинается с конструирования дерева XML. В самом вызове конструктора создается корневой узел по имени `RootElement`. Чтобы создать дочерние узлы, он вызывает вспомогательный метод по имени `Helper`. Не важно, что именно делает этот вспомогательный метод; важно лишь то, что он помогает построить некоторую часть дерева XML, и что вызов этого метода может быть встроен в функциональное конструирование дерева XML.

Листинг 9.7. Использование вспомогательного метода для трансформации документа XML

```
XElement xElement = new XElement("RootElement", Helper());
Console.WriteLine(xElement);
```

Вот результирующий код:

```
<RootElement>
  <Element>A</Element>
  <Element>B</Element>
</RootElement>
```

В главе 7 говорилось, что конструктор `XElement` знает, как справиться с `IEnumerable<T>`, который является типом возврата метода `Helper`. Здорово, правда?

Подавляйте конструирование узла с помощью `null`

Может случиться, что по той или иной причине требуется подавить конструирование некоторых узлов. Возможно, в источнике информации недостает некоторых данных, что вызовет желание пропустить элемент при создании дерева, или же данные окажутся такими, что вы предпочтете пропустить их.

В разделе “Создание элементов с помощью `XElement`” главы 7, при описании конструктора `XElement`, упоминалось, что можно передавать `null` в качестве значения объекта для содержимого элемента, и это может быть удобно при выполнении трансформаций. Подавление конструирования узлов — именно тот случай.

Для примера сначала построим последовательность элементов. Затем приступим к конструированию нового дерева XML на основе этой последовательности. Если значением входного элемента является "A", то создавать выходной элемент для него не нужно. Для этого будет передаваться значение null. Необходимый код приведен в листинге 9.8.

Листинг 9.8. Подавление конструирования узла с помощью null

```
IEnumerable<XElement> elements =
  new XElement[] {
    new XElement("Element", "A"),
    new XElement("Element", "B")};
XElement xElement = new XElement("RootElement",
  elements.Select(e => (string)e != "A" ? new XElement(e.Name, (string)e) : null));
Console.WriteLine(xElement);
```

Как видите, в этом коде строится входная последовательность элементов. Затем конструируется корневой элемент и выполняется перечисление входной последовательности. После этого с применением операции Select, пока значение входного элемента не равно "A", конструируется объект XElement с использованием для этого входного элемента. Если же значение входного элемента эквивалентно "A", возвращается null. Конструктор элемента XElement знает, как обращаться с null: он игнорирует его. В результате любой элемент, чье значение эквивалентно "A", исключается из выходного дерева XML. В коде применяется новое средство извлечения значения узла за счет приведения элемента e к string в лямбда-выражении операции Select.

Ниже показан результат:

```
<RootElement>
  <Element>B</Element>
</RootElement>
```

Обратите внимание, что элемент "A" отсутствует. Конечно, есть и другие способы реализовать ту же логику, не предусматривающие использование null. Например, можно было бы применить операцию where и отфильтровать элементы со значением "A". Но здесь нужно было продемонстрировать эффект от применения null на очень простом примере.

Существуют другие пути использования той же концепции. Предположим, что требуется сгенерировать некоторый XML, который позволял бы иметь пустой элемент в экземплярах, существование которых нежелательно. Рассмотрим код из листинга 9.9.

Листинг 9.9. Пример генерации пустого элемента

```
IEnumerable<XElement> elements =
  new XElement[] {
    new XElement("BookParticipant",
      new XElement("Name", "Joe Rattz"),
      new XElement("Book", "Pro LINQ: Language Integrated Query in C# 2008")),
    new XElement("BookParticipant",
      new XElement("Name", "John Q. Public"))};
XElement xElement =
  new XElement("BookParticipants",
    elements.Select(e =>
      new XElement(e.Name,
        new XElement(e.Element("Name").Name, e.Element("Name").Value),
        new XElement("Books", e.Elements("Book")))));
Console.WriteLine(xElement);
```

В первом операторе приведенного кода генерируется последовательность элементов BookParticipant, состоящая из двух элементов. Обратите внимание, что некоторые из элементов BookParticipant имеют дочерние элементы Book, такие как BookParticipant, у которого дочерний элемент Name имеет значение "Joe Rattz", а некоторые не имеют элементов Book — вроде BookParticipant, у которого дочерний элемент Name имеет значение "John Q. Public".

Во втором операторе строится дерево XML, используя полученную последовательность элементов. В дереве XML создается элемент с тем же именем, что и в исходной последовательности, которым будет BookParticipant. Затем создается его дочерний элемент name и список Book для каждого участника. Вот вывод этого кода:

```
<BookParticipants>
  <BookParticipant>
    <Name>Joe Rattz</Name>
    <Books>
      <Book>Pro LINQ: Language Integrated Query in C# 2008</Book>
    </Books>
  </BookParticipant>
  <BookParticipant>
    <Name>John Q. Public</Name>
    <Books />
  </BookParticipant>
</BookParticipants>
```

XML-разметка получилась такой, как и следовало ожидать на основе приведенного кода, но обратите внимание, что элемент Books второго элемента BookParticipant пуст. Что если пустой элемент Books не нужен, если элементов Books нет? Можно было бы передать null подходящей операции и подавить элементы Books. В листинге 9.10 внесено небольшое изменение в код, производящий XML-разметку.

Листинг 9.10. Пример кода, предотвращающего генерацию пустого элемента

```
IEnumerable<XElement> elements =
  new XElement[] {
    new XElement("BookParticipant",
      new XElement("Name", "Joe Rattz"),
      new XElement("Book", "Pro LINQ: Language Integrated Query in C# 2008")),
    new XElement("BookParticipant",
      new XElement("Name", "John Q. Public"))};
XElement xElement =
  new XElement("BookParticipants",
    elements.Select(e =>
      new XElement(e.Name,
        new XElement(e.Element("Name").Name, e.Element("Name").Value),
        e.Elements("Book").Any() ?
          new XElement("Books", e.Elements("Book")) : null)));
Console.WriteLine(xElement);
```

Существенные изменения в этом коде выделены полужирным. Вместо простого создания элемента Books и указания всех существующих элементов Book в качестве его содержимого используется стандартная операция запроса Any в сочетании с тернарной операцией (if ? then : else) для создания элемента Books только в случае наличия элементов Book. Если элементы Book не существуют, то тернарная операция возвратит null, и конструктор XElement просто проигнорирует null, тем самым исключив создание элемента Books. Это может быть очень удобно. Вот результат выполнения после модификации:

```
<BookParticipants>
  <BookParticipant>
    <Name>Joe Rattz</Name>
    <Books>
      <Book>Pro LINQ: Language Integrated Query in C# 2008</Book>
    </Books>
  </BookParticipant>
  <BookParticipant>
    <Name>John Q. Public</Name>
  </BookParticipant>
</BookParticipants>
```

Как видите, теперь второй элемент `BookParticipant` уже не содержит пустого элемента `Books`, как это было в предыдущем примере.

Обработка множества соседних узлов одного уровня

Иногда при выполнении XML-трансформации в точности известно, сколько нужно выходных элементов каждого типа. Но что произойдет, если будет несколько известных элементов, наряду с переменным количеством повторяющихся элементов, и все на одном уровне дерева для каждого вхождения в исходной XML-разметке? Предположим, что имеется показанная ниже XML-разметка.

Так должна выглядеть исходная XML-разметка

```
<BookParticipants>
  <BookParticipant type="Author">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
    <Nickname>Joey</Nickname>
    <Nickname>Null Pointer</Nickname>
  </BookParticipant>
  <BookParticipant type="Editor">
    <FirstName>Ewan</FirstName>
    <LastName>Buckingham</LastName>
  </BookParticipant>
</BookParticipants>
```

Что если необходимо выровнять эту структуру, чтобы корневой узел `BookParticipants` содержал только повторяющиеся наборы элементов `FirstName`, `LastName` и `Nickname`, вместо того, чтобы помещать их в дочерний элемент `BookParticipant`? Целевая XML-разметка должна выглядеть следующим образом.

Так должна выглядеть XML-разметка после трансформации

```
<BookParticipants>
  <!-- BookParticipant -->
  <FirstName>Joe</FirstName>
  <LastName>Rattz</LastName>
  <Nickname>Joey</Nickname>
  <Nickname>Null Pointer</Nickname>
  <!-- BookParticipant -->
  <FirstName>Ewan</FirstName>
  <LastName>Buckingham</LastName>
</BookParticipants>
```

Комментарии не обязательны, но они могут облегчить людям понимание того, что они видят.

Поскольку этот пример более сложен, он требует дополнительных пояснений. Рассмотрим код в листинге 9.11, выполняющий эту трансформацию.

Листинг 9.11. Обработка множества равноправных узлов в плоской структуре

```
XDocument xDocument = new XDocument(
  new XElement("BookParticipants",
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz"),
      new XElement("Nickname", "Joey"),
      new XElement("Nickname", "Null Pointer")),
      new XElement("BookParticipant",
    new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));
Console.WriteLine("Исходный XML-документ:");
Console.WriteLine("{0}{1}{1}", xDocument, System.Environment.NewLine);
```

В этой точке построено и отображено дерево XML. Оно соответствует XML-разметке, указанной в качестве исходной ранее. Теперь необходимо трансформировать исходную XML-разметку:

```
XDocument xTransDocument = new XDocument(
  new XElement("BookParticipants",
    xDocument.Element("BookParticipants")
      .Elements("BookParticipant")
```

Здесь возникает сложность. Планируется использовать проекцию через операцию `Select` для создания объекта, в котором будет содержаться комментарий, имя, фамилия и любые прозвища. Но какого типа объект должен быть создан? Можно было бы создать элемент и сделать комментарий, имя и остальное его дочерними элементами, но это добавит в XML новый уровень. Поэтому необходимо создать нечто такое, что не приведет к добавлению нового уровня к дереву XML. Для этого подошел бы массив объектов, поскольку в C# массив реализует интерфейс `IEnumerable<T>`, что позволяет работать с массивом как с последовательностью. Как объяснялось в главе 7, когда `IEnumerable` передается в качестве содержимого конструктору `XElement`, выполняется перечисление последовательности, и каждый объект этой последовательности применяется к конструируемому элементу. Для наполнения этого массива комментарием, именем, фамилией и прозвищами применяется средство инициализации коллекций C#:

```
      .Select(e => new object[] {
        new XComment(" BookParticipant "),
        new XElement("FirstName", (string)e.Element("FirstName")),
        new XElement("LastName", (string)e.Element("LastName")),
        e.Elements("Nickname")})));
Console.WriteLine("Трансформированный XML-документ:");
Console.WriteLine(xTransDocument);
```

В этой точке получена проекция массива, содержащего комментарий, элемент `FirstName`, элемент `LastName` и столько элементов `NickName`, сколько есть в исходном XML. И, наконец, трансформированный документ XML отображается в окне консоли.

Этот пример действительно довольно сложен. Обратите внимание, что массив объектов включает объект `XComment`, два объекта `XElement` и `IEnumerable<XElement>`. Проектируя вновь созданный массив как возвращаемое значение операции `Select`, в качестве содержимого вновь сконструированного элемента `BookParticipants` возвращается последовательность из `object[]` — `IEnumerable<object[]>`.

В этом случае каждый объект последовательности является массивом объектов, состоящим из комментария, элементов `FirstName` и `LastName`, а также последователь-

ности элементов NickName. Поскольку, как уже упоминалось, массив объектов не добавляет уровень к дереву XML, он добавляет свои элементы непосредственно в элемент BookParticipants.

Это может показаться запутанным, так что лучше взглянем на результат:

```
Исходный XML-документ:
<BookParticipants>
  <BookParticipant type="Author">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  <Nickname>Joey</Nickname>
  <Nickname>Null Pointer</Nickname>
  </BookParticipant>
  <BookParticipant type="Editor">
    <FirstName>Ewan</FirstName>
    <LastName>Buckingham</LastName>
  </BookParticipant>
</BookParticipants>

Трансформированный XML-документ:
<BookParticipants>
  <!-- BookParticipant -->
  <FirstName>Joe</FirstName>
  <LastName>Rattz</LastName>
  <Nickname>Joey</Nickname>
  <Nickname>Null Pointer</Nickname>
  <!-- BookParticipant -->
  <FirstName>Ewan</FirstName>
  <LastName>Buckingham</LastName>
</BookParticipants>
```

Трансформированный XML-документ в точности отвечает спецификации. Действительно сложная часть этого примера заключена в проекции массива объектов, не являющегося классом XML, для создания равноправных элементов XML без добавления нового уровня XML в дерево.

Проверка достоверности

API-интерфейс для работы с XML был бы неполным без возможности проверки достоверности XML. В LINQ to XML имеются средства проверки XML-документа на соответствие схеме XML.

Расширяющие методы

Потребность в проверке достоверности в LINQ to XML реализуется за счет создания статического класса System.Xml.Schema.Extensions, содержащего методы проверки достоверности. Эти методы проверки достоверности реализованы в виде расширяющих методов.

Прототипы

Ниже приведен список некоторых доступных методов проверки достоверности в классе System.Xml.Schema.Extensions:

```
void Extensions.Validate(this XDocument source, XmlSchemaSet schemas,
  ValidationEventHandler validationEventHandler)

void Extensions.Validate(this XDocument source, XmlSchemaSet schemas,
  ValidationEventHandler validationEventHandler, bool addSchemaInfo)
```

```
void Extensions.Validate(this XElement source,
  XmlSchemaObject partialValidationType, XmlSchemaSet schemas,
  ValidationEventHandler validationEventHandler)
void Extensions.Validate(this XElement source,
  XmlSchemaObject partialValidationType, XmlSchemaSet schemas,
  ValidationEventHandler validationEventHandler, bool addSchemaInfo)
void Extensions.Validate(this XAttribute source,
  XmlSchemaObject partialValidationType, XmlSchemaSet schemas,
  ValidationEventHandler validationEventHandler)
void Extensions.Validate(this XAttribute source,
  XmlSchemaObject partialValidationType, XmlSchemaSet schemas,
  ValidationEventHandler validationEventHandler, bool addSchemaInfo)
```

Для каждого типа объектов, на которых может быть вызван метод, предусмотрено по два прототипа. Эти типы — `XDocument`, `XElement` и `XAttribute`. Во втором прототипе для каждого типа объекта просто предусмотрен аргумент `bool` для указания того, должна ли добавляться информация схемы к объектам `XElement` и `XAttribute` после проверки достоверности. Первый метод для каждого из типов объектов, который не имеет аргумента `bool`, работает так же, как второй, когда в аргументе `addShemaInfo` ему передается `false`. В этом случае после проверки достоверности в объекты LINQ to XML никакой информации схемы не добавляется.

Чтобы получить информацию схемы для объектов `XElement` или `XAttribute`, вызывайте метод `GetSchemaInfo` на объекте. Если информация схемы не добавлена, либо в результате применения первого прототипа, либо второго с передачей `false` в аргументе `addShemaInfo`, то метод `GetSchemaInfo` вернет `null`. В противном случае он вернет объект, реализующий `IXmlSchemaInfo`. Этот объект будет содержать свойство по имени `SchemaElement`, которое вернет объект `XmlSchemaAttribute`, если предположить, что элемент или атрибут допустим. Эти объекты могут использоваться для получения дополнительной информации о схеме.

Важно отметить, что информация о схеме не доступна *во время* проверки достоверности, а только после ее завершения. Это значит, что получить информацию о схеме в обработчике события проверки достоверности нельзя. В этом случае вызов метода `GetSchemaInfo` вернет `null`. Это также означает, что проверка достоверности должна полностью завершиться, и что внутри обработчика события проверки достоверности не допускается генерировать исключения.

Совет. Информация о схеме не доступна во время проверки достоверности, а только после нее. Вызов `GetSchemaInfo` внутри обработчика события проверки достоверности вернет `null`.

Обратите внимание, что прототипы метода `Validate` для элементов и атрибутов требуют передачи `XmlSchemaObject` в качестве одного из аргументов. Это значит, что документ, в котором они находятся, уже должен быть проверен.

И, наконец, если вы передадите `null` в аргументе `ValidationEventHandler`, то в случае ошибки проверки достоверности будет сгенерировано исключение типа `XmlSchemaValidationException`. Это будет простейшим подходом к организации проверки достоверности XML-документа.

Получение схемы XML

Для проверки достоверности XML-документа понадобится иметь файл схемы XSD или знать, как его получить. Ниже будет показано, как в этом может помочь .NET Framework. Рассмотрим пример в листинге 9.12.

Листинг 9.12. Создание схемы XSD за счет ее извлечения из документа XML

```
XDocument xDocument = new XDocument(
  new XElement("BookParticipants",
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));
Console.WriteLine("Исходный XML-документ:");
Console.WriteLine("{0}{1}{1}", xDocument, System.Environment.NewLine);
xDocument.Save("bookparticipants.xml");
XmlSchemaInference infer = new XmlSchemaInference();
XmlSchemaSet schemaSet =
  infer.InferSchema(new XmlTextReader("bookparticipants.xml"));
XmlWriter w = XmlWriter.Create("bookparticipants.xsd");
foreach (XmlSchema schema in schemaSet.Schemas())
{
  schema.Write(w);
}
w.Close();
XDocument newDocument = XDocument.Load("bookparticipants.xsd");
Console.WriteLine("Извлеченная схема:");
Console.WriteLine("{0}{1}{1}", newDocument, System.Environment.NewLine);
```

В приведенном коде сначала создается обычный XML-документ, который уже использовался во многих примерах, и отображается для целей ознакомления. Затем XML-документ сохраняется на диске. После этого создается экземпляр объекта `XmlSchemaInference` и строится `XmlSchemaSet` вызовом метода `InferSchema` на объекте `XmlSchemaInference`. Далее создается объект для записи и выполняется перечисление `XmlSchemaSet` с записью каждой схемы в файл `bookparticipants.xsd`. И, наконец, сгенерированный файл схемы XSD загружается и отображается. Результат выполнения показан ниже.

```
Исходный XML-документ:
<BookParticipants>
  <BookParticipant type="Author">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
  <BookParticipant type="Editor">
    <FirstName>Ewan</FirstName>
    <LastName>Buckingham</LastName>
  </BookParticipant>
</BookParticipants>

Извлеченная схема:
<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="BookParticipants">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" name="BookParticipant">
          <xs:complexType>
            <xs:sequence>
```

```
            <xs:element name="FirstName" type="xs:string" />
            <xs:element name="LastName" type="xs:string" />
          </xs:sequence>
          <xs:attribute name="type" type="xs:string" use="required" />
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

Получить схему подобным образом совсем не трудно. Далее сгенерированный файл XSD схемы по имени `bookparticipants.xsd` используется в примерах проверки достоверности. Также обратите внимание, что в данном примере применяется класс `XmlSchemaSet`, который также будет участвовать в примерах проверки достоверности.

Примеры

В первом примере демонстрируются простейшие средства проверки достоверности документа XML — именно такому подходу отдает предпочтение большинство разработчиков. Для этого нужно просто указать `null` в качестве аргумента `ValidationEventHandler` (листинг 9.13).

Листинг 9.13. Проверка достоверности документа XML с обработкой события проверки по умолчанию

```
XDocument xDocument = new XDocument(
  new XElement("BookParticipants",
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("MiddleInitial", "C"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));
Console.WriteLine("Исходный XML-документ:");
Console.WriteLine("{0}{1}{1}", xDocument, System.Environment.NewLine);
XmlSchemaSet schemaSet = new XmlSchemaSet();
schemaSet.Add(null, "bookparticipants.xsd");
try
{
  xDocument.Validate(schemaSet, null);
  Console.WriteLine("Проверка достоверности документа завершена успешно.");
}
catch (XmlSchemaValidationException ex)
{
  Console.WriteLine("Произошло исключение: {0}", ex.Message);
  Console.WriteLine("Документ не прошел проверку достоверности.");
}
```

В этом примере конструируется обычный XML-документ, в который преднамеренно не добавляется элемент `MiddleInitial`, чтобы сделать его недопустимым. Используется схема, полученная в предыдущем примере. Обратите внимание, что в аргументе `ValidationEventHandler` метода `Validate` передается `null`. Это значит, что если воз-

никнет ошибка проверки достоверности, будет автоматически сгенерировано исключение `XmlSchemaValidationException`. И вот результат:

```
Исходный XML-документ:
<BookParticipants>
  <BookParticipant type="Author">
    <FirstName>Joe</FirstName>
    <MiddleInitial>C</MiddleInitial>
    <LastName>Rattz</LastName>
  </BookParticipant>
  <BookParticipant type="Editor">
    <FirstName>Ewan</FirstName>
    <LastName>Buckingham</LastName>
  </BookParticipant>
</BookParticipants>

Произошло исключение: The element 'BookParticipant' has invalid child element
'MiddleInitial'. List of possible elements expected: 'LastName'.

Элемент 'BookParticipant' имеет недопустимый дочерний элемент
'MiddleInitial'. Список ожидаемых допустимых элементов: 'LastName'.
Документ не прошел проверку достоверности.
```

Все очень просто и нормально работает.

В следующем примере выполняется проверка достоверности типичного XML-документа — того, который был использован для генерации схемы — на предмет соответствия его этой схеме. Конечно, поскольку схема получена на основе этого самого XML-документа, она должна работать. Однако для этого примера понадобится метод `ValidationEventHandler`. Давайте посмотрим, как он выглядит.

Обработчик `ValidationEventHandler`

```
static void MyValidationEventHandler(object o, ValidationEventArgs vea)
{
  Console.WriteLine("Ошибка проверки достоверности во время обработки объекта
типа {0}.", o.GetType().Name);
  Console.WriteLine(vea.Message);
  throw (new Exception(vea.Message));
}
```

В этом обработчике не делается ничего помимо вывода сообщения о проблеме и генерации исключения. Конечно, то, как будет выполнена обработка, полностью зависит от обработчика. Нет необходимости генерировать исключение. Можно было бы организовать более изящную обработку ошибок проверки достоверности — возможно, игнорируя все или определенные ошибки.

Рассмотрим пример использования этого обработчика, приведенный в листинге 9.14.

Листинг 9.14. Успешная проверка достоверности XML-документа по схеме XSD

```
XDocument xDocument = new XDocument(
  new XElement("BookParticipants",
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));
Console.WriteLine("Исходный XML-документ:");
Console.WriteLine("{0}{1}{1}", xDocument, System.Environment.NewLine);
```

```
XmlSchemaSet schemaSet = new XmlSchemaSet();
schemaSet.Add(null, "bookparticipants.xsd");
try
{
  xDocument.Validate(schemaSet, MyValidationEventHandler);
  Console.WriteLine("Проверка достоверности документа завершена успешно.");
}
catch (Exception ex)
{
  Console.WriteLine("Произошло исключение: {0}", ex.Message);
  Console.WriteLine("Документ не прошел проверку достоверности.");
}
```

В этом примере создается типичный XML-документ, который сразу отображается на консоли. Затем создается экземпляр объекта `XmlSchemaSet` и добавляется созданный ранее файл схемы с помощью метода `Add`. После этого на XML-документе вызывается расширяющий метод `Validate` с передачей ему набора схем и метода-обработчика события проверки достоверности. Обратите внимание, что в целях безопасности вызов метода `Validate` помещен в блок `try/catch`. Взглянем на результат:

```
Исходный XML-документ:
<BookParticipants>
  <BookParticipant type="Author">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
  <BookParticipant type="Editor">
    <FirstName>Ewan</FirstName>
    <LastName>Buckingham</LastName>
  </BookParticipant>
</BookParticipants>

Проверка достоверности документа завершена успешно.
```

Как видите, XML-документ успешно прошел проверку достоверности. Теперь рассмотрим пример, приведенный в листинге 9.15, с неправильным документом.

Листинг 9.15. Неудачная проверка достоверности XML-документа по схеме XSD

```
XDocument xDocument = new XDocument(
  new XElement("BookParticipants",
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XAttribute("language", "English"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));
Console.WriteLine("Исходный XML-документ:");
Console.WriteLine("{0}{1}{1}", xDocument, System.Environment.NewLine);
XmlSchemaSet schemaSet = new XmlSchemaSet();
schemaSet.Add(null, "bookparticipants.xsd");
try
{
  xDocument.Validate(schemaSet, MyValidationEventHandler);
  Console.WriteLine("Проверка достоверности документа завершена успешно.");
}
```

```
catch (XmlSchemaValidationException ex)
{
  Console.WriteLine("Произошло исключение: {0}", ex.Message);
  Console.WriteLine("Документ не прошел проверку достоверности.");
}
```

Этот код идентичен коду предыдущего примера, за исключением того, что добавлен дополнительный атрибут — `language`. Поскольку в схеме этот атрибут не указан, документ XML считается недопустимым. Ниже показан результат:

```
Исходный XML-документ:
<BookParticipants>
  <BookParticipant type="Author" language="English">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
  <BookParticipant type="Editor">
    <FirstName>Ewan</FirstName>
    <LastName>Buckingham</LastName>
  </BookParticipant>
</BookParticipants>
Ошибка проверки достоверности во время обработки объекта типа XAttribute.
The 'language' attribute is not declared.
Произошло исключение: The 'language' attribute is not declared.
                      Атрибут 'language' не объявлен.
Документ не прошел проверку достоверности.
```

Как видите, XML-документ не прошел проверку достоверности. В двух предыдущих примерах для обработки проверки достоверности создавался именованный метод `MyValidationEventHandler`. В листинге 9.16 показан тот же самый пример, но в нем вместо именованного метода для `ValidationEventHandler` применяется лямбда-выражение.

Листинг 9.16. Неудачная проверка достоверности XML-документа по схеме XSD с использованием лямбда-выражения

```
XDocument xDocument = new XDocument(
  new XElement("BookParticipants",
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XAttribute("language", "English"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));
Console.WriteLine("Исходный XML-документ:");
Console.WriteLine("{0}{1}{1}", xDocument, System.Environment.NewLine);
XmlSchemaSet schemaSet = new XmlSchemaSet();
schemaSet.Add(null, "bookparticipants.xsd");
try
{
  xDocument.Validate(schemaSet, (o, vea) =>
    {
      Console.WriteLine(
        "Ошибка проверки достоверности во время обработки объекта типа {0}.",
          o.GetType().Name);
      Console.WriteLine(vea.Message);
```

```
    throw (new Exception(vea.Message));
   });
  Console.WriteLine("Проверка достоверности документа завершена успешно.");
}
catch (XmlSchemaValidationException ex)
{
  Console.WriteLine("Произошло исключение: {0}", ex.Message);
  Console.WriteLine("Документ не прошел проверку достоверности.");
}
```

Обратите внимание, что целый метод указан в виде лямбда-выражения. И вот результат:

```
Исходный XML-документ:
<BookParticipants>
  <BookParticipant type="Author" language="English">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
  <BookParticipant type="Editor">
    <FirstName>Ewan</FirstName>
    <LastName>Buckingham</LastName>
  </BookParticipant>
</BookParticipants>

Ошибка проверки достоверности во время обработки объекта типа XAttribute.
The 'language' attribute is not declared.
Произошло исключение: The 'language' attribute is not declared.
                      Атрибут 'language' не объявлен.
Документ не прошел проверку достоверности.
```

В листинге 9.17 приведен пример, в котором выдаются указания о том, какая информация должна быть добавлена к схеме.

Листинг 9.17. Неудачная проверка достоверности XML-документа по схеме XSD с использованием лямбда-выражения и выдача указаний о том, что должно быть добавлено к схеме

```
XDocument xDocument = new XDocument(
  new XElement("BookParticipants",
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("MiddleName", "Carson"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));
Console.WriteLine("Исходный XML-документ:");
Console.WriteLine("{0}{1}{1}", xDocument, System.Environment.NewLine);
XmlSchemaSet schemaSet = new XmlSchemaSet();
schemaSet.Add(null, "bookparticipants.xsd");
xDocument.Validate(schemaSet, (o, vea) =>
  {
    Console.WriteLine("Исключение во время обработки объекта типа {0}.",
      o.GetType().Name);
    Console.WriteLine("{0}{1}", vea.Message, System.Environment.NewLine);
  },
  true);
```

```
foreach(XElement element in xDocument.Descendants())
{
  Console.WriteLine("Элемент {0} отмечен как {1}", element.Name,
    element.GetSchemaInfo().Validity);
  XmlSchemaElement se = element.GetSchemaInfo().SchemaElement;
  if (se != null)
  {
    Console.WriteLine(
      "Элемент схемы {0} должен иметь MinOccurs = {1} и MaxOccurs = {2}{3}",
      se.Name, se.MinOccurs, se.MaxOccurs, System.Environment.NewLine);
  }
  else
  {
    // Неверный элемент не имеет SchemaElement.
    Console.WriteLine();
  }
}
```

Этот пример начинается так же, как и предыдущий — с создания XML-документа. На этот раз, однако, в первый элемент `BookParticipant` добавлен дополнительный элемент `MiddleName`. Это неправильно, т.к. не задано в схеме, по которой выполняется проверка достоверности. В отличие от предыдущего примера исключения в коде обработчика события проверки достоверности не генерируются. Как вы, возможно, помните, для получения добавленной информации схемы проверка достоверности должна быть завершена, поэтому обработчик не должен генерировать исключения. По этой же причине удален блок `try/catch`.

После завершения проверки достоверности осуществляется перечисление по всем элементам документа с отображением информации об их допустимости. Кроме того, из добавленной информации схемы извлекается объект `SchemaElement`. Обратите внимание, что значение `SchemaElement` проверяется на неравенство `null`, поскольку если элемент неправильный, то свойство `SchemaElement` будет равно `null`. В конечном итоге элемент может быть неправильным потому, что его нет в схеме, и тогда информацию схемы для него взять попросту негде. То же самое касается свойства `SchemaAttribute` для недопустимых атрибутов. После получения объекта `SchemaElement` отображаются его свойства `Name`, `MinOccurs` и `MaxOccurs`.

Ниже показан результат:

```
Исходный XML-документ:
<BookParticipants>
  <BookParticipant type="Author">
    <FirstName>Joe</FirstName>
    <MiddleName>Carson</MiddleName>
    <LastName>Rattz</LastName>
  </BookParticipant>
  <BookParticipant type="Editor">
    <FirstName>Ewan</FirstName>
    <LastName>Buckingham</LastName>
  </BookParticipant>
</BookParticipants>

Исключение во время обработки объекта типа XElement.
The element 'BookParticipant' has invalid child element 'MiddleName'. List of
possible elements expected: 'LastName'.
Элемент 'BookParticipant' имеет недопустимый дочерний элемент 'MiddleInitial'.
Список ожидаемых допустимых элементов: 'LastName'.
```

```
Элемент BookParticipants отмечен как Invalid
Элемент схемы BookParticipants должен иметь MinOccurs = 1 и MaxOccurs = 1

Элемент BookParticipant отмечен как Invalid
Элемент схемы BookParticipant должен иметь MinOccurs = 1 и MaxOccurs =
79228162514264337593543950335

Элемент FirstName отмечен как Valid
Элемент схемы FirstName должен иметь MinOccurs = 1 и MaxOccurs = 1

Элемент MiddleName отмечен как Invalid

Элемент LastName отмечен как NotKnown

Элемент BookParticipant отмечен как Valid
Элемент схемы BookParticipant должен иметь MinOccurs = 1 и MaxOccurs =
79228162514264337593543950335

Элемент FirstName отмечен как Valid
Элемент схемы FirstName должен иметь MinOccurs = 1 и MaxOccurs = 1

Элемент LastName отмечен как Valid
Элемент схемы LastName должен иметь MinOccurs = 1 и MaxOccurs = 1
```

В этом выводе нет никаких сюрпризов. Обратите внимание, что свойство `MaxOccurs` для элемента `BookParticipant` содержит очень большое число. Причина в том, что в схеме атрибут `maxOccurs` указан как `"unbounded"` (не ограничен).

В последних двух примерах проверки достоверности используется один из прототипов метода `Validate`, применимый к проверке достоверности элементов. Первое, что здесь можно заметить — это то, что он имеет аргумент, требующий передачи `XmlSchemaObject`. Это значит, что документ уже должен быть проверен. Выглядит странно. То есть получается сценарий, когда нужно повторно выполнять проверку достоверности части дерева XML.

Для целей этого сценария предположим, что XML-документ загружен и для начала проверен. После этого пользователю разрешается обновить данные об одном из участников работы над книгой. Теперь необходимо обновить XML-документ, чтобы отразить эти изменения, поэтому проверка достоверности части дерева XML должны быть выполнена снова, после обновления. И здесь приходят на помощь прототипы метода `Validate` для элементов и атрибутов.

Ввиду сложности, пример, приведенный в листинге 9.18, требует дополнительных пояснений. С целью небольшого разнообразия, а еще потому, что нужна расширенная схема для облегчения редактирования дерева XML, она определяется программно вместо загрузки из файла, как это делалось в предыдущих примерах.

Листинг 9.18. Успешная проверка достоверности элемента XML

```
string schema =
  @"<?xml version='1.0' encoding='utf-8'?>
    <xs:schema attributeFormDefault='unqualified' elementFormDefault='qualified'
    xmlns:xs='http://www.w3.org/2001/XMLSchema'>
    <xs:element name='BookParticipants'>
      <xs:complexType>
        <xs:sequence>
          <xs:element maxOccurs='unbounded' name='BookParticipant'>
            <xs:complexType>
              <xs:sequence>
                <xs:element name='FirstName' type='xs:string' />
                <xs:element minOccurs='0' name='MiddleInitial' type='xs:string' />
                <xs:element name='LastName' type='xs:string' />
              </xs:sequence>
```

```
          <xs:attribute name='type' type='xs:string' use='required' />
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:schema>";
XmlSchemaSet schemaSet = new XmlSchemaSet();
schemaSet.Add("", XmlReader.Create(new StringReader(schema)));
```

Здесь просто скопирована схема из ранее использованного файла. Все двойные кавычки заменены одиночными. Также между элементами `FirstName` и `LastName` добавлен элемент `MiddleInitial`. Обратите внимание, что атрибут `minOccurs` указан равным 0, т.е. данный элемент необязателен. Затем создан набор схем из этой схемы. Теперь самое время построить XML-документ:

```
XDocument xDocument = new XDocument(
  new XElement("BookParticipants",
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));
Console.WriteLine("Исходный XML-документ:");
Console.WriteLine("{0}{1}{1}", xDocument, System.Environment.NewLine);
```

Здесь нет ничего нового — просто создается тот же документ, как обычно в примерах, после чего он отображается. Теперь выполним проверку достоверности документа:

```
bool valid = true;
xDocument.Validate(schemaSet, (o, vea) =>
{
  Console.WriteLine("Исключение во время обработки объекта типа {0}.",
    o.GetType().Name);
  Console.WriteLine(vea.Message);
  valid = false;
}, true);
Console.WriteLine("Документ проверен {0}.{1}",
  valid ? "успешно" : "неудачно",
  System.Environment.NewLine);
```

Обратите внимание, что проверка достоверности выполняется немного иначе, чем в предыдущих примерах. Переменная `valid` типа `bool` инициализируется значением `true`, тем самым утверждая, что документ допустим. Внутри обработчика проверки достоверности она устанавливается в `false`. Таким образом, если возникнет ошибка проверки достоверности, `valid` будет установлена в `false`. Затем после проверки достоверности проверяется значение `valid`, чтобы определить, корректен ли документ, и отображаются сведения о его допустимости. В рассматриваемом примере в этой точке документ корректен.

Теперь предположим, что пользователю разрешено редактировать информацию о любом определенном участнике работы над книгой. Пользователь изменил информацию об участнике по имени "`Joe`". Таким образом, мы получаем ссылку на этот элемент, обновляем его и заново выполняем проверку достоверности после обновления:

```
XElement bookParticipant = xDocument.Descendants("BookParticipant").
  Where(e => ((string)e.Element("FirstName")).Equals("Joe")).First();
bookParticipant.Element("FirstName").
  AddAfterSelf(new XElement("MiddleInitial", "C"));
valid = true;
bookParticipant.Validate(bookParticipant.GetSchemaInfo().SchemaElement, schemaSet,
  (o, vea) =>
  {
    Console.WriteLine("Исключение во время обработки объекта типа {0}.",
      o.GetType().Name);

    Console.WriteLine(vea.Message);
    valid = false;
  }, true);
Console.WriteLine("Элемент проверен {0}.{1}",
  valid ? "успешно" : "неудачно",
  System.Environment.NewLine);
```

Как видите, переменная `valid` инициализируется значением `true` и вызывается метод `Validate` — на этот раз на элементе `bookParticipant` вместо целого документа. Внутри обработчика события проверки достоверности `valid` устанавливается в `false`. После проверки достоверности элемента `BookParticipant` отображается результат его проверки. Вывод показан ниже:

```
Исходный XML-документ:
<BookParticipants>
  <BookParticipant type="Author">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
  <BookParticipant type="Editor">
    <FirstName>Ewan</FirstName>
    <LastName>Buckingham</LastName>
  </BookParticipant>
</BookParticipants>

Проверка достоверности документа завершена успешно.

Элемент проверен успешно.
```

Как видите, элемент успешно прошел проверку достоверности. В последнем примере будет использоваться тот же код, но на этот раз при обновлении элемента `BookParticipont` создается элемент `MiddleName` вместо `MiddleInitial`, который не является правильным. Этот код представлен в листинге 9.19.

Листинг 9.19. Неудачная проверка достоверности элемента XML

```
string schema =
  @"<?xml version='1.0' encoding='utf-8'?>
    <xs:schema attributeFormDefault='unqualified' elementFormDefault='qualified'
     xmlns:xs='http://www.w3.org/2001/XMLSchema'>
     <xs:element name='BookParticipants'>
       <xs:complexType>
         <xs:sequence>
           <xs:element maxOccurs='unbounded' name='BookParticipant'>
             <xs:complexType>
               <xs:sequence>
                 <xs:element name='FirstName' type='xs:string' />
                 <xs:element minOccurs='0' name='MiddleInitial' type='xs:string' />
                 <xs:element name='LastName' type='xs:string' />
               </xs:sequence>
```

```
            <xs:attribute name='type' type='xs:string' use='required' />
          </xs:complexType>
        </xs:element>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
   </xs:schema>";

XmlSchemaSet schemaSet = new XmlSchemaSet();
schemaSet.Add("", XmlReader.Create(new StringReader(schema)));

XDocument xDocument = new XDocument(
  new XElement("BookParticipants",
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));

Console.WriteLine("Исходный XML-документ:");
Console.WriteLine("{0}{1}{1}", xDocument, System.Environment.NewLine);

bool valid = true;
xDocument.Validate(schemaSet, (o, vea) =>
  {
    Console.WriteLine("Исключение во время обработки объекта типа {0}.",
      o.GetType().Name);
    Console.WriteLine(vea.Message);
    valid = false;
  }, true);

Console.WriteLine("Документ проверен {0}.{1}",
  valid ? "успешно" : "неудачно",
  System.Environment.NewLine);

XElement bookParticipant = xDocument.Descendants("BookParticipant").
  Where(e => ((string)e.Element("FirstName")).Equals("Joe")).First();

bookParticipant.Element("FirstName").
  AddAfterSelf(new XElement("MiddleName", "Carson"));

valid = true;

bookParticipant.Validate(bookParticipant.GetSchemaInfo().SchemaElement, schemaSet,
  (o, vea) =>
  {
    Console.WriteLine("Исключение во время обработки объекта типа {0}.",
      o.GetType().Name);
    Console.WriteLine(vea.Message);
    valid = false;
  }, true);

Console.WriteLine("Элемент проверен {0}.{1}",
  valid ? "успешно" : "неудачно",
  System.Environment.NewLine);
```

Этот код идентичен коду предыдущего примера, за исключением добавления элемента `MiddleInitial`. Вместо него специально добавлен неправильный элемент `MiddleName`. Ниже показан результат:

```
Исходный XML-документ:
<BookParticipants>
  <BookParticipant type="Author">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
  <BookParticipant type="Editor">
    <FirstName>Ewan</FirstName>
    <LastName>Buckingham</LastName>
  </BookParticipant>
</BookParticipants>

Проверка достоверности документа завершена успешно.

Исключение во время обработки объекта типа XElement.
The element 'BookParticipant' has invalid child element 'MiddleName'. List of
possible elements expected: 'MiddleInitial, LastName'.
Элемент 'BookParticipant' имеет недопустимый дочерний элемент 'MiddleName'.
Список ожидаемых допустимых элементов: 'MiddleInitial, LastName'.

Элемент проверен неудачно.
```

Как видите, новый элемент является недопустимым. Этот пример может показаться несколько надуманным, поскольку вряд ли разработчик станет создавать интерфейс, который даст пользователю возможность вносить некорректные изменения в процессе редактирования. Но предположим, что "пользователем" является какой-то другой процесс, обрабатывающий XML-документ. Возможно, XML-документ передан в отдельную программу для внесения изменений, и не известно, чего от нее можно ожидать. В таком случае повторная проверка достоверности совершенно оправдана.

XPath

Если вы привыкли использовать XPath, то можете также получить в свое распоряжение некоторые средства запросов XPath, благодаря классу `System.Xml.XPath.Extensions` из пространства имен `System.Xml.XPath`. Этот класс добавляет возможности поиска XPath через расширяющие методы.

Прототипы

Ниже приведен список некоторых прототипов методов, доступных в классе `System.Xml.XPath.Extensions`.

```
XPathNavigator Extensions.CreateNavigator(this XNode node);
XPathNavigator Extensions.CreateNavigator(this XNode node, XmlNameTable nameTable);
object Extensions.XPathEvaluate(this XNode node, string expression);
object Extensions.XPathEvaluate(this XNode node, string expression,
  IXmlNamespaceResolver resolver);
XElement Extensions.XPathSelectElement(this XNode node, string expression);
XElement Extensions.XPathSelectElement(this XNode node, string expression,
  IXmlNamespaceResolver resolver);
IEnumerable<XElement> Extensions.XPathSelectElements(this XNode node,
  string expression);
IEnumerable<XElement> Extensions.XPathSelectElements(this XNode node,
  string expression, IXmlNamespaceResolver resolver);
```

Примеры

Используя расширяющие методы, можно опрашивать документ LINQ to XML с применением поисковых выражений. Пример приведен в листинге 9.20.

Листинг 9.20. Опрос XML с использованием синтаксиса XPath

```
XDocument xDocument = new XDocument(
  new XElement("BookParticipants",
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
    new XElement("LastName", "Buckingham"))));

XElement bookParticipant = xDocument.XPathSelectElement(
  "//BookParticipants/BookParticipant[FirstName='Joe']");

Console.WriteLine(bookParticipant);
```

Здесь опять создан обычный XML-документ. Однако на этот раз он не отображается. Затем на документе вызывается метод `XPathSelectElement` с предоставлением ему выражения поиска XPath для нахождения элемента `BookParticipant`, значение элемента `FirstName` которого равно `"Joe"`. Ниже показан результат:

```
<BookParticipant type="Author">
  <FirstName>Joe</FirstName>
  <LastName>Rattz</LastName>
</BookParticipant>
```

С применением расширяющих методов XPath можно получить ссылку на объект `System.Xml.XPath.XPathNavigator` для осуществления навигации по XML-документу, выполнения запроса XPath для возврата элемента или последовательности элементов либо вычисления выражений запросов XPath.

Резюме

После такого количества примеров, рассмотренных в этой и предыдущих главах, должно стать ясно, что API-интерфейс LINQ to XML исключительно полезен в реальном производственном коде. Поскольку работа с XML в LINQ to XML в основном основана на элементах, а не на документах, и благодаря возможности функционального конструирования, создание XML становится совершенно безболезненным. Легкость создания документов в комбинации с интуитивно понятными средствами навигации и модификации обеспечивает большие преимущества LINQ to XML перед альтернативными технологиями.

Облегчение работы с XML, основанное на мощном и гибком языке запросов, превращает LINQ to XML в одну из самых интересных частей всего LINQ. Если до сих пор вас пугала работа с XML, наверняка этого не будет в случае использования LINQ to XML.

ЧАСТЬ IV

LINQ to DataSet

В этой части

Глава 10. Операции LINQ to DataSet

Глава 11. Дополнительные возможности LINQ to DataSet

ГЛАВА 10

Операции LINQ to DataSet

Хотя API-интерфейс LINQ to SQL еще не рассматривался, следует упомянуть, что для использования LINQ to SQL с определенной базой данных для нее должны быть сгенерированы и скомпилированы классы исходного кода либо же должен быть создан файл отображения. Это значит, что выполнение запросов LINQ с помощью LINQ to SQL на неизвестной до момента выполнения базе данных попросту невозможно. К тому же LINQ to SQL работает только с Microsoft SQL Server. Что же делать разработчику?

Операции LINQ to DataSet позволяют разработчику выполнять запросы LINQ на `DataSet`, и поскольку объект `DataSet` может быть получен с использованием нормальных SQL-запросов ADO.NET, интерфейс LINQ to DataSet позволяет выполнять запросы LINQ к *любой* базе данных, которая может быть опрошена с помощью ADO.NET. Это представляет намного более динамичный интерфейс запросов баз данных, чем LINQ to SQL.

Наверняка возник вопрос: при каких обстоятельствах может случиться так, что база данных будет неизвестна до момента выполнения программы? Действительно, в типовых приложениях база данных известна на этапе разработки, и потому нет необходимости иметь дело с LINQ to DataSet. Но как насчет приложений типа утилит баз данных? Например, рассмотрим такое приложение, как диспетчер SQL Server Enterprise Manager. Ему не известно, какие базы данных будут установлены на сервере до момента его запуска. Приложение Enterprise Manager позволяет проверить, какие базы установлены на сервере и какие таблицы есть в каждой из них. Разработчик приложения Enterprise Manager не имел никакой возможности сгенерировать классы LINQ to SQL во время компиляции именно для *вашей* базы данных. И здесь на помощь приходит LINQ to DataSet.

Хотя эта часть книги названа “LINQ to DataSet”, вы найдете здесь операции, относящиеся к объектам `DataTable`, `DataRow` и `DataColumn`. Не удивляйтесь, если вы нечасто встретите здесь ссылки на объекты `DataSet`. Понятно, что в реальных обстоятельствах объекты `DataTable` почти всегда происходят от объектов `DataSet`. Однако для независимости от баз данных, краткости и ясности в большинстве приведенных примерах намеренно создаются простые объекты `DataTable` программно, а не извлекаются из базы данных.

Операции LINQ to DataSet состоят из множества специальных операций, определенных в нескольких сборках и пространствах имен, которые позволяют разработчику решать следующие задачи.

- Выполнять операции множеств на последовательностях объектов `DataRow`.
- Извлекать и устанавливать значения `DataColumn`.

- Получать из DataTable стандартные последовательности LINQ типа IEnumerable<T>, так что можно вызывать стандартные операции запросов.
- Копировать измененные последовательности из объектов DataRow в DataTable.

В дополнение к этим операциям LINQ to DataSet, как только вы вызовете операцию AsEnumerable, то сможете вызывать стандартные операции запросов из LINQ to Objects на возвращенных последовательностях объектов DataRow, достигая большей мощи и гибкости.

Необходимые сборки

Для выполнения примеров настоящей главы понадобится добавить в проект ссылки на динамические библиотеки сборок System.Data.dll и System.Data.DataSetExtensions, если это не было сделано ранее.

Необходимые пространства имен

Чтобы использовать операции LINQ to DataSet, добавьте в начало кода директивы using для пространств имен System.Linq и System.Data, если это еще не сделано:

```
using System.Data;
using System.Linq;
```

Это позволит коду находить операции LINQ to DataSet.

Общий код для примеров

Почти каждый пример настоящей главы потребует объекта DataTable, чтобы выполнять запросы LINQ to DataSet. В реальном рабочем коде вы обычно будете получать объекты DataTable, опрашивая базу данных. Однако для некоторых примеров будут представлены ситуации, когда условий данных из типичной базы будет недостаточно. Например, понадобятся дублированные записи для демонстрации метода Distinct. Вместо того чтобы пытаться манипулировать базой данных для получения нужной информации, будет просто программно создаваться объект DataTable, содержащий определенные данные, которые понадобятся для примера. Это также избавит от необходимости иметь специальную базу данных для тестирования большинства примеров.

Поскольку для получения объектов DataTable база данных опрашиваться не будет, и для того, чтобы облегчить создание этих объектов, они будут генерироваться из массива объектов предопределенных классов. В качестве предопределенного класса используется приведенный ниже класс Student.

Простой класс с двумя общедоступными членами

```
class Student
{
   public int Id;
   public string Name;
}
```

Нужно просто представить, что опрашивается таблица по имени Students, в которой каждая запись представляет студента, и эта таблица состоит из двух столбцов: Id и Name.

Чтобы облегчить создание базы данных DataTable, а также предотвратить неясности в существенных деталях каждого примера, используется общий метод для преобразования массива объектов Student в объект DataTable. Это позволит легко варьировать данные от примера к примеру. Вот этот общий метод.

Преобразование массива объектов `Student` в `DataTable`

```
static DataTable GetDataTable(Student[] students)
{
   DataTable table = new DataTable();
   table.Columns.Add("Id", typeof(Int32));
   table.Columns.Add("Name", typeof(string));
   foreach (Student student in students)
   {
      table.Rows.Add(student.Id, student.Name);
   }
   return (table);
}
```

В этом методе нет ничего сложного. Здесь просто создается экземпляр объекта `DataTable`, в который добавляются два столбца и затем добавляются строки для всех элементов в переданном массиве `students`.

Для многих примеров применения операций LINQ to DataSet понадобится отображать `DataTable`, чтобы показать результаты работы кода. Хотя реальные данные в `DataTable` будут варьироваться, код, необходимый для отображения заголовка объекта `DataTable`, останется неизменным. Вместо многократного повторения этого кода во всех примерах создается следующий метод, который будет вызываться в каждом примере, где требуется выводить заголовок `DataTable`.

Метод `OutputDataTableHeader`

```
static void OutputDataTableHeader(DataTable dt, int columnWidth)
{
   string format = string.Format("{0}0,-{1}{2}", "{", columnWidth, "}");
   // Отображение заголовков столбцов.
   foreach(DataColumn column in dt.Columns)
   {
      Console.Write(format, column.ColumnName);
   }
   Console.WriteLine();
   foreach(DataColumn column in dt.Columns)
   {
      for(int i = 0; i < columnWidth; i++)
      {
         Console.Write("=");
      }
   }
   Console.WriteLine();
}
```

Назначение этого метода — вывести заголовок `DataTable` в табличной форме.

Операции множеств `DataRow`

Как вы, вероятно, помните, в API-интерфейсе LINQ to Objects есть группа стандартных операций запросов, предназначенных для выполнения операций множеств над последовательностями объектов. Имеются в виду операции `Distinct`, `Except`, `Intersect`, `Union` и `SequenceEqual`. Каждая из этих операций выполняет одну из операций множеств над двумя последовательностями.

Каждой из этих операций для выполнения соответствующей операции с множествами необходим способ определения эквивалентности двух элементов множеств. Они осуществляют сравнение элементов, вызывая методы `GetHashCode` и `Equals` над элементами. Для `DataRow` это означает сравнение ссылок, что является нежелательным

поведением. Это приведет к неправильному определению эквивалентности элементов, что заставит операции возвращать некорректные результаты. Из-за этого каждая из таких операций имеет дополнительный прототип, который был опущен в главах, посвященных LINQ to Objects. Этот дополнительный прототип позволяет предоставлять в качестве аргумента объект `IEqualityComparer`. Удобно, что объект-компаратор по умолчанию — `System.Data.DataRowComparer.Default` — предназначен специально для этих версий операций. Этот класс-компаратор находится в пространстве имен `System.Data` в сборке `System.Data.Entity.dll`. Этот компаратор определяет эквивалентность элементов, сравнивая количество столбцов и статический тип данных каждого из них, а также используя интерфейс `IComparable` на динамическом типе данных столбца, если тип реализует этот интерфейс. В противном случае он вызывает статический метод `Equals` класса `System.Object`.

Каждый из этих дополнительных прототипов операций определен в статическом классе `System.Linq.Enumerable`, наряду со всеми остальными прототипами этих операций.

В настоящем разделе будет показано несколько примеров, чтобы проиллюстрировать неправильный и, что более важно — правильный способ выполнения сравнений последовательностей при работе с объектами `DataSet`.

Distinct

Операция `Distinct` исключает дублированные строки из последовательности объектов. Она возвращает объект, который при перечислении исходной последовательности возвращает последовательность объектов с исключенными дублированными строками. Обычно эта операция определяет дублирование строк, вызывая методы `GetHashCode` и `Equals` типа данных каждого из элементов. Однако для объектов типа `DataRow` это даст неверный результат.

Поскольку будет вызван дополнительный прототип и предоставлен объект-компаратор `System.Data.DataRowComparer.Default`, эквивалентность элементов будет определена правильно. С его помощью дублирование строк определяется сравнением объектов `DataRow`, используя количество столбцов в строке и статический тип данных каждого столбца, а затем используя интерфейс `IComparable` на каждом столбце, если его динамический тип данных реализует этот интерфейс, либо вызывая статический метод `Equals` класса `System.Object` в противном случае.

Прототипы

Операция `Distinct` имеет один прототип, описанный ниже.

Прототип Distinct

```
public static IEnumerable<T> Distinct<T> (
  this IEnumerable<T> source,
  IEqualityComparer<T> comparer);
```

Примеры

В первом примере объект `DataTable` создается из массива объектов `Student` с использованием общего метода `GetDataTable` и массива, содержащего в себе один дубликат. В массиве дублируется запись со значением `Id` равным 1. Затем содержимое `DataTable` отображается. Это доказывает, что запись встречается в `DataTable` два раза. После этого вызовом операции `Distinct` удаляются любые дублированные строки, и `DataTable` снова отображается, показывая, что дублированные строки удалены. Код представлен в листинге 10.1.

Листинг 10.1. Операция `Distinct` с компаратором эквивалентности

```
Student[] students = {
  new Student { Id = 1, Name = "Joe Rattz" },
  new Student { Id = 6, Name = "Ulyses Hutchens" },
  new Student { Id = 19, Name = "Bob Tanko" },
  new Student { Id = 45, Name = "Erin Doutensal" },
  new Student { Id = 1, Name = "Joe Rattz" },
  new Student { Id = 12, Name = "Bob Mapplethorpe" },
  new Student { Id = 17, Name = "Anthony Adams" },
  new Student { Id = 32, Name = "Dignan Stephens" }
};

DataTable dt = GetDataTable(students);

Console.WriteLine("{0}Перед вызовом Distinct(){0}",
  System.Environment.NewLine);
OutputDataTableHeader(dt, 15);

foreach (DataRow dataRow in dt.Rows)
{
  Console.WriteLine("{0,-15}{1,-15}",
    dataRow.Field<int>(0),
    dataRow.Field<string>(1));
}

IEnumerable<DataRow> distinct =
  dt.AsEnumerable().Distinct(DataRowComparer.Default);
Console.WriteLine("{0}После вызова Distinct(){0}",
  System.Environment.NewLine);
OutputDataTableHeader(dt, 15);
foreach (DataRow dataRow in distinct)
{
  Console.WriteLine("{0,-15}{1,-15}",
    dataRow.Field<int>(0),
    dataRow.Field<string>(1));
}
```

Обратите внимание, что для получения последовательности объектов `DataRow` из `DataTable` используется операция `AsEnumerable`, потому что именно на этой последовательности должна быть вызвана операция `Distinct`. Также заметьте, что в массиве `students` запись с `Id = 1` повторяется.

Как видите, на объекте `DataRow` вызывается метод по имени `Field`. Пока достаточно понимать, что это вспомогательный метод, который делает получение значения объекта `DataColumn` из `DataRow` более удобным. В разделе “Операции над полями `DataRow`” далее в главе операция `Field<T>` будет описана более подробно.

Вот результат:

```
Перед вызовом Distinct()
Id            Name
==============================
1             Joe Rattz
6             Ulyses Hutchens
19            Bob Tanko
45            Erin Doutensal
1             Joe Rattz
12            Bob Mapplethorpe
17            Anthony Adams
32            Dignan Stephens
```

```
После вызова Distinct()
Id          Name
==============================
1           Joe Rattz
6           Ulyses Hutchens
19          Bob Tanko
45          Erin Doutensal
12          Bob Mapplethorpe
17          Anthony Adams
32          Dignan Stephens
```

Обратите внимание, что в результатах перед вызовом операции `Distinct` запись со значением `Id = 1` повторялась, а после вызова `Distinct` второе ее появление было удалено.

Во втором примере демонстрируется результат вызова операции `Distinct`, но без указания объекта-компаратора. Код представлен в листинге 10.2.

Листинг 10.2. Операция `Distinct` без компаратора эквивалентности

```
Student[] students = {
  new Student { Id = 1, Name = "Joe Rattz" },
  new Student { Id = 6, Name = "Ulyses Hutchens" },
  new Student { Id = 19, Name = "Bob Tanko" },
  new Student { Id = 45, Name = "Erin Doutensal" },
  new Student { Id = 1, Name = "Joe Rattz" },
  new Student { Id = 12, Name = "Bob Mapplethorpe" },
  new Student { Id = 17, Name = "Anthony Adams" },
  new Student { Id = 32, Name = "Dignan Stephens" }
};
DataTable dt = GetDataTable(students);
Console.WriteLine("{0}Перед вызовом Distinct(){0}",
  System.Environment.NewLine);
OutputDataTableHeader(dt, 15);
foreach (DataRow dataRow in dt.Rows)
{
  Console.WriteLine("{0,-15}{1,-15}",
    dataRow.Field<int>(0),
    dataRow.Field<string>(1));
}
IEnumerable<DataRow> distinct = dt.AsEnumerable().Distinct();
Console.WriteLine("{0}После вызова Distinct(){0}",
  System.Environment.NewLine);
OutputDataTableHeader(dt, 15);
foreach (DataRow dataRow in distinct)
{
  Console.WriteLine("{0,-15}{1,-15}",
    dataRow.Field<int>(0),
    dataRow.Field<string>(1));
}
```

Единственное отличие этого кода от кода предыдущего примера в том, что здесь при вызове операции `Distinct` не указан компаратор эквивалентности. Удалит ли он дублированную строку? Давайте посмотрим:

```
Перед вызовом Distinct()
Id          Name
==============================
1           Joe Rattz
```

```
6           Ulyses Hutchens
19          Bob Tanko
45          Erin Doutensal
1           Joe Rattz
12          Bob Mapplethorpe
17          Anthony Adams
32          Dignan Stephens

После вызова Distinct()
Id          Name
==============================
1           Joe Rattz
6           Ulyses Hutchens
19          Bob Tanko
45          Erin Doutensal
1           Joe Rattz
12          Bob Mapplethorpe
17          Anthony Adams
32          Dignan Stephens
```

Нет, дублированная строка не удалена! Как видите, в этих двух примерах строки сравниваются по-разному.

Except

Операция `Except` предоставляет последовательность объектов `DataRow`, которые есть в первой последовательности объектов `DataRow`, но отсутствуют во второй последовательности объектов `DataRow`. Операция возвращает объект, который при перечислении проходит по второй последовательности объектов `DataRow`, собирая уникальные элементы, затем проходит по первой последовательности объектов `DataRow`, удаляя те из них, которые обнаружены во второй последовательности; сгенерированный результат возвращается.

Чтобы определить уникальность элементов в одной последовательности, а также то, что один элемент из первой последовательности не эквивалентен элементу во второй последовательности, операция должна иметь возможность определять их эквивалентность. Обычно эта операция определяет эквивалентность, вызывая методы `GetHashCode` и `Equals` типа данных каждого элемента. Однако для объектов типа `DataRow` это может дать некорректный результат.

Поскольку планируется вызывать дополнительный прототип и предоставить объект-компаратор `System.Data.DataRowComparer.Default`, эквивалентность элементов будет определена правильно. С его помощью дублирование строк определяется сравнением объектов `DataRow`, используя количество столбцов в строке и статический тип данных каждого столбца, а затем используя интерфейс `IComparable` на каждом столбце, если его динамический тип данных реализует этот интерфейс, либо вызывая статический метод `Equals` класса `System.Object` в противном случае.

Прототипы

Операция `Except` имеет один прототип, описанный ниже.

Прототип Except

```
public static IEnumerable<T> Except<T> (
  this IEnumerable<T> first,
  IEnumerable<T> second,
  IEqualityComparer<T> comparer);
```

Примеры

В этом примере операция `Except` вызывается дважды. Первый раз ей передается объект-компаратор `System.Data.DataRowComparer.Default`, так что результаты первого запроса с помощью операции `Except` будут корректными. При втором вызове операции `Except` объект-компаратор не передается. Это приведет к некорректному результатам запроса. В листинге 10.3 показан код.

Листинг 10.3. Операция `Except` с компаратором эквивалентности и без

```
Student[] students = {
  new Student { Id = 1, Name = "Joe Rattz" },
  new Student { Id = 7, Name = "Anthony Adams" },
  new Student { Id = 13, Name = "Stacy Sinclair" },
  new Student { Id = 72, Name = "Dignan Stephens" }
};

Student[] students2 = {
  new Student { Id = 5, Name = "Abe Henry" },
  new Student { Id = 7, Name = "Anthony Adams" },
  new Student { Id = 29, Name = "Future Man" },
  new Student { Id = 72, Name = "Dignan Stephens" }
};

DataTable dt1 = GetDataTable(students);
IEnumerable<DataRow> seq1 = dt1.AsEnumerable();
DataTable dt2 = GetDataTable(students2);
IEnumerable<DataRow> seq2 = dt2.AsEnumerable();
IEnumerable<DataRow> except =
  seq1.Except(seq2, System.Data.DataRowComparer.Default);
Console.WriteLine("{0}Результаты Except() с компаратором{0}",
  System.Environment.NewLine);
OutputDataTableHeader(dt1, 15);

foreach (DataRow dataRow in except)
{
  Console.WriteLine("{0,-15}{1,-15}",
  dataRow.Field<int>(0),
  dataRow.Field<string>(1));
}

except = seq1.Except(seq2);
Console.WriteLine("{0}Результаты Except() без компаратора{0}",
  System.Environment.NewLine);
OutputDataTableHeader(dt1, 15);

foreach (DataRow dataRow in except)
{
   Console.WriteLine("{0,-15}{1,-15}",
   dataRow.Field<int>(0),
   dataRow.Field<string>(1));
}
```

Здесь создаются два объекта `DataTable`, которые наполняются данными из массивов `Student`. С помощью метода `AsEnumerable` из каждого объекта `DataTable` создается последовательность. Затем вызывается операция `Except` с передачей ей объекта-компаратора `System.Data.DataRowComparer.Default`. Второй раз `Except` вызывается без указания объекта-компаратора.

Нажав <Ctrl+F5>, посмотрим на результат работы этого кода:

```
Результаты Except() с компаратором
Id          Name
==============================
1           Joe Rattz
13          Stacy Sinclair

Результаты Except() без компаратора
Id          Name
==============================
1           Joe Rattz
7           Anthony Adams
13          Stacy Sinclair
72          Dignan Stephens
```

Как видите, операция `Except`, вызванная с объектом-компаратором `System.Data.DataRowComparer.Default`, способна правильно определить эквивалентность элементов в двух последовательностях, в то время как операции `Except`, вызванной без объекта-компаратора, это не удается.

Intersect

Операция `Intersect` порождает последовательность объектов `DataRow`, которая представляет собой пересечение двух последовательностей объектов `DataRow`. Она возвращает объект, который при перечислении проходит по второй последовательности, собирая уникальные элементы, а затем проходит по первой последовательности, возвращая элементы, присутствующие в обеих последовательностях.

Чтобы определить уникальность элементов одной последовательности, а также эквивалентность элемента одной последовательности элементу другой последовательности, операция должна уметь определять эквивалентность элементов. Обычно эта операция определяет эквивалентность элементов вызовом методов `GetHashCode` и `Equals` типа данных каждого элемента. Однако для объектов типа `DataRow` это может дать неправильный результат.

Поскольку планируется вызывать дополнительный прототип и предоставить объект-компаратор `System.Data.DataRowComparer.Default`, эквивалентность элементов будет определена правильно. С его помощью дублирование строк определяется сравнением объектов `DataRow`, используя количество столбцов в строке и статический тип данных каждого столбца, а затем используя интерфейс `IComparable` на каждом столбце, если его динамический тип данных реализует этот интерфейс, либо вызывая статический метод `Equals` класса `System.Object` в противном случае.

Прототипы

Операция `Intersect` имеет один прототип, описанный ниже.

Прототип Intersect

```
public static IEnumerable<T> Intersect<T> (
  this IEnumerable<T> first,
  IEnumerable<T> second,
  IEqualityComparer<T> comparer);
```

Примеры

В этом примере используется тот же базовый код, который применялся в примере `Except`, с заменой вызова операции `Except` на `Intersect`.

В листинге 10.4 показан код.

Листинг 10.4. Операция `Intersect` с компаратором эквивалентности и без

```
Student[] students = {
  new Student { Id = 1, Name = "Joe Rattz" },
  new Student { Id = 7, Name = "Anthony Adams" },
  new Student { Id = 13, Name = "Stacy Sinclair" },
  new Student { Id = 72, Name = "Dignan Stephens" }
};
Student[] students2 = {
  new Student { Id = 5, Name = "Abe Henry" },
  new Student { Id = 7, Name = "Anthony Adams" },
  new Student { Id = 29, Name = "Future Man" },
  new Student { Id = 72, Name = "Dignan Stephens" }
};
DataTable dt1 = GetDataTable(students);
IEnumerable<DataRow> seq1 = dt1.AsEnumerable();
DataTable dt2 = GetDataTable(students2);
IEnumerable<DataRow> seq2 = dt2.AsEnumerable();
IEnumerable<DataRow> except =
  seq1. Intersect(seq2, System.Data.DataRowComparer.Default);
Console.WriteLine("{0}Результаты Intersect() с компаратором{0}",
  System.Environment.NewLine);
OutputDataTableHeader(dt1, 15);
foreach (DataRow dataRow in except)
{
  Console.WriteLine("{0,-15}{1,-15}",
  dataRow.Field<int>(0),
  dataRow.Field<string>(1));
}
except = seq1. Intersect(seq2);
Console.WriteLine("{0}Результаты Intersect() без компаратора{0}",
  System.Environment.NewLine);
OutputDataTableHeader(dt1, 15);
foreach (DataRow dataRow in except)
{
   Console.WriteLine("{0,-15}{1,-15}",
     dataRow.Field<int>(0),
     dataRow.Field<string>(1));
}
```

Здесь нет ничего нового. Из двух массивов `Student` создается пара объектов `DataTable`, из которых получаются последовательности. Затем операция `Intersect` вызывается сначала с объектом-компаратором, а потом без него. После каждого вызова `Intersect` отображаются результаты. Ниже показан вывод этого примера:

```
Результаты Intersect() с компаратором
Id           Name
==============================
7            Anthony Adams
72           Dignan Stephens

Результаты Intersect() без компаратора
Id           Name
==============================
```

Как видите, операция `Intersect` с компаратором способна правильно определить эквивалентность элементов из двух последовательностей, в то время как операция `Intersect` без компаратора — нет.

Union

Операция Union производит последовательность объектов DataRow, которая представляет собой объединение двух последовательностей объектов DataRow. Она возвращает объект, который при перечислении проходит по первой последовательности объектов DataRow, затем по второй последовательности объектов DataRow, возвращая любой объект, который не содержался в первой последовательности.

Для определения того факта, что элемент уже был возвращен, операция должна иметь возможность определять эквивалентность двух элементов. Обычно эта операция определяет эквивалентность элементов посредством вызова методов GetHashCode и Equals типа данных элементов. Однако для объектов типа DataRow это может привести к некорректному результату.

Поскольку планируется вызывать дополнительный прототип и предоставить объект-компаратор System.Data.DataRowComparer.Default, эквивалентность элементов будет определена правильно. С его помощью дублирование строк определяется сравнением объектов DataRow, используя количество столбцов в строке и статический тип данных каждого столбца, а затем используя интерфейс IComparable на каждом столбце, если его динамический тип данных реализует этот интерфейс, либо вызывая статический метод Equals класса System.Object в противном случае.

Прототипы

Операция Union имеет один прототип, описанный ниже.

Прототип Union

```
public static IEnumerable<T> Union<T> (
  this IEnumerable<T> first,
  IEnumerable<T> second,
  IEqualityComparer<T> comparer);
```

Примеры

В этом примере используется тот же базовый код, что и в примере Intersect, но вызов операции Intersect заменяется Union. Код представлен в листинге 10.5.

Листинг 10.5. Операция Union с компаратором эквивалентности и без

```
Student[] students = {
  new Student { Id = 1, Name = "Joe Rattz" },
  new Student { Id = 7, Name = "Anthony Adams" },
  new Student { Id = 13, Name = "Stacy Sinclair" },
  new Student { Id = 72, Name = "Dignan Stephens" }
};
Student[] students2 = {
  new Student { Id = 5, Name = "Abe Henry" },
  new Student { Id = 7, Name = "Anthony Adams" },
  new Student { Id = 29, Name = "Future Man" },
  new Student { Id = 72, Name = "Dignan Stephens" }
};
DataTable dt1 = GetDataTable(students);
IEnumerable<DataRow> seq1 = dt1.AsEnumerable();
DataTable dt2 = GetDataTable(students2);
IEnumerable<DataRow> seq2 = dt2.AsEnumerable();
IEnumerable<DataRow> except =
  seq1.Union(seq2, System.Data.DataRowComparer.Default);
Console.WriteLine("{0}Результаты Union() с компаратором{0}",
  System.Environment.NewLine);
```

```
OutputDataTableHeader(dt1, 15);
foreach (DataRow dataRow in except)
{
  Console.WriteLine("{0,-15}{1,-15}",
  dataRow.Field<int>(0),
  dataRow.Field<string>(1));
}
except = seq1.Union(seq2);
Console.WriteLine("{0}Результаты Union() без компаратора{0}",
  System.Environment.NewLine);
OutputDataTableHeader(dt1, 15);
foreach (DataRow dataRow in except)
{
   Console.WriteLine("{0,-15}{1,-15}",
     dataRow.Field<int>(0),
     dataRow.Field<string>(1));
}
```

И здесь нет ничего нового. Из двух массивов `Student` создается пара объектов `DataTable`, из которых получаются последовательности. Затем операция `Union` вызывается сначала с объектом-компаратором, потом без него. После каждого вызова `Union` отображаются результаты. Выполнение примера дает следующий вывод:

```
Результаты Union() с компаратором
Id           Name
==============================
1            Joe Rattz
7            Anthony Adams
13           Stacy Sinclair
72           Dignan Stephens
5            Abe Henry
29           Future Man

Результаты Union() без компаратора
Id           Name
==============================
1            Joe Rattz
7            Anthony Adams
13           Stacy Sinclair
72           Dignan Stephens
5            Abe Henry
7            Anthony Adams
29           Future Man
72           Dignan Stephens
```

Обратите внимание, что результат выполнения операции `Union` с объектом-компаратором корректен, но без объекта-компаратора — нет.

SequenceEqual

Операция `SequenceEqual` сравнивает две последовательности объектов `DataRow`, чтобы определить, эквивалентны ли они. Она выполняет перечисление двух исходных последовательностей, сравнивая соответствующие объекты `DataRow`. Если две исходные последовательности имеют одинаковое количество записей, и все соответствующие объекты `DataRow` в них эквивалентны, возвращается `true`. В противном случае две последовательности считаются не эквивалентными, и возвращается `false`.

Эта операция должна уметь определять эквивалентность двух элементов. Обычно эта операция определяет эквивалентность элементов вызовом методов `GetHashCode` и

Equals типа данных элементов. Однако для объектов типа DataRow это может привести к некорректному результату.

Поскольку планируется вызывать дополнительный прототип и предоставить объект-компаратор System.Data.DataRowComparer.Default, эквивалентность элементов будет определена правильно. С его помощью дублирование строк определяется сравнением объектов DataRow, используя количество столбцов в строке и статический тип данных каждого столбца, а затем используя интерфейс IComparable на каждом столбце, если его динамический тип данных реализует этот интерфейс, либо вызывая статический метод Equals класса System.Object в противном случае.

Прототипы

Операция SequenceEqual имеет один прототип, который описан ниже.

Прототип

```
public static bool SequenceEqual<T> (
  this IEnumerable<T> first,
  IEnumerable<T> second,
  IEqualityComparer<T> comparer);
```

Примеры

В этом примере применения операции SequenceEqual строятся две идентичные последовательности объектов DataRow, которые сравниваются с помощью операции SequenceEqual с передачей ей объекта-компаратора и без передачи этого объекта. При вызове SequenceEqual с компаратором возвращается true, что указывает на эквивалентность двух последовательностей, а при вызове той же операции без компаратора возвращается значение false, указывающее на то, что последовательности не эквивалентны. В листинге 10.6 показан код.

Листинг 10.6. Операция SequenceEqual с компаратором эквивалентности и без

```
Student[] students = {
  new Student { Id = 1, Name = "Joe Rattz" },
  new Student { Id = 7, Name = "Anthony Adams" },
  new Student { Id = 13, Name = "Stacy Sinclair" },
  new Student { Id = 72, Name = "Dignan Stephens" }
};
DataTable dt1 = GetDataTable(students);
IEnumerable<DataRow> seq1 = dt1.AsEnumerable();
DataTable dt2 = GetDataTable(students);
IEnumerable<DataRow> seq2 = dt2.AsEnumerable();
bool equal = seq1.SequenceEqual(seq2, System.Data.DataRowComparer.Default);
Console.WriteLine("SequenceEqual() с компаратором : {0}", equal);
equal = seq1.SequenceEqual(seq2);
Console.WriteLine("SequenceEqual() без компаратора : {0}", equal);
```

Первый вызов должен сообщить, что две последовательности эквиваленты, а второй — что нет. Результат в точности соответствует ожиданиям:

```
SequenceEqual() с компаратором : True
SequenceEqual() без компаратора : False
```

Операции над полями DataRow

В дополнение к специфичному для DataRow классу компаратора для операций множеств существует потребность в некоторых специфичных для DataRow операциях. Эти операции определены в статическом классе System.Data.DataRowExtensions внутри сборки System.Data.DataSetExtensions.dll.

Вы наверняка заметили, что почти во всех приведенных до сих пор примерах для извлечения значения объекта DataColumn из DataRow использовалась операция Field<T>. У этой операции два предназначения: корректная проверка эквивалентности и обработка значения null.

С объектами DataRow возникает проблема. Их значения DataColumn не сравниваются правильно на предмет эквивалентности, когда обращение происходит через индексатор объекта DataRow, если столбец имеет тип значения. Причина в том, что поскольку тип данных столбца может быть любым, индексатор возвращает объект типа System.Object. Это позволяет индексатору вернуть целое число, строку или любой другой необходимый тип для этого столбца. И это значит, что если типом столбца является int, то это тип значения, который должен быть *упакован* в объект типа Object. Подобное действие в Microsoft .NET Framework так и называется — *упаковка* (boxing). Извлечение типа значения обратно из объекта известно как *распаковка* (unboxing). Именно в упаковке и кроется проблема.

Рассмотрим некоторый простой код. Для начала возьмем пример сравнения целочисленного литерала с другим целочисленным литералом, имеющим то же значение, как показано в листинге 10.7.

Листинг 10.7. Сравнение 3 и 3

```
Console.WriteLine("(3 == 3) равно {0}.", (3 == 3));
```

Результат этого кода такой:

```
(3 == 3) равно True.
```

Здесь нет никакого сюрприза. Но что случится, когда целое число упаковывается? Рассмотрим код в листинге 10.8 и посмотрим на результат его работы.

Листинг 10.8. Сравнение значения 3, приведенного к Object с другим значением 3, приведенным к Object

```
Console.WriteLine("((Object)3 == (Object)3) равно {0}.", ((Object)3 == (Object)3));
```

И вот результат:

```
((Object)3 == (Object)3) равно False.
```

Что же произошло? Дело в том, что за счет приведения целочисленного литерала 3 к типу Object было создано два объекта, и сравнивались ссылки (адреса) двух объектов, которые не эквивалентны. При обращении к объектам DataColumn с использованием индексатора объекта DataRow, если любой из столбцов имеет тип значения, то значения таких столбцов упаковываются и не могут правильно сравниваться на предмет эквивалентности.

Чтобы продемонстрировать это, будет создан более сложный пример, который действительно использует объекты DataColumn. В примере есть два массива, каждый своего типа класса. Один — тот же базовый массив студентов, что применялся ранее. Второй — массив назначений на курсы с внешними ключами, указывающими на массив студентов. Ниже показан класс StudentClass.

Простой класс с двумя общедоступными свойствами

```
class StudentClass
{
   public int Id;
   public string Class;
}
```

Имея два разных типа классов, необходим метод для преобразования этого массива в объект типа DataTable. Вот этот метод:

```
static DataTable GetDataTable2(StudentClass[] studentClasses)
{
  DataTable table = new DataTable();
  table.Columns.Add("Id", typeof(Int32));
  table.Columns.Add("Class", typeof(string));

  foreach (StudentClass studentClass in studentClasses)
  {
    table.Rows.Add(studentClass.Id, studentClass.Class);
  }
  return (table);
}
```

Этот метод — не что иное, как копия существующего метода GetTableData, который был модифицирован для работы с массивами объектов StudentClass. Очевидно, если вы собираетесь работать с массивами в реальном рабочем коде, вам понадобится нечто более абстрактное, чем создание метода для каждого типа класса, для которого нужен объект DataTable. Возможно, подходящим подходом был бы обобщенный расширяющий метод. Но как упоминалось в начале примеров, запросы LINQ to DataSet обычно будут выполняться на данных, взятых из базы, а не из массивов, так что об этом можно не беспокоиться.

Для примера построим последовательность объектов DataRow из каждого массива и попробуем соединить их по общему столбцу Id, который получается посредством индексации в DataRow с именем столбца, которым является Id. Код показан в листинге 10.9.

Листинг 10.9. Соединение двух столбцов типа значения посредством индексации в DataRow

```
Student[] students = {
  new Student { Id = 1, Name = "Joe Rattz" },
  new Student { Id = 7, Name = "Anthony Adams" },
  new Student { Id = 13, Name = "Stacy Sinclair" },
  new Student { Id = 72, Name = "Dignan Stephens" }
};
StudentClass[] classDesignations = {
  new StudentClass { Id = 1, Class = "Sophmore" },
  new StudentClass { Id = 7, Class = "Freshman" },
  new StudentClass { Id = 13, Class = "Graduate" },
  new StudentClass { Id = 72, Class = "Senior" }
};
DataTable dt1 = GetDataTable(students);
IEnumerable<DataRow> seq1 = dt1.AsEnumerable();
DataTable dt2 = GetDataTable2(classDesignations);
IEnumerable<DataRow> seq2 = dt2.AsEnumerable();
string anthonysClass = (from s in seq1
                    where s.Field<string>("Name") == "Anthony Adams"
                    from c in seq2
                    where c["Id"] == s["Id"]
```

```
                    select (string)c["Class"]).
                   SingleOrDefault<string>();
Console.WriteLine("Класс для Anthony Adams: {0}",
  anthonysClass != null ? anthonysClass : "null");
```

В этом запросе следует отметить несколько моментов. Обратите внимание на выделенную полужирным строку. Здесь объект `DataRow` индексируется для получения значений столбцов. Поскольку типом данных значений столбцов является `string`, они упаковываются, что означает потенциальную проблему определения эквивалентности. Вдобавок можно видеть, что в этом примере при сравнении поля `Name` со значением `"Anthony Adams"` используется операция `Field<T>`. Пока не обращайте на это внимания. Просто примите к сведению, что операция `Field<T>` вызывается для предотвращения проблем с упаковкой поля `Name`, которое используется наряду с полем `Id`. Также отметьте, что этот запрос комбинирует синтаксис выражений со стандартным синтаксисом точечной нотации. Как видите, выполняется также соединение двух объектов `DataTable`. Запустим код и посмотрим результат:

```
Класс для Anthony Adams: null
```

Строка `anthonysClass` равна `null`. Это потому, что соединение не может найти в `seq2` записи с эквивалентным значением поля `Id`. Причина — в упаковке поля `Id` при извлечении с использованием индексатора `DataRow`. Теперь можно обработать распаковку самостоятельно, изменив строку

```
where c["Id"] == s["Id"]
```

на

```
where (int)c["Id"] == (int)s["Id"]
```

Листинг 10.10 содержит полный пример с замененной строкой.

Листинг 10.10. Использование приведения для корректной проверки эквивалентности

```
Student[] students = {
  new Student { Id = 1, Name = "Joe Rattz" },
  new Student { Id = 7, Name = "Anthony Adams" },
  new Student { Id = 13, Name = "Stacy Sinclair" },
  new Student { Id = 72, Name = "Dignan Stephens" }
};

StudentClass[] classDesignations = {
  new StudentClass { Id = 1, Class = "Sophmore" },
  new StudentClass { Id = 7, Class = "Freshman" },
  new StudentClass { Id = 13, Class = "Graduate" },
  new StudentClass { Id = 72, Class = "Senior" }
};

DataTable dt1 = GetDataTable(students);
IEnumerable<DataRow> seq1 = dt1.AsEnumerable();
DataTable dt2 = GetDataTable2(classDesignations);
IEnumerable<DataRow> seq2 = dt2.AsEnumerable();

string anthonysClass = (from s in seq1
                    where s.Field<string>("Name") == "Anthony Adams"
                    from c in seq2
                    where (int)c["Id"] == (int)s["Id"]
                    select (string)c["Class"]).
                   SingleOrDefault<string>();
Console.WriteLine("Класс для Anthony Adams: {0}",
  anthonysClass != null ? anthonysClass : "null");
```

После запуска этого кода получается следующий результат:

```
Класс для Anthony Adams: Freshman
```

Это решает проблему упаковки. Однако остается еще одна проблема. Когда вы пытаетесь извлечь значение столбца, используя индексатор объекта DataRow, помните, что значение столбца возвращается как объект типа Object. Поэтому для того, чтобы сравнить его с любым значением или присвоить переменной, необходимо привести его к другому типу данных, как это было сделано путем приведения к int. Поскольку объекты DataSet используют значение DBNull.Value в качестве значения столбца null, то если значением столбца является DBNull.Value, то приведение его к другому типу данных вызовет генерацию исключения.

К счастью, появление LINQ to DataSet привело к исчезновению обеих этих проблем — сравнения упакованных значений и обработки null — благодаря операциям Field<T> и SetField<T>. В листинге 10.11 показан предыдущий пример с использованием операции Field<T>.

Листинг 10.11. Использование операции Field

```
Student[] students = {
  new Student { Id = 1, Name = "Joe Rattz" },
  new Student { Id = 7, Name = "Anthony Adams" },
  new Student { Id = 13, Name = "Stacy Sinclair" },
  new Student { Id = 72, Name = "Dignan Stephens" }
};
StudentClass[] classDesignations = {
  new StudentClass { Id = 1, Class = "Sophmore" },
  new StudentClass { Id = 7, Class = "Freshman" },
  new StudentClass { Id = 13, Class = "Graduate" },
  new StudentClass { Id = 72, Class = "Senior" }
};
DataTable dt1 = GetDataTable(students);
IEnumerable<DataRow> seq1 = dt1.AsEnumerable();
DataTable dt2 = GetDataTable2(classDesignations);
IEnumerable<DataRow> seq2 = dt2.AsEnumerable();
string anthonysClass = (from s in seq1
                    where s.Field<string>("Name") == "Anthony Adams"
                    from c in seq2
                    where c.Field<int>("Id") == s.Field<int>("Id")
                    select (string)c["Class"]).
                   SingleOrDefault<string>();
Console.WriteLine("Класс для Anthony Adams: {0}",
  anthonysClass != null ? anthonysClass : "null");
```

Этот код почти такой же, как в предыдущем примере, за исключением вызова операции Field<T> вместо приведения поля к типу int. Вот результат:

```
Класс для Anthony Adams: Freshman
```

Field<T>

Как только что было показано в листинге 10.11, операция Field<T> позволяет получить значение столбца из объекта DataRow и устраняет описанные выше проблемы приведения DBNull.Value и сравнения упакованных значений.

Прототипы

Операция Field имеет шесть прототипов, которые описаны ниже.

Первый прототип возвращает значение столбца для DataColumn и указанной версии.

Первый прототип Field

```
public static T Field (
  this DataRow first,
  System.Data.DataColumn column,
  System.Data.DataRowVersion version);
```

Второй прототип возвращает значение столбца с указанным именем и версией.

Второй прототип Field

```
public static T Field (
  this DataRow first,
  string columnName,
  System.Data.DataRowVersion version);
```

Третий прототип возвращает значение указанного по порядку столбца заданной версии.

Третий прототип Field

```
public static T Field (
  this DataRow first,
  int ordinal,
  System.Data.DataRowVersion version);
```

Четвертый прототип возвращает текущее значение столбца только для указанного DataColumn.

Четвертый прототип Field

```
public static T Field (
  this DataRow first,
  System.Data.DataColumn column);
```

Пятый прототип возвращает текущее значение только столбца с указанным именем.

Пятый прототип Field

```
public static T Field (
  this DataRow first,
  string columnName);
```

Шестой прототип возвращает текущее значение только для столбца с указанным порядковым номером.

Шестой прототип Field

```
public static T Field (
  this DataRow first,
  int ordinal);
```

Как вы, возможно, заметили, первые три прототипа позволяют указать, какие версии DataRowVersion значений объектов DataColumn должны быть извлечены.

Примеры

До сих пор вы уже не раз сталкивались с разнообразными вызовами операции Field<T>. Но только теперь вы можете увидеть все прототипы в действии. В листинге 10.12 показан тривиальный пример каждого из них.

Листинг 10.12. Пример вызова каждого прототипа операции `Field`

```
Student[] students = {
  new Student { Id = 1, Name = "Joe Rattz" },
  new Student { Id = 7, Name = "Anthony Adams" },
  new Student { Id = 13, Name = "Stacy Sinclair" },
  new Student { Id = 72, Name = "Dignan Stephens" }
};
DataTable dt1 = GetDataTable(students);
IEnumerable<DataRow> seq1 = dt1.AsEnumerable();
int id;
// Применение прототипа 1.
id = (from s in seq1
     where s.Field<string>("Name") == "Anthony Adams"
     select s.Field<int>(dt1.Columns[0], DataRowVersion.Current)).
    Single<int>();
Console.WriteLine("Значение Id для Anthony Adams, полученное с помощью первого
прототипа: {0}", id);
// Применение прототипа 2.
id = (from s in seq1
     where s.Field<string>("Name") == "Anthony Adams"
     select s.Field<int>("Id", DataRowVersion.Current)).
    Single<int>();
Console.WriteLine("Значение Id для Anthony Adams, полученное с помощью второго
прототипа: {0}", id);
// Применение прототипа 3.
id = (from s in seq1
     where s.Field<string>("Name") == "Anthony Adams"
     select s.Field<int>(0, DataRowVersion.Current)).
    Single<int>();
Console.WriteLine("Значение Id для Anthony Adams, полученное с помощью третьего
прототипа ", id);
// Применение прототипа 4.
id = (from s in seq1
     where s.Field<string>("Name") == "Anthony Adams"
     select s.Field<int>(dt1.Columns[0])).
    Single<int>();
Console.WriteLine("Значение Id для Anthony Adams, полученное с помощью четвертого
прототипа: {0}", id);
// Применение прототипа 5.
id = (from s in seq1
     where s.Field<string>("Name") == "Anthony Adams"
     select s.Field<int>("Id")).
    Single<int>();
Console.WriteLine("Значение Id для Anthony Adams, полученное с помощью пятого
прототипа: {0}", id);
// Применение прототипа 6.
id = (from s in seq1
     where s.Field<string>("Name") == "Anthony Adams"
     select s.Field<int>(0)).
    Single<int>();
Console.WriteLine("Значение Id для Anthony Adams, полученное с помощью шестого
прототипа: {0}", id);
```

В коде объявляется массив студентов, из которого создается объект `DataTable`, как и в большинстве других примеров. Затем также получается последовательность объектов `DataRow`. После этого друг за другом используются прототипы операции `Field<T>`

для получения поля по имени Id. Обратите внимание, что в каждом запросе поля Id также применяется операция Field<T>, в части Where запроса. И вот результат:

```
Значение Id для Anthony Adams, полученное с помощью первого прототипа: 7
Значение Id для Anthony Adams, полученное с помощью второго прототипа: 7
Значение Id для Anthony Adams, полученное с помощью третьего прототипа: 7
Значение Id для Anthony Adams, полученное с помощью четвертого прототипа: 7
Значение Id для Anthony Adams, полученное с помощью пятого прототипа: 7
Значение Id для Anthony Adams, полученное с помощью шестого прототипа: 7
```

Прежде чем переходить к операции SetField<T>, рассмотрим пример, демонстрирующий один из прототипов, который позволяет указывать значение DataRowVersion объекта DataColumn, подлежащего извлечению. Для целей этого примера понадобится модифицировать значение одного из объектов DataColumn, используя для этого операцию SetField<T>. Операция SetField<T> подробно обсуждается в следующем разделе.

Поскольку целью этой главы является объяснение операций LINQ to DataSet, а не детальное обсуждение работы класса DataSet, пара дополнительных методов DataSet, которые вызываются в этом примере, упоминаются лишь кратко. Код приведен в листинге 10.13.

Листинг 10.13. Прототип операции Field с указанным DataRowVersion

```
Student[] students = {
  new Student { Id = 1, Name = "Joe Rattz" },
  new Student { Id = 7, Name = "Anthony Adams" },
  new Student { Id = 13, Name = "Stacy Sinclair" },
  new Student { Id = 72, Name = "Dignan Stephens" }
};
DataTable dt1 = GetDataTable(students);
IEnumerable<DataRow> seq1 = dt1.AsEnumerable();
DataRow row = (from s in seq1
             where s.Field<string>("Name") == "Anthony Adams"
             select s).Single<DataRow>();
row.AcceptChanges();
row.SetField("Name", "George Oscar Bluth");
Console.WriteLine("Исходное значение = {0} : Текущее значение = {1}",
  row.Field<string>("Name", DataRowVersion.Original),
  row.Field<string>("Name", DataRowVersion.Current));
row.AcceptChanges();
Console.WriteLine("Исходное значение = {0} : Текущее значение = {1}",
  row.Field<string>("Name", DataRowVersion.Original),
  row.Field<string>("Name", DataRowVersion.Current));
```

В этом примере получается последовательность из массива студентов, обычным образом. Затем запрашивается единственный объект DataRow, в котором можно провести некоторые изменения. Первое, что представляет интерес — метод AcceptChanges, вызванный после получения объекта DataRow. Этот метод вызывается, чтобы заставить объект DataRow принять текущее значение каждого объекта DataColumn внутри исходной версии. Без этого никакой исходной версии значений объектов DataColumn не было бы, и попытка обратиться к исходной версии поля вызвала бы генерацию исключения. Таким образом, объект DataRow готов начать отслеживать изменение значений объектов DataColumn. Это нужно для того, чтобы можно было получить разные версии DataRowVersion значений DataColumn объекта DataRow.

После вызова метода AcceptChanges первый раз поле устанавливается посредством операции SetField. Затем на консоль выводится исходная и текущая версия значения Name объекта DataColumn. В этой точке исходная версия должна быть "Anthony

Adams", а текущая — "George Oscar Bluth". Это позволяет увидеть разные версии, которые можно получить из объекта DataRow.

После этого, чтобы стало еще интереснее, метод AcceptChanges вызывается второй раз, и снова отображаются исходная и текущая версии значения объекта DataColumn. На этот раз исходная и текущая версии должны совпадать и содержать строку "George Oscar Bluth", потому что объекту DataRow было указано принять изменения в качестве текущей версии. Посмотрим результат:

```
Исходное значение = Anthony Adams : Текущее значение = George Oscar Bluth
Исходное значение = George Oscar Bluth : Текущее значение = George Oscar Bluth
```

Все работает. Однако помните, что без первоначального вызова метода AcceptChanges можно было бы сколько угодно раз изменять значение объекта DataColumn, но никакой исходной версии не было бы.

Также ранее упоминалось о том, что одним из дополнительных преимуществ операции Field<T> является ее способность справляться с ситуацией, когда поля содержат значение null. Взглянем на пример в листинге 10.14, где имя студента имеет значение null, но операция Field<T> не используется.

Листинг 10.14. Пример без операции Field, когда присутствует значение null

```
Student[] students = {
  new Student { Id = 1, Name = "Joe Rattz" },
  new Student { Id = 7, Name = null },
  new Student { Id = 13, Name = "Stacy Sinclair" },
  new Student { Id = 72, Name = "Dignan Stephens" }
};
DataTable dt1 = GetDataTable(students);
IEnumerable<DataRow> seq1 = dt1.AsEnumerable();
string name = seq1.Where(student => student.Field<int>("Id") == 7)
  .Select(student => (string)student["Name"])
  .Single();
Console.WriteLine("Имя студента: '{0}'", name);
```

Это чрезвычайно простой пример. Обратите внимание, что значение члена Name записи Student с Id, равным 7, устанавливается в null. Кроме того, вместо использования операции Field<T> производится обращение по индексу к значению внутри DataRow и приведение его к типу string. Посмотрим на результат:

```
Unhandled Exception: System.InvalidCastException: Unable to cast object of type
'System.DBNull' to type 'System.String'.
```

Необработанное исключение: System.InvalidCastException: Невозможно привести объект типа 'System.DBNull' к типу 'System.String'.

...

Так что же произошло? А случилось то, что значение объекта DataColumn равно DBNull, и привести его к string нельзя. Существуют довольно громоздкие решения, которые позволили бы избежать этой сложности, но специально для упрощения предусмотрена операция Field<T>. Рассмотрим такой же пример, но на этот раз с применением операции Field<T> для получения значения объекта DataColumn. Код представлен в листинге 10.15.

Листинг 10.15. Пример с операцией Field, когда присутствует значение null

```
Student[] students = {
  new Student { Id = 1, Name = "Joe Rattz" },
  new Student { Id = 7, Name = null },
```

```
  new Student { Id = 13, Name = "Stacy Sinclair" },
  new Student { Id = 72, Name = "Dignan Stephens" }
};
DataTable dt1 = GetDataTable(students);
IEnumerable<DataRow> seq1 = dt1.AsEnumerable();
string name = seq1.Where(student => student.Field<int>("Id") == 7)
  .Select(student => student.Field<string>("Name"))
  .Single();
Console.WriteLine("Имя студента: '{0}'", name);
```

Все то же самое, но теперь вместо приведения к `string` используется операция `Field<T>`. Посмотрим, что получилось:

```
Имя студента: ''
```

С этим гораздо легче иметь дело.

SetField<T>

Как и в случае извлечения объектов `DataColumn`, значение `null` неблагоприятно влияет и на установку объектов `DataColumn`. Чтобы помочь справиться с этой проблемой, была создана операция `SetField<T>`. Она применяется в ситуации, когда устанавливается значение объекта `DataColumn` из допускающего `null` типа данных, текущее значение которого равно `null`.

Прототипы

Операция `SetField<T>` имеет три прототипа, описанные ниже.

Первый прототип позволяет устанавливать текущее значение столбца для указанного объекта `DataColumn`.

Первый прототип SetField<T>

```
public static void SetField (
  this DataRow first,
  System.Data.DataColumn column,
  T value);
```

Второй прототип позволяет устанавливать текущее значение столбца с указанным именем.

Второй прототип SetField<T>

```
public static void SetField (
  this DataRow first,
  string columnName,
  T value);
```

Третий прототип позволяет вам устанавливать текущее значение столбца с указанным порядковым номером.

Третий прототип SetField<T>

```
public static void SetField (
  this DataRow first,
  int ordinal,
  T value);
```

Примеры

В примере применения операции `SetField<T>`, приведенном в листинге 10.16, сначала отображается последовательность объектов `DataRow`, содержащих сведения о сту-

дентах. Затем из этой последовательности запрашивается один из студентов по имени, после чего имя изменяется с использованием операции `SetField<T>`. После проведенных изменений последовательность объектов `DataRow` снова отображается. Далее все повторяется с каждым прототипом.

Листинг 10.16. Пример с применением каждого прототипа операции `SetField`

```
Student[] students = {
  new Student { Id = 1, Name = "Joe Rattz" },
  new Student { Id = 7, Name = "Anthony Adams"},
  new Student { Id = 13, Name = "Stacy Sinclair" },
  new Student { Id = 72, Name = "Dignan Stephens" }
};

DataTable dt1 = GetDataTable(students);
IEnumerable<DataRow> seq1 = dt1.AsEnumerable();
Console.WriteLine("{0}Результаты перед вызовом всех прототипов:",
  System.Environment.NewLine);
foreach (DataRow dataRow in seq1)
{
  Console.WriteLine("Студент с Id = {0} - {1}", dataRow.Field<int>("Id"),
  dataRow.Field<string>("Name"));
}

// Применение прототипа 1.
(from s in seq1
 where s.Field<string>("Name") == "Anthony Adams"
 select s).Single<DataRow>().SetField(dt1.Columns[1], "George Oscar Bluth");
Console.WriteLine("{0}Результаты после вызова первого прототипа:",
  System.Environment.NewLine);
foreach (DataRow dataRow in seq1)
{
  Console.WriteLine("Студент с Id = {0} - {1}", dataRow.Field<int>("Id"),
  dataRow.Field<string>("Name"));
}

// Применение прототипа 2.
(from s in seq1
 where s.Field<string>("Name") == "George Oscar Bluth"
 select s).Single<DataRow>().SetField("Name", "Michael Bluth");
Console.WriteLine("{0}Результаты после вызова второго прототипа:",
  System.Environment.NewLine);
foreach (DataRow dataRow in seq1)
{
  Console.WriteLine("Студент с Id = {0} - {1}", dataRow.Field<int>("Id"),
  dataRow.Field<string>("Name"));
}

// Применение прототипа 3.
(from s in seq1
 where s.Field<string>("Name") == "Michael Bluth"
 select s).Single<DataRow>().SetField("Name", "Tony Wonder");
Console.WriteLine("{0}Результаты после вызова третьего прототипа:",
  System.Environment.NewLine);

foreach (DataRow dataRow in seq1)
{
  Console.WriteLine("Студент с Id = {0} - {1}", dataRow.Field<int>("Id"),
  dataRow.Field<string>("Name"));
}
```

Этот код не так плох, как может показаться. После получения последовательности студентов и ее отображения на консоли, идет блок кода, повторяющийся три раза — по одному для каждого прототипа. Каждый блок содержит запрос LINQ, извлекающий поле и обновляющий его значение, отображает строку заголовка на консоли, а затем — каждую строку в последовательности, чтобы показать изменение, проведенное в поле.

В этом примере следует отметить две вещи. В каждом запросе LINQ, где запрашивается DataRow по полю Name, синтаксис выражений запросов смешивается со стандартным синтаксисом точечной нотации. Также используется операция Flat<T> для нахождения записи, которая должна быть установлена операцией SetField<T>. После получения последовательности объектов DataRow студентов с ними производится работа с использованием всех прототипов SetField<T>. На протяжении примера ранее измененный элемент запрашивается по значению и снова изменяется. Например, для первого прототипа просто запрашивается элемент, поле Name которого равно "Anthony Adams", и это поле устанавливается в "George Oscar Bluth". Для второго прототипа запрашивается поле со значением "George Oscar Bluth" в Name и затем изменяется на что-то другое, что будет запрошено в следующем прототипе. После обновления значения каждого элемента последовательность выводится на консоль для подтверждения, что значение этого элемента действительно было обновлено.

Одна из изюминок этого примера состоит в том, что запрос элемента и обновление его значения производится в единственном операторе. Это настолько мощно, что может показаться какой-то иллюзией, но как будет показано далее, никакой магии здесь нет.

Ниже приведен вывод этого примера:

```
Результаты перед вызовом всех прототипов:
Студент с Id = 1 - Joe Rattz
Студент с Id = 7 - Anthony Adams
Студент с Id = 13 - Stacy Sinclair
Студент с Id = 72 - Dignan Stephens

Результаты после вызова первого прототипа:
Студент с Id = 1 - Joe Rattz
Студент с Id = 7 - George Oscar Bluth
Студент с Id = 13 - Stacy Sinclair
Студент с Id = 72 - Dignan Stephens

Результаты после вызова второго прототипа:
Студент с Id = 1 - Joe Rattz
Студент с Id = 7 - Michael Bluth
Студент с Id = 13 - Stacy Sinclair
Студент с Id = 72 - Dignan Stephens

Результаты после вызова третьего прототипа:
Студент с Id = 1 - Joe Rattz
Студент с Id = 7 - Tony Wonder
Студент с Id = 13 - Stacy Sinclair
Студент с Id = 72 - Dignan Stephens
```

Как видите, поле Name соответствующего элемента обновляется каждый раз.

Операции DataTable

В дополнение к ориентированным на DataRow операциям в классе DataRowExtensions предусмотрены также некоторые операции, специфичные для DataTable. Эти операции определены в статическом классе System.Data.DataTableExtensions внутри сборки System.Data.Entity.dll.

AsEnumerable

Возможно, вы удивитесь, встретив здесь операцию AsEnumerable. Эта операция специально предназначена для класса DataTable и возвращает последовательность объектов DataRow. Хотя она не упоминалась до сих пор, но эта операция вызывалась почти в каждом примере.

Если вы посмотрите на статический класс System.Data.DataTableExtensions, то увидите, что там есть операция AsEnumerable. Назначение этой операции — возвращать последовательность типа IEnumerable<DataRow> из объекта DataTable.

Прототипы

Операция AsEnumerable имеет один прототип, описанный ниже.

Прототип AsEnumerable

```
public static IEnumerable<DataRow> AsEnumerable (
  this DataTable source
);
```

Эта операция, будучи вызванной на объекте DataTable, возвращает последовательность объектов DataRow. Обычно так выглядит первый шаг при выполнении запроса LINQ to DataSet на DataTable объекта DataSet. За счет вызова этой операции можно получить последовательность IEnumerable<T>, где T является DataRow, что позволяет вызывать множество операций LINQ, которые могут быть вызваны на последовательности типа IEnumerable<T>.

Примеры

Поскольку вызов операции AsEnumerable — это первый шаг при выполнении запроса LINQ to DataSet, почти в каждом примере этой главы вызывается операцию AsEnumerable. Поэтому нет необходимости приводить здесь еще один.

CopyToDataTable<DataRow>

Теперь, когда известно, как запрашивать и модифицировать значение DataColumn объекта DataRow, может возникнуть вопрос, а как поместить эту последовательность модифицированных объектов DataRow в DataTable? Операция CopyToDataTable предназначена именно для этой цели.

Прототипы

Операция CopyToDataTable имеет два прототипа, которые описаны ниже.

Первый прототип вызывается на IEnumerable<DataRow> и возвращает DataTable. Он используется для создания нового объекта DataTable из последовательности объектов DataRow.

Первый прототип CopyToDataTable

```
public static DataTable CopyToDataTable<T> (
  this IEnumerable<T> source
) where T : DataRow;
```

Первый прототип автоматически устанавливает исходные версии для каждого поля, не требуя явного вызова метода AcceptChanges.

Второй прототип вызывается на IEnumerable<DataRow> исходного DataTable, чтобы обновить уже существующий целевой объект DataTable, на основе указанного значения LoadOptions.

Второй прототип CopyToDataTable

```
public static void CopyToDataTable<T> (
  this IEnumerable<T> source,
  DataTable table,
  LoadOption options
) where T : DataRow;
```

Значение переданного аргумента LoadOptions информирует операцию о том, что должны быть изменены только *исходные* значения столбцов, *текущие* значения либо те и другие. Это полезно при управлении изменениями DataTable. Для LoadOption доступны описанные ниже значения.

- OverwriteChanges: в каждом столбце будут обновлены и исходное значение и текущее.
- PreserveChanges: обновляется только исходное значение каждого столбца.
- Upsert: обновляется только текущее значение каждого столбца.

Однако этот аргумент LoadOption создает небольшую проблему. Обратите внимание, что описание каждого возможного значения ссылается на обновление значений столбца. Это, конечно же, означает также обновление столбцов записи, уже находящейся в целевом DataTable. Как операция CopyToDataTable узнает, какая именно запись в целевом DataTable соответствует записи в исходном DataTable? Другими словами, когда операция CopyToDataTable пытается копировать запись из исходного DataTable в целевой и получает параметр LoadOption, каким образом она узнает, следует просто добавить запись из исходного DataTable или же обновить уже существующую запись целевого DataTable? Ответ — никак, если только ей ничего не известно о полях первичного ключа в DataTable.

Поэтому для правильной работы этого прототипа операции CopyDataTable целевой объект DataTable должен иметь соответствующую информацию о полях первичного ключа. Если не указать первичный ключ, этот прототип добавит все записи из исходного объекта DataTable в целевой объект DataTable.

Здесь есть одна дополнительная сложность. Поскольку, используя этот прототип, вы, возможно, заинтересованы в сравнении исходной и текущей версий значений полей, не забудьте, что с этим прототипом операции CopyDataTable поле не имеет исходной версии, если только не был вызван метод AcceptChanges. Попытка обратиться к исходной версии, когда она не существует, приведет к генерации исключения. Тем не менее, перед попыткой получить доступ к исходной версии на каждом объекте DataRow можно вызвать метод HasVersion, чтобы узнать, существует ли она, и предотвратить генерацию исключения этого типа.

Примеры

В примере применения первого прототипа операции CopyToDataTable сначала модифицируется поле в DataTable, с помощью операции CopyToDataTable создается новый объект DataTable из модифицированного, после чего содержимое нового объекта DataTable отображается на консоль. Код представлен в листинге 10.17.

Листинг 10.17. Вызов первого прототипа операции CopyToDataTable

```
Student[] students = {
  new Student { Id = 1, Name = "Joe Rattz" },
  new Student { Id = 7, Name = "Anthony Adams"},
  new Student { Id = 13, Name = "Stacy Sinclair" },
  new Student { Id = 72, Name = "Dignan Stephens" }
};
```

```
DataTable dt1 = GetDataTable(students);
Console.WriteLine("Исходный объект DataTable:");
foreach (DataRow dataRow in dt1.AsEnumerable())
{
  Console.WriteLine("Студент с Id = {0} - {1}", dataRow.Field<int>("Id"),
  dataRow.Field<string>("Name"));
}
(from s in dt1.AsEnumerable()
 where s.Field<string>("Name") == "Anthony Adams"
 select s).Single<DataRow>().SetField("Name", "George Oscar Bluth");
DataTable newTable = dt1.AsEnumerable().CopyToDataTable();
Console.WriteLine("{0}Новый объект DataTable:", System.Environment.NewLine);
foreach (DataRow dataRow in newTable.AsEnumerable())
{
  Console.WriteLine("Student Id = {0} is {1}", dataRow.Field<int>("Id"),
    dataRow.Field<string>("Name"));
}
```

Итак, сначала создается объект DataTable из массива студентов, как обычно делалось в предыдущих примерах. Затем содержимое DataTable отображается на консоль. После этого модифицируется поле Name одного из объектов DataRow. Затем вызовом операции CopyToDataTable создается новый объект DataTable. И, наконец, содержимое вновь созданного объекта DataTable отображается на консоль.

Ниже показаны результаты:

```
Исходный объект DataTable:
Студент с Id = 1 - Joe Rattz
Студент с Id = 7 - Anthony Adams
Студент с Id = 13 - Stacy Sinclair
Студент с Id = 72 - Dignan Stephens

Новый объект DataTable:
Студент с Id = 1 - Joe Rattz
Студент с Id = 7 - George Oscar Bluth
Студент с Id = 13 - Stacy Sinclair
Студент с Id = 72 - Dignan Stephens
```

Как видите, в новом объекте DataTable имеются данные в модифицированной версии, чего и следовало ожидать.

В следующем примере демонстрируется применение второго прототипа операции CopyToDataTable. Как упоминалось ранее, для правильной работы аргумента LoadOption в целевом объекте DataSet должен быть определен первичный ключ. В данном примере они не устанавливаются, чтобы можно было увидеть поведение без ключей. Поскольку этот пример несколько сложнее, приводятся дополнительные пояснения. Код представлен в листинге 10.18.

Листинг 10.18. Вызов второго прототипа операции CopyToDataTable, когда первичный ключ не установлен

```
Student[] students = {
  new Student { Id = 1, Name = "Joe Rattz" },
  new Student { Id = 7, Name = "Anthony Adams"},
  new Student { Id = 13, Name = "Stacy Sinclair" },
  new Student { Id = 72, Name = "Dignan Stephens" }
};

DataTable dt1 = GetDataTable(students);
DataTable newTable = dt1.AsEnumerable().CopyToDataTable();
```

Пока ничего нового. Из массива students создан исходный объект DataTable. Целевой объект DataTable будет создан вызовом операции CopyToDataTable на исходном DataTable. Обратите внимание, что поскольку вызван первый прототип операции CopyToDataTable, незачем вызывать метод AcceptChanges на целевом DataTable. Это важно отметить, потому что в следующем сегменте кода производится ссылка на исходную версию поля Name. Если бы не тот факт, что первый прототип операции CopyToDataTable устанавливает исходные версии полей, в этой ситуации было бы сгенерировано исключение.

