*Включает приемы разработки с применением технологии AJAX*

# Dojo

*Подробное руководство*

O'REILLY®

*Мэтью А. Расселл*

По договору между издательством «Символ-Плюс» и Интернет-магазином «Books.Ru – Книги России» единственный легальный способ получения данного файла с книгой ISBN 978-5-93286-151-6, название «Dojo. Подробное руководство» – покупка в Интернет-магазине «Books.Ru – Книги России». Если Вы получили данный файл каким-либо другим образом, Вы нарушили международное законодательство и законодательство Российской Федерации об охране авторского права. Вам необходимо удалить данный файл, а также сообщить издательству «Символ-Плюс» (piracy@symbol.ru), где именно Вы получили данный файл.

# Dojo

## The Definitive Guide

*Matthew A. Russell*

O'REILLY®

# Dojo

## Подробное руководство

*Мэтью А. Расселл*

Символ®

*Санкт-Петербург — Москва*
*2009*

Мэтью А. Расселл

# Dojo. Подробное руководство

Перевод А. Киселева


| | |
|---|---|
| Главный редактор | *А. Галунов* |
| Зав. редакцией | *Н. Макарова* |
| Выпускающий редактор | *П. Щеголев* |
| Редактор | *Ю. Бочина* |
| Корректор | *О. Макарова* |
| Верстка | *Д. Орлова* |

*Расселл М.*

Dojo. Подробное руководство – Пер. с англ. – СПб.: Символ-Плюс, 2009. – 560 с., ил.

ISBN 978-5-93286-151-6



Dojo – это высоконадежный инструментарий JavaScript, позволяющий быстрее и проще создавать веб-приложения и сайты, основанные на применении JavaScript или технологии Ajax. Это издание представляет собой наиболее полный сборник документации по инструментарию Dojo, снабженный развернутыми комментариями. Демонстрируются эффективные приемы работы с обширным набором утилит, реализация различных пользовательских механизмов, методы воспроизведения анимационных эффектов. Также рассматриваются проекты, входящие в состав библиотеки DojoX, инструменты сборки и платформы модульного тестирования.

Книга предназначена для разработчиков, уже имеющих некоторый опыт работы с технологиями JavaScript и Ajax. Использование Dojo поможет эффективнее воплощать новые идеи по созданию интерактивных веб-приложений, значительно разнообразить интерфейс и предоставить пользователю намного больше удобств в работе.



**ISBN 978-5-93286-151-6**
**ISBN 978-0-596-51648-2 (англ)**



© Издательство Символ-Плюс, 2009

Authorized translation of the English edition © 2008 O'Reilly Media, Inc. This translation is published and sold by permission of O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

Все права на данное издание защищены Законодательством РФ, включая право на полное или частичное воспроизведение в любой форме. Все товарные знаки или зарегистрированные товарные знаки, упоминаемые в настоящем издании, являются собственностью соответствующих фирм.



Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 324-5353, www.symbol.ru. Лицензия ЛП N 000054 от 25.12.98.

Налоговая льгота – общероссийский классификатор продукции
ОК 005-93, том 2; 953000 – книги и брошюры.

Подписано в печать 27.03.2009. Формат 70×100 1/16. Печать офсетная.
Объем 35 печ. л. Тираж 1500 экз. Заказ №

Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12.

*Эта книга посвящается всем веб-разработчикам,*
*кто лишался сна из-за нелепых различий*
*между броузерами.*

# Оглавление

**Вступительное слово** . . . . . 11

**Предисловие** . . . . . 14

**I. Base и Core** . . . . . 35

**1. Обзор комплекта инструментальных средств** . . . . . 37

Обзор архитектуры Dojo . . . . . 37
Подготовка к работе . . . . . 42
Терминология . . . . . 49
Самонастройка . . . . . 52
Исследование Dojo с помощью Firebug . . . . . 60
В заключение . . . . . 71

**2. Утилиты броузера** . . . . . 72

Поиск узлов DOM . . . . . 72
Определение типа . . . . . 73
Утилиты для работы со строками . . . . . 74
Обработка массивов . . . . . 75
Управление исходным программным кодом с помощью модулей . . . . . 81
Утилиты для работы с объектами JavaScript . . . . . 90
Манипулирование контекстом объекта . . . . . 94
Утилиты для работы с деревом DOM . . . . . 99
Утилиты для работы с броузером . . . . . 106
В заключение . . . . . 111

**3. Обработчики событий и организация взаимодействий по подписке** . . . . . 113

Нормализация событий и клавиатуры . . . . . 113
Обработчики событий . . . . . 116
Организация взаимодействий по подписке . . . . . 123
В заключение . . . . . 127

**4. Технология AJAX и взаимодействие с сервером** . . . . . 128

Краткий обзор AJAX . . . . . 128
Работать с AJAX просто . . . . . 131
Объекты Deferred . . . . . 139
Утилиты для работы с формами и HTTP . . . . . 149
Межсайтовый скриптинг с использованием JSONP . . . . . 150
Модуль IO библиотеки Core . . . . . 152
Вызов удаленных процедур . . . . . 161
OpenAjax Hub . . . . . 164
В заключение . . . . . 165

**5. Манипулирование узлами** . . . . . 166

Поиск: универсальная реализация . . . . . 166
NodeList . . . . . 174
Создание расширений для NodeList . . . . . 185
Модуль Behavior . . . . . 187
В заключение . . . . . 191

**6. Интернационализация (i18n)** . . . . . 192

Введение . . . . . 192
Интернационализация модуля . . . . . 193
Даты, числа и денежные суммы . . . . . 197
В заключение . . . . . 201

**7. Перетаскивание элементов** . . . . . 202

Перетаскивание . . . . . 202
Сброс . . . . . 215
В заключение . . . . . 225

**8. Анимация и специальные эффекты** . . . . . 226

Анимация . . . . . 226
Core fx . . . . . 240
Анимация + «перетащил и бросил» = забавно! . . . . . 249
Цвета . . . . . 251
В заключение . . . . . 260

**9. Абстракция данных** . . . . . 261

Изменение схемы работы с данными . . . . . 261
Обзор прикладного интерфейса доступа к данным . . . . . 262
Интерфейсы . . . . . 263
Основные реализации интерфейсов доступа к данным . . . . . 272
В заключение . . . . . 290

**10. Имитация классов и наследование** . . . . . 291

JavaScript – это не Java . . . . . 291
Одна проблема, множество решений . . . . . 293
Имитация классов с использованием средств Dojo . . . . . 296
Множественное наследование посредством смешивания классов . . . . . 307
В заключение . . . . . 312

**II. Dijit и Util** . . . . . 313

**11. Обзор Dijit** . . . . . 315

Причины появления Dijit . . . . . 315
Доступность . . . . . 319
Библиотека Dijit для дизайнеров . . . . . 323
Парсер . . . . . 330
Практика Dijit на примере NumberSpinner . . . . . 334
Обзор базовых диджитов . . . . . 340
Функции прикладного интерфейса библиотеки Dijit . . . . . 345
В заключение . . . . . 347

**12. Анатомия Dijit и жизненный цикл** . . . . . 348

Анатомия библиотеки Dijit . . . . . 348
Методы управления жизненным циклом диджита . . . . . 351
Ваш первый диджит: HelloWorld . . . . . 360
Организация отношений родитель–потомок с помощью классов _Container и _Contained . . . . . 371
Быстрое создание прототипов виджетов в разметке . . . . . 372
В заключение . . . . . 374

**13. Виджеты форм** . . . . . 376

Обзор элементов управления форм . . . . . 376
Диджиты форм . . . . . 380
Разновидности TextBox . . . . . 383
FilteringSelect . . . . . 406
MultiSelect . . . . . 407
Разновидности Textarea . . . . . 408
Разновидности Button . . . . . 409
Slider . . . . . 417
Form . . . . . 423
В заключение . . . . . 425

**14. Виджеты компоновки** . . . . . 426

Общие особенности диджитов компоновки . . . . . 426
ContentPane . . . . . 429
BorderContainer . . . . . 433
StackContainer . . . . . 439
TabContainer . . . . . 441
AccordionContainer . . . . . 444
Проблема видимости и отображения . . . . . 445
В заключение . . . . . 446

**15. Виджеты приложения** . . . . . 447

Tooltip . . . . . 447
Виджеты диалогов . . . . . 449
ProgressBar . . . . . 453
ColorPalette . . . . . 456
Toolbar . . . . . 457
Menu . . . . . 459
TitlePane . . . . . 464
InlineEditBox . . . . . 466
Tree . . . . . 468
Editor . . . . . 482
В заключение . . . . . 490

**16. Инструменты сборки, тестирования и вопросы подготовки к выпуску в эксплуатацию** . . . . . 491

Сборка . . . . . 491
Платформа тестирования Dojo (DOH) . . . . . 504
Тестирование в броузере . . . . . 510
Вопросы производительности . . . . . 512
В заключение . . . . . 515

**A. Учебник по работе с отладчиком Firebug** . . . . . 516

**B. Краткий обзор DojoX** . . . . . 529

**Алфавитный указатель** . . . . . 534

# Вступительное слово

По-правде говоря, именно из-за DHTML меня выкинули из колледжа.

Как сейчас помню моменты, когда в 3 часа ночи бесконечные копания в документации MSDN, в спецификациях W3C и в мириадах сообщений в конференции *comp.lang.javascript* приводили к озарениям: «а что, если...». Все эти маленькие открытия как выжженные клейма зудели в моем мозгу, пока я перебирал все способы заставить броузер делать то, что мне требовалось. В те времена существовало лишь небольшое сообщество энтузиастов, занимавшихся тем же. Стремясь опередить друг друга, они описывали на форумах DHTMLCentral новые приемы программирования и отсылали новые компоненты, позволяющие обеспечить работоспособность тех или иных алгоритмов в броузерах Netscape. К 7 часам утра у латинских спряжений и бесконечных лекций по Java™ не оставалось шансов перед истинной красотой замыканий и тем более перед полным пониманием механизма наследования прототипов. Даже Рождественские каникулы я посвящал изучению и программированию на языке JavaScript. Я знаю, моя девушка и мои родители очень беспокоились за меня, но ничего не говорили мне при этом. Из пепла моей несостоявшейся академической карьеры выросли понимание идей движения за распространение программного обеспечения с открытыми исходными текстами (*http://opensource.org*), крепкая дружба и, в конечном счете, Dojo.

Спустя годы характер работы с DHTML изменился. Мы уже изучили большую часть подвохов, которых можно было ожидать от отдельных броузеров, и научились использовать те области, где поведение различных броузеров совпадает, – достаточно взглянуть на разнообразие библиотек в Dijit и DojoX. Теперь труд разработчика DHTML/Ajax заключается в том, чтобы спрессовать доступные технологии в службы, предназначенные для пользователей и разработчиков, так чтобы они одинаково хорошо отвечали потребностям конечных пользователей и разработчиков. История Dojo – это история такого перехода. Даже самая замечательная архитектура будет отвергнута, если она не в состоянии улучшить жизнь пользователя. Точно так же даже самые замечательные графические интерфейсы и эффекты не будут приняты нами, если они тяжелы в сопровождении, не соответствуют образу мыслей разработчиков и осложняют взаимодействие между дизайнерами и разработчиками. Все мы, кто вовлечен в разработку Dojo, росли

вместе с развитием Всемирной паутины, и с выходом версии Dojo 1.0 и этой книги можно смело сказать, что набор инструментальных средств Dojo полностью готов к эксплуатации. Теперь в планах, с которых мы когда-то начинали, напротив всех пунктов поставлены галочки. Уже существуют сайты, обслуживающие миллиарды обращений в месяц, которые опираются на Dojo для удовлетворения ожиданий пользователей, и крупные коллективы дизайнеров и разработчиков, совместно работающих над этим.

Все эти достижения не являются плодом труда одного человека или даже маленького коллектива. Количество людей, способствовавших развитию Dojo, веривших в проект и работавших вместе, чтобы еще больше улучшить Веб, слишком велико, чтобы можно было упомянуть каждого. Мы скопировали самые лучшие, на наш взгляд, структурные элементы из других проектов, результатом чего явились единая конкурентная среда и правила игры, одинаково справедливые для пользователей, добровольных помощников и спонсоров. Dojo может служить ярким доказательством того, что идея открытого программного обеспечения не только представляет собой удобный способ распространения для закрытых систем, но и того, что открытые проекты могут процветать, когда они следуют политике, вызывающей доверие пользователей, и когда разработчики проекта доверяют друг другу. Я горжусь, что при всех технических достижениях, воплощенных в инструментарии, нам удается сохранять проект открытым, принимая всех, кто желает присоединиться к нам, и не меняя правила игры. Мы развиваем проект таким образом, что содействовать ему можно не только добавлением программного кода. Еще одна задача проекта – поднять разработку открытого программного обеспечения до уровня коллегиального, цивилизованного обсуждения проблем. Проект предназначен для развития силами сообщества и, кроме того, не ставит своей целью размежевать разработчиков и пользователей. Мы и есть пользователи, и эта книга стремится разъяснить философию программирования с открытым кодом, лежащую в основе инструментария, и согласно которой мы предлагаем всем читателям оказывать нам помощь, чтобы продолжить дальнейшее развитие.

К тому времени, когда я встретился с Мэтью Расселлом (Matthew Russell), книга была почти завершена. В мире открытого программного обеспечения часто так бывает – вы можете годами работать с человеком, общаться с ним в почтовых рассылках и по каналам IRC, но полное представление о нем складывается, только когда вам удастся поговорить с ним о мирском и волнующем за добрым местным элем (или, в крайнем случае, за кружкой Гиннеса). Так было и с Мэтью; мы встретились с ним для обмена мнениями в старом маленьком тихом пабе в районе Норт Бич (North Beach) города Сан-Франциско, и тогда мне стало ясно: глубина его познаний, живой интерес и способность говорить с вами на вашем уровне – все это черты великого учителя. По мере того как я просматривал рукописи глав, я вдруг обнаружил, что

мне постоянно приходится удалять свои собственные недавно внесенные критические замечания, просто потому что Мэтью выстраивал понятия в обоснованном, правильном порядке. Его изложение делает библиотеку Dojo легкодоступной, дружественной и полезной. Ощущение удовольствия от открытий, которое возникает, когда начинаешь говорить с Мэтью лично, получаешь в полной мере и при чтении этой книги.

Примерно то же происходило со мной в течение последних четырех лет, пока я активно участвовал в обсуждениях на каналах IRC и в почтовых рассылках. В мире свободного программного обеспечения каждый человек входит в ваш мир в виде технической проблемы, которую требуется решить, ошибки, которую необходимо исправить, или особенности, над добавлением которой следует подумать. Только позднее становится ясно, что это за люди, с которыми вы работаете, – и это практически всегда захватывающее дух открытие! Доброта, бескорыстный труд и талант значительно затмеваются теми жертвами, которые приносятся каждым человеком, который старается стать частью проекта. Книга Мэтью – это дань уважения замечательной команде, с которой мне посчастливилось работать.

Не могу сказать, что я рекомендую бросить колледж ради работы, за которую никто не платит, но если эта работа зажжет в вас огонь интереса, не игнорируйте ее. Если она приведет вас к людям, которые хотя бы наполовину столь же замечательны, как те, которые встречались мне и которых я теперь считаю своими друзьями, это достойная цена всех бессонных ночей.

Алекс Расселл (Alex Russell)

*Сооснователь проекта Dojo Toolkit*
*и президент Dojo Foundation*

# Предисловие

Теперь пользователям требуются веб-приложения, которые выглядят точно так же, как и обычные настольные приложения. Домашние компьютеры давно стали обычным делом, веб-броузеры представляют собой высокоэффективную платформу, поэтому практически любой человек на планете является потенциальным конечным пользователем. Разработчики программного обеспечения тратят больше времени, чем когда-либо, чтобы перенести свои приложения в броузеры и предоставить доступ к ним миллионам – одни стремятся зачерпнуть свою пригоршню из многомиллиардной рекламной волны, а другие стремятся зарабатывать на элегантности и удобстве приложений, доведенных до такого совершенства, что люди готовы платить за доступ к ним.

Конечно, сам факт, что веб-броузер является высокоэффективной платформой, не означает, что он является идеальной платформой – по крайней мере, не в текущей реализации. Политика корпораций, отсутствие унификации в реализации различных спецификаций для веб-броузеров, а также извилистое развитие протоколов и стандартов за последние два десятилетия сделали развертывание приложений в разных веб-броузерах намного более сложным делом, чем кто-либо когда-либо мог предположить.

Но в мире, где потребности заставляют искать выход и изобретать, всегда есть надежда.

К счастью, богатые и мощные возможности языка JavaScript позволяют манипулировать веб-страницами, изменять и дополнять их на лету, образуя тем самым изолирующую прослойку, защищающую разработчика от лязгающих деталей внутреннего механизма веб-броузеров, – причем от всех броузеров одновременно.

Эта книга рассказывает о Dojo, библиотеке инструментальных средств на языке JavaScript, которая обеспечивает этот уровень изоляции между вами и различными несовместимостями, существующими между броузерами, и позволяет эффективно использовать JavaScript и другие веб-технологии везде, где их стоит использовать, а не пытаться построить ломкий поверхностный слой повторной реализации или обходных путей. Библиотека Dojo будет значительным дополнением к проекту, уже использующему YUI![1], или к любой другой серверной

[1] *http://developer.yahoo.com/yui/*

платформе, способной извлекать преимущества от передачи части работы на сторону клиента.

Dojo содержит стандартную библиотеку JavaScript, которая всегда нужна, набор настраиваемых элементов управления HTML и приемов использования CSS, которые приходится реализовывать снова и снова, а также средства сборки и модульного тестирования, которые становятся очень удобными, когда наступает время переносить свои разработки в рабочее окружение. Dojo – это не просто библиотека JavaScript, Dojo – это *уникальная* библиотека JavaScript, и сейчас самое время научиться пользоваться ею, чтобы сделать свою жизнь проще, а конечному пользователю обеспечить максимально возможный комфорт. Набор инструментальных средств Dojo полностью изменяет процесс веб-разработки и позволяет кардинально ускорить его.

Какой бы веб-проект не появился на вашем горизонте, вы можете быть уверены, что библиотека Dojo поможет вам осуществить его быстро и с минимальными затратами, так чтобы у вас получилась максимально ясная и удобная в сопровождении реализация. Я искренне надеюсь, что эта книга описывает библиотеку Dojo настолько эффективно, что вам придется затрачивать минимальное время в поисках ответов на свои вопросы, и вы сможете полностью сосредоточиться на исследовании сложных (и намного более интересных) проблем, которые вам предстоит решить.

## Почему Dojo?

Несомненно, в настоящее время существует множество библиотек JavaScript, поэтому вы уже наверняка задались вопросом, какие возможности предоставляет библиотека Dojo, которые невозможно получить где-либо еще. Следует понимать, что сама природа комплектов инструментальных средств или библиотек, написанных на полностью интерпретируемом языке программирования, предполагает теоретическую возможность реализовать практически все то же самое в рамках другого комплекта инструментальных средств. Поэтому не так важно, что Dojo может обеспечить что-то, чего не могут другие, – скорее речь следует вести о таких аспектах, как эффективность работы, поддержка сообществом, философия и вопросы лицензирования.

Чтобы лучше понять, о чем идет речь, представьте себе следующее: теоретически вполне возможно построить дом с помощью молотка, лопаты, пилы и кучи гвоздей, но какой ценой? Очевидно, что наличие хоть какой-то механизации труда и помощь еще нескольких плотников придется весьма кстати. Ситуация с библиотекой Dojo очень похожа на описанную. В следующем списке сделана попытка выделить (не в каком-то определенном порядке) некоторые из сторон, где библиотека Dojo наиболее ярко проявляет себя.

*Сообщество*

Открытое сообщество Dojo является одной из главных движущих сил проекта, хотя это скорее нетехническая проблема. Dojo Foundation является некоммерческой организацией, созданной с целью обеспечить независимую защиту интеллектуальной собственности, поддержку библиотеки (и других интересных проектов, таких как Cometd[1], DWR[2] и OpenRecord[3]), и поддерживается такими компаниями, как IBM, AOL, Sun, OpenLaszlo, Nexaweb, SitePen, BEA, Renkoo и множеством самых видных в мире специалистов по DHTML. Не правда ли, впечатляющий список друзей проекта, поддерживающих его?

Этот открытый проект имеет свободную лицензию, а для вступления в него требуется совсем немного, и здесь ваш голос обязательно услышат, если вы захотите быть услышанным. Если вы подключитесь к каналу IRC *#dojo* на *freenode.net*, то у вас появится шанс столкнуться с кем-нибудь из участников проекта. Кроме того, каждую неделю по средам с 15:00 до 18:00 (по тихоокеанскому стандартному времени) на канале IRC *#dojo-meeting* проводятся встречи, куда вы можете заходить без предупреждения и присутствовать или даже участвовать в официальных встречах, где обычно обсуждаются не только тактические, но и стратегические проблемы.

Там вы поймете, насколько умело осуществляются стратегическое управление и тактическая разработка Dojo. По мере того как другие инструментальные средства и библиотеки JavaScript все больше превращаются в товар, сообщество Dojo все больше начинает выделяться на их фоне. Организации и отдельные люди, составляющие костяк проекта (не говоря уже о тысячах разработчиков, создающих действующие веб-сайты и приложения), – все они придают проекту Dojo неповторимый характер и обеспечивают его успех.

*Свободная (и понятная) лицензия*

Библиотека Dojo представляет собой программное обеспечение, распространяемое с открытыми исходными текстами, которое может свободно лицензироваться на условиях лицензии BSD (Berkeley Software Distribution) или AFL (Academic Free License) версии 2.1. Кроме отдельных модулей, условия лицензирования которых описывается в отдельных файлах, распространяемых вместе с модулями, вы можете сами определять, под какой из лицензий вы будете производить свое программное обеспечение. Любые содействия

---

[1] Подробнее о Cometd читайте в Интернете, по адресу: *http://www.cometd.com or http://www.cometdaily.com*.

[2] Подробнее о проекте Direct Web Remoting читайте в Интернете, по адресу: *http://getahead.org/dwr*.

[3] Подробнее об OpenRecord читайте в Интернете, по адресу: *http://www.openrecord.org*.

проекту, поступающие извне, должны предоставляться на условиях, совместимых с условиями лицензий BSD и AFL, а все лица, стремящиеся оказать содействие проекту, должны подписать лицензионное соглашение CLA (Contributor License Agreement). Благодаря этому обеспечивается однозначность прав на все пожертвования в Dojo Foundation и защита всех пользователей набора инструментальных средств от хитросплетений лицензирования интеллектуальной собственности. Преимущества такого ясного лицензирования очевидны по сравнению с другими популярными библиотеками JavaScript (которые мы не будем называть).

*Глубина и широта*

Многие наборы инструментальных средств разработаны под определенные прикладные области; в отличие от них Dojo представляет собой комплексное решение для разработки программного обеспечения, работающего под управлением броузера. Все, начиная от стандартных утилит до готовых к использованию виджетов, инструментов сборки и платформы тестирования, – присутствует здесь, поэтому вам практически не потребуется искать что-либо на стороне. Но не надо думать, что такая широта охвата приведет к раздуванию вашего программного кода, потому что инструменты сборки позволяют создавать собственные версии библиотеки, которые могут быть урезаны под требования вашего приложения.

Нередко широта охвата решаемых проблем препятствует углубленной проработке решений, но в отношении Dojo все совсем наоборот. Даже одна только библиотека Base, маленькое ядро, составляющее основу для всего набора инструментальных средств, предоставляет больше возможностей, чем можно было бы представить, – средства обращения к объектной модели документа (DOM) с использованием селекторов CSS3, утилиты для использования технологии AJAX, реализацию универсальной модели событий, ликвидирующей различия между броузерами, и многое другое. И при этом ядро не затрагивает богатейшей библиотеки приложений, форм и средств размещения виджетов или инструментов сборки.

Несмотря на то, что широта и глубина инструментального набора Dojo порождает немало сложностей, его инфраструктура постоянно и скрупулезно пересматривается одними из самых лучших в мире веб-хакеров с целью обеспечить соответствие программного кода самым высоким стандартам, соблюдение соглашений об именовании, высокую производительность, удобство сопровождения и простоту использования для разработчиков приложений. Вы можете быть уверены, что благодаря применению Dojo вам *удастся* удовлетворить любые прихоти пользователей.

*Переносимость*

Несмотря на то, что язык JavaScript является динамичным, мощным и чрезвычайно выразительным, тем не менее при разработке

возникает большое количество рутинных проблем, решение которых составляет основную задачу разработчика. Очень интересно продираться к решению основной алгоритмической или технической проблемы, но в результате любой написанный вами код вам же придется сопровождать, обновлять, отлаживать и документировать.

Это достаточно веская причина, чтобы использовать стандартную библиотеку JavaScript, особенно если учесть существующее состояние дел с совместимостью броузеров, обладающих различающимися функциональными возможностями. Реализация некоторой функции, одинаково хорошо работающей во всех современных броузерах, может не представлять собой интересную интеллектуальную задачу; это может быть работа, способная своей занудностью деморализовать даже самых закаленных профессионалов.

В конце концов, вряд ли хоть один прикладной программист захочет заработать очки (или получить огромное удовольствие), преодолевая все эти препоны. Он скорее отыщет работающий программный код, написанный *сообществом* других разработчиков, отлаженный и протестированный, а затем подумает о возможности внести свой вклад в это сообщество. Хочется надеяться, что желание «внести свой вклад» возникнет вполне естественным путем, после того как вам удастся сэкономить достаточно времени и денег за счет использования открытого программного обеспечения, поддерживаемого сообществом.

*Философия прагматизма*

Библиотека Dojo использует JavaScript как есть, вместо того чтобы рассматривать его как нечто ненадежное и вследствие этого пытаться создать ломкую, искусственную надстройку над ним, почти полностью переопределяя все и вся. Хотя Dojo обладает огромными функциональными возможностями, защищающими программиста от соприкосновения с голым металлом броузера, и реализует такие особенности, как нормализованная модель событий, так что вам даже не придется об этом вспоминать, – тем не менее библиотека не пытается переосмыслить JavaScript. Например, в Dojo отсутствуют специализированные функции для удаления узлов DOM или для обхода дерева DOM, потому что такие операции, как `childNodes`, `firstChild`, `lastChild` и `removeChild`, прекрасно работают во всех броузерах. Однако, в тех областях, где между броузерами наблюдаются отличия, библиотека Dojo вступает в игру, чтобы предоставить в распоряжение программиста инструменты, которые помогут создавать переносимый программный код.

Кроме того, библиотека Dojo не пытается воспрепятствовать или ограничить возможность применения других библиотек JavaScript – это вполне обычное дело, когда Dojo используется совместно с другими технологиями, такими как DWR или YUI!. И, конечно же, вы совершенно свободно можете использовать любые серверные техно-

логии, так как Dojo, будучи исключительно клиентской технологией, совершенно не зависит от серверных технологий.

Детальный обзор всех популярных инструментальных средств JavaScript показал бы, что у всех у них есть много общего, чем, в первую очередь, и обусловлена их популярность. Поэтому, когда приходит время принимать решение о выборе одного или нескольких комплектов инструментальных средств, определенно стоит подумать о важности таких аспектов, как наличие сообщества, прозрачность, условия лицензирования и философия управления развитием технологии, в которую вы собираетесь вкладывать свое время и, возможно, деньги. Вам определенно захочется иметь поддержку (сообщество и документацию) в случае необходимости, вам едва ли захочется вкладываться в проект, слишком сложный в сопровождении или вообще проигрышный, и вы точно захотите минимизировать время, необходимое на то, чтобы штопать дыры, которые уже заштопаны какими-либо инструментальными средствами.

## Структура книги

Первая часть этой книги во многом представляет собой справочник по стандартной библиотеке, в ней будут рассматриваться все углы и закоулки библиотек Base и Core – та часть набора инструментальных средств, которая включает в себя стандартную библиотеку JavaScript. Размер библиотеки Base при передаче по кабелю[1] составляет менее 30 Кбайт. Библиотека тщательно оптимизирована по скорости, размеру и степени полезности. Она включает в себя очень богатые и разнообразные функциональные возможности, среди которых средства для работы с технологией AJAX, выполнение запросов к DOM с использованием синтаксиса селекторов CSS, стандартизованная модель распространения событий и утилиты функционального программирования, такие как `map` и `filter`. Вы очень скоро будете себя спрашивать, как же вы обходились раньше без всего этого. Библиотека Core включает значительный объем дополнительных возможностей, таких как анимация и поддержка технологии «перетащил и бросил» (drag-and-drop), которые, несмотря на предоставляемые удобства, используются не так часто, как механизмы из библиотеки Base.

Одно замечание к первой части: развернутое обсуждение синтаксического анализатора, или парсера, отложено до главы 11, где будет представлена библиотека Dijit, потому что в большинстве случаев парсер используется для анализа виджетов. Впрочем, упоминание о парсере имеется во врезке, в главе 7, потому что

[1] При более подробном обсуждении в последующих главах вы узнаете, что при указании размера при передаче «по кабелю» подразумевается размер содержимого после сжатия, так как это самый обычный способ передачи веб-страниц от веб-сервера клиентам.

его удобно использовать при встраивании функции «перетащил и бросил» (drag-and-drop).

Первая часть книги включает в себя следующие главы:

Глава 1, *Обзор комплекта инструментальных средств*

Представляет собой краткое введение в набор инструментальных средств, включая следующие темы: архитектура Dojo, как получить и установить Dojo, как встраивать Dojo в веб-страницы; кроме того, в некоторых разделах будут приводиться примеры, чтобы вы могли увидеть Dojo в действии.

Глава 2, *Утилиты броузера*

Представляет собой обширный обзор часто применяемых вспомогательных функций, которые обычно используются в любых веб-приложениях. Большая часть этих функций призвана сгладить различия между броузерами, обеспечить удобства там, где их не хватает в реализациях JavaScript или DOM, а в других случаях – уменьшить объем шаблонного программного кода, который вам приходится писать для выполнения некоторых действий.

Глава 3, *Обработчики событий и организация взаимодействий по подписке*

В этой главе будут представлены конструкции управления взаимодействиями в пределах страницы. Здесь будут обсуждаться две основные парадигмы, первая из которых связана с непосредственной обработкой определенного события, возникающего где-либо в дереве DOM, в объекте `Object` или в отдельной функции, а вторая связана с организацией взаимодействий по подписке на события, когда произвольному подписчику разрешается принимать события и отвечать на них по мере необходимости.

Глава 4, *Технология AJAX и взаимодействие с сервером*

Представляет собой краткий обзор технологии AJAX и механизмов в наборе инструментальных средств, обеспечивающих взаимодействие с сервером с использованием объекта *XMLHttpRequest*. Здесь также обсуждаются механизмы отложенного выполнения, которые предоставляют единообразный способ асинхронной обработки событий. У вас может сложиться впечатление, что эти механизмы создают иллюзию многопоточных приложений, хотя JavaScript не поддерживает потоки выполнения. Здесь также обсуждаются такие особенности, как применение формата JSON для организации получения данных из другого домена, удаленный вызов процедур (Remote Procedure Call, RPC) и передача данных посредством плавающих фреймов (IFRAME).

Глава 5, *Манипулирование узлами*

Представляет универсальный механизм, позволяющий выполнять запросы к дереву DOM используя селекторы CSS, обрабатывать спи-

ски узлов, возвращаемых удобными встроенными функциями, что предоставляет возможность строить произвольные цепочки событий и допускает использовать идиому отделения поведения узлов DOM от конкретных действий, определяемых в разметке HTML.

Глава 6, *Интернационализация (i18n)*

Содержит краткий обзор и примеры интернационализации веб-приложений с использованием утилит из комплекта инструментальных средств. Также включает обзор различных конструкций, используемых для управления такими элементами интернационализации, как представление дат, времени, денежных сумм и форматирования чисел.

Глава 7, *Перетаскивание элементов*

Представляет собой самостоятельный учебник о том, как с помощью Dojo можно быстро добавить в приложение возможность перетаскивания элементов мышью.

Глава 8, *Анимация и специальные эффекты*

Представляет собой самостоятельный учебник о встроенных в Dojo механизмах анимации произвольных свойств CSS, что позволяет создавать различные визуальные эффекты, такие как исчезновение, скольжение и плавные изменения. Здесь также будут описаны утилиты, позволяющие изменять и смешивать цвета.

Глава 9, *Абстракция данных*

Обсуждает инфраструктуру абстракции данных в Dojo, которая представляет собой промежуточный уровень между логикой приложения и конкретными используемыми форматами данных, будь то открытые стандарты или закрытые форматы, защищенные патентами.

Глава 10, *Имитация классов и наследование*

Представляет собой переход ко второй части, где рассказывается о библиотеке Dijit. В этой главе описываются механизмы Dojo, позволяющие использовать приемы объектно-ориентированного программирования, которые очень широко используются в Dijit.

Вторая часть книги систематически исследует остальную часть набора инструментальных средств, включая полный охват Dijit, богатейшей библиотеки настраиваемых элементов управления HTML, которые столько раз приходилось писать (и переписывать). Библиотека Dijit спроектирована так, что может применяться непосредственно из разметки HTML с использованием незначительного объема программного кода или вообще без него. Благодаря ей вы узнаете, что можно создавать веб-страницы с фантастическим внешним видом, приложив совсем незначительные усилия, потому что элементы управления, предлагаемые библиотекой, уже выглядят и ведут себя почти как элементы пользовательского интерфейса обычных настольных приложений.

Вторая часть заканчивается обсуждением системы сборки и платформы модульного тестирования, представленной библиотекой Unit. Система сборки включает в себя инструмент ShinkSafe, использующий механизм сжатия программного кода (нередко до трех раз и более) из Rhino JavaScript. Аббревиатура DOH происходит от Dojo Objective Harness (а также совпадает с известным восклицанием героя мультипликационного сериала Гомера Симпсона «D'oh!») и является названием самостоятельной системы модульного тестирования программного кода на языке JavaScript.

Вторая часть книги включает в себя следующие главы:

Глава 11. *Обзор Dijit*

Представляет библиотеку Dijit, рассматривает различные темы, такие как философия дизайна, доступность, парсер (технически является частью библиотеки Core, но наиболее часто используется для парсинга страниц, содержащих элементы управления из библиотеки Dijit) и примеры использования элементов управления. Завершается глава кратким обзором каждого из подпроектов Dijit.

Глава 12. *Анатомия Dijit и жизненный цикл*

Подробно рассматривает, как располагаются на диске файлы, из которых состоят «диджиты»[1] – элементы графического интерфейса, а также, как выглядит жизненный цикл экземпляра диджита после его размещения в памяти. Содержит множество коротких примеров, каждый из которых подчеркивает определенные моменты жизненного цикла. Понимание того, как протекает жизненный цикл диджита, совершенно необходимо для следующей главы.

Глава 13. *Виджеты форм*

Содержит краткий обзор обычных форм HTML и затем переходит к полному обзору виджетов (графических элементов) форм, которые в большинстве своем тесно связаны с имеющейся коллекцией. Виджеты форм могут использоваться в качестве замены всех типичных элементов форм, фактически использующихся в веб-дизайне. Среди рассматриваемых элементов вы найдете разнообразные кнопки, специализированные текстовые поля и движки. Кроме того, в составе коллекции присутствуют такие дополнительные производные элементы, как раскрывающиеся списки, которые так часто приходилось создавать и пересоздавать.

Глава 14. *Виджеты компоновки*

В этой главе будут представлены виджеты компоновки – коллекция виджетов, посредством которых создается скелет сложных схем компоновки, для чего часто привлекаются непростые и утоми-

[1] По аналогии с «виджетами» (widgets) и «гаджетами» (gadgets). – *Прим. перев.*

тельные операции с CSS, обеспечивающих отображение или сокрытие элементов форм в зависимости от состояния приложения, позволяющих реализовать табличную верстку и многое другое.

Глава 15. *Виджеты приложения*

Описывает остальные виджеты в инструментальном наборе, соответствующие типичным элементам управления приложений, таким как всплывающие подсказки, модальные диалоги, меню, деревья и многофункциональные текстовые редакторы.

Глава 16. *Инструменты сборки, тестирования и вопросы подготовки к выпуску в эксплуатацию*

Завершает книгу обсуждением некоторых обычно опускаемых, но очень важных тем, касающихся развертывания приложений. Содержит обширное описание инструментов сборки, которые упрощают операции по сжатию, уменьшению и объединению программного кода на JavaScript, с целью минимизации размера файла и задержки, вызванной передачей данных по протоколу HTTP. Рассказывает о платформе модульного тестирования и рассматривает разные аспекты подготовки приложения к развертыванию, что должно помочь навести последний лоск на ваше приложение.

В книге имеются два дополнительных приложения: краткий обзор DojoX – набор специализированных и экспериментальных расширений – и учебник по работе с отладчиком Firebug. Несмотря на то, что библиотека DojoX является настоящей сокровищницей виджетов и модулей для любых применений, от рисования диаграмм до криптографии и широко любимого, очень гибкого виджета табличного представления данных, тем не менее нет никакой гарантии стабильности или непротиворечивости прикладных интерфейсов в проектах, включенных в состав DojoX, по сравнению с Base, Core и Dijit. Кроме того, полное описание DojoX легко могло бы занять несколько отдельных книг.

В другом приложении представлено простое учебное руководство по использованию отладчика Firebug. С его помощью вы быстро познакомитесь со всеми основными особенностями отладчика, который (посредством своего интерфейса командной строки) позволит вам сэкономить массу времени, когда возникнет потребность отладить или быстро исследовать новые идеи. Для тех, кому ранее не приходилось слышать о Firebug, – это фантастическое дополнение к броузеру Firefox, позволяющее исследовать буквально все аспекты веб-страницы – от исследования и манипулирования стилями узлов DOM до мониторинга сетевой активности и выполнения фрагментов программного кода на JavaScript с помощью интерфейса командной строки.

# Чего нет в этой книге

Несмотря на то, что эта книга пытается обеспечить такую же широту и полноту охвата тем, как и сама библиотека Dojo, тем не менее некоторые темы настолько обширны, что их рассмотрение невозможно было поместить в это издание:

*Веб-разработка 101*

Эта книга представляет собой глубокий обзор библиотеки Dojo, но она не является законченным учебником по веб-разработке, в котором вводились бы такие элементарные понятия, как HTML, JavaScript и CSS.

*Обширное описание прикладного интерфейса*

Подавляющее большинство[1] прикладных интерфейсов библиотеки Dojo в этой книге подробно описано и сведено в таблицы для справки. Библиотека Dojo обладает чрезвычайно широкими возможностями, поэтому мне казалось важным представить вашему вниманию как можно большую часть этих возможностей, чтобы к моменту, когда они потребуются, вы уже знали, что вам доступно. Однако библиотека Dojo продолжает развиваться, поэтому вам постоянно будет необходимо сверяться с наиболее авторитетным источником информации в Интернете, по адресу *http://api.dojotoolkit.org*. В отличие от языков программирования и менее гибких платформ разработки приложений, Dojo является развивающимся проектом, поддерживаемым растущим сообществом, поэтому весьма вероятно, что с выходом новых версий в нужном вам направлении будет расширяться и прикладной интерфейс. Но вы должны знать, что целостность прикладного интерфейса в версии 1.x не будет нарушена, по крайней мере, до выхода версии 2.0, поэтому тот прикладной интерфейс, что рассматривается в этой книге, останется неизменным в течение некоторого времени. Но даже тогда, когда некоторые части прикладного интерфейса претерпят существенные изменения, они предварительно будут подробно документированы.

*Среды выполнения вне броузеров*

В этой книге даже не предпринимаются попытки исследовать или представить примеры использования Dojo в средах выполнения, не являющихся броузерами (таких как Rhino или Adobe AIR), или показать, как можно использовать библиотеку Dojo в комбинации с другими клиентскими платформами, такими как DWR, YUI! или Domino.

---

[1] По скромным оценкам эта книга охватывает примерно 95% API библиотек Base, Core, Dijit и Util.

# Открытое программное обеспечение не прекращает развиваться

Библиотека Dojo относится к открытому программному обеспечению с растущим сообществом, вследствие чего в нее в любой момент могут быть добавлены новые особенности. Эта книга охватывает версию Dojo 1.1, но совершенно очевидно, что в последующих версиях в библиотеку могут быть добавлены новые функциональные возможности. Чтобы гарантировать, что вы обладаете самыми последними сведениями, вам необходимо ознакомиться с примечаниями ко всем выпускам, вышедшим после версии 1.1.

Кроме того, прикладной интерфейс библиотеки Dojo в настоящее время заморожен до выхода версии 2.0, поэтому все примеры и сведения, что приводятся в этой книге, останутся верными и для всех последующих версий, меньших 2.0. Даже если вы читаете эту книгу, когда уже была выпущена версия 2.0, примеры программного кода по-прежнему должны работать, так как неофициальная политика сохранения обратной совместимости состоит в том, что все, что не приветствуется использовать в новой главной версии, скорее всего будет оставлено до выхода очередной новой главной версии. Другими словами, все, что не приветствуется использовать в версии 1.x, будет оставлено в неприкосновенности до выхода версии 2.0, а возможно, просуществует и дольше.

# О вас

Эта книга предполагает, что у вас имеется некоторый опыт веб-разработки с применением клиентских технологий, таких как HTML, JavaScript и CSS. Однако от вас не требуется быть экспертом ни в одной из этих областей, и вам на самом деле вообще не требуется знать, что происходит на стороне веб-сервера, потому что Dojo – это клиентская технология. Простого знания о существовании этих технологий и представления о том, как они используются, будет вполне достаточно.

Если вы действующий веб-разработчик или просто человек, увлеченный веб-разработкой, способный создать простую веб-страницу и применить JavaScript и CSS, чтобы немного оживить ее, тогда вам определенно стоит продолжить чтение. Если вы даже не слышали об HTML, JavaScript или CSS и до сих пор не написали ни одной строчки программного кода, тогда, возможно, вам лучше будет подыскать в дополнение к этой книге хорошее введение в веб-разработку.

## Инструменты разработки

В качестве инструментов разработки можно использовать привычный вам текстовый редактор и любой веб-броузер – для эффективной разработки с использованием библиотеки Dojo этого будет вполне доста-

точно. В этой книге часто упоминается веб-броузер Firefox и замечательное дополнение к нему Firebug, который вы можете использовать для отладки и исследования веб-страниц, а кроме того, вы можете возиться с JavaScript в его консоли. Вы можете использовать версию Firebug Lite в паре с другим броузером, таким как Internet Explorer, однако полная версия Firebug обладает гораздо более широкими возможностями и не разочарует вас. (На практике обычно разработка ведется с использованием Firefox и Firebug, но на этапе тестирования часто привлекается броузер Internet Explorer.) Получить Firefox и Firebug можно по адресам: *http://getfirefox.com* и *http://getfirebug.com*, соответственно.

Совместно с броузером Firefox было бы желательно использовать еще два инструмента. Первый – это Web Developer Toolbar (панель инструментов веб-разработчика), разработанная Крисом Педериком (Chris Pederick) и доступная по адресу *http://chrispederick.com/work/web-developer/*, содержащая ряд дополнительных инструментов, которые могут пригодиться в процессе разработки. И второй – дополнение к Firefox под названием Clear Cache Button (кнопка очистки кэша), доступное по адресу *https://addons.mozilla.org/en-US/firefox/addon/1801*, и представляющее собой кнопку для панели инструментов, с помощью которой можно быстро очистить кэш броузера. Иногда могут возникать ситуации, когда броузер «капризничает» и показывает устаревшие версии страниц, в таких ситуациях очистка кэша может помочь решить эту проблему.

## Основные практические знания

Замыкания, контекст и анонимные функции – это одни из самых важных и фундаментальных концепций в JavaScript, а так как для овладения инструментарием требуется больше, чем поверхностное знание этих тем, прочитайте внимательно этот раздел. Хотя эти понятия на первый взгляд могут показаться слишком сложными, тем не менее они составляют основу для овладения языком JavaScript и действительно необходимы для понимания внутренних механизмов инструментария. Вы можете попробовать приобрести эти знания в ходе чтения книги, но если вы потратите некоторое время сейчас, то позднее вы обнаружите, что многие объяснения в последующих главах будут выглядеть гораздо понятнее.

### Замыкания

Замыкание – это по сути связывание элементов данных с областью видимости, которая содержит (или заключает) эти данные. В обычной ситуации, когда имеется единственная глобальная область видимости, содержащая некоторые функции, нет ничего сложного, но когда появляются вложенные функции, ситуация кардинально меняется. В качестве иллюстрации рассмотрим пример 1.

*Пример 1. Простейшая иллюстрация замыкания*

```
function foo() {
  var x = 10;
  return function bar() {
    console.log(x);
  }
}

var x = 5;
var barReference = foo();
barReference(); // Что будет напечатано? 10 или 5
```

В зависимости от вашего уровня подготовки как программиста, предыдущий фрагмент программного кода может удивить вас. Как оказывается, он выведет число 10, потому что в JavaScript при вычислении значения функции в учет принимается вся цепочка областей видимости. В данном случае цепочка областей видимости связана с функцией bar, значение которой возвращается функцией foo. То есть, когда производится попытка получить значение barReference, поиск значения переменной x будет произведен в процессе выполнения и значение будет найдено в теле функции foo. Такое поведение прямо противоположно поведению, принятому во многих других языках программирования, где искомый контекст находится в самой ближней области видимости.

В языке JavaScript функции являются самыми обычными объектами, которые можно передавать между различными участками программы, присваивать переменным и т. д. Некоторые шаблоны проектирования в Dojo очень широко используют эту особенность.

Замыкания в языке JavaScript обычно считают сложной темой, тем не менее чем раньше вы получите полное понимание, как работают замыкания, тем быстрее вы будете двигаться по пути освоения языка и тем лучше будете понимать многие аспекты философии дизайна библиотеки Dojo.

В практическом смысле это означает, что ваша производительность вырастет, вы будете в состоянии намного быстрее отыскивать сложные ошибки и, возможно, даже стать более интересным человеком. (Даже первые два результата из трех, это уже неплохо.) Прекрасный анализ замыканий вы найдете в легендарной книге Дэвида Флэнагана (David Flanagan) «JavaScript: The Definitive Guide» (O'Reilly).[1]

[1] Дэвид Флэнаган «JavaScript. Полное руководство», 5-е издание. – Пер. с англ. – СПб.: Символ-Плюс, 2008. – *Прим. перев.*

## Контекст

Чрезвычайный динамизм языка JavaScript обеспечивает ему огромную гибкость, и одним из самых интересных и наименее понятных аспектов, связанных с динамизмом, является *контекст*. Возможно, вы уже знаете, что по умолчанию в реализации JavaScript для броузеров контекстная ссылка `this` указывает на глобальный объект `window`. Например, следующая последовательность инструкций должна возвращать значение `true` практически в любом броузере:

```
// контекстом документа по умолчанию является объект window
console.log(window == this); //true
// document - это сокращение для window.document
console.log(document == window.document); //true
```

Внутри объектов-функций ключевое слово `this` используется как ссылка на контекст этого объекта. Например, возможно, вам встречалось такое использование ссылки `this` внутри функций JavaScript, как показано ниже:

```
function Dog(sound) {
    this.sound = sound;
}

Dog.prototype.talk = function(name) {
    console.log(this.sound + "," + this.sound + ". my name is", name);
}

dog = new Dog("woof");
dog.talk("fido"); //woof, woof. my name is fido
```

Если у вас имеется опыт работы с языком Java или другим объектно-ориентированным языком программирования, тогда порядок определения значения `sound` относительно текущего объекта будет выглядеть для вас достаточно знакомым. В этом фрагменте не происходит ничего интересного. Однако, как только начинает использоваться встроенная функция `call`, все происходящее может существенно осложниться. Рассмотрим следующий искусственный пример, где в дело вступает функция `call`:

```
function Dog(sound) {
    this.sound = sound;
}

Dog.prototype.talk = function(name) {
    console.log(this.sound + "," + this.sound + ". my name is", name);
}

dog = new Dog("woof");
dog.talk("fido"); //woof, woof. my name is fido

function Cat(sound) {
    this.sound = sound;
}
```

```
Cat.prototype.talk = function(name) {
    console.log(this.sound + "," + this.sound + ". my name is", name);
}

cat = new Cat("meow");
cat.talk("felix"); //meow, meow. my name is felix

cat.talk.call(dog, "felix") //woof, woof. my name is felix
```

Стоп! Последняя строка выполнила что-то невероятное. Через экземпляр объекта `cat` был вызван метод `talk` прототипа объекта `cat`, которому был передан параметр `name`, используемый как обычно. Однако, вместо значения `sound`, присвоенного объекту `cat`, было использовано значение, связанное с объектом `dog`. Произошло это потому, что в качестве контекста был использован объект `dog`.

Функция `call` стоит того, чтобы потратить несколько минут на знакомство с ней, если вы с ней еще не знакомы. Во многих менее динамичных языках программирования возможность переопределять контекстную ссылку `this` выглядит почти абсурдно. Однако в языке JavaScript эта возможность является мощной особенностью, и инструментальные средства, такие как Dojo, используют такого рода возможности для реализации поразительных действий. Некоторые из этих действий будут представлены уже в следующем разделе.

Цель этой книги состоит вовсе не в том, чтобы обеспечить всесторонний охват языка JavaScript, как это сделано в книге «JavaScript: The Definitive Guide», но для вас будет интересно узнать, что функция `apply` работает точно так же, как и функция `call`, за исключением того, что вместо неопределенного числа параметров, передаваемых целевой функции, она принимает только два параметра, последний из которых является массивом, содержащим неограниченное число значений. Этот массив будет преобразован в свойство `arguments` целевой функции. По сути, вам предоставляется возможность выбора наиболее удобного для вас варианта.

## Анонимные функции

В языке JavaScript объекты типа Function могут передаваться между различными участками программы точно так же, как и объекты любых других типов. Анонимные функции на практике могут служить этаким синтаксическим подсластителем, уменьшающим загромождение программного кода и упрощающим его сопровождение, но, пожалуй, более важная особенность анонимных функций состоит в том, что они обеспечивают возможность создания замыканий, защищающих контекст функций.

Например, как вы думаете, что получится в результате выполнения следующего фрагмента программного кода?

```
// Переменная i не определена.
for (var i=0; i < 10; i++) {
    // выполняются некоторые действия с участием i
}
console.log(i); // ???
```

Если вы думаете, что инструкция `console.log(i)` выведет слово `undefined`, то вы глубоко ошибаетесь. Малозаметная, но очень важная особенность языка JavaScript заключается в том, что он не поддерживает концепцию областей видимости для блоков, меньших чем функции, и по этой причине значение переменной `i`, как и любой другой «временной» переменной, определяемой в операторах циклов, в условных инструкциях и в других подобных ситуациях, продолжает существовать и после того, как выполнение блочной инструкции будет завершено.

При необходимости явно ограничить область видимости одним блоком *можно было бы* обернуть блок встроенным объектом Function. Взгляните на следующую версию этого же фрагмента:

```
(function() {
    for (var i=0; i < 10; i++) {
        // выполняются некоторые действия с участием i
    }
})()
console.log(i); // undefined
```

Несмотря на неуклюжесть синтаксиса, такого рода предосторожности иногда могут уберечь вас от появления неприятных ошибок. Многие функции в библиотеке Base, которые будут представлены в последующих главах, обеспечивают создание замыканий (в дополнение к синтаксическому подсластителю и удобству) в выполняемых блоках программного кода.

## Типографские соглашения, используемые в книге

В этой книге приняты следующие соглашения:

Рубленый шрифт

Применяется для обозначения заголовков и пунктов меню, кнопок и комбинаций клавиш (например, Alt и Ctrl).

*Курсив*

Используется для выделения новых терминов, адресов электронной почты, веб-сайтов, имен файлов и каталогов, утилит операционной системы UNIX.

Моноширинный шрифт

Применяется для выделения команд, параметров, вариантов выбора, переменных, атрибутов, ключей, функций, типов, классов, про-

странств имен, методов, модулей, свойств, аргументов, значений, объектов, событий, обработчиков событий, тегов XML, тегов HTML, макроопределений, содержимого файлов или результатов работы команд.

**Моноширинный жирный**

Используется для выделения команд и другого текста, который должен быть введен пользователем.

*Моноширинный курсив*

Обозначает аргументы функций и элементы, которые в программе необходимо заменить на реальные значения.

Таким способом выделяются советы, предложения и примечания общего характера.

Таким способом выделяются предупреждения и предостережения.

## Соглашения по оформлению

Следует также отметить два дополнительных соглашения по оформлению, так как они способствуют более эффективному взаимодействию с содержимым книги.

*Квалифицированные ссылки*

Полная квалификация пространств имен обычно не используется, когда контекст операции очевиден. Например, если в листинге появляется виджет `dijit.form.Button`, то в дальнейшем обсуждении он может именоваться просто как `Button`.

Некоторые термины, как например, *constructor*, могут использоваться в нескольких смыслах в пределах одного и того же параграфа. Всякий раз, когда это происходит, для подчеркивания различий используется моноширинный шрифт. Например, в предложении: «Виджет создается вызовом обычной функции-конструктора JavaScript, но сам виджет также имеет специальный метод `constructor`, который может использоваться для выполнения действий по инициализации», – предпринята попытка устранить неоднозначность в понимании термина «constructor» за счет применения моноширинного шрифта.

*Представление функций API*

Вообще эта книга стремится предоставить полные сигнатуры функций API, используя соглашение, согласно которому наряду со стандартизованными именами параметров указываются их типы. Например, рассмотрим функцию, которая вызывается как `loadUpAr-`

`ray("foo", 4)` и возвращает `["foo", "foo", "foo", "foo"]`. Сигнатура такой функции API могла бы быть представлена, как показано ниже:

```
loadUpArray(/*String*/ value, /*Integer*/ length) //возвращает Array
```

Язык JavaScript является весьма динамичным языком программирования со слабой типизацией, но, тем не менее ситуации, когда функция может принимать или возвращать значения произвольных типов, встречаются достаточно редко. В таких случаях для обозначения типа будет использоваться слово `Any` (любой). Если параметр является необязательным, вслед за обозначением его типа будет следовать вопросительный знак, например: `/*Integer?*/`.

Начав исследовать программный код библиотеки Dojo, вы можете обратить внимание, что имена некоторых параметров в исходном программном коде библиотеки отличаются от тех, что указаны в сигнатурах функций API в этой книге. Параметры функций в языке JavaScript являются неименованными и позиционными, поэтому такие различия в именах параметров не имеют существенного значения – эта особенность языка была использована, чтобы улучшить понимание описаний функций API. Были приложены все усилия, чтобы обеспечить максимально единообразное представление сигнатур функций API, но иногда могут наблюдаться незначительные отклонения.

## Использование программного кода примеров

Данная книга призвана оказать вам помощь в решении ваших задач. Вообще вы можете свободно использовать примеры программного кода из этой книги в своих приложениях и в документации. Вам не нужно обращаться в издательство за разрешением, если вы не собираетесь воспроизводить существенные части программного кода. Например, если вы разрабатываете программу и используете в ней несколько отрывков программного кода из книги, вам не нужно обращаться за разрешением. Однако в случае продажи или распространения компакт-дисков с примерами из этой книги вам необходимо получить разрешение от издательства O'Reilly. Для цитирования данной книги или примеров из нее при ответе на вопросы не требуется получение разрешения. При включении существенных объемов программного кода примеров из этой книги в вашу документацию, вам *необходимо* будет получить разрешение издательства.

Мы приветствуем, но не требуем добавлять ссылку на первоисточник при цитировании. Под ссылкой на первоисточник мы подразумеваем указание авторов, издательства и ISBN. Например: «Dojo: The Definitive Guide, by Matthew A. Russell. Copyright 2008 Matthew A. Russell, 978-0-596-51648-2».

За получением разрешения на использование значительных объемов программного кода примеров из этой книги обращайтесь по адресу *permissions@oreilly.com*.

## Safari® Books Online

Если на обложке технической книги есть пиктограмма «Safari® Books Online», это означает, что книга доступна в Сети через O'Reilly Network Safari Bookshelf.

Safari предлагает намного лучшее решение, чем электронные книги. Это виртуальная библиотека, позволяющая без труда находить тысячи лучших технических книг, вырезать и вставлять примеры кода, загружать главы и находить быстрые ответы, когда требуется наиболее верная и свежая информация. Она свободно доступна по адресу *http://safari.oreilly.com*.

## Отзывы и предложения

С вопросами и предложениями, касающимися этой книги, обращайтесь в издательство:

O'Reilly Media
1005 Gravenstein Highway North
Sebastopol, CA 95472
(800) 998-9938 (в Соединенных Штатах Америки или в Канаде)
(707) 829-0515 (международный)
(707) 829-0104 (факс)

Список опечаток, файлы с примерами и другую дополнительную информацию вы найдете на сайте книги:

*http://www.oreilly.com/catalog/9780596516482*

Свои пожелания и вопросы технического характера отправляйте по адресу:

*bookquestions@oreilly.com*

Дополнительную информацию о книгах, обсуждения, Центр ресурсов издательства O'Reilly вы найдете на сайте:

*http://www.oreilly.com*

## Благодарности

Работа над этой книгой была больше чем просто работа по совместительству или крещение огнем. В действительности это было целое путешествие, начавшееся значительно раньше, чем я услышал о Dojo, JavaScript или компьютерах. Эта книга явилась логическим продолжением жизненно важных событий, в которые оказались вовлечены

потрясающие личности. Все, что следует дальше, – это сильно сжатая, полухронологическая последовательность отдельных кадров ключевых событий, сыгравших свою роль в страницах, которые вы собираетесь читать.

Эта книга явилась кульминацией самых разнообразных событий в моей жизни, но общим для всех этих событий являются удивительные люди, встречавшиеся мне на каждом этапе моего пути. И с большим восхищением я приношу свои многочисленные благодарности:

- Господу нашему, любовь которого пребывает с нами вечно!
- Люсиль Табор (Lucille Tabor), ставшую малышу матерью и создавшую ему дом.
- Джерри Расселлу (Jerry Russell), купившему сорванцу его первый компьютер.
- Дэвиду Кейду (David Kade), научившему подающего надежды студента думать на другом языке.
- Деборе Пеннингтон (Deborah Pennington), которая практически в одиночку перевернула жизнь молодого панка.
- Келлану Сарлесу (Kellan Sarles), научившему начинающего писателя излагать свои мысли на бумаге.
- Гэри Ламонту (Gary Lamont), заложившему в юный ум способность думать как программист.
- Деррику Стори (Derrick Story), давшему мне шанс стать писателем.
- Эби Мьюзику (Abe Music), первому, кто поведал мне о Dojo.
- Симону Сент-Лорену (Simon St.Laurent), Татьяне Апанди (Tatiana Apandi), Колину Горману (Colleen Gorman), Рэйчел Монаган (Rachel Monaghan) и Самите Макержи (Sumita Mukherji), дававшим мне редакторские указания и рекомендации, которые позволили сгладить множество острых углов в этой книге.
- Великому множеству моих новых друзей в `#dojo`, так великодушно согласившихся ознакомиться с рукописью этой книги и представить свои отзывы, которые оказали существенное влияние на качество книги. Перечислю их без иерархии: Адам Пеллер (Adam Peller), Сэм Фостер (Sam Foster), Карл Тайдт (Karl Tiedt), Билл Кис (Bill Keese), Дастин Маши (Dustin Machi), Пит Хиггинс (Pete Higgins), Джеймс Барк (James Burke), Питер Кристофферссон (Peter Kristoffersson), Алекс Расселл (Alex Russell) и Дилан Шиманн (Dylan Schiemann).
- Басирет – любови всей моей жизни.
- Господу нашему, любовь которого пребывает с нами вечно!

– Мэтью А. Расселл (Matthew A. Russell)

*Июнь 2008*

# I

# Base и Core

Эта часть книги представляет собой экскурс по библиотекам Base и Core – элементам инструментария, содержащим мощную стандартную библиотеку JavaScript. Библиотека *Base* – это ядро набора инструментальных средств. Эта библиотека оптимизирована до такой степени, что содержит впечатляющий объем функциональных возможностей всего в 30 Кбайтах при передаче по кабелю. Каждая особенность, входящая в состав Base, прошла тщательную проверку на полезность и была оптимизирована на скорость выполнения и размер. Начав использовать библиотеку Base, вы быстро обнаружите, что уже не желаете обходиться без нее. Чтобы встроить Base в свою страницу, достаточно добавить единственный тег `SCRIPT`, причем можно даже указать адрес в другом домене, например адрес одного из кэширующих серверов службы AOL (America Online). Кроме того, что библиотека составляет логическую основу всего набора инструментальных средств, все, что в ней содержится, включено в пространство имен базового уровня `dojo`, поэтому, чтобы применить наиболее часто используемые функции и элементы данных, достаточно всего нескольких нажатий на клавиши.

Библиотека *Core* расширяет Base дополнительными функциональными возможностями, которые, вы рано или поздно начнете использовать, но, чтобы сохранить размер библиотеки Base как можно меньше при максимальных возможностях, библиотека Core была оформлена в виде отдельной библиотеки, так как входящие в нее особенности недостаточно универсальны для всех случаев использования. Чтобы подключить ресурсы из библиотеки Core, достаточно просто воспользоваться инструкцией `dojo.require`, которая по своему поведению напоминает директиву `#include` из языка программирования C или инструкцию `import` из языка Java – и с этого момента в вашем распоряжении будут обе библиотеки. Как будет показано в главе 16, где описывается

библиотека Util, с помощью системы сборки Dojo вы можете объединить необходимые дополнительные ресурсы, не входящие в библиотеку Base, в единый сценарий; таким образом, функциональность библиотеки Core будет так же доступна для нормальной работы, как и Base. В составе библиотеки Core вы найдете такие функциональные возможности, как механизм анимации (`dojo.fx`), механизм «перетащил и бросил» (`dojo.dnd`), уровень управления данными (`dojo.data`), функции для работы с cookie (`dojo.cookie`) и многое другое.

Знакомство с арсеналом инструментов в библиотеках Base и Core совершенно необходимо разработчику, чтобы применять инструментальный набор Dojo, и велика вероятность, что эти механизмы пополнят ваш набор инструментов и приемов независимо от того, как вы к ним пришли, или от того, как долго вы их использовали. После овладения Base и Core вам потребуется затрачивать меньше усилий на рутинные задачи, над решением которых многие разработчики проводят больше времени, чем над более интересными аспектами проекта, требующими творческого подхода и нестандартного мышления.

# 1

# Обзор комплекта инструментальных средств

Эта глава представляет краткий обзор архитектуры инструментального набора Dojo, описывает процедуру установки Dojo, знакомит со специализированной терминологией, проводит через процесс настройки и затем демонстрирует некоторые примеры программного кода, чтобы разжечь у вас желание перейти к следующим главам. Как и любое другое введение, эта глава рисует общую картину крупными мазками и задает тон для остальной части книги. Хотелось бы надеяться, что эта глава будет вам полезна, так как вы начинаете свое знакомство с инструментарием.

## Обзор архитектуры Dojo

Как вы увидите дальше, использование термина *инструментальный набор* применительно к Dojo далеко не случайно. В дополнение к тому, что он содержит своего рода стандартную библиотеку языка JavaScript, в состав Dojo также входят: коллекция готовых к употреблению графических компонентов (виджетов), для использования которых если и требуется писать программный код, то только минимального объема; инструменты сборки; платформа тестирования и многое другое. В этом разделе вашему вниманию предлагается обзор архитектуры Dojo с общих позиций, как показано на рис. 1.1. Как вы убедитесь в этом позднее, организация остальной части книги в значительной степени определяется архитектурой инструментария. Хотя библиотека DojoX представлена отдельно от Dijit, тем не менее ресурсы DojoX можно построить на основе ресурсов Dijit, так же как вы можете создавать свои собственные виджеты на любых комбинациях ресурсов из Dijit и DojoX.

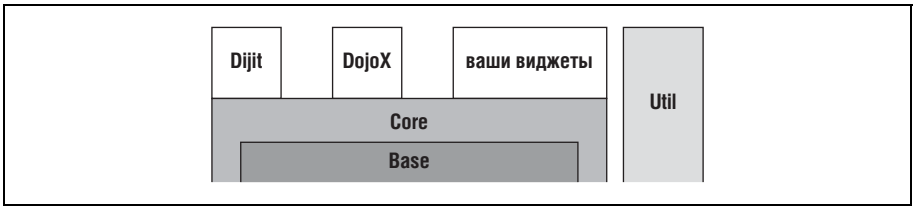



***Рис. 1.1.*** *Один из вариантов представления взаимоотношений между различными компонентами Dojo*

## Base

Ядром Dojo является *Base* – чрезвычайно компактная, высоко оптимизированная библиотека, образующая основу для всего остального инструментария. Кроме всего прочего библиотека Base содержит удобные языковые конструкции и утилиты AJAX, систему пакетов, позволяющую добавлять ресурсы из Dojo прямо во время выполнения, а не во время загрузки страницы. Эта библиотека также предоставляет инструменты для создания и управления иерархиями наследования, средства для выполнения практически универсальных запросов к дереву DOM с использованием селекторов CSS3 и конструкцию, нормализующую модели событий DOM, имеющиеся в разных броузерах. Все, что находится в библиотеке Base, доступно как функции или атрибуты пространства имен верхнего уровня `dojo.*`. Библиотека Base представлена единственным файлом *dojo.js*, который при передаче по сети занимает порядка 30 Кбайт. Если вы задумаетесь над тем, что размер большинства рекламных Flash-роликов, наводнивших Всемирную паутину, составляет значительно больше 30 Кбайт, то это число покажется вам удивительно маленьким.

Если взглянуть на фактический размер файла *dojo.js* на диске, вы увидите, что он составляет что-то около 80 Кбайт, но так как серверы обычно обрабатывают содержимое, отправляемое броузерам «по кабелю», это содержимое сжимается, что позволяет уменьшить время, необходимое для его загрузки. Если вы попробуете вручную сжать файл *dojo.js* с помощью утилиты gzip, у вас должен получиться архив, размер которого составляет примерно одну треть от размера первоначального файла.

Еще одна интереснейшая особенность библиотеки Base состоит в том, что она способна выполнять *самонастройку* Dojo автоматически, достаточно лишь включить файл *dojo.js* в страницу. В упрощенном представлении процедура самонастройки включает определение параметров среды, сглаживание различий между броузерами и загрузку пространства имен `dojo`. Кроме того, могут использоваться дополнительные параметры для выполнения автоматического парсинга любых виджетов

на странице и выполнения других задач по инициализации. (Обо всех этих возможностях будет рассказываться в последующих главах.)

Библиотека Base предоставляет великое множество вспомогательных функций для реализации многих стандартных операций, которые часто бывает необходимо выполнить для достижения поставленных целей. Даже если вы ничего больше не будете использовать из набора инструментальных средств, библиотека Base наверняка станет для вас ценным ресурсом, от которого уже не захочется отказаться, стоит только почувствовать, как она позволяет повысить производительность труда. Без библиотеки Base не будет и Dojo – все, что имеется в комплекте инструментальных средств Dojo, так или иначе, зависит или построено на основе этой библиотеки.

Содержимое библиотеки Base становится все более организованным и постоянным, поэтому со временем внутри проекта термин «Base» используется все меньше и меньше, и вы можете даже услышать, что термины «Base» и «dojo.js» стали взаимозаменяемыми.

## Core

Библиотека Core основана на библиотеке Base и предлагает дополнительные средства для парсинга виджетов, улучшенные анимационные эффекты, средства реализации механизма «перетащил и бросил», интернационализации (i18n), обработки кнопки «Назад», управления cookie и многого другого. Ресурсы, доступные в Core, используются довольно часто и обеспечивают фундаментальную поддержку для выполнения наиболее распространенных операций, но они не считаются достаточно универсальными, чтобы включить их в состав библиотеки Base. Хотя критерии внесения ресурсов в состав библиотеки Core можно считать несовершенными, но система управления пакетами в инструментальном наборе Dojo упрощает подключение дополнительных модулей и ресурсов по мере необходимости с помощью простого механизма, который работает на манер директивы `#include` из языка C или инструкции `import` из языка Java.

Вообще, различия между Base и Core определяются просто: если явно импортируемый ресурс, внешний по отношению к *dojo.js*, принадлежит к пространству имен `dojo`, значит, он является частью библиотеки Core. Обычно средства из библиотеки Core не входят непосредственно в пространство имен библиотеки Base, а составляют пространства имен более низкого уровня, такие как `dojo.fx` или `dojo.data`.

## Dijit

Сказать, что Dojo – это всего лишь стандартная библиотека языка JavaScript, значит ничего не сказать. Кроме всего прочего в состав инструментального набора Dojo входит *Dijit* (сокращенно от «Dojo widget») –

фантастическая библиотека виджетов, готовых к использованию, для работы с которыми часто вообще не требуется писать программный код JavaScript. Диджиты соответствуют общепринятым стандартам доступности, таким как ARIA[1], и поставляются с предварительными интернациональными настройками для большинства распространенных языковых настроек. Библиотека Dijit построена непосредственно на базе Core (и тем самым полностью зависит от целостности Core), поэтому, когда вам потребуется создать свой собственный виджет, вы будете использовать те же самые стандартные блоки, которые использовались для всего, что имеется в Dijit. Виджеты, которые создаются средствами Dojo, обладают высокой степенью переносимости и легко могут разворачиваться на любом веб-сервере или даже запускаться локально, вообще без участия веб-сервера, с помощью протокола *file://*.

Включение диджитов в веб-страницу выполняется простым определением специального атрибута `dojoType` внутри обычного тега HTML – воплотившаяся в реальность мечта дизайнеров и пользователей, которых мало интересует (если вообще интересует) программирование на языке JavaScript. Основное преимущество от использования библиотеки Dijit состоит в том, что она фактически позволяет получать богатые функциональные возможности без необходимости погружаться в детали реализации. Даже если вы занимаетесь созданием библиотек или разработкой собственных виджетов, следование стилю и соглашениям, принятым в Dijit, обеспечит вашим виджетам высокую переносимость и простоту в использовании, составляющим основу любого программного компонента многократного использования.

Вся линейка виджетов библиотеки Dijit грубо может быть разделена на прикладные виджеты общего назначения, такие как индикаторы хода выполнения процесса и модальные диалоги, виджеты верстки, такие как вкладки и раздвижные панели, и виджеты форм, которые представляют собой значительно улучшенные версии стандартных элементов, такие как кнопки и разнообразные поля ввода.

## DojoX

DojoX – это коллекция проектов, название которой происходит от «Dojo Extensions» (расширения для Dojo), хотя ее часто называют «Extensions and Experimental» (расширения и экспериментальные реализации). Проекты «расширений» в DojoX – это стабильные виджеты и ресурсы, которые представляют собой ценные дополнения, назначение которых просто не соответствует тому, чтобы поместить их в библиотеку Core или в библиотеку Dijit. К «экспериментальным» проек-

[1] Стандарт, направленный на улучшение доступности активных Интернет-приложений (Accessible Rich Internet Applications): *http://www.w3.org/WAI/intro/aria*.

там относят виджеты, которые подвержены частым изменениям и находятся на начальной стадии разработки.

В составе каждого проекта, входящего в коллекцию DojoX, должен присутствовать файл *README*, где описывается состояние проекта. Проекты в DojoX обычно стремятся соответствовать инициативам по интернационализации и обеспечению доступности совместимым с Dijit способом, но вообще это редкий случай, когда такое соответствие находится на достаточно высоком уровне. Как бы то ни было, реализация значительного числа механизмов, используемых в действующих приложениях, находится именно в коллекции DojoX, включая виджет табличного представления данных, процедуры преобразования данных для наиболее типичных разновидностей веб-служб и т. д. Кроме того, в состав коллекции DojoX входят испытательный полигон и инкубатор для новых идей, что гарантирует отсутствие угрозы стандартному и стабильному прикладному интерфейсу ресурсов в Core и Dijit. В этом смысле DojoX обеспечивает тонкий баланс в решении критических проблем, занимающих центральное место в любом проекте, поддерживаемом сообществом.

## Util

Util – это коллекция вспомогательных инструментов Dojo, включая платформу модульного тестирования и инструменты сборки, позволяющие создавать свои версии библиотеки инструментальных средств Dojo для собственных нужд. Платформа модульного тестирования, DOH[1], не имеет прямого отношения к Dojo и представляет собой простой набор конструкций, которые могут использоваться для автоматизации проверки качества любого программного кода JavaScript. В конце концов, вы ведь хотите реализовать четкое и систематическое тестирование своего программного кода JavaScript, разве не так?

Основное назначение инструментов сборки состоит в том, чтобы сократить размер вашего программного кода и объединить его в несколько слоев, где каждый слой не что иное, как набор других файлов JavaScript. Сжатие производится посредством инструмента ShrinkSafe – адаптированной проектом Mozilla версии механизма из Rhino JavaScript, который сжимает программный код JavaScript, не повреждая общедоступный программный интерфейс, а объединение выполняется с помощью коллекции сценариев, которые также были разработаны для Rhino. Другие вспомогательные компоненты в библиотеке Util выполняют такие операции, как встраивание строк шаблонов HTML (подробнее об этом будет говориться во время формального представления

[1] Возможно, вы уже обратили внимание на каламбур, поскольку DOH – это также известное восклицание Гомера Симпсона: испытательный механизм может при определенных настройках воспроизводить звуковой эффект, напоминающий восклицание «D'oh!» при неудачном прохождении теста.

библиотеки Dijit в главе 11) в файлы JavaScript – еще один прием, позволяющий уменьшить задержки.

Читая этот раздел, вы, возможно, понимаете, что могут предложить инструменты сборки, но вы можете не совсем четко представлять себе, зачем это может вам потребоваться. Проще говоря, инструменты сборки позволяют объединить и уменьшить объем программного кода JavaScript и тем самым *значительно* сократить время загрузки страницы, что дает существенные преимущества, когда приложение переходит в стадию эксплуатации.

Подобно DOH механизм ShrinkSafe может применяться независимо от Dojo, и нет никаких причин, чтобы не использовать его при подготовке рабочей версии программного кода JavaScript, особенно если учесть, что он позволяет сократить объем кода JavaScript до 50% и больше. Разница во времени между загрузкой множества крупных файлов JavaScript посредством серии синхронных запросов и получением одного или двух сжатых файлов JavaScript может оказаться весьма существенной.

# Подготовка к работе

Чтобы приступить к разработке приложений с использованием Dojo, вам не потребуется приобретать какие-либо дополнительные инструменты или настраивать веб-сервер, такой как Apache. Во всей этой книге есть всего несколько примеров, которые требуют взаимодействия с веб-сервером. Доступ к большинству ресурсов может быть организован с использованием относительных путей на локальной машине или посредством загрузки их из другого домена, поэтому по большей части вам будет достаточно вашего любимого текстового редактора и веб-броузера.

Существует три основных способа загрузить Dojo и подготовить ее к работе: загрузить официальный выпуск, получить самую последнюю версию из репозитария Subversion и использовать кросс-доменную (XDomain) сборку, доступную в специализированной сети доставки содержимого AOL (Content Delivery Network, CDN). Мы поговорим о каждом из этих вариантов. Обычно предпочтительнее загрузить официальный выпуск на локальную машину, однако иногда могут быть причины для выбора других вариантов.

## Получение Dojo

Существует три основных способа воспользоваться инструментальным набором Dojo: загрузить официальный выпуск на локальную машину, получить копию из репозитария Subversion и использовать сборку XDomain, доступную в специализированной сети доставки со-

держимого AOL (Content Delivery Network, CDN). В этом разделе рассматривается каждый из представленных вариантов

## Загрузка официального выпуска

Загрузка самой последней официальной версии Dojo является наиболее традиционным способом подготовиться к работе с этим инструментальным набором. «Официальная» версия – это не более чем снимок репозитария Subversion, тщательно протестированный и сопровождаемый некоторыми полезными примечаниями к выпуску. Официальные выпуски инструментария можно найти по адресу *http://dojotoolkit.org/downloads*. Единственный существенный недостаток такого подхода состоит в том, что официальный выпуск не включает в себя инструменты сборки. Чтобы получить их, вам потребуется либо воспользоваться репозитарием Subversion, либо загрузить выпуск с полными исходными текстами, который можно найти по адресу *http://download.dojotoolkit.org/*.

При распаковке архива вы увидите, что он разворачивается в папку с именем *dojo-release-x.y.z*, где «x», «y» соответствуют старшему, младшему номерам версии, а «z» – номеру обновления конкретного выпуска. Чтобы обеспечить максимальную универсальность, можно переименовать папку, дав ей другое имя, например *js* (от JavaScript). Из других способов можно упомянуть возможность использования директив в конфигурационных файлах веб-сервера с целью создания псевдонима *js* для папки *dojo-release-x.y.z* или использования символических ссылок в операционных системах Linux и UNIX. В любом случае эти действия принесут дополнительные преимущества, позволив использовать для доступа к Dojo такие относительные пути, как *www/js*, вместо, например, *www/dojo-release-x.y.z*.

Создать символическую ссылку очень просто. В операционных системах Linux, Mac OS X или UNIX достаточно просто выполнить в окне терминала следующую команду: `ln -s dojo-release-x.y.z js`. Подробнее о символических ссылках можно прочитать на страницах справочного руководства, выполнив команду `man ln`.

После того как инструментальный набор Dojo будет загружен, вас вначале может удивить тот факт, что это вообще не единственный файл JavaScript, но не стоит беспокоиться. Быстрый взгляд на распакованное содержимое покажет, что весь программный код разбит на те же самые архитектурные компоненты, о которых мы говорили в предыдущем разделе – Base (*dojo/dojo.js*), Core (*dojo*), Dijit (*dijit*), DojoX (*dojox*) и Util (*util*). Хотя мы будем работать со всеми этими компонентами, единственное, что необходимо сделать для подключения библиотеки Base к веб-странице, – это указать относительный путь к файлу *dojo.js* (указав *dojo/dojo.js* в теге `SCRIPT`, точно так же, как при использовании любого другого файла JavaScript). Все очень просто.

## Загрузка из репозитария Subversion

Возможно, вы пожелаете использовать в своей работе последнюю официальную сборку инструментального набора. Однако, если вы заинтересованы в поддержании самой свежей копии репозитария Subversion, то внимательно прочитайте этот раздел – здесь описываются шаги, которые необходимо выполнить для получения инструментального набора Dojo из репозитария Subversion и настройки удобной среды разработки. Использование версии из репозитария Subversion может оказаться полезным, когда возникает необходимость следить за исправлением какой-либо ошибки, когда появляется желание опробовать в работе новую особенность или если вы являетесь первоклассным специалистом, который не может жить спокойно, если не обладает самой последней версией программного обеспечения.

Авторитетное руководство по системе контроля версий Subversion вы найдете в электронной книге «Version Control with Subversion» по адресу: *http://svnbook.red-bean.com/*.[1]

Репозитарий Subversion с инструментальным набором Dojo находится по адресу *http://svn.dojotoolkit.org/src/*, поэтому начните с этого адреса, если вам интересно ознакомиться с некоторым программным кодом с помощью вашего веб-броузера. Возможно, вы пожелаете получить программный код всего набора инструментальных средств одним махом из внешнего представления, доступный по адресу: *http://svn.dojotoolkit.org/src/view/anon/all/trunk*.

Свойство Subversion создавать внешние представления обеспечивает возможность делать рабочие копии, для получения которых в обычной ситуации могло бы потребоваться выполнить несколько отдельных операций с репозитарием. Подробнее об этом можно прочитать по адресу *http://svnbook.red-bean.com/en/1.0/ch07s03.html*.[2]

Чтобы загрузить программный код из репозитария, необходимо выполнить следующую команду в окне терминала (далее в этом разделе предполагается, что загрузка выполняется в папку с именем *www*):

```
svn co http://svn.dojotoolkit.org/src/view/anon/all/trunk ./svn
```

После окончания загрузки из репозитария Subversion у вас появится папка с именем *svn*, содержащая подкаталоги, соответствующие тем же основным компонентам набора инструментальных средств (*dojo*,

---

[1] Самое интересное, что при правильных языковых настройках в броузере открывается русская версия книги с названием «Управление версиями в Subversion». – *Прим. перев.*

[2] Тот же раздел, но на русском языке: *http://svnbook.red-bean.com/nightly/ru/svn.advanced.externals.html*. – *Прим. перев.*

*dijit*, *dojox* и *util*). Однако на этот раз в папке *util* будут находиться сценарии сборки (и, возможно, ряд дополнительных вспомогательных инструментов, используемых для обеспечения поддержки инструментария). Мы не будем касаться тонкостей работы с системой Subversion, но заметим, что вполне возможно иметь несколько версий Dojo, например, последний официальный выпуск, ежедневную сборку и фактическую копию репозитария, и выполнять переключение между ними с помощью директив настройки сервера или другими средствами в зависимости от того, какую версию желательно использовать в каждый конкретный момент времени.

## Версия CDN AOL

В специализированной сети доставки содержимого AOL (AOL CDN) поддерживается кросс-доменная версия Dojo, доступ к которой можно легко организовать несколькими конфигурационными параметрами и включением тега `SCRIPT`, который ссылается на сборку XDomain инструментального набора Dojo на сервере AOL CDN. Поскольку это достаточно просто, все примеры в этой книге будут использовать сборку XDomain, благодаря чему от вас потребуются минимальные усилия, чтобы опробовать их.

Как говорилось в двух предыдущих разделах, обычно загрузка Dojo производится путем указания ссылки на файл *dojo.js* – примерно так, как показано ниже:

```
<script
    type="text/javascript"
    src="www/js/dojo/dojo.js">
</script>
```

Перейти на использование сборки XDomain очень просто: достаточно изменить ссылку в атрибуте `src` и позволить Dojo и AOL сделать все остальное. Следующий ниже тег `SCRIPT` иллюстрирует этот процесс для Dojo 1.1:

```
<script
    type="text/javascript"
    src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js">
</script>
```

Обратите внимание на имя файла *dojo.xd.js* в этом фрагменте – если по ошибке указать имя *dojo.js*, вы наверняка получите сообщение об ошибке вместо Dojo. Примечательно также, что *1.1* в адресе ссылки определяет последний исправленный выпуск указанной версии. Можно было бы запросить конкретный исправленный выпуск, например *1.1.0* или *1.1.1*. Возможно, вам также не будет лишним оставить у себя закладку на страницу *http://dev.aol.com/dojo*, потому что здесь приводится самая достоверная информация о версиях Dojo, доступных через CDN.

## Отладка с помощью Firebug

Если вам уже доводилось заниматься разработкой пусть даже самых тривиальных веб-приложений, вы наверняка знаете, что отладка порой может превратиться в настоящую головную боль. Это особенно верно, когда проблема связана с трудноуловимыми различиями между броузерами или когда поведение броузера несколько отличается от стандартов W3C. К сожалению, добавление в приложение дополнительных мощных инструментальных средств иногда может только осложнить отладку, и это особенно верно для языка JavaScript с его динамическими замыканиями, динамической типизацией и неудобным выводом отладочных сообщений с помощью окон с предупреждениями. И вот появился *Firebug* – удивительное расширение для Firefox, позволяющее упростить отладку и веб-разработку.

Всерьез подумайте над тем, чтобы вести разработку с использованием броузера Firefox, потому что его расширение Firebug, да и он сам, позволяют невероятно ускорить процесс разработки. С помощью этого расширения вы сможете просматривать и управлять любыми элементами в дереве DOM (включая и стили) в режиме реального времени, регистрировать события в консоли Firebug и получать часто достаточно точную информацию об ошибках, которая позволит идентифицировать фактическую проблему. (Сравните это с гениальным окном предупреждения, вопрошающим: «Желаете приступить к отладке?».)

При этом, конечно же, не забывайте проверять степень кросс-платформенности своего проекта, проверяя его работоспособность в IE и в других броузерах. Несмотря на то, что Dojo прилагает максимум усилий для обхода индивидуальных особенностей броузеров, тем не менее иногда могут возникать некоторые аномалии, и чем раньше они будут обнаружены, тем лучше.

Расширение Firebug – это удивительный инструмент. Испытав его один раз, вы будете с трудом без него обходиться. Вы сэкономите массу времени, если будете вести разработку в броузере Firefox и использовать Firebug при решении любых проблем. Кроме того, будет совсем нелишним чаще тестировать разрабатываемое приложение в IE (скажем, через каждые 30 минут), чтобы обнаруживать трудноуловимые ошибки, которые могут прокрасться в приложение. Например, если при определении ассоциативного массива JavaScript поставить запятую после последней пары ключ/значение, броузер Firefox «простит» вам эту оплошность, но IE – нет..., а сообщения об ошибках, которые выводит IE, не отличаются информативностью.

Вне всякого сомнения, наиболее часто используемой функцией из Firebug является `console.log`, которая позволяет выводить информацию в консоль Firebug непосредственно из программного кода JavaScript. (Мы все уже устали от окон с предупреждениями, разве не так?)

Известно, что Dojo стремится настолько тесно интегрироваться с Firebug, насколько это вообще возможно, поэтому в ее состав входит версия Firebug Lite. Поэтому, даже если вы будете вынуждены вести разработку с применением другого броузера, такие функции, как `console.log`, всегда будут доступны, если в них возникнет необходимость.

Броузер Firefox и расширение Firebug можно получить по адресам *http://www.getfirefox.com* и *http://www.getfirebug.com* соответственно. В приложении А содержится учебное руководство по Firebug, который может оказаться для вас полезным.

## Настройка системы безопасности броузера для запуска Dojo на локальной системе

Большинство примеров в этой книге предусматривают возможность запуска с использованием локального протокола *file://*, который в общем и целом работает весьма неплохо. Однако пользователям броузера Firefox 3 *может* потребоваться настроить один параметр, чтобы иметь возможность запускать локальную копию Dojo:

1. В строке адреса нужно ввести `about:config` и нажать клавишу Enter.
2. В параметре `security.fileuri.origin_policy` установить значение 3 или выше.

Подробнее об этой проблеме можно прочитать по адресу: *http://kb.mozillazine.org/Security.fileuri.origin_policy*.

## Легковесный сервер ответов

Практически все примеры в этой книге могут быть опробованы без применения веб-сервера. Однако, для полноценной демонстрации *отдельных компонентов* инструментария полезно организовать получение динамически изменяемого содержимого с сервера. Когда наступит такая необходимость, мы будем использовать CherryPy (версии 3.1+) – чрезвычайно простой в использовании сервер, способный отвечать на запросы, написанный на языке Python. Познакомиться с CherryPy и загрузить его можно на сайте *http://cherrypy.org*. Но не расстраивайтесь раньше времени – вы не погрязнете в трясине подробностей о новом веб-сервере, которые придется изучать попутно с изучением Dojo.

Установить CherryPy совсем несложно, достаточно загрузить его и выполнить несколько коротких инструкций из файла *README*. Модуль CherryPy устанавливается точно так же, как и любой другой модуль Python, поэтому в данном случае не приходится говорить о каталоге установки. В отличие от других, более сложных серверных технологий, модуль CherryPy просто становится доступным для использования всякий раз, как только будет импортирован с помощью инструкции `import`, как и любой другой модуль Python. В действительности CherryPy – это не более чем самостоятельное приложение на языке Python, которое управляет своим собственным многопоточным веб-сервером,

благодаря этому выполнить «сценарий на стороне сервера» так же просто, как запустить простую команду в окне терминала.

Для опробования всех примеров (которых очень немного), требующих получения динамического содержимого или явного ответа от сервера, достаточно выполнить всего одну команду в окне терминала, поэтому не стоит пугаться – для работы с этими примерами, использующими сервер, *не обязательно* обладать железными нервами. Конечно, вы можете просто не выполнять все примеры, затрагивающие использование серверных технологий, а полагаться на сопровождающие их исчерпывающие объяснения.

Например, если бы вас попросили запустить следующее простое приложение, хранящееся в файле *hello.py*, то все, что вам пришлось бы сделать для этого, – просто ввести в командной строке команду `python hello.py`. Это все, что требуется для запуска CherryPy и приема запросов, поступающих в порт с номером 8080. Если модуль CherryPy уже установлен, инструкция `import cherrypy`, как показано в примере 1.1, отыщет его и сделает его доступным для использования.

*Пример 1.1. Очень простое приложение, использующее модуль CherryPy*

```
1 import cherrypy
2
3 class Content:
4
5     @cherrypy.expose
6     def index(self):
7         return "Hello"
8
9     @cherrypy.expose
10    def greet(self, name=None):
11        return "Hello "+name
12
13 cherrypy.quickstart(Content())
```

Если запустить приложение, приведенное в примере 1.1, и в адресной строке веб-броузера ввести адрес *http://localhost:8080/*, тем самым вы обратитесь к методу `index` (строки 6-7) и должны получить в ответ строку «Hello», а если ввести адрес *http://localhost:8080/greet?name=Dojo*, произойдет обращение к методу `greet` (строки 10-11), который обработает параметр `name` из строки запроса и вернет ответ «Hello Dojo». Это самый типичный пример, проще которого невозможно придумать, и к тому же он демонстрирует, насколько просто выглядит программный код на языке Python.

В этой книге от вас не требуется писать или понимать программный код на языке Python; цель предыдущего примера состояла лишь в том, чтобы показать, как легко и просто можно воспользоваться модулем CherryPy и интерпретатором Python, если вам потребуется воспроиз-

вести свое собственное динамическое содержимое. Замечательным руководством по языку Python может служить книга Марка Лутца (Mark Lutz) «Learning Python» (O'Reilly)[1], если вам когда-либо потребуется писать более сложный программный код на языке Python. Значительный объем документации можно также найти на официальном сайте Python *http://www.python org* и на сайте проекта CherryPy *http://cherrypy.org*.

# Терминология

Для нас будет полезно потратить несколько минут на выяснение значений некоторых терминов, которые мы будем использовать по всей книге. Конкретное описание механики работы JavaScript на профессиональном сленге требует применения точной терминологии, и это описание становится еще более расплывчатым, когда вы приступаете к созданию мощного набора инструментов на его основе, не говоря уже о наборе инструментов, в котором предпринимается попытка имитировать классы на языке, где отсутствуют нормальные классы в объектно-ориентированном понимании.

Хотелось бы надеяться, что следующий список терминов будет для вас полезен, когда мы начнем погружаться в эти темные воды:

*Инструментальный набор*

*Инструментальный набор* – это всего лишь совокупность инструментов. Так случилось, что в мире программирования инструментальные наборы часто используются в контексте разработки пользовательского интерфейса. Более точным будет назвать Dojo инструментальным набором, потому что это больше, чем библиотека с программным кодом поддержки, состоящая из набора взаимосвязанных функций и абстракций, – в состав Dojo также входят такие компоненты, как утилиты развертывания приложений, инструменты тестирования и система пакетов. Довольно легко можно запутаться в таких терминах, как библиотека, платформа, инструментальный набор и т. д., но Dojo был назван инструментальным набором, поэтому мы и будем придерживаться этого определения.

*Модуль*

Физически в терминологии Dojo *модуль* – это не более, чем файл или каталог, содержащий совокупность взаимосвязанных файлов с программным кодом JavaScript. Каталог верхнего уровня, в свою очередь, образует *пространство имен* для программного кода, который в нем содержится. С логической точки зрения модули в Dojo концептуально больше напоминают пакеты в других языках программирования, где они используются для выделения категорий

[1] Марк Лутц «Изучаем Python». – Пер. с англ. – СПб.: Символ-Плюс, 2008. – *Прим. перев.*

программных компонентов. Однако, следует заметить, что под *системой пакетов* в Dojo в действительности подразумевается механизм, выполняющий такие операции, как определение и удовлетворение зависимостей, при этом сами модули Dojo не являются «пакетами».

*Ресурс*

Когда это необходимо, модуль Dojo может быть поделен на несколько файлов, но даже когда модуль состоит из единственного файла с программным кодом JavaScript, каждый такой файл называется *ресурсом*. Обычно ресурсы используются исключительно для логической организации различных абстракций, присутствующих в модуле, но иногда деление на ресурсы производится с целью уменьшить размеры файлов JavaScript. При этом важно найти компромисс, чтобы в результате такого деления не приходилось загружать слишком много лишнего программного кода, но и не приходилось загружать слишком много маленьких файлов, так как каждый файл загружается в результате выполнения отдельного синхронного запроса к веб-серверу, что приводит к увеличению нагрузки на него (хотя использование инструментов сборки может снять проблему перегрузки).

*Пространство имен*

Физически *пространства имен* в Dojo являются отображением иерархии расположения модулей и ресурсов в файловой системе. С логической точки зрения пространства имен предотвращают конфликты имен модулей и ресурсов. Примечательно, что пространства имен, не являясь сами по себе ни модулями, ни ресурсами, тем не менее напрямую отражают иерархию модулей и ресурсов. Кроме того, следует заметить, что Dojo предохраняет глобальное пространство имен страницы и любые модули, которые будут создаваться с помощью Dojo, при правильной реализации не будут загрязнять глобальное пространство имен. Вспомните, что все содержимое модуля Base располагается в пространстве имен верхнего уровня `dojo`.

*Первоклассный*

В программировании *первоклассным* называется все, что может передаваться между различными участками программы без каких-либо ограничений, присущих некоторым другим сущностям того же самого языка. Например, во многих языках программирования невозможно передавать функции так же просто, как данные других типов, такие как числовые или строковые значения. В этом конкретном случае функции не могли бы считаться первоклассными объектами. В нашей дискуссии этот термин будет использоваться, чтобы подчеркнуть тот факт, что функции в языке JavaScript являются первоклассными объектами. Позже будет показано, что такие операции, как присваивание функций переменным и/или помеще-

ние их в ассоциативные массивы, представляют основу для многих шаблонов проектирования, используемых в Dojo.

*Функция*

*Функция* – это фрагмент программного кода, который определяется один раз, но используется многократно. В языке JavaScript функции являются первоклассными объектами, которые могут передаваться между различными частями программы точно так же, как любые другие переменные. *Функция-конструктор* – это функция, предназначенная для использования совместно с оператором `new`, с помощью которого создается новый объект JavaScript типа *Function* и выполняется инициализация этого объекта. Примечательно, что все *объекты* JavaScript наследуют встроенный тип `Object` и обладают свойством `prototype`, которое служит основой механизма наследования в JavaScript, базирующегося на построении цепочек прототипов. В терминологии Dojo термин *конструктор* может относиться к анонимной функции, которая является отображением ключевого слова `constructor` в ассоциативном массиве `dojo.declare`, который в свою очередь используется для инициализации *свойств классов* Dojo.

*Объект*

В самом общем смысле под *объектом* в языке JavaScript понимается составной тип данных, который может содержать произвольное число именованных *свойств*. Например, простейшая инструкция `var o = {}` представляет литеральное определение *объекта* JavaScript. Термин «объект» в этой книге используется так часто, что иногда, для обозначения конструкций, содержащих пары ключ/значение, таких как `{a:1, b:2}`, используется не термин «объект», а ассоциативный массив. С технической точки зрения в языке JavaScript имеются только объекты, здесь нет никаких классов, хотя Dojo имитирует понятие класса посредством функции `dojo.declare` – специальной функции, используемой исключительно для этой цели.

*Свойство*

В объектно-ориентированном программировании (ООП) любые элементы данных, хранящиеся в классах, называются *свойствами*. В нашем обсуждении Dojo этот термин будет относиться к данным, хранящимся в объектах типа Function, или к данным, хранящимся в классах Dojo, которые определяются с помощью функции `dojo.declare`.

*Метод*

Функция, являющаяся членом класса, часто называется *методом*, как в контексте ООП, так и в JavaScript и Dojo. Кроме того, в терминологии Dojo анонимные функции, присутствующие в инструкции `dojo.declare`, также называются методами, потому что `dojo.declare` является основой механизма наследования классов. Вообще говоря,

вы вполне можете считать методы функциями, определяемыми в классе, и доступ к которым впоследствии будут осуществляться через контекст объекта.

*Класс*

В Dojo объявление логической сущности посредством функции `dojo.declare` (специальная функция, используемая для имитации классов и иерархий наследования) называется *классом*. Еще раз напомню, что этот термин используется в более широком смысле, чем в таких языках программирования, как Java или C++, потому что язык JavaScript не поддерживает классы.

*Виджет*

В Dojo *виджеты* – это *объекты типа Function*, которые создаются посредством инструкции `dojo.declare`, включающей `dijit._Widget` (базовый класс для всех виджетов) в качестве предка. Обычно виджет имеет визуальное представление на экране и логически связан с разметкой HTML, CSS, JavaScript и статическими ресурсами в единую сущность, которой легко можно манипулировать, сопровождать и передавать, как обычный файл.

## Самонастройка

В этом разделе обсуждаются некоторые темы, с которыми можно для начала ознакомиться лишь поверхностно, а потом, когда вы будете чувствовать себя более уверенно, вернуться и ознакомиться с ними подробнее.

Прежде чем вы сможете приступить к использованию Dojo, необходимо подключить этот набор инструментов к странице. Независимо от того, установили ли вы локальную копию Dojo или используете версию AOL CDN, вам необходимо включить в страницу тег `SCRIPT`, в котором указать ссылку на файл с программным кодом JavaScript, а затем волшебные эльфы, что живут в вашем веб-броузере, чудесным образом сделают так, что «все заработает само собой», так ведь? Так, но не совсем так. В компьютерном мире в большинстве случаев все основано на простой автоматизации, и процесс самонастройки Dojo не является исключением.

Термин «самонастройка» относится к идее «вытаскивания самого себя за волосы». Другими словами, эта идея заключается в том, чтобы программное обеспечение могло самостоятельно приступить к работе без посторонней помощи. По сути процесс самонастройки Dojo – это то же самое, что «загрузка» компьютера после включения питания.

Так, в примере 1.2 показано, какие минимальные усилия требуется приложить, чтобы иметь возможность использовать некоторые воз-

можности XDomain Dojo в странице HTML. Примечательно, что при загрузке Dojo из CDN *по кабелю передается менее 30 Кбайт данных*. Велика вероятность, что выделенный блок программного кода – с небольшими модификациями – вы часто будете использовать в своих страницах. Сэкономьте свое время и скопируйте этот пример в виде шаблона для последующего использования.

*Пример 1.2. Пример минимального приложения*

```
<html>
  <head>
    <title>Title Goes Here</title>
    <!-- Простая таблица стилей, которая обеспечивает одинаковое отображение
         в разных броузерах -->
    <link rel="stylesheet" type="text/css"
          href="http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css" />

    <script
      type="text/javascript"
      src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js">
    </script>

    <script type="text/javascript">
      /* В случае необходимости модули Dojo могут загружаться асинхронно,
         посредством инструкций dojo.require... */

      dojo.addOnLoad(function() {

        /* Все содержимое, которое загружается посредством dojo.require,
           используется здесь... */
      });
    </script>

  </head>
  <body>
    <!-- ... -->
  </body>
</html>
```

Функция `dojo.addOnLoad` принимает в качестве параметра другую функцию. В примерах этой книги в качестве этого параметра передается анонимная функция, однако вполне возможно определить, например, такую функцию: `init = function(){/*...*/}`, и передавать ее. Анонимные функции и некоторые из причин, почему они играют такую важную роль, коротко рассматривались во вступлении.

В предыдущем примере присутствуют две новых конструкции – инструкция `dojo.require` и тело функции `dojo.addOnLoad`. Инструкция `dojo.require` будет обсуждаться на протяжении всего раздела «Управление исходным программным кодом с помощью модулей» в главе 2, но если представить ее вкратце, она добавляет в страницу именованные ресурсы для последующего использования и работает точно так же, как

инструкция `import` в языке Java или директива `#include` в языке C. Одна важнейшая особенность инструкции `dojo.require` состоит в том, что она выполняет загрузку локальных ресурсов *синхронно*, но при загрузке ресурсов из версии XDomain работает *асинхронно*. Такое различие имеет особенно важное значение, поскольку оно имеет отношение к функции `dojo.addOnLoad`.

## dojo.addOnLoad

Поскольку при загрузке ресурсов версии XDomain инструкция `dojo.require` работает асинхронно, не всегда бывает возможным сразу же использовать ресурс, который был указан в инструкции `dojo.require` при загрузке страницы[1], потому что задержка, вызванная передачей данных по сети, и другие факторы могут (а чаще всего так и случается) обусловить некоторую задержку в получении доступа к ресурсу. Если в такой ситуации попытаться обратиться к ресурсу, который еще не успел загрузиться, вы получите сообщение об ошибке и весь процесс самонастройки вероятнее всего прервется. На техническом языке такая ситуация, когда результат не определен из-за непредсказуемости временных ограничений, называется *гонкой за ресурсами*.

По только что упомянутым причинам желательно взять в привычку использовать функцию `dojo.addOnLoad`, потому что она обеспечивает максимальную переносимость страницам независимо от того, загружается ли версия XDomain или нет.

Распространенной ошибкой является отказ от использования `dojo.addOnLoad`, которая позволяет надежно и независимо от конкретного окружения задать логику того, что должно выполняться после загрузки содержимого страницы и после завершения выполнения всех инструкций `dojo.require`. Проблема обычно начинает проявляться, после того как по окончании разработки с использованием локальной копии производится переход на использование версии XDomain.

Учитывая, что в предыдущем фрагменте используется версия инструментального набора XDomain, нам не потребовалось выполнять дополнительных действий для установки локальной копии, поэтому процесс самонастройки всего инструментального набора в этом случае состоял из одного этапа.

Хотя виджеты не будут представлены вашему вниманию еще на протяжении нескольких глав, тем не менее следует упомянуть еще об одной особенности функции `addOnLoad`, пока мы обсуждаем эту тему. Функция `addOnLoad` не будет вызвана, пока не будут

[1] Вообще говоря, по окончании загрузки страницы вызывается либо обработчик события `onload` объекта `window`, либо обработчик события `DOMContentLoaded` в броузерах Mozilla.

разобраны все виджеты на странице (здесь предполагается, что Dojo настроен так, что парсинг виджетов производится при загрузке страницы).

Хотя процесс самонастройки – синоним «загрузки сценария» на высоком уровне, тем не менее в процессе загрузки происходит еще несколько событий, заслуживающих внимания. По большому счету, производятся, по меньшей мере, два действия, хотя и не обязательно именно в таком порядке:

*Настройка платформы*

Учитываются любые нестандартные параметры настройки, которые могут быть определены посредством `djConfig` – ассоциативного массива, который должен быть описан перед тегом `SCRIPT`, выполняющим загрузку Dojo, или в виде атрибута тега `SCRIPT`, выполняющего загрузку Dojo. Подробнее о массиве `djConfig` мы поговорим ниже в этой главе.

Определяется, какая версия Dojo должна быть загружена – локальная или кросс-доменная. Загрузка версии XDomain производится прозрачно – при условии, что есть соединение с Интернетом и во время сборки был настроен загрузчик версии XDomain. По умолчанию настройку загрузки версии XDomain производит *dojo.xd.js* (и другие компоненты **.xd.js*), который представляет собой замену стандартному ресурсу *dojo.js*.

На основе информации об окружении, для которого была создана конкретная сборка Dojo (обычно это броузер, но могут быть и другие варианты, такие как Rhino или мобильное устройство), производится настройка всех особенностей, зависящих от окружения. Хотя вам не придется выполнять какие-либо настройки на конкретный броузер при использовании версии Dojo по умолчанию, собранной для работы в этом броузере, тем не менее библиотека Base содержит такие элементы данных, как `dojo.isIE` и `dojo.isFF`, чтобы с их помощью можно было определить тип броузера в тех случаях, когда это действительно необходимо.

Выполняются действия, зависящие от типа броузера, такие как создание экземпляра объекта `XMLHttpRequest` (XHR), позволяющего выполнять асинхронные запросы при использовании различных асинхронных утилит AJAX в инструментальном наборе Dojo. Производится настройка методов преодоления несовместимостей между броузерами, например, нормализация модели событий DOM, стандартизация таблицы кодов клавиш, и предпринимаются дополнительные меры по минимизации и предотвращению потерь памяти.

*Загружается и вводится в действие пространство имен*

Вводится в действие пространство имен `dojo`, чтобы исключить конфликты между именами из инструментального набора и именами, существующими в странице.

Загружается пространство имен `dojo`, содержащее функции и имена, составляющие содержимое библиотеки Base.

Помимо функции `dojo.addOnLoad` существует еще одна функция – `dojo.addOnUnload`, которая используется гораздо реже и позволяет выполнять операции при закрытии страницы.

## Настройка с помощью djConfig

Большая часть сведений из этого раздела станет понятнее после того, как вы какое-то время поработаете над созданием программного кода, поэтому совсем необязательно сейчас задерживаться на нем. Эти сведения помещены здесь в значительной степени для справки.

В этом разделе рассматривается `djConfig` – массив с настройками, который можно поместить в тег `SCRIPT`, где выполняется самонастройка инструментального набора (или определить этот массив раньше, до начала процесса самонастройки Dojo), чтобы указать, откуда должны загружаться ресурсы, должны ли использоваться инструменты отладки и т. д.

В табл. 1.1 приводится краткое описание пар ключ/значение, которые можно определять в массиве, чтобы управлять процессом самонастройки. (Здесь встречаются и еще не представленные конструкции. Пока просто ознакомьтесь с ними, а когда потребуется, вернетесь обратно.)

Если определить массив `djConfig` после тега `SCRIPT`, который выполняет загрузку инструментального набора, он не будет иметь никакого эффекта.

*Таблица 1.1. Параметры настройки в массиве djConfig*

| Ключ | Тип значения (значение по умолчанию) | Комментарий |
|---|---|---|
| `afterOnLoad` | Boolean<br>(false) | Используется для упрощения добавления ресурсов Dojo в уже загруженную страницу. Удобно использовать при изучении принципа действия страницы или для разработки виджетов, которые должны загружаться, только когда в них возникает необходимость, например, в приложениях социальных сетей и прочих. |
| `baseUrl` | String<br>(undefined) | С технической точки зрения этот параметр позволяет переопределять пути к корню инструментального набора для локальной версии или для версии |

| Ключ | Тип значения (значение по умолчанию) | Комментарий |
|---|---|---|
| | | XDomain. Делается это обычно для разрешения зависимостей с нестандартными модулями. Однако на практике этот параметр используется почти исключительно для разрешения локальных модулей при запуске инструментария версии XDomain. |
| `cacheBust` | String\|Date (undefined) | В случае строкового (`String`) значения это значение будет добавлено в строку запроса модуля, чтобы предыдущая версия страницы не извлекалась броузером из локального кэша. Обычно для этого параметра выбирается случайное строковое значение, которое вы можете воспроизводить самостоятельно для уникальной идентификации версии вашего приложения, с целью предотвратить появление досадных ошибок, которые могут быть вызваны старыми версиями модулей.<br>В ходе разработки можно использовать в качестве значения дату (`Date`), например так: `(new Date()).getTime()`; это гарантирует, что при каждой загрузке страницы будет использоваться новое значение, и предотвратит появление нежелательных проблем, вызванных механизмом кэширования. |
| `debugAtAllCosts` | Boolean (false) | Обычно доставляет более подробную отладочную информацию за счет снижения производительности. Можно попробовать изменить это значение, чтобы точнее определить строку, в которой возникла ошибка, – если в сообщении говорится, что ошибка возникла в одном из файлов сборки, таких как *bootstrap.js*, *dojo.js* или *dojo.xd.js*. |
| `dojoBlankHtmlUrl` | String ("") | Используется, чтобы указать местоположение пустого документа HTML, необходимого при применении транспорта IFRAME через `dojo.io.iframe.create` (рассматривается в главе 4). По умолчанию используется *dojo/resources/blank.html*. |
| `dojoIframeHistoryUrl` | String ("") | Используется, чтобы указать местоположение специального файла, который применяется вместе с модулем `dojo.back` для управления поведением кнопки «Назад» |

*Таблица 1.1 (продолжение)*

| Ключ | Тип значения (значение по умолчанию) | Комментарий |
|---|---|---|
| | | броузера (рассматривается в главе 2). По умолчанию используется файл *dojo/resources/iframe_history.html*. |
| enableMozDomContentLoaded | Boolean (false) | В броузерах на основе механизма Gecko, таких как Firefox, для определения момента, когда страница будет полностью загружена, при желании можно использовать событие DOMContentLoaded, потому что при объеме документа больше 65536 байтов выполнение синхронных XHR-запросов сопряжено с некоторыми техническими проблемами.[a] |
| extraLocale | String или Array ("") | Используется для указания дополнительных языковых настроек, чтобы инструментарий Dojo мог прозрачно взаимодействовать с локализованными модулями. Значение может быть строкой (String) или массивом строк (Array). |
| isDebug | Boolean (false) | Для отладки использует механизм Firebug или Firebug Lite. Обратите внимание, что заглушки для отладочных функций, таких как различные методы console, остаются в месте по умолчанию, чтобы программный код не утратил работоспособность, если вы вдруг решите удалить диагностическую информацию из своего приложения. |
| libraryScriptUri | String ("") | Используется для настройки в окружениях, отличных от броузеров, таких как Rhino и SpiderMonkey (механизмы JavaScript). Действует примерно так же, как параметр baseUrl в броузерах. |
| locale | String (предоставляется броузером) | Используется для переопределения dojo.locale, когда желаемое значение отличается от того, что определяется броузером. |
| modulePaths | Object (undefined) | Совокупность пар ключ/значение, которые связывают имена модулей с относительными путями к ним на диске. Обычно вы будете помещать свои модули в корневой каталог инструментального набора, но в случае необходимости вы |

[a] *http://trac.dojotoolkit.org/ticket/1704*

| Ключ | Тип значения (значение по умолчанию) | Комментарий |
|---|---|---|
| | | сможете использовать этот параметр. При загрузке Dojo из CDN или из другого источника версии XDomain необходимо определить еще и параметр baseUrl. |
| parseOnLoad | Boolean (false) | Указывает, необходимо ли автоматически выполнять парсинг виджетов при загрузке страницы (по сути, управляет вызовом функции dojo.parser.parse() в соответствующий момент времени в процессе самонастройки). |
| require | Array ([]) | Обеспечивает удобный способ определять модули, которые должны быть загружены автоматически после загрузки ядра Base. При использовании совместно с параметром afterOnLoad определяет, какие ресурсы должны быть загружены после полной загрузки страницы. |
| useXDomain | Boolean (false) | Используется для принудительной загрузки версии XDomain. Это действие выполняется по умолчанию при использовании сборки XDomain. (Преимущества использования загрузки версии XDomain описываются в разделе «Преимущества использования сборок XDomain»). |
| xdWaitSeconds | Number (15) | Определяет число секунд ожидания загрузки затребованного ресурса при использовании кросс-доменной версии инструментария. |

Параметр в массиве djConfig называется modulePaths, хотя функция в библиотеке Base, выполняющая настройку путей к отдельным модулям, носит имя dojo.registerModulePath (без символа «s» в конце).

Чаще всего определение массива djConfig находится в том же теге SCRIPT, где выполняется загрузка инструментального набора, как показано в примере ниже:

```
<script
    type="text/javascript"
    src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js"
    djConfig="parseOnLoad:true,isDebug:true">
</script>
```

Однако при желании, например, когда необходимо указать большое число параметров или такой способ просто удобнее для вас в данной конкретной ситуации, массив можно определить в другом теге SCRIPT,

предшествующем тегу, выполняющему загрузку инструментария. Ниже приводится переделанный программный код предыдущего примера, который имеет тот же эффект:

```
<script type="text/javascript">
    djConfig = {
        parseOnLoad : true,
        isDebug : true
    };
</script>

<script
    type="text/javascript"
    src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js ">
</script>
```

# Исследование Dojo с помощью Firebug

Остальная часть книги последовательно рассматривает весь инструментальный набор, однако нам стоит потратить минутку, чтобы узнать, как можно исследовать Dojo с помощью консоли Firebug. В ходе разработки случаются моменты, когда бывает полезно опробовать некоторые идеи отдельно от разрабатываемого приложения, а консоль Firebug как раз обеспечивает среду, которая по своему поведению напоминает интерпретатор.

## Исследование библиотеки Base

Для иллюстрации сначала сохраним страницу HTML, представленную в примере 1.3, в локальном файле. Единственный нюанс в этом сценарии заключен в теге, выполняющем загрузку версии XDomain. Мы еще не рассматривали массив `djConfig`, определение которого присутствует в теге `SCRIPT`, однако это всего лишь способ передачи конфигурационной информации в Dojo на этапе самонастройки. В данном случае мы явно указываем, что средства отладки, такие как консоль Firebug, должны быть доступны для нас. Даже если вы используете другой броузер, такой как IE, параметр `djConfig="isDebug:true"` обеспечит загрузку Firebird Lite.

*Пример 1.3. Очень простая страница HTML для демонстрации некоторых особенностей библиотеки Base*

```
<html>
    <head>
        <title>Fun with Dojo!</title>

        <link rel="stylesheet" type="text/css"
              href="http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css" />

        <script
            type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js"
```

```
            djConfig="isDebug:true">
        </script>

        <style type="text/css">
            .blue {color: blue;}
        </style>
    </head>
    <body>
        <div id="d1" class="blue">A div with id=d1 and class=blue</div>
        <div id="d2">A div with id=d2</div>
        <div id="d2">Another div with id=d2</div>
        <div id="d4">A div with id=d4.
            <span id="s1">
                This sentence is in a span that's contained in d4.
                The span's id is s1.
            </span>
        </div>
        <form name="foo" action="">
            A form with name="foo"
        </form>
        <div id="foo">
            A div with id=foo
        </div>
    </body>
</html>
```

## Добавление Dojo

Если вы решите использовать Dojo для исследования страниц, полученных откуда-то из Сети, вы можете выполнить динамическую вставку сценария с помощью Firebug или букмарклета[1] (bookmarklet). Начиная с версии 1.1 в инструментальный набор были добавлены конфигурационные параметры `afterOnLoad` и `require`, обеспечивающие обычную последовательность обратных вызовов, которая возникает после загрузки страницы.

Ниже приводится фрагмент программного кода, выполняющий вставку Dojo в существующую страницу, который можно скопировать и выполнить в консоли Firebug. Единственное, что вам придется сделать, – это определить массив `djConfig` в виде отдельного объекта, а не внутри тега сценария, если вы хотите *гарантировать* вставку Dojo после загрузки страницы. Эта необходимость обусловлена тем, что броузеры немного по-разному обрабатывают динамические теги сценариев:

---

[1] Букмарклеты (bookmarklet) – это не что иное, как фрагменты программного кода JavaScript, которые можно сохранять в виде закладок. Обычно букмарклеты используют, чтобы наделить страницы некоторыми дополнительными особенностями поведения.

```
/*Определив в djConfig параметр afterOnLoad со значением true,
Вы также можете указать, какие модули должны быть загружены.
Предположим, что нам необходим модуль dojo.behavior.*/
djConfig={afterOnLoad:true,require:['dojo.behavior']}

/* Создать тег SCRIPT, который обычно добавляется в заголовок страницы*/
var e = document.createElement("script");
e.type="text/javascript";
e.src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js";

/* И вставить его. Что нам и требовалось */
document.getElementsByTagName("head")[0].appendChild(e);
```

Или еще лучше: создать букмарклет и загружать Dojo посредством горячей комбинации клавиш или щелкнув мышью на закладке. Для создания букмарклета необходимо обернуть предыдущий фрагмент программного кода в функцию и вызвать эту функцию на выполнение. Создайте в своем броузере закладку с именем «Dojo-ify» и используйте следующий фрагмент (весь программный код должен быть в одной строке) в качестве адреса закладки:

```
(function() {djConfig={afterOnLoad:true,require:['dojo.behavior']};
var e = document.createElement("script"); e.type="text/javascript";
e.src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js";
document.getElementsByTagName("head")[0].appendChild(e);})()
```

При активизации этой закладки вложенный в нее сценарий загрузит Dojo в страницу и обеспечит полный доступ к сборке инструментария Dojo от CDN. С этого момента вы можете заниматься исследованием страницы, как ваша душа пожелает. Только помните, что в Dojo 1.1 нет эквивалента функции `addOnLoad`, с помощью которого можно было бы определить, когда закончится загрузка инструментария (и любых затребованных модулей). Однако начиная с версии Dojo 1.2 такой эквивалент появился. Дополнительные сведения по этому вопросу вы найдете по адресу *http://www.oreillynet.com/onlamp/blog/2008/05/dojo_goodness_part_7_injecting.html*.

Другая интересная возможность добавления Dojo в страницу после того, как она будет загружена, заключается в использовании механизма отложенной загрузки виджетов в профилях приложений социальных сетей, в общественных профилях и т. д.

Сохранив страницу в файле, откройте его в Firefox и щелкните на маленьком зеленом кружочке с галочкой в нем, чтобы открыть консоль Firebug. Затем щелкните на ярлыке с изображением крышечки, рядом с полем ввода для поиска, чтобы открыть Firebug в новом окне, как показано на рис. 1.2. (Если вы еще не прочитали учебное руковод-

ство по Firebug в приложении А, тогда самое время, чтобы сделать это прямо сейчас.) Щелкнув на вкладке «Net» (Сеть), вы сможете убедиться, что файл *dojo.xd.js*, содержащий библиотеку Base версии XDomain, действительно был загружен с сайта CDN.

Если теперь опять щелкнуть на вкладке «Console» (Консоль), ввести `dojo` в строке приглашения к вводу `>>>` и нажать клавишу Enter, вы должны увидеть в консоли Firebug сообщение примерно с таким текстом: `Object _scopeName=dojo _scopeMap=Object`, который говорит о том, что глобальный объект `dojo` действительно жив и здоров. Введя команду `console.dir(dojo)`, вы получите исчерпывающий древовидный список всего, что содержится в Dojo. В этом списке вы увидите большое количество внутренних элементов, имена которых начинаются с символа подчеркивания, но пусть это вас не останавливает. Просмотрите весь список, это позволит вам получить некоторое представление о том, что находится в библиотеке Base.

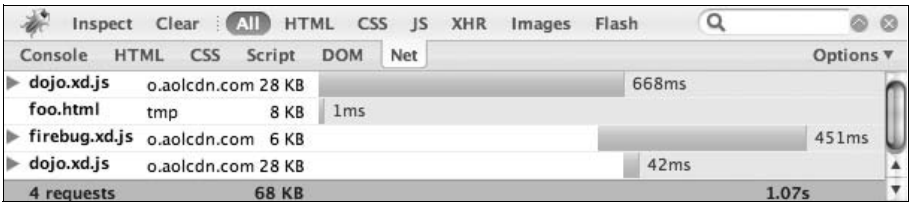



***Рис. 1.2.*** *Отладчик Firebug выводит ценную информацию, которую можно использовать для изучения того, что происходит с сетевыми запросами, и многого другого*

## dojo.byId

В наборе инструментальных средств Dojo имеется функция `dojo.byId`, которая служит заменой методу `document.getElementById`. То есть, если передать функции строковое значение, например, `dojo.byId("s1")`, она вернет ссылку, которую можно было бы сохранить в переменной, как и в случае использования метода `document.getElementById`. Однако, кроме того, что она отыскивает элемент по значению атрибута `id`, функция `dojo.byId` может также играть роль пустой операции, если ей передать ссылку на узел дерева DOM. Функция анализирует полученный аргумент, поэтому при разработке приложения вам даже не придется вспоминать, как она действует. Полная сигнатура этой функции выглядит, как показано ниже:

```
dojo.byId(/*String*/ id | /*DomNode*/ node) // Возвращает ссылку на узел DOM
```

На протяжении всей книги символ вертикальной черты | в сигнатурах функций будет использоваться для обозначения логической операции «ИЛИ» везде, где аргумент может иметь разные типы.

Может показаться, что функция `dojo.byId` выполняет те же действия, что и метод `document.getElementById`, и у вас вполне может возникнуть желание просто забыть о `dojo.byId`, но не торопитесь! Как оказывается, эта функция устраняет некоторые несовместимости между броузерами, которые могут приводить к появлению ошибок, когда вы меньше всего ожидаете этого. Одна из хорошо известных ошибок, связанных с применением метода `document.getElementById`, проявляется в IE6 и IE7. Для демонстрации сказанного введите следующую строку в консоли Firebug Lite после открытия документа с последним представленным примером, и вы увидите то, что приведено на рис. 1.3:

```
document.getElementById("foo") // Разве результат не очевиден?!?
```

Хм-м. Вероятно, вы не ожидали, что в качестве результата получите элемент `FORM`, не так ли? Оказывается, если бы этот элемент не стоял первым в документе, в качестве результата вы получили бы элемент `div`. Эта специфическая ошибка возникает из-за того, что в IE атрибуты `name` и `id` расцениваются как одно и то же. Поэтому так важно помнить о совместимости броузеров даже в самых обычных вещах! На рис. 1.4 показано, как Dojo защищает вас от соприкосновения с холодным металлом броузера, предохраняя вас от необходимости искать обходные пути вокруг множества несовместимостей, которые в противном случае препятствовали бы возможности создания переносимых приложений.

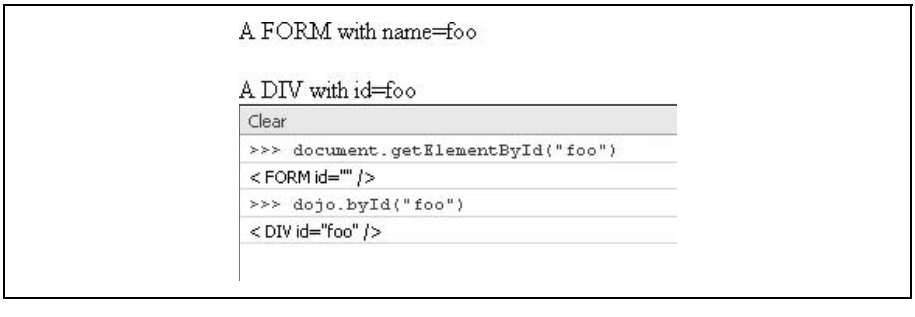



***Рис. 1.3.*** *Различия в поведении между document.getElementById и dojo.byId при работе с предыдущим документом*

***Рис. 1.4.*** *Инструментарий Dojo делает ваш программный код более переносимым, защищая вас от несовместимостей между броузерами*

Кроме того что dojo.byId ликвидирует конкретную несовместимость между броузерами, она также возвращает первый элемент, если в документе имеется несколько элементов с одинаковым значением атрибута id, тем самым нормализуя неоднозначность выбора. Убедиться в том, что dojo.byId всегда возвращает первый встретившийся элемент, можно, попробовав применить к нашему документу следующую инструкцию:

```
dojo.byId("d2").innerHTML
```

Самый главный вывод из этого короткого урока заключается в том, что если в разработке вы применяете инструментальный набор JavaScript, – используйте его прикладной интерфейс, вместо того чтобы искать обходные пути на простом JavaScript. Иногда вам могут встречаться вызовы API, которые на первый взгляд не несут никаких дополнительных преимуществ, и может появиться искушение вернуться к старым, испытанным способам, но не поддавайтесь искушению! Хорошо продуманный прикладной интерфейс не содержит бесполезных функций.

Если вы полагаете, что какая-то функция API «бесполезна», это, скорее всего, является признаком вашей плохой осведомленности об истинном назначении этой функции. Всякий раз, когда возникают подобные подозрения, обращайтесь к документации, чтобы выяснить, что именно вы пропустили. Если же документации не удастся убедить вас, обратитесь в список рассылки или в канал IRC и спросите кого-нибудь об этой функции.

## dojo.connect

Получение ссылки на узел дерева DOM, несомненно, ничем не интересно, поэтому давайте рассмотрим что-нибудь более интересное, например подключение события пользовательского интерфейса, такого как перемещение указателя мыши, к узлу с помощью функции dojo.connect – механизма инструментального набора для динамического добавления и удаления этих типов событий. Сигнатура функции может показаться сложной на первый взгляд, но функцией в действительности совсем просто пользоваться. Взгляните:

```
connect(/*Object|null*/ obj,
        /*String*/ event,
        /*Object|null*/ context,
        /*String|Function*/ method) // Возвращает дескриптор соединения
```

Чтобы опробовать действие функции connect, выполните приведенный далее программный код в консоли Firebug, а затем переместите указатель мыши на текст предложения, содержащегося в теге SPAN, чтобы убедиться, что обработка события mouseover настроена должным образом. (Возможно, вам придется щелкнуть на ярлыке с крышечкой в правом нижнем углу, чтобы переключить командную строку в многострочный режим ввода.)

```
var handle = dojo.connect(
    dojo.byId("s1"), //контекст
    "onmouseover",   //событие
    null,            //контекст
    function(evt) {console.log("mouseover event", evt);} //обработчик события
);
```

Вы должны заметить, что кроме подтверждения в консоли Firebug, о том, что событие произошло, вы получаете фактическую ссылку на событие, щелкнув на которой можно получить очень важные сведения о месте на экране, где произошло событие, и многое другое.

Как оказывается, функция `dojo.connect`, как и `dojo.byId`, проверяет многие вещи, для того чтобы работа с ней была проще, чем это могло сначала показаться. Фактически любые аргументы, которые допускают значение `null`, вообще могут быть опущены. Вследствие этого предыдущий фрагмент можно переписать, как показано ниже, сделав его немного более удобочитаемым:

```
var handle = dojo.connect(
    dojo.byId("s1"), //контекст
    "onmouseover",   //событие
    function(evt) {console.log("mouseover event",evt);} //обработчик события
);
```

Операция отключения функции, предназначенной для обработки события DOM, выполняется еще проще, и она имеет большое значение для предотвращения утечек памяти, если в странице приходится делать большое число подключений и отключений. Для этого достаточно вызвать функцию `dojo.disconnect`, передав ей дескриптор соединения, который был сохранен ранее, а инструментарий Dojo сам позаботится обо всем остальном:

```
dojo.disconnect(handle);
```

Несмотря на всю простоту примера, функция `dojo.connect` демонстрирует ключевой принцип философии Dojo: перемещение из пункта A в пункт B всегда должно быть максимально простым. Уверен, если вы близко знакомы с языком JavaScript, вы могли бы и сами пройти все этапы установки, поддержания и отсоединения обработчика события. Однако при этом вам пришлось бы использовать шаблон, загромождающий программный код, и давайте не будем забывать: каждую строку программного кода, которую вы напишете, вам же и придется сопровождать. Опытные веб-разработчики, которых немало и среди нас, предпочитающие не усложнять задачу, оценят преимущества использования функций `dojo.connect` и `dojo.disconnect`.

Инструментальный набор Dojo не делает ничего такого, что нельзя было бы сделать на чистом JavaScript, то же самое можно сказать и о любом другом наборе инструментальных средств для JavaScript. Главная ценность Dojo состоит в том, что он устраняет несовместимость между

броузерами и делает выполнение наиболее типичных операций настолько простым, насколько это возможно, защищая вас от необходимости писать и сопровождать шаблонный программный код, что позволяет максимально повысить вашу производительность труда.

Другой интересной особенностью, которая демонстрирует огромную мощь, заключенную в маленьком пакете, является функция `dojo.query` – механизм, позволяющий быстро выполнять запросы к странице с использованием синтаксиса CSS3.

Более подробно функция `dojo.query` будет рассматриваться в главе 5, и там же приводится больше сведений о селекторах CSS3. При желании вы можете перейти к этой главе и бегло ознакомиться с ней.

Например, поиск всех элементов `DIV` в странице выполняется очень просто, как показано ниже:

```
dojo.query("div") //отыскивает все элементы div в дереве DOM
```

Если вы попробуете выполнить эту инструкцию в консоли Firebug, то вы увидите, что она действительно возвращает список элементов `DIV`. Поиск конкретного элемента `DIV` с определенным именем также выполняется очень просто:

```
dojo.query("div#d2") //проверяет наличие элемента div с атрибутом id=d2
```

А вот так выполняется поиск элементов определенного класса:

```
dojo.query(".blue") //возвращает список элементов с классом blue.
```

Заговорив о классах, следует заметить, что существует возможность отбирать элементы определенного типа, но, так как в нашем документе класс CSS применяется только к одному элементу `DIV`, нам необходимо применить класс `blue` еще к какому-нибудь элементу. Но, вместо того чтобы редактировать страницу HTML, почему бы не воспользоваться еще одной встроенной функцией из библиотеки Base – `dojo.addClass`, чтобы применить класс к выбранному элементу, как показано ниже:

```
dojo.addClass("s1", "blue"); //добавляет класс blue в элемент SPAN
```

После применения класса `blue` к элементу `s1`, можно продемонстрировать другой вариант запроса с помощью функции `dojo.query`:

```
dojo.query("span.blue") //возвращает только элементы span с классом blue
```

Уловили суть? Уверен, вы *смогли* бы реализовать все это с использованием своих собственных обходных приемов, но разве не проще знать, что инструментальный набор изолирует вас от всего этой чехарды и вместо этого предоставляет единственную и простую в использовании функцию?

## Исследование Dijit

Мы могли бы идти вперед и вперед по пути демонстрации простого в использовании прикладного интерфейса библиотеки Base, но давайте оставим это для последующих глав и рассмотрим короткий пример, показывающий как легко добавить в страницу некоторые из диджитов, не написав ни строчки программного кода.

Предположим, что у нас имеется страница, как показано в примере 1.4.

*Пример 1.4. Пример примитивной формы*

```
<html>
    <head>
        <title>Fun with Dijit!</title>
    </head>
    <body>
        Just Use the form below to sign-up for our great offers:<br /><br />
        <form id="registration_form">
            First Name: <input type="text" maxlength=25 name="first"/><br />
            Last Name: <input type="text" maxlength=25 name="last"/><br />
            Your Email: <input type="text" maxlength=25 name="email"/>
            <br /><br />
            <button onclick="alert('Boo!')">Sign Up!</button>
        </form>
    </body>
</html>
```

На рис. 1.5. показано, как выглядит эта страница, хотя представить ее совсем не сложно.

Возможно, в 92 году такая форма вполне сгодилась бы, но она совершенно неприемлема для наших дней. Остановимся на минутку и посмотрим, что вы обычно делаете в таких случаях: определяете несколько классов, применяете классы, пишете программный код JavaScript с целью обеспечить проверку правильности вводимых данных и т. д.

Чтобы дать вам почувствовать, как могла бы выглядеть страница после внедрения в нее возможностей инструментария Dojo, взгляните на пример 1.5. Не надо сейчас пытаться вникать во все мелочи – впереди нас ждет еще множество страниц, при исследовании которых мы будем за-

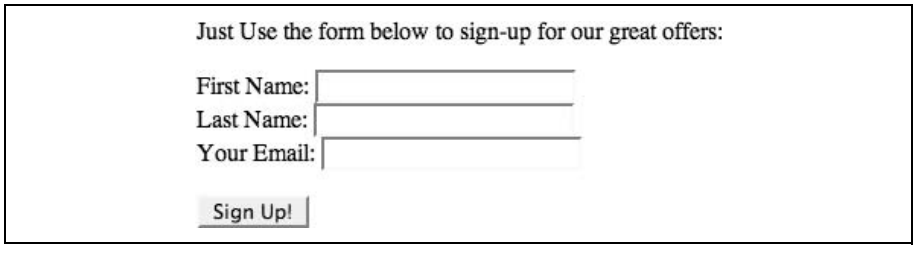



***Рис. 1.5.*** *Хоть и функциональная, но очень некрасивая форма*

глядывать во все углы и закоулки. А пока просто ознакомьтесь с общей структурой страницы и с некоторыми диджитами, внедренными в нее.

*Пример 1.5. Форма, которая (благодаря Dojo) перестала быть примитивной*

```
<html>
  <head>
    <title>Fun with Dijit!</title>

    <!-- Некоторые стили взяты из встроенной темы tundra, которая
    предлагается инструментальным набором Dojo для стилизации страниц,
    что дает вам возможность использовать профессиональный стиль без
    приложения дополнительных усилий. -->
    <link rel="stylesheet" type="text/css"
      href="http://o.aolcdn.com/dojo/1.1/dijit/themes/tundra/tundra.css" />
    <link rel="stylesheet" type="text/css"
      href="http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css" />
    <!-- Дополнительные стили CSS для выравнивания элементов формы -->

    <style type="text/css">
      h3 {
          margin : 10px;
      }
      label,input {
          display: block;
          float: left;
          margin-bottom: 5px;
      }
      label {
          text-align: right;
          width: 70px;
          padding-right: 20px;
      }
      br {
          clear: left;
      }
      .grouping {
          width:300px;
          border:solid 1px rgb(230,230,230);
          padding:5px;
          margin:10px;
      }
    </style>

    <!-- Загрузить Base и указать, что после загрузки страницы требуется
    выполнить парсинг диджитов -->
    <script
        type="text/javascript"
        src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js"
        djConfig="parseOnLoad: true" >
    </script>

    <!-- Загрузить некоторые диджиты с помощью dojo.require так же, как
    если бы вы подключали с помощью директивы #include файлы в языке C
```

```
    или выполняли импортирование модулей в языке Java -->
    <script type="text/javascript">
        dojo.require("dojo.parser");
        dojo.require("dijit.form.TextBox");
        dojo.require("dijit.form.ValidationTextBox");
        dojo.require("dijit.form.Button");
    </script>
  </head>

  <!-- Указать, что встроенная тема tundra должна применяться ко всему,
  что находится в теле страницы. (Библиотека Dijit в значительной степени
  опирается на CSS, поэтому включение соответствующей темы имеет очень важное
  значение.)-->
  <body class="tundra">

    <h3>Sign-up for our great offers:</h3>

    <form id="registration_form">

      <!-- Мы внедрили несколько виджетов в страницу, вставив теги и включив
      в них атрибут dojoType, чтобы парсер мог обнаружить и загрузить их -->
      <div class="grouping">
        <label>First Name:</label>
        <input type="text"
          maxlength=25
          name="first"
          dojoType="dijit.form.TextBox"
          trim="true"
          propercase="true"/><br>

        <label>Last Name:</label>
        <input type="text"
          maxlength=25
          name="last"
          dojoType="dijit.form.TextBox"
          trim="true"
          propercase="true"/><br>

        <label>Your Email:</label>
        <input type="text"
          maxlength=25
          name="email"
          dojoType="dijit.form.ValidationTextBox"
          trim="true"
          lowercase="true"
          regExp="[a-z0-9._%+-]+@[a-z0-9-]+\.[a-z]{2,4}"
          required="true"
          invalidMessage="Please enter a valid e-mail address"/><br>

        <button dojoType="dijit.form.Button"
          onClick="alert('Boo!')">Sign Up!</button>
      </div>
    </form>
  </body>
</html>
```

И вуаля! На рис. 1.6. показано, как теперь выглядит форма, дополненная возможностями проверки введенных данных.

***Рис. 1.6.*** *Форма, имеющая более приятный внешний вид благодаря применению диджитов*

Если вас заинтриговали примеры в этой главе и вы готовы продолжить изучение Dojo, то вы на верном пути. В следующих главах последовательно рассматриваются специфические особенности инструментального набора. Но, прежде чем двинуться дальше, потратим немного времени, чтобы вспомнить все, о чем рассказывалось в этой главе (то же самое мы будем делать в каждой главе).

# В заключение

В этой главе мы только начали наше путешествие в мир Dojo, но при этом нам удалось охватить множество основных идей. После прочтения этой главы вы должны:

- Знать, куда идти, чтобы загрузить инструментальный набор, и как настроить среду разработки
- Понимать архитектуру инструментального набора и ключевые отличия каждого компонента
- Понимать некоторые распространенные термины, используемые при разработке на основе Dojo
- Понимать преимущества использования Firebug в процессе разработки
- Иметь представление о том, как инструментальный набор загружает и настраивает сам себя
- Иметь представление о том, насколько просто использовать библиотеку Base и включать в страницу стандартные диджиты
- Представлять, как выглядит программный код, использующий Dojo
- Быть в нетерпении перейти к следующим главам и узнать еще больше о Dojo

В следующей главе будут обсуждаться утилиты броузера.

# 2

# Утилиты броузера

В этой главе приводится формальное описание вспомогательных функций, которые можно найти в Base. Эти функции были разработаны с целью упростить решение проблем, типичных для языка JavaScript, связанных с несовместимостью броузеров, и поэтому они отличаются высокой переносимостью и были сильно оптимизированы. Независимо от того, будете ли вы использовать что-нибудь еще из инструментального набора, конструкции, представленные в этой главе, заслуживают пристального внимания, потому что они обеспечивают такие удобства, без которых сложно будет обойтись, после того как вы попробуете использовать их. Среди тем, рассматриваемых в этой главе, – манипулирование массивами, клонирование узлов, добавление и удаление классов, а также вычисление размеров отступов и вмещающих прямоугольников для узлов DOM.

## Поиск узлов DOM

В предыдущей главе была представлена функция `dojo.byId` – специализированный механизм поиска узлов DOM более переносимым и предсказуемым способом, чем метод `document.getElementById`. Функция `dojo.byId` является самой часто используемой функцией при разработке с использованием Dojo, поэтому есть смысл еще раз вернуться к дискуссии из главы 1, где были обозначены некоторые проблемы функции `document.getElementById` и говорилось о том, как `dojo.byId` помогает их решать. В качестве напоминания ниже приводится полная сигнатура функции `dojo.byId`:

```
dojo.byId(/*String*/ id | /*DomNode*/ node, /*DomNode*/doc) // Возвращает
                                                            // узел DOM
```

В примере 2.1 приводится ряд других типичных случаев использования этой функции.

*Пример 2.1. Краткое напоминание о функции dojo.byId*

```
var foo = dojo.byId("foo"); //вернет узел с id=foo, если он существует
dojo.byId(foo).innerHTML="bar"; //поиск не выполняется, так как foo -
                                //это узел; затем в свойство innerHTML
                                //записывается значение "bar"
var bar = dojo.byId("bar", baz); //вернет узел с id=bar в документе, на
                                 //который ссылается baz, если он существует
```

# Определение типа

В языках программирования с динамической типизаций, таких как JavaScript, очень часто бывает необходимо (и даже желательно) определить тип переменной, прежде чем выполнять какие-либо операции над ней. На первый взгляд определение типа не должно быть сложной процедурой, но это не всегда так и на практике может приводить к неприятностям и ошибкам из-за трудноуловимых отличий. В библиотеке Base имеется несколько удобных функций, упрощающих эту операцию. Определение типа в разных броузерах из-за их особенностей выполняется немного по-разному, как и решение других проблем, рассматривавшихся до сих пор. Ниже приводится список этих функций:

`isString(/*Any*/ value)`

Возвращает `true`, если значение имеет тип `String`.

`isArray(/*Any*/ value)`

Возвращает `true`, если значение имеет тип `Array`.

`isFunction(/*Any*/ value)`

Возвращает `true`, если значение имеет тип `Function`.

`isObject(/*Any*/ value)`

Возвращает `true`, если значение имеет тип `Object` (включая `Array` и `Function`) или `null`.

`isArrayLike(/*Any*/ value)`

Возвращает `true`, если значение имеет тип `Array`, но при этом относит к массивам некоторые другие типы. Например, встроенное свойство `arguments`, к которому можно обращаться внутри объекта Function, принадлежит одному из таких типов. Этот тип не поддерживает встроенные методы, такие как `push`, однако он напоминает массив в том смысле, что представляет список значений, к которым можно обращаться через индексы.

`isAlien(/*Any*/ value)`

Возвращает `true`, если значение является встроенной функцией или «родной» функцией таких компонентов, как ActiveX, но не опознается таким методом, как применение функции `instanceof`.

## Грубое определение типа

Концепция *грубого определения типа*, присущая языкам программирования с динамической типизацией, таким как Python и JavaScript, лежит в основе большинства только что представленных функций. Суть грубого определения типа можно выразить словами: «если оно ходит как утка и крякает как утка, – значит, это утка». Это означает, что если рассматриваемый экземпляр данных обладает минимально необходимым набором свойств, чтобы быть отнесенным к определенному типу, это является достаточным основанием, чтобы предположить, что он принадлежит именно к этому типу.

Например, функция `isArrayLike` квалифицирует встроенное свойство `arguments` как массив. В языках программирования с динамической типизацией, которые не требуют объявления принадлежности каждой переменной определенному типу данных (динамическое связывание), грубое определение типа представляет собой важный механизм проверки типа объекта, когда в этом возникает необходимость.

Например, если передать оператору `typeof` обычный массив, такой как `[]`, он вернет тип `object`, а функция `isArray` из ядра Base выполнит грубое определение типа и вернет для такого массива значение `true`.

Грубое определение типа является фундаментальной концепцией в JavaScript и во многих инструментальных наборах, поэтому эти пояснения более тесно связаны с решением повседневных задач, чем может показаться на первый взгляд.

Суть состоит в том, что функции определения типа из библиотеки Base могут сэкономить ваше время и уберечь вас от столкновения с несовместимостями броузеров, поэтому пользуйтесь ими и пользуйтесь ими чаще.

# Утилиты для работы со строками

Усечение начальных и завершающих пробельных символов в строках является чрезвычайно часто используемой операцией. В следующий раз, когда вам потребуется выполнить эту операцию, вместо того, чтобы создавать свою функцию, воспользуйтесь функцией `trim` из библиотеки Base.

С большинством подобных тривиальных вспомогательных функций могут быть связаны трудноуловимые проблемы обеспечения высокой производительности, а использование инструментального набора предоставляет в ваше распоряжение коллективный опыт целого сообщества, которое уделяет особое внимание подобным проблемам.

Ниже приводится пример использования функции `trim`:

```
var s = "   this is a value with whitespace padding each side   ";
s = dojo.trim(s); //"this is a value with whitespace padding each side"
```

В состав модуля `string` библиотеки Core входят еще несколько функций для работы со строками. Каждый из следующих примеров предполагает, что загрузка модуля `dojo.string` уже была выполнена с помощью инструкции `dojo.require`.

`dojo.string.pad`

Дополняет строку, гарантируя, что ее длина будет точно соответствовать указанному числу символов. По умолчанию дополнение производится слева. С помощью дополнительного параметра можно организовать дополнение справа:

```
dojo.string.pad("", 5);             // "00000"
dojo.string.pad("", 5, " ");        // "     "
dojo.string.pad("0", 5, "1");       // "11110"
dojo.string.pad("0", 5, "1", true); // "01111"
```

`dojo.string.substitute`

Выполняет замену параметров в строке и при необходимости позволяет указывать функцию преобразования и/или другой объект, являющийся источником информации:

```
//Вернет "Jack and Jill went up a hill."
dojo.string.substitute("${0} and ${1} went up a hill.", ["Jack", "Jill"]);
//Вернет "Jack and Jill went up a hill."
dojo.string.substitute("${person1} and ${person2} went up a hill.",
{person1 :
"Jack", person2: "Jill"});
//Вернет "*Jack* and *Jill* went up a hill."
dojo.string.substitute("${0} and ${1} went up a hill.", ["Jack", "Jill"],
function(x) {
    return "*"+x+"*";
});
```

`dojo.string.trim`

За счет несколько большего размера по сравнению с реализацией одноименной функции в библиотеке Base, функция из модуля `string` библиотеки Core обеспечивает немного более высокую производительность и может использоваться в случаях, когда быстродействие имеет большое значение:

```
dojo.string.trim( /* строковое значение */);
```

# Обработка массивов

Массивы являются одной из наиболее фундаментальных структур данных в любом императивном языке программирования, включая и JavaScript. Однако, к сожалению, стандартные функции для работы с массивами поддерживаются не всеми господствующими броузерами,

и, пока такое положение дел имеет место, очень полезно иметь инструментальный набор, защищающий вас от оголенного металла броузеров. В этом смысле, даже если бы следующие версии всех основных броузеров поддерживали массивы абсолютно идентичным образом, по-прежнему остаются люди, использующие старые версии броузеров, поэтому пока рано думать о возвращении к стандартным функциям, реализованным в броузерах.

Вам будет интересно узнать, что различные языковые инструменты, оптимизированные по скорости выполнения, представляют собой обертки вокруг «родного» типа `Array` везде, где это возможно, и имитируют необходимую функциональность, отсутствующую в таких броузерах, как IE.

К счастью, инструментальный набор Dojo стремится не отставать от возможностей богатой реализации типа `Array`, заключенной в броузерах Mozilla (*http://developer.mozilla.org/en/docs/Core_JavaScript_1.5_Reference*). Пока инструментальный набор будет с вами, ничто не застанет вас врасплох. А если вы уже забыли нашу дискуссию о функции `dojo.byId` в главе 1, напомню, что вы не должны рассчитывать на многое в текущей экосистеме броузеров. Следующий раздел снова должен наполнить вас энтузиазмом и, может быть, даже немного удивить.

## Поиск местоположения элементов

Две самые распространенные операции при работе с массивами включают в себя определение индекса элемента, причем обе они, по сути, проверяют наличие элемента в массиве. Библиотека Base упрощает поиск двумя функциями с говорящими именами: `dojo.indexOf` и `dojo.lastIndexOf`. Каждая из этих функций возвращает целое число, представляющее собой индекс элемента в массиве, если он существует. Если заданный элемент в массиве отсутствует, функции возвращают значение `-1`. В сигнатурах этих функций имеется дополнительный параметр, определяющий начальный или конечный индекс, откуда следует начинать поиск. Обе функции имеют одинаковые сигнатуры:

```
dojo.indexOf(/*Array*/ array, /*Any*/ value, /*Integer?*/ fromIndex)
//возвращает Integer
dojo.lastIndexOf(/*Array*/ array, /*Any*/ value, /*Integer?*/ fromIndex)
//возвращает Integer
```

Если вы исходно выполняли разработку в броузере Firefox, вас может удивить тот факт, что в броузерах IE6 и IE7 «родная» реализация объектов `Array` не поддерживает даже метод `indexOf`. К сожалению, такого рода семантические недоразумения в том, что кажется совершенно очевидным, относятся к одной из самых сложных в обнаружении разновидности ошибок.

Следующий фрагмент программного кода иллюстрирует некоторые основные приемы использования этих методов:

```
var foo = [1,2,3];
var bar = [4,5,6,5,6];
var baz = [1,2,3];

dojo.indexOf([foo, bar], baz); // -1
dojo.indexOf(foo, 3);          // 2
dojo.indexOf(bar, 6, 2);       // 2
dojo.indexOf(bar, 6, 3);       // 4

dojo.lastIndexOf(bar, 6);      // 4
```

Кроме того, следует отметить, что эти методы выполняют поверхностное сравнение. Для таких сложных структур данных, как `Array`, это означает, что сравниваются ссылки на объекты. Следующий фрагмент поясняет это утверждение на конкретном примере:

```
bop = [4,5,6,5,6, foo]; // bop содержит вложенный массив
dojo.indexOf(bop, foo); //5, потому что (ссылка на) foo содержится в bop
dojo.indexOf(bop, [1,2,3]); //-1, потому что foo - это не тот же самый объект
                            //что и [1,2,3]
```

## Проверка элементов на соответствие условию

Очень часто бывает необходимо определить, соответствуют ли все элементы массива некоторому условию или существуют ли в массиве элементы, соответствующие некоторому условию. Для выполнения такого рода проверок библиотека Base предоставляет функции `every` и `some`. Входными параметрами этих функций являются: массив, функция, с помощью которой будут проверяться элементы массива, и дополнительный параметр, который можно использовать для передачи контекста (`this`) функции:

```
dojo.every([2,4,6], function (x) { return x % 2 == 0 });   //true
dojo.every([2,4,6,7], function (x) { return x % 2 == 0 }); //false
dojo.some([3,5,7], function f(x) { return x % 2 == 0 });   //false
dojo.some([3,5,7,8], function f(x) { return x % 2 == 0 }); //true
```

## Обход элементов

Функция `forEach` передает каждый элемент массива функции, принимающей до трех параметров и не возвращающей никакого значения. Первый параметр – текущий элемент массива, обход которого выполняется, второй параметр (необязательный) – индекс текущего элемента массива и третий параметр – сам массив. Вообще функция `forEach` используется для обхода всех элементов массива, как если бы это был обычный цикл `for`. Ниже приводится сигнатура этой функции:

```
dojo.forEach(/*Array*/ array, /*Function*/ function) // Не возвращает ничего
```

В простейшем случае функция forEach применяется, как показано ниже:

```
dojo.forEach([1,2,3], function(x) {
    console.log(x);
});
```

Функция forEach обладает следующими очевидными преимуществами: она меньше загромождает программный код, чем цикл for, для работы которого необходимо еще предусмотреть переменную цикла, и, кроме того, позволяет использовать литералы массивов. Однако самое важное преимущество состоит в том, что на основе функции, определяемой вторым параметром, она позволяет реализовать замыкания, защищая тем самым свой контекст от влияния переменной цикла и любых других переменных, которые могут использоваться в теле цикла. Как и другие вспомогательные функции, forEach способна принимать дополнительный параметр, определяющий контекст для встроенной функции.

Чтобы продемонстрировать, как функция forEach может обезопасить вас от неожиданностей, рассмотрим следующий фрагмент программного кода:

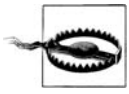
```
var nodes = getSomeNodes();

for(var x=0; x<nodes.length; x++){
    nodes[x].onclick = function(){
        console.debug("clicked:", x);
    }
}
```

Как вы думаете, какое значение переменной x будет здесь получено? Поскольку замыкание выполняется по *лексической переменной* x, а не по ее значению, во все вызовы функции будет передано *последнее значение*. Функция forEach предохраняет нас от такого поведения, создавая новую лексическую область видимости. Следующая версия программного кода демонстрирует, как выполнить обход массива и вывести ожидаемое значение:

```
var nodes = getSomeNodes();
var idx = 0;
dojo.forEach(nodes, function(node, idx){
    node.onclick = function(){
        console.debug("clicked:", idx);
    }
});
```

## Преобразование элементов

Функции map и filter имеют точно такую же сигнатуру, что и forEach, но они принципиально отличаются от последней тем, что, применяя некоторую логику к каждому элементу массива, они возвращают другой массив, оставляя оригинальный массив без изменений.



С технической точки зрения вы можете вносить изменения непосредственно в оригинальный массив, используя собственные версии функций `map` и `filter`. Однако обычно ожидается, что реализации `map` и `filter` будут свободны от каких-либо побочных эффектов. Говоря другими словами, введение побочных эффектов должно производиться с большой осторожностью и сопровождаться недвусмысленными комментариями, явно свидетельствующими об этом.

Программисты, использующие функциональные языки программирования (или даже языки программирования с функциональными расширениями, такие как Python), хорошо знают, что функции `map` и `filter` очень быстро приобретают высокую значимость, потому что они обеспечивают очень широкие возможности при таком кратком синтаксисе.

Тем, кто не сталкивался с функцией `map` прежде, она может показаться загадочной. В действительности же ее имя говорит само за себя, потому что она создает отображение (mapping) массива, получаемое с помощью функции преобразования. Следующий пример иллюстрирует применение функции `map`:

```
var z = dojo.map([2,3,4], function(x) {
    return x + 1
}); //вернет [3,4,5]
```

Для сравнения ниже приводится версия, которая создает аналогичный массив `z`, но без использования функции `map`:

```
var a = [2,3,4];
var z = [];
for (var i=0; i < a.length; i++) {
    z.push(a[i] +1);
}
```

Одно из преимуществ использования функции `map`, как и функции `forEach`, заключается в более высокой выразительности программного кода, что в свою очередь упрощает его сопровождение. Кроме того, вы получаете те же преимущества от создания замыканий с помощью анонимных функций, когда использование переменных, полученных в результате промежуточных вычислений, при отказе от замыканий могло бы привести к загромождению контекста.

Имя функции `filter` тоже говорит само за себя, так как эта функция фильтрует массив согласно некоторому функциональному критерию. Ниже показано, как она используется:

```
dojo.filter([2,3,4], function(x) {
    return x % 2 == 0
}); //вернет [2,4]
```

Реализация эквивалентного фрагмента программного кода не сложна, но требует большего внимания и дополнительных инструкций и, как следствие, увеличивает вероятность допустить опечатку или ошибку:

```
var a = [2,3,4];
var z = [];
for (var i=0; i < a.length; i++) {
    if (a[i] % 2 == 0)
        z.push(a[i]);
}
```

Как и в случае с другими функциями для работы с массивами, присутствующими в библиотеке Base, вы можете передавать функциям `map` и `filter` дополнительный параметр, определяющий контекст, если в этом есть необходимость:

```
function someContext() { this.y = 2; }
var context = new someContext;
dojo.filter([2,3,4], function(x) {return x % this.y==0}, context);
//вернет [2,4]
```

## Аргументы «функции в виде строк»

Библиотека Base предоставляет возможность передавать функциям `forEach`, `map`, `filter`, `every` и `some` сокращенные аргументы «функции в виде строк». Вообще такой подход выглядит проще, чем создание функций-оберток, и особенно удобен в действительно простых случаях, когда выполняется быстрое преобразование. Суть заключается в том, что вместо целой функции вы передаете в виде строки только тело функции. Три ключевых слова имеют специальное значение при использовании в таких строках:

`item`

Представляет ссылку на текущий обрабатываемый элемент.

`array`

Представляет ссылку на весь обрабатываемый массив.

`index`

Представляет ссылку на индекс текущего обрабатываемого элемента.

Взгляните на следующий пример, где для достижения одного и того же результата используются два разных подхода:

```
var a = new Array(1,2,3,...);

//Вариант, требующий много ввода для достижения простой цели
a.forEach(function(x) {console.log(x);}); //первый подход

//Вариант, требующий меньшего объема ввода,
//поэтому вы выполните свою работу быстрее
a.forEach("console.log(item)"); //второй подход
```

Использование подхода, основанного на применении «функций в виде строк» к методам для работы с массивами, может сделать ваш программный код более кратким, но он может осложнять поиск ошибок, поэтому не следует им злоупотреблять. Напри-

мер, взгляните на следующую версию предыдущего фрагмента программного кода:

```
var a = new Array(1,2,3,...);
a.forEach("console.log(items)"); //упс... лишний символ "s"
                                 //в имени items
```

Поскольку здесь закрался один лишний символ «s» в специальном термине `item`, он больше не будет действовать как итератор, превращаясь для функции `forEach` в пустую операцию. Такая опечатка может привести к значительным затратам времени на отладку, если, конечно, вы не обладаете острым глазом на подобные ошибки.

# Управление исходным программным кодом с помощью модулей

Если вы имеете опыт программирования, вы должны знать о концепциях группировки взаимосвязанных фрагментов программного кода в блоки, которые можно называть библиотеками, пакетами или модулями, и подключения этих ресурсов по мере необходимости, с помощью таких механизмов, как инструкция `import` или директива препроцессора `#include`. В Dojo те же самые понятия реализованы посредством инструкций `dojo.provide` и `dojo.require` соответственно.

В терминах Dojo фрагменты программного кода многократного использования называются *ресурсами*, а коллекции взаимосвязанных ресурсов называются *модулями*. В библиотеке Base имеются две невероятно простые конструкции, позволяющие импортировать модули и ресурсы: `dojo.require` и `dojo.provide`. Проще говоря, вы должны включить инструкцию `dojo.provide` в качестве первой строки в файл, который должен быть доступен для подключения к странице инструкцией `dojo.require`. Как оказывается, инструкция `dojo.require` — это не просто метка-заполнитель, как тег `SCRIPT`; она заботится о том, чтобы отобразить имя модуля на конкретное местоположение в файловой системе, извлечь программный код модуля и кэшировать модули и ресурсы, которые ранее были затребованы инструкцией `dojo.require`. Учитывая, что каждая инструкция `dojo.require` влечет за собой, по крайней мере, одно обращение к серверу, если затребованный ресурс еще не был получен, кэширование может существенно повысить производительность; даже кэширование ресурса, который загружается один раз и становится доступен в виде локальной копии, может принести существенную выгоду при следующих запусках приложения.

## Зачем управлять беспорядком

Для любых проектов, кроме самых маленьких, преимущества описываемого подхода не вызывают сомнений. Простота в обслуживании и легкость расширения или встраивания программного кода в разные

страницы являются ключевыми аспектами, которые позволят быстро и эффективно выполнять работу. Разумеется, преимущества импортирования программного кода описываемым способом не всегда были очевидны для веб-разработчиков, и из-за неподобающей организации управления исходным программным кодом многие веб-проекты превратились в сложные для сопровождения чудовища. Например, обычно, когда необходимо взять файл JavaScript, находящийся в постоянном каталоге на веб-сервере, и вставить его в страницу, используется примерно такой тег SCRIPT:

```
<script src="/static/someScript.js" type="text/javascript"></script>
```

Хорошо, возможно, нет ничего страшного, когда в странице присутствует один или два тега SCRIPT, но что, если имеется много страниц, которые пользуются теми же инструментами, требующими загрузки тех же самых сценариев? Тогда, вероятно, вам придется включить те же самые теги SCRIPT во все эти страниц. В конечном счете, у вас может оказаться значительное число таких сценариев, и, когда вам потребуется вручную отследить их все, эта задача может оказаться непосильной. Конечно, в прошлом, когда для работы страницы требовалось всего несколько сотен строк программного кода JavaScript, вам не требовались мощные механизмы управления ресурсами, но современные приложения могут подключать десятки тысяч строк кода JavaScript. Как можно управлять всем этим, не имея хорошего инструмента, выполняющего отложенную загрузку и загрузку по требованию?

Кроме смягчения кошмара управления конфигурацией, который ждал бы вас в противном случае, абстракции dojo.provide и dojo.require также делают возможным применение инструментов сборки из библиотеки Util. Это позволяет, например, упаковывать несколько файлов (не забывайте, что для загрузки каждого из них требуется выполнить синхронный запрос) в один-единственный файл, что влечет за собой уменьшение временных задержек. Без правильных абстракций, явно определяющих зависимости, все функциональные особенности инструментов сборки были бы невозможны.

Еще одно преимущество четко определенной связи, как между dojo.provide и dojo.require, заключается в возможности управления родственными ресурсами путем объединения их в пространства имен, что позволяет минимизировать конфликты имен и упростить организацию и поддержку программного кода. Хотя *пространства имен* в Dojo – это всего лишь иерархии вложенных объектов, доступ к которым осуществляется с применением точечной нотации, тем не менее эти иерархии являются весьма эффективным средством организации пространств имен и хорошо выполняют эту задачу.

Организация ресурсов по пространствам имен является настолько базовой, что инструментальный набор Dojo снабжает библиотеку Base функцией dojo.setObject. Эта функция принимает два аргумента. Пер-

вый аргумент – это иерархия объектов, которая должна быть создана, а второй – значение, которое должно быть отображено в эту иерархию:

```
dojo.setObject(/* String */ object, /* Any */ value, /* Object */ context)
//возвращает Any
```

Пример 2.2 иллюстрирует применение этой функции.

*Пример 2.2. Организация пространств имен с помощью функции dojo.setObject*

```
var foo = {bar : {baz : {qux : 1}}}; //"длинное" объявление
                                     //вложенных объектов
console.log(foo.bar.baz.qux);        //выведет 1

//То же самое можно сделать короче, не беспокоясь о соблюдении
//парности скобок
dojo.setObject("foo.bar.baz.qux", 1); //более очевидный синтаксис
console.log(foo.bar.baz.qux);         //выведет 1

//Если в дополнительном аргументе передать функции контекст, иерархия
//объектов будет создана относительно указанного контекста, а не глобального,
//определяемого dojo.global
var someContext = {};
dojo.setObject("foo.bar.baz.qux", 23, someContext);
console.log(someContext.foo.bar.baz.qux); //выведет 23
```

Функция `dojo.setObject` – это не что иное, как синтаксический подсластитель, но в случае необходимости она может помочь ликвидировать нагромождения в программном коде, избавить от нудной работы по подсчету парных скобок и т. д.

Консорциум производителей OpenAjax Alliance (*http://www.openajax.org*), объединившихся для продвижения открытости и стандартов в области передовых веб-технологий, настоятельно рекомендует следовать практике применения точечной нотации для организации пространств имен.

## Пример создания модуля для применения в составе версии XDomain

Рассмотрим небольшой конкретный пример использования инструкций `dojo.require` и `dojo.provide`. Для начала рассмотрим простой модуль, реализующий такую тривиальную функцию, как функцию вычисления членов последовательности Фибоначчи. В примере 2.3 ресурс связан с модулем. И хотя группировка ресурсов в модули не является строго обязательной, тем не менее практически всегда это целесообразно. В этой книге вам часто будет встречаться имя `dtdg` (от «Dojo: The Definitive Guide[1]»), используемое в качестве общего пространства имен для модулей.

[1] Оригинальный заголовок этой книги. – *Прим. ред.*

## Последовательность чисел Фибоначчи

Числа Фибоначчи – это числовая последовательность[1], получившая свое название в честь математика 13 века Леонардо из Пизы, которая показывает некоторые интригующие свойства чисел. Числа Фибоначчи можно обнаружить в генераторах псевдослучайных чисел, в приемах оптимизации, в музыке, в природе, и они тесно связаны с золотым сечением.

Последовательность Фибоначчи определяется следующим образом:

```
fibonacci(0) = 0
fibonacci(x <= 1) = x
fibonacci(x > 1) = fibonacci(x-1) + fibonacci(x-2)
```

Подробнее об этой последовательности можно прочитать по адресу *http://en.wikipedia.org/wiki/Fibonacci_number*.[2]

---

[1] Имеется в виду целочисленная последовательность. – *Прим. ред.*

[2] На русском языке: *http://ru.wikipedia.org/wiki/Фибоначчи_числа*. – *Прим. перев.*

*Пример 2.3. Определение простого модуля (dtdg.foo)*

```
/*
  Инструкция dojo.provide указывает, что файл .js с исходным программным
  кодом образует модуль dtdg.foo. Семантически модуль dtdg.foo также
  образует пространство имен для функций, входящих в состав модуля. На диске
  этот файл мог бы иметь имя foo.js и находиться в каталоге dtdg.
*/
dojo.provide("dtdg.foo");

//Обратите внимание: функции объявляются относительно пространства имен модуля
dtdg.foo.fibonacci = function(x) {
    if (x < 0)
        throw Exception("Illegal argument");

    if (x <= 1)
        return x;

    return dtdg.foo.fibonacci(x-1) + dtdg.foo.fibonacci(x-2);
}
```

Практически всегда имеет смысл группировать свои ресурсы в логические модули и объединять их в пространства имен. Кроме того, что это рекомендуемый подход, он предотвращает от непреднамеренного переопределения символов в глобальном пространстве имен, а также предотвращает переопределение

введенных вами имен другими модулями. В конечном итоге, это один из основных поводов применения инструкций dojo.provide и dojo.require!

Предположим, когда-нибудь, при создании другой страницы, вы решили использовать свой модуль dtdg.foo, чтобы удивить юных математиков из начальной школы. Вместо того чтобы заново переписывать хорошо протестированную функцию и тем самым повышать вероятность появления ошибок, вы можете снова использовать свой модуль, подключив его с помощью dojo.require. В примере 2.4 показано, как вы можете использовать локальный модуль совместно с инструментальным набором, загружаемым из CDN. В этом примере предполагается, что этот файл HTML сохранен в каталоге, содержащем каталог dtdg, в котором находится модуль из примера 2.3.

*Пример 2.4. Использование локального модуля в процедуре самонастройки XDomain*

```
<html>
    <head>
        <title>Fun With Fibonacci!</title>

        <script
            type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js"
            djConfig="baseUrl:'./'">
        </script>

        <script type="text/javascript">
            dojo.registerModulePath("dtdg", "./dtdg");
            dojo.require("dtdg.foo");
            /* с этого момента dojo.require выполняется асинхронно,
            потому что используется Dojo в сборке Xdomain. Будет лучше,
            если все обращения к dtdg.foo будут происходить в addOnLoad */

            dojo.addOnLoad(function() {
                dojo.body().innerHTML = "guess what? fibonacci(5) = " +
                                        dtdg.foo.fibonacci(5);
            });
        </script>

    </head>
    <body>
    </body>
</html>
```

Этот пример наглядно демонстрирует, насколько просто с помощью dojo.require подключить ресурс к странице и затем использовать его. Кроме того, здесь имеется несколько интересных моментов, которые следует отметить особо.

В случае использования локальной копии инструментарий Dojo с точки зрения модулей выглядит как корневой каталог, но при использо-

вании версии XDomain «настоящий» корневой каталог инструментария находится где-то на одном из серверов компании AOL. Поэтому, чтобы определить начальную точку поиска локальных модулей, в данном случае — `dtdg.foo`, в массиве `djConfig` явно задается значение параметра `baseUrl`.

Массив `djConfig` — это всего лишь средство передачи определенных параметров настройки в инструментальный набор. Подробнее об этом будет говориться в следующем разделе, а пока просто примите это как должное.

Функция `dojo.registerModulePath` просто связывает пространство имен верхнего уровня (первый аргумент функции) с именем каталога относительно `baseUrl` (второй аргумент).

Не забывайте проявлять особую аккуратность при настройке путей к модулям с помощью `dojo.registerModulePath`. Если забыть, что относительные пути к каталогам определяются не относительно корневого каталога инструментария, а относительно каталога с файлом *dojo.js*, то легко ошибиться на один уровень в дереве каталогов. Кроме того, как известно, периодически вызывает проблемы присутствие завершающего символа слеша в пути к модулю, поэтому также необходимо побеспокоиться о том, чтобы этого символа не было.

Все, что определяется в вашем модуле, станет доступно для использования в результате вызова `dojo.require`. Например, если предположить, что модуль `dtdg.foo` содержит дополнительные функции или переменные, они будут доступны после выполнения инструкции `dojo.require("dtdg.foo")`. Как обычно, мы нигде не обращаемся к содержимому dtdg.foo за пределами блока `addOnLoad`.

Совершенно необязательно наличие полного соответствия между файлами с исходным программным кодом и функциями, которые предоставляются этими файлами посредством инструкции `dojo.provide`, но в общем случае это является вопросом стиля. Исключение составляют случаи, когда некоторые функции рассматриваются как частные и не должны быть доступны для общего использования.

Возможно, в предыдущем примере вы также заметили вызов функции `dojo.body()`. По сути, это упрощенный способ получить доступ к телу текущего документа в противоположность менее удобному методу `document.body`.

## Пример с числами Фибоначчи при использовании локальной версии инструментария

Для сравнения в примере 2.5 приводится тот же самый пример, но на этот раз используется локальная версия инструментария Dojo, а мо-

дуль dtdg находится в корневом каталоге инструментария рядом с каталогом *dojo*, содержащим библиотеку Core, поэтому нет необходимости изменять параметр настройки baseUrl или вызывать функцию registerModulePath. Это возможно потому, что Dojo автоматически пытается искать модули в каталоге с библиотекой Core, который является логичным и удобным местоположением для них.

*Пример 2.5. Использование модуля dtdg.foo совместно с локальной инсталляцией инструментария*

```
<html>
    <head>
        <title>Fun With Fibonacci!</title>

        <script
            type="text/javascript"
            src="your/relative/path/from/this/page/to/dojo/dojo.js" >
        </script>

        <script type="text/javascript">
            dojo.require("dtdg.foo");
            /* мы по привычке применяем блок addOnLoad, даже
               при использовании локальной версии */
            dojo.addOnLoad(function() {
                dojo.body().innerHTML = "guess what? fibonacci(5) = " +
                                        dtdg.foo.fibonacci(5);
            });
        </script>

    </head>
    <body>
    </body>
</html>
```

## Создание примера модуля волшебного джинна

Давайте создадим модуль для демонстрации некоторых концепций этой главы. Наша жизнь порой бывает настолько сложной, что было бы просто здорово иметь волшебного джинна, который смог бы давать нам ответы на любые вопросы. (Dojo может выполнять некоторые действия автоматически, что немного похоже на волшебство, но джинны – это *настоящее* волшебство.)

Прежде чем приступить к созданию модуля, вспомните, что всегда рекомендуется использовать пространства имен. Пример 2.6 помещает программный код ресурса *Genie* в пространство имен dtdg, которое мы уже использовали в этой книге. Если вы еще не создали локальный каталог с именем *dtdg*, то сделайте это прямо сейчас. Внутри этого каталога создайте файл с именем *Genie.js*, куда скопируйте волшебный программный код, представленный в примере 2.6.

*Пример 2.6. Реализация модуля волшебного джинна*

```
//всегда начинайте модуль с инструкции dojo.provide
dojo.provide("dtdg.Genie");

//создание пространства имен для джинна
dtdg.Genie = function() {}

//набор предсказаний чем-то напоминает 8 волшебных шаров
dtdg.Genie.prototype._predictions = [
        "As I see it, yes",
        "Ask again later",
        "Better not tell you now",
        "Cannot predict now",
        "Concentrate and ask again",
        "Don't count on it",
        "It is certain",
        "It is decidedly so",
        "Most likely",
        "My reply is no",
        "My sources say no",
        "Outlook good",
        "Outlook not so good",
        "Reply hazy, try again",
        "Signs point to yes",
        "Very doubtful",
        "Without a doubt",
        "Yes",
        "Yes - definitely",
        "You may rely on it"
];

//функция инициализации, конструирующая интерфейс
dtdg.Genie.prototype.initialize = function() {

    var label = document.createElement("p");
    label.innerHTML = "Ask a question. The genie knows the answer...";

    var question = document.createElement("input");
    question.size = 50;

    var button = document.createElement("button");
    button.innerHTML = "Ask!";
    button.onclick = function() {
        alert(dtdg.Genie.prototype._getPrediction());
        question.value = "";
    }

    var container = document.createElement("div");
    container.appendChild(label);
    container.appendChild(question);
    container.appendChild(button);

    dojo.body().appendChild(container);
}
```

```
//основная функция реализации взаимодействия
dtdg.Genie.prototype._getPrediction = function() {
    //получить псевдослучайное число в диапазоне от 0 до 19,
    //которое будет играть роль индекса предсказания
    var idx = Math.round(Math.random()*19)
    return this._predictions[idx];
}
```

Фактически этот пример не делает ничего, кроме как создает объект типа Function с именем `dtdg.Genie` и определяет одну «общедоступную» функцию `initialize`.

Согласно соглашениям, принятым в Dojo, если имя члена начинается с символа подчеркивания, он должен рассматриваться как частный член. Это общепринятое соглашение будет использоваться на протяжении всей книги. Очень важно следовать этому соглашению, потому что «частные» члены могут существенно изменяться от версии к версии.

Пример содержит комментарии, и, с точки зрения веб-разработчика, его логика должна быть достаточно проста для понимания. (Если это не так, то, прежде чем продолжить чтение, вам стоит ознакомиться с основами HTML и JavaScript с помощью какой-нибудь другой книги.)

Чтобы ввести джинна в действие, необходимо изменить базовый шаблон веб-страницы, как показано в примере 2.7.

*Пример 2.7. Веб-страница, предназначенная для общения с волшебным джинном*

```
<html>
    <head>
        <title>Fun With the Genie!</title>

        <script
            type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js">
            djConfig="modulePaths:{dtdg:'./dtdg',baseUrl:'./'}">
        </script>

        <script type="text/javascript">
            // загрузить модуль
            dojo.require("dtdg.Genie");

            // обеспечить безопасное обращение к dtdg.Genie внутри addOnLoad
            dojo.addOnLoad(function() {

                //создать экземпляр
                var g = new dtdg.Genie;

                //инициализировать, а все остальное он сделает сам
                g.initialize();
            });
        </script>
```

```
    </head>
    <body>
    </body>
</html>
```

Этот пример демонстрирует возможность многократного использования и переносимость модуля `dtdg.Genie`. Вам нужно просто загрузить модуль в странице, и после инициализации он будет «просто работать». (И пока пользователь не увидит исходный текст, он будет считать его по-настоящему волшебным.) Наиболее сложная часть сценария, требующая дополнительного пояснения, – это использование `djConfig` для определения параметров настройки Dojo перед загрузкой инструментария: параметр `modulePaths` определяет местоположение модуля относительно параметра `baseUrl`, значение которого соответствует текущему рабочему каталогу. Вследствие этого с точки зрения физического размещения структура файлов может иметь вид, как на рис. 2.1.

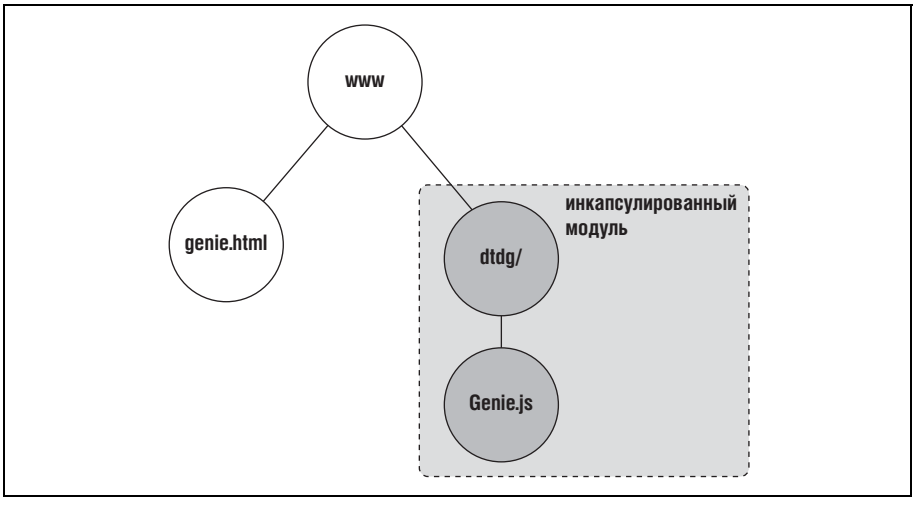



***Рис. 2.1.*** *Так модуль должен располагаться на диске*

# Утилиты для работы с объектами JavaScript

В библиотеке Base имеются три утилиты, упрощающие выполнение операций, которые обычно выполняются над объектами: `mixin`, `extend` и `clone`.

Функции `mixin` и `extend` используются инструментарием Dojo в реализации функции `dojo.declare`, которая представляет собой механизм имитации наследования классов. Более подробно функция `dojo.declare` будет рассматриваться в главе 10.

## Классы-смеси

Язык JavaScript позволяет создавать нечто, напоминающее классы, помещая совокупности свойств и методов внутрь функции-конструктора. После этого вы можете создавать экземпляры таких классов, вызывая функцию-конструктор с оператором `new`. Как можно предположить, иногда бывает очень удобно иметь возможность добавлять к объектам дополнительные свойства – либо в ответ на происходящие события, либо как часть хорошо продуманного дизайна, с целью максимально повысить возможность многократного использования программного кода. В любом из этих случаев функция `mixin` принимает на себя основные сложности реализации.

С точки зрения объектно-ориентированного дизайна функция `mixin` очень широко используется по всему инструментарию с целью максимально обеспечить многократное использование блоков программного кода.

Функция `mixin` принимает неопределенное число объектов, первый из которых служит объектом, к которому *примешиваются* остальные объекты:

```
dojo.mixin(/*Object*/ o, /*Object*/ o, ...) //Возвращает Object
```

Ниже приводится пример использования функции `mixin`:

```
function Man() {
    this.x = 10;
}

function Myth() {
    this.y = 20;
}

function Legend() {
    this.z = 30;
}

var theMan = new Man;
var theMyth = new Myth;
var theLegend = new Legend;

function ManMythLegend() {}
var theManTheMythTheLegend = new ManMythLegend;

//видоизменить объект theManTheMythTheLegend, смешав в нем три других объекта
dojo.mixin(theManTheMythTheLegend, theMan, theMyth, theLegend);
```

Обратите внимание, что все параметры функции `mixin` фактически являются экземплярами объектов, а не объявлениями функций.

## Расширение прототипов объектов

Функция extend из библиотеки Base работает точно так же, как функция mixin, но, в отличие от последней, она добавляет все свойства и методы смеси к *прототипу функции-конструктора*. Вследствие этого все последующие экземпляры, созданные с помощью такого конструктора, автоматически будут приобретать все эти новые свойства и методы:

```
dojo.extend(/*Function*/constructor, /*Object*/props, ... )
//Возвращает Function
```

Ниже приводится пример использования функции extend:

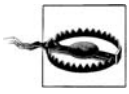
```
function Man() {
    this.x = 10;
}

function Myth() {
    this.y = 20;
}

function Legend() {
    this.z = 30;
}

var theMan = new Man;
var theMyth = new Myth;
var theLegend = new Legend;

function ManMythLegend() {}

var theManTheMythTheLegend = new ManMythLegend;

dojo.extend(ManMythLegend, theMan, theMyth, theLegend);

var theTheManTheMythTheLegend = new ManMythLegend;
```

Самое главное различие между функциями mixin и extend состоит в том, что функция mixin воспроизводит единственный экземпляр объекта, представляющий собой смесь из нескольких объектов, тогда как функция extend в действительности модифицирует свойство prototype функции.

Еще одна важная область применения функции extend — обеспечение более простой возможности создавать классы, чем обычно; когда все, что необходимо, реализуется через свойство prototype объекта. Использование функции extend таким способом — это, скорее, вопрос вкуса, хотя конечный результат при этом обычно получается более компактным. Ниже с целью демонстрации приводится переделанная версия нашего волшебного сценария из примера 2.6:

```
dojo.provide("dtdg.Genie");

//определить объект
dtdg.Genie = function() {}
```

```
//и расширить его
dojo.extend(dtdg.Genie, {
    _predictions : [
        "As I see it, yet",
        "Ask again later",
        "Better not tell you now",
        "Cannot predict now",
        "Concentrate and ask again",
        "Don't count on it",
        "It is certain",
        "It is decidedly so",
        "Most likely",
        "My reply is no",
        "My sources say no",
        "Outlook good",
        "Outlook not so good",
        "Reply hazy, try again",
        "Signs point to yes",
        "Very doubtful",
        "Without a doubt",
        "Yes",
        "Yes - definitely",
        "You may rely on it"
    ],

    initialize : function() {
        var label = document.createElement("p");
        label.innerHTML = "Ask a question. The genie knows the answer...";

        var question = document.createElement("input");
        question.size = 50;

        var button = document.createElement("button");
        button.innerHTML = "Ask!";
        button.onclick = function() {
            alert(dtdg.Genie.prototype._getPrediction());
            question.value = "";
        }

        var container = document.createElement("div");
        container.appendChild(label);
        container.appendChild(question);
        container.appendChild(button);

        dojo.body().appendChild(container);
    },

    getPrediction : function() {
        //получить псевдослучайное число в диапазоне от 0 до 19,
        //которое будет играть роль индекса предсказания
        var idx = Math.round(Math.random()*19)
        return this._predictions[idx];
    }
});
```



Не оставляйте по забывчивости запятую после последнего элемента типа `Object`, что является довольно распространенной ошибкой, особенно при проведении рефакторинга и копировании крупных блоков программного кода. Броузер Firefox простит вам такую оплошность, но в таком его поведении фактически больше вреда, чем пользы, потому что IE непременно разразится сообщением об ошибке.

## Клонирование объектов

Язык JavaScript предусматривает создание лишь поверхностных копий, поэтому, когда в операциях присваивания участвуют объекты JavaScript и узлы DOM, часто возникает необходимость в *клонировании*, или создании полных копий иерархий объектов. В библиотеке Base имеется функция `clone`, которая обеспечивает высокоэффективный способ копирования. Взгляните на следующий простой пример:

```
function foo() {
    this.bar = "baz";
}

var foo1 = new foo;
var foo2 = foo1; //shallow copy

console.log(foo1.bar);
console.log(foo2.bar);

foo1.bar = "qux";      //изменение foo1 приводит к изменению foo2

console.log(foo1.bar); // qux
console.log(foo2.bar); // qux

foo3 = new foo
foo4 = dojo.clone(foo3); //полная копия

foo3.bar = "qux";

console.log(foo3.bar); // qux
console.log(foo4.bar); // baz
```

# Манипулирование контекстом объекта

Глобальный объект `window` обеспечивает в веб-приложениях самый внешний уровень контекста, но иногда бывает необходимо изменить контекст по умолчанию на какой-то другой. Например, у вас может возникнуть потребность сохранять состояние сеанса, когда пользователь завершает работу с приложением. Или у вас, может быть, какая-то особенная среда выполнения, которая уже настроена под определенные обстоятельства. Тогда вместо того чтобы каждый раз заставлять программный код выполнять проверку всех условий для настройки среды, вы могли бы средствами библиотеки Base поменять один существующий контекст на другой.

Следующая функция позволяет изменять объект dojo.global и dojo.doc по своему усмотрению. Обратите внимание: по умолчанию dojo.doc — это простая ссылка на window.document, но несмотря на это, она обеспечивает единообразный способ идентификации контекста текущего документа, что опять же может быть весьма удобно в ситуациях, когда вопрос касается управления контекстами объектов. Функция dojo.body() — это сокращенный способ получить ссылку на тело документа.

Строго говоря, элемент body не определен явно, как часть документа XHTML и некоторых других документов, которые могут вам встретиться.

Вам необходимо знать о существовании как минимум трех функций из библиотеки Base, чтобы успешно управлять контекстом:

```
dojo.doc //Возвращает Document
dojo.body() //Возвращает DomNode
dojo.setContext(/*Object*/globalObject, /*Document*/globalDocument)
```

Наконец, для большей гибкости библиотека Base предоставляет две функции, позволяющие вызывать функцию либо в контексте другого окружения dojo.global, либо в контексте другого документа dojo.doc, отличных от текущих.

```
dojo.withGlobal(/*Object*/globalObject, /*Function*/callback,
                /*Object*/thisObject, /*Array*/callbackArgs)
dojo.withDoc(/*Object*/documentObject, /*Function*/callback,
             /*Object*/thisObject, /*Array*/callbackArgs)
```

Следует заметить, что применение функций Dojo для *активной* работы в другом объекте document или window еще не достаточно хорошо проверено, поэтому вы можете столкнуться с некоторыми проблемами, если пойдете этим путем. Стандартное использование предполагает загрузку инструментария Dojo в каждый документ, где он, как планируется, будет использоваться. Однако, в случае выполнения несложных операций обсуждаемые здесь функции управления контекстом работают замечательно.

## Передача параметров по частям

Функция partial из библиотеки Base позволяет передавать параметры функции частями по мере их готовности и производить фактический вызов функции позднее. Или можно определить значения всех параметров сразу и передать ссылку на функцию, которую можно будет вызвать позднее. Этот прием выглядит проще, чем передача функции вместе с параметрами, и широко используется по всему инструментальному набору. Ниже приводится сигнатура функции partial:

```
dojo.partial(*/Function|String*/func /*, arg1, ..., argN*/) //Возвращает Any
```

Для иллюстрации ниже приводится простой пример использования функции partial для передачи параметров функции, вычисляющей сумму последовательности чисел:

```
function addThree(x,y,z) { console.log(x+y+z);}

//передать два первых аргумента
f = dojo.partial(addThree, 100, 10);

//передать последний аргумент
f = dojo.partial(f, 1);

//а теперь вызвать функцию
f(); //111
```

## Каррингг[1]

Еще одна концепция, родственная функции partial, о которой следует знать, – это *карринг* (*currying*). Операция каррирования функции напоминает по своему действию функцию partial – в том смысле, что она возвращает функцию, частично подготовленную к вызову, которой позднее можно будет передать остальные аргументы, однако эти частично подготовленные функции являются самостоятельными первоклассными функциями и позволяют продолжать передавать им по одному аргументу за раз. Затем, как только каррированная функция получит все аргументы, она будет выполнена автоматически. Реализовать такое поведение оказывается гораздо сложнее, чем могло бы показаться на первый взгляд.

Если быть более точным, функция partial не полностью соответствует операции карринга. Функция partial позволяет передавать аргументы по одному за раз, но она никогда не вызовет функцию автоматически, даже если будут переданы все аргументы. Кроме того, функция partial должна явно участвовать в создании каждого экземпляра частично подготовленной функции вплоть до ее вызова, в ходе которого будет выполнена передача всех подготовленных параметров, то есть каждая промежуточная частично подготовленная функция не может действовать без явного использования самой функции partial.

Если вам интересно, поближе взглянуть на реализацию карринга можно в модуле dojox.lang.functional.curry. В приложении B вы найдете краткий обзор библиотеки DojoX.

---

[1] См. *http://ru.wikipedia.org/wiki/Карринг – Прим. перев.*

## Связывание объекта с определенным контекстом

Функция hitch из библиотеки Base очень напоминает функцию partial, – в том смысле, что тоже позволяет передавать параметры частями. Но у нее есть одна интересная особенность, которая заключается в том, что она позволяет связывать функцию с определенным контекстом выполнения, который будет установлен независимо от того, в каком контексте будет произведен вызов функции. Это может быть особенно удобно в ситуациях, когда имеется функция обратного вызова, и заранее никогда точно не известно, каков будет контекст (то есть значение ссылки this) при вызове функции. Сигнатура функции hitch имеет следующий вид:

```
dojo.hitch(/*Object*/scope, /*Function||String*/method /*, arg1, ... , argN*/)
//Возвращает Any
```

А для иллюстрации ниже приводится простой пример, где переопределяется контекст метода объекта:

```
var foo = {
    name : "Foo",
    greet : function() {
        console.log("Hi, I'm", this.name);
    }
}

var bar = {
    name : "Bar",
    greet : function() {
        console.log("Hi, I'm", this.name);
    }
}

foo.greet(); //Hi, I'm Foo
bar.greet(); //Hi, I'm Bar

/* Связать метод greet объекта bar с другим контекстом */
bar.greet = dojo.hitch(foo, "greet");

/ * Теперь объект bar будет выдавать себя за другой объект */
bar.greet(); // Hi, I'm Foo
```

Дополню: поскольку функция greet явно ссылается на контекст с помощью ссылки this, следующий фрагмент программного кода не переопределяет контекст метода greet:

```
bar.greet = foo.greet;
bar.greet();
```

Вам, вероятно, будет интересно узнать, что с точки зрения реализации функция hitch составляет основу функции partial, а вызов функции hitch со значением null в первом аргументе полностью эквивалентен вызову функции partial.

В главе 4 в разделе «Изменение контекста функций обратного вызова» приводится пример применения функции `hitch` для управления контекстом данных, используемых внутри асинхронной функции обратного вызова. Это один из наиболее часто встречающихся случаев, потому что функция обратного вызова имеет контекст `this`, отличный от содержащего ее объекта.

## Делегирование и наследование

*Делегирование* – это шаблон программирования, когда при выполнении действия один объект опирается на другой объект, а не предусматривает собственную реализацию этого действия. Делегирование – это самое сердце JavaScript как языка, основанного на прототипах, потому что делегирование – это именно тот прием, благодаря которому отыскиваются свойства объекта в *цепочке прототипов*. Несмотря на то, что делегирование составляет самую основу реализации наследования в JavaScript, которая опирается на поиск в цепочке прототипов во время выполнения, тем не менее делегирование по своей сути совершенно отличается от наследования в настоящих объектно-ориентированных языках программирования, таких как Java и C++, в которых часто (но далеко не всегда) разрешение имен в иерархии классов выполняется на этапе компиляции, а не во время выполнения. В этом смысле особенно примечательно, что, будучи особенностью времени выполнения, делегирование в обязательном порядке опирается на механизм *динамического связывания* как на особенность языка.

Функция `delegate` инструментального набора Dojo реализует механизм делегирования функции объекта и имеет следующую сигнатуру:

```
dojo.delegate(/*Object*/delegate, properties) //Возвращает Object
```

Основываясь на предыдущем примере, следующий отрывок демонстрирует, как можно с помощью делегирования получить объект, который делегирует вызов функции другому объекту:

```
function Foo() {
    this.talk = function() {console.log("Hello, my name is", this.name);}
}

// Получить объект типа Function, имеющий свойство name,
// но передает или делегирует обязательство вызова функции talk
// экземпляру Foo,  который передается функции delegate.
var bar = dojo.delegate(new Foo, {name : "Bar"});

// Вызывается метод делегата Foo
bar.talk();
```

В главе 10 рассказывается о механизме наследования, основанном на функции `dojo.declare` из инструментального набора, которая может использоваться для моделирования иерархий классов в JavaScript. Эта глава также обсуждает различные подходы в организации наследования.

# Утилиты для работы с деревом DOM

Не забывайте, что Dojo не пытается заменить основные функциональные возможности JavaScript; напротив, инструментальный набор дополняет JavaScript там, где это необходимо, чтобы позволить вам писать переносимый программный код и как можно реже использовать шаблоны. Поэтому вы не найдете прямых аналогов распространенных операций для работы с деревом DOM, таких как `appendChild`, `removeChild` и т. д. Однако существует множество вспомогательных функций, которые могут существенно упростить манипулирование деревом DOM, и в этом разделе рассказывается о том, как библиотека Base может помочь в таких случаях.

## Определение родительского элемента

В состав библиотеки Base входят несколько удобных функций, которые дополняют перечень общих функций для работы с DOM. Первая из этих функций, `isDescendant`, приведенная в табл. 2.1, имеет название, говорящее само за себя. Функция принимает два аргумента (значения атрибутов `id` или фактические ссылки на узлы), где первый аргумент представляет интересующий нас узел, а второй – потенциальный предок. Если интересующий нас узел в действительности входит в состав дерева DOM потенциального предка, функция возвращает значение `true`.

*Таблицы 2.1. Функция библиотеки Base для работы с деревом DOM*

| Имя | Тип возвращаемого значения | Комментарий |
|---|---|---|
| `dojo.isDescendant(/*String \| DomNode*/node, /* String \| DomNode*/potentialAncestor)` | Boolean | Возвращает логическое значение, которое свидетельствует о том, является ли узел `potentialAncestor` предком узла `node` или нет; работает с многоуровневыми иерархиями, как и следовало ожидать. |

## Селективность

Необходимость предотвратить возможность выделения текста на странице с помощью мыши возникает не так уж редко, и это иногда действительно способствует увеличению простоты и удобства использования. Фактически в каждом броузере имеется свой собственный способ

решить эту задачу, но это не повод для беспокойства, когда у вас имеется Dojo. Всякий раз, когда в этом возникает необходимость, достаточно просто вызвать функцию `dojo.setSelectable`. Ниже приводится сигнатура этой функции:

```
dojo.setSelectable(/*String | DomNode*/node, /*Boolean*/selectable)
```

Следует отметить, что ни одна операция, выполняемая на стороне клиента, не обеспечивает защиты содержимого, так как все, что отображается в броузере, доступно для исследования.

## Изменение стилей узлов

Функция `dojo.style` из библиотеки Base обеспечивает полную возможность получения или изменения значений отдельных атрибутов стиля для конкретного узла. Чтобы получить значение определенного атрибута стиля, достаточно просто передать функции узел и имя атрибута стиля в формате DOM (то есть `borderWidth`, а не `border-width`). Если в третьем необязательном аргументе функции передать значение атрибута стиля, она превратится из метода чтения в метод записи. Например, вызов `dojo.style("foo", "height")` вернет высоту элемента со значением атрибута `id`, равным `"foo"`, а вызов `dojo.style("foo", "height", "100px")` установит высоту элемента равной 100 пикселям. Существует также возможность одним обращением к функции изменить значения сразу нескольких атрибутов, используя значение типа `Object` в качестве второго аргумента, например, так:

```
dojo.style("foo", {
    height : "100px",
    width : "100px",
    border : "1px green"
});
```

Во многих приложениях помимо возможности манипулировать определенными атрибутами стиля с помощью `dojo.style` возникает необходимость добавлять, удалять, переключать и проверять существование определенных классов стилей. Все это можно реализовать с помощью набора функций манипулирования классами, которые к тому же имеют одинаковые сигнатуры. Первым параметром в эти функции передается интересующий узел DOM, а вторым – имя класса, которым требуется манипулировать. Например, чтобы добавить в узел стилевой класс, достаточно выполнить такой вызов: `dojo.addClass("foo", "someClassName")`. Обратите внимание, что в имени класса отсутствует ведущий символ точки, которая необходима при определении класса в таблицах стилей.

В табл. 2.2 приводится перечень функций, предназначенных для манипулирования внешним видом узла.

*Таблица 2.2. Функции из библиотеки Base, предназначенные для работы со стилями*

| Имя | Комментарий |
|---|---|
| `dojo.style(/*DomNode\|String*/ node, /*String?\|Object?*/style, /*String?*/value)` | Обеспечивает возможность получения и изменения значений определенных атрибутов стиля узла. |
| `dojo.hasClass(/*DomNode*/node, /*String*/classString)` | Возвращает `true`, только если к узлу применен указанный класс. |
| `dojo.addClass(/*DomNode*/node, /*String*/classString)` | Добавляет указанный класс к узлу. |
| `dojo.removeClass(/*DomNode*/node, /*String*/classString)` | Удаляет указанный класс из узла. |
| `dojo.toggleClass(/*DomNode*/node, /*String*/classString)` | Добавляет указанный класс к узлу в случае его отсутствия; удаляет его, если класс присутствует. |

## Манипулирование атрибутами

Аналогично приемам работы со стилями узлов, обсуждавшимся в предыдущем разделе, библиотека Base предоставляет функции, позволяющие получать, изменять, удалять и проверять существование атрибутов. Перечень доступных функций приводится в табл. 2.3.

*Таблица 2.3. Функции из библиотеки Base, предназначенные для работы с атрибутами*

| Имя | Комментарий |
|---|---|
| `dojo.attr(/*DOMNode\|String*/node, /*String?\|Object?*/attrs, /*String?*/value)` | Обеспечивает возможность получения и изменения значений определенных атрибутов узла. |
| `dojo.hasAttr (/*DOMNode\|String*/node, /*String*/name)` | Возвращает `true`, только если узел обладает указанным атрибутом. |
| `dojo.removeAttr (/*DOMNode\|String*/node, /*String*/name)` | Удаляет указанный атрибут из узла. |

Функция `dojo.attr` работает точно так же, как функция `dojo.style`, то есть она может устанавливать значение как одного, так и нескольких атрибутов сразу в зависимости от того, будет ли с помощью второго и третьего аргументов задан атрибут и его значение или во втором аргументе будет передан ассоциативный массив, содержащий совокупность атрибутов и их значений. Имена функций `hasAttr` и `removeAttr` говорят сами за себя и работают именно так, как и можно было предположить.

## Добавление узлов

Встроенные методы манипулирования содержимым дерева DOM, такие как `appendChild`, `insertBefore` и т. д., вполне справляются со своей работой, но иногда гораздо удобнее иметь более универсальное средство размещения узлов. Именно таким средством и является функция `dojo.place`, описание которой приводится в табл. 2.4. В двух словах: функция принимает три параметра – узел, который должен быть вставлен, ссылку на другой узел, относительно которого выполняется вставка, и параметр `position`, определяющий отношения между двумя узлами. В параметре `position` могут передаваться следующие значения: `"before"`, `"after"`, `"first"` и `"last"`. Значения `"before"` и `"after"` могут использоваться для размещения узла по соседству с указанным, а значения `"first"` и `"last"` могут использоваться для вставки дочернего узла внутрь родительского узла, представленного ссылкой. В аргументе `position` допускается также передавать целочисленное значение, которое в этом случае будет определять абсолютное положение вставляемого узла среди дочерних узлов узла, заданного ссылкой.

*Таблица 2.4. Добавление узла*

| Имя | Комментарий |
|---|---|
| `dojo.place(/*String\|DomNode*/node, /*String\|DomNode*/refNode, /*String\|Number*/position)` | Расширяет функциональные возможности DOM, предоставляя универсальную функцию добавления одного узла относительно другого. Возвращает логическое значение. |

## Блочная модель

Блочная модель CSS – это достаточно простая тема, но из-за наличия множества несогласованностей в ее реализациях, существующих в Сети, работа с ней быстро превращается в кошмар. Этот короткий раздел лишь немного затрагивает вопросы работы с блочной моделью, поэтому, если вы *действительно* хотите получить полное представление о сути всего этого, обращайтесь к более авторитетным источникам, таким как книга Эрика Мейера (Eric Meyer) «CSS: The Definitive Guide» (O'Reilly).[1]

Помимо различных несоответствий в реализациях блочной модели, существует еще проблема поддержки блочной модели CSS2 и блочной модели CSS3. О блочной модели CSS2 можно прочитать в спецификации CSS2 по адресу *http://www.w3.org/TR/REC-CSS2/box.html*, а с рабочим проектом CSS3 можно ознакомиться по адресу *http://www.w3.org/TR/css3-box/*.

[1] Эрик Мейер, «CSS – каскадные таблицы стилей. Подробное руководство», 3-е издание. – Пер. с англ. – СПб.: Символ-Плюс, 2008. – *Прим. перев.*

Суперкратко можно сказать, что блочная модель была разработана как способ обеспечения гибкого визуального представления, при котором управление высотой и шириной содержимого происходит с помощью последовательности вложенных прямоугольников, окружающих элемент страницы. Прежде чем продолжить обсуждение, взгляните на рис. 2.2, который выражает основную идею блочной модели.

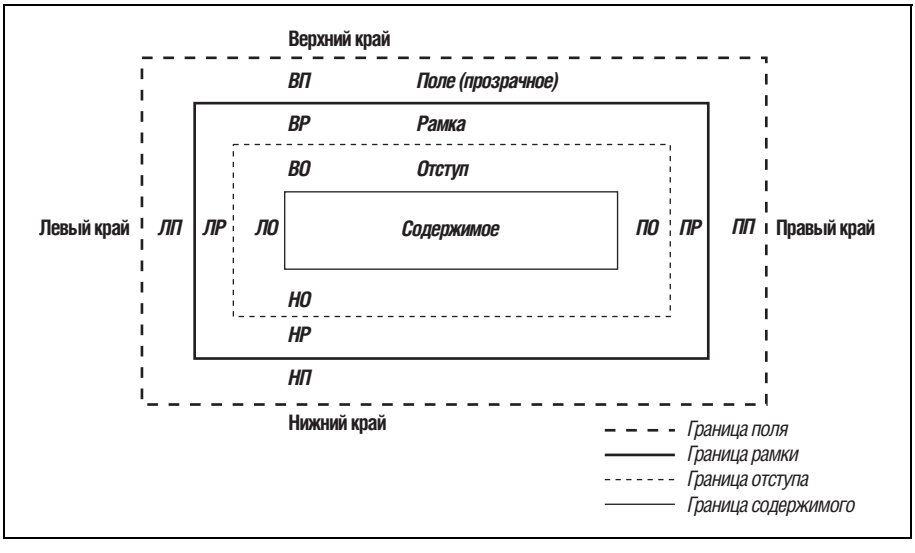



***Рис. 2.2.*** *Определение ширины и высоты содержимого в блочной модели CSS 2.1*

Чтобы свести воедино понятия содержимого, полей, отступов и рамки, рассмотрим следующую выдержку из спецификации:

> Поле, рамка и отступ могут быть разбиты на левые, правые, верхние и нижние сегменты (например, на диаграмме «ЛП» относится к левому полю, «ПО» – к правому отступу, «ВР» – к верхней рамке, и т. д.). Периметр каждой из четырех областей (содержание, отступ, рамка и поле) называются «границей» («edge»), то есть каждый блок имеет четыре границы:
>
> 1 – граница содержимого или внутренняя граница
>
> Граница содержимого окружает отображаемое содержимое элемента.
>
> 2 – граница отступа
>
> Граница отступа задает отступ от блока. Если отступ имеет ширину 0, то граница отступа совпадает с границей содержимого. Граница отступа от блока определяет границы объемлющего прямоугольника для этого блока.

3 – граница рамки

Граница рамки окружает рамку блока. Если рамка имеет ширину 0, то граница рамки совпадает с границей отступа.

4 – граница поля или внешняя граница

Граница поля окружает поля блока. Если поле имеет ширину 0, то граница поля совпадает с границей рамки.

Как оказывается, существует два подхода к пониманию принципов блочной модели. Различие заключается в том, откуда начинать отсчет – от *блока содержимого* или от *блока рамки*. Главное различие между этими двумя пониманиями заключается в ответе на вопрос, как поля и рамка применяются к области содержимого. При подходе, когда за основу принимается блок содержимого, любые области, образуемые отступами и рамками, добавляются снаружи к области содержимого с явно заданной шириной и высотой, тогда как при подходе, когда за основу принимается блок рамки, считается, что ширина рамок и отступов является частью явно заданной ширины и высоты области содержимого. Другими словами, при подходе, когда за основу принимается блок содержимого, явно заданные высота и ширина относятся только к содержимому, тогда как при подходе, когда за основу принимается блок рамки, высота и ширина относятся ко всему, что находится внутри блока рамки.

Многие современные броузеры поддерживают оба режима: стандартный режим и специальный режим. Подход, когда за основу принимается блок содержимого, соответствует стандартному режиму, а подход, когда за основу принимается блок рамки – специальному.

Если вы не собираетесь делать ничего из ряда вон выходящего, а просто хотите разместить некоторое содержимое на странице, то отличия могут быть не очень заметны, и, вообще говоря, тот же эффект можно получить несколькими способами. Однако, если необходимо добиться какого-то особого размещения элементов страницы, все решения могут оказаться принятыми за вас, а задача достижения одинакового отображения страницы в разных броузерах – это как раз то место, где начинается (или заканчивается) самое забавное.

Инструментальный набор пытается нормализовать различия в вычислениях различных аспектов блочной модели, для чего предоставляется атрибут `dojo.boxModel`, который может принимать значение `"content-box"` или `"margin-box"`, и функция `dojo.contentBox`, которая может использоваться для получения координат блоков. По умолчанию атрибут `dojo.boxModel` имеет значение `"content-box"`. Во всех случаях параметр `box` в сигнатурах функций, представленных в табл. 2.5, является ссылкой на объект, содержащий значения ширины и высоты, а также координаты верхнего левого угла прямоугольной области. Например, для узла, верхний левый угол которого отстоит на 50px правее и на 200px

ниже начала координат родительского элемента, при ширине прямоугольника с полями 300px и высоте 150px параметр box, описывающий блок, мог бы выглядеть так: { l: 50, t: 200, w: 300: h: 150 }.

*Таблица 2.5. Свойства для работы с блочной моделью*

| Имя | Тип возвращаемого значения | Комментарий |
|---|---|---|
| dojo.marginBox(/*DomNode\|String*/node, /*Object?*/box) | Object | Возвращает объект, содержащий размеры и координаты блока рамки узла. |
| dojo.contentBox(/*DomNode\|String*/node, /*Object?*/box) | Object | Возвращает объект, содержащий размеры и координаты блока содержимого узла. |
| dojo.coords(/*HTMLElement*/node, /*Boolean*/includeScroll) | Object | Возвращает данные для блока рамки узла, включая данные об абсолютном местоположении. В дополнение к значениям t, l, w и h возвращаются значения x и y, содержащие абсолютные координаты элемента на странице. Эти значения измеряются относительно начала координат видимой области, если параметр includeScroll установлен в значение true. Функция dojo.coords не может играть роль метода, выполняющего изменение значений. |

Чтобы поэкспериментировать самостоятельно, скопируйте следующий фрагмент в локальный файл и откройте его в броузере Firefox:

```
<body style="margin:3px">
    <div id="foo" style="width:4px; height:4px; border:solid 1px;"></div>
</body>
```

Ниже приводятся некоторые результаты, которые можно получить в Firebug, если скопировать страницу и поэкспериментировать с ней, а на рис. 2.3 показано, как эта страница выглядит в окне броузера:

```
console.log("box model", dojo.boxModel); // content-box
console.log("content box", dojo.contentBox("foo")); // l=0 t=0 w=4 h=4
console.log("margin box", dojo.marginBox("foo"));   // l=3 t=3 w=6 h=6
```

Подобно другим функциям, которые встречаются в этой главе, вызов функции с единственным параметром, указывающим на узел, возвращает значение, тогда как вызов со вторым необязательным параметром задает значение для этого узла. В табл. 2.5 перечислены все свойства, предназначенные для работы с блочной моделью.

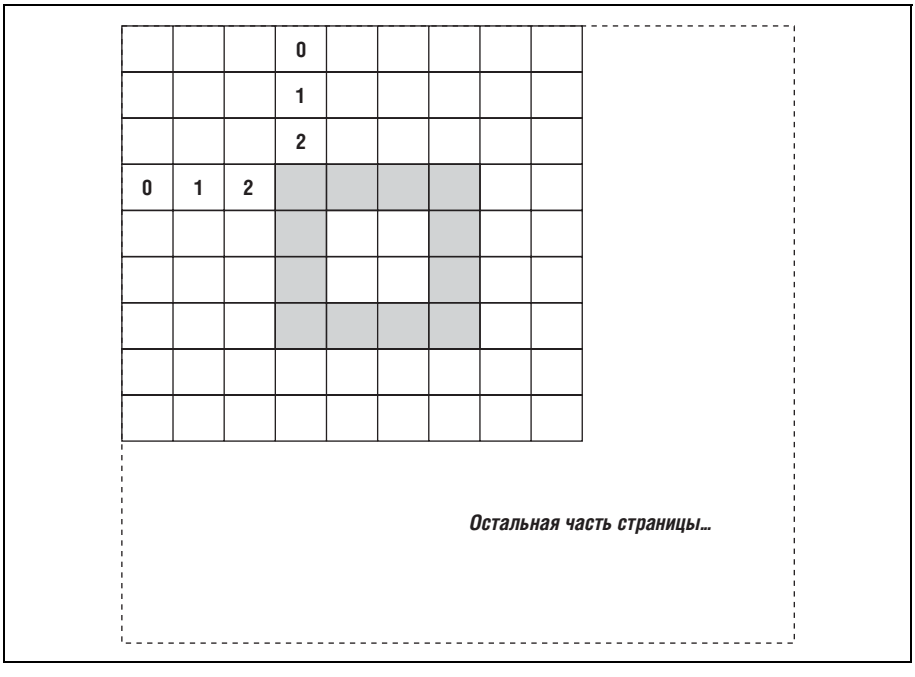



***Рис. 2.3.*** *Страница примера в броузере*

Библиотека Dijit широко использует средства блочной модели, чтобы обеспечить переносимость виджетов между броузерами.

# Утилиты для работы с броузером

В этом разделе представлен краткий обзор вспомогательных функций из инструментального набора, предназначенных для работы с cookies и с кнопкой «Назад» – двух возможностей, поддержка которых является обычной для любого современного веб-приложения. Поскольку все необходимые функции находится в библиотеке Core, прежде чем воспользоваться ими, следует загрузить их с помощью `dojo.require`.

## Cookies

Протокол HTTP является протоколом, не использующим информацию о состоянии, поэтому как только веб-сервер закончит передачу страницы, он не будет иметь никакой информации о клиенте. Такой принцип действия Сети прекрасно зарекомендовал себя во многих отношениях, но он не идеален в ситуациях, когда приложению необходимо отображать страницу с учетом личных предпочтений пользовате-

### Определение типа броузера

Набор инструментальных средств Dojo обычно берет на себя всю тяжесть по сглаживанию расхождений в броузерах, но иногда по каким-либо причинам может возникнуть потребность самостоятельно определить тип броузера. В этой ситуации можно обращаться к следующим атрибутам из библиотеки Base и интерпретировать их значения как логические величины (любое значение, большее или равное 1, интерпретируется как истина), чтобы быстро выполнить проверку и не захламлять свой программный код шаблонными вставками:

- `dojo.isOpera`
- `dojo.isKhtml`
- `dojo.isSafari`
- `dojo.isMozilla`
- `dojo.isFF`
- `dojo.isIE`
- `dojo.isAIR`
- `dojo.isQuirks`

Обратите внимание: атрибут `isMozilla` будет иметь значение `true` при использовании любой разновидности механизма отображения Gecko, тогда как `isFF` определяет только механизм отображения Gecko, применяемый исключительно броузером Firefox.

Атрибут `isQuirks` будет иметь значение `true`, если броузер действует в режиме обратной совместимости. Большинство броузеров по умолчанию переключаются в специальный (quirks) режим, если первым элементом страницы не является определение типа документа (DTD), утверждающее обратное.

ля, выбранных им к этому моменту. Например, для сайта, предоставляющего прогноз погоды, было бы нелишним запоминать код вашего региона, чтобы вам не приходилось вводить его всякий раз при посещении страницы.

Концепция cookies впервые была разработана в компании Netscape с целью смягчить эту проблему и дать броузерам ограниченную форму краткосрочной памяти. Проще говоря, разработчики веб-страниц могут с помощью JavaScript или с помощью сценариев на стороне сервера создать cookie с информацией о посещении страницы в виде пар имя/значение. При очередном посещении страницы сценарии могут извлечь информацию из cookie и динамически подстроить страницу под вас. Вообще cookies имеют конечный срок действия и всегда связаны с определенным доменом, откуда они были получены.

Одна из проблем, возникающих при работе с cookie из JavaScript, заключается в том, что вам приходится помнить некий строгий требуемый синтаксис и самостоятельно конструировать строки. Например, чтобы создать cookie для домена по умолчанию, содержащий пару имя/значение `foo=bar`, с определенным сроком действия, необходимо выполнить примерно следующее действие:

```
document.cookie ='foo=bar; expires=Sun, 15 Jun 2008 12:00:00 UTC; path=/'
```

Безусловно, в этом нет ничего сложного. Но когда потребуется прочитать значения из cookie, вам придется самостоятельно проанализировать строку, которая может содержать большое число пар имя/значение.

В наборе инструментальных средств Dojo имеются функции, выполняющие операции над cookie, пользоваться которыми гораздо легче. Описание этих функций приводится в табл. 2.6.

*Таблица 2.6. Функции dojo.cookie*

| Имя | Комментарий |
|---|---|
| `dojo.cookie(/*String*/name, /*String*/value, /*Object?*/properties)` | Действует как метод чтения значения cookie (возвращает значение типа `String`), когда функции передается только первый аргумент, являющийся именем определенного значения. Если функции передаются два первых аргумента, она действует как метод записи, который записывает в cookie пару имя/значение. Последний параметр properties может содержать следующие пары ключ/значение для определенных свойств cookie:<br>`expires (Date\|String\|Number)`<br>Если в виде значения передается число, оно определяет количество дней, начиная от текущих суток и до истечения срока действия cookie (если получается, что срок действия уже истек, cookie удаляется). Если свойство `expires` отсутствует или равно `0`, срок действия cookie истекает при закрытии броузера.<br>`path (String)`<br>Путь, с которым используется cookie.<br>`domain (String)`<br>Домен, с которым используется cookie.<br>`secure (Boolean)`<br>Определяет, следует ли передавать cookie только через защищенное соединение. |
| `dojo.cookie.isSupported()` | Возвращает значение типа `Boolean`, свидетельствующее о наличии поддержки cookies в броузере. |

Например, установить и прочитать значение cookie можно следующим образом:

```
dojo.cookie("foo","bar", {expires : 30});
//установит пару ключ/значение foo/bar,
//со сроком действия 30 дней от текущей даты
dojo.cookie("foo"); //вернет значение bar для имени foo
```

## Работа с кнопкой «Назад»

В наши дни стало нормой, когда все приложение выполняется в пределах единственной страницы, которая никогда не перезагружается, но это влечет за собой проблему управления действием кнопки «Назад» так, чтобы приложение могло корректно переключать свое состояние при ее нажатии и даже обеспечивать возможность установки закладок. В модуле `back` из библиотеки Core имеется простая вспомогательная функция, которая упрощает работу по отслеживанию состояний приложения, позволяя явно определять их и действовать соответствующим образом при нажатии кнопки «Назад» или «Вперед». Описания функций приводятся в табл. 2.7.

*Таблица 2.7. Функции модуля dojo.back*

| Имя | Комментарий |
|---|---|
| `init()` | Должна вызываться в теге `SCRIPT`, расположенном в теге `BODY` страницы, из-за особенностей Internet Explorer. Если заранее известно, что приложение никогда не будет выполняться под управлением IE, эту функцию можно не вызывать. |
| `setInitialState(/*Object*/args)` | Используется, чтобы определить функцию обратного вызова, которая должна вызываться, когда приложение возвращается в «первоначальное» состояние. Вообще рекомендуется вызывать эту функцию самой первой в блоке `addOnLoad`. |
| `addToHistory(/*Object*/args)` | Обеспечивает возможность сохранения некоторого состояния приложения посредством аргумента `args`, в котором определяются функции-обработчики нажатий на кнопки «Назад» и «Вперед», а также необязательные идентификаторы в URL, которые могут использоваться для установки закладок. Аргумент `args` имеет следующую форму:<br>`back (Function)`<br>Функция обратного вызова, которая выполняется, когда осуществляется переход в заданное состояние нажатием на кнопку «Назад».<br>`forward(Function)`<br>Функция обратного вызова, которая выполняется при осуществлении перехода в заданное состояние нажатием на кнопку «Вперед». |

*Таблица 2.7 (продолжение)*

| Имя | Комментарий |
|---|---|
| | `changeUrl (Boolean\|String)`<br>Если имеет значение `true`, в URL вставляется случайный идентификатор, который будет использоваться внутренними механизмами для нужд сохранения состояния. Если имеет значение типа `String`, данная строка вставляется в URL и используется для тех же целей, а кроме того, обеспечивает удобную возможность создания закладки. Не следует комбинировать использование значений типа `Boolean` и `String`, применяйте что-то одно. |

Будьте последовательны и используйте в свойстве `changeUrl` объекта `args`, который передается функции `addToHistory`, либо значение типа `Boolean`, либо строковые идентификаторы.

Пример 2.8, как хотелось бы надеяться, поясняет основную идею; в нем иллюстрируется тривиальное использование функции `back` для определения функции обратного вызова, которая должна обеспечить необходимое поведение. Обратите внимание: строки, выделенные в листинге, необходимы для обеспечения нормальной работы в IE.

Для тех, кто задается вопросом, зачем потребовалось вставлять в раздел `BODY` страницы тег `SCRIPT`, что выглядит достаточно неуклюже, поясню – это связано с особенностью броузера IE, требующей выполнения метода `document.write`, который не может быть выполнен по окончании загрузки страницы. Это не очень элегантное решение, но оно работает во всех броузерах и позволяет обеспечить обслуживание кнопки «Назад».

*Пример 2.8. Пример обработки нажатия кнопки «Назад»*

```
<html>
    <head>
        <title>Fun with Back!</title>

        <link rel="stylesheet" type="text/css"
            href="http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css" />

        <script
            type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js"
            djConfig="dojoIframeHistoryUrl:'iframe_history.html',isDebug:true"
        ></script>

        <script type="text/javascript">

            dojo.addOnLoad(function() {
                initialState = {
                   back: function() { console.log("Back to initial state"); }
```

```
                };
                state1 = {
                    back:function() { console.log("Back to state 1"); },
                  forward:function() { console.log("Forward to state 1"); },
                  changeUrl:true //можно было бы использовать идентификатор,
                                 //такой как "state1"
                };
                state2 = {
                    back: function() { console.log("Back to state 2"); },
                  forward: function() {console.log("Forward to state 2"); },
                  changeUrl:true //можно было бы использовать идентификатор,
                                 //такой как "state2"
                };

                //установить начальное состояние и переместиться вперед
                //на два шага в истории
                dojo.back.setInitialState(initialState);
                dojo.back.addToHistory(state1);
                dojo.back.addToHistory(state2);
            });
        </script>
    <head>
    <body>
        <script type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/back.js"></script>
        <script type="text/javascript">dojo.back.init( );</script>

        Press the back button and have a look at the console.
    </body>
</html>
```

# В заключение

После прочтения этой главы вы должны:

- Представлять основные возможности библиотеки Base
- Уметь настраивать массив параметров `djConfig`; определять в нем пути к модулям и знать о других параметрах, которые можно передавать в этой структуре для настройки процесса загрузки и инициализации
- Понимать, как используются функции `dojo.addOnLoad` и `dojo.addOnUnload`, и знать, как функция `dojo.addOnLoad` может защитить от попадания в состояние гонки за ресурсами
- Уметь создавать собственные модули и соответствующие им пространства имен с помощью функций `dojo.provide` и `dojo.require`
- Понимать, как (и когда) следует использовать функции `map`, `filter` и `forEach`
- Знать о различиях и уметь эффективно применять функции `mixin` и `extend`

- Эффективно использовать вспомогательные функции Dojo, предназначенные для манипулирования стилями `hasClass`, `removeClass`, `addClass` и `toggleClass`
- Понимать основы блочной модели CSS и уметь использовать такие функции, как `coords` и `marginBox`, предназначенные для манипулирования местоположением узлов DOM
- Знать о существовании в библиотеке Base вспомогательных функций для работы с массивами
- Уметь произвольным образом связывать объекты и события DOM
- Уметь управлять информацией в cookies
- Уметь пользоваться средствами из библиотеки Core, предназначенными для обслуживания кнопки «Назад» в одностраничном приложении

Далее мы рассмотрим обработчики событий и организацию взаимодействий.

# 3

# Обработчики событий и организация взаимодействий по подписке

В состав библиотеки Base входят чрезвычайно удобные и гибкие утилиты для организации взаимодействий между объектами JavaScript, узлами DOM и любыми их комбинациями. В этой главе будут представлены все эти конструкции, а также даны рекомендации, когда и какую из них предпочтительнее использовать. Переносимость программного кода, обрабатывающего события DOM, заведомо зависит от стандартизации модели событий, поэтому будет немного рассказано о том, как внутренние механизмы инструментального набора Dojo сглаживают некоторые несоответствия, имеющиеся между броузерами в области обработки событий мыши и клавиатуры. Заканчивается глава обсуждением организации взаимодействий по подписке, которая возможна благодаря механизму реализации архитектуры с гибко связанными компонентами.

## Нормализация событий и клавиатуры

Самый старый программный код, входящий в инструментальный набор, был написан с целью сгладить несоответствия между моделями событий, реализованными в разных броузерах. Этот раздел представляет собой краткий обзор событий, которые можно считать нормализованными при использовании Dojo для разработки приложений. Основой стандартизации является модель, предложенная консорциумом W3C.

### Нормализация событий от мыши и от клавиатуры

Механизм `dojo.connect`, который обсуждается в следующем разделе, часто имеет отношение к событиям мыши, возникающим в конкретных узлах DOM. Если вы используете Dojo, то можете быть совершенно

уверены, что перечисленные далее события мыши и клавиатуры поддерживаются в соответствии с рекомендациями стандарта W3C:

```
onclick
onmousedown
onmouseup
onmouseover
onmouseout
onmousemove
onkeydown
onkeyup
onkeypress
```

Помимо событий, стандартизованных консорциумом W3C, поддерживаются также нестандартные события `onmouseenter` и `onmouseleave`.

Помимо событий, которые запускаются стандартным способом, можно полагаться на нормализованные объекты событий, передаваемые функциям-обработчикам. На практике, при необходимости произвести нормализацию событий самостоятельно, вы можете воспользоваться следующей функцией из библиотеки Base:

```
dojo.fixEvent(/*DOMEvent*/ evt, /*DOMNode*/ sender) //Возвращает DOMEvent
```

Тип `DOMEvent` — это стандартное соглашение, которое будет использоваться в остальной части книги для ссылки на объекты событий DOM.

Другими словами, функции нужно передать событие и узел, который рассматривается как текущая цель события, и вы получите нормализованное событие, которое будет соответствовать спецификации W3C. В табл. 3.1 приводится краткое описание некоторых из наиболее часто используемых свойств объекта `DOMEvent`.[1]

*Таблица 3.1. Часто используемые свойства объекта DOMEvent*

| Имя | Тип | Комментарий |
|---|---|---|
| `bubbles` | Boolean | Указывает, может ли событие «всплывать» по дереву DOM. |
| `cancelable` | Boolean | Указывает, может ли быть отменено связанное с событием действие по умолчанию. |

[1] В настоящее время в Dojo реализована нормализация в соответствии со спецификацией DOM2, ознакомиться с которой можно по адресу *http://www.w3.org/TR/DOM-Level-2-Events/events.html*. Обзор спецификации модели событий DOM3 вы найдете по адресу *http://www.w3.org/TR/DOM-Level-3-Events/events.html*.

| Имя | Тип | Комментарий |
|---|---|---|
| currentTarget | DOMNode | Текущий узел, чей обработчик выполняет обслуживание события. (Удобно использовать на стадии всплытия события.) |
| target | DOMNode | Целевой узел, которому событие предназначалось изначально. |
| type | String | Тип события, например mouseover. |
| ctrlKey | Boolean | Указывает, удерживалась ли нажатой клавиша Ctrl в момент появления события. |
| shiftKey | Boolean | Указывает, удерживалась ли нажатой клавиша Shift в момент появления события. |
| metaKey | Boolean | Указывает, удерживалась ли нажатой клавиша Meta в момент появления события. (Это специальная клавиша в компьютерах Apple.) |
| altKey | Boolean | Указывает, удерживалась ли нажатой клавиша Alt в момент появления события. |
| screenX | Integer | Координата X на экране, где возникло событие. |
| screenY | Integer | Координата Y на экране, где возникло событие. |
| clientX | Integer | Координата X относительно окна броузера, где возникло событие. |
| clientY | Integer | Координата Y относительно окна броузера, где возникло событие. |

## Стандартизованные коды клавиш

Кроме всего прочего, инструментарий определяет коды клавиш, как показано в табл. 3.2, доступные в виде свойств объекта dojo.keys. Например, чтобы определить, была ли нажата комбинация клавиш Shift+Enter, можно использовать такой фрагмент программного кода:

```
/* ... обрезано ... */
    if (evt.keyCode == dojo.keys.ENTER && evt.shiftKey) {
        /* ... */
    }
/* ... обрезано ... */
```

В табл. 3.2 приводится список констант для обращения к кодам клавиш при обработке событий от клавиатуры.

*Таблица 3.2. Список констант, предоставляемых инструментарием Dojo для обращения к кодам клавиш, в виде свойств объекта dojo.keys*

| | | |
|---|---|---|
| BACKSPACE | DELETE | NUMPAD_DIVIDE |
| TAB | HELP | F1 |
| CLEAR | LEFT_WINDOW | F2 |

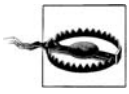
*Таблица 3.2 (продолжение)*

| | | |
|---|---|---|
| ENTER | RIGHT_WINDOW | F3 |
| SHIFT | SELECT | F4 |
| CTRL | NUMPAD_0 | F5 |
| ALT | NUMPAD_1 | F6 |
| PAUSE | NUMPAD_2 | F7 |
| CAPS_LOCK | NUMPAD_3 | F8 |
| ESCAPE | NUMPAD_4 | F9 |
| SPACE | NUMPAD_5 | F10 |
| PAGE_UP | NUMPAD_6 | F11 |
| PAGE_DOWN | NUMPAD_7 | F12 |
| END | NUMPAD_8 | F13 |
| HOME | NUMPAD_9 | F14 |
| LEFT_ARROW | NUMPAD_MULTIPLY | F15 |
| UP_ARROW | NUMPAD_PLUS | NUM_LOCK |
| RIGHT_ARROW | NUMPAD_ENTER | SCROLL_LOCK |
| DOWN_ARROW | NUMPAD_MINUS | |
| INSERT | NUMPAD_PERIOD | |

# Обработчики событий

Прямые каналы взаимодействий конструируются путем явного соединения функций и/или событий DOM так, что выполнение одной функции автоматически влечет за собой вызов другой. Например, вы можете захотеть, чтобы при каждом изменении какого-либо объекта посредством метода «записи» автоматически производилось изменение в визуальном интерфейсе приложения. Или чтобы каждое изменение одного объекта автоматически вызывало обновление свойств другого объекта. Варианты могут быть самые разные.

В схему прямого взаимодействия вовлечены два основных метода – `dojo.connect` и `dojo.disconnect`. Проще говоря, метод `dojo.connect` используется для объединения в цепь последовательности событий. При каждом обращении к методу `dojo.connect` возвращается дескриптор, который необходимо сохранять и явно передавать методу `dojo.disconnect`, когда требуется разорвать связь. Обычно при выгрузке страницы все соединения разрываются автоматически, однако ручное управление дескрипторами может потребоваться, чтобы предотвратить утечки памяти в долгоживущих приложениях, которые устанавливают множество временных соединений. (Это особенно справедливо в отношении IE.) Следующие ниже сигнатуры методов уже были представлены в главе 1.



Не пытайтесь устанавливать связи, пока страница не будет полностью загружена. Попытка использовать `dojo.connect` до полной загрузки страницы – очень распространенная ошибка, которая может заставить вас потратить массу времени на выяснение того, что произошло, т. к. эту ошибку не очень легко отыскать, когда вы впервые сталкиваетесь с ней. Для обеспечения безопасности соединения всегда должны устанавливаться в пределах функции, которая передается функции `addOnLoad`.

Установка и разрыв соединения выполняются достаточно просто. Ниже приводятся сигнатуры используемых функций:

```
/* Устанавливает соединение */
dojo.connect(/*Object|null*/ obj,
             /*String*/ event,
             /*Object|null*/ context,
             /*String|Function*/ method) // Возвращает дескриптор Handle
/* Разрывает соединение */
dojo.disconnect(/*Handle*/handle);
```

С практической точки зрения вы должны рассматривать дескриптор, возвращаемый функцией `dojo.connect`, как некий «черный ящик», который не нужен ни для чего иного, кроме как для последующего разрыва соединения. (Если вам интересно, то этот объект не представляет собой ничего удивительного – просто блок информации, которая используется внутренними механизмами для управления соединением.)

Давайте рассмотрим пример, иллюстрирующий своего рода проблему, которую можно решить с помощью функции `dojo.connect`:

```
function Foo() {
    this.greet = function() { console.log("Hi, I'm Foo"); }
}

function Bar() {
    this.greet = function() { console.log("Hi, I'm Bar"); }
}

foo = new Foo;
bar = new Bar;

foo.greet();

//объект bar должен отвечать на приветствие объекта foo
//всегда, когда объект bar существует.
```

Оказывается, что решить эту маленькую проблему можно всего одной строкой программного кода. Измените предыдущий листинг, как показано ниже, и проверьте его в Firebug:

```
function Foo() {
    this.greet = function() { console.log("Hi, I'm foo"); }
}
```

```
function Bar() {
    this.greet = function() { console.log("Hi, I'm bar"); }
}

foo = new Foo;
bar = new Bar;

//При каждом вызове foo.greet автоматически будет вызываться bar.greet ...
var handle = dojo.connect(foo, "greet", bar, "greet"); //устанавливается
                                                       //соединение

foo.greet(); //теперь объект bar автоматически ответит на приветствие!
```

Вы написали всего одну строчку, а получили такой приятный результат – неплохо, не правда ли? Обратите внимание, что второй и четвертый параметры, переданные функции `dojo.connect`, – строковые литералы для соответствующих им контекстов и что функция возвращает дескриптор, который позднее позволит разорвать это соединение. Вообще говоря, *всегда* наступает некоторый момент, когда следует разорвать соединение – либо приложение достигает некоторого функционального состояния, либо вы выполняете некоторые завершающие действия, например при уничтожении некоторого объекта или при закрытии страницы, примерно так, как показано ниже:

```
var handle = dojo.connect(foo, "greet", bar, "greet");
foo.greet();

dojo.disconnect(handle);

foo.greet(); //на сей раз ответного приветствия не последует
```

Кроме того что `dojo.connect` позволяет добиться такого значительного эффекта такими малыми усилиями, обратите внимание, насколько опрятным и понятным остался программный код. Никаких шаблонных заготовок, никакой путаницы, никаких накрученных решений и никакого кошмара для того, кто будет это сопровождать.

Запуск методов в ответ на события, происходящие в странице, – действительно очень удобная возможность, но помимо этого рано или поздно появится необходимость передавать таким методам какие-нибудь аргументы. Оказывается, функция `connect` обладает возможностью передавать аргументы из контекста одной функции в контекст другой функции. Ниже приводится пример, как это делается:

```
function Foo() {
    this.greet = function(greeting) { console.log("Hi, I'm Foo.",greeting); };
}

function Bar() {
    this.greet = function(greeting) { console.log("Hi, I'm Bar.",greeting); };
}

foo = new Foo;
bar = new Bar;
```

```
var handle= dojo.connect(foo, "greet", bar, "greet");
foo.greet("Nice to meet you");
```

Вы наверняка можете оценить, насколько удобно, когда передача аргументов происходит автоматически, и это особенно верно для функций, связанных с событиями DOM, например щелчок мышью, потому что такая возможность обеспечивает для функции моментальный доступ ко всем важным сведениям о событии, таким как целевой элемент, координаты указателя мыши и т. д. Давайте рассмотрим еще один пример:

```
//Обратите внимание, что третий аргумент опущен, так как обработчик -
//это анонимная функция. Если в третьем аргументе передать значение null,
//мы получим тот же самый эффект.
dojo.connect(
    dojo.byId("foo"), //Некоторый элемент DOM
    "onmouseover",
    function(evt) {
        console.log(evt);
    });
```

Если создать пробную страницу, установить в ней соединение и наблюдать за происходящим в консоли Firebug, можно заметить, что функции-обработчику объект события, содержащий практически все сведения, какие только может потребоваться знать о только что произошедшем событии доступен целиком.

Возникает резонный вопрос: «Если настолько просто определять обработчики событий DOM, зачем тогда заниматься изучением другой библиотечной функции?». Да, возможно, не требуется быть нейрохирургом, чтобы собрать вместе несколько простых обработчиков событий, но как быть, когда имеется сложное приложение, основанное на обработке большого числа сложных событий с учетом предпочтений пользователя, на обработке нестандартных событий или обладающее иным поведением, управляемым событиями? Несомненно, всю необходимую работу можно выполнить вручную, но сможете ли вы устанавливать и разрывать соединение однострочными командами, с единым непротиворечивым интерфейсом, уже написанными и всесторонне проверенными?

Наконец, обратите внимание: хотя в приведенных примерах связывались между собой только пара событий, вам ничто не мешает связать воедино произвольное число обычных функций, методов объектов и событий DOM для их последовательного вызова.

## Распространение событий

Бывают моменты, когда необходимо переопределить обработку некоторых событий DOM, встроенную в броузер, и с помощью функции `dojo.connect` подставить свои функции, которые будут обрабатывать эти события. В качестве двух наиболее типичных примеров можно

привести необходимость предотвратить автоматический переход броузера после щелчка на гиперссылке и необходимость предотвратить автоматическую отправку формы по нажатию на клавишу Enter или в результате щелчка на кнопке Submit.

К счастью, помешать броузеру выполнить действие по умолчанию, заданное для этих событий DOM, очень просто – достаточно лишь вызвать функцию `dojo.stopEvent` или метод `preventDefault` объекта `DOMEvent`, и событие прекратит свое распространение по броузеру. Функция `stopEvent` принимает в качестве параметра объект `DOMEvent`:

```
dojo.stopEvent(/*DOMEvent*/evt)
```

Распространение события DOM можно подавить в последовательности функций, связанных с событием с помощью `dojo.connect`, но нет никакой возможности остановить работу цепочки обычных функций или методов объектов JavaScript, связанных функцией `dojo.connect`.