```
Console.WriteLine("Перед обновлением DataTable:");
foreach (DataRow dataRow in newTable.AsEnumerable())
{
  Console.WriteLine("Студент с Id = {0} : исходное {1} : текущее {2}",
    dataRow.Field<int>("Id"),
    dataRow.Field<string>("Name", DataRowVersion.Original),
    dataRow.Field<string>("Name", DataRowVersion.Current));
}
```

Здесь нет ничего особенного за исключением ссылки на исходную версию поля Name в записи и отсутствия генерации исключения при этом, поскольку данный прототип операции CopyToDataTable устанавливает исходную версию полей автоматически.

```
(from s in dt1.AsEnumerable()
 where s.Field<string>("Name") == "Anthony Adams"
 select s).Single<DataRow>().SetField("Name", "George Oscar Bluth");
dt1.AsEnumerable().CopyToDataTable(newTable, LoadOption.Upsert);
```

Это важный сегмент кода в данном примере. Обратите внимание, что с использованием операции SetField<T> изменяется значение поля Name для одной записи в исходном объекте DataTable. Затем вызывается операция CopyToDataTable с указанием, что должно произойти копирование типа LoadOption.Upsert, имея в виду обновление только текущей версии. Это вызывает проблему. Из-за того, что вызван второй прототип операции CopyToDataTable, который не устанавливает исходные версии записей, вставляемых в базу данных, а также не вызван метод AcceptChanges, попытка обратиться к исходным версиям вставленных записей приводит к генерации исключения. Чтобы предотвратить это в случае вставки любой записи, понадобится применить метод HasVersion. Поскольку первичный ключ не указан, *известно*, что все записи исходной таблицы будут вставлены в целевую таблицу.

```
Console.WriteLine("{0}После обновления DataTable:", System.Environment.NewLine);
foreach (DataRow dataRow in newTable.AsEnumerable())
{
  Console.WriteLine("Студент с Id = {0} : исходное {1} : текущее {2}",
  dataRow.Field<int>("Id"),
  dataRow.HasVersion(DataRowVersion.Original) ?
    dataRow.Field<string>("Name", DataRowVersion.Original) : "-не существует-",
  dataRow.Field<string>("Name", DataRowVersion.Current));
}
```

В этом сегменте кода содержимое DataTable отображается на консоль. Так как первичный ключ для целевой таблицы не указан, при копировании дубликаты записей не определяются, поэтому все скопированные записи из исходного объекта DataTable будут добавлены в целевой объект DataTable.

Также обратите внимание, что обращение к исходной версии данных поля происходит, только если метод HasVersion возвращает true, указывая на наличие исходной версии. Вот результат:

```
Перед обновлением DataTable:
Студент с Id = 1 : исходное Joe Rattz : текущее Joe Rattz
Студент с Id = 7 : исходное Anthony Adams : текущее Anthony Adams
Студент с Id = 13 : исходное Stacy Sinclair : текущее Stacy Sinclair
Студент с Id = 72 : исходное Dignan Stephens : текущее Dignan Stephens

После обновления DataTable:
Студент с Id = 1 : исходное Joe Rattz : текущее Joe Rattz
Студент с Id = 7 : исходное Anthony Adams : текущее Anthony Adams
Студент с Id = 13 : исходное Stacy Sinclair : текущее Stacy Sinclair
Студент с Id = 72 : исходное Dignan Stephens : текущее Dignan Stephens
Студент с Id = 1 : исходное -не существует- : текущее Joe Rattz
Студент с Id = 7 : исходное -не существует- : текущее George Oscar Bluth
Студент с Id = 13 : исходное -не существует- : текущее Stacy Sinclair
Студент с Id = 72 : исходное -не существует- : текущее Dignan Stephens
```

Как видите, несколько записей теперь дублированы, поскольку в целевом `DataTable` не был указан первичный ключ. Даже только что обновленная запись теперь встречается дважды в `DataTable`.

Может вызвать удивление, зачем возиться с вызовом метода `HasVersion`, если можно было просто вызвать метод `AcceptChanges`? Если это сделать, то все текущие значения полей станут исходными версиями значений, и невозможно будет узнать, какие записи были изменены. В рассматриваемых примерах нужно, чтобы при изменении записи значения исходной и текущей версий отличались.

Решение проблемы предыдущего примера заключается в указании первичного ключа целевого объекта `DataTable`. В листинге 10.19 приведен пример, аналогичный предыдущему, но на этот раз первичный ключ установлен.

Листинг 10.19. Вызов второго прототипа операции `CopyToDataTable`, когда первичный ключ установлен

```
Student[] students = {
  new Student { Id = 1, Name = "Joe Rattz" },
  new Student { Id = 7, Name = "Anthony Adams"},
  new Student { Id = 13, Name = "Stacy Sinclair" },
  new Student { Id = 72, Name = "Dignan Stephens" }
};
DataTable dt1 = GetDataTable(students);
DataTable newTable = dt1.AsEnumerable().CopyToDataTable();
newTable.PrimaryKey = new DataColumn[] { newTable.Columns[0] };
Console.WriteLine("Перед обновлением DataTable:");
foreach (DataRow dataRow in newTable.AsEnumerable())
{
  Console.WriteLine("Студент с Id = {0} : исходное {1} : текущее {2}",
    dataRow.Field<int>("Id"),
    dataRow.Field<string>("Name", DataRowVersion.Original),
    dataRow.Field<string>("Name", DataRowVersion.Current));
}
(from s in dt1.AsEnumerable()
 where s.Field<string>("Name") == "Anthony Adams"
 select s).Single<DataRow>().SetField("Name", "George Oscar Bluth");
dt1.AsEnumerable().CopyToDataTable(newTable, LoadOption.Upsert);
Console.WriteLine("{0}После обновления DataTable:", System.Environment.NewLine);
foreach (DataRow dataRow in newTable.AsEnumerable())
{
  Console.WriteLine("Студент с Id = {0} : исходное {1} : текущее {2}",
  dataRow.Field<int>("Id"),
  dataRow.HasVersion(DataRowVersion.Original) ?
```

```
   dataRow.Field<string>("Name", DataRowVersion.Original) : "-does not exist-",
  dataRow.Field<string>("Name", DataRowVersion.Current));
}
```

Единственное различие между этим примером и предыдущим в том, что добавлена строка, устанавливающая первичный ключ в DataTable по имени newTable. И вот результат:

```
Перед обновлением DataTable:
Студент с Id = 1 : исходное Joe Rattz : текущее Joe Rattz
Студент с Id = 7 : исходное Anthony Adams : текущее Anthony Adams
Студент с Id = 13 : исходное Stacy Sinclair : текущее Stacy Sinclair
Студент с Id = 72 : исходное Dignan Stephens : текущее Dignan Stephens

После обновления DataTable:
Студент с Id = 1 : исходное Joe Rattz : текущее Joe Rattz
Студент с Id = 7 : исходное Anthony Adams : текущее George Oscar Bluth
Студент с Id = 13 : исходное Stacy Sinclair : текущее Stacy Sinclair
Студент с Id = 72 : исходное Dignan Stephens : текущее Dignan Stephens
```

Это уже больше похоже на то, что нужно. Обратите внимание, что студент с Id, равным 7, у которого было имя Anthony Adams, теперь имеет имя George Oscar Bluth. Именно это и было целью примера.

Резюме

В этой главе было показано, как использовать операции IEnumerable для выполнения операций с множествами над объектами DataRow, а также как получать и устанавливать значения полей посредством операции Field<T> и SetField<T>. Кроме того, объяснялось, что может пойти не так, если не использовать специфичные прототипы операций с множествами для DataRow. Комбинируя стандартные операции запросов LINQ to Objects со специфичными операциями DataSet, можно строить мощные запросы LINQ к объектам DataSet.

В следующей главе рассматриваются запросы типизированных объектов DataSet с помощью LINQ, а также приводится пример реального запроса LINQ to DataSet к базе данных.

ГЛАВА 11

Дополнительные возможности LINQ to DataSet

В предыдущей главе приводились многочисленные примеры опроса объектов DataTable, которые в реальных приложениях естественным образом поступают от типичных DataSet. Но для простоты объекты DataTable создавались программно, используя объявление статического массива. Однако запросы DataSet — это нечто большее, чем просто создание объектов DataTable из статически объявленных массивов.

К тому же примеры из предыдущей главы выполнялись на нетипизированных DataSet. Иногда может возникнуть необходимость выполнения запроса к типизированному объекту DataSet. API-интерфейс LINQ to DataSet это также позволяет.

В этой главе будет показано, как решается эта задача с помощью LINQ to DataSet. Сначала рассматриваются запросы к типизированным DataSet посредством LINQ to DataSet. Затем, поскольку было указано, что чаще приходится опрашивать DataSet, чем программно создавать объекты DataTable, будет приведен реальный пример запроса базы данных с использованием LINQ to DataSet.

Необходимые пространства имен

Примеры в этой главе ссылаются на классы из пространств имен System.Data, System.Data.SqlClient и System.Linq. Если в коде отсутствуют следующие директивы, добавьте их:

```
using System.Data;
using System.Data.SqlClient;
using System.Linq;
```

Типизированные DataSet

Типизированные DataSet могут быть опрошены с использованием LINQ, как это возможно и с нетипизированными. Однако типизированные DataSet позволяют упростить код запросов LINQ и сделать его более читабельным. Поскольку существует класс для DataSet, при запросе к типизированному DataSet можно обращаться к именам таблиц и столбцов, используя свойства типизированного класса DataSet, вместо индексации в коллекции Tables или применения операций Field<T> и SetField<T>.

Поэтому вместо следующего обращения к таблице объекта DataSet по имени Students:

```
DataTable Students = dataSet.Tables["Students"];
```

можно обратиться так:

```
DataTable Students = dataSet.Students;
```

Вместо получения значения поля вида:

```
dataRow.Field<string>("Name")
```

его можно получить следующим образом:

```
dataRow.Name
```

Это без сомнения делает код более читабельным и упрощает его сопровождение.

Прежде чем продемонстрировать пример, понадобится создать типизированный DataSet. Для этого выполните описанные ниже шаги.

1. Щелкните правой кнопкой мыши на проекте в окне Solution Explorer (Проводник решения).
2. Выберите в контекстном меню пункт Add⇨New Item (Добавить⇨Новый элемент).
3. Разверните узел Categories (Категории) в диалоговом окне Add New Item (Добавить новый элемент). Выберите в узел Data (Данные). Выберите шаблон DataSet в списке Data Templates (Шаблоны данных). Измените имя файла DataSet на StudentsDataSet.xsd и щелкните на кнопке Add (Добавить).
4. После этого должен открыться визуальный конструктор наборов данных (DataSet Designer). Перетащите элемент DataTable из панели инструментов на поверхность визуального конструктора.
5. Щелкните правой кнопкой мыши на заголовке, добавленном DataTable, и выберите в контекстном меню пункт Properties (Свойства).
6. В открывшемся окне Properties (Свойства) измените свойство Name объекта DataTable, присвоив ему значение Students.
7. Снова щелкните правой кнопкой мыши на DataTable и выберите в контекстном меню пункт Add⇨Column (Добавить⇨Столбец).
8. Установите свойство Name вновь добавленного объекта DataColumn в Id, а свойство DataType — в System.Int32.
9. Снова щелкните правой кнопкой мыши на DataTable и выберите в контекстном меню пункт Add⇨Column.
10. Установите свойство Name вновь добавленного объекта DataColumn в Name.
11. Сохраните файл.

В результате создан типизированный DataSet по имени StudentsDataSet, который содержит объект DataTable по имени Students с двумя столбцами данных типа DataColumn — один по имени Id типа Int32 и один по имени Name типа string. Этот типизированный DataSet можно использовать для выполнения запросов LINQ, и поскольку DataSet типизирован, к полям DataRow можно обращаться как к членам класса. Давайте рассмотрим пример.

Имея типизированный DataSet, на нем можно выполнять запросы LINQ, как показано в листинге 11.1.

Листинг 11.1. Пример запроса к типизированному DataSet

```
StudentsDataSet studentsDataSet = new StudentsDataSet();
studentsDataSet.Students.AddStudentsRow(1, "Joe Rattz");
studentsDataSet.Students.AddStudentsRow(7, "Anthony Adams");
studentsDataSet.Students.AddStudentsRow(13, "Stacy Sinclair");
studentsDataSet.Students.AddStudentsRow(72, "Dignan Stephens");
string name =
  studentsDataSet.Students.Where(student => student.Id == 7).Single().Name;
Console.WriteLine(name);
```

В этом примере создается объект StudentsDataSet, к которому добавляются четыре записи о студентах с использованием имен из предыдущей главы. Как правило, в рабочем коде делать это не придется, потому что, скорее всего, эта информация будет извлекаться из базы данных.

После наполнения данными типизированного DataSet к нему выполняется запрос. Обратите внимание, что обращением к таблице DataTable по имени Students производится как к свойству объекта StudentsDataSet. Кроме того, в лямбда-выражении операции Where имеется прямое обращение к свойству Id элемента, который должен быть DataRow — вместо вызова свойства Field этого DataRow. Это можно делать потому, что объект DataSet типизирован. По этой же причине, после получения единственного объекта DataRow вызовом операции Single можно напрямую обращаться к его свойству Name. Результат показан ниже:

```
Anthony Adams
```

Разве не здорово? Типизированные DataSet делают взаимодействие с DataSet таким же простым, как работа с нормальными объектами классов и их свойствами.

Собираем все вместе

В завершение этой главы рассмотрим более полный пример, в котором действительно получается объект DataSet из базы данных, как это обычно происходит в коде реальных приложений.

Нужно признать, что создание примера разумных размеров, в котором извлекаются данные из базы и используется API-интерфейс LINQ to DataSet, выглядит несколько неестественно. В конце концов, планируется выполнить запрос SQL к данным из базы с помощью ADO.NET, чтобы получить DataSet, а затем снова запросить эти данные с помощью LINQ to DataSet, и все это в нескольких строках кода. В реальности кто-то может спросить: почему бы просто не изменить запрос SQL таким образом, чтобы получить все, что нужно, с первого раза? Да, это так. Но речь идет о текущем примере.

В рассматриваемом сценарии работа осуществляется с компанией под названием Northwind. Она уже располагает приложением, запрашивающим в базе данных список заказов. Это конкретное приложение выполняет разнообразный анализ того, какие сотрудники какие наименования товара каким заказчикам продали, и в какие страны были произведены поставки в соответствии с заказами. Так что приложение уже загружает сотрудников, заказчиков и страны поставок для всех заказов в DataSet. Задача состоит в том, чтобы провести один или более видов анализа этих запрошенных данных. Необходимо построить уникальный список каждого сотрудника, который продал что-либо любой компании по всем заказам, поставляемым в Германию.

В этом примере создается экземпляр SqlDataAdapter, за которым следует DataSet, и вызывается метод Fill этого объекта SqlDataAdapter для наполнения DataSet. В настоящем сценарии это уже сделано, поскольку этим занимается существующее

приложение. Так что код должен получить готовый объект DataSet. Но поскольку здесь не участвует полноценное приложение, просто сделаем это в примере. После получения объекта DataSet с результатами запроса SQL все, что останется — это выполнить запрос LINQ to DataSet и отобразить полученный результат. Код представлен в листинге 11.2.

Листинг 11.2. Собираем все вместе

```
string connectionString =
  @"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind;Integrated Security=SSPI;";

SqlDataAdapter dataAdapter = new SqlDataAdapter(
  @"SELECT O.EmployeeID, E.FirstName + ' ' + E.LastName as EmployeeName,
    O.CustomerID, C.CompanyName, O.ShipCountry
    FROM Orders O
    JOIN Employees E on O.EmployeeID = E.EmployeeID
    JOIN Customers C on O.CustomerID = C.CustomerID",
  connectionString);
DataSet dataSet = new DataSet();
dataAdapter.Fill(dataSet, "EmpCustShip");

// Весь предшествующий код унаследован.

var ordersQuery = dataSet.Tables["EmpCustShip"].AsEnumerable()
  .Where(r => r.Field<string>("ShipCountry").Equals("Germany"))
  .Distinct(System.Data.DataRowComparer.Default)
  .OrderBy(r => r.Field<string>("EmployeeName"))
  .ThenBy(r => r.Field<string>("CompanyName"));

foreach(var dataRow in ordersQuery)
{
  Console.WriteLine("{0,-20} {1,-20}", dataRow.Field<string>("EmployeeName"),
  dataRow.Field<string>("CompanyName"));
}
```

Как видите, в коде производится подключение к базе данных Northwind. При необходимости настройте строку подключения под существующие условия.

Обратите внимание, что в предыдущем запросе использовались операции AsEnumerable, Distinct и Field<T>, описанные в предыдущей главе, а также операции Where, OrderBy и ThenBy из LINQ to Objects. Они позволяют составить необходимый запрос. Если запрос будет работать так, как надо, то должен быть получен список всех сотрудников, осуществивших продажу по заказу каждой компании, поставки по которому выполнялись в Германию, в алфавитном порядке имен сотрудников и названий компаний, без дублированных строк. Вот результат:

```
Andrew Fuller Die Wandernde Kuh
Andrew Fuller Königlich Essen
Andrew Fuller Lehmanns Marktstand
Andrew Fuller Morgenstern Gesundkost
Andrew Fuller Ottilies Käseladen
Andrew Fuller QUICK-Stop
Andrew Fuller Toms Spezialitäten
Anne Dodsworth Blauer See Delikatessen
Anne Dodsworth Königlich Essen
Anne Dodsworth Lehmanns Marktstand
Anne Dodsworth QUICK-Stop
...
Steven Buchanan Frankenversand
Steven Buchanan Morgenstern Gesundkost
Steven Buchanan QUICK-Stop
```

Обратите внимание, что для каждого сотрудника в левой части строки ни одна компания не повторяется в правой части. Это важно, потому что демонстрирует необходимость операций работы с множествами LINQ to DataSet. Для целей тестирования измените вызов операции `Distinct` так, чтобы не был указан компаратор по умолчанию `DataRowComparer.Default` — сразу появятся дублированные строки.

Чтобы увидеть другой пример использования синтаксиса выражений, в листинге 11.3 представлен похожий код, но с синтаксисом выражений запросов.

Листинг 11.3. Собираем все вместе; синтаксис выражений запросов

```
string connectionString =
  @"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind;Integrated Security=SSPI;";

SqlDataAdapter dataAdapter = new SqlDataAdapter(
  @"SELECT O.EmployeeID, E.FirstName + ' ' + E.LastName as EmployeeName,
    O.CustomerID, C.CompanyName, O.ShipCountry
    FROM Orders O
    JOIN Employees E on O.EmployeeID = E.EmployeeID
    JOIN Customers C on O.CustomerID = C.CustomerID",
  connectionString);

DataSet dataSet = new DataSet();
dataAdapter.Fill(dataSet, "EmpCustShip");

// Весь предшествующий код унаследован.

var ordersQuery = (from r in dataSet.Tables["EmpCustShip"].AsEnumerable()
                 where r.Field<string>("ShipCountry").Equals("Germany")
                 orderby r.Field<string>("EmployeeName"),
                   r.Field<string>("CompanyName")
                 select r)
                .Distinct(System.Data.DataRowComparer.Default);

foreach (var dataRow in ordersQuery)
{
  Console.WriteLine("{0,-20} {1,-20}", dataRow.Field<string>("EmployeeName"),
    dataRow.Field<string>("CompanyName"));
}
```

Теперь запрос использует синтаксис выражения запроса. Хотя цель была в том, чтобы сделать запрос функционально эквивалентным предыдущему, добиться этого не удалось. Обратите внимание, что операция `Distinct` вызывается теперь в самом конце запроса. Вспомните, что компилятор не может транслировать все операции запроса, указанного в синтаксисе выражений запросов, а только наиболее часто используемые. В этом случае он не знает, как транслировать операцию `Distinct`. Из-за этого в синтаксисе выражений запросов данный вызов невозможно сделать частью запроса. Как видите, он вызван в конце запроса. Полученный результат будет тем же, что и раньше.

Однако есть разница в производительности между запросами в листингах 11.3 и 11.2. В листинге 11.2 операция `Distinct` вызывается сразу после операции `Where`, поэтому дублированные записи исключаются из результирующего набора до его упорядочивания. В листинге 11.3 операция `Distinct` не вызывается до самого конца, так что дублированные записи присутствуют на этапе упорядочивания результата. Это значит, что сначала записи будут отсортированы, а затем исключены при вызове операции `Distinct`. Это излишняя работа, однако, неизбежная, если вы предпочитаете применить синтаксис выражений запросов.

Резюме

Как было показано в этой главе, с помощью LINQ to DataSet можно опрашивать не только обычные объекты DataSet, но также и типизированные DataSet. Типизированные DataSet облегчают сопровождение кода, повышая его читабельность, а LINQ to DataSet позволяет очень легко и просто выполнять запросы к ним. Также был продемонстрирован более реалистичный запрос LINQ to DataSet, обращающийся к базе данных Northwind.

API-интерфейс LINQ to DataSet открывает доступ LINQ к еще одной предметной области. Учитывая наличие большого объема кода, использующего DataSet, LINQ to DataSet имеет широкое поле приложения, поскольку позволяет легко модифицировать унаследованный код .NET, тем самым существенно облегчая запросы данных из DataSet.

Одно преимущество, которое имеется у LINQ to DataSet перед LINQ to SQL, заключается в том, для выполнения запросов LINQ to DataSet не требуется генерации и предварительной компиляции какого-то кода классов, связанных с базой данных. Это делает LINQ to DataSet более динамичным и в большей степени подходящим для построения утилит работы с базами, когда конкретная база данных неизвестна до момента выполнения.

Благодаря использованию операции AsEnumerable для создания последовательностей из объектов DataTable, стало возможным применение стандартных операций запросов LINQ to Objects, что добавило мощи к арсеналу возможностей запросов.

В LINQ to DataSet были добавлены операции для работы с ключевыми классами DataSet: DataTable, DataRow и DataColumn. Не забывайте о проблеме, которая вызвала необходимость в новых, ориентированных на множества, прототипах операций Distinct, Union, Intersect, Except и SequenceEqual, а именно — проблеме сравнения DataRow на эквивалентность. Поэтому при работе с объектами DataSet, DataTable и DataRow всегда отдавайте предпочтение использованию ориентированных на множества прототипов операций Distinct, Union, Intersect, Except и SequenceEqual, когда указывается объект проверки эквивалентности, а не тех версий прототипов, в которые указание такого объекта не предусмотрено.

И, наконец, при получении значения столбца используйте операции Field<T> и SetField<T> для исключения проблем сравнения со значениями null.

При работе с API-интерфейсом LINQ to DataSet становится очевидным одно обстоятельство. Мощь и удобство DataSet совершенно недооценивается. Объекты DataSet предлагают широкие возможности как кэшированные хранилища реляционных данных. И хотя их средства поиска ограничены, благодаря LINQ to DataSet все ограничения устраняются. Теперь для опроса данных из DataSet можно использовать LINQ, что значительно облегчает разработку кода.

ЧАСТЬ V

LINQ to SQL

В этой части

Глава 12. Введение в LINQ to SQL
Глава 13. Советы и инструменты, связанные с LINQ to SQL
Глава 14. Операции LINQ to SQL для баз данных
Глава 15. Сущностные классы LINQ to SQL
Глава 16. Класс `DataContext` в LINQ to SQL
Глава 17. Конфликты параллельного доступа LINQ to SQL
Глава 18. Дополнительные возможности LINQ to SQL

ГЛАВА 12

Введение в LINQ to SQL

Листинг 12.1. Простой пример обновления столбца `ContactName` таблицы `Customer` из базы данных Northwind

```
// Создать DataContext.
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");

// Извлечь заказчика LAZYK.
Customer cust = (from c in db.Customers
                 where c.CustomerID == "LAZYK"
                 select c).Single<Customer>();

// Обновить контактное имя.
cust.ContactName = "Ned Plimpton";

try
{
  // Сохранить изменения.
  db.SubmitChanges();
}

// Обнаружить конфликты совместного доступа.
catch (ChangeConflictException)
{
  // Разрешить конфликты.
  db.ChangeConflicts.ResolveAll(RefreshMode.KeepChanges);
}
```

На заметку! Этот пример требует генерации сущностных классов, о чем рассказывается далее в этой главе.

В листинге 12.1 с помощью LINQ to SQL запрашивается запись, поле `CustomerID` которой имеет значение `"LAZYK"`, из таблицы `Customers` базы данных Northwind, и возвращается объект `Customer`, представляющий эту запись. После этого свойство `ContactName` объекта `Customer` обновляется, а изменения сохраняются в базе данных Northwind посредством вызова метода `SubmitChanges`. Хоть кода не так много, но он вдобавок обнаруживает конфликты параллельного доступа и разрешает их в случае возникновения.

Нажав <Ctrl+F5>, запустите код в листинге 12.1. Здесь нет консольного вывода, но если вы проверите базу данных, то обнаружите, что `ContactName` для заказчика `LAZYK` теперь содержит значение `"Ned Plimpton"`.

На заметку! В этом примере в базу данных вносятся изменения без возможности последующего восстановления. Исходным значением `ContactName` для заказчика `LAZYK` было "`John Steel`". Чтобы последующие примеры работали правильно, это значение потребуется вернуть. Это можно сделать вручную или просто изменить пример кода так, чтобы он восстанавливал старое значение, и запустить его снова.

Введение в LINQ to SQL

К этому моменту уже рассказывалось об использовании LINQ с находящимися в памяти коллекциями данных и массивами, XML и `DataSet`. Теперь перейдем к тому, что многим кажется наиболее веской причиной для использования LINQ — LINQ to SQL.

LINQ to SQL — это API-интерфейс для работы с базами данных SQL Server. В современном мире, где господствуют объектно-ориентированные языки программирования, существует несоответствие между языком программирования и реляционной базой данных. При написании приложения мы моделируем классы как представления объектов реального мира, таких как заказчики, счета, политики и полеты. Нам нужен способ сохранения этих объектов, чтобы при перезапуске приложения все эти объекты с их данными не были потеряны. Однако большинство баз данных промышленного масштаба остаются реляционными и хранят свою информацию в виде записей в таблицах, а не в виде объектов. Пользовательский класс может содержать несколько адресов и номеров телефонов, хранящихся в коллекциях, которые являются дочерними свойствами класса заказчика; при сохранении такая информация, скорее всего, распределится по множеству таблиц, например, в таблицу заказчиков, таблицу адресов и таблицу телефонов.

Вдобавок типы данных, поддерживаемые языком приложения, отличаются от типов базы данных. Разработчикам приходится писать собственный код, который загружает и сохраняет объекты заказчиков из соответствующих таблиц, обрабатывая необходимые преобразования данных между языком приложения и базой данных. Это утомительный, и часто подверженный ошибкам процесс. Наличие упомянутых проблем объектно-реляционного отображения (object-relational mapping — ORM), часто называемых *объектно-реляционной потерей соответствия* (object-relational impedance mismatch), в течение ряда лет привело к появлению широкого разнообразия готовых программных ORM-решений. LINQ to SQL — это реализация ORM начального уровня от Microsoft на основе LINQ, предназначенная для работы с SQL Server.

Обратите внимание — для работы с SQL Sever. API-интерфейс LINQ to SQL предназначен исключительно для SQL Server. Однако существуют реализации LINQ от независимых поставщиков, поддерживающие наиболее распространенные базы данных, включая Oracle, DB2, MySQL, SqlLite и другие.

Также учтите, что LINQ to SQL — реализация ORM *начального уровня*. Если вы найдете ее недостаточно мощной или гибкой, чтобы отвечать имеющимся требованиям, попробуйте обратиться к API-интерфейсу LINQ to Entities, который рассматривается в главе 19.

Большинство инструментов ORM пытаются абстрагировать физическую базу данных в виде бизнес-объектов. С такой абстракцией иногда теряется возможность выполнения запросов SQL, составляющих значительную часть абстракции реляционных баз данных. Именно это отличает LINQ to SQL от большинства его аналогов. Мы не только получаем удобство в виде бизнес-объектов, которые отображаются на базу данных, но также получаем полноценный язык запросов, подобный хорошо знакомому SQL.

Совет. LINQ to SQL — инструмент ORM начального уровня, позволяющий выполнять мощные SQL-запросы.

В дополнение к предоставлению возможностей запросов LINQ, до тех пор, пока запрос возвращает *сущностные объекты* LINQ to SQL, в противоположность одиночным полям, именованным не сущностным классам или анонимным классам, LINQ to SQL также предлагает средства отслеживания изменений и обновлений базы данных, построенных на основе оптимистической модели обнаружения и разрешения конфликтов параллельного доступа, а также транзакционной целостности.

В листинге 12.1 сначала должен был создан экземпляр класса `Northwind`. Этот класс унаследован от класса `DataContext`, который более подробно рассматривается в главе 16. А пока воспринимайте его как "усовершенствованное" соединение с базой данных. Он также обрабатывает обновление базы данных, как будет показано позже, при вызове на нем метода `SubmitChanges`. Далее из базы данных `Northwind` в объект `Customer` извлекается один заказчик. Этот объект `Customer` является экземпляром сущностного класса `Customer`, который может быть либо написан вручную, либо сгенерирован. В данном случае класс `Customer`, как и `Northwind`, был сгенерирован утилитой SQLMetal. После извлечения заказчика было обновлено одно из свойств объекта `Customer` по имени `ContactName` и вызван метод `SubmitChanges` для сохранения модифицированного имени контакта в базе данных. Обратите внимание, что этот вызов помещен в блок `try/catch`, и специально перехватывается исключение `ChangeConflictException`. Это нужно для обработки конфликтов параллельного доступа, которые детально описаны в главе 17.

Прежде чем можно будет запустить этот или любой другой пример, приведенный в этой главе, понадобится создать сущностные классы для базы данных Northwind. Руководство по созданию необходимых сущностных классов читайте в разделе "Предварительные условия для запуска примеров" далее в главе.

LINQ to SQL — сложная тема, и любые примеры требуют участия многих элементов LINQ to SQL. В первом примере в начале этой главы использовался класс, производный от `DataContext` — `Northwind`; сущностный класс `Customer`; обнаружение и разрешение конфликтов параллельного доступа; и обновление базы данных через метод `SubmitChanges`. Объяснить все эти концепции одновременно невозможно. Поэтому сначала будут предоставлены базовые сведения для понимания каждого из перечисленных компонентов, которые позволят разобраться в основах LINQ to SQL. Более подробно все эти концепции будут рассматриваться в последующих главах, посвященных LINQ to SQL.

DataContext

`DataContext` — это класс, устанавливающий соединение с базой данных. Он также предоставляет несколько служб, обеспечивающих отслеживание идентичности, отслеживание изменений и обработку этих изменений. Более детально эти службы описаны в главе 16. А пока просто знайте, что `DataContext` — это класс, соединяющий с базой данных, отслеживающий то, что мы изменяем, и обновляющий базу данных при вызове метода `SubmitChanges`.

В LINQ to SQL принято использовать класс, производный от `DataContext`. Имя производного класса обычно совпадает с именем базы данных, на которую он отображается. В главах о LINQ to SQL на этот производный класс мы часто будем ссылаться как на `[Your]DataContext`, поскольку его имя зависит от базы данных, для которой он создан.

В приводимых примерах производный от `DataContext` класс будет называться `Northwind`. Для его построения применялся инструмент SqlMetal, входящий в состав Visual Studio 2010, который генерирует классы отображения на основе базы данных SQL Server.

Этот производный от DataContext класс, [Your]DataContext, обычно будет иметь общедоступное свойство Table<T> для каждой таблицы базы данных, которая отображается на базу данных, где T — тип *сущностного класса*, экземпляр которого создается для каждой извлеченной записи из конкретной таблицы базы данных. Тип данных Table<T> представляет собой специализированную коллекцию. Например, поскольку в базе Northwind присутствует таблица Customers, класс Northwind, унаследованный от класса DataContext, будет иметь Table<Customers> по имени Customers. Это значит, что обращаться к записям в таблице Customers базы данных можно, непосредственно обращаясь к свойству Customers типа Table<Customers> в классе Northwind. Это было показано в первом примере настоящей главы, в листинге 12.1, где оно кодировалось как db.Customers. Этот код запрашивает запись в таблице Customers базы данных Northwind.

Сущностные классы

LINQ to SQL подразумевает использование сущностных классов, причем каждый из них обычно отображается на единственную таблицу базы данных. Однако, используя наследуемое отображение сущностных классов, при определенных условиях можно отобразить целую иерархию классов на единственную таблицу. Подробнее об этом написано в главе 18. Таким образом, мы имеем сущностные классы, отображенные на таблицы базы данных, и свойства сущностных классов, отображенные на столбцы этих таблиц. Это отображение “класс-таблица” и “свойство-столбец” являются сутью LINQ to SQL.

На заметку! Квинтэссенция LINQ to SQL заключается в отображении сущностных классов на таблицы базы данных и свойств сущностных классов на столбцы таблиц базы данных.

Такое отображение может возникать непосредственно в исходных файлах класса за счет оснащения их соответствующими атрибутами или же может быть задано во внешнем XML-файле отображения. При использовании такого внешнего файла специфичная для LINQ to SQL информация может храниться отдельно от исходного кода. Это может быть очень удобно, если нет исходного кода либо если нужно хранить код отдельно от LINQ to SQL. В большинстве примеров из глав, посвященных LINQ to SQL, используются сущностные классы, сгенерированные инструментом командной строки SQLMetal. Этот инструмент генерирует сущностные классы с информацией об отображении LINQ to SQL, встроенной непосредственно в генерируемые исходные модули. Эта информация представлена в форме атрибутов и их свойств.

Сущностные классы имеют имя в форме единственного числа от имени таблицы базы данных Northwind. Например, в листинге 12.1 используется класс по имени Customer. Поскольку *Customer* — форма единственного числа от *Customers*, а в базе данных Northwind есть таблица по имени Customers, это указывает на то, что Customer — сущностный класс для таблицы Customers из базы данных Northwind.

Инструмент командной строки SQLMetal поддерживает опцию /pluralize, которая вызывает именование сущностного класса в форме единственного числа имени таблицы базы данных. Если при генерации сущностных классов эта опция не указана, то класс получил бы имя Customers, а не Customer, потому что таково имя таблицы — Customers. Не забывайте, что сущностный класс может иметь имя как во множественном, так и в единственном числе.

Ассоциации

Ассоциация (association) — это термин, используемый для назначения первичного ключа для отношений внешнего ключа между двумя сущностными классами. В от-

ношении “один ко многим” результатом ассоциации является то, что родительский класс — тот, что содержит первичный ключ — включает коллекцию дочерних классов, т.е. классов, имеющих внешний ключ. Эта коллекция сохраняется в приватной переменной-члене типа EntitySet<T>, где T будет типом дочернего сущностного класса.

Например, в сущностном классе Customer, сгенерированном инструментом командной строки SQLMetal для базы данных Northwind, есть приватный член типа EntitySet<Order>, названный _ Orders, который содержит все объекты Order для определенного объекта Customer:

```
private EntitySet<Order> _Orders;
```

SQLMetal также сгенерировал общедоступное свойство по имени Orders, используемое для доступа к приватной коллекции _Orders.

На другом конце отношения — в классе, содержащем внешний ключ, — находится ссылка на родительский класс, поскольку это отношение “многие к одному”. Эта ссылка сохраняется в приватной переменной-члене типа EntityRef<T>, где T — родительский класс (здесь термин “родительский” относится к ассоциации, а не к наследованию).

В сгенерированных сущностных классах Northwind сущностный класс Order содержит приватную переменную-член типа EntityRef<Customer> по имени _Customer:

```
private EntityRef<Customer> _Customer;
```

И снова SQLMetal сгенерировал общедоступное свойство по имени Customer для доступа к ссылке на родителя.

Ассоциация, первичные и внешние ключи, а также направление отношения задаются атрибутами и свойствами атрибутов в исходном модуле сгенерированных сущностных классов.

Преимущество ассоциации состоит в возможности доступа к дочерним объектам родительских и, таким образом, к записям базы данных, с той же легкостью, как к любому свойству родительского объекта. Аналогично доступ к родительскому объекту из дочернего заключается в обращении к соответствующему свойству дочернего объекта.

Обнаружение конфликтов параллельного доступа

Одной из ценных служб, предоставляемых вам DataContext, является обработка изменений. Когда вы пытаетесь обновить базу данных вызовом метода SubmitChanges класса DataContext, он автоматически выполняет оптимистическое обнаружение конфликтов параллельного доступа.

Если конфликт обнаружен, генерируется исключение ChangeConflictException. Всякий раз, когда вы вызываете метод SubmitChanges, то должны помещать его в блок try/catch и перехватывать исключение ChangeConflictException. Это — правильный способ обнаружения конфликтов параллельного доступа.

Пример этого можно было видеть в листинге 12.1. Подробное описание деталей обнаружения и разрешения конфликтов параллелизма приведено в главе 17. Во многих примерах в этой и последующих главах, посвященных LINQ to SQL, обнаружение конфликтов параллельного доступа или их разрешение не выполняется; это сделано только в целях краткости и ясности. В реальном коде это нужно делать всегда.

Разрешение конфликтов параллельного доступа

Как только конфликт параллелизма обнаружен, следующий шаг состоит в его разрешении. Это можно сделать несколькими способами.

В листинге 12.1 был предпринят простейший прием за счет вызова метода ResolveAll коллекции ChangeConflicts унаследованного класса DataContext, когда перехвачено исключение ChangeConflictException.

Опять-таки, во многих примерах, приведенных в главах по LINQ to SQL, не включен код для обнаружения и разрешения конфликтов, но вы всегда должны его предусматривать в реальном коде.

Как упоминалось в предыдущем разделе, подробности разрешения конфликтов параллелизма будут детально описаны в главе 17.

Предварительные условия для запуска примеров

Поскольку почти во всех примерах в этой и последующих главах, посвященных LINQ to SQL, используется *расширенная* база данных примеров Northwind, понадобятся сущностные классы и файлы отображения для базы данных Northwind.

Получение соответствующей версии базы данных Northwind

К сожалению, в стандартной базе данных Northwind недостает нескольких вещей, которые понадобятся для полноценной демонстрации LINQ to SQL, таких как табличные и скалярные функции. Поэтому вместо стандартной базы данных Northwind применяется расширенная версия, поставляемая Microsoft для демонстрации возможностей LINQ.

Расширенная версия базы Northwind входит в состав загружаемого кода примеров для этой книги.

Генерация сущностных классов Northwind

В этой главе процесс генерации сущностных классов описан лишь вкратце. Более подробно он рассматривается в главе 13.

Чтобы сгенерировать сущностные классы, должна быть доступна расширенная версия базы данных Northwind, как упоминалось в предыдущем разделе.

Откройте окно командной строки Visual Studio, выбрав в меню пункт Microsoft Visual Studio 2010⇨Visual Studio Tools⇨Visual Studio 2010 Command Prompt (Microsoft Visual Studio 2010⇨Инструменты Visual Studio⇨Командная строка Visual Studio 2010). В открывшемся окне командной строки смените текущий каталог на тот, в котором должны быть созданы сущностные классы и внешние файлы отображения, например:

```
cd \
```

Если вы собираетесь генерировать сущностные классы с использованием файлов базы данных Northwind, не подключая к ним базу данных, используйте следующую команду:

```
sqlmetal /namespace:nwind /code:Northwind.cs /pluralize /functions /sprocs /views
<путь к MDF-файлу Northwind>
```

Внимание! Уделите особое внимание имени файла MDF и регистру его символов, указывая его в командной строке. Имя и регистр генерируемого производного от `DataContext` класса будет соответствовать имени файла, переданному в командной строке, а не физическому имени самого файла. Если вы отступите от точного написания имени класса Northwind, то ни один пример не станет работать без модификации. Поэтому важно передать имя файла базы данных, как `[`*путь*`]\Northwind.mdf`, а не `northwind.mdf`, `NorthWind.mdf` или как-нибудь еще.

Чтобы создать сущностные классы из файла по имени `Northwind.mdf`, введите следующую команду:

```
sqlmetal /namespace:nwind /code:Northwind.cs /pluralize /functions /sprocs /views
"C:\Northwind.mdf"
```

Запуск этой команды вызовет создание модуля сущностного класса по имени `Northwind.cs` в текущем каталоге.

Если вы собираетесь генерировать сущностные классы из базы данных Northwind, которая уже подключена к SQL Server, используйте следующую команду:

```
sqlmetal /server:<сервер> /user:<пользователь> /password:<пароль>
/database:Northwind /namespace:nwind /code:Northwind.cs
/pluralize /functions /sprocs /views
```

Чтобы создать сущностные классы из подключенной базы данных по имени Northwind, введите следующую команду:

```
sqlmetal /server:.\SQLExpress /database:Northwind /namespace:nwind
/code:Northwind.cs /pluralize /functions /sprocs /views
```

На заметку! В зависимости от окружения, в командной строке может понадобиться указать пользователя с помощью опции `/user:[`*пользователь*`]` и пароль — посредством опции `/password:[`*пароль*`]`. За подробностями обращайтесь в главу 13.

Команда, введенная с использованием любого из этих подходов, сообщает утилите SQLMetal о том, что она должна сгенерировать файл исходного кода по имени `Northwind.cs` в текущем каталоге. Все опции этой утилиты будут описаны в следующей главе. Скопируйте сгенерированный файл `Northwind.cs` в проект, добавив его в качестве существующего элемента.

Теперь можно применять LINQ to SQL на базе данных Northwind, используя сущностные классы, которые содержатся в файле `Northwind.cs`.

Совет. Будьте осторожны, внося изменения в исходный файл сгенерированных сущностных классов. Позднее может возникнуть необходимость в их повторной генерации, что вызовет потерю этих изменений. Может понадобиться расширить бизнес-логику за счет добавления методов к сущностным классам. Вместо модификации сгенерированного файла рассмотрите возможность применения частичных классов C#, чтобы хранить добавленные свойства и методы в отдельном модуле исходного кода.

Генерация XML-файла отображения Northwind

Также понадобится сгенерировать файл отображения, который будет использоваться в некоторых примерах. Опять-таки, для этого применяется утилита SQLMetal. В том же текущем каталоге выполните следующую команду:

```
sqlmetal /map:northwindmap.xml "C:\Northwind.mdf" /pluralize /functions /sprocs
/views /namespace:nwind /code:Northwind.cs
```

При указании MDF-файла обратите особое внимание на регистр символов. Эта команда приведет к генерации файла по имени `northwindmap.xml` в текущем каталоге.

Использование LINQ to SQL

Чтобы можно было пользоваться API-интерфейсом LINQ to SQL, в проект понадобится добавить сборку `System.Data.Linq.dll`, если это не было сделано ранее. Кроме того, в исходный код должны быть добавлены директивы `using` для пространств имен `System.Linq` и `System.Data.Linq`:

```
using System.Data.Linq;
using System.Linq;
```

В коде примеров потребуется добавить директиву using для пространства имен, в котором находятся сгенерированные сущностные классы:

```
using nwind;
```

IQueryable<T>

Вы увидите, что во многих примерах LINQ to SQL в этой и последующих главах работа производится с последовательностями типа IQueryable<T>, где T — тип сущностного класса. Это тип последовательностей, которые обычно возвращаются запросами LINQ to SQL. Они часто работают подобно последовательности IEnumerable<T>, и это не случайно. Интерфейс IQueryable<T> реализует интерфейс IEnumerable<T>. Вот его определение:

```
interface IQueryable<T> : IEnumerable<T>, IQueryable
```

Благодаря такому наследованию, последовательность IQueryable<T> можно трактовать как последовательность IEnumerable<T>.

Некоторые общие методы

Как вы вскоре убедитесь, примеры в этой главе и главах, посвященных LINQ to SQL, очень быстро усложняются. Демонстрация конфликтов параллельного доступа требует внесения изменений в базу данных, внешнюю по отношению к LINQ to SQL. Иногда приходится извлекать данные извне LINQ to SQL. Чтобы подчеркнуть код LINQ to SQL и исключить как можно больше тривиальных деталей, в то же время сохраняя реальную пользу примеров, создано несколько общих методов, предназначенных для использования во многих примерах.

Не забудьте добавить эти общие методы в исходные модули при тестировании примеров из глав, посвященных LINQ to SQL.

GetStringFromDb()

Общий метод, который пригодится не раз — это метод для получения простой строки из базы данных с использованием стандартной технологии ADO.NET. Он позволит просматривать то, что в действительности имеется в базе данных, чтобы сравнить с тем, что показывает LINQ to SQL.

GetStringFromDb: метод для извлечения строки с использованием ADO.NET

```
static private string GetStringFromDb(
  System.Data.SqlClient.SqlConnection sqlConnection, string sqlQuery)
{
  if (sqlConnection.State != System.Data.ConnectionState.Open)
  {
    sqlConnection.Open();
  }
  System.Data.SqlClient.SqlCommand sqlCommand =
    new System.Data.SqlClient.SqlCommand(sqlQuery, sqlConnection);
  System.Data.SqlClient.SqlDataReader sqlDataReader = sqlCommand.ExecuteReader();
  string result = null;
  try
  {
    if (!sqlDataReader.Read())
    {
      throw (new Exception(
        String.Format("Unexpected exception executing query [{0}].", sqlQuery)));
    }
```

```
      else
      {
        if (!sqlDataReader.IsDBNull(0))
        {
          result = sqlDataReader.GetString(0);
        }
      }
    }
    finally
    {
      // Всегда вызывать Close по окончании чтения.
      sqlDataReader.Close();
    }
    return (result);
  }
```

При вызове методу `GetStringFromDb` передается объект `SqlConnection` и строка, содержащая запрос SQL. Метод проверяет, открыто ли соединение, и если нет, то открывает его.

После этого создается экземпляр `SqlCommand` передачей конструктору запроса и соединения. Затем в результате вызова метода `ExecuteReader` класса `SqlCommand` получается `SqlDataReader`. Экземпляр `SqlDataReader` выполняет чтение вызовом метода `Read`, и если данные прочитаны и возвращенное значение первого столбца не равно `null`, это значение извлекается методом `GetString`. Наконец, `SqlDataReader` закрывается, и значение первого столбца возвращается вызвавшему методу.

ExecuteStatementInDb()

Иногда нужно будет выполнять операторы SQL, не являющиеся запросами, такие как вставка, обновление и удаление в ADO.NET, чтобы модифицировать состояние базы данных вне LINQ to SQL. Для этой цели предусмотрен метод `ExecuteStatementInDb()`.

ExecuteStatementInDb: метод для выполнения операторов вставки, обновления и удаления в ADO.NET

```
static private void ExecuteStatementInDb(string cmd)
{
  string connection =
    @"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind;Integrated Security=SSPI;";
  System.Data.SqlClient.SqlConnection sqlConn =
    new System.Data.SqlClient.SqlConnection(connection);
  System.Data.SqlClient.SqlCommand sqlComm =
    new System.Data.SqlClient.SqlCommand(cmd);

  sqlComm.Connection = sqlConn;
  try
  {
    sqlConn.Open();
    Console.WriteLine("Выполнение оператора SQL для базы данных с помощью ADO.NET...");
    sqlComm.ExecuteNonQuery();
    Console.WriteLine("База данных обновлена.");
  }
  finally
  {
    // Закрыть соединение.
    sqlComm.Connection.Close();
  }
}
```

При вызове методу `ExecuteStatementInDb` передается строка, содержащая команду SQL. Создаются экземпляры `SqlConnection` и `SqlCommand`. Экземпляру `SqlCommand` назначается `SqlConnection`. Затем `SqlConnection` открывается и команда SQL выполняется вызовом метода `ExecuteNonQuery` объекта `SqlCommand`. И, наконец, `SqlConnection` закрывается.

Резюме

В этой главе было дано введение в LINQ to SQL и его наиболее базовые понятия, такие как объекты `DataContext`, сущностные классы, ассоциации, а также обнаружение и разрешение конфликтов параллельного доступа.

Было показано, как генерировать сущностные классы и внешние файлы отображения для расширенной базы данных Northwind. Эти сущностные классы будут интенсивно использоваться во всех примерах LINQ to SQL. Кроме того, были представлены общие методы, на которые полагаются многие примеры в последующих главах, посвященных LINQ to SQL. В следующей главе будут даны некоторые советы и показано, как использовать необходимые инструменты для работы с LINQ to SQL.

ГЛАВА 13

Советы и инструменты, связанные с LINQ to SQL

В предыдущей главе был представлен API-интерфейс LINQ to SQL вместе с большей частью связанной с ним терминологии. Было показано, как генерировать сущностные классы, которые потребуются большинству примеров в главах, посвященных LINQ to SQL. Также были определены некоторые общие методы, которые будут использоваться в примерах этих глав.

В этой главе рассматривается несколько примеров, полезных для освоения работы с LINQ to SQL. Также будут описаны некоторые инструменты, упрощающие использование LINQ to SQL.

Введение

Сейчас самое время напомнить, что для возможности запускать примеры этой главы должен быть удовлетворен ряд предварительных условий. Первым делом, следует иметь расширенную базу данных Northwind и сгенерированные сущностные классы для нее. Вернитесь к разделу "Предварительные условия для запуска примеров" главы 12 и удостоверьтесь в наличии соответствующей базы данных и сущностных классов.

Поскольку в этой главе демонстрируется код, который использует сущностные классы, сгенерированные как утилитой SQLMetal, так и визуальным конструктором Object Relational Designer, директива `using` для пространства имен `nwind` в примерах не указывается. Вместо этого будет явно указано пространство имен, когда это понадобится для классов `nwind`. В настоящей главе это необходимо для контроля, на какой сущностный класс `Customer` производится ссылка в каждом конкретном примере. Поскольку по умолчанию Object Relational Designer генерирует пространство имен такое же, как и проект, и т.к. примеры уже будут существовать в пространстве имен проекта, указывать пространство имен для сгенерированных Object Relational Designer сущностных классов не нужно, но это понадобится для сущностных классов, сгенерированных SQLMetal.

На заметку! В отличие от большинства глав, посвященных LINQ to SQL, в примерах данной главы не указывайте директиву `using` для пространства имен `nwind`.

Советы

Придерживаясь ранее заявленного стиля, начнем с предоставления набора советов, содержащих информацию, которую нужно обсудить. Вы должны ознакомиться с этими советами *до того*, как они понадобятся, а не после того, как вы набьете себе шишек в процессе самостоятельной работы.

Используйте свойство DataContext.Log

Теперь самое время напомнить о некоторых специфичных для LINQ to SQL советах, которые приводились в главе 1. Один из них выглядел как "Используйте свойство Log из DataContext" и был связан с использованием свойства Log объекта DataContext для отображения транслированного запроса SQL. Это может быть очень полезно не только в целях отладки, но и для анализа производительности. Вы можете обнаружить, что запросы LINQ to SQL транслируются в очень неэффективные запросы SQL. Или же выяснить, что из-за *отложенной загрузки* ассоциированных сущностных классов приходится выполнять больше SQL-запросов, чем необходимо. Свойство DataContext.Log обеспечит получение такой информации.

Чтобы воспользоваться преимуществами этого средства, необходимо просто присвоить свойству DataContext.Log объект System.IO.TextWriter, такой как Console.Out.

Пример приведен в листинге 13.1.

Листинг 13.1. Пример использования свойства DataContext.Log

```
nwind.Northwind db =
  new nwind.Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
db.Log = Console.Out;
var custs = from c in db.Customers
            where c.Region == "WA"
            select new { Id = c.CustomerID, Name = c.ContactName };
foreach (var cust in custs)
{
  Console.WriteLine("{0} - {1}", cust.Id, cust.Name);
}
```

Поскольку в этой главе будут демонстрироваться сущностные классы, сгенерированные и SQLMetal, и Object Relational Designer, для примеров предусматриваются два класса Customer. Как упоминалось ранее, директива using в примеры не включается, поэтому такие сущностные классы, как Customer, не будут неоднозначными. Но в листинге 13.1 должно быть указано пространство имен nwind для класса Northwind, поскольку в этом примере используется сущностный класс, сгенерированный SQLMetal.

Как видите, в листинге 13.1 свойству Log объекта DataContext по имени Northwind присваивается Console.Out. Вот результат работы примера из листинга 13.1:

```
SELECT [t0].[CustomerID], [t0].[ContactName]
FROM [dbo].[Customers] AS [t0]
WHERE [t0].[Region] = @p0
-- @p0: Input String (Size = 2; Prec = 0; Scale = 0) [WA]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.20706.1
LAZYK - John Steel
TRAIH - Helvetius Nagy
WHITC - Karl Jablonski
```

Это позволяет увидеть в точности, как выглядит сгенерированный SQL-запрос. Обратите внимание, что сгенерированный оператор SQL — это не просто форматированная строка, он использует параметры. Поэтому, применяя LINQ to SQL, мы автоматически получаем защиту от атак внедрением SQL.

Внимание! Если вы видите в полученном результате, что имя, ассоциированное с заказчиком LAZYK, выглядит как Ned Plimpton вместо John Steel, как показано в предыдущем примере, значит, вы запустили пример из листинга 13.1, не восстановив затем данные, как рекомендовалось. Это можно исправить перед тем, как будут затронуты все последующие примеры.

В нескольких следующих главах будет продемонстрировано использование этого средства протоколирования для обнаружения потенциальных проблем, связанных с производительностью.

Используйте метод GetChangeSet()

Метод GetChangeSet() объекта DataContext можно использовать для получения всех сущностных объектов, содержащих изменения, которые должны быть сохранены в базе данных при вызове SubmitChanges. Это удобно для протоколирования и отладки. Этот метод подробно рассматривается в главе 16.

Подумайте об использовании частичных классов или файлов отображения

Без сомнений, одной из главных трудностей использования любого инструмента ORM является управление изменениями в базе данных. Если вы держите всю логику бизнес-классов и логику LINQ to SQL в одних и тех же модулях, то этим создаете себе проблемы сопровождения при изменениях базы данных. Рассмотрите возможность применения частичных классов, помещая бизнес-логику в модуль, отдельный от модулей сгенерированных сущностных классов. Используя частичные классы для хранения ваших атрибутов базы данных LINQ to SQL отдельно от бизнес-логики, вы минимизируете необходимость в добавлении кода к любому сгенерированному сущностному классу.

В качестве альтернативы можно разделить бизнес-классы и отображение сущностей LINQ to SQL с помощью внешнего XML-файла отображения. Речь идет об XML-файле, который отображает бизнес-объекты на базу данных, не полагаясь на атрибуты LINQ to SQL. За дополнительными сведениями о файлах отображения обращайтесь в раздел “XML-схема внешнего файла отображения” главы 15, а также в раздел, посвященный конструктору DataContext, главы 16.

Подумайте об использовании частичных методов

Частичные методы позволяют вмешиваться в определенные события, которые происходят в сущностных классах. Изящество их в том, что если тело частичного метода решено не реализовывать, то при этом не возникнет никаких накладных расходов, и компилятор не будет генерировать код, связанный с их вызовом.

В разделе “Вызов соответствующих частичных методов” главы 15 речь пойдет о применении частичных методов в сущностных классах.

Инструменты

Точно так же, как заранее было предложено несколько советов — до того, как они действительно понадобятся — стоит заранее упомянуть об инструментах, которые в дальнейшем могут облегчить работу. Вполне уместно знать о них и о том, как они могут облегчить и ускорить адаптацию к LINQ to SQL.

SQLMetal

Хотя еще предстоит рассмотреть разные способы создания сущностных классов, необходимых для использования LINQ to SQL с базой данных, вы должны знать, что простейший путь генерации сущностных классов для всей базы данных, если еще нет бизнес-классов, заключается в применении утилиты SQLMetal. Ее можно найти в каталоге C:\Program Files\Microsoft SDKs\Windows\v7.0A\Bin\NETFX 4.0 Tools. Утилита SQLMetal — инструмент командной строки, который генерирует все необходимые части сущностных классов LINQ to SQL.

Чтобы увидеть доступные опции программы SQLMetal, откройте окно командной строки Visual Studio, выбрав в меню пункт Microsoft Visual Studio 2010⇨Visual Studio Tools⇨Visual Studio 2010 Command Prompt (Microsoft Visual Studio 2010⇨Инструменты Visual Studio⇨Командная строка Visual Studio 2010).

Введите в окне командной строки `sqlmetal` и нажмите <Enter>:

```
sqlmetal
```

Эта команда приведет к отображению шаблона вызова программы со всеми опциями:

```
Microsoft (R) Database Mapping Generator 2008 version 4.0.30319.1
for Microsoft (R) .NET Framework version 4.0
Copyright (C) Microsoft Corporation. All rights reserved.

SqlMetal [options] [<input file>]
  Generates code and mapping for the LINQ to SQL component of the .NET framework.

  SqlMetal can:

  - Generate source code and mapping attributes or a mapping file from a database.
  - Generate an intermediate dbml file for customization from the database.
  - Generate code and mapping attributes or mapping file from a dbml file.

Options:
  /server:<name>         Database server name.
  /database:<name>       Database catalog on server.
  /user:<name>           Login user ID (default: use Windows Authentication).
  /password:<password>   Login password (default: use Windows Authentication).
  /conn:<connection string>    Database connection string. Cannot be used with
                               /server, /database, /user or /password options.
  /timeout:<seconds>     Timeout value to use when SqlMetal accesses the
                         database (default: 0 which means infinite).
  /views                 Extract database views.
  /functions             Extract database functions.
  /sprocs                Extract stored procedures.
  /dbml[:file]           Output as dbml. Cannot be used with /map option.
  /code[:file]           Output as source code. Cannot be used with /dbml option.
  /map[:file]            Generate mapping file, not attributes. Cannot be used
                         with /dbml option.
  /language:<language>   Language for source code: VB or C# (default: derived
                         from extension on code file name).
  /namespace:<name>      Namespace of generated code (default: no namespace).
  /context:<type>        Name of data context class (default: derived from
                         database name).
  /entitybase:<type>     Base class of entity classes in the generated code
                         (default: entities have no base class).
  /pluralize             Automatically pluralize or singularize class and member
                         names using English language rules.
  /serialization:<option> Generate serializable classes: None or Unidirectional
                         (default: None).
  /provider:<type>       Provider type: SQLCompact, SQL2000, SQL2005, or SQL2008.
                         (default: provider is determined at run time).
<input file>             May be a SqlExpress mdf file, a SqlCE sdf file, or a
                         dbml intermediate file.
Create code from SqlServer:
  SqlMetal /server:myserver /database:northwind /code:nwind.cs /namespace:nwind
Generate intermediate dbml file from SqlServer:
  SqlMetal /server:myserver /database:northwind /dbml:northwind.dbml
  /namespace:nwind
Generate code with external mapping from dbml:
  SqlMetal /code:nwind.cs /map:nwind.map northwind.dbml
```

```
Generate dbml from a SqlCE sdf file:
  SqlMetal /dbml:northwind.dbml northwind.sdf
Generate dbml from SqlExpress local server:
  SqlMetal /server:.\sqlexpress /database:northwind /dbml:northwind.dbml
Generate dbml by using a connection string in the command line:
  SqlMetal /conn:"server='myserver'; database='northwind'" /dbml:northwind.dbml
```

Как видите, даже приведено несколько примеров запуска. Доступные опции описаны в табл. 13.1.

Таблица 13.1. Опции командной строки SQLMetal

Опция/пример	Описание
/server:<имя> /server:.\SQLExpress	Эта опция позволяет указывать имя сервера базы данных, к которому нужно подключиться. Если ее опустить, SQLMetal принимает по умолчанию localhost/sqlexpress. Чтобы заставить SQLMetal сгенерировать сущностные классы из файла MDF, опустите эту опцию вместе с опцией /database и укажите полное путевое имя MDF-файла в конце команды
/database:<имя> /database:Northwind	Это имя базы данных на указанном сервере, для которой генерируются сущностные классы. Чтобы заставить SQLMetal сгенерировать сущностные классы из файла MDF, опустите эту опцию вместе с опцией /server и укажите полное путевое имя MDF-файла в конце команды
/user:<имя> /user:sa	Это пользовательская учетная запись, применяемая для регистрации в указанной базе данных при подключении для создания сущностных классов
/password:<пароль> /password:1590597893	Это пароль, используемый указанной пользовательской учетной записью для регистрации в указанной базе данных при подключении для создания сущностных классов
/conn:<строка подключения> /conn:"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind;Integrated Security=SSPI;"	Это строка подключения к базе данных. Ее можно использовать вместо указания опций /server, /database, /user и /password
/timeout:<секунды> /timeout:120	Эта опция позволяет указывать значение таймаута в секундах для использования утилитой SQLMetal при генерации сущностных классов. Пропуск этой опции аналогичен указанию SQLMetal значения по умолчанию — 0, что означает отсутствие таймаута. Эта опция не управляет таймаутом сгенерированного класса DataContext, который будет использоваться для запросов LINQ to SQL. Для управления таким таймаутом понадобится устанавливать свойство CommandTimeout класса DataContext или, для более тонкого контроля, вызывать метод DataContext.GetCommand для настройки таймаута конкретного запроса. В главе 16 приведен соответствующий пример

Продолжение табл. 13.1

Опция/пример	Описание
/views /views	Указывайте эту опцию для генерации SQLMetal необходимых свойств Table<T> и сущностных классов для поддержки заданных представлений базы данных
/functions /functions	Применяйте эту опцию для генерации SQLMetal методов для вызова указанных пользовательских функций базы данных
/sprocs /sprocs	Применяйте эту опцию для генерации SQLMetal методов для вызова указанных хранимых процедур базы данных
/dbml[:файл] /dbml:Northwind.dbml	Эта опция указывает имя промежуточного DBML-файла. Целью генерации этого файла является управление именами сгенерированных сущностных классов и их свойств. Благодаря этой опции, можно сгенерировать промежуточный DBML-файл, отредактировать его и затем создать модуль исходного кода вызовом SQLMetal на промежуточном файле DBML с указанием опции /code. В качестве альтернативы можно загрузить промежуточный DBML-файл, созданный этой опцией, в Object Relational Designer, отредактировать его в визуальном конструкторе и позволить визуальному конструктору сгенерировать необходимый исходный код. Эта опция не может быть использована вместе с /map
/code[:файл] /code:Northwind.cs	Это имя файла, создаваемого SQLMetal, который содержит класс, производный от DataContext, и сущностные классы на указанном языке программирования. Эта опция не может использоваться вместе с /dbml. Интересно, что если указать обе опции — /code и /map — в одном вызове SQLMetal, то получится код, сгенерированный без атрибутов LINQ to SQL. Конечно, вы должны использовать сгенерированное отображение (map) вместе с генерированным кодом, чтобы иметь возможность применять LINQ to SQL
/map[:файл] /map:northwindmap.xml	Эта опция указывает, что SQLMetal должен генерировать внешний XML-файл отображения, в противоположность модулю исходного кода, заданного опцией /code. Этот внешний XML-файл отображения может затем быть загружен при создании экземпляра DataContext. Это позволяет использовать LINQ to SQL без какого-либо действительного исходного кода LINQ to SQL, скомпилированного вместе с вашим кодом
/language:<язык> /language:C#	Эта опция определяет, для какого языка программирования должен генерировать код SQLMetal. Допустимые опции в настоящее время: csharp, C# и VB
/namespace:<имя> /namespace:nwind	Эта опция диктует пространство имен, в котором будут находиться сгенерированный класс-наследник DataContect и сущностные классы

Окончание табл. 13.1

Опция/пример	Описание
/context:<тип> /context:Northwind	Указывает имя сгенерированного класса-наследника DataContect. Если данная опция опущена, имя класса будет совпадать с именем базы данных, для которой генерируется код
/entitybase:<тип> /entitybase:MyEntityClassBase	Указывает имя класса, используемого в качестве базового для генерируемых SQLMetal сущностных классов. Если данная опция опущена, сгенерированные сущностные классы не наследуются ни от какого другого класса
/pluralize /pluralize	Эта опция заставляет SQLMetal сохранять множественное число в именах таблиц, но единственное — в именах классов, отображенных на эти таблицы. Таким образом, для таблицы базы данных по имени Customers сгенерированный сущностный класс будет назван Customer (в единственном числе), а Table<Customer> будет сгенерирован с именем Customers (во множественном числе). Таким образом, объект Customer хранится в таблице Customers. С грамматической точки зрения это звучит правильно. Без указания этой опции сущностный класс будет назван Customers (во множественном числе), и Table<Customers> будет назван Customers (во множественном числе). Это значит, что объект Customers буде находиться в таблице Customers. С точки зрения грамматики звучит неверно
/serialization:<опция> /serialization:None	Эта опция указывает, должен ли SQLMetal генерировать атрибуты сериализации для классов. Возможные варианты значений — None и Unidirectional. Если эта опция опущена, по умолчанию SQLMetal принимает None
/provider:<тип> /provider:SQL2005	Данная опция используется для указания класса поставщика базы данных. Допустимые значения: SQLCompact, SQL2000, SQL2005 и SQL2008. Утилита SQLMetal будет генерировать атрибут Provider с классом, заданным в этой опции. Каждое допустимое значение отображается на класс поставщика из пространства имен System.Data.Linq.SqlClient. Утилита SQLMetal добавляет строку Provider в конец указанного значения, чтобы получить имя класса поставщика, и генерирует атрибут Provider, определяющий это имя класса поставщика

Обратите внимание, что опции /dbml, /code и /map могут быть указаны без предоставления имени файла. Если имя файла не задано, то сгенерированный код XML выводится на консоль.

Сравнение XML-файла отображения и промежуточного файла DBML

Одним из запутанных аспектов использования SQLMetal является то, что он позволяет указывать два разных типа выходных XML-файлов. Один создается с помощью опции /map, а второй — посредством опции /dbml.

Отличие между этими двумя файлами состоит в том, что опция /map создает внешний XML-файл отображения, предназначенный для загрузки при создании экземпляра DataContext. Опция /map — альтернатива генерации или написанию вручную исходного модуля, содержащего атрибуты LINQ to SQL, которые компилируются. При таком подходе исходный код не включает никакого специфичного для базы данных кода LINQ to SQL, подлежащего компиляции и компоновке вместе с ним. Это допускает в некоторой степени динамическое обращение к базе данных, поскольку нет необходимости ни в каком предварительно генерированном и компилированном коде. Это называется "в некоторой степени динамическим обращением", потому что код должен знать имена таблиц и полей; в противном случае он просто не будет знать, что запрашивать. Внешний XML-файл отображения инструктирует LINQ to SQL о том, какие есть таблицы, столбцы и хранимые процедуры, с которыми он может взаимодействовать, и на какие классы, свойства и методы они должны быть отображены.

Опция /dbml создает промежуточный файл DBML (XML) для обеспечения возможности редактирования имен впоследствии генерируемых сущностных классов и их свойств. Затем понадобится сгенерировать модуль исходного кода, запустив SQLMetal снова — на этот раз на файле DBML, а не на базе данных, и указав опцию /code. Или же можно загрузить промежуточный DBML-файл в Object Relational Designer, отредактировать его и позволить визуальному конструктору сгенерировать исходный код необходимых сущностных классов.

Другая причина того, что SQLMetal может генерировать два типа XML файлов — XML-файла отображения и промежуточного DBML-файла — состоит в путанице из-за их сходства. Так что не удивляйтесь, когда увидите, насколько они похожи. Схема XML-файла отображения рассматривается в главе 15.

Работа с промежуточным файлом DBML

Как уже было сказано, назначение промежуточного файла DBML — позволить вмешаться в процесс между извлечением схемы базы данных и генерацией сущностных классов, чтобы управлять именами генерируемых сущностных классов и их имен. Поэтому, если такая необходимость отсутствует, то генерировать промежуточный файл DBML незачем. Пока продолжим, исходя из того, что такая необходимость есть.

Предполагая, что имеется расширенная база данных Northwind, присоединенная к серверу базы данных SQL Server, вот как можно создать промежуточный файл DBML:

```
sqlmetal /server:.\SQLExpress /database:Northwind /pluralize /sprocs /functions
/views /dbml:Northwind.dbml
```

На заметку! Указание опций /server и /database при запуске SQLMetal требует, чтобы расширенная база данных Northwind была присоединена к SQL Server.

Вдобавок может понадобиться указать соответствующие опции /user и /password, чтобы SQLMetal мог подключиться к базе данных.

Или же можно сгенерировать промежуточный файл DBML из файла MDF:

```
sqlmetal /pluralize /sprocs /functions /views /dbml:Northwind.dbml
"C:\Northwind.mdf"
```

На заметку! Генерация промежуточного файла DBML из файла MDF может потребовать присоединения этого MDF-файла к SQL Server с именем C:\NORTHWIND.MDF или подобным. Для корректной работы примеров необходимо переименовать базу данных в "Northwind" внутри SQL Server Enterprise Manager или SQL Management Studio.

Любой из двух описанных подходов должен привести к генерации идентичных промежуточных DBML-файлов. Указаны только те опции, которые существенны для чтения базы данных и производства DBML-файла. Такие опции, как `/language` и `/code`, имеют значение только при создании модуля исходного кода.

Отредактировав промежуточный XML-файл, можно сгенерировать модуль исходного кода следующим образом:

```
sqlmetal /namespace:nwind /code:Northwind.cs Northwind.dbml
```

Опции, которые были заданы при запуске SQLMetal, важны при генерации исходного кода.

Схема промежуточного файла DBML

Если решено пройти путь создания промежуточного файла DBML, так, чтобы можно было отредактировать его и затем сгенерировать на его основе сущностные классы, нужно знать схему промежуточного файла, а также то, что означают его имена элементов и атрибутов.

Поскольку схема меняется, проконсультируйтесь с документацией Microsoft относительно схемы промежуточного файла DBML, чтобы получить наиболее свежее определение схемы и пояснения к ней. Разобравшись в схеме, вы можете вручную редактировать промежуточный DBML-файл для управления именами генерируемых SQLMetal на основе этого промежуточного файла сущностных классов и их свойств.

Или, что еще лучше — можно загрузить сгенерированный DBML-файл в Object Relational Designer внутри Visual Studio и отредактировать его там. Это средство предоставит графический интерфейс пользователя для манипуляций объектно-реляционной моделью, а также избавит от необходимости знания и понимания схемы. В следующем разделе будет показано, как отредактировать объектно-реляционную модель.

Object Relational Designer

В дополнение к утилите SQLMetal существует также графический инструмент пользователя для генерации сущностных классов, который работает в среде Visual Studio. Этот инструмент называется Object Relational Designer, и на него часто ссылаются, как на LINQ to SQL Designer, O/R Designer или даже DLinq Designer. Визуальный конструктор Object Relational Designer — более избирательный инструмент, чем SQLMetal. Визуальный конструктор предоставляет в распоряжение разработчика средства моделирования сущностных классов методом перетаскивания. При этом беспокоиться не нужно: визуальный конструктор сделает большую часть рутинной работы. Ваше дело — выбрать таблицы базы данных, которые хотите моделировать, если они доступны, и отредактировать имена сущностных классов и их свойств. Конечно, если необходим полный контроль, то в визуальном конструкторе можно выполнить все моделирование вручную.

Создание файла классов LINQ to SQL

Первый шаг в использовании визуального конструктора — создание файла классов LINQ to SQL посредством щелчка правой кнопкой мыши на проекте и выбора в контекстном меню пункта Add⇨New Item (Добавить⇨Новый элемент). Откроется диалоговое окно Add New Item (Добавить новый элемент). Выберите в списке установленных шаблонов LINQ to SQL Classes (Классы LINQ to SQL). Измените имя по своему усмотрению. Имя моделируемой базы данных — обычно подходящий выбор для имени файла классов LINQ to SQL. Расширение файла классов LINQ to SQL выглядит как `.dbml`. В рассматриваемом примере в качестве имени файла используется `Northwind.dbml`.

Внимание! Если вы создаете файл по имени `Northwind.dbml` в уже созданном проекте для примеров этой книги, будьте осторожны, чтобы не вызвать конфликт между кодом, сгенерированным визуальным конструктором, и существующим вашим кодом.

Указав имя для файла, щелкните на кнопке Add (Добавить). После этого появится пустое окно. Это — поверхность проектирования визуального конструктора. На рис. 13.1 показано, как она выглядит.

Рис. 13.1. Поверхность проектирования Object Relational Designer

Щелкнув на этой поверхности и заглянув в окно Properties (Свойства), вы увидите там свойство по имени `Name`. Значением свойства `Name` будет имя сгенерированного класса-наследника `DataContext`. Поскольку файл классов LINQ to SQL назван `Northwind.dbml`, по умолчанию значением свойства `Name` будет `NorthwindDataContext`, что вполне подходит.

В окне Solution Explorer (Проводник решения) непосредственно под файлом `Northwind.dbml` есть еще один файл по имени `Northwind.designer.cs`. Открыв его, вы увидите, что пока он содержит очень мало кода. По сути, он будет содержать конструкторы для нового класса-наследника `DataContext`, который называется `NorthwindDataContext`.

Подключение `DataContext` к базе данных

Следующий шаг — добавление в окне Server Explorer (Проводник сервера) подключения к соответствующему серверу, который обслуживает базу данных Northwind, если это подключение еще не существует.

Совет. Если окно Server Explorer не видно, выберите пункт Server Explorer в меню View (Вид) внутри Visual Studio.

Рис. 13.2 Диалоговое окно Add Connection

Чтобы добавить подключение к базе данных, щелкните правой кнопкой мыши на узле Data Connections (Подключения к данным) в окне Server Explorer и выберите в контекстном меню пункт Add Connection (Добавить подключение) для открытия диалогового окна Add Connection (Добавление подключения), показанного на рис. 13.2. Поле ввода Data source (Источник данных) по умолчанию будет содержать Microsoft SQL Server (SqlClient), что и требуется. Сконфигурируйте соответствующие настройки базы данных Northwind в диалоговом окне Add Connection.

Сконфигурировав подключение, щелкните на кнопке OK. Должен появиться узел, представляющий подключение к базе данных Northwind — прямо под узлом Data Connections в окне Server Explorer. Теперь имеется доступ к базе данных Northwind в проводнике.

Прежде чем двигаться дальше, удостоверьтесь, что файл `Northwind.dbml` можно просмотреть в редакторе Visual Studio.

Добавление сущностного класса

Найдите базу данных Northwind в списке Data Connections в окне Server Explorer. Разверните узел Tables (Таблицы) и вы увидите список таблиц базы данных Northwind. Сущностные классы создаются перетаскиванием таблиц из списка Tables в окне Server Explorer на поверхность проектирования.

Перетащите таблицу `Customers` из окна Server Explorer на поверхность проектирования. Тем самым вы указываете визуальному конструктору создать сущностный класс для таблицы `Customers` по имени `Customer`. После этого поверхность проектирования должна выглядеть, как показано на рис. 13.3.

Может понадобиться изменить размеры некоторых панелей, чтобы ясно все видеть. За счет перетаскивания таблицы `Customers` на поверхность проектирования визуального конструктора добавляется исходный код сущностного класса `Customer` в файл `Northwind.designer.cs`. После построения проекта можно приступать к использованию сущностного класса `Customer` для доступа и обновления данных в базе Northwind. Вот как все просто!

Однако прежде чем строить проект и писать код, использующий сгенерированные сущностные классы, следует создать еще несколько дополнительных вещей, необходимых для использования всех преимуществ LINQ to SQL. Теперь перетащите на поверхность проектирования таблицу `Orders` из окна Server Explorer. Подходящим образом расположите ее на поверхности. Это указывает визуальному конструктору на необходимость создания сущностного класса для таблицы `Orders` по имени `Order`. После этого поверхность визуального конструктора будет выглядеть, как показано на рис. 13.4.

На поверхность проектирования видна пунктирная линия, соединяющая класс `Customer` с классом `Order`. Эта пунктирная линия представляет отношение, которое в LINQ to SQL называется *ассоциацией*, между таблицами `Customers` и `Orders`, заданное ограничением внешнего ключа `FK_Orders_Customers` и существующее в базе данных Northwind. Эта линия указывает на то, что визуальный конструктор также создаст необходимую ассоциацию в сущностных классах для поддержки отношений между этими

двумя сущностными классами. Наличие такой ассоциации позволит получать ссылку на коллекцию заказов определенного заказчика через некоторое свойство объекта `Customer`, а также получать ссылку на заказчика, разместившего конкретный заказ, через соответствующее свойство объекта `Order`.

Если сгенерированная ассоциация не нужна, выберите пунктирную линию, представляющую ее, и удалите нажатием <Delete>, или щелкните правой кнопкой мыши и выберите в контекстном меню пункт Delete (Удалить).

Рис. 13.3. Внешний вид визуального конструктора после добавления таблицы `Customers` на поверхность проектирования

Рис. 13.4. Визуальный конструктор после перетаскивания таблица `Orders` на поверхность проектирования

Использование сгенерированных визуальным конструктором сущностных классов

Теперь все готово к использованию сущностных классов, сгенерированных визуальным конструктором. В листинге 13.2 приведен пример запроса базы данных для получения списка заказчиков из Лондона.

Листинг 13.2. Пример использования сгенерированных визуальным конструктором сущностных классов

```
NorthwindDataContext db = new NorthwindDataContext();
IQueryable<Customer> custs = from c in db.Customers
                             where c.City == "London"
                             select c;
foreach(Customer c in custs)
{
  Console.WriteLine("{0} имеет {1} заказов.", c.CompanyName, c.Orders.Count);
}
```

Это похоже на другой пример с исключением. Обратите внимание, что при создании экземпляра объекта `NorthwindDataContext` не было указано никакой информации о соединении. Дело в том, что визуальный конструктор сгенерировал класс `NorthwindDataContext` с конструктором без параметров, который получает информацию соединения из файла настроек проекта по имени `app.config`. Он даже установил соответствующее значение в файле настроек. Ниже показано, как выглядит сгенерированный конструктор без параметров.

Конструктор `DataContext`, сгенерированный визуальным конструктором

```
public NorthwindDataContext() :
  base(global::LINQChapter13.Properties.Settings.Default.
NorthwindConnectionString,
      mappingSource)
{
  OnCreated();
}
```

Внимание! Если вы загрузили код примеров, сопровождающий эту книгу, не забудьте обновить настройку `connectionString` в файле `app.config`. В частности, источник данных будет содержать имя машины, которое, скорее всего, не совпадет с именем вашей машины.

В предыдущем коде обратите внимание на возможность доступа к заказам полученного заказчика через свойство `Orders` объекта `Customer`. Это позволяет делать ассоциация, автоматически созданная визуальным конструктором. Здорово, не правда ли? А вот результат работы кода из листинга 13.2:

```
Around the Horn имеет 13 заказов.
B's Beverages имеет 10 заказов.
Consolidated Holdings имеет 3 заказов.
Eastern Connection имеет 8 заказов.
North/South имеет 3 заказов.
Seven Seas Imports имеет 9 заказов.
```

Редактирование модели сущностных классов

Естественно, может понадобится управлять именами сущностных классов, их свойствами (настройками сущностных классов), именами этих свойств (членов сущностных классов) и свойствами (членами сущностных классов) свойств (настройками). Система именования, предложенная Microsoft, достаточно запутана. Действительно ли была

необходимость называть члены классов *свойствами*, зная, что в Visual Studio *свойствами* обозначаются настройки?

Гибкость и простота использования для управления именами и свойствами сущностный классов — это то, что делает визуальный конструктор столь привлекательным. Здесь практически все можно делать с помощью щелчков и перетаскивания.

Редактирование имени сущностного класса

Чтобы отредактировать имя сущностного класса, дважды щелкните на его имени либо выберите сущностный класс на поверхности проектирования и модифицируйте свойство `Name` в окне Properties (Свойства).

Редактирование свойств сущностного класса (настроек сущностного класса)

Можно редактировать свойства как настройки сущностного класса, выбрав нужный класс в панели и модифицировав соответствующие его свойства в окне Properties. Есть возможность редактировать имя таблицы базы данных, в которой эти сущности хранятся, переопределенные методы вставки, обновления и удаления, а также другие свойства.

Редактирование имени свойства сущностного класса (члена сущностного класса)

Можно редактировать имя свойства сущностного класса как члена сущностного класса, выполнив тройной щелчок на имени свойства. Пусть не удивляет такая экзотическая вещь, как тройной щелчок — она существует и действует. Или же можно выбрать свойство сущностного класса на поверхности и отредактировать его свойство `Name` в окне Properties.

Редактирование свойств свойства сущностного класса (настройки)

Можно редактировать свойства свойств сущностного класса, выбрав свойство на поверхности проектирования и модифицировав соответствующее свойство в окне Properties. В этом окне вы найдете все свойства, соответствующие свойствам атрибутов сущностного класса, такие как `Name` и `UpdateCheck`, для атрибута `Column` сущностного класса. Атрибуты сущностного класса рассматриваются в главе 15.

Добавление объектов к модели сущностных классов

Перетащить сущностный класс на поверхность достаточно просто — до тех пор, пока есть соответствующая таблица базы данных в Server Explorer. Но бывают случаи, когда вы лишены такого удобства. Возможно, вы определили сущностный класс сначала, и планируете генерировать базу данных, вызвав метод `CreateDatabase` на объекте `DataContext`. Или, может быть, вы собираетесь воспользоваться преимуществом наследования сущностного класса, и нет соответствующей таблицы для отображения.

Добавление нового сущностного класса

Один из способов добавления новых классов к модели сущностных классов заключается в перетаскивании таблиц базы данных из окна Server Explorer, как это делалось в предыдущем разделе. Другой способ создания сущностного класса состоит в перетаскивании на поверхность проектирования объекта Object Relational Designer Class из панели инструментов Visual Studio. Отредактируйте имя и установите свойства сущностного класса, как было описано в предыдущем разделе.

Добавление новых свойств (членов) сущностного класса

Для добавления новых свойств (членов) сущностного класса необходимо выполнить щелчок правой кнопкой мыши на сущностном классе в визуальном конструкторе и выбрать в контекстном меню пункт Add⇨Properties (Добавить⇨Свойства). Как только

свойство добавлено к сущностному классу, следуйте инструкциям по редактированию свойств свойства сущностного класса, приведенным в разделе “Редактирование свойств свойства сущностного класса (настройки)”.

Добавление новой ассоциации

Вместо использования механизма перетаскивания для создания ассоциации, как вы поступали при добавлении нового сущностного класса из панели инструментов Visual Studio, ассоциация создается щелчком на объекте Association (Ассоциация) в панели инструментов, за которым следует щелчок на сущностном классе, задающем сторону *один* в отношении “один ко многим”, а за ним — щелчок на дочернем сущностном классе, задающем сторону *многие* в том же отношении. Каждый из этих классов должен иметь соответствующее свойство перед тем, как вы добавите ассоциацию, чтобы можно было отобразить первичный ключ на стороне *один* на внешний ключ на стороне *многие*. Выбор второго класса (стороны *многие*) ассоциации приводит к открытию диалогового окна Association Editor (Редактор ассоциации), которое позволяет отобразить свойство первого класса на соответствующее свойство второго класса.

Надлежащим образом отобразив свойства и закрыв окно Association Editor, вы увидите пунктирную линию, соединяющую родительский и дочерний сущностные классы.

Выберите ассоциацию щелчком на пунктирной линии и установите соответствующие свойства в окне Properties. За дополнительной информацией о свойствах ассоциации обратитесь к описанию атрибута `Association` и его свойств в главе 15.

Добавление нового наследования

Object Relational Designer можно также использовать для моделирования отношения наследования. Добавление отношения наследования подобно добавлению новой ассоциации. Выберите объект Inheritance (Наследование) в панели инструментов Visual Studio Toolbox и щелкните на сущностном классе, который будет наследником, а затем — на сущностном классе, который будет базовым. Удостоверьтесь, что установлены все соответствующие свойства сущностного класса, как определено атрибутами `InheritanceMapping` и `Column` сущностного класса, которые рассматриваются в главе 15.

Добавление хранимых процедур и функций, определяемых пользователем

Чтобы заставить визуальный конструктор сгенерировать код, необходимый для вызова хранимых процедур и функций, определяемых пользователем, перетащите хранимую процедуру или пользовательскую функцию из Server Explorer в панель Methods (Методы) визуального конструктора. Это будет показано в следующем разделе.

Переопределение методов вставки, обновления и удаления

В главе 14 будет описано переопределение методов вставки, обновления и удаления, используемых LINQ to SQL при проведении изменений в объекте сущностного класса. Добавляя специфические методы в сущностный класс, можно переопределять методы по умолчанию. Приняв такой подход, не забудьте использовать частичные классы, чтобы не модифицировать сгенерированный код. В главе 14 будет показано, как это делается.

Однако переопределение методов вставки, обновления и удаления также легко выполняется в визуальном конструкторе. Предположим, что есть хранимая процедуры по имени `InsertCustomer`, которая будет вставлять запись о новом заказчике в таблицу `Customers` базы данных Northwind. Ниже показана хранимая процедура, которая для этого используется.

Хранимая процедура `InsertCustomer`

```
CREATE PROCEDURE dbo.InsertCustomer
(
  @CustomerID nchar(5),
  @CompanyName nvarchar(40),
  @ContactName nvarchar(30),
  @ContactTitle nvarchar(30),
  @Address nvarchar(60),
  @City nvarchar(15),
  @Region nvarchar(15),
  @PostalCode nvarchar(10),
  @Country nvarchar(15),
  @Phone nvarchar(24),
  @Fax nvarchar(24)
)
AS
  INSERT INTO Customers
  (
    CustomerID,
    CompanyName,
    ContactName,
    ContactTitle,
    Address,
    City,
    Region,
    PostalCode,
    Country,
    Phone,
    Fax
  )
  VALUES
  (
    @CustomerID,
    @CompanyName,
    @ContactName,
    @ContactTitle,
    @Address,
    @City,
    @Region,
    @PostalCode,
    @Country,
    @Phone,
    @Fax
  )
```

На заметку! Хранимая процедура `InsertCustomer` не является частью расширенной базы данных Northwind. Она добавлена вручную в целях демонстрации.

Чтобы переопределить метод вставки сущностного класса `Customer`, сначала убедитесь, что панель Methods видима. Если нет, щелкните правой кнопкой мыши в панели визуального конструктора и выберите в контекстном меню пункт Show Methods Pane (Показать панель методов). Затем откройте окно Server Explorer в Visual Studio, если оно еще не открыто. Найдите и разверните узел Stored Procedures (Хранимые процедуры) в узле соответствующей базы данных. Окно Visual Studio должно выглядеть подобно показанному на рис. 13.5.

Рис. 13.5. Нахождение хранимой процедуры

Найдя хранимую процедуру, перетащите ее в панель Methods, которая представляет собой окно справа от модели сущностных классов. На рис. 13.6 показано окно Visual Studio после перетаскивания хранимой процедуры `InsertCustomer` в панель Methods.

Рис. 13.6. Перетаскивание хранимой процедуры в панель Methods

Перетаскивание хранимой процедуры из окна Server Explorer в панель Methods — способ, которым визуальный конструктор инструктируется о необходимости сгенерировать код, требуемый для вызова хранимой процедуры из LINQ to SQL. Точно так же визуальный конструктор инструктируется о необходимости генерации кода вызова пользовательской функции.

Обеспечение доступа к хранимой процедуре из LINQ to SQL — первый шаг к выполнению операций вставки, обновления или удаления посредством вызова хранимой

процедуры вместо обычного метода по умолчанию. Следующий шаг — переопределение одной из этих операций для вызова доступной теперь хранимой процедуры.

Теперь, когда хранимая процедура `InsertCustomer` находится в панели Methods, выберите класс `Customer` на поверхности проектирования и просмотрите окно свойств Properties. Теперь вы увидите список Default Methods (Методы по умолчанию). Выберите метод `Insert`, щелкнув на нем. Вы получите кнопку выбора с многоточием (...), как показано на рис. 13.7.

Теперь щелкните на кнопке с многоточием для открытия диалогового окна Configure Behavior (Конфигурировать поведение). Выберите переключатель Customize (Настроить) и в раскрывающемся списке выберите хранимую процедуру `InsertCustomer`. Отобразите аргументы метода в списке Method Arguments слева на соответствующие свойства пользовательского класса в списке Class Properties справа, как показано на рис. 13.8.

Рис. 13.7. Выбор метода `Insert` в категории Default Methods окна Properties

Рис. 13.8. Отображение аргументов метода на свойства класса

Теперь все аргументы метода отображены по умолчанию на соответствующие свойства класса.

После отображения всех аргументов метода щелкните на кнопке OK. После этого вы готовы к вставке записей Customer посредством вызова хранимой процедуры InsertCustomer. В листинге 13.3 новый заказчик создается с использованием хранимой процедуры InsertCustomer.

Листинг 13.3. Создание записи Customer с переопределенным методом вставки по умолчанию

```
NorthwindDataContext db = new NorthwindDataContext();

db.Log = Console.Out;

Customer cust =
  new Customer
    {
      CustomerID = "EWICH",
      CompanyName = "Every 'Wich Way",
      ContactName = "Vickey Rattz",
      ContactTitle = "Owner",
      Address = "105 Chip Morrow Dr.",
      City = "Alligator Point",
      Region = "FL",
      PostalCode = "32346",
      Country = "USA",
      Phone = "(800) EAT-WICH",
      Fax = "(800) FAX-WICH"
    };

db.Customers.InsertOnSubmit(cust);
db.SubmitChanges();

Customer customer = db.Customers.Where(c => c.CustomerID == "EWICH").First();
Console.WriteLine("{0} - {1}", customer.CompanyName, customer.ContactName);

// Восстановление базы данных.
db.Customers.DeleteOnSubmit(cust);
db.SubmitChanges();
```

Обратите внимание, что никакое пространство имен для класса Customer не указывается, т.е. используется класс Customer, находящийся в пространстве имен проекта — сгенерированный визуальным конструктором класс Customer.

В листинге 13.3 нет ничего особенного. Здесь просто создается экземпляр DataContext, которым в данном случае является сгенерированный визуальным конструктором NorthwindDataContext. Затем создается новый объект Customer, который вставляется в свойство Customers типа Table<T>. После этого вызывается метод SubmitChanges для сохранения нового заказчика в базе данных. Далее этот заказчик запрашивается из базы данных и отображается на консоли — просто чтобы доказать, что запись действительно была вставлена в таблицу базы. И последнее, что делается в коде — только что вставленный заказчик удаляется вызовом метода DeleteOnSubmit с фиксацией изменений в базе данных с помощью метода SubmitChanges. Это позволяет оставить базу данных в изначальном состоянии. Последующие примеры будут работать правильно, а этот пример может быть выполнен многократно. Рассмотрим вывод программы из листинга 13.3:

```
EXEC @RETURN_VALUE = [dbo].[InsertCustomer] @CustomerID = @p0, @CompanyName = @p1,
@ContactName = @p2, @ContactTitle = @p3, @Address = @p4, @City = @p5, @Region = @p6,
@PostalCode = @p7, @Country = @p8, @Phone = @p9, @Fax = @p10
-- @p0: Input StringFixedLength (Size = 5; Prec = 0; Scale = 0) [EWICH]
-- @p1: Input String (Size = 15; Prec = 0; Scale = 0) [Every 'Wich Way]
-- @p2: Input String (Size = 12; Prec = 0; Scale = 0) [Vickey Rattz]
-- @p3: Input String (Size = 5; Prec = 0; Scale = 0) [Owner]
-- @p4: Input String (Size = 19; Prec = 0; Scale = 0) [105 Chip Morrow Dr.]
-- @p5: Input String (Size = 15; Prec = 0; Scale = 0) [Alligator Point]
-- @p6: Input String (Size = 2; Prec = 0; Scale = 0) [FL]
-- @p7: Input String (Size = 5; Prec = 0; Scale = 0) [32346]
-- @p8: Input String (Size = 3; Prec = 0; Scale = 0) [USA]
-- @p9: Input String (Size = 14; Prec = 0; Scale = 0) [(800) EAT-WICH]
-- @p10: Input String (Size = 14; Prec = 0; Scale = 0) [(800) FAX-WICH]
-- @RETURN_VALUE: Output Int32 (Size = 0; Prec = 0; Scale = 0) []
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.20706.1

SELECT TOP 1 [t0].[CustomerID], [t0].[CompanyName], [t0].[ContactName],
[t0].[ContactTitle], [t0].[Address], [t0].[City], [t0].[Region], [t0].
[PostalCode],
[t0].[Country], [t0].[Phone], [t0].[Fax]
FROM [dbo].[Customers] AS [t0]
WHERE [t0].[CustomerID] = @p0
-- @p0: Input String (Size = 5; Prec = 0; Scale = 0) [EWICH]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.20706.1

Every 'Wich Way - Vickey Rattz
DELETE FROM [dbo].[Customers] WHERE ([CustomerID] = @p0) AND ([CompanyName] = @p1)
AND ([ContactName] = @p2) AND ([ContactTitle] = @p3) AND ([Address] = @p4) AND
([City] = @p5) AND ([Region] = @p6) AND ([PostalCode] = @p7) AND ([Country] = @p8)
AND ([Phone] = @p9) AND ([Fax] = @p10)
-- @p0: Input StringFixedLength (Size = 5; Prec = 0; Scale = 0) [EWICH]
-- @p1: Input String (Size = 15; Prec = 0; Scale = 0) [Every 'Wich Way]
-- @p2: Input String (Size = 12; Prec = 0; Scale = 0) [Vickey Rattz]
-- @p3: Input String (Size = 5; Prec = 0; Scale = 0) [Owner]
-- @p4: Input String (Size = 19; Prec = 0; Scale = 0) [105 Chip Morrow Dr.]
-- @p5: Input String (Size = 15; Prec = 0; Scale = 0) [Alligator Point]
-- @p6: Input String (Size = 2; Prec = 0; Scale = 0) [FL]
-- @p7: Input String (Size = 5; Prec = 0; Scale = 0) [32346]
-- @p8: Input String (Size = 3; Prec = 0; Scale = 0) [USA]
-- @p9: Input String (Size = 14; Prec = 0; Scale = 0) [(800) EAT-WICH]
-- @p10: Input String (Size = 14; Prec = 0; Scale = 0) [(800) FAX-WICH]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.20706.1
```

Хотя это не так просто сразу обнаружить в выводе, но SQL-оператор `insert` здесь не был создан. Вместо него вызывалась хранимая процедура `InsertCustomer`. Визуальный конструктор существенно облегчает задачу переопределения методов вставки, обновления и удаления для сущностного класса.

Совместное использование SQLMetal и Object Relational Designer

Поскольку формат промежуточного DBML-файла SQLMetal совпадает со схемой XML формата Object Relational Designer, вполне допускается использовать их совместно.

Например, можно сгенерировать промежуточный файл DBML для базы данных, используя для этого SQLMetal, а затем загрузить его в Object Relational Designer, чтобы подправить имена сущностных классов или их свойств по своему желанию. Такой под-

ход предоставляет простой способ генерации сущностных классов для всей базы данных, оставляя возможность простой модификации того, что нужно.

Другой пример, где такая взаимозаменяемость может оказаться удобной — возможность переопределения операций вставки, обновления и удаления, которые предназначены для проведения сущностным классом изменений в базе данных. Можно генерировать промежуточный файл DBML для базы данных посредством SQLMetal, но затем загружать его в визуальный конструктор и модифицировать методы вставки, обновления и удаления, как было описано в разделе настоящей главы, посвященном Object Relational Designer.

Резюме

В соответствии с принятым стилем, большая часть информации в этой главе может показаться преждевременной, поскольку нам еще только предстоит знакомство с сущностными классами и `DataContext`. Воспринимайте эти советы как фундамент для лучшего понимания того, что будет изложено далее.

Помните, что доступны два инструмента для моделирования сущностных классов. Первый — SQLMetal — является инструментом командной строки, больше подходящим для генерации сущностных классов для всей базы данных в целом. Второй — Object Relational Designer, часто называемый LINQ to SQL Designer — инструмент с графическим интерфейсом пользователя, позволяющий выполнять моделирование сущностных классов методом перетаскивания, который запускается в среде Visual Studio. Он больше подходит для итеративной и новой разработки. Но, как уже отмечалось, эти два инструмента вполне могут работать совместно. Лучше всего начать с SQLMetal для генерации сущностных классов для всей базы данных, а затем работать с ними в Object Relational Designer.

Вооружившись некоторыми знаниями относительно инструментов LINQ to SQL, вы можете переходить к созданию собственных сущностных классов. В главе 14 будет показано, как выполнять наиболее распространенные операции с базой данных, с которыми вам придется регулярно иметь дело.

ГЛАВА 14

Операции LINQ to SQL для баз данных

В этой главе будут описаны типовые операции LINQ to SQL, предназначенные для баз данных. В частности, будет показано, как выполняются:

- вставки;
- запросы;
- обновления;
- удаления.

После рассмотрения стандартных операций для баз данных будет продемонстрировано, как переопределять методы вставки, обновления и удаления по умолчанию, используемые сущностными классами для сохранения изменений в базе данных.

И последняя тема, которая раскрывается в главе — автоматическая трансляция запросов LINQ to SQL, включая моменты, на которые следует особо обращать внимание при написании запросов.

Для того чтобы говорить о стандартных операциях для баз данных, придется обращаться к `DataContext` и сущностным классам. Пока еще вы не получили достаточно подробной информации о работе сущностных классов и `DataContext`, но она будет приведена в последующих главах. Сущностные классы рассматриваются в главе 15, а `DataContext` — в главе 16. А пока просто запомните, что `DataContext` управляет подключением к базе данных, а также объектами сущностных классов. Объект сущностного класса представляет определенную запись базы данных в форме объекта.

Предварительные условия для запуска примеров

Для запуска примеров этой главы понадобится расширенная версия базы данных Northwind и сгенерированные для нее сущностные классы. Прочтите еще раз и выполните требования раздела “Предварительные условия для запуска примеров” из главы 12.

Некоторые общие методы

Дополнительно для запуска примеров этой главы понадобятся некоторые общие методы, используемые примерами. Прочтите и выполните требования раздела “Некоторые общие методы” главы 12.

Использование API-интерфейса LINQ to SQL

Чтобы запускать примеры этой главы, в проект нужно будет добавить необходимые ссылки и директивы `using`. Прочтите и выполните то, что сказано в разделе "Использование LINQ to SQL" главы 12.

Стандартные операции для баз данных

Хотя детали выполнения запросов LINQ to SQL будут изложены в последующих главах, здесь предоставляются некоторые поверхностные сведения о том, как выполнять рутинные операции с базой данных, без излишнего усложнения. Эти примеры предназначены только для демонстрации базовых концепций. И как таковые, они не будут включать в себя проверку ошибок и обработку исключений.

Например, поскольку многие из базовых операций, о которых пойдет речь, вносят изменения в базу данных, при этом следовало бы позаботиться об обнаружении и разрешении конфликтов параллельного доступа. Но в целях простоты рассматриваемые в главе примеры не будут следовать этим принципам. Обнаружение и разрешение таких конфликтов детально рассматривается в главе 17.

Вставки

Для осуществления вставки необходимо выполнить четыре шага. Первый шаг — создание `DataContext`. Это первый шаг для любого запроса LINQ to SQL.

Второй шаг — создание экземпляра сущностного объекта из сущностного класса (такого как `Customer`).

Третий шаг — вставка этого сущностного объекта в соответствующую коллекцию таблицы типа `Table<T>`, где `T` — тип сущностного класса, хранящегося в таблице, или добавление его к `EntitySet<T>` на сущностном объекте, уже отслеженном `DataContext`, где `T` — тип сущностного класса.

Четвертый и последний шаг — вызов метода `SubmitChanges` на объекте `DataContext`.

В листинге 14.1 приведен пример вставки записи в базу данных.

Листинг 14.1. Вставка записи посредством вставки сущностного объекта в `Table<T>`

```
// 1. Создание DataContext.
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");

// 2. Создание экземпляра сущностного объекта.
Customer cust =
  new Customer
    {
      CustomerID = "LAWN",
      CompanyName = "Lawn Wranglers",
      ContactName = "Mr. Abe Henry",
      ContactTitle = "Owner",
      Address = "1017 Maple Leaf Way",
      City = "Ft. Worth",
      Region = "TX",
      PostalCode = "76104",
      Country = "USA",
      Phone = "(800) MOW-LAWN",
      Fax = "(800) MOW-LAWO"
    };
// 3. Добавление сущностного объекта в таблицу Customers.
db.Customers.InsertOnSubmit(cust);
```

```
// 4. Вызов метода SubmitChanges.
db.SubmitChanges();

// 5. Запрос записи.
Customer customer = db.Customers.Where(c => c.CustomerID == "LAWN").First();
Console.WriteLine("{0} - {1}", customer.CompanyName, customer.ContactName);

// Эта часть кода просто восстанавливает базу данных,
// чтобы пример можно было запустить снова.
Console.WriteLine("Удаление ранее добавленного заказчика LAWN.");
db.Customers.DeleteOnSubmit(cust);
db.SubmitChanges();
```

Об этом примере говорить особо нечего. Во-первых, создается экземпляр объекта `Northwind`, чтобы иметь объект `DataContext` для базы данных Northwind. Во-вторых, создается экземпляр объекта `Customer`, который наполняется с использованием средства инициализации объектов. В-третьих, созданный экземпляр `Customer` вставляется в таблицу `Customers` типа `Table<Customer>`, в класс `DataContext` по имени `Northwind`. В-четвертых, вызывается метод `SubmitChanges` для сохранения вновь созданного объекта `Customer` в базе данных. И, наконец, из базы данных запрашивается только что вставленная запись о заказчике, чтобы доказать, что она на самом деле была вставлена.

На заметку! Если вы запустите этот пример, то новая запись для заказчика `LAWN` будет временно добавлена в таблицу `Customers` базы данных Northwind. Обратите внимание, что после того, как вновь добавленная запись запрошена и отображена, она удаляется. Это делается для того, чтобы пример можно было запускать многократно, и вновь добавленные записи не влияли на последующие примеры. Если любой пример, модифицирующий базу данных, по каким-то причинам не сможет успешно завершиться, вы должны вручную вернуть базу в исходное состояние.

Так выглядит результат запуска листинга 14.1:

```
Lawn Wranglers - Mr. Abe Henry
Удаление ранее добавленного заказчика LAWN.
```

Как видите, вставленная запись успешно найдена в базе данных.

В качестве альтернативы для вставки записи в базу данных можно добавить новый экземпляр сущностного класса в уже существующий сущностный объект, отслеживаемый объектом `DataContext`, как показано в листинге 14.2.

Листинг 14.2. Вставка записи в базу данных Nothwind добавлением к `EntitySet<T>`

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");

Customer cust = (from c in db.Customers
                 where c.CustomerID == "LONEP"
                 select c).Single<Customer>();

// Используется для запроса записи.
DateTime now = DateTime.Now;
Order order = new Order
{
  CustomerID = cust.CustomerID,
  EmployeeID = 4,
  OrderDate = now,
  RequiredDate = DateTime.Now.AddDays(7),
  ShipVia = 3,
  Freight = new Decimal(24.66),
```

```
  ShipName = cust.CompanyName,
  ShipAddress = cust.Address,
  ShipCity = cust.City,
  ShipRegion = cust.Region,
  ShipPostalCode = cust.PostalCode,
  ShipCountry = cust.Country
};
cust.Orders.Add(order);
db.SubmitChanges();

IEnumerable<Order> orders =
  db.Orders.Where(o => o.CustomerID == "LONEP" && o.OrderDate.Value == now);

foreach (Order o in orders)
{
   Console.WriteLine("{0} {1}", o.OrderDate, o.ShipName);
}

// Эта часть кода восстанавливает базу данных
db.Orders.DeleteOnSubmit(order);
db.SubmitChanges();
```

В листинге 14.2 создается объект `DataContext` по имени `Northwind`, извлекается заказчик и добавляется вновь сконструированный сущностный объект к свойству `Orders` типа `EntitySet<Order>` сущностного объекта `Customer`. Затем новая запись запрашивается и выводится на консоль.

В листинге 14.1 вставляемый объект, которым был `Customer`, добавлялся в переменную типа `Table<Customer>`. В листинге 14.2 вставляемый объект, которым является `Order`, добавляется в переменную типа `EntitySet<Order>`.

Ниже показан результат выполнения кода из листинга 14.2:

```
9/2/2007 6:02:16 PM Lonesome Pine Restaurant
```

Вставка присоединенных сущностных объектов

Класс `DataContext` обнаруживает любые зависимые ассоциированные объекты сущностных классов, которые присоединены к данному объекту, так что они также сохраняются при вызове метода `SubmitChanges`. Под зависимыми подразумеваются объекты сущностных классов, содержащие внешний ключ, который указывает на вставляемый объект сущностного класса. Пример приведен в листинге 14.3.

Листинг 14.3. Добавление присоединенных записей

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial
Catalog=Northwind");
Customer cust =
  new Customer {
    CustomerID = "LAWN",
    CompanyName = "Lawn Wranglers",
    ContactName = "Mr. Abe Henry",
    ContactTitle = "Owner",
    Address = "1017 Maple Leaf Way",
    City = "Ft. Worth",
    Region = "TX",
    PostalCode = "76104",
    Country = "USA",
    Phone = "(800) MOW-LAWN",
    Fax = "(800) MOW-LAWO",
    Orders = {
```

```
      new Order {
        CustomerID = "LAWN",
        EmployeeID = 4,
        OrderDate = DateTime.Now,
        RequiredDate = DateTime.Now.AddDays(7),
        ShipVia = 3,
        Freight = new Decimal(24.66),
        ShipName = "Lawn Wranglers",
        ShipAddress = "1017 Maple Leaf Way",
        ShipCity = "Ft. Worth",
        ShipRegion = "TX",
        ShipPostalCode = "76104",
        ShipCountry = "USA"
      }
    }
  };
db.Customers.InsertOnSubmit(cust);
db.SubmitChanges();
Customer customer = db.Customers.Where(c => c.CustomerID == "LAWN").First();
Console.WriteLine("{0} - {1}", customer.CompanyName, customer.ContactName);
foreach (Order order in customer.Orders)
{
  Console.WriteLine("{0} - {1}", order.CustomerID, order.OrderDate);
}
// Эта часть кода восстанавливает базу данных.
db.Orders.DeleteOnSubmit(cust.Orders.First());
db.Customers.DeleteObSubmit(cust);
db.SubmitChanges();
```

В листинге 14.3 создается новый объект `Customer` с присоединенной к нему коллекцией `Orders`, содержащей один вновь созданный экземпляр `Order`. Несмотря на то что вставляется только объект `cust` типа `Customer` в таблицу `Customers`, а заказы явно не вставляются в таблицу `Orders`, при вызове метода `SubmitChanges` этот новый `Order` также будет сохранен в базе данных, поскольку новый `Order` присоединен к новому `Customer`.

Относительно этого примера нужно сделать одно замечание. Обратите внимание, что в коде очистки в конце листинга 14.3 вызывается метод `DeleteOnSubmit` как на новом объекте `Order`, так и на `Customer`. В данном случае удаляется только первый `Order`, но поскольку объект `Customer` — новый, точно известно, что у него был только один `Order`. Необходимость в ручном удалении заказов вызвана тем, что хотя при вставке родительского объекта в базу данных все присоединенные к нему объекты вставляются автоматически, при удалении такого не происходит. Если не удалить заказы вручную, при попытке удалить заказчика было сгенерировано исключение. Более подробно речь об этом пойдет позже, в разделе “Удаления” настоящей главы.

Нажмем <Ctrl+F5> и посмотрим на вывод кода из листинга 14.3:

```
Lawn Wranglers - Mr. Abe Henry
LAWN - 9/2/2007 6:05:07 PM
```

Запросы

Выполнение запросов LINQ to SQL похоже на выполнение любого другого запроса LINQ, но с несколькими исключениями. Рассмотрим их очень кратко.

Чтобы выполнить запрос LINQ to SQL, сначала понадобится создать `DataContext`. Затем можно выполнять запрос в таблицу внутри этого `DataContext`, как показано в листинге 14.4.

Листинг 14.4. Выполнение простого запроса LINQ to SQL на базе данных Northwind

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
Customer cust = (from c in db.Customers
                 where c.CustomerID == "LONEP"
                 select c).Single<Customer>();
```

Когда выполняется этот код, заказчик, CustomerID которого равен "LONEP", извлекается в переменную cust. Однако следует иметь в виду, что, как упоминалось в главе 5, стандартная операция запроса Single сгенерирует исключение, если последовательность, на которой он вызван, не содержит подходящих элементов. Поэтому в данном случае нужно точно знать, что заказчик "LONEP" существует. В действительности стандартная операция запроса SingleOrDefault обеспечивает лучшую защиту на случай, если не найдется записи, соответствующей конструкции where.

В этом примере необходимо отметить еще пару моментов. Во-первых, обратите внимание, что в запросе при сравнении CustomerID с "LONEP" используется синтаксис C#. Об этом говорит применение двойных кавычек вместо одинарных, как того требует синтаксис SQL. К тому же используется операция проверки эквивалентности C# == вместо операции проверки эквивалентности SQL =. Это демонстрирует тот факт, что запрос на самом деле интегрирован в язык: в конце концов, это следует из самого названия LINQ — язык интегрированных запросов. Во-вторых, обратите внимание, что в этом запросе синтаксис выражений запроса смешивается со стандартным синтаксисом точечной нотации. Часть, представленная в синтаксисе выражений запросов, заключена в скобки, а операция Single вызывается с применением стандартной точечной нотации.

А теперь вопрос. Уже не раз на протяжении этой книги говорилось об отложенном выполнении запросов. Вопрос состоит в том, вызовет ли запуск приведенного кода немедленное выполнение запроса? Обдумывая ответ, не забудьте об отложенном выполнении запросов. Ответ: да, стандартная операция запроса Single вызовет немедленное выполнение запроса. Если исключить вызов этой операции и немедленно вернуть результат запроса, то запрос не будет выполнен немедленно.

Код в листинге 14.4 не выдает никакого экранного вывода, поэтому просто чтобы убедиться, что код действительно извлечет соответствующего заказчика, в листинге 14.5 приведен тот же код, но с добавлением вывода на консоль, отображающего извлеченную запись о заказчике.

Листинг 14.5. Выполнение того же запроса, но с выводом на консоль

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
Customer cust = (from c in db.Customers
                 where c.CustomerID == "LONEP"
                 select c).Single<Customer>();
Console.WriteLine("{0} - {1}", cust.CompanyName, cust.ContactName);
```

Ниже показан вывод листинга 14.5:

```
Lonesome Pine Restaurant - Fran Wilson
```

Отступления от нормы

Ранее упоминалось, что запросы LINQ to SQL подобны обычным запросам LINQ, но с некоторыми исключениями. Давайте обсудим эти исключения.

Запросы LINQ to SQL возвращают IQueryable<T>

В то время как запросы LINQ, выполненные на массивах и коллекциях, возвращают последовательности IEnumerable<T>, запросы LINQ to SQL, запрашивающие такую последовательность, возвращают последовательность типа IQueryable<T>. В листинге 14.6 приведен пример запроса, возвращающего последовательность типа IQueryable<T>.

Листинг 14.6. Простой запрос LINQ to SQL, возвращающий последовательность IQueryable<T>

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
IQueryable<Customer> custs = from c in db.Customers
                             where c.City == "London"
                             select c;
foreach(Customer cust in custs)
{
  Console.WriteLine("Заказчик: {0}", cust.CompanyName);
}
```

Как видите, типом возврата этого запроса является IQueryable<Customer>. Вот результат выполнения кода из листинга 14.6:

```
Заказчик: Around the Horn
Заказчик: B's Beverages
Заказчик: Consolidated Holdings
Заказчик: Eastern Connection
Заказчик: North/South
Заказчик: Seven Seas Imports
```

Как было указано в главе 12, поскольку IQueryable<T> реализует IEnumerable<T>, обычно можно трактовать последовательность типа IQueryable<T>, как если бы это была последовательность IEnumerable<T>. Если при этом возникнут проблемы — не забудьте об операции AsEnumerable.

Запросы LINQ to SQL выполняются на объектах Table<T>

В то время как большинство обычных запросов LINQ выполняются на массивах и коллекциях, реализующих интерфейсы IEnumerable<T> или IEnumerable, запрос LINQ to SQL выполняется на классах, реализующих интерфейс IQueryable<T>, таком как Table<T>.

Это значит, что запросам LINQ to SQL доступны дополнительные операции запросов, наряду со стандартными операциями запросов, поскольку IQueryable<T> реализует IEnumerable<T>.

Запросы LINQ to SQL транслируются в SQL

Как говорилось в главе 2, поскольку запросы LINQ to SQL возвращают последовательность типа IQueryable<T>, они не компилируются в код промежуточного языка .NET, как это делают обычные запросы LINQ. Вместо этого они преобразуются в деревья выражений, что позволяет им вычисляться как единое целое и транслироваться в соответствующие и оптимальные конструкции SLQ. Прочтите раздел, озаглавленный "Трансляция SQL" в конце настоящей главы, чтобы узнать больше о трансляции SQL, которая происходит в запросах LINQ to SQL.

Запросы LINQ to SQL выполняются в базе данных

В отличие от запросов LINQ, которые выполняются в памяти локальной машины, запросы LINQ to SQL транслируются в вызовы SQL, которые в действительности выпол-

няются в базе данных. Из-за этого происходит некоторое расхождение, такое как способ обработки проекции, которая не может в действительности случаться в базе данных, поскольку база ничего не знает о сущностных классах, как и о любых других классах.

К тому же, поскольку запрос на самом деле выполняется в базе данных, и база не имеет доступа к коду вашего приложения, то, что можно сделать в запросе, должно транслироваться, что ограничивает его некоторым образом в зависимости от возможностей транслятора. Нельзя просто встроить вызов написанного метода в лямбда-выражение и ожидать, что SQL Server догадается, что нужно делать с вызовом. Из-за этого неплохо бы знать, что может транслироваться, во что оно может быть транслировано, и что случится, когда трансляция невозможна.

Ассоциации

Запрос ассоциированного класса в LINQ to SQL не сложнее простого обращения к переменной-члену сущностного класса. Это объясняется тем, что ассоциированный класс *является* переменной-членом связанного сущностного класса или хранится в коллекции объектов сущностных классов, причем коллекция является переменной-членом связанного сущностного класса. Если ассоциированный класс представляет сторону *многие* в отношении "один ко многим", то объекты класса *многие* будут храниться в коллекции, тип которой — `EntitySet<T>`, а `T` — тип сущностного класса стороны *многие*. Эта коллекция будет членом класса стороны *один*, а ссылка на объект класса *один* станет членом класса стороны *многие*.

Например, рассмотрим случай сущностных классов `Customer` и `Order`, которые были сгенерированы для базы данных Northwind. Заказчик может иметь множество заказов, но заказ принадлежит только одному заказчику. В данном примере класс `Customer` находится на стороне *один* отношения "один ко многим" между сущностными классами `Customer` и `Order`. Класс `Order` находится на стороне *многие* того же отношения. Поэтому заказы, принадлежащие объекту `Customer`, могут быть доступны через переменную-член, обычно именованную `Orders`, типа `EntitySet<Order>` в классе `Customer`. Заказчик в объекте `Order` доступен через переменную-член, обычно именуемую `Customer`, типа `EntityRef<Customer>` в классе `Order` (рис. 14.1).

Рис. 14.1. Отношение ассоциации между родительским и дочерним сущностными классами

Классы ассоциированы за счет указания атрибута `Association` в свойстве класса, которое хранит ссылку на ассоциированный класс, в определении сущностного класса. Поскольку и родительский, и дочерний объекты имеют свойства класса, ссылающиеся друг на друга, атрибут `Association` указывается как в родительском, так и в дочернем сущностном классе. Атрибут `Association` более подробно рассматривается в главе 15.

В листинг 14.7 приведен пример, в котором запрашиваются определенные заказчики и затем отображаются вместе со всеми их заказами.

Листинг 14.7. Использование ассоциации для доступа к связанным данным

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
IQueryable<Customer> custs = from c in db.Customers
                        where c.Country == "UK" &&
                          c.City == "London"
                        orderby c.CustomerID
                        select c;
foreach (Customer cust in custs)
{
  Console.WriteLine("{0} - {1}", cust.CompanyName, cust.ContactName);
  foreach (Order order in cust.Orders)
  {
    Console.WriteLine(" {0} {1}", order.OrderID, order.OrderDate);
  }
}
```

Как видите, каждый заказчик перечисляется и отображается, затем также перечисляются и отображаются все его заказы. В запросе даже никак не указано, что нужны заказы. Вот сокращенный результат работы листинга 14.7:

```
Around the Horn - Thomas Hardy
   10355 11/15/1996 12:00:00 AM
   10383 12/16/1996 12:00:00 AM
   10453 2/21/1997 12:00:00 AM
   10558 6/4/1997 12:00:00 AM
   10707 10/16/1997 12:00:00 AM
   10741 11/14/1997 12:00:00 AM
   10743 11/17/1997 12:00:00 AM
   10768 12/8/1997 12:00:00 AM
   10793 12/24/1997 12:00:00 AM
   10864 2/2/1998 12:00:00 AM
   10920 3/3/1998 12:00:00 AM
   10953 3/16/1998 12:00:00 AM
   11016 4/10/1998 12:00:00 AM
...
Consolidated Holdings - Elizabeth Brown
   10435 2/4/1997 12:00:00 AM
   10462 3/3/1997 12:00:00 AM
   10848 1/23/1998 12:00:00 AM
...
```

Здесь может показаться, что это страшно неэффективно, если нигде нет обращения к заказам данного заказчика.

В действительности это не так. Причина в том, что заказы в действительности не извлекаются до тех пор, пока на них никто не ссылается. Если бы код не обращался к свойству `Orders` заказчика, они бы никогда и не извлекались. Это известно под названием *отложенной загрузки*, которую не следует путать с отложенным выполнением запроса.

Отложенная загрузка

Отложенная загрузка — это такая загрузка, при которой записи на самом деле не загружаются из базы данных до тех пор, пока это не становится абсолютно необходимым, что происходит при первом обращении к ним; таким образом, загрузка записей откладывается.

В листинге 14.7, если бы не были ни одной ссылки на переменную-член `Orders`, то заказы никогда бы не извлекались из базы данных. Для большинства ситуаций отложенная загрузка — вещь хорошая. Она предотвращает выполнение ненужных запросов и экономит пропускную способность сети за счет исключения пересылки ненужных данных.

Однако может возникнуть проблема. В листинге 14.8 показан такой же код, как в листинге 14.7, за исключением того, что на этот раз включено средство протоколирования, обеспечиваемое объектом `DataContext.Log`, чтобы выявить проблему.

Листинг 14.8. Пример, демонстрирующий отложенную загрузку

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");

IQueryable<Customer> custs = from c in db.Customers
                        where c.Country == "UK" &&
                          c.City == "London"
                        orderby c.CustomerID
                        select c;

// Включить протоколирование.
db.Log = Console.Out;

foreach (Customer cust in custs)
{
  Console.WriteLine("{0} - {1}", cust.CompanyName, cust.ContactName);
  foreach (Order order in cust.Orders)
  {
    Console.WriteLine(" {0} {1}", order.OrderID, order.OrderDate);
  }
}
```

В результате запуска этого примера нажатием <Ctrl+F5> получается следующий вывод (приведен с сокращениями):

```
SELECT [t0].[CustomerID], [t0].[CompanyName], [t0].[ContactName],
[t0].[ContactTitle], [t0].[Address], [t0].[City], [t0].[Region], [t0].[PostalCode],
[t0].[Country], [t0].[Phone], [t0].[Fax]
FROM [dbo].[Customers] AS [t0]
WHERE ([t0].[Country] = @p0) AND ([t0].[City] = @p1)
ORDER BY [t0].[CustomerID]
-- @p0: Input String (Size = 2; Prec = 0; Scale = 0) [UK]
-- @p1: Input String (Size = 6; Prec = 0; Scale = 0) [London]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.20706.1

Around the Horn - Thomas Hardy
SELECT [t0].[OrderID], [t0].[CustomerID], [t0].[EmployeeID], [t0].[OrderDate],
[t0].[RequiredDate], [t0].[ShippedDate], [t0].[ShipVia], [t0].[Freight],
[t0].[ShipName], [t0].[ShipAddress], [t0].[ShipCity], [t0].[ShipRegion],
[t0].[ShipPostalCode], [t0].[ShipCountry]
FROM [dbo].[Orders] AS [t0]
WHERE [t0].[CustomerID] = @p0
-- @p0: Input String (Size = 5; Prec = 0; Scale = 0) [AROUT]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.20706.1
```

```
   10355 11/15/1996 12:00:00 AM
   10383 12/16/1996 12:00:00 AM
   10453 2/21/1997 12:00:00 AM
   10558 6/4/1997 12:00:00 AM
   10707 10/16/1997 12:00:00 AM
   10741 11/14/1997 12:00:00 AM
   10743 11/17/1997 12:00:00 AM
   10768 12/8/1997 12:00:00 AM
   10793 12/24/1997 12:00:00 AM
   10864 2/2/1998 12:00:00 AM
   10920 3/3/1998 12:00:00 AM
   10953 3/16/1998 12:00:00 AM
   11016 4/10/1998 12:00:00 AM
B's Beverages - Victoria Ashworth
SELECT [t0].[OrderID], [t0].[CustomerID], [t0].[EmployeeID], [t0].[OrderDate],
[t0].[RequiredDate], [t0].[ShippedDate], [t0].[ShipVia], [t0].[Freight],
[t0].[ShipName], [t0].[ShipAddress], [t0].[ShipCity], [t0].[ShipRegion],
[t0].[ShipPostalCode], [t0].[ShipCountry]
FROM [dbo].[Orders] AS [t0]
WHERE [t0].[CustomerID] = @p0
-- @p0: Input String (Size = 5; Prec = 0; Scale = 0) [BSBEV]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.20706.1
   10289 8/26/1996 12:00:00 AM
   10471 3/11/1997 12:00:00 AM
   10484 3/24/1997 12:00:00 AM
   10538 5/15/1997 12:00:00 AM
   10539 5/16/1997 12:00:00 AM
   10578 6/24/1997 12:00:00 AM
   10599 7/15/1997 12:00:00 AM
   10943 3/11/1998 12:00:00 AM
   10947 3/13/1998 12:00:00 AM
   11023 4/14/1998 12:00:00 AM
Consolidated Holdings - Elizabeth Brown
...
```

Здесь SQL-запросы помечены полужирным, чтобы отличить их от выходных данных заказчика и заказов. В первом SQL-запросе запрашиваются только заказчики, и нет обращения к таблице заказов. Затем отображается имя компании и имя контакта для первой компании, после чего идет вывод второго SQL-запроса. Во втором SQL-запросе выполняется запрос к таблице `Orders` с определенным значением `CustomerID` в конструкции `where`. Поэтому запрос генерируется и выполняется только для определенного заказчика, который отображен на консоли. Далее отображается список заказов для последнего выведенного заказчика, за которым идет следующий заказчик. После этого появляется другой SQL-запрос заказов определенного заказчика.

Как видите, для извлечения заказов каждого заказчика выполняется отдельный запрос. Заказы не запрашиваются, а потому и не загружаются — до тех пор, пока не осуществляется обращение к переменной `Orders EntityRef<T>` во втором цикле `foreach`, который следует немедленно после отображения на консоли информации о заказчике. Так как заказы не извлекаются до тех пор, пока к ним не будет выполнено обращение, их загрузка откладывается.

Поскольку для каждого заказчика генерируется и выполняется отдельный запрос, потенциально в базу данных поступает множество SQL-запросов. Это может привести к проблемам с производительностью.

В коде точно известно, что планируется обращение к заказам извлеченных заказчиков. В данном случае можно было бы добиться лучшей производительности, если бы

удалось извлекать заказы вместе с извлечением заказчиков. То, что необходимо, называется *немедленной загрузкой*.

Немедленная загрузка с помощью класса DataLoadOptions

Хотя отложенная загрузка является поведением по умолчанию для ассоциированных классов, можно инициировать их немедленную загрузку. Немедленная загрузка заставляет ассоциированные классы загружаться до того, как к ним выполнится обращение. Это может дать выигрыш в производительности. С помощью операции LoadWidth<T> класса DataLoadOptions можно заставить DataContext немедленно загрузить ассоциированный класс, указанный в лямбда-выражении этой операции. Благодаря операции LoadWidth<T>, когда происходит действительное выполнение запроса, то при этом извлекается не только первичный класс, но также и ассоциированный с ним класс.

В листинге 14.9 используется тот же базовый код примера, что и в листинге 14.8, но здесь создается объект DataLoadOptions; на этом объекте вызывается операция LoadWith<T> с передачей члена Orders как класса, подлежащего немедленной загрузке вместе с объектом Customer; и объект DataLoadOptions присваивается объекту Northwind типа DataContext. Кроме того, чтобы исключить любые сомнения относительного того, что ассоциированные объекты — заказы загрузятся до того, как к ним произойдет обращение, код, выполняющий перечисление заказов текущего заказчика, опущен, чтобы на них не было никаких ссылок.

Листинг 14.9. Пример, демонстрирующий немедленную загрузку с использованием класса DataLoadOptions

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
DataLoadOptions dlo = new DataLoadOptions();
dlo.LoadWith<Customer>(c => c.Orders);
db.LoadOptions = dlo;
IQueryable<Customer> custs = from c in db.Customers
                        where c.Country == "UK" &&
                          c.City == "London"
                        orderby c.CustomerID
                        select c;
// Включить протоколирование.
db.Log = Console.Out;
foreach (Customer cust in custs)
{
  Console.WriteLine("{0} - {1}", cust.CompanyName, cust.ContactName);
}
```

Единственное отличие этого листинга от листинга 14.8 состоит в создании экземпляра объекта DataLoadOptions, вызове операции LoadWith<T>, присваивании объекта DataLoadOptions объекту Northwind типа DataContext и исключении ссылок на заказы каждого заказчика. В вызове операции LoadWith<T> объект DataLoadOptions инструктируется на немедленную загрузку Orders при загрузке объекта Customer. Теперь посмотрим на вывод программы из листинга 14.9.

```
SELECT [t0].[CustomerID], [t0].[CompanyName], [t0].[ContactName],
[t0].[ContactTitle], [t0].[Address], [t0].[City], [t0].[Region], [t0].[PostalCode],
[t0].[Country], [t0].[Phone], [t0].[Fax], [t1].[OrderID], [t1].[CustomerID] AS
[CustomerID2], [t1].[EmployeeID], [t1].[OrderDate], [t1].[RequiredDate],
[t1].[ShippedDate], [t1].[ShipVia], [t1].[Freight], [t1].[ShipName],
[t1].[ShipAddress], [t1].[ShipCity], [t1].[ShipRegion], [t1].[ShipPostalCode],
[t1].[ShipCountry], (
   SELECT COUNT(*)
```

```
    FROM [dbo].[Orders] AS [t2]
    WHERE [t2].[CustomerID] = [t0].[CustomerID]
    ) AS [count]
FROM [dbo].[Customers] AS [t0]
LEFT OUTER JOIN [dbo].[Orders] AS [t1] ON [t1].[CustomerID] = [t0].[CustomerID]
WHERE ([t0].[Country] = @p0) AND ([t0].[City] = @p1)
ORDER BY [t0].[CustomerID], [t1].[OrderID]
-- @p0: Input String (Size = 2; Prec = 0; Scale = 0) [UK]
-- @p1: Input String (Size = 6; Prec = 0; Scale = 0) [London]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.20706.1

Around the Horn - Thomas Hardy
B's Beverages - Victoria Ashworth
Consolidated Holdings - Elizabeth Brown
Eastern Connection - Ann Devon
North/South - Simon Crowther
Seven Seas Imports - Hari Kumar
```

Как видите, был выполнен единственный запрос SQL для извлечения заказчиков, отвечающих условию конструкции `where`.Несмотря на то что нигде нет обращений к заказам данного заказчика, этот единственный запрос соединяет каждого заказчика с его заказами. Поскольку заказы загружаются до обращения к ним, их загрузка не откладывается и выполняется немедленно. Вместо множества запросов SQL, количество которых равно одному (запрос заказчиков), плюс число извлеченных заказчиков (для заказов каждого заказчика), здесь мы имеем дело с одним SQL-запросом. Если заказчиков достаточно много, разница в производительности будет огромной.

При использовании класса `DataLoadOptions` вы не ограничены немедленной загрузкой объектов единственного ассоциированного класса или единственным иерархическим уровнем ассоциированных классов. Однако немедленная загрузка более одного ассоциированного класса отражается на ее производительности.

Когда немедленная загрузка не такая уж и немедленная

Когда классы не загружаются до того, как к ним будет выполнено обращение, говорят, что их загрузка *отложена*. Если же они загружаются до обращения, то их загрузку называют *немедленной*. Однако иногда немедленная загрузка оказывается не такой уж немедленной, как можно было ожидать.

На примере кода из листинга 14.9 было показано, что за счет передачи ассоциированного класса в качестве аргумента методу `LoadWith<T>` класса `DataLoadOptions` можно заставить загружать заказы вместе с заказчиками. Если вызвать метод `LoadWith<T>` много раз, чтобы немедленно загрузить несколько классов, только один из этих классов будет соединен в запросе с исходным сущностным классом, а другие будут загружаться при ссылке на этот исходный сущностный класс. Когда такое случается, поскольку ассоциированные классы, не соединенные с исходным сущностным классом, все же загружаются до первой ссылки на них, они все равно считаются немедленно загружаемыми, хотя для их получения выполняются отдельные запросы по мере обращения к объектам исходного сущностного класса. Таким образом, хотя их загрузка также считается немедленной, на самом деле она не происходит столь немедленно, как в случае их объединения с исходным классом в едином запросе.

Решение о том, какой ассоциированный класс должен быть соединен в запросе с каким, чтобы загружаться до первого обращения, принимается LINQ to SQL. Это решение оптимизировано на основе общих принципов, применимых к модели классов, т.е. это не имеет отношения к оптимизации, проводимой базой данных. Оно соединяет самую нижнюю ассоциацию в иерархии немедленно загружаемых классов. Все станет понятнее, в разделе, посвященном немедленной загрузке иерархии ассоциированных классов.

Чтобы лучше понять такое поведение, это будет рассматриваться для каждого подхода, где немедленно загружается более одной ассоциации. Эти два подхода следующие: загрузка множества ассоциированных классов одного исходного сущностного класса либо загрузка иерархии ассоциированных классов.

Немедленная загрузка множества ассоциированных классов

Класс `DataLoadOptions` также может использоваться для немедленной загрузки более чем одного класса, ассоциированного с заданным сущностным классом.

Обратите внимание, что в листинге 14.9 сгенерированный запрос SQL не ссылается на ассоциированные с заказчиком демографические данные. Если ссылки на такие данные для извлекаемых заказчиков существовали, то выполнялись бы дополнительные конструкции SQL для каждого заказчика, для которого была бы запрошены демографические данные.

В листинге 14.10 с помощью `DataLoadOptions` будет затребована немедленная загрузка демографических данных заказчика наряду с его заказами.

Листинг 14.10. Немедленная загрузка множества `EntitySet`

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");

DataLoadOptions dlo = new DataLoadOptions();
dlo.LoadWith<Customer>(c => c.Orders);
dlo.LoadWith<Customer>(c => c.CustomerCustomerDemos);
db.LoadOptions = dlo;

IQueryable<Customer> custs = from c in db.Customers
                        where c.Country == "UK" &&
                          c.City == "London"
                        orderby c.CustomerID
                        select c;

// Включить протоколирование.
db.Log = Console.Out;

foreach (Customer cust in custs)
{
  Console.WriteLine("{0} - {1}", cust.CompanyName, cust.ContactName);
}
```

В листинге 14.10 указывается немедленная загрузка заказов и демографических данных. Обратите внимание, что ссылки на них в запросе LINQ отсутствуют. Поэтому любая загрузка этих ассоциированных классов произойдет немедленно, а не будет отложена. В действительности больший интерес вызывают не возвращаемые данные, а выполняемые операторы SQL. Рассмотрим вывод кода из листинга 14.10.

```
SELECT [t0].[CustomerID], [t0].[CompanyName], [t0].[ContactName],
[t0].[ContactTitle], [t0].[Address], [t0].[City], [t0].[Region], [t0].[PostalCode],
[t0].[Country], [t0].[Phone], [t0].[Fax], [t1].[CustomerID] AS [CustomerID2],
[t1].[CustomerTypeID], (
  SELECT COUNT(*)
  FROM [dbo].[CustomerCustomerDemo] AS [t2]
  WHERE [t2].[CustomerID] = [t0].[CustomerID]
  ) AS [count]
FROM [dbo].[Customers] AS [t0]
LEFT OUTER JOIN [dbo].[CustomerCustomerDemo] AS [t1] ON [t1].[CustomerID] =
[t0].[CustomerID]
WHERE ([t0].[Country] = @p0) AND ([t0].[City] = @p1)
ORDER BY [t0].[CustomerID], [t1].[CustomerTypeID]
```

```
-- @p0: Input String (Size = 2; Prec = 0; Scale = 0) [UK]
-- @p1: Input String (Size = 6; Prec = 0; Scale = 0) [London]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.20706.1
SELECT [t0].[OrderID], [t0].[CustomerID], [t0].[EmployeeID], [t0].[OrderDate],
[t0].[RequiredDate], [t0].[ShippedDate], [t0].[ShipVia], [t0].[Freight],
[t0].[ShipName], [t0].[ShipAddress], [t0].[ShipCity], [t0].[ShipRegion],
[t0].[ShipPostalCode], [t0].[ShipCountry]
FROM [dbo].[Orders] AS [t0]
WHERE [t0].[CustomerID] = @x1

-- @x1: Input StringFixedLength (Size = 5; Prec = 0; Scale = 0) [AROUT]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.20706.1
Around the Horn - Thomas Hardy
SELECT [t0].[OrderID], [t0].[CustomerID], [t0].[EmployeeID], [t0].[OrderDate],
[t0].[RequiredDate], [t0].[ShippedDate], [t0].[ShipVia], [t0].[Freight],
[t0].[ShipName], [t0].[ShipAddress], [t0].[ShipCity], [t0].[ShipRegion],
[t0].[ShipPostalCode], [t0].[ShipCountry]
FROM [dbo].[Orders] AS [t0]
WHERE [t0].[CustomerID] = @x1

-- @x1: Input StringFixedLength (Size = 5; Prec = 0; Scale = 0) [BSBEV]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.20706.1
B's Beverages - Victoria Ashworth
...
```

Как видите, демографические данные заказчика соединены с заказчиками во время запроса, но для загрузки заказов каждого заказчика генерируется отдельный запрос SQL. Этот отдельный запрос заказов выполняется тогда, когда производится обращение к каждому заказчику, т.е. в операторе `foreach`. Обратите внимание, что в выводе запрос заказов заказчика находится перед информацией о заказчике, отображаемой на консоли.

Поскольку в коде нет ссылок ни на демографические данные, ни на заказы, кроме как при вызове метода `LoadWith<T>`, загрузка не откладывается, а потому является немедленной.

Немедленная загрузка иерархии ассоциированных классов

В предыдущем разделе было показано, как заставить множество ассоциированных сущностных классов загружаться немедленно. Здесь рассматривается, как обеспечить немедленную загрузку иерархии ассоциированных сущностных классов. Чтобы продемонстрировать это, в листинге 14.11 запрашивается немедленная загрузка не только заказов, но также детальной информации по этим заказам.

Листинг 14.11. Немедленная загрузка иерархии сущностных классов

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");

DataLoadOptions dlo = new DataLoadOptions();
dlo.LoadWith<Customer>(c => c.Orders);
dlo.LoadWith<Order>(o => o.OrderDetails);
db.LoadOptions = dlo;

IQueryable<Customer> custs = from c in db.Customers
                        where c.Country == "UK" &&
                          c.City == "London"
                        orderby c.CustomerID
                        select c;

// Включить протоколирование.
db.Log = Console.Out;
```

```
foreach (Customer cust in custs)
{
  Console.WriteLine("{0} - {1}", cust.CompanyName, cust.ContactName);
  foreach (Order order in cust.Orders)
  {
    Console.WriteLine(" {0} {1}", order.OrderID, order.OrderDate);
  }
}
```

Обратите внимание, что в листинге 14.11 немедленно загружаются заказы заказчика, а для каждого заказа — его детали. Вывод кода из листинга 14.11 показан ниже.

```
SELECT [t0].[CustomerID], [t0].[CompanyName], [t0].[ContactName],
[t0].[ContactTitle], [t0].[Address], [t0].[City], [t0].[Region], [t0].[PostalCode],
[t0].[Country], [t0].[Phone], [t0].[Fax]
FROM [dbo].[Customers] AS [t0]
WHERE ([t0].[Country] = @p0) AND ([t0].[City] = @p1)
ORDER BY [t0].[CustomerID]
-- @p0: Input String (Size = 2; Prec = 0; Scale = 0) [UK]
-- @p1: Input String (Size = 6; Prec = 0; Scale = 0) [London]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.20706.1

SELECT [t0].[OrderID], [t0].[CustomerID], [t0].[EmployeeID], [t0].[OrderDate],
[t0].[RequiredDate], [t0].[ShippedDate], [t0].[ShipVia], [t0].[Freight],
[t0].[ShipName], [t0].[ShipAddress], [t0].[ShipCity], [t0].[ShipRegion],
[t0].[ShipPostalCode], [t0].[ShipCountry], [t1].[OrderID] AS [OrderID2],
[t1].[ProductID], [t1].[UnitPrice], [t1].[Quantity], [t1].[Discount], (
   SELECT COUNT(*)
   FROM [dbo].[Order Details] AS [t2]
   WHERE [t2].[OrderID] = [t0].[OrderID]
   ) AS [count]
FROM [dbo].[Orders] AS [t0]
LEFT OUTER JOIN [dbo].[Order Details] AS [t1] ON [t1].[OrderID] = [t0].[OrderID]
WHERE [t0].[CustomerID] = @x1
ORDER BY [t0].[OrderID], [t1].[ProductID]
-- @x1: Input StringFixedLength (Size = 5; Prec = 0; Scale = 0) [AROUT]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.20706.1

Around the Horn - Thomas Hardy
SELECT [t0].[OrderID], [t0].[CustomerID], [t0].[EmployeeID], [t0].[OrderDate],
[t0].[RequiredDate], [t0].[ShippedDate], [t0].[ShipVia], [t0].[Freight],
[t0].[ShipName], [t0].[ShipAddress], [t0].[ShipCity], [t0].[ShipRegion],
[t0].[ShipPostalCode], [t0].[ShipCountry], [t1].[OrderID] AS [OrderID2],
[t1].[ProductID], [t1].[UnitPrice], [t1].[Quantity], [t1].[Discount], (
   SELECT COUNT(*)
   FROM [dbo].[Order Details] AS [t2]
   WHERE [t2].[OrderID] = [t0].[OrderID]
   ) AS [count]
FROM [dbo].[Orders] AS [t0]
LEFT OUTER JOIN [dbo].[Order Details] AS [t1] ON [t1].[OrderID] = [t0].[OrderID]
WHERE [t0].[CustomerID] = @x1
ORDER BY [t0].[OrderID], [t1].[ProductID]
-- @x1: Input StringFixedLength (Size = 5; Prec = 0; Scale = 0) [BSBEV]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.20706.1

B's Beverages - Victoria Ashworth
...
```

Опять-таки, здесь больше интересуют не извлеченные данные, а выполняющиеся запросы SQL. Обратите внимание, что на этот раз запрос заказчиков не соединяется

ни с заказами, ни с их деталями. Вместо этого при ссылке на каждого заказчика выполняется дополнительный запрос SQL, который соединяет заказы и их детали. Поскольку ни на то, ни на другое нет ссылок в коде, они загружаются до ссылок и по-прежнему рассматриваются как загружаемые немедленно.

На этом примере можно видеть, что LINQ to SQL выполняет единственное соединение ассоциации нижнего уровня иерархии немедленно загружаемых классов, как упоминалось ранее.

Фильтрация и упорядочивание

При обсуждении класса `DataLoadOptions` необходимо, чтобы вы помнили о методе `AssociateWith`, который может использоваться как для фильтрации ассоциированных дочерних объектов, так и для их упорядочивания.

В листинге 14.8 извлекалось несколько заказчиков, которые затем перечислялись с отображением информации о них и их заказах. Вы могли видеть, что в результате даты заказов упорядочены по возрастанию. Чтобы продемонстрировать, как метод `AssociateWith` применяться и для фильтрации, и для упорядочивания ассоциированных классов, в листинге 14.12 сделано то и другое.

Листинг 14.12. Использование класса `DataLoadOptions` для фильтрации и упорядочивания

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");

DataLoadOptions dlo = new DataLoadOptions();
dlo.AssociateWith<Customer>(c => from o in c.Orders
                            where o.OrderID < 10700
                            orderby o.OrderDate descending
                            select o);
db.LoadOptions = dlo;

IQueryable<Customer> custs = from c in db.Customers
                        where c.Country == "UK" &&
                          c.City == "London"
                        orderby c.CustomerID
                        select c;

foreach (Customer cust in custs)
{
  Console.WriteLine("{0} - {1}", cust.CompanyName, cust.ContactName);
  foreach (Order order in cust.Orders)
  {
    Console.WriteLine(" {0} {1}", order.OrderID, order.OrderDate);
  }
}
```

Обратите внимание, что в листинге 14.12 в запрос встроено лямбда-выражение, переданное методу `AssociateWith`. В этом запросе фильтруются все записи, в которых `OrderID` меньше `10700`, и затем они сортируются в порядке убывания `OrderDate`. Рассмотрим результат запуска листинга 14.12.

```
Around the Horn - Thomas Hardy
  10558 6/4/1997 12:00:00 AM
  10453 2/21/1997 12:00:00 AM
  10383 12/16/1996 12:00:00 AM
  10355 11/15/1996 12:00:00 AM
B's Beverages - Victoria Ashworth
  10599 7/15/1997 12:00:00 AM
```

```
  10578 6/24/1997 12:00:00 AM
  10539 5/16/1997 12:00:00 AM
  10538 5/15/1997 12:00:00 AM
  10484 3/24/1997 12:00:00 AM
  10471 3/11/1997 12:00:00 AM
  10289 8/26/1996 12:00:00 AM
Consolidated Holdings - Elizabeth Brown
  10462 3/3/1997 12:00:00 AM
  10435 2/4/1997 12:00:00 AM
Eastern Connection - Ann Devon
  10532 5/9/1997 12:00:00 AM
  10400 1/1/1997 12:00:00 AM
  10364 11/26/1996 12:00:00 AM
North/South - Simon Crowther
  10517 4/24/1997 12:00:00 AM
Seven Seas Imports - Hari Kumar
  10547 5/23/1997 12:00:00 AM
  10523 5/1/1997 12:00:00 AM
  10472 3/12/1997 12:00:00 AM
  10388 12/19/1996 12:00:00 AM
  10377 12/9/1996 12:00:00 AM
  10359 11/21/1996 12:00:00 AM
```

Как видите, возвращены только заказы с `OrderID` меньше `10700`, причем возвращены они в порядке убывания дат.

Случайные соединения

Одним из преимуществ ассоциаций является то, что они, по сути, выполняют соединения автоматически. Когда запрашиваются заказчики из базы данных Northwind, каждый заказчик имеет коллекцию заказов, доступных через свойство `Orders` объекта `Customer`. Таким образом, извлечение заказов для заказчиков осуществляется автоматически. Для получения такого поведения обычно пришлось бы выполнять соединение. Обратное также верно. Когда извлекаются заказы, класс `Order` имеет свойство `Customer`, ссылающееся на соответствующего заказчика.

Несмотря на то что происходит автоматическое соединение, на самом деле это просто счастливая случайность. Соединение случается потому, что имеется объект, скажем, дочерний, имеющий отношение с другим объектом, например, родительским, и мы *ожидаем* от него возможности доступа через ссылку на исходный дочерний объект.

Например, при работе с XML, когда есть ссылка на узел, мы ожидаем получения ссылки на его родительской узел через переменную-член дочернего узла, ссылающуюся на родителя. Мы не готовы к тому, что придется выполнять запрос по всей структуре XML, передавая дочерний узел в качестве ключа поиска. К тому же, когда имеется ссылка на узел, мы ожидаем возможности доступа к его дочерним узлам со ссылками на него самого.

Поэтому, хотя автоматическое соединение, несомненно, удобно, реализация может по-своему обращаться с природой отношений, и наши ожидания относительно того, как они *должны* себя вести, определяют стремление к автоматическому установлению соединений. Поэтому соединения являются в определенной степени случайными.

Соединения

Выше обсуждался тот факт, что многие отношения в базе данных определены как ассоциации, и что можно получить доступ к ассоциированным объектам, просто обращаясь к членам класса. Однако отображаться подобным образом могут только те отношения, которые определены с использованием внешних ключей. Поскольку не каждый

тип отношения определен с применением внешних ключей, иногда приходится соединять таблицы явно.

Внутренние соединения

Внутреннее соединение по эквивалентности можно выполнить с помощью операции `join`. Как это принято при внутреннем соединении, любые записи во внешнем результирующем наборе исключаются, если не существуют связанные с ними записи во внутреннем результирующем наборе. В листинге 14.13 приведен пример.

Листинг 14.13. Выполнение внутреннего соединения

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");

var entities = from s in db.Suppliers
            join c in db.Customers on s.City equals c.City
            select new
            {
              SupplierName = s.CompanyName,
              CustomerName = c.CompanyName,
              City = c.City
            };

foreach (var e in entities)
{
  Console.WriteLine("{0}: {1} - {2}", e.City, e.SupplierName, e.CustomerName);
}
```

В листинге 14.13 выполняется внутреннее соединение поставщиков и потребителей (заказчиков). Если запись о заказчике из того же города, что и у поставщика, не существует, то запись о поставщике будет исключена из результирующего набора. Ниже показан результат работы кода из листинга 14.13.

```
London: Exotic Liquids - Around the Horn
London: Exotic Liquids - B's Beverages
London: Exotic Liquids - Consolidated Holdings
London: Exotic Liquids - Eastern Connection
London: Exotic Liquids - North/South
London: Exotic Liquids - Seven Seas Imports
Sao Paulo: Refrescos Americanas LTDA - Comйrcio Mineiro
Sao Paulo: Refrescos Americanas LTDA - Familia Arquibaldo
Sao Paulo: Refrescos Americanas LTDA - Queen Cozinha
Sao Paulo: Refrescos Americanas LTDA - Tradiзao Hipermercados
Berlin: Heli SЬЯwaren GmbH & Co. KG - Alfred Futterkiste
Paris: Aux joyeux ecclйsiastiques - Paris spйcialitйs
Paris: Aux joyeux ecclйsiastiques - Spйcialitйs du monde
Montréal: Ma Maison - Mère Paillarde
```

Как видите, несмотря на тот факт, что некоторым поставщикам в выводе соответствует по нескольку заказчиков, ряд поставщиков вообще в списке отсутствует. Причина в том, что не найдено поставщиков в тех городах, откуда эти пропущенные заказчики. Если нужно видеть поставщиков независимо от того, есть ли в том же городе заказчики или нет, то понадобится внешнее соединение.

Внешние соединения

В главе 4 при рассмотрении стандартной операции запроса `DefaultIfEmpty` упоминалось, что он может быть использован для выполнения соединений. В листинге 14.14 конструкцию `into` применяется для направления соответствующих результатов `join` во временную последовательность, на которой затем вызываются операции

DefaultIfEmpty. Таким образом, если запись опущена в результате соединения, то будет подставлено значение по умолчанию. Чтобы можно было увидеть сгенерированный оператор SQL, используется средство протоколирования DataContext.

Листинг 14.14. Выполнение внешнего соединения

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");

db.Log = Console.Out;

var entities =
  from s in db.Suppliers
  join c in db.Customers on s.City equals c.City into temp
  from t in temp.DefaultIfEmpty()
  select new
  {
    SupplierName = s.CompanyName,
    CustomerName = t.CompanyName,
    City = s.City
  };

foreach (var e in entities)
{
  Console.WriteLine("{0}: {1} - {2}", e.City, e.SupplierName, e.CustomerName);
}
```

Обратите внимание, что в конструкции join в листинге 14.14 результаты соединения направлены во временную последовательность по имени temp. Имя временной последовательности может быть любым до тех пор, пока оно не конфликтует ни с одним другим именем или ключевым словом. Затем выполняется последующий запрос результатов на последовательности temp, передавая ее операции DefaultEmpty. Хотя она еще не рассматривалась, но операция DefaultEmpty, вызванная в листинге 14.14 — это не та же самая операция, о которой шла речь в главе 4. Как будет вскоре объясняться, запросы LINQ to SQL транслируются в операторы SQL, и эти операторы SQL выполняются базой данных. SQL Server не имеет никакой возможности вызвать стандартную операцию запроса DefaultEmpty. Вместо этого вызов операции транслируется в соответствующий оператор SQL. Вот почему мне понадобилось включить средство протоколирования DataContext.

К тому же доступ к названию города производится через таблицу Suppliers, а не коллекцию temp. Это сделано потому, что известно, что всегда должна существовать запись о поставщике, но для поставщиков, не имеющих соответствующих заказчиков, не будет значения города в присоединенном результате из коллекции temp. Это отличается от предыдущего примера внутреннего соединения, где был получен город из присоединенной таблицы. В том примере не имело значения, из какой таблицы брать город, потому что если соответствующий заказчик не был найдено, то не было бы и соответствующей результирующей записи, поскольку соединение было внутренним.

Взглянем на результат работы листинга 14.14:

```
SELECT [t0].[CompanyName], [t1].[CompanyName] AS [value], [t0].[City]
FROM [dbo].[Suppliers] AS [t0]
LEFT OUTER JOIN [dbo].[Customers] AS [t1] ON [t0].[City] = [t1].[City]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.20706.1

London: Exotic Liquids - Around the Horn
London: Exotic Liquids - B's Beverages
London: Exotic Liquids - Consolidated Holdings
London: Exotic Liquids - Eastern Connection
London: Exotic Liquids - North/South
```

```
London: Exotic Liquids - Seven Seas Imports
New Orleans: New Orleans Cajun Delights -
Ann Arbor: Grandma Kelly's Homestead -
Tokyo: Tokyo Traders -
Oviedo: Cooperativa de Quesos 'Las Cabras' -
Osaka: Mayumi's -
Melbourne: Pavlova, Ltd. -
Manchester: Specialty Biscuits, Ltd. -
Gцteborg: PB Knдckebrцd AB -
Sao Paulo: Refrescos Americanas LTDA - Comйrcio Mineiro
Sao Paulo: Refrescos Americanas LTDA - Familia Arquibaldo
Sao Paulo: Refrescos Americanas LTDA - Queen Cozinha
Sao Paulo: Refrescos Americanas LTDA - Tradiзao Hipermercados
Berlin: Heli SЬЯwaren GmbH & Co. KG - Alfreds Futterkiste
Frankfurt: Plutzer LebensmittelgroЯmдrkte AG -
Cuxhaven: Nord-Ost-Fisch Handelsgesellschaft mbH -
Ravenna: Formaggi Fortini s.r.l. -
Sandvika: Norske Meierier -
Bend: Bigfoot Breweries -
Stockholm: Svensk Sjцfцda AB -
Paris: Aux joyeux ecclйsiastiques - Paris sprйcialitйs
Paris: Aux joyeux ecclйsiastiques - Sprйcialitйs du monde
Boston: New England Seafood Cannery -
Singapore: Leka Trading -
Lyngby: Lyngbysild -
Zaandam: Zaanse Snoepfabriek -
Lappeenranta: Karkki Oy -
Sydney: G'day, Mate -
Montréal: Ma Maison - Mère Paillarde
Salerno: Pasta Buttini s.r.l. -
Montceau: Escargots Nouveaux -
Annecy: Gai pâturage -
Ste-Hyacinthe: Forêts d'érables -
```

В выводе кода из листинга 14.14 видно, что получается как минимум одна запись для каждого поставщика, причем некоторые поставщики не имеют соответствующего заказчика, поскольку выполнено внешнее соединение. Но если возникнут какие-то сомнения, можете взглянуть на сгенерированный оператор SQL, который ясно указывает на выполнение внешнего соединения.

Упрощать или не упрощать

В примерах, которые представлены в листингах 14.13 и 14.14, результаты запроса проектируются на плоскую структуру. Под этим понимается объект анонимного класса, в котором каждое запрошенное поле становится членом этого класса. Вместо создания единственного анонимного класса, содержащего все нужные поля, можно было бы создать анонимный класс, состоящий из объекта `Supplier` и соответствующего ему объекта `Customer`. В этом случае существовал бы верхний уровень анонимного класса и нижний уровень, состоящий из объекта `Supplier` и либо соответствующего ему объекта `Customer`, либо объекта по умолчанию, предоставленного операцией `DefaultIfEmpty`, который был бы равен `null`.

Если принять “плоский” подход, как делалось в предыдущих примерах из-за того, что спроектированный выходной класс не был сущностным классом, то не будет возможности выполнять обновления выходных объектов через объект `DataContext`, который управляет хранением изменений в базе данных. Это хорошо для данных, которые не подвергаются изменениям. Однако иногда может понадобиться возможность изменения извлеченных объектов с сохранением изменений через `DataContext`. Более под-

робно эта тема раскрывается в главе 16. А пока давайте взглянем на листинг 14.15, который содержит “не плоский” пример.

Листинг 14.15. Возврат не плоского результата, допускающего сохранение изменений через `DataContext`

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");

var entities = from s in db.Suppliers
            join c in db.Customers on s.City equals c.City into temp
            from t in temp.DefaultIfEmpty()
            select new { s, t };

foreach (var e in entities)
{
 Console.WriteLine("{0}: {1} - {2}", e.s.City,
   e.s.CompanyName,
   e.t != null ? e.t.CompanyName : "");
}
```

В листинге 14.15 вместо возврата результатов запроса в плоский анонимный объект, состоящий из всех нужных полей, результаты запроса возвращаются в анонимном объекте, состоящем из объекта `Supplier` и, потенциально, из объекта `Customer`. Также обратите внимание, что в вызове метода `Console.WriteLine` учитывается, что временный результат может быть `null`, если нет соответствующего объекта `Customer`. Давайте посмотрим на результаты выполнения кода из листинга 14.15:

```
London: Exotic Liquids - Around the Horn
London: Exotic Liquids - B's Beverages
London: Exotic Liquids - Consolidated Holdings
London: Exotic Liquids - Eastern Connection
London: Exotic Liquids - North/South
London: Exotic Liquids - Seven Seas Imports
New Orleans: New Orleans Cajun Delights -
Ann Arbor: Grandma Kelly's Homestead -
Tokyo: Tokyo Traders -
Oviedo: Cooperativa de Quesos 'Las Cabras' -
Osaka: Mayumi's -
Melbourne: Pavlova, Ltd. -
Manchester: Specialty Biscuits, Ltd. -
Göteborg: PB Knäckebröd AB -
Sao Paulo: Refrescos Americanas LTDA - Comércio Mineiro
Sao Paulo: Refrescos Americanas LTDA - Familia Arquibaldo
Sao Paulo: Refrescos Americanas LTDA - Queen Cozinha
Sao Paulo: Refrescos Americanas LTDA - Tradiçao Hipermercados
Berlin: Heli Süßwaren GmbH & Co. KG - Alfreds Futterkiste
Frankfurt: Plutzer Lebensmittelgroßmärkte AG -
Cuxhaven: Nord-Ost-Fisch Handelsgesellschaft mbH -
Ravenna: Formaggi Fortini s.r.l. -
Sandvika: Norske Meierier -
Bend: Bigfoot Breweries -
Stockholm: Svensk Sjöföda AB -
Paris: Aux joyeux ecclésiastiques - Paris spécialités
Paris: Aux joyeux ecclésiastiques - Spécialités du monde
Boston: New England Seafood Cannery -
Singapore: Leka Trading -
Lyngby: Lyngbysild -
Zaandam: Zaanse Snoepfabriek -
Lappeenranta: Karkki Oy -
```

```
Sydney: G'day, Mate -
Montréal: Ma Maison - Mère Paillarde
Salerno: Pasta Buttini s.r.l. -
Montceau: Escargots Nouveaux -
Annecy: Gai pâturage -
Ste-Hyacinthe: Forêts d'érables -
```

В этом выводе видно, что некоторые поставщики не имеют соответствующих им заказчиков в своих городах. В отличие от последовательности анонимных объектов, возвращенных запросом в листинге 14.14, анонимные объекты, возвращенные запросом из листинга 14.15, содержат сущностные объекты типа `Supplier` и `Customer`. Поскольку это сущностные объекты, можно воспользоваться преимуществом служб, предоставленных `DataContext`, для управления изменениями в них и сохранения их в базе данных.

Отложенное выполнение запроса

Отложенное выполнение запросов представляет собой настолько важную тему, что ее стоит повторять на тот случай, если в процессе чтения вы пропустили какую-то часть книги. Отложенное выполнение запросов означает, что запрос LINQ любого типа — LINQ to SQL, LINQ to XML или LINQ to Object — может и не выполняться в точке его определения. Рассмотрим, к примеру, следующий запрос:

```
IQueryable<Customer> custs = from c in db.Customers
                             where c.Country == "UK"
                             select c;
```

Запрос базы данных на самом деле не выполняется при выполнении этого оператора; он просто определяется и присваивается переменной `custs`. Запрос не будет выполнен до тех пор, пока не начнется перечисление последовательности `custs`.

Последствия отложенного выполнения запросов

Одним из последствий отложенного выполнения запросов является то, что запрос может содержать ошибки, которые вызовут исключения, но только во время действительного выполнения запросов, а не во время его определения. Это может здорово запутать, когда вы проходите отладчиком через оператор запроса, и все идет хорошо, а затем, намного дальше в коде, при перечислении последовательности запроса генерируется исключение. Или, возможно, на последовательности запроса вызывается другая операция, что дает в результате новую последовательность, которая немедленно перечисляется.

Другим последствием является то, что поскольку запрос SQL выполняется при перечислении последовательности запроса, многократное ее перечисление ведет к многократному выполнению запроса SQL. Это определенно может плохо отразиться на производительности. Чтобы предотвратить это, нужно вызвать на последовательности одну из стандартных операций запросов преобразования — `ToArray<T>`, `ToList<T>`, `ToDictionary<T, K>` или `ToLookup<T, K>`. Каждая из этих операций преобразует последовательность, на которой она вызвана, в структуру данных указанного типа, что в результате приведет к кэшированию результата. Перечисление этой новой структуры данных можно выполнять многократно, не вызывая повторного выполнения SQL-запроса с потенциальной вероятностью изменения результатов.

Использование преимуществ отложенного выполнения запросов

Одним из преимуществ отложенного выполнения запросов является повышение производительности и в то же время возможность повторного использования ранее определенных запросов. Поскольку запрос выполняется каждый раз, когда производится перечисление возвращаемой им последовательности, его можно определить один раз,

а перечислять снова и снова, как только того потребует ситуация. И если поток кода идет по некоторому пути, который не требует просмотра результатов запроса посредством его перечисления, то производительность увеличивается за счет того, что запрос не выполняется.

Другая выгода от отложенного выполнения запросов заключается в том, что поскольку запрос на самом деле не выполняется при его определении, можно добавлять дополнительные операции программно по мере необходимости. Представьте приложение, которое позволяет пользователю запрашивать информацию о заказчиках. Также представьте, что пользователь может фильтровать запрошенный список заказчиков. Нарисуйте в воображении один из возможных вариантов интерфейса с фильтрацией, который имеет раскрывающийся список для каждого столбца таблицы заказчиков. Допустим, есть раскрывающийся список для столбца `City` и другой — для столбца `Country`. Каждый раскрывающийся список содержит все города и все страны из всех записей `Customer` в базе данных. В вершине каждого списка есть опция `[ALL]`, принятая по умолчанию для соответствующего ему столбца данных. Если пользователь не изменяет состояние любого из этих раскрывающихся списков, никакой дополнительной конструкции `where` к запросу для соответствующего столбца не добавляется. В листинге 14.16 приведен пример программного построения запроса для такого интерфейса.

Листинг 14.16. Программное построение запроса

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");

// Включить протоколирование.
db.Log = Console.Out;

// Представьте, что значения, приведенные ниже, не закодированы жестко, а вместо
// этого получаются из выбранных значений некоторых раскрывающихся списков.
string dropdownListCityValue = "Cowes";
string dropdownListCountryValue = "UK";

IQueryable<Customer> custs = (from c in db.Customers
                              select c);

if (!dropdownListCityValue.Equals("[ALL]"))
{
  custs = from c in custs
          where c.City == dropdownListCityValue
          select c;
}

if (!dropdownListCountryValue.Equals("[ALL]"))
{
  custs = from c in custs
          where c.Country == dropdownListCountryValue
          select c;
}

foreach (Customer cust in custs)
{
  Console.WriteLine("{0} - {1} - {2}", cust.CompanyName, cust.City, cust.Country);
}
```

В листинге 14.16 эмулируется получение выбранного пользователем города и страны из раскрывающихся списков. Если они не установлены в `"[ALL]"`, к запросу добавляется дополнительная операция `where`. Поскольку запрос на самом деле не выполняется до того, как начнется перечисление возвращенной им последовательности, его можно строить по частям.

Взглянем на результат работы кода из листинга 14.16:

```
SELECT [t0].[CustomerID], [t0].[CompanyName], [t0].[ContactName],
[t0].[ContactTitle], [t0].[Address], [t0].[City], [t0].[Region], [t0].[PostalCode],
[t0].[Country], [t0].[Phone], [t0].[Fax]
FROM [dbo].[Customers] AS [t0]
WHERE ([t0].[Country] = @p0) AND ([t0].[City] = @p1)
-- @p0: Input String (Size = 2; Prec = 0; Scale = 0) [UK]
-- @p1: Input String (Size = 5; Prec = 0; Scale = 0) [Cowes]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.20706.1

Island Trading - Cowes — UK
```

Обратите внимание, что поскольку указано, что выбран город `Cowes` и страна `UK`, то и получены записи о заказчиках из Cowes, United Kingdom. Также обратите внимание, что здесь выполняется единственный оператор SQL. Поскольку выполнение запроса отложено до момента, когда в нем возникнет необходимость, можно продолжить добавлять условия к запросу, чтобы дополнительно ограничить его или упорядочить результат, без каких-либо затрат на множественные запросы SQL.

Для другого теста — в листинге 14.17 — значение переменной `dropdownListCityValue` изменяется на `"[ALL]"`, чтобы посмотреть, как будет выглядеть выполняющийся оператор запроса SQL и каковы будут полученные результаты. Поскольку в качестве города по умолчанию указан `"[ALL]"`, запрос SQL не должен ограничивать результирующий набор никаким городом.

Листинг 14.17. Программное построение другого запроса

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");

// Включить протоколирование.
db.Log = Console.Out;

// Представьте, что приведенные ниже значения не закодированы жестко,
// а вместо этого получаются из выбранных значений раскрывающихся списков.
string dropdownListCityValue = "[ALL]";
string dropdownListCountryValue = "UK";
IQueryable<Customer> custs = (from c in db.Customers
                              select c);
if (!dropdownListCityValue.Equals("[ALL]"))
{
  custs = from c in custs
          where c.City == dropdownListCityValue
          select c;
}
if (!dropdownListCountryValue.Equals("[ALL]"))
{
  custs = from c in custs
          where c.Country == dropdownListCountryValue
          select c;
}
foreach (Customer cust in custs)
{
  Console.WriteLine("{0} - {1} - {2}", cust.CompanyName, cust.City, cust.Country);
}
```

Ниже показан вывод кода из листинга 14.17:

```
SELECT [t0].[CustomerID], [t0].[CompanyName], [t0].[ContactName],
[t0].[ContactTitle], [t0].[Address], [t0].[City], [t0].[Region], [t0].[PostalCode],
[t0].[Country], [t0].[Phone], [t0].[Fax]
```

```
FROM [dbo].[Customers] AS [t0]
WHERE [t0].[Country] = @p0
-- @p0: Input String (Size = 2; Prec = 0; Scale = 0) [UK]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.20706.1

Around the Horn - London - UK
B's Beverages - London - UK
Consolidated Holdings - London - UK
Eastern Connection - London - UK
Island Trading - Cowes - UK
North/South - London - UK
Seven Seas Imports - London - UK
```

Здесь видно, что в части `where` оператора запроса SQL больше не указан город, что и требовалось. Также в выводе результата теперь присутствуют заказчики из разных городов Соединенного Королевства.

Конечно, всегда можно добавить вызов стандартных операций запросов `ToArray<T>`, `ToList<T>`, `ToDictionary<T, K>` или `ToLookup<T, K>`, чтобы принудительно выполнить запрос тогда, когда это необходимо.

Получение SQL-конструкции `IN` с помощью операции `Contains`

Одной из возможностей операций SQL, которых недоставало ранним воплощениям LINQ to SQL, была способность выполнять SQL-конструкцию `IN` — вроде той, что представлена в следующем SQL-запросе:

Запрос SQL с конструкцией `IN`

```
SELECT *
FROM Customers
WHERE (City IN ('London', 'Madrid'))
```

Для решения этой проблемы была добавлена операция `Contains`. Но она работает в направлении, противоположном тому, чего можно было ожидать от реализации SQL-конструкции `IN`. Эта конструкция позволяет указать, что некоторый член сущностного класса должен быть *в* (`IN`) некотором наборе значений. Вместо этого она работает в обратной манере. Взглянем на листинг 14.18, в котором демонстрируется применение операции `Contains`.

Листинг 14.18. Применение операции `Contains`

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
db.Log = Console.Out;
string[] cities = { "London", "Madrid" };
IQueryable<Customer> custs = db.Customers.Where(c => cities.Contains(c.City));

foreach (Customer cust in custs)
{
  Console.WriteLine("{0} - {1}", cust.CustomerID, cust.City);
}
```

Как видно в листинге 14.18, вместо написания запроса так, что город заказчика должен присутствовать в некотором наборе значений, запрос записывается так, что некоторый набор значений содержит город заказчика. В случае листинга 14.18 создан массив городов по имени `cities`. Затем в запросе вызывается операция `Contains` на массиве `cities` и ей передается город заказчика. Если массив `cities` содержит город заказчика, будет возвращено значение `true` в операции `Where`, что вызовет включение объекта `Customer` в выходную последовательность.

Ниже показан вывод кода из листинга 14.18.

```
SELECT [t0].[CustomerID], [t0].[CompanyName], [t0].[ContactName],
[t0].[ContactTitle], [t0].[Address], [t0].[City], [t0].[Region], [t0].
[PostalCode],
[t0].[Country], [t0].[Phone], [t0].[Fax]
FROM [dbo].[Customers] AS [t0]
WHERE [t0].[City] IN (@p0, @p1)
-- @p0: Input String (Size = 6; Prec = 0; Scale = 0) [London]
-- @p1: Input String (Size = 6; Prec = 0; Scale = 0) [Madrid]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.20706.1

AROUT - London
BOLID - Madrid
BSBEV - London
CONSH - London
EASTC - London
FISSA - Madrid
NORTS - London
ROMEY - Madrid
SEVES — London
```

Взглянув на сгенерированный оператор SQL, легко заметить, что операция `Contains` была транслирована в SQL-конструкцию `IN`.

Обновления

Проведение обновлений в базе данных посредством LINQ to SQL столь же просто, как изменение свойств объекта, вызов метода `SubmitChanges` объекта `DataContext` и последующая обработка возможных конфликтов параллельного доступа. Пусть вопрос обработки конфликтов параллельного доступа вас не пугает — существует несколько вариантов обработки подобных конфликтов, и ни один из них не является слишком болезненным. Теме обнаружения и обработки конфликтов параллелизма посвящена глава 17.

Конечно, все это так просто только в случае правильно написанных сущностных классов, корректно отображаемых на базу данных, и согласованности графа. За дополнительной информацией об отображении сущностных классов на базу данных обращайтесь в раздел “Атрибуты сущностных классов и свойства атрибутов” главы 15. Сведения о согласованности графа можно найти в разделе “Согласованность графа” той же главы. Однако SQLMetal и Object Relational Designer выполняют всю необходимую работу.

Простой пример обновления в базе данных приведен в листинге 12.1.

Обновление ассоциированных классов

В соответствии с проектным решением LINQ to SQL позволяет обновлять любую сторону ассоциированных классов, чтобы удалить отношение между ними. Можно обновить ссылку родительского объекта на один из его дочерних объектов либо обновить ссылку дочернего объекта на родителя. Очевидно, что ссылки на каждом конце отношения должны обновляться, но при этом *вам* нужно обновить только одну или другую сторону.

О поддержке согласованности объектной модели при обновлении одной из сторон заботится не LINQ to SQL; за это отвечает сущностный класс. Дополнительные сведения о том, как это должно быть реализовано, можно найти в разделе “Согласованность графа” главы 15.

Однако если поручить создание сущностных классов утилите SQLMetal и средству Object Relational Designer, то они это должным образом обработают.

Обновление ссылки на родителя у дочернего объекта

Поскольку можно обновить любую из сторон отношения, выберем вариант с обновлением ссылки на родителя дочернего объекта. В качестве примера давайте посмотрим в листинге 14.19, как можно было бы изменить сотрудника, которому поручается заказ, в базе данных Northwind. Этот пример более сложен, чем многие другие, поэтому он снабжается дополнительными пояснениями.

Листинг 14.19. Изменение отношения присвоением нового родителя

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
Order order = (from o in db.Orders
               where o.EmployeeID == 5
               orderby o.OrderDate descending
               select o).First<Order>();
// Сохранить текущего сотрудника, чтобы восстановить его в конце.
Employee origEmployee = order.Employee;
```

В приведенном коде после получения `DataContext` запрашивается самый последний заказ сотрудника с `EmployeeID`, равным `5`, за счет упорядочивания заказов этого лица по убыванию дат и вызова операции `First`. Это даст самый свежий заказ. Сохранение ссылки на исходного сотрудника, которому был поручен этот заказ, позволит восстановить его в конце примера. Эта ссылка сохраняется в переменной по имени `origEmployee`.

```
Console.WriteLine("Перед изменением сотрудника.");
Console.WriteLine("OrderID = {0} : OrderDate = {1} : EmployeeID = {2}",
  order.OrderID, order.OrderDate, order.Employee.EmployeeID);
```

Затем на консоль выводится сообщение, уведомляющее, что сотрудник для извлеченного заказа еще не изменен, за которым следует идентификатор заказа и сотрудника. Должно быть видно, что заказ принят сотрудником с идентификатором `5`, поскольку именно его заказ был запрошен.

```
Employee emp = (from e in db.Employees
                where e.EmployeeID == 9
                select e).Single<Employee>();
// Назначить для заказа нового сотрудника.
order.Employee = emp;
db.SubmitChanges();
```

Затем запрашивается другой сотрудник, с `EmployeeID`, равным `9`; этот сотрудник будет назначен ответственным за ранее извлеченный заказ. После этого изменение сохраняется с помощью вызова метода `SubmitChanges`.

Теперь, чтобы доказать, что изменения действительно были проведены на обоих концах, можно просто отобразить назначенного на данный заказ сотрудника, хотя это может показаться не совсем убедительным, поскольку установка свойства `Employee` заказа еще не доказывает изменения этого отношения на стороне заказчика. Было бы более наглядно найти только что измененный заказ в коллекции заказов нового сотрудника, что и делается.

```
Order order2 = (from o in emp.Orders
                where o.OrderID == order.OrderID
                select o).First<Order>();
```

В приведенном коде запрашивается измененный заказ по его `OrderID` в коллекции `Orders` нового заказчика. Если он будет найден, это докажет, что отношение между сотрудником и заказом было обновлено на обоих концах.

```
Console.WriteLine("{0}После изменения сотрудника.", System.Environment.NewLine);
Console.WriteLine("OrderID = {0} : OrderDate = {1} : EmployeeID = {2}",
  order2.OrderID, order2.OrderDate, order2.Employee.EmployeeID);
```

Здесь просто выводится на консоль сообщение, говорящее о том, что ниже будет отображен заказ после проведенного изменения — передачи его новому сотруднику `emp`. Затем выводится информация о заказе. Теперь можно видеть, что `EmployeeID` сотрудника, которому он передан, равен `9`. Ранее он был равен `5`.

```
// Теперь необходимо отменить изменения,
// чтобы пример можно было запускать многократно.
order.Employee = origEmployee;
db.SubmitChanges();
```

Последние две строки кода восстанавливают базу данных в том состоянии, которое она имела до этого, чтобы пример можно было запускать многократно.

Рассмотрим вывод кода из листинга 14.19:

```
Перед изменением сотрудника.
OrderID = 11043 : OrderDate = 4/22/1998 12:00:00 AM : EmployeeID = 5
После изменения сотрудника.
OrderID = 11043 : OrderDate = 4/22/1998 12:00:00 AM : EmployeeID = 9
```

Как видите, сотрудником, которому был назначен заказ перед проведением изменения, был тот, у которого `EmployeeID` равен `5`. После замены назначенного на заказ сотрудника его `EmployeeID` равен `9`. Это доказывает, что заказ был изменен в отношении и со стороны сотрудника.

В данном примере была обновлена ссылка на родителя в дочернем объекте, где дочерним выступает заказ, а родителем — сотрудник. Но есть и другой подход, посредством которого можно было получить тот же результат. Можно было бы обновить ссылку на дочерний объект в родительском объекте.

Обновление ссылки на дочерний объект в родительском объекте

Другой подход к изменению отношения между двумя объектами заключается в удалении дочернего объекта из коллекции `EntitySet<T>` родительского объекта и добавление его в коллекцию `EntitySet<T>` другого родительского объекта. В листинге 14.20 удаляется заказ из коллекции заказов сотрудника. Поскольку этот пример проще, чем пример из листинга 14.19, детальные пояснения приводиться не будут, но существенные отличия в коде выделяются полужирным.

Листинг 14.20. Изменение отношения удалением и добавлением дочернего объекта в `EntitySet` родителя

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
Order order = (from o in db.Orders
               where o.EmployeeID == 5
               orderby o.OrderDate descending
               select o).First<Order>();

// Сохранить текущего сотрудника, чтобы восстановить его в конце.
Employee origEmployee = order.Employee;
Console.WriteLine("Перед изменением сотрудника.");
Console.WriteLine("OrderID = {0} : OrderDate = {1} : EmployeeID = {2}",
  order.OrderID, order.OrderDate, order.Employee.EmployeeID);
Employee emp = (from e in db.Employees
                where e.EmployeeID == 9
                select e).Single<Employee>();
// Удалить заказ из Orders исходного сотрудника.
origEmployee.Orders.Remove(order);
```

```
// Добавить его к заказам нового сотрудника.
emp.Orders.Add(order);
db.SubmitChanges();

Console.WriteLine("{0}После изменения сотрудника.", System.Environment.NewLine);
Console.WriteLine("OrderID = {0} : OrderDate = {1} : EmployeeID = {2}",
  order.OrderID, order.OrderDate, order.Employee.EmployeeID);

// Теперь необходимо отменить изменения, чтобы пример можно было запускать многократно.
order.Employee = origEmployee;
db.SubmitChanges();
```

В этом коде извлекается самый последний заказ для сотрудника с EmployeeID, равным 5, который затем сохраняется в переменной origEmployee, чтобы восстановить в конце примера. Перед сменой сотрудника информация о заказе выводится на консоль. После этого извлекается сотрудник с EmployeeID, равным 9, и ссылка на него сохраняется в переменной emp. До этого момента код совпадает с листингом 14.19.

Затем заказ удаляется из коллекции заказов исходного сотрудника и добавляется в коллекцию заказов нового сотрудника. После этого вызовом метода SubmitChanges изменения в базе данных сохраняются. Далее после проведенных изменений информация о заказе выводится на консоль. И, наконец, заказ восстанавливается в исходном состоянии, чтобы пример можно было запускать многократно.

Ниже показаны результаты работы кода из листинга 14.20.

```
Перед изменением сотрудника.
OrderID = 11043 : OrderDate = 4/22/1998 12:00:00 AM : EmployeeID = 5
После изменения сотрудника.
OrderID = 11043 : OrderDate = 4/22/1998 12:00:00 AM : EmployeeID = 9
```

Удаления

Чтобы удалить запись из базы данных с помощью LINQ to SQL, понадобится удалить сущностный объект из Table<T>, членом которой он является, вызвав метод DeleteOnSubmit объекта Table<T>. После этого, конечно, необходимо вызвать метод SubmitChanges. Пример приведен в листинге 14.21.

Внимание! В отличие от прочих примеров этой главы, в этом примере база данных в исходное состояние не восстанавливается. Причина в том, что одна из участвующих таблиц содержит столбец идентичности, и не так просто программно восстановить данные в состояние, предшествовавшее запуску примера. Поэтому перед запуском примера удостоверьтесь в наличии резервной копии базы данных. Если вы загрузили упакованную расширенную версию базы данных Northwind, то после запуска этого примера можете просто отсоединить базу данных Northwind, заново извлечь файлы базы данных и снова подключить ее к серверу базы.

Листинг 14.21. Удаление записи посредством удаления ее из Table<T>

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
// Получить заказчика для удаления.
Customer customer = (from c in db.Customers
                     where c.CompanyName == "Alfreds Futterkiste"
                     select c).Single<Customer>();
db.OrderDetails.DeleteAllOnSubmit(
  customer.Orders.SelectMany(o => o.OrderDetails));
db.Orders.DeleteAllOnSubmit(customer.Orders);
db.Customers.DeleteOnSubmit(customer);
db.SubmitChanges();
Customer customer2 = (from c in db.Customers
```

```
                    where c.CompanyName == "Alfreds Futterkiste"
                    select c).SingleOrDefault<Customer>();
Console.WriteLine("Заказчик {0} найден.", customer2 != null ? "успешно" : "не");
```

Этот пример достаточно прямолинеен, но не лишен интересных аспектов. Во-первых, поскольку таблица `Orders` содержит внешний ключ, ссылающийся на таблицу `Customers`, нельзя удалить заказчика без предварительного удаления всех его заказов. И поскольку таблица `OrderDetails` содержит внешний ключ, ссылающийся на таблицу `Orders`, нельзя удалить заказ, не удалив его деталей. Таким образом, чтобы удалить заказчика, сначала понадобится удалить детали всех его заказов, затем сами его заказы, и только потом — собственно заказчика.

Удаление всех заказов не представляет труда, благодаря операции `DeleteAllOnSubmit`, которая может удалить последовательность заказов. Однако удаление деталей каждого заказа несколько сложнее. Конечно, можно было бы выполнить перечисление последовательности заказов, вызывая операцию `DeleteAllOnSubmit` на каждой последовательности деталей заказов, но это утомительно. Вместо этого вызывается операция `SelectMany` для получения последовательности последовательностей деталей заказов и затем создается единая совокупная последовательность деталей заказов, которая передается операции `DeleteAllOnSubmit`.

После удаления деталей заказов, самих заказов и их заказчика вызывается метод `SubmitChanges`. Чтобы доказать, что заказчика больше нет, он запрашивается снова с выводом на консоль сообщения.

Посмотрим на вывод кода из листинга 14.21:

```
Заказчик не найден.
```

Вывод не особо впечатляет, но он доказывает, что заказчика больше нет. Хотя целью листинга 14.21 была демонстрация того, что для удаления сущностного объекта он должен быть удален из соответствующей коллекции `Table<T>`, этот пример также наглядно демонстрирует действие операции `SelectMany`.

На заметку! Не забывайте, что в этом примере база данных не восстанавливается в исходном состоянии, так что это должно быть сделано вручную.

Удаление присоединенных сущностных объектов

В отличие от автоматической вставки в базу данных присоединенных ассоциированных зависимых сущностных объектов с помощью `DataContext`, как было описано в листинге 14.3, когда вставлялся их родительский объект, в случае удаления родительского объекта зависимые объекты автоматически не удаляются. Под зависимыми понимаются сущностные объекты, имеющие внешний ключ. Демонстрация процесса удаления была представлена в листинге 14.21, где пришлось удалять записи `OrderDetails` перед записями `Orders`, а записи `Orders` — перед записью `Customers`.

Поэтому, например, если вы попытаетесь удалить заказ в базе данных Northwind, то его детали не будут удалены автоматически. Попытка удаления заказа приведет к нарушению ограничения внешнего ключа. Перед тем, как удалить сущностный объект, потребуется удалить все присоединенные к нему ассоциированные дочерние объекты.

Примеры этого процесса были показаны в листингах 14.21 и 14.3. В каждом из них приходилось удалять ассоциированный присоединенный сущностный объект до того, как удалять его родительский объект.

Удаление отношений

Чтобы удалить отношение между двумя сущностными объектами в LINQ to SQL, ссылке сущностного объекта, указывающей на связанный объект, присваивается значение `null`. Тем самым удаляется отношение данного сущностного объекта с сущностью этого типа. Однако удаление отношения присваиванием `null` на самом деле не удаляет самой записи. Помните, что для действительного удаления записи соответствующий ей сущностный объект должен быть удален из соответствующей `Table<T>`. В листинге 14.22 приведен пример удаления отношения.

Листинг 14.22. Удаление отношения между двумя сущностными объектами

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
// Получить заказ для удаления отношения.
Order order = (from o in db.Orders
               where o.OrderID == 11043
               select o).Single<Order>();

// Сохранить исходного заказчика, чтобы можно было восстановить обратно.
Customer c = order.Customer;
Console.WriteLine("Заказы перед удалением отношения:");
foreach (Order ord in c.Orders)
{
  Console.WriteLine("OrderID = {0}", ord.OrderID);
}

// Удалить отношение с заказчиком.
order.Customer = null;
db.SubmitChanges();
Console.WriteLine("{0}Заказы после удаления отношения:",
  System.Environment.NewLine);

foreach (Order ord in c.Orders)
{
  Console.WriteLine("OrderID = {0}", ord.OrderID);
}

// Восстановить базу данных обратно в исходное состояние.
order.Customer = c;
db.SubmitChanges();
```

В листинге 14.22 запрашивается определенный заказ — тот, у которого `OrderID` равен `11043`. Ссылка на заказчика этого заказа сохраняется для восстановления в конце примера. Далее все заказы этого заказчика выводятся на консоль, ссылке на заказчика для извлеченного заказа присваивается значение `null`, после чего вызывается метод `SubmitChanges` для сохранения изменений в базе данных. Затем снова отображаются все заказы данного заказчика, и на этот раз в списке заказ с `OrderID`, равным `11043`, отсутствует. Вывод кода из листинга 14.22 показан ниже.

```
Заказы перед удалением отношения:
OrderID = 10738
OrderID = 10907
OrderID = 10964
OrderID = 11043

Заказы после удаления отношения:
OrderID = 10738
OrderID = 10907
OrderID = 10964
```

Как видите, после удаления отношения между заказчиком и заказом с `OrderID`, равным `11043`, он исчез из коллекции заказов данного заказчика.

Переопределение операторов модификации базы данных

Если вы думали, что применение LINQ to SQL в существующей среде невозможно, скажем, из-за требований обязательного применения хранимых процедур для проведения всех модификаций базы данных, то вам будет интересно узнать, что действительный код, вызываемый для проведения обновлений, включая вставки и удаления, может быть переопределен.

Переопределение кода, вызываемого для вставки, обновления и удаления, сводится просто к определению соответственно именованного *частичного* метода с соответствующей сигнатурой. Выполнив такое переопределение, вы заставляете процессор изменений `DataContext` вызывать вашу реализацию частичного метода для обновления, вставки и удаления записей базы данных. Это еще одно из преимуществ частичных методов, использованное Microsoft. Вы получаете возможность вмешаться в код без каких-либо накладных расходов, если вы ею не воспользуетесь.

Следует помнить, однако, что принимая этот подход, вы берете на себя ответственность за обнаружение конфликтов параллельного доступа. Внимательно прочтите главу 17, прежде чем делать это.

Определяя эти методы, вы задаете имя частичного метода и тип параметра сущностного типа, которые инструктируют `DataContext` для их вызова в надлежащий момент. Давайте рассмотрим прототипы методов, которые должны быть определены для переопределения операций вставки, обновления и удаления.

Переопределение метода `Insert`

Для переопределения метода, вызываемого для вставки записи в базу данных, реализуется частичный метод со следующим прототипом:

```
partial void Insert[EntityClassName](T instance)
```

Здесь `[EntityClassName]` — имя сущностного класса, метод вставки которого переопределяется, а `T` — тип сущностного класса.

Вот пример переопределяемого прототипа метода вставки для сущностного класса `Snipper`:

```
partial void InsertShipper(Shipper instance)
```

Переопределение метода `Update`

Для переопределения метода, вызываемого для обновления записи в базе данных, реализуется частичный метод со следующим прототипом:

```
partial void Update[EntityClassName](T instance)
```

Здесь `[EntityClassName]` — имя сущностного класса, метод обновления которого переопределяется, а `T` — тип сущностного класса.

Вот пример переопределяемого прототипа метода обновления для сущностного класса `Snipper`:

```
partial void UpdateShipper(Shipper instance)
```

Переопределение метода Delete

Для переопределения метода, вызываемого для удаления записи в базе данных, реализуется частичный метод со следующим прототипом:

```
partial void Delete[EntityClassName](T instance)
```

Здесь [EntityClassName] — имя сущностного класса, метод удаления которого переопределяется, а T — тип сущностного класса.

Вот пример переопределяемого прототипа метода удаления для сущностного класса Snipper:

```
partial void DeleteShipper(Shipper instance)
```

Пример

Для примера, демонстрирующего переопределение методов вставки, обновления и удаления, вместо модификации сгенерированного файла сущностных классов будет создан новый файл, специально для переопределенных частичных методов, чтобы в случае, если придется повторно генерировать файл сущностных классов, переопределения частичных методов не были потеряны. Этот файл называется NorthwindExtended.cs и имеет следующее содержимое.

Файл NorthwindExtended.cs с переопределенными методами обновления базы данных

```
using System;
using System.Data.Linq;
namespace nwind
{
  public partial class Northwind : DataContext
  {
    partial void InsertShipper(Shipper instance)
    {
      Console.WriteLine("Для грузоотправителя {0} вызван переопределенный метод
вставки.", instance.CompanyName);
    }
    partial void UpdateShipper(Shipper instance)
    {
      Console.WriteLine("Для грузоотправителя {0} вызван переопределенный метод
обновления.", instance.CompanyName);
    }
    partial void DeleteShipper(Shipper instance)
    {
      Console.WriteLine("Для грузоотправителя {0} вызван переопределенный метод
удаления.", instance.CompanyName);
    }
  }
}
```

На заметку! В проект Visual Studio потребуется добавить файл, содержащий эти частичные определения класса.

Первое, что следует отметить относительно переопределенного кода — это тот факт, что переопределенные методы являются частичными методами, определенными на уровне DataContext. Они не определены в сущностном классе, к которому относятся.

Как видите, переопределенные методы не делают ничего помимо того, что уведомляют о том, что они были вызваны. Во многих ситуациях переопределение нужно для вызова хранимых процедур, хотя это оставлено на усмотрении разработчика.

Теперь давайте взглянем на листинг 14.23, который содержит код, вызывающий переопределенные методы.

Листинг 14.23. Пример с переопределенными методами обновления, вставки и удаления

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
Shipper ship = (from s in db.Shippers
                where s.ShipperID == 1
                select s).Single<Shipper>();
ship.CompanyName = "Jiffy Shipping";
Shipper newShip =
  new Shipper
  {
    ShipperID = 4,
    CompanyName = "Vickey Rattz Shipping",
    Phone = "(800) SHIP-NOW"
  };
db.Shippers.InsertOnSubmit(newShip);
Shipper deletedShip = (from s in db.Shippers
                       where s.ShipperID == 3
                       select s).Single<Shipper>();
db.Shippers.DeleteOnSubmit(deletedShip);
db.SubmitChanges();
```

В листинге 14.23 сначала извлекается грузоотправитель, у которого `ShipperID` равен `1`, и обновляется поле. Затем вставляется другой грузоотправитель — Vickey Rattz Shipping, и еще один удаляется — тот, у которого `ShipperID` равно `3`. Естественно, поскольку вызываются переопределенные методы, которые только выводят сообщение на консоль, ни одно из изменений в базе данных не сохраняется. Вот как выглядит результат работы листинга 14.23:

```
Для грузоотправителя Jiffy Shipping вызван переопределенный метод обновления.
Для грузоотправителя Vickey Rattz Shipping вызван переопределенный метод вставки.
Для грузоотправителя Federal Shipping вызван переопределенный метод удаления.
```

В этом результате можно видеть, что вызывался каждый из переопределенных методов. Теперь возникает вопрос: а как переопределить методы вставки, обновления и удаления, не отменяя поведения по умолчанию?

Поскольку необходимый код вступил бы в конфликт с частичными методами из предыдущего примера, работающий пример этого приводиться не будет, а будут даны только объяснения, как это сделать. Чтобы получить поведение по умолчанию, в реализациях частичных методов для вставки, обновления и удаления необходимо вызывать методы `DataContext.ExecuteDynamicInsert`, `DataContext.ExecuteDynamicUpdate` и `DataContext.ExecuteDynamicDelete`.

Например, если бы в предыдущем примере нужно было сначала вызвать протоколирование сообщений, а затем нормальный код LINQ to SQL, чтобы сохранить изменения в базе данных, понадобилось бы изменить реализацию частичного метода следующим образом.

Переопределение методов вставки, обновления и удаления, а также вызов поведения по умолчанию

```
namespace nwind
{
  public partial class Northwind : DataContext
  {
```

```
    partial void InsertShipper(Shipper instance)
    {
      Console.WriteLine("Для грузоотправителя {0} вызван переопределенный метод
вставки.", instance.CompanyName);
      this.ExecuteDynamicInsert(instance);
    }
    partial void UpdateShipper(Shipper instance)
    {
      Console.WriteLine("Для грузоотправителя {0} вызван переопределенный метод
обновления.", instance.CompanyName);
      this.ExecuteDynamicUpdate(instance);
    }
    partial void DeleteShipper(Shipper instance)
    {
      Console.WriteLine("Для грузоотправителя {0} вызван переопределенный метод
удаления.", instance.CompanyName);
      this.ExecuteDynamicDelete(instance);
    }
  }
}
```

Обратите внимание, что в каждом из частичных методов вызывается соответствующий метод `ExecuteDynamicInsert`, `ExecuteDynamicUpdate` или `ExecuteDynamicDelete`. Теперь можно расширить поведение при вызове сущностного класса, модифицировать его или даже создать оболочку для существующего поведения по умолчанию.

Переопределение в Object Relational Designer

Не забывайте, что, как говорилось в главе 13, можно переопределить методы вставки, обновления и удаления в Object Relational Designer.

Соображения

Следует помнить, что при переопределении методов обновления, вставки и удаления вы возлагаете на себя ответственность за обнаружение конфликтов параллельного доступа. Это значит, что вы должны хорошо разбираться в том, как работает текущая реализация обнаружения таких конфликтов. Например, Microsoft реализовала ее так, что в условии `where` оператора обновления требуется указание всех существенных полей, вовлеченных в обновление. Затем логика проверяет, сколько записей было обновлено оператором. Вы должны следовать аналогичному шаблону, и в случае обнаружения конфликта параллелизма генерировать исключение `ChangeConflictException`. Прежде чем переопределять эти методы, прочтите главу 17.

Трансляция SQL

При написании запросов LINQ to SQL вы, вероятно, заметили, что при указании выражений, таких как в конструкции `where`, используется “родной” язык программирования, а не SQL. В конце концов, языковая интеграция является частью предназначения LINQ. В настоящей книге эти выражения пишутся на C#.

Например, в листинге 14.2 запрос выглядит так, как показано ниже.

Пример запроса LINQ to SQL

```
Customer cust = (from c in db.Customers
                 where c.CustomerID == "LONEP"
                 select c).Single<Customer>();
```

Обратите внимание, что выражение в конструкции `where` записано в синтаксисе C#, а не в синтаксисе SQL, который бы выглядел следующим образом.

Пример неправильного запроса LINQ to SQL

```
Customer cust = (from c in db.Customers
                 where c.CustomerID = 'LONEP'
                 select c).Single<Customer>();
```

Вместо операции эквивалентности C# (==) в SQL применяется операция эквивалентности (=). Строковый литерал SQL заключается не в двойные ("), а в одинарные ('') кавычки. Одной из целей LINQ является разрешение разработчикам программировать на своем родном языке программирования. Помните, что LINQ — это язык интегрированных запросов. Однако, поскольку база данных не выполняет выражений C#, эти выражения должны быть транслированы в корректный SQL. Поэтому ваши запросы должны транслироваться в SQL.

И это сразу означает, что то, что можно делать, имеет свои ограничения. Хотя в основном трансляция работает достаточно хорошо. Вместо того чтобы дублировать руководство типа справочной системы MSDN для описания этого процесса трансляции с указанием того, что можно и что нельзя транслировать, рассмотрим, чего можно ожидать в тех случаях, когда запрос LINQ to SQL не может быть транслирован.

Первым делом, имейте в виду, что нетранслируемый код может компилироваться. Некорректная трансляция может не проявить себя до тех пор, пока запрос в действительности не будет выполнен. Из-за отложенного выполнения запроса это также означает, что строка кода, определяющая запрос, может выполниться успешно. И только когда дойдет дело до реального извлечения результата, неправильная трансляция заявит о себе в виде исключения, подобного следующему:

```
Unhandled Exception: System.NotSupportedException: Method 'TrimEnd' has no
supported translation to SQL.
```

Необработанное исключение: System.NotSupportedException: Метод 'TrimEnd' не имеет поддерживаемой трансляции в SQL.

...

Это совершенно ясное сообщение об ошибке. Давайте рассмотрим код из листинга 14.24, который генерирует это исключение.

Листинг 14.24. Запрос LINQ to SQL, который не может транслироваться

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
IQueryable<Customer> custs = from c in db.Customers
                             where c.CustomerID.TrimEnd('K') == "LAZY"
                             select c;
foreach (Customer c in custs)
{
  Console.WriteLine("{0}", c.CompanyName);
}
```

Обратите внимание, что метод `TrimEnd`, генерирующий исключение трансляции, вызывается на поле базы данных, а не на локальном строковом литерале. В листинге 14.25 вызов метода `TrimEnd` перенесен в другую сторону равенства.