Следующий пример демонстрирует применение функции `stopEvent`:

```
var foo = dojo.byId("foo"); //некоторый якорный элемент

dojo.connect(foo, "onclick", function(evt) {
    console.log("anchor clicked");
    dojo.stopEvent(evt); //предотвратит переход по ссылке и дальнейшее
                         //всплытие события
});
```

Точно так же просто отменяется автоматическая отправка формы, для этого достаточно передать контекст соединения и связать его с событием `submit`. Однако на этот раз, чтобы отменить действие события по умолчанию, мы воспользуемся методом `preventDefault` объекта `DOMEvent`, который при этом не предотвращает дальнейшее всплытие события в дереве DOM:

```
var bar = dojo.byId("bar"); //some form element

dojo.connect(bar, "onsubmit", function(evt) {
    console.log("form submitted");
    evt.preventDefault();//предотвратит отправку формы, но не запретит
                         //дальнейшее всплытие события
});
```

## Использование замыканий с функцией dojo.connect

В этом разделе рассматривается относительно сложная тема, поэтому вы можете просто бегло ознакомиться с ней – так, чтобы не увязнуть при первом прочтении и вернуться сюда позднее, потому что рано или поздно эти сведения вам потребуются.

## Однократные соединения

Рассмотрим ситуацию, когда необходимо установить соединение, которое должно быть разорвано после первого же срабатывания. Следующий пример показывает, как выполнить эту работу с минимальными усилиями:

```
var handle = dojo.connect(
    dojo.byId("foo"), //некоторый элемент div
    "onmouseover",
    function(evt) {
        //здесь находится тело некоторого обработчика...
        dojo.disconnect(handle);
    }
);
```

Если вы еще недостаточно уверенно чувствуете себя при работе с замыканиями, вашей первой реакцией может быть: «То, что мы только что сделали, попросту невозможно». В конце концов, переменная `handle` получает значение, возвращаемое функцией `dojo.connect`, и при этом ссылка на нее используется внутри функции, которая передается в `dojo.connect` в качестве параметра. Чтобы лучше понять ситуацию, разберем все происходящее в деталях:

1. Вызывается функция `dojo.connect`, и, хотя одним из ее параметров является анонимная функция, она в этот момент еще не выполняется.
2. Любые переменные внутри анонимной функции (такие как `handle`) связаны с ее областью видимости, и хотя они присутствуют в теле функции, фактическое обращение к ним происходит, только когда функция действительно будет вызвана, поэтому в этом программном коде не возникает никакой ошибки.
3. Функция dojo.connect возвращает значение переменной `handle` еще до того, как анонимная функция будет вызвана. Поэтому к моменту вызова анонимной функции значение переменной будет определено и может передаваться функции `dojo.disconnect`.

## Установка соединений в цикле

Еще одна ситуация, часто встречающаяся на практике, – необходимость устанавливать соединения в теле цикла. Предположим, что у нас в странице имеется несколько элементов – `foo0`, `foo1`, ..., `foo9`, и нам необходимо при перемещении указателя мыши над этими элементами выводить уникальное для каждого из них число. При первой попытке вы могли бы прийти к следующему фрагменту программного кода, который, впрочем, *не даст* ожидаемого результата:

```
/* Следующий фрагмент работает не так, как ожидается! */
for (var i=0; i < 10; i++) {
    var foo = dojo.byId("foo"+i);
    var handle = dojo.connect(foo, "onmouseover", function(evt) {
        console.log(i);
```

```
        dojo.disconnect(handle);
    });
}
```

Если вы запустите этот фрагмент в Firebug на странице с серией указанных элементов, вы быстро обнаружите, что тут имеется проблема. А именно, в консоли всегда будет выводиться число 10, то есть всеми функциями-обработчиками будет использоваться последнее значение переменной i, а это означает, что все десять обработчиков по ошибке будут пытаться разорвать одно и то же соединение. Остановимся на минутку, чтобы обдумать ситуацию. Однако такому неожиданному для вас поведению имеется разумное объяснение: внутри замыкания, образованного анонимной функцией, переданной функции dojo.connect, не выполняется разрешение имени i, пока функция действительно не будет вызвана, но к этому моменту переменная i будет иметь последнее полученное в цикле значение.

Следующие изменения устраняют проблему, захватывая значение переменной i в ловушку цепочки областей видимости, чтобы при последующем обращении к переменной возвращалось значение, которое было на момент вызова функции dojo.connect:

```
for (var i=0; i < 10; i++) {
    (function() {
        var foo = dojo.byId("foo"+i);
        var current_i = i; //поймать в ловушку замыкания
        var handle = dojo.connect(foo, "onmouseover",
            function(evt) {
                console.log(current_i);
                dojo.disconnect(handle);
            }
        );
    })(); //выполнить анонимную функцию немедленно
}
```

Поначалу этот фрагмент программного кода может показаться замысловатым, но на самом деле здесь нет ничего сложного. Все тело цикла оформлено в виде анонимной функции, которая тут же и выполняется, а поскольку анонимные функции образуют замыкание для всего, что находится внутри, значение переменной i попадает в «ловушку» переменной current_i, которая может быть разрешена во время выполнения обработчика события. Точно так же правильно будет разрешаться и переменная handle, потому что она тоже существует в пределах замыкания, образованного встроенной анонимной функцией.

Если раньше вы никогда не встречались с такими замыканиями в действии, возможно, вам стоит потратить некоторое время на более внимательное изучение программного кода, чтобы полностью понять его. Вероятно, вы уже устали слушать о замыканиях, но в дальнейшем уверенное понимание этой темы сослужит вам хорошую службу в работе с JavaScript.

## Соединения в тексте разметки

Стоит заметить, что точно так же можно создавать соединения для диджитов вообще без (или с минимальным использованием) программного кода JavaScript при использовании специальных тегов `SCRIPT` с функцией `dojo.connect` в тексте разметки. Подробнее об этом вы можете прочитать в главе 11, когда будет дано формальное введение в библиотеку Dijit.

# Организация взаимодействий по подписке

Существует масса ситуаций, когда прямой «цепочечный» стиль организации взаимодействий посредством функции `dojo.connect` является именно тем средством, которое необходимо для решения проблем. Однако существует не меньше ситуаций, когда требуется более опосредованный «широковещательный» стиль организации взаимодействий, когда различные виджеты взаимодействуют друг с другом анонимно. В таких случаях можно использовать функции `dojo.publish` и `dojo.subscribe`.

Классическим примером является ситуация, когда необходимо организовать взаимодействие между одним объектом JavaScript и несколькими другими объектами по типу отношений «один ко многим». Вместо того чтобы создавать и управлять множественными соединениями, создаваемыми с помощью функции `dojo.connect`, которые больше напоминают одно связное действие, значительно проще организовать передачу уведомления о событии, произошедшем в одном виджете (наряду с данными, сопутствующими этому событию) , чтобы другие виджеты могли подписаться на получение этих извещений и автоматически принимать соответствующие меры. Вся прелесть такого подхода состоит в том, что объект, осуществляющий отправку уведомлений, ничего не должен знать о других объектах и даже не должен делать никаких предположений об их существовании. Другим классическим примером такого рода взаимодействий являются *портлеты* – подключаемые компоненты интерфейса (*http://en.wikipedia.org/wiki/Portlet*[1]), которые обслуживаются веб-порталом, иногда через панель управления.

OpenAjax Hub (*http://www.openajax.org/OpenAjax%20Hub.html*), о котором подробнее будет рассказываться в главе 4, использует взаимодействия по подписке как механизм эффективного задействования множества библиотек JavaScript в пределах одной и той же страницы.

Во многих ситуациях при помощи организации взаимодействий по подписке можно достичь тех же возможностей, что и в результате прямого соединения элементов, поэтому решение о применении механизма

[1] На русском языке: *http://ru.wikipedia.org/wiki/Портлет*. – *Прим. перев.*

обработки событий по подписке часто зависит от практичности метода, от конкретной проблемы, которую требуется решить, и личных предпочтений, отдаваемых одному из подходов.

Для начала, чтобы определить, какой стиль организации взаимодействий лучше подходит для решения проблемы, попробуйте ответить на следующие вопросы:

- Предполагаете ли вы (и насколько это правильно) публиковать прикладной интерфейс виджета, который вы разрабатываете? Если нет, тогда вам определенно следует использовать взаимодействия по подписке, потому что это позволит вам свободно менять архитектуру виджета, не опасаясь вступить в конфликт с его опубликованным API.
- Предполагается ли одновременное использование нескольких однотипных виджетов, которые порождают один и тот же тип событий? Если да, тогда вам определенно следует использовать взаимодействия, основанные на прямых соединениях, потому что в противном случае вам придется разрабатывать дополнительную логику, которая будет определять, какой виджет и на какое событие должен реагировать.
- Разрабатываете ли вы виджет, содержащий другие виджеты в отношении «имеет»? Если да, тогда вам следует использовать взаимодействия, основанные на прямых соединениях.
- Предполагается ли возможность организации взаимодействий по принципу «один ко многим» или «многие ко многим»? Если да, тогда определенно следует использовать взаимодействия по подписке, чтобы минимизировать тяжесть организации взаимодействий, основанных на прямых соединениях.
- Предполагается ли, что взаимодействия должны быть полностью анонимными и должна ли быть обеспечена свобода в выборе взаимодействующих сторон? Если да, тогда следует использовать взаимодействия по подписке.

Не будем долго задерживаться и рассмотрим интерфейс организации взаимодействий по подписке. Обратите внимание, что в случае функции `dojo.subscribe` можно опустить параметр `context`, а функция выполнит необходимую нормализацию аргументов (точно так же, как и в случае с функцией `dojo.connect`):

```
dojo.publish(/*String*/topic, /*Array*/args)
dojo.subscribe(/*String*/topic, /*Object|null*/context,
               /*String|Function*/method) //Возвращает Handle
dojo.unsubscribe(/*Handle*/handle)
```

Объект дескриптора, возвращаемый функцией `dojo.subscribe`, следует считать своеобразным черным ящиком, как и в случае с функцией `dojo.connect`.

Давайте рассмотрим простой пример, основанный на использовании функций `dojo.subscribe` и `dojo.publish`:

```
function Foo(topic) {

    this.topic = topic;

    this.greet = function() {
        console.log("Hi, I'm Foo");

      /* Объект Foo публикует информацию, но не для конкретного получателя */
        dojo.publish(this.topic);
    }
}

function Bar(topic) {

    this.topic = topic;

    this.greet = function() {
        console.log("Hi, I'm Bar");
    }

 / * Объект Bar подписывается на информацию, но не от конкретного источника */
    dojo.subscribe(this.topic, this, "greet");
}

var foo = new Foo("/dtdg/salutation");
var bar = new Bar("/dtdg/salutation");

foo.greet(); //Hi, I'm Foo...Hi, I'm Bar
```

Несмотря на отсутствие формального стандарта, инструментальный набор использует соглашение о начальном символе и использует начальный символ слеша для обозначения названия темы. Преимущество такого подхода заключается в том, что в программном коде JavaScript весьма непривычно видеть начальный символ слеша, поэтому он сразу бросается в глаза (тогда как при использовании точек в названиях тем выделить их было бы намного сложнее).

Как видите, функция `connect` связывает конкретный источник с конкретным адресатом, а функции `publish/subscribe` позволяют работать с широковещательными посылками, которые могут исходить из любого источника и обрабатываться любыми получателями, заинтересованными в получении таких посылок. Отчасти удивительная мощь, заключенная в архитектуре с гибко связанными компонентами, обусловлена тем, что при минимальных усилиях и большой простоте она позволяет получать приложения, которые концептуально представляют собой совокупность связанных компонентов.

Давайте посмотрим, как можно отписаться от получения уведомлений, внеся изменения в реализацию объекта `Bar`. Пусть теперь объект `Bar` отвечает на сообщение, публикуемое объектом `Foo`, только один раз:

```
function Bar(topic) {

    this.topic = topic;

    this.greet = function() {
        console.log("Hi, I'm bar");
        dojo.unsubscribe(this.handle);

        //не тревожь ты меня, я тебе не отвечу
    }

    this.handle = dojo.subscribe(this.topic, this, "greet");
}
```

Обратите внимание, что существует возможность вместе с сообщением передать как второй аргумент функции `publish` массив значений в виде именованных параметров, которые будут переданы обработчику, подписанному с помощью функции `subscribe`.

Очень часто встречается ошибка, когда разработчик забывает, что дополнительные аргументы должны передаваться функции `publish` в виде массива и что обработчик, подписанный с помощью функции `subscribe`, получает их в виде отдельных аргументов.

И, в заключение, предположим, что у нас отсутствует возможность изменить метод `greet` объекта `Foo`, чтобы включить в него вызов `dojo.publish`, из-за наличия каких-то внешних обстоятельств, которые препятствуют этому, например когда программный код не является вашей собственностью или не должен изменяться. Это не причина для беспокойств — мы можем использовать другую функцию, `dojo.connectPublisher`, которая будет выполнять публикацию события всякий раз, когда оно происходит:

```
function Foo() {
    this.greet = function() {
        console.log("Hi, I'm foo");
    }
}

function Bar() {
    this.greet = function() {
        console.log("Hi, I'm bar");
    }
}

var foo = new Foo;
var bar = new Bar;

var topic = "/dtdg/salutation";
dojo.subscribe(topic, bar, "greet");
dojo.connectPublisher(topic, foo, "greet");

foo.greet();
```

Возможно, вам будет интересно узнать, что внутри `connectPublisher` использует функцию `dojo.connect`, чтобы создать соединение между функцией `dojo.publish` и указанной функцией.

Суть этого заключительного примера состоит в том, что функция `dojo.connectPublisher` позволила добиться тех же результатов, что и добавление вызова `dojo.publish` в метод `greet`, но без изменения исходного программного кода. В этом смысле объект `foo` является косвенным источником уведомлений и даже не подозревает, что вообще осуществляется какое-то взаимодействие. С другой стороны, объект `Bar`, как подписчик на получение уведомления, требует явного знания схемы взаимодействий. По существу, это противоположный случай типичного использования функции `dojo.connect`, когда объект, предоставляющий информацию для обмена, должен явно знать о других объектах или функциях, которые играют роль «целей» соединения.

# В заключение

После прочтения этой главы вы должны:

- Знать, что `dojo.connect` стандартизует объект события, который передается функции-обработчику и обеспечивает переносимость между платформами
- Понимать, как `dojo.connect` позволяет произвольно связывать события DOM, объекты событий JavaScript и обычные функции для реализации структуры, управляемой событиями
- Уметь использовать средства организации взаимодействий по подписке и создавать каналы связи с гибко связанными компонентами
- Знать, когда в архитектуре приложения предпочтительнее использовать `dojo.connect`, а когда – взаимодействия по подписке.

В следующей главе рассматриваются технология AJAX и организация взаимодействий с сервером.

# 4

# Технология AJAX и взаимодействие с сервером

Основная тема этой главы – организация взаимодействий с сервером. Выполнение асинхронных запросов, использование плавающего фрейма (IFRAME) для отправки содержимого формы, преобразование в/из формы представления объектов JavaScript (JavaScript Object Notation, JSON) и использование формата JSONP (JSON with Padding – JSON с дополнением) – лишь некоторые из тем, которые будут представлены в этой главе. Вы также узнаете о классе `Deferred`, который составляет основу системы ввода-вывода в инструментальном наборе, обеспечивая однородный интерфейс для обслуживания асинхронных действий.

## Краткий обзор AJAX

Появление AJAX[1] (Asynchronous JavaScript and XML – асинхронный JavaScript и XML) вызвало широкий резонанс и оказало благотворное действие на дальнейшее развитие веб-дизайна. Если раньше для существенного обновления содержимого веб-страницы должны были полностью перезагружаться посредством выполнения синхронного запроса, то теперь благодаря JavaScript и объекту `XMLHttpRequest` они могут вести себя практически как традиционные приложения. Аббревиатура XHR представляет собой сокращение от названия объекта `XMLHttp-`

[1] Хотя символ «X» в аббревиатуре AJAX обозначает XML, в настоящее время термин AJAX обычно относится практически к любой архитектуре, которая использует объект `XMLHttpRequest` для выполнения асинхронных запросов независимо от фактического типа данных, возвращаемых этим запросом. С технической точки зрения правильнее было бы использовать обобщающее понятие XHR, но мы будем следовать общепринятой терминологии и использовать название AJAX в более широком понимании.

`Request` и обычно используется при описании любых операций, выполняемых объектом.

Теперь веб-страницы могут получать содержимое от сервера посредством выполнения асинхронных запросов в фоновом режиме, как показано на рис. 4.1, и обрабатывать его в функциях обратного вызова при

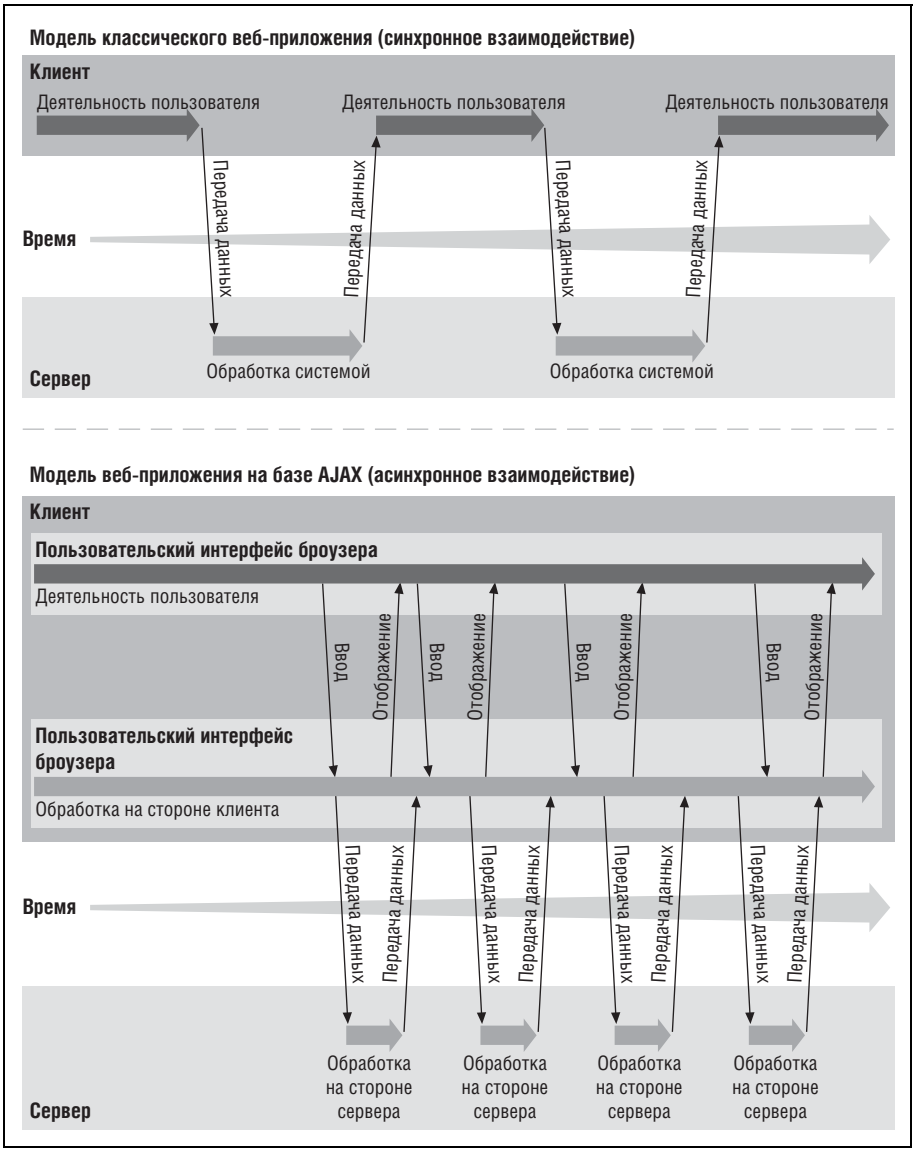



***Рис. 4.1.*** *Различия между синхронными и асинхронными взаимодействиями в веб-приложениях*

получении. (Основой для рис. 4.1 стало изображение, взятое со страницы *http://adaptivepath.com/ideas/essays/archives/000385.php*.) Несмотря на свою простоту, эта концепция существенно изменила представление пользователя и дала начало новой эпохе полнофункциональных Интернет-приложений.

Для непосредственной работы с JavaScript-объектом `XMLHttpRequest` совершенно не требуется обладать незаурядными умственными способностями, но, как и в любом другом деле, здесь часто приходится прибегать к хитростям и использовать шаблонный программный код для решения вполне обычных задач. Например, асинхронные запросы не дают гарантию, что в результате будет возвращено некоторое значение (хотя это происходит в подавляющем большинстве случаев), поэтому, как правило, необходимо предусматривать логику, которая определяла бы, когда истечет предельное время ожидания и как в этом случае реагировать; возможно, вам могут потребоваться некоторые средства для проверки и преобразования строк JSON в объекты JavaScript или вы хотели бы отделить обработку успешных запросов от обработки запросов, завершившихся с признаком ошибки, и т. д.

## JSON

Перед тем как перейти к обсуждениею технологии AJAX, необходимо вкратце рассмотреть формат JSON, потому что он стал практически стандартным форматом для организации обмена простыми данными в AJAX-приложениях. С формальным описанием JSON можно познакомиться по адресу *http://json.org*, но по сути JSON – это просто строковое представление объектов JavaScript. В библиотеке Base имеются две простые функции, выполняющие преобразования объектов JavaScript в строковое представление и обратно. Эти функции берут на себя все хлопоты по экранированию специальных символов, таких как табуляция или перевод строки, и могут даже форматировать получаемый результат, если вам это требуется:

```
dojo.fromJson(/*String*/ json) //Возвращает Object
dojo.toJson(/*Object*/ json, /*Boolean?*/ prettyPrint) //Возвращает String
```

По умолчанию для оформления отступов в строках JSON в режиме форматированного вывода используется символ табуляции. Существует возможность заменить символ табуляции чем-то другим, достаточно лишь поменять значение встроенного атрибута `dojo.toJsonIndentStr`.

Ниже приводится короткий пример, иллюстрирующий процесс преобразования объекта в строку JSON, удобную для восприятия человеком:

```
var o = {a:1, b:2, c:3, d:4};
dojo.toJson(o, true); //форматирование результата
/* получится ...
```

```
'{
  "a": 1,
  "b": 2,
  "c":3,
  "d":4
}'
```

# Работать с AJAX просто

Библиотека Base предоставляет небольшой набор функций для использования в архитектуре RESTful, которые значительно упрощают выполнение операций AJAX. Каждая из этих функций обеспечивает явные механизмы, фактически устраняющие необходимость применения шаблонного программного кода, который раньше приходилось писать. В табл. 4.1 приводится перечень свойств аргумента `args` этих функций.

Ниже приводятся функции поддержки архитектуры RESTful, предназначенные для работы с объектом XHR, которые имеются в инструментальном наборе Dojo версии 1.1. Каждая из этих функций автоматически устанавливает заголовок `X-Requested-With: XMLHttpRequest`. Обсуждение параметра `args` следует ниже.

Все функции XHR возвращают специальный объект `Deferred`, с которым вы познакомитесь поближе в следующем разделе. А пока просто сосредоточьтесь на обсуждении.

### Передача репрезентативного состояния (Representational State Transfer, REST)

Аббревиатура REST происходит от названия «Representational State Transfer» (передача репрезентативного состояния) и описывает архитектурный стиль, который в первую очередь ассоциируется со Всемирной паутиной. Стиль REST в значительной степени ориентирован на ресурсы, и в архитектуре RESTful идентификаторы ресурсов (URI) определяют ресурсы и используются для доступа к ним. Методы HTTP – GET, PUT, POST и DELETE – описывают семантические операции, обычно связанные с действиями над ресурсом. Например, запрос GET по адресу *http://example.com/foo/id/1* подразумевает попытку получить ресурс *foo*, имеющий свойство `id` со значением `1`, а запрос DELETE с тем же адресом URI предполагает, что данный ресурс требуется удалить.

Отличным руководством по архитектуре RESTful может служить книга Леонарда Ричардсона (Leonard Richardson) и Сэма Руби (Sam Ruby) «RESTful Web Services» (O'Reilly).

*Таблица 4.1. Свойства аргумента args*

| Имя | Тип (значение по умолчанию) | Комментарий |
|---|---|---|
| `url` | `String` (`""`) | Базовый URL для передачи запроса. |
| `content` | `Object` (`{}`) | Содержит пары ключ/значение, закодированные в представление, наиболее подходящее для используемого способа передачи. Например, в случае метода GET они сериализуются и добавляются в конец строки запроса в виде пар `name1=value2`, а при использовании метода передачи на основе плавающего фрейма IFRAME они включаются в форму в виде скрытых полей. Следует учитывать, что, хотя протокол HTTP допускает передачу более одного поля с одним и тем же именем (поля с несколькими значениями), это невозможно при использовании свойства `content`, потому что это – ассоциативный массив. |
| `timeout` | `Integer` (`Infinity`) | Предельное время ожидания ответа, в миллисекундах. По истечении этого времени вызывается функция обработки ошибок. Это свойство действует, только когда свойство `sync` имеет значение `false`. |
| `form` | `DOMNode \| String` | Узел DOM или значение атрибута `id` формы, откуда должны быть взяты пары ключ/значение, которые затем будут преобразованы в строку, пригодную для включения в URL в виде строки запроса. (Каждый элемент формы должен иметь определенный атрибут `name`, однозначно идентифицирующий его.) |
| `preventCache` | `Boolean` (`false`) | Если имеет значение `true`, в запрос передается специальный параметр `dojo.preventCache`, значение которого изменяется для каждого запроса (отметка времени). Удобно использовать только в запросах GET. |
| `handleAs` | `String` (`"text"`) | Определяет ожидаемый тип данных ответа, который передается обработчику `load`. Допустимыми типами в зависимости от используемого способа передачи данных являются: `"text"`, `"json"`, `"javascript"` и `"xml"`. |
| `load` | `Function` | Функция, которая будет вызвана в случае успешного получения ответа. Должна иметь сигнатуру `function(response, ioArgs) {/*...*/}`. |
| `error` | `Function` | Функция, которая будет вызвана в случае ошибки. Должна иметь сигнатуру `function(response, ioArgs) {/*...*/}`. |

| Имя | Тип (значение по умолчанию) | Комментарий |
|---|---|---|
| handle | Function | Функция, которая должна вызываться как в случае успеха, так и в случае ошибки, то есть это функция, которая должна вызываться независимо от того, успешно ли выполнился запрос. |
| sync | Boolean (false) | Определяет, будет ли выполняться запрос в синхронном режиме. |
| headers | Object ({}) | Дополнительные заголовки HTTP, включаемые в запрос. |
| postData | String ("") | Данные для передачи в теле запроса POST. Имеет смысл только при использовании функции rawXhrPost. |
| putData | String ("") | Данные для передачи в теле запроса PUT. Имеет смысл только при использовании функции rawXhrPut. |

`dojo.xhrGet(/*Object*/args)`

С помощью XHR выполняет запрос GET.

`dojo.xhrPost(/*Object*/args)`

С помощью XHR выполняет запрос POST.

`dojo.rawXhrPost(/*Object*/args)`

С помощью XHR выполняет запрос POST и позволяет передавать необработанные данные для включения в тело запроса POST.

`dojo.xhrPut(/*Object*/args)`

С помощью XHR выполняет запрос PUT.

`dojo.rawXhrPut(/*Object*/args)`

С помощью XHR выполняет запрос PUT и позволяет передавать необработанные данные для включения в тело запроса PUT.

`dojo.xhrDelete(/*Object*/args)`

С помощью XHR выполняет запрос DELETE.

`dojo.xhr(/*String*/ method, /*Object*/ args, /*Boolean?*/ hasBody)`

Многоцелевая функция для работы с объектом XHR, которая позволяет выбирать метод HTTP для выполнения асинхронного запроса.

Большинство пунктов в таблице достаточно понятны, тем не менее следует особо упомянуть аргументы, которые передаются функциям load и error. Первый параметр, response, представляет то, что вернул сервер, а значение свойства handleAs указывает, как следует интерпретировать ответ. По умолчанию используется значение "text", однако, если, например, указать значение "json", это приведет к тому, что ответ будет

преобразован в объект JavaScript, вследствие чего его можно будет интерпретировать как свойство аргумента response.

Функции load и error всегда должны возвращать значение response. Как вы узнаете далее в этой главе, все функции ввода/вывода, такие как средства для работы с объектом XHR, имеют возвращаемое значение типа Deferred, а возврат значения response позволяет составлять цепочки из функций обратного вызова и обработчиков ошибок, что является важным аспектом взаимодействия с объектами Deferred.

Второй параметр, ioArgs, содержит некоторую информацию об аргументах, переданных серверу при выполнении запроса. Вам нечасто придется использовать параметр ioArgs, но иногда он может оказаться полезным, особенно во время отладки. В табл. 4.2 приводится описание значений, которые вы можете увидеть в ioArgs.

*Таблица 4.2. Свойства аргумента ioArgs*

| Имя | Тип | Комментарий |
|---|---|---|
| args | Object | Оригинальный аргумент функции ввода/вывода. |
| xhr | XMLHttpRequest | Фактический объект XMLHttpRequest, использовавшийся для выполнения запроса. |
| url | String | Окончательный адрес URL, использованный для выполнения запроса. Часто отличается от первоначального, потому что дополняется параметрами запроса и тому подобным. |
| query | String | Определено только для запросов, отличных от GET. Это свойство содержит строку параметров запроса, которая была передана вместе с запросом. |
| handleAs | String | Определяет, как следует интерпретировать ответ. |

## Примеры применения функций XHR

Как минимум, в числе аргументов для выполнения запросов XHR должны быть определены адрес URL и функция load. Однако обычно, и это *очень правильно*, в число аргументов включается еще и обработчик ошибок – не опускайте его, если вы действительно не уверены, что можете обойтись без него. Например:

```
//...обрезано...
dojo.addOnLoad(function() {
    dojo.xhrGet({

        url : "someText.html", //относительный URL

        // Вызвать эту функцию в случае успеха
        load : function(response, ioArgs) {
            console.log("successful xhrGet", response, ioArgs);
```

```
            //Заполнить некоторый элемент содержимым...
            dojo.byId("foo").innerHTML= response;

            return response; //всегда возвращайте аргумент response
        },

        // Вызвать эту функцию в случае неудачи
        error : function(response, ioArgs) {
            console.log("failed xhrGet", response, ioArgs);

            /* обработка ошибки... */

            return response; //всегда возвращайте аргумент response
        }
    });
});
//...обрезано...
```

Совсем необязательно, что вам всегда будет нужен текст в качестве ответа. Иногда может появиться потребность ограничить время ожидания ответа или передать дополнительную информацию в виде строки запроса. К счастью, это не сильно усложнит вашу жизнь. Просто добавьте несколько параметров, как показано ниже:

```
dojo.xhrGet({
    url : "someJSON.html", //Нечто такое: {'bar':'baz'}

    handleAs : "json", //Преобразовать в объект JavaScript

    timeout: 5000, //Вызвать обработчик ошибок через 5 секунд,
                   //если за это время ответ не будет получен

    content: {foo:'bar'}, //Добавить foo=bar в строку запроса

    // Вызвать эту функцию в случае успеха
    load : function(response, ioArgs) {
        console.log("successful xhrGet", request, ioArgs);
        console.log(response);

        //Значение свойства handleAs предписывает Dojo
        //преобразовать данные в объект
        dojo.byId("foo").innerHTML= response.bar;
        //Отобразит обновленные данные, например 'baz'

        return response; //всегда возвращайте аргумент response
    },

    // Вызвать эту функцию в случае неудачи
    error : function(response, ioArgs) {
        console.log("failed xhrGet");
        return response; //всегда возвращайте аргумент response
    }
});
```

Обратите внимание, что отсутствие правильного определения значения для свойства `handleAs` может приводить к появлению ошибок, поиск которых может оказаться непростым делом. Например, если случайно

забыть определить параметр handleAs и попытаться в функции load обратиться к значению ответа как к объекту JavaScript, вы наверняка получите ошибку, которая может увести вас далеко, прежде чем вы заметите, что пытаетесь интерпретировать значение типа String как значение типа Object, потому что в тексте сообщения об ошибке значения строк и объектов могут выглядеть практически идентично.

В большинстве приложений выполняются запросы GET, однако у вас могут сложиться обстоятельства, обязывающие использовать запрос типа PUT, POST или DELETE. Процедура выполнения таких запросов остается прежней, только при этом необходимо включить требуемые данные в свойство putData или postData и вызвать функцию rawXhrPut или rawXhrPost, соответственно, как средство передачи данных серверу. Ниже приводится пример использования функции rawXhrPost:

```
dojo.rawXhrPost({
    url : "/place/to/post/some/raw/data",
    postData : "{foo : 'bar'}", //литерал  JSON
    handleAs : "json",

    load : function(response, ioArgs) {
        /* Здесь выполняются всякие интересные действия */
        return response;
    },

    error : function(response, ioArgs) {
        /* Получше обрабатывайте ошибки */
        return response;
    }
});
```

## Многоцелевая функция выполнения запросов через XMLHttpRequest

В Dojo версии 1.1 появилась более универсальная функция dojo.xhr со следующей сигнатурой:

```
dojo.xhr(/*String*/ method, /*Object*/ args, /*Boolean?*/ hasBody)
```

Как оказывается, каждая из функций, работающих с XHR, представленных в этой главе, фактически является оберткой вокруг этой функции. Например, dojo.xhrGet в действительности реализована следующим образом:

```
dojo.xhrGet = function(args) {
  return dojo.xhr("GET", args); //Имя метода всегда должно записываться
                                //заглавными символами!
}
```

В большинстве случаев на практике используются функции-обертки, представленные в этом разделе, тем не менее в некоторых ситуациях удобнее использовать многоцелевую функцию dojo.xhr: когда необхо-

димо программно настраивать объект XHR или когда нужная функция-обертка отсутствует. Например, следующий фрагмент выполняет запрос HEAD, для которого не предусмотрена функция-обертка:

```
dojo.xhr("HEAD", {
    url : "/foo/bar/baz",
    load : function(response, ioArgs) { /*...*/},
    error : function(response, ioArgs) { /*...*/}
});
```

## Управление контекстом функций обратного вызова

В главе 2 была представлена функция hitch, которая может использоваться, чтобы обеспечить выполнение некоторой функции в определенном контексте. Довольно часто функция hitch используется совместно с объектом XHR, потому что иногда требуется, чтобы контекст функции обратного вызова отличался от контекста блока программного кода, в котором она вызывается. Следующий фрагмент демонстрирует один из случаев применения функции hitch и представляет собой типичный шаблон использования псевдонима для ссылки this, чтобы решить проблему контекста в функции обратного вызова:

```
//Предположим, что имеется следующий блок addOnLoad block,
//который фактически может быть любым объектом JavaScript
dojo.addOnLoad(function() {

    //объект foo связан с контекстом этой анонимной функции
    this.foo = "bar";

    //создать псевдоним для "this", чтобы на него можно было
    //ссылаться внутри функции load...
    var self=this;
    dojo.xhrGet({
        url : "./data",
        load : function(response, ioArgs) {
            //для обращения к объекту foo внутри этой функции
            //следует использовать псевдоним "this"...
            console.log(self.foo, response);
        },
        error : function(response, ioArgs) {
            console.log("error", response, ioArgs);
        }
    });
});
```

Хотя в этом коротком примере все выглядит достаточно понятным, но программный код, в котором для работы многократно создавались новые объекты – псевдонимы контекста this, будет выглядеть беспорядочным. В следующий раз, когда вам потребуется создать псевдоним для this, подумайте о применении следующего шаблона, в котором используется функция hitch:

```
dojo.addOnLoad(function() {

    //объект foo связан с контекстом этой анонимной функции
    this.foo = "bar";

    //связать контекст функции обратного вызова с текущим контекстом,
    //чтобы иметь возможность обращаться к объекту foo
    var callback = dojo.hitch(this, function(response, ioArgs) {
        console.log("foo (in context) is", this.foo);
        //и в вашем распоряжении все еще имеются response и ioArgs...
    });

    dojo.xhrGet({
        url : "./data",
        load : callback,
        error : function(response, ioArgs) {
            console.log("error", response, ioArgs);
        }
    });
});
```

И не забывайте, что функция `hitch` может принимать дополнительные аргументы, благодаря чему вы легко и просто можете передавать ей любые параметры, которые должны быть доступны в функции обратного вызова, например так:

```
dojo.addOnLoad(function() {

    //объект foo связан с контекстом этой анонимной функции
    this.foo = "bar";

    //связать контекст функции обратного вызова с текущим контекстом,
    //чтобы иметь возможность обращаться к объекту foo
    var callback = dojo.hitch(
        this,
        function(extraParam1, extraParam2, response, ioArgs) {
            console.log("foo (in context) is", this.foo);
            //и в вашем распоряжении все еще имеются response и ioArgs...
        },
        "extra", "params"
    );

    dojo.xhrGet({
        url : "./data",
        load : callback,
        error : function(response, ioArgs) {
            console.log("error", response, ioArgs);
        }
    });
});
```

Если число параметров может изменяться, вы можете использовать для доступа к ним свойство `arguments`, не забывая, что два последних значения – это аргументы `response` и `ioArgs`.

# Объекты Deferred

В настоящее время JavaScript не предусматривает поддержку концепции потоков выполнения, но существует возможность асинхронного выполнения запросов `XMLHttpRequest` с задержкой – с помощью функции `setTimeout`. Однако, чтобы запутаться в таком программном коде, потребуется оформить совсем немного асинхронных вызовов. Библиотека Base предоставляет класс с именем `Deferred`, который поможет справиться со сложными ситуациями, часто связанными с рутинными операциями реализации асинхронных событий. Подобно другим абстракциям, объекты `Deferred` позволяют прятать подробности реализации и/или шаблонный программный код за аккуратным и непротиворечивым интерфейсом.

Если попытаться описать ценность `Deferred` в одном предложении, оно могло бы звучать примерно так: эти объекты позволяют обрабатывать все операции сетевого ввода-вывода одинаковым способом независимо от того, являются ли они синхронными или асинхронными. Даже если объект `Deferred` уже запущен, потерпел неудачу или операция завершилась успехом, процедура составления цепочек функций обратного вызова и функций обработки ошибок остается неизменной. Как можно догадаться, такое поведение существенно упрощает работу.

Реализация объекта `Deferred` в Dojo является адаптированной версией реализации из библиотеки MochiKit, которая в свою очередь заимствовала реализацию из библиотеки Twisted. Неплохое описание реализации в библиотеке MochiKit можно найти по адресу: *http://www.mochikit.com/doc/html/MochiKit/Async.html#fn-deferred*. Описание реализации в библиотеке Twisted можно найти по адресу: *http://twistedmatrix.com/projects/core/documentation/howto/defer.html*.

Из ключевых особенностей объектов `Deferred` можно упомянуть возможность составлять цепочки из множества функций обратного вызова и функций обработки ошибок так, что они будут вызываться в заранее заданном порядке, а также возможность передавать объекту `Deferred` процедуру отмены, с помощью которой можно безопасно прерывать выполнение асинхронных запросов. Возможно, вы еще не заметили, но все функции XHR, представленные ранее в этой главе, возвращают объект `Deferred`, просто раньше у нас не было необходимости заниматься его изучением. В действительности все сетевые операции ввода-вывода в инструментальном наборе используют и возвращают объекты `Deferred`, потому что эти объекты обеспечивают высокую гибкость управления асинхронными операциями, которые являются результатом выполнения сетевых запросов.

Прежде чем вернуться к каким-то из предыдущих попыток использования XHR, взгляните на следующий абстрактный пример, демонст-

рирующий применение объекта Deferred и составляющий основу некоторых концепций, которые нам предстоит рассматривать:

```
//Создать объект Deferred
var d = new dojo.Deferred(/* Здесь можно указать функцию отмены */);

//Добавить функцию обратного вызова
d.addCallback(function(response) {
    console.log("The answer is", response);
    return response;
});

//Добавить еще одну функцию обратного вызова,
//которая будет запускаться после предыдущей
d.addCallback(function(response) {
    console.log("Yes, indeed. The answer is", response);
    return response;
});

//Добавить функцию обработки ошибок на тот случай, если что-то пойдет не так
d.addErrback(function(response) {
    console.log("An error occurred", response);
    return response;
});

//Можно добавить еще столько функций обратного вызова
//и обработки ошибок, сколько потребуется

/* Некоторый программный код, выполняющий вычисления */

//Где-то в другом месте запускается цепочка функций обратного вызова
d.callback(46);
```

Если запустить этот пример в Firebug, можно увидеть следующий вывод:

```
The answer is 46
Yes, indeed. The answer is 46
```

Прежде чем перейти к более интересным примерам, наверное, имеет смысл ознакомиться с прикладным интерфейсом объекта Deferred (табл. 4.3).

Объект Deferred может переходить в состояние ошибки в любом из следующих трех случаев:

- Функция обратного вызова или функция обработки ошибок получила аргумент, который является объектом Error.
- Функция обратного вызова или функция обработки ошибок возбудила исключение.
- Функция обратного вызова или функция обработки ошибок вернула значение, которое является объектом Error.

*Таблица 4.3. Методы и свойства объекта Deferred*

| Имя | Тип возвращаемого значения | Комментарий |
|---|---|---|
| `addCallback(/*Function*/handler)` | `Deferred` | Добавляет функцию обратного вызова в цепочку обработчиков, вызываемых в случае успешного выполнения операции. |
| `addErrback(/*Function*/handler)` | `Deferred` | Добавляет функцию обратного вызова в цепочку обработчиков, вызываемых в случае ошибки. |
| `addBoth(/*Function\|Object*/ context, /*String?*/name)` | `Deferred` | Добавляет функцию обратного вызова, которая действует и как обработчик ситуации успешного завершения операции, и как обработчик ошибок. Удобно для добавления программного кода, который должен гарантированно вызываться в любом случае. |
| `addCallbacks(/*Function*/callback, /*Function*/errback)` | `Deferred` | Позволяет одновременно добавлять функцию обратного вызова и функцию обработки ошибок. |
| `callback(/*Any*/value)` | нет | Запускает цепочку функций обратного вызова. |
| `errback(/*Any*/value)` | нет | Запускает цепочку функций обработки ошибок. |
| `cancel()` | нет | Отменяет запрос и вызывает функцию отмены, которая была передана конструктору. |

В наиболее типичных случаях обычно не используются свойства `canceller`, `silentlyCancelled` и `fired` объекта `Deferred`, которые, соответственно, представляют собой ссылку на функцию отмены; признак, что операция была отменена, но в объекте `Deferred` не была зарегистрирована функция отмены; и признак состояния объекта `Deferred`. Свойство `fired` может иметь следующие значения:

- -1: Еще нет результата (начальное состояние)
- 0: Успешно выполнена цепочка функций обратного вызова
- 1: Произошла ошибка

## Исследование объекта Deferred с помощью CherryPy

Давайте для начала подготовим простую процедуру на стороне сервера, которая выполняет короткую задержку, а затем возвращает некоторое содержимое. (Пауза – это лишь способ подчеркнуть асинхронность поведения.)

Ниже следует полное содержимое файла, использующего модуль CherryPy и реализующего упомянутую функциональность:

```
import cherrypy
from time import sleep
import os

# программный код, выполняющий запросы XHR, будет находиться
# в файле foo.html, и это видно в следующей директиве config

current_dir = os.getcwd()
config = {'/foo.html' :
    {
    'tools.staticfile.on' : True,
    'tools.staticfile.filename' : os.path.join(current_dir, 'foo.html')
    }
}
```

### Политика одного источника

Будет полезно отметить, что нам пришлось выполнить дополнительные действия по настройке модуля CherryPy, чтобы он мог поставлять нам статический файл, из которого уже можно будет выполнять запросы XHR. Это было сделано, потому что объект `XMLHttpRequest`, предоставляемый JavaScript, не позволит нам выполнять межсайтовый скриптинг (cross-site scripting) из соображений безопасности. То есть мы не сможем открыть в броузере локальный файл, такой как *file://foo.html*, и использовать функцию `dojo.xhrGet` для выполнения запросов по адресу *http://127.0.0.1:8080/*. Да, оба адреса URL соответствуют одному и тому же компьютеру, но домены их отличаются. Как будет показано в следующем разделе, для преодоления проблемы безопасности и обеспечения возможности загрузки содержимого из других доменов может использоваться прием, известный как JSONP, который обеспечивает возможность создания комбинированных приложений. Среди других подходов, часто применяемых для загрузки содержимого из других доменов, можно назвать использование сокетов через расширения, основанные на технологиях Flash или ActiveX. В любом случае имейте ввиду, что запуск непроверенного программного кода в своем домене представляет определенный риск, к которому не стоит относиться легкомысленно.

```
class Content:

    # именно этот класс возвращает содержимое по запросу
    @cherrypy.expose
    def index(self):
        sleep(3) # выдержать паузу в 3 сек. перед ответом
        return "Hello"

# запустить веб-сервер, который будет прослушивать порт 8080
cherrypy.quickstart(Content(), '/', config=config)
```

Предположим, что предыдущий программный код, использующий CherryPy, сохранен в файле с именем *hello.py*. Тогда, чтобы запустить сервер, достаточно в окне терминала выполнить команду `python.hello.py`. Чтобы убедиться, что сервер запущен, можно просто в броузере перейти по адресу *http://127.0.0.1:8080/*, после чего с короткой задержкой на экране должна появиться надпись «Hello».

## Использование объектов Deferred, возвращаемых функциями XHR

После запуска CherryPy и сохранения следующего ниже кода разметки в файле с именем *foo.html* в том же каталоге, где находится файл *hello.py*, все будет готово к испытаниям. Перейдите в броузере по адресу *http://127.0.0.1:8080/foo.html*, и страница *foo.html* должна загрузиться без каких-либо проблем:

```
<html>
    <head>
        <title>Fun with Deferreds!</title>

        <script type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js">
        </script>

        <script type="text/javascript">
            dojo.addOnLoad(function() {

                //Вызвать функцию, запускающую асинхронный запрос,
                //которая вернет объект Deferred
                var d = dojo.xhrGet({
                    url: "http://localhost:8080",
                    timeout : 5000,
                    load : function(response, ioArgs) {
                        console.log("Load response is:", response);
                        console.log("Executing the callback chain now...");
                        return response;
                    },
                    error : function(response, ioArgs) {
                        console.log("Error!", response);
                        console.log("Executing the errback chain now...");
                        return response;
                    }
                });
```

```
console.log("xhrGet fired. Waiting on callbacks or errbacks");

 //Добавить несколько функций обратного вызова
 d.addCallback(
     function(result) {
         console.log("Callback 1 says that the result is ",
                     result);
         return result;
     }
 );

 d.addCallback(
     function (result) {
         console.log("Callback 2 says that the result is ",
                     result);
         return result;
     }
 );

 //Добавить несколько обработчиков ошибок
 d.addErrback(
     function(result) {
         console.log("Errback 1 says that the result is ",
                     result);
         return result;
     }
 );

 d.addErrback(
     function(result) {
         console.log("Errback 2 says that the result is ",
                     result);
         return result;
     }
 );
});
    </script>
</head>
<body>
    Check the Firebug console.
</body>
</html>
```

После запуска этого примера вы должны увидеть в консоли Firebug следующий вывод:

```
xhrGet fired. Waiting on callbacks or errbacks
Load response is: Hello
Executing the callback chain now...
Callback 1 says that the result is Hello
Callback 2 says that the result is Hello
```

Главный вывод, который следует из этого примера, заключается в том, что объект `Deferred` дает ясный, непротиворечивый интерфейс, позволяющий обрабатывать все ситуации, складывающиеся в результате

работы функции `xhrGet`, – будь то успешное получение ответа на запрос или ошибка, которую надо обработать.

Вы можете попробовать изменить значение предельного времени ожидания в вызове функции `dojo.xhrGet` так, чтобы это время оказалось меньше трех секунд, необходимых серверу для передачи ответа. В результате это приведет к появлению ошибки, и вы сможете увидеть, как запускается цепочка функций обработки ошибок. Эта цепочка запускается еще и тогда, когда что-то пошло не так как надо в одной из функций обратного вызова, поэтому вы можете попробовать внести в функцию обратного вызова программный код, генерирующий ошибку, чтобы увидеть, что перед вызовом цепочки функций обработки ошибок успевает отработать часть цепочки функций обратного вызова.

Не забывайте о необходимости возвращать значение, переданное в функцию обратного вызова и в функцию обработки ошибок, чтобы вся цепочка могла принять участие в обработке результата. Неосторожность, проявленная в этом случае, может вызывать странные результаты, потому что это помешает выполнению цепочки функций обратного вызова или обработки ошибок. Теперь вы знаете, почему так важно не забывать возвращать аргумент `response` в обработчиках `load` и `error` функций XHR.

На рис. 4.2 показано, как протекают события с участием объекта `Deferred`. Одна из ключевых особенностей, о которых следует помнить, состоит в том, что объекты `Deferred` действуют как цепочки.

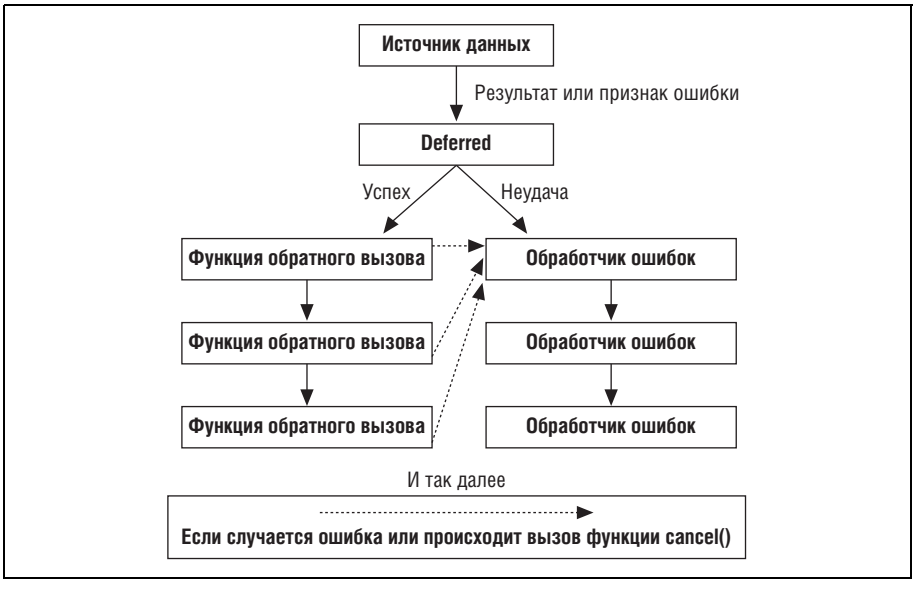



***Рис. 4.2.*** *Основной поток событий, протекающих с участием объекта Deferred*

## Внедрение объектов Deferred в функции XHR

Еще одна замечательная особенность объекта Deferred состоит в том, что он обеспечивает очевидный способ отмены асинхронного действия до его полного завершения. Ниже приводится доработанная версия нашего предыдущего примера, демонстрирующая возможность отменить уже выполняющийся запрос, а также – «внедрение» объекта Deferred в обработчики `load` и `error`:

```
<html>
    <head>
        <title>Fun with Deferreds!</title>

        <script type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js">
        </script>

        <script type="text/javascript">
            dojo.addOnLoad(function() {
                var d = new dojo.Deferred;

                //Добавить несколько функций обратного вызова
                d.addCallback(
                    function(result) {
                        console.log("Callback 1 says that the result is ",
                                    result);
                        return result;
                    }
                );

                d.addCallback(
                    function (result) {
                        console.log("Callback 2 says that the result is ",
                                    result);
                        return result;
                    }
                );

                //Добавить несколько функций обработки ошибок
                d.addErrback(
                    function(result) {
                        console.log("Errback 1 says that the result is ",
                                    result);
                        return result;
                    }
                );

                d.addErrback(
                    function(result) {
                        console.log("Errback 2 says that the result is ",
                                    result);
                        return result;
                    }
                );
```

```
                //Вызвать функцию, запускающую асинхронный запрос,
                //которая вернет объект Deferred
                request = dojo.xhrGet({
                    url: "http://localhost:8080",
                    timeout : 5000,
                    load : function(response, ioArgs) {
                        console.log("Load response is:", response);
                        console.log("Executing the callback chain now...");

                        //внедрить нашу цепочку функций обратного вызова
                        d.callback(response, ioArgs);

                        //позволить продолжить обработку цепочке объекта
                        //Deferred функции xhrGet...
                        return response;
                    },
                    error : function(response, ioArgs) {
                        console.log("Error!", response);
                        console.log("Executing the errback chain now...");

                        //внедрить нашу цепочку обработчиков ошибок
                        d.errback(response, ioArgs);

                        //позволить продолжить обработку цепочке объекта
                        //Deferred функции xhrGet...
                        return response;
                    }
                });
            });
        </script>
    </head>
    <body>
        XHR request in progress. You have about 3 seconds to cancel it.
        <button onclick="javascript:request.cancel()">Cancel</button>
    </body>
</html>
```

Если запустить этот пример, в консоли Firebug можно наблюдать следующий вывод:

```
xhrGet just fired. Waiting on callbacks or errbacks now...
Load response is: Hello
Executing the callback chain now...
Callback 1 says that the result is Hello
Callback 2 says that the result is Hello
```

А щелчок на кнопке «Cancel» (отмена) приведет к появлению следующих результатов:

```
xhrGet just fired. Waiting on callbacks or errbacks now...
Press the button to cancel...
Error: xhr cancelled dojoType=cancel message=xhr cancelleddojo.xd.js (line 20)
Error! Error: xhr cancelled dojoType=cancel message=xhr cancelled
Executing the errback chain now...
```

```
Errback 1 says that the result is Error: xhr cancelled dojoType=cancel
message=xhr
cancelled
Errback 2 says that the result is Error: xhr cancelled dojoType=cancel
message=xhr
cancelled
```

## Собственная функция отмены

Любая функция XHR обладает специальной функцией отмены, которая вызывается при обращении к функции `cancel()`, но для собственных объектов `Deferred` можно создавать индивидуальные функции отмены, как показано ниже:

```
var canceller = function() {
    console.log("custom canceller...");
    //Если не вернуть собственный объект Error, по умолчанию будет
    //возвращен объект Error с сообщением "Deferred Cancelled"
}
var d = new dojo.Deferred(canceller); //передать функцию отмены в конструктор
/* ....здесь происходит нечто интересное...*/
d.cancel(); // обработчики ошибок должны быть готовы особым образом
            // обработать объект Error с сообщением "Deferred Cancelled"
```

## DeferredList

Объект `Deferred` является частью библиотеки Base, а библиотека Core обеспечивает дополнительное приложение `DeferredList`, которое предоставляет средства управления множеством объектов `Deferred`. Среди типичных случаев применения `DeferredList` можно назвать:

- Когда необходимо запустить свою функцию обратного вызова или цепочку, после того как отработают все функции обратного вызова объектов `Deferred` в списке
- Когда необходимо запустить свою функцию обратного вызова или цепочку, если отработает хотя бы одна функция обратного вызова, принадлежащая какому-либо из объектов `Deferred` в списке
- Когда необходимо запустить свою функцию обработки ошибок или цепочку, если отработает хотя бы одна функция обработки ошибок, принадлежащая какому-либо из объектов `Deferred` в списке

Ниже приводится сигнатура функции `DeferredList`:

```
dojo.DeferredList(/*Array*/list, /*Boolean?*/fireOnOneCallback, /*Boolean?*/
    fireOnOneErrback, /*Boolean?*/consumeErrors, /*Function?*/canceller)
```

Сигнатура достаточно очевидна: при вызове конструктора достаточно передать ему единственный аргумент – массив объектов Deferred. По умолчанию объект `DeferredList` запускает свою цепочку функций обратного вызова, только когда отработают цепочки функций обратного вызова всех объектов `Deferred`. Передавая логические аргументы, можно обеспечить запуск цепочек функций обратного вызова и функций

обработки ошибок, если отработала хотя бы одна функция обратного вызова или функция обработки ошибок, соответственно.

Если в аргументе consumeErrors передать значение true, то ошибки будут поглощаться объектом DeferredList, что может быть удобно, когда нежелательно воспроизводить ошибки, порожденные отдельными объектами Deferred в списке. Аргумент canceller обеспечивает возможность передать свою собственную функцию отмены точно так же, как и в обычном объекте Deferred.

# Утилиты для работы с формами и HTTP

Некоторые решения, основанные на технологии AJAX, при правильной реализации могут производить сильное впечатление, но не будем забывать, что некоторые проверенные элементы, такие как формы HTML, еще совсем не устарели и по-прежнему играют важную роль во многих современных решениях, как с применением технологии AJAX, так и без нее. Библиотека Base предоставляет три функции преобразования форм:

```
dojo.formToObject(/*DOMNode||String*/ formNode) //Возвращает Object
dojo.formToQuery(/*DOMNode||String*/ formNode) //Возвращает String
dojo.formToJson(/*DOMNode||String*/ formNode) //Возвращает String
```

Чтобы продемонстрировать действие каждой из этих функций, предположим, что у нас имеется следующая форма:

```
<form id="register">

    <input type="text" name="first" value="Foo">
    <input type="button" name="middle" value="Baz" disabled>
    <input type="text" name="last" value="Bar">

    <select type="select" multiple name="favorites" size="5">
        <option value="red">red</option>
        <option value="green" selected>green</option>
        <option value="blue" selected>blue</option>
    </select>

</form>
```

Ниже приводятся результаты вызова каждой функции. Обратите внимание, что неактивные элементы формы были пропущены при преобразовании:

Функция formToObject вернула:

```
{
    first: "Foo",
    last : "Bar",
    favorites: [
        "green",
        "blue"
    ]
};
```

Функция `formToQuery` вернула:

```
"first=Foo&last=Bar&favorites=green&favorites=blue"
```

Функция `formToJson` вернула:

```
'{"first": "Foo", "last": "Bar", "favorites": ["green", "blue"]}'
```

Библиотека Base предоставляет следующие вспомогательные функции, позволяющие выполнять преобразование строки запроса в объект и обратно. Они достаточно просты в применении, но не забывайте, что *значения в строке запроса преобразуются в строки, даже если они фактически являются числами*:

```
dojo.queryToObject(/*String*/ str) //Возвращает Object
dojo.objectToQuery(/*Object*/ map) //Возвращает String
```

Ниже приводится короткий фрагмент, иллюстрирующий применение этих функций:

```
//вернет объект {foo : "1", bar : "2", baz : "3"}
var o = dojo.queryToObject("foo=1&bar=2&baz=3");

//преобразует объект обратно в строку "foo=1&bar=2&baz=3"
dojo.objectToQuery(o);
```

# Межсайтовый скриптинг с использованием JSONP

Хотя объект `XMLHttpRequest` в рамках обеспечения политики одного источника не позволяет загружать данные из-за пределов домена текущей страницы, тем не менее оказывается, что тег `SCRIPT` не является субъектом этой политики. В результате появился неофициальный стандарт, известный под названием JSONP, позволяющий загружать данные из других доменов. Как вы уже наверняка догадались, именно эта особенность позволяет веб-приложениям[1] *комбинировать* данные из разных источников и представлять их в одном приложении.

## Пример использования JSONP

Как и все прочее, поначалу работа с JSONP кажется немного загадочной, но стоит только понять принцип действия, и все сразу будет выглядеть очень просто. Чтобы разобраться с этой концепцией, представим, что в странице, загруженной с адреса *http://oreilly.com/*, динамически создается тег `SCRIPT`, который добавляется в раздел `HEAD`. С источником этого тега происходит интересная метаморфоза: он свободно

[1] Без установки дополнительных расширений броузера JSONP – единственное средство загружать данные из других доменов. Расширения, подключаемые непосредственно к броузеру, такие как Flash и компоненты ActiveX, имеют собственные возможности преодоления ограничений политики «одного источника».

может загружать данные не только из домена *oreilly.com*, но из любого другого, например из *http://example.com?id=23*. Реализовать такую операцию на JavaScript достаточно просто:

```
e = document.createElement("SCRIPT");
e.src="http://example.com?id=23";
e.type="text/javascript";
document.getElementsByTagName("HEAD")[0].appendChild(e);
```

Обычно тег SCRIPT подразумевает загрузку фактического сценария, но в действительности можно организовать получение любого информационного наполнения, включая и объекты JSON. Однако здесь существует одна проблема – эти объекты можно применять только к разделу HEAD страницы, где невозможно реализовать что-нибудь интересное (разве что испортить внешний вид страницы).

Например, в результате применения этого приема можно получить страницу, как показано ниже, где жирным шрифтом выделен текст, выданный в результате выполнения предыдущего фрагмента JavaScript, который динамически добавляет тег SCRIPT в раздел HEAD:

```
<html>
    <head>
        <title>My Page</title>
        <script type="text/javascript" >
            {foo : "bar"}
        </script>
    </head>
    <body>
        Содержимое страницы.
    </body>
</html>
```

От добавления литерала объекта JavaScript в раздел HEAD мало проку, но представьте, что могло бы произойти, если бы можно было получить данные в формате JSON, содержащие вызов какой-нибудь функции, точнее функции, которая уже определена где-то в теле страницы. Это могло бы способствовать достижению удивительных эффектов, потому что в этом случае можно было бы асинхронно запрашивать внешние данные и сразу же передавать их выбранной функции для обработки. Для достижения этого эффекта все, что необходимо, – это вставить тег SCRIPT, возвращающий не просто данные в формате JSON, такие как `{foo : "bar"}`, а данные, *дополненные* вызовом функции, например `myCallback({foo : "bar"})`. Если предположить, что функция `myCallback` уже определена, то, когда тег SCRIPT завершит загрузку данных, он вызовет функцию, которой передаст данные в виде параметра, фактически воспользовавшись ею как функцией обратного вызова. (Если что-то осталось непонятным, стоит задержаться немного и представить себе, как происходит весь этот процесс.)

Но здесь остается еще одна маленькая проблема: как получить объект JSON, содержащий дополнение в виде вызова функции? Довольно просто: добрые дяди из *example.com* должны предоставить вам возможность указывать дополнительный параметр в строке запроса, позволяющий вам определять имя функции, вызов которой должен быть обернут в объект JSON. Допустим, что в соответствии с их решением, чтобы вызвать свою функцию, вы должны предоставить в строке запроса дополнительный параметр `c` (в котором указать имя своей функции); тогда обращение по адресу *http://example.com?id=23&c=myCallback* должно вернуть `myCallback({foo : "bar"})`. И это все, что требовалось сделать.

# Модуль IO библиотеки Core

В этом разделе рассматриваются возможности модуля `dojo.io`, входящего в состав библиотеки Core. Центральными темами для обсуждения будут – внедрение динамических тегов SCRIPT с целью получения JSON с дополнением и превращение `IFRAME` в надежное транспортное средство.

## Использование JSONP вместе с Dojo

К этому моменту вы уже обладаете достаточным объемом знаний о Dojo, чтобы не удивляться тому, что инструментальный набор упрощает работу, связанную с применением приемов JSONP. Для достижения того же эффекта, что был продемонстрирован в предыдущем примере, можно было бы использовать функцию `dojo.io.script.get`, которая принимает те же параметры, что и различные методы XHR. Единственное, что следует отметить: параметр `handleAs` в действительности не применим к JSONP, и обязательно следует передать параметр `callbackParamName`, чтобы инструментарий Dojo мог передавать имя функции обратного вызова для организации ее вызова от вашего имени.

Ниже демонстрируется, как это можно реализовать на практике:

```
//ресурс dojo.io.script не является частью Base, поэтому не забывайте
//явно подключать его к странице
dojo.require("dojo.io.script");

dojo.io.script.get({
    callbackParamName : "c", //определяется службой jsonp
    url: "http://example.com?id=23",
    load : function(response, ioArgs) {
        console.log(response);
        return response;
    },
    error : function(response, ioArgs) {
        console.log(response);
        return response;
    }
});
```

Следует пояснить, что callbackParamName задает имя параметра строки запроса, который определяется компанией *example.com*. *Это не имя функции, которую вы определяете как функцию обратного вызова*. Для этих целей Dojo создает временную функцию, с помощью которой передает полученные результаты в функцию load, следуя при этом соглашениям, принятым для функций XHR. Поэтому просто позвольте инструментарию Dojo самому разбираться с дополнением и обрабатывайте полученные результаты в функции load.

Если параметр callbackParamName не будет указан вообще или будет определен неправильно, вы получите сообщение об ошибке "<some callback function> does not exist" (некоторая функция обратного вызова не существует), потому что в динамическом теге SCRIPT будет предпринята попытка вызвать несуществующую функцию.

## Подключение к источнику данных Flickr

Следующий пример иллюстрирует выполнение обращения JSONP к источнику данных Flickr. Попробуйте выполнить этот фрагмент в Firebug, чтобы увидеть, что происходит на самом деле. Кроме того, с помощью этого примера будет весьма поучительно исследовать ошибку, которая возникает, если опустить параметр callbackParamName (или указать неправильное значение):

```
dojo.require("dojo.io.script");
dojo.io.script.get({
    callbackParamName : "jsoncallback", //определяется службой Flickr
    url: "http://www.flickr.com/services/feeds/photos_public.gne",
    content : {format : "json"},
    load : function(response, ioArgs) {
        console.log(response);
        return response;
    },
    error : function(response, ioArgs) {
        console.log("error");
        console.log(response);
        return response;
    }
});
```

## Получение программного кода JavaScript с помощью вызова JSONP

Как оказывается, функцию dojo.io.script.get можно использовать для взаимодействия с сервером, который возвращает чистый программный код JavaScript. В этом случае запрос выглядит точно так же, только вместо параметра callbackParamName следует определить значение параметра checkString. Значение параметра checkString представляет собой механизм, позволяющий проверить получение ответа. Если приме-

нение оператора `typeof` к имени, переданному в параметре `checkString`, не возвращает значение `undefined`, то можно предположить, что сценарий JavaScript был полностью загружен. (Другими словами, это грубый прием.) Предположим, что у нас имеется приведенный далее простой сценарий, использующий модуль CherryPy. Для проверки успешной загрузки сценария вы могли бы использовать значение `o` в параметре `checkString`, так как `o` – это переменная, которая, как предполагается, будет установлена в результате вызова JSONP (когда будет выполнено условие `typeof(o) != undefined`, можно будет предположить, что вызов завершился благополучно).

Сначала приведем сценарий CherryPy, возвращающий программный код JavaScript:

```
import cherrypy

class Content:
    @cherrypy.expose
    def index(self):
        return "var o = {a : 1, b:2}"

cherrypy.quickstart(Content())
```

Предположим, что CherryPy прослушивает порт 8080, тогда соответствующая реализация получения программного кода JavaScript с использованием возможностей Dojo может выглядеть так:

```
dojo.require("dojo.io.script");
dojo.io.script.get({
    checkString : "o",
    timeout : 2000,
    url : "http://localhost:8080",
    load : function(response, ioArgs) {
        console.log(o);
        console.log(response)
    },
    error : function(response, ioArgs) {
        console.log("error", response, ioArgs);
        return response;
    }
});
```

Обратите внимание, что функция `dojo.io.script.get` сама определяет, что загружается – программный код JavaScript или объект JSON, на основе присутствия параметра `checkString` или `callbackParamName`.

## Передача данных с помощью IFRAME

Библиотека Core обеспечивает средства передачи данных с помощью тега `IFRAME`, что бывает удобно для выполнения в фоновом режиме задач, которые в нормальных условиях требуют полной перезагрузки

страницы. Функции XHR позволяют получать данные в фоновом режиме, но они мало пригодны для выполнения некоторых задач. В качестве типичных примеров, когда может пригодиться прием передачи данных с помощью IFRAME, можно назвать отправку формы, выгрузку файлов и инициацию загрузки файлов.

Следуя тому же шаблону, что и остальная часть системы ввода-вывода, функции, выполняющие транспортировку данных с помощью тега IFRAME, требуют, чтобы им передавался объект, содержащий именованные параметры, и возвращают объект Deferred. Эти функции позволяют использовать метод GET или POST в качестве метода HTTP и различные значения в параметре handleAs. Фактически можно передавать любые параметры в соответствии с описанием в табл. 4.4.

*Таблица 4.4. Именованные аргументы функций, использующих IFRAME в качестве средства передачи*

| Имя | Тип (значение по умолчанию) | Комментарий |
|---|---|---|
| method | String ("POST") | Используемый метод HTTP. Допускается использовать значение GET или POST. |
| handleAs | String ("text") | Определяет ожидаемый тип данных ответа, который будет передан функции load или функции-обработчику. Допустимыми типами в зависимости от используемого способа передачи данных являются: "text", "html", "javascript" и "json". Для любых значений, кроме "html", ответ сервера должен представлять собой страницу HTML с элементом textarea, содержащим ответ. |
| content | Object | Если среди параметров присутствует параметр с именем form, то свойства объекта content преобразуются в скрытые элементы формы. Если параметр form отсутствует, то объект content преобразуется в строку запроса с помощью функции dojo.objectToQuery(). |

Начиная с версии 1.2 функции, использующие IFRAME, стали способны обрабатывать XML.

## Загрузка файлов с помощью IFRAME

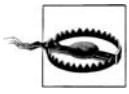
Поскольку запуск загрузки файла посредством тега IFRAME является распространенной операцией, попробуем реализовать ее. Ниже приводится файл CherryPy, который отправляет локальный файл, когда происходит обращение по адресу *http://localhost:8080/*. Мы будем использовать этот адрес URL при обращении к серверу с помощью функции dojo.io.frame.send:

```
import cherrypy
from cherrypy.lib.static import serve_file
import os

# определите здесь абсолютный путь к файлу на вашей машине
local_file_path="/tmp/foo.html"

class Content:

    #обслуживает передачу файла...
    @cherrypy.expose
    def download(self):
        return serve_file(local_file_path, "application/x-download",
                          "attachment")

# запустить веб-сервер, который прослушивает порт 8080
cherrypy.quickstart(Content(), '/')
```

Ниже следует файл HTML, в котором используется `IFRAME`. Вы должны открыть его в броузере, и если в сценарии CherryPy правильно указан путь к этому файлу, то после щелчка на кнопке перед вами появится диалог загрузки файла.

При первом вызове функции `dojo.io.iframe.send` можно заметить, как появляется, а потом тут же исчезает плавающий фрейм IFRAME. Обычно решить эту проблему можно, создав фрейм IFRAME путем отправки пустого запроса во время загрузки страницы, когда этот эффект практически незаметен. Зато потом, когда ваше приложение будет отправлять запросы, этот побочный эффект проявляться не будет.

```
<html>
    <head>
        <title>Fun with IFRAME Transports!</title>

        <script type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js">
        </script>

        <script type="text/javascript">
            dojo.require("dojo.io.iframe");

            dojo.addOnLoad(function() {
                download = function() {
                    dojo.io.iframe.send({
                        url : "http://localhost:8080/download/"
                    });
                };
            });
        </script>
    </head>
    <body>
        <button onclick="javascript:download()">Download!</button>
    </body>
</html>
```



Чтобы иметь возможность использовать кнопку «Download!» (загрузить) несколько раз, возможно, вам потребуется определить в вызове функции `dojo.io.iframe.send` параметр с предельным временем ожидания, чтобы по истечении этого времени она могла быть доступна для выполнения другого запроса.

## Отправка формы с помощью IFRAME

Еще одно типичное использование плавающих фреймов `IFRAME` заключается в том, чтобы отправлять содержимое формы в фоновом режиме, причем это может быть даже форма выгрузки файла, что в обычном случае влечет за собой полную перезагрузку страницы. Ниже следует сценарий CherryPy, который обслуживает операцию выгрузки файла:

```
import cherrypy

# определите здесь, куда следует поместить выгружаемый файл
local_file_path="/tmp/uploaded_file"

class Content:

    #обслуживает передачу файла...
    @cherrypy.expose
    def upload(self, inbound):
        outfile = open(local_file_path, 'wb')
        inbound.file.seek(0)
        while True:
            data = inbound.file.read(8192)
            if not data:
                break
            outfile.write(data)
        outfile.close()

        # вернуть в качестве ответа простой файл HTML
        return "<html><head></head><body>Thanks!</body></html>"

# запустить веб-сервер, который прослушивает порт 8080
cherrypy.quickstart(Content(), '/')
```

А ниже приводится страница HTML, которая выполняет выгрузку. Когда выгрузка файла запускается с помощью IFRAME, она происходит в фоновом режиме, и сама страница при этом не изменяется, но если выгрузка файла инициируется с помощью кнопки отправки формы, происходит переключение страницы. Обратите внимание, что в этом примере параметр запроса `handleAs` имеет значение `"html"`.

```
<html>
    <head>
        <title>Fun with IFRAME Transports!</title>

        <script type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js"
            djConfig="isDebug:true,dojoBlankHtmlUrl:'/path/to/blank.html'">
        </script>
```

```
        <script type="text/javascript">
            dojo.require("dojo.io.iframe");

            dojo.addOnLoad(function() {
                upload = function() {
                dojo.io.iframe.send({
                    form : "foo",
                    handleAs : "html", //тип ответа сервера
                    url : "http://localhost:8080/upload/",
                    load : function(response, ioArgs) {
                        console.log(response, ioArgs);
                        return response;
                    },
                    error : function(response, ioArgs) {
                        console.log("error");
                        console.log(response, ioArgs);
                        return response;
                    }
                });
            };
            });
        </script>
    </head>
    <body>
        <form id="foo" action="http://localhost:8080/upload/" method="post"
              enctype="multipart/form-data">
            <label for="file">Filename:</label>
            <input type="file" name="inbound">
            <br />
            <input type="submit" value="Submit Via The Form">
        </form>

        <button onclick="javascript:upload();">Submit Via the IFRAME Transport
        </button>
    </body>
</html>
```

В следующем разделе рассматриваются особенности получения ответа от сервера, который не является текстом разметки HTML.

## Типы ответов, отличные от HTML

В предыдущем примере сервер возвращал в качестве ответа документ HTML, который можно было извлечь и выполнить над ним некоторые манипуляции. Однако, когда ответ не является разметкой HTML, необходимо соблюсти специальное условие – текст ответа должен заключаться в тег `textarea`. Как оказывается, использование документа HTML является единственно надежным и переносимым способом передачи данных с помощью IFRAME, а тег `textarea` является естественным средством транспортировки содержимого в виде текста. Конечно, внутренние механизмы Dojo извлекают это содержимое и используют его как тело ответа. В следующем примере показано, как следует изменить

предыдущий пример, чтобы в качестве ответа можно было использовать простой текст, а не разметку HTML.

Обратите внимание, что в предыдущих примерах выгрузки и загрузки файлов не требовалось передавать CherryPy локальный файл HTML, как это делается в следующем примере. Это отличие обусловлено тем, что при использовании `IFRAME` необходимо иметь доступ к дереву DOM страницы для извлечения содержимого, что можно квалифицировать как межсайтовый скриптинг (тогда как в предыдущих примерах вообще не привлекались операции с деревом DOM).

В сценарии CherryPy нужно только добавить обслуживание файла *foo.html* и окончательный ответ обернуть в тег `textarea`, как показано ниже:

```
import cherrypy
import os

# программный код, выполняющий запросы XHR, будет находиться в файле foo.html,
# и это видно в следующей директиве config
current_dir = os.getcwd()
config = {'/' :
    {
    'tools.staticfile.on' : True,
    'tools.staticfile.filename' : os.path.join(current_dir, 'foo.html')
    }
}

local_file_path="/tmp/uploaded_file"

class Content:

    #обслуживает передачу файла...
    @cherrypy.expose
    def upload(self, inbound):
        outfile = open(local_file_path, 'wb')
        inbound.file.seek(0)
        while True:
            data = inbound.file.read(8192)
            if not data:
                break
            outfile.write(data)
        outfile.close()
        return "<html><head></head><body><textarea>Thanks!</textarea></
body></html>"

cherrypy.quickstart(Content(), '/', config=config)
```

Единственное, что следует изменить в самом запросе, – это указать иной тип в параметре `handleAs`:

```
dojo.io.iframe.send({
    form : dojo.byId("foo"),
    handleAs : "text", //тип ответа сервера
```

```
        url : "http://localhost:8080/upload/",
        load : function(response, ioArgs) {
            console.log(response, ioArgs); //ответ: "Thanks!"
            return response;
        },
        error : function(response, ioArgs) {
            console.log("error");
            console.log(response, ioArgs);
            return response;
        }
    });
```

## Создание скрытого фрейма IFRAME вручную

В заключение рассмотрим ситуацию, когда в странице необходимо создать скрытый плавающий фрейм IFRAME, чтобы загружать в него некоторое содержимое, и необходимо реализовать передачу извещения, когда содержимое будет загружено. В отличие от функции dojo.io.iframe.send, которая создает элемент IFRAME и тут же посылает запрос на получение содержимого, функция dojo.io.iframe.create создает элемент IFRAME и позволяет определить фрагмент программного кода JavaScript, который будет выполнен после того, как создание и наполнение фрейма завершится. Эта функция имеет следующую сигнатуру:

```
dojo.io.iframe.create(/*String*/frameName, /*String*/onLoadString,
                      /*String?*/url)//Возвращает DOMNode
```

В сущности, вы должны определить имя фрейма, строковое значение, которое будет рассматриваться как функция обратного вызова, и необязательный адрес URL для фрейма. Ниже приводится пример, который загружает содержимое адреса URL в скрытый плавающий фрейм IFRAME на странице и по окончании загрузки вызывает функцию обратного вызова:

```
<html>
    <head>
        <title>Fun with IFRAME Transports!</title>

        <script type="text/javascript"
            src="http://o.aolcdn.com/dojo/1./dojo/dojo.xd.js"
            djConfig="isDebug:true,dojoBlankHtmlUrl:'/path/to/blank.html'"
        </script>

        <script type="text/javascript">
            dojo.require("dojo.io.iframe");

            function customCallback() {
                console.log("callback!");

                //здесь можно сослаться на фрейм с помощью
                //dojo.byId("fooFrame")...
            }

            create = function() {
```

```
                dojo.io.iframe.create("fooFrame", "customCallback()",
                                      "http://www.exmaple.com");
            }
        </script>
    </head>
    <body>
        <button onclick="javascript:create();">Create</button>
    </body>
</html>
```

Имейте в виду, что некоторые страницы включают в себя функции JavaScript, которые разбивают эти страницы на фреймы, что делает приведенное использование этого вида транспорта неэффективным.

Чаще всего вы будете немедленно загружать содержимое в элемент `IFRAME`, но иногда бывает необходимо просто создать пустой фрейм. Если вы пользуетесь версией инструментального набора для локальной работы, то для создания пустого фрейма достаточно всего лишь опустить третий параметр в вызове функции `dojo.io.iframe.create`. Если же вы пользуетесь версией XDomain, то вам необходимо будет указать путь к локальному шаблону, представляющему содержимое. В каталоге с инструментальным набором этот шаблон находится в файле *dojo/resources/blank.html*, который можно скопировать в удобное для вас местоположение. Кроме того, прежде чем создавать `IFRAME`, в массив `djConfig` необходимо добавить дополнительный параметр настройки, как показано в примере следующего раздела.

Помимо средств ввода-вывода в библиотеке Core, средства ввода-вывода также имеются в библиотеке DojoX, в модуле `dojox.io`. Здесь, кроме всего прочего, вы найдете функции для выполнения запросов XHR, состоящих из нескольких частей, и вспомогательные функции для работы с прокси-сервером.

# Вызов удаленных процедур

Возможно, к этому моменту вы уже заметили, что даже после использования различных методов работы с XHR, имеющихся в Dojo, таких как `dojo.xhrGet`, с целью уменьшить объем шаблонного программного кода, по-прежнему приходится реализовывать несколько избыточную и подверженную ошибкам операцию передачи данных в вызов функции и создания функции обратного вызова `load`. К счастью, существует возможность использовать механизм RPC (Remote Procedure Call – вызов удаленных процедур), позволяющий устранить некоторую монотонность этой работы с помощью модуля `dojo.rpc`, входящего в состав библиотеки Core. Проще говоря, вы предоставляете некоторую информацию о конфигурации посредством простого описания методов (Simple Method Description, SMD); отправляя информацию о конфигу-

рации, создаете экземпляр службы RPC и затем используете эту службу вместо xhrGet и других подобных функций. Если ваше приложение придерживается стандартных способов взаимодействий с сервером и предусматривает стандартную обработку ошибок, то при использовании модуля rpc вы получите более прозрачный дизайн, который менее подвержен ошибкам.

В настоящее время библиотека Core предоставляет два класса — JsonService и JsonpService, причем оба наследуют класс RpcService.

В модуле dojox.rpc имеются дополнительные возможности RPC, часть которых в скором времени может быть перемещена в библиотеку Core.

## Пример JSON RPC

Для иллюстрации основ использования механизма RPC рассмотрим пример, в котором для обработки списка чисел используется класс JsonService; в примере вычисляется сумма чисел или сумма квадратов чисел. Клиент определяет SMD, в котором описываются два метода, sum и sumOfSquares, каждый из которых принимает список чисел:

```
<html>
    <head>
        <title>Fun with JSON RPC!</title>

        <script type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js"
            djConfig="isDebug:true">
        </script>

        <script type="text/javascript">
            dojo.require("dojo.rpc.JsonService");
            dojo.addOnLoad(function() {

                //создать описание smd в виде литерала объекта...
                var o = {
                    "serviceType": "JSON-RPC",
                    "serviceURL": "/",
                    "methods":[
                        {
                            "name": "sum",
                            "parameters":[{name : "list"}]
                        },
                        {
                            "name": "sumOfSquares",
                            "parameters":[{name : "list"}]
                        }
                    ]
                }

                //создать экземпляр службы
```

```
                var rpcObject = new dojo.rpc.JsonService(o);

                //вызвать службу и использовать объект Deferred
                //для добавления функций обратного вызова
                var sum = rpcObject.sum([4,8,15,16,23,42]);
                sum.addCallback(function(response) {
                    console.log("the answer is ", response);
                });
                //добавить дополнительные функции обратного вызова
                //и обработки ошибок.

                //функция sumOfSquares вызывается точно так же...
            });
        </script>
    </head>
    <body>
    </body>
</html>
```

Надеюсь, вы поняли, что при большом количестве методов, взаимодействующих с сервером стандартным способом, общая простота вызовов RPC на стороне клиента после их настройки значительно облегчает структуру приложения. Элегантность использования функции `dojo.rpc.JsonService` в значительной степени обусловлена тем, что она возвращает объект `Deferred`, благодаря чему имеется возможность добавлять функции обратного вызова и функции обработки ошибок.

На тот случай, если вы пожелаете опробовать этот пример в действии, ниже приводится сценарий службы. Для простоты примера этот сценарий не использует библиотеку работы с форматом JSON, но на практике вы наверняка пожелаете реализовать нечто более сложное, чем этот сценарий:

```
import cherrypy
import os
# программный код, выполняющий запросы XHR, будет находиться в файле foo.html,
# и это видно в следующей директиве config

current_dir = os.getcwd()
config = {'/foo.html' :
    {
    'tools.staticfile.on' : True,
    'tools.staticfile.filename' : os.path.join(current_dir, 'foo.html')
    }
}

class Content:

    @cherrypy.expose
    def index(self):
        ##############################################################
        # для большей простоты этот пример не использует библиотеку json.
        # для реализации более сложных сценариев, чем этот, вы можете
```

```
        # получить замечательную библиотеку по адресу http://json.org
        ##########################################################

        # прочитать необработанные данные, переданные методом POST
        rawPost = cherrypy.request.body.read()

        # преобразовать в объект
        obj = eval(rawPost) #ВНИМАНИЕ! Это дыра в системе безопасности!
                            #вы предупреждены...

        # обработать данные
        if obj["method"] == "sum":
            result = sum(obj["params"][0])
        if obj["method"] == "sumOfSquares":
            result = sum([i*i for i in obj["params"][0]])

        # вернуть ответ в формате json
        return str({"result" : result})

# запустить веб-сервер, который будет прослушивать порт 8080
cherrypy.quickstart(Content(), '/', config=config)
```

Функция `JsonpService` используется практически так же, как и `JsonService`. В дистрибутиве Dojo имеется пример файла SMD (*dojox/rpc/yahoo.smd*) для доступа к службе Yahoo!, который при желании можно опробовать.

# OpenAjax Hub

OpenAjax Alliance (*http://www.openajax.org/*) – это объединение производителей и организаций, которые посвятили себя выработке веб-технологий взаимодействий на базе AJAX. Один из ключевых вопросов на текущем этапе развития веб-разработки заключается в обеспечении возможности одновременного использования нескольких библиотек JavaScript в пределах одного приложения. В инструментальном наборе Dojo и в некоторых других платформах принимаются меры предосторожности, такие как защита глобального пространства имен, чтобы обеспечить необходимый минимум для организации взаимодействий. тем не менее одновременное использование двух библиотек на практике – так, чтобы они действительно взаимодействовали друг с другом, порождает проблемы, связанные с организацией обмена данными между ними, а также проблемы соблюдения единого стиля программирования и эффективности обучения.

Объединение OpenAjax Alliance внесло предложение, известное как OpenAjax Hub, которое является спецификацией принципов взаимодействий библиотек. Вас, вероятно, не удивит, что в основу взаимодействий между библиотеками была положена идиома архитектуры с гибко связанными компонентами, где передача событий производится по подписке. Со своей стороны, библиотека Core предоставляет модуль `OpenAjax`, который является реализацией спецификации и предлагает к использованию следующие методы через глобальный объект `OpenAjax`:

- `registerLibrary`
- `unregisterLibrary`
- `publish`
- `subscribe`
- `unsubscribe`

Вы можете пребывать в уверенности, что инструментарий Dojo, как лидер в развитии открытых стандартов, будет стремится следовать самым последним требованиям спецификации OpenAjax Hub, которую можно найти по адресу: *http://www.openajax.org/member/wiki/OpenAjax_Hub_Specification*.

# В заключение

После прочтения этой главы вы должны:

- Уметь использовать механизм XHR для выполнения операций с веб-сервером в архитектуре RESTful
- Понимать, как объект `Deferred` создает иллюзию нескольких потоков выполнения, даже при том, что JavaScript не поддерживает эту концепцию
- Знать, что вся система ввода-вывода инструментального набора использует объекты `Deferred` и практически все ее функции возвращают объекты `Deferred`
- Уметь пользоваться функциями из библиотеки Base, выполняющими преобразование объектов в формат JSON
- Уметь пользоваться механизмом передачи данных на базе `IFRAME`, реализованном в библиотеке Core, для выполнения таких распространенных операций, как выгрузка и загрузка файлов
- Понимать, как механизм RPC может упростить реализацию приложения и способствовать созданию программ, более простых в сопровождении
- Знать о существовании инфраструктуры в библиотеке Core, реализующей спецификацию OpenAjax Hub

В следующей главе мы перейдем к вопросам манипулирования узлами.

# 5

# Манипулирование узлами

В этой главе предлагается обзор функции `query`, модуля `behavior` и класса `NodeList`. Эти конструкции предоставляют лаконичные и высокоэффективные механизмы манипулирования узлами в дереве DOM. Основные темы, рассматриваемые в этой главе: поиск в дереве DOM с помощью функции `query` и с использованием синтаксиса селекторов CSS, развязка событий и выполнение манипуляций над элементами HTML средствами модуля `behavior` библиотеки Core и объединение операций в цепочки с помощью синтаксического подсластителя, предлагаемого классом `NodeList`.

## Поиск: универсальная реализация

Если вы имеете богатый опыт программирования на JavaScript, у вас не будет сомнений в необходимости средств, выполняющих поиск узлов в дереве DOM на основе некоторого множества критериев. Если все, что вам требуется, – это отыскать узлы с определенным именем тега, то вполне достаточно воспользоваться функцией `document.getElementsByTagName`. Однако когда необходимо отыскать множество узлов по имени класса, определенному значению атрибута или некоторой их комбинации, вам придется крепко задуматься над тем, почему в JavaScript отсутствует встроенная функция `getElementsByClass`. Очевидно, что любой, кто задавался этим вопросом, пытался написать собственную версию функции – одни более успешно, другие – менее.

Ранние версии Dojo включали в себя специализированные версии функции `getElementsByClass`, но теперь инструментальный набор содержит более универсальную функцию, позволяющую выполнять поиск в дереве DOM, используя синтаксис селекторов CSS. Чтобы продемонстрировать, как этот «швейцарский нож» применяется для поиска

## Поиски вокруг поиска

К моменту написания этой главы произошло много интереснейших событий, связанных с организацией поиска элементов в дереве DOM. Вот некоторые из них:

- В конце 2007 года консорциум W3C дополнил рабочий проект стандарта «Selectors API» (прикладной интерфейс селекторов) (*http://www.w3.org/TR/selectors-api/#documentselector*)
- Проект WebKit объявил о реализации `querySelector` и `querySelectorAll` – ключевых конструкций в прикладном интерфейсе селекторов (*http://webkit.org/blog/156/queryselector-and-queryselectorall/*)
- В Firefox3 была реализована поддержка функции `getElementsByClassName` – основной опоры веб-разработчиков (*http://ejohn.org/blog/getelementsbyclassname-in-firefox-3/*)

Можно с уверенностью сказать, что средства, обсуждаемые в этой главе, еще долго будут основой инструментального набора и приемов, которые вы будете использовать. Даже при наличии в броузерах поддержки прикладного интерфейса селекторов в них все равно будут наблюдаться некоторые расхождения, которые придется обыгрывать и сглаживать, а в некоторых броузерах эта поддержка наверняка будет реализована только частично.

Пока не наступит день, когда все броузеры будут иметь однородную поддержку прикладного интерфейса селекторов, можно пребывать в уверенности, что инструментальный набор обеспечит максимально возможную переносимость и оптимальность вашего программного кода, дополняя имеющиеся реализации там, где это возможно, и имитируя реализации там, где они отсутствуют.

в дереве DOM, рассмотрим героическую попытку реализовать собственную версию функцию `getElementsByClass` (для очень общего случая):

```
// Отыскивает элементы с заданным именем класса,
// При желании поиск может начинаться с определенного родительского узла
function getElementsByClassName(/*String*/className, /*DOMNode?*/node) {
    var regex = new RegExp('(^| )' + className + '( |$)');
    var node = node||document.body;
    var elements = node.getElementsByTagName("*");
    var results = [];

    for (var i=0; i < elements.length; i++) {
        if (regex.test(elements[i].className)) {
            results.push(elements[i]);
        }
```

```
    }
    return results;
```

Хотя эта функция занимает всего 13 строк программного кода, тем не менее эти 13 строк все равно придется писать, отлаживать и сопровождать. Если вам потребуется выполнить поиск по комбинации имени класса и имени тега, то придется добавить еще один параметр для имени тега и передавать его функции getElementsByTagName. Если потребуется что-то еще, вам точно так же придется написать и отладить и эту логику. Теперь к этому всему добавьте тот факт, что наверняка встретится пара крайних случаев, когда функция, представленная выше, будет работать не всегда или не во всех броузерах, а также то обстоятельство, что обычно регулярные выражения не являются интуитивно очевидными.

К счастью, dojo.query избавляет от необходимости иметь свои функции поиска. Ниже приводится сигнатура функции, обеспечивающей универсальные возможности поиска:

```
dojo.query(/*String*/ query, /*String?|DOMNode?*/ root) //Возвращает NodeList
```

Формальное знакомство с классом NodeList состоится лишь несколькими страницами далее, тем не менее уже сейчас вы должны знать, что NodeList — это подкласс Array, обладающий специализированными методами для манипулирования узлами.

Чтобы с помощью query имитировать поведение представленной выше функции getElementsByClassName, достаточно просто передать ей селектор CSS, как показано ниже:

```
dojo.query(".someClassName")
```

Поиск по имени тега, например DIV, и по имени класса выполняется ничуть не сложнее — достаточно просто воспользоваться синтаксисом селекторов CSS:

```
dojo.query("div.someClass")
```

Начали понимать всю прелесть использования простого и однострочного способа поиска в дереве DOM с применением единообразного синтаксиса? Эта функция понравится вам тем больше, чем дальше прочитаете. Однако сначала рассмотрите табл. 5.1, чтобы получить представление о том, насколько широкие возможности доступны при использовании функции query. Полный справочник по селекторам CSS вы найдете по адресу: *http://www.w3.org/TR/css3-selectors/*.

*Таблица 5.1. Наиболее часто используемые селекторы CSS*

| Селектор | Значение | Пример |
|---|---|---|
| * | Любой элемент | dojo.query("*") |
| E | Элементы типа E | dojo.query("div") |

| Селектор | Значение | Пример |
|---|---|---|
| .C | Элементы с классом C | dojo.query(".baz") |
| E.C | Элементы типа E с классом C | dojo.query("div.baz") |
| #ID | Элементы со значением атрибута id, равным ID | dojo.query("#quux") |
| E#ID | Элементы типа E со значением атрибута id, равным ID | dojo.query("div#quux) |
| [A] | Элементы с атрибутом A | dojo.query("[foo]") |
| E[A] | Элементы типа E с атрибутом A | dojo.query("div[foo]") |
| [A="V"] | Элементы с атрибутом A, имеющим значение "V" | dojo.query("[foo='bar']") |
| E[A~='V'] | Элементы типа E со списком атрибутов, разделенных пробелами, один из которых имеет значение "V" | dojo.query(" div[foo~='bar']") |
| E[A^='V'] | Элементы типа E с атрибутом A, значение которого начинается с "V" | dojo.query(" div[foo^='bar']") |
| E[A$='V'] | Элементы типа E с атрибутом A, значение которого заканчивается на "V" | dojo.query(" div[foo$='bar']") |
| E[A*='V'] | Элементы типа E с атрибутом A, значение которого содержит подстроку "V" | dojo.query(" div[foo*='bar']") |
| , | Логическая операция «ИЛИ» | dojo.query("div,span.baz") |
| E > F | Элемент F является дочерним по отношению к элементу E | dojo.query("div > span") |
| E F | Элемент F является произвольным потомком элемента E | dojo.query("E F") |

## Подготовка

Давайте подготовимся к использованию dojo.query на примере простой страницы с разметкой простой структуры, содержащей сценарий. Для краткости в текст примера включена только одна сцена:

```
<div id="introduction" class="intro">
    <p>
        Once upon a time, long ago...
    </p>
</div>

<div id="scene1" class="scene">...</div>

<div id="scene2" class="scene">
    <p>
        At the table in the <span class="place">kitchen</span>, there were
three bowls of <span class="food">porridge</span>. <span class="person"
>Goldilocks</span> was hungry. She tasted the <span class="food">porridge</
span> from the first bowl.
```

```
        </p>
        <p>
            "This <span class="food">porridge</span> is too hot!" she exclaimed.
        </p>
        <p>
            So, she tasted the <span class="food">porridge</span> from
            the second bowl.
        </p>
        <p>
            "This <span class="food">porridge</span> is too cold," she said
        </p>
        <p>
            So, she tasted the last bowl of <span class="food">porridge</span>.
        </p>
        <p>
            "Ahhh, this <span class="food">porridge</span> is just right,"
            she said happily and she ate it all up.
        </p>
    </div>

    <div id="scene3" class="scene">...</div>
```

Как было продемонстрировано в более раннем примере, функция `getElementsByTagName` возвращает массив узлов DOM заданного типа. Чтобы получить тот же результат с помощью `dojo.query`, ей нужно просто передать строку с именем тега в качестве аргумента. Например, чтобы получить список всех элементов `div` в странице, следует использовать `dojo.query("div")`, как показано ниже:

```
dojo.query("div")
//Вернет [div#introduction.intro, div#scene1.scene, div#scene2.scene,
//div#scene3.scene]
```

Обратите внимание: если потребуется отобрать только дочерние узлы определенного элемента, а не все узлы DOM страницы, можно определить второй аргумент, который интерпретируется функцией как корневой узел дерева. Например, чтобы отобрать только элементы параграфов только в элементе с идентификатором `scene2`, во втором аргументе нужно передать либо сам узел, либо значение атрибута `id` этого узла, как показано ниже:

```
dojo.query("p", "scene2")
//Вернет [p, p, p, p, p, p]
```

Поиск элементов страницы определенного класса также выполняется очень просто, достаточно лишь указать имя требуемого класса с использованием синтаксиса CSS, то есть в соответствии со спецификацией это означает, что имя класса должно предваряться символом точки. Например, можно было бы отобрать все элементы, к которым в настоящий момент применен класс `food`, как показано ниже:

```
dojo.query(".food")
//Вернет [span.food, span.food, span.food, span.food, span.food,
//span.food, span.food]
```

Функции addClass и removeClass из библиотеки Base не ожидают, что имена классов будут начинаться с символа точки, и будут возвращать неверные результаты, если добавить этот символ. Эту частность легко забыть, когда вы только начинаете использовать инструментарий.

Выполнить поиск по имени тега и по имени класса очень просто: достаточно лишь объединить две конструкции. Представьте ситуацию, когда желательно отыскать элементы span, к которым применен класс place:

```
dojo.query("span.place")
//Вернет [span.place]
```

Возможность поиска элементов по имени класса удобна сама по себе, но иногда возникает потребность отобрать элементы сразу по нескольким классам. К счастью, эту задачу можно решить, используя все тот же простой подход, который уже наверняка полюбился вам. Например, можно отобрать все элементы, к которым применены классы food и place, как показано ниже:

```
dojo.query(".food,.place")
//Вернет [span.food, span.food, span.food, span.food, span.food, span.food,
//span.food, span.place]
```

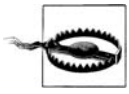
Части выражения CSS, разделенные запятыми, – это самостоятельные, независимые выражения. Они не являются лево-ассоциативными, как некоторые математические операторы или элементы грамматики.

В качестве последнего примера гибкости функции query рассмотрим случай поиска потомков определенного узла. Предположим, что в данный момент нам требуется отыскать все узлы с классом food, которые вложены в узел scene2:

```
dojo.query("#scene2 .food")
//Вернет [span.food, span.food, span.food, span.food, span.food, span.food,
//span.food]
```

Обратите внимание: в данном случае при использовании оператора > был бы получен пустой список, потому что в этой странице нет узлов с классом food, которые являлись бы непосредственными потомками узла scene2:

```
dojo.query("#scene2 > .food")
//Вернет []
```



Очень часто возникает путаница в применении комбинатора дочернего узла (>) и комбинатора потомка (пробел). Комбинатор дочернего узла отбирает только непосредственные дочерние узлы, тогда как комбинатор потомка отбирает любых потомков независимо от уровня вложенности в дереве DOM.

Несмотря на всю краткость этого примера, важно отметить, что максимально возможное сужение области поиска за счет выбора как можно более конкретного критерия может оказывать существенное влияние на производительность.

## Пример наблюдения за состоянием

Кроме очевидного случая применения функции `dojo.query` для поиска узлов в дереве DOM это мощное средство способно изменять способы решения множества общих проблем – благодаря тому, что оно расширяет творческие возможности. В качестве простого примера рассмотрим проблему наблюдения за состоянием приложения, которая является *самой* типичной проблемой при разработке достаточно сложных приложений. Возможно, требуется определить, выделен определенный раздел текста или нет; или, возможно, требуется узнать, было ли выполнено некоторое действие. Можно было бы ввести явные переменные, которые хранили бы всю информацию о состоянии, однако использование классов CSS обеспечивает более изящное решение проблемы.

Например, предположим, что вы разрабатываете ультрасовременную поисковую систему для Всемирной паутины, которая способна определять сущности в документе, и что вам требуется явно выделить имена людей, присутствующие в результатах поиска. Предположим, что документ с результатами поиска содержит пьесу У. Шекспира «Макбет» и вам необходимо выделить персонажей в нем. Документ с результатами мог бы содержать следующий текст:

```
...
<a rel="person">First Witch</a>
When shall we three meet again
In thunder, lightning, or in rain?

<a rel="person">Second Witch</a>
When the hurlyburly's done,
When the battle's lost and won.

<a rel="person">Third Witch</a>
That will be ere the set of sun.

<a rel="person">First Witch</a>
Where the place?

<a rel="person">Second Witch</a>
Upon the heath.

<a rel="person">Third Witch</a>
```

```
There to meet with <a rel="person">Macbeth</a>.
...
```

## Длинный и нестабильный способ

Как разработчик, питающий слабость к удобству, вы могли бы захотеть добавить небольшую панель управления сбоку страницы, с помощью которой можно было бы переключать режим подсветки определенных типов сущностей в результатах поиска. В результате низкоуровневого подхода, связанного с применением JavaScript для прямого манипулирования деревом DOM, мог получиться следующий программный код:

```
function addHighlighting(entityType) {
    var nodes = document.getElementsByTagName("a");
    for (var i=0; i < nodes.length; i++) {
        if (nodes[i].getAttribute('rel')==entityType) {
            nodes[i].className="highlighted";
        }
    }
}

function removeHighlighting(entityType) {
    var nodes = document.getElementByTagName("a");
    for (var i=0; i < nodes.length; i++) {
        if (nodes[i].getAttribute('rel')==entityType) {
            nodes[i].className="";
        }
    }
}
```

Эти функции решают поставленную задачу, но они основаны на наивном предположении, что с результатами поиска не может быть связан никакой другой класс, кроме `highlighted`, потому что обе функции напрямую переопределяют имя класса. Таким образом, нам также потребуется создать функции добавления и удаления классов в узлах, которые, возможно, относятся к нескольким классам, что влечет за собой необходимость приложения дополнительных усилий по организации поиска по строковым значениям свойства `className` и затем добавление или удаление заданного имени класса. Чтобы предотвратить дальнейшее «захламление» программного кода, можно было бы использовать функции `addClass` и `removeClass` из библиотеки Base, о которых вы узнали в главе 2, но это не уменьшит уже имеющиеся «горы хлама».

## Короткий и надежный способ

Ниже приводится способ, который успешно решает поставленную задачу с помощью функции `query`:

```
function addHighlighting(entityType) {
    dojo.query("span[type="+entityType+"]").addClass("highlighted");
}
```

```
function removeHighlighting(entityType) {
    dojo.query("span[type="+entityType+"]").removeClass("highlighted");
}
```

В данном конкретном случае нам удалось избавиться от низкоуровневых манипуляций с деревом DOM, от циклов `for` и условной инструкции в обмен на элегантный синтаксис CSS. Кроме того, эти функции учитывают возможность, что к сущностям в документе с результатами поиска может быть применено сразу несколько классов CSS.

Несмотря на то что `dojo.query` не может ничего такого, что нельзя было бы сделать более длинным способом, тем не менее предыдущее исследование наглядно демонстрирует, что `dojo.query` обеспечивает единый и универсальный интерфейс поиска и манипулирования элементами дерева DOM на очень высоком уровне и что усложняется в этом случае строка критерия, а не логика условных инструкций. Не говоря уже о том, что применение `dojo.query` выглядит более элегантно, чем манипулирование элементами DOM на низком уровне.

Если вы уже стали строить планы, как и где задействовать функцию `query`, подождите немного: стоит опираться на гибкость, которую предлагает класс `NodeList`. Это тип значения, возвращаемого функцией `query`, и ему посвящен следующий раздел.

# NodeList

`Nodelist` — это специализированный подкласс класса `Array`, который специально предназначен для использования в некоторых расширениях, чтобы упростить манипулирование совокупностями узлов в дереве DOM. Одна из наиболее интересных особенностей класса `NodeList` состоит в том, что он обеспечивает возможность составления цепочек операций с помощью оператора точки. Однако существуют и другие специализированные возможности, такие как отображение, фильтрация и поиск индекса существующего узла.

В табл. 5.2 приводится краткий обзор методов класса `NodeList`. Схема именования этих методов следует тем же соглашениям, что и функции для работы с типом `Array` в библиотеке Base. Единственное различие состоит в том, что эти методы возвращают значение типа `NodeList`, а не `Array`.

*Таблица 5.2. Методы класса NodeList*

| Имя | Комментарий |
|---|---|
| `indexOf(/*DOMNode*/n)` | Возвращает индекс первого элемента `n` в `NodeList`. |
| `lastIndexOf(/*DOMNode*/n)` | Возвращает индекс последнего элемента `n` в `NodeList`. |
| `every(/*Function*/f)` | Возвращает `true`, если функция `f` вернула `true` для всех элементов в `NodeList`. |

| Имя | Комментарий |
|---|---|
| some(/*Function*/f) | Возвращает true, если функция f вернула true хотя бы для одного элемента в NodeList. |
| forEach(/*Function*/f) | Для каждого элемента в NodeList выполняет функцию f и возвращает оригинальный NodeList. |
| map(/*Function*/f) | Для каждого элемента в NodeList выполняет функцию f и возвращает результаты в виде NodeList. |
| filter(/*Function*/f) | Для каждого элемента в NodeList выполняет функцию f и возвращает только те элементы, которые соответствуют критерию, реализованному функцией, или применяет к списку узлов фильтр CSS. |
| concat(/*Any*/item, ...) | Возвращает новый NodeList с добавленными элементами. Поведение этого метода аналогично методу Array.concat за исключением того, что возвращаемое значение имеет тип NodeList. |
| splice(/*Integer*/index, /*Integer*/howManyToDelete, /*Any*/item, ...) | Возвращает новый NodeList с добавленными или удаленными элементами. Поведение этого метода аналогично методу Array.splice за исключением того, что возвращаемое значение имеет тип NodeList. |
| slice(/*Integer*/begin, /*Integer*/end) | Возвращает новый NodeList с элементами, попавшими в срез. Поведение этого метода аналогично методу Array.splice за исключением того, что возвращаемое значение имеет тип NodeList. |
| addClass(/*String*/class) | Добавляет класс к каждому узлу. |
| removeClass(/*String*/class) | Удаляет класс из каждого узла. |
| style(/*String\|Object*/ style) | Возвращает или устанавливает указанный стиль для каждого узла, если аргумент style имеет тип String. Работает точно так же, как функция dojo.style, и устанавливает несколько значений стиля, если аргумент style имеет тип Object. |
| addContent(/*String*/ content, /*String?\|Integer?*/ position) | Добавляет в каждый узел текстовую строку или узел в указанную позицию. Допустимыми значениями аргумента position являются: first, last, before и after. Значения first и last являются функциями родительского узла, тогда как значения before и after вычисляются относительно самого узла. |
| place(/*String\|Node*/ queryOrNode, /*String*/ position) | Помещает каждый элемент из списка относительно узла или относительно первого элемента, соответствующего критерию поиска. Допустимыми являются те же самые значения, что и в методе addContent (смотри выше). |

*Таблица 5.2 (продолжение)*

| Имя | Комментарий |
|---|---|
| `coords()` | Возвращает объекты вмещающих прямоугольников для всех элементов в списке в виде массива, а не в виде `NodeList`. Объекты прямоугольников имеют вид: `{ l: 50, t: 200, w: 300: h: 150, x: 100, y: 300 }`, где `l` определяет величину смещения от левого края экрана, `t` — от верхнего края экрана, значения `w` и `h` определяют ширину и высоту прямоугольника, соответственно, а `x` и `y` — абсолютные координаты. |
| `orphan(/*String?*/ filter)` | Удаляет узлы DOM из списка согласно критерию фильтра и возвращает их в виде нового списка `NodeList`. |
| `adopt(/*String\|Array\|DomNode*/ queryOrListOrNode, /*String?*/ position)` | Вставляет узлы DOM относительно первого элемента списка. |
| `connect(/*String*/ methodNameOrDomEvent, /*Object*/ context, /*String*/ funcName)` | Подключает обработчики событий к каждому элементу в `NodeList`, используя функцию `dojo.connect`, благодаря чему автоматически происходит нормализация свойств событий. Аргументы метода полностью соответствуют аргументам функции `dojo.connect`, в которых передаются имя метода или событие DOM, которые будут служить источником, а также необязательный контекст и имя функции. Имена событий DOM должны быть нормализованы и указываться символами в нижнем регистре. В большинстве случаев рекомендуется использовать имена событий, обсуждаемые в разделе «Пакетная обработка событий DOM». |
| `instantiate(/*String\|Object*/ declaredClass, /*Object?*/properties)` | Этот метод удобно использовать для создания большого числа экземпляров виджетов.[a] Предполагая, что `NodeList` содержит множество произвольных исходных узлов, этот метод пытается преобразовать их в класс виджета, определяемый аргументом `declareClass`, при этом каждому виджету присваиваются свойства, определяемые аргументом `properties`. |

[a] Формально виджеты будут представлены в главе 11, соответственно, в данной главе не приводится примеров, демонстрирующих применение метода `instantiate`.

Обзор фундаментальных операций над массивами вы найдете в разделе «Обработка массивов» главы 2.

## Методы, напоминающие методы массивов

Если вы помните, в библиотеку Base включено несколько функций, предназначенных для работы с массивами. Вас наверняка обрадует, что класс NodeList обладает многими из этих методов. В частности, методы indexOf, lastIndexOf, every, some, forEach, map и filter работают точно так же, как одноименные функции, предназначенные для работы с массивами. Однако метод filter класса NodeList обладает некоторыми дополнительными возможностями в зависимости от получаемых им аргументов. (Подробнее об этом будет говориться совсем скоро.)

Для начала нам потребуется создать объект NodeList. Для этого можно использовать тот же синтаксис, что и в случае с массивами, когда явно определяется несколько элементов списка NodeList, или можно воспользоваться встроенным методом concat для создания списка NodeList из существующего массива.

Ниже демонстрируется несколько возможных способов создания нового списка NodeList:

```
var nl = new dojo.NodeList(); //создаст пустой NodeList

var nl = new dojo.NodeList(foo, bar, baz);
//создаст NodeList из нескольких существующих узлов

var a = [foo, bar, baz];
// допустим, что имеется массив объектов с несколькими узлами в нем

a = nl.concat(a); // превратит массив в NodeList
```

Если создавать список NodeList, как показано ниже, можно получить совсем не то, что вы ожидали:

```
var nl = new dojo.NodeList([foo, bar, baz]);
```

В этом случае будет получен список NodeList, содержащий объект Array с тремя элементами, – это в точности соответствует результату, который будет получен вызовом new Array([foo,bar, baz]).

### Создание цепочек обработки списков NodeList

Методы для работы с массивами, включенные в Dojo, чрезвычайно удобны, когда нет необходимости выполнять последовательность операций, передавая получаемые результаты от одной операции к другой, или когда вы должны использовать только массивы. В противном случае вы можете предпочесть использовать NodeList – структуру данных, обладающую очень элегантным синтаксисом. Следующий пример демонстрирует объединение в последовательность нескольких операций:

```
var nl = new dojo.NodeList(node1,node2,node3,node4,...);

nl.map(
    /* Отобразить некоторые элементы... */
```

```
        function(x) {
            /* ... */
        }
    )
    .filter(
        /* Теперь отфильтровать их... */
        function f(x) {
            /* ... */
        }
    )
    .forEach(
        function(x) {
            /* И выполнить некоторое действие над каждым из них... */
        }
    );
```

Взгляните, как неуклюже выглядит реализация тех же действий при использовании стандартных функций Dojo из-за необходимости вводить переменные для хранения промежуточных результатов:

```
var a0 = new Array(node1,node2,node3,node4,...);

/* Отобразить некоторые элементы... */
```

## Подробнее о NodeList

Несмотря на название NodeList, которое подразумевает, что эта структура данных предназначена для манипулирования узлами дерева DOM, имейте в виду, что это полноценный подкласс класса Array и он может хранить элементы любых типов, а не только узлы. Например, если вам действительно понравился синтаксический подсластитель объединения нескольких операций в цепочку с помощью оператора точки, вы можете использовать NodeList вместо массива и пользоваться его возможностями для манипулирования числами и другими примитивами, как показано ниже:

```
//Представим, что имеется список NodeList чисел
var nums = new dojo.NodeList(1,2,3,4,5,6,7,8,9,10)

//Тогда вы могли бы использовать преимущества, которые дает этот
//синтаксический подсластитель вместо создания переменных
//для хранения промежуточных результатов

nums
.filter(function(x) {/* ... */})
.map(function(x) { /* ... */})
//следующая операция...
//смысл должен быть вам понятен
;
```

```
var a1 = dojo.map(a0,
    function(x) {
        /* ... */
    }
);

/* Теперь отфильтровать их... */
var a2 = dojo.filter(a1
    function f(x) {
        /* ... */
    }
);

/* И выполнить некоторое действие над каждым из них... */
dojo.forEach(a2
    function f(x) {
        /* ... */
    }
);
```

Имейте в виду, что возможность объединять операции в цепочки с помощью оператора точки может давать в результате весьма элегантный программный код, в котором отсутствуют переменные, необходимые для хранения промежуточных результатов, но может также существенно осложнять процесс отладки и сопровождения приложения. Как всегда все на ваше усмотрение.

## Аргументы в стиле «функции в виде строк»

Точно так же, как и в методах манипулирования массивами из библиотеки Base, вы можете применять специальные идентификаторы `index`, `array` и `item` при использовании строковых аргументов, как описано в разделе «Обработка массивов» в главе 2. Взгляните на следующий пример, чтобы вспомнить, как это делается:

```
//Предположим, что у нас имеется NodeList с именем...

//Воспользуемся идентификатором item, чтобы избежать необходимости
//писать целую функцию-обертку
nl.forEach("console.log(item)");
```

## Расширенная фильтрация

Помимо традиционных возможностей, подобных тем, что имеются в массивах, метод `filter` класса `NodeList` предоставляет возможность фильтрации с использованием синтаксиса CSS, когда входной аргумент имеет тип `String`. Например, первый блок программного кода, что приводится ниже, демонстрирует возможность передачи функции для обработки каждого отдельного элемента данных в списке. Второй блок кода использует синтаксис CSS для критерия отбора, чтобы обработать только необходимые узлы DOM:

```
dojo.query("div")
.forEach(
    /* Вывести все элементы div */
    function f(x) {
        console.log(x);
    })
.filter(".div2") //отобрать элементы определенного класса и снова вывести их.
.forEach(
    /*Теперь будут выведены только элементы div с классом div2 */
    function f(x) {
        console.log(x);
    }
});
```

## Стиль

Раз есть возможность использовать синтаксис CSS для получения списка узлов, вполне может появиться желание выполнять операции над их стилями. Именно по этой причине NodeList снабжен несколькими методами, которые могут помочь в выполнении этой работы. Особенно примечательным является метод style класса NodeList, так как он может использоваться и как метод чтения, и как метод записи, – в зависимости от того, передается ли ему второй аргумент. Такое его поведение напоминает функцию dojo.style.

Чтобы освежить в памяти, как работает функция dojo.style, вспомните, что вызов dojo.style(someNode, "margin") вернет значение ширины поля узла DOM, а вызов dojo.style(someNode, "margin", "10px") установит ширину поля равной 10 пикселям.

Манипулирование стилями с помощью NodeList выполняется точно так же, за исключением того, что теперь нет никакой необходимости указывать узел в первом параметре. Как и другие методы класса NodeList, этот метод применяется к каждому узлу в списке:

```
// вариант на основе функции dojo.style...
var a = [];

/* здесь создается массив с узлами */

// выполнить обход узлов и применить стиль
dojo.forEach(a, function(x) {
    dojo.style(x, "margin", "10px");
});

//вариант на основе NodeList...
dojo.query( /* некоторый критерий */ )
.style("margin", "10px");
```

Объект NodeList также включает методы addClass и removeClass для добавления и удаления классов, которые тоже очень похожи на одноименные функции dojo.addClass и dojo.removeClass. Таким образом, существует возможность вручную манипулировать свойствами стиля

элементов с помощью метода style или явно добавлять и удалять классы с помощью методов addClass и removeClass. Следует заметить, что метод style особенно удобен в случае отсутствия класса, который позволяет добиться желаемого результата, тогда как методы addClass и removeClass удобно применять, когда необходимые классы уже существуют и остается только переключать их. Как и в случае с методом style, синтаксис использования этих методов вполне предсказуем:

```
dojo.query("span.foo", someDomNode).addClass("foo").removeClass("bar");
dojo.query("#bar").style("color","green");
```

## Размещение

Не удивительно, что в состав NodeList включено несколько методов управления размещением узлов на странице. Среди них имеется метод coords, который, как и одноименная функция dojo.coord, возвращает массив с координатами для каждого узла в списке. А метод place объекта NodeList напоминает функцию dojo.place тем, что позволяет вставить весь список NodeList в определенное место в дереве DOM.