Листинг 14.25. Запрос LINQ to SQL, который может транслироваться

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
IQueryable<Customer> custs = from c in db.Customers
                             where c.CustomerID == "LAZY".TrimEnd('K')
                             select c;
```

```
foreach (Customer c in custs)
{
  Console.WriteLine("{0}", c.CompanyName);
}
```

Вывод листинга 14.25 выглядит так:

Как видите, здесь вообще нет вывода. Но так и должно быть, поскольку это правильный вывод запроса, и никакого исключения трансляции SQL здесь нет.

Таким образом, вызов неподдерживаемого метода на столбце базы данных приводит к исключению, в то время как при вызове того же метода на переданном параметре все в порядке. Это имеет смысл. LINQ to SQL не испытывает никаких проблем с вызовом метода `TrimEnd` на данном параметре, потому что может сделать это до привязки параметра к запросу, что и происходит в среде процесса. Вызов метода `TrimEnd` на столбце базы данных должен был бы выполняться в базе данных, а это значит, что вместо вызова метода в среде процесса этот вызов должен был быть транслирован в оператор SQL, передаваемый базе данных и выполняемый там. А поскольку метод `TrimEnd` не поддерживается в трансляции SQL, генерируется исключение.

Следует иметь в виду одну вещь. Если нужно вызвать неподдерживаемый метод на столбце базы данных, возможно, вместо этого можно вызвать метод, имеющий противоположный эффект, но вызвать его на параметре. Скажем, к примеру, что требуется вызвать метод `ToUpper` на столбце базы данных, а он не поддерживается; возможно, вместо этого удастся вызвать метод `ToLower` на параметре. Однако в этом случае метод `ToUpper` все-таки поддерживается, так что пример не очень удачный. Также следует удостовериться, что вызываемый метод дает прямо противоположный эффект. В рассматриваемом случае столбец базы данных может содержать строку в смешанном регистре, так что вызов `ToLower` не будет иметь в точности противоположного эффекта. Если столбец базы данных содержит значение “Smith”, а параметр — “SMITH”, и производится проверка эквивалентности, то вызов метода `ToUpper` на столбце базы данных будет работать и обеспечит соответствие. Однако если бы метод `ToUpper` не поддерживался, попытка вывернуть эту логику наизнанку вызовом метода `ToLower` на параметре не дала бы соответствия.

Может возникнуть вопрос: как узнать, что метод `TrimEnd` не поддерживается в трансляции SQL? Поскольку природа того, какие примитивные типы и методы поддерживаются, динамична и подвержена изменениям, попытка документировать их все выходит за рамки контекста этой книги. Относительно трансляции существует масса ограничений и исключений. Совершенствование трансляции SQL является областью приложения усилий со стороны Microsoft. Чтобы узнать, что поддерживается, а что нет, загляните в раздел документации MSDN, озаглавленный “.NET Framework Function Translation” для LINQ to SQL. Однако, как видно из предыдущих примеров, узнать, что метод не поддерживается, очень легко.

Резюме

Эта глава выглядит как молниеносный тур по стандартным операциям для баз данных с использованием LINQ to SQL. Приведенные в главе примеры были достаточно простыми, чтобы позволить сосредоточиться на основных шагах, необходимых для выполнения вставок, обновлений и удалений в базе данных. Также были указаны отличия в работе запросов LINQ to SQL и LINQ to Objects.

Имейте в виду, что любой код LINQ to SQL, который изменяет базу данных, должен обнаруживать и разрешать конфликты параллельного доступа, которые подробно рассматриваются в главе 17.

В дополнение к пониманию того, как выполняются основные операции на сущностных объектах, важно также знать, как это затрагивает сущностные объекты, ассоциированные с данным объектом. Помните, что когда вставляется сущностный объект в базу данных, все присоединенные к нему объекты будут вставлены автоматически. Однако это не касается удалений. Чтобы удалить родительский сущностный объект в отношении ассоциации, понадобится сначала удалить дочерние сущностные объекты; в противном случае будет сгенерировано исключение.

Далее было показано, как переопределять методы по умолчанию, которые сгенерированы для модификации записей базы данных, соответствующих сущностному объекту. Такая возможность позволяет разработчику управлять внесением изменений в базу данных, например, выполняя их посредством хранимых процедур.

И, наконец, был раскрыт тот факт, что запросы LINQ to SQL должны транслироваться в операторы SQL. Важно никогда не забывать об этой трансляции, и о том, что она накладывает некоторые ограничения на то, что допускается делать.

До настоящего момента в этой книге уже не раз упоминались сущностные классы, не слишком углубляясь в них. Теперь пришло время обратить на них более пристальное внимание. Этой теме посвящена следующая глава.

ГЛАВА 15

Сущностные классы LINQ to SQL

В предыдущих главах, посвященных LINQ to SQL, много раз упоминались сущностные классы, но пока еще не было приведено их строгое определение и описание. В настоящей главе будет дано определение сущностных классов, а также описаны различные способы их создания. Кроме того, вы узнаете о сложностях, с которыми придется столкнуться, если вы решите создавать собственные сущностные классы.

Но, прежде чем мы приступим к самому интересному, нужно сказать о некоторых предварительных условиях, которые должны быть удовлетворены, чтобы можно было запускать примеры этой главы.

Предварительные условия для запуска примеров

Для того чтобы запускать примеры настоящей главы, необходимо получить расширенную версию базы данных Northwind и сущностные классы, сгенерированные для нее. Прочтите и выполните инструкции, приведенные в разделе “Предварительные условия для запуска примеров” главы 12.

Сущностные классы

Классы, которые отображаются на базу данных SQL Server с использованием LINQ to SQL, называются *сущностными классами* (entity classes). Созданный экземпляр сущностного класса — это сущность данного типа, и это будет называться *сущностным объектом* (entity object). Сущностные классы — это обычные классы C# с дополнительно заданными атрибутами LINQ to SQL. Вместо добавления атрибутов сущностные классы могут быть созданы с применением XML-файла отображения при создании экземпляра объекта `DataContext`. Эти атрибуты или вхождения файла отображения диктуют то, как сущностные классы могут отображаться на базу данных SQL Server с использованием LINQ to SQL.

Применяя сущностные классы, можно опрашивать и обновлять базу данных с использованием LINQ to SQL.

Создание сущностных классов

Сущностные классы — это базовые строительные блоки, участвующие в выполнении запросов LINQ to SQL. Для того чтобы начать использовать LINQ to SQL, необходимы сущностные классы. Есть два способа их получения: их можно сгенерировать, как было

показано в главах 12 и 13, или же можно написать их вручную. К тому же нет причин, запрещающих комбинировать оба способа.

Если еще нет бизнес-классов для сущностей, хранящихся в базе данных, генерация сущностных классов — наверное, лучший подход. Если уже имеется объектная модель, тогда более предпочтительно написание сущностных классов вручную.

Если вы начинаете проект с нуля, рекомендуется рассмотреть сначала моделирование базы данных и сгенерировать сущностные классы из этой базы данных, что предотвратит сложности, связанные с их написанием вручную.

Генерация сущностных классов

В главе 12 было показано, как сгенерировать сущностные классы из базы данных Northwind, чтобы можно было запускать примеры LINQ to SQL, приведенные в этой книге. В главе 13 детально описывалось, как генерировать сущностные классы с использованием либо инструмента командной строки по имени SQLMetal, либо инструмента с графическим интерфейсом пользователя под названием Object Relational Designer.

SQLMetal очень прост в использовании, но не предусматривает никаких опций для управления именованием сгенерированных сущностных классов помимо производства промежуточного XML-файла, который можно редактировать. Затем SQLMetal генерирует сущностные классы для каждой таблицы в указанной базе данных и для каждого поля в каждой из таблиц. Object Relational Designer потребует больше времени на создания полной объектной модели базы данных, но позволит точно указать таблицы и поля, для которых необходимо сгенерировать сущностные классы, а также задать имена этих классов и их свойств. Эти инструменты уже обсуждались в главе 13, так что обращайтесь в эту главу за дополнительными сведениями по их использованию.

Существует разница между генерацией сущностного класса и его использованием. Можно сгенерировать сущностные классы для всех таблиц базы данных, но это не значит, что они все будут использоваться.

А использование сгенерированных сущностных классов не означает, что к ним нельзя добавить специальную функциональность. Например, в главе 12 с помощью SQLMetal был сгенерирован класс `Customer`. Нет причин, по которым нельзя было бы добавить к нему собственные бизнес-методы или непостоянные члены класса. Однако при этом удостоверьтесь, что модификации не касаются сгенерированной части кода сущностного класса. Вместо этого создайте другой модуль класса `Customer` и воспользуйтесь преимуществом того факта, что сущностные классы генерируются как частичные классы. Частичные классы — замечательное приобретение C#, которое, как никогда ранее, облегчает разделение функциональности по разным модулям. Таким образом, если вам по какой-то причине придется повторно сгенерировать сущностный класс, вы не потеряете добавленные члены или методы.

Написание сущностных классов вручную

Написание сущностных классов вручную — более трудный подход. Он требует глубокого понимания атрибутов LINQ to SQL и/или внешней схемы отображения. Однако в то же время написание сущностных классов вручную — замечательный способ действительно изучить LINQ to SQL.

Написание сущностных классов вручную действительно оправдано в случае, когда уже имеется объектная модель для работы. В этом случае было бы не слишком выгодно генерировать сущностные классы из базы данных, поскольку есть объектная модель, используемая приложением. В таких случаях можно либо добавить необходимые атрибуты в существующую объектную модель, либо создать файл отображения. Благодаря гибкости LINQ to SQL нет необходимости, чтобы имена классов соответствовали именам

таблиц, в которых они сохраняются, а имена свойств — именам столбцов этих таблиц. Это значит, что ранее реализованные классы могут быть модифицированы для хранения в базе данных SQL Server.

Чтобы создать сущностные классы вручную с использованием атрибутов, понадобится добавить соответствующие атрибуты к классам — будь то существующие бизнес-классы либо новые классы, — которые должны стать сущностными. Описание доступных атрибутов и свойств можно найти в разделе "Атрибуты сущностных классов и свойства атрибутов" далее в этой главе.

Чтобы создать сущностные классы с использованием внешнего файла отображения, необходимо создать XML-файл, который отвечает схеме, описанной в разделе "XML-схема внешнего файла отображения" далее в главе. Имея такой внешний файл отображения, можно использовать соответствующий конструктор `DataContext` для создания объекта `DataContext`, который загрузит файл отображения. У этого класса есть два конструктора, позволяющие указывать внешний файл отображения.

Дополнительная ответственность сущностных классов

К сожалению, при написании сущностных классов вручную недостаточно только понимать атрибуты и свойства этих атрибутов. Также необходимо знать о некоторой дополнительной ответственности сущностных классов.

Например, следует помнить об уведомлениях, связанных с изменением, и способе их реализации. Понадобится также обеспечивать целостность графа между родительскими и дочерними классами.

Эти дополнительные обязанности учитываются при генерации классов инструментальными средствами SQLMetal и Object Relational Design, но если вы создаете собственный сущностный класс, то должны самостоятельно добавить весь необходимый код.

Уведомления об изменениях

Отслеживание изменений рассматривается в главе 16. В свою очередь, отслеживание изменений не может быть реализовано элегантно и эффективно без участия самих сущностных классов. Если сущностные классы сгенерированы SQLMetal или Object Relational Design, можно расслабиться, поскольку эти инструменты позаботятся обо всем, реализуя код, участвующий в уведомлениях об изменении, при генерации сущностных классов. Но при написании собственных сущностных классов следует понимать механизм уведомления об изменении и самостоятельно реализовывать код, участвующий в этом процессе.

Можно выбрать, должны ли сущностные классы участвовать в уведомлениях об изменении. Если они не участвуют, то `DataContext` обеспечивает это, сохраняя две копии каждого сущностного объекта; одну для исходных значений и другую — для текущих. Он создает копии при первоначальном извлечении сущности из базы данных в начале отслеживания изменений. Этот механизм можно сделать более эффективным, если сущностные классы будут реализовывать два интерфейса уведомления об изменении — `System.ComponentModel.INotifyPropertyChanging` и `System.ComponentModel.INotifyPropertyChanged`.

На протяжении глав, посвященных LINQ to SQL, мы часто будем ссылаться на код, сгенерированный SQLMetal, чтобы продемонстрировать суть способа обработки уведомления об изменении. Чтобы реализовать интерфейсы `INotifyPropertyChanging` и `INotifyPropertyChanged`, потребуется сделать четыре вещи.

Во-первых, необходимо определенить, что сущностный класс реализует интерфейсы `INotifyPropertyChanging` и `INotifyPropertyChanged`.

Из сгенерированного сущностного класса Customer

```
[Table(Name="dbo.Customers")]
public partial class Customer : INotifyPropertyChanging, INotifyPropertyChanged
{ ... }
```

Поскольку сущностный класс реализует эти два интерфейса, DataContext будет известно, что нужно зарегистрировать два обработчика для двух событий, о которых рассказывается ниже.

В приведенном выше коде можно заметить, что указан атрибут Table. В этом разделе перечисляются связанные с контекстом атрибуты, а позже в главе они подробно объясняются. Пока что их можно просто проигнорировать.

Во-вторых, нужно объявить приватную статическую переменную типа PropertyChangingEventArgs и передать ее конструктору значение String.Empty.

Из сгенерированного сущностного класса Customer

```
[Table(Name="dbo.Customers")]
public partial class Customer : INotifyPropertyChanging, INotifyPropertyChanged
{
  private static PropertyChangingEventArgs emptyChangingEventArgs =
    new PropertyChangingEventArgs(String.Empty);
  ...
}
```

Объект emptyChangingEventArgs будет передан одному из ранее упомянутых обработчиков событий, когда возникнет соответствующее событие.

В-третьих, нужно добавить к сущностному классу два члена public event — один типа System.ComponentModel.PropertyChangingEventHandler по имени PropertyChanging и еще один — типа System.ComponentModel.PropertyChangedEventHandler по имени PropertyChanged.

Из сгенерированного сущностного класса Customer

```
[Table(Name="dbo.Customers")]
public partial class Customer : INotifyPropertyChanging, INotifyPropertyChanged
{
  private static PropertyChangingEventArgs emptyChangingEventArgs =
    new PropertyChangingEventArgs(String.Empty);
  ...
  ...
  public event PropertyChangingEventHandler PropertyChanging;
  public event PropertyChangedEventHandler PropertyChanged;
}
```

Когда объект DataContext инициирует отслеживание изменений сущностного объекта, он регистрирует обработчики для этих двух событий, если сущностный класс реализует два интерфейса уведомления об изменении. Если нет, то DataContext создает копию сущностного объекта, как было описано выше.

В-четвертых, каждый раз, когда свойство отображенного сущностного класса изменяется, необходимо сгенерировать событие PropertyChanging непосредственно перед изменением свойства и событие PropertyChanged — сразу после изменения свойства.

Хотя реализовывать генерацию событий подобным образом не обязательно, для согласованности SQLMetal генерирует методы SendPropertyChanging и SendPropertyChanged.

Из сгенерированного сущностного класса Customer

```
protected virtual void SendPropertyChanging()
{
```

```
    if ((this.PropertyChanging != null))
    {
      this.PropertyChanging(this, emptyChangingEventArgs);
    }
  }
  protected virtual void SendPropertyChanged(String propertyName)
  {
    if ((this.PropertyChanged != null))
    {
      this.PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
    }
  }
```

Обратите внимание, что при возбуждении события `PropertyChanged` создается новый объект `PropertyChangedEventArgs`, получающий имя конкретного свойства, которое подвергается изменению. Это позволяет объекту `DataContext` знать точно, какое свойство изменилось. Поэтому когда метод `SendPropertyChanging` вызывается, он инициирует событие `PropertyChanging`, которое приводит к вызову зарегистрированного объектом `DataContext` обработчика. То же самое происходит с методом `SendPropertyChanged` и событием `PropertyChanged`.

Конечно, можно встроить похожую логику в код вместо создания повторно используемых методов, но это более утомительно, к тому же увеличивает объем сопровождаемого кода.

Затем в методе `set` каждого свойства понадобится вызывать два метода `SendPropertyChanging` и `SendPropertyChanged` непосредственно перед изменением свойства и после него.

Из сгенерированного сущностного класса `Customer`

```
[Column(Storage="_ContactName", DbType="NVarChar(30)")]
public string ContactName
{
  get
  {
    return this._ContactName;
  }
  set
  {
    if ((this._ContactName != value))
    {
      this.OnContactNameChanging(value);
      this.SendPropertyChanging();
      this._ContactName = value;
      this.SendPropertyChanged("ContactName");
      this.OnContactNameChanged();
    }
  }
}
```

Опять-таки, обратите внимание, что при вызове метода `SendPropertyChanged` передается имя свойства, в данном случае — `ContactName`. Как только вызывается метод `SendPropertyChanged`, объект `DataContext` узнает, что свойство сущностного объекта `ContactName` было изменено.

Также необходимо отслеживать возникновение соответствующих событий в методах `set` свойств, представляющих ассоциации. Поэтому на стороне *многие* ассоциации “один ко многим” нужно добавить показанный ниже код, которые выделен полужирным.

Из класса `Order`, поскольку `Customer` не имеет свойств `EntityRef<T>`

```
[Association(Name="FK_Orders_Customers", Storage="_Customer",
  ThisKey="CustomerID", IsForeignKey=true)]
public Customer Customer
{
  get
  {
    return this._Customer.Entity;
  }
  set
  {
    Customer previousValue = this._Customer.Entity;
    if (((previousValue != value)
         || (this._Customer.HasLoadedOrAssignedValue == false)))
    {
      this.SendPropertyChanging();
      if ((previousValue != null))
      {
        this._Customer.Entity = null;
        previousValue.Orders.Remove(this);
      }
      this._Customer.Entity = value;
      if ((value != null))
      {
        value.Orders.Add(this);
        this._CustomerID = value.CustomerID;
      }
      else
      {
        this._CustomerID = default(string);
      }
      this.SendPropertyChanged("Customer");
    }
  }
}
```

А на стороне *один* ассоциации “один ко многим” понадобится следующий код, выделенный полужирным:

Из сгенерированного сущностного класса `Customer`

```
public Customer()
{
  ...
  this._Orders = new EntitySet<Order>(new Action<Order>(this.attach_Orders),
                     new Action<Order>(this.detach_Orders));
}
...
private void attach_Orders(Order entity)
{
  this.SendPropertyChanging();
  entity.Customer = this;
  this.SendPropertyChanged("Orders");
}
private void detach_Orders(Order entity)
{
  this.SendPropertyChanging();
  entity.Customer = null;
  this.SendPropertyChanged("Orders");
}
```

Если вы не знакомы с обобщенным делегатом Action, использованным в приведенном коде, знайте, что он находится в пространстве имен System, и был добавлен в версии .NET Framework 2.0. Предыдущий код создает экземпляр объекта Action для сущностного класса Order и передает ему делегат метода attach_Orders. LINQ to SQL использует этот делегат позднее, чтобы присвоить объекту Order соответствующий объект Customer. Аналогично создается экземпляр другого объекта-делегата Action, которому передается делегат метода detach_Orders. Этот делегат LINQ to SQL применяет позже для удаления присвоенного Customer из Order.

Реализуя подобным образом уведомление об изменении, можно увеличить эффективность слежения за изменениями. Теперь объект DataContext знает, когда и какие свойства сущностного класса изменяются.

Когда вызывается метод SubmitChanges, объект DataContext забывает исходные значения своих свойств, и этими исходными значениями свойств становятся текущие их значения, после чего начинается отслеживание изменений. Метод SubmitChanges детально описан в главе 16.

Конечно, как упоминалось ранее, если позволить SQLMetal или Object Relational Designer создавать сущностные классы, все эти сложности исчезают, потому что данные инструменты генерируют весь необходимый рутинный код. И только при написании сущностных классов вручную следует сосредоточиться на реализации уведомлений об изменении.

Согласованность графа

В математике, когда узлы соединяются вместе, полученная в результате сеть называется *графом*. В некотором отношении сеть, представляющая соединения, которые образованы классами, ссылающимися на другие классы, также считается графом. Когда есть два сущностных класса, участвующие в отношении — в том смысле, что между ними создается ассоциация Association, — то поскольку классы имеют ссылки друг на друга, в результате получается граф.

При модификации отношения между двумя сущностными объектами, такими как Customer и Order, ссылка на каждую сторону отношения должна корректно обновляться, чтобы каждый сущностный объект правильно ссылался или более не ссылался на другой. Это верно как для создания отношений, так и для их удаления. Хотя LINQ to SQL требует, чтобы программист, пишущий код, который использует сущностные классы, модифицировал только одну сторону отношения, иногда приходится обрабатывать обновление другой стороны, и LINQ to SQL не делает это автоматически.

За обработку обновления другой стороны отношения отвечает сущностный класс. Если генерация сущностных классов поручается SQLMetal или Object Relational Designer, то ничего дополнительно делать не понадобится. Но в случае создания собственных сущностных классов придется реализовать весь необходимый код самостоятельно.

При гарантированном правильном обновлении каждой стороны отношения граф остается согласованным. Без этого целостность графа может нарушиться и наступит хаос. Объект Customer может быть связан с Order, а объект Order окажется связанным с другим Customer, или вообще не связанным. К счастью, Microsoft предлагает шаблон, который можно использовать для гарантии того, что сущностные классы корректно реализуют согласованность графов. Взглянем на соответствующую реализацию, сгенерированную SQLMetal на базе данных Northwind.

Из сгенерированного сущностного класса Customer

```
public Customer()
{
  ...
  this._Orders = new EntitySet<Order>(new Action<Order>(this.attach_Orders),
```

```
    new Action<Order>(this.detach_Orders));
}
...
private void attach_Orders(Order entity)
{
  this.SendPropertyChanging();
  entity.Customer = this;
  this.SendPropertyChanged("Orders");
}
private void detach_Orders(Order entity)
{
  this.SendPropertyChanging();
  entity.Customer = null;
  this.SendPropertyChanged("Orders");
}
```

В данном примере класс `Customer` будет родительским, или стороной *один* в отношении "один ко многим". Класс `Order` будет дочерним, представляющим сторону *многие* того же отношения.

В приведенном коде можно увидеть, что при инициализации члена `EntitySet<T>` для коллекции дочерних объектов `_Orders` конструктору передаются два объекта-делегата `Action<T>`.

Первый объект `Action<T>` получает делегат метода обратного вызова, который обработает присваивание текущего объекта `Customer`, на который ссылается ключевое слово `this`, как `Customer` объекта `Order`, переданный в метод обратного вызова. В приведенном выше коде таким методом обратного вызова является `attach_Orders`.

Второй параметр конструктора `EntitySet<T>` — это объект-делегат `Action<T>`, в котором передается делегат метода обратного вызова, обрабатывающий удаление присваивания `Customer` переданного объекта `Order`. В приведенном выше коде методом обратного вызова является `detach_Orders`.

Несмотря на то что приведенный выше код находится в родительском классе `Customer`, присваивание объекта дочернего класса `Order` на самом деле обрабатывается свойством `Customer` объекта `Order`. Это можно видеть как в методе `attach_Orders`, так и в `detach_Orders`; все, что они в действительности делают — это изменяют свойство `Customer` объекта `Order`. Свойство `entity.Customer` устанавливается соответственно в `this` и `null`, чтобы присоединить текущий `Customer` и отсоединить подключенный в данный момент `Customer`. Вся работа по поддержке согласованности графа осуществляется в методах `set` и `get` дочернего класса `Order`. Таким образом, реальная работа делегируется дочернему классу. В родительском классе больше ничего делать не нужно.

Однако обратите внимание, что перед тем, как выполнить работу в методах `attach_Orders` и `detach_Orders`, вызовом методов `SendPropertyChanging` и `SendPropertyChanged` инициируются уведомления об изменении.

Теперь посмотрим, что нужно сделать в дочернем классе отношения "родительский–дочерний" для поддержки согласованности графа.

Из сгенерированного сущностного класса `Order`

```
[Association(Name="FK_Orders_Customers", Storage="_Customer",
  ThisKey="CustomerID", IsForeignKey=true)]
public Customer Customer
{
  get
  {
    return this._Customer.Entity;
  }
```

```
    set
    {
      Customer previousValue = this._Customer.Entity;
      if (((previousValue != value)
          || (this._Customer.HasLoadedOrAssignedValue == false)))
      {
        this.SendPropertyChanging();
        if ((previousValue != null))
        {
          this._Customer.Entity = null;
          previousValue.Orders.Remove(this);
        }
        this._Customer.Entity = value;
        if ((value != null))
        {
          value.Orders.Add(this);
          this._CustomerID = value.CustomerID;
        }
        else
        {
          this._CustomerID = default(string);
        }
        this.SendPropertyChanged("Customer");
      }
    }
  }
```

В приведенном коде рассматривается только метод `set` свойства `Customer` — именно потому, что родительская сторона отношения возлагает на него ответственность на поддержание согласованности графа. Поскольку этот метод довольно сложный, вместе с кодом предоставляется дополнительное описание.

```
set
{
  Customer previousValue = this._Customer.Entity;
```

В первой строке кода метода `set` производится сохранение исходного `Customer`, присвоенного `Order`, в переменную `previousValue`. Пусть вас не смущает тот факт, что код ссылается на `this._Customer.Entity`. Вспомните, что переменная-член `_Customer` — это на самом деле объект типа `EntityRef<Customer>`, а не `Customer`. Чтобы получить объект `Customer`, код должен обратиться к свойству `Entity` объекта `EntityRef<T>`. Поскольку здесь `EntityRef<T>` предназначен для хранения `Customer`, типом `Entity` будет `Customer`; никакого приведения не требуется.

```
if (((previousValue != value)
     || (this._Customer.HasLoadedOrAssignedValue == false)))
{
```

Далее приведенная строка кода проверяет, не присвоен ли уже этот `Customer` объекту `Order`, сравнивая переданный параметр `value` с имеющейся в `Order` ссылкой на `Customer`, поскольку в этом случае ничего делать не надо, если только этот `Customer` не был еще загружен или присвоен. Это не только логически оправдано, но, учитывая рекурсивную природу работы этого кода, эта строка становится чрезвычайно важной, поскольку позволяет остановить рекурсию.

```
this.SendPropertyChanging();
```

В этой строке кода вызывается метод `SendPropertyChanging` для генерации события уведомления об изменении.

```
if ((previousValue != null))
{
```

Далее код определяет, не присвоен ли уже родительский объект Customer данному дочернему объекту Order, сравнивая previousValue с null. Вспомните, что в этой точке значение Customer объекта Order все еще совпадает со значением переменной previousValue.

Если Customer назначен Order, в том смысле, что переменная previousValue, представляющая присвоенный Customer, не равна null, то код должен установить свойство Entity объекта Customer EntityRef<T> в null в следующей строке.

```
this._Customer.Entity = null;
```

Свойство Entity устанавливается здесь в null, чтобы остановить рекурсию, которая запускается в следующей строке кода. Поскольку свойство Entity свойства Customer объекта Order теперь равно null и не ссылается на действительный объект Customer, в то время как свойство Orders объекта Customer все еще содержит этот Order в коллекции, в данный момент времени граф находится в несогласованном состоянии.

В следующей строке кода вызывается метод Remove на свойстве Orders объекта Customer, и текущий Order передается в качестве удаляемого Order.

```
  previousValue.Orders.Remove(this);
}
```

Вызов метода Remove инициирует вызов метода detach_Orders класса Customer и передает ему подлежащий удалению Order. В методе detach_Orders свойству Customer переданного объекта Order присваивается null. Ниже еще раз показано, как выглядит метод detach_Orders.

Это код отдельного метода, приведенный здесь для удобства

```
private void detach_Orders(Order entity)
{
  this.SendPropertyChanging();
  entity.Customer = null;
  this.SendPropertyChanged("Orders");
}
```

Когда вызывается метод detach_Orders, свойство Customer переданного Order получает значение null. Это приводит к вызову метода set свойства Customer объекта Order, а этот метод вызывает код, обращающийся, в свою очередь, к методу detach_Orders, так что метод, запустивший этот процесс удаления Order, вызывается рекурсивно, и значение null передается в качестве value в метод set. Поток выполнения теперь находится в рекурсивном вызове метода set свойства Customer.

Метод detach_Orders инициирует рекурсивный вызов метода set

```
set
{
  Customer previousValue = this._Customer.Entity;
  if (((previousValue != value)
       || (this._Customer.HasLoadedOrAssignedValue == false)))
  {
```

В четвертой строке метода set проверяется переданное значение value, и если оно эквивалентно свойству Entity текущего назначенного свойства Customer, этот рекурсивный вызов метода set возвращает управление, ничего не делая. Поскольку в предыдущей строке кода первого не рекурсивного вызова метода set свойству Entity свойства Customer присваивается значение null, и поскольку null был передан как value в метод detach_Orders, они на самом деле эквивалентны: существует рекурсивный вызов метода set, который ничего не делает, и тогда поток управления возвращается обратно — к первому вызову метода set. Именно это имелось в виду в предыдущем абзаце,

когда говорилось, что свойство Entity устанавливается в null, чтобы остановить рекурсию. Итак, как только рекурсивный вызов метода set вернул управление, поток возвращается к последней строке начального вызова метода set, который обсуждался ранее.

Строка кода, повторяющаяся из предыдущего фрагмента

```
  previousValue.Orders.Remove(this);
}
```

Как только метод Orders.Remove завершает работу, свойство Orders объекта Customer более не содержит ссылки на этот Order, и потому граф вновь находится в согласованном состоянии.

Очевидно, что если планируется писать собственные сущностные классы, стоит потратить некоторое время на их отладку. Просто поставьте точки останова в методах detach_Orders и set и наблюдайте, что происходит.

Затем в свойство Entity объекта Customer объекта Order устанавливается новый объект Customer, который был передан методу set в параметре value.

```
this._Customer.Entity = value;
```

В конце концов, это метод set свойства Customer. Мы пытались назначить Order новому Customer. И, опять-таки, в этой точке Order содержит ссылку на вновь назначенный Customer, а вновь назначенный Customer не имеет ссылки на Order, так что граф отныне не согласован.

Далее код проверяет, не равен ли null назначаемый Order объект Customer, потому что если нет, то вновь назначенный Customer следует назначить Order:

```
if ((value != null))
{
```

Если переданный в value объект Customer не равен null, нужно добавить текущий Order в коллекцию объектов Orders переданного Customer:

```
value.Orders.Add(this);
```

Когда в этой строке Order добавляется в коллекцию Orders переданного объекта Customer, то вызывается делегат, переданный методу обратного вызова в конструкторе EntitySet<T> объекта Customer. Поэтому результатом присваивания является вызов метода attach_Orders объекта Customer.

Это, в свою очередь, установит свойство Customer текущего объекта Order в переданный Customer, в результате чего опять вызывается метод set свойства Customer объекта Order. Код рекурсивно входит в метод set, как делал это ранее. Однако всего двумя операторами кода перед предыдущим оператором и перед началом рекурсии свойство Entity свойства Customer объекта Order было установлено в новый Customer, который передан методу set методом attach_Orders. Опять-таки, код метода set вызывается рекурсивно, и в конечном итоге вызывается следующая строка кода.

Следующая строка кода — из другого вызова метода set

```
if (((previousValue != value)
     || (this._Customer.HasLoadedOrAssignedValue == false)))
```

Поскольку текущий объект Customer объекта Order, который теперь хранится в previousValue, и параметр value — одно и то же, метод set возвращает управление, ничего более не делая, в результате чего рекурсия прекращается.

В следующей строке кода член CustomerID текущего объекта Order устанавливается в CustomerID нового объекта Customer:

```
  this._CustomerID = value.CustomerID;
}
```

Если вновь назначаемый Customer равен null, то этот код просто устанавливает член CustomerID текущего объекта Order в значение по умолчанию для его типа данных — в этом случае типа string:

```
else
{
  this._CustomerID = default(string);
}
```

Если бы член CustomerID был типа int, код установил бы его в default(int).

В самой последней строке кода вызывается метод SendPropertyChanged и ему передается имя изменяемого свойства, чтобы инициировать событие уведомления об изменении:

```
  this.SendPropertyChanged("Customer");
}
```

Этот шаблон действителен для отношения “один ко многим”. Для отношения “один к одному” каждая сторона должны была быть реализована, как была реализована дочерняя в этом примере, но с парой изменений. Поскольку в отношении “один к одному” нет логического родительского или дочернего объекта, предположим, что отношение между заказчиками и заказами строится по схеме “один к одному”. Это даст имя для использования в качестве ссылки на каждый класс, поскольку родительский и дочерний объекты более не применимы.

Если вы пишете сущностный класс вручную, и отношение между классом Customer и классом Order — “один к одному”, тогда каждый из этих классов будет содержать свойство типа EntityRef<T>, где T — тип противоположного класса. Класс Customer будет содержать EntityRef<Order>, а класс Order — EntityRef<Customer>. Поскольку ни один из классов не содержит EntitySet<T>, нет вызовов методов Add и Remove, которые присутствуют в ситуации с отношениями “один ко многим”, описанной ранее.

Таким образом, если предположить, что между заказами и заказчиками существует отношение “один к одному”, то метод set свойства Customer объекта Order должен выглядеть в основном так же, как и ранее, за исключением случая, когда удаляется назначение текущего Order исходному Customer. Поскольку исходный Customer имеет единственный Order, мы не удаляем текущий Order из коллекции объектов Order, а просто присвоим null свойству Customer объекта Order.

Таким образом, вместо такой строки кода:

```
previousValue.Orders.Remove(this);
```

будет следующая строка кода:

```
previousValue.Order = null;
```

Аналогично при назначении текущего Order новому Customer, поскольку он имеет единственный Order, вместо вызова метода Add на коллекции объектов Order просто устанавливается свойство Customer нового объекта Order в текущий Order.

Поэтому вместо следующей строки кода:

```
value.Orders.Add(this);
```

будет такая строка:

```
value.Order = this;
```

Как видите, обработка согласованности графа нетривиальна, и в ней легко запутаться. К счастью, существуют два инструмента, которые обо всем позаботятся — SQLMetal и Object Relational Designer. Они адекватно поддерживают согласованность графа и правильную реализацию уведомлений об изменении.

Вызов соответствующих частичных методов

Когда в Microsoft добавили частичные методы для облегчения расширения сгенерированного кода, такого как код сущностных классов, то тем самым возложили на вас немного больше ответственности, если вы собираетесь самостоятельно реализовывать сущностные классы.

Существуют несколько частичных методов, которые должны быть объявлены во вручную написанных сущностных классах:

```
partial void OnLoaded();
partial void OnValidate(ChangeAction action);
partial void OnCreated();
partial void On[Property]Changing(int value);
partial void On[Property]Changed();
```

Для каждого свойства сущностного класса должна существовать пара методов `On[Property]Changing` и `On[Property]Changed`.

Что касается методов `OnLoaded` и `OnValidate`, то не нужно добавлять их вызовы где-либо в коде сущностных классов; их будет вызывать `DataContext`.

Внутрь конструктора сущностного класса потребуется добавить код вызова метода `OnCreate`, как показано ниже.

Вызов частичного метода `OnCreated`

```
public Customer()
{
  OnCreated();
  ...
}
```

Затем для каждого свойства сущностного класса должны быть добавлены вызовы методов `On[Property]Changing` и `On[Property]Changed` непосредственно перед и после изменения этого свойства, как показано ниже.

Вызов методов `On[Property]Changing` и `On[Property]Changed` из метода `set` свойства сущностного класса

```
public string CompanyName
{
  get
  {
    return this._CompanyName;
  }
  set
  {
    if ((this._CompanyName != value))
    {
      this.OnCompanyNameChanging(value);
      this.SendPropertyChanging();
      this._CompanyName = value;
      this.SendPropertyChanged("CompanyName");
      this.OnCompanyNameChanged();
    }
  }
}
```

Обратите внимание, что метод `On[Property]Changing` вызывается перед вызовом метода `SendPropertyChanging`, а метод `On[Property]Changed` — после метода `SendPropertyChanged`.

Объявляя и вызывая эти частичные методы, вы предоставляете другим разработчикам возможность простого расширения без потери производительности, если они такой возможностью не воспользуются. В этом прелесть частичных методов.

Сложности, связанные с EntityRef<T>

В то время как типом данных для ассоциированного класса является EntityRef<T>, общедоступное свойство для этого приватного члена класса должно возвращать тип сущностного класса, а не EntityRef<T>.

Взглянем на то, как SQLMetal генерирует свойство для члена EntityRef<T>.

Общедоступное свойство для члена класса, возвращающее тип сущностного класса вместо EntityRef<T>

```
[Table(Name="dbo.Orders")]
public partial class Order : INotifyPropertyChanging, INotifyPropertyChanged
{
  ...
  private EntityRef<Customer> _Customer;
  ...
  [Association(Name="FK_Orders_Customers", Storage="_Customer",
    ThisKey="CustomerID", IsForeignKey=true)]
  public Customer Customer
  {
    get
    {
      return this._Customer.Entity;
    }
    set
    {
      ...
    }
  }
  ...
}
```

Как видите, несмотря на то, что приватный член класса имеет тип EntityRef<Customer>, свойство Customer возвращает тип Customer, а не EntityRef<Customer>. Это важно, потому что любая ссылка в запросе на тип EntityRef<T> не будет транслироваться в SQL.

Сложности, связанные с EntitySet<T>

В то время как общедоступные свойства для приватных членов класса типа EntityRef<T> должны возвращать тип T вместо EntityRef<T>, то же самое нельзя сказать об общедоступных свойствах для приватных членов класса типа EntitySet<T>. Рассмотрим код, сгенерированный SQLMetal для члена приватного класса типа EntitySet<T>.

Приватный член класса EntitySet<T> и его свойство

```
[Table(Name="dbo.Customers")]
public partial class Customer : INotifyPropertyChanging, INotifyPropertyChanged
{
  ...
  private EntitySet<Order> _Orders;
  ...
  [Association(Name="FK_Orders_Customers", Storage="_Orders", OtherKey="CustomerID",
    DeleteRule="NO ACTION")]
```

```
    public EntitySet<Order> Orders
    {
      get
      {
        return this._Orders;
      }
      set
      {
        this._Orders.Assign(value);
      }
    }
    ...
  }
```

Как видите, свойство возвращает тип EntitySet<Order> — тот же, который имеет и приватный член класса. Поскольку EntitySet<T> реализует интерфейс ICollection<T>, возможно, стоило бы сделать так, чтобы свойство возвращало тип ICollection<T>, если необходимо скрыть детали реализации.

Другая сложность, о которой нужно помнить при написании собственных сущностных классов, заключается в том, что при написании общедоступного метода set для свойства EntitySet<T> должен использоваться метод Assign вместо простого присваивания переданного значения члену класса EntitySet<T>. Это позволит сущностному объекту продолжать применять исходную коллекцию ассоциированных сущностных объектов, поскольку коллекция может уже отслеживаться службой слежения за изменениями объекта DataContext.

Если еще раз взглянуть на предыдущий пример кода, то можно увидеть, что вместо присваивания переменной-члену this._Orders значения value в нем вызывается метод Assign.

Атрибуты сущностных классов и свойства атрибутов

Сущностные классы определяются атрибутами и свойствами атрибутов, которые отображают такой класс на таблицу базы данных, а его свойства — на столбцы таблицы базы данных. Атрибуты определяют наличие отображения, а свойства атрибутов указывают, как оно осуществляется. Например, есть атрибут Table, определяющий, что класс отображается на таблицу, но есть также его свойство Name, указывающее имя таблицы базы данных, на которую отображается класс.

Нет лучшего способа разобраться в атрибутах, их свойствах и в том, как они работают, чем изучить атрибуты, сгенерированные экспертами. Поэтому давайте проанализируем сущностный объект Customer, сгенерированный SQLMetal.

Ниже приведена часть сущностного класса Customer.

Часть сгенерированного SQLMetal сущностного класса Customer

```
[Table(Name="dbo.Customers")]
public partial class Customer : INotifyPropertyChanging, INotifyPropertyChanged
{
  ...
  [Column(Storage="_CustomerID", DbType="NChar(5) NOT NULL", CanBeNull=false,
    IsPrimaryKey=true)]
  public string CustomerID
  {
    get
    {
      return this._CustomerID;
    }
```

```
    set
    {
      if ((this._CustomerID != value))
      {
        this.OnCustomerIDChanging(value);
        this.SendPropertyChanging();
        this._CustomerID = value;
        this.SendPropertyChanged("CustomerID");
        this.OnCustomerIDChanged();
      }
    }
  }
  ...
  [Association(Name="FK_Orders_Customers", Storage="_Orders", OtherKey="CustomerID",
    DeleteRule="NO ACTION")]
  public EntitySet<Order> Orders
  {
    get
    {
      return this._Orders;
    }
    set
    {
      this._Orders.Assign(value);
    }
  }
  ...
}
```

Для краткости все части сущностного класса опущены за исключением тех, что содержат атрибуты LINQ to SQL. Также не показаны все излишние атрибуты.

А вот часть, содержащая хранимую процедуру и определенную пользователем функцию.

Часть, содержащая хранимую процедуру и определенную пользователем функцию

```
[Function(Name="dbo.Get Customer And Orders")]
[ResultType(typeof(GetCustomerAndOrdersResult1))]
[ResultType(typeof(GetCustomerAndOrdersResult2))]
public IMultipleResults GetCustomerAndOrders(
  [Parameter(Name="CustomerID", DbType="NChar(5)")] string customerID)
{
  ...
}
[Function(Name="dbo.MinUnitPriceByCategory", IsComposable=true)]
[return: Parameter(DbType="Money")]
public System.Nullable<decimal> MinUnitPriceByCategory(
  [Parameter(DbType="Int")] System.Nullable<int> categoryID)
{
  ...
}
```

В предыдущих фрагментах кода атрибуты выделены полужирным. Эти фрагменты кода приведены для обеспечения контекста для дискуссии об атрибутах.

Database

Атрибут `Database` указывает для класса-наследника `DataContext` имя по умолчанию отображенной базы данных, если это имя не задано в строке подключения при создании экземпляра `DataContext`. Если атрибут `Database` не указан и база данных не

приведена в строке подключения, то имя класса-наследника DataContext предполагается совпадающим с именем базы данных, к которой выполняется подключение.

Таким образом, для ясности: порядок приоритетов в определении имени базы данных, от высшего к низшему, выглядит следующим образом.

1. Информация подключения, представленная при создании экземпляра DataContext.
2. Имя базы данных, указанное в атрибуте Database.
3. Имя класса-наследника DataContext.

Ниже показана существенная часть сгенерированного SQLMetal класса-наследника DataContext по имени Northwind.

Из класса Northwind, сгенерированного SQLMetal

```
public partial class Northwind : System.Data.Linq.DataContext
{
```

Как видите, атрибут Database не задан в сгенерированном классе Northwind, производном от DataContext. Поскольку класс был сгенерирован Microsoft, можно предположить, что это сделано намеренно. Если вы собираетесь указывать атрибут Database и хотите, чтобы по умолчанию использовалась база данных по имени NorthwindTest, код должен выглядеть следующим образом.

Атрибут Database

```
[Database(Name="NorthwindTest")]
public partial class Northwind : System.Data.Linq.DataContext
{
```

Нет никаких причин избегать атрибута Database. Возможно, это потому, что если указать базу данных в строке соединения, то она переопределит имя класса DataContext и атрибут Database. Возможно, разработчики из Microsoft думали, что если имя базы данных в строке соединения не задано, то будет использоваться имя производного от DataContext класса, что вполне удовлетворительно.

Идею подключения сгенерированного класса-наследника DataContext к базе данных по умолчанию нельзя считать удачной. В этом случае может быть случайно запущено приложение, которое не было однозначно сконфигурировано, и оно подключится к базе данных по умолчанию. Возникает возможность для очень болезненной ошибки, которая рано или поздно случится. Фактически можно согласиться лишь с указанием атрибута Database с намеренно нелепым именем, чтобы предотвратить подключение к базе данных по умолчанию. Возможно, это должно выглядеть примерно так, как показано ниже.

Класс-наследник DataContext, допускающий в высшей степени маловероятное подключение к базе данных по умолчанию

```
[Database(Name=" goopeygobezileywag ")]
public partial class Northwind : System.Data.Linq.DataContext
{
```

И тогда подключения ни к какой базе данных не случится, если только она не будет указана в строке соединения, передаваемой DataContext во время создания его экземпляра.

Name (типа string)

Свойство Name атрибута Database — это строка, указывающая имя базы данных, с которой нужно устанавливать соединение, если имя базы не задано явно в инфор-

мации подключения при создании экземпляра класса-наследника DataContext. Если свойство атрибута Name опущено, а также не указано имя базы данных в информации подключения, тогда имя класса-наследника DataContext служит именем базы данных, к которой необходимо подключиться.

Table

Атрибут Table задает, в какой таблице базы данных будут сохраняться экземпляры сущностного класса. Имя сущностного класса не обязательно должно совпадать с именем таблицы. Ниже приведена часть сущностного класса, содержащая этот атрибут.

Атрибут Table

```
[Table(Name="dbo.Customers")]
public partial class Customer : INotifyPropertyChanging, INotifyPropertyChanged
{
```

Обратите внимание, что в атрибуте Table указывается имя таблицы базы данных в свойстве Name. Если имя сущностного класса совпадает с именем таблицы базы данных, свойство атрибута Name может быть опущено, т.к. имя класса станет по умолчанию именем таблицы, на которую он отображается.

В рассматриваемом примере, поскольку при использовании SQLMetal для генерации сущностных классов Northwind была задана опция множественного числа, имя таблицы базы данных — Customers — преобразуется в форму единственного числа — Customer, которая и является именем класса. Поскольку имя класса не совпадает с именем таблицы базы данных, атрибут Name должен быть указан.

Name (типа string)

Свойство Name атрибута Table — строка, определяющая имя таблицы, на которую отображается данный сущностный класс. Если свойство атрибута Name не задано, то имя сущностного класса по умолчанию будет отображено на одноименную с ним таблицу базы данных.

Column

Атрибут Column указывает на то, что свойство сущностного класса отображается на поле базы данных. Ниже показана иллюстрирующая его часть сущностного класса.

Атрибут Column

```
Private string _CustomerID;
...
[Column(Storage="_CustomerID", DbType="NChar(5) NOT NULL", CanBeNull=false,
   IsPrimaryKey=true)]
public string CustomerID
{
```

В рассматриваемом примере, поскольку указан атрибут Storage, LINQ to SQL сможет непосредственно обращаться к переменной-члену _CustomerID, минуя общедоступное средство доступа CustomerID. Если свойство атрибута Storage не задано, будет использовано средство доступа. Это может быть удобно для обхода специальной логики, которая может быть зашита в общедоступные средства доступа.

Тип базы данных для этого поля определен в атрибуте DbType как NCHAR длиной в пять символов. Поскольку атрибут CanBeNull указан со значением false, значение этого поля в базе данных не может быть NULL, а поскольку присутствует атрибут IsPrimaryKey со значением true, это поле представляет собой в таблице столбец идентичности.

Необязательно каждое свойство сущностного класса отображать на базу данных. Могут существовать свойства времени выполнения, которые не должны сохраняться в базе данных, и это вполне нормально. Для таких свойств просто не указывается атрибут `Column`.

Также могут быть хранимые столбцы, доступные только для чтения. Это обеспечивается отображением столбца и указанием свойства атрибута `Storage` на приватную переменную-член, без реализации метода `set` для свойства класса. `DataContext` может иметь доступ к приватному члену, но поскольку у данного свойства сущностного класса нет метода `set`, никто не может его изменять.

`AutoSync` (типа перечисления `AutoSync`)

Свойство атрибута `AutoSync` — это перечислимый тип `AutoSync`, которое инструктирует исполняющую систему о необходимости извлекать значение отображаемого столбца сразу после операции вставки или обновления записи в базе данных. Попробуйте догадаться, какое именно значение используется по умолчанию. Согласно документации Microsoft, поведение по умолчанию — `Never`.

Установка атрибута свойства переопределяется, когда или свойство `IsDbGenerated`, или свойство `IsVersion` установлено в `true`.

`CanBeNull` (типа `bool`)

Булевское свойство атрибута `CanBeNull` определяет допустимость значения `NULL` для отображаемого столбца базы данных. По умолчанию это свойство атрибута равно `true`.

`DbType` (типа `string`)

Свойство атрибута `DbType` представляет собой строку, определяющую тип столбца базы данных, на который отображается данное свойство сущностного класса. Если свойство `DbType` не указано, то тип столбца будет выведен из типа данных свойства сущностного класса. Это свойство атрибута используется для определения столбца только в случае вызова метода `CreateDatabase`.

`Expression` (типа `string`)

Свойство атрибута `Expression` представляет собой строку, определяющую вычисляемый столбец базы данных. Он используется только в случае вызова метода `CreateDatabase`. По умолчанию имеет значение `String.Empty`.

`IsDbGenerated` (типа `bool`)

Булевское свойство атрибута `IsDbGenerated` указывает на то, что столбец таблицы базы данных, на который отображается свойство класса, генерируется автоматически базой данных. Если первичный ключ задан со свойством атрибута `IsDbGenerated`, равным `true`, то свойство атрибута `DbType` должно быть установлено в `IDENTITY`.

Свойство класса, у которого значение свойства атрибута `IsDbGenerated` установлено в `true`, немедленно синхронизируется после того, как запись вставлена в базу данных, независимо от установки атрибута `AutoSync`, и синхронизированное значение свойства класса будет видимым в свойстве класса после успешного завершения метода `SubmitChanges`. Значение этого атрибута по умолчанию — `false`.

`IsDiscriminator` (типа `bool`)

Свойство `IsDiscriminator` атрибута хранит булевское значение, указывающее, что отображаемое свойство сущностного класса — это такое свойство, которое хранит значение дискриминатора для наследования сущностного класса. По умолчанию это свойство атрибута равно `false`. За дополнительной информацией на эту тему обращай-

тесь в раздел об атрибуте `InheritanceMapping` далее в этой главе, а также в раздел "Наследование сущностных классов" главы 18.

IsPrimaryKey (типа bool)

Свойство `IsPrimaryKey` атрибута хранит булевское значение, указывающее на то, что столбец базы данных, на который отображается данное свойство класса, является частью первичного ключа. В рассматриваемом случае все отображенные столбцы базы являются частями составного первичного ключа. Чтобы сущностный объект был обновляемым, по крайней мере, одно свойство сущностного класса должно иметь свойство атрибута `IsPrimaryKey`, установленное в `true`. В противном случае сущностный объект, отображенный на эту таблицу, будет доступен только для чтения. По умолчанию это свойство атрибута равно `false`.

IsVersion (типа bool)

Свойство `IsVersion` атрибута хранит булевское значение, указывающее, что отображенный столбец базы данных является либо номером версии, либо временной меткой, которая представляет информацию о версии для записи. За счет установки свойства атрибута `IsVersion` в `true` отображенный столбец базы данных будет инкрементирован, если это номер версии, и обновлен, если это временная метка, при всяком обновлении записи таблицы базы данных.

Свойство класса, атрибут `IsVersion` которого установлен в `true`, будет немедленно синхронизирован после вставки записи либо после ее обновления, независимо от установки атрибута `AutoSync`, и синхронизированное значение свойства класса будет видимо в свойстве класса немедленно после успешного завершения метода `SubmitChanges`. По умолчанию это свойство атрибута равно `false`.

Name (типа string)

Свойство `Name` атрибута `Column` — это строка, указывающая имя столбца таблицы, на который отображается это свойство класса. Если свойство `Name` атрибута опущено, то имя свойства класса будет отображено по умолчанию на одноименный столбец базы данных.

Storage (типа string)

Свойство `Storage` атрибута — это строка, указывающая приватную переменную-член, в которой хранится значение свойства сущностного класса. Это позволяет LINQ to SQL миновать общедоступное средство доступа свойства и содержащуюся в нем бизнес-логику, и позволяет получить непосредственный доступ к приватной переменной-члену. Если свойство `Storage` атрибута опущено, то по умолчанию будет использовано общедоступное средство доступа свойства.

UpdateCheck (типа перечисления UpdateCheck)

Свойство атрибута `UpdateCheck` — это перечислимый тип (`enum`) `UpdateCheck`, который управляет поведением определения оптимистического обнаружения параллелизма для свойства класса и отображаемого на него столбца базы данных, если ни одно из отображаемых свойств сущностного класса не имеет свойства атрибута `IsVersion`, установленного в `true`. Три допустимых значения: `UpdateCheck.Always`, `UpdateCheck.WhenChanged` и `UpdateCheck.Never`. Если ни одно из свойств сущностного класса не имеет свойства атрибута `IsVersion`, установленного в `true`, то значение свойства `UpdateCheck` атрибута по умолчанию будет `Always`. Дополнительные сведения об этом свойстве атрибута и его эффекте можно найти в главе 17.

Association

Атрибут Association используется для определения отношений между двумя таблицами, таких как отношение первичного и внешнего ключа. В этом контексте сущность, чья отображенная таблица имеет первичный ключ, является родительской, а сущность, чья отображенная таблица содержит внешний ключ — дочерней. Рассмотрим важные части двух сущностных классов, содержащих ассоциацию.

Ассоциация от родительского (Customer) сущностного класса

```
[Association(Name="FK_Orders_Customers", Storage="_Orders",
OtherKey="CustomerID",
   DeleteRule="NO ACTION")]
public EntitySet<Order> Orders
{
```

Ассоциация от дочернего (Order) сущностного класса

```
[Association(Name="FK_Orders_Customers", Storage="_Customer",
ThisKey="CustomerID",
   IsForeignKey=true)]
public Customer Customer
{
```

В этом обсуждении атрибута Association и его свойств используется сущностный класс Customer в качестве примера родительского элемента и сущностный класс Order в качестве примера дочернего элемента. Таким образом, будут представлены важные атрибуты Association, которые существуют в обоих сущностных классах — Customer и Order.

Если говорить о свойствах атрибута Association, то некоторые его свойства относятся к классу, в котором имеется атрибут Association, а другие свойства — к другому ассоциированному сущностному классу. В этом контексте класс, в котором существует атрибут Association, можно назвать *исходным* классом, а другой ассоциированный сущностный класс — *целевым* классом. Поэтому, если речь идет о свойствах атрибута Association, указанного в сущностном классе Customer, то сущностный класс Customer является исходным классом, а сущностный класс Order — целевым классом. Если говорится о свойствах атрибута Association, указанных в классе Order, то исходным классом выступает Order, а целевым — Customer.

Атрибут Association определяет, что исходный сущностный класс Customer имеет отношение с целевым сущностным классом Order.

В предшествующих примерах свойство Name атрибута указано для определения имени отношения. Значение этого свойства атрибута соответствует имени ограничения внешнего ключа в базе данных и будет использовано для создания этого ограничения при вызове метода CreateDatabase. Свойство Storage атрибута также присутствует. Оно позволяет LINQ to SQL обойти общедоступное средство доступа (accessor), чтобы получить доступ к значению свойства сущностного класса.

В варианте ассоциации первичного ключа к внешнему сущностный класс, являющийся родителем в отношении, будет хранить ссылку на дочерний сущностный класс в коллекции EntitySet<T>, поскольку дочерних классов может быть много. Дочерний сущностный класс будет хранить ссылку на родительский сущностный класс в EntityRef<T>, поскольку он только один. За дополнительными сведениями об ассоциациях и их характеристиках обращайтесь к разделам “EntitySet<T>” и “EntityRef<T>” далее в этой главе, а также к разделу “Немедленная загрузка с помощью класса DataLoadOptions” главы 14.

DeleteOnNull (типа bool)

Свойство DeleteOnNull атрибута принимает булевское значение, которое, будучи установленным в true, указывает, что сущностный объект с дочерней стороны ассоциации должен быть удален в случае, если ссылка на него у родителя устанавливается в null.

Значение этого свойства атрибута определяется SQLMetal, если в базе данных задано правило удаления "CASCADE" для ограничения внешнего ключа и столбец внешнего ключа не допускает null.

DeleteRule (типа string)

Свойство DeleteRule атрибута принимает строку, которая указывает правило удаления (Delete Rule) для ограничения внешнего ключа. Оно используется только LINQ to SQL, когда создается ограничение в базе данных методом CreateDatabase.

Допустимые значения: "NO ACTION", "CASCADE", "SET NULL" и "SET DEFAULT". Определение всех этих значений ищите в документации по SQL Server.

IsForeignKey (типа bool)

Свойство IsForeignKey атрибута хранит булевское значение, которое, будучи установленным в true, говорит о том, что сущностный класс исходной стороны ассоциации содержит внешний ключ; т.е. является дочерней стороной этой ассоциации. По умолчанию это свойство атрибута установлено в false.

В примерах атрибута Association, показанных ранее для сущностных классов Customer и Order, поскольку данный атрибут, указанный в сущностном классе Order, содержит свойство IsForeignKey атрибута, установленное в true, класс Order является дочерним в данном отношении.

IsUnique (типа bool)

Свойство IsUnique атрибута хранит булевское значение, которое, будучи установленным в true, указывает наличие ограничения уникальности на внешнем ключе, т.е. задает отношение “один к одному” между двумя сущностными классами. По умолчанию это свойство атрибута установлено в false.

Name (типа string)

Свойство Name атрибута Association — это строка, указывающая имя ограничения внешнего ключа. Она используется для создания ограничения внешнего ключа при вызове метода CreateDatabase. Также оно будет применяться для различия нескольких отношений между одними и теми же сущностями. В этом случае, если обе стороны отношения определяют имя, оно должно быть одинаковым.

Если нет нескольких отношений между одними и теми же сущностными классами и не вызывается метод CreateDatabase, это свойство атрибута не обязательно. Для этого свойства значение по умолчанию не предусмотрено.

OtherKey (типа string)

Свойство OtherKey атрибута Association — это строка, хранящая разделенный запятыми список всех свойств целевого сущностного класса, составляющих ключ, будь то внешний или первичный, в зависимости от того, какую сторону отношения занимает целевая сущность. Если это свойство атрибута не указано, то по умолчанию используются члены первичного ключа целевого сущностного класса.

Важно понимать, что атрибут Association, указанный на каждой стороне отношения ассоциации — Customer и Order — определяет, где находятся ключи каждой из сторон. Атрибут Association, заданный в сущностном классе Customer, говорит

о том, какие свойства сущностных классов Customer и Order составляют ключ для отношения.

Иногда может показаться, что каждая сторона всегда указывает местоположения ключа обеих сторон. Поскольку обычно на родительской стороне отношения используется первичный ключ таблицы, свойство ThisKey атрибута не должно быть указано, поскольку первичный ключ используется по умолчанию. А на дочерней стороне свойство OtherKey атрибута не обязано указываться, т.к. по умолчанию применяется первичный ключ родителя. Таким образом, часто можно видеть свойство OtherKey определенным только на родительской стороне, а свойство ThisKey атрибута — только на дочерней стороне. Но благодаря значениям по умолчанию, и родительский, и дочерний элементы знают ключи на обеих сторонах.

Storage (типа string)

Свойство Storage атрибута Association — это строка, указывающая приватную переменную-член, в которой хранится значение свойства сущностного класса. Это позволяет LINQ to SQL миновать общедоступное средство доступа и непосредственно обратиться к приватной переменной-члену, пропуская любую бизнес-логику из средств доступа. Если свойство Storage атрибута не задано, то по умолчанию используются общедоступные средства доступа данного свойства.

В Microsoft рекомендуют, чтобы оба члена отношения ассоциации были свойствами сущностного класса с отдельными переменными-членами для хранения данных с указанными свойствами Storage атрибута отношения.

ThisKey (типа string)

Свойство ThisKey атрибута представляет собой строку, которая содержит разделенный запятыми список всех свойств сущностного класса, составляющих ключ, будь то внешний или первичный, в зависимости от того, какую сторону отношения представляет исходная сущность, что определяется свойством IsForeignKey того же атрибута. Если свойство ThisKey атрибута не указано, то члены первичного ключа исходного сущностного класса используются по умолчанию.

Поскольку в ранее приведенном примере атрибута Association для сущностного класса Order имеется свойство IsForeignKey, известно, что сущностный класс Customer представляет родительскую сторону отношения — ту, что содержит первичный ключ. Поскольку атрибут Association не указывает свойства ThisKey, мы знаем, что значение первичного ключа таблицы Customer становится внешним ключом в ассоциированной таблице Orders.

Так как атрибут Association, показанный ранее для сущностного класса Order, определяет атрибут IsForeignKey со значением true, известно, что таблица Orders будет стороной ассоциации, содержащей внешний ключ. И, поскольку атрибут Association задает значение свойства ThisKey этого атрибута равным CustomerID, мы знаем, что столбец CustomerID таблицы Orders будет местом хранения внешнего ключа.

Важно понимать, что атрибут Association, указанный на каждой стороне отношения ассоциации — Customer и Order — определяет местонахождение ключей обеих сторон. Атрибут Association, заданный в сущностном классе Customer, указывает, какие свойства этого класса и класса Orders составляют ключ для отношения. Аналогично атрибут Association, определенный в классе Orders, указывает, какие свойства этого класса и класса Customer составляют ключ отношения.

Иногда может показаться, что каждая сторона всегда указывает местоположения ключа обеих сторон. Поскольку обычно на родительской стороне отношения используется первичный ключ таблицы, свойство ThisKey атрибута не обязано быть определено, поскольку первичный ключ применяется по умолчанию. А на дочерней стороне

свойство OtherKey атрибута не обязано задаваться, т.к. по умолчанию используется первичный ключ родителя. Таким образом, часто можно видеть свойство OtherKey указанным только на родительской стороне, а свойство ThisKey атрибута — только на дочерней стороне. Но благодаря значениям по умолчанию, и родительский, и дочерний элементы знают ключи на обеих сторонах.

Function

Атрибут Function определяет, что метод класса вызывает хранимую процедуру либо пользовательскую функцию со скалярным или табличным значением возврата. Ниже приведена важная часть класса, производного от DataContext, которая связана с хранимой процедурой.

Атрибут Function, отображающий метод на хранимую процедуру в базе данных Northwind

```
[Function(Name="dbo.Get Customer And Orders")]
[ResultType(typeof(GetCustomerAndOrdersResult1))]
[ResultType(typeof(GetCustomerAndOrdersResult2))]

public IMultipleResults GetCustomerAndOrders(
    [Parameter(Name="CustomerID", DbType="NChar(5)")] string customerID)
{
    ...
}
```

Здесь видно, что есть метод по имени GetCustomerAndOrders, который вызовет хранимую процедуру по имени Get Customer And Orders. Мы знаем, что этот метод отображается на хранимую процедуру, а не на определяемую пользователем функцию, поскольку свойство IsComposable атрибута не указано и потому по умолчанию равно false, что отображает метод на хранимую процедуру. Также можно видеть, что он возвращает множественные формы результатов, потому что указано два атрибута ResultTypes.

Написать класс-наследник DataContext, чтобы он мог вызывать хранимую процедуру — не такая тривиальная задача, как отображение сущностного класса на таблицу. В дополнение к соответствующим атрибутам понадобится также вызывать соответствующую версию метода ExecuteMethodCall класса DataContext. Этот метод рассматривается в главе 16.

Конечно, как правило, это необходимо только при написании собственного класса-наследника DataContext, поскольку это сделают SQLMetal и Object Relational Designer.

Существенная часть класса-наследника DataContext, ссылающаяся на определенную пользователем функцию, представлена ниже.

Атрибут Function, отображающий метод на определенную пользователем функцию в базе данных Northwind

```
[Function(Name="dbo.MinUnitPriceByCategory", IsComposable=true)]
[return: Parameter(DbType="Money")]

public System.Nullable<decimal> MinUnitPriceByCategory(
    [Parameter(DbType="Int")] System.Nullable<int> categoryID)
{
  ...
}
```

Здесь видно, что имеется метод по имени MinUnitPriceByCategory, который будет вызывать определенную пользователем функцию по имени MinUnitPriceByCategory. Известно, что метод отображается именно на определенную пользователем функцию, а не на хранимую процедуру, поскольку свойство IsComposable атрибута установлено в true.

Также можно видеть по атрибуту return, что определенная пользователем функция вернет значение типа Money.

Написание наследника класса DataContext так, чтобы он мог вызывать определяемые пользователем функции, не настолько просто, как отображение сущностного класса на таблицу. В дополнение к соответствующим атрибутам потребуется также вызывать ExecuteMethodCall класса DataContext для функций, возвращающих скалярные значения, или же метод CreateMethodCallQuery для функций, возвращающих табличные значения. Вы также прочтете об этих методах в главе 16.

Конечно, как правило, это необходимо при написании собственного класса-наследника DataContext, потому что SQLMetal и Object Relational Designer делают это сами.

IsComposable (типа bool)

Свойство IsComposable атрибута содержит булевское значение, указывающее на то, что отображающий метод вызывает хранимую процедуру или определенную пользователем функцию. Если значение IsComposable равно true, метод отображается на определяемую пользователем функцию. Если значение IsComposable равно true, метод отображается на хранимую процедуру. Это свойство атрибута по умолчанию имеет значение false, если не указано, т.е. метод, отображаемый атрибутом Function, по умолчанию отображается на хранимую процедуру, если свойство IsComposable атрибута не задано.

Name (типа string)

Свойство Name атрибута содержит строку, указывающую действительное имя хранимой процедуры или определяемой пользователем функции в базе данных. Если свойство Name атрибута не задано, то имя хранимой процедуры или пользовательской функции предполагается совпадающим с именем метода.

return

Атрибут return используется для указания возвращаемого типа данных из хранимой процедуры или определяемой пользователем функции. Обычно содержит атрибут Parameter.

Атрибут return для класса Northwind

```
[Function(Name="dbo.MinUnitPriceByCategory", IsComposable=true)]
[return: Parameter(DbType="Money")]
public System.Nullable<decimal> MinUnitPriceByCategory(
    [Parameter(DbType="Int")] System.Nullable<int> categoryID)
{
  ...
}
```

В приведенном коде видно, что определенная пользователем функция вернет значение типа Money, на что указывает атрибут return и свойство DbType встроенного атрибута Parameter.

ResultType

Атрибут ResultType отображает тип данных, возвращенный хранимой процедурой, на класс .NET, в котором сохраняются возвращенные данные. Хранимые процедуры, возвращающие множественные формы, указывают несколько атрибутов ResultType в соответствующем порядке.

Атрибуты ResultType класса Northwind

```
[Function(Name="dbo.Get Customer And Orders")]
[ResultType(typeof(GetCustomerAndOrdersResult1))]
```

```
[ResultType(typeof(GetCustomerAndOrdersResult2))]
public IMultipleResults GetCustomerAndOrders(
   [Parameter(Name="CustomerID", DbType="NChar(5)")] string customerID)
{
  ...
}
```

В приведенном коде видно, что хранимая процедура, на которую отображается этот метод, сначала возвратит форму типа `GetCustomerAndOrdersResult1`, за которой следует форма типа `GetCustomerAndOrdersResult2`.

Parameter

Атрибут `Parameter` отображает параметр метода на параметр хранимой процедуры или пользовательской функции базы данных. Ниже приведена существенная часть кода класса-наследника `DataContext`.

Атрибут `Parameter` из класса Northwind

```
[Function(Name="dbo.Get Customer And Orders")]
[ResultType(typeof(GetCustomerAndOrdersResult1))]
[ResultType(typeof(GetCustomerAndOrdersResult2))]
public IMultipleResults GetCustomerAndOrders(
   [Parameter(Name="CustomerID", DbType="NChar(5)")] string customerID)
{
  ...
}
```

Здесь видно, что метод `GetCustomerAndOrders`, отображаемый на хранимую процедуру базы данных по имени `Get Customer And Orders`, передает хранимой процедуре параметр типа `NChar(5)`.

DbType (типа string)

Свойство `DbType` атрибута — это строка, указывающая тип данных базы и модификаторы параметра хранимой процедуры или определенной пользователем функции.

Name (типа string)

Свойство `Name` атрибута содержит строку, указывающую действительное имя параметра хранимой процедуры или определенной пользователем функции. Если свойство `Name` атрибута не задано, то имя хранимой процедуры или пользовательской функции предполагается совпадающим с именем параметра метода.

InheritanceMapping

Атрибут `InheritanceMapping` используется для отображения *кода дискриминатора* (классификатора) на базовый класс или подкласс базового класса. Код дискриминатора — это значение столбца сущностного класса, указанного в качестве дискриминатора, который определен как свойство сущностного класса, чье свойство `IsDiscriminator` атрибута установлено в `true`.

Например, рассмотрим следующий атрибут `InheritanceMapping`.

Атрибут `InheritanceMapping`

```
[InheritanceMapping(Code = "G", Type = typeof(Shape), IsDefault = true)]
```

Приведенный атрибут `InheritanceMapping` определяет, что если запись базы данных имеет значение `"G"` в столбце дискриминатора, т.е. ее код дискриминатора — `"G"`, то нужно создать запись как объект `Shape` с использованием класса `Shape`. Поскольку свойство `IsDefault` атрибута установлено в `true`, если код дискриминатора записи не

соответствует ни одному значению Code атрибута InheritanceMapping, эта запись будет создана в виде объекта класса Shape.

Чтобы использовать отображение наследования, при объявлении сущностного класса одному из его свойств назначается свойство IsDiscriminator атрибута Column, установленное в true. Это значит, что значение этого столбца будет определять по дискриминатору, какого класса экземпляр — базового или одного из его подклассов — хранит запись таблицы. Атрибут InheritanceMapping указан в базовом классе для каждого из подклассов, а также для самого базового класса. Из всех этих атрибутов InheritanceMapping один и только один должен иметь свойство IsDefault со значением true. Также эта запись таблицы базы данных, дискриминатор которой не соответствует ни одному из кодов дискриминатора, указанных в атрибутах InheritanceMapping, может стать экземпляром этого класса. Наверно, чаще всего атрибут InheritanceMapping задают как атрибут InheritanceMapping по умолчанию.

Опять-таки, все атрибуты InheritanceMapping, указанные на одном из базовых классов, лишь ассоциируют код дискриминатора для базового класса или одного из его подклассов.

Поскольку база данных Northwind не содержит никаких таблиц, используемых таким образом, рассмотрим три класса-примера.

Некоторые примеры классов, демонстрирующие отображение наследования

```
[Table]
[InheritanceMapping(Code = "G", Type = typeof(Shape), IsDefault = true)]
[InheritanceMapping(Code = "S", Type = typeof(Square))]
[InheritanceMapping(Code = "R", Type = typeof(Rectangle))]
public class Shape
{
  [Column(IsPrimaryKey = true, IsDbGenerated = true,
    DbType = "Int NOT NULL IDENTITY")]
  public int Id;

  [Column(IsDiscriminator = true, DbType = "NVarChar(2)")]
  public string ShapeCode;

  [Column(DbType = "Int")]
  public int StartingX;
  [Column(DbType = "Int")]
  public int StartingY;
}
public class Square : Shape
{
  [Column(DbType = "Int")]
  public int Width;
}
public class Rectangle : Square
{
  [Column(DbType = "Int")]
  public int Length;
}
```

Здесь видно, что класс Shape отображен на таблицу, и поскольку свойство атрибута Name не указано, класс Shape будет отображен по умолчанию на таблицу по имени Shape.

Кроме того, заданы три атрибута InheritanceMapping. Первый определяет, что если значение столбца дискриминатора записи таблицы базы данных Shape равно "G", то из записи должен быть создан экземпляр объекта Shape с использованием класса

Shape. В примере "G" выбрано для обозначения *generic* (обобщения), в том смысле, что это обобщенная неизвестная фигура (shape). Поскольку в классе Shape есть свойство ShapeCode, которое является дискриминатором, в том смысле, что оно имеет свойство IsDiscriminator атрибута, установленное в true, если запись имеет значение ShapeCode, равное "G", из этой записи будет создан объект Shape.

Также видно, что первый атрибут InheritanceMapping имеет свойство IsDefault атрибута, установленное в true, так что если значение столбца ShapeCode записи Shape не соответствует ни одному из указанных кодов дискриминатора — "G", "S" и "R", — то используется отображение по умолчанию, а запись станет экземпляром объекта Shape.

Второй атрибут InheritanceMapping ассоциирует код дискриминатора "S" с классом Square. Таким образом, если значение столбца ShapeCode записи Shape равно "S", то из этой записи будет создан объект Square.

Третий атрибут InheritanceMapping ассоциирует код дискриминатора "R" с классом Rectangle. Поэтому если значение столбца ShapeCode записи Shape равно "R", то запись станет экземпляром объекта Rectangle.

Любая запись с ShapeCode, отличающимся от указанного, превратится в экземпляр объекта Shape, потому что Shape — класс по умолчанию, как определено в свойстве IsDefault атрибута.

На заметку! Отображение наследования вместе с примерами применения рассматривается в главе 18.

Code (типа object)

Свойство Code атрибута задает код дискриминатора для отображения на специфицированный класс, который будет указан свойством Type атрибута.

IsDefault (типа bool)

Свойство атрибута IsDefault содержит булевское значение, которое определяет, какой атрибут InheritanceMapping должен использоваться, если столбец дискриминатора записи таблицы базы данных не соответствует ни одному коду дискриминатора, указанному в любом из атрибутов InheritanceMapping.

Type (типа Type)

Свойство Type атрибута указывает тип класса для создания экземпляра, представляющего запись, когда столбец дискриминатора соответствует отображенному коду дискриминатора.

Совместимость типов данных

Некоторые атрибуты сущностного класса имеют свойство атрибута DbType, в котором можно задавать тип данных столбца таблицы. Это свойство атрибута используется только при создании базы данных методом CreateDatabase. Поскольку отображение между типами данных .NET и типами данных SQL Server не строится по схеме “один к одному”, нужно указать свойство DbType атрибута, если планируется вызывать метод CreateDatabase.

Поскольку типы данных общеязыковой исполняющей среды .NET (Common Language Runtime — CLR), используемые в коде LINQ, не совпадают с типами данных, применяемыми базой данных, следует обратиться к документации MSDN, касающейся отображения типов между SQL и CLR (LINQ to SQL). В этой документации приведена матрица, которая определяет поведение при преобразовании между типа данных CLR и типа данных SQL. Следует иметь в виду, что некоторые преобразования типов данных не

поддерживаются, а другие могут привести к потере данных, в зависимости от типов, участвующих в преобразовании и его направления.

Однако в большинстве случаев должно удовлетворить преобразование, и оно не составит проблемы. При написании примеров для глав, посвященных LINQ to SQL, никогда не возникало проблем, вызванных преобразованием типов данных. Конечно, нужно придерживаться здравого смысла. Если вы попытаетесь отображать очевидно несовместимые типы, такие как числовые типы данных .NET на символьный тип данных SQL, то неизбежно столкнетесь с некоторыми проблемами.

XML-схема внешнего файла отображения

Как говорилось в разделе главы 13, посвященном SQLMetal, можно не только отображать сущностные классы на базу данных, но также использовать внешний XML-файл отображения. О том, как применять внешний XML-файл отображения вы узнаете при описании конструкторов для класса `DataContext` в главе 16.

В главе 13 упоминалось, что простейший способ получить внешний XML-файл отображения заключается в вызове программы SQLMetal с указанием опции `/map` — в этом случае он будет сгенерирован автоматически. Однако если вы намерены создавать файл отображения вручную, следует знать его схему.

Схема внешнего отображения описана в разделе “External Mapping Reference (LINQ to SQL)” документации MSDN.

Сравнение проекций на сущностные и на несущностные классы

При выполнении запросов LINQ to SQL есть два выбора для проекции возвращенных результатов. Результаты можно спроецировать на сущностный класс или на несущностный класс, который может быть именованным или анонимным. Между проецированием на сущностный и несущностный класс имеется существенное отличие.

При проецировании на сущностный класс он использует преимущества служб отслеживания идентичности объектов `DataContext`, слежения за изменениями и обработки изменений. Это значит, что изменять несущностные классы и сохранять их посредством LINQ to SQL нельзя. Это оправдано, поскольку такой класс не будет иметь необходимых атрибутов или файла отображения класса на базу данных. И с другой стороны, если он имеет атрибуты или файл отображения, то он по определению является сущностным классом.

Ниже приведен пример запроса, проецируемого на сущностный класс.

Проецирование на сущностный класс представляет службы `DataContext`

```
IEnumerable<Customer> custs = from c in db.Customers
                              select c;
```

После этого запроса можно было бы вносить изменения в любой из сущностных объектов `Customer` в последовательности `custs` и сохранять их вызовом метода `SubmitChanges`. Далее показан пример запроса, проецируемого на несущностный класс.

Проецирование на несущностный класс не обеспечивает доступа к службам `DataContext`

```
var custs = from c in db.Customers
select new { Id = c.CustomerID, Name = c.ContactName };
```

Выполнив проецирование результата на анонимный класс, сохранить никакие внесенные изменения в каждый объект последовательности `custs` вызовом метода `SubmitChanges` не получится.

Ранее упоминалось о том, что есть одно специализированное исключение, имеющее отношение к получению преимуществ отслеживания идентичности и обработки изме-

нений при проецировании несущностных классов. Это исключение происходит, когда проецируемый класс содержит члены типа сущностных классов. Пример приведен в листинге 15.1.

Листинг 15.1. Проецирование на несущностный класс, содержащий сущностные классы

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
var cusorders = from o in db.Orders
                where o.Customer.CustomerID == "CONSH"
                orderby o.ShippedDate descending
                select new { Customer = o.Customer, Order = o };

// Захватить первый заказ.
Order firstOrder = cusorders.First().Order;

// Теперь сохранить страну поставки первого заказа,
// чтобы можно было восстановить ее позже.
string shipCountry = firstOrder.ShipCountry;
Console.WriteLine("Заказ первоначально поставлялся в {0}", shipCountry);

// Теперь изменить страну поставки с Великобритании на США.
firstOrder.ShipCountry = "USA";
db.SubmitChanges();

// Запрос для проверки того, была ли страна на самом деле изменена.
string country = (from o in db.Orders
                  where o.Customer.CustomerID == "CONSH"
                  orderby o.ShippedDate descending
                  select o.ShipCountry).FirstOrDefault<string>();
Console.WriteLine("Заказ теперь поставляется в {0}", country);

// Восстановить состояние базы данных, чтобы пример можно было запустить снова.
firstOrder.ShipCountry = shipCountry;
db.SubmitChanges();
```

В листинге 15.1 запрашиваются заказы, поступившие от заказчика `"CONSH"`. Возвращенные заказы проецируются на анонимный тип, содержащий `Customer` и каждый `Order`. Сам анонимный класс не принимает служб `DataContext`, таких как отслеживание идентичности, отслеживание и обработка изменений, но его компоненты типа `Customer` и `Order` делают это, будучи сущностными классами. Затем на результатах предыдущего выполняется другой запрос, чтобы получить первый `Order`. После этого сохраняется копия исходного свойства `ShipCountry` объекта `Order`, чтобы можно было восстановить его в конце примера, и это значение выводится на консоль. Затем `ShipCountry` в `Order` изменяется с сохранением изменения с помощью метода `SubmitChanges`. После этого `ShipCountry` для этого заказа снова запрашивается из базы данных и выводится на консоль, просто чтобы продемонстрировать, что в базе данных действительно произошло изменение. Это доказывает правильность работы метода `SubmitChanges`, а также тот факт, что компоненты сущностного класса анонимного типа получают в свое распоряжение службы объекта `DataContext`. Далее `ShipCountry` сбрасывается в исходное значение и сохраняется, чтобы пример можно было запустить опять, и никакие последующие примеры не пострадали.

Вот результат запуска программы из листинга 15.1:

```
Заказ первоначально поставлялся в UK
Заказ теперь поставляется в USA
```

В листинге 15.1 содержится пример, в котором результаты запроса проецируются на тип несущностного класса, но поскольку он включает в себя объекты сущностного класса, можно воспользоваться преимуществами отслеживания идентичности, а также отслеживания и обработки изменений, которые предоставлены классом `DataContext`.

Есть одно интересное замечание относительно приведенного выше кода. Вы заметите, что запрос, получающий ссылку на первый `Order`, выделен полужирным. Это сделано, чтобы привлечь внимание. Обратите внимание, что операция `First` вызывается перед выбором части интересующего элемента последовательности — члена `Order`. Это делается для повышения производительности, потому что чем быстрее можно сузить поле поиска, тем выше будет производительность.

При проецировании отдавайте предпочтение инициализации объектов перед параметризацией конструирования

Вы вольны проецировать результат на классы до окончания запроса для последующих операций запроса, но при этом отдавайте предпочтение инициализации объектов перед параметризованным конструированием. Чтобы понять почему, заглянем в листинг 15.2, который использует инициализацию объектов при проецировании.

Листинг 15.2. Проецирование с использованием инициализации объектов

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
db.Log = Console.Out;
var contacts = from c in db.Customers
               where c.City == "Buenos Aires"
               select new { Name = c.ContactName, Phone = c.Phone } into co
               orderby co.Name
               select co;
foreach (var contact in contacts)
{
  Console.WriteLine("{0} - {1}", contact.Name, contact.Phone);
}
```

Обратите внимание, что в листинге 15.2 результат проецируется на анонимный класс и с помощью инициализации объектов осуществляется наполнение создаваемых анонимных объектов. Ниже показан вывод кода из листинга 15.2:

```
SELECT [t0].[ContactName] AS [Name], [t0].[Phone]
FROM [dbo].[Customers] AS [t0]
WHERE [t0].[City] = @p0
ORDER BY [t0].[ContactName]
-- @p0: Input String (Size = 12; Prec = 0; Scale = 0) [Buenos Aires]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.20706.1

Patricio Simpson - (1) 135-5555
Sergio Gutiérrez - (1) 123-5555
Yvonne Moncada - (1) 135-5333
```

Здесь даже не интересует вывод результатов запроса. В действительности необходимо увидеть сгенерированный запрос SQL. Поэтому возникает вопрос, а зачем тогда нужен цикл `foreach`? Дело в том, что из-за отложенного выполнения без `foreach` запрос вообще бы не выполнился.

Существенная для обсуждения часть запроса LINQ to SQL заключена в конструкциях `select` и `orderby`. Запрос LINQ to SQL инструктируется о необходимости создания члена по имени `Name` в анонимном классе, который заполняется полем `ContactName` из таблицы `Customers`. Затем запросу указывается отсортировать результат по члену `Name` анонимного объекта, на который выполняется проекция. Объект `DataContext` содержит всю переданную ему информацию. Инициализация объекта эффективно отображает исходное поле `ContactName` из класса `Customer` на целевое поле `Name` анонимного класса, и объект `DataContext` отвечает за это отображение. На основании этой информации он способен понять, что на самом деле `Customers` сортируется по полю `ContactName`,

поэтому он может сгенерировать запрос SQL, который сделает это. Взглянув на сгенерированный SQL-запрос, можно убедиться, что именно это он и делает.

Теперь давайте посмотрим, что происходит, когда результат проецируется на именованный класс с использованием параметризованного конструирования. Первым делом, нужен именованный класс, в качестве которого используется приведенный ниже.

Именованный класс, используемый в листинге 15.3

```
class CustomerContact
{
  public string Name;
  public string Phone;
  public CustomerContact(string name, string phone)
  {
    Name = name;
    Phone = phone;
  }
}
```

Обратите внимание, что здесь есть единственный конструктор, принимающий два параметра — name и phone. Теперь взглянем на тот же код, что и в листинге 15.2, но с тем отличием, что в листинге 15.3 код будет модифицирован для проецирования на класс CustomerContact с использованием параметризованного конструктора.

Листинг 15.3. Проецирование с использованием параметризованного конструирования

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
db.Log = Console.Out;
var contacts = from c in db.Customers
               where c.City == "Buenos Aires"
               select new CustomerContact(c.ContactName, c.Phone) into co
               orderby co.Name
               select co;
foreach (var contact in contacts)
{
  Console.WriteLine("{0} - {1}", contact.Name, contact.Phone);
}
```

Опять-таки, обратите внимание на конструкции select и orderby. Как показано в листинге 15.3, вместо проецирования на анонимный класс осуществляется проекция на класс CustomerContact. А вместо инициализации созданных объектов применяется параметризованный конструктор. Этот код компилируется успешно, но что произойдет при запуске примера? Будет сгенерировано следующее исключение:

```
Unhandled Exception: System.NotSupportedException: The member
'LINQChapter15.CustomerContact.Name' has no supported translation to SQL...
```

Необработанное исключение: System.NotSupportedException: Член 'LINQChapter15.CustomerContact.Name' не имеет поддерживаемой трансляции в SQL...

Так что же произошло? Взглянув на предшествующий запрос LINQ to SQL, может возникнуть вопрос: откуда DataContext знает, какое поле класса Customer отображается на член CustomerContact.Name, по которому предпринимается попытка выполнить упорядочивание? В листинге 15.2, поскольку это передавалось в именах полей анонимного класса, известно, что исходным полем класса Customer было ContactName, а целевым полем анонимного класса — поле Name. В листинге 15.3 это отображение происходит не в запросе LINQ to SQL, а в конструкторе класса CustomerContact, о чем DataContext не имеет понятия. Поэтому он не знает, по какому полю исходного класса Customer следует выполнить упорядочивание при генерации оператора SQL. Отсюда и проблемы.

Однако использовать параметризованное конструирование вполне безопасно до тех пор, пока ничего в запросе проекции не ссылается на члены именованного класса, как показано в листинге 15.4.

Листинг 15.4. Проецирование с использованием параметризованного конструирования без ссылок на члены

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
db.Log = Console.Out;
var contacts = from c in db.Customers
               where c.City == "Buenos Aires"
               select new CustomerContact(c.ContactName, c.Phone);
foreach (var contact in contacts)
{
  Console.WriteLine("{0} - {1}", contact.Name, contact.Phone);
}
```

Поскольку в листинге 15.4 применяется синтаксис выражений запросов, и этот синтаксис требует, чтобы запрос заканчивался конструкцией `select`, вполне допускается использовать параметризованное конструирование с последней конструкции `select` запроса. Это можно делать потому, что после конструкции `select`, содержащей вызов параметризованного конструктора, который ссылается на именованные члены класса, ничего не следует. Вот результат работы программы из листинга 15.4:

```
SELECT [t0].[ContactName], [t0].[Phone]
FROM [dbo].[Customers] AS [t0]
WHERE [t0].[City] = @p0
-- @p0: Input String (Size = 12; Prec = 0; Scale = 0) [Buenos Aires]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.20706.1

Patricio Simpson - (1) 135-5555
Yvonne Moncada - (1) 135-5333
Sergio Gutiérrez - (1) 123-5555
```

Однако, поскольку использование стандартного точечного синтаксиса не требует, чтобы запрос оканчивался конструкцией `select`, небезопасно предполагать, что запрос будет работать просто потому, что проекция на именованный класс с применением параметризованного конструирования происходит в последней проекции. В листинге 15.5 содержится пример, использующий стандартный точечный синтаксис с последней проекцией, которая применяет параметризованное конструирование, но т.к. последующая часть запроса ссылается на члены именованного класса, запрос генерирует исключение.

Листинг 15.5. Проецирование с использованием параметризованного конструирования, ссылающееся на члены

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
db.Log = Console.Out;
var contacts = db.Customers.Where(c => c.City == "Buenos Aires").
               Select(c => new CustomerContact(c.ContactName, c.Phone)).
               OrderBy(c => c.Name);
foreach (var contact in contacts)
{
  Console.WriteLine("{0} - {1}", contact.Name, contact.Phone);
}
```

Запрос в листинге 15.5 очень похож на запрос из листинга 15.4 за исключением того, что здесь используется стандартный синтаксис точечной нотации вместо синтаксиса выражения запроса, а также операция `OrderBy` перемещается в конец запроса. В финальной проекции применяется параметризованное конструирование, но это не работает, потому что операция `OrderBy` ссылается на член именованного класса. Вот результат работы кода из листинга 15.5:

```
Unhandled Exception: System.NotSupportedException: The member 'LINQChapter15.
CustomerContact.Name' has no supported translation to SQL...
```

```
Необработанное исключение: System.NotSupportedException: Член
'LINQChapter15.CustomerContact.Name' не имеет поддерживаемой трансляции в SQL...
```

Из-за этих сложностей рекомендуется, где это возможно, вместо параметризованных конструкций применять инициализацию объектов.

Расширение сущностных классов с помощью частичных методов

О частичных методах говорилось еще в главе 2, и вот мы добрались до ситуации, при которой такие методы чрезвычайно полезны. В Microsoft определили, когда в течение жизненного цикла сущностного класса разработчики наиболее вероятно заинтересованы в получении уведомлений, и добавили вызовы частичных методов.

Ниже представлен список поддерживаемых частичных методов.

Поддерживаемые частичные методы для сущностного класса

```
partial void OnLoaded();
partial void OnValidate(ChangeAction action);
partial void OnCreated();
partial void On[Property]Changing([Type] value);
partial void On[Property]Changed();
```

Последние два метода из этого списка получают имя, заменяя часть `"[Property]"` именем свойства, а часть `"[Type]"` — типом данных этого свойства. Чтобы продемонстрировать некоторые частичные методы, поддерживаемые сущностными классами, добавим к сущностному классу `Contact` следующий код.

Дополнительное объявление для класса `Contact` для реализации некоторых частичных методов

```
namespace nwind
{
  public partial class Contact
  {
    partial void OnLoaded()
    {
      Console.WriteLine("Вызван OnLoaded().");
    }
    partial void OnCreated()
    {
      Console.WriteLine("Вызван OnCreated().");
    }
    partial void OnCompanyNameChanging(string value)
    {
      Console.WriteLine("Вызван OnCompanyNameChanging().");
    }
    partial void OnCompanyNameChanged()
    {
```

```
        Console.WriteLine("Вызван OnCompanyNameChanged().");
      }
    }
  }
```

Обратите внимание на указание пространства имен nwind. Это необходимо, поскольку пространство имен для объявления класса должно совпадать с пространством имен расширяемого класса. Поскольку при генерации сущностного класса в SQLMetal указано пространство имен nwind, частичный класс Contact также должен быть объявлен в том же пространстве имен nwind. В реальном рабочем коде, скорее всего, для этого объявления частичного класса будет предусмотрен отдельный модуль.

Как видите, предоставлены простые реализации для методов OnLoaded, OnCreated, OnCompanyNameChanging и OnCompanyNameChanged, которые лишь выводят сообщения на консоль. Теперь давайте взглянем на некоторый код, демонстрирующий использование частичных методов. В листинге 15.6 из базы данных запрашивается запись Contact и модифицируется ее свойство CompanyName.

Листинг 15.6. Запрос класса с помощью реализованных частичных методов

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
Contact contact = db.Contacts.Where(c => c.ContactID == 11).SingleOrDefault();
Console.WriteLine("CompanyName = {0}", contact.CompanyName);
contact.CompanyName = "Joe's House of Booze";
Console.WriteLine("CompanyName = {0}", contact.CompanyName);
```

В приведенном коде нет ничего особенного, за исключением реализации ряда частичных методов, поддерживаемых сущностными классами. Сначала запрашивается контакт и на консоль выводится название его компании. Затем название компании изменяется и снова выводится на консоль. Давайте посмотрим на вывод этого кода:

```
Вызван OnCreated().
Вызван OnLoaded().
CompanyName = B's Beverages
Вызван OnCompanyNameChanging().
Вызван OnCreated().
Вызван OnCompanyNameChanged().
CompanyName = Joe's House of Booze
```

Как видите, сначала был вызван метод OnCreate, а после него — метод OnLoaded. В этой точке из базы данных была извлечена запись и загружена в сущностный объект Contact. На консоль было выведено наименование компании. Затем был вызван метод OnCompanyNameChanging, после которого, как ни удивительно, произошел еще один вызов OnCreate. Очевидно, что DataContext создает еще один сущностный объект Contact как часть процедуры отслеживания изменений. Далее вызывается метод OnCompanyNameChanged, за которым производится вывод нового имени компании на консоль.

Это демонстрирует, как можно расширить сущностные классы, используя частичные методы без модификации сгенерированного кода.

Важные классы API-интерфейса System.Data.Linq

В пространстве имен System.Data.Linq существует группа классов, которые будут регулярно применяться при использовании LINQ to SQL. В следующем разделе представлен краткий обзор этих классов, их свойств и места в схеме LINQ to SQL.

EntitySet<T>

Сущностный класс на стороне *один* в отношении “один ко многим” хранит ассоциированные с ним сущностные классы стороны *многие* в своем члене класса типа EntitySet<T>, где T — тип ассоциированного сущностного класса.

Поскольку в базе данных Northwind отношение между заказами и заказчиками построено по схеме “один ко многим”, в классе Customer соответствующие ему Orders хранятся в EntitySet<Order>.

```
private EntitySet<Order> _Orders;
```

Класс EntitySet<T> представляет собой специальную коллекцию, используемую LINQ to SQL. Он реализует интерфейс IEnumerable<T>, который позволяет выполнять на нем запросы LINQ. Кроме того, он реализует интерфейс ICollection<T>.

EntityRef<T>

Сущностный класс на стороне *многие* отношения “один ко многим” хранит ассоциированный с ним класс стороны *один* в члене типа EntityRef<T>, где T — тип ассоциированного сущностного класса.

Поскольку в базе данных Northwind отношение между заказами и заказчиками построено по схеме “один ко многим”, класс Customer сохраняется в классе Order в члене типа EntityRef<Customer>:

```
private EntityRef<Customer> _Customer;
```

Entity

Когда мы ссылаемся на ассоциированный сущностный класс, представляющий сторону *один* в отношении “один ко многим”, то склонны думать, что соответствующая переменная-член имеет тот же тип, что и сущностный класс. Например, когда мы ссылаемся на Customer объекта Order, то думаем, что объект Customer хранится в члене класса Order типа Customer. Однако на самом деле следует помнить, что Customer хранится в EntityRef<Customer>. Когда нужно обратиться к действительной ссылке на объект Customer из члена типа EntityRef<Customer>, это должно делаться через член объекта EntityRef<Customer> по имени Entity.

Бывают случаи, когда важно осознавать этот факт — например, при написании собственных сущностных классов. Если вы взглянете на класс Order, сгенерированный SQLMetal, то заметите, что общедоступные методы get и set свойства Customer используют свойство Entity объекта EntityRef<Customer>, чтобы обратиться к Customer.

Общедоступное свойство, использующее свойство EntityRef<T>.Entity для доступа к действительному сущностному объекту

```
private EntityRef<Customer> _Customer;
...
public Customer Customer
{
  get
  {
    return this._Customer.Entity;
  }
  set
  {
    Customer previousValue = this._Customer.Entity;
    ...
  }
}
```

HasLoadedOrAssignedValue

Это булевское свойство позволяет узнать, присвоено ли свойству класса, хранящемуся в EntityRef<T>, значение, или же оно было загружено в него.

Обычно оно используется в методах set для ссылок на сторону *один* в ассоциации "один ко многим", чтобы предотвратить несогласованность между свойством класса, хранящим идентификаторы стороны *один*, и EntityRef<T>, хранящим ссылку на сторону *один*.

Например, давайте посмотрим на методы set для свойств CustomerID и Customer сущностного класса Order.

Метод set для свойства CustomerId

```
public string CustomerID
{
  get
  {
    return this._CustomerID;
  }
  set
  {
    if ((this._CustomerID != value))
    {
      if (this._Customer.HasLoadedOrAssignedValue)
      {
        throw new System.Data.Linq.ForeignKeyReferenceAlreadyHasValueException();
      }
      this.OnCustomerIDChanging(value);
      this.SendPropertyChanging();
      this._CustomerID = value;
      this.SendPropertyChanged("CustomerID");
      this.OnCustomerIDChanged();
    }
  }
}
```

Обратите внимание в методе set для свойства CustomerID, что если EntityRef<T>, хранящий Customer, имеет свойство HasLoadedOrAssignedValue, равное true, то генерируется исключение. Это предохраняет разработчика от изменения CustomerID сущностного объекта Order, если этот Order уже имеет присвоенную ему сущность Customer. Благодаря такой защите, даже при желании невозможно достичь рассогласованности между CustomerID и Customer сущностного объекта Order.

В отличие от этого, в методе set для свойства Customer ссылка Customer может быть присвоена, только если свойство HasLoadedOrAssignedValue установлено в false.

Метод set для свойства Customer

```
public Customer Customer
{
  get
  {
    return this._Customer.Entity;
  }
  set
  {
    Customer previousValue = this._Customer.Entity;
    if (((previousValue != value)
         || (this._Customer.HasLoadedOrAssignedValue == false)))
```

```
      {
        this.SendPropertyChanging();
        if ((previousValue != null))
        {
          this._Customer.Entity = null;
          previousValue.Orders.Remove(this);
        }
        this._Customer.Entity = value;
        if ((value != null))
        {
          value.Orders.Add(this);
          this._CustomerID = value.CustomerID;
        }
        else
        {
          this._CustomerID = default(string);
        }
        this.SendPropertyChanged("Customer");
      }
    }
  }
```

Проверка значения свойства `HasLoadedOrAssignedValue` в каждом методе `set` предохраняет разработчика от внесения несогласованности между ссылками `CustomerID` и `Customer`.

Table<T>

Этот тип данных LINQ to SQL использует для взаимодействия с таблицей или представлением в базе данных SQL Server. Обычно производный от `DataContext` класс, который в главах по LINQ to SQL часто назывался `[Your]DataContext`, будет иметь общедоступное свойство типа `Table<T>`, где `T` — сущностный класс, для каждой таблицы базы данных в наследнике `DataContext`. Он должен выглядеть примерно так, как показано ниже.

Свойство `Table<T>` для таблицы `Customers` базы данных

```
public System.Data.Linq.Table<Customer> Customers
{
  get
  {
    return this.GetTable<Customer>();
  }
}
```

`Table<T>` реализует интерфейс `IQueryable<T>`, который сам реализует `IEnumerable<T>`. Это значит, что на нем можно выполнять запросы LINQ to SQL. Это — начальный источник данных для большинства запросов LINQ to SQL.

IExecuteResult

Когда методом `ExecuteMethodCall` вызывается хранимая процедура или определяемая пользователем функция, то результаты возвращаются в объекте, реализующем интерфейс `IExecuteResult`, как показано ниже.

Метод `ExecuteMethodCall` возвращает `IExecuteResult`

```
IExecuteResult result = this.ExecuteMethodCall(...);
```

Интерфейс `IExecuteResult` предоставляет одно свойство по имени `ReturnValue` и один метод по имени `GetParametrValue` для доступа к возвращенному значению и выходному параметру соответственно.

ReturnValue

Все результаты хранимой процедуры, кроме выходных параметров и результатов функций, определяемых пользователем, возвращаются через переменную IExecuteResult.ReturnValue.

Чтобы получить доступ к возвращенному значению хранимой процедуры или функции со скалярным типом возврата, вы обращаетесь к члену ReturnValue возвращенного объекта. Ваш код должен выглядеть примерно так, как показано ниже.

Обращение к возвращенному значению хранимой процедуры, возвращающей целое число

```
IExecuteResult result = this.ExecuteMethodCall(...);
int returnCode = (int)(result.ReturnValue);
```

В главе 16 рассматривается метод ExecuteMethodCall, а также пример, возвращающий целое, которое возвращено хранимой процедурой.

Если хранимая процедура возвращает данные помимо ее возвращенного значения, то переменная ReturnValue будет реализовывать интерфейс ISingleResult<T> или IMultipleResults, который больше подходит, в зависимости от того, сколько форм данных возвращается из хранимой процедуры.

GetParameterValue

Для получения доступа к выходным параметрам хранимой процедуры на возвращенном объекте вызывается метод GetParameterValue с передачей ему начинающегося с нуля индекса параметра, значение которого требуется получить. Если предположить, что хранимая процедура возвращает CompanyName в третьем параметре, то код может выглядеть примерно так, как показано ниже.

Доступ к возвращенным параметрам хранимой процедуры

```
IExecuteResult result = this.ExecuteMethodCall(..., param1, param2, companyName);
string CompanyName = (string)(result.GetParameterValue(2));
```

Метод ExecuteMethodCall вместе с примером доступа к выходным параметрам хранимой процедуры рассматриваются в главе 16.

ISingleResult<T>

Когда хранимая процедура возвращает свои результаты в единственной форме, то они возвращаются в объекте, реализующем интерфейс ISingleResult<T>, где T — сущностный класс. Этот возвращенный объект, реализующий ISingleResult<T> — переменная IExecuteResult.ReturnValue. Код должен выглядеть подобно приведенному ниже.

Обращение к возвращенным результатам, когда они имеют одну форму

```
IExecuteResult result = this.ExecuteMethodCall(...);
ISingleResult<CustOrdersOrdersResult> results =
  (ISingleResult<CustOrdersOrdersResult>)(result.ReturnValue);
```

Обратите внимание, что здесь просто выполняется приведение члена ReturnValue объекта IExecuteResult к ISingleResult<T>, чтобы получить результаты. Поскольку интерфейс ISingleResult<T> унаследован от IEnumerable<T>, имеется возможность доступа к возвращенным результатам, как к любой другой последовательности LINQ.

Обращение к результатам из ISingleResult<T>

```
foreach (CustomersByCityResult cust in results)
{
  ...
}
```

Метод ExecuteMethodCall вместе с примером доступа к выходным параметрам хранимой процедуры, когда она возвращает их в одной форме, рассматриваются в главе 16.

ReturnValue

Интерфейс ISingleResult<T> представляет свойство ReturnValue, которое работает так же, как в интерфейсе IExecuteResult. Прочтите предыдущий раздел о свойстве ReturnValue интерфейса IExecuteResult, чтобы понять, как обращаться к этому свойству.

IMultipleResults

Когда хранимая процедура возвращает результаты в нескольких формах, то эти результаты возвращаются в объекте, реализующем интерфейс IMultipleResults. Этот возвращенный объект, реализующий IMultipleResults, является переменной IExecuteResult.ReturnValue. Код должен выглядеть примерно так, как показано ниже.

Обращение к возвращенным результатам, имеющим несколько форм

```
IExecuteResult result = this.ExecuteMethodCall(...);
IMultipleResults results = (IMultipleResults)(result.ReturnValue);
```

Чтобы получить доступ к множественным возвращенным формам, вызывайте метод IMultipleResults.GetResult<T>, описанный ниже.

Метод ExecuteMethodCall вместе с примером доступа к выходным параметрам хранимой процедуры, когда она возвращает их в нескольких формах, рассматриваются в главе 16.

Интерфейс IMultipleResults предоставляет одно свойство по имени ReturnValue для доступа к возвращенному хранимой процедурой значению и один метод по имени GetResult<T> для извлечения IEnumerable<T> для каждой возвращенной формы, где T — сущностный класс, соответствующий форме.

ReturnValue

Интерфейс IMultipleResults<T> предоставляет свойство ReturnValue, которое работает, как и в интерфейсе IExecuteResults. Прочтите предыдущий раздел о свойстве ReturnValue интерфейса IExecuteResults, чтобы понять, каким образом обращаться к этому свойству.

GetResult<T>

Интерфейс IMultipleResults предоставляет метод GetResult<T>, где T представляет тип данных, хранящий возвращенную форму. Метод GetResult<T> используется для получения повторяющихся записей указанной результирующей формы, и эти записи возвращаются в IEnumerable<T>, где T представляет сущностный класс, используемый для хранения записи формы. Код должен выглядеть примерно так, как показано ниже.

Обращение к нескольким формам, возвращенным хранимой процедурой

```
[StoredProcedure(Name="A Stored Procedure")]
[ResultType(typeof(Shape1))]
[ResultType(typeof(Shape2))]
...
IExecuteResult result = this.ExecuteMethodCall (...);
IMultipleResults results = (IMultipleResults)(result.ReturnValue);
foreach(Shape1 x in results.GetResult<Shape1>()) {...}
foreach(Shape2 y in results.GetResult<Shape2>()) {...}
```

Здесь включены атрибуты, которые должны находится перед методом, содержащим этот код, чтобы можно было видеть контекст атрибутов `ResultType` и формы, возвращенные хранимой процедурой.

В приведенном коде известно, что записи, отображаемые на `Shape1`, будут возвращены хранимой процедурой первыми, после чего пойдут записи, отображаемые на `Shape2`. Поэтому сначала выполняется перечисление последовательности `IEnumerable<Shape1>`, которая возвращается первым вызовом метода `GetResult<T>`, а затем — перечисление последовательности `IEnumerable<Shape2>`, возвращенной вторым вызовом метода `GetResult<T>`. Важно то, что записи `Shape1` возвращаются первыми, а за ними — записи `Shape2`, и потому они извлекаются с помощью метода `GetResult<T>` именно в таком порядке.

Метод `ExecuteMethodCall` вместе с примером доступа к выходным параметрам хранимой процедуры, когда она возвращает их в нескольких формах, рассматривается в главе 16.

Резюме

В этой главе было предложено углубленное рассмотрение сущностных классов LINQ to SQL, сложностей, сопровождающих их самостоятельное написание, а также их атрибутов и свойств этих атрибутов.

Важно помнить, что при написании собственных сущностных классов вы отвечаете за реализацию уведомлений об изменении и обеспечение согласованности графа. Эти детали нетривиальны и могут оказаться сложными в реализации. К счастью, как указывалось в этой главе, и SQLMetal, и Object Relational Designer благополучно справляются со всеми этими сложностями.

Для написания собственных сущностных классов также необходимо иметь четкое представление об их атрибутах и свойствах этих атрибутов. В этой главе было описано каждое из них с предоставлением соответствующей реализации за счет обсуждения сгенерированных SQLMetal сущностных классов для базы данных Northwind.

Также были описаны преимущества проецирования результатов запроса на сущностные классы по сравнению с несущностными. Если необходимость в модификации данных и сохранении изменений отсутствует, то несущностные классы обычно подходят. Но если нужна возможность изменять возвращенные данные и сохранять их обратно в базе данных, то есть только один способ — проецирование результата на сущностные классы.

И, наконец, были рассмотрены некоторые часто используемые классы из пространства имен `System.Data.Linq`, а также их применение в LINQ to SQL.

К этому моменту вы уже должны хорошо понимать сущностные классы. Они достаточно глубоко обсуждались, и объяснялось устройство сгенерированного кода. Конечно, к этим сущностным классам обычно обращается класс, унаследованный от `DataContext`, который еще предстоит рассмотреть детально. Этим мы займемся в следующей главе.

ГЛАВА 16

Класс DataContext в LINQ to SQL

В этой главе объясняется устройство класса DataContext, что он может делать и как он это делает. Мы рассмотрим основные его методы и приведем примеры вызова каждого. Понимание класса DataContext необходимо для успешного применения LINQ to SQL, и по окончании чтения этой главы вы должны научиться пользоваться классом DataContext.

Предварительные условия для запуска примеров

Для того чтобы запускать примеры настоящей главы, понадобится расширенная версия базы данных Northwind и сгенерированные для нее сущностные классы. Прочтите раздел "Предварительные условия для запуска примеров" главы 12 и выполните то, что там сказано.

Некоторые общие методы

Вдобавок для запуска примеров этой главы понадобятся некоторые общие методы, которые будут использованы в примерах. Прочтите раздел "Некоторые общие методы" главы 12 и выполните то, что там сказано.

Использование API-интерфейса LINQ to SQL

Для запуска примеров этой главы может понадобиться добавить соответствующие ссылки и директивы using к проекту. Прочтите раздел "Использование LINQ to SQL" главы 12 и выполните то, что там указано.

Дополнительно для некоторых примеров настоящей главы также потребуется добавить директиву using для пространства имен System.Data.Linq.Mapping:

```
using System.Data.Linq.Mapping;
```

Класс [Your]DataContext

Хотя он пока подробно не рассматривался, но одним из классов LINQ to SQL, который будет часто использоваться, является System.Data.Linq.DataContext. Это класс, который будет применяться для установления подключения к базе данных.

При создании или генерации сущностных классов принято создавать класс, унаследованный от DataContext. Этот производный класс обычно будет называться по имени базы данных, к которой он будет подключаться. Поскольку в примерах этой главы ис-

пользуется база данных Northwind, производный класс базы данных будет называться Northwind. Однако, т.к. имя производного класса изменяется в зависимости от используемой базы данных, на него часто ссылаются как на [Your]DataContext. Это — намек на то, что речь идет о созданном вами или сгенерированном классе, унаследованном от DataContext.

Класс DataContext

Класс DataContext обрабатывает подключение к базе данных. Он также обрабатывает запросы, обновления, вставки в базу данных, отслеживает идентичность, отслеживает изменения, обрабатывает их, обеспечивает целостность транзакций и даже создание базы данных. Класс DataContext транслирует запросы сущностных классов в операторы SQL, которые выполняются на подключенной базе данных.

Производный от DataContext класс [Your]DataContext предоставляет доступ к целой группе методов базы данных, таких как ExecuteQuery, ExecuteCommand и SubmitChanges. В дополнение к этим унаследованным методам класс [Your] DataContext будет содержать свойства типа System.Data.Linq.Table<T> для каждой таблицы и представления в базе данных, с которой будет использоваться LINQ to SQL, где каждый тип T — это сущностный класс, отображенный на конкретную таблицу или представление.

Например, взглянем на класс Northwind, который был сгенерирован инструментом SQLMetal. Это — класс [Your]DataContext для базы данных Northwind. Ниже приведена наиболее примечательная часть этого класса с выделением важных частей полужирным.

Часть сгенерированного класса Northwind

```
public partial class Northwind : System.Data.Linq.DataContext
{
  ...
  static Northwind()
  {
  }

  public Northwind(string connection) :
     base(connection, mappingSource)
  {
   OnCreated();
  }
  public Northwind(System.Data.IDbConnection connection) :
     base(connection, mappingSource)
  {
   OnCreated();
  }
  public Northwind(string connection,
                System.Data.Linq.Mapping.MappingSource mappingSource) :
   base(connection, mappingSource)
  {
   OnCreated();
  }
  public Northwind(System.Data.IDbConnection connection,
                System.Data.Linq.Mapping.MappingSource mappingSource) :
   base(connection, mappingSource)
  {
   OnCreated();
  }
  ...
```

```
    public System.Data.Linq.Table<Customer> Customers
    {
      get
      {
        return this.GetTable<Customer>();
      }
    }
    ...
  }
```

Как видите, этот класс действительно наследуется от DataContext. Легко заметить, что в нем объявлены пять конструкторов. Обратите внимание, что конструктор по умолчанию является приватным (private), поскольку для него не указан модификатор доступа, так что создать экземпляр [Your]DataContext без параметров не получится. Каждый из общедоступных конструкторов [Your]DataContext соответствует одному из конструкторов DataContext и вызывает эквивалентный конструктор класса DataContext в своем инициализаторе. В теле конструкторов вызывается только частичный метод OnCreated. Это позволяет разработчику реализовывать частичный метод OnCreated, который вызывается при каждом создании объекта [Your]DataContext.

Кроме того, в классе Northwind есть свойство по имени Customers типа Table<Customer>, где тип Customer представляет сущностный класс. Этот сущностный класс Customer отображается на таблицу Customers базы данных Northwind.

На самом деле не обязательно писать код, использующий класс [Your]DataContext. Вполне можно работать и со стандартным классом DataContext. Однако применение класса [Your]DataContext делает написание остального кода более удобным. Например, если вы используете класс [Your]DataContext, то каждая таблица представлена свойством, которое доступно непосредственно через объект [Your]DataContext. Пример показан в листинге 16.1.

Листинг 16.1. Пример, демонстрирующий доступ к таблице через свойство

```
Northwind db =
  new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
IQueryable<Customer> query = from cust in db.Customers
                             where cust.Country == "USA"
                             select cust;
foreach(Customer c in query)
{
  Console.WriteLine("{0}", c.CompanyName);
}
```

На заметку! В примерах, приведенных в этой главе, может понадобиться изменить строки соединения, чтобы заставить их работать.

В предыдущем коде, поскольку осуществляется подключение с использованием класса [Your]DataContext по имени Northwind, есть возможность обращаться к таблице заказчиков Table<Customer> как к свойству Customers класса [Your]DataContext. Вот результат работы этого кода:

```
Great Lakes Food Market
Hungry Coyote Import Store
Lazy K Kountry Store
Let's Stop N Shop
Lonesome Pine Restaurant
```

```
Old World Delicatessen
Rattlesnake Canyon Grocery
Save-a-lot Markets
Split Rail Beer & Ale
The Big Cheese
The Cracker Box
Trail's Head Gourmet Provisioners
White Clover Markets
```

Если бы вместо этого подключение производилось с применением самого класса DataContext, нужно было бы использовать метод GetTable<T> объекта DataContext, как показано в листинге 16.2.

Листинг 16.2. Пример, демонстрирующий доступ к таблице через метод GetTable<T>

```
DataContext dc =
  new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
IQueryable<Customer> query = from cust in dc.GetTable<Customer>()
                        where cust.Country == "USA"
                        select cust;
foreach(Customer c in query)
{
  Console.WriteLine("{0}", c.CompanyName);
}
```

Этот код дает тот же самый результат:

```
Great Lakes Food Market
Hungry Coyote Import Store
Lazy K Kountry Store
Let's Stop N Shop
Lonesome Pine Restaurant
Old World Delicatessen
Rattlesnake Canyon Grocery
Save-a-lot Markets
Split Rail Beer & Ale
The Big Cheese
The Cracker Box
Trail's Head Gourmet Provisioners
White Clover Markets
```

Так что использование класса [Your]DataContext просто удобнее, и это стоит взять на вооружение.

Класс DataContext реализует интерфейс IDisposable

Класс DataContext реализует интерфейс IDisposable и потому должен трактоваться правильно как освобождаемый (disposabe) объект. Это значит, что если создается новый класс, содержащий внутри себя класс DataContext или [Your]DataContext, что означает отношение “имеет” (has-a) между новым классом и классом DataContext или [Your]DataContext, то новый класс также должен реализовывать интерфейс IDisposable. Вопросы проектирования классов для правильной реализации интерфейса IDisposable выходят за рамки контекста настоящей книги, но есть множество онлайновых ресурсов, посвященных этой теме. Другое преимущество реализации интерфейса IDisposable классом DataContext состоит в том, что теперь можно применять оператор using для управления объектом DataContext или [Your]DataContext.

Основное назначение

В дополнение ко всем методам, которые рассматриваются в этой главе, класс `DataContext` предоставляет три основных службы: отслеживание идентичности, отслеживание изменений и обработка изменений.

Отслеживание идентичности

Одной из проблем, для преодоления которых предназначен LINQ to SQL, является проблема *объектно-реляционной потери соответствия* (object-relational impedance mismatch). Этот термин указывает на неизбежные трудности, вызванные тем фактом, что большинство наиболее распространенных баз данных являются реляционными, в то время как большинство современных языков программирования — объектно-ориентированными. Из-за этого различия и возникают проблемы.

Одним из проявлений объектно-реляционной потери соответствия является ожидаемый способ поведения идентичности. При запросе одной и той же записи из базы данных во многих местах кода ожидается, что возвращенные данные будут находиться в разных местах в памяти. Кроме того, модификация полей записей в одной части кода не должна затрагивать поля той же записи, извлекаемые в другой части кода. Такое поведение ожидается потому, что извлеченные данные хранятся в разных переменных, находящихся по разным адресам в памяти.

Однако от объектов ожидается другое поведение. Мы ожидаем, что когда есть объект в памяти, скажем, `Customer`, то можно рассчитывать, что во всех местах код, имеющий ссылку на одного и того же заказчика, будет ссылаться на одно и то же местоположение в памяти. При обновлении свойства, представляющего имя объекта заказчика, в одном месте программы можно ожидать, что заказчик, ссылка на которого имеется, в другой части кода будет иметь новое имя.

Такое поведение обеспечивается службой отслеживания идентичности `DataContext`. Когда запись запрашивается из базы данных первый раз с момента создания экземпляра объекта `DataContext`, эта запись сохраняется в таблице идентичности с использованием ее первичного ключа, после чего создается объект идентичности и сохраняется в кэше. Последующие запросы, которые определяют, что должна быть возвращена та же самая запись, сначала проверяют таблицу идентичности, и если эта запись там присутствует, то из кэша возвращается уже существующий сущностный объект. Эта важная концепция, которую необходимо понимать, поэтому мы еще раз рассмотрим ее, но немного иным путем. Если при выполнении запроса запись в базе данных отвечает критерию поиска *и* если ее сущностный объект уже кэширован, то возвращается этот кэшированный объект. Это значит, что действительные данные, возвращенные запросом, могут быть не теми же самыми, что содержатся в записи базы данных. Запрос определяет, *какие сущности* должны возвращаться, на основе данных в базе. Но служба слежения за идентичностью объекта `DataContext` определит, *какие данные* возвращаются. Это может привести к проблеме, которая называется *несоответствием кэша результирующего набора*.

Несоответствие кэша результирующего набора

Несоответствие кэша результирующего набора может произойти, когда запись в базе данных не согласована с сущностным объектом, представляющим ее в кэше объекта `DataContext`. Когда выполняется запрос, действительная база данных опрашивается на предмет поиска записи, отвечающей запросу. Если запись в базе соответствует критерию поиска, то сущностный объект этой записи будет включен в возвращенный результирующий набор. Однако если сущностный объект записи из результирующего набора уже кэширован в объекте `DataContext`, то кэшированный сущностный объект

будет возвращен запросом вместо того, чтобы прочесть последнюю версию записи из базы данных.

В результате получается, что если есть сущностный объект, кэшированный в DataContext, а другой контекст обновляет поле записи этого сущностного объекта в базе данных, и выполняется запрос LINQ с указанием этого поля в критерии поиска, так что он соответствует новому значению из базы данных, то такая запись будет включена в результирующий набор. Однако поскольку она уже имеется в кэше, получится кэшированный сущностный объект с полем, которое не отвечает заданному критерию поиска.

Возможно, станет понятнее, если рассмотреть конкретный пример. Сначала запросим определенного заказчика, который, как известно, не соответствует критерию поиска, указанному в последующем запросе. Будем использовать заказчика LONEP. Регионом этого заказчика является OR, поэтому производится поиск заказчиков из региона WA. Затем заказчики из региона WA выводятся на консоль. После этого регион заказчика LONEP заменяется значением WA, используя для этого ADO.NET — как если бы это сделал некоторый другой контекст, внешний по отношению к текущему процессу. В этой точке LONEP будет иметь регион OR в сущностном объекте, но WA — в базе данных. Далее тот же самый запрос выполняется снова, чтобы извлечь всех заказчиков из региона WA. Взглянув на код, вы не увидите там нового определения запроса. Вы просто увидите, что производится перечисление на возвращенной последовательности custs. Помните, что из-за отложенного выполнения запросов нужно лишь перечислить результаты, чтобы снова запустить запрос. Поскольку в базе данных регион заказчика LONEP теперь WA, эта запись будет включена в результирующий набор. Но поскольку сущностный объект этой записи уже находится в кэше, будет возвращен именно этот кэшированный сущностный объект, а у этого объекта значение региона — по-прежнему OR. Затем регионы каждого возвращенного сущностного объекта выводятся на консоль. Когда очередь дойдет до заказчика LONEP, его регионом окажется OR, несмотря на тот факт, что в запросе указана необходимость в заказчиках из региона WA. В листинге 16.3 представлен код, демонстрирующий это несоответствие.

Листинг 16.3. Пример, демонстрирующий несоответствие кэша результирующего набора

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");

// Получить заказчика для модификации, который находится
// вне запрашиваемого региона == 'WA'.
Customer cust = (from c in db.Customers
              where c.CustomerID == "LONEP"
              select c).Single<Customer>();
Console.WriteLine("Заказчик {0} имеет регион = {1}.{2}",
  cust.CustomerID, cust.Region, System.Environment.NewLine);

// Регионом LONEP является OR.
// Теперь получить последовательность заказчиков из 'WA',
// которая не включает LONEP, поскольку его регион — OR.
IEnumerable<Customer> custs = (from c in db.Customers
                          where c.Region == "WA"
                          select c);
Console.WriteLine("Заказчики из WA перед изменением посредством ADO.NET - начало...");
foreach(Customer c in custs)
{
  // Отобразить Region каждого сущностного объекта.
  Console.WriteLine("Заказчик {0} имеет регион {1}.", c.CustomerID, c.Region);
}
Console.WriteLine("Заказчики из WA перед изменением посредством ADO.NET - конец.{0}",
  System.Environment.NewLine);
```

```
// Теперь изменить регион заказчика LONEP на WA, что приведет
// к включению его в результаты предыдущего запроса.
// Изменить регион заказчика через ADO.NET.
Console.WriteLine("Обновление региона для LONEP на WA в ADO.NET...");
ExecuteStatementInDb(
  "update Customers set Region = 'WA' where CustomerID = 'LONEP'");
Console.WriteLine("Регион для LONEP обновлен.{0}", System.Environment.NewLine);
Console.WriteLine("Таким образом, регионом для LONEP является WA в базе данных, но...");
Console.WriteLine("Заказчик {0} имеет регион = {1} в сущностном объекте.{2}",
  cust.CustomerID, cust.Region, System.Environment.NewLine);

// Теперь в базе данных регионом для LONEP является WA,
// но в сущностном объекте - по-прежнему WA.
// Снова выполнить запрос.
// И снова отобразить регион сущностного объекта заказчика.
Console.WriteLine("Запрос сущностных объектов после изменения посредством ADO.NET
- начало...");

foreach(Customer c in custs)
{
  // Отобразим Region каждого сущностного объекта.
  Console.WriteLine("Заказчик {0} имеет регион {1}.", c.CustomerID, c.Region);
}

Console.WriteLine("Запрос сущностных объектов после изменения посредством ADO.NET
- конец.{0}", System.Environment.NewLine);

// Вернуть измененное значение в исходное состояние, чтобы
// данный код можно было запустить более одного раза.
Console.WriteLine("{0}Сброс данных в исходные значения.",
  System.Environment.NewLine);
ExecuteStatementInDb(
  "update Customers set Region = 'OR' where CustomerID = 'LONEP'");
```

Ниже показан результат выполнения этого кода:

```
Заказчик LONEP имеет регион = OR.

Заказчики из WA перед изменением посредством ADO.NET — начало ...
Заказчик LAZYK имеет регион WA.
Заказчик TRAIH имеет регион WA.
Заказчик WHITC имеет регион WA.
Заказчики из WA перед изменением посредством ADO.NET - конец.

Обновление региона для LONEP на WA в ADO.NET...
Выполнение оператора SQL для базы данных с помощью ADO.NET...
База данных обновлена.
Регион для LONEP обновлен.

Таким образом, регионом для LONEP является WA в базе данных, но...
Заказчик LONEP имеет регион = OR в сущностном объекте.

Запрос сущностных объектов после изменения посредством ADO.NET - начало...
Заказчик LAZYK имеет регион WA.
Заказчик LONEP имеет регион OR.
Заказчик TRAIH имеет регион WA.
Заказчик WHITC имеет регион WA.
Запрос сущностных объектов после изменения посредством ADO.NET - конец.

Сброс данных в исходные значения.
Выполнение оператора SQL для базы данных с помощью ADO.NET...
База данных обновлена.
```

Как видите, даже несмотря на то, что запрашивались заказчики из региона WA, заказчик LONEP был включен в результат, при том, что его регионом является OR. Конечно, в базе данных у LONEP в качестве региона указано WA, но у объекта, на который имеется ссылка в коде, это не так.

Другим аспектом этого поведения является тот факт, что вставленные сущности не могут быть запрошены, а удаленные — могут, если это происходит до вызова SubmitChanges. Опять-таки, это объясняется тем, что хотя сущность и вставлена, при выполнении запроса результирующий набор определяется действительным содержимым базы данных, а не кэшем объекта DataContext. Поскольку изменения не были зафиксированы, вставленной сущности в базе данных еще нет. С удаленными сущностями — обратная картина. В листинге 16.4 приведен пример, демонстрирующий это поведение.

Листинг 16.4. Другой пример, демонстрирующий несоответствие кэша результирующего набора

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");

Console.WriteLine("Сначала добавляется заказчик LAWN.");
db.Customers.InsertOnSubmit(
  new Customer
  {
    CustomerID = "LAWN",
    CompanyName = "Lawn Wranglers",
    ContactName = "Mr. Abe Henry",
    ContactTitle = "Owner",
    Address = "1017 Maple Leaf Way",
    City = "Ft. Worth",
    Region = "TX",
    PostalCode = "76104",
    Country = "USA",
    Phone = "(800) MOW-LAWN",
    Fax = "(800) MOW-LAWO"
  });

Console.WriteLine("Далее заказчик LAWN запрашивается.");
Customer cust = (from c in db.Customers
                 where c.CustomerID == "LAWN"
                 select c).SingleOrDefault<Customer>();
Console.WriteLine("Заказчик LAWN {0}.{1}",
  cust == null ? "не существует" : "существует",
  System.Environment.NewLine);

Console.WriteLine("Теперь удаляется заказчик LONEP.");
cust = (from c in db.Customers
        where c.CustomerID == "LONEP"
        select c).SingleOrDefault<Customer>();
db.Customers.DeleteOnSubmit(cust);

Console.WriteLine("Далее заказчик LONEP запрашивается.");
cust = (from c in db.Customers
        where c.CustomerID == "LONEP"
        select c).SingleOrDefault<Customer>();
Console.WriteLine("Заказчик LONEP {0}.{1}",
  cust == null ? "не существует" : "существует",
  System.Environment.NewLine);

// Возвращать базу в исходное состояние незачем,
// поскольку метод SubmitChanges() не вызывался.
```

В предыдущем коде сначала вставляется заказчик LAWN, который затем запрашивается, чтобы проверить его существование. После этого удаляется другой заказчик — LONEP — и выполняется запрос, чтобы проверить его наличие. Все это делается без вызова метода SubmitChanges, так что кэшированные сущностные объекты не сохраняются в базе данных. Вот результат выполнения этого кода:

```
Сначала добавляется заказчик LAWN.
Далее заказчик LAWN запрашивается.
Заказчик LAWN не существует.

Теперь удаляется заказчик LONEP.
Далее заказчик LONEP запрашивается.
Заказчик LONEP существует.
```

Разработчики из Microsoft утверждают, что задуманное поведение состоит в том, что данные, извлеченные запросом, считаются устаревшими на момент извлечения, а данные, кэшированные в DataContext, не предназначены для долговременного хранения в кэше. Если нужна лучшая изоляция и согласованность, они рекомендуют помещать все в транзакцию. Примеры того, как это делается, можно найти в разделе "Пессимистический параллелизм" главы 17.

Отслеживание изменений

Как только служба идентичности создает сущностный объект в кэше, служба отслеживания изменений начинает наблюдать за этим объектом. Отслеживание изменений работает, сохраняя первоначальные значения сущностного объекта. Эта служба продолжает свою работу до тех пор, пока не будет вызван метод SubmitChanges. Вызов этого метода сохраняет изменения сущностного объекта в базе данных, исходные значения отбрасываются, а измененные значения становятся исходными. Это позволяет снова приступать к отслеживанию изменений.

Все это работает хорошо до тех пор, пока сущностные объекты извлекаются из базы данных. Однако простое создание нового экземпляра сущностного объекта не вызывает никакого отслеживания идентичности или изменений до тех пор, пока DataContext не узнает об его существовании. Чтобы сообщить объекту DataContext о существовании сущностного объекта, просто вставьте сущностный объект в одно из свойств Table<T>. Например, в классе Northwind есть свойство типа Table<Customer> по имени Customers. На этом свойстве можно вызвать метод InsertOnSubmit и вставить сущностный объект Customer в Table<Customer>. Когда это сделано, DataContext начнет отслеживать идентичность и изменения этого сущностного объекта. Ниже приведен пример кода, вставляющего нового заказчика.

```
db.Customers.InsertOnSubmit(
  new Customer {
    CustomerID = "LAWN",
    CompanyName = "Lawn Wranglers",
    ContactName = "Mr. Abe Henry",
    ContactTitle = "Owner",
    Address = "1017 Maple Leaf Way",
    City = "Ft. Worth",
    Region = "TX",
    PostalCode = "76104",
    Country = "USA",
    Phone = "(800) MOW-LAWN",
    Fax = "(800) MOW-LAWO"});
```

Как только вызывается метод InsertOnSubmit, начинается отслеживание идентичности и изменений заказчика LAWN.

Поначалу служба отслеживания изменений может показаться несколько запутанной. Понять базовую концепцию достаточно просто, но уверенность в обращении с ней приходит не сразу. Понимание отслеживания изменений становится еще более важным при написании сущностных классов вручную. Чтобы получить более полное представление о работе службы слежения за изменениями, прочтите еще раз раздел "Уведомления об изменениях" в главе 15.

Обработка изменений

Одним из наиболее важных служб, предоставляемых классом `DataContext`, является отслеживание изменений в сущностных объектах. При вставке, изменении или удалении сущностного объекта `DataContext` выполняет мониторинг того, что происходит. Изменения кэшируются в `DataContext` до тех пор, пока не будет вызван метод `SubmitChanges`.

Когда вызывается метод `SubmitChanges`, процессор изменений объекта `DataContext` управляет обновлением базы данных. Сначала процессор изменений вставит все вновь вставленные сущностные объекты в свой список отслеживаемых объектов. Затем он упорядочит все измененные сущностные объекты на основе их зависимостей, обусловленных внешними ключами и ограничениями уникальности. Затем, если в области определения нет никаких транзакций, он создаст транзакцию, так что все команды SQL, выполняемые во время вызова метода `SubmitChanges`, будут обладать транзакционной целостностью. Он использует уровень изоляции SQL Server по умолчанию — `ReadCommited`, который означает, что прочитанные данные не будут физически повреждены и только зафиксированные (commited) данные будут прочитаны, но поскольку блокировка является разделяемой (shared), ничто не помешает данным измениться до окончания транзакции. И, наконец, он выполняет перечисление упорядоченного списка измененных сущностных объектов, создает необходимые операторы SQL и выполняет их.

Если в процессе перечисления измененных сущностных объектов происходят любые ошибки и если метод `SubmitChanges` использует для `ConflictMode` вариант `FailOnFirstConflict`, то процесс перечисления прерывается, транзакция откатывается, отменяя все изменения в базе данных, и генерируется исключение. Если для `ConflictMode` указан вариант `ContinueOnConflict`, то все измененные сущностные объекты будут перечислены и обработаны, несмотря на любые возникающие конфликты, пока `DataContext` строит список конфликтов. Но, опять-таки, транзакция откатывается, отменяя все изменения в базе данных, и генерируется исключение. Однако пока данные не сохранены в базе данных, все изменения сущностных объектов присутствуют в этих объектах. Это дает возможность разработчику попробовать решить проблему и снова вызвать метод `SubmitChanges`.

Если все изменения внесены в базу данных успешно, транзакция фиксируется, и информация об изменениях сущностных объектов удаляется, так что отслеживание изменений может начать работать заново.

Время жизни контекста данных

Одним из часто задаваемых вопросов является вопрос о том, насколько долго объект `DataContext` должен сохраняться актуальным и использоваться. Как упоминалось в разделе "Несоответствие кэша результирующего набора", данные, извлеченные и кэшированные объектом `DataContext`, считаются устаревшими на момент извлечения. Это означает, что чем дольше сохраняется актуальным объект `DataContext`, тем сильнее устаревают находящиеся в памяти данные. И потому это означает не только дополнительные накладные расходы, но и повышает вероятность несоответствия данных кэша результирующего набора реальным данным. Следовательно, настоятельно рекомендуется делать объекты `DataContext` как можно более кратковременными.

Создавать объект DataContext рекомендуется каждый раз, когда в нем возникает необходимость, и затем позволять ему покидать область видимости после вызова метода SubmitChanges. Конечно, ситуации бывают разные, так что это лишь рекомендация. Следует придерживаться эмпирического правила, что объект DataContext должен существовать несколько минут, а не часов.

У некоторых разработчиков возникает соблазн сохранять объект DataContext в течение более длительного времени, полагаясь на метод Refresh, который будет описан в конце этой главы, чтобы предотвратить несоответствие кэша результирующего набора. Это нельзя считать удачным подходом, потому что при этом приходится принимать решение о частоте и времени вызова метода Refresh. Будете ли вы вызывать его во время каждого использования объекта DataContext? Излишние вызовы метода Refresh приводят полному обновлению всего кэша из базы данных. Если DataContext существует достаточно долго, это может вызвать проблемы с производительностью. Слишком дорогая цена за то, чтобы избежать затрат на создание объекта DataContext.

DataContext() и [Your]DataContext()

Обычно от класса DataContext производится наследование для создания класса [Your]DataContext. Он существует для подключения к базе данных и обработки всего взаимодействия с базой данных. Для создания объекта DataContext или [Your] DataContext будет использоваться один из представленных далее конструкторов.

Прототипы

Конструктор DataContext имеет четыре прототипа, которые описаны ниже.

Первый прототип конструктора DataContext

```
DataContext(string fileOrServerOrConnection);
```

Этот прототип конструктора принимает строку соединения ADO.NET и, возможно, он будет использоваться большую часть времени. Этот прототип применяется также в большинстве примеров LINQ to SQL, приведенных в этой книге.

Второй прототип конструктора DataContext

```
DataContext (System.Data.IDbConnection connection);
```

Поскольку System.Data.SqlClient.SqlConnection наследуется от System.Data. Common.DbConnection, который реализует System.Data.IDbConnection, можно создать экземпляр DataContext или [Your]DataContext с только что созданным объектом SqlConnection. Этот прототип конструктора удобен, когда приходится смешивать код LINQ to SQL с существующим кодом ADO.NET.

Третий прототип конструктора DataContext

```
DataContext(string fileOrServerOrConnection,
          System.Data.Linq.MappingSource mapping);
```

Этот прототип конструктора удобен, когда нет класса [Your]DataContext, а вместо него применяется XML-файл отображения. Иногда уже может быть существующий бизнес-класс, к которому нельзя добавить соответствующие атрибуты LINQ to SQL. Возможно, даже исходный код его отсутствует. Файл отображения можно сгенерировать с помощью утилиты SQLMetal либо написать вручную, чтобы он работал с существующим бизнес-классом или любым другим классом того же рода. Для установки соединения предоставляется нормальная строка соединения ADO.NET.

Четвертый прототип конструктора DataContext

```
DataContext (System.Data.IDbConnection connection,
          System.Data.Linq.MappingSource mapping)
```

Этот прототип позволит создать соединение LINQ to SQL из существующего соединения ADO.NET и предоставить XML-файл отображения. Эта версия прототипа удобна в случаях комбинирования кода LINQ to SQL с существующим кодом ADO.NET при отсутствии сущностных классов, оснащенных атрибутами.

Примеры

Для примера первого прототипа конструктора DataContext в листинге 16.5 производится подключение к физическому файлу .mdf, используя строку соединения типа ADO.NET.

Листинг 16.5. Первый прототип конструктора DataContext, подключаемого к файлу базы данных

```
DataContext dc = new DataContext(@"C:\Northwind.mdf");
IQueryable<Customer> query = from cust in dc.GetTable<Customer>()
                        where cust.Country == "USA"
                        select cust;
foreach (Customer c in query)
{
  Console.WriteLine("{0}", c.CompanyName);
}
```

На заметку! Понадобится модифицировать путь, переданный конструктору DataContext, чтобы он мог найти ваш файл .mdf.

Чтобы создать экземпляр объекта DataContext, просто предоставляется путь к файлу .mdf. Поскольку создается объект DataContext, а не [Your]DataContext, для доступа к заказчикам в базе данных должен быть вызван метод GetTable<T>. Вот результат:

```
Great Lakes Food Market
Hungry Coyote Import Store
Lazy K Kountry Store
Let's Stop N Shop
Lonesome Pine Restaurant
Old World Delicatessen
Rattlesnake Canyon Grocery
Save-a-lot Markets
Split Rail Beer & Ale
The Big Cheese
The Cracker Box
Trail's Head Gourmet Provisioners
White Clover Markets
```

В листинге 16.6 демонстрируется тот же базовый код, но на этот раз используется класс [Your]DataContext, которым в данном случае является Northwind.

Листинг 16.6. Первый прототип конструктора [Your]DataContext, подключаемого к файлу базы данных

```
DataContext db = new Northwind(@"C:\Northwind.mdf");
IQueryable<Customer> query = from cust in db.Customers
                        where cust.Country == "USA" select cust;
```

```
foreach (Customer c in query)
{
  Console.WriteLine("{0}", c.CompanyName);
}
```

Обратите внимание, что вместо вызова метода GetTable<T> для доступа к заказчикам в базе данных просто производится ссылка на свойство Customers. Неудивительно, что код выдает те же самые результаты:

```
Great Lakes Food Market
Hungry Coyote Import Store
Lazy K Kountry Store
Let's Stop N Shop
Lonesome Pine Restaurant
Old World Delicatessen
Rattlesnake Canyon Grocery
Save-a-lot Markets
Split Rail Beer & Ale
The Big Cheese
The Cracker Box
Trail's Head Gourmet Provisioners
White Clover Markets
```

В целях полноты картины ниже представлен еще один пример применения первого прототипа, но на этот раз для действительного подключения к серверу базы данных SQL Server, содержащему присоединенную базу Northwind, используется строка соединения. И, поскольку обычная практика заключается в применении класса [Your]DataContext, это же делается и в листинге 16.7.

Листинг 16.7. Первый прототип конструктора [Your]DataContext, подключаемого к базе данных

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
IQueryable<Customer> query = from cust in db.Customers
                        where cust.Country == "USA"
                        select cust;
foreach (Customer c in query)
{
  Console.WriteLine("{0}", c.CompanyName);
}
```

Результаты все те же:

```
Great Lakes Food Market
Hungry Coyote Import Store
Lazy K Kountry Store
Let's Stop N Shop
Lonesome Pine Restaurant
Old World Delicatessen
Rattlesnake Canyon Grocery
Save-a-lot Markets
Split Rail Beer & Ale
The Big Cheese
The Cracker Box
Trail's Head Gourmet Provisioners
White Clover Markets
```

Поскольку второй прототип для класса DataContext удобен при комбинировании кода LINQ to SQL с кодом ADO.NET, этому будет посвящен следующий пример в лис-

тинге 16.8. Первым делом, создается объект `SqlConnection`, через который вставляется запись в таблицу `Customers`. Затем с помощью `SqlConnection` создается экземпляр класса `[Your]DataContext`. И, наконец, посредством ADO.NET вставленная ранее запись удаляется из таблицы `Customers`, таблица вновь запрашивается, на этот раз с применением LINQ to SQL, и результаты выводятся на консоль.

Листинг 16.8. Второй прототип конструктора `[Your]DataContext`, подключаемого с помощью разделяемого соединения ADO.NET

```
System.Data.SqlClient.SqlConnection sqlConn =
  new System.Data.SqlClient.SqlConnection(
  @"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind;Integrated Security=SSPI;");

string cmd = @"insert into Customers values ('LAWN', 'Lawn Wranglers',
  'Mr. Abe Henry', 'Owner', '1017 Maple Leaf Way', 'Ft. Worth', 'TX',
  '76104', 'USA', '(800) MOW-LAWN', '(800) MOW-LAWO')";

System.Data.SqlClient.SqlCommand sqlComm =
  new System.Data.SqlClient.SqlCommand(cmd);
sqlComm.Connection = sqlConn;

try
{
  sqlConn.Open();
  // Вставить запись.
  sqlComm.ExecuteNonQuery();

  Northwind db = new Northwind(sqlConn);
  IQueryable<Customer> query = from cust in db.Customers
                               where cust.Country == "USA"
                               select cust;

  Console.WriteLine("Заказчики после вставки, но перед удалением.");
  foreach (Customer c in query)
  {
    Console.WriteLine("{0}", c.CompanyName);
  }

  sqlComm.CommandText = "delete from Customers where CustomerID = 'LAWN'";
  // Удалить запись.
  sqlComm.ExecuteNonQuery();
  Console.WriteLine("{0}{0}Заказчики после удаления.", System.Environment.NewLine);
  foreach (Customer c in query)
  {
    Console.WriteLine("{0}", c.CompanyName);
  }
}
finally
{
  // Закрыть соединение.
  sqlComm.Connection.Close();
}
```

Обратите внимание, что запрос LINQ определен лишь однажды, но выполняется дважды — за счет двукратного перечисления возвращенной последовательности. Напомню, что из-за отложенного выполнения запроса определение запроса LINQ в действительности не запускает его выполнения. Запрос выполняется только при перечислении результатов. Это доказывает тот факт, что результаты отличаются между двумя перечислениями. Код в листинге 16.8 также демонстрирует замечательную интеграцию ADO.NET и LINQ to SQL и то, насколько хорошо они работают вместе. Вот результат:

```
Заказчики после вставки, но перед удалением.
Great Lakes Food Market
Hungry Coyote Import Store
Lawn Wranglers
Lazy K Kountry Store
Let's Stop N Shop
Lonesome Pine Restaurant
Old World Delicatessen
Rattlesnake Canyon Grocery
Save-a-lot Markets
Split Rail Beer & Ale
The Big Cheese
The Cracker Box
Trail's Head Gourmet Provisioners
White Clover Markets

Заказчики после удаления.
Great Lakes Food Market
Hungry Coyote Import Store
Lazy K Kountry Store
Let's Stop N Shop
Lonesome Pine Restaurant
Old World Delicatessen
Rattlesnake Canyon Grocery
Save-a-lot Markets
Split Rail Beer & Ale
The Big Cheese
The Cracker Box
Trail's Head Gourmet Provisioners
White Clover Markets
```

В примере применения третьего прототипа даже не будет использоваться сущностный класс `Northwind`. Предположим, что он отсутствует. Вместо этого применяется написанный вручную класс `Customer` и сокращенный файл отображения. По правде говоря, вручную написанный класс `Customer` — это сгенерированный SQLMetal класс `Customer`, который был несколько сокращен за счет удаления из него атрибутов LINQ to SQL. Взглянем на вручную написанный класс `Customer`.

Написанный вручную класс `Customer`

```
namespace Linqdev
{
  public partial class Customer
  {
    private string _CustomerID;
    private string _CompanyName;
    private string _ContactName;
    private string _ContactTitle;
    private string _Address;
    private string _City;
    private string _Region;
    private string _PostalCode;
    private string _Country;
    private string _Phone;
    private string _Fax;
    public Customer()
    {
    }
    public string CustomerID
    {
```

```
    get
    {
      return this._CustomerID;
    }
    set
    {
      if ((this._CustomerID != value))
      {
        this._CustomerID = value;
      }
    }
  }
  public string CompanyName
  {
    get
    {
      return this._CompanyName;
    }
    set
    {
      if ((this._CompanyName != value))
      {
        this._CompanyName = value;
      }
    }
  }
  public string ContactName
  {
    get
    {
      return this._ContactName;
    }
    set
    {
      if ((this._ContactName != value))
      {
        this._ContactName = value;
      }
    }
  }
  public string ContactTitle
  {
    get
    {
      return this._ContactTitle;
    }
    set
    {
      if ((this._ContactTitle != value))
      {
        this._ContactTitle = value;
      }
    }
  }
  public string Address
  {
    get
    {
      return this._Address;
    }
```

```
    set
    {
      if ((this._Address != value))
      {
        this._Address = value;
      }
    }
  }
  public string City
  {
    get
    {
      return this._City;
    }
    set
    {
      if ((this._City != value))
      {
        this._City = value;
      }
    }
  }
  public string Region
  {
    get
    {
      return this._Region;
    }
    set
    {
      if ((this._Region != value))
      {
        this._Region = value;
      }
    }
  }
  public string PostalCode
  {
    get
    {
      return this._PostalCode;
    }
    set
    {
      if ((this._PostalCode != value))
      {
        this._PostalCode = value;
      }
    }
  }
  public string Country
  {
    get
    {
      return this._Country;
    }
    set
    {
      if ((this._Country != value))
```

```
        {
          this._Country = value;
        }
      }
    }
    public string Phone
    {
      get
      {
        return this._Phone;
      }
      set
      {
        if ((this._Phone != value))
        {
          this._Phone = value;
        }
      }
    }
    public string Fax
    {
      get
      {
        return this._Fax;
      }
      set
      {
        if ((this._Fax != value))
        {
          this._Fax = value;
        }
      }
    }
  }
}
```

Наверное, это наихудший написанный вручную сущностный класс за все времена. Здесь не обрабатываются уведомления об изменениях, и удалена значительная часть кода, которая делала этот сущностный класс полноценным. О том, как правильно писать полноценные сущностные классы, рассказывается в главе 15.

В коде указано, что этот класс находится в пространстве имен `Linqdev`. Это важно, потому что понадобится не только указать это в коде примера, чтобы отличать данный класс `Customer` от того, который имеется в пространстве `nwind`, но это же пространство имен должно быть задано во внешнем файле отображения.

Однако что важнее всего в этом примере, так это наличие свойства для каждого поля базы данных, отображенного во внешнем файле. Теперь давайте взглянем на этот внешний файл отображения, который будет использоваться для этого примера.

Сокращенный внешний XML-файл отображения

```
<?xml version="1.0" encoding="utf-8"?>
<Database Name="Northwind"
  xmlns="http://schemas.microsoft.com/linqtosql/mapping/2007">
  <Table Name="dbo.Customers" Member="Customers">
    <Type Name="Linqdev.Customer">
      <Column Name="CustomerID" Member="CustomerID" Storage="_CustomerID"
        DbType="NChar(5) NOT NULL" CanBeNull="false" IsPrimaryKey="true" />
      <Column Name="CompanyName" Member="CompanyName" Storage="_CompanyName"
```

```
        DbType="NVarChar(40) NOT NULL" CanBeNull="false" />
      <Column Name="ContactName" Member="ContactName" Storage="_ContactName"
        DbType="NVarChar(30)" />
      <Column Name="ContactTitle" Member="ContactTitle" Storage="_ContactTitle"
        DbType="NVarChar(30)" />
      <Column Name="Address" Member="Address" Storage="_Address"
        DbType="NVarChar(60)" />
      <Column Name="City" Member="City" Storage="_City" DbType="NVarChar(15)" />
      <Column Name="Region" Member="Region" Storage="_Region"
        DbType="NVarChar(15)" />
      <Column Name="PostalCode" Member="PostalCode" Storage="_PostalCode"
        DbType="NVarChar(10)" />
      <Column Name="Country" Member="Country" Storage="_Country"
        DbType="NVarChar(15)" />
      <Column Name="Phone" Member="Phone" Storage="_Phone" DbType="NVarChar(24)" />
      <Column Name="Fax" Member="Fax" Storage="_Fax" DbType="NVarChar(24)" />
    </Type>
  </Table>
</Database>
```

Обратите внимание на указание, что класс `Customer`, на который выполняются отображения, находится в пространстве имен `Linqdev`.

Этот код XML помещен в файл по имени `abbreviatednorthwindmap.xml`, а этот файл — в каталог `bin\Debug`.

В листинге 16.9 данный вручную написанный класс `Customer` используется вместе с внешним файлом отображения для выполнения запроса LINQ to SQL без использования атрибутов.

Листинг 16.9. Третий прототип конструктора `DataContext`, подключаемого к базе данных и использующего файл отображения

```
string mapPath = "abbreviatednorthwindmap.xml";
XmlMappingSource nwindMap =
  XmlMappingSource.FromXml(System.IO.File.ReadAllText(mapPath));

DataContext db = new DataContext(
  @"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind;Integrated Security=SSPI;",
  nwindMap);

IQueryable<Linqdev.Customer> query =
  from cust in db.GetTable<Linqdev.Customer>()
  where cust.Country == "USA"
  select cust;
foreach (Linqdev.Customer c in query)
{
  Console.WriteLine("{0}", c.CompanyName);
}
```

На заметку! В этом примере файл `abbreviatednorthwindmap.xml` помещен в каталог `bin\Debug` проекта Visual Studio, поскольку выполняется компиляция и запуск с отладочной (Debug) конфигурацией.

Как видите, из файла отображения создается экземпляр объекта `XmlMappingSource`, который затем передается `XmlMappingSource` конструктору `DataContext`. Также обратите внимание, что нельзя просто обратиться к свойству `Customers Table<Customer>` в объекте `DataContext` для запроса LINQ to SQL, потому что вместо `[Your]DataContext` используется базовый класс `DataContext`, который здесь не существует.

Кроме того, везде, где производится ссылка на класс `Customer`, также явно указывается пространство `Linqdev` — просто чтобы иметь уверенность, что случайно не будет применен сгенерированный SQLMetal класс `Customer`, используемый большинством прочих примеров.

Вот результат запуска примера из листинга 16.9:

```
Great Lakes Food Market
Hungry Coyote Import Store
Lazy K Kountry Store
Let's Stop N Shop
Lonesome Pine Restaurant
Old World Delicatessen
Rattlesnake Canyon Grocery
Save-a-lot Markets
Split Rail Beer & Ale
The Big Cheese
The Cracker Box
Trail's Head Gourmet Provisioners
White Clover Markets
```

Хотя в этом примере используется низкоуровневый класс `Customer`, которому недостает большей части кода, необходимого полноценному сущностному классу, нужно было продемонстрировать пример использования файла отображения и класса, лишенного атрибутов LINQ to SQL.

Четвертый прототип является комбинацией второго и третьего. В листинге 16.10 показан пример его применения.

Листинг 16.10. Четвертый прототип конструктора `DataContext`, подключаемого к базе данных с помощью разделяемого соединения ADO.NET и использующего файл отображения

```
System.Data.SqlClient.SqlConnection sqlConn =
  new System.Data.SqlClient.SqlConnection(
  @"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind;Integrated Security=SSPI;");
string cmd = @"insert into Customers values ('LAWN', 'Lawn Wranglers',
  'Mr. Abe Henry', 'Owner', '1017 Maple Leaf Way', 'Ft. Worth', 'TX',
  '76104', 'USA', '(800) MOW-LAWN', '(800) MOW-LAWO')";
System.Data.SqlClient.SqlCommand sqlComm =
  new System.Data.SqlClient.SqlCommand(cmd);
sqlComm.Connection = sqlConn;
try
{
  sqlConn.Open();
  // Вставить запись.
  sqlComm.ExecuteNonQuery();
  string mapPath = "abbreviatednorthwindmap.xml";
  XmlMappingSource nwindMap =
    XmlMappingSource.FromXml(System.IO.File.ReadAllText(mapPath));
  DataContext db = new DataContext(sqlConn, nwindMap);
  IQueryable<Linqdev.Customer> query =
    from cust in db.GetTable<Linqdev.Customer>()
    where cust.Country == "USA"
    select cust;
  Console.WriteLine("Заказчики после вставки, но перед удалением.");
  foreach (Linqdev.Customer c in query)
  {
    Console.WriteLine("{0}", c.CompanyName);
  }
```

```
  sqlComm.CommandText = "delete from Customers where CustomerID = 'LAWN'";

  // Удалить запись.
  sqlComm.ExecuteNonQuery();
  Console.WriteLine("{0}{0}Заказчики после удаления.", System.Environment.NewLine);

  foreach (Linqdev.Customer c in query)
  {
    Console.WriteLine("{0}", c.CompanyName);
  }
}

finally
{
  // Закрыть соединение.
  sqlComm.Connection.Close();
}
```

Код в листинге 16.10 зависит от класса `Linqdev.Customer` и внешнего файла `abbreviatednorthwindmap.xml`, как и код в листинге 16.9.

Это замечательный пример использования LINQ to SQL для запроса базы данных без оснащенного атрибутами кода сущностного класса и интеграции с кодом ADO.NET. Результат именно такой, какой следовало ожидать:

```
Заказчики после вставки, но перед удалением.
Great Lakes Food Market
Hungry Coyote Import Store
Lawn Wranglers
Lazy K Kountry Store
Let's Stop N Shop
Lonesome Pine Restaurant
Old World Delicatessen
Rattlesnake Canyon Grocery
Save-a-lot Markets
Split Rail Beer & Ale
The Big Cheese
The Cracker Box
Trail's Head Gourmet Provisioners
White Clover Markets

Заказчики после удаления.
Great Lakes Food Market
Hungry Coyote Import Store
Lazy K Kountry Store
Let's Stop N Shop
Lonesome Pine Restaurant
Old World Delicatessen
Rattlesnake Canyon Grocery
Save-a-lot Markets
Split Rail Beer & Ale
The Big Cheese
The Cracker Box
Trail's Head Gourmet Provisioners
White Clover Markets
```

Как видно по предыдущим примерам, получить подключенный объект `DataContext` или `[Your]DataContext` совсем не сложно.

SubmitChanges()

Объект DataContext будет кэшировать все изменения, проведенные в сущностных объектах, до тех пор, пока не будет вызван метод SubmitChanges. Метод SubmitChanges инициирует запуск процессор изменений, и эти изменения в сущностных объектах будут сохранены в базе данных.

Если объекту DataContext недоступна объемлющая транзакция, которую можно было бы задействовать при вызове метода SubmitChanges, транзакция будет создана, и все изменения будут проведены в ее рамках. В этом случае, если транзакция потерпит неудачу, все ее изменения в базе данных будут отменены.

В случае возникновения конфликтов параллельного доступа будет сгенерировано исключение ChangeConflictException, что даст возможность попытаться решить конфликты и повторить попытку фиксации изменений. И что действительно хорошо — DataContext содержит коллекцию ChangeConflicts, которая предоставляет метод ResolveAll, чтобы выполнить такое разрешение конфликтов автоматически.

Конфликты параллельного доступа детально рассматриваются в главе 17.

Прототипы

Метод SybmitChanges имеет два прототипа, которые описаны ниже.

Первый прототип SubmitChanges

```
void SubmitChanges()
```

Этот прототип метода не принимает аргументов и по умолчанию устанавливает значение ConflictMode в FailOnFirstConflict.

Второй прототип SubmitChanges

```
void SubmitChanges(ConflictMode failureMode)
```

Этот прототип метода позволяет указывать ConflictMode. Его возможные значения: ConflictMode.FailOnFirstConflict и ConflictMode.ContinueOnConflict. Режим ConflictMode.FailOnFirstConflict вынуждает метод SubmitChanges генерировать исключение ChangeConflictException при самом первом обнаруженном конфликте. Режим ConflictMode.ContinueOnConflict пытается провести все обновления базы данных, чтобы, когда сгенерировано исключение ChangeConflictException, можно было получить отчет о них всех и разрешить за один прием.

Конфликты оцениваются в терминах количества конфликтующих записей, а не количества конфликтующих полей. Можно иметь два поля из одной записи, приведших к конфликту, но это вызовет только один конфликт.

Примеры

Поскольку во многих примерах в главе 14 вызывался метод SubmitChanges, тривиальный пример этого метода, вероятно, покажется старым знакомым. Вместо еще одного базового примера вызова метода SubmitChanges для простого сохранения изменений в базе данных предлагается рассмотреть несколько более сложный пример.

В примере вызова первого прототипа SubmitChanges будет сначала доказано, что изменения не проводятся в базе данных до тех пор, пока не будет вызван метод SubmitChanges. Поскольку этот пример сложнее многих предыдущих, он сопровождается дополнительными пояснениями. Код примера приведен в листинге 16.11.

Листинг 16.11. Пример вызова первого прототипа SubmitChanges

```
System.Data.SqlClient.SqlConnection sqlConn =
  new System.Data.SqlClient.SqlConnection(
```

```
  @"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind;Integrated Security=SSPI;");
try
{
  sqlConn.Open();
  string sqlQuery = "select ContactTitle from Customers where CustomerID = 'LAZYK'";
  string originalTitle = GetStringFromDb(sqlConn, sqlQuery);
  string title = originalTitle;
  Console.WriteLine("Звание из записи базы данных: {0}", title);
  Northwind db = new Northwind(sqlConn);
  Customer c = (from cust in db.Customers
                where cust.CustomerID == "LAZYK"
                select cust).
            Single<Customer>();
  Console.WriteLine("Звание из сущностного объекта: {0}", c.ContactTitle);
```

В приведенном коде создается и открывается подключение к базе данных ADO.NET. Затем в базе `ContactTitle` запрашивается заказчик `LAZYK`, используя общий метод `GetStringFromDb`, а результат выводится на консоль. Далее с использованием подключения к базе данных ADO.NET создается объект `Northwind`, запрашивается тот же самый заказчик с помощью LINQ to SQL и отображается информация `ContactTitle`. В этой точке два полученных `ContactTitle` должны совпадать.

```
  Console.WriteLine(String.Format(
    "{0}Изменить звание на 'Director of Marketing' в сущностном объекте:",
    System.Environment.NewLine));
  c.ContactTitle = "Director of Marketing";
  title = GetStringFromDb(sqlConn, sqlQuery);
  Console.WriteLine("Звание из записи базы данных: {0}", title);

  Customer c2 = (from cust in db.Customers
                 where cust.CustomerID == "LAZYK"
                 select cust).
                 Single<Customer>();
  Console.WriteLine("Звание из сущностного объекта: {0}", c2.ContactTitle);
```

В этом фрагменте кода значение `ContactTitle` сущностного объекта LINQ to SQL, представляющего заказчика, изменяется. Затем значения `ContactTitle` запрашиваются из базы данных и из сущностного объекта, после чего отображаются на консоли. На этот раз значения `ContactTitle` не должны совпадать, потому что изменения еще не были сохранены в базе данных.

```
  db.SubmitChanges();
  Console.WriteLine(String.Format(
    "{0}Вызван метод SubmitChanges().",
    System.Environment.NewLine));
  title = GetStringFromDb(sqlConn, sqlQuery);
  Console.WriteLine("Звание из записи базы данных: {0}", title);

  Console.WriteLine("Восстановление исходного значение ContactTitle...");
  c.ContactTitle = "Marketing Manager";
  db.SubmitChanges();
  Console.WriteLine("Значение ContactTitle восстановлено.");
}
finally
{
  sqlConn.Close();
}
```

В предыдущем коде вызывается метод SubmitChanges и вновь извлекается ContactTitle, чтобы отобразить его снова. На этот раз значение в базе данных должно быть обновлено, потому что метод SubmitChanges сохранил изменения в базе.

И, наконец, ContactTitle устанавливается обратно в исходное значение и сохраняется в базе данных с использованием метода SubmitChanges. Это позволит вернуть базу данных в исходное состояние, чтобы пример можно было запускать несколько раз и это никак не повлияло на остальные примеры.

Этот код много чего делает, но его цель — доказать, что изменения, проведенные в сущностном объекте, не сохраняются в базе данных до тех пор, пока не будет вызван метод SubmitChanges. Когда вы видите вызов метода GetStringFromDb, это значит, что он извлекает ContactTitle непосредственно из базы данных с использованием ADO.NET. Ниже показаны результаты выполнения кода.

```
Звание из записи базы данных: Marketing Manager
Звание из сущностного объекта: Marketing Manager

Изменить звание на 'Director of Marketing' в сущностном объекте:
Звание из записи базы данных: Marketing Manager
Звание из сущностного объекта: Director of Marketing

Вызван метод SubmitChanges().
Звание из записи базы данных: Director of Marketing
Восстановление исходного значение ContactTitle...
Значение ContactTitle восстановлено.
```

Как видите, значение ContactTitle не изменялось в базе данных до тех пор, пока не был вызван метод SubmitChanges.

В примере применения второго прототипа SubmitChanges намеренно вызываются ошибки параллельного доступа к двум записям за счет их обновления через ADO.NET между моментом запроса записей и моментом их обновления с помощью LINQ to SQL. Будут созданы *две* конфликтующих записи, чтобы продемонстрировать разницу между ConflictMode.FailOnFirstConflict и ConflictMode.ContinueOnConflict.

Также далее будет показан код, который вернет в базе данных значения ContactTitle в исходное состояние. Это позволит запускать пример многократно. Если вы прекратите выполнение кода в отладчике, то может понадобиться вручную сбросить эти значения.

В первом примере второго прототипа метода SubmitChanges в листинге 16.12 для ConflictMode указывается ContinueOnConflict, чтобы сначала можно было увидеть, как он обрабатывает множественные конфликты. Поскольку пример довольно сложен, он сопровождается дополнительными пояснениями.

Листинг 16.12. Второй прототип SubmitChanges, демонстрирующий ContinueOnConflict

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");

Console.WriteLine("Запрос заказчика LAZYK с помощью LINQ.");
Customer cust1 = (from c in db.Customers
                  where c.CustomerID == "LAZYK"
                  select c).Single<Customer>();

Console.WriteLine("Запрос заказчика LONEP с помощью LINQ.");
Customer cust2 = (from c in db.Customers
                  where c.CustomerID == "LONEP"
                  select c).Single<Customer>();
```

В этом коде создается объект Northwind типа DataContext и запрашиваются два заказчика — LAZYK и LONEP.

```
string cmd = @"update Customers set ContactTitle = 'Director of Marketing'
               where CustomerID = 'LAZYK';
             update Customers set ContactTitle = 'Director of Sales'
               where CustomerID = 'LONEP'";
ExecuteStatementInDb(cmd);
```

Далее для обоих заказчиков обновляется значение ContactValue в базе данных с помощью общего метода ExecuteStatementInDb, который использует ADO.NET для проведения изменений. В этой точке создается потенциальная возможность конфликта параллельного доступа для каждой записи.

```
Console.WriteLine("Изменение ContactTitle в сущностных объектах для LAZYK и LONEP.");
cust1.ContactTitle = "Vice President of Marketing";
cust2.ContactTitle = "Vice President of Sales";
```

В этой части кода обновляется ContactTitle для каждого заказчика, так что при вызове метода SubmitChanges в следующей части процессор изменений объекта DataContext попытается сохранить изменения для этих двух заказчиков и обнаружить конфликты параллельного доступа.

```
try
{
  Console.WriteLine("Вызов SubmitChanges()...");
  db.SubmitChanges(ConflictMode.ContinueOnConflict);
  Console.WriteLine("Метод SubmitChanges() вызван успешно.");
}
```

Здесь вызывается метод SubmitChanges. Это заставит процессор изменений объекта DataContext попытаться сохранить этих двух заказчиков, но поскольку значение каждого ContactTitle каждого заказчика в базе данных отличается от того, что было изначально из нее загружено, будет обнаружен конфликт параллельного доступа.

```
catch (ChangeConflictException ex)
{
  Console.WriteLine("Во время вызова SubmitChanges() обнаружен(ы) конфликт(ы): {0}.",
    ex.Message);
  foreach (ObjectChangeConflict objectConflict in db.ChangeConflicts)
  {
    Console.WriteLine("Возник конфликт для {0}.",
      ((Customer)objectConflict.Object).CustomerID);
    foreach (MemberChangeConflict memberConflict in objectConflict.MemberConflicts)
    {
      Console.WriteLine("  Значение LINQ = {0}{1}  Значение базы данных = {2}",
      memberConflict.CurrentValue,
      System.Environment.NewLine,
      memberConflict.DatabaseValue);
    }
  }
}
```

В этом фрагменте перехватывается исключение ChangeConflictException. Именно здесь начинается самое интересное. Обратите внимание, что сначала выполняется перечисление коллекции ChangeConflicts объекта DataContext по имени db. Эта коллекция будет хранить объекты ObjectChangeConflict. Объекты ObjectChangeConflict имеют свойство по имени Object, ссылающееся на действительный сущностный объект, при сохранении которого случается конфликт параллельного доступа. В коде просто выполняется приведение члена Object к типу данных сущностного класса, чтобы обратиться к значениям свойств сущностного объекта. В данном случае производится обращение к свойству CustomerID.

Затем для каждого объекта ObjectChangeConflict производится перечисление его коллекции объектов MemberChangeConflict с отображением информации для каждого из тех, которые интересуют. В данном случае на консоль выводится значение LINQ и значение из базы данных.

```
Console.WriteLine("{0}Сброс данных к исходным значениям.",
  System.Environment.NewLine);
cmd = @"update Customers set ContactTitle = 'Marketing Manager'
          where CustomerID = 'LAZYK';
        update Customers set ContactTitle = 'Sales Manager'
          where CustomerID = 'LONEP'";
ExecuteStatementInDb(cmd);
```

В этом фрагменте кода база данных просто восстанавливается в ее исходное состояние, чтобы пример можно было запускать много раз.

Был приведен довольно большой объем кода. Имейте в виду, что ни одно из этих перечислений по различным коллекциям конфликтов не является необходимым. Здесь просто показано, как это можно делать, и предоставлена доступная информация о конфликтах, которую можно видеть.

Также обратите внимание, что в этом примере ничего не делается в плане разрешения конфликтов. О них просто сообщается.

Ниже показан результат выполнения этого кода:

```
Запрос заказчика LAZYK с помощью LINQ.
Запрос заказчика LONEP с помощью LINQ.
Выполнение оператора SQL для базы данных с помощью ADO.NET...
База данных обновлена.
Изменение ContactTitle в сущностных объектах для LAZYK и LONEP.
Вызов SubmitChanges()...
Во время вызова SubmitChanges() обнаружен(ы) конфликт(ы): 2 of 2 updates failed.
Возник конфликт для LAZYK.
  Значение LINQ = Vice President of Marketing
  Значение базы данных = Director of Marketing
Возник конфликт для LONEP.
  Значение LINQ = Vice President of Sales
  Значение базы данных = Director of Sales

Сброс данных к исходным значениям.
Выполнение оператора SQL для базы данных с помощью ADO.NET...
База данных обновлена.
```

Как видите, было обнаружено два конфликта — по одному для каждой из двух записей, для которых был создан конфликт. Это демонстрирует, что процессор изменений не прекратил попыток сохранить изменения в базе данных после того, как произошел первый конфликт. Это связано с тем, что при вызове SubmitChanges было передано значение ConflictMode, равное ContinueOnConflict.

В листинге 16.13 содержится такой же код, за исключением того, что в нем при вызове SubmitChanges передается значение ConflictMode, равное FailOnFirstConflict.

Листинг 16.13. Второй прототип SubmitChanges, демонстрирующий FailOnFirstConflict

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
Console.WriteLine("Запрос заказчика LAZYK с помощью LINQ.");
Customer cust1 = (from c in db.Customers
                  where c.CustomerID == "LAZYK"
                  select c).Single<Customer>();
Console.WriteLine("Запрос заказчика LONEP с помощью LINQ.");
```

```
Customer cust2 = (from c in db.Customers
                  where c.CustomerID == "LONEP"
                  select c).Single<Customer>();
string cmd = @"update Customers set ContactTitle = 'Director of Marketing'
                where CustomerID = 'LAZYK';
               update Customers set ContactTitle = 'Director of Sales'
                where CustomerID = 'LONEP'";
ExecuteStatementInDb(cmd);

Console.WriteLine("Изменение ContactTitle в сущностных объектах для LAZYK и LONEP.");
cust1.ContactTitle = "Vice President of Marketing";
cust2.ContactTitle = "Vice President of Sales";

try
{
  Console.WriteLine("Вызов SubmitChanges()...");
  db.SubmitChanges(ConflictMode.FailOnFirstConflict);
  Console.WriteLine("Метод SubmitChanges() вызван успешно.");
}

catch (ChangeConflictException ex)
{
  Console.WriteLine("Во время вызова SubmitChanges() обнаружен(ы) конфликт(ы): {0}",
    ex.Message);
  foreach (ObjectChangeConflict objectConflict in db.ChangeConflicts)
  {
    Console.WriteLine("Возник конфликт для {0}.",
      ((Customer)objectConflict.Object).CustomerID);
    foreach (MemberChangeConflict memberConflict in objectConflict.MemberConflicts)
    {
      Console.WriteLine("  Значение LINQ = {0}{1}  Значение базы данных = {2}",
      memberConflict.CurrentValue,
      System.Environment.NewLine,
      memberConflict.DatabaseValue);
    }
  }
}
Console.WriteLine("{0}Сброс данных к исходным значениям.",
  System.Environment.NewLine);

cmd = @"update Customers set ContactTitle = 'Marketing Manager'
          where CustomerID = 'LAZYK';
        update Customers set ContactTitle = 'Sales Manager'
          where CustomerID = 'LONEP'";
ExecuteStatementInDb(cmd);
```

На этот раз результат должен указывать на то, что обработка изменений в сущностных объектах останавливается, как только обнаружен первый конфликт параллельного доступа. Взглянем на результат:

```
Запрос заказчика LAZYK с помощью LINQ.
Запрос заказчика LONEP с помощью LINQ.
Выполнение оператора SQL для базы данных с помощью ADO.NET...
База данных обновлена.
Изменение ContactTitle в сущностных объектах для LAZYK и LONEP.
Вызов SubmitChanges()...
Во время вызова SubmitChanges() обнаружен(ы) конфликт(ы): Row not found or
changed.
Возник конфликт для LAZYK.
  Значение LINQ = Vice President of Marketing
  Значение базы данных = Director of Marketing
```

```
Сброс данных к исходным значениям.
Выполнение оператора SQL для базы данных с помощью ADO.NET...
База данных обновлена.
```

Как видите, даже несмотря на то, что было инициировано два конфликта, процессор изменений прекращает попытки сохранить изменения в базе данных, как только возникает конфликт, о чем свидетельствует единственное сообщение о конфликте.

DatabaseExists()

Метод DatabaseExists может использоваться для определения существования базы данных. Определение существования базы данных основано на строке соединения, указанной при создании экземпляра DataContext. Если задается путь к файлу .mdf, метод ищет базу данных в пути с указанным именем. Если же задается сервер, метод проверит сервер.

Метод DatabaseExists часто применяется в сочетании с методами DeleteDatabase и CreateDatabase.

Прототипы

Метод DatabaseExists имеет один прототип, описанный ниже.

Единственный прототип DatabaseExists

```
bool DatabaseExists()
```

Этот метод вернет true, если база данных, указанная в строке соединения, существует при создании экземпляра DataContext. В противном случае он вернет false.

Примеры

К счастью, этот метод продемонстрировать очень просто. В листинге 16.14 просто создается экземпляр DataContext и вызывается метод DatabaseExists, чтобы посмотреть, существует ли база данных Northwind. И, конечно же, известно, что эта база существует.

Листинг 16.14. Пример вызова метода DatabaseExists

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
Console.WriteLine("База данных Northwind {0}.",
   db.DatabaseExists() ? "существует" : "не существует");
```

Вот результат:

```
База данных Northwind существует.
```

Если отключить базу данных Northwind и запустить пример снова, получится следующий результат:

```
База данных Northwind не существует.
```

Не забудьте затем вновь подключить базу данных Northwind, чтобы могли работать другие примеры.

CreateDatabase()

Поскольку сущностные классы знают настолько много о структуре базы данных, на которую они отображаются, был предоставлен метод CreateDatabase, предназначенный для создания базы данных.

Однако следует осознавать, что он может создавать только те части базы данных, о которых ему известно через сущностные классы или файл отображения. Поэтому *содержимое* таких вещей, как хранимые процедуры, триггеры, определенные пользователем функции и ограничения, не будет включено в базу данных, создаваемую подобным способом, поскольку нет атрибутов, задающих эту информацию. Тем не менее, для простого приложения это может оказаться вполне приемлемым.

На заметку! В отличие от большинства прочих изменений, которые вносятся в базу данных через `DataContext`, метод `CreateDatabase` выполняется немедленно. Нет необходимости вызывать метод `SubmitChanges`, и выполнение не является отложенным. Это предоставляет преимущество в виде возможности создать базу данных и начать вставлять в нее данные немедленно.

Прототипы

Метод `CreateDatabase` имеет один прототип, описанный ниже.

Единственный прототип `CreateDatabase`

```
void CreateDatabase()
```

Этот метод не принимает аргументов и ничего не возвращает.

Примеры

Применение этого метода продемонстрировать просто, и листинг 16.15 доказывает это.

Листинг 16.15. Пример вызова метода `CreateDatabase`

```
Northwind db = new Northwind(@"C:\Northwnd.mdf");
db.CreateDatabase();
```

На заметку! В листинге 16.15 намеренно записано имя файла `Northwnd` без буквы *i*, чтобы это не затронуло базу Northwind, если она установлена.

Этот код не производит никакого экранного вывода, так что нет результатов, которые стоило бы показать. Однако если заглянуть в каталог `C:\`, там можно увидеть файлы `Northwnd.mdf` и `Northwnd.ldf`. Кроме того, в окне SQL Server Management Studio можно увидеть присоединенный файл `C:\Northwnd.mdf`. Этот метод лучше всего сочетать с методом `DatabaseExists`. Если попытаться вызвать метод `CreateDatabase`, когда база данных уже существует, будет сгенерировано исключение. Чтобы продемонстрировать сказанное, просто запустите код из листинга 16.15 второй раз, не удаляя или не отключая файл `C:\Northwnd.mdf` с помощью SQL Server Management Studio или Enterprise Manager. Получится следующий вывод:

```
Unhandled Exception: System.Data.SqlClient.SqlException: Database 'C:\Northwnd.mdf'
already exists. Choose a different database name.
```

Необработанное исключение: System.Data.SqlClient.SqlException: База данных 'C:\Northwnd.mdf' уже существует. Выберите другое имя для базы данных.

...

Кроме того, не допускайте ошибку, предполагая, что можно просто удалить два созданных файла базы данных Northwind из файловой системы, чтобы избавиться от базы данных и получить возможность запустить пример снова. SQL Server по-прежнему будет содержать их в своем каталоге. Чтобы второй запуск метода `CreateDatabase` удался, необходимо в правильном порядке отсоединить и удалить базу данных.

Можно удалить или отключить вновь созданную базу данных, чтобы предотвратить путаницу в будущем, или же просто оставить ее на месте для того, чтобы ее удалил следующий пример, показанный в листинге 16.16.

DeleteDatabase()

LINQ to SQL предоставляет возможность удаления базы данных с помощью метода `DeleteDatabase` объекта `DataContext`. Попытка удалить базу данных, которая не существует, приведет к генерации исключения, так что этот метод лучше вызывать после проверки существования базы данных с помощью метода `DatabaseExists`.

Внимание! В отличие от большинства других изменений, которые вносятся в базу данных через `DataContext`, метод `DeleteDatabase` выполняется немедленно. Нет необходимости вызывать `SubmitChanges`, и выполнение не является отложенным.

Прототипы

Метод `DeleteDatabase` имеет единственный прототип, описанный ниже.

Единственный прототип `DeleteDatabase`

```
void DeleteDatabase()
```

Этот метод не принимает аргументов и ничего не возвращает.

Примеры

В листинге 16.16 удаляется база данных, которая была создана в листинге 16.15.

Листинг 16.16. Пример вызова метода `DeleteDatabase`

```
Northwind db = new Northwind(@"C:\Northwnd.mdf");
db.DeleteDatabase();
```

Этот пример не создает никакого экранного вывода при запуске — до тех пор, пока указанная база данных существует, но после запуска его обнаружится, что два файла базы данных, которые были созданы вызовом `CreateDatabase`, исчезли.

Вызов этого метода при несуществующей базе данных вызовет генерацию следующего исключения:

```
Unhandled Exception: System.Data.SqlClient.SqlException: An attempt to attach
an auto-named database for file C:\Northwnd.mdf failed. A database with the same
name exists, or specified file cannot be opened, or it is located on UNC share.
```

Необработанное исключение: System.Data.SqlClient.SqlException: Попытка присоединения к автоматически именованной базе данных для файла C:\Northwnd.mdf завершилась сбоем. База данных с тем же именем уже существует, или указанный файл не может быть открыт, или он расположен на разделяемом ресурсе UNC.

...

CreateMethodCallQuery()

Первое, что нужно знать о методе `CreateMethodCallQuery` — это защищенный (`protected`) метод. Значит, он не может быть вызван в коде приложения, и чтобы получить такую возможность, понадобится унаследовать свой класс от `DataContext`.

Метод `CreateMethodCallQuery` используется для вызова возвращающих табличные значения пользовательских функций. Метод `ExecuteMethodCall` применяется для вызова пользовательских функций, возвращающих скалярные значения, и он рассматривается далее в этой главе.

Прототипы

Метод CreateMethodCallQuery имеет один прототип, описанный ниже.

Единственный прототип CreateMethodCallQuery

```
protected internal IQueryable<T> CreateMethodCallQuery<T>(
  object instance,
  System.Reflection.MethodInfo methodInfo,
  params object[] parameters)
```

Метод CreateMethodCallQuery принимает ссылку на объект DataContext или [Your] DataContext, членом которого является метод, вызывающий CreateMethodCallQuery, объект MethodInfo для этого вызывающего метода и массив params параметров для возвращающей табличное значение пользовательской функции.

Примеры

Поскольку метод CreateMethodCallQuery является protected и может быть вызван только из класса DataContext или его наследника, вместо представления примера, который действительно вызывает метод CreateMethodCallQuery, рассмотрим метод, сгенерированный SQLMetal для пользовательской функции, которая возвращает табличное значение ProductsUnderThisUnitPrice из расширенной базы данных Northwind.

Сгенерированный SQLMetal метод, вызывающий CreateMethodCallQuery

```
[Function(Name="dbo.ProductsUnderThisUnitPrice", IsComposable=true)]
public IQueryable<ProductsUnderThisUnitPriceResult>
  ProductsUnderThisUnitPrice(
    [Parameter(DbType="Money")] System.Nullable<decimal> price)
{
  return this.CreateMethodCallQuery<ProductsUnderThisUnitPriceResult>(
    this, ((MethodInfo)(MethodInfo.GetCurrentMethod())), price);
}
```

В приведенном выше коде видно, что метод ProductsUnderThisUnitPrice снабжен атрибутом Function, поэтому известно, что он собирается вызывать либо хранимую процедуру, либо определяемую пользователем функцию по имени ProductsUnderThisUnitPrice. Поскольку свойство IsComposable атрибута установлено в true, это определяемая пользователем функция, а не хранимая процедура. Так как сгенерированный код вызывает метод CreateMethodCallQuery, это значит, что указанная пользовательская функция ProductsUnderThisUnitPrice возвращает табличное значение, а не *скалярное*.

Из аргументов, переданных методу CreateMethodCallQuery, первым аргументом является ссылка на производный от DataContext класс, сгенерированный SQLMetal. Второй аргумент — объект MethodInfo текущего метода. Это позволит методу CreateMethodCallQuery получить доступ к атрибутам, так что он будет иметь необходимую информацию для вызова определяемой пользователем функции, возвращающей *табличное* значение, такую как ее имя. Третий аргумент, переданный методу CreateMethodCallQuery — это единственный параметр, принимаемый указанной пользовательской функцией, которым в данном случае является цена.

Значение, возвращенное вызовом метода CreateMethodCallQuery, будет возвращено методом ProductsUnderThisUnitPrice, и им будет последовательность объектов ProductsUnderThisUnitPriceResult. Инструмент SQLMetal также сгенерировал класс ProductsUnderThisUnitPriceResult.

Код, о котором шла речь, показывает, как вызывать метод `CreateMethodCallQuery`, но чтобы задать некоторый контекст, давайте рассмотрим пример вызова сгенерированного метода `ProductsUnderThisUnitPriceResult` в действии.

Листинг 16.17. Пример вызова метода `ProductsUnderThisUnitPrice`

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");

IQueryable<ProductsUnderThisUnitPriceResult> results =
  db.ProductsUnderThisUnitPrice(new Decimal(5.50));
foreach(ProductsUnderThisUnitPriceResult prod in results)
{
  Console.WriteLine("{0} - {1:C}", prod.ProductName, prod.UnitPrice);
}
```

Вот результат работы этого примера:

```
Guaraná Fantástica - $4.50
Geitost - $2.50
```

ExecuteQuery()

Нет сомнений, что API-интерфейс LINQ to SQL великолепен. Использование стандартной точечной нотации LINQ или синтаксиса выражений превращает построение запросов LINQ в пару пустяков. Но рано или поздно возникает потребность просто выполнить запрос SQL. С помощью LINQ to SQL можно делать и это. На самом деле так можно поступить и получить в результате те же сущностные объекты.

Метод `ExecuteQuery` позволяет указать запрос SQL как строку и даже включать в нее параметры для подстановки, как это делается при вызове метода `String.Format`, причем полученный результат будет транслирован в последовательность сущностных объектов.

Это очень просто. Возможно, вы спросите, а как насчет ошибок внедрения SQL? Разве соответствующий способ делать это требует использования параметров? Да, требует. И метод `ExecuteQuery` позаботится обо всем этом. Давайте рассмотрим пример.

Прототипы

Метод `ExecuteQuery` имеет один прототип, описанный ниже.

Единственный прототип `ExecuteQuery`

```
IEnumerable<T> ExecuteQuery<T>(string query, params object[] parameters)
```

Этот метод принимает, как минимум, один аргумент — запрос SQL и ноль или более параметров. Строка запроса и необязательные параметры работают как метод `String.Format`. Метод возвращает последовательность типа `T`, где `T` — сущностный класс.

Следует помнить, что если указывается значение столбца в конструкции `where` самой строки запроса, то столбцы символьного типа должны заключаться в одиночные кавычки, как это принято в нормальном запросе SQL. Но если значение столбца подставляется как параметр, то нет необходимости заключать спецификатор параметра, такой как `{0}`, в одиночные кавычки.

Чтобы столбец в запросе распространился на сущностный объект, имя столбца должно соответствовать одному из отображенных полей сущностного объекта. Разумеется, этого можно достичь, добавляя "`as <имя_столбца>`" к действительному имени столбца, где `<имя_столбца>` — отображенный в сущностном объекте столбец.

Каждое отображенное поле не обязательно должно быть возвращено запросом, но первичные ключи — безусловно, должны. В запросе можно извлекать поля, которые не

отображаются ни на одно поле сущностного объекта, но они не будут распространяться на сущностный объект.

Примеры

Для демонстрации простого примера вызова метода `ExecuteQuery` в листинге 16.18 обратимся к таблице `Customers`.

Листинг 16.18. Простой пример вызова метода `ExecuteQuery`

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");

IEnumerable<Customer> custs = db.ExecuteQuery<Customer>(
  @"select CustomerID, CompanyName, ContactName, ContactTitle
    from Customers where Region = {0}", "WA");
foreach (Customer c in custs)
{
  Console.WriteLine("ID = {0} : Name = {1} : Contact = {2}",
    c.CustomerID, c.CompanyName, c.ContactName);
}
```

Этот пример очень прост. Обратите внимание, что поскольку используется средство подстановки параметров метода за счет указания в качестве параметра `"WA"` вместо жесткого кодирования его в запросе, спецификатор формата незачем заключать в одиночные кавычки. Вот результат:

```
ID = LAZYK : Name = Lazy K Kountry Store : Contact = John Steel
ID = TRAIH : Name = Trail's Head Gourmet Provisioners : Contact = Helvetius Nagy
ID = WHITC : Name = White Clover Markets : Contact = Karl Jablonski
```

Если бы нужно было выполнить тот же запрос, но без подстановки параметров, понадобилось бы заключить часть `"WA"` в одиночные кавычки, как это делается в нормальном SQL-запросе. Соответствующий код показан в листинге 16.19.

Листинг 16.19. Другой простой пример вызова метода `ExecuteQuery`

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");

IEnumerable<Customer> custs = db.ExecuteQuery<Customer>(
  @"select CustomerID, CompanyName, ContactName, ContactTitle
    from Customers where Region = 'WA'");
foreach (Customer c in custs)
{
  Console.WriteLine("ID = {0} : Name = {1} : Contact = {2}",
    c.CustomerID, c.CompanyName, c.ContactName);
}
```

В случае если вы сразу не заметили — `WA` теперь заключено в одиночные кавычки в строке запроса. Результат выполнения этого кода такой же, как и предыдущего:

```
ID = LAZYK : Name = Lazy K Kountry Store : Contact = John Steel
ID = TRAIH : Name = Trail's Head Gourmet Provisioners : Contact = Helvetius Nagy
ID = WHITC : Name = White Clover Markets : Contact = Karl Jablonski
```

К этому еще можно добавить указанное имя столбца, если реальное его имя не соответствует имени столбца в базе данных. Поскольку в строке запроса допускается выполнять соединения, можно опрашивать столбцы с разными именами из разных таблиц, но указывать их имя как одно из отображенных полей сущностного класса. Пример этого представлен в листинге 16.20.

Листинг 16.20. Пример вызова метода `ExecuteQuery` с указанием имени отображенного поля

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");

IEnumerable<Customer> custs = db.ExecuteQuery<Customer>(
  @"select CustomerID, Address + ', ' + City + ', ' + Region as Address
    from Customers where Region = 'WA'");
foreach (Customer c in custs)
{
  Console.WriteLine("Id = {0} : Address = {1}",
    c.CustomerID, c.Address);
}
```

Интересная часть этого примера касается соединения нескольких столбцов базы данных и строковых литералов с указанием имени отображенного поля, чтобы получить адрес, город и регион в единственном члене `Address` сущностного объекта. В этом случае все поля взяты из одной таблицы, но они вполне могли поступать из соединения с другой таблицей. Вот результат:

```
Id = LAZYK : Address = 12 Orchestra Terrace, Walla Walla, WA
Id = TRAIH : Address = 722 DaVinci Blvd., Kirkland, WA
Id = WHITC : Address = 305 - 14th Ave. S. Suite 3B, Seattle, WA
```

Конечно, применяя такой трюк, не забывайте о том, что если один из возвращенных сущностных объектов будет модифицирован, а затем будет вызван метод `SubmitChanges`, могут быть получены некоторые записи базы данных с сомнительной информацией. Но при правильном применении такой прием может быть весьма удобным.

Translate()

Метод `Translate` подобен `ExecuteQuery` в том, что он транслирует результаты SQL-запроса в последовательность сущностных объектов. Отличие заключается в том, что вместо передачи строки, содержащей оператор SQL, ему передается объект типа `System.Data.Common.DbDataReader`, такой как `SqlDataReader`. Этот метод удобен для интеграции кода LINQ to SQL с существующим кодом ADO.NET.

Прототипы

Метод `Translate` имеет один прототип, описанный ниже.

Единственный прототип `Translate`

```
IEnumerable<T> Translate<T>(System.Data.Common.DbDataReader reader)
```

Методу `Translate` передается объект типа `System.Data.Common.DbDataReader`, а возвращает он последовательность сущностных объектов.

Примеры

В листинге 16.21 создается запрос с помощью ADO.NET. Затем применяется метод `Translate` для трансляции результатов запроса в последовательность сущностных объектов `Customer`. Поскольку листинг 16.21 сложнее, чем обычно, будут даны дополнительные пояснения.

Листинг 16.21. Пример вызова метода `Translate`

```
System.Data.SqlClient.SqlConnection sqlConn =
  new System.Data.SqlClient.SqlConnection(
  @"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind;Integrated Security=SSPI;");
```

```
string cmd = @"select CustomerID, CompanyName, ContactName, ContactTitle
            from Customers where Region = 'WA'";

System.Data.SqlClient.SqlCommand sqlComm =
  new System.Data.SqlClient.SqlCommand(cmd);

sqlComm.Connection = sqlConn;
try
{
  sqlConn.Open();
  System.Data.SqlClient.SqlDataReader reader = sqlComm.ExecuteReader();
```

Для этого примера давайте предположим, что весь предыдущий код уже существует. Представим, что это унаследованный код, который нужно обновить, и для решения задачи желательно воспользоваться преимуществом LINQ. Как видите, в предыдущем коде нет никаких ссылок на LINQ. Подключение `SqlConnection` установлено, запрос сформирован, команда `SqlCommand` создана, соединение открыто, а запрос выполнен — все так, как в самом обычном запросе ADO.NET. Теперь добавим некоторый код LINQ.

```
Northwind db = new Northwind(sqlConn);
IEnumerable<Customer> custs = db.Translate<Customer>(reader);
foreach (Customer c in custs)
{
  Console.WriteLine("ID = {0} : Name = {1} : Contact = {2}",
    c.CustomerID, c.CompanyName, c.ContactName);
}
```

В этом коде создается экземпляр `DataContext` по имени `Northwind` с использованием объекта `SqlConnection` из ADO.NET. Затем вызывается метод `Translate` с передачей ему уже созданного объекта `reader`, чтобы результаты запроса могли быть преобразованы в сущностные объекты, которые можно перечислять и отображать результаты.

Обычно, поскольку этот код унаследован, в нем присутствует еще что-то, что делается с результатами, но в рассматриваемом примере это не обязательно. Все, что остается добавить — код очистки метода:

```
}
finally
{
  sqlComm.Connection.Close();
}
```

Предыдущий код просто закрывает соединение. Этот пример демонстрирует, насколько изящно LINQ to SQL может взаимодействовать с ADO.NET. Взглянем на результаты работы кода из листинга 16.21:

```
ID = LAZYK : Name = Lazy K Kountry Store : Contact = John Steel
ID = TRAIH : Name = Trail's Head Gourmet Provisioners : Contact = Helvetius Nagy
ID = WHITC : Name = White Clover Markets : Contact = Karl Jablonski
```

ExecuteCommand()

Подобно `ExecuteQuery`, метод `ExecuteCommand` позволяет указывать действительный оператор SQL, чтобы выполнить его в базе данных. Это значит, что его можно использовать для выполнения операторов вставки, обновления или удаления, а также для вызова хранимых процедур. Кроме того, как и методу `ExecuteQuery`, ему можно передавать параметры.

При вызове метода `ExecuteCommand` следует помнить об одной вещи: он выполняется немедленно, потому вызывать метод `SubmitChanges` незачем.

Прототипы

Метод ExecuteCommand имеет единственный прототип, описанный ниже.

Единственный прототип ExecuteCommand

```
int ExecuteCommand(string command, params object[] parameters)
```

Этот метод принимает строку command и ноль или более необязательных параметров, и возвращает целое число, указывающее количество строк, затронутых запросом.

Имейте в виду, что если указывается значение столбца в конструкции where самой строки запроса, то столбцы символьного типа должны быть заключены в одиночные кавычки, как это принято в нормальном запросе SQL. Но если подставляется значение столбца как параметр, то нет необходимости заключать спецификатор параметра, такой как {0}, в одиночные кавычки.

Примеры

В листинге 16.22 вставляется запись с использованием метода ExecuteCommand. Поскольку изменения, вносимые в базу данных, всегда отменяются, чтобы не пострадали последующие примеры, здесь также применяется метод ExecuteCommand для удаления вставленной записи.

Листинг 16.22. Пример использования метода ExecuteCommand для вставки и удаления записи

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");

Console.WriteLine("Вставка заказчика...");
int rowsAffected = db.ExecuteCommand(
  @"insert into Customers values ({0}, 'Lawn Wranglers',
    'Mr. Abe Henry', 'Owner', '1017 Maple Leaf Way', 'Ft. Worth', 'TX',
    '76104', 'USA', '(800) MOW-LAWN', '(800) MOW-LAWO')",
  "LAWN");

Console.WriteLine("Вставка завершена.{0}", System.Environment.NewLine);

Console.WriteLine("Оказано влияние на {0} строк. Находится ли заказчик в базе данных?",
  rowsAffected);

Customer cust = (from c in db.Customers
                 where c.CustomerID == "LAWN"
                 select c).DefaultIfEmpty<Customer>().Single<Customer>();
Console.WriteLine("{0}{1}",
  cust != null ?
    "Да, находится." : "Нет, не находится.",
  System.Environment.NewLine);
Console.WriteLine("Удаление заказчика...");
rowsAffected =
  db.ExecuteCommand(@"delete from Customers where CustomerID = {0}", "LAWN");

Console.WriteLine("Удаление завершено.{0}", System.Environment.NewLine);
```

Как видите, в этом примере нет ничего особенного. Здесь вызывается метод ExecuteCommand с передачей строки команды, а также некоторых параметров. Затем выполняется запрос с использованием LINQ to SQL — просто для проверки, что запись действительно вставлена в базу данных. Результаты запроса выводятся на консоль. Для возврата базы в исходное состояние снова вызывается метод ExecuteCommand, чтобы удалить только что вставленную запись.

Ниже показаны результаты выполнения этого кода:

```
Вставка заказчика...
Вставка завершена.

Оказано влияние на 1 строк. Находится ли заказчик в базе данных?
Да, находится.

Удаление заказчика...
Удаление завершено.
```

ExecuteMethodCall()

Первое, что нужно знать о методе ExecuteMethodCall — то, что он является защищенным (protected). Это значит, что вызывать этот метод в коде приложения нельзя, а для этого понадобится унаследовать свой класс от DataContext.

Метод ExecuteMethodCall используется для вызова хранимых процедур и пользовательских функций, возвращающих скалярные значения. Чтобы вызывать пользовательские функции, возвращающие табличные значения, прочтите раздел настоящей главы, посвященный методу CreateMethodCallQuery.

Прототипы

Метод ExecuteMethodCall имеет один прототип, описанный ниже.

Единственный прототип ExecuteMethodCall

```
protected internal IExecuteResult ExecuteMethodCall(
  object instance,
  System.Reflection.MethodInfo methodInfo,
  params object[] parameters)
```

Методу ExecuteMethodCall передается ссылка на объект DataContext или [Your]DataContext, членом которого является метод, вызвавший ExecuteMethodCall, объект MethodInfo для вызова метода и массив params параметров хранимой процедуры или пользовательской функции, возвращающей скалярное значение.

Обратите внимание, что поскольку необходимо передавать объект MethodInfo, метод должен быть оснащен соответствующим атрибутом хранимой процедуры или пользовательской функции, с соответствующими свойствами атрибута. LINQ to SQL затем использует объект MethodInfo для доступа к атрибуту метода Function, чтобы получить имя хранимой процедуры или пользовательской функции, возвращающей скалярное значение. Также он использует объект MethodInfo для получения имен и типов параметров.

Метод ExecuteMethodCall возвращает объект, реализующий интерфейс IExecuteResult, который был описан в главе 15.

Если для генерации сущностных классов применяется инструмент SQLMetal, он создаст методы сущностных классов, которые вызовут ExecuteMethodCall для всех хранимых процедур базы данных, если указана опция /sprocs, и для пользовательских функций базы данных — если указана опция /function.

Примеры

Прежде чем ознакомиться с кодом первого примера, рассмотрим метод по имени CustomersCountByRegion, который SQLMetal сгенерировал для вызова хранимой процедуры базы данных Customers Count By Region. Вот как выглядит этот генерированный метод.

Использование метода ExecuteMethodCall для вызова хранимой процедуры

```
[Function(Name="dbo.Customers Count By Region")]
[return: Parameter(DbType="Int")]
```

```
public int CustomersCountByRegion([Parameter(DbType="NVarChar(15)")] string param1)
{
  IExecuteResult result =
    this.ExecuteMethodCall(
      this,
      ((MethodInfo)(MethodInfo.GetCurrentMethod())), param1);
  return ((int)(result.ReturnValue));
}
```

Как видите, методу `CustomersCountByRegion` передается параметр `string`, который затем передается в качестве параметра методу `ExecuteMethodCall`, который, в свою очередь, передает его хранимой процедуре `Customers Count By Region`.

Метод `ExecuteMethodCall` возвращает переменную, реализующую `IExecuteResult`. Чтобы получить целочисленное возвращаемое значение, метод `CustomersCountByRegion` просто ссылается на свойство возвращенного объекта `ReturnValue` и выполняет приведение его к `int`. Теперь давайте заглянем в листинг 16.23, в котором приведен некоторый код, вызывающий сгенерированный метод `CustomersCountByRegion`.

Листинг 16.23. Пример вызова сгенерированного метода `CustomersCountByRegion`

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
int rc = db.CustomersCountByRegion("WA");
Console.WriteLine("В WA есть {0} заказчиков.", rc);
```

Это очень тривиальный пример, без каких-либо сюрпризов. Вот результат:

```
В WA есть 3 заказчиков.
```

Теперь посмотрим, как вызывать хранимую процедуру, которая возвращает выходной параметр. Опять-таки, имея сгенерированные SQLMetal сущностные классы для базы данных Northwind, проанализируем метод `CustOrderTotal`, сгенерированный для вызова хранимой процедуры `CustOrderTotal`.

Пример использования метода `ExecuteMethodCall` для вызова хранимой процедуры, которая возвращает выходной параметр

```
[Function(Name="dbo.CustOrderTotal")]
[return: Parameter(DbType="Int")]
public int CustOrderTotal(
  [Parameter(Name="CustomerID", DbType="NChar(5)")] string customerID,
  [Parameter(Name="TotalSales", DbType="Money")] ref System.Nullable<decimal>
    totalSales)
{
  IExecuteResult result =
    this.ExecuteMethodCall(
      this,
      ((MethodInfo)(MethodInfo.GetCurrentMethod())),
      customerID,
      totalSales);
  totalSales = ((System.Nullable<decimal>)(result.GetParameterValue(1)));
  return ((int)(result.ReturnValue));
}
```

Обратите внимание, что во втором параметре метода `CustOrderTotal` — `totalSales` — указано ключевое слово `ref`. Это намек на то, что хранимая процедура собирается возвращать это значение. Для того чтобы получить значение после вызова метода `ExecuteMethodCall`, код вызывает метод `GetParameterValue` на возвращенном объекте, реализующем `IExecuteResult`, и передает ему `1`, поскольку интересует второй параметр. Код в листинге 16.24 вызывает метод `CustOrderTotal`.

Листинг 16.24. Пример вызова сгенерированного метода `CustOrderTotal`

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
decimal? totalSales = 0;

int rc = db.CustOrderTotal("LAZYK", ref totalSales);

Console.WriteLine("Заказчик LAZYK имеет общий объем продаж на сумму {0:C}.", totalSales);
```

Для второго параметра `totalSales` было указано ключевое слово `ref`. Вот результат:

```
Заказчик LAZYK имеет общий объем продаж на сумму $357.00.
```

Теперь рассмотрим пример, в котором вызывается хранимая процедура с помощью метода `ExecuteMethodCall`.

Пример использования метода `ExecuteMethodCall` для вызова хранимой процедуры, возвращающей одну форму

```
[Function(Name="dbo.Customers By City")]
public ISingleResult<CustomersByCityResult>
  CustomersByCity([Parameter(DbType="NVarChar(20)")] string param1)
{
  IExecuteResult result =
    this.ExecuteMethodCall(
      this,
      ((MethodInfo)(MethodInfo.GetCurrentMethod())),
      param1);
  return ((ISingleResult<CustomersByCityResult>)(result.ReturnValue));
}
```

Обратите внимание, что сгенерированный метод возвращает объект типа `ISingleResult<CustomersByCityResult>`. Сгенерированный метод получает этот объект, выполняя приведение свойства `ReturnValue` возвращенного объекта к этому типу. Инструмент SQLMetal также сгенерировал класс `CustomersByCityResult`, который здесь не рассматривается. В листинге 16.25 приведен код вызова сгенерированного метода `CustomersByCity`.

Листинг 16.25. Пример вызова сгенерированного метода `CustomersByCity`

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");

ISingleResult<CustomersByCityResult> results = db.CustomersByCity("London");

foreach (CustomersByCityResult cust in results)
{
  Console.WriteLine("{0} - {1} - {2} - {3}", cust.CustomerID, cust.CompanyName,
    cust.ContactName, cust.City);
}
```

Как видите, осуществляется перечисление возвращенного объекта типа `ISingleResult<CustomersByCityResult>`, как если бы это была последовательность LINQ. Это объясняется тем, что он унаследован от `IEnumerable<T>`, о чем упоминалось в главе 15. Результаты затем выводятся на консоль и выглядят следующим образом:

```
AROUT - Around the Horn - Thomas Hardy - London
BSBEV - B's Beverages - Victoria Ashworth - London
CONSH - Consolidated Holdings - Elizabeth Brown - London
EASTC - Eastern Connection - Ann Devon - London
NORTS - North/South - Simon Crowther - London
SEVES - Seven Seas Imports - Hari Kumar — London
```

Теперь давайте взглянем на некоторые примеры, возвращающие результат в нескольких формах. Если вы не знакомы с термином *форма* (shape) в данном контексте, вот вам пояснение: форма результата определяется типом возвращенных данных. Когда запрос возвращает идентификатор и имя заказчика — это форма. Если запрос возвращает идентификатор заказа, дату заказа и код поставки — это еще одна форма. Если запрос возвращает и то, и другое, запись, содержащую идентификатор заказчика и имя, и что-то еще, а также запись, содержащую идентификатор заказа, дату заказа и код поставки, то такой запрос возвращает несколько результирующих форм. Поскольку хранимые процедуры обладают такой способностью, LINQ to SQL необходим какой-то способ, чтобы справиться с этим, и такой способ есть.

Для первого примера, возвращающего несколько форм, рассмотрим сценарий, где форма результата определяется некоторым условием. К счастью, расширенная база данных Northwind имеет хранимую процедуру такого типа. Ее имя — `Whole Or Partial Customers Set`. Инструмент SQLMetal сгенерировал метод для вызова этой хранимой процедуры по имени `WholeOrPartialCustomersSet`, который показан ниже.

Пример, использующий вызов метода `ExecuteMethodCall` для вызова хранимой процедуры, которая возвращает результаты различной формы в зависимости от условия

```
[Function(Name="dbo.Whole Or Partial Customers Set")]
[ResultType(typeof(WholeOrPartialCustomersSetResult1))]
  [ResultType(typeof(WholeOrPartialCustomersSetResult2))]
public IMultipleResults WholeOrPartialCustomersSet(
[Parameter(DbType="Int")] System.Nullable<int> param1)
{
  IExecuteResult result =
    this.ExecuteMethodCall(
      this,
      ((MethodInfo)(MethodInfo.GetCurrentMethod())),
      param1);
  return ((IMultipleResults)(result.ReturnValue));
}
```

Обратите внимание, что есть два атрибута `ResultType`, указывающих две возможные формы результата. Инструмент SQLMetal сгенерировал два указанных класса. Разработчик, вызывающий метод `WholeOrPartialCustomersSet`, должен иметь в виду, что хранимая процедура возвращает разные результаты в зависимости от значения `param1`. Поскольку речь идет о хранимой процедуре, известно, что если `param1` равен `1`, то хранимая процедура вернет все поля из таблицы `Customer`, и потому метод вернет последовательность объектов типа `WholeOrPartialCustomersSetResult1`. Если же значение `param1` равно `2`, то будет возвращен сокращенный набор полей в последовательности объектов типа `WholeOrPartialCustomersSetResult2`.

Также заметьте, что типом возврата метода `WholeOrPartialCustomersSet` является `IMultipleResults`. Метод получает его, приводя свойство `ReturnValue` объекта, возвращенного методом `ExecuteMethodCall` к `IMultipleResults`. Этот интерфейс рассматривался в главе 15.

В листинге 16.26 представлен пример вызова метода `WholeOrPartialCustomersSet`.

Листинг 16.26. Пример вызова сгенерированного метода `WholeOrPartialCustomersSet`

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");

IMultipleResults results = db.WholeOrPartialCustomersSet(1);
foreach (WholeOrPartialCustomersSetResult1 cust in
```

```
   results.GetResult<WholeOrPartialCustomersSetResult1>())
{
  Console.WriteLine("{0} - {1} - {2} - {3}", cust.CustomerID, cust.CompanyName,
    cust.ContactName, cust.City);
}
```

Обратите внимание, что результат имеет тип `IMultipleResults`. Так как передано значение `1`, то известно, что будет получена последовательность типа `WholeOrPartialCustomersSetResult1`. Кроме того, для получения результатов вызывается метод `GetResult<T>` на переменной `IMultipleResults`, где `T` — тип возвращенных данных. Вот результаты:

```
LAZYK - Lazy K Kountry Store - John Steel - Walla Walla
TRAIH - Trail's Head Gourmet Provisioners - Helvetius Nagy - Kirkland
WHITC - White Clover Markets - Karl Jablonski — Seattle
```

Эта хранимая процедура извлекает только заказчиков из региона `"WA"`. Если бы при вызове метода `WholeOrPartialCustomersSet` передавалось значение `2`, была бы получена последовательность типа `WholeOrPartialCustomersSetResult2`, так что везде в предыдущем коде, где указан тип `WholeOrPartialCustomersSetResult1`, нужно было бы заменить его на `WholeOrPartialCustomersSetResult2`.

Теперь осталось рассмотреть только случай, когда хранимая процедура возвращает несколько форм за один вызов. В этом поможет хранимая процедура из расширенной базы данных Northwind под названием `Get Customer And Orders`. Для начала давайте посмотрим на метод, сгенерированный SQLMetal для вызова этой хранимой процедуры.

Пример использования метода `ExecuteMethodCall` для вызова хранимой процедуры, возвращающей множественные формы

```
[Function(Name="dbo.Get Customer And Orders")]
[ResultType(typeof(GetCustomerAndOrdersResult1))]
[ResultType(typeof(GetCustomerAndOrdersResult2))]
public IMultipleResults GetCustomerAndOrders(
  [Parameter(Name="CustomerID", DbType="NChar(5)")] string customerID)
{
  IExecuteResult result =
    this.ExecuteMethodCall(
      this,
      ((MethodInfo)(MethodInfo.GetCurrentMethod())),
      customerID);
  return ((IMultipleResults)(result.ReturnValue));
}
```

Как видите, типом возврата метода является `IMultipleResults`. Поскольку хранимая процедура возвращает результат в нескольких формах, на вас возлагается ответственность в определении порядка возвращаемых форм. Поскольку хранимая процедура `Get Customer And Orders` была изучена, известно, что она сначала вернет запись из таблицы `Customers`, а за ней — связанные записи из таблицы `Orders`.

Код в листинг 16.27 вызывает сгенерированный метод из предыдущего кода.

Листинг 16.27. Пример вызова сгенерированного метода `GetCustomerAndOrders`

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
IMultipleResults results = db.GetCustomerAndOrders("LAZYK");
GetCustomerAndOrdersResult1 cust =
  results.GetResult<GetCustomerAndOrdersResult1>().Single();
Console.WriteLine("Заказы {0}:", cust.CompanyName);
```

```
foreach (GetCustomerAndOrdersResult2 order in
   results.GetResult<GetCustomerAndOrdersResult2>())
{
  Console.WriteLine("{0} - {1}", order.OrderID, order.OrderDate);
}
```

Поскольку известно, что хранимая процедура вернет единственную запись, соответствующую типу GetCustomerAndOrdersResult1, то можно вызвать операцию Single на последовательности, содержащей этот тип — до тех пор, пока есть уверенность, что нужный заказчик с переданным CustomerID существует. Если же такой уверенности нет, то всегда можно вызвать операцию SingleOrDefault. Также известно, что после возврата единственного объекта GetCustomerAndOrdersResult1 должно быть возвращено ноль или более объектов GetCustomerAndOrdersResult2, поэтому выполняется их перечисление с отображениме интересующих данных. Ниже показаны результаты:

```
Заказы Lazy K Kountry Store:
10482 - 3/21/1997 12:00:00 AM
10545 - 5/22/1997 12:00:00 AM
```

На этом примеры вызовов хранимых с помощью метода ExecuteMethodCall завершены. В начале раздела, посвященного этому методу, было упомянуто, что этот метод используется и для вызова пользовательских функций, возвращающих скалярные значения. Давайте рассмотрим пример вызова пользовательской функции, возвращающей скалярное значение.

Сначала посмотрим на сгенерированный SQLMetal метод под названием ExecuteMethodCall для вызова пользовательской функции скалярного значения.

Пример использования метода ExecuteMethodCall для вызова определяемой пользователем функции, возвращающей скалярное значение

```
[Function(Name="dbo.MinUnitPriceByCategory", IsComposable=true)]
[return: Parameter(DbType="Money")]
public System.Nullable<decimal> MinUnitPriceByCategory(
  [Parameter(DbType="Int")] System.Nullable<int> categoryID)
{
  return ((System.Nullable<decimal>)(this.ExecuteMethodCall(this,
    ((MethodInfo)(MethodInfo.GetCurrentMethod())), categoryID).ReturnValue));
}
```

Обратите внимание, что скалярное значение, возвращенное хранимой процедурой, получается через ссылку на свойство ReturnValue возвращенного методом ExecuteMethidCall объекта.

Можно было бы создать простой пример вызова сгенерированного метода MinUnitPriceByCategory. Однако вся прелесть определяемой пользователем функции проявляется при включении ее в запрос, подобно встроенным функциям SQL.

Взглянем на пример, представленный в листинге 16.28, который встраивает метод MinUnitPriceByCategory в запрос для идентификации всех продуктов с минимальной ценой в своей категории.

Листинг 16.28. Пример встраивания определенной пользователем функции в запрос

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
IQueryable<Product> products = from p in db.Products
                          where p.UnitPrice ==
                            db.MinUnitPriceByCategory(p.CategoryID)
                          select p;
foreach (Product p in products)
```

```
{
  Console.WriteLine("{0} - {1:C}", p.ProductName, p.UnitPrice);
}
```

В этом примере в конструкцию where встраивается вызов метода MinUnitPriceByCategory, который, в свою очередь, вызывает одноименную скалярную пользовательскую функцию. Вот результат:

```
Aniseed Syrup - $10.00
Konbu - $6.00
Teatime Chocolate Biscuits - $9.20
Guaraná Fantástica - $4.50
Geitost - $2.50
Filo Mix - $7.00
Tourtière - $7.45
Longlife Tofu - $10.00
```

GetCommand()

Еще один потенциально полезный метод — это GetCommand. Когда вызывается метод GetCommand на объекте DataContext и ему передается интерфейс LINQ to SQL по имени IQueryable, то возвращается объект типа System.Data.Common.DbCommand. Возвращенный объект DbCommand содержит доступ к нескольким ключевым компонентам, которые будут использованы запросом.

Извлекая объект DbCommand с помощью метода GetCommand, можно получить ссылку на объекты CommandText, CommandTimeout, Connection, Parameters и Transaction, наряду с некоторыми другими, относящимися к переданному запросу. Это позволяет не только просматривать эти объекты, но и модифицировать их, изменяя значения по умолчанию, *без* модификации одних и тех же запросов, которые будут выполнены текущим экземпляром DataContext. Возможно, для конкретного запроса понадобится увеличить значение CommandTimeout, но так, чтобы не менять значение таймаута для всех остальных запросов, выполняемых с данным объектом DataConext.

Прототипы

Метод GetCommand имеет один прототип, описанный ниже.

Единственный прототип GetCommand

```
System.Data.Common.DbCommand GetCommand(IQueryable query)
```

Этому методу передается запрос LINQ to SQL в форме IQueryable, а возвращает он System.Data.Common.DbCommand этого запроса.

Примеры

В листинге 16.29 получается объект DbCommand для изменения значения CommandTimeout запроса. На консоль выводится значение CommandText, которым будет сам запрос SQL.