При этом метод addContent не имеет аналога ни в одной из библиотек инструментального набора. Он обеспечивает возможность добавлять узлы или текстовые строки в относительное местоположение для каждого узла в списке NodeList.

Ниже приводится пример использования метода addContent для вставки текстовой строки (которая оформлена как элемент span) после каждого контейнера в странице. Этот пример может быть полезен, в частности когда на экране отображается стопка контейнеров с вкладками:

```
/* Добавить текст после каждого контейнера с классом  pageContainer */
var nl = dojo.query("div.pageContainer").addContent("footer goes here!",
                    "after");
```

Помня, что метод place помещает в страницу все элементы из списка NodeList относительно другого узла, для вставки всего списка в контейнерный узел, идентифицируемый значением debugPane в атрибуте id, можно сделать следующее:

```
var nl = dojo.query("div.someDebugNodes").place("#debugPane", "last");
```

Метод coords, как и одноименная функция, возвращает объект, содержащий пары ключ/значение, представляющие координаты каждого элемента в NodeList. Запомните, что объект, возвращаемый методом coords, включает значения смещений от левого и верхнего краев, ширину и высоту, а также абсолютные координаты x и y, которые могут быть преобразованы в относительные координаты в окне броузера.

Результатом работы метода coords является массив, а не NodeList. Проверьте результат работы следующего фрагмента в консоли Firebug и убедитесь в этом лично:

```
dojo.forEach(
    dojo.query("div").coords(),
    function(x) { console.log(x); }
);
```

Объект NodeList обладает одним уникальным методом размещения, аналога которому нет ни в одной библиотеке Dojo, – это метод orphan, который применяет простой фильтр (единственный селектор CSS; использование оператора «запятая» не разрешено) к каждому элементу в списке и каждый *дочерний элемент*, соответствующий критерию, удаляется из дерева DOM. Эти удаленные, или «осиротевшие» (orphaned), дочерние элементы возвращаются в виде нового списка NodeList. Метод orphan часто используется для удаления узлов из дерева DOM; и этот способ более устойчив к ошибкам, чем использование функций доступа к DOM, когда типичное удаление узла foo выглядит примерно так:

```
foo.parentNode.removeChild(foo).
```

Например, чтобы в дереве DOM удалить из элементов span все дочерние гиперссылки и получить их в виде списка NodeList, достаточно произвести такой вызов метода:

```
var nl = dojo.query("span > a").orphan()
```

Селектор > требует наличия пробелов, то есть слева и справа от селектора должны стоять пробелы.

Метод adopt по существу представляет собой противоположность методу orphan, то есть он позволяет вставлять элементы в дерево DOM. Функция настолько гибкая, что позволяет ей передавать определенный узел DOM, строку критерия или список NodeList. Вставляемые узлы размещаются относительно *первого элемента*, когда в качестве аргумента методу adopt передается список NodeList. Во втором аргументе передается обычная информация о позиционировании (first, last, after или before):

```
var n = document.createElement("div");
n.innerHTML="foo";
dojo.query("#bar").adopt(n, "last");
```

## Пакетная обработка событий DOM

Привыкнув к мысли, что с помощью NodeList можно реализовать практически все, что угодно, вы наверняка не слишком удивитесь тому, что существует возможность определить для группы узлов реакцию на такие события DOM, как потеря фокуса ввода, перемещение указателя мыши и нажатие клавиш. Выполнение некоторых действий в ответ на одно или более событий является таким обыденным делом, что NodeList предоставляет встроенный метод для упрощения решения этой задачи.

С помощью NodeList можно реализовать пакетную обработку следующих событий DOM:

- onmouseover
- onmouseenter
- onmousedown
- onmouseup
- onmouseleave
- onmouseout
- onmousemove
- onfocus
- onclick
- onkeydown
- onkeyup
- onkeypress
- onblur

В качестве примера рассмотрим случай захвата событий перемещения указателя мыши над определенным элементом. Для этого достаточно просто определить функцию обработки события onmouseover, как показано ниже:

```
dojo.query(".foobar").onmousemove(
  function(evt) {
    console.log(evt);// здесь можно реализовать что-нибудь более интересное!
  }
);
```

Объекты событий, доступные через методы модели событий DOM Event, являются стандартизованными, потому что подключение обработчиков выполняется внутри с помощью функции dojo.connect. Модель событий, реализованная в виде dojo.connect, стандартизована в соответствии со спецификацией консорциума W3C.

В настоящее время нет прямого способа управлять или разрывать соединения, созданные с помощью метода connect объекта NodeList, хотя в одной из будущих версий 1.x такая возможность может появиться. Если вам недостаточно того, что соединения разрываются автоматически во время выгрузки страницы, то можно воспользоваться обычной функцией dojo.connect внутри метода forEach класса NodeList при условии, что у вас действительно есть веские причины, вынуждающие вас самостоятельно управлять соединениями.

Например, когда необходимо вручную управлять соединениями из предыдущего примера, это можно сделать примерно следующим образом:

```
var handles =
    dojo.query("a").map(function(x) {
        return dojo.connect(x, "onclick",
            function(evt) { /* ... */ });
    });
```

```
/* Где-то дальше в приложении... */
dojo.forEach(handles, function(x) {
    dojo.disconnect(x);
});
```

## Анимация

Возможно, для начала вам достаточно просто бегло просмотреть этот раздел, чтобы затем вернуться к нему вновь, после того как вы прочитаете главу 8, где дается полный охват тем анимации содержимого.

Создание анимационных эффектов средствами DHTML часто воспринималось как утомительное занятие, да, собственно, так оно и есть. Однако NodeList делает решение этой задачи таким же простым, как и все остальное, что можно сделать с помощью NodeList. С точки зрения разработки приложений это означает, что эффект исчезновения реализуется до смешного просто, а кроме того, с помощью метода _Animation.animateProperty легко можно реализовать гораздо более сложные эффекты.

Имя упомянутого выше объекта _Animation начинается с символа подчеркивания. В данном конкретном случае начальный символ подчеркивания означает, что его интерфейс еще не окончательный и вообще объект _Animation должен рассматриваться как нечто, еще не готовое к эксплуатации. Сведения, представленные в этом разделе, относятся к версии Dojo 1.1, и интерфейс _Animation считается достаточно устойчивым, но в будущих версиях Dojo он может измениться.

Методы, перечисленные в табл. 5.3, представляют анимационные эффекты, доступные в настоящее время, но чтобы воспользоваться ими, необходимо явно подключить ресурс вызовом dojo.require("dojo.NodeList-fx"). Каждый из этих методов принимает ассоциативный массив пар ключ/значение, в котором определяются значения таких параметров, как продолжительность анимации, информация о позиции, цвет и т. д.

*Таблица 5.3. Расширения NodeList для воспроизведения анимационных эффектов*

| | |
|---|---|
| fadeIn | Воспроизводит эффект проявления каждого элемента в списке. |
| fadeout | Воспроизводит эффект растворения каждого элемента в списке. |
| wipeIn | Воспроизводит эффект появления каждого элемента в списке. |
| wipeout | Воспроизводит эффект исчезновения каждого элемента в списке. |

| | |
|---|---|
| slideTo | Передвигает каждый элемент в списке в определенную позицию. |
| animateProperties | Воспроизводит анимационный эффект для каждого элемента в списке, используя указанные свойства. |
| anim | Похож на метод animateProperties за исключением того, что возвращает объект анимационного эффекта, который уже воспроизводится. За дополнительной информацией обращайтесь к описанию функции dojo.anim. |

Вы, вероятно, и до этой книги считали, что анимационные эффекты стоят того, чтобы с ними разобраться. Dojo делает возможность воспроизведения анимационных эффектов чрезвычайно простым делом. Как и в любом другом случае, вы можете просто открыть консоль Firebug и приступить к экспериментам. Для начала можно поэкспериментировать с простым эффектом растворения, как показано ниже:

```
dojo.require("dojo.NodeList-fx");
//После того как ресурс NodeList-fx будет загружен...
dojo.query("p").fadeOut().play()
```

Затем, когда вы будете готовы приступить к изучению более сложных анимационных эффектов, добавьте несколько пар ключ/значение в ассоциативный массив и посмотрите, что произойдет:

```
dojo.require("dojo.NodeList-fx");
//После того как ресурс NodeList-fx будет загружен...
dojo.query("div").animateProperty({
    duration: 5000,
    properties: {
        color: {start: "black", end: "green"}
    }
}).play();
```

Обратите внимание, что в качестве результата различные методы анимационных эффектов возвращают объект _Animation, а метод play — это стандартный механизм его активации.

# Создание расширений для NodeList

Хотя имеющийся набор встроенных методов NodeList очень функционален, но пройдет совсем немного времени, и вы обнаружите, что вам очень не хватает какого-нибудь особенного метода. К счастью, чтобы расширить возможности NodeList, нужно приложить совсем немного усилий. Рассмотрим следующий случай использования функции query для получения значения свойства innerHTML каждого элемента в списке NodeList:

```
dojo.query("p").map(function(x) {return x.innerHTML;});
```

По сравнению со случаем, когда вам пришлось бы разрабатывать собственное решение этой задачи, вы получили достаточно краткое решение, но это решение можно упростить еще больше, использовав более краткий синтаксис «функций в виде строк», – внеся следующее усовершенствование:

```
dojo.query("p").map("return item.innerHTML;"); //Используется специальный
                                               //идентификатор item
```

Определенно, это – усовершенствование, но как вам кажется – может ли ваш программный код выглядеть еще более удобочитаемым *и еще более кратким*? Взгляните на следующее расширение к `NodeList`, которое воплощает отображение в более удобочитаемую и элегантную функцию, имя которой совершенно отчетливо сообщает о выполняемом ею действии:

```
//Расширить прототип объекта NodeList новой функцией
dojo.extend(dojo.NodeList, {
    innerHTML : function() {
        return this.map("return item.innerHTML");
    }
});

//Вызов новой функции
dojo.query("p").innerHTML();
```

Что действительно очень удобно при расширении возможностей `NodeList`, так это отсутствие необходимости слишком далеко забегать вперед при планировании – вы можете одновременно существенно упрощать структуру приложения и делать его менее сложным в сопровождении.

При создании подобных расширений рекомендуется использовать модульный подход, суть которого заключается в создании подмодуля с именем *ext-dojo* с файлом ресурса *NodeList.js* внутри, чтобы инструкция `dojo.require` была кристально ясной для тех, кто будет читать ваш программный код. В конечном счете, от такого решения выиграют все. В окончательном варианте порядок использования вашего расширения может выглядеть примерно так, как показано ниже:

```
/* ... */
dojo.require("dtdg.ext-dojo.NodeList");

/* ...*/

dojo.query("p").innerHTML();
```

Очевидно, если соблюдать точность во всех мелочах, вы могли бы назвать свой файл ресурса *NodeList-innerHTML.js*; делайте так, как сочтете удобным, – при условии, что от этого не возникнет противоречий.

# Модуль Behavior

В состав библиотеки Core входит легковесное расширение, построенное поверх функции query, которое обеспечивает возможность отделения событий и манипулирования элементами в дереве DOM от разметки HTML с помощью модуля behavior. Возможно, на первый взгляд это не совсем очевидно, но возможность определять *поведение* (*behavior*) узлов независимо от разметки может обеспечить приложению высокую гибкость. Например, это позволяет упростить решение таких задач, как связывание обработчиков щелчка мышью со всеми якорными элементами без необходимости знать, где они находятся и сколько якорных элементов имеется. Используя селекторы CSS, которые были приведены в табл. 5.1, можно выделить узлы и присоединить к ним заданное поведение, – так что возможности становятся поистине неограниченными.

В настоящее время прикладной интерфейс модуля содержит две функции – метод add позволяет составлять коллекции из свойств поведения и метод apply запускает поведение:

```
dojo.behavior.add(/*Object*/ behaviorObject)
dojo.behavior.apply()
```

Метод add используется, чтобы связать новое свойство поведения с совокупностью узлов DOM, но это свойство не будет реализовано, пока не будет вызван метод apply. Одной из причин такой двухступенчатой организации процесса является то, что выполнение множественных вызовов метода add до выполнения заключительного вызова метода apply сводится к соответствующему числу асинхронных взаимодействий, описывавшихся в главе 4.

В главе 4 была преведена структура данных с именем Deferred, которая является основой подсистемы ввода-вывода инструментария Dojo. Объекты Deferred представляют собой средство асинхронного выполнения действий и обеспечивают возможность последовательного применения множества функций обратного вызова и функций обработки ошибок. После знакомства с принципами использования объектов Deferred смысл предоставления двух отдельных функций add и apply должен быть для вас очевиден.

Объект, который передается методам add и apply, обладает очень высокой гибкостью и может принимать самые разные формы. В двух словах: объект поведения содержит пары ключ/значение, которые отображают селекторы CSS на объекты, поставляющие обработчики событий DOM. Обработчики событий DOM также представляют собой пары ключ/значение. Однако, прежде чем перейти к примерам, просмотрите содержимое табл. 5.4, где коротко описываются возможные варианты.

*Таблица 5.4. Особенности объекта, описывающего поведение*

| Ключ | Значение | Комментарий |
|---|---|---|
| Селектор (`String`) | `Object` | Значение типа `Object` должно содержать пары ключ/значение, которые должны отображать имена событий DOM или специальный идентификатор `found` на обработчики событий или на темы широковещательных сообщений.<br>Например:<br>`{`<br>`onclick : function(evt) {/*...*/},`<br>`onmouseover : "/dtdg/foo/moveover",`<br>`found : function(node) {/*...*/},`<br>`found : "/dtdg/bar/found"`<br>`}`<br>В случае использования в качестве значения темы широковещательного сообщения подписавшемуся обработчику события передается стандартизованный объект события.<br>В случае использования в качестве значения обработчика события функции передается стандартизованный объект события.<br>В случае использования в качестве ключа специального идентификатора `found` либо обработчику передается соответствующий узел, либо выполняется публикация сообщения с указанной темой. |
| Селектор (`String`) | `Function` | Для каждого узла, соответствующего селектору, обработчику передается сам узел в качестве аргумента. |
| Селектор (`String`) | `String` | Для каждого узла, соответствующего селектору, выполняется публикация сообщения с указанной темой. Обработчику, подписавшемуся на получение сообщений с этой темой, передается сам узел. |

Не забывайте, что ключи в значениях типа `Object` должны иметь тип `String`, когда это требуется синтаксисом селектора CSS. Например, объект поведения `{div : function(evt) {/*...*/}` является корректным, тогда как `{#foo : "/dtdg/foo/topic"}` содержит ошибку, потому что `#foo` не является допустимым идентификатором.

Найдите время на изучение примера 5.1, который иллюстрирует некоторые особенности в действии.

*Пример 5.1. Пример dojo.behavior в действии*

```
<html>
    <head>
        <title>Fun with Behavior!</title>

        <link rel="stylesheet" type="text/css"
              href="http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css" />
        <script
            type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js"
            djConfig="isDebug:true"
        ></script>

        <script type="text/javascript">
            dojo.require("dojo.behavior");

            dojo.addOnLoad(function() {
                /* Передать объект, описывающий поведение в dojo.behavior.
                   Этот объект автоматически будет добавлен
                   сразу после загрузки страницы */
                dojo.behavior.add({

                    /* Объект с описанием поведения может содержать
                       в качестве ключей любые комбинации селекторов CSS,
                       которые могут отображаться на один тип поведения
                       или на совокупность особенностей поведения */

                    /* Отображение ключей на функции эквивалентно
                       отображению {found : function(node) { ... } } */
                    ".container" : function(node) {
                        //применить некоторый стиль

                        dojo.style(node, {
                            border : "solid 1px",
                            background : "#eee"
                        });
                    },

                   /*Отобразить ключ на совокупность особенностей поведения*/
                    "#list > li" : {
                       /* Для событий DOM действует аналогично dojo.connect,
                          позволяя выполнять обработку события */
                        onmouseover : function(evt) {dojo.style(evt.target,
                            "background", "yellow");},
                        onmouseout : function(evt) {dojo.style(evt.target,
                            "background", "");},

                       /*Строковые значения публикуются как темы сообщений*/
                        onclick : "/dtdg/behavior/example/click",

                        /* "found" - это универсальный обработчик,
                           позволяющий манипулировать узлом */
                        found : function(node) {dojo.style(node, "cursor",
                            "pointer")}
```

```
                    }
                });

                /* Где-то в другом месте... производится подписка... */
            dojo.subscribe("/dtdg/behavior/example/click", function(evt) {
                    console.log(evt.target.innerHTML, "was clicked");
                });
            });
        </script>
    <head>

    <body>
        <div class="container" style="width:300px">
            Grocery List:
            <ul id="list">
                <li>Bananas</li>
                <li>Milk</li>
                <li>Eggs</li>
                <li>Orange Juice</li>
                <li>Frozen Pizzas</li>
            </ul>
        </div>
    </body>
</html>
```

Как демонстрирует этот пример, любое поведение, которое вы определяете до того, как страница загрузится полностью, вступает в силу автоматически. Однако, когда поведение определяется после загрузки страницы, сначала нужно добавить поведение с помощью метода `add`, а затем применить его с помощью метода `apply`. В следующем дополнительном фрагменте к списку элементов добавляется *другой* обработчик события `onclick`:

```
dojo.behavior.add({
    "#list > li" : {
        onclick : "/dtdg/behavior/example/another/click"
    }
});

dojo.behavior.apply();

dojo.subscribe("/dtdg/behavior/example/another/click", function(evt) {
    console.log("an additional event handler...");
});
```

Одно из ключевых наблюдений, которые вы должны были сделать, заключается в том, насколько независимо фактическое описание *поведения* узлов от самой разметки. Вы просто пользуетесь преимуществом, которое обеспечивается за счет создания соединения с помощью функции `apply` модуля `behavior`: любое поведение, которое вы определяете, добавляется поверх существующего. Другими словами, новое поведение не затирает существовавшее поведение, благодаря чему вы полу-

чаете возможность добавлять новые уровни поведения, обработка которых будет выполняться без вашего вмешательства.

# В заключение

После прочтения этой главы вы должны:

- Уметь использовать `dojo.query` для организации поиска узлов в странице
- Понимать базовый синтаксис селекторов
- Иметь представление об объекте `NodeList` и знать аналоги некоторых функций инструментария, таких как функции для работы с массивами
- Уметь передавать результаты по цепочке методов `NodeList` для более быстрой и более очевидной обработки элементов DOM
- Знать о возможности использования средства `NodeList` вместо других утилит в инструментарии
- Уметь использовать `NodeList` для размещения узлов DOM, воспроизводить анимационные эффекты, устанавливать соединения и управлять стилями
- Понимать значимость возможности расширения `NodeList` собственными реализациями операций, которые позволяют минимизировать усилия по обработке результатов, получаемых от функции `dojo.query`
- Знать о преимуществах отделения событий DOM от разметки HTML и понимать, как этого можно добиться с помощью модуля `behavior`

В следующей главе рассматриваются проблемы интернационализации.

# 6

# Интернационализация (i18n)

В этой главе дается краткий обзор инструментов Dojo, предназначенных для интернационализации модулей. Ключевыми темами главы будут порядок создания определения региональных настроек и использование возможностей библиотеки Core для представления дат, денежных сумм и чисел. Для тех, кто еще не знает, – слово «internationalization» обычно сокращается до *i18n*, просто потому что это слово слишком длинное, чтобы вводить его всякий раз, когда оно потребуется. Данное сокращение образовано первой и последней буквами, а число 18 обозначает, что имеется еще 18 букв между ними.

## Введение

Если вам повезет, и разработанное вами приложение станет пользоваться некоторой популярностью, вам наверняка потребуется обеспечить поддержку более чем одного языка человеческого общения. Библиотека Dijit, которая будет рассматриваться во второй части книги, уже содержит параметры интернационализации для некоторых языков, и, тем не менее определенные действия над вашими модулями и виджетами потребуются. К счастью, инструментальный набор обеспечивает все необходимое для организации поддержки более чем одного языка единообразным способом, позволяя вам избежать мороки с изобретением своей собственной системы отображения лексем. Инструментарий Dojo сам управляет процессом загрузки, поэтому вам также не придется размышлять над вопросом оптимизации загрузки. Кроме того, дополнительные утилиты обеспечивают поддержку таких распространенных операций, как форматирование чисел, денежных сумм и многое другое.

Следует заметить, что средства i18n технически являются частью библиотеки Core, а не Base. В свою очередь, сборка XDomain включает мо-

дуль dojo.i18n как часть *dojo.xd.js*, добавляя 2 Кбайта программного кода, предназначенного решать проблемы загрузки региональных настроек. Но независимо от этого, вам по-прежнему необходимо добавлять инструкцию dojo.require("dojo.i18n"), чтобы использовать эти средства.

Интернационализация собственного модуля выполняется достаточно просто, потому что таблица строк хранится отдельно, в специальном каталоге *nls*, в каталоге модуля, где находятся исходные файлы JavaScript. Имя *nls* происходит от *native language support* (поддержка родного языка). Сам каталог *nls* содержит серию подкаталогов с переводами, имена которых соответствуют сокращенным названиям отдельных языков, определенным в соответствии с RFC 3066 («Tags for the Identification of Languages»).[1]

Например, общее обозначение английского языка – *en*, а диалекта английского языка, на котором говорят в Соединенных Штатах, – *en-us*. Общее обозначение испанского языка – *es*. В процессе автоматической самонастройки Dojo запросит у броузера информацию об используемом языке и сохранит ее в памяти приложения. Ее можно увидеть, если ввести команду dojo.locale в консоли Firebug. Значение dojo.locale используется инструментарием Dojo, чтобы определить наиболее подходящий перевод при загрузке модуля, для которого выполнена интернационализация.

# Интернационализация модуля

Предположим, что у вас нашлось время и вы расширили модуль с волшебным джинном (раздел «Создание примера модуля волшебного джинна» в главе 2), создав модуль psychic. Допустим, что по умолчанию вы используете английский язык и вы решили, что первым дополнительным языком, поддержку которого требуется обеспечить, является русский язык.[2]

## Расположение файлов на диске

Как и любой другой модуль, ваш модуль psychic должен быть представлен простым файлом с исходными текстами, находящимся в типичном дереве каталогов:

```
dtdg/
  psychic/
    Psychic.js /* Самое основное находится здесь */
```

[1] *http://www.ietf.org/rfc/rfc3066.txt*.

[2] В оригинале говорится об испанском языке, но, как мне кажется, нашему читателю будет гораздо интереснее узнать, как выполняется интернационализация для русского языка. – *Прим. перев.*

Неудивительно, что ваш невероятный модуль psychic обладает возможностью предсказывать будущее. Пользователи модуля могли бы добавлять его в свои страницы и использовать его возможности, как показано ниже:

```
<script type="text/javascript">
    dojo.require("dtdg.psychic");
    dojo.addOnLoad(function() {
        dtdg.psychic.predictFuture();
    });
</script>
```

В функции predictFuture может находиться много чего, *по-настоящему* волшебного, но сейчас нас интересует та ее часть, где строковое значение фактически выводится на экран, потому что именно в этот момент и выполняется интернационализация. Как оказывается, вывод информации в нашем модуле реализован, как показано ниже:

```
dojo.byId("reading").innerHTML = predictFuture( /* волшебство */ );
```

На первом этапе интернационализации мы будем считать, что мы имеем дело просто с английским и русским языком, игнорируя конкретные диалекты. С учетом этого решения содержимое каталога *nls* могло бы выглядеть, как показано ниже:

```
dtdg/
  psychic/
    Psychic.js
    nls/
      readings.js /* Английский перевод по умолчанию */
      ru/
        readings.js /* Русский перевод */
      en/
        /* Каталог с английским переводом по умолчанию пуст, поэтому
           Dojo будет искать файл выше, в каталоге nls/ */
```

В соответствии с соглашениями каждый из файлов *.js*, содержащий информацию о переводе, называется *привязкой* (*bundle*). Согласно соглашению, привязка с переводом по умолчанию должна находиться в самом каталоге *nls*, а не в каталоге для данного конкретного языка. Это соглашение основано на том, что всегда удобнее иметь перевод по умолчанию в каталоге *nls*, как в наиболее логичном местоположении, и нет смысла создавать точную копию привязки в ее собственном каталоге (*en*, в данном случае), потому что в противном случае этот файл превратился бы еще в одну вещь, за которой необходимо следить.

## Определение таблиц строк

Ниже приводятся выдержки из каждого файла *readings.js*, где демонстрируются некоторые строки, являющиеся частью окончательного перевода.

Сначала файл *readings.js* с переводом по умолчанию:

```
{
/* ... */
reading101 : "You're a Libra, aren't ya darling?",
reading102: "Can you please tell me your first name only, and your
             birthday please?",
reading103: "Yep, that's the Daddy."
/* ... */
}
```

А теперь файл *ru/readings.js*:

```
{
/* ... */
reading101 : "Вы – Весы, не правда ли, мой друг?",
reading102: "Пожалуйста, назовите только свое имя и дату рождения?",
reading103: "Да, это папа."
/* ... */
}
```

Одна из замечательных особенностей процедуры интернационализации приложений с применением средств Dojo заключается в простой форме представления списка лексем.

## Объединим все вместе

Пришло время объединить вместе полученные сведения и продемонстрировать, насколько просто организовать поддержку нескольких языков, но сначала рассмотрим функции, имеющие отношение к интернационализации, которые перечислены в табл. 6.1.

*Таблица 6.1. Функции локализации*

| Имя | Комментарий |
|---|---|
| `dojo.i18n.getLocalization(/*String*/moduleName, /*String*/bundleName, /*String?*/locale)` | Возвращает объект, содержащий локализацию для заданного ресурса привязки в пакете. По умолчанию аргумент `locale` соответствует значению `dojo.locale`, однако, передача явного значения позволит использовать конкретный перевод. |
| `dojo.i18n.normalizeLocale(/*String?*/locale)` | Возвращает название языка в канонической форме. |
| `dojo.requireLocalization(/*String*/moduleName, /*String*/bundleName, /*String?*/locale)` | Загружает ресурсы с переводами точно так же, как это делает `dojo.require` при загрузке модулей. Примечательно, что эта функция находится в библиотеке Base, а не является частью модуля `i18n` из библиотеки Core. |

Если раньше для получения значения `reading102` требовалось обратиться к значению хеш-функции, например `psychic.reading102`, то теперь это можно сделать с помощью инструментария. Если имеется перевод для языка, используемого пользователем, вся интернационализация будет выполняться «сама по себе». Поиск строк в различных переводах выполняется очень просто, как показано в следующем фрагменте:

```
/* Сначала нужно подключить утилиты i18n... */
dojo.require("dojo.i18n");

/* Затем подключить различные переводы */
dojo.requireLocalization("psychic", "readings");

function predictFuture() {

    /* Где-то в глубине функции predictFuture... */
    var future= dojo.i18n.getLocalization("psychic", "readings").reading597;
    return future;
}
```

Обратите внимание, что имеется возможность изменить значение `dojo.locale`, чтобы проверить варианты на других языках. Лучше всего изменять это значение в блоке `djConfig`. Ниже показано, как можно проверить русский перевод при использовании локальной версии Dojo:

```
<head>
    <script type="text/javascript" src="your/path/to/dojo.js"
    djConfig="dojo.locale:'ru'">
</script>
</head>

<!--
    Теперь все интернационализированные модули будут использовать русский язык
-->
```

Точно так же, как и в случае с любым другим модулем или ресурсом, не следует вызывать `dojo.i18n.getLocalization` как часть определения свойства объекта. Вместо этого функция `dojo.i18n.getLocalization` должна вызываться в блоке `dojo.addOnLoad`:

```
dojo.addOnLoad(function() {
    //Вернет локализованный объект
    var foo = {bar : dojo.i18n.getLocalization( /* ...*/)}
});
```

Имеется нюанс, о котором следует знать, – когда языком по умолчанию является английский и вы тестируете русский перевод, загружаются сразу две привязки: *nls/readings.js* и *nls/ru/readings.js*. То есть привязка по умолчанию, находящаяся в каталоге *nls/*, загружается всегда. Чтобы убедиться в наличии этой особенности, можно воспользоваться инструментом *Net* в Firebug.

Хотя в данном конкретном примере не использовались диалекты ни одного из вовлеченных языков, однако, следует заметить, что чаще всего при загрузке привязок во внимание принимаются именно диалекты. Например, если вы используете язык en-us и в наличии имеется привязка *en-us*, то Dojo попытается загрузить обе привязки, *en-us* и *en*, объединив их в единую коллекцию, обращение к которой будет производиться различными вызовами `dojo.i18n.getLocalization`. При работе, например, над английским переводом, необходимо как можно больше информации, общей для всех диалектов, поместить в привязку *en*, а затем переопределять или дополнять недостающие элементы, характерные для конкретного диалекта, такого как *en-us*.

### Использование инструментов сборки для повышения производительности

Последнее, но от этого не менее важное, замечание об интернационализации: инструментальный набор Dojo содержит в библиотеке Util инструменты сборки, которые при сборке специализированной версии модуля автоматически могут позаботиться о бесчисленных деталях, чтобы минимизировать число синхронных вызовов и избыточность данных. На первый взгляд это может показаться несущественным, но инструменты сборки могут объединить воедино множество маленьких файлов ресурсов и тем самым помочь избежать дополнительных попыток отыскать их и избежать задержек, связанных с этим. Эти инструменты действительно могут существенно повысить скорость загрузки страницы. Обсуждение библиотеки Util и инструментов сборки приводится в главе 16.

## Даты, числа и денежные суммы

Дополнительные средства библиотеки Core обеспечивают поддержку манипулирования интернационализированными представлениями дат, чисел и денежных сумм с помощью модулей `dojo.date`, `dojo.number` и `dojo.currency`, соответственно. Во второй части книги вы узнаете, что библиотека Dijit очень широко использует эти модули, чтобы обеспечить улучшенную поддержку наиболее часто используемых виджетов. А здесь дается краткий обзор этих возможностей.

### Даты

В табл. 6.2 дается краткий обзор модуля `dojo.date`. Здесь вы найдете поистине бесценные возможности, если вам когда-нибудь придется столкнуться с обработкой встроенного объекта `Date`.

В версии 1.1 инструментария функция `getTimeZoneName` не локализована.

*Таблица 6.2. Перечень функций модуля date*

| Имя | Тип возвращаемого значения | Комментарий |
|---|---|---|
| dojo.date.getDaysInMonth (/*Date*/date) | Integer | Возвращает число дней в месяце для указанной даты. |
| dojo.date.isLeapYear (/*Date*/date) | Boolean | Возвращает true, если год для указанной даты является високосным. |
| dojo.date.getTimezoneName (/*Date*/date) | String | Возвращает информацию о часовом поясе, которая определяется броузером. Объект Date необходим, потому что часовой пояс может меняться из-за перехода на летнее или зимнее время. |
| dojo.date.compare(/*Date*/ date1, /*Date*/ date2, /*String?*/ portion) | Integer | Возвращает 0, если аргументы равны. Возвращает положительное число, если date1 > date2. Возвращает отрицательное число, если date1 < date2. По умолчанию сравниваются и даты и время, однако в аргументе portion можно указать значение "date" или "time", чтобы сравнению подвергались только дата или только время суток, соответственно. |
| dojo.date.add(/*Date*/date, /*String*/ interval, /*Integer*/ amount) | Date | Обеспечивает удобный способ добавить интервал времени к объекту Date, указав в аргументе amount число единиц, а в аргументе interval — единицы измерения. Допустимыми единицами измерения являются: "year", "month", "day", "hour", "minute", "second", "millisecond", "quarter", "week" и "weekday". |
| dojo.date.difference (/*Date*/date1, /*Date*/ date2, /*String*/ interval) | Integer | Обеспечивает удобный способ вычислить разницу между двумя датами в указанных единицах измерения. Допустимыми единицами измерения являются: "year", "month", "day", "hour", "minute", "second", "millisecond", "quarter", "week" и "weekday". |

## Числа

Модуль dojo.number содержит несколько удобных функций, список которых приводится в табл. 6.3 и табл. 6.4 и которые предназначены для преобразования строковых значений в числа, форматирования значений типа Number в соответствии с заданным шаблоном или округления до определенного числа знаков после десятичной точки.

*Таблица 6.3. Параметры форматирования в модуле number, используемые функциями dojo.number.format и dojo.number.parse, которые приводятся в табл. 6.4*

| Имя | Тип | Комментарий |
|---|---|---|
| pattern | String | Может использоваться для переопределения шаблона форматирования. |
| type | String | Формат представления, основанный на региональных настройках. Допустимыми значениями являются: "decimal", "scientific", "percent", "currency" и "decimal". Значение по умолчанию: "decimal". |
| places | Number | Содержит число отображаемых десятичных знаков. Переопределяет информацию, которая может иметься в параметре pattern. |
| round | Number | Определяет параметры округления на основе множителя. Например, при значении 5 округление будет произведено до ближайшего числа с точностью 0.5, а при значении 0 – до ближайшего целого числа. |
| currency | String | Код валюты в соответствии со стандартом ISO4217. Например, код "USD" обозначает доллар США. |
| symbol | String | Локализованный символ, обозначающий валюту. |
| locale | String | Позволяет задать региональные настройки, которые определяют правила форматирования. |

*Таблица 6.4. Перечень функций модуля number*

| Имя | Тип возвращаемого значения | Комментарий |
|---|---|---|
| dojo.number.format (/*Number*/value, /*Object*/options) | String | Преобразует объект Number в строку, используя параметры региональных настроек. Аргумент options может принимать одно из значений, перечисленных в табл. 6.3. |
| dojo.number.round (/*Number*/value, /*Number*/places) | Number | Округляет число до заданного аргументом places числа знаков после десятичной точки. |

*Таблица 6.4 (продолжение)*

| Имя | Тип возвращаемого значения | Комментарий |
|---|---|---|
| dojo.number.parse (/*String*/value, /*Object*/options) | Number | Преобразует правильно сформированную строку в объект Number, используя параметры региональных настроек. Допустимыми значениями аргумента options являются значения, перечисленные в табл. 6.3: pattern, type, locale, strict и currency. |

## Денежные суммы

Модуль dojo.currency, описанный в табл. 6.5 и табл. 6.6, по своим возможностям напоминает модуль dojo.number, – в том смысле, что обеспечивает форматирование числовых значений, только на этот раз с учетом указанного кода валюты в соответствии со стандартом ISO4217.[1]

*Таблица 6.5. Параметры форматирования в модуле currency, используемые функциями dojo.currency.format и dojo.currency.parse*

| Имя | Тип | Комментарий |
|---|---|---|
| currency | String | Трехсимвольный код валюты в соответствии со стандартом ISO4217. |
| symbol | String | Значение, которое может применяться для переопределения символа валюты, используемого по умолчанию. |
| pattern | String | Используется для переопределения шаблона форматирования по умолчанию. |
| round | Number | Используется для ограниченного округления: -1 означает «без округления», 0 – округление до ближайшего целого числа, а 5 – округление до ближайшего числа с точностью 0.5. |
| locale | String | Переопределяет параметры региональных настроек, которые содержат правила форматирования по умолчанию. |
| places | Number | Содержит число отображаемых десятичных знаков. (По умолчанию определяется типом валюты.) |

[1] *http://en.wikipedia.org/wiki/ISO_4217*. На русском языке: *http://ru.wikipedia.org/wiki/ISO_4217*.

*Таблица 6.6. Перечень функций модуля currency*

| Имя | Тип возвращаемого значения | Комментарий |
|---|---|---|
| `dojo.currency.format (/*Number*/value, /*Object*/options)` | `String` | Преобразует объект `Number` в строку, используя параметры региональных настроек. Аргумент `options` может принимать одно из значений, перечисленных в табл. 6.5. |
| `dojo.currency.parse (/*String*/expression, /*Object*/options)` | `Number` | Преобразует правильно сформированную строку в объект `Number`. Допустимыми значениями аргумента `options` являются значения, перечисленные в табл. 6.5. |

Некоторые инструменты сборки, входящие в состав Dojo, могут использоваться для включения поддержки произвольных региональных настроек и валют, поскольку значительная часть этой работы сопряжена с созданием таблиц для поиска информации. Дополнительные сведения можно найти в файле *util/buildscripts/cldr/README*.

# В заключение

После прочтения этой главы вы должны:

- Уметь выполнять интернационализацию модулей для более чем одного набора региональных настроек
- Знать, что библиотека Core содержит утилиты для работы с денежными суммами, числами и датами, которые могут быть полезны для интернационализации приложения

В следующей главе рассматривается механизм перетаскивания элементов.

# 7

# Перетаскивание элементов

Механизм «перетащил и бросил» (Drag-and-Drop, DnD) может придать вашему приложению функциональные возможности, сближающие их с обычными настольными приложениями и обеспечивающие простоту и удобство использования, которые могут выгодно выделять их среди других приложений. Эта глава подробно рассматривает данный вопрос и предоставляет множество наглядных примеров и исходного программного кода. Вы можете использовать эти примеры, чтобы добавить визуальные эффекты в свои приложения или даже сделать более смелый шаг – например, создать на основе концепций и механизмов, предоставляемых инструментарием Dojo, игры, в которые смогут играть ваши пользователи. В любом случае это очень интересная глава, поэтому начнем.

## Перетаскивание

Операция перетаскивания уже более двух десятилетий является неотъемлемой частью обычных приложений, однако веб-приложения не спешили реализовать эту возможность. Отчасти такая неспешность обусловлена тем, что имеющиеся механизмы манипулирования деревом DOM сами по себе весьма примитивны, а управляемая событиями природа механизма «перетащил и бросил» серьезно осложняет создание универсальной платформы, которая одинаково хорошо работала бы во всех броузерах. К счастью, эти сложности были преодолены в инструментальном наборе Dojo, и теперь Dojo предоставляет средства, которые избавляют от необходимости тратить время и силы на самостоятельную разработку этого механизма.

## Простые перемещения

Для усвоения материала этой главы необходимо иметь некоторые знания о том, как работают CSS. На сайте W3C Schools, по адресу *http://www.w3schools.com/css/default.asp*, имеется учебное руководство по CSS. Кроме того, замечательным настольным руководством может служить книга Эрика Мейера (Eric Meyer) «CSS: The Definitive Guide» (O'Reilly).[1]

Для начала рассмотрим самый простой пример, какой только возможен: попробуем организовать возможность перетаскивания объекта[2] по экрану. Пример 7.1 демонстрирует структуру страницы с разметкой. Особое внимание обратите на выделенные строки, где присутствуют ссылки на класс `Moveable`, особенности которого будут рассматриваться ниже.

*Пример 7.1. Простой перемещаемый объект*

```
<html>
    <head>
        <title>Fun with Moveables!</title>

        <style type="text/css">
            .moveable {
                background: #FFFFBF;
                border: 1px solid black;
                width: 100px;
                height: 100px;
                cursor: pointer;
            }
        </style>

        <script
            type="text/javascript"
            djConfig="parseOnLoad:true,isDebug:true"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js">
        </script>

        <script type="text/javascript">
            dojo.require("dojo.dnd.Moveable");
            dojo.require("dojo.parser");
        </script>
    </head>
    <body>
        <div class="moveable" dojoType="dojo.dnd.Moveable" ></div>
```

[1] Эрик Мейер «CSS – каскадные таблицы стилей. Подробное руководство», 3-е издание. – Пер. с англ. – СПб.: Символ-Плюс, 2008.

[2] В этой главе термин *объект* используется для обозначения узлов дерева DOM. В данном случае этот термин не имеет ничего общего с объектами из объектно-ориентированного программирования.

```
    </body>
</html>
```

Как вы уже наверняка обратили внимание, создание объекта, который можно перемещать по экрану, выглядит очень просто. После подключения ресурса `Moveable` к странице остается лишь с помощью атрибута `dojoType` определить перемещаемый элемент страницы и выполнить парсинг страницы после загрузки, добавив дополнительный параметр в массив `djConfig`. В этой странице действительно нет ничего особенного, за исключением таблицы стилей, которая была создана специально, чтобы перемещаемый узел выглядел несколько иначе, чем просто фрагмент текста, хотя и обычный фрагмент текста можно перетаскивать точно так же.

## Парсер

Парсер является одним из наиболее часто используемых ресурсов библиотеки Core. Однако, т. к. оказывается, что чаще всего он используется для парсинга виджетов на странице, то подробное обсуждение парсера откладывается до главы 11, где будет дано формальное представление библиотеки Dijit. Хотя вам ничто не мешает просмотреть указанную главу прямо сейчас, эта врезка предоставляет тот объем информации, которого вам сейчас будет достаточно для понимания.

Чаще всего парсер используется для поиска и создания экземпляров виджетов, прежде чем будет вызвана функция `addOnLoad`. В поисках виджетов парсер просматривает все теги, выискивая атрибут `dojoType`, где определяется имя ресурса виджета, который требуется загрузить. Все, что для этого необходимо, – подключить парсер с помощью функции `dojo.require`, как любой другой ресурс, добавить в массив `djConfig` директиву `parseOnLoad:true` и включить атрибут `dojoType` во все элементы страницы, которые будут играть роль виджетов. (В действительности парсинг виджетов можно выполнить и вручную, но мы отложим обсуждение этого вопроса до главы 11.)

Листинги с программным кодом, которые будут встречаться в этой главе, используют парсер сходным способом, потому что выполняют очень похожую функцию для реализации механизма «перетащил и бросил». Если говорить точнее, выполняется сканирование страницы в поисках тегов с атрибутом `dojoType`, которые идентифицируют ресурсы механизма перетаскивания, и проделывается вся необходимая работа, чтобы сделать их интерактивными.

Вообще говоря, все, что выполняется парсером во время загрузки страницы, можно выполнить программным способом и после загрузки страницы. Ниже приводится практически тот же самый пример, в котором перемещаемый объект создается программно:

```
        <!-- ... Обрезано ... -->
        <script type="text/javascript">
            dojo.require("dojo.dnd.Moveable");

            dojo.addOnLoad(function() {
                var e = document.createElement("div");
                dojo.addClass(e, "moveable");
                dojo.body().appendChild(e);
                var m = new dojo.dnd.Moveable(e);
            });
        </script>
    </head>
    <body></body>
</html>
```

В табл. 7.1 перечислены методы, необходимые для создания и уничтожения перемещаемых объектов.

*Таблица 7.1. Функции создания и уничтожения перемещаемых объектов*

| Имя | Комментарий |
|---|---|
| `Moveable(/*DOMNode*/node, /*Object*/params)` | Функция-конструктор, идентифицирующая узел, который станет перемещаемым. Аргумент `params` может принимать следующие значения:<br>`handle (String \| DOMNode)`<br>Узел или значение атрибута `id` узла, который можно будет захватить мышью. По умолчанию используется сам перемещаемый узел.<br>`skip (Boolean)`<br>Определяет, следует ли пропустить обычное применение механизма перетаскивания к текстовым элементам форм, которое при нормальных условиях запускается в момент нажатия кнопки мыши (по умолчанию: `false`).<br>`mover (Object)`<br>Конструктор собственного объекта `Mover`.<br>`delay (Number)`<br>Число пикселей задержки перемещения (по умолчанию: 0) |
| `destroy()` | Делает объект неперемещаемым, удаляя все ссылки, которые затем будут утилизированы сборщиком мусора. |

Объект `Mover` представляет собой еще более низкоуровневый механизм перетаскивания, который используется внутри реализации конструктора `Moveable`. Объекты Mover не рассматриваются в этой главе, они были упомянуты лишь для полноты картины.

Давайте возьмем за основу наш предыдущий пример, чтобы продемонстрировать, как можно с помощью параметра `skip` обеспечить возможность редактирования в текстовых элементах форм, создав на экране простой объект, имитирующий наклейку с примечанием, которую можно перемещать и текст в которой можно редактировать. Рабочий программный код представлен в примере 7.2.

*Пример 7.2. Использование функции Moveable для создания имитации наклейки с примечанием*

```
<html>
    <head>
        <title>Even More Fun with Moveables! </title>

        <style type="text/css">
            .note {
                background: #FFFFBF;
                border-bottom: 1px solid black;
                border-left: 1px solid black;
                border-right: 1px solid black;
                width: 302px;
                height: 300px;
                margin : 0px;
                padding : 0px;
            }
            .noteHandle {
                border-left: 1px solid black;
                border-right: 1px solid black;
                border-top: 1px solid black;
                cursor :pointer;
                background: #FFFF8F;
                width : 300px;
                height: 10px;
                margin : 0px;
                padding : 0px;
            }
        </style>

        <script
            type="text/javascript"
            djConfig="parseOnLoad:true,isDebug:true"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js">
        </script>

        <script type="text/javascript">
            dojo.require("dojo.dnd.Moveable");
            dojo.require("dojo.parser");
```

```
        </script>
    </head>
    <body>
        <div dojoType="dojo.dnd.Moveable" skip=true>
            <div class="noteHandle"></div>
            <textarea class="note">Type some text here</textarea>
        </div>
    </body>
</html>
```

Эффект действия параметра `skip` не совсем очевиден, поэтому будет очень поучительно убрать текст `skip=true` из внешнего элемента `DIV`, чтобы лично убедиться, что произойдет, если забыть указать, что элементы форм должны быть пропущены.

Хотя в нашем объекте, имитирующем наклейку, совершенно *необязательно* определять элемент-ручку, который можно захватить мышью, потому что внутренний элемент `div` и является единственной областью, которую можно передвинуть, тем не менее точно такого же эффекта можно было добиться, определив такой элемент: если отметить в объекте `Moveable` элемент, к которому применимо перетаскивание (связать с ним атрибут `handle`), то любой элемент формы, расположенный за его пределами, будет доступен для редактирования. В этом можно убедиться, заменив строки, выделенные в предыдущем примере, следующим фрагментом:

```
<div id="note" dojoType="dojo.dnd.Moveable" handle='dragHandle'>
    <div id='dragHandle' class="noteHandle"></div>
    <textarea class="note">
        This form element can't trigger drag action
    </textarea>
</div>
```

## События, возникающие при перетаскивании

Вероятно, вам потребуется определять момент начала и конца перетаскивания для воспроизведения специальных эффектов, таких как визуальное выделение перетаскиваемого объекта. Определение этих событий легко выполняется с помощью функций `dojo.subscribe` и `dojo.connect`. В примере 7.3 приводится измененная версия примера 7.2.

*Пример 7.3. Соединение и подписка на получение событий перетаскивания*

```
<html>
    <head>
        <title>Yet More Fun with Moveable!</title>

        <style type="text/css">
            .note {
                background: #FFFFBF;
                border-bottom: 1px solid black;
```

```
            border-left: 1px solid black;
            border-right: 1px solid black;
            width: 302px;
            height: 300px;
            margin : 0px;
            padding : 0px;
        }
        .noteHandle {
            border-left: 1px solid black;
            border-right: 1px solid black;
            border-top: 1px solid black;
            cursor :pointer;
            background: #FFFF8F;
            width : 300px;
            height: 10px;
            margin : 0px;
            padding : 0px;
        }
        .movingNote {
            background : #FFFF3F;
        }
    </style>

    <script
        type="text/javascript"
        src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js">
    </script>

    <script type="text/javascript">
        dojo.require("dojo.dnd.Moveable");
        dojo.addOnLoad(function() {
            //создать и сохранить ссылки на перемещаемые объекты
            //для последующего использования.
            var m1 = new dojo.dnd.Moveable("note1",
                                           {handle : "dragHandle1"});
            var m2 = new dojo.dnd.Moveable("note2",
                                           {handle : "dragHandle2"});
            //темы широковещательных сообщений для всех
            //перемещаемых элементов.
            dojo.subscribe("/dnd/move/start", function(node){
                console.log("Start moving", node);
            });
            dojo.subscribe("/dnd/move/stop", function(node){
                console.log("Stop moving", node);
            });

            //подсветить фон перемещаемого объекта...
            //подключить только к перемещаемым объектам.
            dojo.connect(m1, "onMoveStart", function(mover){
                console.log("note1 start moving with mover:", mover);
                dojo.query("#note1 > textarea").addClass("movingNote");
            });
```

```
                dojo.connect(m1, "onMoveStop", function(mover){
                    console.log("note1 stop moving with mover:", mover);
                  dojo.query("#note1 > textarea").removeClass("movingNote");
                });
            });
        </script>
    </head>
    <body>
        <div id="note1">
            <div id='dragHandle1' class="noteHandle"></div>
            <textarea class="note">Note1</textarea>
        </div>
        <div id="note2">
            <div id='dragHandle2' class="noteHandle"></div>
            <textarea class="note">Note2</textarea>
        </div>
    </body>
</html>
```

При вызове функции `dojo.query` следует помнить, что параметр `"#note1 > textarea"` подразумевает узлы `textarea`, являющиеся дочерними по отношению к узлу со значением `"note1"` в атрибуте `id`. Перечень наиболее часто используемых селекторов CSS3, которые допускается передавать функции `dojo.query`, приводится в табл. 5.1.

Обратите внимание, что в программном коде из предыдущего листинга не выполняется непосредственное подключение к узлу. Вместо этого соединение выполняется с элементом `Moveable`, который создается вызовом функции `dojo.dnd.Moveable`.

Как видите, существует возможность подписаться на получение широковещательных сообщений или напрямую подключиться к определенным узлам `Moveable`. В табл. 7.2 приводится перечень событий, для получения которых можно подключаться с помощью функции `dojo.connect`.

Для организации взаимодействий по рассылке можно использовать функцию `dojo.subscribe`, чтобы подписаться на получение сообщений с темами `"dnd/move/start"` и `"dnd/move/stop"`.

*Таблица 7.2. События, возникающие в объектах Moveable*

| Событие | Комментарий |
|---|---|
| `onMoveStart(/*dojo.dnd.Mover*/mover)` | Возникает перед каждым перемещением. |
| `onMoveStop(/*dojo.dnd.Mover*/mover)` | Возникает после каждого перемещения. |
| `onFirstMove(/*dojo.dnd.Mover*/mover)` | Вызывается при самом первом перемещении. Удобно для выполнения действий по инициализации. |

*Таблица 7.2 (продолжение)*

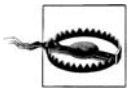
| Событие | Комментарий |
|---|---|
| onMove(/*dojo.dnd.Mover*/mover), (/* Object */ leftTop) | Возникает при каждом изменении позиции элемента в процессе перемещения. По умолчанию сначала возникает событие onMoving, затем производится изменение координат элемента Moveable, а затем происходит событие onMoved. |
| onMoving(/*dojo.dnd.Mover*/mover), (/*Object*/leftTop) | Возникает непосредственно перед событием onMove. |
| onMoved(/*dojo.dnd.Mover*/mover), (/*Object */leftTop) | Возникает сразу после события onMove. |

## Координата Z

Наш рабочий пример про наклейки становится все более и более сложным. Еще одна особенность, которая может стать проблемой, заключается в том, что начальные значения координаты z (z-index) остаются без изменений: одна из наклеек всегда будет выше, а другая всегда ниже. Однако, более естественным было бы, если бы наклейка, выбранная последней, всегда оказывалась выше, то есть имела бы более высокое значение координаты z. К счастью, изменить координату z можно достаточно просто – в функции, которая соединена с событием onMoveStart.

Решение, представленное ниже, требует изменения логики функции addOnLoad и обладает определенным изяществом благодаря применению замыкания для хранения значения локальной переменной с информацией о состоянии вместо явного использования глобальной переменной уровня модуля:

```
dojo.addOnLoad(function() {
    //создать и сохранить ссылки на перемещаемые объекты
    //для последующего использования.
    var m1 = new dojo.dnd.Moveable("note1", {handle : "dragHandle1"});
    var m2 = new dojo.dnd.Moveable("note2", {handle : "dragHandle2"});

    var zIdx = 1; // образует замыкание вместе с этой анонимной функцией

    dojo.connect(m1, "onMoveStart", function(mover){
        dojo.style(mover.host.node, "zIndex", zIdx++);
    });
    dojo.connect(m2, "onMoveStart", function(mover){
        dojo.style(mover.host.node, "zIndex", zIdx++);
    });
});
```



В главе 2 говорилось, что функции dojo.style следует передавать имена свойств, которые используются DOM, а не в таблицах стилей. Например, если попытаться установить свойство с именем "z-index", этот прием работать не будет.

## Ограничение перемещений

Возможность перемещать объекты по экрану без ограничений и с минимальными усилиями – это, конечно, хорошо, но рано или поздно наверняка потребуется определять границы перемещения, возможность перекрытия и задавать другие ограничения. К счастью, реализация механизма «перетащил и бросил» позволяет уменьшить объем программного кода, который обычно приходится писать, чтобы наложить ограничения на возможность перемещения.

В состав модуля dojo.dnd включены три основных средства, позволяющие наложить ограничения на перемещение объектов: можно определить собственную функцию, реализующую ограничения, которая будет динамически вычислять границы допустимой области (объект constrainedMoveable), можно определить статические границы в момент создания перемещаемого объекта (объект boxConstrainedMoveable) и ограничить область перемещения рамками другого, родительского узла (объект parentConstrainedMoveable). Формат представления каждой разновидности допустимой области следует соглашениям, описанным в разделе «Блочная модель» главы 2.

Ниже приводится измененная версия предыдущего примера с наклейками, где демонстрируется наложение ограничений посредством функции constrainedMoveable:

```
<html>
    <head>
        <title>Moving Around</title>
        <style type="text/css">
            .note {
                background: #FFFFBF;
                border-bottom: 1px solid black;
                border-left: 1px solid black;
                border-right: 1px solid black;
                width: 302px;
                height: 300px;
                margin : 0px;
                padding : 0px;
            }
            .noteHandle {
                border-left: 1px solid black;
                border-right: 1px solid black;
                border-top: 1px solid black;
                cursor :pointer;
                background: #FFFF8F;
```

```
            width : 300px;
            height: 10px;
            margin : 0px;
            padding : 0px;
        }
        .movingNote {
            background : #FFFF3F;
        }
        #note1, #note2 {
            width : 302px
        }
    </style>
    <script
        type="text/javascript"
        src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js">
    </script>

    <script type="text/javascript">
        dojo.require("dojo.dnd.Moveable");
        dojo.require("dojo.dnd.move");

        dojo.addOnLoad(function() {
            var f1 = function() {
                //вычисления, определяющие границы допустимой
                //области удерживать элемент note1 не более
                //чем на 50 пикселей правее и ниже узла note2
                var mb2 = dojo.marginBox("note2");
                b = {};
                b["t"] = 0;
                b["l"] = 0;
                b["w"] = mb2.l + mb2.w + 50;
                b["h"] = mb2.h + mb2.t + 50;
                return b;
            }
            var m1 = new dojo.dnd.move.constrainedMoveable("note1",
                {handle : "dragHandle1", constraints : f1,
                 within : true});

            var m2 = new dojo.dnd.Moveable("note2", {handle :
                                            "dragHandle2"});
            var zIdx = 1;

            dojo.connect(m1, "onMoveStart", function(mover){
                dojo.style(mover.host.node, "zIndex", zIdx++);
            });
            dojo.connect(m2, "onMoveStart", function(mover){
                dojo.style(mover.host.node, "zIndex", zIdx++);
            });
        });
    </script>
</head>
<body>
    <div id="note1">
```

```
            <div id='dragHandle1' class="noteHandle"></div>
            <textarea class="note">Note1</textarea>
        </div>
        <div id="note2">
            <div id='dragHandle2' class="noteHandle"></div>
            <textarea class="note">Note2</textarea>
        </div>
    </body>
</html>
```

При вычислении границ допустимой области для объекта `Moveable` необходимо явно определять высоту и ширину самого внешнего контейнерного элемента, который будет перемещаться по экрану. Например, если оставить без ограничений ширину внешнего элемента `div`, который является контейнером для нашей наклейки, это будет приводить к неустойчивым результатам, потому что перемещаемый элемент `div` намного шире, чем желтый прямоугольник, который вы видите на экране. Вследствие этого вычисление ограничений через границы полей внешнего элемента `div` не приводит к ожидаемым результатам.

В этом примере для внешнего элемента `div`, образующего имитацию наклейки, явно была задана ширина, чтобы можно было точно определить его границы полей с помощью функции `dojo.marginBox`. А функция, задающая границы области, определяет такие ограничения, чтобы элемент `note1` нельзя было переместить правее и ниже элемента `note2` более чем на 50 пикселей.

Попытка использовать объект `constrainedMoveable` без определения функции, вычисляющей границы допустимой области, приведет к появлению массы ошибок, поэтому, если вы решите не применять функцию вычисления границ области, то вам следует вернуться к использованию обычного объекта `Moveable`.

Вариант с использованием статических границ допустимой области реализуется еще проще. Вместо функции, вычисляющей границы, нужно просто передать явные значения. Измените предыдущий пример так, чтобы превратить элемент `note2` в объект `boxConstrainedMoveable`, как показано ниже, и посмотрите, что из этого получится:

```
var m2 = new dojo.dnd.move.boxConstrainedMoveable("note2",
{
    handle : "dragHandle2",
    box : {l : 20, t : 20, w : 500, h : 300}
});
```

Этот пример работает точно так же, как и прежде, за исключением того, что объект `note2` нельзя переместить за определенные границы.

Наконец, подобным образом действует объект `parentConstrainedMoveable`: вы просто создаете перемещаемые объекты и определяете ширину

и высоту родительского элемента, достаточные для обеспечения необходимого рабочего пространства. При использовании специализированного класса не придется делать никакой дополнительной работы. Ниже приводится еще одна работающая версия нашего примера:

```
        <!-- ... обрезано ... -->
        .parent {
            background: #BFECFF;
            border: 10px solid lightblue;
            width: 400px;
            height: 700px;
            padding: 10px;
            margin: 10px;
        }
        <!-- ... обрезано ... -->
        <script type="text/javascript">
            dojo.require("dojo.dnd.move");
            dojo.addOnLoad(function() {
                new dojo.dnd.move.parentConstrainedMoveable("note1",
                {
                    handle : "dragHandle1", area: "margin", within: true
                });
                new dojo.dnd.move.parentConstrainedMoveable("note2",
                {
                    handle : "dragHandle2", area: "padding", within: true
                });
            });
        </script>
    </head>
    <body>
        <div class="parent" >
            <div id="note1">
                <div id='dragHandle1' class="noteHandle"></div>
                <textarea class="note">Note1</textarea>
            </div>
            <div id="note2">
                <div id='dragHandle2' class="noteHandle"></div>
                <textarea class="note">Note2</textarea>
            </div>
        </div>
    </body>
</html>
```

Особый интерес в функции `parentConstrainedMoveables` представляет параметр `area`. Для обозначения допустимой области в пределах родительского элемента можно использовать значения `"margin"`, `"padding"`, `"content"` и `"border"`.

В случае использования описанных выше объектов можно выполнять подключения обработчиков или организовать взаимодействия по подписке, чтобы определять реакцию на события

перемещения. Так как классы constrainedMoveable и boxConstrainedMoveable наследуют класс Moveable, имена событий для функции dojo.connect и темы сообщений для dojo.subscribe остаются теми же, что были перечислены в табл. 7.2.

# Сброс

До настоящего времени основное внимание в этой главе уделялось этапу *перемещения* объектов на экране. Этот раздел завершает обсуждение, сконцентрировав внимание на этапе *сбрасывания*. Для начала рассмотрим класс dojo.dnd.Source – специализированный контейнерный класс, являющийся реализацией перемещаемого объекта в механизме «перетащил и бросил». Класс Source может играть роль зоны сброса, но, как будет показано ниже, с помощью класса dojo.dnd.Target можно определить «настоящую» зону сброса. Объекты класса Source могут действовать и как перемещаемые объекты, и как зоны сброса, а объекты класса Target могут играть роль только зоны сброса.

Объекты класса Source создаются точно так же, как и объекты класса Moveable, – вызывается функция-конструктор, которой в первом аргументе передается узел, а во втором – объект с параметрами. В табл. 7.3 перечислены основные методы.

*Таблица 7.3. Создание и уничтожение объектов класса Source*

| Имя | Комментарий |
|---|---|
| dojo.dnd.Source(/*DOMNode*/node, /*Object*/params) | Конструктор, используемый для создания объекта. Допустимые значения для аргумента params перечислены в табл. 7.4. |
| destroy() | Подготавливает объект к утилизации сборщиком мусора. |

В табл. 7.4. перечислены ключевые параметры, используемые при создании объектов класса Source.

*Таблица 7.4. Параметры настройки для аргумента params конструктора класса Source из табл. 7.3*

| Параметр | Тип | Комментарий |
|---|---|---|
| isSource | Boolean | Значение по умолчанию true. Значение false делает невозможным перемещение объекта. |
| horizontal | Boolean | Значение по умолчанию false. Значение true обеспечивает горизонтальное размещение элементов (требуются встраиваемые элементы HTML). |
| copyOnly | Boolean | Значение по умолчанию false. Значение true обеспечивает создание копии элемента вместо его перемещения (без использования клавиши Ctrl). |

*Таблица 7.4 (продолжение)*

| Параметр | Тип | Комментарий |
|---|---|---|
| skipform | Boolean | Значение по умолчанию false. Как и в случае с объектом класса Moveable, обеспечивает доступность текстовых элементов управления форм для редактирования. |
| withHandles | Boolean | Значение по умолчанию false. Значение true обеспечивает возможность захвата мышью только за определенную область. |
| accept | Array | Значение по умолчанию ["text"]. Определяет тип объекта, который может быть принят зоной сброса. |

Основное назначение класса Source состоит в том, чтобы устранить лишние операции, связанные с перемещением и сбросом элементов, упорядоченных в виде списка, как показано в следующем примере:

```
<html>
    <head>
        <title>Fun with Source!</title>
        <link rel="stylesheet" type="text/css"
         href="http://o.aolcdn.com/dojo/1.1/dijit/themes/tundra/tundra.css" />
        <link rel="stylesheet" type="text/css"
         href="http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css" />
        <link rel="stylesheet" type="text/css"
         href="http://o.aolcdn.com/dojo/1.1/dojo/resources/dnd.css" />
        <link rel="stylesheet" type="text/css"
         href="dndDefault.css" />
        <script
            type="text/javascript"
            djConfig="parseOnLoad:true"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js">
        </script>

        <script type="text/javascript">
            dojo.require("dojo.dnd.Source");
            dojo.require("dojo.parser");
        </script>
    </head>
    <body>
        <div dojoType="dojo.dnd.Source" class="container">
            <div class="dojoDndItem">foo</div>
            <div class="dojoDndItem">bar</div>
            <div class="dojoDndItem">baz</div>
            <div class="dojoDndItem">quux</div>
        </div>
    </body>
</html>
```

В Dojo 1.1 файл *dndDefault.css* недоступен в версии AOL CDN, но он присутствует в исходных текстах, в каталоге *dojo/tests/dnd*. Вам необ-

ходимо добавить ссылку на него, чтобы получить оформление страницы, как показано на рис. 7.1.

Хотя этот начальный пример выглядит не очень впечатляюще, тем не менее в нем заключен значительный объем функциональных возможностей. Обратите внимание, что в нем напрямую используется единственный класс из инструментария – класс `Source`. На основе этого класса создается контейнер элементов, которые можно перемещать в пределах контейнера, для чего следует определить атрибут `dojoType` в соответствующем теге и произвести парсинг элемента – как и в большинстве других примеров, для этого в массив `djConfig` добавляется параметр `parseOnLoad`.

Поэкспериментируйте с этим примером. Совершенно очевидно, что он позволяет перетаскивать элементы по одному за раз, однако важно отметить, что он обладает полной палитрой предлагаемых функциональных возможностей:

- Щелчком мыши выделяется единственный элемент, а все остальные элементы остаются невыделенными.
- Щелчок при нажатой клавише Ctrl переключает состояние выделения элемента и позволяет выделять несколько элементов сразу. Кроме того, это позволяет снять выделение с отдельных элементов, если выделена целая группа.
- Щелчок при нажатой клавише Shift позволяет выделить диапазон элементов, начиная от выделенного элемента до элемента, на кото-

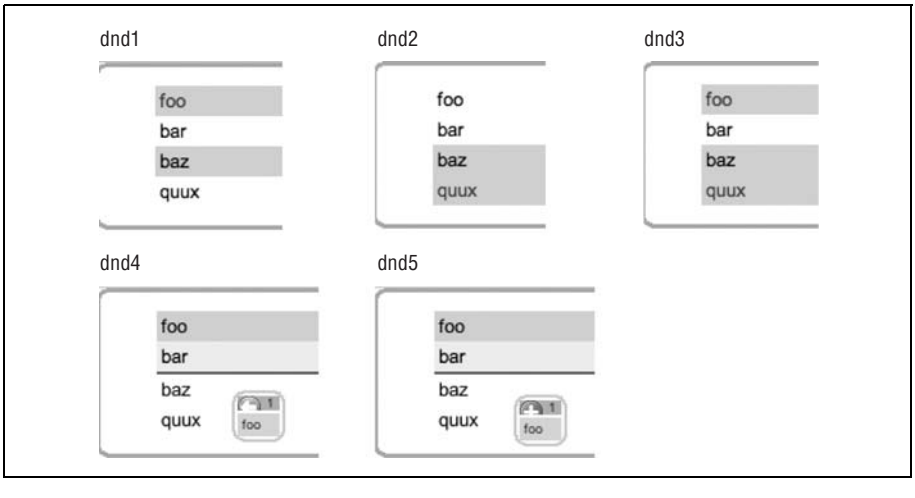



***Рис. 7.1.*** *dnd1 – выделение щелчком мыши при нажатой клавише Ctrl; dnd2 – результат щелчка на элементе quux при нажатой клавише Shift; dnd3 – результат щелчка на элементе quux при нажатых клавишах Shift+Ctrl; dnd4 – операция перемещения без нажатой клавиши Ctrl; dnd5 – операция копирования, когда перемещение выполняется с нажатой клавишей Ctrl*

ром производится щелчок. Все остальные элементы, оказавшиеся за пределами диапазона, теряют выделение.

- Удержание клавиши Ctrl во время перетаскивания приводит к созданию и перемещению копии выделенных элементов. Рисунок 7.1 иллюстрирует некоторые из этих действий.

## Истинная зона сброса

Как уже упоминалось выше, иногда у вас будет возникать необходимость использовать класс `Target`, объекты которого могут играть роль только зоны сброса. После того как элементы будут перемещены в эту зону, они больше не могут перемещаться и переупорядочиваться. Внесите простые изменения, как показано ниже, чтобы увидеть класс `Target` в действии.

```
<body>
    <div dojoType="dojo.dnd.Source" class="container">
        <div class="dojoDndItem">foo</div>
        <div class="dojoDndItem">bar</div>
        <div class="dojoDndItem">baz</div>
        <div class="dojoDndItem">quux</div>
    </div>
    <!--Элементы, перемещенные в зону сброса, не могут больше перемещаться
       и переупорядочиваться -->
    <div dojoType="dojo.dnd.Target" class="container"></div>
</body>
```

Как вы уже могли заметить, всего в несколько строк программного кода можно заключить огромный объем функциональности, и, хотя в примере использовались элементы `div`, следует отметить, что с таким же успехом можно применять и другие стандартные элементы HTML. Очень часто, например, используются элементы неупорядоченных списков, `ul` и `li`.

## Собственные аватары

Маленький графический значок, который появляется при перемещении элемента, созданного на основе класса `Source`, называется *аватарой*. Стандартная аватара имеет достаточно привлекательный внешний вид, тем не менее существует возможность создать собственную аватару. Следующий фрагмент программного кода достаточно наглядно демонстрирует, как можно назначить свой текст для аватары, переопределив метод `creator`, используемый для создания аватары для одного или более узлов. В данном конкретном случае мы предпочли переопределить метод `creator` прямо в тексте разметки. Кроме того, была изменена схема размещения элементов на горизонтальную, чтобы заодно продемонстрировать, как изменять расположение элементов:

```
<body>
    <div dojoType="dojo.dnd.Source" horizontal=true class="container">
```

```
        <span class="dojoDndItem ">foo</span>
        <span class="dojoDndItem ">bar</span>
        <span class="dojoDndItem ">baz</span>
        <span class="dojoDndItem ">quux</span>

        <script type="dojo/method" event="creator" args="item,hint">
            // переопределить функцию creator и вернуть соответствующий тип
            var node = dojo.doc.createElement("span");
            node.id = dojo.dnd.getUniqueId();
            node.className = "dojoDndItem";
            node.innerHTML = "<strong style='color: red'>Custom</strong>"+
                             item;
            return {node: node, data: item, type: ["text"]};
        </script>

    </div>
    <div dojoType="dojo.dnd.Target" horizontal=true class="container"></div>
</body>
```

Обратите внимание на аргументы, которые передаются функции `creator` — `item` и `hint`. Аргумент `item` представляет фактически перемещаемый элемент, а аргумент `hint` определяет, какого рода подсказка должна быть создана. Если вы не будете заниматься собственной реализацией низкоуровневых механизмов, аргумент `hint` всегда будет содержать текст `"avatar"`. Ожидается, что функция `creator` вернет объект, описывающий представление элемента `item` с ключами, определяющими узел DOM, представление данных и тип представления. Имейте ввиду, что `"text"` — это тип представления, принятый для объекта `Source` по умолчанию.

## События сброса

Подписка на события или подключение обработчиков с помощью функций `dojo.subscribe` и `dojo.connect` выполняется так же просто, как и в случае с объектами класса `Moveable`. В табл. 7.5 приводится перечень событий, доступных для организации взаимодействий по подписке или для непосредственного подключения, а далее следует пример программного кода.

*Таблица 7.5. События сброса*

| Тип | Событие | Параметры | Описание |
|---|---|---|---|
| по подписке | `"/dnd/source/over"` | `/* Node */ source` | Публикуется, когда указатель мыши перемещается над контейнером `Source`. Параметр `source` указывает на контейнер. Когда указатель мыши выходит за пределы контейнера `Source`, выполняется публикация события с этой же темой, но в параметре `source` передается значение `null`. |

*Таблица 7.5 (продолжение)*

| Тип | Событие | Параметры | Описание |
|---|---|---|---|
| по подписке | "/dnd/start" | /* Node */ source<br>/* Array */ nodes<br>/* Boolean */ copy | Публикуется, когда начинается перемещение. В параметре source передается контейнер Source, в котором определены координаты начала операции перемещения. В параметре copy передается значение true, когда выполняется операция копирования, и false — когда выполняется операция перемещения. В параметре nodes передается массив элементов, вовлеченных в операцию перемещения. |
| по подписке | "/dnd/drop" | /* Node */ source<br>/* Array */ nodes<br>/* Boolean */ copy | Публикуется, когда выполняется сброс (и фактически заканчивается перемещение). В параметре source передается контейнер Source, в котором определены координаты операции сброса. В параметре copy передается значение true, когда выполняется операция копирования, и false — когда выполняется операция перемещения. В параметре nodes передается массив элементов, вовлеченных в операцию сброса. |
| по подписке | "/dnd/cancel" | нет | Публикуется в случае отмены операции сброса (например, когда была нажата клавиша Esc). |
| соединение | onDndSourceOver | /* Node */ source | Возникает, когда указатель мыши перемещается над контейнером Source. Параметр source указывает на контейнер. Когда указатель мыши выходит за пределы контейнера Source, возникает другое событие onDndSourceOver, а в параметре source передается значение null. |
| соединение | onDndStart | /* Node */ source<br>/* Array */ nodes<br>/* Boolean */ copy | Возникает, когда начинается перемещение. В параметре source передается контейнер Source, в котором определены координаты начала операции перемещения. В параметре copy передается значение true, когда выполняется операция копирования, и false — когда выполняется операция перемещения. В параметре nodes передается массив элементов, вовлеченных в операцию перемещения. |

| Тип | Событие | Параметры | Описание |
|---|---|---|---|
| соединение | onDndDrop | /* Node */ source<br>/* Array */ nodes<br>/* Boolean */ copy | Возникает, когда выполняется сброс (и фактически заканчивается перемещение). В параметре source передается контейнер Source, в котором определены координаты операции сброса. В параметре copy передается значение true, когда выполняется операция копирования, и false — когда выполняется операция перемещения. В параметре nodes передается массив элементов, вовлеченных в операцию сброса. |
| соединение | onDndCancel | нет | Возникает в случае отмены операции сброса (например, когда была нажата клавиша Esc). |

Двинемся дальше, загрузим следующий полноценный пример и воспользуемся расширением Firebug, чтобы вывести в консоль различные темы событий, получаемых по подписке. Не забывайте, что существует возможность перетаскивать элементы из любых контейнеров Source. На рис. 7.2 показаны полученные результаты. Отличный пример!

```
<html>
    <head>
        <title>More Fun with Drop!</title>
        <link rel="stylesheet" type="text/css"
        href="http://o.aolcdn.com/dojo/1.1/dijit/themes/tundra/tundra.css" />
        <link rel="stylesheet" type="text/css"
         href="http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css" />
        <link rel="stylesheet" type="text/css"
         href="dndDefault.css" />
```

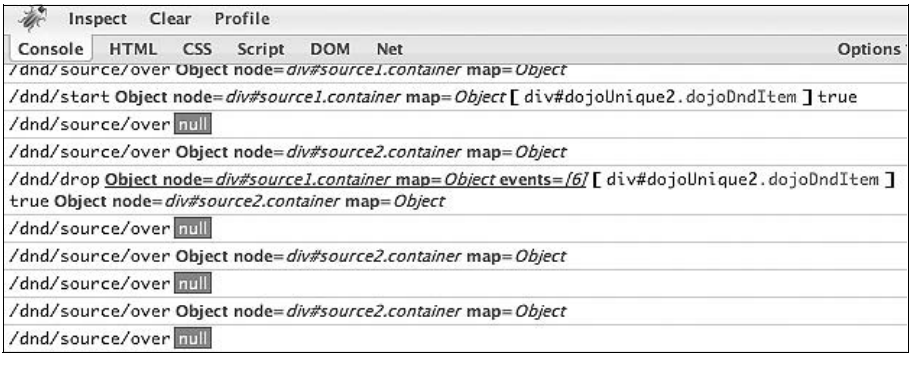



***Рис. 7.2.*** *Расширение Firebug отлично подходит для детального изучения действия механизма «перетащил и бросил»*

```
        <script
            type="text/javascript"
            djConfig="parseOnLoad:true"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js">
        </script>

        <script type="text/javascript">
            dojo.require("dojo.dnd.Source");
            dojo.require("dojo.parser");

            dojo.addOnLoad(function() {
                dojo.subscribe("/dnd/source/over", function(source) {
                    console.log("/dnd/source/over", source);
                });
                dojo.subscribe("/dnd/start", function(source, nodes, copy) {
                    console.log("/dnd/start", source, nodes, copy);
                });
                dojo.subscribe("/dnd/drop", function(source, nodes, copy) {
                    console.log("/dnd/drop", source, nodes, copy);
                });
                dojo.subscribe("/dnd/cancel", function() {
                    console.log("/dnd/cancel");
                });
            });
        </script>
    </head>
    <body>
        <div id="source1" dojoType="dojo.dnd.Source" class="container">
            <div class="dojoDndItem">foo</div>
            <div class="dojoDndItem">bar</div>
            <div class="dojoDndItem">baz</div>
            <div class="dojoDndItem">quux</div>
        </div>
        <div id="source2" dojoType="dojo.dnd.Source" class="container">
            <div class="dojoDndItem">FOO</div>
            <div class="dojoDndItem">BAR</div>
            <div class="dojoDndItem">BAZ</div>
            <div class="dojoDndItem">QUUX</div>
        </div>
    </body>
</html>
```

Все, что требуется, чтобы продемонстрировать прямое подключение к событиям, – внести изменение в функцию `addOnLoad`. Поскольку в этом случае необходима ссылка на созданный объект класса `Source` (а не на узел DOM), нам следует программно создать объект `Source` и не полагаться на парсер, создающий экземпляры виджетов, определяемые разметкой страницы. Замените фрагмент программного кода, как показано ниже, выключите флаг `parseOnLoad` в массиве `djConfig` и посмотрите на результаты в консоли Firebug еще раз:

```
dojo.addOnLoad(function() {
    //Сохранить ссылку на объект Source для выполнения соединений.
    var s1 = new dojo.dnd.Source("source1");

    dojo.connect(s1, "onDndSourceOver", function(source) {
        console.log("onDndSourceOver for", s1, source);
    });
    dojo.connect(s1, "onDndStart", function(source, nodes, copy) {
        console.log("onDndStart for ", s1, source, nodes, copy);
    });
    dojo.connect(s1, "onDndStop", function(source, nodes, copy, target) {
        console.log("onDndStop for", s1, source, nodes, copy, target);
    });
    dojo.connect(s1, "onDndCancel", function() {
        console.log("onDndCancel for ", s1);
    });
});
```

## Взаимодействие со сбрасываемыми объектами

Предыдущий пример демонстрирует возможность использования функции-конструктора Source, чтобы наделить узел возможностью перетаскивать его мышью, однако набор функциональных возможностей, которые можно использовать в сценариях, существенно шире. В табл. 7.6 перечислены функциональные возможности, предлагаемые низкоуровневым классом Selector, определение которого находится в модуле dojo.dnd. Класс Source наследует класс Selector, поэтому перечисленные функции также доступны и в объектах класса Source, однако вы с легкостью можете найти применение и классу Selector.

*Таблица 7.6. Прикладной интерфейс класса Selector*

| Метод | Комментарий |
|---|---|
| getSelectedNodes() | Возвращает массив выделенных узлов. |
| selectNone() | Снимает выделение со всех узлов. |
| selectAll() | Выделяет все узлы. |
| deleteSelectedNodes() | Удаляет все выделенные узлы. |
| insertNodes(/* Boolean */ addSelected, /* Array */ data, /* Boolean */ before, /* Node */ anchor) | Вставляет массив узлов, которые могут быть выделены вызовом функции addSelected. В случае отсутствия аргумента anchor узлы вставляются перед первым дочерним узлом объекта Selector. В противном случае они вставляются либо перед, либо после узла anchor в соответствии со значением аргумента before. |
| destroy() | Подготавливает объект к утилизации сборщиком мусора. |

*Таблица 7.6 (продолжение)*

| Метод | Комментарий |
|---|---|
| `onMouseDown(/* Object */ event)` | Может подключаться с помощью `dojo.connect` для обработки события `onmousedown`. Однако, прежде чем принимать решение об использовании этого события, следует рассмотреть возможность применения событий `onDnd`. В аргументе `event` передается стандартная информация о событии. |
| `onMouseUp(/* Object */ event)` | Может подключаться с помощью `dojo.connect` для обработки события `onmouseup`. Однако, прежде чем принимать решение об использовании этого события, следует рассмотреть возможность применения событий `onDnd`. В аргументе `event` передается стандартная информация о событии. |
| `onMouseMove(/* Object */ event)` | Может подключаться с помощью `dojo.connect` для обработки события перемещения указателя мыши. Однако, прежде чем принимать решение об использовании этого события, следует рассмотреть возможность применения событий `onDnd`. В аргументе `event` передается стандартная информация о событии. |
| `onOverEvent(/* Object */ event)` | Может подключаться с помощью `dojo.connect` для обработки события перемещения указателя мыши в область, занимаемую объектом. Однако, прежде чем принимать решение об использовании этого события, следует рассмотреть возможность применения событий `onDnd`. В аргументе `event` передается стандартная информация о событии. |
| `onOutEvent(/* Object */ event)` | Может подключаться с помощью `dojo.connect` для обработки события выхода указателя мыши за пределы области, занимаемой объектом. Однако, прежде чем принимать решение об использовании этого события, следует рассмотреть возможность применения событий `onDnd`. В аргументе `event` передается стандартная информация о событии. |

# В заключение

Более практичный пример действия механизма «перетащил и бросил» вы найдете в разделе «Операция перетаскивания в деревьях», в главе 15, где этот механизм используется для решения проблемы управления диджитом `Tree`. Деревья сами по себе являются феноменальным достижением, а применение возможности перетаскивания делает их еще лучше!

После прочтения этой главы вы должны:

- Уметь создавать объекты класса `Moveable`, которые можно перемещать по экрану без каких-либо ограничений
- Уметь накладывать ограничения на объекты `Moveable`, чтобы контролировать их поведение
- Уметь создавать контейнеры `Source` и `Target` с коллекциями элементов, которые можно перемещать в/из/внутри контейнеров
- Уметь создавать собственные аватары для взаимодействия с пользователем при выполнении операций перемещения
- Уметь использовать функции `dojo.connect` и `dojo.subscribe` для получения событий и извещений от перемещаемых объектов

В следующей главе мы рассмотрим анимационные и специальные эффекты.

# 8

# Анимация и специальные эффекты

Анимационные эффекты могут добавить выразительности ничем неприметному приложению. В этой главе последовательно будут рассматриваться функции, связанные с воспроизведением анимационных эффектов, встроенные непосредственно в библиотеку Base и в модуль `dojo.fx` (модуль эффектов), который входит в состав библиотеки Core. В этой главе вы найдете множество примеров с программным кодом, а сведения, которые здесь даются, основаны лишь на тех концепциях, что рассматривались в предыдущих главах. Кроме того, эта глава может рассматриваться как самостоятельный справочник.

## Анимация

Библиотека Base инструментального набора включает в себя основные средства анимации и дополняет их расширенными возможностями, находящимися в модуле `dojo.fx`. Основные возможности воспроизведения анимации в библиотеке Base реализованы в виде класса `_Animation`, который действует как делегат – в том смысле, что он вызывает функции обратного вызова в соответствии с настройками. Эти функции обратного вызова осуществляют манипулирование свойствами узла – и вид узла меняется. После создания экземпляра класса `_Animation` все, что останется сделать – это вызвать его метод `play`.

Начальный символ подчеркивания в имени класса `_Animation` означает, по крайней мере, следующее:

- Прикладной интерфейс определен еще не окончательно, хотя уже обладает достаточно высокой стабильностью и, скорее всего, не претерпит существенных изменений (если такие изменения вообще произойдут) в процессе разработки версии 1.1 инструментария, когда он действительно станет стабильным.

- Обычно класс _Animation не используется непосредственно. Вместо этого следует опираться на применение вспомогательных функций, имеющихся в библиотеке Base и в модуле dojo.fx и позволяющих создавать и использовать этот класс. Однако, для запуска анимационных эффектов вам наверняка придется вызывать метод play класса.

## Простые эффекты растворения и проявления

Прежде чем углубиться в исследование сложных аспектов воспроизведения анимационных эффектов, давайте сначала разберемся с одним из самых простых примеров: создадим на экране простой квадрат, который будет как бы растворяться после щелчка мышью на нем, как показано на рис. 8.1. В этом примере используется одна из функций fade, входящих в состав библиотеки Base. Функции fadeOut и fadeIn принимают три именованных аргумента, которые перечислены в табл. 8.1. На рис. 8.1 показана временная диаграмма протекания процесса.

*Таблица 8.1. Параметры функций fade из библиотеки Base*

| Параметр | Тип | Комментарий |
|---|---|---|
| node | DOM Node | Узел, который будет воспроизводить эффект анимации. |
| duration | Integer | Продолжительность воспроизведения эффекта в миллисекундах. Значение по умолчанию: 350. |
| easing | Function | Переходная функция, описывающая ускорение и/или замедление процесса. По умолчанию реализует формулу: `(0.5 + ((Math.sin((n + 1.5) * Math.PI))/2)`. Обратите внимание, что функция определена только в интервале от 0 до 1 как для fadeIn, так и для fadeOut. |

Параметры node и duration не требуют дополнительных пояснений, но назначение переходной функции easing может вызывать недопонимание. Проще говоря, переходная функция – это обычная функция, которая управляет скоростью изменения чего-либо, в данном случае – объекта класса _Animation. Самая простая реализация линейной переходной функции выглядит как `function(x) { return x; }` – для каждого входного значения возвращается то же самое значение. То есть для диапазона десятичных значений от 0 до 1 можно заметить, что такая функция всегда возвращает значение, равное значению входного аргумента. Если построить график такой функции, получится обычная прямая линия с постоянным углом наклона, как показано на рис. 8.2. Постоянство угла гарантирует равномерное протекание анимационного эффекта с неизменной скоростью.

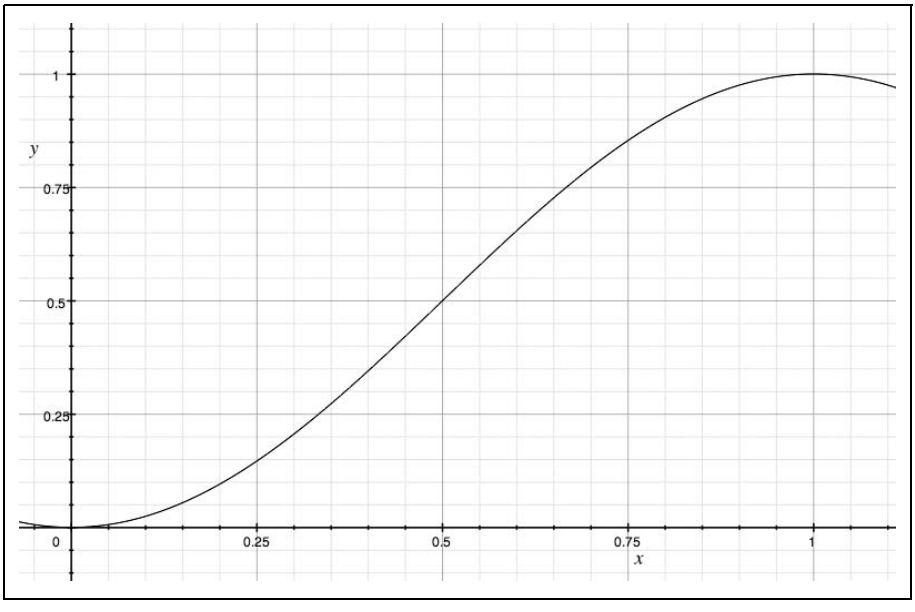



***Рис. 8.1.*** *Кривая скорости протекания анимационного эффекта при использовании переходной функции по умолчанию – переходная функция для fadeIn и fadeOut определена только в диапазоне от 0 до 1*

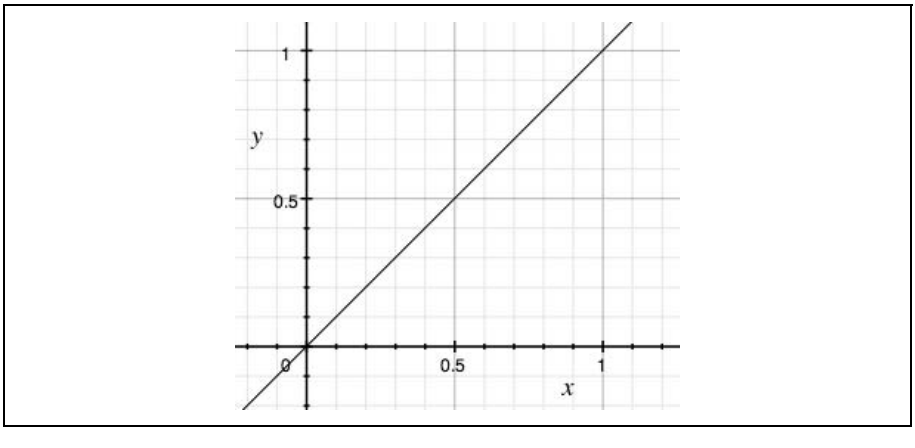



***Рис. 8.2.*** *Простая переходная функция, линейная в диапазоне от 0 до 1*

Пример 8.1 демонстрирует, как реализовать постепенное растворение части изображения на экране, используя значения параметров по умолчанию.

*Пример 8.1. Реализация визуального растворения узла*

```
<html>
    <head>
        <title>Fun with Animation!</title>
        <style type="text/css">
            @import "http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css";
            .box {
                width : 200px;
                height : 200px;
                margin : 5px;
                background : blue;
                text-align : center;
            }
        </style>
        <script
            type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js">
        </script>
        <script type="text/javascript">
            dojo.addOnLoad(function() {
                var box = dojo.byId("box");
                dojo.connect(box, "onclick", function(evt) {
                    var anim = dojo.fadeOut({node:box});
                    anim.play();
                });
            });
        </script>
    </head>
    <body>
        <div id="box" class="box">Fade Me Out</div>
    </body>
</html>
```

Чтобы продемонстрировать влияние переходной функции, ниже приводится измененный блок функции `addOnLoad`, где реализована переходная функция, график которой представлен на рис. 8.3. Обратите внимание, что переходная функция по умолчанию реализует относительно равномерное увеличение возвращаемого значения в диапазоне от 0 до 1, тогда как при использовании измененной версии практически все изменения протекают в самом конце временного интервала. Кроме того, в этом примере используется оператор точки для запуска метода `play` класса `_Animation`, чтобы избежать необходимости сохранять явную ссылку, что является более понятным и общепринятым.

```
dojo.addOnLoad(function() {
    var box = dojo.byId("box");
    dojo.connect(box, "onclick", function(evt) {
        var easingFunc = function(x) {
            return Math.pow(x,10);
        }
```

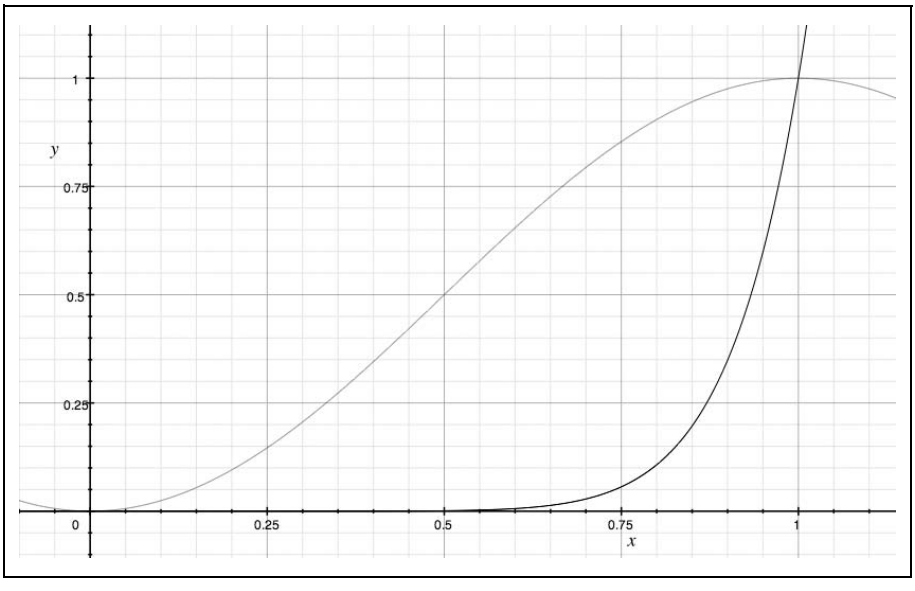



***Рис. 8.3.*** *Пример переходной функции, для сравнения приводится график переходной функции по умолчанию*

```
            dojo.fadeOut({
                node:box,
                easing : easingFunc,
                duration : 3000
            }).play();
    });});
```

В модуле `dojo.fx.easing` имеется несколько замечательных переходных функций. Обратите на них внимание, если вам потребуется реализовать более творческий подход.

Учитывая, что простые эффекты изменения прозрачности получили достаточно широкое распространение, они были помещены на расстояние одного вызова функции – в библиотеку Base. Однако пройдет совсем немного времени, и вас начнет интересовать вопрос о том, какие еще эффекты можно воспроизводить средствами класса `_Animation`.

## Анимирование произвольных свойств CSS

Давайте на основе полученных знаний рассмотрим оставшуюся функцию библиотеки Base – `animateProperty`, способную принимать один или более параметров, список которых приводится в табл. 8.2, и работающую точно так же, как функции `fadeIn` и `fadeOut`.

*Таблица 8.2. Параметры функции animateProperty*

| Параметр | Тип | Комментарий |
|---|---|---|
| node | DOM Node | Узел или значение атрибута id узла, который будет воспроизводить эффект анимации. |
| duration | Integer | Продолжительность воспроизведения эффекта в миллисекундах. Значение по умолчанию: 350. |
| easing | Function | Переходная функция, описывающая ускорение и/или замедление процесса. По умолчанию реализует формулу:<br>(0.5 + ((Math.sin((n + 1.5) * Math.PI))/2). |
| repeat | Integer | Количество повторов воспроизведения эффекта. Значение по умолчанию: 0. |
| rate | Integer | Продолжительность паузы в миллисекундах перед переходом к следующему «кадру». Этот параметр определяет, как часто объект _Animate будет выполнять обновления. Например, значение 1000 в аргументе rate означает относительную скорость изменения 1 кадр в секунду. Если предположить, что продолжительность эффекта задана равной 10000, то в объекте _Animate будет воспроизведено 10 дискретных обновлений. Значение по умолчанию: 10. |
| delay | Integer | Продолжительность паузы в миллисекундах между вызовом метода play и фактическим началом воспроизведения эффекта. |
| properties | Object | Определяет свойства CSS, которые подвергнутся воздействию. В объекте указываются начальные значения, конечные значения и единицы измерения. Начальное (start) и конечное (end) значения могут быть представлены литералами или функциями, используемыми для вычисления этих значений:<br>start (String)<br>Начальное значение свойства<br>end (String)<br>Конечное значение свойства<br>unit (String)<br>Единица измерения: px (по умолчанию), em и т. д. |

Заменив функцию addOnLoad фрагментом, что приводится ниже, можно проверить работу функции animateProperty. В данном конкретном случае производится изменение ширины узла от 200px до 400px:

```
dojo.addOnLoad(function() {
    var box = dojo.byId("box");
    dojo.connect(box, "onclick", function(evt) {
        dojo.animateProperty({
```

```
            node : box,
            duration : 3000,
            properties : {
                width : {start : '200', end : '400'}
            }
        }).play();
    });
});
```

Функция `animateProperty` стоит того, чтобы потратить некоторое время на эксперименты с ней для получения полного представления о том, какие творческие возможности она в себе таит. Она составляет основу большинства анимационных эффектов, реализованных в модуле `dojo.fx`, и есть вероятность, что вы будете использовать ее очень часто при совершении стандартных действий. Она принимает практически любые свойства CSS посредством одного и того же унифицированного интерфейса. Пример 8.2 демонстрирует, как анимационные эффекты воздействуют на информационное наполнение страницы, не вовлеченное в анимацию. Щелчок на синем прямоугольнике заставляет его увеличить свои размеры по осям $x$ и $y$, что приводит к соответствующему изменению местоположения красного и зеленого прямоугольников.

*Пример 8.2. Увеличение размеров узла*

```
<html>
    <head>
        <title>More Fun With Animation!</title>
        <style type="text/css">
            @import "http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css";
            .box {
                width : 200px;
                height : 200px;
                margin : 5px;
                text-align : center;
            }
            .blueBox {
                background : blue;
                float : left;
            }
            .redBox {
                background : red;
                float : left;
            }
            .greenBox {
                background : green;
                clear : left;
            }
        </style>
        <script
            type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js">
```

```
        </script>
        <script type="text/javascript">

            dojo.addOnLoad(function() {
                var box = dojo.byId("box1");
                dojo.connect(box, "onclick", function(evt) {
                    dojo.animateProperty({
                        node : box,
                        duration : 3000,
                        properties : {
                            height : {start : '200', end : '400'},
                            width : {start : '200', end : '400'}
                        }
                    }).play();
                });
            });
        </script>
    </head>
    <body>
        <div id="box1" class="box blueBox">Click Here</div>
        <div id="box2" class="box redBox"></div>
        <div id="box2" class="box greenBox"></div>
    </body>
</html>
```

Если некоторые аргументы функции `animateProperty` до сих пор остаются неясными, предыдущий пример предоставляет прекрасную возможность познакомиться поближе с действием разных параметров. Например, внесите следующие изменения в вызове функции `animateProperty`, чтобы вместо плавного воспроизведения эффекта получить 10 дискретных кадров (вспомните, что продолжительность эффекта, поделенная на значение параметра `rate`, дает в результате число кадров):

```
dojo.addOnLoad(function() {
    var box = dojo.byId("box1");
    dojo.connect(box, "onclick", function(evt) {
        dojo.animateProperty({
            node : box,
            duration : 10000,
            rate : 1000,
            properties : {
                height : {start : '200', end : '400'},
                width : {start : '200', end : '400'}
            }
        }).play();
    });
});
```

Учитывая, что используемая переходная функция по умолчанию является довольно гладкой, потратьте некоторое время на эксперименты, чтобы выяснить, какое влияние оказывают на эффект анимации другие функции, имеющие более скачкообразный график. Например,

в следующем фрагменте используется параболическая переходная функция, график которой представлен на рис. 8.4, значение которой увеличивается тем больше, чем ближе исходное значение к границам интервала, а разбивка эффекта на 10 кадров делает это влияние еще более очевидным:

```
dojo.addOnLoad(function() {
    var box = dojo.byId("box1");
    dojo.connect(box, "onclick", function(evt) {
        dojo.animateProperty({
            node : box,
            duration : 10000,
            rate : 1000,
            easing : function(x) { return x*x; },
            properties : {
                height : {start : '200', end : '400'},
                width : {start : '200', end : '400'}
            }
        }).play();
    });
});
```

Несмотря на то что до сих пор в примерах использовались только монотонные[1] переходные функции, это не является обязательным требованием. Например, попробуйте изменить рабочий пример, применив в нем немонотонную переходную функцию, график которой показан на рис. 8.5, чтобы увидеть, какой эффект она позволяет получить:

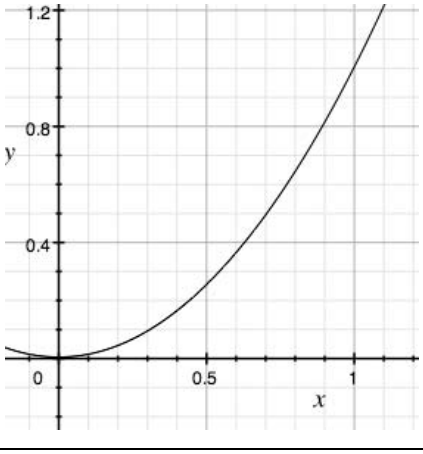



***Рис. 8.4.*** *Пример параболической переходной функции*

[1] Монотонная функция – это функция, приращение которой не меняет знака, то есть функция всегда возрастает либо всегда убывает.

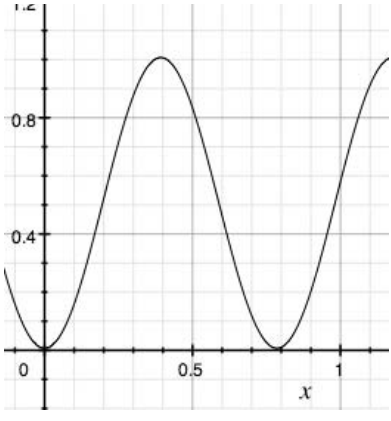



***Рис. 8.5.*** *Переходная функция, которая то возрастает, то убывает*

```
dojo.addOnLoad(function() {
    var box = dojo.byId("box1");
    dojo.connect(box, "onclick", function(evt) {
        dojo.animateProperty({
            node : box,
            duration : 10000,
            easing : function(x) {return Math.pow(Math.sin(4*x),2);},
            properties : {
                height : {start : '200', end : '400'},
                width : {start : '200', end : '400'}
            }
        }).play();
    });
});
```

## Упрощение синтаксиса

В версию 1.1 был добавлен упрощенный синтаксис описания аргумента `properties`. Раньше воспроизведение эффекта изменения текущей ширины узла до нового значения можно было описать, как показано ниже:

```
dojo.animateProperty({
    node: "foo",
    properties: { width: { end: 500 } } //много фигурных скобок
}).play();
```

Теперь же, если в аргументе `properties` передается только одно целое число, подразумевается, что оно определяет значение параметра `end`, то есть предыдущий фрагмент можно упростить до:

```
dojo.animateProperty({
    node: "foo", properties: { width: 500 } //всего пара фигурных скобок
}).play();
```

В версии 1.1 появилось еще одно нововведение – функция dojo.anim, которая добавила пару расширений. Она работает практически как функция animateProperty, за исключением того, что автоматически запускает воспроизведение анимационного эффекта и не требует явного вызова метода play, то есть, когда она возвращает объект _Animation, он уже находится в состоянии воспроизведения. (Вызов метода play() в этом случае воспринимается как пустая операция.) Кроме того, наиболее часто используемые свойства объекта-аргумента были сделаны позиционными аргументами.

Полная сигнатура функции выглядит, как показано ниже:

```
dojo.anim(/*DOMNode|String*/node, /*Object*/props, /*Integer?*/
duration,
/*Function?*/easing, /*Function?*/onEnd, /*Integer?*/delay)
//Возвращает объект _Animation, находящийся в состоянии воспроизведения
```

Если эти расширения смущают вас, вам совсем необязательно использовать их – они были реализованы как дополнительные удобства для упрощения разработки.

## Программное управление анимацией

Хотя в общем случае вам не придется создавать объекты _Animation самостоятельно, вы имеете возможность управлять ими в большинстве обычных ситуаций. Например, в процессе воспроизведения анимационного эффекта имеется возможность приостановить его, перезапустить или остановить совсем, запросить информацию о состоянии процесса воспроизведения или дать команду переместиться в определенную точку. Для выполнения всех этих действий объекты _Animation предоставляют методы, перечисленные в табл. 8.3.

*Таблица 8.3. Функции управления объектом _Animations*

| Метод | Параметры | Комментарий |
|---|---|---|
| stop | /* Boolean */ goToEnd | Останавливает воспроизведение анимации. Если параметр goToEnd имеет значение true, объект _Animation перемещается в конец, поэтому при следующем вызове метода play он начнет воспроизведение с самого начала. По умолчанию параметр goToEnd имеет значение false. |

| Метод | Параметры | Комментарий |
|---|---|---|
| pause | нет | Приостанавливает воспроизведение. |
| play | /* Integer */ delay<br>/* Boolean */ goToStart | Запускает воспроизведение анимации. С помощью параметра delay позволяет определять время задержки в миллисекундах перед фактическим началом воспроизведения. В случае запуска ранее приостановленной анимации значение true в параметре goToStart позволяет принудительно начать воспроизведение с самого начала, а не с того места, где оно было приостановлено. |
| status | нет | Возвращает состояние воспроизведения анимации. Возможными возвращаемыми значениями являются: "paused", "playing" и "stopped". |
| gotoPercent | /* Decimal */ percent<br>/* Boolean */ andPlay | Останавливает воспроизведение, затем переходит в точку, определяемую параметром percent, который может принимать значения от 0.0 до 1.0. Если параметр andPlay имеет значение true (значение по умолчанию – false), запускает воспроизведение анимации. |

Обратите внимание, как правильно записывается имя параметра, – *gotoPercent*, а не *goToPercent*. Это одна из нескольких функций в инструментарии, которые не следуют привычному соглашению об использовании символов верхнего регистра в именах и при написании имен которых легко ошибиться.

Кроме всего прочего, при использовании функции animateProperty имеется возможность определить во входных параметрах любой из методов, перечисленных в табл. 8.4. Эта таблица описывает предоставляемые функциональные возможности, а в следующем ниже фрагменте программного кода показано, как следует изменить вызов функции animateProperty, чтобы можно было поэкспериментировать с этим множеством методов.

*Таблица 8.4. Входные методы для функции animateProperty*

| Метод | Параметры | Комментарий |
|---|---|---|
| beforeBegin | нет | Вызывается перед началом воспроизведения анимации, обеспечивая тем самым возможность доступа к объекту _Animation и к узлу непосредственно перед тем, как что-то начнет происходить. |

*Таблица 8.4 (продолжение)*

| Метод | Параметры | Комментарий |
|---|---|---|
| onBegin | /* Object */ value | Вызывается непосредственно после начала воспроизведения, то есть этот метод по сути является асинхронным. Параметр value – это объект, содержащий текущие значения свойств стилей. |
| onAnimate | /* Object */ value | Вызывается для каждого кадра в процессе воспроизведения. Параметр value – это объект, содержащий текущие значения свойств стилей. |
| onEnd | нет | Вызывается автоматически по завершении воспроизведения анимации. |
| onPlay | /* Object */ value | Вызывается всякий раз, когда происходит вызов метода play (включая самый первый раз). Параметр value – это объект, содержащий текущие значения свойств стилей. |
| onPause | /* Object */ value | Вызывается всякий раз, когда происходит вызов метода pause. Параметр value – это объект, содержащий текущие значения свойств стилей. |
| onStop | /* Object */ value | Вызывается всякий раз, когда происходит вызов метода stop. Параметр value – это объект, содержащий текущие значения свойств стилей. |

Ниже приводится небольшой фрагмент программного кода, который можно использовать, чтобы поэкспериментировать с этими методами:

```
dojo.animateProperty({
    node : "box1",
    duration:10000,
    rate : 1000,
    beforeBegin:function(){ console.log("beforeBegin: ", arguments); },
    onBegin:function(){ console.log("onBegin: ", arguments); },
    onAnimate:function(){ console.log("onAnimate: ", arguments); },
    onEnd:function(){ console.log("onEnd: ", arguments); },
    onPlay:function(){ console.log("onPlay: ", arguments); },
    properties : {height : {start : "200", end : "400"} }
}).play();
```

Следующий фрагмент показывает, как изменить рабочий пример, чтобы поближе познакомиться с основными методами управления объектом _Animate:

```
<!-- обрезано -->
        <script type="text/javascript">
```

```
            dojo.addOnLoad(function() {
                var box = dojo.byId("box1");
                var anim;
                dojo.connect(box, "onclick", function(evt) {
                    anim = dojo.animateProperty({
                        node : box,
                        duration : 10000,
                        rate : 1000,
                       easing : function(x) { console.log(x); return x*x; },
                        properties : {
                            height : {start : '200', end : '400'},
                            width : {start : '200', end : '400'}
                        }
                    });
                    anim.play();
                  dojo.connect(dojo.byId("stop"), "onclick", function(evt) {
                        anim.stop(true);
                        console.log("status is ", anim.status());
                    });
                   dojo.connect(dojo.byId("pause"),"onclick",function(evt) {
                        anim.pause();
                        console.log("status is ", anim.status());
                    });
                  dojo.connect(dojo.byId("play"), "onclick", function(evt) {
                        anim.play();
                        console.log("status is ", anim.status());
                    });
                  dojo.connect(dojo.byId("goTo50"),"onclick",function(evt) {
                        anim.gotoPercent(0.5, true);
                        console.log("advanced to 50%");
                    });
                });
            });
        </script>
    </head>
    <body>
        <div>
            <button id="stop" style="margin : 5px">stop</button>
            <button id="pause" style="margin : 5px">pause</button>
            <button id="play" style="margin : 5px">play</button>
            <button id="goTo50" style="margin : 5px">50 percent</button>
        </div>
        <div id="box1" class="box blueBox">Click Here</div>
        <div id="box2" class="box redBox"></div>
        <div id="box2" class="box greenBox"></div>
    </body>
</html>
```

# Core fx

До этого момента все наше внимание было сосредоточено на средствах создания анимационных эффектов, которые имеются в библиотеке Base. Возможностей функций `fadeIn`, `fadeOut` и `animateProperty`, представленных в библиотеке Base, вполне достаточно для подавляющего большинства случаев. Однако в модуле `fx` из библиотеки Core имеются еще функции, доступ к которым можно получить ценой одной дополнительной инструкции `dojo.require`. Среди них вы найдете функции для воспроизведения таких эффектов, как скольжение узлов, сворачивание и разворачивание узлов, а также составление последовательностей, комбинирование и переключение анимационных эффектов.

## Пристальный взгляд на объект _Animation

Как отмечалось ранее, для воспроизведения анимационных эффектов чаще используется не сам объект `_Animation`, а вспомогательные функции, особенно `animateProperty`. Функции, подобные `animateProperty`, действуют как обертки, позволяющие выполнить настройку объектов `_Animation`, хотя при этом они возвращают эти объекты, чтобы обеспечить возможность запускать анимацию, приостанавливать ее и выполнять другие операции над объектами.

Если вас заинтриговал объект `_Animation` и вам интересно точно знать, что он делает, могу сказать, что в действительности все просто. За исключением узла и свойств стиля функция-конструктор принимает те же параметры, что и функция `animateProperty`, представленные в табл. 8.2 (что совершенно неудивительно, потому что `animateProperty` действует как функция-обертка, которая создает и управляет объектом анимации), а также еще один дополнительный параметр, `curve`, который задает область определения переходной функции. Встроенные функции анимации, такие как `animateProperty`, `fadeIn` и другие, имеют область определения по умолчанию в диапазоне от 0 до 1, а параметр `curve` позволяет задавать область определения в произвольном диапазоне. (Если термин *curve* (кривая) в данном случае покажется вам неуместным, вы будете далеко не первым, кто подумал об этом. В данном случае термин *кривая* характеризует одномерную функцию, тогда как под термином «кривая» обычно подразумевается понятие двумерное.)

Различные параметры – `curve`, `easing`, `duration` и `rate` – тесно связаны друг с другом, и потому напомню их назначение еще раз:

`duration`

Продолжительность анимационного эффекта.

rate

Частоты смены кадров – разделив значение duration на rate, вы получите общее число отдельных кадров. Понятие частоты смены кадров имеет большое значение, потому что текущая позиция анимации эквивалентна отношению номера текущего кадра к номеру последнего кадра.

curve

Диапазон допустимых значений, которые передаются функции onAnimate. Каждое значение, передаваемое этой функции, вычисляется путем проецирования результата переходной функции easing на этот диапазон. Параметр curve представлен массивом из двух числовых значений, являющихся началом и концом допустимого диапазона значений.

easing

Функция, которая принимает входные значения в диапазоне от 0 до 1, соответствующие текущему положению процесса анимации. Результат переходной функции проецируется на диапазон «кривой», и полученное значение передается функции onAnimate. Примечательно, что если переходная функция вернет значение больше, чем 1.0, функция onAnimate получит значение выше конечного значения диапазона.

В завершение обсуждения параметров функции-конструктора объекта _Animate давайте предположим, что имеется некоторая анимация, продолжительность которой составляет 10 секунд, со скоростью смены кадров 1 раз в секунду. Область определения, представленная параметром curve, лежит в диапазоне от 50 до 100, а переходная функция имеет вид: function(x) { return 2*x; }.

Учитывая эти значения параметров, можно сказать, что переходная функция будет принимать 10 различных значений: 0.1, 0.2, 0.3,..., 1.0 и будет возвращать удвоенные значения. Значения, возвращаемые переходной функцией будут проецироваться на область определения, представленную параметром curve, и затем передаваться функции onAnimate, то есть функция onAnimate будет получать значения 50, 60, 70, 80, 90,..., 150.

Ниже приводится программный код, который вы можете опробовать:

```
new dojo._Animation({
    duration:10000,
    rate : 1000,
    curve: [50,100],
    easing : function(x) {
        console.log("easing: ", 2*x);
        return 2*x;
```

```
        },
        onAnimate:function(x){
            console.log("onAnimate: ", x);
        },
        onEnd:function(){
            console.log('all done.');
        }
    }).play();
```

## Скольжение

Эффект скольжения узла воспроизводится так же просто, как и эффект растворения/проявления. Для этого функции `dojo.fx.slideTo` передается ассоциативный массив с параметрами, как и в случае с функцией `animateProperty`. Допустимые параметры перечислены в табл. 8.5.

*Таблица 8.5. Параметры функции slideTo из библиотеки Core*

| Параметр | Тип | Комментарий |
|---|---|---|
| `node` | DOM Node | Узел, который будет скользить. |
| `duration` | Integer | Продолжительность воспроизведения эффекта в миллисекундах. Значение по умолчанию: 350. |
| `easing` | Function | Переходная функция, описывающая ускорение и/или замедление процесса. По умолчанию реализует формулу: `(0.5 + ((Math.sin((n + 1.5) * Math.PI))/2)`. Обратите внимание, что функция определена только в интервале от 0 до 1. |
| `left` | Integer | Координата x, где должен оказаться верхний левый угол узла по окончании скольжения. |
| `top` | Integer | Координата y, где должен оказаться верхний левый угол узла по окончании скольжения. |

Пример 8.3 иллюстрирует применение функции `sliteTo`. Та часть, которая отличает этот пример от предыдущего, выделена жирным шрифтом.

*Пример 8.3. Эффект скольжения узла*

```
<html>
    <head>
        <title>Animation Station</title>
        <style type="text/css">
            @import "http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css";
            .box {
                width : 200px;
                height : 200px;
```

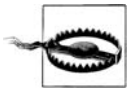
```
                margin : 5px;
                background : blue;
                text-align : center;
            }
        </style>
        <script
            type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js">
        </script>
        <script type="text/javascript">
            dojo.require("dojo.fx");
            dojo.addOnLoad(function() {
                var box = dojo.byId("box");
                dojo.connect(box, "onclick", function(evt) {
                    dojo.fx.slideTo({
                        node:box,
                        top : "200",
                        left : "200"
                    }).play();
                });
            });
        </script>
    </head>
    <body>
        <div id="box" class="box">Slide Me</div>
    </body>
</html>
```

## Свертывание

Кроме столь распространенных эффектов скольжения и проявления/исчезновения, не менее часто используются эффекты свертывания и развертывания, для реализации которых также имеются отдельные функции. Принцип их использования не будет для вас новым. Они принимают практически те же самые аргументы; их перечень приводится в табл. 8.6.

*Таблица 8.6. Параметры функций wipe из библиотеки Core*

| Параметр | Тип | Комментарий |
|---|---|---|
| node | DOM Node | Узел, который будет сворачиваться или разворачиваться. |
| duration | Integer | Продолжительность воспроизведения эффекта в миллисекундах. Значение по умолчанию: 350. |
| easing | Function | Переходная функция, описывающая ускорение и/или замедление процесса. По умолчанию реализует формулу: `(0.5 + ((Math.sin((n + 1.5) * Math.PI))/2)`. Обратите внимание, что функция определена только в интервале от 0 до 1. |



Имейте в виду, что в некоторых случаях значения `border`, `margin` и `padding`, связанные с узлом, оказывают воздействие на расположение окружающих элементов по окончании анимации.

Ниже следует еще один набор примеров для этой главы, основу которого составляет пример 8.4.