Листинг 16.29. Пример вызова метода GetCommand

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");

IQueryable<Customer> custs = from c in db.Customers
                        where c.Region == "WA"
                        select c;

System.Data.Common.DbCommand dbc = db.GetCommand(custs);
Console.WriteLine("Таймаут запроса: {0}{1}", dbc.CommandTimeout,
```

```
  System.Environment.NewLine);
dbc.CommandTimeout = 1;
Console.WriteLine("SQL-код запроса: {0}{1}",
  dbc.CommandText, System.Environment.NewLine);
Console.WriteLine("Таймаут запроса: {0}{1}", dbc.CommandTimeout,
  System.Environment.NewLine);
foreach (Customer c in custs)
{
  Console.WriteLine("{0}", c.CompanyName);
}
```

Об этом примере говорить особо нечего. Здесь просто объявляется запрос, который передается методу `GetCommand`. Затем отображается значение `CommandTimeout` для возвращенного объекта `DbCommand`. Далее значение `CommandTimeout` устанавливается в `1` и выводится сам запрос SQL вместе с новым значением `CommandTimeout`. И последнее — выполняется перечисление результатов, возвращенных запросом.

Ниже показаны результаты запуска этого кода:

```
Таймаут запроса: 30

SQL-код запроса: SELECT [t0].[CustomerID], [t0].[CompanyName], [t0].[ContactName],
[t0].[ContactTitle], [t0].[Address], [t0].[City], [t0].[Region], [t0].[PostalCode],
[t0].[Country], [t0].[Phone], [t0].[Fax]
FROM [dbo].[Customers] AS [t0]
WHERE [t0].[Region] = @p0

Таймаут запроса: 1

Lazy K Kountry Store
Trail's Head Gourmet Provisioners
White Clover Markets
```

Разумеется, если этот запрос потребует слишком много времени для выполнения, он будет прерван по таймауту, и результаты окажутся другими.

GetChangeSet()

Иногда удобно иметь возможность получить список всех сущностных объектов, которые *будут* вставлены, изменены или удалены при вызове метода `SubmitChanges`. Метод `GetChangeSet` служит именно для этого.

Прототипы

Метод `GetChangeSet` имеет один прототип, описанный ниже.

Единственный прототип GetChangeSet

```
ChangeSet GetChangeSet()
```

Этот метод не принимает ничего и возвращает объект `ChangeSet`. Объект `ChangeSet` содержит коллекции типа `IList<T>` для вставленных, модифицированных и удаленных объектов, где `T` — сущностный класс. Эти свойства-коллекции называются `Inserts`, `Updates` и `Deletes` соответственно.

Затем можно выполнить перечисление каждой из этих коллекций для того, чтобы просмотреть содержащиеся в них объекты.

Примеры

В листинге 16.30 сущностный объект модифицируется, вставляется и удаляется. Затем с помощью метода `GetChangeSet` извлекается `ChangeSet` и производится перечисление каждой коллекции.

Листинг 16.30. Пример вызова метода `GetChangeSet`

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");

Customer cust = (from c in db.Customers
                 where c.CustomerID == "LAZYK"
                 select c).Single<Customer>();

cust.Region = "Washington";
db.Customers.InsertOnSubmit(
  new Customer
  {
    CustomerID = "LAWN",
    CompanyName = "Lawn Wranglers",
    ContactName = "Mr. Abe Henry",
    ContactTitle = "Owner",
    Address = "1017 Maple Leaf Way",
    City = "Ft. Worth",
    Region = "TX",
    PostalCode = "76104",
    Country = "USA",
    Phone = "(800) MOW-LAWN",
    Fax = "(800) MOW-LAWO"
  });

Customer cust2 = (from c in db.Customers
               where c.CustomerID == "LONEP"
               select c).Single<Customer>();

db.Customers.DeleteOnSubmit(cust2);
cust2 = null;
ChangeSet changeSet = db.GetChangeSet();
Console.WriteLine("{0}Добавленные сущности:", System.Environment.NewLine);

foreach (Customer c in changeSet.Inserts)
{
  Console.WriteLine("Заказчик {0} будет добавлен.", c.CompanyName);
}
Console.WriteLine("{0}Модифицированные сущности:", System.Environment.NewLine);

foreach (Customer c in changeSet.Updates)
{
  Console.WriteLine("Заказчик {0} будет обновлен.", c.CompanyName);
}
Console.WriteLine("{0}Удаленные сущности:", System.Environment.NewLine);

foreach (Customer c in changeSet.Deletes)
{
  Console.WriteLine("Заказчик {0} будет удален.", c.CompanyName);
}
```

В предыдущем примере сначала модифицируется `Region` заказчика `LAZYK`. Затем заказчик `LAWN` вставляется, а заказчик `LONEP` удаляется. После этого с помощью метода `GetChangeSet` получается `ChangeSet`. И, наконец, осуществляется перечисление каждой коллекции — `Inserts`, `Updates` и `Deletes` — с отображением каждого сущностного объекта в соответствующей коллекции.

Ниже показаны результаты:

```
Добавленные сущности:
Заказчик Lawn Wranglers будет добавлен.

Модифицированные сущности:
Заказчик Lazy K Kountry Store будет обновлен.

Удаленные сущности:
Заказчик Lonesome Pine Restaurant будет удален.
```

Конечно, в предыдущем примере можно выполнять перечисление каждой коллекции, предполагая, что каждый элемент является объектом `Customer`, потому что известно, что так оно и есть. Однако во многих случаях в коллекции может существовать более одного типа объектов, и делать такие предположения не получится. В таких ситуациях понадобится написать собственное перечисление для обработки множественных типов данных. Помочь в этом может операция `OfType`.

GetTable()

Метод `GetTable` используется для получения ссылки на последовательность `Table` из `DataContext` для определенной таблицы базы данных. Этот метод обычно применяется только вместе с классом `DataContext`, а не `[Your]DataContext`. Использование класса `[Your]DataContext` является предпочтительной техникой, поскольку этот класс уже будет иметь свойство-последовательность `Table`, ссылающееся на каждую отображенную таблицу.

Прототипы

Метод `GetTable` имеет два прототипа, которые описаны ниже.

Первый прототип GetTable

```
Table<T> GetTable<T>()
```

Этому методу предоставляется указанный отображаемый сущностный тип `T`, и он возвращает последовательность `Table` объектов типа `T`.

Второй прототип GetTable

```
ITable GetTable(Type type)
```

Этот метод принимает `Type` сущностного объекта и возвращает интерфейс таблицы. Затем этот интерфейс `ITable` можно использовать по своему усмотрению. Чтобы применять интерфейс `ITable`, как если бы это была таблица, не забудьте привести его к `IQueryable<T>`.

Примеры

В примере первого прототипа в листинге 16.31 для извлечения определенного заказчика используется стандартный класс `DataContext` вместо класса `[Your]DataContext` по имени `Northwind`.

Листинг 16.31. Пример вызова первого прототипа GetTable

```
DataContext db =
  new DataContext(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
Customer cust = (from c in db.GetTable<Customer>()
                 where c.CustomerID == "LAZYK"
                 select c).Single<Customer>();
Console.WriteLine("Заказчик {0} извлечен.", cust.CompanyName);
```

Здесь вызывается метод GetTable для получения ссылки на таблицу Customer, чтобы иметь возможность извлечь конкретного заказчика. Вот результат:

```
Заказчик Lazy K Kountry Store извлечен.
```

Для примера второго прототипа метода GetTable используется DataContext вместо [Your]DataContext. В листинге 16.32 представлен тот же базовый пример, что и предыдущий, но в нем применяется второй прототип.

Листинг 16.32. Пример вызова второго прототипа GetTable

```
DataContext db =
  new DataContext(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
Customer cust = (from c in ((IQueryable<Customer>)db.GetTable(typeof(Customer)))
                 where c.CustomerID == "LAZYK"
                 select c).Single<Customer>();
Console.WriteLine("Заказчик {0} извлечен.", cust.CompanyName);
```

Не должно быть сюрпризом, что результат работы кода из листинга 16.32 совпадает с результатом выполнения кода из листинга 16.31:

```
Заказчик Lazy K Kountry Store извлечен.
```

Refresh()

Метод Refresh позволяет вручную обновлять сущностные объекты из базы данных. В некоторых ситуациях это делается при вызове метода ResolveAll коллекции ChangeConflicts объекта DataContext в случае возникновения конфликтов во время вызова метода SubmitChanges. Однако могут быть ситуации, в которых метод SubmitChanges никогда не вызывается, но нужно получать обновления из базы данных.

Примером может служить приложение, которое отображает данные о состоянии некоторого сущностного объекта, системы или процесса, доступные только для чтения. Могут понадобиться регулярные обновления данных из базы через какие-то интервалы времени. Для этой цели пригодится метод Refresh. С помощью метода Refresh можно обновлять отдельный сущностный объект либо последовательность сущностных объектов — имеется в виду результат запроса LINQ to SQL.

Прототипы

Метод Refresh имеет три прототипа, которые описаны ниже.

Первый прототип Refresh

```
void Refresh(RefreshMode mode, object entity)
```

Этот метод принимает режим обновления и единственный сущностный объект и ничего не возвращает.

Второй прототип Refresh

```
void Refresh(RefreshMode mode, params object[] entities)
```

Этот метод принимает режим обновления и массив params сущностных объектов и ничего не возвращает.

Третий прототип Refresh

```
void Refresh(RefreshMode mode, System.Collections.IEnumerable entities)
```

Этот метод принимает режим обновления и последовательность сущностных объектов и ничего не возвращает.

Перечисление RefreshMode имеет три возможных значения: KeepChanges, KeepCurrentValues и OverwriteCurrentValues.

В документации Visual Studio о перечислении `RefreshMode` эти значения определены так, как описано в табл. 16.1.

Таблица 16.1. Перечисление `RefreshMode`

Имя члена	Описание
`KeepCurrentValues`	Заставляет метод `Refresh` обменять исходное значение объекта со значениями, извлеченными из базы данных
`KeepChanges`	Заставляет метод `Refresh` сохранять текущее значение, которое было изменено, но обновляет другие значения прочитанными из базы
`OverwriteCurrentValues`	Заставляет метод `Refresh` заменить все текущие значения значениями, взятыми из базы

Поведение каждого из этих значений более подробно описано в главе 17.

Примеры

Для примера первого прототипа в листинге 16.33 запрашивается заказчик с использованием LINQ to SQL и на консоль выводится имя и должность контактного лица. Затем с помощью ADO.NET имя контактного лица этого заказчика изменяется в базе данных. Должность контактного лица модифицируется в сущностном объекте. И чтобы доказать, что текущий сущностный объект не имеет понятия об изменении в базе данных, но имеет измененную должность контакта, снова отображается имя и должность контактного лица, и вы увидите, что имя контактного лица не изменилось, тогда как должность изменилась.

После этого вызывается метод `Refresh` с `RefreshMode`, равным `KeepChanges`, и контактное имя и должность из сущностного объекта выводится на консоль еще раз. Можно убедиться, что сущностный объект действительно содержит новое значение имени контактного лица из базы данных, в то же время сохраняя изменение, внесенное в должность контакта.

В конце контактное имя возвращается в исходное значение, чтобы пример можно было запускать многократно. В листинге 16.33 показан необходимый код.

Листинг 16.33. Пример вызова первого прототипа метода `Refresh`

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
Customer cust = (from c in db.Customers
                 where c.CustomerID == "GREAL"
                 select c).Single<Customer>();
Console.WriteLine("Исходное имя заказчика - {0}, ContactTitle - {1}.{2}",
  cust.ContactName, cust.ContactTitle, System.Environment.NewLine);
ExecuteStatementInDb(String.Format(
  @"update Customers set ContactName = 'Brad Radaker' where CustomerID = 'GREAL'"));
cust.ContactTitle = "Chief Technology Officer";
Console.WriteLine("Имя заказчика перед обновлением - {0}, ContactTitle - {1}.{2}",
  cust.ContactName, cust.ContactTitle, System.Environment.NewLine);
db.Refresh(RefreshMode.KeepChanges, cust);
Console.WriteLine("Имя заказчика после обновления - {0}, ContactTitle - {1}.{2}",
  cust.ContactName, cust.ContactTitle, System.Environment.NewLine);
// Необходимо восстановить старые значения, чтобы код
// можно было запускать более одного раза.
Console.WriteLine("{0}Сброс данных в исходные значения.",
  System.Environment.NewLine);
ExecuteStatementInDb(String.Format(
  @"update Customers set ContactName = 'John Steel' where CustomerID = 'GREAL'"));
```

В предыдущем коде выполняется запрос LINQ to SQL для получения ссылки на объект Customer. Затем на консоль выводятся значения ContactName и ContactTitle объекта Customer.

После этого с помощью ADO.NET значение ContactName заказчика обновляется в базе данных и модифицируется ContactTitle в извлеченном сущностном объекте Customer. В этой точке сущностный объект Customer не знает, что ContactName было изменено в базе данных, и это доказывается выводом на консоль значений полей ContactName и ContactTitle объекта Customer.

Далее вызывается метод RefreshMethod с KeepChanges из RefreshMode. Это должно заставить загрузиться в сущностный объект любые свойства объекта Customer, которые были изменены в базе данных. В рассматриваемом случае, поскольку значение ContactName было изменено в базе данных, оно должно быть обновлено из этой базы данных.

Затем на консоль выводятся значения ContactName и ContactTitle объекта Customer, при этом для ContactName должно быть отображено значение, взятое из базы данных, а для ContactTitle — значение, измененное в сущностном объекте.

И, наконец, исходные значения в базе данных восстанавливаются, чтобы этот пример можно было запустить снова, и чтобы последующие примеры сохранили работоспособность.

Ниже показаны результаты работы кода из листинга 16.33:

```
Исходное имя заказчика - John Steel, ContactTitle - Marketing Manager.

Выполнение оператора SQL для базы данных с помощью ADO.NET...
База данных обновлена.
Имя заказчика перед обновлением - John Steel, ContactTitle - Chief Technology
Officer.

Имя заказчика после обновления - Brad Radaker, ContactTitle - Chief Technology
Officer.

Сброс данных в исходные значения.
Выполнение оператора SQL для базы данных с помощью ADO.NET...
База данных обновлена.
```

Как видите, сущностный объект не знает об изменении ContactName на "Brad Radaker" в базе данных, пока не будет вызван метод Refresh.

Для примера вызова второго прототипа в листинге 16.34 производится извлечение заказчиков из региона "WA" с использованием LINQ to SQL. После этого осуществляется перечисление членов возвращенной последовательности объектов Customer с выводом на консоль их свойств CustomerID, Region и Country. Затем с помощью ADO.NET обновляется поле Country у каждого заказчика в базе данных, чьим регионом является "WA". В этой точке значение поля Country для этих заказчиков в базе данных будет отличаться от значения, которое есть в соответствующем свойстве извлеченных ранее сущностных объектов. Снова выполняется перечисление последовательности извлеченных заказчиков — просто чтобы доказать, что сущностные объекты не имеют понятия об изменении поля Region в базе данных.

Далее на последовательности объектов Customer вызывается операция ToArray, чтобы получить массив, содержащий объекты Customer.

Наконец, в последний раз выполняется перечисление последовательности сущностных объектов Customer с выводом на консоль их свойств CustomerID, Region и Country, чтобы доказать, что поле Country на самом деле было обновлено из базы данных.

Разумеется, придется также восстановить базу данных в исходное состояние, поэтому используется ADO.NET, чтобы установить Country заказчика обратно в исходное значение.

Код приведен в листинге 16.34.

Листинг 16.34. Пример вызова второго прототипа метода `Refresh`

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
IEnumerable<Customer> custs = (from c in db.Customers
                               where c.Region == "WA"
                               select c);
Console.WriteLine("Сущностные объекты перед изменением с помощью ADO.NET и вызовом
Refresh():");
foreach (Customer c in custs)
{
  Console.WriteLine("Заказчик {0} имеет регион {1} и страну {2}.",
    c.CustomerID, c.Region, c.Country);
}
Console.WriteLine("{0}Обновление страны заказчиков на United States в ADO.NET...",
  System.Environment.NewLine);
ExecuteStatementInDb(String.Format(
  @"update Customers set Country = 'United States' where Region = 'WA'"));
Console.WriteLine("Страна заказчиков обновлена.{0}", System.Environment.NewLine);
Console.WriteLine("Сущностные объекты после изменения с помощью ADO.NET, но перед
вызовом Refresh():");
foreach (Customer c in custs)
{
  Console.WriteLine("Заказчик {0} имеет регион {1} и страну {2}.",
    c.CustomerID, c.Region, c.Country);
}
Customer[] custArray = custs.ToArray();
Console.WriteLine("{0}Обновление массива params сущностных объектов заказчиков...",
  System.Environment.NewLine);
db.Refresh(RefreshMode.KeepChanges, custArray[0], custArray[1], custArray[2]);
Console.WriteLine("Массив params сущностных объектов заказчиков обновлен.{0}",
  System.Environment.NewLine);
Console.WriteLine("Сущностные объекты после изменения с помощью ADO.NET и вызова Refresh():");
foreach (Customer c in custs)
{
  Console.WriteLine("Заказчик {0} имеет регион {1} и страну {2}.",
    c.CustomerID, c.Region, c.Country);
}
// Нужно вернуть измененные значения в исходное состояние,
// чтобы можно было запустить пример более одного раза.
Console.WriteLine("{0}Сброс данных в исходные значения.",
  System.Environment.NewLine);
ExecuteStatementInDb(String.Format(
  @"update Customers set Country = 'USA' where Region = 'WA'"));
```

В предыдущем коде не происходит ничего интересного вплоть до момента вызова операции `ToArray`. Как только получается массив объектов `Customer`, сразу же вызывается `RefreshMethod` с передачей ему `custArray[0]`, `custArray[1]` и `custArray[2]`.

Взглянем на результаты:

```
Сущностные объекты перед изменением с помощью ADO.NET и вызовом Refresh():
Заказчик LAZYK имеет регион WA и страну USA.
Заказчик TRAIH имеет регион WA и страну USA.
Заказчик WHITC имеет регион WA и страну USA.
```

```
Обновление страны заказчиков на United States в ADO.NET...
Выполнение оператора SQL для базы данных с помощью ADO.NET...
База данных обновлена.
Страна заказчиков обновлена.

Сущностные объекты после изменения с помощью ADO.NET, но перед вызовом Refresh():
Заказчик LAZYK имеет регион WA и страну USA.
Заказчик TRAIH имеет регион WA и страну USA.
Заказчик WHITC имеет регион WA и страну USA.

Обновление массива params сущностных объектов заказчиков...
Массив params сущностных объектов заказчиков обновлен.

Сущностные объекты после изменения с помощью ADO.NET и вызова Refresh():
Заказчик LAZYK имеет регион WA и страну United States.
Заказчик TRAIH имеет регион WA и страну United States.
Заказчик WHITC имеет регион WA и страну United States.

Сброс данных в исходные значения.
Выполнение оператора SQL для базы данных с помощью ADO.NET...
База данных обновлена.
```

В этом результате видно, что изменения, внесенные в поле `Country` в базе данных, не отражаются в сущностном объекте `Customer` до тех пор, пока не будет вызван метод `Refresh`.

В листинге 16.34 каждый обновленный сущностный объект был одного и того же типа — `Customer`. Во втором прототипе метода `Refresh` не обязательно, чтобы все переданные сущностные объекты были одного и того же типа. Ему можно передавать сущностные объекты разных типов. В случае листинга 16.34 на самом деле было бы проще передать последовательность сущностных объектов методу `Refresh`, потому что такая последовательность была. К счастью, третий прототип метода `Refresh` позволяет передавать последовательность.

Итак, для примера применения третьего прототипа в листинге 16.35 используется тот же базовый код, что и в листинге 16.34, но на этот раз передается последовательность извлеченных сущностных объектов `Customer`.

Листинг 16.35. Пример вызова третьего прототипа метода `Refresh`

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");

IEnumerable<Customer> custs = (from c in db.Customers
                               where c.Region == "WA"
                               select c);
Console.WriteLine("Сущностные объекты перед изменением с помощью ADO.NET и вызовом
Refresh():");

foreach (Customer c in custs)
{
  Console.WriteLine("Заказчик {0} имеет регион {1} и страну {2}.",
    c.CustomerID, c.Region, c.Country);
}

Console.WriteLine("{0} Обновление страны заказчиков на United States в ADO.NET...",
  System.Environment.NewLine);

ExecuteStatementInDb(String.Format(
  @"update Customers set Country = 'United States' where Region = 'WA'"));

Console.WriteLine("Страна заказчиков обновлена.{0}", System.Environment.NewLine);

Console.WriteLine("Сущностные объекты после изменения с помощью ADO.NET, но перед
вызовом Refresh():");
```

```
foreach (Customer c in custs)
{
  Console.WriteLine("Заказчик {0} имеет регион {1} и страну {2}.",
    c.CustomerID, c.Region, c.Country);
}
Console.WriteLine("{0}Обновление последовательности сущностных объектов заказчиков...",
  System.Environment.NewLine);

db.Refresh(RefreshMode.KeepChanges, custs);

Console.WriteLine("Последовательность сущностных объектов заказчиков обновлена.{0}",
  System.Environment.NewLine);
Console.WriteLine("Сущностные объекты после изменения с помощью ADO.NET и вызова
Refresh():");

foreach (Customer c in custs)
{
  Console.WriteLine("Заказчик {0} имеет регион {1} и страну {2}.",
    c.CustomerID, c.Region, c.Country);
}

// Нужно вернуть измененные значения в исходное состояние,
// чтобы можно было запустить пример более одного раза.
Console.WriteLine("{0} Сброс данных в исходные значения.",
  System.Environment.NewLine);
ExecuteStatementInDb(String.Format(
  @"update Customers set Country = 'USA' where Region = 'WA'"));
```

Код в листинге 16.35 тот же, что и в листинге 16.34, за исключением того, что при вызове метода `Refresh` я передаю ему последовательность `custs`. Взглянем на результат:

```
Сущностные объекты перед изменением с помощью ADO.NET и вызовом Refresh():
Заказчик LAZYK имеет регион WA и страну USA.
Заказчик TRAIH имеет регион WA и страну USA.
Заказчик WHITC имеет регион WA и страну USA.

Обновление страны заказчиков на United States в ADO.NET...
Выполнение оператора SQL для базы данных с помощью ADO.NET...
База данных обновлена.
Страна заказчиков обновлена.

Сущностные объекты после изменения с помощью ADO.NET, но перед вызовом Refresh():
Заказчик LAZYK имеет регион WA и страну USA.
Заказчик TRAIH имеет регион WA и страну USA.
Заказчик WHITC имеет регион WA и страну USA.

Обновление последовательности сущностных объектов заказчиков...
Последовательность сущностных объектов заказчиков обновлена.

Сущностные объекты после изменения с помощью ADO.NET и вызова Refresh():
Заказчик LAZYK имеет регион WA и страну United States.
Заказчик TRAIH имеет регион WA и страну United States.
Заказчик WHITC имеет регион WA и страну United States.

Сброс данных в исходные значения.
Выполнение оператора SQL для базы данных с помощью ADO.NET...
База данных обновлена.
```

Как видите, несмотря на тот факт, что значение `Country` у извлеченных заказчиков было изменено на "`United States`" в базе данных, изменения в сущностных объектах не отразились до тех пор, пока не был вызван метод `Refresh`.

Резюме

Обретение понимания всего того, что может делать класс `DataContext`, потребовало немало времени. API-интерфейс LINQ to SQL не тривиален, потому что предполагает знание LINQ, запросов данных и SQL. Из-за этого для овладения LINQ to SQL нужно узнать достаточно много, и большая часть этих знаний касается класса `DataContext` в его переплетении с сущностными классами; поэтому что-то познается в начале, а что-то — в конце.

Хотя в этой главе было представлено много информации, возможно, наиболее важной темой здесь является работа трех служб `DataContext`: отслеживание идентичности, отслеживание изменений и обработка изменений. Конечно, ни одна из этих служб ничего не стоит, если не создан экземпляр объекта `DataContext` или `[Your]DataContext`, так что конструкторы классов `DataContext` и `[Your]DataContext` важны не менее.

Помимо конструкторов `DataContext` и `[Your]DataContext`, скорее всего, наиболее часто будет использоваться метод `SubmitChanges`, потому что этот метод вызывается для сохранения изменений в базе данных.

Важно помнить, что при попытке сохранения данных в базе иногда могут возникнуть конфликты параллельного доступа, сопровождаемые генерацией исключений. До сих пор в главах, посвященных LINQ to SQL, не раз упоминалось о конфликтах параллелизма, но пока они детально не рассматривались. Этому как раз посвящена следующая глава.

ГЛАВА 17

Конфликты параллельного доступа LINQ to SQL

Необходимость обнаружения и разрешения конфликтов параллельного доступа упоминалась неоднократно. В большинстве предыдущих глав, посвященных LINQ to SQL, говорилось о конфликтах параллельного доступа, но пока они не рассматривались настолько подробно, как того заслуживают. В настоящей главе это упущение восполнено.

Предварительные условия для запуска примеров

Для того чтобы запустить примеры этой главы, понадобится расширенная версия базы данных Northwind и сгенерированные для нее сущностные классы. Прочтите и выполните требования, изложенные в разделе "Предварительные условия для запуска примеров" главы 12.

Некоторые общие методы

Дополнительно для запуска примеров этой главы понадобятся общие методы, которые будут использованы в примерах. Прочтите и выполните то, что написано в разделе "Некоторые общие методы" главы 12.

Использование API-интерфейса LINQ to SQL

Чтобы запустить примеры этой главы, в проект может понадобиться добавить соответствующие ссылки и директивы `using`. Прочтите и выполните то, что сказано в разделе "Использование LINQ to SQL" главы 12.

Конфликты параллелизма

Когда одно соединение с базой данных пытается обновить часть данных, которая была изменена в другом соединении после того, как запись была прочитана первым соединением, возникает конфликт параллельного доступа. То есть, если первый процесс читает данные, после чего те же данные читает второй процесс, и второй процесс обновляет те же данные до того, как это сможет сделать первый процесс, возникает конфликт, когда первый процесс попытается обновить эти данные. Также верно, что

если первый процесс обновляет данные перед вторым процессом, то второй процесс столкнется с конфликтом параллельного доступа при попытке обновить данные. Если несколько соединений могут обращаться к базе данных и проводить изменения, то возникновение конфликтов — вопрос лишь времени и везения.

Когда возникает конфликт, приложение должно предпринять какие-то действия для его разрешения. Например, администратор веб-сайта может просматривать страницу, отображающую данные для обычного пользователя, которая позволяет администратору обновлять эти данные. Если после того, как страница администратора прочитает пользовательские данные из базы данных, а обычный пользователь обратится к странице, отображающей его данные, и внесет изменение, то возникнет конфликт, когда администратор сохранит свои изменения в базе данных. Если же конфликт не возникнет, то изменения обычного пользователя будут переписаны и утеряны. Может случиться иначе: изменения обычного пользователя будут сохранены, а изменения администратора — утеряны. Какое поведение следует считать правильным в каждом конкретном случае — это сложная проблема. Первый шаг — обнаружить ее. Второй шаг — разрешить.

Существуют два базовых подхода к разрешению конфликтов параллельного доступа — *оптимистический* и *пессимистический*.

Оптимистический параллелизм

Как следует из названия, оптимистический подход к разрешению конфликтов параллельного доступа исходит из того, что в большинстве случаев конфликты не происходят. Поэтому никаких блокировок на данные во время их чтения из базы не устанавливается. Когда вдруг случится конфликт при попытке обновить одни и те же данные, тогда им и следует заняться. Оптимистическая обработка конфликтов параллелизма более сложна, чем пессимистическая, но работает лучше в современных приложениях с очень большим количеством пользователей. Представьте, насколько раздражало бы, если бы каждый раз, когда необходимо взглянуть на лот на любимом аукционном сайте, это не удавалось бы сделать потому, что кто-то другой в этот момент просматривает тот же самый лот, и нужная запись блокирована, потому что кто-то предлагает свою цену за него. Вряд ли такое проектное решение можно назвать удачным.

В LINQ to SQL используется оптимистический подход к обработке конфликтов параллелизма. К счастью, LINQ to SQL в состоянии обнаружить и разрешить конфликты параллельного доступа настолько просто, насколько это вообще возможно. В нем даже предусмотрен метод обработки разрешения, если вам это подходит.

Обнаружение конфликтов

Как уже упоминалось, первым шагом является обнаружение конфликтов. В LINQ to SQL реализовано два подхода к обнаружению конфликтов параллелизма. Если указано свойство `IsVersion` атрибута `Column` свойства сущностного класса, и оно имеет значение `true`, то значение этого, и только этого, свойства сущностного класса, будет использовано для определения возникновения конфликта параллельного доступа.

Если ни одно из свойств сущностного класса не имеет установленного в `true` свойства `IsVersion` атрибута `Column`, то LINQ to SQL позволяет управлять тем, какие именно свойства сущностного класса участвуют в обнаружении конфликтов параллельного доступа через свойство `UpdateCheck` атрибута `Column`, заданного на отображаемом свойстве сущностного класса. Перечисление `UpdateCheck` предусматривает три возможных значения: `Never`, `Always` и `WhenChanged`.

UpdateCheck

Если свойство UpdateCheck атрибута для отображаемого свойства сущностного класса установлено в UpdateCheck.Never, то это свойство сущностного класса не будет участвовать в обнаружении конфликтов параллельного доступа. Если же оно установлено в UpdateCheck.Always, то такое свойство сущностного класса всегда будет участвовать в разрешении конфликтов параллелизма, независимо от того, было ли изменено его значение с момента первоначального извлечения и помещения в кэш объекта DataContext. Если свойство UpdateCheck установлено в UpdateCheck.WhenChanged, то это свойство сущностного класса будет участвовать в проверке обновления, только если его значение было изменено с момента первоначальной загрузки в кэш объекта DataContext. Если свойство UpdateCheck атрибута не указано, по умолчанию принимается UpdateCheck.Always.

Помочь разобраться в формальной работе обнаружения конфликтов может помочь понимание его текущей реализации. Когда вызывается метод SubmitChanges, процессор изменений генерирует необходимые операторы SQL, чтобы сохранить все изменения сущностных объектов в базе данных. Когда требуется обновить запись, то вместо того, чтобы указывать только первичный ключ в конструкции where для нахождения соответствующей записи для обновления, наряду с первичным ключом он перечисляет все столбцы, участвующие в обнаружении конфликта. Если свойство UpdateCheck атрибута сущностного класса установлено в UpdateCheck.Always, столбец, на который отображено это свойство сущностного класса, и его исходное значение будут всегда присутствовать в конструкции where. Если свойство UpdateCheck атрибута сущностного класса установлено в UpdateCheck.WhenChanged, то в процессе будет участвовать лишь текущее значение свойства сущностного объекта, которое было изменено по сравнению с его исходным значением, в конструкции where будет указано его исходное значение. Если свойство UpdateCheck имеет значение UpdateCheck.Never, то столбец, на который отображено данное свойство сущностного класса, в конструкции where присутствовать не будет.

Например, предположим, что в сущностном объекте Customer указано свойство UpdateCheck для CompanyName как UpdateCheck.Always, ContactName — как UpdateCheck.WhenChanged и ContactTitle — как UpdateCheck.Never. Если все эти три свойства сущностного класса были модифицированы в объекте заказчика, то сгенерированный оператор SQL будет выглядеть следующим образом:

```
Update Customers
Set CompanyName = 'Art Sanders Park',
  ContactName = 'Samuel Arthur Sanders',
  ContactTitle = 'President'
Where CompanyName = 'Lonesome Pine Restaurant' AND
  ContactName = 'Fran Wilson' AND
  CustomerID = 'LONEP'
```

В приведенном примере значения столбцов в выражении where — это исходные значения, прочитанные из базы данных при первом извлечении сущностного объекта, успешном завершении вызова метода SubmitChanges или же вызове метода Refresh.

Поскольку свойство UpdateCheck атрибута свойства CompanyName указано как UpdateCheck.Always, соответствующий ему столбец будет включен в конструкцию where, независимо от того, было ли это свойство изменено в сущностном объекте. А так как свойство UpdateCheck атрибута свойства ContactName определено как UpdateCheck.WhenChanged, и значение этого свойства сущностного класса было изменено в сущностном объекте, оно также включается в выражение where. Из-за того, что свойство UpdateCheck атрибута свойства ContactTitle задано как UpdateCheck.Never, соответствующий ему столбец не включен в выражение where, несмотря на то, что значение этого свойства также было изменено.

При выполнении этого оператора SQL, если любое из значений свойств сущностного класса, указанных в части `where`, не будет соответствовать тому, что есть в базе данных, то запись не будет найдена, а, следовательно, и не будет обновлена. Именно так обнаруживается конфликт параллельного доступа. В случае обнаружения конфликта генерируется исключение `ChangeConflictException`. Чтобы увидеть, как выглядит сгенерированный оператор `update`, давайте рассмотрим код в листинге 17.1.

Листинг 17.1. Инициализация обновления в базе данных для того, чтобы увидеть, как обнаруживаются конфликты параллельного доступа

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");

db.Log = Console.Out;

Customer cust = db.Customers.Where(c => c.CustomerID == "LONEP").SingleOrDefault();
string name = cust.ContactName; // чтобы восстановить позднее.
cust.ContactName = "Neo Anderson";

db.SubmitChanges();

// Восстановление базы данных.
cust.ContactName = name;
db.SubmitChanges();
```

Об этом запросе сказать особо нечего. Фактически единственное, что здесь следует отметить — это вызов операции `SingleOrDefault` вместо `Single`, которая используется обычно — просто, чтобы подстраховаться на случай, если запись не будет найдена. В этом случае известно, что запись будет найдена, но нужно напомнить, что всегда необходимо заботиться о том, чтобы код мог правильно справиться с подобными ситуациями.

Все, что действительно интересует — это сгенерированный оператор `update`. Посмотрим на результат:

```
SELECT [t0].[CustomerID], [t0].[CompanyName], [t0].[ContactName],
[t0].[ContactTitle], [t0].[Address], [t0].[City], [t0].[Region], [t0].[PostalCode],
[t0].[Country], [t0].[Phone], [t0].[Fax]
FROM [dbo].[Customers] AS [t0]
WHERE [t0].[CustomerID] = @p0
-- @p0: Input String (Size = 5; Prec = 0; Scale = 0) [LONEP]
-- Context: SqlProvider(Sql2008) Model: AttributedMetaModel Build: 3.5.30729.4926

UPDATE [dbo].[Customers]
SET [ContactName] = @p11
WHERE ([CustomerID] = @p0) AND ([CompanyName] = @p1) AND ([ContactName] = @p2) AND
([ContactTitle] = @p3) AND ([Address] = @p4) AND ([City] = @p5) AND ([Region] = @p6)
AND ([PostalCode] = @p7) AND ([Country] = @p8) AND ([Phone] = @p9) AND ([Fax] = @p10)
-- @p0: Input StringFixedLength (Size = 5; Prec = 0; Scale = 0) [LONEP]
-- @p1: Input String (Size = 24; Prec = 0; Scale = 0) [Lonesome Pine Restaurant]
-- @p2: Input String (Size = 11; Prec = 0; Scale = 0) [Fran Wilson]
-- @p3: Input String (Size = 13; Prec = 0; Scale = 0) [Sales Manager]
-- @p4: Input String (Size = 18; Prec = 0; Scale = 0) [89 Chiaroscuro Rd.]
-- @p5: Input String (Size = 8; Prec = 0; Scale = 0) [Portland]
-- @p6: Input String (Size = 2; Prec = 0; Scale = 0) [OR]
-- @p7: Input String (Size = 5; Prec = 0; Scale = 0) [97219]
-- @p8: Input String (Size = 3; Prec = 0; Scale = 0) [USA]
-- @p9: Input String (Size = 14; Prec = 0; Scale = 0) [(503) 555-9573]
-- @p10: Input String (Size = 14; Prec = 0; Scale = 0) [(503) 555-9646]
-- @p11: Input String (Size = 12; Prec = 0; Scale = 0) [Neo Anderson]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.20706.1
```

```
UPDATE [dbo].[Customers]
SET [ContactName] = @p11
WHERE ([CustomerID] = @p0) AND ([CompanyName] = @p1) AND ([ContactName] = @p2) AND
([ContactTitle] = @p3) AND ([Address] = @p4) AND ([City] = @p5) AND ([Region] = @p6)
AND ([PostalCode] = @p7) AND ([Country] = @p8) AND ([Phone] = @p9) AND ([Fax] = @p10)
-- @p0: Input StringFixedLength (Size = 5; Prec = 0; Scale = 0) [LONEP]
-- @p1: Input String (Size = 24; Prec = 0; Scale = 0) [Lonesome Pine Restaurant]
-- @p2: Input String (Size = 12; Prec = 0; Scale = 0) [Neo Anderson]
-- @p3: Input String (Size = 13; Prec = 0; Scale = 0) [Sales Manager]
-- @p4: Input String (Size = 18; Prec = 0; Scale = 0) [89 Chiaroscuro Rd.]
-- @p5: Input String (Size = 8; Prec = 0; Scale = 0) [Portland]
-- @p6: Input String (Size = 2; Prec = 0; Scale = 0) [OR]
-- @p7: Input String (Size = 5; Prec = 0; Scale = 0) [97219]
-- @p8: Input String (Size = 3; Prec = 0; Scale = 0) [USA]
-- @p9: Input String (Size = 14; Prec = 0; Scale = 0) [(503) 555-9573]
-- @p10: Input String (Size = 14; Prec = 0; Scale = 0) [(503) 555-9646]
-- @p11: Input String (Size = 11; Prec = 0; Scale = 0) [Fran Wilson]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.20706.1
```

Обратите внимание, что в первом операторе `update` конструкция `where` указывает, что `ContactName` должно быть равно `"Fran  Wilson"`, т.е. первоначальному значению `ContactName`. Если какой-то другой процесс изменит `ContactName` после его прочтения, ни одна запись не будет соответствовать условию `where`, поэтому ни одна запись не будет обновлена.

Поскольку ни одно из свойств сущностного класса `Customer` не указывает свойство атрибута `UpdateCheck`, все они по умолчанию устанавливаются в `UpdateCheck.Always`, так что в конструкции `where` этого оператора `update` присутствуют все отображенные свойства сущностного класса.

SubmitChanges()

Обнаружение конфликтов параллельного доступа происходит при вызове метода `SubmitChanges`. Во время вызова этого метода можно указать, должен ли прерываться процесс сохранения изменений в базе данных в случае первого обнаружения конфликта или же необходимо пытаться провести все изменения, накапливая конфликты. Это поведение управляется посредством аргумента `ConflictMode`, который может быть передан методу `SubmitChanges`. Если передать значение `ConflictMode.FailOnFirstConflict`, то процесс будет прерван после первого же конфликта. Если же передать `ConflictMode.ContinueOnConflict`, то процесс продолжит попытки провести все необходимые изменения, даже если возникнут конфликты. Если вы предпочтете не указывать `ConflictMode`, то метод `SubmitChanges` по умолчанию примет `ConflictMode.FailOnFirstConflict`.

Независимо от заданного `ConflictMode`, если объемлющая транзакция не активна в данном контексте при вызове метода `SubmitChanges`, транзакция будет создана для всех попыток изменения базы данных, предпринимаемых в рамках данного вызова метода `SubmitChanges`. Если же активна объемлющая транзакция, то `DataContext` использует ее. При генерации исключения во время вызова метода `SubmitChanges` транзакция будет отменена. Это значит, что даже изменения не конфликтующих сущностных объектов, успешно помещенные в базу данных, также будут отменены.

ChangeConflictException

Если случается конфликт параллельного доступа, то независимо от того, установлен `ConflictMode` в `FailOnFirstConflict` или `ContinueOnConflict`, исключение `ChangeConflictException` будет сгенерировано.

За счет перехвата исключения `ChangeConflictException` обнаруживается возникновение конфликта.

Разрешение конфликтов

Как только конфликт параллельного доступа обнаружен путем перехвата исключения `ChangeConflictException`, следующим шагом, скорее всего, будет его разрешение. Вы можете предпочесть предпринять какое-то другое действие, но разрешение конфликтов — наиболее вероятный выход. Поначалу этот процесс может показаться чудовищно сложным, но, к счастью, LINQ to SQL также облегчает эту задачу, предоставляя в наше распоряжение метод `ResolveAll` и два метода `Resolve`.

`RefreshMode`

При действительном разрешении конфликта с использованием встроенной функциональности разрешения LINQ to SQL посредством вызова метода `ResolveAll` или `Resolve` с помощью режима `RefreshMode` можно управлять способом разрешения конфликта. Его три допустимых значения — `KeepChanges`, `KeepCurrentValues` и `OverwriteCurrentValues`. Эти опции управляют тем, какие данные остаются в свойствах сущностного объекта, когда `DataContext` выполняет разрешение конфликта.

Опция `RefreshMode.KeepChanges` сообщает методу `ResolveAll` или `Resolve` о том, что нужно загрузить изменения из базы данных в текущие значения свойств сущностного класса для каждого столбца, измененного с момента начальной загрузки данных, если только текущий пользователь также не изменил это свойство — в этом случае его значение сохраняется. Порядок приоритетов предохранения данных, от низшего к высшему, такой: первоначальные значения свойств сущностного класса, перезагруженные измененные значения столбцов базы данных, значения свойств сущностного класса, измененные текущим пользователем.

Опция `RefreshMode.KeepCurrentValues` сообщает методу `ResolveAll` или `Resolve` о том, что нужно сохранить первоначальные значения свойств сущностного класса и изменения текущего пользователя, отклонив все изменения, проведенные в базе после начальной загрузки. Порядок приоритетов предохранения данных, от низшего к высшему, такой: первоначальные значения свойств сущностного класса, измененные текущим пользователем значения свойств сущностного класса.

Опция `RefreshMode.OverwriteCurrentValues` сообщает методу `ResolveAll` или `Resolve` о том, что нужно загрузить изменения из базы данных для любого столбца, измененного с момента начальной загрузки данных, и отбросить изменения свойств сущностного класса, проведенные текущим пользователем. Порядок приоритетов предохранения данных, от низшего к высшему, такой: первоначальные значения свойств сущностного класса, затем — перезагруженные значения столбцов.

Подходы к разрешению конфликтов

Существуют три подхода к разрешению конфликтов: простейший, легкий и ручной. Простейший подход заключается в вызове метода `ResolveAll` на коллекции `DataContext.ChangeConflicts` с передачей `RefreshMode` и необязательного булевского значения, говорящего о том, нужно ли автоматически разрешать удаленные записи.

Автоматическое разрешение удаленных записей означает маркировку соответствующего удаленного сущностного объекта как успешно удаленного, даже когда оно не произошло из-за конфликта параллельного доступа, чтобы при следующем вызове метода `SubmitChanges` объект `DataContext` не пытался удалить снова записи базы данных, соответствующие удаленным сущностным объектам. По сути, LINQ to SQL говорится, что он должен представлять, будто бы записи были успешно удалены, потому что кто-то удалил их раньше.

Легкий подход состоит в перечислении всех объектов `ObjectChangeConflict` из коллекции `DataContext.ChangeConflicts` с вызовом метода `Resolve` на каждом из них.

Если же, однако, нужна какая-то специальная обработка, всегда есть выбор обработать разрешение конфликта самостоятельно, перечислив элементы коллекции ChangeConflicts объекта DataContext, а затем перечислив все элементы коллекции MemberConflicts объекта ObjectChangeConflict, вызывая метод Resolve на каждом объекте MemberChangeConflict из этой коллекции. Даже при ручной обработке имеющиеся методы позволяют решить эту задачу достаточно легко.

DataContext.ChangeConflicts.ResolveAll()

Разрешение конфликтов не сложно. Для этого просто перехватывается исключение ChangeConflictException и вызывается метод ResolveAll на коллекции DataContext.ChangeConflicts. Все что требуется — это решить, какой режим RefreshMode использовать, и нужно ли автоматически разрешать конфликты удаленных записей.

Применение такого подхода вызовет одинаковое разрешение всех конфликтов на базе переданного RefreshMode. Если необходим более тонкий контроль при разрешении конфликтов, используйте один из более сложных подходов, которые рассматриваются ниже.

В листинге 17.2 конфликт разрешается с использованием этого подхода. Поскольку пример довольно сложен, он сопровождается дополнительными пояснениями.

Листинг 17.2. Пример разрешения конфликтов с помощью DataContext.ChangeConflicts.ResolveAll()

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
Customer cust = db.Customers.Where(c => c.CustomerID == "LAZYK").SingleOrDefault();
ExecuteStatementInDb(String.Format(
  @"update Customers
    set ContactName = 'Samuel Arthur Sanders'
    where CustomerID = 'LAZYK'"));
```

В коде создается Northwind DataContext, с помощью LINQ to SQL запрашивается заказчик и вносится изменение в значение столбца ContactName извлеченного заказчика в базе данных посредством ADO.NET. Тем самым закладывается фундамент для потенциального конфликта параллельного доступа.

Теперь нужно провести изменение в сущностном объекте и попытаться сохранить его в базе данных:

```
cust.ContactTitle = "President";
try
{
  db.SubmitChanges(ConflictMode.ContinueOnConflict);
}
catch (ChangeConflictException)
{
```

Обратите внимание, что вызов метода SubmitChanges помещен в блок try/catch. Чтобы правильно обнаруживать конфликты параллельного доступа, перехватывается исключение ChangeConflictException. Понадобится только вызвать метод ResolveAll и снова попытаться сохранить изменения.

```
db.ChangeConflicts.ResolveAll(RefreshMode.KeepChanges);
try
{
  db.SubmitChanges(ConflictMode.ContinueOnConflict);
  cust = db.Customers.Where(c => c.CustomerID == "LAZYK").SingleOrDefault();
```

```
    Console.WriteLine("ContactName = {0} : ContactTitle = {1}",
      cust.ContactName, cust.ContactTitle);
  }
  catch (ChangeConflictException)
  {
    Console.WriteLine("Конфликт возник снова, завершение.");
  }
}
```

В приведенном коде вызывается метод `ResolveAll`, которому передается `RefreshMode` со значением `KeepChanges`. Затем вновь вызывается метод `SubmitChanges`, помещенный в собственный блок `try/catch`. После этого из базы данных опять запрашивается заказчик и на консоль выводятся значения `ContactName` и `ContactTitle` заказчика, чтобы доказать, что ни изменение ADO.NET, ни изменение LINQ to SQL не утеряны. Если этот вызов метода `SubmitChanges` сгенерирует исключение, об этом просто выдается сообщение и дальнейшие попытки прекращаются.

Все, что осталось сделать — это восстановить базу данных в исходное состояние, чтобы пример можно было запускать более одного раза:

```
// Сбросить базу данных в исходное состояние.
ExecuteStatementInDb(String.Format(
  @"update Customers
    set ContactName = 'John Steel', ContactTitle = 'Marketing Manager'
    where CustomerID = 'LAZYK'"));
```

Если присмотреться, то если не считать кода, вызвавшего конфликт, который обычно писать не придется, и кода восстановления базы данных в конце примера, который также писать не придется, то разрешение конфликтов параллелизма при таком подходе чрезвычайно просто. Вы помещаете вызов метода `SubmitChanges` в блок `try/catch`, перехватываете исключение `ChangeConflictException`, вызываете метод `ResolveAll` и повторяете вызов метода `SubmitChanges`. Вот и все. Взглянем на результат запуска кода из листинга 17.2.

```
Выполнение оператора SQL для базы данных с помощью ADO.NET...
База данных обновлена.
ContactName = Samuel Arthur Sanders : ContactTitle = President
Выполнение оператора SQL для базы данных с помощью ADO.NET...
База данных обновлена.
```

Как видно по результатам, и изменение ADO.NET в `ContactName`, и изменение LINQ to SQL в `ContactTitle` сохранены в базе данных. Это очень простой подход к разрешению конфликтов параллельного доступа.

ObjectChangeConflict.Resolve()

Если разрешение всех конфликтов с тем же самым `RefreshMode` не работает, можно выбрать подход с перечислением всех конфликтов из коллекции `DataContext.ChangeConflicts` и обрабатывать их индивидуально. Можно обрабатывать каждый из них, вызывая на нем метод `Resolve`. Это позволит передавать разные значения `RefreshMode` для каждого конфликта.

Разрешение конфликтов на этом уровне сродни разрешению их на уровне сущностного объекта. Переданный `RefreshMode` будет применен к каждому свойству сущностного класса в конфликтующем сущностном объекте. Если нужен более тонкий контроль, чем позволяет такой подход, рассмотрите использование ручного подхода, который рассматривается далее.

В листинге 17.3 этот подход будет продемонстрирован на примере. Код будет таким же, как в листинге 17.2, только вызов метода `DataContext.ChangeConflicts.ResolveAll` будет заменен перечислением коллекции `ChangeConflicts`.

Листинг 17.3. Пример разрешения конфликтов с помощью `ObjectChangeConflict.Resolve()`

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
Customer cust = db.Customers.Where(c => c.CustomerID == "LAZYK").SingleOrDefault();
ExecuteStatementInDb(String.Format(
  @"update Customers
    set ContactName = 'Samuel Arthur Sanders'
    where CustomerID = 'LAZYK'"));
cust.ContactTitle = "President";
try
{
  db.SubmitChanges(ConflictMode.ContinueOnConflict);
}
catch (ChangeConflictException)
{
  foreach (ObjectChangeConflict conflict in db.ChangeConflicts)
  {
    Console.WriteLine("Конфликт произошел в заказчике {0}.",
      ((Customer)conflict.Object).CustomerID);
    Console.WriteLine("Вызов Resolve...");
    conflict.Resolve(RefreshMode.KeepChanges);
    Console.WriteLine("Конфликт разрешен.{0}", System.Environment.NewLine);
  }
  try
  {
    db.SubmitChanges(ConflictMode.ContinueOnConflict);
    cust = db.Customers.Where(c => c.CustomerID == "LAZYK").SingleOrDefault();
    Console.WriteLine("ContactName = {0} : ContactTitle = {1}",
      cust.ContactName, cust.ContactTitle);
  }
  catch (ChangeConflictException)
  {
    Console.WriteLine("Конфликт возник снова, завершение.");
  }
}
// Сбросить базу данных в исходное состояние.
ExecuteStatementInDb(String.Format(
  @"update Customers
    set ContactName = 'John Steel', ContactTitle = 'Marketing Manager'
    where CustomerID = 'LAZYK'"));
```

Обратите внимание, что вместо вызова метода `DataContext.ChangeConflicts.ResolveAll` выполняется перечисление коллекции `ChangeConflicts` с вызовом метода `Resolve` на каждом объекте `ObjectChangeConflict` из этой коллекции. Затем, как и в предыдущем листинге, снова вызывается метод `SubmitChanges`, запрашивается заказчик и на консоль выводятся важнейшие свойства сущностного класса. Конечно, после этого база данных восстанавливается в исходном состоянии.

Вот результат работы листинга 17.3:

```
Выполнение оператора SQL для базы данных с помощью ADO.NET...
База данных обновлена.
Конфликт произошел в заказчике LAZYK.
Вызов Resolve...
Конфликт разрешен.

ContactName = Samuel Arthur Sanders : ContactTitle = President
Выполнение оператора SQL для базы данных с помощью ADO.NET...
База данных обновлена.
```

Все работает, как и хотелось. В реальном рабочем коде может понадобиться запустить цикл с вызовами метода `SubmitChanges` и разрешением конфликта — просто чтобы противостоять невезению, связанному с появлением дополнительных конфликтов при таком редком стечении обстоятельств. Если вы сделаете это, не забудьте каким-то образом ограничить количество итераций цикла, чтобы предотвратить бесконечное его выполнение в случае, когда в системе возникнут серьезные неполадки.

`MemberChangeConflict.Resolve()`

При первом подходе вызывается метод для разрешения всех конфликтов одним способом. Это — простейший способ разрешения конфликтов. При втором подходе вызывается метод для разрешения конфликта для единственного конфликтующего объекта. Это обеспечивает гибкость для разрешения конфликтов с каждым сущностным объектом по-своему. Это — легкий способ. Что осталось? Остался подход к разрешению конфликтов вручную.

Пусть объяснения не пугают. Даже при нормальном подходе обнаружение конфликтов параллелизма, вероятно, проще, чем можно было ожидать. Принятие такого подхода позволит применять разные значения `RefreshMode` для индивидуальных свойств сущностного объекта.

Подобно второму подходу к разрешению конфликтов, производится перечисление объектов `ObectChangeConflict` из коллекции `DataContext.ChangeConflicts`. Но вместо вызова метода `Resolve` на каждом объекте `ObectChangeConflict` выполняется перечисление коллекции `MemberConflicts` и вызов метода `Resolve` каждого объекта `MemberChangeConflict`.

На этом уровне объект `MemberChangeConflict` принадлежит к определенному свойству сущностного класса из конфликтующего сущностного объекта. Это позволяет отступать от общего `RefreshMode` для любого свойства сущностного класса по своему усмотрению.

Метод `Resolve` позволяет передавать либо `RefreshMode`, либо действительное значение, которое должно сменить текущее значение. Это обеспечивает замечательную гибкость.

В примере ручного разрешения конфликтов, приведенном в листинге 17.4, предполагается существование требования, что если случится конфликт со столбцом `ContactName` в базе данных, код должен оставить значение базы как есть, но любые другие столбцы в записи могут быть обновлены.

Чтобы реализовать это, используется тот же базовый код, что и в листинге 17.3, но вместо вызова метода `Resolve` на объекте `ObjectChangeConflict` выполняется перечисление членов коллекции `MemberConflicts` каждого объекта. Затем для каждого объекта `MemberConflict` из этой коллекции, если свойством сущностного объекта, вызвавшего конфликт, является `ContactName`, то поддерживается значение в базе данных за счет передачи `RefreshMode`, равного `RefreshMode.OverwriteCurrentValues`, методу `Resolve`. Если же конфликтующее свойство сущностного объекта — не `ContactName`, поддерживается измененное значение за счет передачи методу `Resolve` значения `RefreshMode`, равного `RefreshMode.KeepChanges`.

Кроме того, чтобы сделать пример более интересным, когда база данных обновляется с помощью ADO.NET для создания конфликтной ситуации, также обновляется столбец `ContactTitle`. Это заставит конфликтовать два свойства сущностного объекта. Одно — `ContactName` — должно быть обработано так, чтобы сохранилось значение из базы данных. Второе — `ContactTitle` — должно быть обработано так, чтобы сохранилось значение, установленное LINQ to SQL.

Рассмотрим код в листинге 17.4.

Листинг 17.4. Пример ручного разрешения конфликтов

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
Customer cust = db.Customers.Where(c => c.CustomerID == "LAZYK").SingleOrDefault();

ExecuteStatementInDb(String.Format(
  @"update Customers
    set ContactName = 'Samuel Arthur Sanders',
      ContactTitle = 'CEO'
    where CustomerID = 'LAZYK'"));

cust.ContactName = "Viola Sanders";
cust.ContactTitle = "President";
try
{
  db.SubmitChanges(ConflictMode.ContinueOnConflict);
}
catch (ChangeConflictException)
{
  foreach (ObjectChangeConflict conflict in db.ChangeConflicts)
  {
    Console.WriteLine("Конфликт произошел в заказчике {0}.",
    ((Customer)conflict.Object).CustomerID);
    foreach (MemberChangeConflict memberConflict in conflict.MemberConflicts)
    {
      Console.WriteLine("Вызов Resolve для {0}...",
      memberConflict.Member.Name);
      if (memberConflict.Member.Name.Equals("ContactName"))
      {
        memberConflict.Resolve(RefreshMode.OverwriteCurrentValues);
      }
      else
      {
        memberConflict.Resolve(RefreshMode.KeepChanges);
      }
      Console.WriteLine("Конфликт разрешен.{0}", System.Environment.NewLine);
    }
  }
  try
  {
    db.SubmitChanges(ConflictMode.ContinueOnConflict);
    cust = db.Customers.Where(c => c.CustomerID == "LAZYK").SingleOrDefault();
    Console.WriteLine("ContactName = {0} : ContactTitle = {1}",
      cust.ContactName, cust.ContactTitle);
  }
  catch (ChangeConflictException)
  {
    Console.WriteLine("Конфликт возник снова, завершение.");
  }
}

// Сбросить базу данных в исходное состояние.
ExecuteStatementInDb(String.Format(
  @"update Customers
    set ContactName = 'John Steel', ContactTitle = 'Marketing Manager'
    where CustomerID = 'LAZYK'"));
```

Одним из существенных изменений является то, что также обновляется `ContactTitle` через ADO.NET. Это приводит к конфликту в двух свойствах объекта при вызове `SubmitChanges`. Затем вместо вызова метода `Resolve` на объекте

ObjectChangeConflict перечисляются члены коллекции MemeberCinflicts с проверкой каждого свойства сущностного объекта. Если текущим свойством являеься ContactName, вызывается метод Resolve с RefreshMode, установленным в RefreshMode. KeepChanges, чтобы сохранить значение, определенное кодом LINQ to SQL.

Взглянем на результаты работы кода из листинга 17.4.

```
Выполнение оператора SQL для базы данных с помощью ADO.NET...
База данных обновлена.
Конфликт произошел в заказчике LAZYK.
Вызов Resolve для ContactName...
Конфликт разрешен.

Вызов Resolve для ContactTitle...
Конфликт разрешен.

ContactName = Samuel Arthur Sanders : ContactTitle = President
Выполнение оператора SQL для базы данных с помощью ADO.NET...
База данных обновлена.
```

Здесь видно, что оба свойства — ContactName и ContactTitle — конфликтовали, и эти конфликты были разрешены. Также, просматривая вывод свойств ContactName и ContactTitle в конце, можно заметить, что значение из базы данных было сохранено для свойства ContactName, но проигнорировано для свойства ContactTitle, которое сохранило значение, установленное кодом LINQ to SQL. Именно этого и планировалось добиться.

Действительный код, обрабатывающий разрешение конфликта вручную, не так плох. Но, конечно, все эти усилия оправданы только для специализированного разрешения конфликтов.

Пессимистический параллелизм

Как следует из названия, пессимистический параллелизм предполагает худшее, и можно просто рассчитывать на тот факт, что запись, которая читается, вызовет конфликт на момент ее обновления. К счастью, есть возможность справиться и с этим. Для этого нужно просто поместить чтение и обновление в базе данных внутрь одной транзакции.

При пессимистическом подходе к параллелизму нет конфликтов, которые нужно разрешать, потому что база данных блокирована транзакцией, так что никто другой не сможет модифицировать ее в это время.

Чтобы проверить это, создается объект TransactionScope и получается сущностный объект для заказчика LAZYK. Затем создается другой объект TransactionScope со свойством TransactionScopeOption, равным RequiresNew. При этом код ADO.NET не участвует в объемлющей транзакции, принадлежащей ранее созданному объекту TransactionScope. После этого предпринимается попытка обновить ту же запись в базе данных с помощью ADO.NET. Но поскольку уже существует открытая транзакция, блокирующая базу данных, оператор обновления ADO.NET будет блокирован и в конечном итоге отменен по таймауту. Далее обновляется свойство ContactName сущностного объекта, вызывается метод SubmitChanges, снова запрашивается запись заказчика для вывода на консоль значения ContactName, чтобы доказать, что оно было обновлено LINQ to SQL, и транзакция завершается.

На заметку! Чтобы следующий пример скомпилировался, в проект потребуется добавить ссылку на сборку System.Transactions.dll.

Код описанного примера приведен в листинге 17.5.

Листинг 17.5. Пример пессимистического параллелизма

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");

using (System.Transactions.TransactionScope transaction =
  new System.Transactions.TransactionScope())
{
  Customer cust =
    db.Customers.Where(c => c.CustomerID == "LAZYK").SingleOrDefault();
  try
  {
    Console.WriteLine("Попытаемся обновить ContactName для LAZYK с помощью ADO.NET.");
    Console.WriteLine("  Понадобится подождать истечения таймаута...");
    using (System.Transactions.TransactionScope t2 =
      new System.Transactions.TransactionScope(
        System.Transactions.TransactionScopeOption.RequiresNew))
    {
      ExecuteStatementInDb(String.Format(
        @"update Customers
          set ContactName = 'Samuel Arthur Sanders'
          where CustomerID = 'LAZYK'"));
       t2.Complete();
    }
    Console.WriteLine("Значение ContactName для LAZYK обновлено.{0}",
      System.Environment.NewLine);
  }
  catch (Exception ex)
  {
    Console.WriteLine(
     "При попытке обновления LAZYK с помощью ADO.NET возникло исключение:{0} {1}{0}",
        System.Environment.NewLine, ex.Message);
  }
  cust.ContactName = "Viola Sanders";
  db.SubmitChanges();
  cust = db.Customers.Where(c => c.CustomerID == "LAZYK").SingleOrDefault();
  Console.WriteLine("Имя контактной персоны заказчика: {0}", cust.ContactName);

  transaction.Complete();
}
// Вернуть базу данных в исходное состояние.
ExecuteStatementInDb(String.Format(
  @"update Customers
    set ContactName = 'John Steel',
      ContactTitle = 'Marketing Manager'
    where CustomerID = 'LAZYK'"));
```

Совет. Если при работе с любым примером, использующим объект `TransactionScope`, вы получаете исключение типа `MSDTC on server '[сервер]\SQLEXPRESS' is unavailable` (Служба MSDTC на сервере '[сервер]\SQLEXPRESS' не доступна), удостоверьтесь, что запущена служба по имени Distributed Transaction Coordinator (Распределенный координатор транзакций).

Этот код не так сложен, как может показаться на первый взгляд. Первое, что здесь делается — это создание объекта `TransactionScope`. Теперь используется пессимистический подход к параллелизму, предотвращающий модификацию данных кем-либо. Затем с помощью LINQ to SQL запрашивается заказчик. Далее создается другой объект `TransactionScope`, чтобы помешать коду ADO.NET, который будет вызывать-

ся, участвовать в транзакции исходного объекта TransactionScope. После создания второго объекта TransactionScope предпринимается попытка обновить заказчика в базе данных с использованием ADO.NET. Код ADO.NET не получит возможности выполнить обновление из-за начальной транзакции, в результате чего будет сгенерировано исключение таймаута. Затем изменяется значение ContactName заказчика, изменение сохраняется в базе данных вызовом метода SubmitChanges, заказчик запрашивается снова и его значение ContactName выводится на консоль, чтобы доказать, что изменение сохранено. После этого исходная транзакция завершается вызовом на ней метода Complete.

Как обычно, в конце кода состояние базы данных восстанавливается. Ниже показаны результаты выполнения кода из листинга 17.5:

```
Попытаемся обновить ContactName для LAZYK с помощью ADO.NET.
  Понадобится подождать истечения таймаута...
Выполнение оператора SQL для базы данных с помощью ADO.NET...
При попытке обновления LAZYK с помощью ADO.NET возникло исключение:
  Timeout expired. The timeout period elapsed prior to completion of the
operation or the server is not responding.
The statement has been terminated.
```

Таймаут истек. Время таймаута истекло до завершения операции или сервер не отвечает.
Оператор принудительно завершен.

```
Имя контактной персоны заказчика: Viola Sanders
Выполнение оператора SQL для базы данных с помощью ADO.NET...
База данных обновлена.
```

Обратите внимание, что при попытке обновить базу данных посредством ADO.NET генерируется исключение таймаута. Пусть не вводит в заблуждение отложенное выполнение запроса. Помните, что многие из операций LINQ являются отложенными. В случае этого примера мой запрос LINQ to SQL вызывает операцию SingleOrDefault, так что запрос не является отложенным, что требует объявления запроса внутри контекста объекта TransactionScope. Если бы операция SingleOrDefault не вызывалась, этот запрос мог быть объявлен перед созданием объекта TransactionScope, если действительное выполнение запроса происходило бы внутри контекста TransactionScope. Таким образом, можно было бы просто заставить запрос LINQ вернуть последовательность IEnumerable<T> до создания объекта TransactionScope и затем внутри контекста объекта TransactionScope вызвать операцию SingleOrDefault на возвращенной последовательности, возвращая единственный объект Customer, соответствующий запросу.

При таком подходе всегда необходимо оценивать, насколько много работы будет выполняться внутри контекста объекта TransactionScope, потому что на все это время база данных будет блокирована.

Альтернативный подход для средних звеньев и серверов

Существует и альтернативный подход для обработки конфликтов параллельного доступа, когда они случаются в среднем звене или на сервере. Иногда, когда возникают такие конфликты, может быть проще только создать новый объект DataContext, применить изменения и вызвать SubmitChanges.

Рассмотрим пример веб-приложения ASP.NET. Из-за характера коммуникаций между клиентом-браузером и веб-сервером, предполагающим работу без установки соединения, вы с успехом можете создавать новый объект DataContext при всякой отправке HTTP на веб-сервер, когда требуется выполнить запрос LINQ to SQL. Помните, что поскольку прочтенные из базы данных считаются немедленно устаревающими, будет не

слишком хорошей идеей удерживать объект DataContext открытым в течение длительного времени, если вы намерены проводить изменения.

Когда пользователь впервые обращается к веб-странице и для него извлекаются данные, нет особого смысла заставлять объект DataContext ожидать обратной отправки клиента с попыткой обновить данные. DataContext все равно не будет доступен в процессе этого ожидания, если только не сохранять его каким-то образом между подключениями, например, в состоянии сеанса. Но даже если он будет доступен, задержки между соединениями могут быть очень длинными. Чем дольше вы ожидаете между первоначальным чтением базы данных для отображения страницы и попыткой обновления этой базы данных в последующей обратной отправке, тем более устареют данные. Вместо того чтобы пытаться удерживать DataContext в таком сценарии, может быть более оправдано создавать новый DataContext при каждой обратной отправке, когда необходимо сохранить данные. Если это так, и возникает конфликт параллельного доступа, не будет большого вреда от создания еще одного DataContext, повторного применения изменений и нового вызова метода SubmitChanges. И поскольку задержка между первоначальным чтением данных при обратной отправке, применением изменений и вызовом метода SubmitChanges будет очень короткой, маловероятно, что вы столкнетесь с конфликтом параллельного доступа при первой попытке, и гораздо меньше — при второй.

Если решено использовать такой подход, то при обратной отправке после конструирования нового DataContext можно извлечь необходимый сущностный объект, как было только что описано, или же поступить иначе. Вместо извлечения сущностного объекта можно создать новый сущностный объект, наполнить необходимые его свойства соответствующими значениями и присоединить их к соответствующей таблице, применив метод Attach объекта Table<T>. В этой точке все будет выглядеть так, будто сущностный объект *был* извлечен из базы данных, несмотря на тот факт, что не каждое поле этого объекта могло быть заполнено значениями.

Прежде чем присоединять сущностный объект к Table<T>, потребуется установить необходимые свойства сущностного класса в соответствующие значения. Это не значит, что для получения этих значений должна запрашиваться база данных; они могут поступать откуда угодно, например, из другого звена. Необходимые свойства сущностного класса включают все свойства, составляющие первичный ключ или устанавливающие идентичность, все свойства, которые планируется изменить, а также все свойства, участвующие в проверке обновления. Чтобы DataContext мог правильно отследить идентичность объекта сущностного класса, понадобится включить свойства сущностного класса, устанавливающие идентичность. Все свойства, которые планируется изменить, должны быть включены для того, чтобы они могли быть обновлены, и правильно работало обнаружение конфликтов параллельного доступа. Также потребуется включить все свойства сущностного класса, участвующие в проверке обновления, для обнаружения конфликтов параллельного доступа. Если сущностный класс имеет свойство, указывающее свойство IsVersion атрибута Column со значением true, такое свойство класса должно быть установлено перед вызовом метода Attach.

Давайте в листинге 17.6 посмотрим, как это делается.

Листинг 17.6. Пример использования Attach() для подключения вновь созданного сущностного объекта

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");

// Создать сущностный объект.
Console.WriteLine("Конструирование пустого объекта Customer.");
Customer cust = new Customer();
```

```
// Сначала должны быть установлены все поля, определяющие идентичность.
Console.WriteLine("Установка первичных ключей.");
cust.CustomerID = "LAZYK";

// Затем установить каждое изменяемое поле.
Console.WriteLine("Установка полей, которые будут изменяться.");
cust.ContactName = "John Steel";

// И, наконец, установить все поля, участвующие в проверке обновления.
// К сожалению, для сущностного класса Customer это будут все поля.
Console.WriteLine("Установка всех полей, принимающий участие в проверке обновления.");
cust.CompanyName = "Lazy K Kountry Store";
cust.ContactTitle = "Marketing Manager";
cust.Address = "12 Orchestra Terrace";
cust.City = "Walla Walla";
cust.Region = "WA";
cust.PostalCode = "99362";
cust.Country = "USA";
cust.Phone = "(509) 555-7969";
cust.Fax = "(509) 555-6221";

// Присоединение к Table<T> по имени Customers;
Console.WriteLine("Присоединение к Table<T> по имени Customers.");
db.Customers.Attach(cust);

// В этой точке можно внести изменения и вызвать SubmitChanges().
Console.WriteLine("Внесение изменений и вызов SubmitChanges().");
cust.ContactName = "Vickey Rattz";
db.SubmitChanges();
 cust = db.Customers.Where(c => c.CustomerID == "LAZYK").SingleOrDefault();
Console.WriteLine("ContactName в базе данных = {0}", cust.ContactName);
 Console.WriteLine("Восстановление изменений и вызов SubmitChanges().");
cust.ContactName = "John Steel";
db.SubmitChanges();
```

Как видите, в коде устанавливаются свойства класса, составляющие первичный ключ, свойства класса, которые планируется изменять, и свойства класса, участвующие в проверке обновления. Как уже упоминалось, эти свойства должны быть установлены в соответствующие значения. Однако это не означает необходимость запроса базы данных. Возможно, нужные значения хранятся в скрытых переменных или в состоянии представления, либо же они были получены от другого звена системы. Затем вызывается метод `Attach` на `Customers Table<Customer>`. Далее изменения вносятся и, наконец, вызывается метод `SubmitChanges`. После этого из базы данных запрашивается заказчик, значение `ContactName` которого выводится на консоль, чтобы доказать, что оно было изменено в базе данных. И, наконец, как всегда, база данных восстанавливается в исходное состояние. Вывод кода из листинга 17.6 показан ниже.

```
Конструирование пустого объекта Customer.
Установка первичных ключей.
Установка полей, которые будут изменяться.
Установка всех полей, принимающий участие в проверке обновления.
Присоединение к Table<T> по имени Customers.
Внесение изменений и вызов SubmitChanges().
ContactName в базе данных = Vickey Rattz
Восстановление изменений и вызов SubmitChanges().
```

Вставка и удаление объектов сущностного класса не требует такого подхода. Их можно просто вставлять или удалять перед вызовом метода `SubmitChanges`. Читайте подразделы “Вставки” и “Удаления” в разделе “Стандартные операции для баз данных” главы 14.

Резюме

В этой главе были представлены исчерпывающие сведения об обнаружении и разрешении конфликтов параллельного доступа. Как было показано, обнаружение и разрешение конфликтов параллельного доступа в LINQ to SQL реализуется очень просто.

Исследования API-интерфейса LINQ to SQL практически завершены. В следующей главе будет дана разнородная информация о LINQ to SQL, для которой не нашлось места в предыдущих главах.

ГЛАВА 18

Дополнительные возможности LINQ to SQL

В этой завершающей главе, посвященной LINQ to SQL, мы обсудим только несколько разных тем. Среди них: список представлений базы данных, наследование сущностных классов и дополнительные сведения о транзакциях.

Предварительные условия для запуска примеров

Для того чтобы запускать примеры из этой главы, понадобится расширенная версия базы данных Northwind и сгенерированные для нее сущностные классы. Прочтите и выполните то, что описано в разделе "Предварительные условия для запуска примеров" главы 12.

Использование API-интерфейса LINQ to SQL

Чтобы запускать примеры этой главы, в проект может потребоваться добавить соответствующие ссылки и директивы `using`. Прочтите и выполните то, что изложено в разделе "Использование LINQ to SQL" главы 12.

Использование API-интерфейса LINQ to XML

Некоторые примеры этой главы требуют дополнительной директивы `using` для пространства имен `System.Xml.Linq`.

Представления базы данных

При генерации сущностных классов для базы данных Northwind в главе 12 указывалась опция `/views`, чтобы получить сущностные классы, отображаемые на представления базы данных, но пока ничего не говорилось о том, как выполнять запросы к ним. Инструменты генерации сущностных классов — SQLMetal и Object Relational Designer — генерируют свойство `Table<T>` в классе `[Your]DataContext` для каждого представления базы данных и создают соответствующий класс `T`. Они опрашиваются подобно таблицам. В общем случае они ведут себя подобно таблицам, отличаясь лишь тем, что доступны только для чтения.

Поскольку сущностные классы, сгенерированные для представлений, не содержат свойств сущностных классов, отображаемых на первичные ключи, они доступны только для чтения. Если вспомнить, что без первичных ключей DataContext не имеет эффективной возможности обеспечить отслеживание идентичности, это очевидно имеет смысл.

Например, база данных Northwind включает представление, которое называется Category Sales for 1997. В связи с этим инструмент SQLMetal сгенерировал общедоступное (public) свойство по имени CategorySalesFor1997s.

Общедоступное свойство для представления базы данных

```
public System.Data.Linq.Table<CategorySalesFor1997> CategorySalesFor1997s
{
  get
  {
    return this.GetTable<CategorySalesFor1997>();
  }
}
```

Инструмент SQLMetal также сгенерировал сущностный класс CategorySalesFor1997. В листинге 18.1 показано, как опрашивается представление базы данных.

Листинг 18.1. Запрос к представлению базы данных

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");

IQueryable<CategorySalesFor1997> seq = from c in db.CategorySalesFor1997s
                                where c.CategorySales > (decimal)100000.00
                                orderby c.CategorySales descending
                                select c;
foreach (CategorySalesFor1997 c in seq)
{
  Console.WriteLine("{0} : {1:C}", c.CategoryName, c.CategorySales);
}
```

Обратите внимание, что в листинге 18.1 представление запрашивается точно так же, как таблица. Посмотрим на результат:

```
Dairy Products : $114,749.78
Beverages : $102,074.31
```

Как уже упоминалось, представления доступны только для чтения. В листинге 18.2 предпринимается попытка вставить запись в представление.

Листинг 18.2. Неудачная попытка вставить запись в представление

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
 db.CategorySalesFor1997s.InsertOnSubmit(
  new CategorySalesFor1997
    { CategoryName = "Legumes", CategorySales = (decimal) 79043.92 });
```

Обратите внимание, что в листинге 18.2 даже не предпринимается попытка вызвать метод SubmitChanges. Причина в том, что код не сможет сделать этого без генерации исключения. Посмотрим на результат:

```
Unhandled Exception: System.InvalidOperationException: Can't perform Create, Update
or Delete operations on 'Table(CategorySalesFor1997)' because it has no primary key.
```

Необработанное исключение: System.InvalidOperationException:
Не удается выполнить операции создания, обновления и удаления на
'Table(CategorySalesFor1997)', потому что она не имеет первичного ключа.
...

Здесь следует сделать одно предупреждение. Хотя методы InsertOnSubmit и DeleteOnSubmit сгенерируют исключение при вызове Table<T>, отображенной на представление базы данных, ничто не помешает внести изменения в свойство сущностного объекта. Можно изменить значение свойства и даже вызвать метод SubmitChanges без генерации исключений, но изменение в свойстве сущностного объекта не будет сохранено в базе данных.

Наследование сущностных классов

До сих пор при обсуждении LINQ to SQL всегда присутствовал единственный сущностный класс, отображаемый на единственную таблицу, для которой имелся такой отображаемый класс. Поэтому отображение между сущностными классами и таблицами до сих пор строилось по схеме "один к одному".

Внимание! В примере, используемом в этом разделе, создается модель данных, которая содержит классы Square и Rectangle. С точки зрения геометрии квадрат является прямоугольником, но прямоугольник не всегда является квадратом. Однако в модели данных, построенной для этого примера, справедливо обратное отношение. Эта модель классов определяет прямоугольник как наследник квадрата. Поэтому прямоугольник — есть квадрат, но квадрат — не обязательно прямоугольник. Обоснование этого изложено в тексте.

В LINQ to SQL также предлагается альтернатива этому, которая называется *наследованием сущностных классов*. Наследование сущностных классов позволяет отобразить иерархию классов на единственную таблицу базы данных. Для этой единственной таблицы должен существовать базовый сущностный класс с соответствующими атрибутами отображения класса на таблицу базы данных. Этот базовый класс будет содержать все свойства, общие для всех классов в иерархии, унаследованных от базового класса, в то время как производные классы будут содержать лишь свойства, специфичные для этого производного класса, как это принято в любой объектной модели. Ниже приведен пример базового сущностного класса без отображенных производных классов.

Базовый сущностный класс без отображенных сущностных классов

```
[Table]
public class Shape
{
  [Column(IsPrimaryKey = true, IsDbGenerated = true,
    DbType = "Int NOT NULL IDENTITY")]
  public int Id;

  [Column(IsDiscriminator = true, DbType = "NVarChar(2)")]
  public string ShapeCode;

  [Column(DbType = "Int")]
  public int StartingX;

  [Column(DbType = "Int")]
  public int StartingY;
}
```

Как видите, здесь указан атрибут Table, и поскольку свойство Name этого атрибута было задано, базовый сущностный класс отображается на одноименную с классом таблицу, т.е. на Shape. Не беспокойтесь, что таблицы Shape пока еще нет. Позже с помощью метода CreateDatabase объекта DataContext база данных будет создана. К этому моменту никакие производные классы не отображены. Позднее этот базовый сущностный класс будет участвовать при отображении некоторых производных классов.

Идея, положенная в основу наследования сущностных классов, заключается в том, что единственная таблица базы данных `Shape` содержит столбец, значение которого определяет, объект какого именно сущностного класса должен быть сконструирован из записи таблицы, когда он будет извлечен LINQ to SQL. Этот столбец называется *столбцом дискриминатора*, и он указывается с помощью свойства `IsDiscriminator` атрибута `Column`.

Значение столбца дискриминатора известно под названием *значение дискриминатора* или *код дискриминатора*. При отображении базового сущностного класса на таблицу базы данных в дополнение к атрибуту `Table` указываются атрибуты `InheritanceMapping` для отображения кодов дискриминатора на классы-наследники базового сущностного класса. Но на этот раз в предыдущем классе `Shape` никакое наследование не отображается.

Обратите внимание, что имеется несколько общедоступных членов, каждый из которых отображен на столбец базы данных, и указаны типы столбцов базы данных. Указание типов столбцов базы данных в этом случае необходимо, потому что позднее будет вызываться метод `CreateDatabase`, а для этого должны быть известны соответствующие типы. Кроме того, для члена `ShapeCode` задано свойство `IsDiscriminator` атрибута со значением `true`, что создает столбец дискриминатора. Это значит, что столбец базы данных `ShapeCode` будет диктовать тип сущностного класса, используемый для конструирования сущностного объекта из каждой записи.

В этом классе предусмотрены члены для `Id`, `ShapeCode` и начальных координат `X` и `Y` для фигуры на экране. Пока это единственные члены, наличие которых можно предвидеть в каждой фигуре.

Затем можно создать иерархию классов фигур, наследуя новые классы от этого базового. Производные классы должны наследоваться от базового сущностного класса. В производных классах не будет указан атрибут `Table`, но будут предусмотрены атрибуты `Column` для каждого общедоступного члена, который отображается на базу данных. Ниже представлены производные сущностные классы.

Производные сущностные классы

```
public class Square : Shape
{
  [Column(DBType = "Int")]
  public int Width;
}
 public class Rectangle : Square
{
  [Column(DBType = "Int")]
  public int Length;
}
```

Для начала, при рассмотрении этого примера следует забыть о геометрическом определении квадрата и прямоугольника; с геометрической точки зрения квадрат — это прямоугольник, но прямоугольник — не обязательно квадрат. В этом примере с наследованием сущностных классов, поскольку стороны квадрата должны быть равны, необходимо значение только одного измерения — ширины. Так как прямоугольник требует ширину и длину, он наследуется от квадрата и добавляет к нему член для представления длины. В этом смысле с точки зрения наследования классов прямоугольник является квадратом, но квадрат — не есть прямоугольник. Хотя это противоположно геометрическому определению, оно вполне укладывается в модель наследования сущностных классов.

Общедоступные члены этих классов специфичны для каждого класса. Например, поскольку `Square` нужна ширина, он имеет свойство `Width`. Поскольку `Rectangle` на-

следуется от Square, в дополнение к унаследованному свойству Width ему понадобится свойство для представления длины — Length.

Итак, есть производные классы. Единственное, что пока отсутствует — это отображение значений дискриминатора на базовый и производные сущностные классы. После добавления необходимых атрибутов InheritanceMapping базовый класс будет выглядеть так, как показано ниже.

Базовый сущностный класс с отображениями производных классов

```
[Table]
[InheritanceMapping(Code = "G", Type = typeof(Shape), IsDefault = true)]
[InheritanceMapping(Code = "S", Type = typeof(Square))]
[InheritanceMapping(Code = "R", Type = typeof(Rectangle))]
public class Shape
{
  [Column(IsPrimaryKey = true, IsDbGenerated = true,
    DbType = "Int NOT NULL IDENTITY")]
  public int Id;

  [Column(IsDiscriminator = true, DbType = "NVarChar(2)")]
  public string ShapeCode;

  [Column(DbType = "Int")]
  public int StartingX;

  [Column(DbType = "Int")]
  public int StartingY;
}
```

Добавленные отображения связывают различные значения столбца дискриминатора с сущностными классами. Поскольку столбец ShapeCode является столбцом дискриминатора, если запись имеет значение "G" в этом столбце, то из этой записи конструируется объект класса Shape. Если же запись имеет значение "S" в столбце ShapeCode, то из нее конструируется объект класса Square. А если запись имеет значение "R" в этом столбце, то из этой записи конструируется объект Rectangle.

Вдобавок почти всегда должно быть предусмотрено отображение по умолчанию, когда значение столбца дискриминатора не соответствует ни одному из значений, отображенных на сущностный класс. Отображение, принимаемое по умолчанию, указывается свойством IsDefault атрибута. В данном примере отображение на класс Shape принято по умолчанию. Поэтому, если запись имеет, скажем, значение "Q" в столбце ShapeCode, из этой записи по умолчанию будет сконструирован объект Shape, поскольку это значение не соответствует ни одному из заданных кодов дискриминатора.

Все изложенное почти полностью раскрывает концепцию отображения наследования сущностных классов. Теперь давайте рассмотрим полный код класса DataContext.

Полный класс DataContext

```
public partial class TestDB : DataContext
{
  public Table<Shape> Shapes;
   public TestDB(string connection) :
    base(connection)
  {
  }
  public TestDB(System.Data.IDbConnection connection) :
    base(connection)
  {
  }
```

```
    public TestDB(string connection,
              System.Data.Linq.Mapping.MappingSource mappingSource) :
     base(connection, mappingSource)
    {
    }

    public TestDB(System.Data.IDbConnection connection,
              System.Data.Linq.Mapping.MappingSource mappingSource) :
     base(connection, mappingSource)
    {
    }
  }

  [Table]
  [InheritanceMapping(Code = "G", Type = typeof(Shape), IsDefault = true)]
  [InheritanceMapping(Code = "S", Type = typeof(Square))]
  [InheritanceMapping(Code = "R", Type = typeof(Rectangle))]

  public class Shape
  {
    [Column(IsPrimaryKey = true, IsDbGenerated = true,
      DbType = "Int NOT NULL IDENTITY")]
    public int Id;

    [Column(IsDiscriminator = true, DbType = "NVarChar(2)")]
    public string ShapeCode;

    [Column(DbType = "Int")]
    public int StartingX;

    [Column(DbType = "Int")]
    public int StartingY;
  }

  public class Square : Shape
  {
    [Column(DbType = "Int")]
    public int Width;
  }

  public class Rectangle : Square
  {
    [Column(DbType = "Int")]
    public int Length;
  }
```

Помимо помещения ранее упомянутых классов в класс `[Your]DataContext` по имени `TestDB` и добавления к нему некоторых конструкторов здесь нет ничего нового. В листинге 18.3 вызывается код для создания базы данных.

Листинг 18.3. Код создания базы данных для примера наследования сущностных классов

```
TestDB db = new TestDB(@"Data Source=.\SQLEXPRESS;Initial Catalog=TestDB");
db.CreateDatabase();
```

Этот код не дает никакого экранного вывода, но если заглянуть на сервер базы данных, можно увидеть, что в нем появилась база данных по имени `TestDB` с единственной таблицей `Shapes`. Проверьте таблицу `Shape`, чтобы удостовериться, что в ней нет записей. Теперь, имея таблицу, давайте занесем в нее некоторые данные, используя код LINQ to SQL из листинга 18.4.

Листинг 18.4. Код создания некоторых данных в базе для примера наследования сущностных классов

```
TestDB db = new TestDB(@"Data Source=.\SQLEXPRESS;Initial Catalog=TestDB");

db.Shapes.InsertOnSubmit(new Square { Width = 4 });
db.Shapes.InsertOnSubmit(new Rectangle { Width = 3, Length = 6 });
db.Shapes.InsertOnSubmit(new Rectangle { Width = 11, Length = 5 });
db.Shapes.InsertOnSubmit(new Square { Width = 6 });
db.Shapes.InsertOnSubmit(new Rectangle { Width = 4, Length = 7 });
db.Shapes.InsertOnSubmit(new Square { Width = 9 });

db.SubmitChanges();
```

В этом коде также нет ничего нового. Создается объект `DataContext`, а затем объекты сущностных классов, которые вставляются в таблицу `Shapes`. Далее вызывается метод `SubmitChanges`, чтобы сохранить их в базе данных. После запуска этого кода в таблице `Shapes` базы данных `TestDB` должны появиться записи, представленные в табл. 18.1.

Таблица 18.1. Результаты выполнения кода предыдущего примера

Id	ShapeCode	StartingX	StartingY	Length	Width
1	S	0	0	NULL	4
2	R	0	0	6	3
3	R	0	0	5	11
4	S	0	0	NULL	6
5	R	0	0	7	4
6	S	0	0	NULL	9

Поскольку `Id` — столбец идентичности, при многократном запуске этого кода его значения будут изменяться.

Теперь можно выполнить несколько запросов к этой таблице. Сначала в листинге 18.5 запрашиваются квадраты, к которым также будут относиться и прямоугольники, поскольку прямоугольники наследуются от квадратов. Затем запрашиваются только прямоугольники.

Листинг 18.5. Код, выполняющий запросы к базе данных для примера наследования сущностных классов

```
TestDB db = new TestDB(@"Data Source=.\SQLEXPRESS;Initial Catalog=TestDB");

// Сначала получить все квадраты, включая прямоугольники.
IQueryable<Shape> squares = from s in db.Shapes
                            where s is Square
                            select s;
Console.WriteLine("Существуют следующие квадраты.");
foreach (Shape s in squares)
{
  Console.WriteLine("{0} : {1}", s.Id, s.ToString());
}
// Теперь получить только прямоугольники.
IQueryable<Shape> rectangles = from r in db.Shapes
                           where r is Rectangle
                           select r;
Console.WriteLine("{0}Существуют следующие прямоугольники.",
   System.Environment.NewLine);
```

```
foreach (Shape r in rectangles)
{
  Console.WriteLine("{0} : {1}", r.Id, r.ToString());
}
```

В листинге 18.5, по сути, один и тот же запрос выполняется дважды, но только в первом случае запрашиваю лишь записи, из которых получаются экземпляры квадратов; согласно схеме наследования, к ним относятся и прямоугольники. Во втором случае запрашиваются записи, которые превращаются в экземпляры прямоугольников, исключая квадраты. Результаты запуска кода показаны ниже:

```
Существуют следующие квадраты.
1 : LINQChapter18.Square
2 : LINQChapter18.Rectangle
3 : LINQChapter18.Rectangle
4 : LINQChapter18.Square
5 : LINQChapter18.Rectangle
6 : LINQChapter18.Square

Существуют следующие прямоугольники.
2 : LINQChapter18.Rectangle
3 : LINQChapter18.Rectangle
5 : LINQChapter18.Rectangle
```

Наследование сущностных классов может оказаться удобным приемом для конструирования иерархии сущностей из базы данных.

Транзакции

Ранее уже говорилось о том, что при отсутствии активной транзакции во время вызова метода `SubmitChanges` будет создана новая транзакция. В результате все попытки модификаций базы данных, произведенные в течение единственного вызова `SubmitChanges`, будут помещены в контекст единой транзакции. Это очень удобно, но что если нужна транзакция, распространяющаяся за пределы единственного вызова `SubmitChanges`?

Давайте рассмотрим пример, демонстрирующий, как можно поместить обновления, сделанные несколькими вызовами метода `SubmitChanges`, в одну транзакцию. Более того, вызовы метода `SubmitChanges` будут обновлять разные таблицы. В коде из листинга 18.6 вносятся изменения в записи базы данных Northwind и базы данных TestDB, созданной в разделе “Наследование сущностных классов”. Обычно каждый вызов метода `SubmitChanges` на каждом из этих объектов `DataContext` должен быть помещен внутрь собственной индивидуальной транзакции. В приведенном примере оба вызова `SubmitChanges` должны быть помещены в одну и ту же транзакцию.

Код в листинге 18.6 сопровождается дополнительными пояснениями.

На заметку! Для следующего примера в проект должна быть добавлена ссылка на сборку `System.Transactions.dll`.

Листинг 18.6. Включение в объемлющую транзакцию

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
TestDB testDb = new TestDB(@"Data Source=.\SQLEXPRESS;Initial Catalog=TestDB");

Customer cust = db.Customers.Where(c => c.CustomerID == "LONEP").SingleOrDefault();
cust.ContactName = "Barbara Penczek";
Rectangle rect = (Rectangle)testDb.Shapes.Where(s => s.Id == 3).SingleOrDefault();
rect.Width = 15;
```

В приведенном коде для каждой базы данных создается объект DataContext. Затем из каждой базы запрашивается по сущностному объекту, в который вносятся изменения.

```
try
{
  using (System.Transactions.TransactionScope scope =
    new System.Transactions.TransactionScope())
  {
    db.SubmitChanges();
    testDb.SubmitChanges();
    throw (new Exception("Just to rollback the transaction."));
    // Здесь компилятор выдаст предупреждение,
    // поскольку следующая строка кода недостижима.
    scope.Complete();
  }
}
catch (Exception ex)
{
  Console.WriteLine(ex.Message);
}
```

На заметку! Имейте в виду, что поскольку здесь после генерации исключения присутствует код, компилятор выдаст предупреждение, т.к. вызов метода scope.Complete недостижим для выполнения.

В приведенном коде создается экземпляр объекта TransactionScope, так что появляется объемлющая транзакция для включения объектов DataContext с их вызовами метода SubmitChanges. После вызова метода SubmitChanges на каждом DataContext намеренно генерируется исключение, так что метод scope.Complete не вызывается и происходит откат транзакции.

Если не поместить вызовы метода SubmitChanges в контекст объекта TransactionScope, то каждый вызов SubmitChanges получит собственную транзакцию, и его изменения будут зафиксированы немедленно по окончании успешного вызова SubmitChanges.

Как только в предыдущем коде генерируется исключение, транзакция выходит из контекста, и поскольку метод Complete не был вызван, транзакция откатывается. В этой точке все изменения, внесенные в базе данных, отменяются.

```
db.Refresh(System.Data.Linq.RefreshMode.OverwriteCurrentValues, cust);
Console.WriteLine("Contact Name = {0}", cust.ContactName);

testDb.Refresh(System.Data.Linq.RefreshMode.OverwriteCurrentValues, rect);
Console.WriteLine("Rectangle Width = {0}", rect.Width);
```

Важно помнить, что несмотря на то, что изменения не были успешно сохранены в базе данных, сущностные объекты продолжают содержать модифицированные данные. Помните, что даже если метод SubmitChanges не был завершен успешно, изменения поддерживаются в сущностных объектах, чтобы можно было разрешить конфликты параллельного доступа и снова вызвать метод SubmitChanges. В этом случае вызовы метода SubmitChanges завершены успешно. Кроме того, как говорилось в разделе “Несоответствие кэша результирующего набора” главы 16, новый запрос объектов из базы данных не даст в результате текущих значений из базы данных. Запрос к базе лишь определит, какие сущности должны быть включены в результирующий набор. Если эти сущности уже кэшированы в DataContext, то будут возвращены именно эти кэшированные сущностные объекты. Поэтому, чтобы действительно знать, какие зна-

чения ранее опрошенных сущностных объектов содержатся в базе данных, сущностные объекты должны сначала быть обновлены вызовом метода `Refresh`.

Таким образом, каждый из двух извлеченных сущностных объектов сначала обновляется, а затем на консоль выводится измененное свойство для доказательства, что изменения действительно были отменены. Взглянем на результат:

```
Just to rollback the transaction.
Contact Name = Fran Wilson
Rectangle Width = 11
```

Как видите, изменения значений в базе данных были отменены.

Совет. Если при работе с любым их примеров, в которых используется объект `TransactionScope`, возникает исключение типа `MSDTC on server '[сервер]\SQLEXPRESS' is unavailable` (Служба MSDTC на сервере '[сервер]\SQLEXPRESS' не доступна), удостоверьтесь, что запущена служба Distributed Transaction Coordinator (Распределенный координатор транзакций).

Резюме

В этой главе было продемонстрировано, как выполняются запросы к представлениям базы данных. Помните, что они на самом деле отображаются как таблицы, доступные только для чтения, так что должно быть понятно, как их опрашивать.

Кроме того, в главе была раскрыта тема наследования сущностных классов. Это удобный прием, позволяющий создавать экземпляры разных, но связанных отношением наследования, классов из записей одной и той же таблицы. И, наконец, были приведены дополнительные сведения о транзакциях и показано, как поместить обновления базы данных с помощью LINQ to SQL в одну объемлющую транзакцию.

ЧАСТЬ VI

LINQ to Entities

В этой части

Глава 19. Введение в LINQ to Entities
Глава 20. Операции LINQ to Entities
Глава 21. Классы LINQ to Entities

ГЛАВА 19

Введение в LINQ to Entities

Листинг 19.1. Простой пример обновления контактного имени заказчика в базе данных Northwind

```
// Создать ObjectContext.
NorthwindEntities context = new NorthwindEntities();

// Извлечь заказчика LAZYK
Customer cust = (from c in context.Customers
                 where c.CustomerID == "LAZYK"
                 select c).Single<Customer>();

// Изменить контактное имя.
cust.ContactName = "Ned Plimpton";

// Сохранить изменения.
try {
   context.SaveChanges();
} catch (OptimisticConcurrencyException) {
   context.Refresh(RefreshMode.ClientWins,
      context.Customers);
   context.SaveChanges();
}
```

На заметку! Этот пример требует генерации сущностной модели данных, о которой речь пойдет далее в этой главе.

В листинге 19.1 с помощью LINQ to Entities был произведен запрос записи, значение поля `CustomerID` которой равно `"LAZYK"`, из таблицы `Customers` базы данных Northwind и возврат объекта `Customer`, представляющего эту запись. Затем было обновлено свойство `ContactName` объекта `Customer` с сохранением изменения в базе данных, для чего был вызван метод `SaveChanges`. Нажмите <Ctrl+F5> для запуска программы из листинга 19.1. Консольного вывода нет, но если вы заглянете в базу данных, то увидите там, что поле `ContactName` для заказчика `"LAZYK"` теперь содержит значение `"Ned Plimpton"`.

На заметку! Код в этом примере вносит изменение в базу данных, не восстанавливая старого значения. Исходным значением `ContactName` заказчика `LAZYK` было `"John Steel"`. Его понадобится восстановить, чтобы последующие примеры работали правильно. Это можно сделать вручную или же соответствующим образом изменить исходный код и запустить пример снова.

В этой книге используется расширенная версия базы данных Northwind. Подробности ищите в разделе “Получение соответствующей версии базы данных Northwind” далее в главе.

Введение

В главе 12 объяснялось, что LINQ to SQL — это система объектно-реляционного отображения начального уровня. LINQ to Entities — это часть платформы ADO.NET Entity Framework, предоставляющая более высокую гибкость и больше средств, чем LINQ to SQL, но следующая за LINQ to SQL в отношении адаптации, из-за повышенной сложности и ранних выпусков, которым пока недостает ключевых средств.

Код в листинге 19.1 делает то же самое, что и код в листинге 12.1, который использовался для представления LINQ to SQL. Не поленитесь сравнить листинги 19.1 и 12.1; вы увидите, что они довольно похожи.

API-интерфейс Entity Framework спроектирован для работы с любыми базами данных, поддерживающими ADO (а не только с SQL Server), и даже включает собственный диалект независимого от поставщика языка SQL, который можно применять в качестве альтернативы LINQ. Фактически Entity Framework обладает настолько широким набором средств, что для их описания понадобилась бы отдельная книга. Здесь будет показано, как запустить и использовать только важнейшие части Entity Framework, относящиеся к LINQ to Entities.

Имеется некоторая путаница с терминами. В конце концов, разве мы не потратили последние несколько глав на обсуждение сущностных классов как части LINQ to SQL? Да, действительно. API-интерфейсы LINQ to SQL и Entity Framework делают нечто схожее, а потому не удивительно, что в них используется общая терминология.

Подобно LINQ to SQL, LINQ to Entities позволяет работать с объектами, которые представляют информацию из базы данных — выполнять LINQ-запросы, изменять значения, добавлять и удалять объекты. И так же, как LINQ to SQL, первый шаг в направлении использования этих средств предусматривает генерацию классов, отображающих содержимое базы данных на объекты — то, что делается при создании сущностной модели данных (entity data model — EDM). Модель EDM состоит из набора объектов и свойств, которые используются для взаимодействия с данными.

В листинге 19.1 сначала создается экземпляр класса `NorthwindEntities`. Этот класс унаследован от класса `System.Data.Objects.ObjectContext`, который подробно рассматривается в последующих главах. Это точка входа в EDM, похожая на класс `DataContext` для LINQ to SQL. Класс `NorthwindEntities` устанавливает соединение с базой данных при создании его нового экземпляра и берет на себя ответственность за сохранение изменений при вызове метода `SaveChanges`.

Затем из базы данных извлекается одиночный заказчик, который помещается в объект `Customer`. Объект `Customer` — это экземпляр класса `Customer`, являющегося частью сущностной модели данных. Далее в этой главе будет показано, как генерировать EDM для базы данных Northwind. После извлечения `Customer` обновляется одно из его свойств и вызывается метод `SaveChanges` для сохранения изменений в базе данных. Вызов метода `SaveChanges` помещен в блок `try/catch`, чтобы можно было разрешить любые потенциальные конфликты, связанные с параллелизмом. Более подробно о разрешении этих конфликтов будет рассказываться в главе 20.

Прежде чем получится запустить этот пример, равно как и любой другой пример в этой главе, понадобится создать сущностную модель данных для базы Northwind. Читайте подробности в разделе “Предварительные условия для запуска примеров” далее в главе.

Как это делалось с LINQ to SQL, начнем с обзора ключевых частей LINQ to Entities. Кое-что из того, что будет рассказано о LINQ to Entities, излагается в форме сравнения с LINQ to SQL, так что если вы не читали этих глав, сделайте это перед тем, как двигаться дальше. В первом примере, приведенном в начале этой главы, использовался класс-наследник `ObjectContext` по имени `NorthwindEntities`, сущностный класс `Customer`, средства обнаружения и разрешения конфликтов, а также обновление базы данных через метод `SaveChanges`. Для начала рассмотрим некоторые основы этих компонентов, чтобы обеспечить базовое понимание основ LINQ to Entities и ADO.NET Entity Framework в целом.

ObjectContext

Класс `ObjectContext` — это ключ для доступа к сущностной модели данных и эквивалент класса `DataContext` из LINQ to SQL. Класс `ObjectContext` отвечает за создание и управление соединением с базой данных, отслеживает изменения и управляет постоянством. Подробности будут изложены позднее, а пока достаточно знать, что именно класс `ObjectContext` соединяет с базой данных, когда создается новый экземпляр `NorthwindEntities`, и этот же класс отслеживает изменения, которые вносятся в объект `Customer`, а также транслирует их в оператор SQL, сохраняющий изменения по вызову метода `SaveChanges`.

Обычно используется класс, унаследованный от `ObjectContext`, который создается при генерации EDM из базы данных. Далее в этой главе будет показано, как это делается для базы данных Northwind. Имя класса выбирается на основе имени базы данных — в форме `[База_данных]Entities`. В листинге 19.1 для базы данных Northwind предусмотрен класс `NorthwindEntities`.

Производный класс `[База_данных]Entities` для каждой таблицы базы, выбранной при создании EDM, будет иметь свойство `ObjectSet<T>`, представляющее таблицу, причем `T` здесь — это тип сущностного класса, созданного для представления записи в таблице. Например, класс `NorthwindEntities`, использованный в листинге 19.1, имеет общедоступное свойство `Customers` типа `OrderSet<Customer>`. Оно применялось для выполнения запроса LINQ к множеству заказчиков.

Сущностные классы

Сущностные классы в Entity Framework имеют много общего с классами, описанными в главе о LINQ to SQL. Это типы .NET, которые представляют собой отображение на реляционную структуру базы данных. Платформа Entity Framework позволяет выполнять очень сложные отображения между сущностными классами и реляционными данными, которые могут охватывать разные базы данных и быть абстрагированы различными интересными способами. Здесь все эти тонкости не рассматриваются, потому что внимание сосредоточено на аспектах LINQ, но если нужны развитые средства ORM, то определенно следует обратиться к Entity Framework.

Сущностные классы в примерах обнаруживаются по наличию классов и объектов, имеющих имена таблиц базы данных в форме существительного единственного числа. Например, в листинге 19.1 используется класс по имени `Customer`. Поскольку `Customer` — форма единственного числа от `Customers`, и в базе данных Northwind есть таблица `Customers`, это намек на то, что `Customer` является сущностным классом, который представляет записи из таблицы `Customer` базы данных Northwind.

Мастер построения сущностной модели, с которым вы вскоре ознакомитесь, имеет опцию поддержки имен таблиц во множественном числе при создании сущностных классов, поэтому, когда он находит таблицу базы под названием `Customers`, то создает сущностный класс по имени `Customer` для представления элемента таблицы. Тот же

подход обеспечивается опцией /pluralize утилиты SQLMetal, которая была показана в главе 12, и это обеспечивает значительное повышение читабельности кода.

Ассоциации

Ассоциация — термин, применяемый для обозначения отношения первичного ключа к внешнему ключу между двумя сущностными классами. При отношении "один ко многим" результатом ассоциации является то, что родительский класс, включающий в себя первичный ключ, содержит коллекцию дочерних классов, имеющих внешний ключ.