*Пример 8.4. Сворачивание узла*

```
<html>
    <head>
        <title>Animation Station</title>
        <style type="text/css">
            @import "http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css";
            .box {
                width : 200px;
                height : 200px;
                text-align : center;
                float : left;
                position : absolute;
                margin : 5px;
            }
        </style>
        <script
            type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js">
        </script>
        <script type="text/javascript">
            dojo.require("dojo.fx");

            dojo.addOnLoad(function() {
                var box = dojo.byId("box");
                dojo.connect(box, "onclick", function(evt) {
                    dojo.fx.wipeOut({
                        node:box
                    }).play();
                });
            });
        </script>
    </head>
    <body>
        <div class="box">Now you don't</div>
        <div id="box" style="background : blue" class="box">
            Now you see me...
        </div>
    </body>
</html>
```

Как и в прежнем примере, может оказаться интересным эксперимент с нестандартной переходной функцией. Подставьте нашу ранее подготовленную немонотонную переходную функцию и получите «пружинистый» эффект внесением следующих изменений в `addOnLoad`:

```
dojo.addOnLoad(function() {
```

```
    var box = dojo.byId("box");
    dojo.connect(box, "onclick", function(evt) {
        dojo.fx.wipeOut({
            node:box,
            easing : function(x) { return Math.pow(Math.sin(4*x),2); },
            duration : 5000
        }).play();
    });
});
```

Поскольку возвращаемые значения переходной функции то увеличиваются, то уменьшаются, то опять увеличиваются, объект `_Animation`, используемый функцией `wipeOut`, соответствующим образом изменяет высоту узла.

## Создание цепочек и комбинирование эффектов

Есть что-то удивительное в том, как объект скользит по экрану, растворяется и сворачивается, но на этом многообразие эффектов не заканчивается: вы можете использовать еще одну функцию в модуле `fx` библиотеки Core – `dojo.fx.chain`, позволяющую составлять цепочки из анимационных эффектов. Эта функция чрезвычайно проста, в том смысле, что принимает единственный аргумент – массив объектов `_Animation` и возвращает другой объект `_Animation`, который воспроизводит последовательность указанных эффектов. Давайте воспользуемся этой функцией, чтобы заставить прямоугольник сделать что-нибудь более интересное. В табл. 8.7 перечислены функции, используемые для комбинирования и составления цепочек эффектов.

*Таблица 8.7. Комбинирование и объединение в цепочки анимационных эффектов*

| Функция | Комментарий |
|---|---|
| `dojo.fx.chain(/* Array */ animations)` | Объединяет в цепочку анимационные эффекты, представленные массивом `animations`, и возвращает результат объединения, который можно запустить на воспроизведение как обычно. Получившаяся анимация воспроизводится как последовательное выполнение всех анимационных эффектов друг за другом. |
| `dojo.fx.combine(/* Array */ animations)` | Объединяет анимационные эффекты, представленные массивом `animations`, и возвращает результат объединения, который можно запустить на воспроизведение как обычно. Получившаяся анимация воспроизводится как одновременное выполнение всех анимационных эффектов. |

В версии Dojo 1.1 функции `chain` и `combine`, описываемые в этом разделе, имеют несколько известных проблем, связанных с тем, как вызываются методы `beforeBegin` и `onEnd`, когда воспроизво-

дится несколько анимационных эффектов. Суть проблемы заключается в том, что если вы в своем приложении опираетесь на обработчики этих событий, то вам лучше было бы воспользоваться функциями `dojo.connect` и `dojo.subscribe` для реализации своих собственных цепочек и комбинаций. Но для несложных задач функции `chain` и `combine` вполне пригодны.

Пример 8.5 демонстрирует прямоугольник, выполняющий зигзагообразные движения по экрану. Обратите внимание, что в примере определяется собственная переходная функция.

*Пример 8.5. Объединение анимационных эффектов в цепочку*

```
dojo.addOnLoad(function() {
    var box = dojo.byId("box");
    dojo.connect(box, "onclick", function(evt) {
        var easing = function(x) { return x; };
        var a1 = dojo.fx.slideTo({
            node:box,
            easing : easing,
            duration : 1000,
            top : "150",
            left : "300"
        });
        var a2 = dojo.fx.slideTo({
            node:box,
            easing : easing,
            duration : 400,
            top : "20",
            left : "350"
        });
        var a3 = dojo.fx.slideTo({
            node:box,
            easing : easing,
            duration : 800,
            top : "350",
            left : "400"
        });
        dojo.fx.chain([a1,a2,a3]).play();
    });
});
```

А теперь представим, что вам требуется одновременно воспроизвести эффект скольжения и растворения. Нет проблем. Функция `dojo.fx.combine`, имеющая такую же сигнатуру, как и `dojo.fx.chain`, сделает это в одно мгновение. Все анимационные эффекты, находящиеся в массиве, который передается в качестве аргумента, будут воспроизводиться одновременно. Для начала рассмотрим простую комбинацию эффектов скольжения и растворения. Пример 8.6 демонстрирует, как следует изменить функцию `addOnLoad`.

*Пример 8.6. Комбинированный эффект анимации*

```
dojo.addOnLoad(function() {
    var box = dojo.byId("box");
    dojo.connect(box, "onclick", function(evt) {
        var a1 = dojo.fx.slideTo({
            node:box,
            top : "150",
            left : "300"
        });
        var a2 = dojo.fadeOut({
            node:box
        });
        dojo.fx.combine([a1,a2]).play();
    });
});
```

Очень легко забыть, что функция `slideTo` находится в модуле `dojo.fx`, а функции `fadeIn` и `fadeOut` – в библиотеке Base, поэтому уделите минуту, чтобы убедиться, что вызов `dojo.fx.fadeIn` вызовет появление ошибки. Если вы забудете выполнить инструкцию `dojo.require("dojo.fx")` перед тем, как вызывать функции из модуля `dojo.fx`, это также приведет к появлению ошибки.

Учитывая, что функция `chain` возвращает единственный объект `_Animation`, давайте попробуем воспроизвести какой-нибудь более сложный эффект (но в действительности по-прежнему простой, потому что он будет основан на тех же принципах): в примере 8.7 мы объединили в цепочку несколько эффектов растворения/проявления и скомбинировали ее с цепочкой эффектов скольжения.

*Пример 8.7. Комбинирование цепочек анимационных эффектов*

```
dojo.addOnLoad(function() {
    var box = dojo.byId("box");
    dojo.connect(box, "onclick", function(evt) {

        //объединить в цепочку несколько эффектов скольжения
        var a1 = dojo.fx.slideTo({
            node:box,
            top : "150",
            left : "300"
        });
        var a2 = dojo.fx.slideTo({
            node:box,
            top : "20",
            left : "350"
        });
        var a3 = dojo.fx.slideTo({
            node:box,
            top : "350",
            left : "400"
        });
```

```
        var slides = dojo.fx.chain([a1,a2,a3]);

        //объединить в цепочку несколько эффектов растворения/проявления
        var a1 = dojo.fadeIn({
            node:box
        });
        var a2 = dojo.fadeOut({
            node:box
        });
        var a3 = dojo.fadeIn({
            node:box
        });
        var fades = dojo.fx.chain([a1,a2, a3]);

        //теперь создадим комбинированный эффект из двух цепочек
        dojo.fx.combine([slides, fades]).play();
    });
});
```

## Переключение

Класс `dojo.fx.Toggler` по сути является оберткой, позволяющей настраивать анимационный эффект *переключения* (отображения и сокрытия) узла. Конструктор класса принимает ассоциативный массив параметров, который включает функции `show` и `hide`, а также продолжительность действия функций `show` и `hide`. Класс `Toggler` хорош тем, что не даст запутаться, какой эффект он воспроизводит. Вы просто сообщаете ему, какие функции использовать, указываете продолжительности действия функций и затем вручную вызываете функции `show` и `hide`. Обе эти функции принимают необязательный параметр, который определяет величину задержки перед фактическим выполнением соответствующего действия (табл. 8.8).

*Таблица 8.8. Параметры функции wipe из библиотеки Core*

| Параметр | Тип | Комментарий |
|---|---|---|
| `node` | DOM Node | Узел, который будет переключаться. |
| `showFunc` | Function | Функция, возвращающая объект `_Animation`, который воспроизводит эффект отображения узла. Значение по умолчанию: `dojo.fadeIn`. |
| `hideFunc` | Function | Функция, возвращающая объект `_Animation`, который воспроизводит эффект сокрытия узла. Значение по умолчанию: `dojo.fadeOut`. |
| `showDuration` | Integer | Продолжительность выполнения функции `showFunc` в миллисекундах. Значение по умолчанию: 200 (миллисекунд) |
| `hideDuration` | Integer | Продолжительность выполнения функции `hideFunc` в миллисекундах. Значение по умолчанию: 200 (миллисекунд) |

В таблице 8.9 дается краткое описание методов класса.

*Таблица 8.9. Методы класса Toggler*

| Метод | Комментарий |
|---|---|
| show(/*Integer*/delay) | С помощью функции showFunc воспроизводит эффект отображения узла в течение времени, определяемого параметром showDuration. Необязательный параметр delay определяет время задержки перед фактическим началом эффекта. |
| hide(/*Integer*/delay) | С помощью функции hideFunc воспроизводит эффект сокрытия узла в течение времени, определяемого параметром hideDuration. Необязательный параметр delay определяет время задержки перед фактическим началом эффекта. |

Пример 8.8 содержит необходимый программный код следующей модификации функции addOnLoad для нашего рабочего примера, представленного в примере 8.4.

*Пример 8.8. Переключение узла*

```
dojo.addOnLoad(function() {
    var box = dojo.byId("box");
    var t = new dojo.fx.Toggler({
        node : box,
        showDuration : 1000,
        hideDuration : 1000
    });
    var visible = true;
    dojo.connect(box, "onclick", function(evt) {
        if (visible)
            t.hide();
        else
            t.show();
        visible = !visible;
    });
});
```

Если вы запустите этот пример, вы должны заметить, что после щелчка мышью на прямоугольнике с надписью «Now you see me...» (теперь меня видно), он постепенно растворяется, а после щелчка на прямоугольнике с надписью «Now you don't» (а теперь – нет) происходит проявление первого прямоугольника.

# Анимация + «перетащил и бросил» = забавно!

Действие механизма «перетащил и бросил» в сочетании с анимационными эффектами представляет собой весьма выразительную комбинацию. Потратим немного времени, чтобы поэкспериментировать со

следующим фрагментом программного кода, который объединяет базовую концепцию «перетащил и бросил» из предыдущей главы с тем, что вы узнали в этой. На рис. 8.6 приводится график переходной функции, используемой в этом примере.

```
<html>
    <head>
        <title>Animation + Drag and Drop = Fun!</title>

        <script
            type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js">
        </script>

        <script type="text/javascript">
            dojo.require("dojo.fx");
            dojo.require("dojo.dnd.move");
            dojo.addOnLoad(function(){
                var move = new dojo.dnd.Moveable(dojo.byId("ball"));
                var coords;
                dojo.subscribe("/dnd/move/start",function(e){
                    // сохранить координаты в момент начала перетаскивания
                    coords = dojo.coords(e.node);
                });

                //теперь вернуть изображение в начальное местоположение
                dojo.subscribe("/dnd/move/stop",function(e){
                    dojo.fx.slideTo({
                        node: e.node,
                        top: coords.t,
                        left: coords.l,
                        duration:1200,
                        easing : function(x) { return Math.pow(x,5);}
```

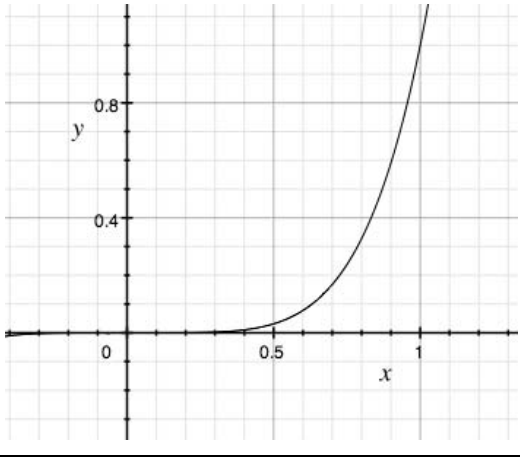



***Рис. 8.6.*** *График переходной функции x^5*

```
                    }).play( );
                }});
            });
        </script>
    </head>
    <body>
        <!-- Вставьте в страницу любое изображение вместо ball.png -->
        <img style="position : absolute; left : 300px; top : 300px;"
             id="ball"
             src="ball.png"/>
    </body>
</html>
```

Программный код этого примера определяет момент начала перетаскивания и сохраняет начальные координаты узла. Затем он ожидает, пока не возникнет событие окончания перетаскивания, и в этот момент начинает перемещение изображения в первоначальное местоположение в соответствии с переходной функцией. Переходная функция обеспечивает невысокую начальную скорость перемещения, которая быстро увеличивается ближе к концу пути.

# Цвета

Анимационные и другие эффекты на странице часто зависят от вычисления значения цвета. В библиотеке Base имеется элементарный класс `Color`, который реализует основные операции, связанные с вычислением значений цвета, преобразованием их в шестнадцатеричное представление и обратно и многие другие. Кроме того, в библиотеке присутствуют несколько вспомогательных функций, предназначенных для выполнения наиболее общих операций над цветами.

## Создание и смешивание цветов

Класс `Color` имеет довольно гибкий конструктор, который может принимать названия цветов в виде строк, значения цветов в шестнадцатеричном представлении и в виде массива значений RGB.[1] В примере 8.9 демонстрируется создание двух объектов `Color` и использование функций из библиотеки Base для смешивания цветов.

*Пример 8.9. Смешивание цветов, представленных объектами Color*

```
<html>
    <head>
```

[1] Аббревиатура RGB, от английского «red green blue» (красный зеленый синий), используется для обозначения одного из стандартных способов представления цветов в CSS. Аббревиатура RGBA, от английского «red green blue alpha» (красный зеленый синий альфа), обозначает четырехкомпонентные значения цвета, в которых присутствует значение степени прозрачности.

```
        <title></title>
        <script
            type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js">
        </script>
        <script type="text/javascript">
            dojo.addOnLoad(function() {
                var blue = new dojo.Color("#0000ff"); //можно использовать
                                                      //значение "blue"
                var red = new dojo.Color([255, 0, 0]);
                var purple = dojo.blendColors(blue, red, 0.5);
                dojo.style("foo", "background", purple.toCss());
            });
        </script>
    </head>
    <body>
        <div id="foo" style="width:200px; height:200px; padding:5px;"></div>
    </body>
</html>
```

Функция `blendColors` получает объекты `Colors` с красным и синим цветами и смешивает их в пропорции 50/50, воспроизводя значение RGB (128, 0, 128) – нейтральный оттенок фиолетового цвета. Вы могли бы реализовать эти вычисления самостоятельно – в этом нет ничего сложного, но нет и ничего увлекательного!

В табл. 8.10 приводится краткое описание класса `Color`, реализованного в библиотеке Base.

*Таблица 8.10. Функциональные возможности класса Color из библиотеки Base*

| Метод | Комментарий |
|---|---|
| `Color(/* Array \| String */color)` | Функция-конструктор, которая принимает массив значений RGB или RGBA, или строку с названием цвета, например `"blue"`, или строку с шестнадцатеричным значением, например `"#0000ff"`. При вызове без аргументов создает объект `Color` со значением цвета в формате RGBA в виде кортежа (255, 255, 255, 1). |
| `setColor(/* Array \| String \| Object */ color)` | Оперирует существующим объектом `Color`, определяя значение цвета, как это делает функция-конструктор. Предпочтительный способ повторного использования существующего объекта `Color`. |
| `toRgb()` | Возвращает строковое значение, выражающее значение цвета объекта `Color` в формате RGB, например (128, 0, 128). |

| Метод | Комментарий |
|---|---|
| toRgba() | Возвращает строковое значение, выражающее значение цвета объекта Color в формате RGBA, например (128, 0, 128, 0.5). |
| toHex() | Возвращает строковое значение, выражающее значение цвета объекта Color в шестнадцатеричном формате, например "#800080". |
| toCss(/* Boolean */ includeAlpha) | Возвращает строку с допустимым в CSS значением цвета в формате RGB, например (128, 0, 128). По умолчанию аргумент includeAlpha имеет значение false. Этот метод часто используется для преобразования объектов Color с целью использовать их в определениях стилей. |
| toString() | Возвращает стандартное строковое значение для объекта Color в формате RGBA. |

Большинство броузеров в настоящее время отклоняются от требований спецификации CSS2 и не поддерживают кортежи RGBA для обозначения цвета, поэтому функция toCss() (без входного параметра), скорее всего, будет для вас наиболее предпочтительным способом получения значения цвета, допустимого для передачи таким методам, как dojo.style. Если вам потребуется определить степень прозрачности узла, используйте для этого свойство CSS opacity.[1]

## Поддержка RGBA в CSS3

Формат RGBA представления значения цвета, включенный в состав спецификации CSS3, является очень выразительным и удобным. В следующем фрагменте кода разметки демонстрируется создание двух перекрывающихся квадратов синего и красного цвета, а на рис. 8.7 показано, как в результате перекрытия получается квадрат фиолетового цвета:

```
<div style="width:200px; height:200px; padding:5px;
            background:rgba(0,0,255,0.5)"></div>
<div style="position:absolute; left:100px; top:100px; width:200px;
            height: 200px; padding:5px;
            background:rgba(255,0,0,0.5)"></div>
```

Броузер Firefox 3 поддерживает формат RGBA. Подробнее о поддержке CSS3 в Firefox можно прочитать по адресу: *http://developer.mozilla.org/en/docs/CSS_improvements_in_Firefox_3*.

[1] Это свойство определяет не степень прозрачности, а степень непрозрачности, о чем говорится в примечании «СЛЕДЫ» ниже. – *Прим. перев.*

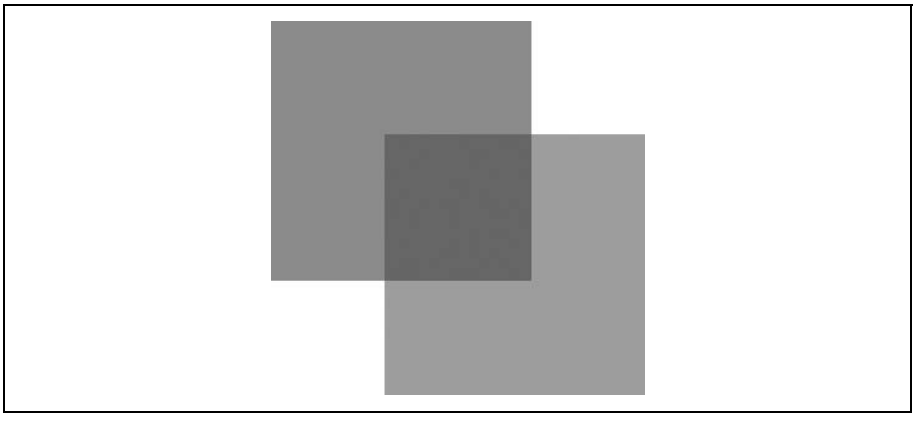


***Рис. 8.7.*** *Пример действия эффекта полупрозрачности при использовании формата rgba в описаниях цвета в Firefox 3*

Прозрачность и непрозрачность – это два противоположных понятия. Если значение непрозрачности равно 1.0, следовательно, объект полностью непрозрачен и значение его прозрачности равно 0.0. Если значение непрозрачности равно 0.1, то объект будет на 90% прозрачным и будет едва виден.

## Именованные значения цветов, доступных в библиотеке Base

В библиотеке Base присутствуют предопределенные названия цветов, хранящиеся в ассоциативном массиве `dojo.Color.named`, который отображает названия цветов в значения RGB. Например, чтобы быстро получить значение темно-бордового (maroon) цвета в формате RGB в виде массива `[128, 0, 0]`, достаточно просто сослаться на имя `dojo.Color.named.maroon`. В табл. 8.11 приводится список названий цветов, встроенных в библиотеку Base. Хотя вы, скорее всего, предпочтете использовать возможность напрямую управлять объектами `Color`, тем не менее массив `dojo.Color.named` по-прежнему может оказаться полезным во время разработки.

*Таблица 8.11. Названия цветов, доступные в библиотеке Base*

| Название | Red (красный) | Green (зеленый) | Blue (синий) |
|---|---|---|---|
| black (черный) | 0 | 0 | 0 |
| silver (светло-серый) | 192 | 192 | 192 |
| gray (серый) | 128 | 128 | 128 |

| Название | Red (красный) | Green (зеленый) | Blue (синий) |
|---|---|---|---|
| white (белый) | 255 | 255 | 255 |
| maroon (темно-бордовый) | 128 | 0 | 0 |
| red (красный) | 255 | 0 | 0 |
| purple (фиолетовый) | 128 | 0 | 128 |
| fuchsia (розовый) | 255 | 0 | 255 |
| green (зеленый) | 0 | 128 | 0 |
| lime (светло-зеленый) | 0 | 255 | 0 |
| olive (желтовато-зеленый, оливковый) | 128 | 128 | 0 |
| yellow (желтый) | 255 | 255 | 0 |
| navy (темно-синий) | 0 | 0 | 128 |
| blue (синий) | 0 | 0 | 255 |
| teal (бирюзовый) | 0 | 128 | 128 |
| aqua (голубой) | 0 | 255 | 255 |

Идентификатор `dojo.Color.named` недоступен как свойство объекта `Color`. Это статическая коллекция значений цвета, для использования которой не требуется создавать объект. Попытка обратиться к свойству `.named` экземпляра объекта приведет к появлению ошибки.

## Дополнительные значения цвета в библиотеке Core

Массив `dojo.Color.named` можно расширить более чем сотней дополнительных названий цветов, включая названия, определяемые спецификациями CSS3, и заканчивая вариантами названий из SVG 1.0, для чего следует выполнить инструкцию `dojo.require("dojo.colors")` (табл. 8.12). Обратите также внимание на то, что с помощью функции `animateProperty`, о которой вы узнали выше, можно реализовать постепенное изменение свойства `backgroundColor`. Например, в качестве значений `start` и `end` можно было бы использовать `"black"` и `"white"`, `"white"` и `"#43fab4"`, и т. д.

Помимо дополнительных цветов в массиве `dojo.Colors.named`, модуль `dojo.colors` содержит дополнительное расширение конструктора класса `Color`, обеспечивая тем самым возможность передачи значения цвета в форматах HSL и HSLA. Формат HSL обеспечивает более точное описание цвета, чем формат RGB, в терминах оттенка (Hue), контрастности (Saturation) и яркости (Lightness). Подробнее о модуле CSS Color можно прочитать по адресу: *http://www.w3.org/TR/css3-iccprof*.

*Таблица 8.11. Дополнительные названия цветов, доступные в библиотеке Core*[1]

| Название | Red (красный) | Green (зеленый) | Blue (синий) |
|---|---|---|---|
| aliceblue | 240 | 248 | 255 |
| antiquewhite | 250 | 235 | 215 |
| aquamarine | 127 | 255 | 212 |
| azure | 240 | 255 | 255 |
| beige | 245 | 245 | 220 |
| bisque | 255 | 228 | 196 |
| blanchedalmond | 255 | 235 | 205 |
| blueviolet | 138 | 43 | 226 |
| brown | 165 | 42 | 42 |
| burlywood | 222 | 184 | 135 |
| cadetblue | 95 | 158 | 160 |
| chartreuse | 127 | 255 | 0 |
| chocolate | 210 | 105 | 30 |
| coral | 255 | 127 | 80 |
| cornflowerblue | 100 | 149 | 237 |
| cornsilk | 255 | 248 | 220 |
| crimson | 220 | 20 | 60 |
| cyan | 0 | 255 | 255 |
| darkblue | 0 | 0 | 139 |
| darkcyan | 0 | 139 | 139 |
| darkgoldenrod | 184 | 134 | 11 |
| darkgray | 169 | 169 | 169 |
| darkgreen | 0 | 100 | 0 |
| darkgrey | 169 | 169 | 169 |
| darkkhaki | 189 | 183 | 107 |
| darkmagenta | 139 | 0 | 139 |
| darkolivegreen | 85 | 107 | 47 |
| darkorange | 255 | 140 | 0 |
| darkorchid | 153 | 50 | 204 |
| darkred | 139 | 0 | 0 |

[1] Как выглядят эти цвета, можно посмотреть по адресу: *http://ru.wikipedia.org/wiki/Википедия:Таблица_цветов*. – *Прим. перев.*

| Название | Red (красный) | Green (зеленый) | Blue (синий) |
|---|---|---|---|
| darksalmon | 233 | 150 | 122 |
| darkseagreen | 143 | 188 | 143 |
| darkslateblue | 72 | 61 | 139 |
| darkslategray | 47 | 79 | 79 |
| darkslategrey | 47 | 79 | 79 |
| darkturquoise | 0 | 206 | 209 |
| darkviolet | 148 | 0 | 211 |
| deeppink | 255 | 20 | 147 |
| deepskyblue | 0 | 191 | 255 |
| dimgray | 105 | 105 | 105 |
| dimgrey | 105 | 105 | 105 |
| dodgerblue | 30 | 144 | 255 |
| firebrick | 178 | 34 | 34 |
| floralwhite | 255 | 250 | 240 |
| forestgreen | 34 | 139 | 34 |
| gainsboro | 220 | 220 | 220 |
| ghostwhite | 248 | 248 | 255 |
| gold | 255 | 215 | 0 |
| goldenrod | 218 | 165 | 32 |
| greenyellow | 173 | 255 | 47 |
| grey | 128 | 128 | 128 |
| honeydew | 240 | 255 | 240 |
| hotpink | 255 | 105 | 180 |
| indianred | 205 | 92 | 92 |
| indigo | 75 | 0 | 130 |
| ivory | 255 | 255 | 240 |
| khaki | 240 | 230 | 140 |
| lavender | 230 | 230 | 250 |
| lavenderblush | 255 | 240 | 245 |
| lawngreen | 124 | 252 | 0 |
| lemonchiffon | 255 | 250 | 205 |
| lightblue | 173 | 216 | 230 |
| lightcoral | 240 | 128 | 128 |
| lightcyan | 224 | 255 | 255 |

| Название | Red (красный) | Green (зеленый) | Blue (синий) |
|---|---|---|---|
| lightgoldenrodyellow | 250 | 250 | 210 |
| lightgray | 211 | 211 | 211 |
| lightgreen | 144 | 238 | 144 |
| lightgrey | 211 | 211 | 211 |
| lightpink | 255 | 182 | 193 |
| lightsalmon | 255 | 160 | 122 |
| lightseagreen | 32 | 178 | 170 |
| lightskyblue | 135 | 206 | 250 |
| lightslategray | 119 | 136 | 153 |
| lightslategrey | 119 | 136 | 153 |
| lightsteelblue | 176 | 196 | 222 |
| lightyellow | 255 | 255 | 224 |
| limegreen | 50 | 205 | 50 |
| linen | 250 | 240 | 230 |
| magenta | 255 | 0 | 255 |
| mediumaquamarine | 102 | 205 | 170 |
| mediumblue | 0 | 0 | 205 |
| mediumorchid | 186 | 85 | 211 |
| mediumpurple | 147 | 112 | 219 |
| mediumseagreen | 60 | 179 | 113 |
| mediumslateblue | 123 | 104 | 238 |
| mediumspringgreen | 0 | 250 | 154 |
| mediumturquoise | 72 | 209 | 204 |
| mediumvioletred | 199 | 21 | 133 |
| midnightblue | 25 | 25 | 112 |
| mintcream | 245 | 255 | 250 |
| mistyrose | 255 | 228 | 225 |
| moccasin | 255 | 228 | 181 |
| navajowhite | 255 | 222 | 173 |
| oldlace | 253 | 245 | 230 |
| olivedrab | 107 | 142 | 35 |
| orange | 255 | 165 | 0 |
| orangered | 255 | 69 | 0 |
| orchid | 218 | 112 | 214 |

| Название | Red (красный) | Green (зеленый) | Blue (синий) |
|---|---|---|---|
| palegoldenrod | 238 | 232 | 170 |
| palegreen | 152 | 251 | 152 |
| paleturquoise | 175 | 238 | 238 |
| palevioletred | 219 | 112 | 147 |
| papayawhip | 255 | 239 | 213 |
| peachpuff | 255 | 218 | 185 |
| peru | 205 | 133 | 63 |
| pink | 255 | 192 | 203 |
| plum | 221 | 160 | 221 |
| powderblue | 176 | 224 | 230 |
| rosybrown | 188 | 143 | 143 |
| royalblue | 65 | 105 | 225 |
| saddlebrown | 139 | 69 | 19 |
| salmon | 250 | 128 | 114 |
| sandybrown | 244 | 164 | 96 |
| seagreen | 46 | 139 | 87 |
| seashell | 255 | 245 | 238 |
| sienna | 160 | 82 | 45 |
| skyblue | 135 | 206 | 235 |
| slateblue | 106 | 90 | 205 |
| slategray | 112 | 128 | 144 |
| slategrey | 112 | 128 | 144 |
| snow | 255 | 250 | 250 |
| springgreen | 0 | 255 | 127 |
| steelblue | 70 | 130 | 180 |
| tan | 210 | 180 | 140 |
| thistle | 216 | 191 | 216 |
| tomato | 255 | 99 | 71 |
| transparent | 0 | 0 | 0 |
| turquoise | 64 | 224 | 208 |
| violet | 238 | 130 | 238 |
| wheat | 245 | 222 | 179 |
| whitesmoke | 245 | 245 | 245 |
| yellowgreen | 154 | 205 | 50 |

# В заключение

В этой главе последовательно рассматривались инструменты в библиотеках Base и Core, предназначенные для создания анимационных эффектов. При разумном использовании анимационные эффекты действительно могут придать вашему приложению особую *привлекательность*, отличающую его от всех остальных приложений. После прочтения этой главы вы должны:

- Уметь использовать функцию `animateProperty` из библиотеки Base для управления любыми свойствами CSS
- Понимать назначение переходной функции, параметров `duration` и `rate` в объекте `_Animation`
- Знать о средствах, предоставляемых библиотекой Core, расширяющих поддержку анимации, присутствующей в библиотеке Base
- Уметь пользоваться средствами библиотеки Core, обеспечивающими поддержку дополнительных эффектов, включая сворачивание и скольжение
- Уметь с помощью функции `dojo.fx.chain` составлять цепочки анимационных эффектов, выполняющихся последовательно, а также одновременно воспроизводить несколько анимационных эффектов с помощью функции `dojo.fx.combine`
- Уметь с помощью класса `dojo.fx.Toggler` отображать и скрывать узлы, используя его простой и универсальный интерфейс
- Понимать, как комбинировать анимационные эффекты с механизмом «перетащил и бросил» для получения хорошо организованных интерактивных страниц
- Уметь создавать и эффективно использовать объекты `Color`, чтобы ликвидировать необходимость вручную выполнять вычисления значений цвета в программном коде

В модуле `dojox.gfx` имеется множество удивительных инструментов для работы с графикой и анимационными эффектами, основанных на применении средств SVG, VML и Silverlight.

В следующей главе будет рассматриваться тема абстракции данных.

# 9

# Абстракция данных

Настоящим проклятием веб-разработки является необходимость создавать функции преобразования данных, возвращаемых сервером, в формат, удобный для использования внутри приложения. Несмотря на то что была разработана масса замечательных функций для разбора таких распространенных форматов представления данных, как CSV (comma-separated values – значения, разделенные запятыми) и JSON, тем не менее все еще приходится использовать большой объем шаблонного программного кода для приема данных от сервера, отправки обновленных значений обратно на сервер, синхронизации локальных данных с данными на сервере и т. д. В этой главе будут представлены функции инструментального набора Dojo, обеспечивающие единообразный интерфейс для работы с источниками данных независимо от того, где они находятся, как осуществляется доступ к ним на транспортном уровне и в каком формате они поставляют данные.

## Изменение схемы работы с данными

Механизм взаимодействия с источниками данных, входящий в состав инструментария, нельзя отнести к разряду высоких технологий, но для его работы необходимо изменить модель представления данных, чтобы позволить рассматривать источники данных как локальные ресурсы, доступные посредством унифицированного прикладного интерфейса. Традиционные подходы, как правило, требуют восприятия данных как удаленного ресурса, что всегда влечет за собой необходимость писать шаблонный программный код, выполняющий прием данных от сервера, отправку обновленных значений обратно на сервер, синхронизацию локальных данных с данными на сервере и обработку данных в различных форматах. Одна из основных проблем, сложившихся исторически, состоит в том, что практически каждому разработчику приходилось заново изобретать колесо и практически

в каждом приложении предлагался свой собственный одноразовый подход к решению проблем, связанных с управлением данными.

Инструментальный набор Dojo обеспечивает набор функций в виде модуля `dojo.data`, которые предоставляют стандартные средства взаимодействия с произвольными источниками данных, как показано на рис. 9.1. Это позволяет прикладному программисту избежать необходимости решать запутанные проблемы, связанные с получением, анализом и управлением данными. Модуль `dojo.data` обеспечивает стандартизованный способ взаимодействия как с локальными данными, так и с удаленными, достоинства которого сложно переоценить, когда приходится иметь дело со все большими и большими объемами данных по мере развития приложения. Что особенно важно: как только вы разработали интерфейс для работы с данными в определенном формате, он превращается в стандартный ресурс, который можно повторно использовать и при желании распространять. Вообще говоря, такого рода стандартные ресурсы позволяют разработчикам приложений работать более эффективно, сосредотачиваясь на решении более интересных задач, чем управление вводом-выводом.

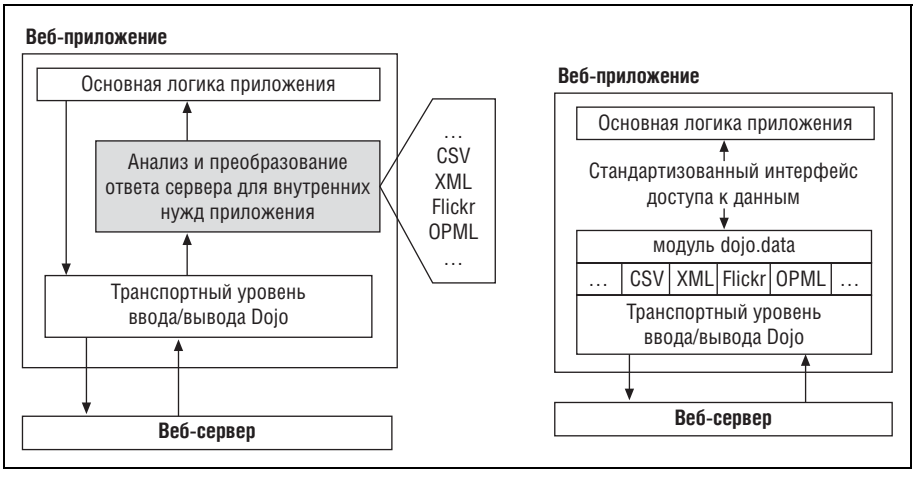



***Рис. 9.1.*** *Слева: традиционный шаблон доступа к произвольным источникам данных из приложения; справа: использование абстракции dojo.data для доступа к произвольным источникам данных*

# Обзор прикладного интерфейса доступа к данным

Базовая единица модуля `dojo.data` называется *элементом*, которая состоит из пар ключ/значение, называемых в терминологии `dojo.data` *атрибутами* и *значениями атрибутов;* для понимания сущности вы можете представлять себе элемент как обычный объект JavaScript. Несмотря на то что в основе реализации может лежать объект Java-

Script, тем не менее следует быть внимательным при использовании предоставляемых функций для доступа к нему, так как внутренняя реализация может совершенно не совпадать с этим представлением. Например, для хранения данных определенных типов некоторые абстракции данных могут использовать модель DOM из соображений эффективности или производить загрузку данных в процессе выполнения, хотя может создаваться впечатление, будто они уже стали локальными. В подобных случаях обращение к элементу как к простому объекту JavaScript, скорее всего, приведет к появлению ошибки. К конкретным функциям, обеспечивающим доступ к элементам, мы вернемся в следующем разделе.

Сказать, что элемент имеет атрибут, но этот атрибут не имеет значения – все равно, что сказать, что у элемента вообще нет такого атрибута. Другими словами, бессмысленно думать об атрибутах без значений, потому что атрибуты всегда находятся в конкретном состоянии.

Прежде чем углубляться в возможности какого-либо конкретного интерфейса, полезно окинуть взглядом общую картину. Ниже приводится обзор прикладных интерфейсов модуля `dojo.data` с кратким описанием того, что конкретный интерфейс предоставляет в распоряжение разработчика приложений. Это лишь интерфейсы, а не реализации; каждый конкретный механизм хранения данных в `dojo.data` реализует один или более следующих прикладных интерфейсов:

`dojo.data.api.Read`

Обеспечивает универсальное средство чтения, поиска, сортировки и фильтрации элементов данных.

`dojo.data.api.Write`

Обеспечивает универсальное средство создания, удаления и обновления элементов данных.

`dojo.data.api.Identity`

Обеспечивает универсальное средство доступа к элементам посредством уникальных идентификаторов.

`dojo.data.api.Notification`

Обеспечивает универсальное средство уведомления о таких изменениях, как создание, удаление или обновление элементов данных.

Далее в этой главе мы подробно разберем каждый из этих интерфейсов и рассмотрим множество примеров, чтобы вы могли максимально использовать возможности модуля `dojo.data` в своих приложениях.

# Интерфейсы

В этом разделе дается описание прикладных интерфейсов доступа к данным. Если у вас пока недостаточно опыта, просто просмотрите этот

раздел, чтобы получить общее представление о возможностях, а затем вернитесь сюда, когда прочитаете остальную часть главы, где исследуются более конкретные примеры.

## Интерфейс Read

Каждый из механизмов хранения данных должен реализовать интерфейс `dojo.data.api.Read`, потому что этот интерфейс обеспечивает возможность получения, обработки и использования данных, что является необходимой предпосылкой для любых других операций. Полная спецификация интерфейса приводится в табл. 9.1. В следующем разделе обсуждается реализация `ItemFileReadStore`, входящая в состав инструментального набора.

В сигнатурах функций, которые приводятся ниже, используется описатель типа `dojo.data.api.Item` – чтобы подчеркнуть, что подразумевается понятие элемента в терминах модуля `dojo.data`, хотя *элемент* – это достаточно абстрактная концепция.

*Таблица 9.1. Интерфейс dojo.data.api.Read*

| Имя | Комментарий |
|---|---|
| `getValue(/*dojo.data.api.Item*/item, /*String*/attribute, /*Any?*/default)` | Исходя из аргументов `item` и `attribute`, возвращает значение атрибута. Если атрибут отсутствует, возвращается значение `undefined` (значение `null` возвращается, только если атрибут действительно имеет значение `null`). При желании можно использовать необязательный аргумент `default`, значение которого будет возвращаться в случае отсутствия атрибута. |
| `getValues(/*dojo.data.api.Item*/item, /*String*/attribute)` | Действует так же, как и функция `getValue`, за исключением того, что позволяет извлекать данные из атрибутов, способных иметь множество значений. Всегда возвращает массив независимо от числа возвращаемых значений. Вы *всегда* должны использовать функцию `getValues` при обращении к атрибутам, способным иметь множество значений. |
| `getAttributes(/*dojo.data.api.Item*/item)` | Анализирует элемент и возвращает массив строковых значений, соответствующих именам атрибутов элемента. Если элемент не имеет атрибутов, возвращается пустой массив. |
| `hasAttribute(/*dojo.data.api.Item*/item, /*String*/attribute)` | Возвращает значение `true`, если элемент имеет указанный атрибут. |

| Имя | Комментарий |
|---|---|
| containsValue(/*dojo.data.api.Item*/item, /*String*/attribute, /*Any*/value) | Возвращает значение true, если элемент имеет указанный атрибут, а атрибут имеет значение, то есть когда функция getValues может вернуть непустой массив. |
| isItem(/*Any*/item) | Возвращает значение true, если элемент происходит из указанного хранилища данных. Возвращает значение false, если элемент происходит из любого другого хранилища. (Эта функция особенно удобна, когда локальная переменная ссылается на элементы устаревших данных, что достаточно обычно для хранилищ, реализующих интерфейс Write.) |
| isItemLoaded(/*Any*/item) | Возвращает значение true, если элемент был загружен и доступен локально. |
| loadItem(/*Object*/args) | Загружает элемент, для того чтобы последующий вызов isItemLoaded вернул значение true. Объект args предоставляет следующие ключи:<br>item<br>Объект, содержащий критерии выбора загружаемого элемента (возможно, это будет идентификатор или подмножество идентификационной информации).<br>onItem(/*dojo.data.api.Item*/item)<br>Функция обратного вызова, которая запускается по окончании загрузки элемента, – загруженный элемент передается функции в виде аргумента.<br>onError(/*Object*/error)<br>Функция обратного вызова, которая запускается в случае появления ошибки во время загрузки элемента, – в качестве аргумента ей передается объект с информацией об ошибке.<br>scope<br>Объект, являющийся контекстом функций обратного вызова. |
| fetch(/*Object*/args) | Выполняет запрос и обеспечивает асинхронное обращение к функциям обратного вызова, имеющим отношение к обработке событий, происходящих в ходе выполнения запроса. Возвращает dojo.data.api.Request.<br>Основная роль объекта args – предоставить метод abort(), который может вызываться, чтобы прервать работу функции fetch. В число аргументов могут входить следующие девять параметров, которые обязательно должны предусматриваться любыми реализациями: |

*Таблица 9.1 (продолжение)*

| Имя | Комментарий |
|---|---|
| | `query`<br>Строка или объект, содержащий критерии выбора (аналог предложения SELECT в языке SQL). Обратите внимание, что синтаксис зависит от конкретной реализации.<br>`queryOptions`<br>Объект, содержащий дополнительные параметры запроса. Все реализации должны стараться обеспечить поддержку таких параметров, как: `ignoreCase` (`Boolean`, значение по умолчанию `false`), обеспечивающий возможность поиска без учета регистра символов, и `deep` (`Boolean`, значение по умолчанию `false`), обеспечивающий возможность запрашивать не только корневые элементы, но и все дочерние элементы.<br>`onBegin (/*Integer*/size,`<br>`/*dojo.data.api.Request*/request)`<br>Вызывается перед первым обращением к функции обратного вызова `onItem`. Аргумент `size` определяет общее число элементов, а аргумент `request` содержит оригинальный запрос, переданный в параметре `query`. Если значение `size` не известно, должно передаваться значение `-1`. Значение аргумента `size` может не соответствовать общему числу возвращаемых элементов, так как их количество может быть уменьшено параметрами `start` и `count`.<br>`onComplete (/*Array*/items,`<br>`/*dojo.data.api.Request*/request)`<br>Вызывается сразу после последнего обращения к функции обратного вызова `onItem`. Если функция `onItem` не определена, аргумент `items` должен быть представлен массивом всех элементов, соответствующих запросу, в противном случае этот аргумент может иметь значение `null`. Аргумент `request` содержит оригинальный запрос, переданный в параметре `query`.<br>`onError(/*Object*/error,`<br>`/*dojo.data.api.Request*/request)`<br>Вызывается, если в ходе выполнения запроса возникла ошибка. Аргумент `error` содержит информацию об ошибке, а аргумент `request` содержит оригинальный запрос, переданный в параметре `query`.<br>`onItem(/*dojo.data.api.Item*/item,`<br>`/*dojo.data.api.Request*/request)`<br>Вызывается для каждого элемента, который возвращается как элемент. Аргумент `request` содержит оригинальный запрос, переданный в параметре `query`. |

| Имя | Комментарий |
|---|---|
| | `scope (Object)`<br>Если определен, все функции обратного вызова должны запускаться в указанном контексте, в противном случае они должны запускаться в глобальном контексте.<br>`start (Integer)`<br>Начальное смещение в массиве возвращаемых результатов (аналог предложения OFFSET в языке SQL).<br>`count (Integer)`<br>Ограничивает число возвращаемых элементов (аналог предложения LIMIT в языке SQL).<br>`sort (Array)`<br>Массив объектов JavaScript, содержащих критерии сортировки для каждого атрибута. Все объекты применяются последовательно и каждый должен определять ключ атрибута, идентифицирующий имя атрибута, и ключ, задающий направление сортировки – по возрастанию или по убыванию. |
| `getFeatures()` | Возвращает объект, содержащий пары ключ/значение, определяющие, как интерфейсы `dojo.data` реализованы в данном механизме. Например, любая реализация интерфейса `dojo.data.api.Read` должна возвращать `{'dojo.data.api.Read' : true}` для данных, доступных только для чтения. |
| `close(/*dojo.data.api.Request*/request)` | Используется для выгрузки всей информации, связанной с определенным запросом `request`, что может включать в себя очистку кеша, закрытие соединений и прочее. Ожидается, что в качестве аргумента функция получит объект, возвращенный функцией `fetch`. Для некоторых реализаций эта функция может ничего не выполнять. |
| `getLabel(/*dojo.data.api.Item*/item)` | Используется для получения удобочитаемой метки элемента, которая обычно содержит некоторое идентификационное описание. Текст метки может состоять из некоторой комбинации атрибутов. |
| `getLabelAttributes(/*dojo.data.api.Item*/item)` | Возвращает массив атрибутов, которые участвуют в создании меток элементов. Удобна для разработчиков пользовательского интерфейса, позволяя им оценить, какие атрибуты удобнее для отображения, чтобы избыточную информацию можно было скрыть, когда необходимо выводить метки элементов. |

## Интерфейс Identity

Интерфейс Identity, который приводится в табл. 9.2, основан на интерфейсе Read и предоставляет несколько дополнительных функций извлечения элементов на основе их идентификационной информации. Обратите внимание, что интерфейс Read не делает никаких предположений об уникальности элементов и, определенно, существуют ситуации, когда исследовать элементы на идентичность не требуется; как раз здесь и проходит граница, разделяющая эти два интерфейса. Если проводить аналогии с базами данных, интерфейс Identity, с определенными допущениями, можно представлять как первичный ключ, который уникальным образом идентифицирует каждую запись. Довольно часто интерфейс Identity бывает необходим виджетам, предназначенным для работы с данными, особенно когда обеспечиваются функциональные особенности интерфейса Write. (Интерфейс Write рассматривается следующим.)

*Таблица 9.2. Интерфейс dojo.data.api.Identity*

| Имя | Комментарий |
|---|---|
| getFeatures() | Смотрите описание dojo.data.api.Read. Возвращает:<br>{<br>'dojo.data.api.Read' : true,<br>'dojo.data.api.Identity' : true<br>} |
| getIdentity(/*dojo.data.api.Item*/item) | Возвращает уникальный идентификатор элемента item, который может быть строкой или объектом, обладающим методом toString. |
| getIdentityAttributes (/*dojo.data.api.Item*/item) | Возвращает массив имен атрибутов, которые используются для идентификации. В большинстве случаев это единственный атрибут, который явно обеспечивает однозначную идентификацию, но может быть и несколько атрибутов – в зависимости от особенностей фактического источника данных. Эта функция часто используется при отображении информации для сокрытия атрибутов, используемых в целях идентификации. |
| fetchItemByIdentity(/*Object*/ args) | Извлекает элемент, используя идентификационную информацию. Если запрошенный элемент отсутствует, реализация должна возвращать значение null. Аргумент args может содержать следующие ключи:<br>identity<br>Строка или объект с функцией toString, который используется, чтобы получить ссылку на требуемый элемент. |

| Имя | Комментарий |
|---|---|
| | `onError(/*Object*/error)`<br>Вызывается в случае появления ошибки во время выполнения запроса. Аргумент `error` содержит информацию об ошибке.<br>`onItem(/*dojo.data.api.Item*/item)`<br>Вызывается для каждого возвращаемого элемента. Сам элемент передается в виде аргумента `item`.<br>`scope (Object)`<br>Если определен, все функции обратного вызова должны запускаться в указанном контексте, в противном случае они должны запускаться в глобальном контексте. |

## Интерфейс Write

Интерфейс `Write`, который приводится в табл. 9.3, дополняет интерфейс `Read` и включает средства создания, удаления и изменения элементов, что влечет за собой такие проблемы управления данными, как определение *грязных* (*dirty*) элементов, когда локальная копия в памяти не синхронизирована с копией на сервере, и выполнение операций ввода/вывода, таких как сохранение элементов.

*Таблица 9.3. Интерфейс dojo.data.api.Write*

| Имя | Комментарий |
|---|---|
| `getFeatures()` | Смотрите описание `dojo.data.api.Read`. Возвращает:<br>`{`<br>`'dojo.data.api.Read' : true,`<br>`'dojo.data.api.Identity' : true`<br>`}` |
| `newItem(/*Object?*/args, /*Object?*/parentItem)` | Возвращает вновь созданный элемент, устанавливая атрибуты в соответствии с парами ключ/значение в объекте `args`, которые обычно непосредственно отображаются на значения атрибутов. В реализациях, поддерживающих возможность создания иерархий элементов, аргумент `parentItem` обеспечивает идентификационную информацию о родителе нового элемента и атрибуте родителя, которому должен быть присвоен новый элемент (обычно это предполагает, что этот атрибут родителя может содержать сразу несколько значений и вновь созданный элемент просто добавляется к уже имеющимся значениям). |

*Таблица 9.3 (продолжение)*

| Имя | Комментарий |
|---|---|
| deleteItem(/*dojo.data.api.Item*/ item) | Удаляет элемент из хранилища и возвращает значение типа Boolean, свидетельствующее об успехе операции. |
| setValue(/*dojo.data.api.Item*/item, /*String*/attribute, /*Any*/value) | Устанавливает значение атрибута элемента, замещая существующее значение. Возвращает значение типа Boolean, свидетельствующее об успехе операции. |
| setValues(/*dojo.data.api.Item*/item, /*String*/attribute, /*Array*/values) | Устанавливает значения атрибута элемента, замещая существующие значения. Возвращает значение типа Boolean, свидетельствующее об успехе операции. |
| unsetAttribute(/*dojo.data.api.Item*/ item, /*String*/attribute) | Удаляет атрибут, удаляя все его значения. Возвращает значение типа Boolean, свидетельствующее об успехе операции. |
| save(/*Object*/args) | Сохраняет локальные изменения в памяти, а вывод передается функции обратного вызова, определенной в аргументе args, который может иметь следующие ключи:<br>onError(/*Object*/error)<br>Вызывается в случае появления ошибки. Аргумент error содержит информацию об ошибке.<br>onComplete()<br>Вызывается в случае успешного завершения операции. Как правило, не имеет параметров.<br>scope (Object)<br>Если определен, все функции обратного вызова должны запускаться в указанном контексте, в противном случае они должны запускаться в глобальном контексте.<br>Интерфейс Write предоставляет точку расширения, _saveCustom. Если ее переопределить, вы получите возможность самостоятельно отправлять данные на сервер. |
| revert() | Отменяет все локальные изменения. Возвращает значение типа Boolean, свидетельствующее об успехе операции. |
| isDirty(/*dojo.data.api.Item?*/item) | Возвращает значение типа Boolean, свидетельствующее о том, изменялся ли элемент с момента последней операции save. |

| Имя | Комментарий |
|---|---|
| | При вызове без аргумента возвращает значение типа Boolean, свидетельствующее о том, имеются ли вообще измененные элементы. |

## Интерфейс Notification

Интерфейс Notification, который приводится в табл. 9.4, основан на интерфейсе Read и дополняет интерфейс Write, обеспечивая унифицированный интерфейс для организации обработки таких событий, как create, update и delete. Интерфейс Notification, в частности, очень удобен для визуальных компонентов, обеспечивая возможность корректного отображения данных, которые могут изменяться или обновляться с использованием функций из интерфейсов Read и Write. (Яркими примерами таких компонентов могут служить dijit.Tree и dojox.grid.Grid.)

*Таблица 9.4. Интерфейс dojo.data.api.Notification*

| Имя | Комментарий |
|---|---|
| getFeatures() | Смотрите описание dojo.data.api.Read. Возвращает:<br>{<br>'dojo.data.api.Read' : true,<br>'dojo.data.api.Identity' : true<br>} |
| onSet(/*dojo.data.api.Item*/item, /*String*/attribute, /*Object\|Array*/old, /*Object\|Array*/new) | Вызывается всякий раз, когда изменяется элемент с помощью функций setValue, setValues или unsetAttribute, обеспечивая возможность контроля за изменениями в элементе или в атрибутах элемента. В аргументах old и new передаются старое и новое значение (или значения), соответственно. |
| onNew(/*dojo.data.api.Item*/item, /*Object?*/parentItem) | Вызывается всякий раз, когда создается новый элемент, где item — это вновь созданный элемент. Аргумент parentItem не передается, если вновь созданный элемент находится на корневом уровне, в противном случае в нем передается родительский элемент. (Обратите внимание, что при этом для родительского элемента parentItem не генерируется событие onSet, свидетельствующее об изменении его атрибута, потому что в данном случае предполагается, что parentItem |

*Таблица 9.4 (продолжение)*

| Имя | Комментарий |
|---|---|
| | предоставляет необходимый доступ к своей информации.) |
| `onDelete(/*dojo.data.api.Item*/item)` | Вызывается всякий раз, когда элемент удаляется из хранилища. В аргументе `item` передается удаленный элемент. |

# Основные реализации интерфейсов доступа к данным

В предыдущем разделе был дан обзор четырех основных интерфейсов доступа к данным, представленных на текущий момент. В этом разделе будут рассмотрены две реализации, входящие в состав библиотеки Core, – `ItemFileReadStore` и `ItemFileWriteStore`. Как будет показано ниже, механизм `ItemFileReadStore` реализует интерфейсы `Read` и `Identity`, а механизм `ItemFileWriteStore` реализует все четыре обсуждаемых интерфейса. Хорошее понимание этих двух механизмов позволит вам дополнять их по мере надобности или разработать свои собственные.

В книге не рассматривается подпроект `dojox.data`, который содержит широкий набор реализаций интерфейсов `dojo.data`, предназначенных для решения таких задач общего характера, как работа с распространенными хранилищами данных – CSV, Flickr, XML, OPML, Picasa и другими. Но так как все они поддерживают одни и те же интерфейсы, то знания, полученные здесь, помогут вам без затруднений в них разобраться.

## ItemFileReadStore

Хотя может оказаться, что в вашей конкретной ситуации лучше подошла бы собственная реализация `dojo.data.api.Read`, позволяющая повысить ее эффективность и лучше адаптировать к конкретным условиям, тем не менее в состав инструментария входит механизм `ItemFileReadStore`, который реализует интерфейсы `Read` и `Identity` и использует для представления данных гибкий формат JSON. В ситуациях, когда необходимо быстро организовать взаимодействие, достаточно в приложение, реализующее бизнес-логику, добавить совсем немного – вывод данных в формате, который понимает механизм `ItemFileReadStore`; *и всё*, после этого вы сможете использовать хранилище данных по своему усмотрению.

Еще одна особенность, о которой следует знать заранее, состоит в том, что реализация `ItemFileReadStore` загружает полный набор данных в память при первом же запросе, поэтому такие операции, как `isItemLoaded` и `loadItem`, являются достаточно бесполезными.

## Иерархии и ссылки в формате JSON

Механизм `ItemFileReadStore` не только реализует интерфейс `Read`, но и добавляет к нему некоторые свои особенности, включая свой формат данных, свой синтаксис запросов, свое средство десериализации значений атрибутов, свой способ идентификации элементов и т. д. Однако, прежде чем перейти к изучению этих особенностей, рассмотрим некоторые примеры данных, совместимых с `ItemFileReadStore`. Существуют две особенности, связанные с представлением данных: иерархии в формате JSON и ссылки в формате JSON. Иерархии в формате JSON состоят из вложенных ссылок, представляющих конкретные экземпляры элементов, а ссылки в формате JSON содержат данные, которые указывают на фактические элементы данных.

Чтобы понять различия между этими двумя разновидностями, сначала взгляните на два следующих примера. Первый пример – иерархия в формате JSON:

```
{
    identifier : id,
    items : [
        {
            id : 1, name : "foo", children : [
                {id : 2, name : "bar"},
                {id : 3, name : "baz"}
            ]
        }
        /* еще элементы... */
    ]
}
```

А теперь – пример ссылок в формате JSON:

```
{
    identifier : id,
    items : [
        {
            id : 1, name : "foo", children : [
                {_reference: 2},
                {_reference: 3}
            ]
        },
        {id : 2, name : "bar"},
```

```
            {id : 3, name : "baz"}
            /* еще элементы... */
        ]
    }
```

В обоих случаях присутствует элемент foo, имеющий два дочерних элемента, но в примере с иерархией дочерние элементы явно вложены в родительский элемент, тогда как в примере со ссылками используются вложенные указатели на элементы. Главное преимущество формата JSON со ссылками заключается в его гибкости – он позволяет объявлять элементы дочерними более чем в одном родительском элементе, а также обеспечивает возможность объявлять все элементы на корневом уровне. Обе эти возможности очень удобны и широко используются во многих приложениях.

Диджит Tree, который будет представлен в главе 15, является ярким примером, демонстрирующим мощь и гибкость (а также некоторые недостатки) формата представления данных JSON со ссылками.

## Анализ ItemFileReadStore

Чтобы поближе познакомиться с механизмом ItemFileReadStore, рассмотрим коллекцию иерархических данных в формате JSON, приведенную в примере 9.1, где каждый элемент определяется идентификатором name. Обратите внимание, что только ключи identifier, label и items присутствуют на самом верхнем уровне в хранилище данных.

*Пример 9.1. Пример набора данных с описаниями сортов кофе*

```
{
    identifier : "name",
    label : "name",
    items : [
       {name : "Light Cinnamon", description : "Very light brown, dry, tastes
like toasted grain with distinct sour tones, baked, bready"},
        {name : "Cinnamon", description : "Light brown and dry, still toasted
grain with distinct sour acidy tones"},
       {name : "New England", description : "Moderate light brown, still sour
but not bready, the norm for cheap Eastern U.S. coffee"},
        {name : "American or Light", description : "Medium light brown, the
traditional norm for the Eastern U.S .”},
        {name : "City, or Medium", description : "Medium brown, the norm for
most of the Western US, good to taste varietal character of a bean.”},
        {name : "Full City", description : "Medium dark brown may have some
slight oily drops, good for varietal character with a little bittersweet.”},
        {name : "Light French", description : "Moderate dark brown with oily
drops, light surface oil, more bittersweet, caramelly flavor, acidity muted.”
},
       {name : "French", description : "Dark brown oily, shiny with oil, also
popular for espresso; burned undertones, acidity diminished"},
```

```
        {name : "Dark French", description : "Very dark brown very shiny,
burned tones become more distinct, acidity almost gone."},
        {name : "Spanish", description : "Very dark brown, nearly black and
very shiny, charcoal tones dominate, flat."}
    ]
}
```

Предположим, что файл с этими данными хранится на диске под именем *coffee.json*. Страница, представленная в примере 9.2, загружает этот файл и обеспечивает доступ к данным через глобальную переменную JavaScript – `coffeeStore`.

*Пример 9.2. Программная загрузка данных механизмом ItemFileReadStore*

```
<html>
    <head>
        <title>Fun with ItemFileReadStore!</title>
        <script
            type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js">
        </script>
        <script type="text/javascript">
            dojo.require("dojo.data.ItemFileReadStore");

            dojo.addOnLoad(function() {
                coffeeStore = new dojo.data.ItemFileReadStore({
                    url:"coffee.json"
                });
            });
        </script>
    </head>
    <body>
    </body>
</html>
```

Парсер будет представлен только в главе 11, но из-за его широкого использования есть смысл явно упомянуть, что тот же самый эффект можно было получить с помощью разметки, представленной в примере 9.3.

*Пример 9.3. Загрузка данных механизмом ItemFileReadStore в разметке*

```
<html>
    <head>
        <title>Fun with ItemFileReadStore!</title>
        <script
            type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js"
            djConfig="parseOnLoad:true">
        </script>
        <script type="text/javascript">
            dojo.require("dojo.parser");
            dojo.require("dojo.data.ItemFileReadStore");
```

```
        </script>
    </head>
    <body>
        <div dojoType="dojo.data.ItemFileReadStore" url="./coffee.json"
             jsId="coffeeStore"></div>
    </body>
</html>
```

Независимо от того, как будут объявлены данные, прикладной интерфейс работает одинаково в обоих случаях. Определенно стоит потратить несколько минут, чтобы поэкспериментировать с существующими данными в консоли Firebug. Далее в этом разделе приводится серия фрагментов программного кода, команд и возвращаемых значений для большинства функций интерфейсов `Read` и `Identity`, которые вы можете просматривать и использовать для более быстрого изучения механизма `ItemFileReadStore`.

Помимо параметра `url`, указывающего на файл с данными, имеется также возможность передать механизму `ItemFileReadStore` параметр `data` со ссылкой на уже существующий в памяти объект JavaScript.

### Извлечение элемента по идентификатору

Извлечение элементов данных с помощью механизма `ItemFileReadStore` может быть произведено двумя способами, которые, впрочем, очень похожи между собой. Чтобы извлечь элемент по идентификатору, следует использовать функцию `fetchItemByIdentity`, принадлежащую интерфейсу `Identity`. Эта функция принимает набор именованных атрибутов, включая идентификатор требуемого элемента и ссылку на функцию, которая должна вызываться в случае ошибки. Например, запрос информации об испанском кофе можно было бы реализовать, как показано в примере 9.4.

*Пример 9.4. Извлечение элемента по идентификатору и его исследование*

```
var spanishCoffeeItem;
coffeeStore.fetchItemByIdentity({
    identity: "Spanish",
    onItem : function(item, request) {
        //получить элемент ... или что-нибудь сделать с ним
        spanishCoffeeItem = item;
    },
    onError : function(item, request) {
        /* Обработать ошибку... */
    }
});

// теперь можно сделать что-нибудь с переменной spanishCoffeeItem ...

//например, получить описание кофе...
coffeeStore.getValue(spanishCoffeeItem, "description"); //Very dark brown...
```

```
//или его название...
coffeeStore.getValue(spanishCoffeeItem, "name"); // Spanish

//в данном случае идентификатор совпадает с меткой...
coffeeStore.getLabel(spanishCoffeeItem); // Spanish
//если у вас имеется элемент, но вы не знаете, как он идентифицируется...
coffeeStore.getIdentity(spanishCoffeeItem); //Spanish
```

Часто начинающие программисты совершают ошибку, путая идентификатор элемента с самим элементом. Такую ошибку отыскать очень непросто, потому что программный код «выглядит правильным». Такая попытка извлечь данные с описанием испанского кофе `var item = coffeeStore.fetchItemByIdentity("Spanish")` выглядит правильной, но при более близком изучении функции интерфейса можно заметить, по меньшей мере, две ошибки: функция не возвращает элемент и функция принимает коллекцию именованных аргументов, а не значение атрибута `identity`.

### Извлечение элемента по произвольному критерию

Если необходимо извлечь элемент по атрибуту, не используемому для идентификации, вместо функции `fetchItemByIdentity` можно использовать более универсальную функцию `fetch`, как показано ниже:

```
coffeeStore.fetch({
    query: {name : "Spanish"},
    onItem : function(item, request){console.log(item);}
});
```

Однако, помимо полностью квалифицированных значений атрибутов, функция `fetch` также принимает небольшие, но надежные коллекции критериев, допуская использовать несложные регулярные выражения. Например, чтобы отыскать кофе, в описании которого присутствует слово «dark», без учета регистра символов, можно было бы выполнить запрос, который демонстрируется в примере 9.5.

*Пример 9.5. Извлечение элемента по произвольному критерию*

```
coffeeStore.fetch({
    query: {description : "*dark*"},
    queryOptions:{ignoreCase : true},
    onItem : function(item, request) {
        console.log(coffeeStore.getValue(item, "name"));
    }
    /* здесь можно подключить другие функции обратного вызова... */
});
```

Всегда используйте функцию `getValue` для получения значений атрибутов. Не пытайтесь обращаться к ним напрямую, потому что реализация механизма хранения данных может не позволять этого. Например, не следует обращаться к элементу в обра-

ботчике события `onItem` следующим образом: `onItem: function(item, request) { console.log(item.name); }`. Огромное преимущество этой абстракции состоит в том, что она обеспечивает возможность кэширования и выполнения других оптимизаций, повышающих эффективность механизма доступа к данным.

Если вы собираетесь создавать собственную реализацию хранилища данных, вам будет полезно знать, что модуль `dojo.data.util.filter` обеспечивает точно такую же возможность организации поиска совпадений по образу и подобию регулярных выражений, которая используется в функции `fetch` механизма `ItemFileReadStore`, а модуль `dojo.data.util.simpleFetch` предоставляет логику для восьми аргументов: `onBegin`, `onItem`, `onComplete`, `onError`, `start`, `count`, `sort` и `scope`.

## Запрос дочерних элементов

Приведенный пример данных с описаниями сортов кофе является достаточно примитивным, так как представляет собой плоский список элементов. Пример 9.6 несколько усложняет его, добавляя несколько дополнительных элементов с дочерними записями, чтобы воспроизвести вложенную структуру. Механизм `ItemFileReadStore` явно использует атрибут `children` для хранения списка дочерних элементов, а мы будем использовать разновидность формата JSON со ссылками, чтобы обеспечить группировку сортов кофе по степени обжарки. Обратите внимание, что для иллюстрации возможности использования ссылок кофе `Light French` был преднамеренно отнесен и к категории `Medium Roasts`, и к категории `Dark Roasts`. Так как каждый элемент должен иметь уникальный идентификатор, было бы невозможно включить один и тот же элемент в разные родительские элементы другим способом.

В оставшейся части главы используется пример данных с двухуровневой организацией, тем не менее нет никаких причин, которые препятствовали бы использовать набор данных с произвольной глубиной вложенности.

*Пример 9.6. Обновленный пример набора данных с описаниями сортов кофе, имеющий иерархическую структуру*

```
{
    identifier : "name",
    items : [
        {
            name : "Light Roasts",
            description : "A number of delicious light roasts",
            children : [
                {_reference: "Light Cinnamon"},
                {_reference: "Cinnamon"},
                {_reference: "New England"}
            ]
        },
```

```
        {
            name : "Medium Roasts",
            description : "A number of delicious medium roasts",
            children : [
                {_reference: "American or Light"},
                {_reference: "City, or Medium"},
                {_reference: "Full City"},
                {_reference: "Light French"}
            ]
        },
        {
            name : "Dark Roasts",
            description : "A number of delicious dark roasts",
            children : [
                {_reference: "Light French"},
                {_reference: "French"},
                {_reference: "Dark French"},
                {_reference: "Spanish"}
            ]
        },
        {name : "Light Cinnamon", description : "Very light brown, dry ,
tastes like toasted grain with distinct sour tones, baked, bready"},
        ...
    ]
}
```

Достаточно часто встречаются ситуации, когда необходимо запросить дочерние элементы. В данном случае это означает необходимость отыскать названия кофе, относящегося к той или иной категории по степени обжарки. Решение этой задачи иллюстрируется в примере 9.7 для элемента `Dark Roasts`.

*Пример 9.7. Извлечение элемента и обход его дочерних элементов*

```
coffeeStore.fetch({
    query: {name : "Dark Roasts"},
    onItem : function(item, request) {
        dojo.forEach(coffeeStore.getValues(item, "children"),
            function(childItem) {
                console.log(coffeeStore.getValue(childItem, "name"));
        });
    }
});
```

В этом примере выполняется простой запрос на получение родительского элемента `Dark Roasts`, и, как только мы получаем его, с помощью функции `getValues` извлекается атрибут `children` со множеством значений, после чего с помощью функции `dojo.forEach` производится обход всех дочерних элементов – не забываем при этом использовать функцию `getValue`, чтобы получить доступ к значению атрибута элемента.

Обратите внимание, что конструкция `{_reference: someIdentifier}` является лишь особенностью реализации. Вам никогда не потребуется строить запросы на основе атрибута `_reference`, потому что в действительности нет такой вещи, как атрибут `_reference`, – это всего лишь стандартизованный способ вести учет. Насколько это касается прикладного программиста, все, что находится в хранилище `dojo.data`, следует рассматривать как старый добрый элемент.

Как вы уже имели возможность убедиться, механизм `ItemFileReadStore` обладает весьма широкими и гибкими возможностями, что делает его вполне подходящим способом хранения данных для самых разных ситуаций, особенно при разработке прототипов приложений и когда необходимо быстро получить результат. Благодаря простой спецификации совсем несложно будет написать на стороне сервера процедуру, отправляющую данные, которые веб-клиент, использующий `dojo.data`, сможет обработать. Однако не забывайте при этом, что вы всегда сможете дополнить существующий механизм своими расширениями или создать собственную реализацию.

## ItemFileWriteStore

Нет никаких сомнений, что хорошая абстракция способна значительно уменьшить объем рутинных действий, когда появляется необходимость обрабатывать и отображать данные, получаемые от сервера. Однако при этом вы не избавлены от необходимости отправлять данные обратно на сервер при изменении в данных, и тут-то вам на помощь придет `ItemFileWriteStore`. Как `ItemFileReadStore` обеспечивает отличную абстракцию чтения данных, точно так же `ItemFileWriteStore` обеспечивает подобную абстракцию таких операций записи, как создание новых элементов, удаление элементов и изменение элементов. В терминах интерфейсов `dojo.data` механизм `ItemFileWriteStore` реализует все четыре интерфейса: `Read`, `Identity`, `Write` и `Notification`.

Чтобы познакомиться с механизмом `ItemFileWriteStore`, мы рассмотрим его особенности таким же способом, каким исследовали особенности механизма `ItemFileReadStore`, – используя тот же самый набор данных *coffee.json*. Как вы увидите дальше, здесь нет никаких сюрпризов – имена функций интерфейсов в значительной степени говорят сами за себя.

### Изменение существующего элемента

Вам часто придется использовать функцию `setValue`, приведенную в примере 9.8, для изменения значения атрибута элемента; ей передаются элемент, атрибут, который требуется изменить, и новое значение атрибута. Если элемент не имеет атрибута с указанным именем, он автоматически будет добавлен.

*Пример 9.8. Запись значения атрибута элемента*

```
//Извлечь элемент обычным способом...
var spanishCoffeeItem
```

```
coffeeStore.fetchItemByIdentity({
    identity: "Spanish",
    onItem : function(item, request) {
        spanishCoffeeItem = item;
    }
});

        coffeeStore.setValue(spanishCoffeeItem, "foo", "bar");

        coffeeStore.getValue(spanishCoffeeItem, "foo"); //bar

        //Точно так же можно изменить любой другой атрибут, за исключением
        //используемых для идентификации
        coffeeStore.setValue(spanishCoffeeItem, "description",
                             "El Matador...?!?");
```

Как и в других схемах хранения данных, обычно не имеет никакого смысла изменять идентификацию элемента, поскольку понятие идентификации обозначает неизменяемую характеристику. Механизм `ItemFileWriteStore` не поддерживает эту операцию, и не рекомендуется стремиться реализовать ее в своих собственных механизмах.

Следует отметить одну особенность: запись пустой строки в атрибут не то же самое, что удаление атрибута. Это особенно важно понять, если вы предполагаете использовать функцию `hasAttribute` из интерфейса `Write` для проверки наличия атрибута. Пример 9.9 иллюстрирует эту особенность.

*Пример 9.9. Установка значения атрибута и удаление*

```
coffeeStore.hasAttribute(spanishCoffeeItem, "foo");   //true

coffeeStore.setValue(spanishCoffeeItem, "foo", "");   //foo=""

coffeeStore.hasAttribute(spanishCoffeeItem, "foo");   //true

coffeeStore.unsetAttribute(spanishCoffeeItem, "foo"); //удалит атрибут

coffeeStore.hasAttribute(spanishCoffeeItem, "foo");   //false
```

Приведенные ранее примеры этого раздела продемонстрировали, как можно изменять существующие элементы, но эти изменения нельзя считать окончательными, потому что явно не была выполнена операция сохранения. Внутри механизм `ItemFileWriteStore` отслеживает изменения и обслуживает коллекцию *грязных* элементов – элементов, которые были изменены, но еще не были сохранены. Например, после изменения элемента `spanishCoffeeItem` можно было бы с помощью функции `isDirty` выяснить, как показано в примере 9.10, что элемент изменился, но не был сохранен. Однако, после выполнения операции сохранения элемент перестает быть «грязным». Пока что под сохранением мы понимаем не что иное, как обновление копии в памяти, а об операции сохранения на сервере мы поговорим чуть позже.

*Пример 9.10. Определить, был ли изменен элемент*

```
/* Выполнить первое изменение spanishCoffeeItem... */
coffeeStore.isDirty(spanishCoffeeItem); //true
coffeeStore.save();                     //обновит копию в памяти
coffeeStore.isDirty(spanishCoffeeItem); //false
```

Хотя прямо сейчас это и неочевидно, но преимущество такого требования явно вызывать операцию `save` для подтверждения изменений состоит в том, что оно обеспечивает возможность отмены изменений, если одна из последующих операций, выполняемых в рамках той же транзакции, приводит к появлению ошибки или возникают другие обстоятельства, которые требуют отмены изменений. В реляционных базах данных это часто называется *откат* (*rollback*). Пример 9.11 демонстрирует выполнение отмены изменений в хранилище данных `dojo.data` и подчеркивает очень тонкую, но очень важную особенность, связанную с локальными переменными, хранящими ссылки на элементы.

*Пример 9.11. Отмена изменений механизмом ItemFileWriteStore*

```
var spanishCoffeeItem
coffeeStore.fetchItemByIdentity({
    identity: "Spanish",
    onItem : function(item, request) {
        spanishCoffeeItem = item;
    }
});
coffeeStore.getValue(spanishCoffeeItem, "description"); //Very dark...
coffeeStore.setValue(spanishCoffeeItem, "description",
                     "El Matador...?!?");

//Сейчас как в spanishCoffeeItem, так и в хранилище находится
//измененное описание. Попробуем извлечь этот же элемент еще раз,
//чтобы проверить это утверждение...
coffeeStore.fetchItemByIdentity({
    identity: "Spanish",
    onItem : function(item, request) {
        coffeeStore.getValue(item, "description"); //El Matador...?!?
        coffeeStore.isDirty(item); //true
)};
coffeeStore.revert(); //отменить изменения.

// В результате выполнения revert() локальная переменная
// spanishCoffeeItem перестала быть элементом данных
coffeeStore.isItem(spanishCoffeeItem); //false

//Извлечь элемент еще раз, чтобы продемонстрировать его
//содержимое...
coffeeStore.fetchItemByIdentity({
    identity: "Spanish",
    onItem : function(item, request) {
        coffeeStore.isDirty(item); //false
```

```
            coffeeStore.getValue(item, "description"); //Very dark...
        }
    });
```

Теоретически вполне возможно реализовать собственный механизм хранилища данных, который предотвращал бы устаревание локальных ссылок на элементы, – путем хитроумной организации взаимодействий с помощью функции `dojo.connect` или использования взаимодействий по подписке. Однако механизм `ItemFileWriteStore` не погружается в такие глубины, и поэтому вы должны использовать функцию `isItem`, когда возникают подозрения в том, что ссылка на элемент могла устареть.

### Создание и удаление элементов

Как только вы разберетесь со всеми нюансами из предыдущего раздела, связанными с операцией изменения существующих элементов, у вас не будет никаких проблем с пониманием того, как выполняется добавление или удаление элементов из хранилища. Здесь в той же мере действуют принципы, связанные с операциями сохранения и отмены. Для начала давайте попробуем добавить и удалить элемент верхнего уровня в существующем хранилище, как показано в примере 9.12. Добавление элемента связано с необходимостью предоставить объект JSON в том же виде, в каком он был бы включен в оригинальный набор данных на стороне сервера.

*Пример 9.12. Добавление и удаление элемента средствами механизма ItemFileWriteStore*

```
var newItem = coffeeStore.newItem({
    name : "Really Dark",
    description : "Left brewing in the pot all day...extra octane."
});

coffeeStore.isItem(newItem); //true
coffeeStore.isDirty(newItem); //true

/* Запросить элемент, сохранить изменения, отменить изменения, и т. д. */

//Или удалить элемент...
coffeeStore.deleteItem(newItem);
coffeeStore.isItem(newItem); //false
```

Несмотря на то что добавление и удаление элементов верхнего уровня выглядит достаточно тривиально, тем не менее для добавления нового элемента приходится приложить чуть больше усилий, если элемент является дочерним и необходимо добавить ссылку на него где-то в другом месте. Пример 9.13. демонстрирует, как это выполняется. Суть состоит в том, что необходимо сначала создать элемент верхнего уровня, затем получить ссылку на коллекцию дочерних элементов, к которой следует присоединить данный элемент и, наконец, добавить его в эту коллекцию.

*Пример 9.13. Добавление дочернего элемента в набор данных, имеющий формат JSON со ссылками*

```
// Создать новый элемент
var newItem = coffeeStore.newItem({
    name : "Really Dark",
    description : "Left brewing in the pot all day...extra octane."
});

var darkRoasts;
//Получить ссылку на родительский элемент с дочерними элементами
coffeeStore.fetchItemByIdentity({
    identity : "Dark Roasts",
    onItem : function(item, request) {
        darkRoasts = item;
    }
});

//С помощью getValues получить ссылку на коллекцию дочерних элементов
var darkRoastChildren = coffeeStore.getValues(darkRoasts, "children");

//И добавить новый элемент в эту коллекцию с помощью setValues
coffeeStore.setValues(darkRoasts, "children",
                      darkRoastChildren.concat(newItem)
);

//Теперь можно выполнить обход дочерних элементов
//чтобы убедиться в том, что новый элемент был добавлен в коллекцию...
dojo.forEach(darkRoastChildren, function(x) {
    console.log(coffeeStore.getValue(x, "name"));
});
```

**Не забывайте, что для извлечения атрибутов, которые могут иметь несколько значений, следует использовать функцию `getValues`, а не `getValue`.**

Удаление выполняется именно так, как и следовало ожидать. Удаление элемента верхнего уровня приводит к удалению самого элемента из набора данных, но оставляет его дочерние элементы нетронутыми, как показано в примере 9.14.

*Пример 9.14. Удаление элемента верхнего уровня средствами механизма ItemFileWriteStore*

```
var darkRoasts;
coffeeStore.fetchItemByIdentity({
    identity : "Dark Roasts",
    onItem : function(item, request) {
        darkRoasts = item;
    }
});
```

```
coffeeStore.deleteItem(darkRoasts);
coffeeStore.fetch({
    query : {name : "*"},
    onItem : function(item, request) {
        //Вы не увидите элемент "Dark Roasts" в этих результатах...
        console.log(coffeeStore.getValue(item, "name"));
    },
    onComplete : function(items, request) {
        /* Выполнить операцию сохранения, или отмены, или... */
    }
});
```

Очевидно, что можно было бы сначала получить коллекцию всех дочерних элементов, удалить их, а затем удалить сам элемент верхнего уровня.

### Собственные реализации операции сохранения

Вы уже наверняка подумали, что сохранение изменений в памяти – это, конечно, замечательно, но как быть, если необходимо вернуть данные серверу? Оказывается, механизм ItemFileWriteStore предоставляет точку подключения расширения `_saveCustom`, которую можно использовать для вызова собственной процедуры всякий раз, когда вызывается метод `save`, то есть, помимо сохранения изменений в локальной копии и сброса признака «*грязный*» (*dirty*) у всех измененных элементов, можно также выполнять синхронизацию с сервером или выполнять еще что-нибудь. В вашем распоряжении остаются все те же функции интерфейса, которые вы использовали все время, но вообще, операция «полного сохранения», вероятно, могла бы включать в себя обход всего набора данных, сериализацию данных в некоторый формат – скорее всего, с помощью функции `dojo.toJson` – и передачу их на сервер. Согласно определению интерфейса `Write`, имеется возможность указать в именованных аргументах необязательные функции обратного вызова `onComplete` и `onError`, которые будут вызываться в случае успеха или ошибки. Существует дополнительная возможность определить аргумент `scope`, задающий контекст выполнения для всех этих функций обратного вызова. Однако, эти аргументы передаются функции `save`, но они не передаются расширению `_saveCustom`.

В примере 9.15 демонстрируется, как можно реализовать обработчик `_saveCustom`, выполняющий передачу данных на сервер при вызове функции `save()`. Как видно из примера, все достаточно предсказуемо.

*Пример 9.15. Собственный обработчик операции сохранения для механизма ItemFileWriteStore*

```
<html>
    <head>
        <title>Fun with ItemFileWriteStore!</title>
        <script
            type="text/javascript"
```

```
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js">
        </script>
        <script type="text/javascript">
            dojo.require("dojo.data.ItemFileWriteStore");

            dojo.addOnLoad(function( ) {
                coffeeStore = new dojo.data.ItemFileWriteStore(
                    {url:"coffee.json"});
                coffeeStore._saveCustom = function( ) {
                    /* Здесь выполняются действия, необходимые для
                       передачи данных на сервер. Это расширение вызывается
                       всякий раз, когда происходит обращение к функции
                       save(). */
                }
            });
        </script>
    </head>
    <body>
    </body>
</html>
```

Оказывается, точка расширения `_saveCustom` используется не так часто, как можно было бы подумать, так как ее использование связано с передачей на сервер всего набора данных, что обычно не является необходимым, если работа не начинается с пустого набора данных, и нет нкужды выполнять начальную отправку всех данных. Чаще всего, особенно в случае очень больших наборов данных, предпочтительнее использовать интерфейс `Notification`, который будет представлен в следующем разделе, для передачи изменений небольшими порциями.

## Обработка уведомлений

В завершение этого раздела и всей главы мы коротко рассмотрим реализацию интерфейса `Notification` в механизме `ItemFileWriteStore`, потому что он очень удобен в случаях, когда необходимо реализовать обработку различных уведомлений `onNew`, `onDelete` или `onSet`, связанных с созданием новых элементов, удалением существующих элементов или изменением элементов, соответственно.

Поскольку к настоящему времени вы уже наверняка способны, прочитав, понять, как используются интерфейсы в различных реализациях, пример, который добавляет, изменяет и удаляет элемент из набора данных, наверняка покажется вам очевидным. Но на всякий случай замечу, что пример 9.16 является дополненной версией примера 9.13.

*Пример 9.16. Использование интерфейса Notification для перехвата событий средствами ItemFileWriteStore*

```
/* Начало блока обработчиков извещений */
coffeeStore.onNew = function(item, parentItem) {
    var itemName = coffeeStore.getValue(item, "name");
```

```
    console.log("Just added", itemName, "which had parent", parentItem);
}
coffeeStore.onSet = function(item, attr, oldValue, newValue) {
    var itemName = coffeeStore.getValue(item, "name");
    console.log("Just modified the ", attr, "attribute for", itemName);
    /* Поскольку атрибут children может содержать сразу несколько значений,
       oldValue и newValue являются массивами, которые можно обойти
       и выполнить какие-либо действия, хотя чаще всего достаточно будет
       просто отправить newValue на сервер, чтобы зафиксировать изменения */
}
coffeeStore.onDelete = function(item) {
    // coffeeStore.isItem(item) вернет false, поэтому не пытайтесь
    // обращаться к элементу
    console.log("Just deleted", item);
}
/* Конец блока обработчиков извещений */
/* Программный код, использующий обработчики извещение */
//Добавление элемента верхнего уровня вызывает появление извещения
var newItem = coffeeStore.newItem({
    name : "Really Dark",
    description : "Left brewing in the pot all day...extra octane."
});
var darkRoasts ;
coffeeStore.fetchItemByIdentity({
    identity : "Dark Roasts",
    onItem : function(item, request) {
        darkRoasts = item;
    }
});
var darkRoastChildren = coffeeStore.getValues(darkRoasts, "children");
//Изменение элемента вызывает появление извещения
coffeeStore.setValues(darkRoasts,
                      "children",darkRoastChildren.concat(newItem))
);
//Удаление элемента вызывает сразу два извещения
coffeeStore.deleteItem(newItem)
```

В результате запуска этого примера вы должны увидеть вывод, как показано ниже:

```
Just added Really Dark, which had parent null
Just modified the children attribute for Dark Roasts
Just modified the children attribute for Dark Roasts
Just deleted Object _0=13 name=[1] _RI=true description=[1]
```

Другими словами, вы получаете ожидаемое уведомление, когда создаете элемент верхнего уровня; когда изменяете значение атрибута `children` другого элемента, добавляя вновь созданный элемент в коллекцию дочерних элементов родителя; еще одно уведомление получаете при удалении ссылки на дочерний элемент из коллекции и последнее уведомление – когда удаляется сам элемент.

В примере 9.16 имеется одна особенность, о которой стоит упомянуть. Она состоит в том, что обработчик `onDelete` получает ссылку на уже удаленный элемент, поэтому ее полезность несколько ограничена, т. к. ее уже нельзя использовать в стандартных операциях реализации.

## Сериализация и десериализация собственных типов данных

Хотя до сих пор об этом ничего не говорилось, тем не менее пришла пора узнать о существовании одной дополнительной особенности, присутствующей в `ItemFileReadStore` и `ItemFileWriteStore`, которая позволяет упаковывать и распаковывать данные произвольного типа. Причина, по которой может потребоваться выполнять *отображение типов*, часто обусловлена тем, что приходится иметь дело с атрибутами, значения которых не являются данными простых типов, литерами объектов или массивами. В таких случаях у вас есть возможность либо вручную собирать атрибуты, что привносит дополнительную сложность в логику приложения, либо запрятать логику сериализации в другом месте.

### Неявное отображение типов

*Неявное отображение типов* в `ItemFileReadStore` происходит автоматически, если в данных присутствуют два специальных атрибута `_type` и `_value`. Атрибут `_type` идентифицирует функцию-конструктор, которой должно передаваться значение атрибута `_value`. Объекты JavaScript типа `Date` являются наиболее типичным примером данных, при использовании которых можно ощутить преимущества отображения типов. Типичный элемент данных из нашего существующего набора, в котором используется значение даты, мог бы выглядеть, как показано в примере 9.17.

*Пример 9.17. Использование возможности отображения типа для десериализации значения*

```
...
{
    name : "Light Cinnamon",
    description : "Very light brown, dry , tastes like toasted grain with
distinct sour tones, baked, bready"
    lastBrewed : {
        '_type' : "Date",
        '_value':"2008-06-15T00:00:00Z"}
    }
}
...
```

Этот пример выглядит достаточно просто, но только при предположении, что функция-конструктор `Date` уже определена! После десериали-

зации данных любые значения атрибута lastBrewed будут являться полноценными объектами типа Date, а не обычными строками:

```
var coffeeItem;
coffeeStore.fetchItemByIdentity({
    identity : "Light Cinnamon",
    onItem : function(item, request) {
        coffeeItem = item;
    }
});
coffeeStore.getValue(coffeeItem, "lastBrewed"); //Настоящий объект Date
```

## Отображение собственных типов

При необходимости вы можете определить объект JavaScript, где задать функцию с именем deserialize и свойством type, которые могли бы использоваться для конструирования значения. Для механизма ItemFileWriteStore также можно определить функцию serialize. Ниже в примере 9.18 демонстрируется пример объекта, представляющий собой отображение объектов типа Date, который можно указать при конструировании набора данных для механизма ItemFileWriteStore.

*Пример 9.18. Использование объекта отображения типа при создании набора данных для ItemFileWriteStore*

```
dojo.require('dojo.date');
dojo.addOnLoad(function( ) {
    var map = {
        "Date": {
            type: Date,
            deserialize: function(value){
                return dojo.date.stamp.fromISOString(value);
            },
            serialize: function(object){
                return dojo.date.stamp.toISOString(object);
            }
        }
    };

    coffeeStore = new dojo.data.ItemFileReadStore({
        url:"coffee.json",
        typeMap : map
    });
});
```

Мы преднамеренно не рассматривали в этой главе подпроекты dojox.data, но было бы несправедливо не упомянуть о проекте dojox.data.QueryReadStore, который является классическим средством взаимодействия с очень объемными источниками данных, расположенными на стороне сервера. По адресу *http://www.oreillynet.com/onlamp/blog/2008/04/dojo_goodness_part_6_a_million.html* вы найдете лаконичный пример использования

этого механизма с собственной процедурой, работающей на стороне сервера. В этом примере в частности демонстрируется, как *эффективнее* организовать работу с одним миллионом записей в знаменитом виджете DojoX Grid.

# В заключение

После прочтения этой главы вы должны:

- Иметь представление об интерфейсах `dojo.data` и понимать назначение каждого из них
- Понимать, что интерфейсы `Read`, `Identity`, `Write` и `Notification` являются абстракциями и что возможны любые их реализации
- Знать, что среди подпроектов `dojox.data` можно найти несколько действительно полезных реализаций, которые позволят вам сэкономить время при решении таких типичных задач, как обеспечение доступа к данным в формате CSV, Flickr и т. д.
- Знать, что инструментальный набор предоставляет `ItemFileReadStore` и `ItemFileWriteStore` — универсальные и мощные реализации интерфейсов `dojo.data`, которые можно расширять или использовать как основу для собственных реализаций
- Понимать ценность возможности отображения типов, которая позволит сэкономить время на сериализации и десериализации данных вручную

В следующей главе мы рассмотрим механизмы имитации классов и наследования.

# 10

# Имитация классов и наследование

Хотя язык JavaScript вводит некое подобие классов с помощью объектов типа Function и на основе прототипов реализует механизм наследования, тем не менее это несколько отличается от того, с чем вы могли встречаться раньше, если у вас имеется опыт программирования на таких языках, как Java или C++. Разработчики Dojo проделали огромный труд, построив конструкции, имитирующие классы и механизм наследования классов, поверх конструкций языка JavaScript. В этой главе обсуждается функция `dojo.declare` – механизм инструментария, предназначенный для создания классов и организации наследования, и тем самым открывается путь к пониманию инфраструктуры библиотеки Dijit, которая будет рассматриваться во второй части книги.

## JavaScript – это не Java

Прежде чем перейти к обсуждению имитации в Dojo классов и иерархий наследования, вы должны осознать, что, во-первых, JavaScript – это не Java, а во-вторых, инструментарий Dojo не пытается побороть стиль языка, за кулисами переделать отдельные части языка JavaScript и заставить его заниматься несвойственными ему вещами, – и здорово, что это так! JavaScript – это динамический, слабо типизированный язык программирования, тогда как Java – гораздо более строго типизированный язык, с настоящими классами и иерархиями наследования на основе классов, которые определяются на этапе компиляции. В JavaScript имеется механизм наследования, основанный на прототипах, который легко интерпретируется и может использоваться для моделирования классов.

Используя язык JavaScript по прямому его предназначению и усиливая некоторые языковые конструкции максимально бережным способом, Dojo позволяет пользоваться преимуществами расширения языка

и избежать необходимости сопровождать массивные пласты шаблонного программного кода. В конце концов, инструментарий предоставляет в ваше распоряжение четко организованную, гибкую реализацию, которая поможет вам следовать философии современной технологической эры: «выпускай раньше, обновляй чаще».

За исключением диджитов форм, которые будут представлены в главе 13, вы не увидите в инструментарии Dojo объектно-ориентированной архитектуры, потому что это не его стиль. Стиль Dojo состоит в том, чтобы использовать наследование на основе прототипов и конструкции из библиотеки Base, такие как `mixin` и `extend`, которые в свою очередь используют сильные стороны JavaScript. В то же самое время инструментарий Dojo стремится быть прагматичным, и для реализации некоторых его частей действительно были использованы принципы объектно-ориентированной разработки, без которых реализация выглядела бы очень неуклюже. На самом деле, главная причина, почему эта глава завершает первую часть книги, состоит в том, что во второй части будет представлена библиотека Dijit, которая *является той самой частью*, которую можно отнести к объектно-ориентированной архитектуре.

При чтении этой главы помните о сказанном выше, потому что если вы обладаете большим опытом программирования на объектно-ориентированных языках, таких как Java или C++, у вас постоянно будет возникать искушение перенести объектно-ориентированные парадигмы этих языков в Dojo и попытаться реализовать все подряд с использова-

### Не надо бороться со стилем!

JavaScript – это интерпретирующий язык программирования с динамической типизацией, обладающий множеством недооцененных возможностей, к которым нужно привыкнуть, если раньше вы использовали языки программирования со статической типизацией, такие как C++ или Java. Если у вас начинают возникать сложности с решением какой-либо проблемы или вы слишком много времени потратили на исправление ошибки, приостановитесь на минуту, убедитесь, что понимаете тонкости того, чего пытаетесь добиться, и спросите себя, как лучше использовать встроенные особенности языка.

Если вы противоречите стилю настолько, что пытаетесь буквально построчно переложить реализацию алгоритма из такого языка, как Java, на язык JavaScript, то в конечном счете вы добьетесь своего, но заплатите за это временем, простотой сопровождения, эффективностью и нервами. Поэтому не стремитесь бороться со стилем программирования – стремитесь сначала понять его, а потом использовать для достижения поставленной цели.

нием механизма наследования, что оказывается не совсем естественным с точки зрения стиля программирования или производительности. В этой главе действительно демонстрируется возможность использовать Dojo для имитации глубоко вложенных иерархий объектов, тем не менее предполагается, что вы будете использовать эту возможность с благоразумной осторожностью.

# Одна проблема, множество решений

Для тех, кто может освободиться от парадигмы наследования, в этом разделе будут продемонстрированы разные способы достижения похожих результатов при решении простой задачи: моделирование объекта окружности. Кроме того что этот раздел является хорошим примером демонстрации нестандартного мышления, здесь также используются некоторые инструменты языка, о которых вы узнали в главе 2.

## Наследование в JavaScript

Программисты, использующие JavaScript, имитируют классы с помощью объектов Function – иногда неправильно используя возможности языка, иногда эффективно решая определенные проблемы. По своей природе объект Function в языке JavaScript представляет собой тот самый механизм, который обеспечивает основу имитации классов. А именно, он играет роль *функции-конструктора*, которая используется совместно с оператором `new` для создания экземпляров объектов, и роль шаблона, на основе которого создаются эти объекты.

Для иллюстрации в примере 10.1 вашему вниманию предлагается короткий фрагмент программного кода, который имитирует простой класс `Shape` на языке JavaScript. Обратите внимание: в соответствии с общепринятыми соглашениями имена классов начинаются с заглавных символов.

Для простоты примеры в этой главе не используют пространства имен для полной квалификации объектов. Однако в действительности вам придется использовать пространства имен, и мы вернемся к ним в главе 12.

*Пример 10.1. Типичный класс на JavaScript*

```
// Определение класса
function Shape(centerX, centerY, color)
{
    this.centerX = centerX;
    this.centerY = centerY;
    this.color = color;
};

// Создание экземпляра
s = new Shape(10, 20, "blue");
```

Как только интерпретатор JavaScript выполнит определение функции, объект Shape появится в памяти и сможет играть роль объекта-прототипа для создания экземпляров с помощью функции-конструктора и оператора new.

Для полноты картины следует заметить, что объявление объекта Shape можно было бы выполнить более компактно, воспользовавшись функцией extend из библиотеки Base:

```
// Создать объект типа Function
function Shape() {}

// Расширить этот прототип некоторыми значениями по умолчанию
dojo.extend(Shape, {
    centerX : 0,
    centerY : 0,
    color : ""
});
```

К сожалению, этот класс покажется вам интересным не более трех секунд, потому что быстро надоест, и вам захочется смоделировать нечто более конкретное, например определенную фигуру. Для этого *можно было бы* воспроизвести совершенно новый класс, например класс окружностей, но более практичный подход заключается в наследовании класса окружности от класса фигуры, который уже определен, потому что любая окружность является частным случаем фигуры. К тому же у вас уже имеется класс фигуры Shape, так почему бы не использовать его?

Пример 10.2 демонстрирует один из способов, обеспечивающих такое наследование в JavaScript.

*Пример 10.2. Типичное наследование в JavaScript*

```
// Определение подкласса
function Circle(centerX, centerY, color, radius)
{
    // Сначала необходимо обеспечить наследование свойств суперкласса,
    // создав ссылку на функцию-конструктор суперкласса и затем
    // вызвав функцию-конструктор внутри подкласса.
    this.base = Shape;
    this.base(centerX, centerY, color);

    // Создать остальные свойства подкласса
    this.radius = radius;
};

// Явно присоединить прототип подкласса к суперклассу, чтобы любые новые
// свойства, динамически добавляемые к суперклассу, появлялись и в подклассе
Circle.prototype = new Shape;

// Создание экземпляра
c = new Circle(10, 20, "blue", 2);

// Окружность ЯВЛЯЕТСЯ фигурой
```

Несмотря на то что этот пример достаточно интересен, тем не менее вы могли обнаружить, что все не так просто и гладко, как можно было бы подумать в самом начале, но и не настолько сложно по сравнению с действительно крутым веб-приложением, которое вы пытаетесь закончить.

## Шаблон смешивания

Для демонстрации альтернативной парадигмы типичного представления о наследовании рассмотрим пример 10.3, где для моделирования фигуры и окружности используется иной подход, основанный на смешанных классах. Следует особо отметить, что в случае смешанных классов широко используется грубое определение типа данных и отношение типа *имеет*. Вспомните: понятие грубого определения типа данных основано на идее, что если нечто крякает как утка и ведет себя как утка, значит, это утка. В нашем случае это понятие транслируется так: если объект обладает свойствами фигуры или окружности, это достаточный повод, чтобы считать объект фигурой или окружностью. Другими словами, совершенно не важно, чем *является* объект на самом деле, если он обладает необходимыми свойствами.

*Пример 10.3. Смешивание – альтернатива наследованию*

```
//Создать простой объект для моделирования фигуры
var shape = {}
    //Добавить все, что необходимо, чтобы объект "выглядел как фигура
    //и крякал как фигура"
    dojo.mixin(shape, {
    centerX : 10,
    centerY : 20,
    color : "blue"
});

//Если позднее потребовалось что-то еще. Нет проблем,
//просто нужно добавить то, что необходимо
dojo.mixin(shape, {
    radius : 2
});

//Теперь фигура ИМЕЕТ радиус
```

Следует заметить, что этот пример смешанного класса не может служить точной заменой предыдущего примера, в котором используется наследование прототипов. Этот пример скорее предназначен, чтобы продемонстрировать различные способы решения проблемы.

## Шаблон делегирования

Существует еще один способ моделирования взаимоотношений между фигурой и окружностью – делегирование, который демонстрируется в примере 10.4. В то время как при использовании смешанных клас-

сов все свойства фактически копируются в единственный экземпляр объекта, в случае делегирования ответственность за наличие некоторых свойств возлагается на другой объект, который уже обладает ими.

*Пример 10.4. Делегирование – альтернатива наследованию*

```
//Создать простой объект
var shape = {}

//Добавить в него все, чем должен обладать этот экземпляр
dojo.mixin(shape, {
    centerX : 10,
    centerY : 20,
    color : "blue"
});

//объект окружности делегирует ответственность за свойства centerX, centerY,
//и color объекту фигуры и добавляет свое свойство radius
circle = dojo.delegate(shape, {
    radius : 2
});
```

Главная особенность этого примера в том, что свойство `radius`, определенное в виде объекта-литерала, добавляется в объект окружности, а остальные свойства фигуры – нет. Вместо этого всякий раз, когда происходит обращение к свойствам объекта окружности, которых он не имеет, он делегирует эти обращения к объекту фигуры. Таким образом:

- Значение свойства `radius` возвращается непосредственно объектом `circle`, потому что свойство `radius` было добавлено непосредственно в этот объект.
- Значения свойств `centerX`, `centerY` и `color` извлекаются из объекта `shape`, потому что они отсутствуют в объекте `circle` (если говорить в общих чертах).
- При обращении к любым другим свойствам по определению возвращается значение `undefined`, потому что они отсутствуют в объектах `circle` и `shape`.

Из-за простоты примеров шаблон, основанный на смешивании, выглядит более предпочтительным, тем не менее шаблон делегирования используется достаточно широко, особенно в ситуациях, когда имеется большое число объектов, обладающих одним общим подмножеством свойств.

## Имитация классов с использованием средств Dojo

Теперь, когда вам была предоставлена возможность сравнить различные формы наследования, пришло время познакомиться с основной конструкцией в инструментальном наборе, которая используется для

объявления классов и имитации иерархий наследования. Инструментарий Dojo обеспечивает максимальную простоту, скрывая все особенности реализации, связанные с объявлением классов и наследованием, за маленькой и элегантной функцией в библиотеке Base, которая называется `dojo.declare`. Эту функцию легко запомнить, потому что с ее помощью выполняется *объявление* (*declaring*) класса. Краткое описание функции приводится в табл. 10.1.

*Таблица 10.1. Функция dojo.declare*

| Имя | Комментарий |
|---|---|
| `dojo.declare (/*String*/ className, /*Function\|Function[]*/ superclass, /*Object*/ props)` | Обеспечивает компактный способ объявления функции-конструктора. В аргументе `className` ей передается имя создаваемой функции-конструктора, в аргументе `superclass` – либо единственный объект Function, либо массив объектов Function, которые будут являться предками, а в аргументе `props` передается объект, свойства которого будут скопированы в прототип функции-конструктора. |

Как вы уже могли предположить, функция `declare` создана на основе шаблонов, реализованных такими функциями, как `extend`, `mixin` и `delegate`, и обеспечивает более богатую абстракцию, чем любой из этих трех шаблонов.

В примере 10.5 показано, как можно использовать функцию `dojo.declare` для создания иерархии наследования между фигурой и окружностью. Пока рассмотрим этот пример без привязки к реальности, а наиболее интересные его моменты обсудим немного ниже.

*Пример 10.5. Имитация наследования классов с помощью dojo.declare*

```
// "Объявить" класс Shape
dojo.declare(
    "Shape", //Имя класса
    null,    //Предков нет, поэтому указано значение null
    {
        centerX : 0, // Атрибуты
        centerY : 0,
        color : "",

        // Функция-конструктор, которая вызывается как "new Shape"
        constructor: function(centerX, centerY, color)
        {
            this.centerX = centerX;
            this.centerY = centerY;
            this.color = color;
        }
```

```
    }
);

// К этому моменту можно было бы создать экземпляр объекта:
// var s = new Shape(10, 20, "blue");

// "Объявить" класс Circle
dojo.declare(
    "Circle", //Имя класса
    Shape,    //Класс-предок
    {
        radius : 0,

        // Функция-конструктор, которая вызывается как "new Circle"
        constructor: function(centerX, centerY, color, radius)
        {
            // Конструктор класса Shape будет вызван автоматически
            // с теми же аргументами. Обратите внимание, что он
            // просто проигнорирует аргумент radius, потому
            // что использует только первые три аргумента
            this.radius = radius; //присвоить значение свойству класса Circle
        }
    }
);

// Функции-конструктору параметры передаются через
// конструктор dojo.declare
c = new Circle(10,20,"blue",2);
```

Хотелось бы надеяться, что вы находите использование функции `dojo.declare` удобочитаемым, простым и вполне очевидным. В зависимости от того, как вы используете отступы и разрывы строк, объявление класса может даже «напоминать» обычные объектно-ориентированные языки. Единственное, что может показаться странным, – это то, что конструктор класса `Shape` вызывается с теми же параметрами, которые передаются функции `constructor` класса `Circle`. Однако это не влечет за собой никаких проблем, так как функция `constructor` класса `Shape` принимает только три именованных аргумента, попросту игнорируя любые дополнительные аргументы. (К этому мы вскоре еще вернемся.)

Разговоры о функциях-конструкторах JavaScript, которые используются совместно с оператором `new` для создания объектов JavaScript, и функциях-конструкторах, которые передаются в третьем параметре `dojo.declare`, могут показаться запутывающими. Чтобы далее избежать неясности, всякое упоминание третьего параметра функции `dojo.declare` будет сопровождаться ключевым словом `constructor`, а когда речь будет идти о функции-конструкторе JavaScript, это ключевое слово будет отсутствовать.

### Параметры функций JavaScript

Во многих языках программирования не допускается передавать функциям переменное число параметров, если это не определено явно. Например, в языке C++, чтобы объявить, что функция может принимать произвольное число параметров, используется нотация `...`.

Однако в языке JavaScript функции благополучно могут принимать любое число параметров. Если они получают больше параметров, чем указано в объявлении функции, все лишние параметры они просто игнорируют, а если они получают меньшее число параметров, недостающим параметрам присваивается значение `undefined`.

Разумеется, вы всегда можете получить полный доступ ко всем параметрам функции с помощью специальной переменной `arguments`. Однако, имейте в виду: несмотря на то что вы можете обращаться к этой переменной как к массиву, на самом деле она не является массивом JavaScript. Например, у нее отсутствуют методы `push` и `pop`.

В большинстве ситуаций, имеющих отношение к наследованию, вам не придется использовать переменную `arguments`, потому что вам придется опираться на подмножество именованных параметров в каждой функции `constructor` классов, объявленных средствами Dojo, как показано в примере 10.5.

## Основной шаблон создания класса

Функция `dojo.declare` является основным способом создания классов, который важно понимать, потому что библиотека Dijit полагается на него при разработке гибкого шаблона, который эффективно автоматизирует решение различных задач, связанных с созданием виджетов. В главе 12 эта тема раскрыта подробнее.

Хотя эта глава практически полностью посвящена изучению функции `constructor`, потому что это, вне всяких сомнений, самый используемый метод, тем не менее следующий пример показывает, что есть еще две функции, предоставляемые функцией `dojo.declare`: `preamble`, которая объявляется перед функцией `constructor`, и `postscript`, которая объявляется после нее:

```
preamble(/*Object*/ params, /*DOMNode*/node)
    //предшествует функции constructor

constructor(/*Object*/ params, /*DOMNode*/node)
    // вызывает функции constructor всех суперклассов
```

```
        // вызывает функции constructor всех подмешанных классов
        // вызывает функцию constructor локального класса, если имеется

    postscript(/*Object*/ params, /*DOMNode*/node)
        // чаще всего начинает создание виджета
```

Для проверки можете попробовать выполнить программный код, представленный в примере 10.6.

*Пример 10.6. Основной способ создания класса с помощью dojo.declare*

```
dojo.addOnLoad(function() {
    dojo.declare("Foo", null, {

        preamble: function() {
            console.log("preamble", arguments);
        },

        constructor : function() {
            console.log("constructor", arguments);
        },

        postscript : function() {
            console.log("postscript", arguments);
        }
    });

    var foo = new Foo(100); //последовательно будут вызваны preamble,
                            //constructor и postscript
});
```

Функция `constructor` — это место, где выполняется большая часть операций, связанных с моделированием класса, однако функциям `preamble` и `postscript` также находится применение. Функция `preamble` в основном используется для манипулирования аргументами суперкласса. Аргументы, которые передаются функции `constructor`, в данном случае — `new Foo(100)`, последовательно передаются функциям `preamble`, `constructor` и `postscript`, но такое положение вещей не является обязательным, когда имеется иерархия наследования. Мы еще вернемся к этой теме во врезке «Упреждающее манипулирование аргументами» ниже в этой главе, после того как будет дано формальное определение наследования в следующем разделе. Функция `postscript` используется главным образом при создании виджетов. Глава 12 практически целиком посвящена исследованию всего жизненного цикла виджетов.

## Пример простого наследования

Давайте перейдем к изучению разнообразных примеров, демонстрирующих некоторые особенности `dojo.declare`. Первый пример снабжен обширными комментариями и демонстрирует несколько более сложный вариант наследования, показывая важные особенности использования внутреннего метода `constructor` функции `dojo.declare`:

```
<html>
    <head>
        <title>Fun with Inheritance!</title>

        <script
            type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js">
        </script>

        <script type="text/javascript">
            dojo.addOnLoad(function() {
                //Здесь объявляется обычный объект Function.
                function Point(x,y) {}
                dojo.extend(Point, {
                    x : 0,
                    y : 0,
                    toString : function() {return "x=",this.x," y=",this.y;}
                });

                dojo.declare(
                    "Shape",
                    null,
                    {
                        //Сначала необходимо определить члены класса,
                        //но инициализироваться они будут в функции
                        //constructor. Никогда не инициализируйте объект
                        //Function в этом ассоциативном массиве, если
                        //не хотите, чтобы он был доступен *всем*
                        //экземплярам класса, что обычно нетипично.

                        //В соответствии с общепринятыми соглашениями
                        //начальный символ подчеркивания обозначает
                        //"частные" члены

                        _color: "",
                        _owners: null,

                        //Инструментарий Dojo предусматривает для классов
                        //специальную функцию constructor. Это она и есть.
                        //Примечательно, что эта функция constructor
                        //будет вызываться с теми же аргументами,
                        //которые будут переданы функции constructor
                        //класса Circle, даже при том, что конструктор
                        //суперкласса нигде напрямую не вызывается.
                        constructor: function(color)
                        {
                            this._color = color;
                            this._owners = [0]; //Об инициализации объектов
                            //говорится в комментарии выше

                            console.log("Created a shape with color",
                                this._color, "owned by", this._owners);
                        },

                        getColor : function() {return this._color;},
```

```
        addOwner : function(oid) {this._owners.push(oid);},
        getOwners : function() {return this._owners;}

        //Не оставляйте завершающую запятую после
        //последнего элемента. Не все броузеры прощают
        //эту ошибку (или выводят достаточно осмысленное
        //сообщение об ошибке). Сделайте татуировку
        //с этим замечанием на тыльной стороне кистей рук.
    }
);

//Важное соглашение:
//В случае простого наследования сначала следует список
//аргументов конструктора суперкласса, за ними следуют
//аргументы, характерные для подкласса.
//Конструктор подкласса вызывается с полным набором
//аргументов, поэтому при правильном определении
//аргументов и если вы преднамеренно не манипулируете
//аргументами суперкласса в конструкторе подкласса,
//все будет работать безошибочно.

//Запомните: первый аргумент функции dojo.declare -
//строка, второй - объект Function.

dojo.declare(
    "Circle",
    Shape,
    {
        _radius: 0,
        _area: 0,
        _point: null,

        constructor : function(color,x,y,radius)
        {
            this._radius = radius;
            this._point = new Point(x,y);
            this._area = Math.PI*radius*radius;

            //Обратите внимание, что унаследованный член
            //_color уже определен и готов к использованию!
            console.log("Circle's inherited color is " +
                        this._color);
        },

        getArea: function() {return this._area;},
        getCenter : function() {return this._point;}
    }
);

console.log(Circle.prototype);

console.log("Circle 1, coming up...");
c1 = new Circle("red", 1,1,100);
console.log(c1.getCenter());
console.log(c1.getArea());
```

```
                console.log(c1.getOwners());
                c1.addOwner(23);
                console.log(c1.getOwners());

                console.log("Circle 2, coming up...");
                c2 = new Circle("yellow", 10,10,20);
                console.log(c2.getCenter());
                console.log(c2.getArea());
                console.log(c2.getOwners());
            });
        </script>
    </head>
    <body>
    </body>
</html>
```

Завершающие запятые наверняка превратятся в ошибки в броузерах, отличных от Firefox, поэтому особенно тщательно проверяйте, чтобы не оставить их в программном коде. Некоторые языки программирования, такие как Python, допускают наличие завершающих запятых, поэтому, если вы программируете на одном из таких языков, будьте особенно внимательны.

После запуска этого примера вы должны увидеть результаты в консоли Firebug, как показано на рис. 10.1.

Важный вывод, который следует из этого примера, заключается в том, что к моменту, когда `dojo.declare` завершит свою работу, объект Func-

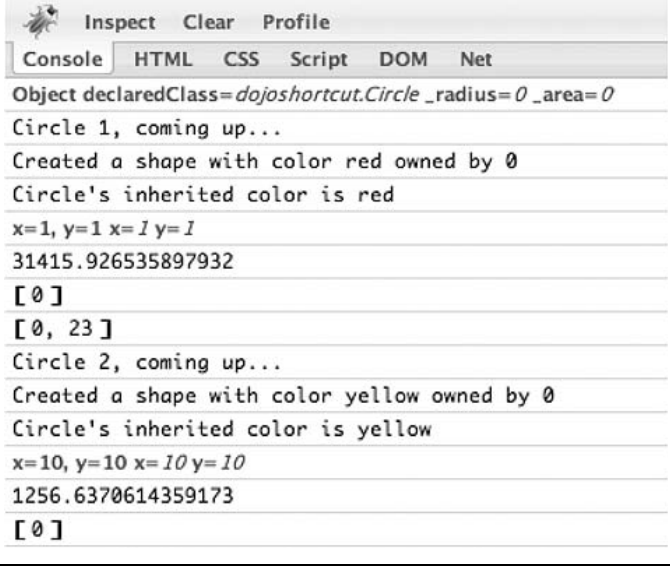



***Рис. 10.1.*** *Результат работы примера 10.6 в консоли Firebug*

tion будет существовать в памяти. Истинный объект Function будет существовать неявно, а его прототип будет содержать все, что было передано функции `dojo.declare` в третьем параметре. Этот объект служит прототипом для всех объектов, которые будут созданы в дальнейшем. Эта особенность может вызывать сложности, если не иметь полных сведений о ней, и является темой следующего раздела.

## Типичные ошибки при использовании механизма наследования на базе прототипов

Как известно, объект Point не имеет совершенно никакого отношения к Dojo. Это обычный объект JavaScript типа Function. Кроме того, его не требуется инициализировать вместе с другими свойствами внутри ассоциативного массива класса `Shape`. В противном случае он будет вести себя как статический член класса, совместно используемый всеми объектами `Shape`, которые будут созданы впоследствии, и *это может привести к действительно странному поведению, если только вы специально не добивались такого результата*.

Проблема возникает из-за того, что внутри функции `declare` смешиваются все свойства в прототипе объекта, а свойства прототипа совместно используются всеми экземплярами. Для таких неизменяемых типов, как числа или строки, изменение свойства приводит к изменению локальной копии. В случае изменяемых типов, таких как `Object` или `Array`, изменение свойства одного экземпляра распространяется на все экземпляры. В примере 10.7 показано, как проявляется эта проблема.

*Пример 10.7. Свойства прототипа совместно используются всеми экземплярами*

```
function Foo() {}
Foo.prototype.bar = [100];

//Создать два экземпляра Foo
foo1 = new Foo;
foo2 = new Foo;

console.log(foo1.bar); // [100]
console.log(foo2.bar); // [100]

// Эта инструкция изменяет прототип, используемый всеми экземплярами объекта
foo1.bar.push(200);

//...поэтому изменения обнаруживаются в обоих экземплярах.
console.log(foo1.bar); // [100,200]
console.log(foo2.bar); // [100,200]
```

Чтобы предотвратить даже появление *мысли* о выполнении действий, которые могут привести к появлению ошибок, вызванных инициализацией сложных типов данных, выполняйте инициализацию всех свойств, даже инициализацию свойств простых типов, внутри стандартной функции `constructor`, и возьмите этот стиль программирования

в привычку. Чтобы обеспечить максимальную удобочитаемость определений ваших классов, старайтесь перечислять все свойства класса вместе с необходимыми комментариями.

Чтобы продемонстрировать неприятные последствия на действующем примере, внесите следующие изменения, выделенные жирным шрифтом, в определение класса Shape и исследуйте полученные результаты в консоли Firebug:

```
//...обрезано...

dojo.declare("Shape", null,
    {
        _color: null,
        //_owners: null,
        _owners: [0], //это изменение превратит свойство _owners
                      //в статический член класса!

        constructor : function(color) {
            this._color = color;
            //this._owners = [0];
            console.log("Created a shape with color ",this._color,
                        " owned by ", this._owners);
        },

        getColor : function() {return this._color;},
        addOwner : function(oid) {this._owners.push(oid);},
        getOwners : function() {return this._owners;}
    }
);
//...обрезано...
```

После внесения этих изменений и попытки обновить страницу в броузере Firefox, вы увидите в консоли Firebug результаты, как показано на рис. 10.2.

## Вызов унаследованных методов

В других объектно-ориентированных языках программирования обычным явлением считается переопределение метода суперкласса в подклассе и вызов унаследованного метода суперкласса перед выполнением каких-либо операций, специфических для подкласса. Хоть и не всегда, но возникают ситуации, когда бывает необходимо вызывать метод базового класса, и подкласс реализует свое поведение на основе реализации базового класса. Любой класс, созданный с помощью dojo.declare, обладает доступом к специальному методу inherited, который вызывает соответствующий унаследованный метод суперкласса. (Обратите внимание, что унаследованные функции constructor вызываются автоматически и для этого не требуется обращаться к методу inherited.)

Пример 10.8 иллюстрирует эту возможность.

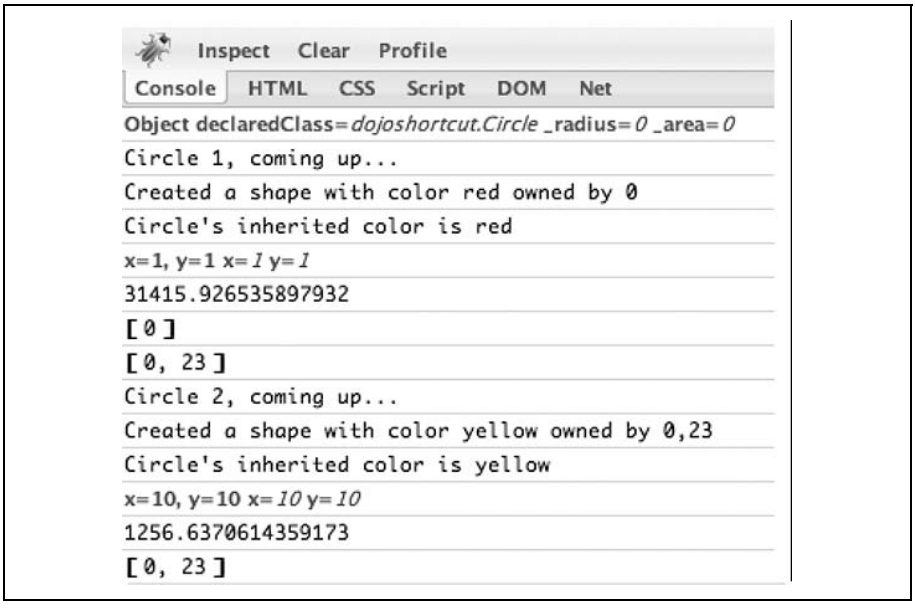



***Рис. 10.2.*** *Результаты в консоли Firebug*

*Пример 10.8. Вызов унаследованного метода суперкласса из метода подкласса*

```
dojo.addOnLoad(function() {
    dojo.declare("Foo", null, {

        constructor : function() {
            console.log("Foo constructor", arguments);
        },

        custom : function() {
            console.log("Foo custom", arguments);
        }
    });

    dojo.declare("Bar", Foo, {

        constructor : function() {
            console.log("Bar constructor", arguments);
        },

        custom : function() {
            //автоматически вызывает метод 'custom' класса Foo
            //и передает ему те же аргументы, хотя в случае необходимости
            //над ними можно выполнить предварительные манипуляции
            this.inherited(arguments);
            //без этого вызова метод Foo.custom никогда не будет вызван
            console.log("Bar custom", arguments);
        }
```

```
    });

    var bar = new Bar(100);
    bar.custom(4,8,15,16,23,42);
});
```

А ниже представлены соответствующие результаты в консоли Firebug:

```
Foo constructor [100]
Bar constructor [100]
Foo custom [4, 8, 15, 16, 23, 42]
Bar custom [4, 8, 15, 16, 23, 42]
```

## Множественное наследование посредством смешивания классов

В предыдущем разделе было начато обсуждение темы имитации механизма наследования классов в Dojo и было указано на некоторые критические проблемы, связанные с тонкостями языка JavaScript, которые оказывают существенное влияние на разработку приложений с использованием Dojo. В первом примере было продемонстрировано простое наследование, в котором суперкласс `Shape` составлял основу для подкласса `Circle`, однако инструментальный набор Dojo обеспечивает и ограниченную форму множественного наследования.

Процесс определения отношений наследования, в которых участвуют несколько объектов Function, называется *организация цепочки прототипов*, потому что в языке JavaScript иерархии создаются с использованием свойства `Object.prototype`. (Эта концепция была продемонстрирована в примере 10.2, где отношения между классом окружности и классом фигуры определялись вручную.)

При создании иерархий простого наследования инструментарий Dojo имитирует работу механизма наследования, опираясь на концепцию организации цепочек прототипов. Однако механизм множественного наследования действует несколько иначе, потому что в языке JavaScript объект Function обладает только одним свойством `prototype`.

Как вы уже могли догадаться, существует несколько способов решения этой проблемы. Решение, использованное в Dojo, основано на таком применении цепочек прототипов, когда определяется единственный предок прототипа, являющийся основой для организации цепочек прототипов и в то же время позволяющий внедрять в прототип предка другие подмешиваемые классы. Проще говоря, класс может иметь только один прототип, но объекты Function, образующие эти классы, при желании можно «снабдить» любым количеством функций-конструкторов. Формально прототипы этих функций-конструкторов не будут приниматься во внимание позднее, уже в течение срока жизни объекта, но, тем не менее они могут использоваться самыми разными способами. Эти подмешиваемые классы можно представить

себе как «интерфейсы, которые что-то делают» или «интерфейс + реализация».

В случае множественного наследования наследуемые классы передаются функции `dojo.declare` в виде списка. Первый элемент списка называется предком прототипа, тогда как остальные – это обычные подмешиваемые предки, или просто подмешиваемые классы.

Ниже показано, на что похоже множественное наследование в интерпретации инструментария Dojo. Единственное отличие здесь, которое наблюдается в третьем параметре функции `dojo.declare`, – это список вместо объекта Function. Первый элемент в этом списке является предком прототипа, а остальные – подмешиваемыми классами:

```
<html>
    <head>
        <title>Fun with Multiple Inheritance!</title>

        <script
            type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js">
        </script>

        <script type="text/javascript">
            dojo.addOnLoad(function() {
                //Типичный класс Dojo с привычным конструктором
                //и без прямых предков.
                dojo.declare("Tiger", null, {
                    _name: null,
                    _species: null,

                    constructor : function(name)
                    {
                        this._name = name;
                        this._species = "tiger";
                        console.log("Created ",this._name,
                                    " the ",this._species);
                    }
                });

                //Еще один типичный класс Dojo с обычным конструктором
                //и без прямых предков.
                dojo.declare("Lion", null, {
                    _name: null,
                    _species: null,

                    constructor: function(name) {
                        this._name = name;
                        this._species = "lion";
                        console.log("Created ",this._name,
                                    " the ",this._species);
                    }
```

```
                    });

                    //Класс Dojo с более чем одним предком. Первый предок -
                    //это предок прототипа, а второй (и все последующие
                    //функции) - это подмешиваемые классы. Обратите особое
                    //внимание на то, что все конструкторы суперклассов
                    //выполняются перед вызовом конструктора подкласса
                    //и в действительности нет никакого способа изменить
                    //такой порядок вещей.
                    dojo.declare("Liger", [Tiger, Lion], {
                        _name: null,
                        _species: null,

                        constructor : function(name) {
                            this._name = name;
                            this._species = "liger";
                            console.log("Created ",this._name, " the ",
                                        this._species);
                        }
                    });

                    lucy = new Liger("Lucy");
                    console.log(lucy);
                });
            </script>
        </head>
        <body>
        </body>
    </html>
```

Если открыть предыдущий пример в броузере и посмотреть в консоли Firebug на результаты, которые приводятся на рис. 10.3, можно увидеть, что конструкторы классов `Tiger` и `Lion` были вызваны перед конструктором класса `Liger`. Как и в предыдущем примере с фигурами, вы получили свой подкласс, но только после того, как были созданы все необходимые суперклассы – обращением к соответствующим конструкторам.

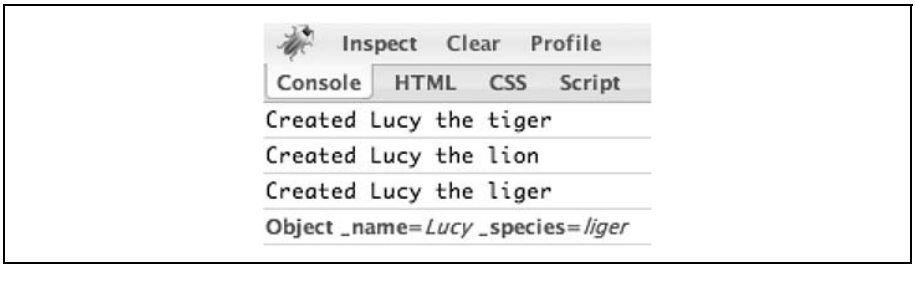



***Рис. 10.3.*** *В конечном счете вы получите свой объект класса Liger, но только после того, как будут созданы и инициализированы необходимые суперклассы*

## Особенности механизма множественного наследования

В предыдущих примерах с фигурами не было никакой необходимости задумываться о списке аргументов, передаваемых классом `Circle` в конструктор класса `Shape`, потому что класс `Circle` построен непосредственно на базе класса `Shape`. Кроме того, имело смысл, и даже было удобно определить первые аргументы функции `constructor` класса `Shape` как первые аргументы функции `constructor` класса `Circle`. В последнем примере с классами `Lion`, `Tiger` и `Liger` функции `constructor` всех трех классов принимают единственный аргумент, который во всех трех случаях имеет одно и то же назначение, поэтому и здесь не возникало никаких проблем.

Но, минутку, а как следовало бы поступить, если бы функции `constructor` классов `Tiger` и `Lion` имели различные сигнатуры? Например, функция `constructor` класса `Tiger` могла бы принимать в качестве аргументов не только имя, но и количество полосок, а функция `constructor` класса Lion могла бы принимать имя и длину гривы. Как тогда следует определить функцию `constructor` класса `Liger`, чтобы она обрабатывала такую ситуацию? Все аргументы, что передаются функции `constructor` класса `Liger`, будут переданы функциям `constructor` классов `Tiger` и `Lion`, а в этом нет никакого смысла.

В данном конкретном случае, когда конструкторы двух или более суперклассов требуют передачи собственных аргументов, лучшее, что можно было бы сделать, – это передавать именованные параметры в виде ассоциативного массива и использовать эти параметры в своей функции `constructor`, не полагаясь на список аргументов в свойстве `arguments`. Передача специфических параметров конструкторам суперклассов при использовании множественного наследования еще недостаточно хорошо поддерживается в версии Dojo 1.1, хотя проблему согласования аргументов предполагается решить в одной из последующих версий.

Вообще, при проектировании иерархии классов с множественным наследованием будет удобнее, если функции `constructor` всех суперклассов не будут иметь входных параметров. Преимущество такого подхода состоит в том, что преднамеренное создание конструкторов суперклассов так, чтобы они не имели входных аргументов, позволяет в конструкторах подклассов принимать и обрабатывать любое число специфических параметров, какое только потребуется, и при этом гарантируется, что ни один из суперклассов не будет подвержен нежелательному влиянию. В конце концов, аргументы не могут повлиять на суперклассы, если они не используются их конструкторами.

## Упреждающее манипулирование аргументами

В Dojo 1.0 появилась новая особенность, имеющая отношение к функции `dojo.declare`, позволяющая решать проблему соответствия аргументов в ситуациях, когда необходимо выполнять манипуляции над аргументами, которые передаются функции `constructor` суперкласса.

В двух словах, перед вызовом функции `constructor` всегда вызывается функция `preamble`, и вы можете использовать ее для изменения аргументов, которые передаются функции `constructor` *суперкласса*. Начиная с версии Dojo 1.1 возвращаемые значения, полученные от функции `preamble`, передаются функциям `constructor` всех суперклассов. Несмотря на то что это никак не снимает проблему, обозначенную выше, в примере с классом `Liger`, тем не менее такая возможность может оказаться весьма полезной во многих других ситуациях.

Ниже приводится небольшой фрагмент программного кода, который демонстрирует, как работает функция `preamble`:

```
dojo.declare("Foo", null, {
    preamble: function(){
        console.log("Foo preamble: ", arguments);
    },

    constructor: function(){
        console.log("Foo constructor: ", arguments);
    }
});

dojo.declare("Bar", null, {
    preamble: function(){
        console.log("Bar preamble: ", arguments);
    },

    constructor: function(){
        console.log("Bar constructor: ", arguments);
    }
});

dojo.declare("Baz", [Foo, Bar], {
    preamble: function(){
        console.log("Baz preamble: ", arguments);
        return ["overridden", "baz", "arguments"];
    },

    constructor: function(){
        console.log("Baz constructor: ", arguments);
```

```
    }
});

var obj = new Baz("baz", "arguments", "go", "here");
```

Вывод, полученный в консоли Firebug, показывает хронологический порядок вызовов функций `preamble` и `constructor`, вовлеченных в создание экземпляра класса `Baz`. Обратите внимание, что в качестве аргументов функциям `constructor` суперклассов Foo и Bar передаются возвращаемые значения, полученные от функции `preamble`. Однако, функция `constructor` класса `Baz` принимает те же самые аргументы, что и функция `preamble` этого класса:

```
Baz preamble: ["baz", "arguments", "go", "here"]
Foo preamble: ["overridden", "baz", "arguments"]
Foo constructor: ["overridden", "baz", "arguments"]
Bar preamble: ["overridden", "baz", "arguments"]
Bar constructor: ["overridden", "baz", "arguments"]
Baz constructor: ["baz", "arguments", "go", "here"]
```

# В заключение

После изучения содержимого этой главы вы должны:

- Понимать, как использовать функцию `dojo.declare` для имитации классов в Dojo
- Уметь реализовывать отношения простого и множественного наследования средствами инструментария Dojo
- Понимать опасности, связанные с инициализацией объектов JavaScript за пределами функции `constructor`
- Знать, что объекты Function являются тем самым механизмом, который используется для имитации классов в языке JavaScript. Запомните, в JavaScript нет «настоящих» классов
- Понимать некоторое различие между наследованием на основе прототипов и наследованием на основе классов
- Иметь общее представление о том, как инструментарий Dojo использует наследование на основе прототипов для имитации наследования на основе классов

Далее следует вторая часть книги, где будут рассматриваться библиотеки Dijit и Util.

# II

# Dijit и Util

В первой части книги рассматривались Base и Core – библиотеки JavaScript, способные упростить разработку практически любых веб-приложений. Вторая часть книги охватывает визуальные элементы инструментария; библиотека *Dijit* содержит множество очень интересных и нужных виджетов, а библиотека *Util* содержит инструменты сборки и самостоятельную платформу модульного тестирования.

Библиотека Dijit представляет собой фантастический слой виджетов и предоставляет полную коллекцию готовых к использованию элементов форм, контейнеров размещения и других элементов управления общего назначения, способных удовлетворить самого взыскательного пользователя. Библиотека Dijit основана непосредственно на библиотеках Base и Core и является ярким примером того, чего можно достичь, используя мощную стандартную библиотеку, изолирующую вас от несовместимостей между броузерами и уменьшает объем шаблонного программного кода, который приходится писать, отлаживать, тестировать и сопровождать. В этом смысле библиотека Dijit является естественным продолжением библиотек Base и Core.

В первой части книги были представлены фундаментальные строительные блоки инструментария, которые дают вам возможность качественно повысить уровень разработки приложений на языке JavaScript. В этой части книги вашему вниманию будут представлены все диджиты (виджеты Dojo), которые можно просто брать с полки, вставлять их в страницу, и они «просто будут работать», при этом вам почти не придется писать программный код. Дизайнеры получат особое удовольствие от этой части книги, потому что она предоставляет всесторонний обзор доступных диджитов и описывает, как можно добавлять их прямо в разметку HTML страницы.

И, конечно, перед вами открывается широчайший диапазон возможностей для создания собственных диджитов – с самого начала или на

основе существующих диджитов – поэтому будет описываться и эта сторона библиотеки. Глубокое изучение анатомии и жизненного цикла виджетов; параллельное рассмотрение возможностей объявления виджетов в разметке и создание их программным способом; обсуждение вопросов обеспечения доступности (в англоязычной литературе употребляется сокращение a11y – от слова «accessibility» (доступность), так как это слово начинается с символа «a», заканчивается символом «y» и между ними еще 11 символов) – все это вы найдете во второй части.

После полного охвата библиотеки Dijit во второй части рассматривается библиотека Util, удивительная коллекция инструментов сборки, в состав которой входит ShrinkSafe – система сжатия кода JavaScript, основанная на проверенном механизме Rhino JavaScript, и Dojo Objective Harness (DOH) – автономная платформа модульного тестирования, упрощающая процедуру тестирования и проверки качества вашего приложения.

# 11

# Обзор Dijit

Библиотека Dijit – это фантастический слой виджетов, предоставляемых инструментарием в качестве замены стандартных веб-элементов управления HTML. Эта глава начинает вторую часть книги с общего описания основ построения библиотеки Dijit, ее философии и преследуемых целей, которое затем плавно перетекают в обсуждение того, как дизайнеры и рядовые создатели страниц могут использовать диджиты в разметке без программного кода или с небольшим его количеством. Заканчивается глава полным обзором всего того, что включено в состав библиотеки Dijit.

## Причины появления Dijit

Веб-разработка – это в значительной степени техническая задача; она имеет весьма интересную историю развития, полную остроумных решений и находок. Несмотря на то что с концептуальной точки зрения веб-броузер может представлять собой идеальную платформу, тем не менее с технической точки зрения практически любой интересный случай улучшения восприятия работающего приложения пользователем связан с поиском обходных путей и решений того или иного вида. По *весьма скромным* оценкам, неполное соответствие стандартам продукции, выпускаемой большинством игроков на рынке, привело к появлению не менее полудюжины конкурирующих между собой платформ, а если учесть нарастающее развитие мобильных устройств, обладающих возможностью доступа во Всемирную паутину, это число возрастает еще больше.

Следовательно, разработка удобных в сопровождении веб-приложений стала еще более трудоемким делом, чем когда-либо – если не поддерживать максимально возможное число программных платформ, можно частично потерять долю рынка, популярность и доходы. Попытка

обеспечить поддержку различных платформ за счет применения громоздких, но все еще привычных приемов смешивания HTML, CSS и JavaScript для каждого конкретного случая приводит к необходимости прикладывать практически невозможные усилия.

Вы уже знаете, что библиотеки Base и Core изолируют вас от проблем несовместимости между броузерами и позволяют уменьшить время поиска обходных путей. Библиотека Dijit использует все самое лучшее из Base и Core и предоставляет расширяемую платформу для создания модульных визуальных элементов пользовательского интерфейса многократного применения.

Многие пользователи Dojo не совсем корректно считают названия «Dijit» и «Dojo» синонимами, потому что виджеты из библиотеки Dijit пользуются большой популярностью. Библиотека Dijit – это самостоятельный проект, развивающийся в рамках набора инструментальных средств, а его логическая изолированность от остального инструментального набора улучшает управляемость и способность к совершенствованию. В дополнение к коллекции готовых к употреблению виджетов библиотека Dijit предоставляет вам основу для построения своих собственных виджетов – ту же самую инфраструктуру, которая используется библиотекой Dijit.

Из конкретных задач, решаемых библиотекой Dijit, можно назвать:

- Создание стандартного набора виджетов для веб-разработки – по аналогии с тем, как библиотека Swing предоставляет компоненты интерфейса для приложений на языке Java, или библиотека Cocoa предоставляет компоненты интерфейса для приложений OS X
- Использование существующих механизмов из библиотек Core и Base, чтобы максимально обеспечить простоту и переносимость виджетов
- Обеспечение соответствия стандартам доступности согласно спецификации ARIA (Accessibility for Rich Internet Applications – стандарт доступности активных Интернет-приложений), определяющей, как обеспечить поддержку людей с ограниченными возможностями
- Обеспечение глобализации всех виджетов, что упрощает проблему интернационализации, гарантирует локализацию виджетов и поддержку региональных форматов и двунаправленного письма
- Поддержание непротиворечивого прикладного интерфейса, чтобы обработчики могли делиться знаниями о виджетах и использовать чужой опыт при решении проблем
- Поддержка унифицированного внешнего вида с помощью таблиц стилей и обеспечение простоты настройки виджетов
- Обеспечение возможности такого же простого создания виджетов в коде разметки, как и в программном коде JavaScript (или даже проще)

- Простота добавления виджетов в существующие страницы и объединения многочисленных виджетов в полноценные приложения
- Полная поддержка двунаправленного письма (реализовано в версии 1.1)
- Поддержка большинства наиболее распространенных броузеров в самых разных операционных системах, включая Internet Explorer 6+, Firefox 2+ и Safari 3+[1]

## Независимость и самодостаточность

Пожалуй, самое важное преимущество, которое привносит библиотека Dijit в веб-разработку, – это возможность *инкапсулировать* компоненты пользовательского интерфейса в самостоятельные виджеты. Если у вас уже имеется опыт разработки веб-приложений, то вам наверняка приходилось сталкиваться с проблемой объединения файлов HTML, CSS и JavaScript, содержащих реализацию пользовательского интерфейса, в переносимые пакеты, которые можно использовать для обеспечения требуемой функциональности в нужный момент времени и с минимальными усилиями.

В программировании проблема разработки модульных компонентов пользовательского интерфейса хорошо описывается такими терминами, как *независимость* и *самодостаточность*. Мера самодостаточности определяет, насколько полно исходный программный код и ресурсы обеспечивают необходимые функциональные возможности, а мера независимости определяет степень независимости от других модулей. При разработке таких программных компонентов, как виджеты, практически всегда главной целью является обеспечение максимальной самодостаточности и минимальной зависимости.

Библиотека Dijit прекрасно соответствует этим требованиям, и, что особенно замечательно, она образует инфраструктурный слой, который вам не придется писать, отлаживать и сопровождать. Применяя функцию `dojo.declare` из библиотеки Base для моделирования классов, как показано на рис. 11.1, библиотека Dijit добавляет к стандартным методам создания и удаления, управляющим жизненным циклом объекта, стандартизованные средства реагирования на события, такие

[1] Официально библиотека Dijit не поддерживает тот же набор броузеров, что и библиотеки Base и Core. Это решение обусловлено отсутствием достаточно широкой аудитории пользователей, ради которой было бы оправдано выполнять разработку программного кода, производить тестирование и обеспечивать поддержку таких броузеров, как Opera или Konqueror. Однако, отсутствие «официальной» поддержки вовсе не означает, что будет слишком сложно обеспечить работоспособность Dijit на этих платформах, особенно если учесть, что Konqueror, Firefox и WebKit (ядро броузера Safari) являются открытыми проектами.

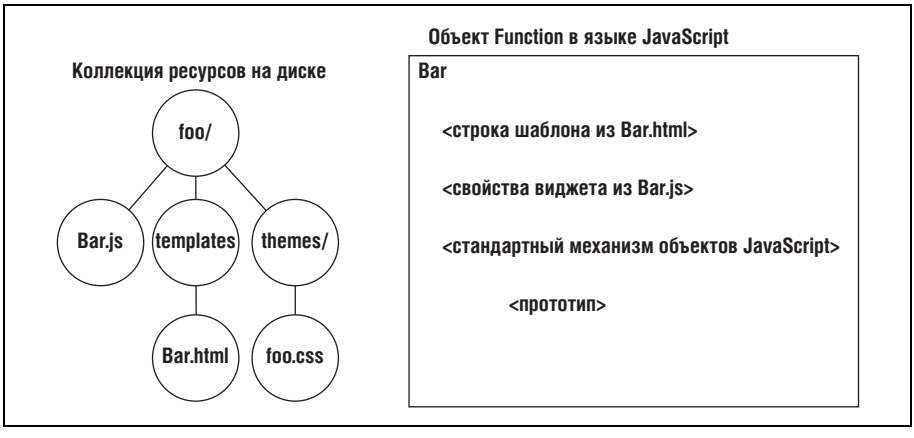



***Рис. 11.1.*** *Представление диджита как коллекции физических ресурсов на диске и как объекта типа Function*

как нажатия клавиш на клавиатуре и перемещение указателя мыши, а также возможность обеспечивать визуальное представление. Кроме того, она обеспечивает возможность реализовать управление всем этим как в коде разметки HTML, так и в программном коде JavaScript, предоставляя одни и те же функциональные возможности разным группам разработчиков.

Дизайнер легко сможет добавить диджит в страницу – для этого потребуется лишь включить в тег специальный атрибут `dojoType`. Парсер проанализирует этот атрибут и создаст экземпляр объекта DHTML, управляемого событиями. Например, ниже приводится измененный фрагмент из главы 1, который наглядно демонстрирует, насколько просто, используя только код разметки, добавить в форму текстовое поле ввода, в котором в первом приближении проверяется правильность ввода адреса электронной почты:

```
<input type="text"
    length=25
    name="email"
    dojoType="dijit.form.ValidationTextBox"
    trim="true"
    lowercase="true"
    regExp="[a-z0-9._%+-]+@[a-z0-9-]+\.[a-z]{2,4}"
    required="true"
    invalidMessage="Please enter a valid e-mail address"/>
```

Именно так. Чтобы *использовать* виджет, не потребовалось написать ни строчки программного кода JavaScript. Без сомнения, многим разработчикам приходится создавать или расширять виджеты, что влечет за собой необходимость писать программный код на языке JavaScript, но вся прелесть заключается в том, что после того, как этот код

будет написан, он становится частью вашего стандартного арсенала. После того как страница будет загружена, парсер отыщет атрибут dojoType, запросит у сервера все необходимые дополнительные ресурсы (если таковые имеются) и пересадит виджет DHTML в страницу. Создание пользовательского интерфейса при таком подходе не должно вызывать сложностей!

Безусловно, все, что можно реализовать в коде разметки, можно реализовать и на JavaScript. Вы можете создавать те же самые виджеты динамически, как обычные объекты JavaScript, и вставлять их в страницу, приложив незначительные усилия.

В типичном случае функция-конструктор диджита имеет следующую сигнатуру, согласно которой она принимает коллекцию параметров настройки и ссылку на узел:

```
dijit.WidgetName(/*Object*/props, /*DOMNode|String*/node)
```

Каждый диджит обладает ссылкой на узел DOM, который является его визуальным представлением, и все, что необходимо, чтобы обеспечить взаимодействие диджита с пользователем, – это добавить узел в страницу. Как только он станет видимым, все его обработчики событий становятся доступными, и диджит будет вести себя так, как будто он все время присутствовал здесь. Ниже показано, как программным способом можно создать тот же самый диджит, выполняющий проверку корректности адреса электронной почты. Параллель между этими двумя подходами вполне очевидна:

```
<script type="text/javascript">
    var w = new dijit.form.ValidationTextBox({
        length : 25,
        name : "email",
        trim : true,
        lowercase : true,
        regExp : "[a-z0-9._%+-]+@[a-z0-9-]+\.[a-z]{2,4}",
        required : true,
        invalidMessage : "Please enter a valid e-mail address"
    }, n); // n - это ссылка на узел, находящийся где-то на странице
</script>
```

# Доступность

Доступность – чрезвычайно важная тема в век информации. Важнейшая ее цель выражается в доставке содержимого широкому кругу пользователей (обычных или с ограниченными возможностями) в соответствии с законодательными актами, например Section 508[1], который

[1] Section 508 – это поправка к закону США о реабилитации (Rehabilitation Act) от 1973 года, требующая, чтобы федеральные органы обеспечивали необходимые удобства для американцев с ограниченными возможностями.

устанавливает стандартный минимум, обеспечивающий доступность технологий лицам с ограниченными возможностями. Помимо этого существуют еще и экономические стимулы. Так, по оценкам министерства труда США, дискреционные расходы[1] лиц с ограниченными возможностями составляют порядка 175 миллиардов долларов (*http://www.usdoj.gov/crt/ada/busstat.htm*). Неважно, как вы смотрите на это и какие мотивы движут вами, но проблема обеспечения доступности имеет слишком большое значение, чтобы ее можно было игнорировать.

## Типичные проблемы доступности

Хотя в этом коротком разделе невозможно даже начать обсуждение бесчисленных тонкостей, имеющих отношение к успешной реализации веб-приложений, тем не менее он должен дать вам представление об имеющихся проблемах и обозначить способы, которые используются библиотекой Dijit для их решения. Две наиболее типичные задачи обеспечения доступности состоят в поддержке пользователей с ослабленным зрением, которым приходится пользоваться устройствами чтения с экрана, и пользователей, которым требуется возможность полностью управлять приложением только посредством клавиатуры.

По умолчанию библиотека Dijit обеспечивает поддержку обеих групп пользователей. Необходимость обеспечения доступности для пользователей с ослабленным зрением определяется по факту работы броузера и операционной системы в режиме повышенной контрастности и по запрету на показ изображений в Internet Explorer или Firefox.[2] Когда обнаруживается любой из признаков, свидетельствующих о необходимости обеспечения доступности, диджиты начинают отображаться с соответствующими стилями, изображениями и шаблонами.

Например, на рис. 11.2 показано, как отображается `dijit.ProgressBar` в высококонтрастном и обычном режимах.

Несмотря на то что некоторые особенности могут быть достаточно утомительны в реализации, тем не менее железное правило, которому необходимо следовать при создании виджетов, доступных для слабовидящих, заключается в том, чтобы не использовать изображения (фоновые изображения CSS или изображения, которые размещаются с помощью тега `IMG`) таким образом, чтобы их отсутствие могло отрицательно сказаться на функциональных возможностях страницы. Из этого правила следует, что необходимо добавлять в теги с изображе-

---

[1] Дискреционные расходы – расходы сверх жизненно необходимых. – *Прим. лит. ред.*

[2] Работу в режиме повышенной контрастности легко определить в Internet Explorer для Windows, но не так легко в Mac и в других броузерах. К сожалению, не все платформы или броузеры поддерживают аспекты доступности в одинаковой степени (если вообще поддерживают), поэтому ваш опыт может быть несколько иным.

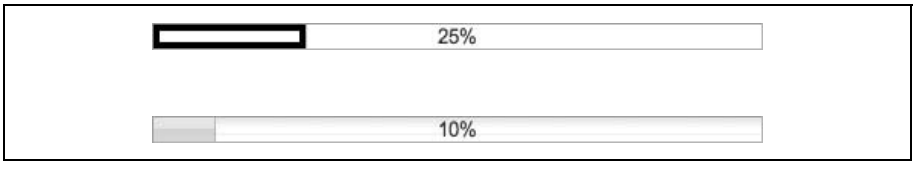



***Рис. 11.2.*** *Вверху: автоматически выбранный режим отображения dijit.ProgressBar при наличии условий, свидетельствующих о необходимости обеспечения доступности; внизу: стандартный режим отображения dijit.ProgressBar*

ниями атрибут `alt` с описанием. Такой прием может показаться простым «захламлением» страницы ненужной информацией, но зачастую он позволяет страницам делать свою работу.

Основные виджеты обеспечивают полноценную поддержку клавиатуры с помощью стандартного атрибута `tabIndex`, который управляет перемещением фокуса ввода в приложении. Кроме того, внутренние механизмы явно управляют фокусом ввода в сложных элементах управления, чтобы всплывающие подсказки могли отображаться без ошибок, которые иногда возможны при использовании устройств чтения с экрана.[1]

## WAI-ARIA

Инициативы по обеспечению доступности для пользователей с ослабленным зрением и для тех, кто не имеет возможности пользоваться мышью, продолжают расширяться, но в эпоху активных Интернет-приложений необходимо внедрение дополнительной поддержки. Типичными примерами такой дополнительной поддержки являются обеспечение пользователей информацией об изменении состояния объекта XHR, который позволяет не выполнять полную перезагрузку страницы, и адекватная обработка кнопки «Назад» броузера для некоторых действий.

Инициатива консорциума W3C по обеспечению доступности активных Интернет-приложений (Web Accessibility Initiative for Accessible Rich Internet Applications, WAI-ARIA) – это попытка определить рекомендации, которым должны следовать приложения, использующие технологии AJAX, для имитации функциональности обычных настольных приложений, чтобы обеспечить их доступность для лиц с ограниченными возможностями. В начале 1990 годов устройства чтения с экрана могли обеспечить чтение только страниц с привычной разметкой

[1] *Устройство чтения с экрана* – это вспомогательное устройство, которое озвучивает факт перемещения фокуса ввода и корректные варианты действий, которые могут быть выполнены в активном элементе управления.

HTML. Однако в настоящее время виджеты располагаются внутри большого количества вложенных элементов `DIV` и управляются технологиями AJAX, которые не подразумевают взаимодействие с устройством чтения с экрана. Рекомендации WAI-ARIA оговаривают семантику, соблюдение которой необходимо для эффективной передачи информации людям с ослабленным зрением. Например, следуя этой семантике, приложение может информировать устройство чтения с экрана, что определенный набор вложенных элементов `DIV` представляет собой древовидную структуру, некоторый узел дерева в настоящий момент обладает фокусом ввода и нажатие клавиши Tab вызовет перемещение фокуса к «следующему» элементу.

В библиотеке Dijit имеется набор функций, реализующих поддержку рекомендаций WAI-ARIA и предназначенных для обеспечения доступности виджетов. Рабочий проект консорциума W3C «Roadmap for Accessible Rich Internet Applications» (план по развитию мер обеспечения доступности активных интернет-приложений) (*http://www.w3.org/TR/ariaroadmap/*) представляет собой прекрасную отправную точку в изучении ARIA и всей инициативы по обеспечению доступности (Web Accessibility Initiative) в целом. Конкретное описание ролей приводится в документе «Roles for Accessible Rich Internet Applications» (роли активных интернет-приложений) (*http://www.w3.org/TR/aria-role/*), а описание модуля состояний и атрибутов обеспечения доступности активных интернет-приложений приводится в документе «States and Properties Module for Accessible Rich Internet Applications» (*http://www.w3.org/TR/aria-state/*).

Перечень функций, реализующих поддержку WAI, приводится в табл. 11.1.

*Таблица 11.1. Функции поддержки рекомендаций WAI*

| Функция | Комментарий |
|---|---|
| `onload()` | Вызывается автоматически, чтобы определить, работает ли страница в режиме высококонтрастного отображения и запрещен ли показ изображений. Обычно вам не придется вызывать этот метод непосредственно, потому что он вызывается автоматически после загрузки страницы. |
| `hasWaiRole(/* DOMNode */ node)` | Возвращает значение `true`, если узел имеет атрибут `role`. |
| `getWaiRole(/* DOMNode */ node)` | Возвращает атрибут `role` узла. |
| `setWaiRole(/* DOMNode */ node, /* String */ role)` | Устанавливает атрибут `role` узла. |
| `removeWaiRole(/* DOMNode */ node)` | Удаляет атрибут `role` узла. |

| Функция | Комментарий |
|---|---|
| `hasWaiState(/* DOMNode */ node, /* String */ state)` | Возвращает значение `true`, если узел находится в указанном состоянии. |
| `getWaiState(/* DOMNode */ node, /* String */ state)` | Возвращает значение атрибута состояния `state` для узла. |
| `setWaiState(/* DOMNode */ node, /* String */ state, * String */ value)` | Устанавливает значение атрибута `state` узла. |
| `removeWaiState(/* DOMNode */ node, /* String */ state)` | Удаляет атрибут `state` узла. |

В терминах WAI-ARIA атрибут `role` описывает назначение элемента управления. Примерами значений атрибута `role` могут служить `link`, `checkbox`, `toolbar` или `slider`. Атрибут `state` описывает состояние элемента управления и не обязательно может иметь только два значения. Например, элемент управления с ролью `checkbox` может иметь состояние «checked» (отмечен), которое установлено в значение `mixed` при частичном выборе. В качестве примеров других состояний можно назвать `checked` и `disabled`, каждое из которых может иметь одно из двух значений (`true/false`).

# Библиотека Dijit для дизайнеров

Основные принципы использования готовых диджитов в разметке очень просты: с помощью атрибута `dojoType` определяется тип диджита, который следует разместить на странице, а с помощью дополнительных атрибутов задаются входные данные для виджета и *точки расширения*, позволяющие переопределять существующее поведение виджета. Сам атрибут `dojoType` является обязательным, а остальные вспомогательные атрибуты обычно имеют достаточно разумные значения по умолчанию. Точно так же точки расширения всегда имеют реализацию поведения по умолчанию.

Различия между «методами» и «точками расширения» носят исключительно семантический характер: *методы* – это операции, с помощью которых прикладной программист может осуществлять непосредственное управление диджитом. *Точки расширения* – это методы, которые прикладной программист не вызывает непосредственно – они вызываются самим диджитом, когда появляются соответствующие условия. Например, виджет может иметь такой явно заданный метод, как `setValue`, который можно настраивать программно, а также такой метод, как `onKeyUp`, который представляет собой точку расширения и вызывается автоматически при каждом нажатии клавиши.

Существует несколько атрибутов, перечисленных в табл. 11.2, знать о существовании которых при использовании виджетов особенно важно, потому что эти атрибуты устанавливаются непосредственно в элементе DOM виджета. Эти атрибуты максимально гарантируют возможность настройки и «правильного» поведения диджита в разметке HTML.

## Несколько слов о проверке правильности объявления DOCTYPE

С технической точки зрения вполне возможно включать диджиты в страницу, чтобы при этом страница соответствовала требованиям HTML 4.01 Strict DOCTYPE, но только если диджиты создаются программным способом, а их шаблоны соответствуют спецификации. При добавлении в разметку атрибута `dojoType`, а также других нестандартных атрибутов страница перестает соответствовать правилам, применяемым большинством валидаторов (средств проверки), несмотря на то, что существует возможность создавать свои собственные определения типа документа (Document Type Definition, DTD), которые включают объявление нестандартных атрибутов, таких как `dojoType`. По этой причине ни один из примеров, приводимых в книге, не включает объявление DOCTYPE в начале страницы.

Соблюдение стандартов имеет важное значение, но не менее важно правильно оценивать и нарушение требований стандартов в некоторых случаях. Например, использование нестандартных атрибутов Dojo – проекта надежного, открытого, поддерживаемого сообществом; проекта, настолько прозрачного, насколько это вообще возможно, – это нечто иное, чем использование нестандартных атрибутов вследствие небрежности или невежества. Инструментарий Dojo занимает передовые позиции и находится под неусыпным наблюдением величайших в мире хакеров DHTML, поэтому в нем не могут появиться решения, которые наверняка вызовут неувязки.

При этом никогда не утверждалось, что проект Dojo полностью соответствует стандартам или какому-то одному показателю (проверка на соответствие HTML 4.01 Strict DOCTYPE является таким же показателем, как и любые другие). тем не менее инструментарий гарантирует определенную функциональность на определенном подмножестве броузеров, что само по себе, как вы уже догадались, является еще одним показателем.

*Таблица 11.2. Общие атрибуты диджитов*

| Атрибут | Тип | Комментарий |
|---|---|---|
| id | String | Уникальный идентификатор виджета. По умолчанию это значение генерируется автоматически, что гарантирует его уникальность. Если явно указано значение, которое уже используется, оно будет проигнорировано и будет сгенерировано уникальное значение. |
| lang | String | Язык, используемый при отображении виджета. По умолчанию используются настройки языка в броузере. Если желательно определить дополнительные настройки, это можно сделать с помощью параметра djConfig.extraLocale, в результате чего будет загружен дополнительный пакет. (Вообще этот атрибут не используется без явной необходимости наличия нескольких языков на одной странице.) |
| dir | String | Поддержка двунаправленного письма в соответствии с атрибутом DIR языка разметки HTML. По умолчанию этот атрибут имеет значение ltr (left to right – слева направо), если явно не указано значение rtl (right to left – справа налево). Любые другие значения являются недопустимыми. |
| style | String | Атрибут style языка разметки HTML, который должен передаваться самому внешнему узлу DOM виджета. По умолчанию никакие дополнительные атрибуты стиля не передаются. |
| title | String | Атрибут title языка разметки HTML, который может использоваться для отображения всплывающих подсказок при наведении указателя мыши на узел DOM диджита. |
| class | String | Информация о классе CSS, который должен применяться к самому внешнему узлу DOM. Этот атрибут удобно использовать для переопределения всех или некоторых свойств стиля заданной по умолчанию темы. |

## Темы

Темы в библиотеке Dijit – это целостная коллекция правил CSS, применяемых ко всему набору виджетов. Можно даже сказать, что виджеты из библиотеки Dijit поддерживают возможность смены скинов («шкур»), где темы играют роль используемых скинов. Если необходимо быстро взять несколько виджетов и поместить их на страницу, особенно удобно использовать темы, потому что при разработке не нужно беспокоиться о CSS. Начиная с версии 1.1 инструментарий Dojo включает три готовые и достаточно привлекательные темы:

*Tundra*

Преобладающими являются светло-серые и светло-синие тона, а название тема получила в честь ландшафта Арктики.

*Soria*

Преобладающими являются синие оттенки. Элементы управления решены в глянце и лоске стиля «Web 2.0 sheen». Гамма этой темы навеяна оттенками синего неба на фотографиях, сделанных в Испании, в провинции Soria.

*Nihilo*

Преобладающими являются белый с мягкими серыми контурами, с серовато-синим текстом. В некоторых элементах управления используются желтоватые оттенки в качестве выделяющего цвета. Ходят слухи, что побудительным мотивом к созданию этой темы стала концепция *ex nihilo* (из ничего) (создать что-то из ничего), целью которой является обеспечение минималистской элегантности – с некоторым трудом, но вы ее обнаружите.

В исходных текстах инструментария можно найти средство тестирования тем *dijit/themes/themeTester.html*, которое отображает некоторые диджиты с применением выбираемой вами темы. Рассмотреть темы на экране – лучший способ составить себе представление о них, так как черно-белые страницы книги не смогут передать вам впечатление от них.

Структура каталогов с темами следует шаблону, что приводится ниже, но при этом каждый основной файл CSS содержит инструкции `@import`, которые подключают другие файлы CSS, являющиеся частями общего дизайна (инструменты сборки объединяют файлы CSS, поэтому пользователю доставляется только один файл, что минимизирует задержки при передаче ресурсов по протоколу HTTP):

```
themes/
    tundra/
        tundra.css
        images/
            комплект статических изображений
    soria/
        soria.css
        images/
            комплект статических изображений
    nihilo/
        nihilo.css
        images/
            комплект статических изображений
<здесь, следуя тому же шаблону, могут располагаться ваши собственные темы...>
```

В примере 11.1 выделены те части страницы, которые имеют отношение к теме.

*Пример 11.1. Использование темы*

```
<html>
    <head>
```

```
        <title>Fun With the Themes!</title>

        <!-подключить тему tundra -->
        <link rel="stylesheet" type="text/css"
         href="http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css" />
        <link rel="stylesheet" type="text/css"
        href="http://o.aolcdn.com/dojo/1.1/dijit/themes/tundra/tundra.css" />

        <script
            type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js"
        ></script>

        <script type="text/javascript">
            //разместите здесь необходимые диджиты
        </script>
    <head>
    <body class="tundra">
        <!-используйте здесь свои диджиты -->
    </body>
</html>
```

Обратите внимание, насколько просто использовать тему – для этого достаточно подключить таблицу стилей и применить соответствующее имя класса к тегу BODY, хотя точно так же можно было бы применить имя класса к любому другому тегу на странице, если у вас есть веские причины варьировать стиль страницы. Темы допускается применять к любому произвольному элементу страницы, ко всей странице, к конкретному тегу DIV или к определенному виджету. В этом примере также производится подключение файла *dojo.css*, который содержит некоторый базовый стиль.

Переключение темы также выполняется очень просто – достаточно лишь заменить все ссылки на тему *tundra* ссылками на тему *soria* или *nihilo*. Проще не придумаешь.

Мы не будем углубляться в изучение тем, потому что это всего лишь система хорошо продуманных правил CSS и, несмотря на то, что темы играют важную и положительную роль в обеспечении эффективности библиотеки Dijit, они не являются необходимой частью текущего обсуждения. Однако, если вы заинтересовались вопросом использования тем, вооружитесь хорошим справочником по CSS и начните изучать содержимое различных файлов CSS. В них вы увидите такие определения, как `.tundra.dojoButton { /* здесь находится определение стиля */}`, которые достаточно легко читаются и без труда обнаруживаются в файлах шаблонов библиотеки Dijit или в страницах при помощи Firebug.

## Узлы и диджиты, события DOM и методы диджитов

Между диджитами и узлами DOM существуют важные отличия: диджит – это объект JavaScript типа Function, экземпляры которых создаются из коллекции ресурсов, включая разметку HTML, CSS, JavaScript

и такие статические ресурсы, как изображения; визуальное представление диджита вставляется в страницу путем присваивания его атрибуту domNode ссылки на узел DOM (внешнего узла в его шаблоне).

Различия между диджитом и узлом DOM можно обозначить еще сильнее, если сравнить функцию dojo.byId, которая возвращает узел DOM, заданный аргументом, с собственным методом библиотеки Dijit – dijit.byId, который возвращает диджит, ассоциированный с указанным узлом DOM. Сравнительное описание этих двух функций приводится в табл. 11.3. Выполнение двух команд в Firebug для следующего ниже определения диджита Button позволит понять отличия:

```
<button id="foo" dojoType="dijit.form.Button">click me</button>
```

*Таблица 11.3. Различия между dojo.byId и dijit.byId*

| Команда | Результат в консоли Firebug |
|---|---|
| dojo.byId("foo") | <button<br>id="foo"<br>class="dijitStretch<br>dijitButtonNode<br>dijitButtonContents"<br>waistate="labelledby-foo_label"<br>wairole="button"<br>type="button"<br>dojoattachpoint="focusNode,titleNode"<br>role="wairole:button"<br>labelledby="foo_label"<br>tabindex="0"<br>valuenow=""<br>disabled="false"> |
| dijit.byId("foo") | [Widget dijit.form.Button, foo] _connects=[4] _attaches=[0]<br>id=foo |

Функция dojo.byId возвращает узел DOM, который служит визуальным представлением экземпляра dijit.form.Button, тогда как функция dijit.byId возвращает объект JavaScript типа Function, с помощью которого можно исследовать все стандартные механизмы диджитов.

Очень часто встречается ошибка, когда программист пытается вызывать методы, используя результат, возвращаемый функцией dojo.byId. Запомните, что узлы DOM не имеют методов, свойственных диджитам.

Вследствие различий между диджитами и узлами DOM существуют различия между событиями DOM и событиями библиотеки Dijit. Несмотря на то что у многих диджитов имеется событие onClick, тем не менее это событие не имеет никакого отношения к событию onclick уз-

ла DOM, вопреки очевидной схожести имен. Найдите минуту времени, чтобы загрузить и выполнить следующую страницу в консоли Firebug – полученные результаты помогут прояснить ситуацию:

```
<html>
    <head>
        <title>Fun with Button Clicking!</title>

        <link rel="stylesheet" type="text/css"
         href="http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css" />
        <link rel="stylesheet" type="text/css"
        href="http://o.aolcdn.com/dojo/1.1/dijit/themes/tundra/tundra.css" />

        <script
            djConfig="parseOnLoad:true"
            type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js"
        ></script>

        <script type="text/javascript">
            dojo.require("dojo.parser");
            dojo.require("dijit.form.Button");
            dojo.addOnLoad(function() {
                dojo.connect(dojo.byId("foo"), "onclick", function(evt) {
                    console.log("connect fired for DOM Node onclick");
                });

                dojo.connect(dijit.byId("foo"), "onclick", function(evt) {
                   console.log("connect fired for dijit onclick"); //никогда!
                });

                dojo.connect(dijit.byId("foo"), "onClick", function(evt) {
                    console.log("connect fired for dijit onClick");
                });
            });
        </script>
    </head>
    <body class="tundra">
        <button id="foo" dojoType="dijit.form.Button" onclick="foo">click me
            <script type="dojo/method" event="onClick" args="evt">
                console.log("Button fired onClick");
            </script>
        </button>
    </body>
</html>
```

В этой странице, в разметке HTML, определяется простой метод для простого диджита `Button`, который содержит реализацию метода `onClick` и создает три подключения: одно – для события `onclick` узла DOM, и два – для событий диджита `onclick` и `onClick`. Однако у диджитов нет события `onclick`, таким образом, этот пример демонстрирует распространенную ошибку, которую очень трудно будет отыскать и исправить, заключающуюся в неудачной попытке установить соединение.

# Парсер

Парсер в инструментарии Dojo является ресурсом, входящим в состав библиотеки Core. Он является стандартным средством создания экземпляров виджетов, определенных в разметке страницы, и гарантирует, что их визуальные представления будут вставлены в страницу посредством связывания их с узлами DOM. Как только свойству domNode диджита будет присвоена ссылка на узел DOM, броузер тут же отобразит его на странице. Но, несмотря на то что узел DOM является жизненно важной составляющей диджита, поскольку позволяет ему стать видимым, тем не менее комплекс всех особенностей диджитов намного шире. В этом разделе дается введение в парсер и подробно рассказывается о том, как он работает.

## Парсинг виджетов во время загрузки страницы

Кроме некоторых упоминаний во вводном материале главы 1 и в описаниях примеров использования механизма «перетащил и бросил» в главе 7, формального представления парсера еще не было, потому что наиболее типичное его применение связано с созданием экземпляров виджетов в странице. Без лишней суеты рассмотрим штатный пример того, как парсер создает экземпляры виджетов на основе разметки страницы. Обратите внимание на выделенные строки в примере 11.2, так как они имеют прямое отношение к действиям парсера.

*Пример 11.2. Автоматический парсинг виджетов*

```
<html>
    <head>
        <title>Fun With the Parser!</title>

        <!-- добавить стандартные CSS для стилизации базовых диджитов -->

        <link rel="stylesheet" type="text/css"
         href="http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css" />
        <link rel="stylesheet" type="text/css"
        href="http://o.aolcdn.com/dojo/1.1/dijit/themes/tundra/tundra.css" />

        <script
            type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js"
            djConfig="parseOnLoad:true"
        ></script>

        <script type="text/javascript">
            dojo.require("dojo.parser");
            dojo.require("dijit.form.Button");
        </script>
    </head>

    <body class="tundra">
        <button dojoType="dijit.form.Button">Sign Up!</button>
```

```
    </body>
</html>
```

В этом примере создается простая страница, содержащая кнопку из библиотеки Dijit, которая, кроме как красуется на странице, ничего больше не делает. Но для целей представления парсера без погружения в специфику библиотеки Dijit этого вполне достаточно. Единственное, что необходимо знать о диджите `Button` прямо сейчас, – это то, что он извлекается с помощью вызова `dojo.require` и вставляется в страницу благодаря атрибуту `dojoType`.

Любые действия, которые вы определяете в вызове функции `addOnLoad`, выполняются уже после парсинга виджетов, делая безопасными любые попытки сослаться на них.

В примере 11.2 вы не увидите прямых обращений к парсеру – это сделано преднамеренно. В большинстве случаев вам достаточно будет предусмотреть загрузку диджита с помощью функции `dojo.require`, установить флаг `parseOnLoad` в массиве `djConfig`, и все остальное будет сделано автоматически. В действительности это все, что происходит в этом примере. Просто задумайтесь на минутку, насколько изящно такое решение, когда можно просто взять готовый диджит и несколькими нажатиями клавиш вставить его в страницу. Никакой мороки, никаких проблем и никаких хлопот.

## Парсинг виджетов вручную

Иногда возникают ситуации, когда необходимо выполнять парсинг страницы или отдельных узлов DOM вручную. К счастью, выполнить это можно за счет единственного вызова функции. Взгляните на пример 11.3, где выполняется парсинг виджета на странице вручную.

*Пример 11.3. Парсинг страницы вручную*

```
<html>
    <head>
        <title>Hello Parser</title>

        <!-- добавить стандартные CSS для стилизации базовых диджитов -->

        <link rel="stylesheet" type="text/css"
         href="http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css" />
        <link rel="stylesheet" type="text/css"
        href="http://o.aolcdn.com/dojo/1.1/dijit/themes/tundra/tundra.css" />

        <script
            type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js"
            djConfig="parseOnLoad: false"
        ></script>

        <script type="text/javascript">
```

```
            dojo.require("dojo.parser");
            dojo.require("dijit.form.Button");
            dojo.addOnLoad(function() {
                dojo.parser.parse(); //выполнение парсинга вручную после
                                     //загрузки страницы
            });
        </script>
    <head/>
    <body class="tundra" >
        <button dojoType="dijit.form.Button" >Sign Up!</button>
    </body>
</html>
```

Хотя и полезно выполнить парсинг всей страницы, тем не менее чаще вам будет необходимо вручную выполнять парсинг отдельных узлов DOM. Функция `dojo.parser.parse` может принимать необязательный аргумент – корневой узел DOM, откуда следует начинать сканирование в поисках атрибутов `dojoType` и создавать виджеты. То есть функции `parse` передается родительский узел элемента, для которого требуется выполнить парсинг. Ниже приводится один из возможных вариантов фрагмента предыдущего программного кода, иллюстрирующий такую возможность:

```
<script type="text/javascript">
  dojo.require("dojo.parser");
  dojo.require("dijit.form.Button");
  dojo.addOnLoad(function() {
    //Парсер выполняет обход дерева DOM, начиная с указанного узла,
    //и создает виджеты. В данном случае кнопка является одним
    //из листьев этого дерева, поэтому такого вызова функции будет
    //вполне достаточно, чтобы выполнить ее парсинг.
    dojo.parser.parse(document.getElementsByTagName("button")[0].parentNode);
  });
</script>
```

Попытка выполнить парсинг виджета, передав функции `parse` узел DOM, в котором определен виджет, закончится неудачей, и при этом вы не получите никаких сообщений, свидетельствующих об ошибке. К счастью, если вы можете получить ссылку на узел, то получить ссылку на его родительский узел можно всего несколькими нажатиями клавиш, добавив ссылку на свойство `parentNode`.

## Раскрываем тайны парсера

Хотя действия парсера больше похожи на волшебство, на самом деле все сводится к строгой, хорошо продуманной автоматизации. Как вы теперь знаете, парсер имеет две основные области применения: парсинг страницы во время загрузки при включении параметра `djConfig="parseOnLoad:true"` и обеспечение возможности парсинга виджетов вруч-

ную. Данный раздел во всех подробностях описывает, что происходит в этих двух случаях.

Для выполнения парсинга во время загрузки страницы необходимо соблюсти следующие три требования:

- Включить параметр `parseOnLoad:true`, как пару ключ/значение в ассоциативный массив `djConfig`, который будет обнаружен парсером после его загрузки и использован для автоматического запуска парсинга.
- Подключить парсер посредством инструкции `dojo.require("dojo.parser")`, что обеспечит загрузку парсера и автоматический вызов функции `dojo.parser.parse()` по окончании загрузки страницы. Так как в этом случае функция вызывается без аргумента, полем деятельности парсера является все тело страницы.
- Добавить в разметку (там, где это необходимо) атрибуты `dojoType` с определениями виджетов, которые должны быть проанализированы парсером.

При выполнении парсинга вручную для виджетов, которые уже определены в разметке, необходимо выполнить похожие требования:

- Подключить парсер посредством инструкции `dojo.require("dojo.parser")`. Поскольку при использовании описываемого подхода параметр `parseOnLoad` не имеет значение `true`, автоматический вызов функции `dojo.parser.parse()` не происходит.
- Добавить в разметку атрибуты `dojoType` с определениями виджетов там, где это необходимо, – возможно, даже динамически, уже после загрузки страницы.
- Вручную вызвать функцию `dojo.parser.parse()`, при необходимости передав ей в качестве аргумента определенный узел DOM, который станет начальной точкой для выполнения операции парсинга.

Но как фактически протекает процесс парсинга? Вы уже понимаете, что отчасти суть парсинга заключается в поиске атрибутов `dojoType` и в создании виджетов из них? Напомню еще раз, что весь процесс – это простая автоматизация – при условии, что вы правильно понимаете ее. Ниже описывается, как именно происходит парсинг:

- Вызывается функция `dojo.query("[dojoType]")`, которая выбирает узлы, требующие парсинга.
- Из каждого узла извлекается информация о классе (подразумеваются классы, создаваемые функцией `dojo.declare`), выполняется обход всех атрибутов и производится приведение типов. Напомню еще раз, что в атрибутах может определяться информация для функции `constructor` класса.
- Внутри узлов отыскиваются и намечаются для обработки все теги `SCRIPT` с типом `dojo/method` или `dojo/connect`. (Подробнее об этом рассказывается в следующем разделе «Определение методов в разметке».)

- Создается экземпляр класса с помощью функции `constructor`, если не определен метод `markupFactory`, который используется в противном случае. `markupFactory` — это специальный метод, позволяющий определять собственные функции-конструкторы для виджетов, которые требуют иного способа инициализации при использовании в разметке, чем при программном способе создания. Все диджиты наследуют базовый класс `_Widget`, содержащий ряд стандартных методов управления жизненным циклом. Один из таких методов вставляет `domNode` диджита в страницу, что обеспечивает его видимость. Методы жизненного цикла подробно будут рассматриваться в следующей главе.
- Если в определении виджета присутствует атрибут `jsId`, то экземпляр класса отображается в глобальное пространство имен JavaScript. (Обычно такая возможность используется применительно к хранилищам данных и к виджетам, когда есть веские основания, чтобы сделать их глобальными переменными.)
- Выполняется обработка всех тегов `SCRIPT` с типом `dojo/connect` или `dojo/method` (подробнее об этом рассказывается далее в этой же главе), и для каждого виджета вызывается метод управления жизненным циклом `startup`. Метод `startup` — это еще один стандартный метод управления жизненным циклом, унаследованный от класса `_Widget` (описывается в следующей главе), позволяющий манипулировать любыми виджетами, содержащимися в создаваемом виджете.

Надеюсь, эти сведения не заставили вас пережить те же чувства, которые появились, когда вы узнали, что Дед Мороз ненастоящий, но когда-то же вы должны были это узнать. В следующей главе будут обсуждаться исключительно вопросы, касающиеся методов управления жизненным циклом диджитов, где будет представлен полный охват этих концепций.

# Практика Dijit на примере NumberSpinner

В этом разделе даются основы практического использования диджитов на примере довольно простого диджита `dijit.form.NumberSpinner` с целью подготовить вас к знакомству со следующими главами. Для начала мы рассмотрим создание диджита в коде разметки, а затем изучим вопрос создания его программным способом.

## Создание в разметке

Как вы узнали из предыдущего раздела, рассказывавшего о парсере, добавление диджита в страницу выполняется достаточно просто. Вы подключаете необходимые ресурсы, добавляете атрибут `dojoType` в тег и обеспечиваете автоматический запуск парсера. Для полноты картины в примере 11.4 показано, как, следуя этому шаблону, создать диджит `NumberSpinner`.

*Пример 11.4. Создание виджета NumberSpinner в разметке*

```
<html>
    <head>
        <title>Number Spinner Fun!</title>

        <link rel="stylesheet" type="text/css"
         href="http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css" />
        <link rel="stylesheet" type="text/css"
        href="http://o.aolcdn.com/dojo/1.1/dijit/themes/tundra/tundra.css" />

        <script
            type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js"
            djConfig="parseOnLoad: true"
        ></script>

        <script type="text/javascript">
            dojo.require("dojo.parser");
            dojo.require("dijit.form.NumberSpinner");
        </script>
    </head>
    <body class="tundra">
        <form> <!-- в некотором смысле действительно удивительная форма -->
            <input dojoType="dijit.form.NumberSpinner"
                   constraints="{min:0,max:10000}" value=1000>
            </input>
        </form>
    </body>
</html>
```

## Создание программным способом

Чаще всего вам придется создавать диджиты в разметке, но создание их программным способом выполняется ничуть не сложнее, и процесс этот мало чем отличается от создания любого другого объекта типа Function, потому что диджит является именно таким объектом – объектом типа Function. В общем случае конструктор диджита принимает два аргумента. Первый – это объект, содержащий свойства, значения которых требуется передать диджиту, причем это те самые свойства, которые определяются в теге при создании диджита в разметке. Второй аргумент – это исходный узел или значение атрибута `id` исходного узла, который будет использоваться диджитом для отображения:

```
var d = new module.DijitName(/*Object*/props, /*DOMNode|String*/node)
```

Пример 11.5 демонстрирует создание `NumberSpinner` программным способом и воспроизводит тот же эффект, что и пример 11.4.

*Пример 11.5. Создание виджета NumberSpinner программным способом*

```
<html>
    <head>
```

```
        <title>Number Spinner Fun!</title>

        <link rel="stylesheet" type="text/css"
         href="http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css" />
        <link rel="stylesheet" type="text/css"
        href="http://o.aolcdn.com/dojo/1.1/dijit/themes/tundra/tundra.css" />

        <script
            type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js"
        ></script>

        <script type="text/javascript">
            dojo.require("dijit.form.NumberSpinner");
            dojo.addOnLoad(function() {
                var ns = new dijit.form.NumberSpinner(
                    //свойства
                    {
                        constraints : {min:0,max:10000},
                        value : 1000
                    },
                    "foo" //значение атрибута id узла
                );

                // выполнить здесь какие-либо другие действия...
            });
        </script>
    <head>
    <body class="tundra">
        <form>
            <input id="foo"></input>
        </form>
    </body>
</html>
```

## Приятные мелочи

Этот пример выводит элемент управления с небольшим текстовым полем, который позволяет изменять значение либо с помощью клавиш управления курсором на клавиатуре, либо щелчками мыши на элементах управления, либо путем прямого ввода значения с клавиатуры. В любом случае значения `min` и `max`, являющиеся частью *атрибута* `constraints`, могут быть настроены для определения минимального и максимального значений, которые нельзя будет преодолеть, нажимая клавиши управления курсором или щелкая мышью на элементах управления. Значение `value` содержит значение по умолчанию, как и в обычном элементе HTML. Однако при вводе значений вручную все же можно выйти за диапазон допустимых значений, что может вызвать появление сообщения об ошибке в виде всплывающей подсказки. Как это происходит, показано на рис. 11.3.

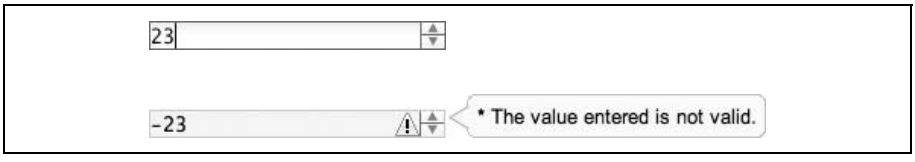



***Рис. 11.3.*** *Вверху: изменение значения в диджите NumberSpinner производится с помощью клавиш управления курсором или с помощью мыши; внизу: при вводе с клавиатуры значения, которое выходит за допустимый диапазон, появляется сообщение об ошибке*

Не забывайте, что Dojo старается лишь дополнять существующие механизмы разработки веб-приложений, не переопределяя их. То есть типичные атрибуты элементов форм, такие как `value` в элементе `input`, использованном в предыдущем примере, по-прежнему будут работать так, как это ожидается.

При определении в разметке пар ключ/значение в виде литерала объекта `djConfig` мы не используем фигурные скобки, например: `djConfig="parseOnLoad:true,isDebug:true"`, но это скорее исключение, чем правило. Библиотека Dijit требует, чтобы атрибуты объектов в разметке записывались с использованием фигурных скобок, например: `constraints="{min:0, max:100}"`.

Вероятно, вы уже связали возможность использования клавиатуры при работе с виджетом `NumberSpinner` с обеспечением доступности, однако существуют и другие приятные мелочи, на которые стоит взглянуть. Вне всяких сомнений, вы уже обратили внимание на форматирование, которое автоматически применяется к числам больше 999. Особенно примечательно, что при отображении страницы с разными региональными настройками будут использоваться разные разделители разрядов, причем это происходит автоматически: в случае использования региональных настроек с кодом *en-us* в качестве разделителя разрядов используется запятая, например 1,000. Для Испании, код *es-es*, в качестве разделителя разрядов используется точка, например 1.000. Применение различных разделителей показано на рис. 11.4. Попробуйте реализовать то же самое, изменив региональные настройки по умолчанию с помощью `djConfig`. Например, чтобы установить региональные настройки для Испании, можно использовать такое объявление:

```
djConfig="locale:'es-es'"
```

Не забывайте, что любые строковые значения в `djConfig` должны окружаться дополнительными кавычками. Синтаксис объявления встроенного ассоциативного массива способствует тому, чтобы быстро позабыть это правило, и, к сожалению, сообщения об ошибках, которые являются результатом такой забывчивости, порою могут направить поиски ошибки по ложному пути. Любые настройки в `djConfig` должны загружаться до начала самонастройки библиотеки Base.

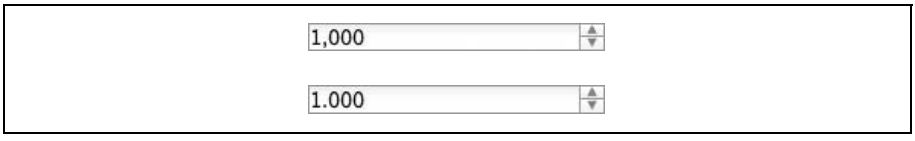



***Рис. 11.4.*** *Диджиты изначально поддерживают специальные возможности форматирования с учетом региональных настроек, не требуя выполнения дополнительной настройки; вверху NumberSpinner был настроен для региона с кодом en-us, а внизу – для региона с кодом es-es; все необходимые действия по форматированию, связанные с региональными настройками, реализованы внутри виджета*

Еще одна важная встроенная особенность, которую поддерживает библиотека Dijit, – это возможность *автоповтора*, то есть диджиты особым способом реагируют на нажатие и удерживание клавиши на клавиатуре или кнопки мыши. Если вы попытаетесь нажать кнопку мыши, когда указатель находится на одном из элементов управления диджита `NumberSpinner`, вы заметите, что значения сначала изменяются достаточно медленно, примерно в пределах первых 10 изменений, а потом скорость изменения начинает постепенно увеличиваться. Неудивительно, что работа с клавиатурой реализована точно таким же способом. Кроме того, клавиши Page Up и Page Down по умолчанию изменяют значение сразу на 10 единиц.

## Определение методов в разметке

В дополнение к возможности создавать методы для управления и расширения виджетов непосредственно в программном коде JavaScript Dojo также обеспечивает возможность определять программный код JavaScript непосредственно в разметке – с помощью специального атрибута `type="dojo/method"` в тегах `SCRIPT`. Это может быть очень удобно для дизайнеров, а также для всех тех, кто хочет быстро опробовать новую идею. В возможности определять методы непосредственно в разметке есть одна примечательная особенность, которая заключается в том, что ключевое слово `this` ссылается на сам виджет, благодаря этому у вас имеется непосредственный доступ к требуемому контексту.

Рассмотрим следующий дополненный пример, где выполняется определение методов в разметке:

```
<!-- обрезано -->
        <script type="text/javascript">
            dojo.require("dojo.parser");
            dojo.require("dijit.form.NumberSpinner");
            dojo.require("dijit.form.Button");
        </script>
    </head>
    <body class="tundra">
```

```
        <form>
            <div dojoType="dijit.form.NumberSpinner" jsId="mySpinner"
                constraints="{min:0,max:10000}" value=1000>
                <script type="dojo/method">
                    dojo.mixin(this, {
                        reset : function() { this.setValue(1000); }
                    });
                </script>
            </div>
        </form>
        <button dojoType="dijit.form.Button" onClick="mySpinner.reset()">
            Reset
        </button>
    </body>
</html>
```

В результате изменений атрибут jsId присвоил виджет NumberSpinner глобальной переменной mySpinner, которая используется в методе onClick кнопки. Фактическое тело метода reset реализовано в виде специализированного тега script внутри диджита. Тег script с типом dojo/method, в котором реализована анонимная функция, выполняется после запуска функции constructor, поэтому будут доступны любые значения, переданные в виде атрибутов в разметке.

Кроме того, обратите внимание, что в предыдущем примере для создания виджета использовался элемент input, а в новом примере – тег div. Причина, по которой тег input не может использоваться в данном случае, заключается в том, что он не обладает свойством innerHTML. Чтобы иметь возможность добавить определение метода в разметку, элемент должен обладать свойством innerHTML. Если вы зададитесь вопросом, почему тег div не использовался и раньше, замечу, что это восходит к следующей исходной задаче: возможности иметь семантически правильную страницу, способную выполнять свои функции без участия JavaScript. Проще говоря, предыдущая форма, где используется элемент input, будет содержать семантически корректный элемент ввода, даже если поддержка JavaScript в броузере окажется отключенной (по тем или иным причинам), тогда как страница с виджетом в теге div – нет. В большинстве случаев это не является проблемой, но, создавая страницы, допускающие деградацию функциональных возможностей, очень важно знать основы языка разметки HTML и иметь представление о подобных подводных камнях.

Теги script с типом dojo/method и dojo/connect не могут использоваться внутри элементов разметки, не имеющих свойства innerHTML. Это не причуды Dojo, такова спецификация HTML. Хотя в представленном примере это и не показано, но тег SCRIPT, содержащий атрибут type="dojo/connect", позволяет устанавливать соединения с обработчиками событий в разметке, используя аналогичный подход.

Хотя дополнительная кнопка сброса может оказаться отличным дополнением для организации управления с помощью мыши, заметьте, что нажатие клавиши Escape на клавиатуре точно так же сбрасывает значение счетчика в исходное состояние, причем без всякой дополнительной работы с нашей стороны.

Ниже приводится измененный вариант тега `script` с типом `dojo/method`, который производит тот же эффект, но за счет меньшего объема программного кода:

```
<script type="dojo/method" event="reset">
    this.setValue(1000);
</script>
```

Вместо однократного вызываемого автоматически выполнения сразу после конструктора, который реализуется в случае с анонимной функцией в теге `script` с типом `dojo/method`, такой подход фактически выполняет создание метода `reset`, который затем присоединяется к виджету. Если бы этот метод требовал наличия входных аргументов, в разметке можно было бы определить дополнительный атрибут `args`. Например, атрибут `args="foo,bar,baz"` позволил бы передачу трех именованных параметров методу, определенному в разметке.

# Обзор базовых диджитов

В коллекции виджетов Dojo легко потеряться, поскольку она чрезвычайно обширна. В этом разделе приводится краткая опись диджитов, чтобы ознакомить вас с тем, что имеется в вашем распоряжении.

## Диджиты форм

Название категории «диджиты форм» подразумевает, что диджиты из этой категории предназначены для использования внутри форм. Хотя это, конечно, верно, но диджиты форм могут кроме того использоваться за пределами форм или внутри специального диджита `dijit.form.Form`, который предоставляет некоторые дополнительные методы и точки расширения. Ниже приводится краткий обзор диджитов, включенных в эту категорию. Не забывайте, что все диджиты совместимы с требованиями к обеспечению доступности и легко интернационализируются при необходимости.

По адресу *http://archive.dojotoolkit.org/nightly/* вы найдете тестовые страницы Dijit, которые содержат все описываемые диджиты. Это отличный способ получить представление о широте и глубине коллекции.

`Form`

Специализированный контейнер для виджетов форм, который предоставляет удобные методы и точки расширения для сериализации

данных в формате JSON, проверки правильности содержимого формы, установки сразу всех значений формы и обработки события отправки формы.

*Различные разновидности* `Button`

Встроенная замена обычных кнопок, созданных на базе элемента `BUTTON`, а также других элементов управления, подобных кнопкам, на базе элементов `INPUT`, таких как флажки и радиокнопки. Из дополнительных разновидностей кнопок присутствуют кнопки с меню, содержащие список значений (своего рода раскрывающиеся списки), которые часто можно увидеть на панелях инструментов, и кнопки переключения режима, такие как кнопки `bold` или `italic` в текстовых процессорах.

`ComboBox`

Соединяет в себе функциональности обычного раскрывающегося списка `SELECT` и текстового поля ввода, определяемого с помощью элемента `INPUT`, позволяя пользователям выбирать предопределенные значения из списка или вводить свои значения.

`FilteringSelect`

Используется в качестве замены обычному элементу `SELECT`. Может заполняться динамически, что делает его незаменимым в ситуациях, когда количество предлагаемых на выбор значений может быть очень большим.

`NumberSpinner`

Напоминает текстовое поле ввода на базе элемента `INPUT`, за исключением того, что этот элемент управления позволяет производить пошаговое изменение значения.

`Slider`

Перемещаемый бегунок, соединенный с линейкой, которая может располагаться вертикально или горизонтально. Этот виджет обеспечивает более интерактивный способ настройки значения и часто используется для изменения размеров двухмерных объектов в режиме реального времени.

`Textarea`

Используется в качестве замены обычному элементу `TEXTAREA`, но, в отличие от последнего, изменяет свои размеры по мере необходимости, в зависимости от объема содержимого, благодаря чему можно предотвратить потерю ценного экранного пространства, когда объем содержимого трудно предсказать заранее или когда требуется вывести только аннотацию.

`SimpleTextarea`

Используется в качестве замены обычному элементу `TEXTAREA`. Обладает некоторыми дополнительными механизмами взаимодействия с контейнерным диджитом `Form` и с диджитами размещения.

`MultiSelect`

Используется в качестве замены обычному элементу SELECT с атрибутом `multiple=true`. Так же, как и SimpleTextarea, обладает некоторыми дополнительными механизмами взаимодействия с диджитом `Form`.

*Различные разновидности* `TextBox`

Целое семейство многофункциональных виджетов на базе элемента `INPUT` с особым упором на возможность организации проверки значений и форматирования таких данных, как даты, время, денежные суммы, числа и т. д. Это семейство обладает невероятным объемом функциональных возможностей.

## Диджиты размещения

Традиционно для реализации сложных схем размещения элементов привлекались каскадные таблицы стилей (CSS). Несмотря на то что работа с CSS не отличается высокой сложностью, тем не менее создание, тестирование в различных броузерах и дальнейшее сопровождение такого кода требует существенных усилий, особенно если вы не являетесь искушенным знатоком CSS. Диджиты размещения позволяют создавать схемы расположения элементов прямо в разметке, причем без необходимости использовать вложенные таблицы, что делает их реализацию гораздо более простым делом. Диджиты размещения могут вкладываться друг в друга произвольными способами, что позволяет создавать весьма сложные схемы расположения элементов за более короткое время, чем при использовании традиционных приемов, основанных на применении CSS. Ниже приводится краткий список того, что предоставляет библиотека Dijit:

`ContentPane`

Самый основной строительный блок схемы размещения. Обеспечивает мозаичную схему размещения. Может использоваться самостоятельно, но обычно один или более диджитов `ContentPane` существуют как часть другого контейнерного виджета.

`TabContainer`

Обеспечивает уже знакомую схему размещения с вкладками. Вкладки могут располагаться вертикально или горизонтально. Простые схемы размещения с применением `TabContainer` обычно содержат комбинации диджитов `TabContainer` и `ContentPane`, хотя всегда существует возможность произвольного вложения элементов. Содержимое вкладок, которые изначально не отображаются, может быть загружено позднее.

`StackContainer`

Обеспечивает возможность отображения нескольких контейнеров, но в каждый конкретный момент времени будет виден только один

из них. Например, множество диджитов ContentPane могут содержать отдельные слайды презентации, а диджит StackContainer мог бы использоваться для компоновки их в единую презентацию. Кроме того, диджиты StackContainer очень удобны в приложениях, имеющих несколько «экранов», которые требуется отображать, не прибегая к полной перезагрузке страницы.

AccordionContainer

Отображает один сегмент из множества. Когда выбирается другой сегмент, предыдущий плавно сворачивается. Содержимое сегментов, которые изначально не отображаются, может быть загружено позднее.

BorderContainer

Обеспечивает удобный и простой способ реализовать схему размещения в стиле «заголовок» или «врезка», когда на экране имеется несколько сегментов, одни из которых занимают всю ширину или высоту определенной области, а другие – нет. Даже создание схемы размещения из пяти элементов в стиле «граница» (когда один элемент располагается в центре, а четыре других по границам) не представляет никаких сложностей.

## Диджиты приложения

Диджиты приложения – это диджиты из категории «прочие». Все они являются типичными элементами любого приложения, которое только начинает приближаться к функциональности активных Интернет-приложений. Меню, панели инструментов, диалоги и текстовые редакторы – все они входят в эту категорию. Эти диджиты настолько удобны, что вы не сможете избежать соблазна использовать их:

Menu

Реализует контекстное меню, подобное тому, которое часто можно увидеть в обычных приложениях, щелкнув правой кнопкой мыши. Диджит Menu нередко используется для создания сложных кнопок, таких как ComboButton и DropDownButton, для обеспечения дополнительных функциональных возможностей.

Toolbar

Контейнер для размещения таких сложных кнопок, как ToggleButton, являющихся элементами управления, которые находятся на панели инструментов. Однако на панели инструментов можно располагать любые другие кнопки из коллекции диджитов.

Dialog

Имитирует обычное диалоговое окно, полностью исключающее возможность взаимодействия с содержимым, находящимся «под» диалогом. В большинстве случаев диджиты диалогов являются фантастической и простой в сопровождении альтернативой всплывающим

окнам, особенно когда требуется производить манипулирование деревом DOM или реализовать взаимодействие между несколькими окнами.[1]

`TooltipDialog`

Комбинация диджитов `Tooltip` и `Dialog`, которая позволяет во всплывающей подсказке реализовать возможность ввода, как в диалоге. Главное различие между диджитами `Dialog` и `TooltipDialog` заключается в том, что `TooltipDialog` можно закрыть, щелкнув на любом месте за его пределами, тогда как `Dialog` предотвращает возможность взаимодействия с остальной частью страницы, пока диджит не будет закрыт явно.

`ProgressBar`

Моделирует обычный индикатор хода процесса, который часто можно увидеть в обычных приложениях. Диджиты `ProgressBar` обеспечивают стандартный способ организации обратной связи при выполнении длительных операций или при выполнении асинхронных запросов к серверу, длительность которых превышает несколько секунд. Диджиты `ProgressBar` могут быть определенными, показывая процент выполнения, или неопределенными, воспроизводя некоторый анимационный эффект, чтобы показать, что процесс продолжается.

`TitlePane`

Обеспечивает возможность отображения панели с информацией, снабженной областью заголовка вверху. Область содержимого может сворачиваться или разворачиваться щелчком мыши на заголовке, но сама область заголовка видна всегда.

`Tooltip`

Намного более гибкая альтернатива обычному атрибуту `title` в обычных элементах управления HTML. Время отображения и разметка HTML, отображаемая в качестве текста всплывающей подсказки, могут выбираться произвольно.

`InlineEditBox`

Своего рода обертка вокруг виджета, отображающая значение виджета и по своему виду напоминающая метку; однако, стоит только щелкнуть на тексте, как виджет трансформируется в поле редактирования. (Очень интересная функциональность.)

`ColorPalette`

По умолчанию отображает матрицу 3H4 или 7H10 наиболее часто используемых цветов и позволяет пользователю сделать свой вы-

[1] В некоторых броузерах манипулирование элементами DOM из другого окна просто невозможно в силу ограничений системы безопасности.

бор. Виджет `ColorPalette` допускает возможность расширения с целью отображения произвольных цветовых конфигураций.

`Editor`

Обеспечивает функциональность, минимально необходимую для редактора с поддержкой форматирования. Виджет укомплектован панелью инструментов с уже установленными на ней кнопками для выполнения таких операций, как вырезать/копировать/вставить, отменить/вернуть, выровнять текст кнопками выбора основных режимов оформления текста, такими как жирный/курсив/перечеркнутый, и кнопкой создания маркированного списка. Панель инструментов позволяет настраивать ее под конкретные нужды. Впечатляющий объем возможностей упакован в этот диджит, и при этом он намного более легкий, чем можно было бы подумать, так как диджит `Editor` построен на основе определенных «родных» элементов управления, таких как текстовый редактор Midas в Firefox.

`Tree`

Реализует дерево с узлами, которые могут произвольным образом вкладываться друг в друга, а также сворачиваться и разворачиваться, как потребуется. Этот визуальный элемент управления обычно используется для представления длинных, иерархических списков информации. Содержимое узлов, которые не развернуты по умолчанию, может быть загружено позже – для получения данных этот диджит использует потрясающий прикладной интерфейс `dojo.data`.

## Функции прикладного интерфейса библиотеки Dijit

Функции, перечисленные в табл. 11.4, используются слишком часто, чтобы можно было умолчать о них. Они входят в базовую часть библиотеки Dijit и загружаются всякий раз, когда страница подключает ресурсы Dijit. Их можно загрузить, выполнив инструкцию `dojo.require("dijit.dijit")`, так как они включены в стандартную конфигурацию сборки, о которой подробнее рассказывается в главе 16.

Полное описание прикладного интерфейса можно найти в электронной документации проекта Dojo по адресу: *http://api.dojotoolkit.org*.

Безусловно, это не все методы прикладного интерфейса библиотеки, о которых следует знать, – это перечень лишь наиболее часто используемых функций, и они позволят вам сэкономить массу времени, если вы не забудете об их существовании.

*Таблица 11.4. Часто используемые функции из библиотеки Dijit*

| Функция/член | Комментарий |
|---|---|
| `dijit.registry` | Реестр содержит полный перечень всех диджитов на странице и может использоваться, чтобы явно обойти их, проверить существование определенных диджитов и т. д. Например, можно было бы организовать единообразное управление всеми диджитами на странице с помощью функции `dijit.registry.forEach` или выяснить наличие на странице определенного виджета с помощью функции `dijit.registry.byClass` (где под «классом» понимается класс в объектно-ориентированном смысле). |
| `dijit.byNode(/* DOM Node */ node)` | По заданному узлу DOM возвращает диджит, который представляется этим узлом. |
| `dijit.getEnclosingWidget(/* DOM Node */ node)` | По заданному узлу возвращает диджит, дерево DOM которого содержит этот узел. Этот метод особенно удобен в ситуациях, когда во время обработки события DOM необходимо быстро отыскать диджит. Например, этот метод можно использовать для поиска диджита по значению свойства `target` объекта события, который передается обработчику щелчка мыши, когда пользователь щелкает на некоторой части диджита. |
| `dijit.getViewport()` | Возвращает размеры и позиции прокрутки видимой области в окне броузера – чрезвычайно удобно для реализации программного размещения объектов на экране, когда разрешение экрана и его размеры не могут быть известны заранее. Часто используется при реализации анимационных эффектов. |
| `dijit.byId(/* String */ id)` | Отыскивает диджит на странице по значению атрибута `id`, включенного в тег с атрибутом `dojoType` или присвоенного при программном способе создания. Эта функция отличается от `dojo.byId` тем, что `dojo.byId` возвращает узел DOM, а эта функция возвращает сам диджит (объект Function). |

# В заключение

После прочтения этой главы вы должны понимать:

- Философию, лежащую в основе дизайна библиотеки Dijit
- Важность обеспечения независимости и самодостаточности при разработке сложных приложений и способы использования этих концепций библиотекой Dijit для инкапсуляции функциональных возможностей в диджиты
- Важность обеспечения доступности приложений, инициативы консорциума W3C по обеспечению доступности активных Интернет-приложений (W3C Web Accessibility Initiative for Accessible Rich Internet Applications) и основные решения, используемые библиотекой Dijit
- Что определения диджитов можно вставлять непосредственно в текст разметки так, что они смогут обеспечить те же самые функциональные возможности, как если бы они создавались программным способом, а также как применять теги `SCRIPT` с типами `dojo/method` и `dojo/connect` в разметке при определении диджитов
- Различия между узлом DOM и диджитом; различия между `dojo.byId` и `dijit.byId`; различия между событиями DOM и событиями диджитов
- Основные этапы работы парсера при создании экземпляра диджита, определенного в разметке
- Основное архитектурное деление диджитов на такие категории, как диджиты форм, диджиты размещения и диджиты общего назначения уровня приложения, а кроме того, как отыскать определенный тип диджита

В следующей главе мы рассмотрим внутреннее устройство библиотеки Dijit и жизненный цикл диджитов.

# 12

# Анатомия Dijit и жизненный цикл

Подобно объектно-ориентированным концепциям в парадигмах других языков программирования, виджеты Dojo – диджиты – поддерживают типичные события жизненного цикла, такие как создание и разрушение, конструируются в соответствии с особенностями внутреннего устройства и описываются с помощью специализированного словаря. В этой главе будет представлен обзор этих тем и будут обсуждаться фундаментальные основы дизайна диджитов.

## Анатомия библиотеки Dijit

Вы уже знаете, что термин «диджит» – это сокращение от «Dojo виджет», тем не менее будет полезно внести некоторые уточнения, прежде чем продолжить их изучение. Если быть более точным, диджит – это любой класс Dojo, наследующий базовый класс из библиотеки Dijit – `_Widget`. Этот класс входит в основной модуль `dijit` инструментального набора, поэтому полное имя класса будет выглядеть так: `dijit._Widget`. В библиотеке Dijit существует еще несколько основополагающих классов, с которыми вы вскоре познакомитесь, но следует помнить, что для любого диджита исходным предком является класс `_Widget`.

Как уже говорилось в главе 10, функция `dojo.declare` избавляет вас от необходимости написания большого объема шаблонного программного кода; диджиты следуют ее примеру и прячут основную сложность своей реализации в классах, таких как `_Widget`. Ниже вы увидите, что в этом классе существует множество методов-заглушек, которые можно переопределять для достижения требуемого поведения, избежав необходимости разрабатывать свой собственный шаблон.

У вас может возникнуть вопрос, почему имя класса `_Widget` начинается с символа подчеркивания. Когда дело касается диджитов, начальный символ подчеркивания практически всегда

означает, что не следует создавать экземпляры такого класса. Обычно такие классы используются только как суперклассы в дереве наследования.

Давайте начнем обсуждение диджитов со знакомых конструкций, посредством которых производится определение диджитов на физическом уровне – HTML, CSS, JavaScript и других статических ресурсов, с которыми вам уже приходилось иметь дело. После этого, обретя устойчивость, мы погрузимся в изучение модели жизненного цикла диджитов, опираясь на знание функции `dojo.declare`, и рассмотрим несколько все более усложняющихся примеров программного кода, связанных с созданием собственных диджитов.

## Обзор принципов разработки веб-приложений

Любой, кому приходилось иметь дело с компьютерами, знает, что HTML де-факто является стандартом отображения информации в веб-броузере. С помощью HTML вы можете стандартизовать заголовки, параграфы, разделы, поля форм и вообще создать любую разметку для отображения текстовой информации и изображений. Однако, применение только одного языка разметки HTML не способно воспроизводить страницы, приятные для глаз, так как отсутствует возможность дополнительного оформления, а содержимое остается статическим. Общая структура такой страницы не отличается сложностью, и она не изменяется в ответ на действия пользователя. Учитывая достижения, появившиеся за последние годы, можно сказать, что если бы Всемирная паутина использовала только язык разметки HTML, она была бы невыносимо скучной.

Стоит добавить CSS, и положение дел существенно меняется. Эстетическая привлекательность страниц сразу изменяется в лучшую сторону. Если язык разметки HTML обеспечивает возможность наполнения страниц информацией, но не слишком заботится о визуальном отображении, то каскадные таблицы стилей CSS добавляют возможностей в настройке расположения элементов на странице и улучшают визуальное оформление. Но добавление одних только таблиц стилей не изменяют статическую природу страниц, что оставляет неудовлетворенным свойственное человеку стремление к динамизму взаимодействий, жажду иметь нечто более живое, более подвижное. С помощью HTML и CSS можно создавать прекрасно оформленные страницы, но этого мало.

Появление JavaScript добавило в стилизованный HTML динамизм, которого так недоставало, и привело к появлению DHTML, принесшего с собой более живой способ общения и провозгласившего эру современных активных Интернет-приложений. JavaScript привносит жизнь в веб-страницы и обеспечивает то удовлетворение, которое мы получаем, когда простой щелчок мышью, выбор в раскрывающемся списке или нажатие клавиши вызывает ощущение, что компьютер угадывает наши мысли.

Все мы без труда опознаем хорошо спроектированные интерактивные страницы, когда сталкиваемся с ними, но обретение навыка создания таких страниц остается весьма трудным делом – программный код JavaScript, управляющий HTML и CSS, часто может быть весьма непростым, и даже очень умные реализации могут оказаться слишком сложными в сопровождении или ничем не примечательными. Корень всех проблем в том, что часто оказывается весьма непросто интегрировать HTML, CSS и JavaScript в единую связную оболочку. Из-за слабой связи между составляющими и их узкой специализации бывает сложно добиться их гармоничного взаимодействия, и потому значительная доля усилий затрачивается на реализацию малоинтересного, шаблонного программного кода. К сожалению, из-за этого трудоемкого процесса может пострадать инициатива и творческий потенциал при разработке действительно значимых частей приложения.

## Диджиты спешат на помощь

К счастью, диджиты значительно упрощают такое положение дел, обеспечивая фундамент, на основе которого вы можете выстраивать более сложный дизайн. Диджиты объединяют HTML, CSS и JavaScript в единую среду, пусть и несовершенную, и позволяют вам мыслить в объектно-ориентированном контексте более быстро, чем это было бы возможно без них: в результате вы напишете основу своего приложения быстрее, чем иссякнет энергия и творческий потенциал. Раньше вам самим пришлось бы обеспечить средства объединения HTML, CSS и JavaScript, теперь же Dojo делает эту утомительную работу за вас, позволяя вам заняться более интересными делами.

Будучи самостоятельными классами, диджиты являются самодостаточными и хранятся в едином каталоге, играющем роль пространства имен. Однако, кроме отдельного файла с программным кодом JavaScript каталог содержит также различные зависимости, такие как файлы изображений и таблиц стилей. Врожденная близость к структуре каталогов обусловливает врожденную переносимость, что обеспечивает простоту использования, развертывания и обновления диджитов. Обслуживание диджитов также несложно – благодаря отсутствию двоичных файлов, с которыми было бы необходимо разбираться, и каждый компонент диджита можно хранить в системе управления версиями, такой как Subversion, в виде отдельного файла.

Хотя все ресурсы, составляющие диджит, можно было бы просто хранить в общем каталоге – по принципу «все в одном флаконе», но на рис. 12.1 отображена обычно применяемая схема расположения файлов на диске для хранения диджитов. Основу этого соглашения составляет стремление расположить различные компоненты в разных подкаталогах, чтобы упростить управление ими.

Диджиты объединяют HTML, CSS и JavaScript, которые являются очень важными составляющими в веб-разработке и предоставляют

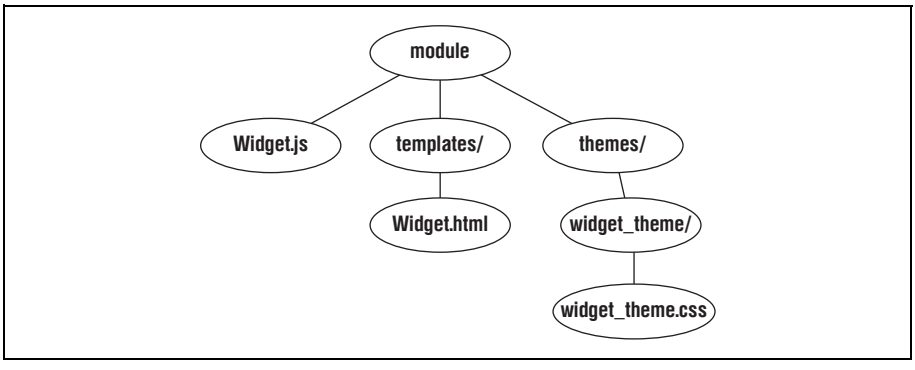



***Рис. 12.1.*** *Структура каталогов для размещения диджита*

унифицированное средство структурирования творческих усилий, необходимых при разработке веб-приложений. В конечном итоге это позволит вам сэкономить время и силы, и, вероятно, создать более эффективный дизайн. Обратите внимание, что в случае простого диджита, не имеющего шаблона или CSS, структура каталогов состоит из единственного каталога с файлом JavaScript.

Структура каталогов, представленная на рис. 12.1, предусматривает хранение шаблона в отдельном файле HTML, причем такое решение используется во время разработки очень часто, так как позволяет членам группы разработчиков заниматься созданием шаблона, CSS и файлов JavaScript независимо друг от друга.

Для загрузки шаблона необходимо, чтобы механизм JavaScript выполнил синхронный запрос к серверу, однако инструментарий Dojo обеспечивает замечательный способ оптимизации синхронных запросов: вы можете включить шаблон в виде встроенной строки в файл JavaScript. Ниже приводится множество примеров такого подхода, демонстрирующих, насколько просто это сделать.

## Методы управления жизненным циклом диджита

Теперь обратим наше внимание на основные методы управления жизненным циклом диджитов, предоставляемые классом `_Widget`. Как будет показано ниже, класс `_Widget` содержит множество мощных методов, для использования которых от вас потребуется приложить самый минимум усилий. Просто добавьте в объявление своего класса класс `_Widget` в качестве основного предка, и ваш подкласс получит непосредственный доступ ко всем стандартным методам управления жизненным циклом, которые можно переопределять, чтобы обеспечить реализацию собственного поведения в процессе конструирования или разрушения диджита.

Например, класс _Widget предоставляет методы, которые можно переопределить, вызываемые перед тем, как диджит появится на экране, сразу после того, как диджит станет видимым, и непосредственно перед уничтожением диджита. Каждая из этих контрольных точек может оказаться очень полезной, когда необходимо выполнить синхронизацию с серверной частью приложения, явно разрушить некоторые объекты (чтобы избежать утечек памяти) или выполнить некоторые манипуляции с деревом DOM. Независимо от конкретных требований этот механизм вам уже не придется разрабатывать – он уже есть, и вы можете использовать его, когда в этом возникнет необходимость.

Чтобы продемонстрировать, что может предложить класс _Widget, ниже приводится пример 12.1, где объявляется простой класс, наследующий класс _Widget и переопределяющий ключевые методы, связанные с созданием и разрушением объекта, которые выводят отладочные сообщения в консоли Firebug. Учитывая сведения, которые приводятся в предыдущем разделе, вы могли бы назвать этот файл *Foo.js* и поместить его в каталог с именем модуля, то есть тем самым отобразить класс в пространство имен.

Ключевой особенностью этого примера, на которую следует обратить внимание, как и следовало ожидать, является факт переопределения методов класса _Widget. Рассмотрите их, а затем мы разберем каждый из методов более подробно.

*Пример 12.1. Подкласс класса _Widget*

```
dojo.require("dijit._Widget");
dojo.addOnLoad(function() {
    dojo.declare(
        "dtdg.Foo",    // подкласс
        dijit._Widget, // суперкласс
        {
            /* Методы создания в хронологическом порядке вызова */
            constructor : function() {console.log("constructor");},
            postMixInProperties : function()
            {
                console.log("postMixInProperties") ;
            },
            postCreate : function() {console.log("postCreate");},

            /* Здесь можно разместить логику работы своего класса */
            talk : function() {console.log("I'm alive!");},

            /* Канонический деструктор, неявно вызывается
               методом destoryRecursive() */
            uninitialize : function() {console.log("uninitialize");}
        }
    );
});

foo = new dtdg.Foo();
```

```
foo.talk();
foo.destroyRecursive(); /* Вызвать все методы uninitialized */
```

Запустив этот пример, вы должны увидеть в консоли Firebug следующее:

```
constructor
postMixInProperties
postCreate
I'm alive!
uninitialize
```

## Жизненный цикл _Widget

Чтобы замкнуть круг обсуждений порядка создания объекта, определяемого функцией `dojo.declare`, начатый в главе 10, ниже показано, как выглядит жизненный цикл объекта класса `_Widget`:

```
preamble (/*Object*/ params, /*DOMNode*/node)
    //вызывается перед функцией constructor;
    //позволяет манипулировать аргументами для конструктора суперкласса

constructor(/*Object*/ params, /*DOMNode*/node)
    // вызывает конструкторы всех суперклассов
    // вызывает все конструкторы подмешиваемых классов
    // вызывает конструктор самого класса, если таковой имеется

postscript (/*Object*/ params, /*DOMNode*/node)
    //Класс _Widget реализует метод postscript, который вызывает метод create
    _Widget.create(/*Object*/params, /*DOMNode*/node)
    _Widget.postMixInProperties()
    _Widget.buildRendering()
    _Widget.postCreate()
```

Отсюда следуют два вывода:

- Класс `_Widget` создается поверх того, что предоставляется функцией `dojo.declare`, и переопределяет метод `postscript`, чтобы выполнить метод `create`, который последовательно вызывает методы управления жизненным циклом класса `_Widget`.
- Виджет, предок класса `_Widget`, является обычным объектом типа Function. С одной стороны, получаем шик и блеск, но опирается все в конце концов на знакомый и надежный фундамент.

### Методы управления жизненным циклом

Ниже дается упрощенное описание жизненного цикла с кратким изложением того, что делает каждый из методов управления жизненным циклом класса `_Widget`. Это упрощенное описание начинается с метода `preamble`, так как вообще-то весьма редко возникает потребность переопределить метод `postscript` или `create` (такая потребность *может* возникнуть, если вы задумаете разработать свои собственные методы управления жизненным циклом, вместо того чтобы использовать

стандартные). Расширенные примеры, которые более подробно охватывают каждый метод, следуют ниже в этой же главе.

*preamble (наследуется от* `dojo.declare`*)*

Метод `preamble` обеспечивает возможность манипулирования аргументами, прежде чем они будут переданы функции `constructor`. Если вы собираетесь переопределить метод `preamble`, вы должны знать, что ему передаются те же самые аргументы, которые обычно передаются методу `constructor`, и все, что возвращает метод `preamble`, передается методу `constructor`. Этот метод имеет довольно узкую область применения и используется гораздо реже остальных методов управления жизненным циклом, таких как `postCreate`.

*constructor (наследуется от* `dojo.declare`*)*

Это самый первый метод, который вам придется переопределять, чтобы назначить свою логику процесса создания диджита. Обычно в методе `constructor` выполняются две операции. Одна из них заключается в инициализации свойств диджита, имеющих сложный тип. (В главе 10 уже говорилось, что в случае встроенного объявления свойств, имеющих сложный тип, такой как объект или список, такие свойства становятся общими для всех экземпляров класса.) Другая операция заключается в создании любых дополнительных свойств, на которые опираются остальные методы управления жизненным циклом.

*postMixInProperties (наследуется от* `dijit._Widget`*)*

Этот метод вызывается сразу после того, как будет произведен обход дерева наследования и выполнено подмешивание к классу всех предков. То есть имя метода *postMixInProperties* буквально указывает на момент времени, когда все свойства виджета будут смешаны в конкретном экземпляре объекта. Таким образом, к тому моменту, когда будет вызван этот метод, ваш класс получит полный доступ ко всем унаследованным свойствам и сможет выполнить с ними некоторые манипуляции, перед тем как диджит появится на экране. Вскоре будет показан пример диджита, который воспроизводится из шаблона, где будет продемонстрировано, как можно использовать этот метод для изменения или получения меток-заполнителей (определяемых в виде `${someWidgetProperty}`), присутствующих в разметке шаблона.

*buildRendering (наследуется от* `dijit._Widget`*)*

В реализации класса `_Widget` этот метод просто устанавливает значение внутреннего свойства `_Widget.domNode`, записывая в него ссылку на фактический элемент DOM, чтобы диджит физически стал частью страницы. Учитывая, что этот метод выполняется сразу вслед за методом `postMixInProperties`, должно стать понятно, почему именно метод `postMixInProperties` используется для изменения шаблона виджета.

Вскоре вы познакомитесь с еще одним основополагающим классом в библиотеке Dijit – с классом `_Templated`, который переопределяет этот метод для выполнения бесчисленного множества мелочей, связанных с получением и обработкой шаблона диджита. Наконец, следует отметить, что сразу после вызова метода `buildRendering` сам диджит добавляется в объект Dojo, осуществляющий управление диджитами, чтобы диджит мог быть корректно уничтожен при явном вызове метода деструктора или при выгрузке страницы. В некоторых броузерах случаются утечки памяти, которые становятся значимыми при работе с долгоживущими приложениями, а отслеживая виджеты с помощью централизованного реестра, Dojo позволяет смягчить эту проблему. Необходимость переопределять этот метод возникает крайне редко – чаще используется реализация по умолчанию, имеющаяся в классе `_Widget` или `_Templated`.

*postCreate (наследуется от* `dijit._Widget`*)*

Этот метод вызывается сразу же, как только диджит будет создан и его визуальное представление появится на странице, поэтому его можно использовать для выполнения любых действий, которые невозможны или неуместны до данного момента. Обратите особое внимание, что такие операции, как применение стиля или изменение местоположения диджита на экране, должны выполняться до того, как диджит станет видимым – в методе `postMixInProperties`. Выполнение этих действий в методе `postCreate` может иногда вызывать резкие изменения на экране, потому что производиться они будут, когда диджит уже видим. Такого рода ошибки бывает очень сложно отыскать и исправить, если забыть фундаментальные различия между методами `postMixInProperties` и `postCreate`. Кроме того, обратите внимание, что если диджит содержит какие-либо дочерние диджиты, то будет небезопасно обращаться к ним в этом методе. Чтобы обезопасить попытки обращения к дочерним диджитам, производите их в методе управления жизненным циклом `startup`. Чтобы обезопасить попытки обращения к другим, не дочерним виджетам, дождитесь окончания загрузки страницы и производите эти попытки в функции `addOnLoad`.

*startup (наследуется от* `dijit._Widget`*)*

Этот метод вызывается автоматически, как только будут созданы сам виджет и *все* его дочерние виджеты, определенные в разметке. Это, к тому же, первый метод, где можно безопасно обращаться к дочерним виджетам. Эту задачу часто пытаются решить в методе `postCreate`, что приводит к неустойчивости в работе, причины которой достаточно сложно определить и исправить. Для виджетов, создаваемых программно и содержащих дочерние виджеты через отношение *имеет*, метод `startup` необходимо вызывать вручную, после того как вы будете уверены, что все дочерние виджеты созданы и доступны. Причина, по которой этот метод придется вызывать

вручную – в случае создания виджета программным способом, состоит в том, что нет смысла продолжать работать с размерами и отображением виджета, пока не будут добавлены все дочерние виджеты. (В противном случае это может приводить к неправильной работе с самого начала.) Этот метод является последним методом-заглушкой, который можно переопределять при реализации собственной логики создания диджита.

*destroyRecursive (наследуется от* `dijit._Widget`*)*

Этот метод является обычным деструктором, который вызывается, чтобы уничтожить диджит и все его дочерние диджиты. В ходе своей работы этот метод вызывает метод `uninitialize`, который является главным кандидатом на переопределение для выполнения нестандартных операций. *Не переопределяйте* метод `destroyRecursive`. Реализуйте свои операции в методе `uninitialize` и вызывайте этот метод (так как он не вызывается автоматически), чтобы он смог позаботиться обо всем остальном.

*uninitialize (наследуется от* `dijit._Widget`*)*

Переопределяйте этот метод для реализации собственной логики поведения диджита при его уничтожении. Например, этот метод можно было бы использовать для выполнения обращений к серверу, чтобы сохранить состояние сеанса или выполнить заключительные манипуляции с деревом DOM. Этот метод является типичным местом, которое должно использоваться всеми диджитами для выполнения действий по уничтожению.

Понимание того, чем отличаются друг от друга различные методы управления жизненным циклом, совершенно необходимо. Поэтому проявите особое тщание, чтобы запомнить, какие операции и в каких методах можно выполнять.

Среди типичных ошибок встречаются:

- Попытки манипулировать шаблоном после вызова метода `postMixInProperties`
- Изменение первоначального внешнего вида виджета после вызова метода `postMixInProperties`
- Попытки получить доступ к дочерним виджетам в методе `postMixInProperties`, а не в методе `startup`
- Забывание выполнения каких-то из обязательных операций по уничтожению диджита в методе `uninitialize`
- Вызов вместо метода `destroyRecursive` метода `uninitialize`

## Основные свойства

В дополнение к только что описанным методам класс `_Widget` обладает также некоторыми особенно примечательными свойствами. Как и в случае методов, вы можете обращаться к этим свойствам, используя то-

чечную нотацию. Обычно эти свойства используются исключительно для чтения:

id

Значением этого свойства является уникальный идентификатор, который присваивается диджиту. Если вы сами не предоставляете это значение, оно автоматически будет присвоено самим инструментарием Dojo. Вы никогда не должны изменять это значение, и в большинстве случаев вам вообще не придется пользоваться им.

lang

Инструментарий Dojo поддерживает возможность интернационализации, и значение этого свойства может использоваться для настройки таких особенностей, как язык, используемый для отображения диджита. По умолчанию значение этого свойства соответствует настройкам броузера, которые в свою очередь обычно соответствуют настройкам операционной системы.

domNode

Это свойство содержит ссылку на самый внешний узел DOM диджита. Это свойство указывает на узел, являющийся визуальным представлением диджита на экране. Обычно следует избегать прямых манипуляций с этим свойством, однако манипуляции с этим свойством бывают удобны при отладке. Как уже упоминалось ранее, в реализации класса `_Widget` по умолчанию это свойство устанавливается в методе `buildRendering`, и любые методы, переопределяющие `buildRendering`, должны принимать на себя ответственность за установку этого свойства, в противном случае могут происходить странные, почти мистические вещи. Если диджит не отображается на экране, значение свойства равно `undefined`.

Чтобы внести ясность, ниже приводится фрагмент простого программного кода, а в соответствующем выводе в консоли Firebug можно увидеть все эти свойства. Отмечу еще раз, что все свойства унаследованы от класса `_Widget` и доступны через ключевое слово `this`, когда оно ссылается на контекст ассоциативного массива, который передается в третьем аргументе функции `dojo.declare`:

```
dojo.require("dijit._Widget");
dojo.addOnLoad(function() {
    dojo.declare(
        "dtdg.Foo",
        dijit._Widget,
        {
            talk() : function() {
                console.log("id:", this.id);
                console.log("lang:", this.lang);
                console.log("dir:", this.dir);
                console.log("domNode:", this.domNode);
            }
```

```
        }
    );
});
foo = new dtdg.Foo();
foo.talk();
```

## Подмешивание класса _Templated

В то время как класс _Widget предоставляет методы-заглушки, через которые можно переопределять события создания и разрушения, возникающие в течение жизненного цикла, класс _Templated, упоминавшийся ранее как предок, фактически составляет основу для определения шаблона виджета в виде разметки и использования подстановок и точек подключения для расширения его функциональных возможностей. В общем и целом это деление инструментария, которое позволяет размежевать труд дизайнера и программиста.

Большая часть работы класса _Templated связана с парсингом и подстановкой значений в файлах шаблонов. Выполнение важной части этой работы влечет за собой переопределение метода buildRendering класса _Widget, в котором выполняются все действия по применению шаблона. При работе с шаблонами используются три важных понятия:

*Подстановка*

Для выполнения подстановок в шаблонах библиотека Dijit применяет модуль dojo.string, отыскивая и используя с его помощью синтаксические конструкции вида ${xxx}. Это удобно для получения передаваемых атрибутов виджетов и использования их для настройки шаблонов.

*Точки подключения*

Когда в шаблонном узле используется специальный атрибут dojoAttachPoint, тем самым обеспечивается возможность прямой ссылки на узел через значение атрибута. Например, если в шаблоне появляется такой узел: ..., вы получаете возможность напрямую обратиться к нему, как this.foo (в методе postCreate или позже).

*Точки событий*

Как и в случае точек подключения, имеется возможность использовать специальный атрибут dojoAttachEvent, в результате чего устанавливается связь между событием DOM узла в шаблоне и методом виджета, который должен вызываться в ответ на указанное событие DOM. Например, если в шаблоне появляется такой узел: ..., метод foo виджета будет вызываться всякий раз, когда в узле будет возникать событие onclick. Допускается объявлять несколько событий, разделяя объявления запятыми.

Подобно классу _Widget, класс _Templated вскоре будут демонстрироваться в некоторых примерах. А теперь познакомимся с ним поближе, чтобы у вас сложилось общее представление о его назначении.

## Методы управления жизненным циклом

Наиболее заметный эффект от подмешивания класса _Templated заключается в том, что он переопределяет метод buildRendering класса _Widget. Ниже приводится краткое описание метода buildRendering.

*buildRendering*

Класс _Widget предоставляет этот метод, а класс _Templated переопределяет его, реализуя действия, связанные с получением и наполнением файла шаблона диджита для отображения на экране. Вообще говоря, вам едва ли придется писать свою реализацию метода buildRendering. Однако, даже если вы соберетесь переопределить этот метод, убедитесь сначала, что полностью понимаете, как реализован класс _Templated.

## Основные свойства

Ниже приводятся краткие описания некоторых основных свойств класса _Templated:

templatePath

Содержит относительный путь к файлу (обычный файл HTML) шаблона диджита. Обратите внимание, что для получения файла шаблона диджита необходимо выполнить синхронный запрос, впрочем, инструментарий Dojo поместит строку шаблона в кэш, после того как он будет получен в первый раз. В главе 16 ведется обсуждение вопросов создания собственных сборок диджитов с помощью инструментов из библиотеки Util, так чтобы все строки шаблонов оказались помещенными в сборку.

templateString

Для диджитов, которые были спроектированы или собраны так, что строки с их шаблонами встроены в файл JavaScript, значение этого свойства представляет шаблон. Если определены оба свойства, templatePath и templateString, предпочтение отдается свойству templateString.

widgetsInTemplate

Если диджиты определены внутри шаблона (либо в файле, либо в строке шаблона), это свойство явно должно быть установлено в значение true, чтобы парсер Dojo знал, что необходимо отыскать и создать экземпляры этих диджитов. Значение по умолчанию: false. Включение диджитов в состав шаблона может быть очень удобно. Обычно для передачи значений дочерним виджетам, присутствующим в шаблоне родительского виджета, применяется нотация

`${someWidgetProperty}`, которая используется для подстановки фактических значений.

`containerNode`

Значение этого свойства ссылается на элемент DOM, который отображается в атрибут `dojoAttachPoint` в веб-странице, содержащей диджит. Оно также определяет элемент, куда будут добавляться новые дочерние диджиты, если ваш диджит играет роль контейнера для списка диджитов. (Диджиты, которые могут играть роль контейнера, наследуют класс `_Container`, а диджиты, которые могут добавляться в контейнер, наследуют класс `_Contained`.)

# Ваш первый диджит: HelloWorld

После такого вступления вы безусловно готовы к тому, чтобы посмотреть программный код в действии. В этом разделе вашему вниманию будет представлен набор все более усложняющихся примеров из серии «HelloWorld», которые демонстрируют фундаментальные концепции, связанные с разработкой собственных диджитов.

Давайте создадим классический диджит HelloWorld и поближе познакомимся с некоторыми обсуждавшимися проблемами. Хотя в этом разделе все внимание будет сосредоточено на создании максимально простого диджита, тем не менее здесь вам встретятся некоторые сложности, обычные при разработке диджитов, которые мы проясним в процессе обсуждения.

На рис. 12.2 приводится структура каталогов для расположения файлов диджита HelloWorld на диске. В этой структуре нет ничего особенного – она прямо следует типичной структуре каталогов, представленной ранее.

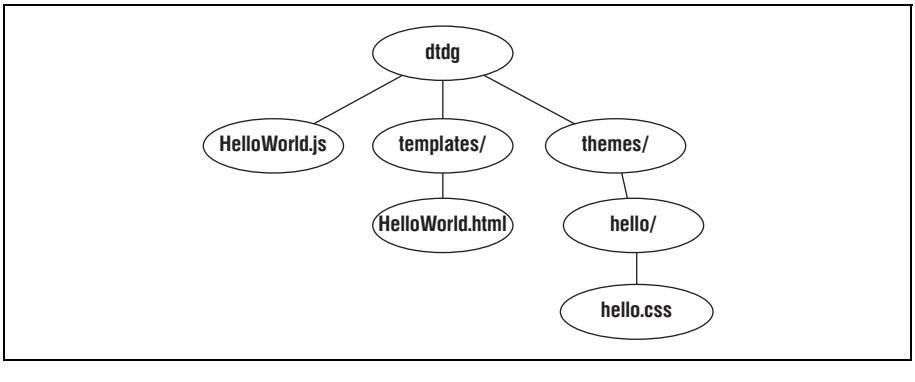



***Рис. 12.2.*** *Структура каталогов для минимального диджита HelloWorld*

## Диджит HelloWorld (Дубль 1: Основа)

В первой попытке создания диджита HelloWorld будет полностью создано тело каждого компонента. Для краткости и ясности во всех последующих итерациях будут описываться только те части компонентов, которые будут подвергаться изменениям. Относительно структуры каталогов на диске предполагается, что главный файл HTML, включающий виджеты, находится в одном каталоге с каталогом модуля *dtdg*, где содержится программный код реализации виджета.

### Страница HTML

Сначала рассмотрим страницу HTML, содержащую диджит, которая приводится в примере 12.2. Подробные комментарии, присутствующие в странице, достаточно ясно объясняют ее.

*Пример 12.2. HelloWorld (Дубль 1 )*

```
<html>
    <head>
        <title>Hello World, Take 1</title>
        <!--
        Поскольку мы получаем версию Dojo с сервера AOL, нам необходимо
        определить несколько дополнительных параметров настройки
        в массиве djConfig, чтобы загрузить версию XDomain
        сборки (dojo.xd.js).

        Например, мы связываем пространство имен "dtdg" с конкретным
        относительным путем к каталогу на диске, определив
        параметр baseUrl и коллекцию отображений пространств имен.
        Если бы мы использовали локальную копию Dojo, мы могли
        бы просто поместить каталог dtdg рядом с каталогом dojo
        и он был бы найден автоматически.

        Указание, что на странице имеются диджиты, которые требуют
        выполнения парсинга при загрузке страницы, является стандартным
        действием для ситуаций, когда на странице встречается
        атрибут dojoType.
        -->

        <script
            type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js"
            djConfig=isDebug:true,parseOnLoad:true,baseUrl:'./',
                     modulePaths:{dtdg:'dtdg'}">
        </script>

        <!--
        Помимо своих собственных таблиц стилей вам обычно придется
        подключать файл dojo.css. Не забывайте, что, если вы
        не используете версию сборки AOL XDomain, вам следует указать
        ссылку на свой локальный файл dojo.css.
        -->
```

```
        <link
            rel="stylesheet"
            type="text/css"
            href="http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css">
        </link>

        <link
            rel="stylesheet"
            type="text/css"
            href="dtdg/themes/hello/hello.css">
        </link>

        <script type="text/javascript">
            dojo.require("dojo.parser");

            //Сообщить Dojo о необходимости получить диджит с именем
            //HelloWorld, расположенный в пространстве имен dtdg, чтобы
            //его можно было использовать в теле страницы. Для поиска
            //местоположения диджита Dojo будет использовать значение
            //параметра djConfig.modulePaths.
            dojo.require("dtdg.HelloWorld");
        </script>
    </head>
    <body>
        <!--
        Здесь парсер Dojo подставит диджит, загруженный инструкцией
        dojo.require, поскольку был определен параметр parseOnLoad:true.
        Любые стили, применяемые к диджиту, будут получены
        из импортированной таблицы стилей.
        -->
        <div dojoType="dtdg.HelloWorld"></div>
    </body>
</html>
```

То, что вы только что увидели, является тем минимумом, который должен присутствовать в любой странице, содержащей диджит. Здесь присутствуют ссылки на все необходимые таблицы стилей, улучшающие внешний вид диджитов, обычная ссылка на библиотеку Base, которая выполнит самонастройку Dojo, и далее инструкции `dojo.require`, явно выполняющие загрузку парсера и диджита HelloWorld, который используется в теле страницы. Из всего этого единственный момент, требующий особого внимания, – правильное указание пути к каталогу, в котором находится модуль `dtdg`, в параметре `djConfig.modulePaths`.

## CSS

Стиль отображения виджета определяется простой каскадной таблицей стилей CSS и дополнительными статическими ресурсами, такими как файлы изображений. Однако следует понимать, что фактический стиль диджита определяется в его шаблоне, а не в элементе DOM, где указывается атрибут `dojoType`. Это добавляет элегантности в реализацию, так как, по сути, обеспечивает возможность применения к дид-

житу различных стилей или, выражаясь терминологией Dojo, обеспечивает возможность определять *темы* для диджита и изменять эти темы, с помощью таблицы стилей.