Коллекция сохраняется в EntityCollection<T>, где T — тип дочернего сущностного класса. В отношении "многие ко многим" каждый сущностный класс поддерживает EntityCollection<T>, где T — тип противоположной сущности в отношении.

Коллекции сущностей доступны через общедоступные свойства по имени внешнего ключа. Поэтому, например, чтобы добраться к заказам, ассоциированным с заказчиком в базе данных Northwind, следует обратиться к свойству Customer.Orders, которое вернет EntityCollection<Order>.

Преимущество ассоциаций между сущностными типами заключается в том, что они позволяют прозрачно выполнять навигацию по данным, не принимая во внимание того факта, что они могут быть распределены среди множества таблиц или даже множества баз данных.

Предварительные условия для запуска примеров

В этой и последующей главах, посвященных LINQ to Entities, используется одна и та же расширенная база данных Northwind, которую применялась также и в главах по LINQ to SQL. Для базы Northwind понадобится сгенерировать сущностную модель данных.

Получение соответствующей версии базы данных Northwind

Для согласованности будет использоваться та же самая расширенная версия базы данных примеров Microsoft под названием Northwind, которая применялась в главах, посвященных LINQ to SQL. Она входит в состав загружаемых примеров для книги.

Генерация сущностной модели данных Northwind

Сгенерировать модель EDM можно либо с помощью инструмента командной строки EdmGen, либо в среде Visual Studio 2010. Далее будет показано, как это делать с использованием графического мастера Visual Studio. Сначала щелкните правой кнопкой мыши на проекте, выберите в контекстном меню пункт Add⇨New Irtem (Добавить⇨Новый элемент) и затем укажите в списке вариант ADO.NET Entity Data Model (Сущностная модель данных ADO.NET). Измените имя модели данных. Поскольку используется база данных Northwind, в качестве имени имеет смысл указать NorthwindDataModel.edmx. Щелкните на кнопке Add (Добавить), после чего запустится мастер создания сущностной модели данных (Entity Data Model Wizard), окно которого показано на рис. 19.1.

Сущностную модель данных можно создать с нуля или же сгенерировать ее на основе имеющейся базы данных. Здесь необходимо сгенерировать EDM из базы данных Northwind, поэтому выберите значок Generate from database (Генерировать из базы данных) и щелкните на кнопке Next (Далее) для перехода на экран подключения к данным, который показан на рис. 19.2.

Этот экран используется для выбора базы данных, на основе которой будет сгенерирована EDM-модель. На рис. 19.2 выбрана существующая база данных Northwind, которая ранее была подключена к SQL Server 2008.

Рис. 19.1. Первый экран мастера Entity Data Model Wizard

Рис. 19.2. Экран подключения к базе данных мастера Entity Data Model Wizard

Содержимое окна может отличаться в зависимости от местоположения базы данных. По крайней мере, имя сервера будет отличаться от `shuttle`. Выберите нужное соединение и щелкните на кнопке Next для перехода к следующему экрану мастера, показанному на рис. 19.3.

Рис. 19.3. Экран объектов сущностной модели базы данных мастера Entity Data Model Wizard

На этом экране мастера выбираются таблицы, представления и хранимые процедуры из базы данных, которые будут включены в EDM-модель. Можно также выбрать множественную или единственную форму имен объектов (например, чтобы объекты, сгенерированные из таблицы `Customers` назывались `Customer`) и добавить внешние ключи. Для текущих целей понадобится включить в модель все содержимое базы данных, поэтому отметьте все флажки на экране, показанном на рис. 19.3. Щелкните на кнопке Finish (Готово) для закрытия окна мастера и генерации модели. Когда процесс завершится, среда Visual Studio должна выглядеть примерно так, как показано на рис. 19.4.

В основной части окна показана создаваемая сущностная модель. Здесь можно видеть свойства каждого сущностного класса и отношения между ними. Кроме того, также выводится множество предупреждений относительно модели данных; они вызваны тем, что с расширенной базой данных Northwind связан ряд недостатков. Пока эти ошибки игнорируются, но в реальном проекте они должны быть тщательно изучены. Наконец, обратите внимание, что мастер создания EDM добавил в проект ряд новых ссылок; они требуются для API-интерфейса Entity Framework и не должны удаляться. На этом все — сущностная модель данных для расширенной базы данных Northwind сгенерирована. В следующем разделе будет приведен очень краткий обзор ее использования.

Рис. 19.4. EDM-модель Northwind в Visual Studio

Использование API-интерфейса LINQ to Entities

Сборки, которые понадобятся для использования LINQ to Entities, добавляются к проекту автоматически при генерации сущностной модели данных. В отличие от LINQ to SQL, импортировать пространство имен для использования сущностных классов не понадобится, т.к. мастер Entity Data Model генерирует сущностные модели данных в пространстве имен по умолчанию проекта.

IQueryable<T>

Во многих примерах настоящей и последующих глав, посвященных LINQ to Entities, работа будет проводиться с последовательностями типа `IQueryable<T>`, где `T` — тип сущностного класса. Этот тип последовательностей обычно возвращается запросами LINQ to Entities, как и в LINQ to SQL. Обычно работа с ними выглядит, как с последовательностью `IEnumerable<T>`, и это не случайно. Интерфейс `IQueryable<T>` реализует интерфейс `IEnumerable<T>`. Ниже приведено определение `IQueryable<T>`:

```
interface IQueryable<T> : IEnumerable<T>, IQueryable
```

Благодаря этому наследованию последовательность `IQueryable<T>` можно трактовать как последовательность `IQueryable<T>`.

Некоторые общие методы

В процессе демонстрации средств LINQ to Entities нужна возможность запрашивать или модифицировать базу данных, внешнюю по отношению к Entity Framework. Чтобы выделить код LINQ to Entities и исключить как можно больше тривиальных деталей (в то же время обеспечивая полезные примеры), были разработаны некоторые общие методы. Добавьте их к своему исходному коду при тестировании примеров.

GetStringFromDo()

Общий метод, который весьма пригодится — это метод для получения простой строки из базы данных с помощью стандартного механизма ADO.NET (листинг 19.2). Это позволит посмотреть, что на самом деле есть в базе данных, и сравнить с тем, что отображает LINQ to Entities.

Листинг 19.2. GetStringFromDb: метод для извлечения строки посредством ADO.NET

```
static private string GetStringFromDb(string sqlQuery) {

   string connection =
     @"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind;Integrated Security=SSPI;";

   System.Data.SqlClient.SqlConnection sqlConn =
     new System.Data.SqlClient.SqlConnection(connection);

   if (sqlConn.State != ConnectionState.Open) {
      sqlConn.Open();
   }

   System.Data.SqlClient.SqlCommand sqlCommand =
     new System.Data.SqlClient.SqlCommand(sqlQuery, sqlConn);

   System.Data.SqlClient.SqlDataReader sqlDataReader = sqlCommand.ExecuteReader();
   string result = null;

   try {
      if (!sqlDataReader.Read()) {
        throw (new Exception(
          String.Format("Unexpected exception executing query [{0}].", sqlQuery)));
      } else {
        if (!sqlDataReader.IsDBNull(0)) {
          result = sqlDataReader.GetString(0);
        }
      }
   } finally {
      // Всегда вызывать Close, когда чтение завершено.
      sqlDataReader.Close();
      sqlConn.Close();
   }

   return (result);
}
```

При вызове методу `GetStringFromDb` передается строка запроса SQL. Метод создает и открывает новое соединение с базой данных.

Затем создается объект `SqlCommand`, конструктору которого передается запрос и соединение. Затем `SqlDataReader` получается вызовом метода `ExecuteReader` на `SqlCommand`. Объект `SqlDataReader` читает данные с помощью своего метода `Read`, и если данные были прочитаны, а значение возвращенного первого столбца не `null`, то это значение извлекается методом `GetString`. Наконец, `SqlDataReader` и `SqlConnection` закрываются, и значение первого столбца возвращается вызывающему методу.

ExecuteStatementInDb()

Иногда для изменения состояния базы данных вне Entity Framework возникает необходимость в выполнении оператора SQL, отличного от запроса вроде `insert`, `update` и `delete` в ADO.NET. Для этой цели был создан метод `ExecuteStatementInDb`, приведенный в листинге 19.3.

Листинг 19.3. `ExecuteStatementInDb`: метод для выполнения вставок, обновлений и удалений в ADO.NET

```
static private void ExecuteStatementInDb(string cmd) {
   string connection =
     @"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind;Integrated Security=SSPI;";

   System.Data.SqlClient.SqlConnection sqlConn =
     new System.Data.SqlClient.SqlConnection(connection);
   if (sqlConn.State != ConnectionState.Open) {
      sqlConn.Open();
   }

   System.Data.SqlClient.SqlCommand sqlComm =
     new System.Data.SqlClient.SqlCommand(cmd);

   sqlComm.Connection = sqlConn;
   try {
     Console.WriteLine("Выполнение оператора SQL в базе данных с помощью ADO.NET...");
       sqlComm.ExecuteNonQuery();
       Console.WriteLine("База данных обновлена.");
   } finally {
       // Закрыть соединение.
       sqlComm.Connection.Close();
   }
}
```

При вызове методу `ExecuteStatementInDb` передается аргумент `string`, содержащий команду SQL. Создается экземпляр `SqlConnection`, за которым следует `SqlCommand`. Объект `SqlConnection` затем открывается и команда SQL выполняется посредством вызова метода `ExecuteNonQuery` объекта `SqlCommand`. Наконец, `SqlCommand` закрывается.

Резюме

В этой главе было дано введение в платформу Entity Framework и API-интерфейс LINQ to Entities. Также были рассмотрены некоторые его базовые элементы, такие как объекты `ObjectContext`, сущностные классы и ассоциации.

В главе было продемонстрировано, как генерируется сущностная модель данных для расширенной базы данных Northwind, содержащая сущностные классы, которые используются для работы с данными Northwind. Эти сущностные классы применяются во всех примерах LINQ to Entities. Также были представлены методы общего назначения, на которые полагаются некоторые примеры в последующих главах, посвященных LINQ to Entities. Следующая глава посвящена демонстрации использования LINQ to Entities и Entity Framework для выполнения общих операций с базами данных.

ГЛАВА 20

Операции LINQ to Entities

В этой главе будет показано, как выполнять типичные операции с базой данных в LINQ to Entities, в частности:

- вставки;
- запросы;
- обновления;
- удаления.

Для демонстрации этих операций понадобится класс `ObjectContext` и средства сущностных классов. Все это детально объясняется в главе 21, а пока просто следует помнить, что класс `ObjectContext` поддерживает и управляет соединением с базой данных, а сущностные классы представляют данные в таблицах базы и отношения между таблицами.

Если вы читали главы LINQ to SQL, то уже должны понимать все основные концепции, которые увидите в этой главе. Если же вы не читали часть, посвященную LINQ to SQL, то стоит сейчас восполнить этот пробел. Для удобства сравнения, где это возможно, используются те же самые примеры с LINQ to Entities, что и при рассмотрении LINQ to SQL.

Предварительные условия для запуска примеров

Для запуска примеров этой главы понадобится расширенная версия базы данных Northwind и сгенерированная для нее сущностная модель данных. Прочтите раздел “Предварительные условия для запуска примеров” главы 19.

Некоторые общие методы

Вдобавок для запуска примеров этой главы потребуются общие методы, которые будут использоваться в примерах. Прочтите и выполните то, что указано в разделе “Некоторые общие методы” главы 19.

Стандартные операции базы данных

В этом разделе будет показано, как выполнять стандартные операции базы данных. Эти примеры призваны продемонстрировать только основные концепции, поэтому они не содержат проверки ошибок или обработки исключений.

Например, поскольку многие стандартные операции вносят изменения в базу данных, в примерах, в которых они применяются, должны обнаруживаться и разрешаться конфликты параллелизма. Но для простоты этот прием в примерах не используется вплоть до раздела “Управление параллельным доступом” в конце главы.

Вставки

Для выполнения вставки необходимо предпринять четыре шага. Первый из них — создание ObjectContext. Это первый шаг для всех операций LINQ to Entities, и он встречается во всех примерах. Получив объектный контекст, можно создать новый экземпляр сущностного типа, например, Customer, и наполнить данными его поля. Заполненный сущностный тип затем добавляется к ObjectSet<T>. Последний шаг — сохранение новых данных посредством метода SaveChanges. В листинге 20.1 продемонстрированы эти четыре шага.

Листинг 20.1. Четыре шага для вставки записи

```
// Шаг 1. Создание ObjectContext.
NorthwindEntities context = new NorthwindEntities();

// Шаг 2. Создание нового объекта Customer.
Customer cust = new Customer() {
   CustomerID = "LAWN",
   CompanyName = "Lawn Wranglers",
   ContactName = "Mr. Abe Henry",
   ContactTitle = "Owner",
   Address = "1017 Maple Leaf Way",
   City = "Ft. Worth",
   Region = "TX",
   PostalCode = "76104",
   Country = "USA",
   Phone = "(800) MOW-LAWN",
   Fax = "(800) MOW-LAWO"
};

// Шаг 3. Добавление в ObjectSet<Customer>.
context.Customers.AddObject(cust);

// Шаг 4. Сохранение изменений.
context.SaveChanges();

// Запрос записи.
Customer customer = context.Customers.Where(c => c.CustomerID == "LAWN").First();
Console.WriteLine("{0} - {1}", customer.CompanyName, customer.ContactName);

// Возврат базы данных в исходное состояние, чтобы можно было запускать пример многократно
Console.WriteLine("Удаление добавленного заказчика LAWN.");
context.DeleteObject(customer);
context.SaveChanges();
```

Как видите, в предыдущем коде шаги пронумерованы. На первом шаге строится ObjectContext за счет создания нового экземпляра производного класса NorthwindEntities. На втором шаге создается новый экземпляр сущностного типа Customer и применяется инициализация объекта для заполнения полей. На третьем шаге в коллекцию Customers добавляется новый экземпляр Customer вызовом метода AddObject свойства Customers объекта ObjectContext. Вспомните из главы 19, что это свойство в базе данных имеет тип ObjectSet<Customer>. На последнем шаге вызывается метод SaveChanges для сохранения новой записи в базе данных.

В оставшейся части кода данные запрашиваются, чтобы убедиться, что новая запись была создана, после чего запись удаляется, чтобы этот же пример можно было запустить повторно без каких-либо проблем. Где это возможно, в конце примеров база данных сбрасывается в исходное состояние.

Создание частично заполненных сущностных типов

В листинге 20.1 явно создается новый сущностный тип, но при этом используется другой подход. Сущностные типы Entity Framework включают статический метод по имени `Create[T]`, где `[T]` — имя типа. Например, сущностный тип `Customer` будет включать метод под названием `CreateCustomer`, а тип `Order` — метод по имени `CreateOrder`. Эти методы имеют параметры для каждого поля сущностного типа, которое не допускает значения `null`. На рис. 20.1 показана таблица `Customers` базы Northwind в среде SQL Server Management Studio. Здесь видно, что все столбцы могут содержать значения `null`, за исключением `CustomerID` и `CompanyName`.

Рис. 20.1. Таблица `Customers` базы данных Northwind

Таким образом, статический метод `Customer.CreateCustomer` имеет следующую сигнатуру, которая требует параметров для поля с обязательным заполнением:

```
public static Customer CreateCustomer(
   String customerID,
   String companyName);
```

Преимущество использования метода `Create[T]` состоит в том, что он позволяет избежать исключений при попытке сохранить сущностный тип со значением `null` в не допускающем `null` поле. В листинге 20.2 приведен код создания и добавления записи посредством этого приема.

Листинг 20.2. Создание сущностного типа методом `Create[T]`

```
// Создать ObjectContext.
NorthwindEntities context = new NorthwindEntities();

// Создать новый объект Customer.
Customer cust = Customer.CreateCustomer("LAWN", "Lawn Wranglers");

// Заполнить поля, допускающие null.
cust.ContactName = "Mr. Abe Henry";
cust.ContactTitle = "Owner";
cust.Address = "1017 Maple Leaf Way";
cust.City = "Ft. Worth";
cust.Region = "TX";
cust.PostalCode = "76104";
cust.Country = "USA";
cust.Phone = "(800) MOW-LAWN";
cust.Fax = "(800) MOW-LAWO";

// Добавить нового заказчика в коллекцию Customers типа ObjectSet.
context.Customers.AddObject(cust);

// Сохранить изменения.
context.SaveChanges();

// Запросить запись.
Customer customer = context.Customers.Where(c => c.CustomerID == "LAWN").First();
Console.WriteLine("{0} - {1}", customer.CompanyName, customer.ContactName);

// Сбросить базу данных в исходное состояние, чтобы можно было запускать пример многократно
Console.WriteLine("Удаление добавленного заказчика LAWN.");
context.DeleteObject(customer);
context.SaveChanges();
```

Как видите, здесь был создан новый экземпляр `Customer` с использованием метода `CreateCustomer`, которому переданы значения двух полей, не допускающие `null`. Затем с помощью общедоступных свойств типа `Customer` устанавливаются другие необходимые значения.

Вставка присоединенных сущностных объектов

Класс `Object` обнаруживает присоединения между сущностными объектами и гарантирует их автоматическое сохранение в базе данных с помощью метода `SaveChanges`. Помните, что сущностные объекты присоединяются тогда, когда между ними существует отношение внешнего ключа. В листинге 20.3 показано, как это работает.

Листинг 20.3. Вставка присоединенного сущностного объекта

```
// Создать ObjectContext.
NorthwindEntities context = new NorthwindEntities();
Customer cust = new Customer {
   CustomerID = "LAWN",
   CompanyName = "Lawn Wranglers",
   ContactName = "Mr. Abe Henry",
   ContactTitle = "Owner",
   Address = "1017 Maple Leaf Way",
   City = "Ft. Worth",
   Region = "TX",
   PostalCode = "76104",
   Country = "USA",
   Phone = "(800) MOW-LAWN",
```

```
    Fax = "(800) MOW-LAWO",
    Orders = {
      new Order {
        CustomerID = "LAWN",
        EmployeeID = 4,
        OrderDate = DateTime.Now,
        RequiredDate = DateTime.Now.AddDays(7),
        ShipVia = 3,
        Freight = new Decimal(24.66),
        ShipName = "Lawn Wranglers",
        ShipAddress = "1017 Maple Leaf Way",
        ShipCity = "Ft. Worth",
        ShipRegion = "TX",
        ShipPostalCode = "76104",
        ShipCountry = "USA"
      }
    }
};

// Добавить новый Customer.
context.Customers.AddObject(cust);

// Сохранить изменения.
context.SaveChanges();

// Запрос для проверки наличия записи.
Customer customer = context.Customers.Where(c => c.CustomerID == "LAWN").First();
Console.WriteLine("{0} - {1}", customer.CompanyName, customer.ContactName);
foreach (Order order in customer.Orders) {
    Console.WriteLine("{0} - {1}", order.CustomerID, order.OrderDate);
}

// Эта часть кода сбрасывает базу данных в исходное состояние.
context.DeleteObject(cust);
context.SaveChanges();
```

В этом примере создается новый объект `Customer`, а свойство-коллекция `Orders` инициализируется единственным новым заказом. При вызове метода `SaveChanges` объект `ObjectContext` сохраняет и `Customers`, и `Orders`. Явно добавлять `Order` в коллекцию `Orders` типа `ObjectSet` не понадобится.

Создавать присоединенные объекты вместе подобным образом не обязательно. Их можно создавать отдельно и затем ассоциировать друг с другом. В листинге 20.4 показано, как это делается.

Листинг 20.4. Присоединение объектов после их создания

```
// Создать ObjectContext.
NorthwindEntities context = new NorthwindEntities();

// Создать новый Customer.
Customer cust = new Customer {
    CustomerID = "LAWN",
    CompanyName = "Lawn Wranglers",
    ContactName = "Mr. Abe Henry",
    ContactTitle = "Owner",
    Address = "1017 Maple Leaf Way",
    City = "Ft. Worth",
    Region = "TX",
    PostalCode = "76104",
    Country = "USA",
```

```
   Phone = "(800) MOW-LAWN",
   Fax = "(800) MOW-LAWO"
};

// Создать новый Order.
Order ord = new Order {
       CustomerID = "LAWN",
       EmployeeID = 4,
       OrderDate = DateTime.Now,
       RequiredDate = DateTime.Now.AddDays(7),
       ShipVia = 3,
       Freight = new Decimal(24.66),
       ShipName = "Lawn Wranglers",
       ShipAddress = "1017 Maple Leaf Way",
       ShipCity = "Ft. Worth",
       ShipRegion = "TX",
       ShipPostalCode = "76104",
       ShipCountry = "USA"
};

// Присоединить Order к Customer.
cust.Orders.Add(ord);

// Добавить новый Customer.
context.Customers.AddObject(cust);

// Сохранить изменения.
context.SaveChanges();

// Запрос для проверки наличия записи.
Customer customer = context.Customers.Where(c => c.CustomerID == "LAWN").First();
Console.WriteLine("{0} - {1}", customer.CompanyName, customer.ContactName);
foreach (Order order in customer.Orders) {
   Console.WriteLine("{0} - {1}", order.CustomerID, order.OrderDate);
}

// Эта часть кода сбрасывает базу данных в исходное состояние.
context.DeleteObject(cust);
context.SaveChanges();
```

В коде объекты `Customer` и `Order` создавались отдельно и затем были присоединены друг к другу вызовом метода `Orders.Add`, который поместил `Order` в коллекцию `ObjectSet<Order>`, поддерживаемую `Customer`. При вызове метода `SaveChanges` объект `ObjectContext` обнаруживает новый `Order` и сохраняет его в базе данных.

В листингах 20.3 и 20.4 объекты были ассоциированы в отношении “один ко многим” с помощью вызова метода `AddObject` на объекте `Customer` (стороны *один*) и передачей нового объекта `Order` (стороны *многие*). Ассоциацию можно установить и в другом направлении, например, указав в качестве значения для свойства `Order.Customer` новый объект `Customer`. При этом `ObjectContext` все равно обнаружит новые сущностные объекты и сохранит их. Это демонстрируется в листинге 20.5.

Листинг 20.5. Присоединение объектов в другом направлении

```
// Создать ObjectContext.
NorthwindEntities context = new NorthwindEntities();

// Создать новый Customer.
Customer cust = new Customer {
   CustomerID = "LAWN",
   CompanyName = "Lawn Wranglers",
   ContactName = "Mr. Abe Henry",
```

```
    ContactTitle = "Owner",
    Address = "1017 Maple Leaf Way",
    City = "Ft. Worth",
    Region = "TX",
    PostalCode = "76104",
    Country = "USA",
    Phone = "(800) MOW-LAWN",
    Fax = "(800) MOW-LAWO"
};
// Создать новый Order.
Order ord = new Order {
    CustomerID = "LAWN",
    EmployeeID = 4,
    OrderDate = DateTime.Now,
    RequiredDate = DateTime.Now.AddDays(7),
    ShipVia = 3,
    Freight = new Decimal(24.66),
    ShipName = "Lawn Wranglers",
    ShipAddress = "1017 Maple Leaf Way",
    ShipCity = "Ft. Worth",
    ShipRegion = "TX",
    ShipPostalCode = "76104",
    ShipCountry = "USA"
};
// Присоединить Customer к Order.
ord.Customer = cust;

// Добавить новый Order к контексту.
context.Orders.AddObject(ord);

// Сохранить изменения.
context.SaveChanges();

// Запрос для проверки наличия записи.
Customer customer = context.Customers.Where(c => c.CustomerID == "LAWN").First();
Console.WriteLine("{0} - {1}", customer.CompanyName, customer.ContactName);
Console.WriteLine("Заказчик имеет {0} заказов", customer.Orders.Count());

// Эта часть кода сбрасывает базу данных в исходное состояние.
context.DeleteObject(ord);
context.DeleteObject(cust);
context.SaveChanges();
```

Как видите, на этот раз пришлось удалять объекты `Customer` и `Order` по отдельности. В случае установки ассоциации в этом направлении они должны удаляться явно. Скомпилировав и запустив этот код, получится несколько неожиданный результат:

```
Lawn Wranglers - Mr. Abe Henry
Заказчик имеет 0 заказов
Для продолжения нажмите любую клавишу . . .
```

Что произошло? Объект `Customer` найден, но что случилось с `Order`? Подождите объяснений до главы 21. А пока просто знайте, что лучший способ присоединять объекты в отношении "один ко многим" предусматривает добавление их к соответствующему `ObjectSet` на стороне родительского объекта в отношении. Кроме того, если вы *можете* что-то сделать, это еще не значит, что вы *должны* это делать.

Запросы

Выполнение запросов с использованием LINQ to Entities подобно выполнению запросов с помощью LINQ to SQL. Тем не менее, есть некоторые нюансы и отличия.

Базовые запросы

Подобно LINQ to SQL, запросы LINQ to Entities возвращают IQueryable<T>. Результат запроса LINQ to Entities можно использовать точно так, как запрос LINQ to SQL. В листинге 20.6 приведен пример.

Листинг 20.6. Получение результата IQueryable<T> от LINQ to Entities

```
// Создать ObjectContext.
NorthwindEntities context = new NorthwindEntities();
IQueryable<Customer> custs = from c in context.Customers
                        where c.City == "London"
                        select c;
foreach (Customer cust in custs) {
   Console.WriteLine("Заказчик: {0}", cust.CompanyName);
}
```

Как видите, запрос выполняется с использованием свойства Customers объекта ObjectContext в качестве источника, а результатом является IQueryable<Customer>. Ниже показан вывод после запуска кода из листинга 20.6:

```
Заказчик: Around the Horn
Заказчик: B's Beverages
Заказчик: Consolidated Holdings
Заказчик: Eastern Connection
Заказчик: North/South
Заказчик: Seven Seas Imports
```

Скомпилированные запросы

Для повышения производительности LINQ to Entities поддерживает скомпилированные запросы. Статический метод CompiledQuery.Compile получает запрос и возвращает Func, принимающий ObjectContext и до 16 параметров запроса. Лучше всего объяснить это на примере. В листинге 20.7 содержатся два запроса LINQ to Entities, которые получают множество заказчиков, находящихся в Лондоне и Париже.

Листинг 20.7. Похожие запросы LINQ to Entities

```
// Создать ObjectContext.
NorthwindEntities context = new NorthwindEntities();

// Запросить лондонских заказчиков.
IQueryable<Customer> londonCustomers = from customer in context.Customers
                                 where customer.City == "LONDON"
                                 select customer;

// Вывести имена лондонских заказчиков.
foreach (Customer cust in londonCustomers) {
   Console.WriteLine("Лондонский заказчик: {0}", cust.CompanyName);
}

// Запросить парижских заказчиков.
IQueryable<Customer> parisCustomers = from customer in context.Customers
                                where customer.City == "PARIS"
                                select customer;

// Вывести имена парижских заказчиков
foreach (Customer cust in parisCustomers) {
   Console.WriteLine("Парижский заказчик: {0}", cust.CompanyName);
}
```

Для каждого города определяется один и тот же запрос, меняется только название города. Запуск кода из листинга 20.7 дает следующие результаты:

```
Лондонский заказчик: Around the Horn
Лондонский заказчик: B's Beverages
Лондонский заказчик: Consolidated Holdings
Лондонский заказчик: Eastern Connection
Лондонский заказчик: North/South
Лондонский заказчик: Seven Seas Imports
Парижский заказчик: Paris spécialités
Парижский заказчик: Spécialités du monde
```

Чтобы создать скомпилированную версию запроса из листинга 20.7, вызывается метод `CompiledQuery.Compile`, как показано ниже. Первый аргумент — всегда `ObjectContext` для сущностной модели данных. Последний аргумент — это результат запроса, в данном случае `IQueryable<Customer>`. Прочие аргументы — параметры, которые необходимо передать запросу, чтобы сделать его многократно используемым. В конце концов, нет смысла компилировать запрос, если он не может применяться более одного раза. В рассматриваемом примере понадобится возможность указания разных городов, поэтому есть один аргумент `string`.

```
Func<NorthwindEntities, string, IQueryable<Customer>> compiledQuery
   = CompiledQuery.Compile<NorthwindEntities, string, IQueryable<Customer>>(
      (ctx, city) =>
        from customer in ctx.Customers
        where customer.City == city
        select customer);
```

Возвращаемым типом метода `Compile` является `Func`, строго типизированный в соответствии с типами, указанными для метода `Compile`. В нашем случае получается `Func<NorthwindEntities, String, IQueryable<Customer>>`. Теперь для повторного использования запроса просто вызывается функция и передаются параметры. В листинге 20.8 показано, как это делается.

Листинг 20.8. Использование скомпилированного запроса LINQ to Entities

```
// Определить скомпилированный запрос.
Func<NorthwindEntities, string, IQueryable<Customer>> compiledQuery
   = CompiledQuery.Compile<NorthwindEntities, string, IQueryable<Customer>>(
      (ctx, city) =>
         from customer in ctx.Customers
         where customer.City == city
         select customer);

// Создать ObjectContext.
NorthwindEntities context = new NorthwindEntities();

// Определить интересующие города.
string[] cities = new string[] { "London", "Paris" };

// Вызвать скомпилированный запрос для каждого города.
foreach (string city in cities) {
   IQueryable<Customer> custs = compiledQuery(context, city);
   foreach (Customer cust in custs) {
      Console.WriteLine("Заказчик из {0}: {1}", city, cust.CompanyName);
   }
}
```

Мы определяем функцию компилированного запроса и затем вызываем ее для каждого интересующего нас города. Запрос компилируется при первом его использовании,

что обеспечивает повышение производительности, — особенно для сложных запросов. Результат выполнения кода из листинга 20.8 выглядит так:

```
Заказчик из London: Around the Horn
Заказчик из London: B's Beverages
Заказчик из London: Consolidated Holdings
Заказчик из London: Eastern Connection
Заказчик из London: North/South
Заказчик из London: Seven Seas Imports
Заказчик из Paris: Paris spécialités
Заказчик из Paris: Spécialités du monde
```

Просмотр оператора SQL

Часто бывает полезно увидеть оператор SQL, в который транслируется запрос LINQ to Entities. К сожалению, не существует удобного пути сделать это для всех операторов SQL, которые создает экземпляр `ObjectContext`. Однако просмотреть оператор SQL, который генерирует одиночный запрос LINQ to Entities, можно, приведя результат `IQueryable<T>` от запроса LINQ to Entities к конкретному классу `ObjectQuery` и вызвав метод `ToTraceString`. Соответствующий код приведен в листинге 20.9.

Листинг 20.9. Отображение оператора SQL

```
// Создать ObjectContext.
NorthwindEntities context = new NorthwindEntities();

// Запросить лондонских заказчиков.
IQueryable<Customer> londonCustomers = from customer in context.Customers
                                       where customer.City == "LONDON"
                                       select customer;

// Удостовериться, что соединение с базой данных открыто.
if (context.Connection.State != ConnectionState.Open) {
   context.Connection.Open();
}

// Отобразить оператор SQL.
string sqlStatement = (londonCustomers as ObjectQuery).ToTraceString();
Console.WriteLine(sqlStatement);
```

В листинге 20.9 определен запрос, который выберет всех заказчиков Nothwind, находящихся в Лондоне. Затем осуществляется проверка наличия открытого соединения с базой данных. Используемые здесь члены `ObjectContext` подробно рассматриваются в главе 21, а пока просто знайте, что, не имея открытого соединения, вы получите исключение, если попытаетесь получить оператор SQL из запроса.

Чтобы получить оператор SQL, результирующее перечисление из LINQ-запроса `IQueryable<Customer>` приводится к `ObjectQuery` и вызывается метод `ToTraceString`. Он возвращает строку, содержащую оператор SQL, в который транслируется запрос, и строка затем выводится на консоль. Компиляция и запуск кода из листинга 20.9 даст следующий вывод:

```
SELECT
[Extent1].[CustomerID] AS [CustomerID],
[Extent1].[CompanyName] AS [CompanyName],
[Extent1].[ContactName] AS [ContactName],
[Extent1].[ContactTitle] AS [ContactTitle],
[Extent1].[Address] AS [Address],
[Extent1].[City] AS [City],
[Extent1].[Region] AS [Region],
```

```
[Extent1].[PostalCode] AS [PostalCode],
[Extent1].[Country] AS [Country],
[Extent1].[Phone] AS [Phone],
[Extent1].[Fax] AS [Fax]
FROM [dbo].[Customers] AS [Extent1]
WHERE N'LONDON' = [Extent1].[City]
```

Получение этого SQL-оператора еще не означает немедленное выполнение запроса — просто запрос LINQ to Entities транслируется в оператор SQL. Понятно, что это не слишком элегантный прием. Хотя он применяется в приведенном примере, все же предпочтительнее пользоваться инструментом профилирования SQL Server Profiler. Если этот инструмент отсутствует (например, он не входит в состав версии SQL Server Express Edition, поставляемой с Visual Studio 2010), рекомендуется воспользоваться бесплатным средством профилирования SQL с открытым исходным кодом от Anjlab, который доступен по адресу `http://sites.google.com/site/sqlprofiler`. Инструмент профилирования позволит увидеть все операторы SQL, посылаемые базе данных, а не только от каждого конкретного запроса.

Загрузка связанных объектов

Сущностные типы ассоциируются, когда между ними устанавливается отношение внешнего ключа. Сущностные объекты (т.е. экземпляры сущностных типов) связаны друг с другом через специфическое значение внешнего ключа. Например, сущностные типы `Customer` и `Order` из базы данных Northwind ассоциированы, и также связаны объекты `Customer` для `Round the Horn` и `Order` для `Round the Horn`. LINQ to Entities облегчает навигацию по данным, автоматически отслеживая ассоциации между ними. Взаимосвязанные объекты загружаются “за кулисами”, так что код работает прозрачно. Однако стоит уделить внимание тому, как загружаются связанные объекты.

Ленивая загрузка

“Ленивая” загрузка объектов является поведением по умолчанию LINQ to Entities. Связанные объекты загружаются из базы данных, только когда происходит обращение к ассоциированному свойству сущностного типа. То, что не нужно, никогда не загружается — это называется стратегией *оперативной* (just-in-time) загрузки, но это означает возможность получить неожиданно большое количество запросов SQL, генерируемых кодом. Данная проблема демонстрируется в листинге 20.10.

Листинг 20.10. Эффект от ленивой загрузки объектов

```
// Создать ObjectContext.
NorthwindEntities context = new NorthwindEntities();
IQueryable<Customer> custs = from c in context.Customers
                        where c.Country == "UK" &&
                          c.City == "London"
                        orderby c.CustomerID
                        select c;
foreach (Customer cust in custs) {
   Console.WriteLine("{0} - {1}", cust.CompanyName, cust.ContactName);
   Order firstOrder = cust.Orders.First();
   Console.WriteLine("   {0}", firstOrder.OrderID);
}
```

В листинге 20.10 запрашиваются заказчики из Лондона с упорядочиванием результатов по полю `CustomerID`. Затем на консоль выводится название и контактное лицо в каждой компании, наряду с `OrderID` первого заказа, ассоциированного с заказчиком.

Компиляция и запуск кода, приведенного в листинге 20.10, приводит к следующим результатам, включающим шесть заказчиков, соответствующих критерию запроса:

```
Around the Horn - Thomas Hardy
   10355
B's Beverages - Victoria Ashworth
   10289
Consolidated Holdings - Elizabeth Brown
   10435
Eastern Connection - Ann Devon
   10364
North/South - Simon Crowther
   10517
Seven Seas Imports - Hari Kumar
   10359
```

Разумеется, объекты `Order`, связанные с каждым `Customer`, не загружаются, пока не будет произведено обращение к полю `Customer.Orders`. Когда это делается, Entity Framework незаметно запрашивает базу данных и загружает необходимые данные. Никакие другие объекты, связанные с типом `Customer`, не загружаются.

В зависимости от конкретного проекта, такой подход может быть либо гениальным, либо совершенно безумным. Гениальным он может быть потому, что из базы данных получается только то, что необходимо, и только тогда, когда нужно. Безумным же он может быть потому, что даже простой запрос LINQ может превратиться во множество запросов к базе данных. В случае, приведенном в листинге 20.10, получаются семь сгенерированных запросов SQL: один для получения списка заказчиков из Лондона и шесть — для получения заказов от каждого из заказчиков. Для некоторых проектов семь запросов для столь простого куска кода будет чересчур, и в последующих разделах будут продемонстрированы альтернативные подходы.

В отношении листинга 20.10 говорится “до семи” запросов, потому что Entity Framework кэширует данные для повышения производительности. Некоторые данные, которые соответствуют запросу, уже могут быть кэшированы. Чтобы отключить ленивую загрузку, необходимо установить опцию в `ObjectContext`, как показано ниже:

```
context.ContextOptions.LazyLoadingEnabled = false;
```

При отключенной ленивой загрузке попытка обратиться к связанному сущностному объекту приводит к генерации исключения, если только не используется один из описанных ниже приемов обеспечения загрузки данных.

Немедленная загрузка

Когда точно известно, какие данные понадобятся в коде, как это было в предыдущем листинге, можно в качестве части запроса LINQ to Entities использовать метод `Include` для загрузки связанных сущностных объектов. Метод `Include` применяется к запросу указанием имени свойства ассоциации между запрашиваемым типом и типом, который требуется загрузить в виде строки; в рассматриваемом случае свойством, ассоциирующим тип `Customer` с типом `Order`, будет `Orders`, так что необходимо вызвать метод `Include` со строковым аргументом `"Orders"`. В листинге 20.11 демонстрируется немедленная загрузка для запроса, который использовался в листинге 20.10.

Листинг 20.11. Немедленная загрузка данных `Orders`

```
// Создать ObjectContext.
NorthwindEntities context = new NorthwindEntities();
IQueryable<Customer> custs = from c in context.Customers
                             .Include("Orders")
```

```
                        where c.Country == "UK" &&
                          c.City == "London"
                        orderby c.CustomerID
                        select c;

foreach (Customer cust in custs) {
   Console.WriteLine("{0} - {1}", cust.CompanyName, cust.ContactName);
   Order firstOrder = cust.Orders.First();
   Console.WriteLine("    {0}", firstOrder.OrderID);
}
```

Скомпилировав и запустив код из листинга 20.11, получится тот же результат, что и у кода из листинга 20.10, но в базу данных будет отправлен только один запрос, показанный ниже:

```
[Project1].[Freight] AS [Freight],
[Project1].[ShipName] AS [ShipName],
[Project1].[ShipAddress] AS [ShipAddress],
[Project1].[ShipCity] AS [ShipCity],
[Project1].[ShipRegion] AS [ShipRegion],
[Project1].[ShipPostalCode] AS [ShipPostalCode],
[Project1].[ShipCountry] AS [ShipCountry]
FROM ( SELECT
      [Extent1].[CustomerID] AS [CustomerID],
      [Extent1].[CompanyName] AS [CompanyName],
      [Extent1].[ContactName] AS [ContactName],
      [Extent1].[ContactTitle] AS [ContactTitle],
      [Extent1].[Address] AS [Address],
      [Extent1].[City] AS [City],
      [Extent1].[Region] AS [Region],
      [Extent1].[PostalCode] AS [PostalCode],
      [Extent1].[Country] AS [Country],
      [Extent1].[Phone] AS [Phone],
      [Extent1].[Fax] AS [Fax],
      1 AS [C1],
      [Extent2].[OrderID] AS [OrderID],
      [Extent2].[CustomerID] AS [CustomerID1],
      [Extent2].[EmployeeID] AS [EmployeeID],
      [Extent2].[OrderDate] AS [OrderDate],
      [Extent2].[RequiredDate] AS [RequiredDate],
      [Extent2].[ShippedDate] AS [ShippedDate],
      [Extent2].[ShipVia] AS [ShipVia],
      [Extent2].[Freight] AS [Freight],
      [Extent2].[ShipName] AS [ShipName],
      [Extent2].[ShipAddress] AS [ShipAddress],
      [Extent2].[ShipCity] AS [ShipCity],
      [Extent2].[ShipRegion] AS [ShipRegion],
      [Extent2].[ShipPostalCode] AS [ShipPostalCode],
      [Extent2].[ShipCountry] AS [ShipCountry],
      CASE WHEN ([Extent2].[OrderID] IS NULL) THEN CAST(NULL AS int) ELSE 1 END AS
[C2]
      FROM [dbo].[Customers] AS [Extent1]
      LEFT OUTER JOIN [dbo].[Orders] AS [Extent2] ON [Extent1].[CustomerID] =
[Extent2].[CustomerID]
      WHERE (N'UK' = [Extent1].[Country]) AND (N'London' = [Extent1].[City])
) AS [Project1]
ORDER BY [Project1].[CustomerID] ASC, [Project1].[C2] ASC
```

Немедленно загружать можно любое количество связанных сущностных типов, применяя метод `Include` к каждому необходимому типу. В листинге 20.20 показан запрос для `Orders`, который немедленно загружает связанные сущностные типы `Shipper` и `Customer`.

Листинг 20.12. Немедленная загрузка множества связанных сущностных типов

```
// Создать ObjectContext.
NorthwindEntities context = new NorthwindEntities();
IQueryable<Order> orders = context.Orders
   .Include("Shipper")
   .Include("Customer")
   .Where(c => c.ShipCountry == "France")
   .Select(c => c);
foreach (Order ord in orders) {
   Console.WriteLine("OrderID: {0}, Shipper: {1}, Contact: {2}",
      ord.OrderID,
      ord.Shipper.CompanyName,
  ord.Customer.ContactName);
}
```

В запрос LINQ включены сущностные типы `Shipper` и `Customer`, что порождает один запрос к базе данных, который извлекает поля трех разных сущностных типов. Результат выполнения этого запроса выглядит следующим образом:

```
OrderID: 10248, Shipper: Federal Shipping, Contact: Paul Henriot
OrderID: 10251, Shipper: Speedy Express, Contact: Mary Saveley
OrderID: 10265, Shipper: Speedy Express, Contact: Frédérique Citeaux
OrderID: 10274, Shipper: Speedy Express, Contact: Paul Henriot
OrderID: 10295, Shipper: United Package, Contact: Paul Henriot
OrderID: 10297, Shipper: United Package, Contact: Frédérique Citeaux
OrderID: 10311, Shipper: Federal Shipping, Contact: Janine Labrune
OrderID: 10331, Shipper: Speedy Express, Contact: Laurence Lebihan
...
```

Явная загрузка

Если необходим полный контроль, то лучше всего подойдет явная загрузка. Загружаемые связанные объекты указываются с использованием метода `EntityCollection.Load`. В листинге 20.13 демонстрируется выборочная загрузка связанных объектов.

Листинг 20.13. Явная загрузка для управления запросами к базе данных

```
// Создать ObjectContext.
NorthwindEntities context = new NorthwindEntities();

// Отключить ленивую загрузку.
context.ContextOptions.LazyLoadingEnabled = false;
IQueryable<Customer> custs = context.Customers
   .Where(c => c.Country == "UK" && c.City == "London")
   .OrderBy(c => c.CustomerID)
   .Select(c => c);

// Выполнить явную загрузку заказов для каждого заказчика.
foreach (Customer cust in custs) {
   if (cust.CompanyName != "North/South") {
      cust.Orders.Load();
   }
}
```

```
foreach (Customer cust in custs) {
   Console.WriteLine("{0} - {1}", cust.CompanyName, cust.ContactName);
   // Проверить загрузку заказов для каждого заказчика.
   if (cust.Orders.IsLoaded) {
      Order firstOrder = cust.Orders.First();
      Console.WriteLine("    {0}", firstOrder.OrderID);
   } else {
      Console.WriteLine("  Данные заказа не загружены");
   }
}
```

Для использования явной загрузки ленивая загрузка должна быть отключена. В противном случае Entity Framework все равно загрузит связанные объекты автоматически. В приведенном выше примере выполняется запрос LINQ для получения всех заказчиков из Лондона, а затем для всех заказчиков, кроме `North/South`, явно загружаются связанные объекты `Order`. После этого результаты снова перечисляются с выводом на консоль запрошенных данных. С помощью метода `IsLoaded` определяется то, были ли загружены связанные объекты. Использование явной загрузки может быть чревато ошибками, если только тщательно не проверить загруженные объекты перед тем, как к ним обращаться. Но если необходим полный контроль над загружаемыми данными, то это — идеальное решение.

Опрос представлений

При генерации сущностной модели для базы данных можно включить в нее поддержку любых существующих представлений. Если вы следовали инструкциям из предыдущей главы, то выбрали все представления базы данных Northwind во время генерации сущностной модели данных для примеров. Запрос к представлению подобен запросу к таблице. В листинге 20.14 демонстрируется использование представления `Customers and Suppliers by City` из базы данных Northwind.

Листинг 20.14. Использование LINQ для запроса к представлению базы данных

```
// Создать ObjectContext.
NorthwindEntities context = new NorthwindEntities();
IQueryable<Customer_and_Suppliers_by_City> res
   = context.Customer_and_Suppliers_by_Cities
   .Where(c => c.City == "LONDON")
   .Select(c => c);
foreach (Customer_and_Suppliers_by_City r in res) {
   Console.WriteLine("{0}, {1}", r.CompanyName, r.ContactName);
}
```

Сущностная модель данных определяет сущностный тип по имени `Customer_and_Suppliers_by_City`, который собирается в свойстве `Customer_and_Suppliers_by_City` объекта `ObjectContext`. Имя представления принимает форму множественного числа посредством мастера Entity Data Model Wizard, что можно отключить при генерации модели. Помимо причудливых имен типов опрос представления во всем подобен опросу таблицы; результат компиляции и выполнения кода из листинга 20.14 показан ниже:

```
Around the Horn, Thomas Hardy
B's Beverages, Victoria Ashworth
Consolidated Holdings, Elizabeth Brown
Eastern Connection, Ann Devon
Exotic Liquids, Charlotte Cooper
North/South, Simon Crowther
Seven Seas Imports, Hari Kumar
```

Опрос хранимых процедур

Использование хранимых процедур несколько сложнее, чем использование представлений. Хранимая процедура должна быть явно импортирована в сущностную модель данных. Однако волноваться не стоит, т.к. большую часть работы Visual Studio сделает самостоятельно. Первый шаг для импорта хранимой процедуры — это открытие окна Model Browser (Браузер модели) в Visual Studio 2010. Для этого выберите пункт меню View⇨Other Windows⇨Entity Data Model Browser (Вид⇨Другие окна⇨Браузер сущностной модели данных). На рис. 20.2 показан внешний вид браузера для сущностной модели данных Northwind.

Планируется импортировать и использовать хранимую процедуру `Customers_By_City`, которая выделена на рисунке 20.2. Чтобы приступить к импорту, дважды щелкните на имени хранимой процедуры для открытия диалогового окна Add Function Import (Добавить импорт функции), показанного на рис. 20.3.

В поле Function Import Name (Имя импорта функции) указывается имя свойства `ObjectContext`, которое будет использовано для вызова хранимой процедуры. В рассматриваемом примере вполне подойдет имя, предлагаемое по умолчанию. Наиболее важный момент — установка типа возврата для хранимой процедуры. Если процедура возвращает поля, необходимые для наполнения имеющегося сущностного типа, то можно выбрать этот тип в раскрывающемся списке. Если же типа возврата нет либо процедура возвращает коллекцию скалярных типов, можно указать это поведение в диалоговом окне.

Хранимая процедура, которую необходимо использовать, не отображается удобно на имеющийся сущностный тип, поэтому часть импорта процедуры будет создание нового типа. Для этого щелкните на кнопке Get Column Information (Получить информацию о столбце), а затем — на кнопке Create New Complex Type (Создать новый сложный тип). Диалоговое окно должно выглядеть примерно так, как на рис. 20.4.

Рис. 20.2. Браузер сущностной модели данных

Рис.20.3. Диалоговое окно Add Function Import

Имя нового типа можно изменить, однако вариант по умолчанию вполне устраивает. Все, что теперь осталось — щелкнуть на кнопке OK. После этого в браузере модели должны появиться две сущности: одна для импортированной процедуры и одна — для нового типа результата. Это видно на рис. 20.5.

Итак, хранимая процедура импортирована, и новый метод `Customers_By_City` можно вызывать в классе-наследнике `ObjectContext`. В нашем случае хранимая процедура принимает единственный параметр (название города для запроса) и возвращает последовательность нового сложного типа, который был создан — `IEnumerable<Customers_By_City_Result>`. В листинге 20.15 демонстрируется использование хранимой процедуры для получения деталей о заказчиках, находящихся в Лондоне.

Листинг 20.15. Опрос импортированной хранимой процедуры

```
// Создать ObjectContext.
NorthwindEntities context = new NorthwindEntities();
IEnumerable<Customers_By_City_Result> custs = context.Customers_By_City("London");
foreach (Customers_By_City_Result cust in custs) {
   Console.WriteLine("{0}, {1}", cust.CompanyName, cust.ContactName);
}
```

Соединения

Способ, которым Entity Framework обрабатывает ассоциации и связанные объекты, может быть интуитивно понятным и удобным, но он работает только при наличии отношения внешнего ключа в базе данных. Когда нужно выполнять запросы между типами, не имеющими ассоциаций, придется явно соединять таблицы.

Рис. 20.4. Генерация нового сложного типа для поддержки хранимой процедуры

Рис. 20.5. Импортированная хранимая процедура и новый сложный тип результата

Внутренние соединения

Соединения в LINQ to Entities выполняются подобно тому, как было описано в главе 14 при рассмотрении LINQ to SQL — с использованием операции `Join`. Как это принято во внутренних соединениях, любые записи в выходном результирующем наборе будут опущены, если соответствующая запись не существует во внутреннем результирующем наборе. В листинге 20.16 приведен пример.

Листинг 20.16. Внутреннее соединение LINQ to Entities

```
// Создать ObjectContext.
NorthwindEntities context = new NorthwindEntities();
var entities = from s in context.Suppliers
            join c in context.Customers on s.City equals c.City
            select new {
               SupplierName = s.CompanyName,
               CustomerName = c.CompanyName,
               City = c.City
            };
foreach (var e in entities) {
   Console.WriteLine("{0}: {1} - {2}", e.City, e.SupplierName, e.CustomerName);
}
```

В листинге 20.16 выполняется внутреннее соединение поставщиков и заказчиков. Если отсутствует запись заказчика в городе поставщика, запись поставщика будет исключена из результирующего набора. Вот результат запуска программы из листинга 20.16:

```
London: Exotic Liquids - Around the Horn
London: Exotic Liquids - B's Beverages
London: Exotic Liquids - Consolidated Holdings
London: Exotic Liquids - Eastern Connection
London: Exotic Liquids - North/South
London: Exotic Liquids - Seven Seas Imports
Sao Paulo: Refrescos Americanas LTDA - Comércio Mineiro
Sao Paulo: Refrescos Americanas LTDA - Familia Arquibaldo
Sao Paulo: Refrescos Americanas LTDA - Queen Cozinha
Sao Paulo: Refrescos Americanas LTDA - Tradição Hipermercados
Berlin: Heli Süßwaren GmbH & Co. KG - Alfreds Futterkiste
Paris: Aux joyeux ecclésiastiques - Paris spécialités
Paris: Aux joyeux ecclésiastiques - Spécialités du monde
Montréal: Ma Maison - Mère Paillarde
```

Как видите, несмотря на то, что некоторые поставщики присутствуют в выводе с несколькими соответствующими заказчиками, другие поставщики отсутствуют вообще. Причина в том, что для них не найдено заказчиков из тех же городов. Если же необходимо получать поставщиков независимо от наличия в их городах заказчиков, понадобится выполнить внешнее соединение.

Внешние соединения

Как и в LINQ to SQL, для выполнения внешних соединений в LINQ to Entities может применяться стандартная операция запроса `DefaultIfEmpty`. В листинге 20.17 используется конструкция `into` для направления соответствующих результатов оператора `join` во временную последовательность, на которой затем будет вызвана операция `DefaultIfEmpty`. Таким образом, если запись отсутствует в объединенном результате, будет представлено значение по умолчанию.

Листинг 20.17. Внешнее соединение LINQ to Entities

```
// Создать ObjectContext.
NorthwindEntities context = new NorthwindEntities();
var entities =
  from s in context.Suppliers
  join c in context.Customers on s.City equals c.City into temp
  from t in temp.DefaultIfEmpty()
  select new {
      SupplierName = s.CompanyName,
      CustomerName = t.CompanyName,
      City = s.City
  };
foreach (var e in entities) {
    Console.WriteLine("{0}: {1} - {2}", e.City, e.SupplierName, e.CustomerName);
}
```

Обратите внимание, что в операторе `join` в листинге 20.17 результат соединения направляется во временную последовательность по имени `temp`. Имя этой временной последовательности может быть любым, если только оно не конфликтует с другим именем или ключевым словом. Затем выполняется последовательный запрос на результате последовательности `temp`, переданной операции `DefaultIfEmpty`. Если трассировать оператор SQL, передаваемый базе данных, можно увидеть вот что:

```
SELECT
1 AS [C1],
[Extent1].[CompanyName] AS [CompanyName],
[Extent2].[CompanyName] AS [CompanyName1],
[Extent1].[City] AS [City]
FROM [dbo].[Suppliers] AS [Extent1]
LEFT OUTER JOIN [dbo].[Customers] AS [Extent2] ON ([Extent1].[City] =
[Extent2].[City]) OR (([Extent1].[City] IS NULL) AND ([Extent2].[City] IS NULL))
```

Если вы сравните это с оператором SQL, генерируемым LINQ to SQL в главе 14 для того же внешнего запроса, то заметите, что LINQ to Entities транслирует запрос по-другому. Запуск кода из листинга 20.17 выдаст следующий результат:

```
London: Exotic Liquids - Around the Horn
London: Exotic Liquids - B's Beverages
London: Exotic Liquids - Consolidated Holdings
London: Exotic Liquids - Eastern Connection
London: Exotic Liquids - North/South
London: Exotic Liquids - Seven Seas Imports
New Orleans: New Orleans Cajun Delights -
Ann Arbor: Grandma Kelly's Homestead -
Tokyo: Tokyo Traders -
Oviedo: Cooperativa de Quesos 'Las Cabras' -
Osaka: Mayumi's -
Melbourne: Pavlova, Ltd. -
Manchester: Specialty Biscuits, Ltd. -
Göteborg: PB Knäckebröd AB -
Sao Paulo: Refrescos Americanas LTDA - Comércio Mineiro
Sao Paulo: Refrescos Americanas LTDA - Familia Arquibaldo
Sao Paulo: Refrescos Americanas LTDA - Queen Cozinha
Sao Paulo: Refrescos Americanas LTDA - Tradição Hipermercados
Berlin: Heli Süßwaren GmbH & Co. KG - Alfreds Futterkiste
Frankfurt: Plutzer Lebensmittelgroßmärkte AG -
Cuxhaven: Nord-Ost-Fisch Handelsgesellschaft mbH -
Ravenna: Formaggi Fortini s.r.l. -
```

```
Sandvika: Norske Meierier -
Bend: Bigfoot Breweries -
Stockholm: Svensk Sjöföda AB -
Paris: Aux joyeux ecclésiastiques - Paris spécialités
Paris: Aux joyeux ecclésiastiques - Spécialités du monde
Boston: New England Seafood Cannery -
Singapore: Leka Trading -
Lyngby: Lyngbysild -
Zaandam: Zaanse Snoepfabriek -
Lappeenranta: Karkki Oy -
Sydney: G'day, Mate -
Montréal: Ma Maison - Mère Paillarde
Salerno: Pasta Buttini s.r.l. -
Montceau: Escargots Nouveaux -
Annecy: Gai pâturage -
Ste-Hyacinthe: Forêts d'érables -
: Sharp As You Like -
```

Как видно в выводе листинга 20.17, для каждого поставщика получается, по крайней мере, одна запись, причем некоторые поставщики не имеют соответствующего заказчика, т.к. выполняется внешнее соединение. Но если возникают какие-то сомнения, всегда можно посмотреть сгенерированный оператор SQL, и станет ясно, что здесь действительно выполняется внешнее соединение.

Обновления

Обновления сущностных типов заключаются в простом изменении свойств сущностного объекта, вызове метода `SaveChanges` объекта `ObjectContext` с разрешением конфликтов параллельного доступа, если это необходимо. Разрешение этих конфликтов в Entity Framework рассматривается далее в главе. В листинге 20.18 приведен простой пример обновления.

Листинг 20.18. Простое обновление сущностного объекта

```
// Создать ObjectContext.
NorthwindEntities context = new NorthwindEntities();

// Извлечь заказчика LAZYK.
Customer cust = (from c in context.Customers
              where c.CustomerID == "LAZYK"
              select c).Single<Customer>();

// Обновить контактное имя.
cust.ContactName = "Ned Plimpton";

// Сохранить изменения.
context.SaveChanges();

// Восстановить базу данных.
cust.ContactName = "John Steel";
context.SaveChanges();
```

В листинге 20.18 выполняется запрос для нахождения `Customer` со значением `CustomerID`, равным `LAZYK`. Затем его значение `ContactName` изменяется на `Ned Plimpton`. После этого вызывается метод `SaveChanges` для сохранения изменений в базе данных. Для восстановления базы данных значение `ContactName` изменяется обратно на `John Steel` и вызывается `SaveChanges` опять, сохраняя базу данных в исходном виде.

Обновление ассоциированных объектов

Entity Framework заботится об управлении отношениями между ассоциированными типами данных. Вы просто должны внести нужные изменения и вызвать метод `SaveChanges`. Пример приведен в листинге 20.19.

Листинг 20.19. Обновление отношения ассоциированных типов

```
// Создать ObjectContext.
NorthwindEntities context = new NorthwindEntities();
Order order = (from o in context.Orders
            where o.EmployeeID == 5
            orderby o.OrderDate descending
            select o).First<Order>();

// Сохранить текущего сотрудника, чтобы можно было восстановить его в конце.
Employee origEmployee = order.Employee;
Console.WriteLine("Перед изменением сотрудника.");

Console.WriteLine("OrderID = {0} : OrderDate = {1} : EmployeeID = {2}",
  order.OrderID, order.OrderDate, order.Employee.EmployeeID);

Employee emp = (from e in context.Employees
             where e.EmployeeID == 9
             select e).Single<Employee>();

// Назначить нового сотрудника на заказ.
order.Employee = emp;
context.SaveChanges();
Order order2 = (from o in emp.Orders
             where o.OrderID == order.OrderID
             select o).First<Order>();

Console.WriteLine("{0}После изменения сотрудника.", System.Environment.NewLine);
Console.WriteLine("OrderID = {0} : OrderDate = {1} : EmployeeID = {2}",
  order2.OrderID, order2.OrderDate, order2.Employee.EmployeeID);

// Теперь необходимо отменить изменения, чтобы можно было пример запускать многократно.
order.Employee = origEmployee;
context.SaveChanges();
```

В листинге 20.19 содержится тот же пример, что использовался для демонстрации обновлений ассоциированных классов в LINQ to SQL, но код изменен для работы с LINQ to Entities. Здесь запрашивается `Order` и обновляется его отношение с типом `Employee`, после чего база данных восстанавливается в исходном состоянии. Как и LINQ to SQL, LINQ to Entities позаботится об управлении изменениями в связанных данных на основе ассоциаций в сущностных объектах.

Удаления

Чтобы удалить запись из базы данных, необходимо просто передать сущностный объект, представляющий эту запись, в качестве аргумента методу `ObjectContext.DeleteObject`. Пример использования этого метода приведен в листинге 20.20.

Внимание! В примерах этого раздела база данных не восстанавливается в исходном состоянии. Чтобы можно было запускать другие примеры, не получая неожиданных результатов, потребуется отключить базу данных Northwind от SQL Server 2008 и присоединить ее исходную версию, которая загружалась вместе с примерами.

Листинг 20.20. Удаление записи удалением сущностного объекта

```
// Создать ObjectContext.
NorthwindEntities context = new NorthwindEntities();
// Получить детали заказа 10248.
IQueryable<Order_Detail> ods = from o in context.Order_Details
                           where o.OrderID == 10248
                           select o;
// Вывести результаты запроса на консоль.
Console.WriteLine("Перед удалением.");
foreach (Order_Detail od in ods) {
   Console.WriteLine("Детали заказа {0}, {1}, {2}",
      od.ProductID, od.UnitPrice, od.Quantity);
}
// Удалить детали первого заказа.
context.DeleteObject(ods.First());
// Сохранить изменения.
context.SaveChanges();
// Вывести результаты запроса на консоль.
Console.WriteLine("После удаления.");
foreach (Order_Detail od in ods) {
   Console.WriteLine("Детали заказа {0}, {1}, {2}",
      od.ProductID, od.UnitPrice, od.Quantity);
}
```

В листинге 20.20 запрашиваются все сущностные объекты `Order_Detail` со значением `OrderID`, равным `10248`. Затем с помощью метода `First` первый из них выбирается и передается в качестве аргумента методу `ObjectContext.DeleteObject`. Для сохранения изменений вызывается метод `ObjectContext.SaveChanges`, который отправляет команду удаления в базу данных.

Класс `EntitySet` также имеет метод `DeleteObject`, а это значит, что можно получить тот же эффект, что и в листинге 20.20, с использованием сущностного объекта, представляющего таблицу, из которой нужно удалить запись. Листинг 20.21 содержит пример.

Листинг 20.21. Удаление записи с использованием сущностного класса

```
// Создать ObjectContext.
NorthwindEntities context = new NorthwindEntities();
// Получить детали заказа 10248.
IQueryable<Order_Detail> ods = from o in context.Order_Details
                           where o.OrderID == 10248
                           select o;
// Вывести результаты запроса на консоль.
Console.WriteLine("Перед удалением.");
foreach (Order_Detail od in ods) {
   Console.WriteLine("Order detail {0}, {1}, {2}",
      od.ProductID, od.UnitPrice, od.Quantity);
}
// Удалить детали первого заказа.
context.Order_Details.DeleteObject(ods.First());
// Сохранить изменения .
context.SaveChanges();
// Вывести результаты запроса на консоль.
Console.WriteLine("После удаления.");
foreach (Order_Detail od in ods) {
   Console.WriteLine("Order detail {0}, {1}, {2}",
      od.ProductID, od.UnitPrice, od.Quantity);
}
```

В листинге 20.21 выполняется тот же запрос, что и в листинге 20.20, но здесь используется метод EntirySet.DeleteObject. Поскольку необходимо удалить экземпляр сущностного типа Order_Detail, вызывается метод DeleteObject на свойстве Order_Details объекта ObjectContext, который является экземпляром EntitySet, представляющим таблицу Order_Details в базе данных. Компиляция и запуск кода, приведенного в листингах 20.20 и 20.21, даст одинаковый результат, показанный ниже:

```
Перед удалением.
Order detail 11, 14.0000, 12
Order detail 42, 9.8000, 10
Order detail 72, 34.8000, 5
После удаления.
Order detail 42, 9.8000, 10
Order detail 72, 34.8000, 5
```

Удаление связанных объектов

Удаление сущностного объекта в Entity Framework не приводит к удалению автоматически связанных с ним объектов. При удалении сущностного объекта, имеющего связанные с ним объекты, следует проявлять осторожность: в зависимости от схемы базы данных, могут либо появиться "висячие" данные (данные, внешний ключ которых ссылается на более несуществующий первичный ключ), либо произойти исключение о нарушении ограничения схемы.

В листинге 20.22 демонстрируется то, что случится, когда объект удаляется без обработки связанных с ним объектов. В данном случае предпринимается попытка удалить сущностный объект Order.

Листинг 20.22. Удаление сущностного объекта без предварительной обработки связанных объектов

```
// Создать ObjectContext.
NorthwindEntities context = new NorthwindEntities();

// Запросить первый заказ для LAZYK.
Order firstOrder = context.Orders
    .Where(o => o.CustomerID == "LAZYK")
    .Select(o => o)
    .First();

// Удалить заказ.
context.DeleteObject(firstOrder);

// Сохранить изменения.
context.SaveChanges();
```

Компиляция и запуск кода из листинга 20.22 приведет к генерации следующего исключения:

```
Unhandled Exception: System.Data.UpdateException: An error occurred while
updating the entries. See the inner exception
for details. ---> System.Data.SqlClient.SqlException: The DELETE statement
conflicted with the REFERENCE constraint "FK_Order_Details_Orders". The conflict
occurred in database "Northwind", table "dbo.Order Details", column 'OrderID'.
The statement has been terminated.

Необработанное исключение: System.Data.UpdateException: Во время обновления
записей возникла ошибка. Для получения сведений см. внутреннее исключение.
---> System.Data.SqlClient.SqlException: Оператор DELETE конфликтует
со ССЫЛОЧНЫМ органичением "FK_Order_Details_Orders". Конфликт произошел в базе
данных "Northwind", таблица "dbo.Order Details", столбец 'OrderID'.
Оператор принудительно завершен.
```

Что же произошло? Было нарушено ограничение схемы базы данных. Исключение сообщает о том, что в таблице `Order Details` имеется ограничение по имени `FK_Order_Details_Orders`. На рис. 20.6 показано окно SQL Server Management Studio для базы данных Northwind. Здесь опция Enforce Foreign Key Constraint (Навязывать ограничение внешнего ключа) установлена в Yes (Да), а это означает, что удалить `Order` не получится, пока есть связанные с ним записи в таблице `Order Details`.

Рис. 20.6. Ограничение внешнего ключа

Совет. Инструмент SQL Server Management Studio не устанавливается вместе с SQL Server Express, когда используется программа установки Visual Studio 2010. Этот инструмент доступен для бесплатной загрузки на сайте Microsoft.

Существуют два подхода к безопасному удалению связанных объектов. Это можно сделать вручную или воспользоваться каскадным удалением в базе данных, и тогда сущностная модель данных обработает их автоматически. Мы предпочитаем автоматический подход, но поскольку бывают случаи, когда невозможно модифицировать схему базы данных, приходится применять ручной подход. Ниже продемонстрированы оба метода.

Ручное удаление связанных объектов

Возможно, простейший способ удаления связанных объектов состоит в простой передаче каждого из них методу `ObjectContext.DeleteObject`. При этом необходимо соблюдать осторожность. Во-первых, следует убедиться, что порядок их удаления не нарушает ограничений схемы. Например, если требуется удалить `Order`, то сначала следует удалить связанные объекты `Order_Detail`, а только потом удалять `Order`. Если сделать наоборот, получится исключение вроде того, что видели ранее.

Во-вторых, нужно убедиться, что они удалены все — что-то оставлять нельзя, иначе будет получено исключение или появится несколько строк “висячих” данных.

Наконец, необходимо убедиться, что связанные объекты не имеют других зависимых объектов. Сложные базы данных могут иметь множество отношений внешнего ключа, и придется разобрать их все досконально, чтобы получить возможность корректно удалить граф объектов.

Таким образом, чтобы вручную удалить Order, сначала понадобится удалить все связанные с ним объекты Order_Detail. В листинге 20.23 показано, как это делается.

Листинг 20.23. Ручное удаление графа связанных объектов

```
// Создать ObjectContext.
NorthwindEntities context = new NorthwindEntities();
// Запросить первый заказ LAZYK.
Order firstOrder = context.Orders
   .Where(o => o.OrderID == 10248)
   .Select(o => o)
   .First();
// Удалить объекты Order_Detail данного заказа.
foreach (Order_Detail od in firstOrder.Order_Details.ToArray()) {
   Console.WriteLine("Deleting order detail {0}, {1}, {2}, {3}",
      od.OrderID, od.ProductID, od.UnitPrice, od.Quantity);
   context.DeleteObject(od);
}
// Удалить заказ.
context.DeleteObject(firstOrder);
// Сохранить изменения.
context.SaveChanges();
```

В этом примере запрашивается объект Order со значением свойства OrderID, равным 10248. Он выбран именно потому, что в базе данных с ним связано более одного Order_Detail. Перед удалением собственно Order объекты Order_Detail перечисляются и удаляются один за другим.

Обратите внимание на вызов метода ToArray на Order_Details EntityCollection и перечисление результата. Если этого не сделать, пришлось бы удалять объекты из обрабатываемого перечисления и тогда после удаления первого объекта Order_Details возникло бы исключение.

Каскадное удаление связанных объектов

Другой способ обработки связанных объектов предусматривает применение каскадного удаления. Каскадное удаление означает, что в случае удаления из базы данных записи, такой как Order, связанные записи, имеющие с ней отношение внешнего ключа, вроде записей из таблицы Order Details, также будут удалены автоматически. Для демонстрации работы такого удаления в листинге 20.23 понадобится включить средство каскадного удаления в базе данных и в EDM. В примерах Northwind будет показано, как это делается, но точные детали в проектах, скорее всего, будут слегка отличаться.

На заметку! Для каждого отношения внешнего ключа, которое необходимо изменить, должно быть включено каскадное удаление в базе данных и в сущностной модели данных.

Включение каскадного удаления в базе данных Northwind

Подключитесь к серверу SQL Server, используя SQL Server Management Studio, и перейдите к FK_Order_Details_Orders в папке Database\NorthwindTables\dbo.Order Details\Keys, как показано на рис. 20.7.

Известно, что это элемент, который нужен, потому что он упомянут в исключении, полученном от кода из листинга 20.22. Щелкните правой кнопкой мыши и выберите в контекстном меню пункт Modify (Модифицировать). Раскройте часть дерева INSERT and UPDATE Specification (Спецификация INSERT и UPDATE) и измените параметр Delete Rule (Правило удаления) на Cascade (Каскадное). На рис. 20.8 показано, как это должно выглядеть.

Рис. 20.7. Элемент — ключ `FK_Order_Details_Orders`

Рис. 20.8. Модификация ограничения по ключу в SQL Server Management Studio

Внеся это изменение, щелкните на кнопке Close (Закрыть) и затем выберите пункт Save Order Details (Сохранить `Order Details`) или Save All (Сохранить все) в меню File (Файл). Подобным образом базе данных сообщается, что из таблицы `Orders` удалена строка и нужно, чтобы она автоматически удалила строки из таблицы `Order Details` с тем же значением ключа/внешнего ключа.

Включение каскадного удаления в сущностной модели данных

Каскадное удаление для таблицы `Order_Details` включено в базе данных, и теперь то же самое нужно сделать для сущностного типа `Order_Detail` в сущностной модели данных. Было бы хорошо, если бы мастер Entity Data Model Wizard обнаруживал каскадные удаления при генерации и обновлении модели, но, к сожалению, он не может этого сделать, так что придется делать это самостоятельно.

Модель должна быть модифицирована для соответствия базе данных, чтобы данные, кэшированные Entity Framework, обрабатывались правильно. Если изменить базу данных, но не менять сущностную модель, то получатся кэшированные сущностные объекты, которые не отображаются на базу данных, а это неприемлемо.

Откройте сущностную модель данных двойным щелчком на файле `.edmx` в Solution Explorer. Откройте окно браузера модели (Model Browser), выбрав пункт меню View⇨Other Windows⇨Entity Data Model Browser (Вид⇨Другие окна⇨Браузер сущностной модели данных) в среде Visual Studio 2010. Откройте древовидное представление `NorthwindDataModel\NorthwindModel\Associations`. В списке будет присутствовать элемент `FK_Order_Details_Orders`, как показано на рис. 20.9.

Выберите элемент `FK_Order_Details_Orders` для открытия детальной информации в окне Properties (Свойства) и измените значение End1 OnDelete на Cascade, как показано на рис. 20.10.

После внесения этого изменения выберите в меню Visual Studio 2010 пункт Save NorthwindDataModel.edmx (Сохранить `NorthwindDataModel.edmx`). После изменения базы данных и модели данных можно удалять объекты `Order`, а связанные с ними объекты `Order_Details` будут удалены автоматически. Это демонстрируется в листинге 20.24.

Рис. 20.9. Просмотр ограничения внешнего ключа

Рис. 20.10. Изменение действия `OnDelete`

Листинг 20.24. Каскадное удаление

```
// Создать ObjectContext.
NorthwindEntities context = new NorthwindEntities();

// Запросить заказ.
Order firstOrder = context.Orders
    .Where(o => o.OrderID == 10248)
    .Select(o => o)
    .First();

// Удалить заказ.
context.DeleteObject(firstOrder);

// Сохранить изменения.
context.SaveChanges();
```

Управление параллельным доступом

По умолчанию в Entity Framework используется модель оптимистического параллелизма. Оптимистичность означает то, что она основана на предположении, что никто другой не модифицирует данные, пока они используются, и данные сохраняются в базе без проведения проверки, не были ли они изменены кем-то другим. Это поведение показано в листинге 20.25.

Листинг 20.25. Пример проблемы параллельного доступа

```
// Создать ObjectContext.
NorthwindEntities context = new NorthwindEntities();
Customer cust = context.Customers
    .Where(c => c.CustomerID == "LAZYK")
    .Select(c => c)
    .First();
Console.WriteLine("Исходное значение: {0}", cust.ContactName);

// Изменить запись вне Entity Framework.
ExecuteStatementInDb(String.Format(
@"update Customers
    set ContactName = 'Samuel Arthur Sanders'
    where CustomerID = 'LAZYK'"));

// Получить значение базы данных вне Entity Framework.
string dbValue = GetStringFromDb(String.Format(
@"select ContactName from Customers
    where CustomerID = 'LAZYK'"));
    Console.WriteLine("Значение в базе данных: {0}", dbValue);

// Модифицировать заказчика.
cust.ContactName = "John Doe";

// Сохранить изменения.
context.SaveChanges();
```

С помощью LINQ to Entities загружается сущностный объект для `Customer` со значением `CustomerID`, равным `LAZYK`. Затем запись обновляется непосредственно, за пределами Entity Framework, и значение `ContactName` становится `Samuel Arthir Sanders`. После этого значение читается из базы данных, также за пределами Entity Framework, в результате чего сущностный объект `Customer` перестает быть синхронизированным с базой данных.

С использованием сущностного объекта `Customer` изменяется значение `ContactName` на `John Doe` и вызывается метод `SaveChanges`. Entity Framework записывает внесенное

изменение в базу данных и, поскольку оптимистическая стратегия означает надежду, что никто другой не изменял данные, переписывает изменения, внесенные вне Entity Framework.

Включение проверок параллелизма

Прежде чем записать изменения, можно заставить Entity Framework проверить, не была ли модифицирована база данных кем-то другим. Это также оптимистический параллелизм, поскольку в базе данных ничего не блокируется, пока работа ведется с сущностными объектами, однако при этом исключается проблема, продемонстрированная в листинге 20.25.

Понадобится включить проверку параллельного доступа на уровне поля. Если нужно, чтобы конфликты параллелизма проверялись для всех полей сущностного объекта, следует убедиться, что отредактированы все поля. Нет способа сообщить Entity Framework о необходимости, чтобы каждое изменение сущностного типа или даже каждое изменение всей модели данных проверялось автоматически.

Для решения проблемы, которую можно было видеть в листинге 20.25, понадобится включить проверку параллельного доступа на поле `ContactName` сущностного типа `Customer`. Первый шаг — открытие файла `.edmx` двойным щелчком в окне Solution Explorer и нахождением сущностного типа `Customer` в представлении визуального конструктора. На рис. 20.11 показано, как это должно выглядеть.

Щелкните на свойстве `ContactName` для открытия подробной информации в окне Properties и измените значение Concurency Mode на Fixed, как показано на рис. 20.12.

Наконец, сохраните изменения, выбрав пункт меню File⇨Save NorthwindEntityModel.edmx в Visual Studio.

Рис. 20.11. Сущностный тип `Customer`

Рис. 20.12. Установка режима параллельного доступа для свойства `ContactName`

Обработка конфликтов параллелизма

Как только включить проверку конфликтов параллелизма для поля сущностного объекта, сразу возникает исключение `OptimisticCincurrencyException` при попытке обновить данные, которые были модифицированы с момента загрузки сущностных объектов. Это демонстрируется в листинге 20.26.

Листинг 20.26. Обработка конфликтов параллельного доступа

```
// Создать ObjectContext.
NorthwindEntities context = new NorthwindEntities();
Customer cust = context.Customers
    .Where(c => c.CustomerID == "LAZYK")
    .Select(c => c)
    .First();
Console.WriteLine("Исходное значение: {0}", cust.ContactName);
// Изменить запись вне Entity Framework.
ExecuteStatementInDb(String.Format(
    @"update Customers
set ContactName = 'Samuel Arthur Sanders'
where CustomerID = 'LAZYK'"));
// Модифицировать заказчика.
cust.ContactName = "John Doe";
// Сохранить изменения.
try {
    context.SaveChanges();
} catch (OptimisticConcurrencyException) {
    Console.WriteLine("Обнаружен конфликт параллельного доступа - разрешить.");
} finally {
    string dbValue = GetStringFromDb(String.Format(
        @"select ContactName from Customers
    where CustomerID = 'LAZYK'"));
    Console.WriteLine("Значение в базе данных: {0}", dbValue);
    Console.WriteLine("Кэшированное значение: {0}", cust.ContactName);
}
```

Здесь выполняется та же последовательность запросов, что и в листинге 20.25: получается сущностный объект Customer для записи со значением CustomerID, равным LAZYK, изменяется поле ContactName за пределами Entity Framework, вносятся те же изменения в Entity Framework и затем вызывается метод SaveChanges.

Вызов SaveChanges помещен в блок try...catch...finally. Поскольку включена проверка параллельного доступа в поле ContactName, известно, что при попытке обновления базы данных будет получено исключение OptimisticConcurrencyException. В блоке finally на консоль выводится значение ContactName из базы данных и значение из сущностного объекта. Компиляция и запуск кода из листинга 20.26 даст следующий вывод:

```
Исходное значение: John Doe
Выполнение оператора SQL для базы данных с помощью ADO.NET...
База данных обновлена.
Обнаружен конфликт параллельного доступа - разрешить.
Значение в базе данных: Samuel Arthur Sanders
Кэшированное значение: John Doe
```

В конечном итоге получается база данных с одним значением и кэшированный сущностный объект, имеющий конфликтующее значение тех же данных. Это уже шаг вперед — по крайней мере, запись в базу данных не выполняется без предварительной их проверки. Но теперь необходимо решить проблему различий в значениях данных, чтобы привести их в соответствие и попытаться (дополнительно) заново обновить. Это делается с помощью метода ObjectContext.Refresh. В листинге 20.27 приведен пример.

Листинг 20.27. Использование метода Refresh

```
// Создать ObjectContext.
NorthwindEntities context = new NorthwindEntities();
Customer cust = context.Customers
    .Where(c => c.CustomerID == "LAZYK")
    .Select(c => c)
    .First();

Console.WriteLine("Исходное значение: {0}", cust.ContactName);

// Изменить запись вне Entity Framework.
ExecuteStatementInDb(String.Format(
       @"update Customers
       set ContactName = 'Samuel Arthur Sanders'
       where CustomerID = 'LAZYK'"));

// Модифицировать заказчика.
cust.ContactName = "John Doe";

// Сохранить изменения.
try {
    context.SaveChanges();
} catch (OptimisticConcurrencyException) {
    Console.WriteLine("Обнаружен конфликт параллельного доступа — обновить данные.");
    context.Refresh(RefreshMode.StoreWins, cust);
} finally {
    string dbValue = GetStringFromDb(String.Format(
       @"select ContactName from Customers
    where CustomerID = 'LAZYK'"));
    Console.WriteLine("Значение в базе данных: {0}", dbValue);
    Console.WriteLine("Кэшированное значение: {0}", cust.ContactName);
}
```

В этом примере при перехвате исключения OptimisticConcurrencyException вызывается метод Refresh. Метод Refresh принимает два аргумента. Первый — значение из перечисления RefreshMode, а второй — объект, который должен быть обновлен. Перечисление RefreshMode включает два значения — StoreWins и ClientWins. Значение StoreWins обновляет указанный объект на основе информации из базы данных. Поэтому в примере можно рассчитывать на то, что и значение в сущностном объекте, и значение в базе данных будет равно Samuel Arthur Adams. Компиляция и запуск кода дают ожидаемый результат:

```
Исходное значение: John Steel
Выполнение оператора SQL для базы данных с помощью ADO.NET...
База данных обновлена.
Обнаружен конфликт параллельного доступа — обновить данные.
Значение в базе данных: Samuel Arthur Sanders
Кэшированное значение: Samuel Arthur Sanders
```

Давайте еще раз посмотрим, что здесь происходит. Предпринимается попытка записать обновление в строку базы данных, которая была модифицирована кем-то другим. Entity Framework обнаруживает конфликт параллельного доступа и генерирует исключение OptimisticConcurrencyException, чтобы уведомить о наличии проблемы. Сущностный объект, который был модифицирован, обновляется из базы данных и приводится в согласованное состояние.

Но что происходит с обновлением? Ничего — оно не применяется. Если все-таки необходимо применить изменения, несмотря на то, что кто-то модифицировал те же самые данные, которые используются, то следует применить значение ClientWins из перечисления RefreshMode и вызвать SaveChanges заново. Пример приведен в листинге 20.28.

Листинг 20.28. Запись обновления после конфликта параллельного доступа

```
// Создать ObjectContext.
NorthwindEntities context = new NorthwindEntities();
Customer cust = context.Customers
   .Where(c => c.CustomerID == "LAZYK")
   .Select(c => c)
   .First();
Console.WriteLine("Исходное значение: {0}", cust.ContactName);
// Изменить запись вне Entity Framework.
ExecuteStatementInDb(String.Format(
   @"update Customers
   set ContactName = 'Samuel Arthur Sanders'
   where CustomerID = 'LAZYK'"));
// Модифицировать заказчика.
cust.ContactName = "John Doe";
// Сохранить изменения.
try {
   context.SaveChanges();
} catch (OptimisticConcurrencyException) {
   Console.WriteLine("Обнаружен конфликт параллельного доступа — обновить данные.");
   context.Refresh(RefreshMode.ClientWins, cust);
   context.SaveChanges();
} finally {
   string dbValue = GetStringFromDb(String.Format(
       @"select ContactName from Customers
       where CustomerID = 'LAZYK'"));
      Console.WriteLine("Значение в базе данных: {0}", dbValue);
      Console.WriteLine("Кэшированное значение: {0}", cust.ContactName);
}
```

На этот раз указано значение ClientWins, что соответствует утверждению “о конфликте параллельного доступа известно, но необходимо сохранить изменения”. Снова понадобится вызвать SaveChanges. Вызов метода Refresh просто очищает конфликт параллельного доступа для Entity Framework и не записывает изменений. Если скомпилировать и запустить код из листинга 20.28, получится следующий результат:

```
Исходное значение: John Steel
Выполнение оператора SQL для базы данных с помощью ADO.NET...
База данных обновлена.
Обнаружен конфликт параллельного доступа — обновить данные.
Значение в базе данных: John Doe
Кэшированное значение: John Doe
```

Здесь видно, что изменение, проведенное через Entity Framework, было записано в базу данных. Следует отметить один момент, касающийся правильной обработки конфликтов параллельного доступа: кто-то может снова изменить данные, пока сущностные объекты обновляются. Это значит, что второй вызов SaveChanges может привести к еще одному исключению OptimiscticConcurrencyException. Чтобы справиться с этим, можно воспользоваться циклом, который многократно пытается применить изменение. Этот подход продемонстрирован в листинге 20.29.

Листинг 20.29. Повторение запроса на сохранение

```
// Создать ObjectContext.
NorthwindEntities context = new NorthwindEntities();
Customer cust = context.Customers
   .Where(c => c.CustomerID == "LAZYK")
   .Select(c => c)
   .First();

Console.WriteLine("Исходное значение: {0}", cust.ContactName);

// Изменить запись вне Entity Framework.
ExecuteStatementInDb(String.Format(
   @"update Customers
   set ContactName = 'Samuel Arthur Sanders'
   where CustomerID = 'LAZYK'"));

// Модифицировать заказчика.
cust.ContactName = "John Doe";
int maxAttempts = 5;
bool recordsUpdated = false;

for (int i = 0; i < maxAttempts && !recordsUpdated; i++) {
   Console.WriteLine("Выполнение попытки записи {0}", i);

   // Сохранить изменения.
   try {
      context.SaveChanges();
      recordsUpdated = true;
   } catch (OptimisticConcurrencyException) {
      Console.WriteLine("Обнаружен конфликт параллельного доступа — обновить данные.");
      context.Refresh(RefreshMode.ClientWins, cust);
   }
}

string dbValue = GetStringFromDb(String.Format(
   @"select ContactName from Customers
   where CustomerID = 'LAZYK'"));
Console.WriteLine("Значение в базе данных: {0}", dbValue);
Console.WriteLine("Кэшированное значение: {0}", cust.ContactName);
```

Для реализации попыток применить изменение к базе данных несколько раз здесь используется цикл. Значение `recordsUpdated` типа `bool` будет установлено в `true` только в том случае, если метод `SaveChanges` не сгенерирует исключение. Это может быть удобным приемом, но его следует применять осторожно.

Во-первых, чем больше предпринимается попыток записать изменения, тем больше игнорируется чужих изменений. Чтобы продолжать попытки сохранить изменения, необходимо иметь абсолютную уверенность, что предпринимаемое обновление важнее всех прочих.

Во-вторых, применяется счетчик цикла для пяти попыток записи обновлений, но не более. Существует очень немного ситуаций, в которых необходимо пытаться сохранять изменения в бесконечном цикле. Следует не только иметь уверенность в том, что текущие данные являются наилучшим вариантом. Здесь также возникает вопрос о корректности проектного решения кода или ценности генерируемых данных. Если одни и те же строки будут обновляться вновь и вновь, то большинство других изменений будут просто отброшены, когда процесс применит, наконец, свои изменения к базе данных. Ввиду вышесказанного, при столкновении с конфликтом параллельного доступа следует проявлять особую осторожность, пытаясь автоматически сохранять данные.

Резюме

В этой главе вы ознакомились с основными операциями базы данных, которые можно выполнять средствами Entity Framework и LINQ to Entities. Было показано, как запрашивать данные посредством LINQ to Entities, а также вставлять, модифицировать и удалять данные с помощью Entity Framework. Также объяснялось, каким образом разрешать конфликты параллельного доступа; важность этого будет возрастать по мере роста загрузки сервера баз данных. В следующей главе речь пойдет о некоторых ключевых классах Entity Framework и о том, как добиться более полного контроля над созданием и использованием сущностных объектов.

ГЛАВА 21

Классы LINQ to Entities

В предыдущей главе множество классов Entity Framework использовалось для демонстрации средств LINQ to Entities без подробного их определения. В этой главе мы восполним этот пробел, детализируя ключевые члены наиболее важных классов Entity Framework.

Как уже объяснялось, Entity Framework — ценный и сложный набор инструментов, поэтому удастся лишь кратко коснуться его аспектов, чтобы сосредоточить внимание на стороне LINQ to Entities. Это продолжается в настоящей главе. Здесь будут выборочно описаны классы и их члены, что позволит приступить к использованию Entity Framework и LINQ to Entities.

Предварительные условия для запуска примеров

Для выполнения примеров настоящей главы понадобится получить расширенную версию базы данных Northwind и сгенерированную для нее сущностную модель данных. Прочтите инструкции в разделе "Предварительные условия для запуска примеров" главы 19. Вдобавок для запуска примеров этой главы потребуются общие методы, которые будут использоваться в примерах. Ознакомьтесь с инструкциями, которые даны в разделе "Некоторые общие методы" главы 19.

Внимание! В некоторых примерах этой главы база данных модифицируется (и даже удаляется). После запуска этих примеров понадобится отключить базу данных Northwind от SQL Server и присоединить исходную версию, загруженную вместе с исходным кодом примеров.

Класс `ObjectContext`

Класс `ObjectContext` лежит в основе платформы Entity Framework и LINQ to Entities. Когда создается сущностная модель данных, то создается класс, производный от `ObjectContext`, со свойствами, представляющими сущностные типы и коллекции, специфичные для базы данных. В сущностной модели данных Northwind, созданной в главе 19, такой производный класс назывался `NorthwindEntities`. Этот класс использовался во всех примерах LINQ to Entities для обработки соединения с базой данных, загрузки сущностных объектов и сохранения изменений после выполнения модификаций. И, поскольку он производный, класс `NorthwindEntities` имеет общедоступные члены — коллекции сущностных объектов, представляющих содержимое базы данных Northwind. Класс `ObjectContext` определен в пространстве имен `System.Data.Objects`.

Конструктор

Прежде чем можно будет использовать какие-либо средства Entity Framework или LINQ to Entities, описанные в предыдущих главах, потребуется создать новый экземпляр производного класса `ObjectContext`. Созданный экземпляр отвечает за управление соединением с базой данных и используется для загрузки и сохранения данных в базе.

Прототипы

Существуют два прототипа конструктора `ObjectContext`, которые описаны ниже.

Первый унаследованный от `ObjectContext` конструктор

```
public NortwindEntities();
```

Это конструктор по умолчанию, создающий новый экземпляр производного класса `ObjectContext` с использованием строки подключения к базе данных, которая была добавлена в файл `App.Config` при создании сущностной модели данных. Это прототип, который применялся в примерах на протяжении всех глав, посвященных LINQ to Entities.

Второй унаследованный от `ObjectContext` конструктор

```
public NortwindEntities(string connectionString);
```

Этот прототип конструктора позволяет указать детали, которые будут использованы для подключения к базе данных, либо имя строки подключения из файла `App.config`.

Примеры

В коде, приведенном в листинге 21.1, для создания нового экземпляра производного от `ObjectContext` типа используется прототип конструктора по умолчанию.

Листинг 21.1. Использование конструктора по умолчанию, унаследованного от `ObjectContext`

```
NorthwindEntities context = new NorthwindEntities();

IQueryable<Customer> custs = context.Customers
   .Where(c => c.City == "London")
   .Select(c => c);

foreach (Customer cust in custs) {
   Console.WriteLine("Имя заказчика: {0}", cust.CompanyName);
}
```

В листинге 21.2 используется второй прототип конструктора, которому передается имя строки подключения в файле `App.Config`. Результирующий экземпляр, производный от `ObjectContext`, затем применяется для запроса к базе данных.

Листинг 21.2. Указание имени для свойства строки подключения

```
string connectionString = "name=NorthwindEntities";
NorthwindEntities context = new NorthwindEntities(connectionString);

IQueryable<Customer> custs = context.Customers
   .Where(c => c.City == "London")
   .Select(c => c);

foreach (Customer cust in custs) {
   Console.WriteLine("Имя заказчика: {0}", cust.CompanyName);
}
```

Можно также воспользоваться вторым прототипом конструктора, чтобы указать полностью специальную строку соединения. Самый простой путь сделать это — с помощью классов EntityConnectionStringBuilder и SqlConnectionStringBuilder, которые находятся в пространствах имен System.Data.EntityClient и System.Data.SqlClient. В листинге 21.3 демонстрируется использование этих классов для создания некоторой строки подключения, помещаемой в файл App.Config при создании сущностной модели данных в главе 19.

Листинг 21.3. Создание и использование специальной строки подключения

```
SqlConnectionStringBuilder scsb = new SqlConnectionStringBuilder();
scsb.DataSource = @".\sqlexpress";
scsb.InitialCatalog = "Northwind";
scsb.IntegratedSecurity = true;
scsb.MultipleActiveResultSets = true;
EntityConnectionStringBuilder ecsb = new EntityConnectionStringBuilder();
ecsb.Provider = "System.Data.SqlClient";
ecsb.ProviderConnectionString = scsb.ToString();
ecsb.Metadata = @"res://*/NorthwindEntityModel.csdl|
   res://*/NorthwindEntityModel.ssdl
   |res://*/NorthwindEntityModel.msl";

NorthwindEntities context = new NorthwindEntities(ecsb.ToString());

IQueryable<Customer> custs = context.Customers
   .Where(c => c.City == "London")
   .Select(c => c);

foreach (Customer cust in custs) {
   Console.WriteLine("Имя заказчика: {0}", cust.CompanyName);
}
```

DatabaseExists()

Метод DatabaseExists возвращает true, если база данных, которая указана в строке подключения, использованной для создания ObjectContext, существует, и false — в противном случае.

Прототипы

Метод DatabaseExists имеет один прототип, описанный ниже.

Единственный прототип DatabaseExists

```
public bool DatabaseExists();
```

Метод DatabaseExists возвращает true, если база данных, которая указана в строке соединения, использованной для создания ObjectContext, существует, и false — в противном случае.

Примеры

В листинге 21.4 создается экземпляр производного от ObjectContext класса для базы данных Northwind и используется метод DatabaseExists.

Листинг 21.4. Использование метода DatabaseExists

```
NorthwindEntities context = new NorthwindEntities();
bool databaseExists = context.DatabaseExists();
Console.WriteLine("Признак существования базы данных: {0}", databaseExists);
```

DeleteDatabase()

Метод DeleteDatabase удаляет базу данных, которая указана в строке соединения, использованной для создания производного от ObjectContext экземпляра. Этот метод обычно применяется в сочетании с методом DatabaseExists.

Прототипы

Метод DeleteDatabase имеет один прототип.

```
public void DeleteDatabase();
```

Примеры

В листинге 21.5 метод DeleteDatabase используется для удаления базы данных Northwind.

Внимание! После компиляции и запуска примера из листинга 21.5 понадобится восстановить исходную версию базы данных Northwind.

Листинг 21.5. Удаление базы данных

```
NorthwindEntities context = new NorthwindEntities();
if (context.DatabaseExists()) {
   context.DeleteDatabase();
}
```

CreateDatabase()

Метод CreateDatabase использует сущностную модель данных для создания новой базы данных, хотя, конечно, в ней не будет никаких данных. Этот метод обычно применяется в сочетании с методом DatabaseExists.

Прототипы

Метод CreateDatabase имеет один прототип.

```
public void CreateDatabase();
```

Примеры

В листинге 21.6 создается новая база данных с использованием сущностной модели базы данных Northwind.

Листинг 21.6. Создание новой базы данных

```
NorthwindEntities context = new NorthwindEntities();
if (!context.DatabaseExists()) {
   context.CreateDatabase();
}
```

SaveChanges()

Метод SaveChanges сохраняет модификации, проведенные в сущностных объектах базы данных. Этот метод генерирует исключение OptomisticConcurencyException, если включена проверка параллельного доступа и возникает конфликт обновления. Подробности управления параллельным доступом изложены в главе 20.

Прототипы

Метод SaveChanges имеет один прототип, описанный ниже.

Прототип SaveChanges

```
public int SaveChanges();
```

Возвращаемое значение указывает количество добавленных, обновленных или удаленных сущностных объектов.

Примеры

Код в листинге 21.7 модифицирует поле ContactName сущностного типа Customer и вызывает метод SaveChanges для сохранения модификаций в базе данных.

Листинг 21.7. Использование метода SaveChanges

```
NorthwindEntities context = new NorthwindEntities();
Customer cust = (from c in context.Customers
   where c.CustomerID == "LAZYK"
   select c).First();
cust.ContactName = "John Doe";
int modificationCount = context.SaveChanges();
Console.WriteLine("Количество: {0}", modificationCount);
```

Refresh()

Для повышения производительности сущностные объекты в Entity Framework кэшируются. Это значит, что данные, с которыми ведется работа, могут устареть, когда другие пользователи и процессы обновляют базу данных. Модификация устаревших данных и попытка записать их в базу приводит к конфликту параллельного доступа.

Метод Refresh() имеет два предназначения. Он может использоваться для предварительного обновления одного или более сущностных объектов по последней информации из базы данных, а также для разрешения конфликтов параллельного доступа в случае их возникновения.

Прототипы

Метод Refresh имеет два прототипа. Оба используют перечисление RefreshMode, имеющее два значения: StoreWins и ClientWins. При предварительном обновлении данных должно использоваться значение StoreWins, потому что оно указывает на то, что изменения, проведенные в сущностных объектах, должны быть отброшены в пользу изменений, внесенных в базу данных. При разрешении конфликтов параллельного доступа можно использовать любое из значений перечисления; подробности и примеры ищите в главе 20.

Первый прототип Refresh

```
public void Refresh(RefreshMode refreshMode,
   Object entity);
```

Этот прототип обновляет одиночный сущностный объект в режиме RefreshMode.

Второй прототип Refresh

```
public void Refresh(RefreshMode refreshMode,
   IEnumerable collection);
```

Этот прототип обновляет коллекцию сущностных объектов в режиме RefreshMode. Прототип может использоваться для обновления одной из коллекций сущностных объектов в производном от ObjectContext классе, который представляет таблицу.

Примеры

В коде, приведенном в листинге 21.8, предварительно обновляются единственный сущностный объект `Customer` и коллекция сущностных объектов `Customer` в свойстве `context.Customers`.

Листинг 21.8. Использование метода `Refresh`

```
NorthwindEntities context = new NorthwindEntities();
Customer cust = (from c in context.Customers
                 where c.CustomerID == "LAZYK"
                 select c).First();
// Обновить единственный сущностный объект.
context.Refresh(RefreshMode.StoreWins, cust);

// Обновить всю коллекцию сущностных объектов.
context.Refresh(RefreshMode.StoreWins, context.Customers);
```

AddObject()

Метод `AddObject` добавляет новый сущностный объект к одной из коллекций, поддерживаемой классом-наследником `ObjectContext`.

Прототипы

У метода `AddObject` один прототип.

```
public void AddObject(
   string entitySetName,
   Object entity);
```

Первый аргумент — имя коллекции, в которую объект должен быть добавлен. Второй аргумент — сущностный объект, который вы хотите добавить. Заметьте, что база данных не обновляется данными, содержащимися в новом сущностном объекте, пока не будет вызван метод `SaveChanges`.

Примеры

В коде, показанном в листинге 21.9, создается новый экземпляр сущностного типа `Customer`, а также устанавливаются значения полей. Метод `AddObject` используется для добавления сущностного объекта в коллекцию `Customers`. Наконец, метод `SaveChanges` применяется для записи новой записи `Customer` в базу данных.

Листинг 21.9. Добавление сущностного объекта к `ObjectContext`

```
NorthwindEntities context = new NorthwindEntities();

// Создать новый объект Customer.
Customer cust = Customer.CreateCustomer("LAWN", "Lawn Wranglers");

// Заполнить поля, допускающие null.
cust.ContactName = "Mr. Abe Henry";
cust.ContactTitle = "Owner";
cust.Address = "1017 Maple Leaf Way";
cust.City = "Ft. Worth";
cust.Region = "TX";
cust.PostalCode = "76104";
cust.Country = "USA";
cust.Phone = "(800) MOW-LAWN";
cust.Fax = "(800) MOW-LAWO";
context.AddObject("Customers", cust);
context.SaveChanges();
```

CreateObject()

Метод CreateObject создает новый сущностный объект. Новый объект не содержит данных и должен быть добавлен к одной из коллекций сущностных объектов в ObjectContext. Только после этого метод SaveChanges сохранит объект в базе данных. Если определение схемы базы данных требует, чтобы некоторые поля не содержали null, необходимо обеспечить наполнение сущностного объекта перед вызовом SaveChanges.

Прототипы

Метод CreateObject строго типизирован, в том смысле, что для каждого сущностного типа, поддерживаемого классом-наследником ObjectContext, существует один прототип в следующей форме, где T — сущностный тип, экземпляр которого необходимо создать:

```
public T CreateObject<T>();
```

В случае базы данных Northwind это означает, что для сущностного типа Customer есть такой прототип:

```
public Customer CreateObject<Customer>();
```

Для сущностного типа Order прототип выглядит следующим образом:

```
public Order CreateObject<Order>();
```

Прототипы для каждого из сущностных типов, содержащихся в сущностной модели данных, похожи на показанные выше.

Примеры

В коде, приведенном в листинге 21.10, метод CreateObject используется для создания нового сущностного объекта Customer. Поля заполняются (включая поля CustomerID и CompanyName, которые не допускают null), и объект добавляется в коллекцию Customers с помощью метода AddObject. Новый Customer сохраняется в базе данных вызовом метода SaveChanges.

Листинг 21.10. Создание экземпляра сущностного типа с использованием ObjectContext

```
NorthwindEntities context = new NorthwindEntities();

// Создать новый объект Customer.
Customer cust = context.CreateObject<Customer>();

// Заполнить все поля.
cust.CustomerID = "LAWN";
cust.CompanyName = "Lawn Wranglers";
cust.ContactName = "Mr. Abe Henry";
cust.ContactTitle = "Owner";
cust.Address = "1017 Maple Leaf Way";
cust.City = "Ft. Worth";
cust.Region = "TX";
cust.PostalCode = "76104";
cust.Country = "USA";
cust.Phone = "(800) MOW-LAWN";
cust.Fax = "(800) MOW-LAWO";
context.AddObject("Customers", cust);
context.SaveChanges();
```

DeleteObject()

Метод DeleteObject удаляет объект из кэша сущностей и удаляет соответствующие данные из базы при вызове метода SaveChanges. При удалении объектов, у которых есть связанные с ними объекты, следует соблюдать осторожность; подробнее об этом читайте в главе 20.

Прототипы

Метод DeleteObject имеет один прототип:

```
public void DeleteObject(Object entity);
```

Аргумент в этом прототипе — это сущностный объект, который должен быть удален.

Примеры

Код листинга 21.11 запрашивает сущностные объекты Order_Detail, имеющие OrderID, равный 10248. Результаты перечисляются в цикле foreach и удаляются с помощью метода DeleteObject. Чтобы сохранить удаление, вызывается метод SaveChanges.

Листинг 21.11. Удаление сущностных объектов

```
// Создать ObjectContext.
NorthwindEntities context = new NorthwindEntities();

// Получить детали заказа 10248.
IQueryable<Order_Detail> ods = (from o in context.Order_Details
                           where o.OrderID == 10248
                           select o);

foreach (Order_Detail od in ods) {
   context.DeleteObject(od);
}

// Сохранить изменения.
context.SaveChanges();
```

EntityObject

Сущностные типы, создаваемые в сущностной модели данных для представления схемы базы данных, наследуются от класса EntityObject, который является частью пространства имен System.Data.Objects.DataClasses.

Конструктор

Создать новый экземпляр сущностного типа можно с использованием конструктора, но при этом нужно обеспечить заполнение полей, отображаемых на столбцы базы данных, которые не допускают хранения значений null. Если этого не сделать, то при попытке сохранить новый сущностный объект в базе данных будет сгенерировано исключение.

Прототипы

Существует только один прототип конструктора, где T — сущностный тип:

```
public T();
```

Примеры

В листинге 21.12 демонстрируется создание нового экземпляра сущностного типа `Customer`, заполнение полей данных и сохранение его в базе данных посредством метода `SaveChanges` класса `ObjectContext`.