В нашем примере для отдельного элемента DIV используется очень простой стиль, но он наглядно демонстрирует, как можно применять стили к диджитам. Наша тема HelloWorld состоит из единственного файла CSS, который содержит следующее определение стиля:

```
div.hello_class {
    color: #009900;
}
```

## Шаблон

Как и стиль, шаблон HTML диджита HelloWorld содержит самый минимум. Этот шаблон предписывает Dojo взять указанный тег DIV, находящийся в нашей странице HTML, и заменить его тем, что находится в шаблоне, – в данном случае шаблон просто содержит другой элемент DIV, к которому применен некоторый стиль и в котором содержится текст «Hello World».

Фактически файл шаблона содержит всего лишь следующую строку:

```
<div class="hello_class">Hello World</div>
```

## JavaScript

При первом взгляде на программный код JavaScript создается впечатление, что он слишком объемный, однако большую его часть составляют подробные комментарии. Мы по-прежнему используем только самые основные конструкции, которые уже рассматривали ранее, и, изучив листинг, вы убедитесь, что в нем нет ничего сложного. Итак, рассмотрите листинг, что приводится ниже, а затем мы перечислим, что в нем имеется. Вы можете заметить, что файл JavaScript – просто стандартный модуль:

```
//Описание: Пример диджита HelloWorld, демонстрирующий основные
//принципы создания диджитов Dojo

//Первая строка любого модуля должна содержать инструкцию dojo.provide,
//определяющую ресурс и принадлежность к родительским модулям.
//Имя ресурса должно совпадать с именем файла .js.
dojo.provide("dtdg.HelloWorld");

//Всегда подключайте необходимые ресурсы, прежде чем использовать их.
//Нам потребуются следующие два ресурса, потому что они являются
//частью иерархии наследования нашего диджита.
dojo.require("dijit._Widget");
dojo.require("dijit._Templated");

//Богатый возможностями конструктор, позволяющий объявлять "классы" Dojo.
dojo.declare(
    "dtdg.HelloWorld",
```

```
        //dijit._Widget - это предок прототипа, который обеспечивает
        //важные методы-заглушки, часть которых используется ниже.
        //dijit._Templated - это подмешиваемый класс, переопределяющий
        //метод buildRendering класса dijit._Widget, который создает
        //визуальное представление диджита из шаблона.
        [dijit._Widget, dijit._Templated],
        {
            //Путь к шаблону этого диджита. Класс dijit._Templated использует
            //его для получения шаблона в виде указанного файла посредством
            //выполнения синхронного вызова.
            templatePath: dojo.moduleUrl("dtdg", "templates/HelloWorld.html")
        }
    );
```

В цепочке наследования класс `_Widget` является предком прототипа, который должен наследовать наш диджит, чтобы действительно стать диджитом. Поскольку это самый первый пример, мы не будем переопределять какие-либо методы управления жизненным циклом класса `_Widget`, но в последующих примерах эти методы будут переопределяться. Подмешиваемый предок, класс `_Templated`, реализует функциональные возможности работы с шаблонами, переопределяя метод `buildRendering` класса `_Widget`. Фактический шаблон определяется свойством `templatePath`. Использование свойства `templatePath` вместо `templateString` влечет за собой дополнительный синхронный вызов сервера, однако полученный шаблон помещается в кэш, что позволяет избежать необходимости выполнять другие синхронные вызовы, если на той же странице будут присутствовать другие диджиты HelloWorld.

Когда Dojo впервые загружает файл шаблона, это влечет за собой некоторую задержку из-за выполнения синхронного запроса к серверу. При последующих попытках получить шаблон он извлекается из кэша.

При выполнении этот пример просто выводит текст сообщения на экран, однако внутри происходит гораздо больше событий, чем неявный вызов инструкции `print`. Более того, для создания диджита HelloWorld требуется приложить лишь самый минимум усилий, намного меньше, чем при создании любого другого диджита.

Давайте закрепим ваши знания, полученные ранее, и переопределим некоторые методы, чтобы расширить возможности диджита. Но вместо того чтобы идти напрямик, мы пойдем обходным путем. В конце концов, какой другой путь лучше способствует обучению?

## Диджит HelloWorld (Дубль 2: Изменение шаблона)

Предположим, что вам необходимо сделать свой диджит немного сложнее. Например, для начала было бы здорово реализовать возможность динамического отображения различных сообщений при каждой загрузке страницы, вместо того чтобы всякий раз выводить одно и то

же сообщение. Один из замечательных механизмов Dojo, используемых для поддержки концепции самодостаточности диджита, позволяет в файле шаблона ссылаться на свойства диджита, объявляемые в исходном программном коде JavaScript. Использование свойств диджита внутри шаблона полезно только до выполнения метода buildRendering класса _Templated, тем не менее вы увидите, что инициализация некоторой части отображаемого диджита, перед тем как он появится на экране, является очень часто используемой операцией.

Ссылка на свойство диджита внутри файла шаблона оформляется очень просто. Взгляните на следующую версию файла шаблона HelloWorld:

```
<div class="hello_class">${greeting}</div>
```

Проще говоря, вы можете обратиться к любому существующему свойству диджита изнутри файла шаблона и использовать его для изменения изображения виджета, его стиля и т. д. Однако здесь имеется маленькое, но очень важное ограничение: делать это следует в нужное время. В частности, для работы со свойствами диджита, которые используются в шаблонах, лучше всего подходит метод postMixInProperties. Напомню, что метод postMixInProperties вызывается перед методом builRendering, который является той точкой, где диджит вставляется в дерево DOM и становится видимым.

Напомню, что общепринятым местом в жизненном цикле диджита, где выполняются манипуляции со строками шаблонов, является метод postMixInProperties, который наследуется от класса _Widget. Попытки манипулирования строками шаблонов после этого момента могут приводить к нежелательным эффектам резкой смены изображения на экране.

Без лишних слов перейдем к примеру 12.3, где приводится содержимое файла JavaScript с реализацией диджита, предусматривающей возможность манипулирования свойствами в шаблоне для отображения другого текста приветствия.

*Пример 12.3. HelloWorld (Дубль 2: postMixInProperties)*

```
//Пример правильной организации манипулирования свойством диджита в методе
//postMixInProperties, ссылка на которое присутствует в строке шаблона
dojo.provide("dtdg.HelloWorld");

dojo.require("dijit._Widget");
dojo.require("dijit._Templated");

dojo.declare(
    "dtdg.HelloWorld",
     [dijit._Widget, dijit._Templated],
    {
        greeting : "",

        templatePath: dojo.moduleUrl(
```

```
            "dtdg",
            "templates/HelloWorld.html"
        ),

        postMixInProperties: function() {
            //Манипулирование свойствами, ссылки на которые
            //присутствуют в шаблонах.
            this.greeting = "Hello World"; //использовать статический текст.
        }
    }
)
```

## Диджит HelloWorld (Дубль 3: Внедрение шаблона)

Как уже говорилось ранее, чтобы сэкономить время, затрачиваемое на выполнение синхронного запроса к серверу, можно определить строку шаблона непосредственно в файле JavaScript. В примере 12.4 приводится измененная версия HelloWorld, где показано, насколько просто это можно реализовать вручную, но имейте в виду, что сценарии сборки из библиотеки Util могут автоматизировать этот процесс для всех ваших диджитов в процессе подготовки их к развертыванию.

*Пример 12.4. HelloWorld (Дубль 3: templateString)*

```
dojo.provide("dtdg.HelloWorld");
dojo.require("dijit._Widget");
dojo.require("dijit._Templated");

dojo.declare(
    "dtdg.HelloWorld",
    [dijit._Widget, dijit._Templated],
    {
        greeting : "",

        //Добавить встроенную строку шаблона...
        templateString : "<div class='hello_class'>${greeting}</div>",

        postMixInProperties: function() {
            console.log ("postMixInProperties");

            //Мы по-прежнему имеем возможность оказывать влияние
            //на строку шаблона
            this.greeting = "Hello World";
        }
    }
);
```

В этом примере свойство `templateString` предоставляет встроенный шаблон, благодаря этому отпала необходимость в отдельном файле шаблона. В свою очередь, это экономит время на выполнение синхронного запроса к серверу. Представьте себе множество диджитов с большим количеством строк шаблонов – совершенно очевидно, что такое встраивание шаблонов может привести к существенному уменьшению

времени загрузки страницы. В ситуации подготовки рабочей версии приложения вам на помощь придет система сборки из библиотеки Util (глава 16), которая автоматизирует подобные оптимизации производительности.

## Диджит HelloWorld (Дубль 4: Передача параметров)

В качестве еще одного усовершенствования нашего диджита HelloWorld рассмотрим порядок передачи параметров диджиту в его шаблоне. Опираясь на предыдущий пример, предположим, что нам необходимо определить свой текст приветствия для диджита, который определяется в разметке страницы с помощью атрибута `dojoType`. Делается это просто, как показано ниже:

```
<div dojoType="dtdg.HelloWorld" greeting="Hello World"></div>
```

Передача параметров виджету, создаваемому программным способом, выполняется ничуть не сложнее:

```
var hw = new dtdg.HelloWorld({greeting : "Hello World"}, theWidgetsDomNode);
```

Конечно, вы не ограничены только теми свойствами, что упоминаются в шаблоне. Вы можете передавать любые другие параметры, которые могут использоваться разными способами. Взгляните на следующий ниже элемент `DIV`, содержащий ссылку на диджит HelloWorld, где определяются две дополнительные пары ключ/значение:

```
<div foo="bar" baz="quux" dojoType="dtdg.HelloWorld"></div>
```

Разве это не удобно – иметь возможность передавать диджиту дополнительные данные для использования в процессе инициализации, что позволило бы разработчикам приложений даже не прикасаться к исходному программному коду, а только слегка подправлять шаблон? Итак, дамы и господа, это возможно, и файл с программным кодом JavaScript, как показано в примере 12.5, демонстрирует, как это делается.

*Пример 12.5. HelloWorld (Дубль 4: дополнительные параметры)*

```
dojo.provide("dtdg.HelloWorld");

dojo.require("dijit._Widget");
dojo.require("dijit._Templated");

dojo.declare(
    "dtdg.HelloWorld",
    [dijit._Widget, dijit._Templated],
    {
        templateString : "<div class='hello_class'>Hello World</div>",

        foo : "",

        //вы не сможете устанавливать несуществующие свойства диджита
        //baz : "",
```

```
        //атрибуты, указанные в теге с атрибутом dojoType, передаются
        //только функции constructor, если предварительно они были
        //определены как свойства диджита. То есть наличие
        //атрибута baz="quux" не будет оказывать никакого влияния в этом
        //примере, потому что диджит не имеет свойства с именем baz
        constructor: function() {
            console.log("constructor: foo=" , this.foo);
            console.log("constructor: baz=" , this.baz);
        }
    }
);
```

Возможно, вы уже обратили внимание на тот факт, что *вы можете передавать значения только для тех свойств диджита, которые существуют,* – невозможно создать новое свойство в диджите, поместив все, что вам заблагорассудится в элемент, содержащий атрибут `dojoType`. Если запустить предыдущий пример и проверить результаты в консоли Firebug, вы увидите следующий вывод:

```
constructor: foo=bar
constructor: baz=undefined
```

Хотя передача строковых значений диджиту – очень полезная возможность, тем не менее строковые значения имеют ограниченную область применения, потому что жизнь, как правило, не так проста; но и тут не стоит беспокоиться: инструментарий Dojo позволяет также передавать диджитам списки и ассоциативные массивы. Все, что для этого требуется, это определить свойства диджита соответствующего типа в файле JavaScript, а обо всем остальном инструментарий Dojo позаботится сам.

Следующий пример демонстрирует, как передавать списки и ассоциативные массивы в диджиты в шаблоне.

Добавление параметров в тег с атрибутом `dojoType` выполняется очень просто:

```
<div
    foo="[0,20,40]"
    bar="[60,80,100]"
    baz="{'a':'b', 'c':'d'}"
    dojoType="dtdg.HelloWorld"
></div>
```

И изменения в файле JavaScript столь же предсказуемы:

```
dojo.provide("dtdg.HelloWorld");

dojo.require("dijit._Widget");
dojo.require("dijit._Templated");

dojo.declare(
    "dtdg.HelloWorld",
    [dijit._Widget, dijit._Templated],
```

```
        {
            templateString : "<div class='hello_class'>Hello World</div>",

            foo : [], //привести значение к типу Array
            bar : "", //привести значение к типу String
            baz : {}, //привести значение к типу Object

            postMixInProperties: function() {
                console.log("postMixInProperties: foo[1]=" , this.foo[1]);
                console.log("postMixInProperties: bar[1]=" , this.bar[1]);
                console.log("postMixInProperties: baz['a']=", this.baz['a']);
            }
        }
    );
```

Ниже приводится результат работы этого примера, полученный в консоли Firebug:

```
postMixInProperties: foo[1]=20
postMixInProperties: bar[1]=6
postMixInProperties: baz['a']=b
```

Обратите внимание на свойство `bar`: в странице, где используется диджит, это свойство *выглядит* как список, но в файле JavaScript оно определено как свойство строкового типа. По этой причине Dojo рассматривает его как строку и при попытке индексирования обращается с ним как со строкой. Вообще парсер пытается интерпретировать значения и преобразовывать их в соответствующие типы, используя возможности грубого определения типа.

Особое внимание уделяйте корректному определению типов параметров в файле JavaScript, в противном случае это может привести к необходимости тратить время на отладку!

## Диджит HelloWorld (Дубль 5: Обработка событий в диджитах)

В качестве еще одного улучшения диджита HelloWorld рассмотрим возможность обработки таких событий DOM, как щелчок мыши или наведение указателя мыши на диджит. Инструментарий Dojo упрощает возможность связывания событий с диджитом. Для этого нужно просто указать пары ключ/значение в форме `DOMEvent:dijitMethod` внутри атрибута `dojoAttachEvent`, который является частью шаблона. Допускается указывать несколько пар ключ/значение или более одного типа события DOM, отделяя их друг от друга запятыми.

Давайте посмотрим, как можно использовать атрибут `dojoAttachEvent` для применения заданного стиля, определенного в виде класса в таблице стилей, при появлении события `mouseover`, и удаления стиля при появлении события `mouseout`. Поскольку элементы DIV занимают

всю ширину кадра, мы заменим его элементом SPAN, чтобы события от мыши возникали только тогда, когда указатель находится непосредственно над текстом. Давайте применим стиль pointer к указателю.

Изменения в стиле диджита минимальны. Мы использовали ссылку на элемент SPAN вместо элемента DIV и изменили форму указателя мыши на pointer:

```
span.hello_class {
    cursor: pointer;
    color: #009900;
}
```

Файл JavaScript, представленный в примере 12.6, включает в себя измененную строку шаблона, демонстрируя, насколько просто использовать атрибут dojoAttachEvent.

*Пример 12.6. HelloWorld (Дубль 5: dojoAttachEvent)*

```
dojo.provide("dtdg.HelloWorld");

dojo.require("dijit._Widget");
dojo.require("dijit._Templated");

dojo.declare(
    "dtdg.HelloWorld",
    [dijit._Widget, dijit._Templated],
    {
        templateString :
            "<span class='hello_class'
            dojoAttachEvent='onmouseover:onMouseOver, onmouseout:onMouseOut'>
             Hello World</span>",

        onMouseOver : function(evt) {
            dojo.addClass(this.domNode, 'hello_class');
            console.log("applied hello_class...");
            console.log(evt);
        },

        onMouseOut : function(evt) {
            dojo.removeClass(this.domNode, 'hello_class');
            console.log("removed hello_class...");
            console.log(evt);
        }
    }
);
```

Убедились, как это просто? Когда указатель мыши наводится на текст в элементе SPAN, возникает событие onmouseover и применяется стиль с помощью функции dojo.addClass из библиотеки Base. Затем, когда возникает событие onmouseout, стиль удаляется. Отличная штука!

Обратили ли вы внимание на то, что обработчики событий принимают аргумент evt, – может быть, в нем передается важная информация

о событии? Возможно, вы уже догадались, что за кулисами работает функция `dojo.connect`, которая стандартизирует объект события. Ниже приводится вывод, полученный в консоли Firebug в ходе работы этого программного кода, который наглядно демонстрирует, какая информация передается обработчикам событий диджита:

```
applied hello_class...
mouseover clientX=64, clientY=11
removed hello_class clientX=65, clientY=16
mouseover clientX=65, clientY=16
```

Особое внимание уделяйте правильному написанию имен событий DOM и записывайте эти имена только символами нижнего регистра. Например, имена `ONMOUSEOVER` и `onMouseOver` не могут использоваться для назначения обработчика события `onmouseover` и, к большому сожалению, отладчик Firebug не сможет подсказать вам, где кроется ошибка. Тот факт, что вы можете называть свои обработчики событий в диджите как угодно (с любым порядком следования символов верхнего и нижнего регистра), только способствует тому, чтобы забыть это правило.

Кроме того, для достижения полной ясности, обратите внимание, что в предыдущем примере событие `onmouseover` отображено на метод `onMouseOver`, а событие `onmouseout` – на метод `onMouseOut`. Это не является обязательным требованием, но вполне разумно следовать такому правилу именования, так как это повышает удобочитаемость программного кода. Кроме того, важно заметить, что такие события, как `onmouseover` и `onmouseout`, являются *событиями DOM*, тогда как `onMouseOver` и `onMouseOut` – это *методы, ассоциированные с данным конкретным диджитом*. Возможно, на первый взгляд разница не видна, потому что имена читаются одинаково, но это очень важная концепция, которую вам необходимо усвоить в процессе обретения мастерства владения возможностями библиотеки Dijit. В семантическом смысле эти имена похожи, но они совершенно отличны друг от друга во всех остальных отношениях.

## Организация отношений родитель–потомок с помощью классов _Container и _Contained

После некоторой практики работы с классами `_Widget` и `_Templated` вы быстро поймете, насколько удобны виджеты, которые могут содержать другие дочерние виджеты. Отношения типа «имеет» часто используются в программировании, и инструментарий Dojo в этом смысле не является исключением. Подмешиваемые классы `_Container` и `_Contained` предназначены для упрощения возможности ссылаться из родительских диджитов на дочерние и наоборот, что часто бывает необходимо. В табл. 12.1 приводится краткое описание интерфейсов этих классов.

*Таблица 12.1. Подмешиваемые классы _Container и _Contained*

| Имя | Комментарий |
|---|---|
| `removeChild(/*Object*/ dijit)` | Удаляет дочерний виджет из родительского. (Завершает работу без ошибки, если указанный виджет не является дочерним или если контейнер не имеет дочерних виджетов.) |
| `addChild(/*Object*/ dijit, /*Integer?*/ insertIndex)` | Добавляет дочерний виджет в родительский. При необходимости для помещения в требуемую позицию может использоваться необязательный параметр `insertIndex`. |
| `getParent()` | Позволяет дочернему виджету сослаться на родительский виджет. Возвращает экземпляр диджита. |
| `getChildren()` | Позволяет родительскому виджету получить список всех дочерних виджетов. Возвращает массив экземпляров диджитов. |
| `getPreviousSibling()` | Позволяет дочернему виджету сослаться на предыдущий, то есть на «тот, что слева», дочерний виджет. Возвращает экземпляр диджита. |
| `getNextSibling()` | Позволяет дочернему виджету сослаться на следующий, то есть на «тот, что справа», дочерний виджет. Возвращает экземпляр диджита. |

# Быстрое создание прототипов виджетов в разметке

Теперь, когда у вас имеется полное представление о том, как протекает жизненный цикл виджета, и после знакомства со множеством примеров, пришло время продемонстрировать способ простого и быстрого создания прототипов. В библиотеке Dijit имеется ресурс `Declaration`, позволяющий объявлять виджеты в разметке без необходимости создавать отдельный файл JavaScript. Этот подход может существенно упростить процесс разработки, когда необходимо быстро реализовать и проверить некоторую идею.

В примере 12.7. демонстрируется реализация нашего первого виджета HelloWorld с использованием ресурса `Declaration`, применение которого позволило создать виджет целиком в одной самостоятельной странице.

*Пример 12.7. HelloWorld (Дубль 6: Declaration)*

```
<html>
    <head>
        <title>Hello World, Take 6</title>
```

```
        <script
            type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js"
            djConfig="isDebug:true,parseOnLoad:true">
        </script>

        <link
            rel="stylesheet"
            type="text/css"
            href="http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css">
        </link>

        <!-- объявление встроенных таблиц стилей CSS -->
        <style type="text/css">
            span.hello_class {
                color: #009900;
                cursor: pointer;
            }
        </style>

        <script type="text/javascript">
            dojo.require("dijit.Declaration");
            dojo.require("dojo.parser");
        </script>
    </head>
    <body>
        <!-- полное объявление виджета в разметке -->
        <div
            dojoType="dijit.Declaration"
            widgetClass="dtdg.HelloWorld"
            defaults="{greeting:'Hello World'}">

            <span class="hello_class"
           dojoAttachEvent='onmouseover:onMouseOver, onmouseout:onMouseOut'>
                ${greeting}
            </span>

            <script type="dojo/method" event="onMouseOver" args="evt">
                dojo.addClass(this.domNode, 'hello_class');
                console.log("applied hello_class...");
                console.log(evt);
            </script>

            <script type="dojo/method" event="onMouseOut" args="evt">
                dojo.removeClass(this.domNode, 'hello_class');
                console.log("removed hello_class...");
                console.log(evt);
            </script>
        </div>

        <!-- теперь виджет можно включить в страницу как обычно -->
        <div dojoType="dtdg.HelloWorld"></div>
    </body>
</html>
```

Хотелось бы надеяться, что вы быстро поймете, какие *потрясающие* возможности предоставляет инструмент `Declaration` для быстрого создания примеров. Благодаря ему исключается необходимость создавать несколько файлов и отслеживать их изменения, объявлять пути к модулям и тратить время на решение других проблем, не связанных с основной задачей, – вы можете полностью сосредоточиться на решении поставленной задачи и делать свою работу максимально эффективно. В табл. 12.2 приводится краткое описание прикладного интерфейса `Declaration`.

*Таблица 12.2. Атрибуты класса Declaration*

| Атрибут | Комментарий |
|---|---|
| `widgetClass` | Класс виджета |
| `defaults` | Значения атрибутов, которые обычно передаются конструктору в виде параметров |
| `mixins` | Массив всех подмешиваемых классов предков |

Атрибут `mixins` класса `Declaration` при объявлении в разметке должен иметь тип `Array`. В этом заключается важное отличие от функции `dojo.declare`, которая позволяет указывать в качестве предка либо один объект, либо массив объектов.

Обычно после опробования идеи с помощью `Declaration` вам придется набело реализовать свои наработки, но в действительности не существует более быстрого способа проверить состоятельность своей идеи на скорую руку.

# В заключение

После прочтения этой главы вы должны:

- Понимать, как диджиты инкапсулируют HTML, CSS и JavaScript в единый переносимый модуль
- Понимать ключевые события жизненного цикла, доступ к которым предоставляется классом `_Widget` с помощью методов управления жизненным циклом, а также порядок их следования
- Понимать, как действует класс `_Templated` в качестве подмешиваемого предка к классу `_Widget` и какие дополнительные возможности поддержки шаблонов он обеспечивает
- Понимать различия в использовании свойств `templatePath` и `templateString` для виджетов с шаблонами
- Уметь безопасно манипулировать шаблоном диджита до того, как он появится на экране
- Уметь передавать параметры диджитам через шаблоны

- Уметь создавать и добавлять виджеты в страницу программным способом
- Знать, как реализовать в виджетах обработку таких событий DOM, как `onmouseover`
- Уметь использовать класс `Declaration` для быстрого создания прототипов диджитов в разметке

В следующей главе будут рассматриваться виджеты форм.

# 13

# Виджеты форм

В этой главе последовательно описываются различные диджиты, позволяющие создавать формы с фантастическим внешним видом, затрачивая минимальные усилия. Как и все остальное в библиотеке Dijit, элементы управления, описываемые в этой главе, могут целиком определяться в разметке, требуют минимального объема программного кода JavaScript и были разработаны с учетом требований к обеспечению доступности. Кроме того, следует заметить, что вы приступаете к чтению очень сложной главы. Модуль `dijit.form` обладает чрезвычайно широкими функциональными возможностями – диджиты форм составляют большинство объектно-ориентированных виджетов, входящих в состав инструментария, поэтому здесь вы встретитесь с более глубокими иерархиями классов, созданными посредством функции `dojo.declare`, чем в каком-либо другом месте этой книги.

## Обзор элементов управления форм

Спецификация HTML 4.01 (*http://www.w3.org/TR/html401/*) содержит исчерпывающие сведения о формах и сама по себе достойна внимательного изучения, тем не менее в этом разделе мы попытаемся рассмотреть наиболее полезные концепции, которые помогут вам извлечь максимум выгоды из этой главы. Раз вы читаете эту книгу, то можно с определенной долей уверенности предположить, что вам уже приходилось создавать свои собственные формы; поэтому нет никакого смысла напоминать вам, что *форма* – это коллекция, состоящая из одного или более элементов управления, предназначенных для ввода информации и передачи ее на сервер для последующей обработки.

Однако, важно отметить, что революция AJAX действительно видоизменила парадигму передачи данных на сервер. Ранее данные могли передаваться серверной процедуре, которая разбивала их на удобные

в использовании пары ключ/значение, использовала в своей работе полученные из них ассоциативные массивы и затем возвращала новую страницу, отражающую выбор, сделанный пользователем. Чтобы сделать процесс передачи данных более элегантным, форма могла бы включаться в тег `iframe`, чтобы избежать необходимости полной перезагрузки страницы. Однако теперь благодаря объекту `XMLHttpRequest` (XHR) имеется возможность асинхронной отправки небольших порций данных на сервер без необходимости явно отправлять форму на сервер или перезагружать страницу.

Конечно, объект XHR, AJAX и другие хитроумные приемы организации взаимодействий с пользователем не ликвидируют потребности в формах. Формы все еще остаются надежным, проверенным стандартом – они способны выполнять свои функции даже при отключенной поддержке JavaScript и играют важную роль в создании доступных реализаций. Практически всегда есть смысл предусматривать возможность деградации функциональных возможностей и обеспечивать высокую доступность при применении самых причудливых способов отправки данных на сервер. Проще говоря, это не вопрос выбора «формы или AJAX», это вопрос обеспечения консолидации «форм и AJAX».

Для начала рассмотрим пример 13.1, представляющий собой расширенную версию самой простой формы из главы 1.

*Пример 13.1. Простая форма*

```
<html>
    <head>
        <title>Register for Spam</title>

        <script type="text/javascript">
            function help() {
              var msg = "Basically, we want to sell your info to a 3rd party.";
                alert(msg);
                return false;
            }

            //простейшая проверка
            function validate() {
                var f = document.getElementById("registration_form");

               if (f.first.value==""||f.last.value==""||f.email.value==="") {
                    alert("All fields are required.");
                    return false;
                }

                return true;
            }
        </script>
    </head>
    <body>
        <p>Just Use the form below to sign-up for our great offers:</p>
        <form id="registration_form"
              method="POST"
```

```
                onsubmit="javascript:return validate()"
                action="http://localhost:8080/register/">

            First Name: <input type="text" name="first"/><br>
            Last Name: <input type="text" name="last"/><br>
            Your Email: <input type="text" name="email"/><br>
            <button type="submit">Sign Up!</button>
            <button type="reset">Reset</button>
            <button type="button" onclick="javascript:help()">Help</button>

        </form>
    </body>
</html>
```

Несмотря на всю свою простоту, эта форма обладает достаточным объемом функциональных возможностей и будет вести себя одинаково практически во всех типах броузеров, а объединение ее с таблицей стилей CSS может придать ей довольно привлекательный внешний вид. Здесь имеется даже кнопка Help (справка), которая даст пользователю возможность узнать, для чего *в действительности* предназначена эта форма. На стороне сервера обрабатывать эту форму мог бы простой сценарий, возможно, уже после того, как веб-сервер извлечет данные из именованных полей формы. Обработать эту форму мог бы простой сценарий CherryPy, который приводится в примере 13.2.

*Пример 13.2. Сценарий CherryPy, выполняющий обработку формы*

```
import cherrypy
class Content:
    """
    Процедура обработки формы.
    Доступ к именованным полям формы реализуется очень просто.
    """
    @cherrypy.expose
    def register(self, first=None, last=None, email=None):

        #здесь выполняется добавление информации о пользователе
        #в черный список...

        #отправить в ответ настроенную страницу html
        return """
        <html>
            <head><title>You're now on our spam list!</title></head>
            <body>
                <p>Congratulations %s %s, you're gonna get spammed!</p>
            </body>
        </html>
        """ % (first, last) #подстановка значений переменных

cherrypy.quickstart(Content())
```

Несмотря на всю простоту, предыдущий пример затрагивает некоторые основы, касающиеся использования форм:

- Элементы управления форм должны заключаться в тег `FORM`.

- Тег FORM практически всегда включает атрибуты `name`, `method`, `onsubmit`, `enctype` и `action`, предоставляющие информацию о том, как следует обрабатывать форму.
- Атрибут `onsubmit` предоставляет стандартный способ выполнения действий по проверке содержимого формы на стороне клиента. Если процедура проверки возвращает значение `false`, это препятствует отправке формы на сервер.
- Атрибут `action` содержит URL отправки формы.
- Поля формы, представляющие значимые состояния, должны включать атрибуты `name`, которые собираются серверными платформами в пары ключ/значение и передаются процедурам, выполняющим обработку отправленных форм.
- Формы изначально обеспечивают возможность доступа к ним при помощи клавиатуры – нажатие на клавишу табуляции вызывает перемещение между полями[1], а нажатие клавиши Enter – отправку формы. Хотя это и не было продемонстрировано в примере, следует помнить, что порядок навигации между полями формы можно изменить с помощью атрибута `tabindex`.
- Вообще существует большое разнообразие типов элементов управления, определяемых с помощью атрибута `type`. В данном конкретном примере демонстрируется использование кнопок трех разных типов: одна вызывает событие `onsubmit`, вторая выполняет сброс формы в исходное состояние и третья обрабатывает нестандартное действие.
- Отправка формы обязательно вызывает полную загрузку новой страницы, переданной сервером в ответ на получение формы. Если атрибут `action` отсутствует, необходимо предусмотреть выполнение дополнительных действий с помощью JavaScript или DHTML для подключения обработчиков событий DOM, таких как `onclick`.

На протяжении этой главы термин «атрибут» часто будет использоваться как для описания атрибутов форм, так и для описания атрибутов объектов. Однако применение термина должно быть очевидным из контекста обсуждения и не должно вызывать затруднений.

Несмотря на всю краткость этого обзора, хотелось бы надеяться, что он подготавливает почву для последующего обсуждения различных диджитов форм. За дополнительной справочной информацией о формах HTML можно порекомендовать обратиться к книге «HTML & XHTML: The Definitive Guide» Чака Маскиано (Chuck Musciano) и Билла Кеннеди (Bill Kennedi) (O'Reilly).

---

[1] Пользователям Firefox 2+ в MacOS X может потребоваться загрузить дополнение Configuration Mania со страницы *https://addons.mozilla.org/en-US/firefox/addon/4420*, чтобы обеспечить возможность перехода между кнопками посредством клавиши табуляции.

# Диджиты форм

Диджиты форм, которые явно обозначены как допустимые для использования в обычных формах HTML, определяемых с помощью тега `FORM`, являются частью пространства имен `dijit.form`. В этом разделе вам будут представлены все диджиты, включенные в это пространство имен, примеры программного кода и снимки с экрана, сделанные при использовании темы *tundra*. Но сначала есть смысл еще раз напомнить, что диджиты форм разрабатывались с учетом предположительной полной деградации функциональных возможностей и отвечают требованиям доступности. Проще говоря, они сохраняют свою функциональность даже при отключенной поддержке JavaScript, CSS и изображений, а также в случаях, когда клавиатура является единственным устройством ввода. Кроме того, в среде Windows атрибуты доступности поддерживают режим высокой контрастности и поддерживают устройства чтения с экрана.

На рис. 13.1 показана общая структура дерева наследования в модуле `dijit.form`. На этой диаграмме отсутствуют отдельные подмешиваемые

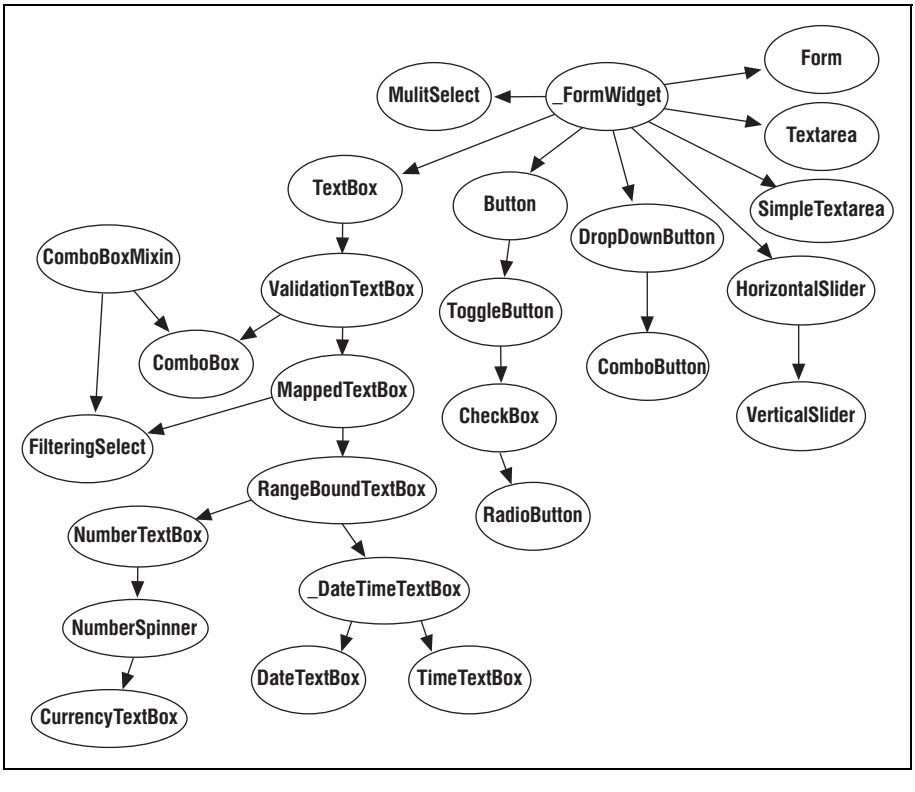



***Рис. 13.1.*** *Дерево наследования в модуле dijit.form*

классы, но хорошо отображается общая схема отношений между виджетами. Хотелось бы надеяться, что она поможет вам лучше понять, как располагается исходный программный код на диске, и использовать это знание, когда придет время поближе познакомиться с ним.

В дополнение к стандартным атрибутам диджитов, унаследованным от класса `_Widget`, таким как `domNode`, и обычным атрибутам HTML, включенным в спецификацию HTML 4.01, таким как `disabled` и `tabIndex`, все диджиты форм наследуют свойства базового класса, который явно поддерживает атрибуты, методы и точки расширения, перечисленные в табл. 13.1.

*Таблица 13.1. Поддерживаемые атрибуты, методы и точки расширения диджитов форм, унаследованные от класса _FormWidget*

| Имя | Тип данных | Категория | Комментарий |
|---|---|---|---|
| `value` | String | Атрибут | Текущее значение диджита. Имеет то же назначение, что и его одноименный эквивалент в языке HTML. |
| `name` | String | Атрибут | Имя значения диджита. Имеет то же назначение, что и его одноименный эквивалент в языке HTML. Удобно для использования процедурами обработки форм на стороне сервера. |
| `alt` | String | Атрибут | Альтернативный текст, который используется, когда броузер не может отобразить элемент, что довольно необычно для элементов форм и чаще используется для элементов изображений. Имеет то же назначение, что и его одноименный эквивалент в языке HTML. |
| `type` | String | Атрибут | Определяет тип элемента, когда существует несколько разновидностей. Например, элемент `button` может иметь тип `type="submit"`, в этом случае она вызывает событие `onsubmit`. Имеет то же назначение, что и его одноименный эквивалент в языке HTML. По умолчанию этот атрибут имеет значение `"text"`. |
| `tabIndex` | Integer | Атрибут | Используется для явного определения порядка перемещения между элементами формы с помощью клавиши табуляции. Имеет то же назначение, что и его одноименный эквивалент в языке HTML. По умолчанию этот атрибут имеет значение `"0"`. |
| `disabled` | Boolean | Атрибут | Делает элемент управления неактивным так, что он не может получить фокус ввода и при навигации с помощью клавиши табуляции пропускается. Не пытайтесь использовать этот атрибут в элементах, которые |

*Таблица 13.1 (продолжение)*

| Имя | Тип данных | Категория | Комментарий |
|---|---|---|---|
| | | | его не поддерживают. В число элементов, поддерживающих этот атрибут, согласно спецификации HTML 4.01 входят: button, input, optgroup, option, select и textarea. Значения элементов управления, находящихся в неактивном состоянии, не отправляются вместе с формой. По умолчанию этот атрибут имеет значение false. |
| readOnly | Boolean | Атрибут | Делает элемент управления недоступным для изменения, однако он по-прежнему может получать фокус ввода, доступен при навигации с помощью клавиши табуляции и его значение передается на сервер вместе с формой. Не пытайтесь использовать этот атрибут в элементах, которые его не поддерживают. В число элементов, поддерживающих этот атрибут, согласно спецификации HTML 4.01 входят: input и textarea. По умолчанию этот атрибут имеет значение false. |
| intermediateChanges | Boolean | Атрибут | Указывает, необходимо ли вызывать точку расширения onChange при каждом изменении значения. По умолчанию этот атрибут имеет значение false. |
| setAttribute (/* String */ attr, /* Any */ value) | Function | Метод | Обеспечивает корректный способ установки значения атрибута диджита. Например, записать в атрибут value значение "foo" можно было бы следующим способом: <dijit_name>.setAttribute("value", "foo"). |
| focus() | Function | Метод | Устанавливает фокус ввода в элемент управления, способный принимать его. |
| isFocusable() | Function | Метод | Возвращает значение, свидетельствующее о том, обладает ли элемент фокусом ввода. |

| Имя | Тип данных | Категория | Комментарий |
|---|---|---|---|
| forWaiValuenow() | Function | Точка расширения | По умолчанию возвращает текущее состояние виджета, которое может использоваться для определения значения состояния valuenow WAI-ARIA, устанавливающееся функциями dijit.removeState и dijit.setWaiState. |
| onChange(/* Any */ val) | Function | Точка расширения | Обеспечивает возможность назначения функции обратного вызова, которая будет вызываться при каждом изменении значения. |

Прекрасную документацию о формах HTML 4.01 в электронном виде можно найти по адресу *http://www.w3.org/TR/html401/interact/forms.html*.[1]

# Разновидности TextBox

Обычное поле ввода текстовой информации, безусловно, является наиболее часто используемым элементом форм. Несчетные часы потрачены программистами на реализацию форматирования и проверки информации, которая обычно представлена небольшими фрагментами текста, а вспомогательные сценарии, обеспечивающие поддержку полей ввода, вне всяких сомнений, можно считать самым распространенным шаблонным программным кодом, который когда-либо писался для поддержки веб-страниц. Если хотя бы одно из этих утверждений вызывает глубокий отклик в вашей душе, значит, вы оцените дары, приносимые семейством элементов TextBox.

Давайте рассмотрим каждого представителя семейства TextBox и улучшим пример формы, представленный ранее в этой главе. Большинство представителей этого семейства являются самыми обычными полями ввода текстовой информации, снабженными некоторыми дополнительными особенностями форматирования и предоставляющими вам возможность назначать свои собственные процедуры форматирования с помощью точек расширения format и parse. В списке, следующем ниже, приводятся все атрибуты и точки расширения элементов семейства TextBox. Технически элементы TextBox являются разновидностями элемента input, поэтому не забывайте, что к ним по-прежнему применимы стандартные атрибуты HTML, отсутствующие в списке.

[1] Перевод на русский язык: *http://www.umade.ru/resources/specifications/html401/forms.html*. – *Прим. перев.*

Атрибуты и точки расширения элементов TextBox наследуются всеми остальными диджитами семейства – знать их особенно важно, так как они находят широкое применение.

## TextBox

В табл. 13.2 приводится перечень основных свойств базового диджита TextBox.

*Таблица 13.2. Атрибуты и точки расширения элемента TextBox*

| Имя | Категория | Комментарий |
|---|---|---|
| trim | Атрибут | Указывает, следует ли удалять начальные и конечные пробельные символы. Значение по умолчанию: false. |
| uppercase | Атрибут | Указывает, следует ли преобразовывать все символы в верхний регистр. Значение по умолчанию: false. |
| lowercase | Атрибут | Указывает, следует ли преобразовывать все символы в нижний регистр. Значение по умолчанию: false. |
| propercase | Атрибут | Указывает, следует ли преобразовывать первый символ каждого слова в верхний регистр. Значение по умолчанию: false. |
| maxLength | Атрибут | Используется для передачи значения через стандартный атрибут HTML maxLength. Значение по умолчанию: "". |
| format(/* String */ value, /*Object*/constraints) | Точка расширения | Позволяет указывать свою функцию преобразования значения в правильно отформатированную строку. Реализация по умолчанию возвращает результат вызова метода toString, если таковой имеется, в противном случае возвращает неформатированное значение. Для значений null и undefined возвращает пустую строку. |
| parse(/* String */ value) | Точка расширения | Может использоваться, чтобы указать свою функцию преобразования строки в значение, которая имеется у всех диджитов форм и вызывается при попытке получить значение диджита. Реализация по умолчанию возвращает простое строковое значение. |

| Имя | Категория | Комментарий |
|---|---|---|
| setValue(/*String*/value) | Метод | Используется для записи строкового значения в диджиты класса TextBox или любого из его подклассов. Для этой иерархии подклассов не следует использовать метод setAttribute('value', /*...*/) класса _FormWidget. |
| getValue() | Метод | Используется для получения строкового значения из диджитов класса TextBox или любого из его подклассов. Не обращайтесь к свойству value напрямую, минуя этот метод. |

Начиная с версии 1.1 методы setValue и getValue класса _FormWidget были объявлены устаревшими и было рекомендовано использовать метод setAttribute('value', /*...*/) для изменения значения и свойство .value – для получения значения. Однако, класс TextBox, его подклассы и некоторые другие диджиты возвращают законность использования методов setValue и getValue. То есть методы setValue и getValue должны использоваться только для получения *значений виджетов*. Например, диджит TextBox имеет значение (следовательно, при работе с ним следует использовать методы setValue и getValue), тогда как при работе с другими диджитами, такими как Button, следует использовать метод setAttribute, потому что они не имеют значения виджета, несмотря на то, что им соответствуют значения, отправляемые вместе с формой.

Чтобы продемонстрировать наиболее типичное использование, в примере 13.3 к нашему предыдущему примеру формы были добавлены некоторые текстовые поля и включены атрибуты propercase и trim для полей first и last.

*Пример 13.3. Форма, дополненная полями TextBox и применением темы*

```
<html>
    <head>
        <title>Register for Spam</title>

        <link rel="stylesheet" type="text/css"
         href="http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css" />
        <link rel="stylesheet" type="text/css"
         href="http://o.aolcdn.com/dojo/1.1/dijit/themes/tundra/tundra.css" />

        <script
            djConfig="parseOnLoad:true",
            type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js">
        </script>
```

```
        <script type="text/javascript">
            dojo.require("dojo.parser");
            dojo.require("dijit.form.TextBox");

            function help() {
              var msg="Basically, we want to sell your info to a 3rd party.";
               alert(msg);
               return false;
            }

            //простая проверка
            function validate() {
                var f = document.getElementById("registration_form");
               if (f.first.value==""||f.last.value==""||f.email.value==="") {
                    alert("All fields are required.");
                    return false;
                }

                return true;
            }
        </script>
    </head>
    <body class="tundra">
        <p>Just Use the form below to sign-up for our great offers:</p>
        <form id="registration_form"
              method="POST"
              onsubmit="javascript:return validate()"
              action="http://localhost:8080/register/">
            First Name:
                <input dojoType="dijit.form.TextBox" propercase=true
                       trim=true name="first"><br>
            Last Name:
                <input dojoType="dijit.form.TextBox" propercase=true
                       trim=true name="last"><br>
            Your Email:
                <input dojoType="dijit.form.TextBox"
                       length=25 name="email"><br>
                <button type="submit">Sign Up!</button>
                <button type="reset">Reset</button>
                <button type="button"
                        onclick="javascript:help()">Help</button>
        </form>
    </body>
</html>
```

Если попытаться использовать диджиты без подключения файла *dojo.css* и без подходящей темы, они по-прежнему будут выполнять свои функции, но при этом будут иметь ужасный внешний вид. Обычная ошибка новичков, приводящая их в недоумение, заключается в том, что они забывают загрузить таблицы стилей CSS или определить соответствующее значение для атрибута `class` в теге `BODY`.

Помимо улучшенного внешнего вида диджит TextBox поможет сэкономить около десятка строк программного кода JavaScript. Конечно, вы можете переопределить точку расширения format, реализовав свою функцию форматирования и присвоив ее атрибуту format. Например, следующая функция форматирования принимает строку и выполняет *ЧеРеДоВаНиЕ РеГиСтРа СиМвОлОв*:

```
function mixedCapitalization(value) {
    var newValue = "";
    var upper = true;

    dojo.forEach(value.toLowerCase(), function(x) {
        if (upper)
            newValue += x.toUpperCase();
        else
            newValue += x;

        upper = !upper;
    });

    return newValue;
}
```

Задействовать эту функцию для работы совместно с диджитом TextBox очень просто:

```
<input dojoType="dijit.form.TextBox" format="mixedCapitalization"
       trim=true name="first">
```

Если при взаимодействии с формой вызвать появление события blur, переместив указатель мыши за ее пределы, можно увидеть, как выполняется это преобразование. Точно так же, как и format, можно переопределить функцию parse, которая стандартизует возвращаемое значение диджита. Одной из типичных операций является преобразование числовых типов в значения типа Number или стандартизация строковых значений.

Функции, назначенные точкам расширения format и parse, вызываются всякий раз, когда вызываются операции setValue и getValue, а не просто в ответ на взаимодействие пользователя с формой.

## ValidationTextBox

Еще одна навязшая в зубах проблема – функция проверки, – гарантирующая, что поле будет заполнено, и в действительности не сильно соответствующая своему названию, потому что не обеспечивает надежную проверку корректности адреса электронной почты. Здесь вам на помощь придет ValidationTextBox!

В табл. 13.3 приводится полный перечень дополнительных функциональных возможностей, предлагаемых диджитом ValidationTextBox.

*Таблица 13.3. Атрибуты диджита ValidationTextBox*

| Имя | Тип | Комментарий |
|---|---|---|
| required | Boolean | Атрибут, определяющий, является ли поле обязательным. Если поле ввода с этим атрибутом оставить незаполненным, оно не пройдет проверку. Значение по умолчанию: false. |
| promtMessage | String | Атрибут, определяющий текст подсказки, которая выводится при наведении указателя мыши на поле. |
| invalidMessage | String | Атрибут, определяющий текст сообщения, которое выводится, если поле не пройдет проверку. |
| constraints | Object | Атрибут с объектом пользователя, который может определять (даже динамически в случае необходимости) ограничения для атрибута regExpGen. Этот атрибут широко используется в других диджитах, таких как DateTextBox, для определения собственных форматов отображения. |
| regExp | String | Атрибут, содержащий регулярное выражение, используемое для проверки. Если вы определяете атрибут regExpGen, то этот атрибут задаваться не должен. Значение по умолчанию: ".*" (регулярное выражение, допускающее появление любых символов в любом месте). |
| regExpGen | Function | Атрибут, определяющий пользовательскую функцию, применяемую для генерирования регулярных выражений, опирающихся на пары ключ/значения в атрибуте constraints, что бывает удобно для организации динамических форм. Если вы определяете атрибут regExp, то этот атрибут определяться не должен. Значение по умолчанию: функция, возвращающая строку ".*" (регулярное выражение, допускающее появление любых символов в любом месте). |
| tooltipPosition | Array | Атрибут, определяющий, где должна выводиться всплывающая подсказка – выше, ниже, левее или правее элемента управления. По умолчанию этот атрибут возвращает значение dijit.Tooltip.defaultPosition, которое определено внутри виджета dijit.Tooltip. |
| isValid() | Function | Метод, который вызывается точкой расширения validator для выполнения проверки. Возвращает значение типа Boolean. |

| Имя | Тип | Комментарий |
|---|---|---|
| validator(/* String */ value, /* Object */ constraints) | Function | Точка расширения, которая вызывается по событиям DOM onblur, oninit и onkeypress. |
| displayMessage(/* String */ message) | Function | Точка расширения, которая может переопределяться в случае реализации собственного механизма вывода сообщений об ошибках проверки или подсказок. По умолчанию используется dijit.Tooltip. |

Диджит dijit.Tooltip описывается в главе 15.

Приспособить ValidationTextBox для использования в нашем примере очень просто, достаточно добавить к различным элементам управления атрибут require и добавить дополнительное регулярное выражение для проверки адреса электронной почты. Изменения, которые приводятся в примере 13.4, – это добавление виджетов ValidationTextBox и устранение всего программного кода JavaScript, который был написан ранее. Кнопка Help (справка) также была удалена, потому что та же самая задача более элегантно решается с помощью всплывающих подсказок.

*Пример 13.4. Форма, измененная под использование виджетов ValidationTextBox*

```
<html>
    <head>
        <title>Register for Spam</title>

        <link rel="stylesheet" type="text/css"
         href="http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css" />
        <link rel="stylesheet" type="text/css"
        href="http://o.aolcdn.com/dojo/1.1/dijit/themes/tundra/tundra.css" />

        <script
            djConfig="parseOnLoad:true",
            type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js">
        </script>

        <script type="text/javascript">
            dojo.require("dojo.parser");
            dojo.require("dijit.form.ValidationTextBox");
        </script>

        <!-- была удалена масса уродливого программного кода JavaScript -->

    </head>
```

```
    <body class="tundra">
        <p>Just Use the form below to sign-up for our great offers:</p>
        <form id="registration_form"
              method="POST"
              action="http://localhost:8080/register/">

              First Name:
                <input dojoType="dijit.form.ValidationTextBox"
                    properCase="true" trim=true required="true"
                    invalidMessage="Required." name="first"><br>

              Last Name:
                <input dojoType="dijit.form.ValidationTextBox"
                    properCase="true" trim=true required="true"
                    invalidMessage="Required." name="last"><br>

              Your Email:
                <input dojoType="dijit.form.ValidationTextBox"
                    promptMessage="Basically, we want to sell your info to
                                                  a 3rd party."
                    regExp="[a-z0-9._%+-]+@[a-z0-9-]+\.[a-z]{2,4}"
                    required name="email"><br>

              <button type="submit">Sign Up!</button>
              <button type="reset">Reset</button>
            <!-- кнопку Help заменили всплывающие подсказки -->
        </form>
    </body>
</html>
```

Затратив незначительные усилия, вы неожиданно получили более широкие функциональные возможности, попутно улучшив привлекательность интерфейса, а объем программного кода при этом уменьшился.

Теперь нам необходимо проделать некоторую работу с кнопками, что мы и сделаем в следующем разделе, но сначала познакомимся с остальными членами семейства TextBox.

## MappedTextBox и RangeBoundTextBox

Существует еще два замечательных класса диджитов форм, о которых пока не говорилось в этой главе, — это MappedTextBox и RangeBoundTextBox. Класс MappedTextBox предоставляет несколько методов, выполняющих сериализацию данных в строку посредством собственного метода toString, а класс RangeBoundTextBox упрощает возможность ограничивать значение допустимым диапазоном, который определяется свойствами min и max объекта constraints. Интуитивно может показаться, что класс ValidationTextBox должен предусматривать решение таких задач, как проверка диапазона, однако это не так, поскольку ValidationTextBox выполняет проверку строковых значений с помощью регулярного выражения, а класс RangeBoundTextBox явно обрабатывает числовые типы.

Проще говоря, эти два класса являются промежуточными механизмами, которые используются для нужд остальных диджитов форм, описываемых в этой главе, и в значительной степени предназначены для упрощения внутренней реализации. Знать о существовании этих двух классов необходимо, так как они могут пригодиться при создании специфических форм, но в действительности они имеют весьма ограниченный круг применений.

## TimeTextBox и DateTextBox

Специализированные процедуры проверки дат и времени – еще одна разновидность задач, которую приходилось решать практически каждому веб-разработчику в тот или иной момент времени. Несмотря на то что у дат и времени имеются четко определенные форматы представления, тем не менее слишком универсальный элемент `INPUT` языка разметки HTML не предлагает их поддержки, поэтому для решения проблемы проверки правильности приходится писать программный код JavaScript. К счастью, библиотека Dijit предполагает максимально простую возможность ввода даты и времени. Эти диджиты предусматривают представление дат и времени с учетом большинства наиболее распространенных региональных настроек, а добавить новые настройки, не входящие в стандартный набор, не составляет труда.

Диджиты `TimeTextBox` и `DateTextBox` применяют Григорианский календарь, который используется по умолчанию всеми средствами модуля `dojo.date`.

Предположим, что вместо забрасывания вас электронными письмами некая организация решила досаждать вам после тяжелого трудового дня, добиваясь вас по домашнему телефону. Естественно, чтобы не перетрудиться, бесплодно пытаясь дозвониться до вас в ваше отсутствие, они должны вытащить из вашей головы информацию о том, когда вы будете дома. Допустим, что они оказались настолько грамотными, чтобы использовать инструментарий Dojo и сэкономить на разработке программного обеспечения; тогда они могли бы предусмотреть такие поля в форме:

```
<!-- Не забудьте с помощью dojo.require подключить эти диджиты! -->

Best Day to call:
<input dojoType="dijit.form.DateTextBox"><br>

Best Time to call:
<input dojoType="dijit.form.TimeTextBox"><br>
```

Вот и все! Для этого не потребовалось никаких дополнительных усилий. При получении фокуса ввода диджит `DateTextBox` автоматически показывает небольшой привлекательный календарь для выбора даты, как показано на рис. 13.2, а диджит `TimeTextBox` – прокручиваемый

***Рис. 13.2.*** *Диджит DateTextBox автоматически показывает небольшой привлекательный календарь*

***Рис. 13.3.*** *Диджит TimeTextBox выводит прокручиваемый список со значениями времени*

список со значениями времени с 15-минутными интервалами, как показано на рис. 13.3.

Точно так же легко можно создать эти диджиты и программным способом:

```
var t = new dijit.form.TimeTextBox();
var d = new dijit.form.DateTextBox();

/* теперь их следует поместить на страницу с помощью их атрибутов domNode */
```

Кроме того что они просты в использовании, эти диджиты позволяют также реализовать собственное форматирование отображаемых значений, для чего можно использовать модуль `dojo.date`, применяемый этими диджитами в своей работе. В частности, для воспроизведения

### Zulu, Гринвич и Григорианский календарь... Что это?

В этой главе упоминается множество терминов, на поиск объяснений которых у вас, возможно, нет времени. Ниже дается краткое их описание, которое частично удовлетворит ваше любопытство.

- Zulu обозначает универсальное скоординированное время (Coordinated Universal Time, UTC), которое используется в части Западной Европы и отсчитывается от Гринвичской обсерватории в Лондоне. Этот часовой пояс исторически обозначался символом Z, которому в фонетическом алфавите, применяемом в вооруженных силах и в других организациях при чтении алфавитно-цифровых символов (чтобы не перепутать с символами, имеющими похожее звучание), соответствует слово Zulu.
- Среднее гринвичское время (Greenwich Mean Time, GMT) – по сути то же, что и UTC, когда речь идет о часовых поясах.
- Григорианский календарь получил такое название в честь папы Григория XIII, который провел реформу Юлианского календаря в конце 16 века (установленного Юлием Цезарем при реформе Римского календаря в первом веке нашей эры), в котором год был немного длиннее, из-за чего возникали проблемы при вычислении даты Христианской Пасхи.

соответствующих эффектов можно использовать свойства `formatLength`, `timePattern` и `datePattern` объекта `constraints`.

В табл. 13.4 и табл. 13.5 приводится перечень доступных возможностей. Обычно в зависимости от желаемой степени детализации изменяется либо длина формата, либо шаблон представления даты или времени.

*Таблица 13.4. Атрибуты диджита DateTextBox*

| Атрибут | Комментарий |
|---|---|
| `formatLength` | Используется для форматирования значений с учетом региональных настроек. Допустимыми значениями являются: `full`, `long`, `medium` и `short`. Существует возможность указывать специфические значения для отдельных регионов. Например, для *en-us* используются следующие форматы представления:<br>`full`<br>Thursday, January 10, 2008<br>`long`<br>January 10, 2008<br>`medium`<br>Jan 10, 2008<br>`short` (по умолчанию)<br>1/16/2008 |

*Таблица 13.4 (продолжение)*

| Атрибут | Комментарий |
|---|---|
| datePattern | Используется для реализации собственного форматирования независимо от региональных настроек. Принимает строку, сформированную в соответствии с соглашениями для языка Java. Подробности вы найдете по адресу *http://www.w3.org/TR/NOTE-datetime*. Типичные примеры приводятся ниже:<br>yyyy<br>2008<br>yyyy-MM<br>2008-01<br>MMM dd, yyyy<br>Jan 08, 2008 |
| strict | Когда установлено значение true, допускаются некоторые отступления в обозначениях и количестве пробелов. Значение по умолчанию: false. |
| locale | Для отдельных виджетов позволяет переопределить региональные настройки, используемые по умолчанию. Не забудьте настроить дополнительные регионы с помощью параметра djConfig.extraLocale, в противном случае вы будете получать сообщения об ошибках или непредсказуемые результаты. |
| selector | При отправке формы значение атрибута selector определяет, что следует отправлять – дату, время или и то и другое вместе, хотя на экране отображается либо только дата, либо только время. По умолчанию передаются оба значения. Чтобы отправить только дату или только время следует указать значение date или time, соответственно. |

*Таблица 13.5. Атрибуты диджита TimeTextBox*

| Атрибут | Комментарий |
|---|---|
| clickableIncrement | Строка, представляющая шаг увеличения временного значения во всех полях окна установки значения времени, которые управляются щелчком мыши. Это значение должно указываться без обозначения часового пояса и целое число раз укладываться в интервал, определяемый атрибутом visibleIncrement. Например, значение по умолчанию, "T00:15:00", соответствует 15-минутным интервалам. |
| visibleIncrement | Строка, представляющая значение временного интервала, через который будет обновляться отображение текущего времени. Значению по умолчанию "T01::00:00" соответствуют часовые интервалы. Это значение должно указываться без обозначения часового пояса. |

| Атрибут | Комментарий |
|---|---|
| visibleRange | Строка, представляющая отображаемый интервал времени. Значение по умолчанию, "T05:00:00", соответствует пяти часам. Это значение должно указываться без обозначения часового пояса. |
| formatLength | Строковое значение, используемое для форматирования с учетом региональных настроек. Допустимыми значениями являются long и short. Существует возможность указывать специфические значения для отдельных регионов. Например, для *en-us* используются следующие форматы представления:<br>long<br>10:00:00PM CST<br>short<br>10:00 PM |
| timePattern | Используется для реализации собственного форматирования независимо от региональных настроек. Принимает строку, сформированную в соответствии с соглашениями для языка Java. Подробности вы найдете по адресу *http://www.w3.org/TR/NOTE-datetime*. Типичные примеры приводятся ниже:<br>hh:mm<br>08:00<br>h:mm<br>8:00<br>h:mm a<br>8:00 PM<br>HH:mm<br>22:00<br>hh:mm:ss<br>08:00:00<br>hh:mm:ss.SSS<br>08:00:00.000 |
| strict | Когда установлено значение true, допускаются некоторые отступления в обозначениях (например, am и a.m. и т. д.) и количестве пробелов. Значение по умолчанию: false. |
| locale | Для отдельных виджетов позволяет переопределить региональные настройки, используемые по умолчанию. Не забудьте настроить дополнительные регионы с помощью параметра djConfig.extraLocale, в противном случае вы будете получать сообщения об ошибках или непредсказуемые результаты. |
| selector | При отправке формы значение атрибута selector определяет, что следует отправлять – дату, время или и то и другое вместе, хотя на экране отображается либо только дата, либо только время. По умолчанию передаются оба значения. Чтобы отправить только дату или только время следует указать значение date или time, соответственно. |

Объект constraints передается в разметке, как любой другой атрибут:

```
<input constraints="{datePattern:'MMM dd, yyyy'}"
       dojoType="dijit.form.DateTextBox">
```

Точно так же просто он передается программным способом:

```
var d = new dijit.form.DateTextBox({datePattern:'MMM dd, yyyy'});
```

## Общие особенности DateTextBox и TimeTextBox

У диджитов DateTextBox и TimeTextBox имеются два дополнительных метода: getDisplayedValue и setDisplayedValue. Расхождения между этими методами и обычными методами getValue и setValue обусловлены различием между тем, что фактически отображается в диджите, и тем типом данных, который используется внутри диджита. Внутри диджиты DateTextBox и TimeTextBox используют объект JavaScript типа Date, и получение этого объекта Date и является способом получения значения.

Напомню, что механизмы, унаследованные от класса RangeBoundTextBox, также позволяют указывать значения min и max, которые удобно использовать, чтобы предотвратить выбор недопустимых значений. Например, ограничить возможность выбора даты диапазоном между 1 декабря 2007 года и 30 июня 2008 года можно следующим образом:

```
<input constraints="{min:'2007-12', max:'2008-06',
       datePattern:'MMM dd, yyyy'}" dojoType="dijit.form.DateTextBox">
```

Кроме того, диджит MappedTextBox обладает средствами сериализации данных с помощью метода toString. Благодаря этому в случае необходимости вы можете получать строку в формате, соответствующем стандарту ISO-8601, что может быть очень удобно для отправки данных на сервер.

Важно понимать обособленность свойств datePattern и timePattern с одной стороны и спецификацией ISO-8601 – с другой: фактически они никак не связаны между собой. Значения datePattern и timePattern используются для закулисного манипулирования форматированием значения виджета, тогда как форматирование в соответствии со стандартом ISO-8601 используется парсером для передачи значения серверу.

Что касается двух дополнительных методов, getDisplayedValue и setDisplayedValue, предоставляемых этими двумя диджитами: метод setDisplayedValue производит тот же эффект, что и метод setAttribute('value', /*...*/), а метод getDisplayedValue возвращает значение, отображаемое диджитом, тогда как при обращении к свойству .value возвращается объект JavaScript типа Date.

В табл. 13.6 приводится краткое описание дополнительных особенностей, предоставляемых обоими диджитами – DateTextBox и TimeTextBox.

*Таблица 13.6. Общие особенности диджитов DateTextBox и TimeTextBox*

| Имя | Комментарий |
|---|---|
| getDisplayedValue() | Возвращает форматированное значение, которое фактически отображается элементом формы, тогда как метод getValue возвращает фактический объект Date. |
| setDisplayedValue(/*Date*/ date) | Устанавливает отображаемое и внутреннее значение диджита. (Тот же самый эффект может быть достигнут с помощью метода setValue) |
| toString() | Возвращает дату или время в виде строки, сформированной в соответствии со стандартом ISO-8601. |
| значения min и max объекта constraints | Позволяет ограничивать диапазон значений, доступных через всплывающий элемент выбора. |
| serialize() | Точка расширения, которая может использоваться для предоставления нестандартной реализации метода toString. Эта точка расширения манипулирует представлением значения, отправляемым серверу при передаче формы. |

## Сериализация данных перед отправкой серверу

Как оказывается, точка расширения serialize может оказаться особенно полезной при организации обмена данными с серверным компонентом, который ожидает получить дату, отформатированную определенным образом. Так, в примере 13.5 приводится программный код, расширяющий диджит DateTextBox и обеспечивающий возможность форматирования при вызове метода toString. В этом примере демонстрируется, как организовать отправку значения диджита DateTextBox в формате, отличном от того, что используется для отображения на экране.

*Пример 13.5. Сериализация значения DateTextBox перед отправкой серверу*

```
<html>
    <head>
        <title>Custom DateTextBox</title>

        <link rel="stylesheet" type="text/css"
         href="http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css" />
        <link rel="stylesheet" type="text/css"
        href="http://o.aolcdn.com/dojo/1.1/dijit/themes/tundra/tundra.css" />

        <script
            djConfig="parseOnLoad:false",
```

```
            type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js">
        </script>

        <script type="text/javascript">
            dojo.require("dojo.parser");
            dojo.require("dijit.form.DateTextBox");
            dojo.addOnLoad(function() {
                dojo.declare("dtdg.CustomDateTextBox",
                             [dijit.form.DateTextBox], {
                    serialize: function(d, options) {
                        return dojo.date.locale.format(d,
                        {
                            selector:'date',
                            datePattern:'dd-MMM-yyyy'}).toUpperCase();
                        }
                });
                dojo.parser.parse(dojo.body());
            });
        </script>
    </head>
    <body class="tundra">
        <form action="http://localhost:8080" type="POST">
            <input dojoType="dtdg.CustomDateTextBox" name="customDate"/>
            <input type="submit" value="Submit"/>
        </form>
    </body>
</html>
```

Ниже приводится минимальный класс CherryPy, способный принимать и отображать эту форму:

```
import cherrypy

class Content:
    @cherrypy.expose
    def index(self, **kwargs):
        return str(kwargs)

cherrypy.quickstart(Content())
```

## Не забывайте об унаследованных свойствах

Несмотря на то что иерархия наследования к этому моменту стала немного более глубокой, не стоит забывать, что все методы, которые определяются классами `TextBox` и `ValidationTextBox`, по-прежнему остаются доступными и являются основными во многих случаях. Кроме того, полезно будет освежить некоторые подробности о модуле `dojo.date`, связанные с этими диджитами, вернувшись к разделу с его описанием в главе 6.

## NumberTextBox

Класс NumberTextbox наследует все замечательные особенности класса RangeBoundTextBox и его предков и дополняет их использованием средств модуля dojo.number, предназначенных для работы с числовыми типами. Проще говоря, он представляет числовое значение, применяя форматирование с учетом текущих региональных настроек, и позволяет вводить ограничения, перечисленные в табл. 13.7

*Таблица 13.7. Ограничения в NumberTextBox*

| Имя | Комментарий |
|---|---|
| min и max | Используются для проверки границ допустимых значений, так же как и в других наследниках RangeBoundTextBox. |
| pattern | Используется для указания обязательного числа цифр после десятичной точки, а также для определения дополнительных элементов форматирования, таких как завершающий знак процента. |
| type | Используется для указания типа значения — decimal или percentage. |
| places | Используется для ограничения числа цифр после десятичной точки (это ограничение при его наличии переопределяет ограничение pattern). |

Например, чтобы гарантировать, что число будет иметь ровно два знака после десятичной точки и после него будет добавляться знак процента, можно использовать следующий прием:

```
<input constraints="{pattern: '#.##%'}" dojoType="dijit.form.NumberTextBox">
```

Хотя перед десятичной точкой указан только один знак решетки, на его месте может находиться большее число цифр. Однако, если появление цифр перед десятичной точкой нежелательно, можно определить значение для ограничения pattern без ведущего символа решетки: {pattern:'.##%'}. Кроме того, следует заметить, что с началом редактирования отображаемое значение автоматически преобразуется в обычное число, а по завершении редактирования происходит обратное преобразование в форматированное представление.

Напомню, что модуль dojo.number, как описывалось в главе 6, является универсальным средоточием десятков средств форматирования чисел и сопутствующих операций. Диджит NumberTextBox напрямую зависит от этих средств.

## NumberSpinner

Диджит NumberSpinner был рассмотрен в главе 11 и представляет собой аналог NumberTextBox с двумя маленькими кнопками с краю, который позволяет увеличивать или уменьшать значение с определенным шагом. Кнопки предусматривают работу в режиме *автоповтора*, то есть при нажатии и удержании они продолжают воздействовать на значение.

Кроме того, в диджите NumberSpinner ограничения min и max действуют несколько иначе, то есть при наличии этих ограничений кнопки диджита NumberSpinner не позволят преодолеть установленные границы.

Атрибуты, предоставляемые диджитом `NumberSpinner`, перечислены в табл. 13.8.

*Таблица 13.8. Атрибуты диджита NumberSpinner*

| Имя | Комментарий |
|---|---|
| `defaultTimeout` | Число миллисекунд, в течение которых должна удерживаться нажатой клавиша или кнопка, прежде чем будет включен режим автоповтора. Значение по умолчанию: `500`. |
| `timeoutChangeRate` | Множитель, используемый для изменения значения таймера автоповтора. Значение `1.0` означает, что события автоповтора будут запускаться через один и тот же интервал `defaultTimeout`. Если значение меньше `1.0`, тогда каждое последующее событие будет происходить немножко раньше пропорционально этому значению. Значение по умолчанию: `0.90`. |
| `smallDelta` | Шаг изменения счетчика положения при использовании клавиш управления курсором или кнопок. Значение по умолчанию: `1`. |
| `largeDelta` | Шаг изменения счетчика положения при использовании клавиш Page Up и Page Down. Значение по умолчанию: `10`. |

Создается диджит `NumberSpinner` так же просто, как и любые другие диджиты:

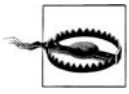
```
<input dojoType="dijit.form.NumberSpinner" smallDelta="2" largeDelta="4"
constraints="{min:100,max:120}" value="100">
```

## CurrencyTextBox

Диджит `CurrencyTextBox` является самым дальним потомком общего предка `NumberTextBox` и для форматирования значений использует средства модуля `dojo.currency`.

Однако, этот диджит предоставляет только один дополнительный атрибут – `currency`, который определяет форматирование с учетом региональных настроек. Значением атрибута `currency` должна быть одна из трехсимвольных последовательностей, определяемых стандартом обозначений валют ISO-4217, ознакомиться с которыми можно по адресу *http://en.wikipedia.org/wiki/ISO_4217*.[1]

[1] Версия этой страницы на русском языке (*http://ru.wikipedia.org/wiki/ISO_4217*) еще далека от завершения. – *Прим. перев.*



Всякий раз, когда для обозначения символов будут использоваться символы национального алфавита, у вас будет возникать потребность определить кодировку, используемую броузером, чтобы обеспечить корректное отображение всех символов. Всегда существует вероятность, что веб-сервер не будет включать эту информацию в заголовок страницы.

Стандартный способ определить кодировку в страницах HTML заключается в использовании специального тега META, помещаемого в заголовок страницы, и проект Dijit приветствует использование этого приема как наиболее лучшего метода. В следующем примере приводится тег META для набора символов UTF-8, что практически всегда является лучшим выбором:

```
<META http-equiv="Content-Type"
content="text/html; charset=UTF-8"/>
```

Обратите внимание: начиная с версии 1.1 вам необходимо применять этот тег, если вы используете версию Dojo от AOL CDN, потому что в настоящий момент этот сервер не включает информацию о кодировке символов в заголовки HTTP, что также позволило бы достигнуть требуемого результата. (В противном случае обозначения валют и некоторые символы юникода могут отображаться некорректно.)

Следующий фрагмент иллюстрирует применение диджита CurrencyTextBox для отображения денежных сумм в долларах США, когда требуется явно указывать количество центов через десятичную точку, для чего используется ограничение fractional – единственное ограничение, предоставляемое этим диджитом, помимо тех, что унаследованы от предков:

```
<input dojoType="dijit.form.CurrencyTextBox"
constraints="{min:1,max:100,fractional:true}" currency="USD"/>
```

Как и в диджите NumberTextBox, значение этого диджита с началом редактирования превращается в обычное числовое значение, а по завершении редактирования происходит обратное преобразование в форматированное представление.

## ComboBox

Диджит ComboBox – это раскрывающийся список значений, очень напоминающий элемент SELECT в языке разметки HTML, только в отличие от последнего ComboBox основан на обычном элементе input, поэтому, если требуемое значение отсутствует в списке предлагаемых вариантов, его можно ввести с клавиатуры. Диджит ComboBox наследует свойства и методы ValidationTextBox, благодаря чему вы получаете полную палитру возможностей выполнения проверки правильности. Из дополнительных возможностей – ComboBox предоставляет возможность фильтровать список вариантов в соответствии с учетом уже введенных символов. Список вариантов может быть статическим, определенным *заранее*,

или динамическим, который может извлекаться с сервера с помощью механизмов `dojo.data`.

В самом простом случае диджит `ComboBox` может использоваться для предоставления статического списка значений с возможностью ввода пользователем своего собственного значения. Следующий фрагмент демонстрирует применение статического списка с активированной функцией автодополнения.

```
<select name="coffee" dojoType="dijit.form.ComboBox" autoComplete="true">
    <option>Verona</option>
    <option>French Roast</option>
    <option>Breakfast Blend</option>
    <option selected>Sumatra</option>

    <script type="dojo/method" event="onChange" args="newValue">
        console.log("value changed to ", newValue);
    </script>
</select>
```

## Несколько слов о кодировке символов

UCS (Universal Character Set – универсальный набор символов) – это международный стандарт, утвержденный как международной организацией по стандартизации (International Organization for Standardization, ISO), так и международной электротехнической комиссией (International Electrotechnical Commission, IEC), в котором определено более 100 000 символов и соответствующих им уникальных чисел, называемых «кодами символов».

Юникод (Unicode) – это отраслевой стандарт, который разрабатывался совместно с UCS и может рассматриваться как реализация стандарта UCS или его основополагающая идея, хотя точные взаимоотношения между этими двумя стандартами неоднозначны и часто являются предметом философских дебатов. Самое главное, что эти два стандарта синхронизируются между собой.

Кодировки занимают очень важное положение, потому что от информационных систем все чаще и чаще требуется корректная обработка символов языков, даже древних, не говоря уже о тех, что существуют в наше время, а благодаря стандарту Юникод ключевые игроки получают стандартную возможность обеспечивать максимальную совместимость. Вполне очевидно, что возможность общения имеет огромное значение для веб-серверов и броузеров, почтовых серверов и почтовых клиентов и т. д.

Несмотря на то, что 7-битовый американский стандартный код для обмена информацией (American Standard Code for Information Interchange, ASCII) прекрасно подходит для отображения символов английского алфавита, его никак не достаточно для представления национальных символов других языков. Кодировка UTF-8 (Unicode Transformation Format – формат преобразования Юникода) способна представить любой символ в стандарте Юникода и обеспечивает обратную совместимость со стандартом ASCII, поэтому она стала такой популярной и широко используется для представления веб-страниц, электронной почты и т. д.

Интересная особенность UTF-8, обеспечивающая обратную совместимость со стандартом ASCII, заключается в использовании кодов переменной длины, когда для представления одного символа может использоваться от одного до четырех 8-битовых байтов. Таким образом, для корректного отображения символов очень важно, чтобы либо веб-сервер идентифицировал используемую кодировку с помощью заголовков протокола HTTP, либо чтобы веб-страница содержала специальный тег `META` с определением кодировки символов. Не имея информации о кодировке, броузер не сможет правильно преобразовать поток байтов в отдельные символы.

Подробнее о Юникоде можно прочитать на сайте *http://www.unicode.org*.

Объединение диджита `ComboBox` с механизмом `ItemFileReadStore` выполняется очень просто, и для этого придется приложить усилий чуть больше, чем просто указать диджиту источник данных. Например, пусть у нас имеется источник данных, хранящий информацию о различных видах обжарки кофе с их описаниями, как показано ниже:

```
{identifier : "name",
    items : [
        {name : "Light Cinnamon", description : "Very light brown, dry ,
tastes like toasted grain with distinct sour tones, baked, bready"},
        {name : "Cinnamon", description : "Light brown and dry, still toasted
grain with distinct sour acidy tones"},

        ...еще множество видов...
    ]
}
```

Предположим, что вам потребовалось заполнить `ComboBox` значениями поля `name` и при выборе того или иного вида обжарки делать что-либо с его описанием. Решить эту задачу можно, как показано в примере 13.6.

*Пример 13.6. Диджит ComboBox в действии*

```
<html>
    <head>
        <title>Pick a coffee roast, any coffee roast</title>

        <link rel="stylesheet" type="text/css"
         href="http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css" />
        <link rel="stylesheet" type="text/css"
        href="http://o.aolcdn.com/dojo/1.1/dijit/themes/tundra/tundra.css" />

        <script
            djConfig="parseOnLoad:true",
            type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js">
        </script>

        <script type="text/javascript">
            dojo.require("dojo.parser");
            dojo.require("dojo.data.ItemFileReadStore");
            dojo.require("dijit.form.ComboBox");
            dojo.require("dijit.form.Button");
            dojo.require("dijit.form.Form");
        </script>
    </head>
    <body class="tundra">

        <div dojoType="dojo.data.ItemFileReadStore"
             jsId="coffeeStore" url="./coffee.json"></div>

        <form action="localhost" dojoType="dijit.form.Form">
            <select name="coffee" dojoType="dijit.form.ComboBox"
                    store="coffeeStore" searchAttr="name">

                <script type="dojo/method" event="onChange" args="newValue">
                    console.log("value changed to ", newValue);
                    var f = function(item) {
                        console.log("new description is ",
                                    coffeeStore.getValue(item, "description")
                        );
                    };
                    coffeeStore.fetchItemByIdentity(
                        {identity : newValue, onItem : f}
                    );
                </script>
            </select>
            <button dojoType="dijit.form.Button">Submit</button>
        </form>
    </body>
</html>
```

В этом фрагменте ComboBox соединяется с ItemFileReadStore через атрибут store, и посредством атрибута serachAttr диджиту ComboBox сообщается, значение какого поля должно отображаться. После этого, когда

происходит изменение, метод onChange диджита ComboBox определяет, какое значение было выбрано, и использует его для поиска соответствующего ему описания в хранилище.

Внутри ComboBox реализует лишь подмножество функций прикладного интерфейса dojo.data.Read/Notification, необходимых для его работы. В частности, реализованы следующие методы:

- getValue
- isItemLoaded
- fetch
- close
- getLabel
- getIdentity
- fetchItemByIdentity
- fetchSelectedItem

Для полноты описания в табл. 13.9 приводится список специфических атрибутов, доступных в ComboBox.

*Таблица 13.9. Атрибуты диджита ComboBox*

| Имя | Тип | Комментарий |
|---|---|---|
| item | Object | Текущий выбранный элемент списка. Значение по умолчанию: null. |
| pageSize | Integer | Количество результатов на одну страницу (через ключ count в методе извлечения данных механизма ItemFileReadStore). Удобно для обращения к объемным хранилищам данных. Значение по умолчанию: Infinity. |
| store | Object | Ссылка на механизм доступа к данным, такой как ItemFileReadStore. Значение по умолчанию: null. |
| query | Object | Запрос, который можно передать механизму доступа к данным, чтобы выполнить первоначальную фильтрацию, прежде чем продолжить отбирать данные на основе атрибута searchAttr и текущего введенного ключа. Значение по умолчанию: {}. |
| autoComplete | Boolean | Определяет, следует ли отображать список вариантов, подходящих для уже введенной последовательности символов (в качестве поискового критерия используется значение атрибута queryExpr). Значение по умолчанию: true. |
| searchDelay | Integer | Количество миллисекунд задержки от момента нажатия на клавишу до момента запуска процесса поиска значения. Значение по умолчанию: 100. |
| searchAttr | String | Шаблон поиска значений, которые должны отображаться. Значение по умолчанию: name. |

*Таблица 13.9 (продолжение)*

| Имя | Тип | Комментарий |
|---|---|---|
| queryExpr | String | Шаблон выражения запроса dojo.data. (Выражение по умолчанию отыскивает любые значения, начинающиеся с уже введенной последовательности символов.) Значение по умолчанию: "${0}*". |
| ignoreCase | Boolean | Определяет, является ли запрос чувствительным к регистру символов. Значение по умолчанию: true. |
| hasDownArrow | Boolean | Определяет, следует ли отображать стрелку, направленную вниз, в качестве индикатора раскрытия списка. Значение по умолчанию: true. |

# FilteringSelect

FilteringSelect — это расширенная версия обычного HTML-элемента select, которая предоставляет раскрывающийся список допустимых значений и отправляет наряду с отображаемыми значениями еще и скрытые значения. Несмотря на то что FilteringSelect выглядит похожим на ComboBox и имеет много аналогичных возможностей, включая возможность фильтрации списка по мере ввода текста и способность извлекать данные из хранилища, тем не менее этот диджит построен на основе HTML-элемента SELECT.

Между диджитами FilteringSelect и ComboBox существуют три важных отличия, о которых следует помнить:

- Диджит FilteringSelect строится на основе обычного элемента select, который – по событию submit – передает серверу скрытое значение, невидимое в элементе управления. Это важная особенность, потому что диджит FilteringSelect допускает возможность деградации функциональных возможностей вплоть до поведения обычного элемента SELECT.
- Диджит FilteringSelect наследует свойства и методы MappedTextBox (TextBox с возможностью сериализации значения), а не ValidationTextBox, потому что проверка правильности в данном случае не требуется, так как пользователь не имеет возможности вводить произвольный текст.
- Диджит FilteringSelect в качестве своего значения может отображать не только простой текст, но и разметку HTML. То есть он допускает возможность включать в отображаемый им текст произвольную разметку и даже изображения.

В дополнение к общим операциям модуля dijit.form, таким как getValue, setValue, getDisplayedValue, setDisplayedValue, и различным свойствам ComboBox, диджит FilteringSelect предоставляет два дополнительных атрибута и одну дополнительную функцию, которые перечислены в табл. 13.10.

*Таблица 13.10. Дополнительные особенности диджита FilteringSelect*

| Имя | Комментарий |
|---|---|
| labelAttr | Текст, отображаемый элементом управления. Если значение не определено, используется значение searchAttr. |
| labelType | Указывает, является ли отображаемый текст разметкой или обычным текстом. Допустимыми значениями являются: 'text' и 'html'. |
| labelFunc (/*Object*/ item, /*dojo.data.store*/ store) | Обработчик события, который вызывается при изменении отображаемого значения – возвращает значение, которое должно отображаться. |

# MultiSelect

Диджит MultiSelect – это простая обертка (с атрибутами, перечисленными в табл. 13.11) вокруг элемента SELECT (с атрибутом multi=true), который наследует класс _FormWidget. Основная причина, по которой этот диджит был включен в состав библиотеки Dijit, состоит в том, что он облегчает взаимодействие в оберткой dijit.Form (которая будет рассматриваться ниже в этой главе) и устраняет необходимость разрабатывать свой элемент SELECT.

*Таблица 13.11. MultiSelect*

| Имя | Комментарий |
|---|---|
| size | Число элементов, отображаемых на одной странице. Значение по умолчанию: 7. |
| addSelected(/*dijit.form.MultiSelect*/ select) | Перемещает выбранные узлы из другого диджита MultiSelect в этот диджит. |
| getSelected() | Возвращает выделенные узлы в данном виджете. |
| setValue(/*Array*/values) | Устанавливает значения всех узлов виджета в соответствии со значениями элементов массива values. |
| invertSelection(/*Boolean*/fireOnChange) | Инвертирует выделения. Если аргумент fireOnChange имеет значение true, возбуждается событие onChange. |

Так как MultiSelect является всего лишь тонкой оберткой вокруг эквивалентного элемента HTML, он не обладает большим числом особенностей, достойных упоминания. Определить MultiSelect в разметке можно, как показано в примере 13.7.

*Пример 13.7. Типичный диджит MultiSelect в разметке*

```
<select multiple="true" name="foo" dojoType="dijit.form.MultiSelect"
style="height:100px; width:100px; border:3px solid black;">

    <option value="TN" selected="true">Tennessee</option>
    <option value="VA">Virginia</option>
    <option value="WV">West Virginia</option>
    <option value="OH">Ohio</option>

</select>
```

# Разновидности Textarea

Элемент TEXTAREA традиционно отравлял жизнь веб-разработчиков, так как он занимает фиксированное пространство на странице, из-за чего порой приходится прибегать к черной магии для определения, какой объем пространства необходимо выделить для него, чтобы это было не в ущерб остальному содержимому, отображаемому на экране.

## Textarea

Диджит `Textarea` наследует класс `_FormWidget` и предоставляет все лучшее из обоих миров, т. к. он поддерживает стандартные атрибуты HTML, обычные для элемента `textarea`, и выглядит как элемент, допускающий изменение вертикального размера при фиксированной ширине. Прикладной интерфейс диджита `Textarea` прост, поскольку обычно вам будут требоваться только методы `setValue` и `getValue`. Точка расширения `onChange` может использоваться для назначения функции обратного вызова, которая будет запускаться в случае изменений:

```
<textarea dojoType="dijit.form.Textarea" style="width:300px">
    One fish, two fish...
</textarea>
```

## SimpleTextarea

Способность диджита `Textarea` изменяться в размерах – удобное свойство во многих случаях, но не в тех, когда полный размер диджита определяется объемлющим контейнером (например, диджитами компоновки, с которыми вы познакомитесь в главе 14). По этой причине был создан диджит `SimpleTextarea`. С практической точки зрения `SimpleTextarea` ведет себя точно так же, как и обычный элемент `TEXTAREA`, за исключением того, что он может изменять свои размеры в установленных пределах. При работе с `SimpleTextarea` можно использовать те же самые атрибуты, что и при работе с обычным элементом `TEXTAREA`, такие как `rows` и `cols`, и так же, как в случае с диджитом `Textarea`, применять методы `setValue` и `getValue` для манипулирования текстом.

# Разновидности Button

Библиотека Dijit содержит большое разнообразие диджитов, допускающих возможность деградации функциональных возможностей и способных заменить стандартные кнопки и флажки, таких как кнопки панелей инструментов. Для начала познакомимся с самой обычной кнопкой – диджитом `Button`, а затем перейдем к более сложным разновидностям.

## Button

На рис. 13.4 показана кнопка, а в табл. 13.12 перечислены основные особенности диджита `Button`, который наследует класс `_FormWidget`.

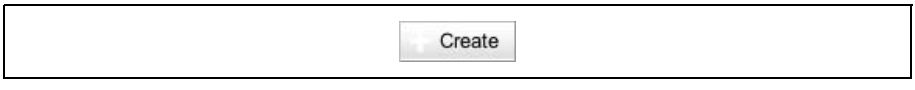



***Рис. 13.4.*** *Типичный диджит Button*

*Таблица 13.12. Свойства диджита Button*

| Имя | Комментарий |
|---|---|
| `label` | Используется для создания метки на кнопке в разметке или программным способом. |
| `showLabel` | Значение типа Boolean, которое указывает, следует ли отображать текстовую метку на кнопке. Значение по умолчанию: `true`. |
| `iconClass` | Класс, определяющий изображение, которое выводится на кнопке в виде ярлыка. |
| `onClick(/* DOM Event*/ evt)` | Точка расширения, которая вызывается в ответ на щелчок мышью. Обычно переопределяется в приложениях. |
| `setLabel(/* String */ label)` | Метод, принимающий разметку HTML, которая будет отображаться в качестве метки на кнопке. |

В отличие от `TextBox` и его наследников, виджеты `Button` требуют использовать функцию `setAttribute('value', /*...*/)`, унаследованную от класса `_FormWidget`, для установки значения, потому что, в отличие от других виджетов, кнопки не имеют *значения виджета*, которое отправлялось бы на сервер.

Давайте выбьем пыль из примера 13.4 и придадим ему блеск, заменив уродливые кнопки, как показано в примере 13.8. Не забудьте при этом добавить в заголовок страницы обязательную инструкцию `dojo.require("dojo.form.Button")`. Сама замена выполняется очень просто. Обратите

внимание, насколько удобно в данной ситуации определяется обработчик события onClick в разметке.

*Пример 13.8. Типичный пример использования диджита Button*

```
<button dojoTye="dijit.form.Button" type="submit">Sign Up!
    <script type="dojo/method" event="onClick" args="evt">
        alert("You just messed up...but it's too late now! Mwahahaha");
    </script>
</button>
<button dojoTye="dijit.form.Button" type="reset">Reset</button>
```

Особенно привлекательным в кнопках является атрибут iconClass, который не просто замещает кнопку ярлыком, а помещает изображение ярлыка на поверхность кнопки рядом с надписью, если значение атрибута label определено и атрибут showLabel имеет значение true. Например, если у вас имеется маленькая пиктограмма размером 20×20 пикселей, которую вам хотелось бы поместить на кнопку «Sign Up!» (зарегистрируйтесь), то в тег с определением кнопки можно было бы включить атрибут iconClass="spamIcon" и добавить в страницу следующий класс:

```
.spamIcon {
    background-image:url('spam.gif');
    background-repeat:no-repeat;
    height:20px;
    width:20px;
}
```

Конечно, вы можете определять любые классы CSS и придавать кнопкам любой желаемый внешний вид с помощью встроенных определений стиля в теге или с помощью классов.

## ToggleButton

Диджиты форм широко используют преимущества наследования и зачастую, чтобы обеспечить сходные функциональные возможности в классах-наследниках, они порождаются от общих предков. ToggleButton – один из таких классов. Он наследует свойства и методы класса Button и дополняет их возможностью, позволяющей кнопке иметь два состояния – включено и выключено, подобно диджитам RadioButton или CheckBox. Единственный значимый атрибут, который добавляет эта разновидность кнопок, – checked, значение которого можно изменять с помощью метода setAttribute.

Несмотря на то что в подобных случаях для переключения между двумя состояниями вы, вероятно, могли бы использовать более типичный элемент управления, такой как CheckBox, тем не менее вы можете воспользоваться диджитом ToggleButton непосредственно или создать от него свой подкласс и реализовать собственную версию ToggleButton. Точка расширения onChange (обычная для всех диджитов форм) является еще одной полезной особенностью:

```
<button dojoType="dijit.form.ToggleButton">
    <script type="dojo/method" event="onChange" args="newValue">
        console.log(newValue);
    </script>
</button>
```

Для создания многих кнопок на панели инструментов, например реализующих управление форматированием текста, таких как кнопки выбора курсивного стиля начертания, жирного, с подчеркиванием и т. д., используется диджит ToggleButton. Диджиты Menu и MenuItem будут представлены в главе 15.

Реализация некоторых диджитов кнопок не выделена в отдельные файлы ресурсов. В частности, если вы предполагаете использовать Button, ToggleButton, DropDownButton или ComboButton, вам необходимо будет добавить инструкцию dojo.require("dijit.form.Button"). На первый взгляд необходимость подключать одно, когда на самом деле используется другое, может показаться странной, тем не менее этому есть свое объяснение – реализации различных кнопок (опирающиеся на наследование) имеют столько общего, что они были объединены в одном физическом файле, чтобы уменьшить накладные расходы, связанные с получением ресурсов. Кроме того, не забывайте, что соответствие между классами, имитируемыми с помощью dojo.declare, и ресурсами не обязательно должно быть взаимнооднозначным (хотя традиционная философия объектно-ориентированного программирования часто предполагает это).

Такой подход остается источником споров в кругах разработчиков Dojo, так как накладные расходы, связанные с выполнением синхронных запросов к серверу можно было бы уменьшить за счет применения к окончательной версии приложения средств из библиотеки Util, выполняющих оптимизацию каждой страницы.

Такого рода нюансы являются результатом наличия многих (полных благих намерений) конкурирующих интересов в сообществе Dojo.

## CheckBox

CheckBox является прямым наследником ToggleButton и представляет собой стандартную замену обычного элемента <input type="checkbox">. Чтобы использовать этот диджит, достаточно подключить необходимый ресурс и добавить атрибут dojoType в нужный тег. Мы могли бы использовать этот диджит в странице из примера 13.4, активируя кнопку «Sign Up!» (зарегистрируйтесь) после щелчка пользователя на CheckBox в знак подтверждения того, что он знает о ваших тайных намерениях завалить его спамом:

```
<div name="confirmation" dojoType="dijit.form.CheckBox">
    <script type="dojo/method" event="onClick" args="evt">
        if (this.checked)
```

```
            dijit.byId("signup").setAttribute('disabled', false);
        else
            dijit.byId("signup").setAttribute('disabled', true);
    </script>
</div> I understand that you intend to spam me.<br>

<button id="signup" disabled dojoType="dijit.form.Button" type="submit">
    Sign Up!
</button>
```

На рис. 13.5 показано, как выглядят диджиты CheckBox.

***Рис. 13.5.*** *Группа диджитов CheckBox*

Причина, по которой вместо тега INPUT используется тег DIV, обусловлена невозможностью включить тег SCRIPT в тег INPUT, и, если вы попытаетесь сделать это, практически наверняка броузер проигнорирует его. Поэтому помните, что когда есть необходимость включить в диджит тег SCRIPT, вы не можете использовать тег INPUT. Если проблема деградации функциональных возможностей имеет настолько большое значение, что вы не можете допустить этого, то просто оформляйте свои методы в виде обычного JavaScript, а не в разметке.

Чтобы программно установить флажок, можно вызвать метод setValue(true), который запишет значение true не только в атрибут value диджита, но и в атрибут checked.

Если действительно важно гарантировать возможность деградации функциональных возможностей, можно пойти еще дальше и явно включить в тег обычные атрибуты HTML. Например, вместо простого объявления <input dojoType="dijit.form.CheckBox"/> можно добавить в тег дополнительный атрибут type: <input dojoType="dijit.form.CheckBox type="checkbox"/>.

Однако, как и у обычных элементов checkbox в языке разметки HTML, *состояние* и *значение* флажка являются различными понятиями. Состояние флажка может быть одним из двух – отмечен или не отмечен, а определить состояние можно с помощью атрибута checked. Однако в атрибуте value, когда флажок отмечен, при отправке формы серверу можно передавать любые, не только логические, значения. Например, тег <input name="pleaseSpamMe" value="yes"/> добавит в строку запроса параметр pleaseSpamMe=yes, если для отправки формы используется запрос типа GET. (По умолчанию атрибут value имеет значение "on".)

Тот факт, что метод getValue и атрибут value не обязательно возвращают одно и то же значение, может вызывать недоумение. Дело в том, что метод getValue возвращает значение, свидетельствующее о том, был ли отмечен флажок независимо от фактического значения атрибута value. Причина такого поведения состоит в том, что в большинстве случаев метод getValue используется для определения видимого состояния – включено/выключено, а не для получения фактического значения атрибута value, которое может не отражать состояние включено/выключено.

Поскольку в таких различиях, существующих между некоторыми возможностями, легко запутаться, рассмотрим несколько наиболее типичных примеров использования диджита CheckBox:

```
<input id="foo" dojoType="dijit.form.CheckBox"></input>
```

В примере 13.9 демонстрируется несколько вариантов манипулирования диджитом, а также даются комментарии, описывающие происходящее.

*Пример 13.9. Типичные примеры использования CheckBox*

```
/* Проверить начальное состояние */
dijit.byId("foo").checked    // false
dijit.byId("foo").getValue() // "on"

/* Вызвать setValue со значением true */
dijit.byId("foo").setValue(true) // отметить флажок и установить значение true
dijit.byId("foo").checked        // true
dijit.byId("foo").getValue()     // true

/* Вызвать setValue со значением false */
dijit.byId("foo").setValue(false)// снять флажок и установить значение false
dijit.byId("foo").checked        // false
dijit.byId("foo").getValue()     // false

/* Вызвать setValue со строковым аргументом */
dijit.byId("foo").setValue("bar")//отметить флажок и установить значение "bar"
dijit.byId("foo").checked        // true
dijit.byId("foo").getValue()     // "bar"
```

В большинстве случаев использования CheckBox метод setValue вызывается с логическим значением в качестве аргумента, поэтому велика вероятность, что вам не придется разбираться с таинственными эффектами, которые могут возникнуть при смешивании понятий состояния и значения.

Ниже приводится одна из особенно запутанных ситуаций, подчеркивающая некоторые из проблем, связанных со смешиванием понятий *состояние* и *значение*, о которых вам следует знать:

```
dijit.byId("foo").setAttribute("value", "foo")
// изменяется атрибут value, но не состояние флажка
```

```
dijit.byId("foo").value // "foo"
dijit.byId("foo").getValue()
//false, потому что флажок не установлен
```

Самым дезориентирующим здесь является то, что после установки значения вы не ожидаете, что метод `getValue()` вернет значение `false`, потому что согласно общей идиоме языка JavaScript, связанной с проверкой строкового значения, если строка не является пустой строкой (`""`), значением `null` или `undefined`, возвращается значение `true`. Однако, не следует забывать, что метод `getValue()` возвращает признак – установлен флажок или нет независимо от фактического значения атрибута `value`. В данном случае флажок не установлен, поэтому метод `getValue()` возвращает значение `false`.

Точно так же событие `onChange` не будет возникать в результате вызова метода `dijit.byId("foo").setAttribute("value", "foo")`, потому что видимое состояние флажка не изменилось.

## RadioButton

Диджит `RadiButton` является заменой эквивалентного элемента HTML и наследником `CheckBox`. Подобно своему эквиваленту в языке HTML, он концептуально представляет собой группу флажков, в которой только один может быть установлен в каждый конкретный момент времени. Вспомните, что все кнопки, объединенные в группу, имеют одно и то же значение атрибута `name`, но разные значения атрибута `value`. На рис. 13.6 показано, как выглядит группа диджитов `RadioButton`.

***Рис. 13.6.*** *Группа диджитов RadioButton*

Мы могли еще больше усовершенствовать свой пример (пример 13.4), спросив у пользователя, сколько раз в день мы могли бы его побеспокоить. Для достижения этой цели проще всего было бы использовать радиокнопки, как показано в примере 13.10, *не забыв сначала подключить к странице ресурс* `dijit.form.CheckBox`.

На первый взгляд создается ощущение, что в последнем предложении допущена опечатка, но это не так. Не забывайте, что функция `dojo.require` загружает ресурсы, а не отдельные диджиты. Оказывается, что ресурс `dijit.form.CheckBox` содержит как диджит `dijit.form.CheckBox`, так и `dijit.form.RadioButton`.

Это предупреждение стоит в одном ряду с приведенным ранее предупреждением, в котором говорилось о том, что ресурс `dijit.form.Button` содержит реализации нескольких диджитов.

*Пример 13.10. Типичный пример использования диджита RadioButton*

```
<input name="spamFrequency" value="1 per day"
       dojoType="dijit.form.RadioButton">
    1 per day<br>
<input name="spamFrequency" value="2 per day"
       dojoType="dijit.form.RadioButton">
    2 per day<br>
<input name="spamFrequency" value="3+ per day"
       dojoType="dijit.form.RadioButton">
    3+ per day<br>
```

## DropDownButton

DropDownButton — это простой потомок Button, который по щелчку мышью выводит раскрывающееся меню с пунктами, доступными для выбора. Такого рода кнопки часто можно увидеть на панелях инструментов. Диджиты DropDownButton и dijit.Menu тесно связаны между собой, так как Menu является одним из компонентов, часто используемых для реализации раскрывающихся списков; для этих же целей часто используется диджит TooltipDialog. На рис. 13.7 показано, как выглядит диджит DropDownButton.

***Рис. 13.7.*** *Диджит DropDownButton*

Более полное описание диджита Menu (и составляющих его отдельных пунктов MenuItem) приводится в главе 15. Однако пример 13.11, демонстрирующий диджит DropDownButton в действии, должен дать некоторое представление о нем. Обратите внимание, что первый дочерний элемент родительского узла DropDownButton является меткой, которая выводится на поверхности кнопки.

*Пример 13.11. Пример типичного использования DropDownMenu*

```
<button dojoType="dijit.form.DropDownButton">
    <span>Save...</span>
    <div dojoType="dijit.Menu">
        <div dojoType="dijit.MenuItem" label="Save">
            <script type="dojo/method" event="onClick" args="evt">
                console.log("you clicked", this.label);
            </script>
        </div>
        <div dojoType="dijit.MenuItem" label="Save as...">
            <script type="dojo/method" event="onClick" args="evt">
```

```
                console.log("you clicked", this.label);
            </script>
        </div>
        <div dojoType="dijit.MenuItem" label="Save to FTP...">
            <script type="dojo/method" event="onClick" args="evt">
                console.log("you clicked", this.label);
            </script>
        </div>
    </div>
</button>
```

Если вам потребуется, чтобы значение DropDownButton отправлялось вместе с формой, можно поступить следующим образом: создайте скрытый элемент INPUT и программно устанавливайте его значение из метода onClick диджита MenuItem. Однако в большинстве случаев DropDownButton используется исключительно в прикладных целях, например для сохранения документа.

Вообще, поля формы, которые передаются серверу при отправке формы, должны быть видимы пользователю в момент отправки. Поэтому может показаться, что модуль dijit.form — не место для DropDownButton, т. к. этот диджит не является *таким* элементом управления форм. Причина, по которой данный диджит рассматривается в этом разделе, в том, что он является потомком Button, и нет смысла рассматривать наследника Button в каком-нибудь другом месте.

## ComboButton

Диджит ComboButton является наследником DropDownButton, но обладает дополнительной особенностью: в нем имеется выделенная область, при щелчке по которой раскрывается меню, а щелчки на «остальных» участках изначально видимой кнопки вызывают действие по умолчанию. Например, с помощью этого диджита можно было бы реализовать кнопку «Save» (сохранить), которая по щелчку мышью запускает самую обычную операцию сохранения, а по щелчку на отведенной области с изображением стрелки вниз раскрывает меню с такими дополнительными пунктами, как «Save» (сохранить), «Save as...» (сохранить как...), «Save to FTP site» (сохранить на сервере FTP) и т. д. На рис. 13.8 показано, как выглядит кнопка ComboButton до открытия меню и после.

***Рис. 13.8.*** *Слева: кнопка ComboButton до щелчка на области открытия меню; справа: кнопка ComboButton после щелчка на области открытия меню*

Пример 13.12 иллюстрирует использование ComboButton.

*Пример 13.12. Пример типичного использования диджита ComboButton*

```
<button dojoType="dijit.form.ComboButton">
    <span>Save</span>

    <script type="dojo/method" event="onClick" args="evt">
        console.log("you clicked the button itself");
    </script>

    <div name="foo" dojoType="dijit.Menu">
        <div dojoType="dijit.MenuItem" label="Save">
            <script type="dojo/method" event="onClick" args="evt">
                console.log("you clicked", this.label);
            </script>
        </div>
        <div dojoType="dijit.MenuItem" label="Save As...">
            <script type="dojo/method" event="onClick" args="evt">
                console.log("you clicked", this.label);
            </script>
        </div>
        <div dojoType="dijit.MenuItem" label="Save to FTP...">
            <script type="dojo/method" event="onClick" args="evt">
                console.log("you clicked", this.label);
            </script>
        </div>
    </div>
</button>
```

Заметьте, что метка для кнопки ComboButton по-прежнему определяется через первый дочерний элемент; в данном случае это элемент Save, и раскрывающееся меню с пунктами определяются точно так же, как и в случае с диджитом DropDownButton.

# Slider

Движок – может, и не совсем «родной» элемент для HTML, но мало кто будет спорить о том, насколько полезны бегунки для создания наглядного интерфейса. Неважно, какова ваша конечная цель – корректировать степень прозрачности изображения, устанавливать глубину цвета или изменять размеры некоторого элемента управления на экране, – движок поможет решить эту задачу интуитивно понятным способом. Библиотека Dijit предлагает два вида движков – горизонтальный и вертикальный.

Диджит Slider содержит довольно интересное инженерное решение. Подобно некоторым другим диджитам он хранит текущее значение в виде скрытого значения формы, чтобы это значение передавалось на сервер вместе с формой, как и любое другое значение.

Чтобы подключить к странице все необходимые механизмы диджита Slider, достаточно просто выполнить инструкцию dojo.require("dijit.form.Slider"). При этом помимо VerticalSlider и HorizontalSlider вы также получите классы поддержки линеек и меток. Начнем с самого простого примера, а потом постепенно будем наращивать сложность, чтобы дать вам почувствовать, насколько гибким является этот фантастический маленький виджет.

## HorizontalSlider

Предположим, что, будучи любителем кофеина, вы захотели создать горизонтальный движок, с помощью которого можно регулировать уровень кофеина в различных напитках. Первая попытка добавления простенького движка в страницу может выглядеть, как показано в примере 13.13. Не забудьте при этом сначала подключить к странице ресурс dijit.form.Slider.

*Пример 13.13. HorizontalSlider (Дубль 1)*

```
<div dojoType="dijit.form.HorizontalSlider" name="caffeine"
     value="100"
     maximum="175"
     minimum="2"
     style="margin: 5px;width:300px; height: 20px;">

    <script type="dojo/method" event="onChange" args="newValue">
        console.log(newValue);
    </script>
</div>
```

В этом фрагменте создается простой движок без каких-либо меток; значения, которые можно получить, перемещая движок, лежат в диапазоне от 2 до 175, а размеры диджита определяются стилем. Значение по умолчанию равно 100, а при изменении значения метод onClick извлекает его и выводит в консоль. Обратите внимание, что щелчок мышью в пределах диджита приводит к перемещению бегунка в позицию щелчка и к соответствующему изменению значения. Пока все идет неплохо.

Для дальнейшего усовершенствования движка уберем кнопки, что располагаются по краям движка, добавив атрибут showButtons="false", и поместим сверху линейку HorizontalRule с метками HorizontalRuleLabels. Все, что необходимо, уже подключено к странице, поэтому нам не потребуется подключать какие-либо дополнительные ресурсы. Однако, мы подключим модуль dojo.number, чтобы обрести средства форматирования значений при выводе на консоль.

Просто добавьте в тело существующего движка дополнительную разметку, как показано в примере 13.14.

*Пример 13.14. HorizontalSlider (Дубль 2)*

```
<div dojoType="dijit.form.HorizontalSlider" name="caffeine"
     value="100"
     maximum="175"
     minimum="2"
     showButtons="false"
     style="margin: 5px;width:300px; height: 20px;">

    <script type="dojo/method" event="onChange" args="newValue">
        console.log(dojo.number.format(newValue,{places:1,pattern:'#mg'}));
    </script>

    <ol dojoType="dijit.form.HorizontalRuleLabels" container="topDecoration"
        style="height:10px;font-size:75%;color:gray;" count="6">
    </ol>

    <div dojoType="dijit.form.HorizontalRule" container="topDecoration"
        count=6 style="height:5px;">
    </div>
</div>
```

Вот и все! Теперь с дополнительными рисками и метками, обозначающими проценты, движок выглядит более привлекательно. Обратите внимание, что совсем не обязательно обеспечивать точное соответствие между количеством рисок на линейке и количеством меток, но в данном случае такой подход оказался вполне оправданным. Кроме того, в атрибуте `container` указано одно из значений перечисления – `topDecoration`, которое определяет местоположение линейки и меток.

Несмотря на то что движок уже получил линейку с процентами, было бы неплохо вывести некоторые данные в нижней части движка. Делается это точно так же, как и прежде, только вместо значения `topDecoration` будем использовать значение `bottomDecoration`. Однако мы не будем полагаться на возможности диджита вычислять числовые значения и определим содержимое списка сами, как показано в примере 13.15, включая явные теги `<br>` в метках, состоящих из нескольких слов, чтобы обеспечить привлекательность отображения. Результат приводится на рис. 13.9.

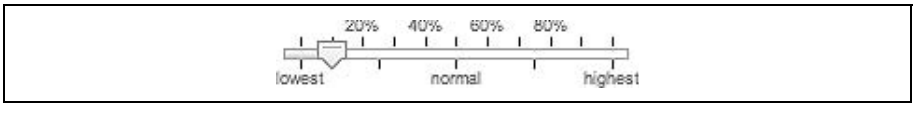



***Рис. 13.9.*** *HorizontalSlider*

*Пример 13.15. HorizontalSlider (Дубль 3)*

```
<div dojoType="dijit.form.HorizontalSlider" name="caffeine"
     value="100"
     maximum="175"
     minimum="2"
```

```
        showButtons="false"
        style="margin: 5px;width:300px; height: 20px;">

    <script type="dojo/method" event="onChange" args="newValue">
        console.log(newValue);
    </script>

    <ol dojoType="dijit.form.HorizontalRuleLabels" container="topDecoration"
        style="height:10px;font-size:75%;color:gray;" count="6">
    </ol>

    <div dojoType="dijit.form.HorizontalRule" container="topDecoration"
        count=6 style="height:5px;">
    </div>

    <div dojoType="dijit.form.HorizontalRule" container="bottomDecoration"
        count=5 style="height:5px;">
    </div>

    <ol dojoType="dijit.form.HorizontalRuleLabels"
        container="bottomDecoration"
        style="height:10px;font-size:75%;color:gray;">

        <li>green<br>tea</li>
        <li>coffee</li>
        <li>red<br>bull</li>
    </ol>
</div>
```

## VerticalSlider

Диджит `VerticalSlider` работает точно так же, как и `HorizontalSlider`, за исключением того, что он располагается вдоль оси $y$, а в качестве значения атрибута `container` для линеек и меток вместо значений `topDecoration` и `bottomDecoration` следует использовать `leftDecoration` и `rightDecoration`, а кроме того необходимо скорректировать стиль, указав размеры не по вертикали, а по горизонтали. В примере 13.6 определяется тот же самый движок, но расположенный по вертикали. Результат работы примера показан на рис. 13.10.

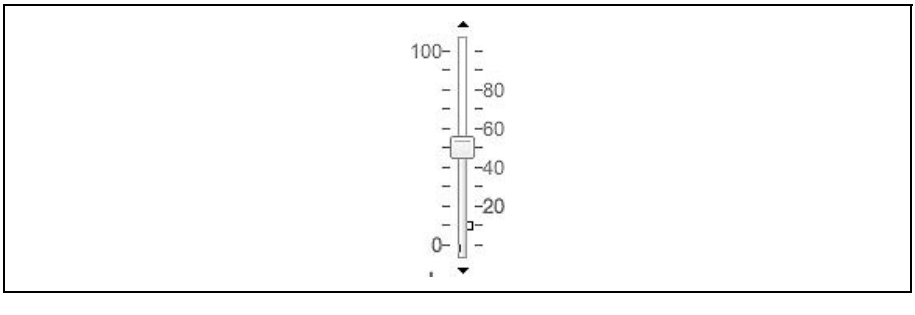



***Рис. 13.10.*** *VerticalSlider*

*Пример 13.16. VerticalSlider*

```
<div dojoType="dijit.form.VerticalSlider" name="caffeine"
     value="100"
     maximum="175"
     minimum="2"
     showButtons="false"
     style="margin: 5px;width:75px; height: 300px;">

    <script type="dojo/method" event="onChange" args="newValue">
        console.log(newValue);
    </script>
    <ol dojoType="dijit.form.VerticalRuleLabels" container="leftDecoration"
        style="height:300px;width:25px;font-size:75%;color:gray;" count="6">
    </ol>

    <div dojoType="dijit.form.VerticalRule" container="leftDecoration"
        count=6 style="height:300px;width:5px;">
    </div>

    <div dojoType="dijit.form.VerticalRule" container="rightDecoration"
        count=5 style="height:300px;width:5px;">
    </div>

    <ol dojoType="dijit.form.VerticalRuleLabels" container="rightDecoration"
        style="height:300px;width:25px;font-size:75%;color:gray;">
        <li>green tea</li>
        <li>coffee</li>
        <li>red bull</li>
    </ol>
</div>
```

В табл. 13.13, 13.14 и 13.15 перечислены наиболее важные особенности, входящие в состав ресурса `dijit.form.Slider`, а именно: сами движки, линейки и метки. Не забывайте, что все привычные механизмы форм, такие как `setValue` и прочие, наследуются и работают как обычно.

*Таблица 13.13. HorizontalSlider и VerticalSlider*

| Имя | Тип | Комментарий |
|---|---|---|
| `showButtons` | Boolean | Указывает, следует ли отображать кнопки инкремента/декремента на концах движка. Значение по умолчанию: `true`. |
| `minimum` | Integer | Минимальное допустимое значение. Значение по умолчанию: `0`. |
| `maximum` | Integer | Максимальное допустимое значение. Значение по умолчанию: `100`. |
| `discreteValues` | Integer | Число промежуточных значений между минимумом и максимумом (включая крайние значения). Значение `Infinity` определяет непрерывную шкалу. Значения больше 1 приводят к появлению эффекта дискретизации. Значение по умолчанию: `Infinity`. |

*Таблица 13.13 (продолжение)*

| Имя | Тип | Комментарий |
|---|---|---|
| pageIncrement | Integer | Шаг перемещения бегунка по нажатию клавиш Page Up и Page Down. Значение по умолчанию: 2. |
| clickSelect | Boolean | Указывает, следует ли перемещать бегунок в положение указателя мыши при щелчке на движке. Значение по умолчанию: true. |
| slideDuration | Number | Время в миллисекундах, за которое бегунок должен преодолевать расстояние от значения 0% до 100%. Удобно использовать при программном управлении значением движка. Значение по умолчанию: 1000. |
| increment() | Function | Увеличивает значение положения движка на единицу. |
| decrement() | Function | Уменьшает значение положения движка на единицу. |

*Таблица 13.14. HorizontalRule и VerticalRule*

| Имя | Тип | Комментарий |
|---|---|---|
| ruleStyle | String | Класс CSS, применяемый к отдельным штрихам. |
| count | Integer | Число штрихов. Значение по умолчанию: 3. |
| container | DOM Node | Положение меток относительно движка: topDecoration или bottomDecoration для HorizontalSlider. leftDecoration или rightDecoration для VerticalSlider. |

HorizontalRuleLabel и VerticalRuleLabel наследуют свойства и методы HorizontalRule и VerticalRule, соответственно.

*Таблица 13.15. HorizontalRuleLabel и VerticalRuleLabel*

| Имя | Тип | Комментарий |
|---|---|---|
| labelStyle | String | Класс CSS, применяемый к текстовым меткам. |
| labels | Array | Массив текстовых меток, которые будут равномерно располагаться по длине движка слева направо или сверху вниз. Значение по умолчанию: [ ]. |
| numericMargin | Integer | Количество числовых меток, которые не должны отображаться с обоих концов движка. (Полезно, когда необходимо опустить очевидные значения в начале и в конце, такие как 0, по умолчанию, и 100.) |
| minimum | Integer | Когда массив меток не определен, это значение указывает крайнюю левую метку, которая должна отображаться как метка. Значение по умолчанию 0. |

| Имя | Тип | Комментарий |
|---|---|---|
| maximum | Integer | Когда массив меток не определен, это значение определяет крайнюю правую метку, которая должна отображаться как метка. Значение по умолчанию: 1. |
| constraints | Object | Шаблон (для модуля dojo.number), используемый при генерировании числовых меток, когда массив labels не определен. Значение по умолчанию: {pattern:"#%"}. |
| getLabels | Function | Возвращает массив labels. |

# Form

Хотя диджиты форм могут быть обернуты тегом HTML form, библиотека Dijit предоставляет диджит dijit.form.Form, обеспечивающий некоторые дополнительные возможности, которые могут оказаться весьма полезны. Этот раздел завершает главу обзором обычных форм HTML и описывает специфические особенности диджита dijit.form.Form. Начинающие пользователи библиотеки Dijit часто путаются, ожидая, что с помощью библиотеки Dijit можно сделать нечто, что и так прекрасно работает в виде обычной разметки HTML. Не забывайте важную часть философии Dojo: не изобретать повторно веб-технологии, которые уже работают; вместо этого Dojo добавляет новые и дополняет существующие особенности там, где ощущается их недостаток или отсутствует стандартизация.