Листинг 21.12. Создание нового сущностного объекта с использованием конструктора по умолчанию

```
NorthwindEntities context = new NorthwindEntities();

// Создать новый объект Customer.
Customer cust = new Customer();

// Заполнить все поля.
cust.CustomerID = "LAWN";
cust.CompanyName = "Lawn Wranglers";
cust.ContactName = "Mr. Abe Henry";
cust.ContactTitle = "Owner";
cust.Address = "1017 Maple Leaf Way";
cust.City = "Ft. Worth";
cust.Region = "TX";
cust.PostalCode = "76104";
cust.Country = "USA";
cust.Phone = "(800) MOW-LAWN";
cust.Fax = "(800) MOW-LAWO";
context.AddObject("Customers", cust);
context.SaveChanges();
```

Фабричный метод

Статический фабричный метод добавляется к сущностным типам, когда они создаются мастером Entity Data Model Wizard. (За более подробной информацией относительно применения этого мастера с базой данных Northwind обращайтесь в главу 19.) Фабричный метод может использоваться для создания новых экземпляров сущностного типа и имеет преимущество перед конструктором по умолчанию, требуя значений для всех полей с обязательными значениями в базе данных. Это позволяет изящно избежать проблемы создания нового экземпляра, которому не хватает одного из таких значений. Проблема связана с генерацией исключения при попытке сохранить экземпляр в базе данных методом `SaveChanges`. Как и с конструктором по умолчанию, сущностные объекты, создаваемые фабричными методами, не сохраняются до тех пор, пока они не будут добавлены к одной из коллекций сущностных типов, поддерживаемых производным от `ObjectContext` классом (для примера просмотрите метод `AddObject` класса `ObjectContext`).

Прототипы

Прототип для фабричного метода варьируется в зависимости от сущностного типа, однако следует общему шаблону. Если сущностный тип представляет строку из базы данных, в которой все значения могут быть установлены в `null`, то у фабричного метода не будет аргументов, и прототип будет выглядеть так, как показано ниже.

Фабричный метод сущностного типа по умолчанию

```
public static T CreateT();
```

Поэтому, например, если есть сущностный тип `MyType`, представляющий данные из таблицы, в которой все столбцы допускают хранение `null`, то прототип для фабричного метода будет таким:

```
public static MyType CreateMyType();
```

Если сущностный тип представляет строки таблицы, содержащей столбцы, которые не могут иметь значения `null`, то у него будет предусмотрено по аргументу для каждого обязательного значения данных. Например, если посмотреть на таблицу `Customers` базы данных Northwind в SQL Server Management Studio (рис. 21.1), то можно заметить, что столбцы `CustomerID` и `CompanyName` не имеют отмеченных флажков Allow Nulls (Допускает значения `null`).

SHUTTLE\SQLEXPR...- dbo.Customers

Column Name	Data Type	Allow Nulls
CustomerID	nchar(5)	☐
CompanyName	nvarchar(40)	☐
ContactName	nvarchar(30)	☑
ContactTitle	nvarchar(30)	☑
Address	nvarchar(60)	☑
City	nvarchar(15)	☑
Region	nvarchar(15)	☑
PostalCode	nvarchar(10)	☑
Country	nvarchar(15)	☑
Phone	nvarchar(24)	☑
Fax	nvarchar(24)	☑
		☐

Рис. 21.1. Таблица `Customers` базы данных Northwind

Это поля данных для сущностного типа `Customer`, которые потребуются в качестве аргументов статического фабричного метода, имеющего показанный ниже прототип.

Прототип фабричного метода сущностного типа `Customer` базы Northwind

```
public static Customer CreateCustomer(String customerID, String companyName);
```

Простейший способ работы с прототипом фабричного метода предусматривает использование средства IntelliSense среды Visual Studio или даже обращение к исходному коду сущностного типа.

Примеры

В коде, приведенном в листинге 21.13, создается новый экземпляр сущностного типа `Customer` с помощью фабричного метода, которому передаются значения для обязательных полей данных. Затем устанавливаются остальные поля данных (хотя для них можно оставить значения по умолчанию). Новый объект затем добавляется к коллекции сущностей `Customers`, поддерживаемой производным классом `ObjectContext` и сохраняемой вызовом `SaveChanges`.

Листинг 21.13. Использование фабричного метода для создания нового сущностного объекта

```
NorthwindEntities context = new NorthwindEntities();
// Создать новый объект Customer.
Customer cust = Customer.CreateCustomer("LAWN", "Lawn Wranglers");
// Заполнить остальные поля.
cust.ContactName = "Mr. Abe Henry";
cust.ContactTitle = "Owner";
cust.Address = "1017 Maple Leaf Way";
cust.City = "Ft. Worth";
```

```
cust.Region = "TX";
cust.PostalCode = "76104";
cust.Country = "USA";
cust.Phone = "(800) MOW-LAWN";
cust.Fax = "(800) MOW-LAWO";
context.AddObject("Customers", cust);
context.SaveChanges();
```

Примитивные свойства

Каждый сущностный тип имеет набор общедоступных свойств, соответствующих столбцам таблицы, с которой он ассоциирован. Эти свойства позволяют получать и устанавливать значения данных для новой строки таблицы, представленной определенным экземпляром сущностного типа.

Прототипы

Набор свойств, которые имеет сущностный тип, зависит от проектного решения таблицы базы данных, с которой он ассоциирован. Общий прототип показан ниже, где `T` — тип данных, а `ColumnName` — имя поля данных.

Общий прототип примитивного свойства сущностного типа

```
public T ColumnName {get; set};
```

От общего прототипа толку мало. Намного полезнее взглянуть на действительную реализацию. На рис. 21.1 были показаны столбцы таблицы `Customer` в базе данных Northwind, строки которой представлены сущностным типом `Customer` в сущностной модели данных, созданной в главе 19. Для каждого из столбцов, показанных на рисунке, будет найдено общедоступное свойство, позволяющее получать и устанавливать ассоциированные значения данных. Например, столбец `City` будет иметь следующий прототип.

Прототип примитивного свойства `City` класса `Customer`

```
public String City {get; set};
```

Примеры

В большинстве примеров, приведенных в главах по LINQ to Entities, используются примитивные свойства в той или иной форме. Код в листинге 21.14 читает значение свойства `City` из сущностного объекта `Customer`, модифицирует значение и сохраняет изменение в базе данных, используя метод `SaveChanges`. Изменения значений свойств не записываются в базу данных до вызова метода `SaveChanges`.

Листинг 21.14. Чтение и изменение значения примитивного свойства

```
NorthwindEntities context = new NorthwindEntities();

// Запросить запись Customer.
Customer cust = (from c in context.Customers
                 where c.CustomerID == "LAZYK"
                 select c).First();

// Получить доступ к текущему значению данных.
Console.WriteLine("Исходное значение City: {0}", cust.City);

// Изменить значение.
cust.City = "Seattle";

// Вывести на консоль новое значение (но не постоянное значение).
Console.WriteLine("Новое значение City: {0}", cust.City);

// Сохранить изменения.
context.SaveChanges();
```

После компиляции и запуска кода листинга 21.14 получается следующий результат:

```
Исходное значение City: Walla Walla
Новое значение City: Seattle
```

Навигационные свойства

Навигационные свойства позволяют легко работать со связанными сущностными объектами, особенно когда объекты связаны по внешнему ключу.

Предположим, что необходимо найти набор заказов, которые заказчик разместил в базе данных Northwind, и единственные данные, от которых можно оттолкнуться — это название компании. Без навигационных свойств пришлось бы выполнять два запроса LINQ — один для нахождения сущностного объекта `Customer`, а другой для получения всех сущностных объектов `Order`, связанных по внешнему ключу с найденным ранее `Customer`. В листинге 21.15 показано, как это могло бы работать.

Листинг 21.15. Запрос без навигационных свойств

```
NorthwindEntities context = new NorthwindEntities();

// Запросить запись для заказчика.
Customer cust = (from c in context.Customers
                 where c.CompanyName == "Lazy K Kountry Store"
                 select c).First();

// Запросить заказы, размещенные компанией.
IQueryable<Order> orders = from o in context.Orders
                           where o.CustomerID == cust.CustomerID
                           select o;

// Вывести заказы на консоль.
foreach (Order ord in orders) {
   Console.WriteLine("Идентификатор заказа: {0}, дата: {1}", ord.OrderID, ord.OrderDate);
}
```

Компиляция и запуск кода из листинга 21.15 даст следующий результат:

```
Идентификатор заказа: 10482, дата: 21/03/1997 00:00:00
Идентификатор заказа: 10545, дата: 22/05/1997 00:00:00
```

Был получен нужный результат, но за счет использования навигационных свойств можно избежать явного выполнения второго запроса.

Прототипы

Для каждого отношения внешнего ключа в базе данных существует пара навигационных свойств в сущностной модели данных — по одному для каждого затронутого сущностного типа. Прототип для свойства зависит от множественности отношения. Если сущностный тип может быть связан с множеством экземпляров другого сущностного типа (как `Customer` из базы Northwind может быть связан с множеством экземпляров сущностного типа `Order`), то прототип будет выглядеть, как показано ниже, где `T` — связанный сущностный тип, а `TableNameOfT` — имя таблицы базы данных, чьи записи представляет `T`.

Прототип навигационного свойства множественного отношения

```
public EntityCollection<T> TableNameOfT {get; set};
```

Описание прототипов проще разобрать на примере. В базе данных Northwind таблицы `Customers` и `Orders` разделяют отношение внешнего ключа, т.е. множество

строк в Orders могут иметь внешний ключ, указывающий на одну строку в Customers. Строки таблицы Customers представлены сущностным типом Customer, а строки таблицы Orders — сущностным типом Order. Все это значит, что в типе Customer должно присутствовать навигационное свойство с показанным ниже прототипом. Класс EntityCollection рассматривается далее в этой главе.

Прототип навигационного свойства отношения Customer.Orders

```
public EntityCollection<Order> Orders {get; set};
```

Если в отношении может быть не более одного связанного объекта, то прототип выглядит, как показано ниже, при этом T — это сущностный тип.

Прототип навигационного свойства одиночного отношения

```
public EntityCollection<T> TReference {get; set};
```

В случае сущностного типа Order базы данных Northwind может быть только один связанный объект Customer, поэтому прототип будет выглядеть так, как показано ниже.

Прототип навигационного свойства отношения Order.Customer

```
public EntityReference<Customer> CustomerReference {get; set};
```

Мастер Entity Data Model Wizard также создает встречное свойство для отношений подобного рода. Прототип выглядит, как показано ниже, при этом T — это связанный сущностный тип.

Прототип встречного навигационного свойства одиночного отношения

```
public Customer Customer {get; set};
```

Примеры

В листинге 21.16 показано, как использовать навигационное свойство Orders в сущностном типе Customer базы данных Northwind для получения всех заказов определенного заказчика. Здесь решается та же самая задача, что и в листинге 21.15, но без необходимости в явном втором запросе. Под “явным” подразумевается, что данные все равно будут получены из базы данных, но навигационное свойство облегчает задачу кодирования (подробные сведения о различных способах влияния на загрузку данных из базы можно найти в главе 20).

Листинг 21.16. Использование навигационного свойства “один ко многим”

```
NorthwindEntities context = new NorthwindEntities();

// Запросить запись заказчика.
Customer cust = (from c in context.Customers
                 where c.CompanyName == "Lazy K Kountry Store"
                 select c).First();
EntityCollection<Order> orders = cust.Orders;
foreach (Order ord in orders) {
   Console.WriteLine("Идентификатор заказа: {0}, дата: {1}",
       ord.OrderID, ord.OrderDate);
}
```

В этом листинге явно использовался класс EntityCollection, но если посмотреть на ряд других примеров в главах, посвященных LINQ to Entities, можно заметить, что повсюду свободно применялись навигационные свойства, но класс явно не объявлялся. После компиляции и запуска кода из листинга 21.16 будет получен следующий результат, который полностью совпадает с выводом кода из листинга 21.15:

```
Идентификатор заказа: 10482, дата: 21/03/1997 00:00:00
Идентификатор заказа: 10545, дата: 22/05/1997 00:00:00
```

В листинге 21.17 демонстрируется использование навигационного свойства одиночного экземпляра между типом Order и соответствующим ему Customer.

Листинг 21.17. Использование навигационного свойства одиночного экземпляра

```
NorthwindEntities context = new NorthwindEntities();

// Запросить заказ.
Order ord = (from o in context.Orders
             where o.CustomerID == "LAZYK"
             select o).First();

// Получить ссылку на сущность.
EntityReference<Customer> customerRef = ord.CustomerReference;
Console.WriteLine("Имя заказчика: {0}", customerRef.Value.CompanyName);

// Получить заказчика через встречное свойство.
Customer cust = ord.Customer;
Console.WriteLine("Имя заказчика: {0}", cust.CompanyName);
```

Экземпляр EntityReference<Customer> получается через свойство CustomerReference. Для получения типа Customer от EntityReference следует обратиться к свойству Value. Класс EntityReference рассматривается далее в этой главе.

Обращаться к сущностному типу более удобно непосредственно через свойство Customer. В листинге 21.17 продемонстрированы оба подхода с выводом в обоих случаях на консоль значения свойства CompanyName:

```
Имя заказчика: Lazy K Kountry Store
Имя заказчика: Lazy K Kountry Store
```

EntityReference

Класс EntityReference используется для манипуляций навигационными свойствами одиночного экземпляра между сущностными типами; подробности устройства навигационных свойств были описаны в предыдущем разделе. Это не тот класс, с которым придется работать очень часто. Обычно проще использовать встречное свойство, создаваемое при генерации сущностного типа мастером Entity Data Model Wizard. Для полноты картины ниже приводятся описания ключевых членов.

Load()

Метод Load используется при явной загрузке данных, которая была описана в главе 20.

Примеры

В листинге 21.18 демонстрируется явная загрузка сущностного объекта, ассоциированного с EntityReference. Обратите внимание, что для того, чтобы этот метод возымел действие, ленивая загрузка должна быть отключена; подробности и примеры см. в главе 20.

Листинг 21.18. Использование метода Load класса EntityReference

```
NorthwindEntities context = new NorthwindEntities();

// Отключить ленивую загрузку.
context.ContextOptions.LazyLoadingEnabled = false;
```

```
// Запросить заказ.
Order ord = (from o in context.Orders
             where o.CustomerID == "LAZYK"
             select o).First();

// Получить ссылку на сущность.
EntityReference<Customer> customerRef = ord.CustomerReference;

// Явно загрузить заказ.
customerRef.Load();
```

Value

Свойство `Value` возвращает лежащий в основе сущностный тип, на который полагается `EntityReference`. В листинге 21.17 свойство `Value` вызывалось для получения сущностного объекта `Customer`, связанного с сущностным объектом `Order`, с которым выполнялась работа.

EntityCollection

Класс `EntityCollection` предназначен для хранения коллекций сущностных объектов, чаще всего на одном конце навигационного свойства. Например, в сущностном типе `Customer` базы данных Northwind свойство `Orders` — это `EntityCollection<Order>`, и оно используется для содержания объектов `Order`, относящихся к данному объекту `Customer`.

Наиболее распространенные способы использования `EntityCollection` заключаются в перечислении элементов в коллекции с помощью цикла `foreach` как основы запроса LINQ. Класс `EntityCollection` реализует интерфейсы, допускающие перечисление — `IEnumerable<T>` и `IEnumerable`, где `T` является сущностным типом, экземпляры которого нужно собирать. Примеры обоих подходов приведены в главах, посвященных LINQ to Entities. Класс `EntityCollection` реализует некоторые другие полезные методы, описанные ниже.

Add()

Добавление сущностного типа в `EntityCollection` устанавливает с ним отношение внешнего ключа и делает его связанным объектом. Entity Framewrok самостоятельно установит значения полей внешнего ключа. Если это новый объект, то при вызове `SaveChanges` будет создана строка в базе данных. Если есть готовый сущностный объект, то при вызове `SaveChanges` отношение внешнего ключа будет обновлено.

Прототипы

Метод `Add` имеет один прототип, представленный ниже.

Прототип метода Add

```
public void Add(Object entity);
```

Эффект от вызова метода `Add` немедленно отражается в кэшированных данных, поддерживаемых Entity Framework, но не отражается в базе данных до тех пор, пока не будет вызван метод `SaveChanges` в производном от `ObjectContext` классе. Подробные сведения о сохранении изменений даны в главе 20, а в этой главе выше приведена дополнительная информация о классе `ObjectContext`.

Примеры

В листинге 21.19 создается новый сущностный тип `Order` и вызывается метод `Add` на `Customer.Orders.EntityCollection` для связывания `Order` с `Customer`.

Листинг 21.19. Использование метода Add для связывания объектов

```
NorthwindEntities context = new NorthwindEntities();

// Запросить запись заказчика.
Customer cust = (from c in context.Customers
                 where c.CompanyName == "Lazy K Kountry Store"
                 select c).First();

Order ord = Order.CreateOrder(1234);
cust.Orders.Add(ord);
Console.WriteLine("CustomerID для заказа: {0}", ord.CustomerID);
```

После компиляции и запуска кода из листинга 21.19 получится следующий результат, который демонстрирует, что между объектами `Customer` и `Order` установлено отношение внешнего ключа.

```
CustomerID для заказа: LAZYK
```

Обратите внимание, что метод `SaveChanges` не вызывался, а это значит, что модифицированы только кэшированные данные в Entity Framewrok, но никакие изменения в базу данных не вносились. Если вызвать `Refresh` для обновления кэшированных данных, то изменения будут утеряны; подробные сведения о методе `Refresh` ищите в разделе “Класс `ObjectContext`” ранее в главе.

С помощью метода `Add` можно изменить отношение между сущностными объектами. В листинге 21.20 содержится пример.

Листинг 21.20. Использование метода `Add` для изменения отношения внешнего ключа

```
NorthwindEntities context = new NorthwindEntities();

// Получить заказчика LAZYK.
Customer cust1 = (from c in context.Customers
                  where c.CustomerID == "LAZYK"
                  select c).First();

// Получить заказчика AROUT.
Customer cust2 = (from c in context.Customers
                  where c.CustomerID == "AROUT"
                  select c).First();

// Получить первый заказ для LAZYK.
Order firstOrder = cust1.Orders.First();
Console.WriteLine("Первый заказ для LAZYK, идентификатор заказчика: {0},
  идентификатор заказа: {1}",
  firstOrder.CustomerID, firstOrder.OrderID);

// Добавить заказ LAZYK к набору заказов AROUT.
cust2.Orders.Add(firstOrder);
Console.WriteLine("Первый заказ для LAZYK, идентификатор заказчика: {0},
  идентификатор заказа: {1}",
  firstOrder.CustomerID, firstOrder.OrderID);
```

В этом примере запрашиваются два заказчика со значениями `CustomerID`, равными `LAZYK` и `AROUT`. Затем извлекается первый заказ для заказчика `LAZYK` и вызывается `Add` для добавления в `EntityCollection` объектов `Order` заказчика `AROUT`. На консоль вы-

водятся значения полей CustomerID и OrderID объекта Order до и после вызова метода Add. Если скомпилировать и запустить код из листинга 21.20, получится следующий результат:

```
Первый заказ для LAZYK, идентификатор заказчика: LAZYK, идентификатор заказа: 10482
Первый заказ для LAZYK, идентификатор заказчика: AROUT, идентификатор заказа: 10482
```

Как видите, Entity Framework интеллектуально обновляет отношение внешнего ключа, так что объект Order теперь связан с объектом Customer, имеющим идентификатор AROUT.

Remove()

Метод Remove() удаляет сущностный объект из коллекции и устанавливает поле внешнего ключа в null. Это значит, что объект не появится при перечислении EntityCollection и когда будет вызван метод SaveChanges. Строка в базе данных, соответствующая удаленному объекту, будет обновлена значением внешнего ключа NULL.

Прототипы

Метод Remove имеет один прототип, описанный ниже.

Прототип Remove

```
public bool Remove(Object entity);
```

Аргумент entity — это сущностный объект, который должен быть удален из коллекции. Метод Remove возвращает true, если сущностный объект был удален успешно, и false — в противном случае. Вызов метода Remove оказывает немедленный эффект на сущностные объекты, кэшированные Entity Framework, но не затрагивает базу данных, пока не будет вызван метод SaveChanges.

Если схема базы данных не допускает значений NULL в столбце внешнего ключа, то при использовании метода Remove следует соблюдать осторожность. В этом случае вызов метода SaveChanges приведет к генерации исключения.

Примеры

В листинге 21.21 демонстрируется применение метода Remove для удаления объекта Order из коллекции EntityCollection<Order> объекта Customer в сущностной модели базы данных Northwind.

Листинг 21.21. Использование метода Remove для разрыва отношения внешнего ключа

```
NorthwindEntities context = new NorthwindEntities();

// Получить заказчика LAZYK.
Customer cust = (from c in context.Customers
                 where c.CustomerID == "LAZYK"
                 select c).First();

// Получить первый заказ заказчика LAZYK.
Order order = cust.Orders.First();
Console.WriteLine("Заказ имеет следующий CustomerID: {0}", order.CustomerID);

// Удалить заказ из коллекции.
Console.WriteLine("Удаление заказа с идентификатором: {0}", order.OrderID);
cust.Orders.Remove(order);
Console.WriteLine("Заказ имеет следующий CustomerID: {0}", (order.CustomerID == null?
   "NULL" : order.CustomerID));

// Сохранить изменения.
context.SaveChanges();
```

В коде выполняется запрос LINQ to Entities для получения сущностного объекта Customer и извлечения первого экземпляра Order из коллекции, который передается в качестве аргумента методу Remove. Перед и после вызова метода Remove значение CustomerID объекта Order выводится на консоль, чтобы можно было видеть значение поля внешнего ключа. Ниже показан результат запуска кода:

```
Заказ имеет следующий CustomerID: LAZYK
Удаление заказа с идентификатором: 10482
Заказ имеет следующий CustomerID: NULL
```

Как видите, значение CustomerID объекта Order установлено в null. В рассмотренном примере вызывается метод SaveChanges для сохранения изменений, что приводит к обновлению строки таблицы, представляющей Order, значением поля CustomerID, равным NULL. Никакие исключения при сохранении данных не генерируются, потому что внешний ключ не навязывается схемой базы данных, но появляется "висячая" запись, которая более не ассоциирована с заказчиком.

Clear()

Метод Clear удаляет все сущностные объекты из EntityCollection и устанавливает их значения внешних ключей в null. Это эквивалентно удалению каждого объекта коллекции индивидуально с помощью метода Remove.

Прототипы

Метод Clear имеет один прототип.

Прототип метода Clear

```
public void Clear();
```

Примеры

В листинге 21.22 удаляются все объекты Order из EntityCollection<Order> для заданного объекта Customer из базы данных Northwind.

Листинг 21.22. Использование метода Clear для удаления сущностных объектов из коллекции

```
NorthwindEntities context = new NorthwindEntities();

// Получить заказчика LAZYK.
Customer cust = (from c in context.Customers
                 where c.CustomerID == "LAZYK"
                 select c).First();

// Очистить коллекцию Orders.
cust.Orders.Clear();

// Сохранить изменения.
context.SaveChanges();
```

Как и с методами Add или Remove, изменения не сохраняются в базе данных, пока не будет вызван метод SaveChanges. В примере вызов метода SaveChanges обновит все строки таблицы Orders со значением CustomerID, равным LAZYK, установив в этом поле значение NULL.

Contains()

Метод Contains позволяет определить, содержит ли EntityCollection заданный сущностный объект.

Прототипы

Метод `Contains` имеет один прототип, в котором `T` — сущностный тип элементов `EntityCollection<T>`. Метод `Contains` возвращает `true`, если сущностный объект содержится в коллекции, и `false` — в противном случае.

Единственный прототип `Contains`

```
public bool Contains(T entity);
```

Примеры

В листинге 21.23 демонстрируется использование метода `Contains`. В коде запрашиваются объекты `Customer` базы данных Northwind, затем из коллекции `EntityCollection<Order>` извлекается первый объект `Order`. После этого вызывается метод `Contains`, объект `Order` удаляется из коллекции с помощью метода `Remove` и снова вызывается `Contains`, чтобы сравнить эффект.

Листинг 21.23. Использование метода `Contains`

```
// Получить заказчика LAZYK.
Customer cust = (from c in context.Customers
                 where c.CustomerID == "LAZYK"
                 select c).First();

// Получить первый заказ заказчика.
Order ord = cust.Orders.First();

// Воспользоваться методом Contains.
Console.WriteLine("Содержат ли заказы определенный заказ: {0}",
  cust.Orders.Contains(ord));

// Удалить заказ из коллекции.
Console.WriteLine("Удаление заказа.");
cust.Orders.Remove(ord);

// Воспользоваться методом Contains.
Console.WriteLine("Содержат ли заказы определенный заказ: {0}",
  cust.Orders.Contains(ord));
```

Компиляция и запуск кода из листинга 21.23 даст следующий результат:

```
Содержат ли заказы определенный заказ: True
Удаление заказа.
Содержат ли заказы определенный заказ: False
```

Load()

Метод `Load` используется для явной загрузки сущностных объектов в коллекцию из базы данных, когда отключена ленивая загрузка. Подробности о загрузке данных и причинах, по которым она может понадобиться, читайте в главе 20.

Count

Свойство `Count` возвращает количество сущностных объектов в коллекции.

Прототипы

Есть только один прототип свойства `Count`.

Единственный прототип свойства `Count`

```
public int Count {get;};
```

Примеры

В листинге 21.24 свойство `Count` применяется для определения количества объектов `Orders`, ассоциированных с определенным объектом `Customer` базы данных Northwind. Затем `Order` удаляется и `Count` вызывается опять.

Листинг 21.24. Подсчет количества сущностных объектов с помощью свойства `Count`

```
NorthwindEntities context = new NorthwindEntities();

// Получить заказчика LAZYK.
Customer cust = (from c in context.Customers
                 where c.CustomerID == "LAZYK"
                 select c).First();

// Подсчитать количество заказов.
Console.WriteLine("Количество заказов: {0}", cust.Orders.Count);

// Получить первый заказ заказчика.
Order ord = cust.Orders.First();

// Удалить заказ из коллекции.
Console.WriteLine("Удаление заказа.");
cust.Orders.Remove(ord);

// Подсчитать количество заказов.
Console.WriteLine("Количество заказов: {0}", cust.Orders.Count);
```

Компиляция и запуск кода из листинга 21.24 дает результат, показанный ниже. Как и следовало ожидать, удаление экземпляра из коллекции уменьшает значение, возвращаемое свойством `Count`.

```
Количество заказов: 2
Удаление заказа.
Количество заказов: 1
```

Резюме

В этой главе были описаны ключевые члены ключевых классов, которые используются с LINQ to Entities. Были рассмотрены основные сложности Entity Framework и предоставлено достаточно информации для того, чтобы приступить к самостоятельной работе и увидеть, насколько много общего имеется у LINQ to Entities и LINQ to SQL. На этом часть книги, посвященная LINQ to Entities, завершена. Следующая глава посвящена введению в новейшее средство LINQ под названием Parallel LINQ.

ЧАСТЬ VII

Parallel LINQ

В этой части

Глава 22. Введение в Parallel LINQ

Глава 23. Использование Parallel LINQ

Глава 24. Операции Parallel LINQ

ГЛАВА 22

Введение в Parallel LINQ

Листинг 22.1. Простой пример Parallel LINQ

```
string[] presidents = {
   "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
   "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
   "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
   "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
   "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt",
   "Taft", "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};

string president = presidents.AsParallel()
   .Where(p => p.StartsWith("Lin")).First();

Console.WriteLine(president);
```

В главе 3 был написан простой пример LINQ to Objects, который находил первого президента США, чье имя начиналось с “Lin”. Просмотрите еще раз его. А теперь взгляните на листинг 22.1, в котором выполняется тот же запрос, но с применением Parallel LINQ. Чтобы было легче заметить разницу, мы выделили отличающийся фрагмент — новый вызов метода `AsParallel`. Как объясняется позже, запрос из листинга 22.1 не слишком подходит для Parallel LINQ, но он работает и демонстрирует важный момент, а именно: переход от обычного запроса LINQ к запросу Parallel LINQ может быть не сложнее добавления вызова метода `AsPareallel`.

В этой и последующих главах будет показано, как использовать Parallel LINQ эффективно, и описаны его сходства и различия с остальным семейством LINQ.

Введение

В своей основе Parallel LINQ, он же PLINQ — это версия LINQ to Objects, в которой объекты исходного перечисления обрабатываются параллельно. В этом определении заключено очень многое, так что давайте подробно разберем его, чтобы понять, что к чему.

В версии .NET 4 появился целый набор расширенных средств для упрощения параллельного программирования. Эти новые средства достаточно развиты, так что заслуживают написания отдельной книги.

Средства параллельного программирования существуют уже в течение длительного времени, но они были сложны в применении и многие программисты испытывали затруднения в их эффективном использовании. Средства .NET 4 были разработаны в расчете на более широкую аудиторию, и опираются на преимущества широкого распространения многопроцессорных и многоядерных машин.

Если посмотреть на исходный запрос в листинге 3.1, можно заметить, что в нем имя каждого президента обрабатывается по очереди. На рис. 22.1 проиллюстрировано, как это работает.

Рис. 22.1. Последовательная обработка LINQ

Механизм LINQ начинает с того, что проверяет, не начинается ли `Adams` с `Lin`. Затем переходит к `Arthur` и осуществляет проверку снова, после чего проверяет `Buchanan`, `Bush`, `Carter` и т.д. LINQ проходит последовательно по всем именам в массиве. Разумеется, это называется *последовательным* выполнением. Проблема последовательного выполнения состоит в том, что в каждый момент времени используется только одно ядро или один центральный процессор (далее будем говориться только о ядрах, но подразумеваться и то, и другое). На четырехядерных машинах три из четырех ядер не делают ничего при выполнении LINQ подобным образом.

Parallel LINQ изменяет правила игры, разбивая исходные данные на части и обрабатывая их параллельно, как показано на рис. 22.2.

Рис. 22.2. Выполнение Parallel LINQ

Имена `Arthur`, `Coolidge`, `Grant` и `Jackson` обрабатываются одновременно — каждый в отдельном ядре машины. После того, как каждое ядро завершит обработку имени, оно переходит к следующему, независимо от других ядер. Parallel LINQ заботится о разбиении данных, определяя, сколько элементов может быть обработано одновременно (хотя обычно принимается решение, что одного элемента данных на ядро достаточно), и координируя работу ядер таким образом, что результат получается так, как от любого другого запроса LINQ. Если скомпилировать и запустить код из листинга 21.1, получится следующий результат:

```
Lincoln
```

Так зачем вообще связываться с Parallel LINQ? Ответ прост — производительность. Взгляните на листинг 22.2. В нем определены два запроса LINQ, которые делают одно и то же. Один запрос последовательный, а другой использует Parallel LINQ. Оба запроса `select` выбирают четные числа между `0` и `Int32.MaxValue` и подсчитывают количество совпадений. Для генерации последовательности целочисленных значений применяются методы `Enumerable.Range` и `ParallelEnumerable.Range`. Диапазоны будут рассматриваться в главе 23, а пока просто имейте в виду, что оба метода создают `IEnumerable<int>`, содержащую все нужные целочисленные значения. Хоть этот пример не особенно полезен, но он позволяет понять идею.

Листинг 22.2. Сравнение производительности последовательного и параллельного выполнения

```
// Создать последовательный диапазон чисел.
IEnumerable<int> numbers1 = Enumerable.Range(0, Int32.MaxValue);

// Запустить секундомер.
Stopwatch sw = Stopwatch.StartNew();

// Выполнить запрос LINQ.
int sum1 = (from n in numbers1
            where n % 2 == 0
            select n).Count();

// Вывести результат последовательного выполнения.
Console.WriteLine("Результат последовательного выполнения: {0}", sum1);

// Вывести время последовательного выполнения.
Console.WriteLine("Время последовательного выполнения: {0} мс", sw.ElapsedMilliseconds);

// Создать параллельный диапазон чисел.
IEnumerable<int> numbers2 = ParallelEnumerable.Range(0, Int32.MaxValue);

// Перезапустить секундомер.
sw.Restart();

// Выполнить параллельный запрос LINQ.
int sum2 = (from n in numbers2.AsParallel()
            where n % 2 == 0
            select n).Count();

// Вывести результат параллельного выполнения.
Console.WriteLine("Результат параллельного выполнения: {0}", sum2);

// Вывести время параллельного выполнения.
Console.WriteLine("Время параллельного выполнения: {0} мс", sw.ElapsedMilliseconds);
```

Для измерения времени выполнения каждого запроса используется класс `Stopwatch` из пространства имен `System.Diaqgnostics`. После компиляции и запуска кода в листинге 22.2 были получены образы экрана диспетчера задач Windows при выполнении каждого запроса. Экран диспетчера задач Windows во время последовательного выполнения показан на рис. 22.3.

Как видите, использование центрального процессора составило 25%. Это четырехядерная машина, а потому можно было ожидать 25% загрузки при работе одного ядра — именно так последовательные запросы и выполняются. Экран диспетчера задач Windows во время выполнения запроса Parallel LINQ приведен на рис. 22.4.

В результате выполнения этого кода получается вывод, который показан ниже. Для последовательного запроса LINQ на обработку всех целочисленных значений и выдачи результата понадобилось около 38 секунд. Запрос Parallel LINQ сделал то же самое за 9 секунд. Это впечатляет, учитывая, что код запросов выглядит очень похоже и, как видно из вывода, оба варианта дают одинаковый результат.

За счет применения Parallel LINQ можно с минимальными усилиями получить значительный выигрыш в производительности.

```
Результат последовательного выполнения: 1073741824
Время последовательного выполнения: 38521 мс
Результат параллельного выполнения: 1073741824
Время параллельного выполнения: 9498 мс
```

Рис. 22.3. Использование центрального процессора во время последовательного выполнения запроса

Parallel LINQ предназначен для объектов

Как уже говорилось, Parallel LINQ — это параллельная реализация API-интерфейса LINQ to Objects. Поэтому он выполняет запросы LINQ to Objects в параллельном режиме. Он не реализует параллельных средств для других видов LINQ, описанных в этой книге.

Это не значит, что с помощью Parallel LINQ не удастся обработать результаты других разновидностей запросов LINQ (например, выбрать объекты `Order` из базы данных Northwind с использованием LINQ to Entities или LINQ to SQL, а затем применить Parallel LINQ для дальнейшей обработки результатов), но Parallel LINQ не работает ни на чем, кроме объектов.

Далеко не все запросы LINQ to Objects являются хорошими кандидатами для запросов Parallel LINQ. Есть еще накладные расходы, связанные с разбиением данных на фрагменты, установкой и управлением классами, выполняющими параллельные задачи. Если запрос не слишком долго выполняется последовательно, возможно, не имеет смысла выполнять его параллельно, т.к. накладные расходы могут свести на нет весь выигрыш в производительности.

Рис. 22.4. Использование центрального процессора во время параллельного выполнения запроса

Использование API-интерфейса Parallel LINQ

Для использования Parallel LINQ никаких специальных шагов предпринимать не понадобится. Все ключевые классы содержатся в пространстве имен `System.Linq`, где также находятся и все обычные классы LINQ to Objects. Наиболее важные методы и ключевые операции будут описаны в главе 24.

Резюме

В этой главе был представлен API-интерфейс Parallel LINQ, который параллельно обрабатывает множество элементов данных в запросе LINQ to Objects. При грамотном применении Parallel LINQ обеспечивает существенный рост производительности запросов LINQ to Objects. Было продемонстрировано несколько простых запросов со сравнением производительности их последовательной и параллельной версий. Следующая глава посвящена использованию полного диапазона средств Parallel LINQ.

ГЛАВА 23

Использование Parallel LINQ

В этой главе будет показано, как использовать Parallel LINQ, начиная с основ и заканчивая расширенными опциями управления параллельным выполнением. С Parallel LINQ легко начать работать, и с самого начала могут получаться неплохие результаты, но если нужно достичь самых лучших результатов, потребуется некоторое планирование и приложением определенных усилий.

Создание запроса Parallel LINQ

По большей части использование Parallel LINQ, обычно называемого PLINQ, очень похоже на применение LINQ to Object. Фактически это одна из привлекательных сторон PLINQ. В обычном запросе LINQ to Querey источником данных является `IEnumerable<T>`, где `T` — обрабатываемый тип данных. Механизм LINQ автоматически переключается на использование PLINQ, когда источником данных является экземпляр типа `ParallelQuery<T>`. И здесь есть один трюк: любой `IEnumerable<T>` может быть преобразован в `ParallelQuery<T>` просто с использованием метода `AsParallel`. Давайте рассмотрим код. В листинге 23.1 показан запросы LINQ to Object и PLINQ, которые делают одно и то же.

Листинг 23.1. Сравнение запросов LINQ и Parallel LINQ

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt",
  "Taft", "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
// Последовательный запрос LINQ.
IEnumerable<string> results = from p in presidents
                              where p.Contains('o')
                              select p;
foreach (string president in results) {
  Console.WriteLine("Результат последовательного запроса: {0}", president);
}
// Запрос Parallel LINQ.
results = from p in presidents.AsParallel()
          where p.Contains('o')
          select p;
foreach (string president in results) {
  Console.WriteLine("Результат параллельного запроса: {0}", president);
}
```

Первый запрос использует обычный LINQ to Objects для обработки каждого из президентов с целью нахождения имен, содержащих букву “o”. В качестве результата получается IEnumerable<string> и все подходящие имена выводятся на консоль.

Второй запрос делает то же самое, но вдобавок вызывается метод AsParallel. С его помощью источник данных преобразуется в ParallelQuery, что автоматически подразумевает применение Parallel LINQ. И как явно следует из кода, никаких других изменений не требуется. Просто вызвав AsParallel, мы получаем PLINQ. Запуск кода из листинга 23.1 дает следующие результаты:

```
Результат последовательного запроса: Clinton
Результат последовательного запроса: Coolidge
Результат последовательного запроса: Eisenhower
Результат последовательного запроса: Fillmore
Результат последовательного запроса: Ford
Результат последовательного запроса: Harrison
Результат последовательного запроса: Hoover
Результат последовательного запроса: Jackson
Результат последовательного запроса: Jefferson
Результат последовательного запроса: Johnson
Результат последовательного запроса: Lincoln
Результат последовательного запроса: Madison
Результат последовательного запроса: Monroe
Результат последовательного запроса: Nixon
Результат последовательного запроса: Polk
Результат последовательного запроса: Roosevelt
Результат последовательного запроса: Taylor
Результат последовательного запроса: Washington
Результат последовательного запроса: Wilson
Результат параллельного запроса: Lincoln
Результат параллельного запроса: Roosevelt
Результат параллельного запроса: Clinton
Результат параллельного запроса: Ford
Результат параллельного запроса: Madison
Результат параллельного запроса: Taylor
Результат параллельного запроса: Coolidge
Результат параллельного запроса: Harrison
Результат параллельного запроса: Monroe
Результат параллельного запроса: Washington
Результат параллельного запроса: Eisenhower
Результат параллельного запроса: Hoover
Результат параллельного запроса: Nixon
Результат параллельного запроса: Wilson
Результат параллельного запроса: Fillmore
Результат параллельного запроса: Jackson
Результат параллельного запроса: Polk
Результат параллельного запроса: Jefferson
Результат параллельного запроса: Johnson
```

Относительно этих результатов необходимо сделать одно замечание. Результаты последовательного запроса отсортированы в алфавитном порядке, а результаты параллельного — нет. Почему? Причина объясняется немного ниже.

Ключевой момент, который важно было продемонстрировать — это простота создания запроса PLINQ. Достаточно просто вызвать метод AsParallel с выражениями запроса (как это делалось в листинге 23.1) или воспользоваться расширяющими методами для структурирования запроса. В листинге 23.2 показаны два запроса PLINQ, в которых это применяется.

Листинг 23.2. Параллельные запросы, написанные с использованием выражений запросов и расширяющих методов

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt",
  "Taft", "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
IEnumerable<string> results = from p in presidents.AsParallel()
                              where p.StartsWith("M")
                              select p;
foreach (string president in results) {
  Console.WriteLine("Результат с использованием выражений запросов: {0}", president);
}
results = presidents.AsParallel()
  .Where(p => p.StartsWith("M"))
  .Select(p => p);
foreach (string president in results) {
  Console.WriteLine("Результат с использованием расширяющих методов: {0}", president);
}
```

Первый запрос написан с использованием выражений запросов, а второй — с применением расширяющих методов. Конечно, оба запроса вызывают метод `AsParallel`. Без этого был бы получен последовательный запрос LINQ. Запуск кода из листинга 23.2 после компиляции даст следующие результаты, которые демонстрируют, что нет разницы между способами выполнения запросов:

```
Результат с использованием выражений запросов: Madison
Результат с использованием выражений запросов: McKinley
Результат с использованием выражений запросов: Monroe
Результат с использованием расширяющих методов: Madison
Результат с использованием расширяющих методов: McKinley
Результат с использованием расширяющих методов: Monroe
```

Известно, что это не идеальный запрос для применения с PLINQ. В предыдущей главе объяснялось, что накладные расходы в небольших простых запросах могут привести к ухудшению производительности по сравнению с последовательным выполнением. Однако нужно было продемонстрировать средства, и чем меньше времени будет потрачено на избыточно сложные примеры, тем проще понять то, что планировалось донести. В этой главе будут использоваться простые запросы. И это один из моментов, когда следует делать не так, как делается здесь.

Предохранение порядка результатов

Результаты, полученные при выполнении кода из листинга 23.1, не были организованы в алфавитном порядке. С другой стороны, результат выполнения листинга 23.2 упорядочен по алфавиту. Почему?

Ответ заключается в способе, которым PLINQ обрабатывает данные. Данные, представленные в качестве источника для запроса PLINQ, разбиты на части и разделены для параллельной обработки. Несколько разделов могут обрабатываться одновременно. Однако каждый из этих разделов обрабатывается последовательно. Задумайтесь об этом — параллельное выполнение происходит из последовательной обработки нескольких разделов данных одновременно. Это показано на рис. 23.1.

Рис. 23.1. Параллельное выполнение последовательных обработок разделов данных

Когда PLINQ разделяет данные, получается нечто вроде показанного на рис. 23.1, но в этом нельзя быть уверенным, потому что механизм PLINQ анализирует запрос и данные, выполняя разбиение "за кулисами". Но давайте предположим, что есть то, что показано на рис. 23.1, т.е. множество разделов, каждый из которых содержит имена пяти президентов. PLINQ назначает один раздел для обработки одному из ядер машины, и каждое ядро обрабатывает назначенный ему раздел последовательно.

Таким образом, продолжая пример, первое ядро проверит `Adams` на содержание буквы `o`. Затем будет выполнена проверка `Arthur`, `Buchanan`, `Bush` и `Carter`. Пока это происходит, второе ядро проверяет `Cleveleand`, `Coolidge`, `Eisenhower` и т.д. В то же время третье и четвертое ядра обрабатывают свои разделы.

Найденное соответствие добавляется в результирующий набор. На рис. 23.1 имена президентов, содержащие букву "o" помечены, и легко заметить, что если элементы обрабатывались примерно в одно и то же время каждым из ядер, то первым результатом будет `Hoover`, за ним последуют `Coolidge`, `Jackson`, `Eisenhower`, `Jefferson` и т.д.

Начало результирующего списка выглядит так:

```
Hoover, Coolidge, Jackson, Eisenhower, Jefferson...
```

Таким образом, можно видеть, что PLINQ генерирует результаты, которые не упорядочены так, как исходные данные. Хуже того, поскольку заранее не известно то, как PLINQ разобьет данные, то и нельзя предсказать, каким будет порядок. Еще хуже то, что разделы не обрабатываются за один шаг. Другие процессы на машине могут приостанавливать выполнение приложения .NET на одном или более ядер, а это означает, что в действительности при многократном запуске одного и того же запроса будут получены результаты, упорядоченные по-разному.

Иногда порядок результата не важен. Например, если необходимо только узнать, сколько имен президентов содержат букву "o", порядок формирования результата значения не имеет. Запрос подобного рода показан в листинге 23.3.

Листинг 23.3 Запрос PLINQ, в котором порядок результата не важен

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt",
  "Taft", "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};

int count = presidents.AsParallel()
   .Where(p => p.Contains("o"))
   .Select(p => p)
   .Count();

Console.WriteLine("Количество элементов в результате: {0}", count);
```

Независимо от того, как PLINQ разделит и разместит данные, получается один и тот же результат — 19 соответствий:

```
Количество элементов в результате: 19
```

Однако бывают случаи, когда приходится заботиться о порядке результата. Это особенно верно при преобразовании существующих запросов LINQ в PLINQ. Например, где-то могут существовать предположения о порядке результатов. Чтобы сохранить порядок, необходимо воспользоваться расширяющим методом `AsOrdered` на объекте `ParallelQuery`, который создан посредством метода `AsParallel`. Например, для сохранения порядка имен президентов можно было бы выполнить следующий вызов:

```
presidents.AsParallel().AsOrdered()
```

Вызов метода `AsOrdered` заставляет PLINQ предохранить порядок результатов. Применение этого метода демонстрируется в листинге 23.4.

Листинг 23.4. Предохранение порядка результатов запроса PLINQ методом `AsOrdered`

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt",
  "Taft", "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};

// Запрос Parallel LINQ.
IEnumerable<string> results = from p in presidents.AsParallel().AsOrdered()
                              where p.Contains('o')
                              select p;

foreach (string president in results) {
   Console.WriteLine("Результат параллельного запроса: {0}", president);
}
```

Для расширяющих методов `AsParallel` и `AsOrdered` ключевых слов выражений запросов не предусмотрено. Эти методы должны вызываться непосредственно. В листинге 23.4 ключевые слова запросов смешаны с вызовами расширяющих методов. Если скомпилировать и запустить код из листинга 23.4, получится следующий результат:

```
Результат параллельного запроса: Clinton
Результат параллельного запроса: Coolidge
Результат параллельного запроса: Eisenhower
```

```
Результат параллельного запроса: Fillmore
Результат параллельного запроса: Ford
Результат параллельного запроса: Harrison
Результат параллельного запроса: Hoover
Результат параллельного запроса: Jackson
Результат параллельного запроса: Jefferson
Результат параллельного запроса: Johnson
Результат параллельного запроса: Lincoln
Результат параллельного запроса: Madison
Результат параллельного запроса: Monroe
Результат параллельного запроса: Nixon
Результат параллельного запроса: Polk
Результат параллельного запроса: Roosevelt
Результат параллельного запроса: Taylor
Результат параллельного запроса: Washington
Результат параллельного запроса: Wilson
```

Если просмотреть еще раз результаты выполнения кода из листинга 23.1, можно увидеть, что все совпадает. Метод `AsOrdered` очень полезен, но не стоит привыкать вызывать его автоматически, поскольку он требует от PLINQ выполнения дополнительной работы по упорядочиванию результатов. Учитывая то, что главной целью PLINQ является повышение производительности, следует избегать лишней работы, где это возможно.

Управление параллелизмом

PLINQ анализирует запрос и решает, сколько разделов можно обработать сразу. В Microsoft утверждают, что работают над совершенствованием этого процесса, так что вы не должны делать никаких предположений относительно этого поведения, основываясь на результатах, полученных в текущем выпуске.

Принудительное параллельное выполнение

В некоторых случаях PLINQ может решить, что запрос лучше выполнять последовательно. Этим можно управлять за счет применения расширяющего метода `WithExecutionMode` к типу `ParallelQuery`. Метод `WithExecutionMode` принимает значение из перечисления `ParallelExecutionMode`. Этих значений два: режим по умолчанию (который позволяет PLINQ решать, как выполнять обработку) и `ForceParallelism` (использовать PLINQ даже если накладные расходы, связанные с параллельным выполнением, превосходят выигрыш от него). В листинге 23.5 показано, как вызвать принудительно параллельное выполнение.

Листинг 23.5. Принудительное параллельное выполнение с использованием метода `WithExecutionMode`

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt",
  "Taft", "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
// Запрос Parallel LINQ.
IEnumerable<string> results = presidents
    .AsParallel()
    .WithExecutionMode(ParallelExecutionMode.ForceParallelism)
    .Where(p => p.Contains('o'))
    .Select(p => p);
```

```
foreach (string president in results) {
   Console.WriteLine("Результат параллельного запроса: {0}", president);
}
```

Ограничение степени параллелизма

Можно потребовать, чтобы PLINQ ограничивал количество разделов, обрабатываемых параллельно, с использованием расширяющего метода `WithDegreeofParallelism`, который работает с типом `ParallelQuery`. Этот метод принимает аргумент `int`, который устанавливает максимальное количество разделов, которые должны быть обработаны одновременно; это называется *степенью параллелизма*. Установка степени параллелизма не принуждает PLINQ использовать именно столько разделов, а лишь указывает верхний предел. PLINQ может решить использовать меньшее количество разделов, чем указано, или же, если метод `WithExecutionMode` не использовался, может вообще выполнять запрос последовательно. Применение этого метода продемонстрировано в листинге 23.6.

Листинг 23.6. Установка степени параллелизма в запросе PLINQ

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt",
  "Taft", "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
// Запрос Parallel LINQ.
IEnumerable<string> results = presidents
   .AsParallel()
   .WithDegreeOfParallelism(2)
   .Where(p => p.Contains('o'))
   .Select(p => p);
foreach (string president in results) {
  Console.WriteLine("Результат параллельного запроса: {0}", president);
}
```

В листинге 23.6 задана максимальная степень параллелизма `2`, а это означает, что обрабатывать одновременно понадобится максимум два раздела. Это может быть удобно, если необходимо ограничить влияние запроса на машину, которая должна выполнять и другие задачи.

Обработка исключений

Если в последовательном запросе LINQ что-то идет не так, генерируемое исключение прерывает любую дальнейшую обработку. Например, если при обработке имени президента `Arthur` генерируется исключение, ни одно из имен, следующих за `Arthur`, не будет обработано, как показано на рис. 23..2.

Таким образом, мы должны получить результаты, предшествующие `Arthur`, но не следующие за ним. Давайте проверим. В листинге 23.7 содержится последовательный запрос, который выбирает имена всех президентов и выводит их на консоль. Однако в код добавлен один трюк. При достижении запросом `Arthur` генерируется исключение.

Рис. 23.2. Исключение в последовательном запросе

Листинг 23.7. Принудительное исключение в последовательном запросе

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt",
  "Taft", "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
// Запрос Parallel LINQ.
IEnumerable<string> results = presidents
   .Select(p => {
      if (p == "Arthur")
         throw new Exception(String.Format("Problem with President {0}", p));
      return p;
   });
try {
  foreach (string president in results) {
    Console.WriteLine("Результат: {0}", president);
  }
} catch (Exception ex) {
  Console.WriteLine(ex.Message);
}
```

После компиляции и запуска кода из листинга 23.7 получаются показанные ниже результаты, которых можно было ожидать. Элемент `Adams` обрабатывается корректно, а затем возникает проблема (вызванная намеренно) с `Arthur`. Сгенерированное исключение прекращает выполнение оставшейся части запроса.

```
Результат: Adams
Problem with President Arthur
```

Но с PLINQ все обстоит иначе. Вспомните, что данные разбиты на разделы, которые затем обрабатываются независимо и параллельно. Может случиться так, что будет сгенерировано более одного исключения, и поскольку одновременно обрабатывается несколько разделов, то первое исключение не останавливает обработку. На рис. 23.3 показано, как это происходит.

На рис. 23.3 четыре раздела обрабатываются параллельно. В первом разделе возникает проблема с `Arthur`, которая приводит к генерации исключения. Однако это не останавливает обработку других разделов, а проблема с `Harding` вызывает генерацию второго исключения. Так что же делать?

К счастью, существует изящное решение. PLINQ собирает все исключения, которые находит, и упаковывает их в `System.AggregateException`, которое затем генерирует в коде. В листинге 23.8 содержится запрос PLINQ, который сгенерирует исключения для значений `Arthur` и `Harding`.

Рис. 23.3. Выполнение параллельного запроса

Листинг 23.8. Инициация исключения в запросе PLINQ

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt",
  "Taft", "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};

// Запрос Parallel LINQ.
IEnumerable<string> results = presidents
  .AsParallel()
  .Select(p => {
    if (p == "Arthur" || p == "Harding")
      throw new Exception(String.Format("Problem with President {0}", p));
      return p;
    });
try {
  foreach (string president in results) {
    Console.WriteLine("Результат: {0}", president);
  }
} catch (AggregateException agex) {
  agex.Handle(ex => {
    Console.WriteLine(ex.Message);
    return true;
  });
}
```

Когда начинается перечисление результатов, цикл `foreach` помещается в блок `try/catch`, который ищет `AggregateException`. Класс `AggregateException` имеет метод `Handle`, позволяющий обрабатывать каждое исключение по очереди. Ему передается исключение и возвращается `true`, если оно обработано, либо `false`, если обработать исключение не удается.

Если исключение не обрабатывается, оно распространяется дальше и в конечном итоге прекращает работу программы. С другой стороны, не имеет смысла обрабатывать исключения, которые не ожидались, и не известно, что с ними делать. Это путь к непредсказуемому поведению и трудно обнаруживаемым ошибкам.

Результаты, которые получаются от запроса PLINQ, когда возникло исключение, непредсказуемы. Все зависит от того, как PLINQ разделяет данные и сколько разделов обрабатывается параллельно. Например, запуск кода из листинга 23.8 дает следующие результаты:

```
Результат: Reagan
Результат: Roosevelt
Результат: Taft
Результат: Taylor
Результат: Truman
Результат: Tyler
Результат: Van Buren
Результат: Washington
Результат: Wilson
Problem with President Arthur
Problem with President Harding
```

Запустив тот же код второй раз, получаются совершенно другие результаты:

```
Problem with President Arthur
Problem with President Harding
```

Запросы без результатов

PLINQ включает полезное средство в расширяющем методе `ForAll`. Использованный с объектом `ParallelQuery` (который, как известно, возвращается методом `AsParallel`), `ForAll` выполняет `System.Action` на каждом элементе в последовательности. В одном из повторяющихся примеров этой главы ищутся все имена президентов, содержащие букву “o”. Для фильтрации соответствующих имен использовалась конструкция `where`, а выбранные имена добавлялись к результирующей коллекции `IEnumerable<string>`. Затем результаты перечисляются в цикле `foreach` с выводом их на консоль посредством `Console.WriteLine`.

С помощью метода `ForAll` можно сделать то же самое, но намного элегантнее. Рассмотрим код в листинге 23.9.

Листинг 23.9. Использование метода `ForAll`

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt",
  "Taft", "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};

// Запрос Parallel LINQ.
presidents.AsParallel()
   .Where(p => p.Contains('o'))
   .ForAll(p => Console.WriteLine("Имя: {0}", p));
```

Здесь по-прежнему используется метод `Where` для фильтрации последовательности, но вместо сбора результатов имена выводятся на консоль непосредственно, с применением лямбда-выражения, переданного методу `ForAll`. Чтобы понять работу этого средства, может потребоваться некоторое время. В конце концов, все прочие примеры, приведенные в этой книге, функционировали иначе. Однако метод `ForAll` стоит знать.

Ниже показаны результаты запуска кода из листинга 23.9:

```
Имя: Ford
Имя: Clinton
Имя: Lincoln
Имя: Harrison
Имя: Roosevelt
Имя: Taylor
Имя: Coolidge
Имя: Madison
Имя: Hoover
Имя: Jackson
Имя: Jefferson
Имя: Johnson
Имя: Eisenhower
Имя: Fillmore
Имя: Monroe
Имя: Nixon
Имя: Polk
Имя: Washington
Имя: Wilson
```

Внутри объекта `Action`, передаваемого методу `ForAll`, можно делать довольно много всего за исключением возврата результата. Можно даже фильтровать данные без помощи конструкции `Where`. Пример приведен в листинге 23.10.

Листинг 23.10. Фильтрация данных без конструкции `Where`

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt",
  "Taft", "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};

int count = 0;
presidents.AsParallel()
   .ForAll(p => {
      if (p.Contains('o')) {
         System.Threading.Interlocked.Increment(ref count);
      }
   });

Console.WriteLine("Совпадений: {0}", count);
```

В листинге 23.10 метод `ForAll` использовался для выполнения действия над каждым элементом в последовательности данных. Для каждого элемента проверяется наличие в имени буквы “o” и увеличивается значение счетчика, если она там есть. Результат запуска кода из листинга 23.10 выглядит следующим образом:

```
Совпадений: 19
```

Не забывайте, что метод `ForAll` является частью PLINQ, а это значит, что указываемое в объекте `Action` действие выполняется в разделах последовательности данных параллельно. Это дает выигрыш в производительности за счет параллельного выполнения, но может привести к проблемам с разделяемыми данными, такими как значение `int`, используемое в качестве счетчика совпадений. Для обеспечения точности подсчета применяется класс `Interlocked` из пространства имен `System.Threading`. Этот прием называется *синхронизацией*, и представляет собой расширенную технику параллельного программирования.

Создание диапазонов и повторов

Иногда возникает необходимость в обработке последовательности числовых величин либо последовательности, хранящей одно и то же значение. Разумеется, такие последовательности можно создавать вручную с помощью примерно такого кода:

```
int[] sequence = new int[50000];
for (int i = 0; i < sequence.Length; i++) {
   sequence[i] = i;
}
```

В качестве альтернативы такого рода последовательность можно сгенерировать с использованием статического метода `Range` из пространства имен `System.Linq.ParallelEnumerable`. В листинге 23.11 приведен код создания той же последовательности, что и ранее, и выполнения запроса к ней.

Листинг 23.11. Генерация и использование параллельной последовательности

```
IEnumerable<int> evens
  = ((ParallelQuery<int>) ParallelEnumerable.Range(0, 50000))
    .Where(i => i % 2 == 0)
    .Select(i => i);
```

В листинге 23.11 применяется метод `Range` для создания последовательности из 50 000 целых чисел, начиная с нуля. Первый аргумент метода — это начальный индекс; второй — количество нужных значений. Обратите внимание, что результат метода `Range` должен быть приведен к типу `ParallelQuery<int>`. Если этого не сделать, то LINQ не распознает последовательность как поддерживающую параллельное выполнение, и запрос будет последовательным.

`ParallelEnumerable` содержит связанный с этим метод, хотя и не используемый так часто, как `Range`. Статический метод `Repeat` принимает объект и счетчик, а в результате создает последовательность, в которой объект повторяется заданное количество раз. В качестве примера в листинге 23.12 создается последовательность, в которой повторяется одно и то же целое значение.

Листинг 23.12. Генерация и использование повторяющейся последовательности

```
int sum = ParallelEnumerable.Repeat(1, 50000)
    .Select(i => i)
    .Sum();
Console.WriteLine("Сумма: {0}", sum);
```

В листинге 32.12 выбираются все элементы последовательности, и для суммирования значений вызывается расширяющий метод `Sum`. Запуск кода из листинга 23.12 дает следующий ожидаемый результат:

```
Сумма: 50000
```

Резюме

В этой главе было описано наиболее полезное средство Parallel LINQ. Оно позволяет с минимальными изменениями в коде обеспечить обработку данных с привлечением всех ядер машины. PLINQ прост в использовании и может принести существенную выгоду. Лучше всего то, что его легко применять к существующим запросам LINQ, так что получается выигрыш в производительности с минимальными усилиями.

Несмотря на то что это может показаться не очень соответствующем остальной части LINQ, не забывайте об удобном методе `ForAll`. В следующей главе вы ознакомитесь с ключевыми членами наиболее важных классов PLINQ.

ГЛАВА 24

Операции Parallel LINQ

В этой заключительной главе рассматриваются ключевые операции Parallel LINQ (PLINQ). Вы должны были заметить, что операции PLINQ выражаются с помощью набора расширяющих методов класса `ParallelEnumerable`, применяемых к типу `ParallelQuery`. Ниже будут описаны параллельные операции и показано, как они сочетаются друг с другом.

В главу включены только те операции, которые позволяют создавать экземпляры `ParallelQuery` или управлять выполнением параллельного запроса. Большинство операций PLINQ идентичны аналогичным операциям LINQ to Objects, кроме того, что они применяются к запросам `ParallelQuery`. Операции LINQ to Objects были описаны в главах 3 и 4.

Операции создания `ParallelQuery`

Ниже перечислены операции создания `ParallelQuery`.

`AsParallel`

Метод `AsParallel` — своего рода "входная дверь" в PLINQ. Он преобразует последовательность данных в `ParallelQuery`. Механизм LINQ обнаруживает использование `ParallelQuery` в качестве источника в запросе и переключается на выполнение PLINQ автоматически. Метод `AsParallel`, скорее всего, будет применяться при каждом использовании PLINQ.

Прототипы

Метод `AsParallel` имеет два прототипа, которые описаны ниже.

Первый прототип `AsParallel`

```
public static ParallelQuery<T> AsParallel<T> (
   this IEnumerable<T> source
)
```

Этот прототип работает на `IEnumerable<T>` и возвращает экземпляр `ParallelQuery<T>`, который может быть использован в качестве основы запроса PLINQ. Этот метод применялся во всех примерах PLINQ предыдущей главы, и будет использоваться почти во всех примерах настоящей.

Второй прототип `AsParallel`

```
public static ParallelQuery<T> AsParallel(
   this IEnumerable source
)
```

Второй прототип создает экземпляр ParallelQuery из IEnumerable и предназначен для поддержки унаследованных коллекций, таких как System.Collections.ArrayList. Объект ParallelQuery не является строго типизированным и не может использоваться в качестве основы запроса PLINQ без преобразования в ParallelQuery<T>. Приведение ParallelQuery к ParallelQuery<T> осуществляется с помощью операции Cast<T> или путем фильтрации последовательности для получения элементов — экземпляров типа T с применением операции OfType<T>.

Примеры

В листинге 24.1 используется первый прототип AsParallel для создания объекта ParallelQuery, который затем используется в качестве источника для запроса PLINQ. Этот пример аналогичен примеру из предыдущей главы — нахождение имен президентов, содержащих букву “o”.

Листинг 24.1. Создание ParallelQuery с помощью первого прототипа AsParallel

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt",
  "Taft", "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
ParallelQuery<string> pq = presidents.AsParallel();
IEnumerable<string> results = from p in pq
                              where p.Contains('o')
                              select p;
foreach (string president in results) {
   Console.WriteLine("Совпадение: {0}", president);
}
```

Скомпилировав и запустив код из листинга 24.1, получаются следующие результаты:

```
Совпадение: Roosevelt
Совпадение: Clinton
Совпадение: Ford
Совпадение: Lincoln
Совпадение: Taylor
Совпадение: Coolidge
Совпадение: Harrison
Совпадение: Madison
Совпадение: Washington
Совпадение: Eisenhower
Совпадение: Hoover
Совпадение: Monroe
Совпадение: Wilson
Совпадение: Fillmore
Совпадение: Jackson
Совпадение: Nixon
Совпадение: Jefferson
Совпадение: Polk
Совпадение: Johnson
```

Как видите, результаты выстроены не в том порядке, в котором находились элементы исходной последовательности. За дополнительной информацией о порядке результатов PLINQ обращайтесь к предыдущей главе. Предохранить порядок результатов в запросе PLINQ позволяет операция AsOrdered.

Применение второго прототипа продемонстрировано в листинге 24.2. Здесь определяется список ArrayList (унаследованная не строго типизированная коллекция), содержащий некоторые имена президентов. Затем вызывается метод AsParallel для создания экземпляра ParallelQuery и Cast<string> — для получения экземпляра ParallelQuery<string>, который далее может использоваться в качестве основы для запроса PLINQ.

Листинг 24.2. Использование второго прототипа операции AsParallel

```
ArrayList list = new ArrayList() {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson"};
IEnumerable<string> results = list
   .AsParallel()
   .Cast<string>()
   .Where(p => p.Contains('o'))
   .Select(p => p);
foreach (string president in results) {
   Console.WriteLine("Совпадение: {0}", president);
}
```

В этом листинге фактически эмулируется то, что понадобится делать, когда необходимо использовать PLINQ с унаследованной коллекцией данных. Недостаточно просто вызвать AsParallel; кроме этого также должна быть вызвана операция Cast<T>, чтобы получить нечто, с чем может работать PLINQ. Скомпилировав и запустив код из листинга 24.2, получатся следующие результаты:

```
Совпадение: Fillmore
Совпадение: Coolidge
Совпадение: Clinton
Совпадение: Eisenhower
Совпадение: Ford
Совпадение: Harrison
Совпадение: Hoover
Совпадение: Jackson
```

Код в листинге 24.2 работает хорошо, если все объекты в унаследованной коллекции относятся к одному и тому же типу. Скомбинировав второй прототип AsParallel с операцией OfType<T>, можно отфильтровать только те объекты, которые относятся к заданному типу. В листинге 24.3 приведен пример.

Листинг 24.3. Создание ParallelQuery<T> посредством фильтрации ParallelQuery

```
ArrayList list = new ArrayList();
list.Add("Adams");
list.Add(23);
list.Add("Arthur");
list.Add(DateTime.Now);
list.Add("Buchanan");
list.Add(new string[] { "apple", "orange" });
IEnumerable<string> results = list
   .AsParallel()
   .OfType<string>()
   .Select(p => p);
foreach (string president in results) {
  Console.WriteLine("Совпадение: {0}", president);
}
```

В листинге 24.3 создается список ArrayList, содержащий имена первых трех президентов и три других объекта. Затем на ArrayList используется второй прототип AsParallel, после чего вызовом OfType<string> в последовательности отфильтровываются элементы string. В запросе используются только элементы типа string из последовательности. Скомпилировав и запустив код из листинга 24.3, получатся следующие результаты:

```
Совпадение: Adams
Совпадение: Arthur
Совпадение: Buchanan
```

Range

Метод Range создает объект ParallelQuery<int>, содержащий возрастающую последовательность целых чисел. Это статический метод класса ParallelEnumerable, а не расширяющий метод.

Прототипы

У метода Range единственный прототип.

Прототип операции Range

```
public static ParallelQuery<int> Range(
   int start,
   int count
)
```

Методу Range передаются два аргумента. Первый — это целочисленное значение, с которого должна начинаться последовательность, а второй — количество элементов, которые должны быть в последовательности. Метод Range возвращает объект ParallelQuery<int>, содержащий возрастающие значения.

Примеры

В листинге 24.4 демонстрируется использование параллельного диапазона. Здесь вызывается статический метод Range для создания ParallelQuery<int>, содержащего 10 целых чисел, начиная со значения 0. Затем в цикле foreach перебираются все элементы последовательности с выводом их на консоль. Та же последовательность используется в качестве основы для запроса PLINQ, где выбираются и выводятся на консоль четные целочисленные значения.

Листинг 24.4. Использование последовательности диапазона

```
ParallelQuery<int> pq = ParallelEnumerable.Range(0, 10);
foreach (int i in pq) {
   Console.WriteLine("Значение {0}", i);
}
IEnumerable<int> results = from i in pq
                           where i % 2 == 0
                           select i;
foreach (int i in results) {
   Console.WriteLine("Совпадение: {0}", i);
}
```

Запуск этого примера дает следующие результаты:

```
Значение 0
Значение 1
```

```
Значение 2
Значение 3
Значение 4
Значение 5
Значение 6
Значение 7
Значение 8
Значение 9
Совпадение: 0
Совпадение: 4
Совпадение: 6
Совпадение: 8
Совпадение: 2
```

Repeat

Repeat, как и Range — статический метод класса ParallelEnumerable, а не операция расширяющего метода. Метод Repeat создает объект ParallelQuery<T>, содержащий одиночное значение типа T, которое повторяется заданное количество раз.

Прототипы

Метод Repeat имеет один прототип.

Прототип метода Repeat

```
public static ParallelQuery<T> Repeat<T>(
   T element,
   int count
)
```

Метод Repeat принимает два аргумента. Первый — это элемент, который должен повторяться. Второй — количество его повторений в последовательности. Метод Repeat возвращает ParallelQuery<T>, где T — тип элемента, переданного в первом аргументе.

Примеры

В листинге 24.5 показано создание повторяющейся последовательности с использованием метода Repeat.

Листинг 24.5. Использование метода Repeat

```
ParallelQuery<int> pq = ParallelEnumerable.Repeat(2, 10);
foreach (int i in pq) {
   Console.WriteLine("Значение {0}", i);
}
```

Здесь создается последовательность, в которой целое значение 2 повторяется 10 раз. В цикле foreach последовательность перечисляется с выводом всех ее элементов на консоль. Запуск кода из листинга 24.5 дает следующие результаты:

```
Значение 2
Значение 2
Значение 2
Значение 2
Значение 2
Значение 2
Значение 2
Значение 2
Значение 2
Значение 2
```

Empty

Статический метод ParallelEnumerable.Empty создает объект ParallelQuery<T>, не содержащий элементов.

Тип T для ParallelQuery<T> указывается вызовом Empty<T>(). Для создания ParallelQuery<string> понадобится вызвать Empty<string>().

Прототипы

Метод Repeat имеет один прототип.

Прототип метода Empty

```
public static ParallelQuery<T> Empty<TResult>()
```

Операции управления выполнением

Использовать PLINQ можно, просто вызывая AsParallel или одну из описанных выше операций создания. Но если нужен более тонкий контроль над формированием запроса PLINQ, то для этого понадобится работать с одной или более операциями, описанными в этом разделе.

AsOrdered

Операция AsOrdered предохраняет порядок результатов в соответствии с порядком исходной последовательности. В главе 23 объяснялось, почему параллельная обработка по умолчанию не предохраняет порядка результатов.

Прототипы

Операция AsOrdered имеет два прототипа.

Первый прототип AsOrdered

```
public static ParallelQuery<T> AsOrdered<T>(
   this ParallelQuery<T> source
)
```

Первый прототип AsOrdered принудительно устанавливает порядок результатов для ParallelQuery<T>. Этот прототип будет применяться наиболее часто. Результатом операции также является объект ParallelQuery<T>, который можно использовать как входную последовательность для запроса PLINQ. Второй прототип AsOrdered работает на слабо типизированном ParallelQuery.

Второй прототип AsOrdered

```
public static ParallelQuery AsOrdered(
   this ParallelQuery source
)
```

Второй прототип AsOrdered работает на слабо типизированном ParallelQuery. Это разновидность ParallelQuery, которая получается при вызове AsOrdered на унаследованной коллекции. Прежде чем можно будет использовать последовательность данных в запросе PLINQ, необходимо применить операцию OfType или Cast.

Примеры

В листинге 24.6 демонстрируется использование первого прототипа AsOrdered. В отношении результатов операции AsParallel, примененной к последовательности имен президентов, выполняется операция AsOrdered.

Листинг 24.6. Использование первого прототипа `AsOrdered`

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt",
  "Taft", "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
IEnumerable<string> results = presidents
   .AsParallel()
   .AsOrdered()
   .Where(p => p.Contains('o'))
   .Select(p => p);
foreach (string president in results) {
   Console.WriteLine("Совпадение: {0}", president);
}
```

Запуск кода из листинга 24.6 дает показанные ниже результаты. Здесь видно, что порядок исходной последовательности предохранен.

```
Совпадение: Clinton
Совпадение: Coolidge
Совпадение: Eisenhower
Совпадение: Fillmore
Совпадение: Ford
Совпадение: Harrison
Совпадение: Hoover
Совпадение: Jackson
Совпадение: Jefferson
Совпадение: Johnson
Совпадение: Lincoln
Совпадение: Madison
Совпадение: Monroe
Совпадение: Nixon
Совпадение: Polk
Совпадение: Roosevelt
Совпадение: Taylor
Совпадение: Washington
Совпадение: Wilson
```

В листинге 24.7 применяется второй прототип. Он включен здесь только для полноты картины; обычно применять эту операцию к слабо типизированному `ParellelQuery` не придется.

Листинг 24.7. Использование второго прототипа `AsOrdered`

```
ArrayList list = new ArrayList() {
   "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
   "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
   "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson"};
IEnumerable<string> results = list
   .AsParallel()
   .AsOrdered()
   .Cast<string>()
   .Where(p => p.Contains('o'))
   .Select(p => p);
foreach (string president in results) {
   Console.WriteLine("Совпадение: {0}", president);
}
```

В этом листинге создается унаследованная коллекция, содержащая некоторые имена президентов. Затем вызывается метод AsParallel, который возвращает экземпляр ParallelQuery. После этого применяется операция AsOrdered, а также другие методы PLINQ, и вызывается операция Cast, чтобы трансформировать ParallelQuery в ParallelQuery<string>. После компиляции и запуска кода из листинга 24.7 получаются следующие результаты:

```
Совпадение: Clinton
Совпадение: Coolidge
Совпадение: Eisenhower
Совпадение: Fillmore
Совпадение: Ford
Совпадение: Harrison
Совпадение: Hoover
Совпадение: Jackson
```

AsUnordered

Операция AsUnordered отменяет действие операции AsOrdered. Это может пригодиться в запросах, состоящих из множества частей, когда нужно упорядочить одну часть, но избежать накладных расходов по упорядочиванию результатов другой части. Более подробно вопросы упорядочивания результатов рассматривались в предыдущей главе.

Прототипы

Операция AsUnordered имеет один прототип. Операция применяется к ParallelQuery<T>, который можно использовать в качестве основы для запроса PLINQ.

Прототип AsUnordered

```
public static ParallelQuery<T AsUnordered<T>(
   this ParallelQuery<T> source
)
```

Примеры

В листинге 24.8 демонстрируется использование операции AsUnordered в запросе PLINQ, состоящем из двух стадий.

Листинг 24.8. Смешанное упорядочивание в запросе PLINQ

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt",
  "Taft", "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
IEnumerable<string> results = presidents
   .AsParallel()
   .AsOrdered()
   .Where(p => p.Contains('o'))
   .Take(5)
   .AsUnordered()
   .Where(p => p.Contains('e'))
   .Select(p => p);
foreach (string president in results) {
   Console.WriteLine("Совпадение: {0}", president);
}
```

В коде сначала находятся все имена президентов, содержащие букву “o” с предохранением порядка результатов с помощью операции AsOrdered. Это значит, что соответствующие имена будут получены в алфавитном порядке, поскольку это порядок исходной последовательности данных. Результаты упорядочены, потому что необходимо получить первые пять соответствий, которые затем использовать для нахождения всех имен, содержащих букву “e”. Упорядочивать эту часть не нужно, поэтому вызывается операция AsUnordered для исключения накладных расходов PLINQ, связанных с сортировкой результатов. Скомпилировав и запустив код из листинга 24.8, получаем следующие результаты:

```
Совпадение: Fillmore
Совпадение: Coolidge
Совпадение: Eisenhower
```

AsSequential

Операция AsSequential — противоположность операции AsParallel. Она навязывает последовательное выполнение, преобразуя ParallelQuery<T> в IEnumerable<T>.

Прототипы

Операция AsSequential имеет один прототип, который оперирует с ParallelQuery<T> и возвращает IEnumerable<T>. Запросы, выполняемые на результате этой операции, будут последовательными.

Прототип операции AsSequential

```
public static IEnumerable<T> AsSequential<T>(
   this ParallelQuery<T> source
)
```

Примеры

Операция AsSequential больше всего нужна, когда требуется включать или отключать параллельное выполнение запроса с множеством частей. Пример приведен в листинге 24.9.

Листинг 24.9. Переход от параллельного к последовательному выполнению в запросе с множеством частей

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt",
  "Taft", "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
IEnumerable<string> results = presidents
   .AsParallel()
   .AsOrdered()
   .Where(p => p.Contains('o'))
   .Take(5)
   .AsSequential()
   .Where(p => p.Contains('e'))
   .Select(p => p);
foreach (string president in results) {
   Console.WriteLine("Совпадение: {0}", president);
}
```

Код в этом примере является вариантом кода из листинга 24.9. Для второй части запроса решено, что накладные расходы, связанные с параллельным выполнением, не оправданы, поскольку заранее известно, что требуется обработать всего пять элементов. По этой причине с помощью операции `AsSequential` было произведено переключение с PLINQ на LINQ, когда осуществляется выбор имен, содержащих букву “e”. Переключаться с параллельного на последовательное выполнение с использованием операций `AsParallel` и `AsSequential` можно столько раз, сколько необходимо. Ниже показаны результаты запуска кода из листинга 24.9. Поскольку последняя часть запроса выполнялась последовательно, результаты получаются в том же порядке, в котором расположены данные в исходной последовательности.

```
Совпадение: Coolidge
Совпадение: Eisenhower
Совпадение: Fillmore
```

AsEnumerable

Операция `AsEnumerable` дает тот же эффект, что и `AsSequential`. Она преобразует `ParallelQuery<T>` в `IEnumerable<T>` и тем самым навязывает последовательное выполнение запроса.

Прототипы

Операция `AsEnumerable` имеет один прототип.

Единственный прототип AsEnumerable

```
public static IEnumerable<T> AsSequential<T>(
   this ParallelQuery<T> source
)
```

Примеры

Код в листинге 24.10 идентичен коду в листинге 24.9 за исключением того, что операция `AsSequential` заменена операцией `AsEnumerable`.

Листинг 24.10. Использование операции AsEnumerable

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt",
  "Taft", "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};

IEnumerable<string> results = presidents
   .AsParallel()
   .AsOrdered()
   .Where(p => p.Contains('o'))
   .Take(5)
   .AsEnumerable()
   .Where(p => p.Contains('e'))
   .Select(p => p);

foreach (string president in results) {
   Console.WriteLine("Совпадение: {0}", president);
}
```

WithDegreeOfParallelism

Операция WithDegreeOfParallelism устанавливает верхний предел количества разделов, которые будут обработаны PLINQ одновременно. PLINQ разбивает исходную последовательность на *разделы*, которые затем обрабатываются одновременно. Дополнительную информацию по этому поводу ищите в предыдущей главе.

Механизм PLINQ анализирует машину, запрос и исходные данные, и решает, сколько разделов можно обработать сразу. Указать точное количество разделов, которое PLINQ будет использовать, нельзя, но можно задать верхний предел.

Прототипы

Операция WithDegreeOfParallelism имеет один прототип.

Прототип операции WithDegreeOfParallelism

```
public static ParallelQuery<T> WithDegreeOfParallelism<T>(
   this ParallelQuery<T> source,
   int degreeOfParallelism
)
```

Эта операция применяется к ParallelQuery<T> и принимает единственный целочисленный аргумент, в котором указывается верхний предел. Обратите внимание, что PLINQ может использовать более низкую степень параллелизма или даже выполнить запрос последовательно, если в результате анализа станет ясно, что параллельное выполнение, скорее всего, не даст ощутимого роста производительности. Указание предела 1 в этой операции приводит к последовательному выполнению запроса.

Примеры

В листинге 24.11 эта операция используется в стандартном запросе для нахождения имен президентов, содержащих букву “o”. Для операции WithDegreeOfParallelism указан аргумент 2, означающий, что, по крайней мере, две порции данных из исходной последовательности будут обработаны одновременно.

Листинг 24.11. Установка предела степени параллелизма

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt",
  "Taft", "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
IEnumerable<string> results = presidents
   .AsParallel()
   .WithDegreeOfParallelism(2)
   .Where(p => p.Contains('o'))
   .Select(p => p);
foreach (string president in results) {
   Console.WriteLine("Совпадение: {0}", president);
}
```

WithExecutionMode

Операция WithExecutionMode позволяет игнорировать анализ, выполненный механизмом PLINQ, и принудительно осуществлять параллельное выполнение, даже если производительность параллельного режима окажется хуже, чем у последовательного.

Прототипы

Операция WithExecutionMode имеет один прототип.

Прототип операции WithExecutionMode

```
public static ParallelQuery<T> WithExecutionMode<T>(
   this ParallelQuery<T> source,
   ParallelExecutionMode executionMode
)
```

Единственный аргумент операции — это значение из перечисления ParallelExecutionMode. Оно включает два варианта — Default (решение принимает PLINQ) и ForceParallelizm (означает параллельное выполнение независимо от результатов анализа запроса).

Примеры

В листинге 24.12 демонстрируется использование операции WithExecutionMode для принудительного параллельного выполнения.

Листинг 24.12. Принудительное параллельное выполнение

```
string[] presidents = {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
  "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt",
  "Taft", "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
IEnumerable<string> results = presidents
   .AsParallel()
   .WithExecutionMode(ParallelExecutionMode.ForceParallelism)
   .Where(p => p.Contains('o'))
   .Select(p => p);
foreach (string president in results) {
   Console.WriteLine("Совпадение: {0}", president);
}
```

WithMergeOptions

Операция WithMergeOptions позволяет управлять буферизацией результатов по мере их производства запросом. По умолчанию PLINQ создает буфер, хранящий несколько результирующих элементов, и поставляет их потребителю результата только при наполнении буфера. Это поведение можно изменить так, чтобы отправлять потребителю результаты только после их накопления или же передавать каждый результат немедленно.

Прототипы

Операция WithMergeOptions имеет один прототип. Операция применяется к экземплярам ParallelQuery<T> и принимает один аргумент — значение из перечисления ParallelMergeOptions.

Прототип операции WithMergeOptions

```
public static ParallelQuery<T> WithMergeOptions<T>(
   this ParallelQuery<T> source,
   ParallelMergeOptions mergeOptions
)
```

Перечисление ParallelMergeOptions включает четыре значения. Значение NotBuffered заставляет каждый результирующий элемент передаваться по мере получения. Значение FullBuffered обеспечивает ожидание, пока все результаты не будут получены, прежде чем передавать их потребителю. Значение AutoBuffered позволяет системе выбирать размер буфера и передавать элементы по мере наполнения буфера. Последнее из значений перечисления — Default — это то же самое, что и AutoBuffered.

Примеры

В листинге 24.13 содержится пример использования операции WithMergeOptions со значением FullBuffered из перечисления ParallelMergeOptions.

Листинг 24.13. Полная буферизация результатов PLINQ

```
IEnumerable<int> results = ParallelEnumerable.Range(0, 10)
   .WithMergeOptions(ParallelMergeOptions.FullyBuffered)
   .Select(i => {
      System.Threading.Thread.Sleep(1000);
      return i;
   });
Stopwatch sw = Stopwatch.StartNew();
foreach (int i in results) {
   Console.WriteLine("Значение: {0}, время: {1}", i, sw.ElapsedMilliseconds);
}
```

В этом примере к конструкции select запроса добавлено ожидание, чтобы перед обработкой каждого элемента созданной последовательности происходила задержка в 1 секунду. Затем результаты запроса перечисляются (что инициирует отложенное выполнение) и каждый элемент выводится на консоль. Для каждого вызова Console.Writeline выводится временная метка с использованием класса StopWatch, чтобы приблизительно видеть, сколько времени проходит между получением результирующих элементов.

Ниже показаны результаты запуска кода из листинга 24.13:

```
Значение: 0, время: 3013
Значение: 1, время: 3014
Значение: 2, время: 3014
Значение: 3, время: 3014
Значение: 4, время: 3014
Значение: 5, время: 3014
Значение: 6, время: 3014
Значение: 7, время: 3014
Значение: 8, время: 3014
Значение: 9, время: 3014
```

Взглянув на значение времени рядом с каждым результирующим значением, можно заметить, что проходит около трех секунд перед получением каких-либо элементов, но затем они все приходят одним блоком. Поскольку использовалась опция FullyBuffered, это именно то, чего следовало ожидать. Все результаты будут переданы, только когда все они будут обработаны.

Листинг 24.14. Запрос PLINQ без буферизации результатов

```
IEnumerable<int> results = ParallelEnumerable.Range(0, 10)
   .WithMergeOptions(ParallelMergeOptions.NotBuffered)
```

```
   .Select(i => {
      System.Threading.Thread.Sleep(1000);
      return i;
   });
Stopwatch sw = Stopwatch.StartNew();
foreach (int i in results) {
   Console.WriteLine("Значение: {0}, время: {1}", i, sw.ElapsedMilliseconds);
}
```

Скомпилировав и запустив код листинга 2414, получаются следующие результаты:

```
Значение: 6, время: 1012
Значение: 8, время: 1012
Значение: 0, время: 1013
Значение: 3, время: 1013
Значение: 1, время: 2012
Значение: 4, время: 2012
Значение: 7, время: 2012
Значение: 9, время: 2012
Значение: 2, время: 3012
Значение: 5, время: 3012
```

Это ожидаемый результат, хотя поначалу это может быть не очевидно. Стоит напомнить, что код этого примера был запущен на четырехядерной машине разработчика, и потому PLINQ смог обработать четыре раздела одновременно и одновременно произвести результаты. Поэтому то, что видно в выводе — это четыре результата, полученные примерно через секунду (вспомните о секундной задержке в конструкции `select`), секундой позже — еще четыре результата и спустя еще одну секунду получаются последние два элемента. Именно этого и следовало ожидать при выполнении запроса на четырехядерной машине с отключенной буферизацией.

Операции преобразования

Ранее уже упоминалось, что можно получить объект `ParallelQuery` из унаследованной коллекции с помощью метода `AsParallel`, но для получения объекта `ParallelQuery<T>`, который можно использовать с PLINQ, понадобится предпринять дополнительные действия. В этом разделе рассматриваются операции, позволяющие выполнять нужные преобразования.

Cast

Операция `Cast` преобразует `ParallelQuery` в `ParallelQuery<T>`. Должен быть указан тип, и если во входной последовательности есть какие-то элементы, которые не относятся к типу `T`, будет сгенерировано исключение.

Прототипы

Операция `Cast` имеет один прототип. Чтобы создать `ParallelQuery<string>`, необходимо вызвать `Cast<string>()`. Если нужен `ParallelQuery<MyObject>`, следует вызвать `Cast<MyObject>()`.

Прототип операции `Cast`

```
public static ParallelQuery<T> Cast<T>(
   this ParallelQuery source
)
```

Примеры

В листинге 24.15 демонстрируется применение операции Cast для использования унаследованной коллекции в качестве источника для запроса PLINQ. Операция AsParallel применяется к ArrayList и затем вызывается Cast<string> для создания объекта ParallelQuery<string>, используемого в запросе PLINQ.

Листинг 24.15. Приведение унаследованного источника данных

```
ArrayList list = new ArrayList() {
  "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
  "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
  "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson"};
IEnumerable<string> results = list
   .AsParallel()
   .Cast<string>()
   .Where(p => p.Contains('o'))
   .Select(p => p);
foreach (string president in results) {
   Console.WriteLine("Совпадение: {0}", president);
}
```

Компиляция и запуск кода из листинга 24.15 дает следующие результаты:

```
Совпадение: Clinton
Совпадение: Eisenhower
Совпадение: Coolidge
Совпадение: Fillmore
Совпадение: Ford
Совпадение: Harrison
Совпадение: Hoover
Совпадение: Jackson
```

OfType

Операция OfType создает ParallelQuery<T> из ParallelQuery, выбирая только те элементы последовательности, которые относятся к типу T. Это позволяет выборочно использовать элементы из унаследованной коллекции, содержащей смешанные типы, не беспокоясь об исключениях, которые могут возникнуть в случае применения операции Cast.

Прототипы

Операция OfType имеет один прототип.

Прототип операции OfType

```
public static ParallelQuery<T> OfType<T>(
   this ParallelQuery source
)
```

Примеры

В листинге 24.16 содержится пример использования этой операции. Выбираемый тип указывается в угловых скобках. Если, например, необходимо получить ParallelQuery<string>, то следует вызвать OfType<string>. В этом листинге создается унаследованная коллекция, содержащую смесь типов, и затем используется операция OfType для создания экземпляра ParallelQuery<string>, который содержит строковые значения из коллекции. После этого объект ParallelQuery<string> применяется в запросе PLINQ.

Листинг 24.16. Использование операции `OfType`

```
ArrayList list = new ArrayList();
list.Add("Adams");
list.Add(23);
list.Add("Arthur");
list.Add(DateTime.Now);
list.Add("Buchanan");
list.Add(new string[] { "apple", "orange" });
IEnumerable<string> results = list
   .AsParallel()
   .OfType<string>()
   .Select(p => p);
foreach (string president in results) {
   Console.WriteLine("Совпадение: {0}", president);
}
```

Операция `ForAll`

Операция `ForAll` уникальна для PLINQ и не имеет эквивалента в LINQ to Objects, поэтому ей посвящен отдельный раздел. Эта операция позволяет указывать действие, которое будет выполнено над каждым элементом в исходной последовательности данных при выполнении запроса.

Прототипы

У операции `ForAll` один прототип. Аргументом является экземпляр `System.Action`, который будет выполнен над каждым элементом последовательности. Возвращать результирующее значение из `Action` нельзя.

Прототип операции `ForAll`

```
public static void ForAll<T>(
   this ParallelQuery<T> source,
   Action<T> action
)
```

Примеры

Дополнительную информацию и примеры применения операции `ForAll` ищите в главе 23.

Резюме

В этой главе рассматривались ключевые операции создания экземпляров `ParallelQuery` и управления выполнением PLINQ. Одним из преимуществ PLINQ является то, что по большей части это — прямая замена LINQ to Objects. Если нужно параллельное выполнение, достаточно просто вызвать операцию `AsParallel`.

За счет гибкости, удобства и интеграции в .NET Framework технологии PLINQ и в целом LINQ превращаются в средство, замечательно дополняющее язык. Практически в каждом случае использование LINQ во всех его формах позволяло быстро и легко решать проблемы, решение которых было трудоемким и чреватым ошибками во времена, предшествовавшие появлению LINQ. Мы надеемся, что вы чувствуете то же самое, и что эта книга помогла понять, оценить и даже полюбить LINQ.

Предметный указатель

D

DOM (Document Object Model), 182

I

IL (Intermediate Language), 37

L

LINQ (Language Integrated Query), 23
LINQ to DataSet, 22; 320; 350
LINQ to Entities, 22; 559; 593
LINQ to Objects, 22; 62
LINQ to SQL, 22; 358; 368; 390; 539
LINQ to XML, 19; 20; 22; 182; 187; 260; 285

O

Object Relational Designer, 376; 387; 539

P

PLINQ (Parallel LINQ), 614; 619; 631

S

SQLMetal, 539

X

XML (Extensible Markup Language), 19
  аннотации XML, 251
  модификация данных XML, 232
  события XML, 254
  трансформации XML, 293
XML-документ, 216
XPath, 316
XSLT (eXtensible Stylesheet Language Transformations), 293

А

Агрегация, 168
Аннотации XML, 251
Ассоциация (association), 361; 553
  добавление новой ассоциации, 382
Атрибут
  добавление, 247
  модификация, 247
  обновление, 249
  обход атрибутов, 244
  создание, 244
  удаление, 247

В

Выражение
  дерево выражения (expression tree), 37
  запроса, 49
    грамматика выражений запросов, 51
    трансляция выражений запросов, 51
  лямбда (lambda expression), 25; 32

Г

Граф
  согласованность графа, 434
Группирование, 111

Д

Данные
  совместимость типов данных, 455
Дерево выражения (expression tree), 37
Документ
  XML, 216

З

Загрузка
  “ленивая”, 569
  немедленная, 400; 403; 570
  отложенная, 398; 401
  явная, 572
Запись
  вставка, 560
    в базу данных, 390
  удаление, 418; 580
Запрос
  LINQ to SQL, 393; 424
  PLINQ, 619; 623
  базовый, 566
  операции запросов, 68
  отложенный, 28; 197; 411
  параллельный, 621
  с использованием LINQ to XML, 20
  скомпилированный, 566
  сложный, 288

И

Идентификатор
  прозрачный, 52
Инициализация
  коллекций, 41
  объектов, 40
Интерфейс
  IComparer<T>, 95
  IDisposable, 472
  IEnumerable<T>, 63
  IExecuteResult, 465
  IMultipleResults, 467
  IQueryable<T>, 365; 556

ISingleResult<T>, 466
Исключение
обработка исключений, 625

К

Квантификаторы, 163
Класс
DataContext, 360; 469; 472; 479; 543
DataLoadOptions, 402; 405
EntityCollection, 607
EntityObject, 600
EntityReference, 606
EntityRef<T>, 463
EntitySet<T>, 463
Extensions, 316
Object, 562
ObjectContext, 552; 593
XAttribute, 202; 244
XCData, 211
XComment, 203
XContainer, 203
XDeclaration, 204
XDocument, 206
XDocumentType, 204
XElement, 200
XName, 206
XNamespace, 207
XNode, 207
XObject, 251
XProcessingInstruction, 207
XStreamingElement, 209; 210
XText, 210
сущностный (entity), 428
наследование, 541
расширение с помощью частичных методов, 461
Ключевое слово
var, 26; 38
Коллекция
инициализация коллекций, 41
Конфликты
обнаружение конфликтов, 523
разрешение конфликтов, 527

Л

Лямбда-выражение (lambda expression), 25; 32

М

Метод
анонимный, 34
расширяющий (extension), 43; 46
вызов, 46
объявление, 46
приоритет, 47
частичный (partial), 47
Множества, 116
Модификация данных XML, 232

Н

Наследование
добавление нового наследования, 382

О

Обновление
ассоциированных объектов, 579
отношения ассоциированных типов, 579
простое обновление сущностного объекта, 578
Объект, 43
инициализация объектов, 40
Объектная модель LINQ to XML, 196
Операции
LINQ to XML, 260
Parallel LINQ (PLINQ), 631
Операция
Aggregate, 178
All, 164
Ancestors, 261
AncestorsAndSelf, 265
Any, 163
AsEnumerable, 125; 344; 640
AsOrdered, 636
AsSequential, 639
AsUnordered, 638
Attributes, 267
Average, 176
Cast, 121; 644
Concat, 90
Contains, 166; 414
CopyToDataTable, 344
Count, 168
DefaultIfEmpty, 128
DescendantNodes, 269
DescendantNodesAndSelf, 270
Descendants, 272
DescendantsAndSelf, 274
Distinct, 116; 323
ElementAt, 161
ElementAtOrDefault, 162
Elements, 276
Empty, 133
Except, 120; 326
Field, 336
First, 152
FirstOrDefault, 153
ForAll, 646
GroupBy, 111
GroupJoin, 109
InDocumentOrder, 278

Intersect, 118; 328
Join, 107
Last, 155
LastOrDefault, 156
LongCount, 169
Max, 174
Min, 172
Nodes, 280
OfType, 123; 222; 645
OrderBy, 93
OrderByDescending, 97
Range, 131
Remove, 281
Reverse, 106
Select, 75
SelectMany, 80
SequenceEqual, 149; 331
SetField, 341
Single, 158
SingleOrDefault, 159
Skip, 87
SkipWhile, 88
Sum, 170
Take, 84
TakeWhile, 85
ThenBy, 100
ThenByDescending, 103
ToArray, 138
ToDictionary, 141
ToList, 139
ToLookup, 145
Union, 117; 330
Where, 73
WithDegreeOfParallelism, 641
WithExecutionMode, 641
WithMergeOptions, 642
агрегатная, 168
генерации, 131
группирования, 111
запроса, 68
-квантификатор, 163
конкатенации, 90
множества, 116
не отложенная (nondeferred), 135
ограничения (restriction), 73
отложенная, 71
преобразования, 121
проекции, 75
разбиения (partitioning), 83
соединения (join), 107
упорядочивания, 92
эквивалентности, 149
элемента, 127; 151

П

Параллелизм
  оптимистический, 523
  пессимистический, 533
  принудительное параллельное выполнение, 624
Последовательность (sequence), 21
Предикат (predicate), 45
Проблема Хэллоуина, 197

Р

Распаковка (unboxing), 333

С

События XML, 254
Соединения
  внешние, 407; 576
  внутренние, 407; 576
  случайные, 406

Т

Тип данных
  анонимный, 42
  совместимость типов данных, 455
Транзакция, 546
Трансляция SQL, 424
Трансформации XML, 293

У

Удаление
  записи, 580
  каскадное, 586
  связанных объектов, 581; 583
Упаковка (boxing), 333
Упорядочивание, 405
Утилита
  SQLMetal, 370

Ф

Файл
  DBML, 375
Фильтрация, 405
Форум, посвященный LINQ, 31

Я

Язык
  C#, 32
  IL, 37
  XML, 19
  XSLT, 293
  интегрированных запросов (LINQ), 23

VISUAL C# 2010
ПОЛНЫЙ КУРС

Карли Уотсон
и др.

www.dialektika.com

Этот исчерпывающий источник для начинающих поможет подготовиться к работе с новым выпуском языка программирования C#. Книга предлагает исчерпывающее описание синтаксиса C#. Читатели узнают о таких фундаментальных основах, как переменные, управление потоком выполнения и объектно-ориентированном программировании. Кроме того, будет показано, как строить Windows- и веб-приложения, формы Windows и работать с данными. Пошаговые упражнения позволят лучше усвоить материал, предлагаемый в каждой главе. Прочтя эту книгу, читатели сразу же смогут приступить к написанию собственного кода для решения разнообразных реальных задач.
Книга рассчитана на начинающих программистов, а также будет полезна студентам и преподавателям дисциплин, связанных с программированием и разработкой для .NET.

ISBN 978-5-8459-1699-0 **в продаже**

VISUAL C++ 2010
ПОЛНЫЙ КУРС

Айвор Хортон

www.dialektika.com

По существу, в этой книге рассматриваются две обширные темы: язык программирования C++ и программирование приложений Windows с использованием MFC или .NET Framework. Прежде чем вы сможете разработать полнофункциональное приложение Windows, необходимо приобрести хороший уровень знаний языка C++, поэтому упражнения здесь на первом месте.

В первой части книги поэтапно изложены основные темы программирования на языке C++. Вы изучите синтаксис и использование базового языка C++, а также приобретете уверенность и опыт применения его на практике. Модификацию C++/CLI базового языка C++ вы также изучите на практических примерах.

Кроме того, вы узнаете о мощных инструментальных средствах, предоставляемых стандартной библиотекой шаблонов STL, для базовой версии языка C++ и версии C++/CLI. Одна из глав посвящена библиотеке шаблонов для параллельных вычислений, которая позволяет использовать мощь многоядерных PC для приложений с интенсивными вычислениями.

ISBN 978-5-8459-1698-3 **в продаже**

WCF 4
WINDOWS COMMUNICATION FOUNDATION И .NET 4
ДЛЯ ПРОФЕССИОНАЛОВ

Пабло Сибраро и др.

www.dialektika.com

В этой книге рассказывается о платформе Windows Communication Foundation в .NET 4.0. Платформа WCF является технологией .NET, используемой при создании приложений, предназначенных для широкого круга запросов, обмена сообщениями в различных коммуникационных случаях и выполнения потоков задач, состоящих из действий служб. Эта новая книга поможет понять принцип ориентации на службы, изучить примеры коммуникации и научиться описывать бизнес-процессы декларативно. Вы также научитесь эффективно использовать среду разработки Visual Studio 2010 для создания решений, в которых максимально применяются новые возможности платформы WCF 4.0.

В данной книге описывается взаимодействие разработчика и архитектора при создании приложений, интегрирующихся в новую парадигму программирования платформы WCF 4.0. Вы также узнаете, как установить решение в этой новой архитектурной форме, основанной на платформе WCF 4.0, с помощью технологий и служб .NET.

ISBN 978-5-8459-1713-3 **в продаже**

WPF
WINDOWS PRESENTATION FOUNDATION В .NET 4 С ПРИМЕРАМИ НА C# 2010 ДЛЯ ПРОФЕССИОНАЛОВ

Мэтью Мак-Дональд

www.williamspublishing.com

Книга ведущего специалиста в области технологий .NET представляет собой учебное и справочное пособие по технологии WPF, входящей в состав .NET 4, для разработчиков высококлассных приложений, которые ориентированы на семейство операционных систем семейства Windows. Данная технология позволяет сочетать в рамках приложения пользовательский интерфейс, документы и медиа-содержимое, располагая при этом полной поддержкой со стороны операционной системы.
Функциональность WPF расширена поддержкой Tablet PC и других форм устройств ввода. Теперь WPF предоставляет более развитый конвейер рисования и печати, инфраструктуру доступности и автоматизации пользовательских интерфейсов, управляемые данными интерфейсы и виртуализацию, а также точки интеграции приложений с командной оболочкой Windows. В этой книге описано, как в действительности работает WPF.

ISBN 978-5-8459-1657-0

в продаже

ЯЗЫК ПРОГРАММИРОВАНИЯ C# 2010 И ПЛАТФОРМА .NET 4

5-е издание

Эндрю Троелсен

www.williamspublishing.com

Версия .NET 4 привнесла множество новых API-интерфейсов в библиотеках базовых классов, а также новых синтаксических конструкций в языке C#.

В этой книге вы найдете полное описание всех нововведений в характерной для автора дружественной к читателю манере. Помимо общих вопросов, подробно рассматривается среда Dynamic Language Runtime (DLR); библиотека Task Parallel Library (TPL, включая PLINQ); технология ADO.NET Entity Framework (а также LINQ to EF); расширенное описание API-интерфейса Windows Presentation Foundation (WPF); улучшенная поддержка взаимодействия с COM.

В книге рассматриваются следующие темы

- Особенности платформы .NET 4 и языка Visual C# 2010
- Детали технологии .NET – лидера в производстве современного программного обеспечения
- Полезные советы по разработке от эксперта в .NET, который изучает эту платформу, начиная с ее первой версии
- Полное описание технологий WPF, WCF и WF, поддерживаемых ядром платформы .NET

ISBN 978-5-8459-1682-2

в продаже

C# 4.0
ПОЛНОЕ РУКОВОДСТВО

Герберт Шилдт

www.williamspublishing.com

В этом полном руководстве по C# 4.0 — языку программирования, разработанному специально для среды .NET, — детально рассмотрены все основные средства языка: типы данных, операторы, управляющие операторы, классы, интерфейсы, методы, делегаты, индексаторы, события, указатели, обобщения, коллекции, основные библиотеки классов, средства многопоточного программирования и директивы препроцессора. Подробно описаны новые возможности C#, в том числе PLINQ, библиотека TPL, динамический тип данных, а также именованные и необязательные аргументы. Это справочное пособие снабжено массой полезных советов авторитетного автора и сотнями примеров программ с комментариями, благодаря которым они становятся понятными любому читателю независимо от уровня его подготовки.
Книга рассчитана на широкий круг читателей, интересующихся программированием на C#.

ISBN 978-5-8459-1684-6 **в продаже**

C# 4 И ПЛАТФОРМА .NET 4 ДЛЯ ПРОФЕССИОНАЛОВ

Кристиан Нейгел
Билл Ивьен
Джей Глинн
Карли Уотсон
Морган Скиннер

www.dialektika.com

Книга известных специалистов в области разработки приложений с использованием .NET Framework посвящена программированию на языке C# 2010 в среде .NET Framework 4 и в предшествующих версиях. Книгу отличает простой и доступный стиль изложения, изобилие примеров и рекомендаций по написанию высококачественных программ. Подробно рассматриваются такие вопросы, как основы языка программирования C#, организация среды .NET, работа с данными, написание Windows- и веб-приложений, взаимодействие через сеть, создание веб-служб и многое другое. Немалое внимание уделено проблемам безопасности и сопровождения кода. Тщательно подобранный материал позволит без труда разобраться с тонкостями использования Windows Forms и построения веб-страниц. Читатели ознакомятся с работой в Visual Studio 2010, а также с применением различных технологий, встроенных в .NET. Книга рассчитана на программистов разной квалификации.

ISBN 978-5-8459-1656-3

в продаже