## Обзор тега form

Диджит dijit.form.Form корректно обслуживает все стандартные атрибуты тега form в соответствии со спецификацией HTML 4.01. Предполагается, что все значения атрибутов задаются как строковые значения, заключенные в кавычки, исключение составляют события DOM, такие как onclick, которые позволяют указывать действие в виде программного кода, например, onclick="javascript:someScriptAction()" или onclick="javascript:return someValidationAction()". Под событиями от мыши подразумевается событие щелчка левой кнопкой.

## Form

Диджит Form дополняет стандартный набор атрибутов тега form, добавляя несколько методов для прямого манипулирования самим диджитом и одну точку расширения, которая вызывается внутренней реализацией диджита в ответ на действие пользователя. Ключевые особенности диджита Form перечислены в табл. 13.16.

*Таблица 13.16. Методы и точки расширения диджита Form*

| Имя | Категория | Комментарий |
|---|---|---|
| getValues() | Метод | Возвращает структуру для формы в формате JSON, содержащую пары ключ/значение. |
| isValid() | Метод | Возвращает значение true, если методы isValid всех активных элементов формы вернули значение true. |
| setValues (/*Object*/ values) | Метод | Обеспечивает простой способ установки сразу всех значений в форме. Принимает структуру в формате JSON, в которой каждому ключу соответствует имя поля в форме. |
| submit() | Метод | Используется для отправки формы программным способом. |
| reset() | Метод | Последовательно вызывает методы reset() каждого диджита в форме для сброса их значений в начальное состояние. |
| onSubmit() | Точка расширения | Вызывается внутренней реализацией диджита при обращении к методу submit(). Эта точка расширения предоставляет возможность отменить отправку формы, для чего функция должна вернуть значение false. Реализация по умолчанию возвращает результат вызова метода isValid(). |
| validate() | Метод | Возвращает значение true, если проверка формы прошла успешно, то есть то же самое, что и метод isValid(), но кроме этого подсвечивает все диджиты формы, которые благополучно прошли проверку, и вызывает метод focus() первого диджита в форме, не прошедшего проверку. |

Обертывание всей формы диджитом dijit.form.Form выполняется точно так же, как замена любого другого элемента соответствующим диджитом, как показано в примере 13.17.

*Пример 13.17. Типичный пример использования диджита Form*

```
<form id="registration_form" dojoType="dijit.form.Form">

    <!-- здесь располагаются элементы формы -->

    <!-здесь выполняется переопределение точек расширения...-->

    <script type="dojo/method" event="onSubmit" args="evt">
        //вернуть false, если форма не должна отправляться.
        //По умолчанию onSubmit возвращает результат метода isValid()
        //диджита dijit.form.Form
    </script>
</form>
```

# В заключение

В этой главе были рассмотрены некоторые важные основы. После ее прочтения вы должны:

- Понимать принцип действия обычных форм HTML
- Понимать, как использовать диджиты, замещающие стандартные элементы форм
- Представлять общую классификацию диджитов форм, в общих чертах понимать взаимоотношения наследования между ними
- Уметь создавать диджиты форм как программно, так и в разметке
- Понимать различия между методами, атрибутами и точками расширения
- Понимать, что означает понятие *деградации функциональных возможностей* формы, и уметь учитывать различные факторы, связанные с разработкой форм, допускающих деградацию

А теперь можно перейти к изучению виджетов компоновки.

# 14

# Виджеты компоновки

К сожалению, при разработке веб-приложений невероятный объем времени тратится на всякие ухищрения с CSS, только чтобы добиться определенных схем размещения, которые на самом деле уже были реализованы множество раз и не должны вызывать сложностей. В этой главе будут представлены *диджиты компоновки* – множество удобных контейнеров, используемых для реализации наиболее типичных схем компоновки в разметке. Диджиты компоновки позволят вам автоматизировать такие типичные задачи, как воспроизводство схемы компоновки с вкладками или компоновки с произвольным мозаичным расположением элементов, без привлечения своих стилей CSS для плавающего содержимого, без необходимости вычислять относительные смещения и тому подобное. В отличие от предыдущей главы, где были представлены виджеты форм, эта глава существенно короче, проще и понятнее, потому что виджетов компоновки совсем немного; все они обладают лишь несколькими параметрами настройки, и в отношении их практически нет оговорок.

## Общие особенности диджитов компоновки

Все диджиты компоновки располагаются в пределах пространства имен `dijit.layout` и обладают общим набором основных особенностей, о которых вам следует знать. Помимо наследования классов `_Widget`, `_Container` и `_Contained`, все они обладают еще целым рядом общих особенностей. В этом разделе дан краткий обзор этих особенностей, сведенных в табл. 14.1, запомнить которые не составит никакого труда.

Особенно важным в табл. 14.1 являются взаимоотношения между методами `layout` и `resize`. Если говорить более точно, метод `resize` используется для изменения размеров виджета и это практически всегда является основанием для вызова метода `layout`, чтобы поменять размеры

дочерних виджетов. Проще говоря, дочерние узлы не занимаются собственным размещением. Родительский узел выполняет их размещение внутри своего метода layout. В обычной ситуации метод управления жизненным циклом startup вызывает метод resize, который в свою очередь вызывает метод layout.

*Таблица 14.1. Общие методы диджитов компоновки*

| Имя | Комментарий |
|---|---|
| isLayoutContainer | Возвращает признак, свидетельствующий о том, является ли виджет контейнером компоновки. |
| layout() | Переопределяется виджетами с целью изменять размеры и положение своего содержимого (дочерних виджетов). Вызывается после метода startup, когда уже точно будет создано все, что содержится в этом виджете, а также всякий раз, когда вызовом метода resize изменяется размер виджета. |
| resize(/*Object*/ size) | Используется, чтобы явно установить размеры виджета компоновки. Принимает объект, содержащий координаты левого верхнего угла, ширину и высоту, в виде: {w : Integer, h: Integer, l : Integer, t : Integer}. (Когда вы будете *переопределять* метод resize, не забывайте в своей реализации вызывать метод layout, потому что layout — это стандартное место, где выполняются изменения размеров и позиций дочерних виджетов.) |

Диджиты компоновки особенно широко используют особенности классов _Container и _Contained, поэтому есть смысл познакомиться и с ними. Список этих особенностей приводится в табл. 14.2.

*Таблица 14.2. Механизм контейнеров, используемый диджитами компоновки*

| Имя | Комментарий |
|---|---|
| removeChild(/*Object*/ dijit) | Удаляет дочерний виджет из контейнера. (Если виджет dijit не является дочерним или если контейнер вообще не имеет дочерних виджетов, это не приводит к появлению ошибки.) |
| addChild(/*Object*/ dijit, /*Integer?*/ insertIndex) | Добавляет дочерний виджет в контейнер; можно использовать необязательный аргумент insertIndex, чтобы указать место, где он должен быть вставлен. |
| getParent() | Обычно используется дочерними виджетами, чтобы получить ссылку на родительский контейнер. Возвращает экземпляр диджита. |
| getChildren() | Обычно используется контейнерным диджитом, чтобы получить список дочерних диджитов. Возвращает массив экземпляров диджитов. |

*Таблица 14.2 (продолжение)*

| Имя | Комментарий |
|---|---|
| getPreviousSibling() | Используется наследниками StackContainer, чтобы получить ссылку на предыдущий соседний диджит, то есть на тот, что «слева». Возвращает экземпляр диджита. |
| getNextSibling() | Используется наследниками StackContainer, чтобы получить ссылку на следующий соседний диджит, то есть на тот, что «справа». Возвращает экземпляр диджита. |

## Создание программным способом

Как будет показано в следующих ниже примерах, порядок создания диджитов компоновки ничем не отличается от порядка создания других диджитов, когда в первом аргументе конструктору передается набор свойств, а во втором – ссылка на исходный узел, предназначенный для диджита компоновки. Как только диджит компоновки будет создан, ссылка на исходный узел будет записана в свойство domNode диджита. Все эти действия выполняются посредством метода create, унаследованного от класса _Widget, который был представлен при обсуждении жизненного цикла диджитов в главе 12. Однако, в отличие от других диджитов, с которыми вы уже познакомились, при создании диджитов компоновки программным способом вам практически всегда придется явно вызывать метод startup, потому что эти диджиты практически всегда содержат дочерние виджеты, а метод startup сигнализирует, что контейнер закончил добавление дочерних виджетов и с этого момента можно приступать к их компоновке. В конце концов, для виджета не было бы никакого смысла просто располагать себя среди соседних элементов и периодически запускать процесс компоновки. Поэтому метод startup родительского виджета обычно вызывает метод startup каждого дочернего виджета, зажигая зеленый свет к началу отображения.

Если вы реализуете свой собственный контейнер, метод startup – последний шанс, где можно выполнить манипуляции над дочерними виджетами, прежде чем они появятся на экране.

## Поддержка клавиатуры

Подобно другим диджитам, виджеты компоновки обладают полноценной поддержкой клавиатуры. Позднее вы увидите, что практически всегда клавиши функционируют «очевидным» образом. Например, для навигации в AccordionPane можно использовать клавиши со стрелками вверх и вниз, а также клавиши Page Up и Page Down. Кроме обеспечения поддержки доступности, которая является одной из целей библиотеки Dijit, возможность широкого использования клавиатуры обеспечивает дополнительные удобства для пользователей.

# ContentPane

Диджит ContentPane является одним из основных диджитов компоновки: он реализует мозаичную схему размещения. Он наследует свойства и методы непосредственно от класса _Widget и концептуально напоминает навороченный элемент iframe, но в отличие от него привносит в страницу множество дополнительных особенностей, из которых далеко не последними является возможность отображать произвольные фрагменты HTML (а не только целые документы), перезагружать содержимое с помощью XHR, отображать виджеты и соблюдать тему страницы. Чаще всего диджит ContentPane располагается внутри других виджетов, таких как TabContainer, хотя в практике встречаются интересные способы применения его как самостоятельного диджита.

При наиболее типичном использовании диджит ContentPane не делает ничего особенного; это показано в примере 14.1.

*Пример 14.1. Создание диджита ContentPane в разметке*

```
<html>
    <head><title>Fun with ContentPane!</title>

        <link rel="stylesheet" type="text/css"
         href="http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css" />
        <link rel="stylesheet" type="text/css"
        href="http://o.aolcdn.com/dojo/1.1/dijit/themes/tundra/tundra.css" />

        <script
            djConfig="parseOnLoad:true",
            type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js">
        </script>

        <script type="text/javascript">
            dojo.require("dijit.layout.ContentPane");
        </script>
    </head>
    <body class="tundra">
        <div dojoType="dijit.layout.ContentPane">
            Nothing special going on here.
        </div>
    </body>
</html>
```

Просматривая исходный программный код или читая документацию о Dojo, вы можете заметить упоминание о диджите LinkPane. За время своего развития диджит ContentPane включил в себя многие особенности, присущие диджиту LinkPane, и поэтому весьма вероятно, что LinkPane будет объявлен устаревшим, так как предлагает незначительный объем функциональности по сравнению с ContentPane.

При этом вы можете без лишних сложностей загрузить произвольное содержимое с сервера и отобразить его на странице, для чего достаточно просто указать URL на стороне сервера. Предположим, что URL *foo* на стороне сервера ссылается на фрагмент текста, тогда для его отображения можно было бы использовать ContentPane, как показано ниже:

```
<div id="foo" preload="false" dojoType="dijit.layout.ContentPane" href="foo">
```

В данном конкретном случае при обращении к URL *foo* возвращается простое строковое значение, хотя вполне возможно организовать возврат виджета, который автоматически будет подвергнут парсингу и отображен на странице. Однако, следует помнить, что при этом виджет уже должен быть подключен к странице с помощью функции dojo.require. Например, предположим, что при обращении к URL возвращается , тогда по умолчанию этот диджит будет автоматически создан и отображен на странице.

В табл. 14.3 приводится перечень всего, что поддерживает ContentPane.

*Таблица 14.3. Прикладной интерфейс ContentPane*

| Имя | Тип | Комментарий |
|---|---|---|
| href | String | Определяет адрес внешних данных, которые должны быть загружены. |
| extractContent | Boolean | Если имеет значение true, извлекается видимое содержимое между тегами BODY документа, который получает ContentPane, и помещается на панель. Значение по умолчанию: false. |
| parseOnLoad | Boolean | Если имеет значение true, любые диджиты, присутствующие в загружаемом содержимом, автоматически будут создаваться и отображаться. Значение по умолчанию: true. |
| preventCache | Boolean | Действует точно так же, как параметр preventCache для dojo.xhrGet. Если имеет значение true, с каждым запросом передается дополнительный параметр, препятствующий попыткам извлекать содержимое из кэша. Значение по умолчанию: false. |
| preload | Boolean | Используется для принудительной загрузки содержимого в диджит в случае, если он изначально невидим. (Если для узла определен параметр стиля display:none, содержимое может не загружаться, если не установить атрибут в значение true.) Значение по умолчанию: false. |
| refreshOnShow | Boolean | Указывает, должен ли диджит перезагружать содержимое всякий раз, когда переходит из невидимого состояния в видимое. Значение по умолчанию: false. |

| Имя | Тип | Комментарий |
|---|---|---|
| loadingMessage | String | Определяется в dijit.nls.loading. Содержит сообщение, которое отображается пользователю в процессе загрузки. Значение по умолчанию: "Loading...". |
| errorMessage | String | Определяется в dijit.nls.loading. Содержит сообщение, которое отображается пользователю в случае ошибки. Значение по умолчанию: "Sorry, an error occurred". |
| isLoaded | Boolean | Указывает, было ли загружено содержимое. Полезно, когда содержимое обновляется достаточно часто. |
| refresh() | Function | Выполняет принудительное обновление содержимого, повторно загружая его. |
| setHref(/*String*/ href) | Function | Изменяет адрес местоположения внешнего содержимого для диджита. Если атрибут preload имеет значение false, содержимое не будет загружаться, пока виджет снова не станет видимым. |
| setContent(/*String \| DOMNode \| NodeList */data) | Function | Явно устанавливает локальное содержимое панели. |
| cancel() | Function | Прерывает процесс загрузки содержимого. |
| onLoad(/*Event*/evt) | Function | Точка расширения. Вызывается после завершения загрузки (и, возможно, парсинга виджетов). |
| onUnload(/*Event*/evt) | Function | Точка расширения. Вызывается перед тем, как содержимое будет очищено вызовом одного из методов: refresh, setHref или setContent. |
| onDownloadStart | Function | Точка расширения. Вызывается непосредственно перед началом загрузки содержимого. По умолчанию возвращает строку, которая будет отображаться в процессе загрузки. |
| onContentError(/*Error*/ e) | Function | Точка расширения. Вызывается, когда возникает ошибка DOM. Возвращает строку с сообщением об ошибке, которая будет отображаться для пользователя. |
| onDownloadError(/*Error*/ e) | Function | Точка расширения. Вызывается в случае появления ошибки во время загрузки. По умолчанию возвращает строку, которая будет отображаться в качестве сообщения об ошибке. |
| onDownloadEnd() | Function | Точка расширения. Вызывается по завершении процесса загрузки. |

## Расширение возможностей ContentPane

Одна из замечательных особенностей интерпретирующего, динамического языка программирования, такого как JavaScript, состоит в том, что при его использовании вы можете расширять функциональность программных компонентов прямо в процессе выполнения программы. Когда мы приводили пример, в котором с помощью ContentPane загружался такой диджит, как Textarea, то напоминали, что при этом необходимо предварительно подключить к странице ресурс с реализацией Textarea. Обычная рекомендация – добавить инструкцию dojo.require require("dijit.form.Textarea") где-нибудь в заголовке страницы, и под нее подпадает большинство случаев. Но как быть, если требуется, чтобы сервер мог вернуть *произвольный* диджит для отображения в ContentPane?

Это не проблема, если клиент запрашивает в строке запроса URL определенный диджит. Тогда вы можете просто вызвать функцию dojo.require во время выполнения программы, прежде чем производить обновление содержимого ContentPane. Если же сервер может вернуть произвольный, заранее неизвестный диджит, то дело несколько осложняется, но не настолько, чтобы это препятствие нельзя было преодолеть. Просто запишите в атрибут parseOnLoad диджита ContentPane значение false, чтобы предотвратить автоматический парсинг (в противном случае возникнет ошибка, поскольку соответствующая инструкция dojo.require еще не была выполнена), и в точке расширения onLoad отыщите узел с диджитом, подключите его и запустите парсинг вручную.

Ниже приводится пример реализации в разметке:

```
<div dojoType="dijit.layout.ContentPane"
     href="bar" parseOnLoad="false">
    <script type="dojo/method" event="onLoad">
        dojo.query("[dojoType]", this.domNode)
        .forEach(function(x) {
            var _resource = dojo.attr(x, "dojoType");
            dojo.require(_resource);
            //не выполнять парсинг, пока модуль не будет загружен
            var _interval = setInterval(function() {
                if (eval(_resource)) { //объект существует?
                    clearInterval(_interval);
                    dojo.parser.parse(x.parentNode);
                }
            }, 100);
        });
    </script>
</div>
```

Если предположить, что исходный узел имеет атрибут id со значением foo, то создать ContentPane программным способом можно было бы, как показано ниже:

```
var contentPane = new dijit.layout.ContentPane({ /* properties*/, "foo");
contentPane.startup(); // возьмите за правило всегда вызывать метод startup
```

Поскольку диджит ContentPane не является наследником класса _Container, в нем отсутствуют встроенные методы добавления дочерних диджитов. Однако, можно использовать ссылку domNode этого диджита, чтобы с ее помощью добавлять в диджит другие узлы, используя старый добрый JavaScript. Например, используя существующий диджит ContentPane, созданный в предыдущем примере:

```
contentPane.domNode.appendChild(someOtherDijit.domNode);
```

Вы можете задаться вопросом, почему ContentPane не поддерживает интерфейс класса _Container. Неофициальный ответ таков: диджит ContentPane вообще не должен предпринимать какие-либо конкретные действия, когда по каким-то причинам в него добавляется дочерний диджит. Причины, которые обусловливают необходимость добавления дочерних диджитов в ContentPane, могут быть самыми разными. Однако, если вам действительно это необходимо, вы можете с помощью функций mixin или extend дополнить ContentPane функциональными возможностями класса _Container.

# BorderContainer

BorderContainer — это новый диджит компоновки, появившийся в версии 1.1. Его появление привело к тому, что диджиты LayoutContainer и SplitContainer были объявлены устаревшими, так как BorderContainer по сути является объединением этих двух диджитов. Несмотря на то что в Интернете вам наверняка встретятся примеры использования LayoutContainer и SplitContainer, тем не менее лучше воздержаться от использования устаревших компонентов в своих приложениях. По этой причине указанные два виджета не рассматриваются в этой книге.

Диджит BorderContainer предоставляет удобную возможность определить одну из следующих схем размещения, состоящих из нескольких более простых неперекрывающихся элементов: сверху/снизу/слева/справа/в центре, сверху/снизу/в центре или слева/справа/в центре страницы. Можно разрешить изменение размеров этих элементов. BorderContainer является особенно ценным виджетом, т. к. он позволяет легко и просто превратить то, что в противном случае могло бы превратиться в изнурительную работу, в действительно простое решение на языке виджетов. Как вы уже наверняка догадались, он получил название контейнера (container) с «границами» (border), потому что по его

границам располагаются четыре неперекрывающихся элемента, окружающие то, что заполняет центр.

Прикладной интерфейс диджита приводится в табл. 14.4.

*Таблица 14.4. Прикладной интерфейс BorderContainer*

| Имя | Тип | Комментарий |
|---|---|---|
| design | String | Допустимыми значениями являются "headline" (по умолчанию) и "sidebar". Определяет, будут ли верхний и нижний элементы растягиваться на всю ширину контейнера или левый и правый элементы будут растягиваться на всю высоту контейнера. |
| liveSplitters | Boolean | Определяет, как будут изменяться размеры – синхронно с перемещением указателя мыши или по событию onmouseup. |
| persist | Boolean | Следует ли сохранять размеры элементов в виде cookie. |

При работе с диджитом BorderContainer можно использовать дополнительные атрибуты, доступные через ContentPane, на которые он опирается и которые перечислены в табл. 14.5.

Вам наверняка интересно будет узнать, что имеется в виду, когда говорится про дополнительные атрибуты, доступные через ContentPane. Дело в том, что в файле ресурса BorderContainer производится расширение прототипа класса _Widget за счет добавления этих свойств. Это достаточно остроумное решение, использующее динамическую природу JavaScript для обеспечения дополнительных возможностей по требованию, вместо того чтобы предусматривать решение проблемы *заранее*, что породило бы ненужные связи с реализацией ContentPane.

*Таблица 14.5. Атрибуты, доступные виджетам, дочерним по отношению к BorderContainer*

| Имя | Тип | Комментарий |
|---|---|---|
| minSize | Integer | Если определено, указывает минимальный размер в пикселях диджита ContentPane. Значение по умолчанию: 0. |
| maxSize | Integer | Если определено, указывает максимальный размер в пикселях диджита ContentPane. Значение по умолчанию: Infinity. |
| splitter | Boolean | Если имеет значение true, на границе ContentPane появляется область (splitter), ухватив которую мышью можно изменять размер ContentPane. Значение по умолчанию: false, то есть содержимое не может изменять размер. |
| region | String | Виджет BorderContainer в качестве элементов схемы компоновки использует виджеты ContentPane, в каждом из |

| Имя | Тип | Комментарий |
|---|---|---|
| | | которых должен быть определен атрибут region, определяющий, как располагается виджет. Допустимыми значениями являются: "top", "bottom", "left", "right" и "center". Значение по умолчанию: пустая строка. Допускается также использовать значения "leading" и "trailing", которые отличаются от значений "left" и "right" тем, что они имеют отношение к двунаправленным схемам размещения. |

Схема размещения, при которой верхняя и нижняя границы растягиваются на всю ширину контейнера, называется схемой *заголовка* (*headline*), а схема размещения, при которой левая и правая границы растягиваются на всю высоту контейнера, называется схемой *врезки* (*sidebar*). В любом из этих двух случаев компоновка может включать дополнительные элементы, что увеличивает число областей от трех до пяти. Но при любой схеме компоновки оставшееся пространство в центре занимает центральный элемент.

Давайте рассмотрим простой пример реализации схемы заголовка в разметке, как показано в примере 14.2. Вверху будет располагаться панель красного цвета, внизу – синего, а в середине – белого. Верхняя панель имеет минимальную высоту 10 пикселей и максимальную – 100 пикселей (это ее высота по умолчанию).

*Пример 14.2. Создание диджита BorderContainer в разметке*

```
<html>
    <head><title>Fun with BorderContainer!</title>

        <link rel="stylesheet" type="text/css"
         href="http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css" />
        <link rel="stylesheet" type="text/css"
        href="http://o.aolcdn.com/dojo/1.1/dijit/themes/tundra/tundra.css" />

        <script
            djConfig="parseOnLoad:true",
            type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js">
        </script>

        <script type="text/javascript">
            dojo.require("dijit.layout.ContentPane");
            dojo.require("dijit.layout.BorderContainer");
            dojo.require("dojo.parser");
        </script>
    </head>
    <body class="tundra">

        <div dojoType="dijit.layout.BorderContainer" design="headline"
             style="height:500px;width:500px;border:solid 3px;">

            <div dojoType="dijit.layout.ContentPane" region="top"
```

```
                    style="background-color:blue;height:100px;" splitter="true"
                    minSize=10 maxSize=100>top</div>

               <div dojoType="dijit.layout.ContentPane"
                    region="center">center</div>

               <div dojoType="dijit.layout.ContentPane" region="bottom"
                    style="background-color:red;height:100px;"
                    splitter="true">bottom</div>

          </div>
     </body>
</html>
```

Чтобы добавить два элемента, заполняющие левую и правую области, достаточно вставить два дополнительных диджита `ContentPane`, как показано в следующей ниже версии тега `BODY`. Получившийся результат приводится на рис. 14.1.

```
<body class="tundra">

    <div dojoType="dijit.layout.BorderContainer"
        design="headline" style="height:500px;width:500px;border:solid 3px;">

        <div dojoType="dijit.layout.ContentPane" region="top"
             style="background-color:blue;height:100px;" splitter="true"
             minSize=10 maxSize=100>top</div>

       <div dojoType="dijit.layout.ContentPane" region="center">center</div>

        <div dojoType="dijit.layout.ContentPane" region="bottom"
             style="background-color:red;height:100px;"
             splitter="true">bottom</div>
```

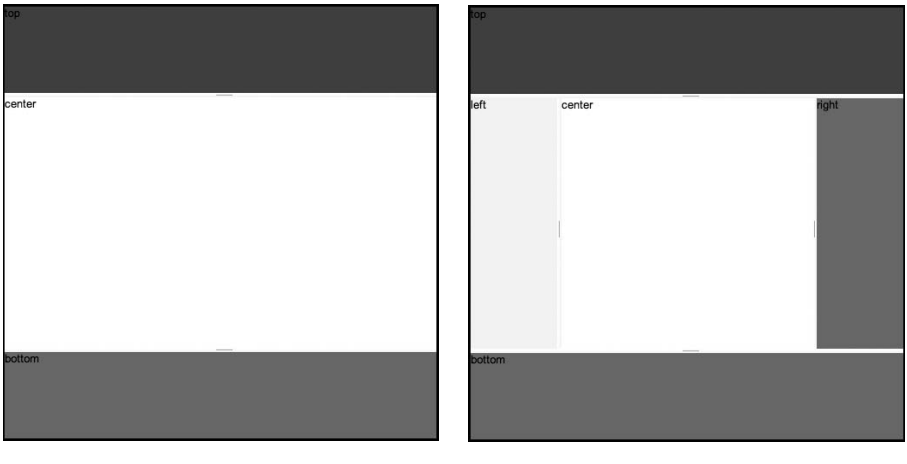



***Рис. 14.1.*** *Слева: BorderContainer перед добавлением дополнительных панелей слева и справа; справа: BorderContainer после добавления дополнительных панелей слева и справа*

```
        <div dojoType="dijit.layout.ContentPane" region="left"
             style="background-color:yellow;width:100px;"
             splitter="true">left</div>

        <div dojoType="dijit.layout.ContentPane" region="right"
             style="background-color:green;width:100px;"
             splitter="true">right</div>

    </div>
</body
```

Как и для всех других диджитов, создание диджита BorderContainer программным способом связано с вызовом того же самого базового конструктора, которому передаются список свойств и ссылка на исходный узел. Также и добавление дочерних диджитов ContentPane состоит в поочередном их создании, одного за другим. Хотя программный способ более утомителен, чем создание диджитов в разметке, тем не менее порядок создания остается одним и тем же. В примере 14.3 показано, как реализовать пример 14.2 программным способом.

*Пример 14.3. Создание BorderContainer программным способом*

```
<html>
    <head><title>Fun with BorderContainer!</title>

        <link rel="stylesheet" type="text/css"
         href="http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css" />
        <link rel="stylesheet" type="text/css"
        href="http://o.aolcdn.com/dojo/1.1/dijit/themes/tundra/tundra.css" />

        <script
            djConfig="parseOnLoad:true",
            type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js">
        </script>

        <script type="text/javascript">
            dojo.require("dijit.layout.BorderContainer");
            dojo.require("dijit.layout.ContentPane");
            dojo.require("dojo.parser");
            dojo.addOnLoad(function() {

                //BorderContainer
                var bc = new dijit.layout.BorderContainer(
                    {
                        design: "headline",
                        style: "height:500px;width:500px;border:solid 3px"
                    },
                    "bc"
                );

                var topContentPane = new dijit.layout.ContentPane(
                    {
                        region: "top",
                        style: "background-color:blue;height:100px;",
```

```
                splitter: true,
                minSize : 10,
                maxSize : 100
            },
            document.createElement("div")
        );

        var centerContentPane = new dijit.layout.ContentPane(
            {
                region: "center"
            },
            document.createElement("div")
        );

        var bottomContentPane = new dijit.layout.ContentPane(
            {
                region: "bottom",
                style: "background-color:red;height:100px;",
                splitter: true
            },
            document.createElement("div")
        );

        bc.startup(); // выполнить начальное размещение
                      //(несмотря на отсутствие дочерних виджетов)

        //Теперь добавить дочерний виджет.
        bc.addChild(topContentPane);
        bc.addChild(centerContentPane);
        bc.addChild(bottomContentPane);
    });
    </script>
</head>
<body class="tundra">
    <div id="bc"></div>
</body>
</html>
```

**В предыдущем примере сначала вызывается метод `startup()` для выполнения первоначального размещения, а затем с помощью метода `addChild()` производится добавление дочерних виджетов. Следующий подход также будет работать:**

```
bc.domNode.appendChild(topContentPane.domNode);
bc.domNode.appendChild(centerContentPane.domNode);
bc.domNode.appendChild(bottomContentPane.domNode);
bc.startup();
```

**Диджиты `BorderContainer` обладают высокой гибкостью и могут вкладываться друг в друга произвольным образом, если ситуация требует этого. Они также предоставляют превосходный способ оформления компоновки в стиле заголовка или в стиле врезки практически без каких-либо усилий, но, с другой стороны, в случаях, когда применение**

виджетов не дает дополнительной выгоды, вы всегда можете вернуться к старым добрым таблицам стилей CSS.

# StackContainer

StackContainer — это диджит компоновки, который отображает целую последовательность элементов одновременно. Концептуально StackContainer напоминает слайд-шоу, когда можно переходить взад и вперед от страницы к странице в «стопке» элементов. Так как StackContainer является контейнерным диджитом компоновки, вы заботитесь лишь о том, чтобы передать ему некоторое число дочерних виджетов, а он позаботится об их отображении. При таком типичном использовании, как показано в примере 14.4, вы получаете возможность просто перелистывать страницы.

*Пример 14.4. Создание диджита StackContainer в разметке*

```
<div id="stack" dojoType="dijit.layout.StackContainer"
style="width:100px; height:100px; margin:5px; border:solid 1px;">

    <div dojoType="dijit.layout.ContentPane">
        One fish...
    </div>

    <div dojoType="dijit.layout.ContentPane">
        Two fish...
    </div>

    <div dojoType="dijit.layout.ContentPane">
        Red fish...
    </div>

    <div dojoType="dijit.layout.ContentPane">
        Blue fish...
    </div>

</div>

<button dojoType="dijit.form.Button">&lt;
    <script type="dojo/method" event="onClick" args="evt">
        dijit.byId("stack").back();
    </script>
</button>

<button dojoType="dijit.form.Button">&gt;
    <script type="dojo/method" event="onClick" args="evt">
        dijit.byId("stack").forward();
    </script>
</button>
```

Диджит StackContainer обладает методами контейнеров и общими методами диджитов компоновки, а кроме того, дополнительными особенностями, список которых приводится в табл. 14.6.

*Таблица 14.6. Прикладной интерфейс диджита StackContainer*

| Имя (по умолчанию) | Тип | Комментарий |
|---|---|---|
| doLayout | Boolean | Используется для изменения размеров отображаемого в данный момент дочернего диджита для приведения их в соответствие с размерами контейнера. Значение по умолчанию: true. |
| selectedChildWidget | Object | Ссылка на текущий выбранный дочерний виджет. Значение по умолчанию: null. |
| selectChild(/*Object*/ page) | Function | Используется, чтобы выбрать определенный дочерний виджет. |
| forward() | Function | Используется для перехода на страницу вперед, к следующему дочернему виджету. |
| back() | Function | Используется для перехода на страницу назад, к предыдущему дочернему виджету. |

Кроме всего прочего StackContainer поддерживает некоторые дополнительные особенности:

- Метод closeChild(/*Object*/ child)
- Точка расширения onClose()
- Дочерние виджеты могут предоставлять атрибуты closeable, title и selected
- При добавлении, удалении или выборе дочерних виджетов публикуются широковещательные сообщения с темами <id>-addChild, <id>-removeChild и <id>-selectChild, соответственно

Поскольку эти особенности обычно ассоциируются с TabContainer (который наследует свойства и методы StackContainer), их официальное представление будет отложено до следующего раздела.

Если вы внимательно читали предыдущий пример создания диджитов компоновки программным способом, то пример 14.5 должен показаться вам знакомым.

*Пример 14.5. Создание диджита StackContainer программным способом*

```
var container = new dijit.layout.StackContainer({}, "foo");

var leftChild = new dijit.layout.ContentPane({});
leftChild.domNode.innerHTML="page 1";

var rightChild = new dijit.layout.ContentPane({});
rightChild.domNode.innerHTML="page 2";

container.addChild(leftChild);
```

```
container.addChild(rightChild);

container.startup();

/* Перепрыгнуть с первой страницы на вторую... */
dijit.byId("foo").forward();
```

## Промедление (или отложенная загрузка) может повысить производительность

В предыдущем примере используются кнопки для явного листания страниц, но не так уж редко StackContainer используется как контейнер приложения, управляющий ходом выполнения приложения, состоящего из нескольких страниц. Например, ваше приложение первоначально может отображать страницу со строкой поиска. Как только для запуска поиска будет нажата кнопка, вы можете выполнить переход к следующей странице, где отображаются результаты поиска. Предположим, что вы оформили все страницы приложения как дочерние виджеты в StackContainer, тогда при таком подходе вам не придется выполнять явную перезагрузку страницы – особенно привлекательная черта интерфейсов в стиле Web 2.0.

Несмотря на то что загрузка всего приложения целиком *может* быть лучшим выбором в некоторых случаях, тем не менее вы могли бы воспользоваться преимуществами отложенной загрузки содержимого, настроив дочерний виджет ContentPane на выполнение отложенной загрузки через атрибут href. Вспомните, что такое поведение виджета ContentPane определяется атрибутом preload, когда при значении false в этом атрибуте (значение по умолчанию) он не выполняет загрузку содержимого, пока не станет видимым. Чтобы убедиться в таком поведении, можно воспользоваться консолью Firebug. Например, если представить, что имеется URL, ссылающийся на ресурс *blueFish*, содержащий текст «Blue fish...» из примера 14.4, то следующее изменение приведет к отложенной загрузке содержимого четвертой страницы виджета StackContainer:

```
<div dojoType="dijit.layout.ContentPane" href="blueFish"></div>
```

Отложенная загрузка идеально подходит в ситуациях, когда приложение обладает некоторыми важными особенностями, которые используются не так часто. Панель настроек – первый кандидат на отложенную загрузку, в ходе которой, кстати, могут загружаться наборы элементов управления, не используемые ни в одной другой странице приложения.

# TabContainer

В действительности TabContainer – это всего лишь более причудливая версия диджита StackContainer, главное отличие которого заключается в том, что TabContainer уже имеет набор вкладок, которые в любой

момент времени могут использоваться для выбора требуемой страницы. Фактически TabContainer наследует StackContainer и добавляет к нему лишь несколько новых особенностей, относящихся к списку вкладок. Пример 14.6 иллюстрирует типичный способ использования диджита TabContainer.

*Пример 14.6. Создание диджита TabContainer в разметке*

```
<div dojoType="dijit.layout.TabContainer"
     style="width:225px; height:100px; margin:5px; border:solid 1px;">

    <div dojoType="dijit.layout.ContentPane" title="one">
        One fish...
    </div>

    <div dojoType="dijit.layout.ContentPane" title="two">
        Two fish...
    </div>

    <div dojoType="dijit.layout.ContentPane" title="red"
    closable="true">Red fish...
        <script type="dojo/method" event="onClose" args="evt">
            console.log("Closing", this.title);
            return true; //чтобы панель закрылась, следует вернуть true!
        </script>
    </div>

    <div dojoType="dijit.layout.ContentPane" title="blue">
        Blue fish...
    </div>
</div>
```

Обратите особое внимание, что позиции вкладок устанавливаются автоматически – вам достаточно просто определить значение атрибута title для каждого дочернего виджета в TabContainer, а обо всем остальном позаботятся внутренние механизмы, не требуя от вас прямого вмешательства (и это лучшее решение). Кроме того, следует отметить, что вы можете сделать вкладки закрывающимися – с помощью атрибута closeable, а дополнительная точка расширения onClose позволит выполнить какие-либо действия при закрытии вкладки. Однако, будьте внимательны: чтобы вкладка закрылась, точка расширения onClose должна вернуть значение true.

В табл. 14.7 перечислены особенности, присущие диджиту TabContainer.

Кнопки, которые вы видите как вкладки, – это обычные кнопки dijit.form.Button. С ними вы можете работать как с обычными кнопками.

Так же, как и в случае с диджитом StackContainer, в TabContainer можно реализовать отложенную загрузку содержимого, установив в виджетах ContentPane атрибут preload в значение false.

*Таблица 14.7. Прикладной интерфейс диджита TabContainer*

| Имя | Тип | Комментарий |
|---|---|---|
| title | String | Свойство, подмешиваемое классом StackContainer к свойствам, унаследованным от класса _Widget. Используется дочерними виджетами для предоставления заголовка вкладки. |
| closeable | Boolean | Свойство, подмешиваемое классом StackContainer к свойствам, унаследованным от класса _Widget. Используется дочерними виджетами, чтобы указать, что вкладка может закрываться. Если вкладка допускает закрытие, на ней появляется маленький ярлык, щелкнув на котором, можно закрыть вкладку. Значение по умолчанию: false. |
| onClose() | Function | Точка расширения, подмешиваемая классом StackContainer к свойствам, унаследованным от класса _Widget, которая предоставляет возможность дочерним виджетам определить свое поведение при закрытии вкладки. По умолчанию возвращает значение true. |
| tabPosition | String | Указывает, где в списке вкладок должна находиться данная вкладка. Допустимыми значениями являются "top" (по умолчанию), "button", "left-h" и "right-h". |
| <id>-addChild<br><id>-removeChild<br><id>-selectChild | темы для dojo.publish | Эти возможности унаследованы от класса StackContainer. Эти темы публикуются, когда выполняется добавление, удаление или выбор дочернего виджета. Часть <id> обозначает значение атрибута id виджета TabContainer. |

А теперь взгляните на пример 14.7, демонстрирующий программный способ создания диджита.

*Пример 14.7. Создание диджита TabContainer программным способом*

```
var container = new dijit.layout.TabContainer({
    tabPosition: "left-h",
    style : "width:200px;height:200px;"
}, "foo");

var leftChild = new dijit.layout.ContentPane({title : "tab1"});
leftChild.domNode.innerHTML="tab 1";

var rightChild = new dijit.layout.ContentPane({title : "tab2",
                                               closable: true});
rightChild.domNode.innerHTML="tab 2";

container.addChild(leftChild);
container.addChild(rightChild);

container.startup();
```

# AccordionContainer

Подобно TabContainer, диджит AccordionContainer наследует StackContainer и является средством, позволяющим отображать дочерние виджеты по одному в каждый конкретный момент времени. Визуальное отличие этого диджита состоит в том, что он изображается в виде гармошки и воспроизводит анимационный эффект при выборе любого дочернего виджета.

Еще одно важная особенность AccordionContainer заключается в необходимости использовать специальный дочерний контейнер AccordionPane, который служит оберткой для всех его дочерних виджетов. Обсуждение фактических причин, объясняющих такое положение дел, не представляет особого интереса; достаточно знать, что это связано с внутренней реализацией AccordionContainer. Вообще достаточно воспринимать AccordionPane как ContentPane, и все будет в порядке.

Начиная с версии 1.1 инструментария диджит AccordionPane не поддерживает вложенные виджеты компоновки, такие как SplitContainer, однако он прекрасно работает со всеми другими видами содержимого.

В примере 14.8 демонстрируется простой виджет AccordionContainer в действии.

*Пример 14.8. Создание диджита AccordionContainer в разметке*

```
<div id="foo" dojoType="dijit.layout.AccordionContainer"
     style="width:150px; height:150px; margin:5px">

    <div dojoType="dijit.layout.AccordionPane" title="one">
        <p>One fish...</p>
    </div>
    <div dojoType="dijit.layout.AccordionPane" title="two">
        <p>Two fish...</p>
    </div>
    <div dojoType="dijit.layout.AccordionPane" title="red">
        <p>Red fish...</p>
    </div>
    <div id="blue" dojoType="dijit.layout.AccordionPane" title="blue">
        <div dojoType="dijit.layout.ContentPane" href="blueFish"></div>
    </div>
</div>
```

Что касается прикладного интерфейса: сам диджит AccordionContainer предоставляет единственный дополнительный атрибут помимо тех, что предлагаются диджитом StackContainer, приведенный в табл. 14.8.

Мы могли бы оставить реализацию примера создания диджита программным способом в качестве самостоятельного упражнения для тех, кому это интересно, однако в данном случае порядок создания, как показано в примере 14.9, несколько отличается от описанного ранее, потому что AccordionPane – это отдельный диджит.

*Таблица 14.8. Прикладной интерфейс диджита AccorfionContainer*

| Имя (по умолчанию) | Тип | Комментарий |
|---|---|---|
| duration (250) | Integer | Атрибут диджита AccordionContainer, который указывает время в миллисекундах, в течение которого будет воспроизводиться эффект плавного открытия выбранной панели. |

*Пример 14.9. Создание диджита AccordionContainer программным способом*

```
var container = new dijit.layout.AccordionContainer({}, "foo");

var content1 = dojo.doc.createElement("p");
content1.innerHTML = "content 1";

var ap1 = new dijit.layout.AccordionPane({title: "pane1", selected : true},
                                         content1);
container.addChild(ap1);

var content2 = dojo.doc.createElement("p");
content2.innerHTML = "content 2";

var ap2 = new dijit.layout.AccordionPane({title: "pane2"}, content2);
container.addChild(ap2);

container.startup();
```

# Проблема видимости и отображения

Работая с примерами этой главы, вы, возможно, обратили внимание, что размещение элементов производится после загрузки страницы. Например, вы могли заметить, как сначала появляется обычный текст, а затем, как по волшебству, все вдруг занимает свои места в соответствии с выбранной схемой размещения. Нельзя сказать, что это вообще недопустимо, но в большинстве ситуаций вам вряд ли захочется видеть такую перемену в отображении.

Обычно эта проблема решается за счет того, что изначально тело страницы делается невидимым, а затем, когда загрузка будет закончена, оно делается видимым. Реализуется этот прием очень просто: вам нужно определить стиль (или класс), указывающий, что тело страницы должно быть скрыто, например так: `<body style="visibility:hidden;">`. Только не забудьте добавить соответствующий вызов, который сделает страницу видимой в нужное время. Предположим, что невидимым было объявлено все тело страницы, тогда вызов `dojo.style(dojo.body(), "visibility", "visible")`, добавленный в `dojo.addOnLoad`, сделает страницу видимой. Вместо `addOnLoad` можно воспользоваться любой другой функцией обратного вызова, если по каким-то причинам необходимо задержать отображение страницы до наступления некоторого события (например, в обработчике асинхронного события, который выполняет передачу данных некоторому виджету).

Не забывайте, что между стилями CSS, управляющими видимостью, существуют определенные различия, заключающиеся в том, как распределяется пространство страницы. Вообще, узлы со стилем `visibility:hidden` невидимы, но продолжают занимать место, а узлы со стилем `display:none` невидимы и не занимают места – в результате на экране происходит сдвиг содержимого, когда свойство `display:none` изменяется на `display:block`.

Следует заметить, однако, что контейнерные диджиты компоновки не всегда правильно реагируют, когда создаются в изначально невидимом состоянии. Если вы обнаружите, что диджиты компоновки остаются скрытыми, когда они должны быть уже видимы, вам может потребоваться вручную вызвать метод `resize()`, чтобы принудительно заставить их отобразить свое содержимое. Исторически эта проблема часто наблюдалась при отображении диджитов компоновки внутри `dijit.Dialog`.

Диджиты компоновки не всегда отображаются корректно, если они были созданы в контексте, который не требует их видимости. Практически всегда эту проблему можно исправить вызовом метода `resize()` контейнера.

# В заключение

По прочтении этой главы вы должны уметь:

- Оценивать, какие типичные задачи дизайна (например, организация размещения содержимого во вкладках) лучше решать с применением диджитов компоновки
- Понимать, какими основными особенностями обладают различные диджиты компоновки
- Создавать произвольные схемы размещения с помощью диджитов компоновки как в разметке, так и программным способом
- Использовать `BorderContainer` для создания мозаичных схем размещения, которые могут произвольно изменять размеры
- Использовать `ContentPane` для отложенной загрузки содержимого либо в качестве самостоятельного диджита, либо в составе других диджитов
- Использовать `StackContainer` и `TabContainer` для отображения данных в приложении на нескольких страницах
- Понимать некоторые проблемы, связанные с отображением, если изначально диджиты компоновки находились в невидимом состоянии
- Понимать ограничения `AccordionPane`, связанные с вложением других диджитов компоновки
- Понимать, какую роль играют базовые классы `_Container` и `_Contained` в диджитах компоновки

В следующей главе мы рассмотрим прикладные диджиты.

# 15

# Виджеты приложения

В этой главе последовательно рассматриваются все прикладные виджеты общего назначения, присутствующие в библиотеке Dijit. Так или иначе, но они являются самыми удивительными диджитами, предоставляемыми инструментарием, потому что они не так широко известны, как элементы форм, и, в отличие от диджитов компоновки, обеспечивают значительные функциональные возможности по обеспечению интерактивности. `ProgressBar`, `Toolbar`, `Editor` и `Tree` — вот лишь некоторые из этих замечательных диджитов, с которыми вам предстоит познакомиться. Вероятно, вы оцените высший пилотаж применения приемов DHTML, использованных в этой главе, особенно ближе к ее концу.

Все виджеты, описываемые в этой главе, точно так же, как и другие виджеты из библиотеки Dijit, обеспечивают максимальный уровень доступности, хотя явно это и не указывается.

## Tooltip

Всплывающие подсказки представляют собой замечательное средство обеспечения пользователя справочной информацией о назначении того или иного элемента управления, находящегося на странице. Обычный атрибут `title` языка разметки HTML неплохо подходил для приложений 1990-х годов, но наступившая эра веб-приложений требует наличия средств воспроизведения более богатого разнообразия подсказок. Диджит `Tooltip` как раз является таким средством, обеспечивая возможность отображения произвольной разметки HTML вместо простого фрагмента текста. В предыдущей главе вы уже встречались с диджитом `Tooltip`, когда рассматривали диджит `ValidationTextBox` и его наследников, теперь вы узнаете, что диджит `Tooltip` может использоваться самостоятельно.

Взгляните на пример 15.1, где используются некоторые ключевые особенности Tooltip для воспроизведения результата, представленного на рис. 15.1.

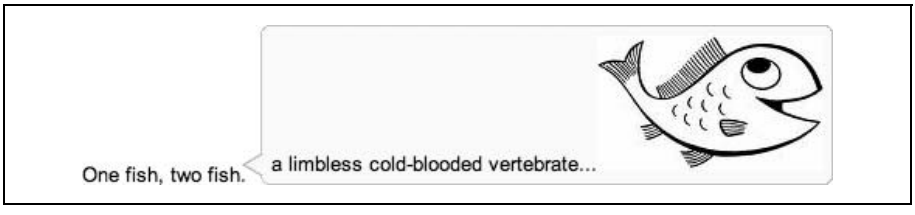



***Рис. 15.1.*** *Подсказка, появляющаяся при наведении указателя мыши на любой из тегов со словом fish (рыба)*

*Пример 15.1. Пример типичного использования диджита Tooltip*

```
One <span id="one">fish</span>, two <span id="two">fish</span>.

<div dojoType="dijit.Tooltip" connectId="one,two">
    A limbless cold-blooded vertebrate...<img src='./fish.jpeg'/>
</div>
```

Обратите внимание на синтаксис передачи нескольких значений в атрибут connectId — он не совпадает с синтаксисом массивов в языке JavaScript: несколько значений указываются без квадратных скобок и без внутренних кавычек: connectId="id1,id2". Вероятно, в одном из будущих выпусков этот синтаксис будет приведен к стандартному, чтобы ликвидировать это исключение из правил.

Как было показано в примере, вы просто определяете произвольную разметку HTML, которая будет отображаться диджитом Tooltip, и больше ничего. Диджит Tooltip должен использоваться для отображения содержимого, доступного только для чтения, а для интерактивного содержимого, такого как поля ввода и кнопки, следует использовать диджит TooltipDialog, который будет представлен в следующем разделе. В табл. 15.1 приводится полный перечень особенностей диджита Tooltip.

*Таблица 15.1. Прикладной программный интерфейс диджита Tooltip*

| Имя (по умолчанию) | Тип | Комментарий |
|---|---|---|
| connectId ("") | String | Список значений, разделенных запятыми, в котором перечисляются значения атрибутов id элементов, для которых будет отображаться всплывающая подсказка. |

| Имя (по умолчанию) | Тип | Комментарий |
|---|---|---|
| label ("") | String | Текст, отображаемый диджитом Tooltip. Несмотря на то что атрибут label может содержать произвольную разметку HTML, ее лучше включать между открывающим и закрывающим тегами. |
| showDelay (400) | Integer | Количество миллисекунд задержки, после которой подсказка будет показана пользователю. |

# Виджеты диалогов

Библиотека Dijit предлагает два виджета, обладающих функциональностью диалогов: Dialog, взаимодействие с которым напоминает взаимодействие с обычными диалогами (только имеет более привлекательный внешний вид и обладает более высокой гибкостью), и TooltipDialog, который напоминает обычные всплывающие подсказки, за исключением того, что способен отображать другие виджеты и обладает большей интерактивностью, чем обычный Tooltip.

## Dialog

Диджит Dialog концептуально напоминает всплывающее окно, сквозь которое просвечивает то, что находится под ним. Несмотря на то что содержимое страницы, лежащее ниже, остается видимым, тем не менее вы не можете взаимодействовать с ним, что делает этот диджит идеальным для ситуаций, когда необходимо временно запретить доступ к элементам управления на странице или вынудить пользователя отреагировать на предупреждение.

Но в дополнение к очевидному использованию диджит Dialog может также использоваться в любой другой ситуации, когда требуется появление другого окна. С точки зрения реализации использование диджита Dialog часто выглядит более простым, чем организация взаимодействия с другим окном, потому что все, что находится внутри диджита Dialog, является частью дерева DOM текущей страницы.[1] Вы можете обращаться к нему, манипулировать элементами, имеющимися в странице, даже теми, что в настоящий момент невидимы.

Диджит Dialog может содержать любое дерево DOM элементов, которые вам могут потребоваться, будь то простой фрагмент HTML, диджит со сложной схемой размещения или ваш собственный диджит. В примере 15.2 показан наиболее типичный способ использования диджита

[1] В действительности некоторые броузеры просто не позволят манипулировать деревом DOM в одном окне из другого окна, даже несмотря на то, что оба окна имеют общее происхождение.

Dialog – в данном случае он появляется автоматически по завершении загрузки страницы.

Как отмечалось в предыдущей главе, вам может потребоваться вручную вызывать метод resize диджитов компоновки, чтобы вынудить их перерисовать свое содержимое, если при создании они находились в невидимом состоянии – такая ситуация складывается, если вы сначала создаете их, а потом встраиваете в Dialog.

*Пример 15.2. Пример типичного использования диджита Dialog*

```
<html>
    <head>
        <title>Fun With Dialog!</title>

        <link rel="stylesheet" type="text/css"
         href="http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css" />
        <link rel="stylesheet" type="text/css"
        href="http://o.aolcdn.com/dojo/1.1/dijit/themes/tundra/tundra.css" />

        <script
            type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js"
            djConfig="parseOnLoad:true">
        </script>

        <script type="text/javascript">
            dojo.require("dojo.parser");
            dojo.require("dijit.Dialog");
            dojo.addOnLoad(function() {
                dijit.byId("dialog").show();
            });
        </script>
    </head>
    <body class="tundra">
        <div id="dialog" dojoType="dijit.Dialog">
            So foul and fair a day I have not seen...
        </div>
    </body>
</html>
```

При создании диджита Dialog программным способом его внутреннее наполнение легко производится с помощью метода setContent, который может принять произвольный узел DOM. Ниже приводится пример, в котором пользователь вынуждается к щелчку на кнопке, помещенной внутрь диджита Dialog, хотя приложение явно просит не делать этого:

```
dojo.addOnLoad(function() {
    var d = new dijit.Dialog();

    //спрятать от пользователя обычную кнопку закрытия диалога...
    dojo.style(d.closeButtonNode, "visibility", "hidden");
```

```
        var b = new dijit.form.Button({label: "Do not press this button"});
        var handle = dojo.connect(b, "onClick", function() {
            d.hide();
            dojo.disconnect(handle);
        });
        d.setContent(b.domNode);
        d.show();
    });
```

Шаблон диджита Dialog содержит ряд полезных точек подключения дополнительных элементов, включая точку closeButtonNode, которая была использована в предыдущем примере для сокрытия ярлыка, обычно используемого для закрытия диалога.

Как и в случае с другими диджитами, при работе с диджитом Dialog вы часто будете использовать лишь некоторые из общих методов и атрибутов; в табл. 15.2 приводятся остальные особенности, которые могут вам потребоваться.

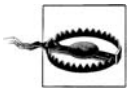
Диджит Dialog наследует ContentPane, поэтому в нем доступны все атрибуты, методы и точки расширения класса ContentPane. Порядок использования ContentPane демонстрируется в примере 14.3.

*Таблица 15.2. Прикладной программный интерфейс диджита Dialog, построенный на основе ContentPane*

| Имя | Тип | Комментарий |
|---|---|---|
| open | Boolean | Состояние диалога. Если диалог открыт, имеет значение true, в противном случае – false (значение по умолчанию). |
| duration | Integer | Продолжительность в миллисекундах эффекта проявления и растворения диалога. Значение по умолчанию: 400. |
| hide() | Function | Скрывает диалог. |
| layout() | Function | Выполняет позиционирование диалога и всего, что находится внутри него. |
| show() | Function | Отображает диалог. |

## TooltipDialog

Диджит TooltipDialog наследует Dialog, но предоставляемая им функциональность больше напоминает меню DropDownButton, за исключением того, что с этим диджитом вы можете взаимодействовать. Фактически в своем текущем проявлении TooltipDoalog мог бы размещаться в DropDownButton или ComboButton, хотя теоретически вы могли бы определить такой стиль кнопки, чтобы полностью изменить ее внешний вид. Можно считать, что концепция TooltipDialog предполагает порядок взаимодействия, который можно встретить в приложениях электронных таблиц.



Чтобы иметь возможность использовать диджит TooltipDialog, необходимо выполнить инструкцию dojo.require("dijit.Dialog"), потому что объявление TooltipDialog находится в файле ресурса Dialog.

Функциональные возможности TooltipDialog очень напоминают возможности диджита Dialog за исключением поддержки метода show(), что обуславливает невозможность использовать TooltipDialog в качестве самостоятельного диджита. Кроме того, он предоставляет стандартный атрибут title, который можно заполнить, если иметь в виду обеспечение высокого уровня доступности.

Хорошим примером использования TooltipDialog может служить интерактивное средство пометки изображения. Например, можно было бы использовать диджит DropDownButton для отображения картинок с помощью атрибута iconClass и реализовать TooltipDialog, который будет выводиться после щелчка мышью на изображении. Следующий фрагмент в общих чертах демонстрирует, с чего можно было бы начать разработку инструмента пометки изображений, а результат работы этого фрагмента приводится на рис. 15.2.

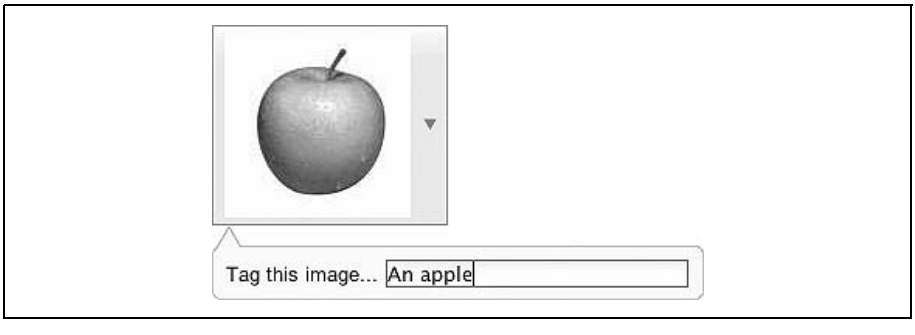



***Рис. 15.2.*** *Средство пометки изображений, построенное на основе DropDownButton и TooltipDialog*

```
<!-- где-то там...
<style type="text/css">
.customImage {
                background-image : url('/static/path/to/apple.jpeg');
                backgrond-repeat : no-repeat;
                width : 120px;
                height : 120px;
            }
</style>
-->

<button dojoType="dijit.form.DropDownButton" iconClass="customImage"
        showLabel="false">
```

```
    <span>This label is hidden...</span>

    <div dojoType="dijit.TooltipDialog">
        <span>Tag this image...<span>
        <div dojoType="dijit.form.TextBox"></div>
    </div>
</button>
```

# ProgressBar

Диджит ProgressBar, индикатор хода выполнения какой-то операции, напоминает другие аналогичные индикаторы, которые можно встретить в приложениях, и предусматривает возможность работы в детерминированном и в недетерминированном режиме. Самое замечательное в этом диджите то, что о нем можно много не говорить. Фактически пример 15.3 прекрасно говорит сам за себя.

*Пример 15.3. Типичный пример использования недетерминированного индикатора ProgressBar*

```
<div dojoType="dijit.ProgressBar" indeterminate="true"
     style="width:300px"></div>
```

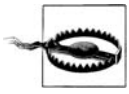
Конечно, иногда вместо использования недетерминированного индикатора будет возникать необходимость получать данные от сервера и отображать фактический ход выполнения операции. Предположим, что у нас имеется серверная процедура, которая возвращает некоторое значение, свидетельствующее о ходе выполнения операции. Следующий фрагмент представляет собой модель такой процедуры:

```
import cherrypy

config = {
    #передать этот статический файл...
    '/foo.html' :
    {
        'tools.staticfile.on' : True,
        'tools.staticfile.filename' : '/absolute/path/to/foo.html'
    }

}
class Content:
    def __init__(self):
        self.progress = 0

    @cherrypy.expose
    def getProgress(self):
        self.progress += 10
        return str(self.progress)

cherrypy.quickstart(Content(), '/', config=config)
```

Файл *foo.html*, содержащий диджит ProgressBar, может выглядеть, как показано ниже:

```
<html>
    <head>
        <title>Fun with ProgressBar!</title>

        <link rel="stylesheet" type="text/css"
         href="http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css" />
        <link rel="stylesheet" type="text/css"
        href="http://o.aolcdn.com/dojo/1.1/dijit/themes/tundra/tundra.css" />

        <script
            type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js"
            djConfig="parseOnLoad:true">
        </script>

        <script type="text/javascript">
            dojo.require("dojo.parser");
            dojo.require("dijit.ProgressBar");
            dojo.addOnLoad(function() {
                var progressInterval = setInterval(function() {
                    dojo.xhrGet({
                        url : "http://localhost:8080/getProgress",
                        load : function(response, ioArgs) {
                            console.log("load", response);
                            if (response <= 100) {
                                dijit.byId("pb").update(
                                    {progress : response});
                            }
                            else {
                                clearInterval(progressInterval);
                            }
                        }
                    });
                }, 1000);
            });
        </script>
    </head>
    <body style="padding:100px" class="tundra">
        <div>Loading...</div>
        <div id="pb" dojoType="dijit.ProgressBar" style="width:300px"></div>
    </body>
</html>
```

Функция addOnLoad в этом примере каждую секунду обращается к URL /getProgress, чтобы получить некоторое значение и с помощью функции update диджита ProgressBar использовать его для индикации хода выполнения операции. Применение функции setInterval вполне обычно при работе с ProgressBar.



Не путайте функцию `setInterval` с функцией `setTimeout`. Первая из них продолжает многократно вызывать указанную ей функцию через установленный интервал времени, тогда как вторая вызывает ее всего один раз – *по истечении* установленного интервала времени.

Полный перечень особенностей диджита `ProgressBar` приводится в табл. 15.3.

*Таблица 15.3. Прикладной программный интерфейс диджита ProgressBar*

| Имя | Тип | Комментарий |
|---|---|---|
| `indeterminate` | Boolean | Определяет режим работы: недетерминированный (когда просто отображается анимированное изображение) или детерминированный, когда с помощью метода `update` отображается фактический ход выполнения операции. |
| `maximum` | Float | Максимально возможное значение. Наиболее часто используется диапазон значений от `0` до `100` (значение по умолчанию), но существует возможность определить иной диапазон. |
| `places` | Number | Число знаков после запятой, которые будут отображаться в детерминированном режиме. Значение по умолчанию: `0`. |
| `progress` | String | Начальное значение для отображения в `ProgressBar`. Допускается указывать знак процента, например `"50%"`, для обозначения относительных величин. |
| `update(/*Object*/ progress)` | Function | Используется для обновления информации о ходе выполнения операции. В объекте `progress` допускается передавать значения свойств `progress`, `maximum` и `determinate` для настройки `ProgressBar` при каждом вызове метода `update`. |
| `onChange()` | Function | Точка расширения. Вызывается после каждого обращения к методу `update`. |

Наконец напомню, что если потребуется отображать диджит `ProgressBar` при одновременном блокировании доступа к странице, чтобы заставить пользователя дождаться завершения операции, вы всегда можете поместить его внутрь диджита `Dialog`. Сделать это можно примерно так, как показано ниже:

```
var pb = new dijit.ProgressBar;
var d = new dijit.Dialog;
d.setContent(pb.domNode);
d.show();
```

# ColorPalette

ColorPalette – это еще один простой самостоятельный виджет, используемый для организации более наглядного и интерактивного способа выбора цвета, что может быть очень удобно в ситуациях, когда вы разрешаете пользователю настраивать тему приложения. По умолчанию палитра, заполненная предопределенными цветами, наиболее популярными в веб-дизайне, может иметь два фиксированных размера: 3H4 и 7H10.

Возможно, вы уже задались вопросом, почему нельзя определить свой набор цветов для палитры. Оказывается, палитра, которая появляется на экране, – это изображение, а не набор панелей в виде разметки HTML, и сделано это с целью обеспечения доступности, хотя такое решение и не кажется идеальным. Но если вам потребуется расширить ColorPalette, чтобы отобразить свой набор цветов, это вполне выполнимо – вам нужно только заглянуть в исходные тексты и внести изменения в некоторые частные атрибуты.

Использовать диджит ColorPalette в разметке очень просто, это показано в следующем листинге:

```
<div dojoType="dijit.ColorPalette">
    <script type="dojo/method" event="onChange" args="selectedColor">
        /* возможно, здесь же следует предусмотреть действия
           по сокрытию палитры? */
        console.log(selectedColor);
    </script>
</div>
```

Как и ProgressBar, ColorPalette является замечательным и простым отдельным диджитом. Перечень его особенностей приводится в табл. 15.4.

*Таблица 15.4. Прикладной программный интерфейс диджита ColorPalette*

| Имя | Тип | Комментарий |
|---|---|---|
| defaultTimeout | Integer | Продолжительность задержки, прежде чем включится режим автоповтора нажатой клавиши. Значение по умолчанию: 500. |
| timeoutChangeRate | Number | Множитель, используемый для изменения значения таймера автоповтора. Значение 1.0 означает, что события автоповтора буду запускаться через один и тот же интервал. Если значение меньше 1.0, то каждое последующее событие будет происходить немножко раньше пропорционально этому значению. Значение по умолчанию: 0.90. |

| Имя | Тип | Комментарий |
|---|---|---|
| palette | String | Размер отображаемой палитры: 7H10 (по умолчанию) или 3H4. |
| onChange(/*String*/ hexColor) | Function | Точка расширения, которая вызывается в момент выбора цвета |

Создание диджита программным способом тоже выполняется достаточно просто:

```
var cp = new dijit.ColorPalette({/*здесь определяются атрибуты */});
/* Теперь следует добавить палитру в страницу...*/
dojo.body().appendChild(cp.domNode);
```

# Toolbar

Панель инструментов – это еще один хорошо знакомый элемент управления, который упрощает доступ пользователя к наиболее часто используемым командам. Проще говоря, диджит Toolbar (панель инструментов) представляет собой всего лишь место для размещения коллекции диджитов Button, которые при соответствующей стилизации могут иметь весьма привлекательный внешний вид. В состав библиотеки Dijit входят различные предопределенные темы, содержащие классы для многих распространенных операций, таких как вырезать/вставить, жирный/курсив и прочих, которые можно указывать в качестве значения атрибута iconClass диджитов Button.

Следующий листинг демонстрирует, как поместить диджит Toolbar в страницу, и затем последовательно присоединяет к каждой кнопке свой обработчик события.

В этом конкретном примере предпринимается попытка автоматизировать процесс привязывания обработчиков к кнопкам. Обратите внимание, что внутри блока forEach подключение производится к x.parentNode, а не к x, – это связано с особенностями реализации кнопки. Оказывается, фактический щелчок мышью принимает на себя покрывающий кнопку ярлык, в чем легко можно убедиться благодаря возможностям Firebug.

```
<html>
    <head>
        <title>Fun with Toolbar!</title>

        <link rel="stylesheet" type="text/css"
         href="http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css" />
        <link rel="stylesheet" type="text/css"
        href="http://o.aolcdn.com/dojo/1.1/dijit/themes/tundra/tundra.css" />

        <script
            type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js"
```

```
            djConfig="parseOnLoad:true,isDebug:true">
        </script>

        <script type="text/javascript">
            dojo.require("dojo.parser");
            dojo.require("dijit.Toolbar");
            dojo.require("dijit.form.Button");

            dojo.addOnLoad(function() {
                var bold = function() {console.log("bold");}
                var italic= function() {console.log("italic");}
                var underline = function() {console.log("underline");}
                var superscript = function() {console.log("superscript");}
                var subscript = function() {console.log("subscript");}

                dojo.query(".dijitEditorIcon").forEach(function(x) {
                    if (dojo.hasClass(x, "dijitEditorIconBold"))
                        dojo.connect(x.parentNode, "onclick", bold);
                    else if (dojo.hasClass(x, "dijitEditorIconItalic"))
                        dojo.connect(x.parentNode, "onclick", italic);
                    else if (dojo.hasClass(x, "dijitEditorIconUnderline"))
                        dojo.connect(x.parentNode, "onclick", underline);
                    else if (dojo.hasClass(x, "dijitEditorIconSubscript"))
                        dojo.connect(x.parentNode, "onclick", superscript);
                    else if (dojo.hasClass(x, "dijitEditorIconSuperscript"))
                        dojo.connect(x.parentNode, "onclick", subscript);
                });
            });
        </script>
    </head>
    <body style="padding:100px" class="tundra">
        <div dojoType="dijit.Toolbar" style="width:175px">
            <button dojoType="dijit.form.Button"
              iconClass="dijitEditorIcon dijitEditorIconBold" ></button>
            <button dojoType="dijit.form.Button"
              iconClass="dijitEditorIcon dijitEditorIconItalic" ></button>
            <button dojoType="dijit.form.Button"
             iconClass="dijitEditorIcon dijitEditorIconUnderline" ></button>

            <span dojoType="dijit.ToolbarSeparator"></span>

            <button dojoType="dijit.form.Button"
              iconClass="dijitEditorIcon dijitEditorIconSubscript"></button>
            <button dojoType="dijit.form.Button"
             iconClass="dijitEditorIcon dijitEditorIconSuperscript"></button>
        </div>
    </body>
</html>
```

Интересно отметить, что в настоящее время библиотека Dijit определяет ряд наглядных ярлыков для редактора `Editor` (определения находятся в таблицах стилей темы), которые могут содержаться в диджите `Toolbar`. (Диджит `Editor` будет подробно рассматриваться в одном из следующих разделов.)

Диджит Toolbar имеет простой прикладной программный интерфейс, как показано в табл. 15.5, который можно считать типичным для любого потомка класса _Container.

*Таблица 15.5. Прикладной программный интерфейс диджита Toolbar*

| Имя | Тип | Комментарий |
|---|---|---|
| addChild(/*Object*/ child, /*Integer?*/ insertIndex) | Function | Вставляет диджит child в Toolbar. |
| getChildren() | Function | Возвращает массив дочерних диджитов, содержащихся в диджите Toolbar. |
| removeChild(/*Object*/ child) | Function | Удаляет дочерний диджит (очищает его свойство domNode, но не разрушает сам диджит, – для этого вы должны вручную вызвать метод destroyRecursive()). |

# Menu

Диджит Menu моделирует контекстное меню, подобное тому, что появляется, если щелкнуть правой кнопкой мыши на ярлыке приложения или на поверхности рабочего стола. Диджит Menu содержит в себе виджеты MenuItem, являющиеся обычными пунктами меню, которые можно выбирать, или виджеты PopupMenuItem, образующие еще один слой элементов меню (напоминая меню кнопки Пуск (Start) в операционной системе Windows). Дочерним для PopupMenuItem является другой диджит Menu. Теоретически с помощью диджитов PopupMenuItem и MenuItem можно было бы реализовать иерархические меню любой степени вложенности, хотя такое решение нельзя назвать мудрым с точки зрения простоты и удобства.

Для начала рассмотрим простое меню, состоящее только из диджитов MenuItem, как показано в примере 15.4.

Выполнив инструкцию dojo.require("dijit.Menu"), вы также получаете в свое распоряжение диджиты MenuItem и PopupMenuItem.

*Пример 15.4. Пример типичного использования диджита Menu*

```
<body class="tundra">
    <!-- щелчок правой кнопкой мыши здесь
         приводит к появлению контекстного меню -->
    <div id="context" style="background:#eee; height:300px;
         width:300px;"></div>

    <div dojoType="dijit.Menu" targetNodeIds="context" style="display:none">
        <div dojoType="dijit.MenuItem">foo
            <script type="dojo/method" event="onClick" args="evt">
                console.log("foo");
```

```
            </script>
        </div>
        <div dojoType="dijit.MenuItem">bar
            <script type="dojo/method" event="onClick" args="evt">
                console.log("bar");
            </script>
        </div>
        <div dojoType="dijit.MenuItem">baz
            <script type="dojo/method" event="onClick" args="evt">
                console.log("baz");
            </script>
        </div>
    </div>
</body>
```

При определении диджита `Menu` ему, так же как и диджиту `Tooltip`, передается строка со списком значений, разделенных запятыми, без объемлющих квадратных скобок, которые необходимы при определении обычных массивов языка JavaScript. В будущих выпусках инструментария это отклонение может быть ликвидировано.

## Нарезка изображений для уменьшения задержки

Вместо того чтобы определять бесчисленное множество маленьких изображений, для получения каждого из которых придется выполнять синхронный запрос HTTP к серверу, темы в библиотеке Dijit используют иной подход – все изображения объединены в один блок, вплотную друг к другу. Для вырезания нужной части изображения, которая должна отображаться в виде ярлыка, используются стили CSS, как показано ниже:

```
.tundra .dijitEditorIcon
/* Все ярлыки, встроенные в Editor, имеют этот класс */
{
    background-image: url('images/editor.gif');
    background-repeat: no-repeat;
    width: 18px;
    height: 18px;
    text-align: center;
}

/* Отдельные  ярлыки вырезаются следующим образом... */
.tundra .dijitEditorIconUnderline { background-position: -648px; }
```

Ниже приводится перечень всех ярлыков, доступных в диджите Editor:

- `dijitEditorIconSep`
- `dijitEditorIconBackColor`

- dijitEditorIconBold
- dijitEditorIconCancel
- dijitEditorIconCopy
- dijitEditorIconCreateLink
- dijitEditorIconCut
- dijitEditorIconDelete
- dijitEditorIconForeColor
- dijitEditorIconHiliteColor
- dijitEditorIconIndent
- dijitEditorIconInsertHorizontalRule
- dijitEditorIconInsertImage
- dijitEditorIconInsertOrderedList
- dijitEditorIconInsertTable
- dijitEditorIconInsertUnorderedList
- dijitEditorIconItalic
- dijitEditorIconJustifyCenter
- dijitEditorIconJustifyFull
- dijitEditorIconJustifyLeft
- dijitEditorIconJustifyRight
- dijitEditorIconLeftToRight
- dijitEditorIconListBulletIndent
- dijitEditorIconListBulletOutdent
- dijitEditorIconListNumIndent
- djitEditorIconListNumOutdent
- dijitEditorIconOutdent
- dijitEditorIconPaste
- dijitEditorIconRedo
- dijitEditorIconRemoveFormat
- dijitEditorIconRightToLeft
- dijitEditorIconSave
- dijitEditorIconSpace
- dijitEditorIconStrikethrough
- dijitEditorIconSubscript
- dijitEditorIconSuperscript
- dijitEditorIconUnderline
- dijitEditorIconUndo
- dijitEditorIconWikiword
- dijitEditorIconToggleDir

Как видите, в разметке приходится делать совсем немного, что только упрощает возможность использования меню. Единственное, на что следует обратить внимание, это строка, где значение атрибута стиля display устанавливается равным none, – это очень важно, потому что без этого ваше меню будет видимо изначально.

Теперь предположим, что нам необходимо, чтобы пункт baz был реализован диджитом PopupMenuItem, а само меню было бы контекстным для всего окна. Сделать это можно, как показано ниже:

```
<div dojoType="dijit.Menu" style="display:none"
     contextualMenuForWindow="true">

    <div dojoType="dijit.MenuItem">foo
        <script type="dojo/method" event="onClick" args="evt">
            console.log("foo");
        </script>
     </div>
    <div dojoType="dijit.MenuItem">bar
        <script type="dojo/method" event="onClick" args="evt">
            console.log("bar");
        </script>
    </div>
    <div dojoType="dijit.PopupMenuItem">
        <span>baz</span>
        <div dojoType="dijit.Menu">
            <!-- определить обработчики события onClick
                 для каждого элемента -->
            <div dojoType="dijit.MenuItem">yabba</div>
            <div dojoType="dijit.MenuItem">dabba</div>
            <div dojoType="dijit.MenuItem">doo</div>
        </div>
    </div>
</div>
```

Хотелось бы надеяться, что самой хитрой для вас частью в установке PopupMenuItem оказалась необходимость явного определения первого дочернего узла, играющего роль заголовка.

В завершение раздела о Menu в табл. 15.6 приводится перечень особенностей, составляющих программный интерфейс диджита. Обратите внимание, что, будучи наследником класса _Container, диджит Menu имеет ключевые методы добавления, удаления и получения дочерних элементов, так же как Toolbar и другие подобные диджиты. Состав прикладного программного интерфейса диджитов MenuItem и PopupMenuItem приводится в табл. 15.7.

*Таблица 15.6. Прикладной программный интерфейс диджита Menu*

| Имя | Тип (по умолчанию) | Комментарий |
|---|---|---|
| contextMenuForWindow | Boolean (false) | Если имеет значение true, меню будет открываться щелчком правой кнопки мыши на любой части окна. Если имеет значение false, в атрибуте targetNodeIds должен быть указан один или более узлов, щелчком на которых может быть открыто меню. |
| popupDelay | Integer (500) | Число миллисекунд, по прошествии которых с момента щелчка будет открыто контекстное меню. (Повторный щелчок, произведенный до истечения этого срока, производит сброс таймера в начальное состояние.) |
| targetNodeIds | Array ([ ]) | Список значений атрибутов id узлов, поддерживающих это меню. |
| parentMenu | Object (null) | Ссылка на родительский диджит Menu, если таковой имеется. |
| addChild(/*Object*/ child, /*Integer?*/ insertIndex) | Function | Вставляет диджит child в Menu. |
| getChildren() | Function | Возвращает массив дочерних диджитов, содержащихся в диджите Menu. |
| removeChild(/*Object*/ child) | Function | Удаляет дочерний диджит (очищает его свойство domNode, но не разрушает сам диджит, – для этого вы должны вручную вызвать метод destroyRecursive()). |
| bindDomNode(/*String\|DOMNode*/node) | Function | Подключает меню к определенному узлу. (Удобно, например, для контекстных меню.) |
| unBindDomNode(/*String\|DOMNode*/node) | Function | Отсоединяет меню от определенного узла. |
| onClick(/*Object*/item, /*Event*/evt) | Function | Точка расширения, предназначенная для обработки щелчков мышью. |
| onItemHover(/*MenuItem*/ item) | Function | Вызывается, когда указатель мыши оказывается над MenuItem. |
| onItemUnhover(/*MenuItem*/ item) | Function | Вызывается, когда указатель мыши покидает пределы MenuItem. |

*Таблица 15.6 (продолжение)*

| Имя | Тип (по умолчанию) | Комментарий |
|---|---|---|
| onCancel() | Function | Точка расширения, которая вызывается, когда пользователь закрывает меню, не щелкнув на каком-либо пункте. |
| onExecute() | Function | Точка расширения, которая вызывается, когда пользователь запускает текущее меню. |

*Таблица 15.7. Прикладной программный интерфейс диджитов MenuItem и PopupMenuItem*

| Имя | Тип | Комментарий |
|---|---|---|
| label | String | Текст, который должен отображаться в MenuItem. |
| iconClass | String | Класс, который применяется к MenuItem, придавая ему внешний вид ярлыка (используется свойство CSS background-image). |
| disabled | Boolean | Указывает, должен ли MenuItem быть в неактивном (запрещенном) состоянии. Значение по умолчанию: false. |
| setDisabled (/*Boolean*/ value) | Function | Используется для программного управления атрибутом disabled в MenuItem. |
| onClick (/*DomEvent*/ evt) | Function | Используется для подключения к MenuItem обработчика события щелчка мышью. |

# TitlePane

TitlePane — это виджет, который всегда отображает заголовок, но его тело может разворачиваться или сворачиваться по мере необходимости. Фактическое изменение размера виджета сопровождается анимационным эффектом. Будучи наследником ContentPane, диджит TitlePane имеет доступ ко всем унаследованным методам загрузки удаленного содержимого, хотя явно они в этом разделе не рассматриваются. (Полное описание диджита ContentPane вы найдете в предыдущей главе.) В примере 15.5 демонстрируется типичное использование TitlePane.

*Пример 15.5. Пример типичного использования диджита TitlePane*

```
<div dojoType="dijit.TitlePane" title="Grocery list:" style="width:300px">
    <ul>
        <li>Eggs</li>
        <li>Milk</li>
        <li>Bananas</li>
```

```
        <li>Coffee</li>
    </ul>
</div>
```

Особенности, которые поддерживаются диджитом TitlePane, перечислены в табл. 15.8.

*Таблица 15.8. Прикладной программный интерфейс диджита TitlePane*

| Имя | Тип | Комментарий |
|---|---|---|
| title | String | Заголовок панели. |
| open | Boolean | Указывает состояние панели – закрыта или открыта. Значение по умолчанию: true. |
| duration | Integer | Длительность воспроизведения анимационного эффекта в миллисекундах. Значение по умолчанию: 250. |
| setContent(/*DomNode\|String*/) | Function | Используется для установки содержимого панели программным способом. |
| setTitle(/* String */ title) | Function | Устанавливает заголовок. |
| toggle() | Function | Если панель открыта, этот метод закрывает ее. Если закрыта, открывает. |

Виджет TitlePane может использоваться как простой статический элемент страницы, однако ему можно найти более интересное применение – как более интерактивному элементу управления. В следующем примере демонстрируется использование TitlePane в качестве имитации стикеров с примечанием, которые можно увидеть во многих приложениях. Для этого вполне достаточно просто вставить такой виджет, как Textarea, внутрь TitlePane и изменять заголовок всякий раз, когда последний из них закрывается, как показано в примере 15.6.

*Пример 15.6. Имитация стикера с примечанием с помощью TitlePane*

```
dojo.addOnLoad(function() {
    var ed = new dijit.form.Textarea({id : "titlePaneContent"});
    dijit.byId("tp").setContent(ed.domNode);
});

// А теперь диджит TitlePane, который можно объявить в разметке:
<div id="tp" dojoType="dijit.TitlePane" style="width:300px">
    <script type="dojo/connect" event="toggle">
        if (!this.open) {
            var t = dijit.byId("titlePaneContent").getValue();
            if (t.length > 15)
                t = t.slice(0,12)+"...";
            this.setTitle(t);
        }
    </script>
</div>
```

Дополнительная стилизация и возможность перетаскивания объектов мышью – это практически все, что необходимо для создания небольшого приложения для работы с такого рода заметками.

# InlineEditBox

InlineEditBox часто описывается как виджет-обертка, которая предоставляет возможность статического отображения того, что в действительности является элементом ввода. Когда потребуется изменить содержимое элемента, пользователь должен просто выбрать его. Например, вместо того чтобы добавлять в страницу элемент редактирования фиксированного размера, такой как TextBox, всегда видимый на экране, его можно обернуть диджитом InlineEditBox, который на экране отображается как обычный текст (подобно метке), но при выборе преобразуется в TextBox, готовый к редактированию. По завершении редактирования, например по нажатию клавиши Enter или по щелчку мыши за пределами диджита, он опять превращается в обычный текст.

В простейшем случае можно просто обернуть TextBox диджитом InlineEditBox, сделав его, например, частью шаблона письма, как показано в следующем примере. Обратите внимание, что та часть, которая должна была бы отображаться как TextBox и загромождала бы изображение, выглядит как обычная разметка, но после щелчка мышью преобразуется в элемент редактирования:

```
Dear <span dojoType="dijit.InlineEditBox" autoSave="false"
    editor="dijit.form.TextBox">Valued Customer</span>:

<div>We have received your request to be removed from our spam list. Not to
worry, we'll remove you when we're good and ready. In the meanwhile, please
do not hesitate to contact us with further complaints.</div>

<div>Sincerely,</div>
<span dojoType="dijit.InlineEditBox" autosave="false"
    editor="dijit.form.TextBox">Customer Service</span>
```

В этом примере атрибут autosave устанавливается в значение false, благодаря чему в элементе управления появляются две кнопки Сохранить (Save)[1] и Отменить (Cancel) (в противном случае текст сохранялся бы автоматически по мере набора и эти кнопки вообще не отображались бы). Это очень важно. Теперь попробуем расширить базовую концепцию и создадим другой шаблон для составления письма.

Ниже приводится короткий пример, где диджитом InlineEditBox обернут диджит Textarea. Обратите внимание, что атрибут renderAsHtml позволяет отображать и редактировать исходный текст разметки:

---

[1] Благодаря интернационализации надписи на кнопках действительно на русском языке! – *Прим. перев.*

```
Dear <span dojoType="dijit.InlineEditBox" autoSave="false"
editor="dijit.form.TextBox">Valued Customer</span>:

<div dojoType="dijit.InlineEditBox" autoSave="false"
editor="dijit.form.Textarea" renderAsHtml="true">
  Insert<br>
  Form<br>
  Letter<br>
  Here<br>
</div>

<div>Sincerely,</div>

<span dojoType="dijit.InlineEditBox"
    autoSave="false" editor="dijit.form.TextBox">Customer Service</span>
```

Как и для других диджитов, рассматривавшихся выше в этой главе, типичное использование InlineEditBox не вызывает сложностей. В табл. 15.9 приводится описание некоторых дополнительных особенностей, о существовании которых следует знать и помнить.

*Таблица 15.9. Прикладной программный интерфейс диджита InlineEditBox*

| Имя | Тип | Комментарий |
|---|---|---|
| editing | Boolean | Индикатор состояния, в котором находится InlineEditBox. Имеет значение true, когда виджет находится в состоянии редактирования. |
| autoSave | Boolean | Указывает, должен ли автоматически сохраняться введенный текст без необходимости явно подтверждать это. Значение по умолчанию: true. |
| buttonSave | String | Текст, отображаемый на кнопке сохранения. Значение по умолчанию: пустая строка. |
| buttonCancel | String | Текст, отображаемый на кнопке отмены. Значение по умолчанию: пустая строка. |
| renderAsHtml | Boolean | Если имеет значение true, элемент редактирования в InlineEditBox отображает исходный текст как разметку HTML. Значение по умолчанию: false. |
| editor | String | Имя класса диджита, который будет играть роль элемента редактирования. Значение по умолчанию: dijit.form.TextBox. |
| editorParams | Object | Любые параметры, которые должны быть переданы конструктору элемента редактирования в InlineEditBox. |
| width | String | Ширина элемента редактирования. Значение по умолчанию: 100%. |
| value | String | Значение, отображаемое виджетом, когда он не находится в режиме редактирования. |

*Таблица 15.9 (продолжение)*

| Имя | Тип | Комментарий |
|---|---|---|
| noValueIndicator | String | Строка-заполнитель, которая должна отображаться, когда в виджете отсутствует текстовое значение (чтобы пользователь мог щелкнуть на виджете и перевести его в режим редактирования). Значение по умолчанию: строка специального вида. |
| setDisabled(/*Boolean*/ disabled) | Function | Используется для деактивации и активации виджета. |
| setValue(/*String*/val) | Function | Устанавливает значение виджета. |
| save | Function | Сохраняет содержимое элемента редактирования и переводит виджет в режим отображения. |
| cancel | Function | Отменяет результаты редактирования и переводит виджет в режим отображения. |
| onChange | Function | Точка расширения, которая может использоваться для обработки события изменения значения. |
| enableSave | Function | Допускает замену пользовательской функцией, которая может активировать или деактивировать кнопку сохранения. (Например, кнопку сохранения можно деактивировать при несоблюдении некоторых условий в редакторе.) |

# Tree

Диджит Tree является одним из удивительнейших инженерных достижений. Построенный исключительно на основе использования DHTML, он выглядит и действует, как иерархическое дерево. Он поддерживает операции перетаскивания элементов дерева мышью и обладает достаточной гибкостью, чтобы его можно было связать с произвольным источником данных. Как и в случае с любым другим сложным механизмом, прежде чем приступить к манипулированию им, необходимо получить некоторые фундаментальные сведения о нем; они станут легко понятны, как только все факты будут связаны воедино. Это один из самых длинных разделов в главе, потому что диджит Tree является чрезвычайно мощным и предлагает очень широкий набор возможностей. Мы не будем углубляться в вопросы, связанные с обеспечением доступности, тем не менее вы должны знать, что диджит Tree полностью может управляться при использовании клавиатуры с помощью клавиш управления курсором, клавиши Enter и т. д.

Хорошее понимание принципов, заложенных в dojo.data, будет особенно полезно при работе с диджитом Tree. За дополнительной информацией обращайтесь к главе 9.

Прежде чем перейти к изучению программного кода, совсем нелишним будет иметь представление о следующем:

*Деревья и леса*

*Дерево* – это иерархическая структура данных, содержащая единственный корневой элемент. С другой стороны, *лес* – это иерархическая структура данных, напоминающая дерево, но в отличие от него имеющая несколько корневых узлов. Как будет показано ниже, различение дерева и леса – это распространенная проблема, потому что во многих случаях данные удобно представлять в виде дерева с единственным корневым узлом, несмотря на то, что фактически данные представлены в виде леса с несколькими неявными корневыми узлами.

*Узлы*

Дерево – это иерархическая организация узлов и связей между ними. Специальный тип узла, используемый в dijit.Tree, – dijit._TreeNode. Начальный символ подчеркивания в имени в данном случае говорит о том, что диджиты _TreeNode никогда не должны использоваться за пределами диджита Tree. Однако, _TreeNode обладает некоторыми свойствами, которыми удобно управлять непосредственно, в чем вы убедитесь при изучении следующих ниже примеров.

*Независимость от данных*

Диджит Tree не делает никаких предположений об источнике данных, наполняющих его. До версии 1.1 он напрямую обращался к реализации интерфейсов dojo.data, обладающих высокой гибкостью и предоставляющих единообразный способ доступа к данным, но начиная с версии 1.1 между моделью dojo.data и диджитом Tree были добавлены промежуточные уровни. Это, соответственно, dijit.tree.TreeStoreModel и dijit.tree.ForestStoreModel. В значительной степени такое изменение было обусловлено стремлением сделать диджит Tree более устойчивым и управляемым при выполнении операций «перетащил и бросил».

В результате выполнения инструкции dojo.require("dijit.Tree") ForestStoreModel и TreeStoreModel автоматически подключаются вместе с диджитом Tree.

## Простое дерево

В качестве введения в возможности, которые может предложить диджит Tree, предположим, что имеются простые данные, доступ к кото-

рым может быть организован с помощью механизма dojo.data.ItemFileReadStore:

```
{
    identifier : 'name',

    label : 'name',

    items : [
        {
            name : 'Programming Languages',
            children: [
                {name : 'JavaScript'},
                {name : 'Python'},
                {name : 'C++'},
                {name : 'Erlang'},
                {name : 'Prolog'}
            ]
        }
    ]
}
```

Пока все просто. Чтобы на стороне клиента не выполнять разбор данных вручную, эту работу можно поручить dojo.data. Задействование механизма ItemFileReadStore выполняется очень просто, для этого достаточно указать URL источника данных и запросить эти данные. Приведенный далее тег, после того как он будет проанализирован парсером, мог бы справиться с этой задачей – при условии, что в рабочем каталоге присутствует файл с данными, который называется *programmingLanguages.json*, а сам диджит имеет глобальный идентификатор dataStore, обеспечивающий доступ к нему:

```
<div dojoType="dojo.data.ItemFileReadStore"
    jsId="dataStore" url="./programmingLanguages.json"></div>
```

Однако, прежде чем данные попадут в диджит Tree, их необходимо передать промежуточному звену – TreeStoreModel. (К использованию ForestStoreModel мы подойдем чуть ниже.) Полный перечень особенностей, составляющих прикладной программный интерфейс TreeStoreModel, будет представлен немного ниже, а пока достаточно будет знать, что в TreeStoreModel следует настроить ссылку на ItemFileReadStore и определить текст запроса. Следующий диджит TreeStoreModel обращается к механизму dojo.data с глобальным идентификатором dataStore за получением всех значений атрибута name:

```
<div dojoType="dijit.tree.TreeStoreModel" jsId="model" store="dataStore"
     query="{name:'*'}"></div>
```

Теперь единственное, что осталось сделать, – это определить ссылку на TreeStoreModel внутри диджита Tree, как показано ниже:

```
<div dojoType="dijit.Tree" model="model"></div>
```

Ниже, в примере 15.7, все соединяется в одно целое, а полученный результат демонстрируется на рис. 15.3.

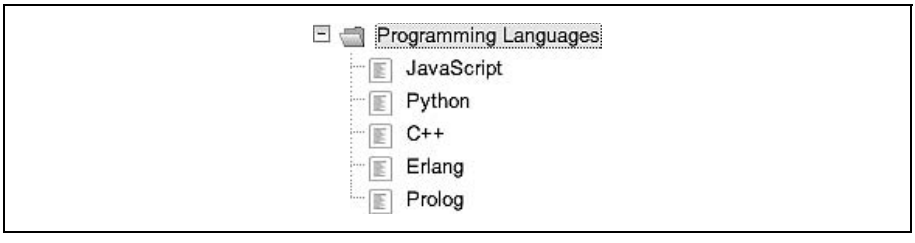



***Рис. 15.3.*** *Диджит Tree, отображающий данные, полученные из хранилища; щелчок на развернутом узле приводит к его сворачиванию*

*Пример 15.7. Простое дерево с одним корневым узлом*

```
<html>
    <head>
        <title>Tree Fun!</title>

        <link rel="stylesheet" type="text/css"
         href="http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css" />
        <link rel="stylesheet" type="text/css"
        href="http://o.aolcdn.com/dojo/1.1/dijit/themes/tundra/tundra.css" />

        <script
            type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js"
            djConfig="parseOnLoad:true,isDebug:true">
        </script>

        <script type="text/javascript">
            dojo.require("dijit.Tree");
            dojo.require("dojo.data.ItemFileReadStore");
            dojo.require("dojo.parser");
        </script>
    </head>
    <body class="tundra">
        <div dojoType="dojo.data.ItemFileReadStore" jsId="dataStore"
             url="./programmingLanguages.json"></div>
        <div dojoType="dijit.tree.TreeStoreModel" jsId="model"
             store="dataStore" query="{name:'*'}"></div>
        <div dojoType="dijit.Tree" model="model"></div>
    </body>
</html>
```

## Простой лес

Во многих приложениях используются данные, для представления которых необходимо несколько корневых узлов, поэтому скорректируем

предыдущий пример так, чтобы он работал с лесом, а не с деревом, и можно было бы заметить различия между подходами. Для начала нам необходимы данные, имеющие несколько корневых узлов. Взгляните на следующий пример, где языки программирования перечислены в виде леса, полученного в результате исключения корневого узла «programming languages» (языки программирования):

```
{
    identifier : 'name',
    label : 'name',
    items : [
        {
            name : 'Object-Oriented',
            type : 'category',
            children: [
                {name : 'JavaScript', type : 'language'},
                {name : 'Java', type : 'language'},
                {name : 'Ruby', type : 'language'}
            ]
        },
        {
            name : 'Imperative',
            type : 'category',
            children: [
                {name : 'C', type : 'language'},
                {name : 'FORTRAN', type : 'language'},
                {name : 'BASIC', type : 'language'}
            ]
        },
        {
            name : 'Functional',
            type : 'category',
            children: [
                {name : 'Lisp', type : 'language'},
                {name : 'Erlang', type : 'language'},
                {name : 'Scheme', type : 'language'}
            ]
        }
    ]
}
```

Как видите, в измененных данных, представленных в формате JSON, отсутствует единый корневой узел, поэтому такие данные можно представить только в виде леса. Единственное существенное изменение по сравнению с примером 15.7 заключается в необходимости добавления в диджит `Tree` параметра `showRoot`, чтобы скрыть его корень; в изменении запроса, чтобы идентифицировать узлы верхнего уровня, и в замене `TreeStoreModel` на `ForestStoreModel`. В примере 15.8 приводится измененный код разметки, в котором различия выделены жирным шрифтом.

*Пример 15.8. Изменения, необходимые для представления данных не в виде дерева, а в виде леса*

```
<body class="tundra">
    <div dojoType="dojo.data.ItemFileReadStore" jsId="dataStore"
         url="./programmingLanguages.json"></div>
    <div dojoType="dijit.tree.ForestStoreModel" jsId="model" store="dataStore"
         query="{type:'category'}"></div>
    <div dojoType="dijit.Tree" model="model" showRoot=false></div>
</body>
```

То, что данные имеют структуру, которую можно отобразить как лес, еще не означает, что их нельзя отобразить в виде дерева. В примере 15.9 демонстрируется, как в данных, получаемых с помощью механизмов `dojo.data`, можно сфабриковать корневой узел с помощью атрибутов `rootId` и `rootLabel` в `ForestStoreModel`.

*Пример 15.9. Изменения, создающие корневой узел, чтобы лес выглядел как дерево*

```
<body class="tundra">
    <div dojoType="dojo.data.ItemFileReadStore" jsId="dataStore"
         url="./programmingLanguages.json"></div>
    <div dojoType="dijit.tree.ForestStoreModel" jsId="model" store="dataStore"
         query="{type:'category'}" rootId="root"
         rootLabel="Programming Languages"></div>
    <div dojoType="dijit.Tree" model="model" ></div>
</body>
```

С практической точки зрения сфабрикованный корневой узел теперь может участвовать в операциях, выполняемых средствами интерфейса `dojo.data`, такими как `getLabel` и `getValue`. На первый взгляд в этом нет ничего особенного, но благодаря такому подходу теперь можно заниматься работой с данными, не отвлекаясь на необходимость обрабатывать особые случаи. На рис. 15.4 показано, как выглядит простой лес.

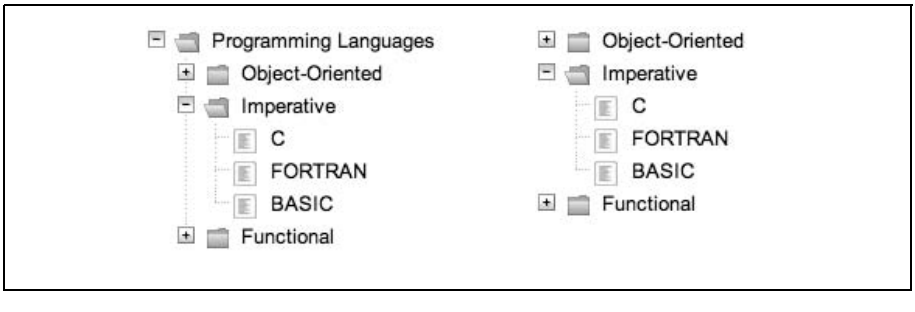



***Рис. 15.4.*** *Слева: диджит Tree (со сфабрикованным корневым узлом), отображающий те же самые данные; справа: диджит Tree (без корневого узла), отображающий данные в виде леса*

## Реакция на событие щелчка мышью

Иметь возможность отображения данных в виде дерева хорошо уже само по себе, но разве не было бы еще лучше иметь возможность обрабатывать такие события, как щелчок мышью? Давайте реализуем точку расширения onClick, чтобы продемонстрировать возможность обрабатывать щелчки мышью на различных элементах. В функцию onClick для обработки передаются как сам узел _TreeNode, на котором произошел щелчок, так и элемент данных dojo.data. В примере 15.10 показаны изменения, которые необходимо внести, чтобы реализовать обработку щелчка.

*Пример 15.10. Обработка щелчков мышью в диджите Tree*

```
<body class="tundra">
    <div dojoType="dojo.data.ItemFileReadStore" jsId="dataStore"
         url="./programmingLanguages.json"></div>
    <div dojoType="dijit.tree.ForestStoreModel" jsId="model" store="dataStore"
         query="{type:'category'}" rootId="root"
         rootLabel="Programming Languages"></div>
    <div dojoType="dijit.Tree" model="model" >
        <script type="dojo/method" event="onClick" args="item,treeNode">
            //использовать элемент данных или узел по желанию...
           console.log("onClick:",dataStore.getLabel(item)); //вывести метку
        </script>
    </div>
</body>
```

Обратите внимание: несмотря на наличие между Tree и dojo.data промежуточной модели, обеспечивающей дополнительный уровень абстракции данных, мы по-прежнему имеем возможность прямого доступа к элементам данных. Здесь нет никакой необходимости использовать промежуточную модель, которая упрощает возможность *отображения* данных, но является совершенно ненужной прослойкой между элементом dojo.data и обычными средствами доступа к нему.

## Прикладной программный интерфейс деревьев

Если вы следовали за примерами и понимаете, как работают интерфейсы dojo.data, то вы уже знаете о диджите Tree больше, чем может показаться. В табл. 15.10 приводится полный перечень особенностей прикладного программного интерфейса диджита Tree, с которыми желательно ознакомиться, прежде чем переходить к следующему разделу, где будет описываться возможность реализации механизма «перетащил и бросил» для диджита Tree. Как видите, сам диджит Tree предоставляет лишь несколько простых атрибутов, при этом основные механизмы реализации убраны за кулисы, в класс dijit.tree.model.

В версии 1.1 до сих пор сохраняется возможность прямого подключения диджита `Tree` к хранилищу данных через интерфейс `dojo.data`. Однако такая возможность, скорее всего, будет исключена из версии 2.0, поэтому она не рассматривается в этой главе и соответствующие ей элементы прикладного программного интерфейса не включены в следующий ниже список.

*Таблица 15.10. Прикладной программный интерфейс диджита Tree*

| Имя | Тип | Комментарий |
|---|---|---|
| `model` | `dijit.tree.model` | Интерфейс, обеспечивающий универсальный доступ к данным. |
| `query` | Object | Запрос на получение данных, который возвращает элемент(ы) верхнего уровня в дереве. Если запрос возвращает единственный элемент, в качестве промежуточной модели следует использовать `TreeStoreModel`, в противном случае — `ForestStoreModel`. |
| `showRoot` | Boolean | Указывает, должен ли диджит `Tree` отображать корневой узел. Обычно используется для сокрытия корневого узла в случае применения модели `ForestStoreModel`. |
| `childrenAttr` | Array | Массив строк, в котором перечислены имена атрибутов, хранящих дочерние узлы в дереве данных. Значение по умолчанию: `["children"]`. |
| `openOnClick` | Boolean | Если имеет значение `true`, щелчок мышью на метке узла открывает его (в противоположность точке расширения `onClick`, которая должна выполнять открытие узла наряду с другими действиями). Значение по умолчанию: `false` |
| `persist` | Boolean | Указывает, следует ли использовать cookies для сохранения состояния узлов (свернут/развернут). Значение по умолчанию: `true`. |
| `onClick(/*dojo.data.Item*/item, /*TreeNode*/node)` | Function | Точка расширения, используемая для обработки события щелчка мышью (а также события нажатия клавиши Enter) на узле дерева. Обработчику передаются как сам узел дерева, так и элемент данных. |

Далее в табл. 15.11 приводится список особенностей, составляющих прикладной программный интерфейс dijit.Tree.model. Все, что представлено в этой таблице, относится к модели, такой как TreeStoreModel, использовавшейся в предыдущих примерах. Как и в случае с любым другим интерфейсом, это означает, что, по сути, вы можете создать любую абстракцию, необходимую для заполнения диджита Tree, при условии, что она будет отвечать требованиям спецификации независимо от используемого источника данных, будь то dojo.data, какой-то другой открытый или полностью закрытый интерфейс.

*Таблица 15.11. Прикладной программный интерфейс dijit.Tree.TreeStoreModel*

| Имя | Комментарий |
|---|---|
| getRoot(/*Function*/onItem, /*Function*/onError) | Используется для обхода элементов дерева. Вызывает функцию onItem, передавая ей корневой узел дерева, который может быть сфабрикованным узлом. Возбуждает ошибку onError в случае ошибки. |
| mayHaveChildren(/*dojo.data.Item*/ item) | Используется для обхода элементов дерева. Возвращает информацию о том, может ли указанный узел иметь дочерние узлы, что очень удобно, так как не всегда эффективно проверять фактическое наличие дочерних узлов, перед тем как развернуть узел дерева. |
| getChildren(/*dojo.data.Item*/ parentItem, /*Function*/onComplete) | Используется для обхода элементов дерева. Вызывает функцию onComplete, которой передаются все дочерние узлы данного узла parentItem. |
| getIdentity(/*dojo.data.Item*/ item) | Используется для проверки узлов. Возвращает идентификатор элемента данных. |
| getLabel(/*dojo.data.Item*/item) | Используется для проверки узлов. Возвращает метку элемента данных. |
| newItem(/*Object?*/args, /*dojo.data.Item?*/parent) | Составляет часть интерфейса Write. Создает новый элемент данных dojo.data в соответствии с требованиями, предъявляемыми к интерфейсу dojo.data.api.Write. |
| pasteItem(/*dojo.data.Item*/ childItem, /*dojo.data.Item*/oldParentItem, /*dojo.data.Item*/newParentItem, /*Boolean*/copy) | Составляет часть интерфейса Write. Перемещает или копирует элемент данных из одного родительского элемента в другой. Используется в операциях «перетащил и бросил». Если параметр oldParentItem указан и параметр copy содержит значение false, дочерний элемент удаляется из oldParentItem. Если параметр newParentItem указан, элемент childItem присоединяется к нему. |

| Имя | Комментарий |
|---|---|
| onChange(/*dojo.data.Item*/item) | Функция обратного вызова, используемая для изменения метки или ярлыка. Изменения в дочерних элементах приводят к вызову onChildrenChange, поэтому такие изменения в onChange, скорее всего, должны игнорироваться. |
| onChildrenChange(/*dojo.data.Item*/ parent, /*Array*/ newChildren) | Функция обратного вызова, используемая для обработки событий добавления новых, изменения или удаления существующих элементов. |
| destroyRecursive() | Уничтожает объект и разрывает соединения с хранилищем данных. Освободившиеся ресурсы будут утилизированы сборщиком мусора. |

Модель ForestStoreModel (табл. 15.12), реализованная поверх TreeStoreModel, предоставляет две дополнительные функции, отвечающие на события, связанные со сфабрикованным корневым узлом, а именно – на события добавления и удаления элементов верхнего уровня. Эти функции необходимы для корректировки критерия запроса, чтобы верхний уровень дерева был доступен и после внесения изменений. Будучи независимым от представления данных, диджит Tree сам по себе не несет никакой ответственности за обновление или манипулирование элементами данных – это бремя ложится на плечи программиста, который должен гарантировать соответствие критериям запроса. То есть главная задача этих дополнительных функций состоит в том, чтобы дать ему такую возможность.

*Таблица 15.12. Прикладной программный интерфейс dijit.Tree.TreeStoreModel*

| Имя | Комментарий |
|---|---|
| onAddToRoot(/*dojo.data.Item*/item) | Вызывается, когда элемент item добавляется на верхний уровень дерева. Этот метод необходимо переопределять для изменения элемента так, чтобы он соответствовал критерию запроса, отбирающего элементы данных верхнего уровня. |
| onLeaveRoot(/*dojo.data.Item*/item) | Вызывается, когда элемент item удаляется с верхнего уровня дерева. Этот метод необходимо переопределять для изменения элемента так, чтобы он больше не соответствовал критерию запроса, отбирающего элементы данных верхнего уровня. |

Для корректировки элемента данных в примере 15.9, чтобы он соответствовал критерию запроса, выбирающему элементы верхнего уровня, достаточно заменить значение "language" атрибута type значением "category". Например, можно было бы элемент "Java" переместить на верхний уровень, определив его тип как "category" и предоставив операцию для добавления отдельных реализаций Java (со значением "language" в атрибуте type) в качестве дочерних элементов. Как будет показано в следующем разделе, наиболее типичными случаями, когда возникает необходимость следовать этим соглашениям, являются ситуации реализации механизма «перетащил и бросил».

## Операция «перетащил и бросил» в дереве

Расширения интерфейса dijit.tree.model, обсуждавшиеся в предыдущем разделе, реализованы так объемно, чтобы сделать операции «перетащил и бросил» для диджита Tree более простыми и непротиворечивыми. Вообще говоря, операции «перетащил и бросил» нельзя отнести к разряду универсальных операций, поэтому готовьтесь к тому, что вам придется засучить рукава, если вы хотите в конечном итоге получить надежную реализацию любого сложного виджета, способного участвовать в операциях перетаскивания мышью. Особенно важно уделить достаточное время, чтобы сформулировать ответы на следующие общие вопросы:

- Что должно произойти в момент начала перетаскивания?
- Что должно произойти в момент сброса?
- Что должно произойти в случае отмены операции сброса?

Текущая архитектура дерева, на основе которой необходимо реализовать механизм «перетащил и бросил», влечет за собой необходимость реализации обширного набора функций, определяемых модулем dojo.dnd (представленном в главе 7), и передачу их в диджит Tree посредством атрибута dndController. Поскольку начинать такую реализацию на пустом месте достаточно сложно, версия 1.1 включает в себя модуль dijit._tree, содержащий шаблонную реализацию, из которой вы можете использовать все, что сочтете нужным. Вы можете создать подкласс и переопределить в нем требуемые части, вы можете подмешивать в него другие классы или использовать имеющуюся реализацию в качестве примера для создания своей реализации с самого начала. Поскольку в результате ваших усилий должен получиться класс, напоминающий класс dojo.dnd.Source и способный соответствующим образом взаимодействовать с реализацией dijit.tree.model, лежащей в основе диджита Tree, то вы должны быть в хорошей спортивной форме. В частности, ваша реализация класса Source должна содержать, по крайней мере, ключевые методы, перечисленные в табл. 15.13, которые ожидает получить диджит Tree через атрибут dndController.

*Таблица 15.13. Интерфейс dndController для диджита Tree*

| Имя | Комментарий |
|---|---|
| `onDndDrop(/*Object*/source, /*Array*/nodes, /*Boolean*/copy)` | Обработчик темы события `/dnd/drop`, вызываемый для завершения операции сброса, которая влечет за собой необходимость изменения элементов данных в соответствии с начальной и конечной точкой операции перемещения, чтобы дерево могло обновить себя. |
| `onDndCancel()` | Обработчик темы события `/dnd/cancel`, обслуживающий факт отмены операции. |
| `checkAcceptance(/*Object*/source, /*Array*/nodes)` | Используется для проверки цели на способность принять узлы `nodes`, перемещаемые из источника `source`. Часто используется, чтобы запретить сброс, основываясь на некоторых свойствах узлов. |
| `checkItemAcceptance(/*DOMNode*/target, /*Object*/source)` | Используется для проверки цели на способность принять узлы `nodes`, перемещаемые из источника `source`. Часто используется, чтобы запретить сброс, основываясь на некоторых свойствах цели. |
| `itemCreator(/*Array*/nodes)` | Когда перемещение выполняется между двумя различными источниками данных, в принимающем наборе данных необходимо создать соответствующие элементы. Этот метод обеспечивает средство создания таких элементов, если исходная и конечная точки перемещения опираются на разные источники данных. |

Важное замечание по поводу функций `dndController`: если ссылки на эти функции присутствуют в разметке, они должны быть объявлены как глобальные переменные к моменту, когда парсер начнет парсинг диджита `Tree`. То есть они не могут быть объявлены в блоке `dojo.addOnLoad`, потому что он выполняется уже после того, как парсер завершит свою работу. Однако вы можете вообще не использовать функции интерфейса `dndController` в разметке, а связывание их выполнять в блоке `addOnLoad`. Такой подход используется в примере, следующем ниже.

Чрезвычайно важно понять, что в операцию перетаскивания вовлечены узлы DOM, а не диджиты `_TreeNode`. Однако чаще всего вам будут требоваться именно `_TreeNode`, потому что только они содержат информацию об интересующих вас данных, а узлы DOM такой информации

не имеют. Всякий раз, когда будет возникать такая потребность, а она всегда будет возникать внутри методов из табл. 15.13, можно использовать функцию dijit.getEnclosingWidget, которая преобразует узел DOM в соответствующий ему диджит _TreeNode.

## Пример реализации операции «перетащил и бросил» в дереве

Эти методы являются настолько общими, что они могут передаваться диджиту Tree на этапе конструирования, что особенно замечательно, так как позволяет максимально использовать существующую реализацию в dijit._tree. Теперь настало время для еще одного примера.

Возьмем за основу пример 15.9 и дополним его возможностью перемещать элементы дерева мышью. Чтобы минимизировать прилагаемые для этого усилия, будем опираться на шаблонную реализацию dijit._tree. Кроме того, обратите внимание, что нам придется отказаться от механизма ItemFileReadStore и задействовать ItemFileWriteStore, так как сама природа операции перетаскивания не предполагает доступ только для чтения.

На первый взгляд может создаться ощущение, что диджит Tree сам обновляет себя при взаимодействии с ним, т. к. его отображение меняется при выполнении операции перетаскивания, однако не забывайте, что диджит Tree всего лишь отображает данные. Любые внешние изменения являются результатом изменений, производимых в наборе данных, то есть изменение в данных запускает изменение в отображении.

Чтобы пример не противоречил здравому смыслу, нам необходимо предотвратить возможность сброса элементов на другие элементы, так как элементы по своей сути отличаются от категорий элементов, как мы их определили в уже описанном ранее хранилище dojo.data. Реализация приводится в примере 15.11, а на рис. 15.5 показано, как это выглядит.

*Пример 15.11. Простое дерево с возможностью перетаскивания элементов*

```
<html>
    <head>
        <title>Drag and Droppable Tree Fun!</title>

        <link rel="stylesheet" type="text/css"
         href="http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css" />
        <link rel="stylesheet" type="text/css"
        href="http://o.aolcdn.com/dojo/1.1/dijit/themes/tundra/tundra.css" />

        <script
            type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js"
            djConfig="parseOnLoad:true,isDebug:true">
        </script>
```

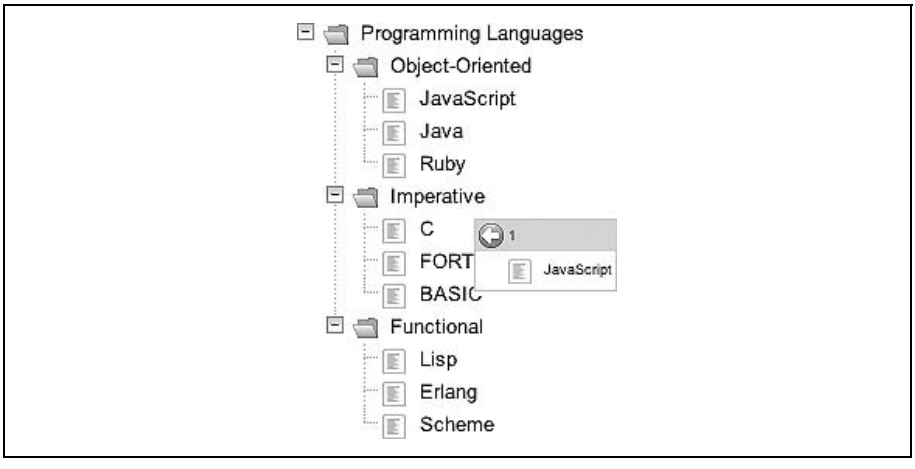



***Рис. 15.5.*** *Перемещение языка программирования в другую категорию*

```
        <script type="text/javascript">
            dojo.require("dijit.Tree");
            dojo.require("dojo.data.ItemFileWriteStore");
            dojo.require("dijit._tree.dndSource");
            dojo.require("dojo.parser");

            dojo.addOnLoad(function() {
                //связать обработчик checkItemAcceptance...
                dijit.byId("tree").checkItemAcceptance = function(target,
                                                                  source) {
                    //преобразовать цель (узел DOM) в узел дерева
                    //и затем получить элемент данных
                    var item = dijit.getEnclosingWidget(target).item;

                    //запретить сброс в корневой (сфабрикованный) узел
                    //и не допускать сброс на элементы, не являющиеся
                    //категориями
                    return (item.id != "root" && item.type == "category");
                }
            });
        </script>
    </head>
    <body class="tundra">
        <div dojoType="dojo.data.ItemFileWriteStore" jsId="dataStore"
             url="./programmingLanguages.json"></div>
        <div dojoType="dijit.tree.ForestStoreModel" jsId="model"
             store="dataStore" query="{type:'category'}" rootId="root"
             rootLabel="Programming Languages"></div>
        <div id="tree" dojoType="dijit.Tree" model="model"
             dndController="dijit._tree.dndSource"></div>
    </body>
</html>
```

Когда у вас появится потребность реализовать возможность перетаскивания элементов внутри диджита `Tree`, стоит потратить некоторое время на изучение шаблонной реализации в `dijit._tree`. Любой случай применения механизма перетаскивания обычно имеет свои особенности, поэтому вероятность найти готовое решение, не требующее приложения дополнительных усилий, близка к нулю.

# Editor

Во время работы над этой книгой диджит `Editor` и его расширяемая архитектура не раз подвергались существенным дополнениям и изменениям. По этой причине в данном разделе не так подробно, как в других разделах, обсуждаются технические детали.

С каждым годом увеличивается число приложений, использующих возможность редактирования текста со сложным форматированием. Можно даже сказать, что если приложение, имея привлекательный интерактивный интерфейс, вдруг предоставляет пользователю обычный элемент `textarea` (пусть даже диджит `Textarea`), это будет восприниматься, по меньшей мере, как нечто неестественное. К счастью, диджит `Editor` обладает типичной функциональностью по редактированию текста со сложным форматированием, а его использование не требует от вас приложения чрезмерных усилий.

Перед прочтением этого раздела вам будет интересно ознакомиться со следующим справочным руководством: *http://developer.mozilla.org/en/docs/Rich-Text_Editing_in_Mozilla*.

Инструментарий Dojo опирается на использование элементов управления, реализованных в самом броузере, позволяющих редактировать содержимое. В качестве короткого экскурса в историю: в Internet Explorer 4.0 была реализована концепция *режима разработки* (*design mode*), в котором стало возможным редактировать текст способом, напоминающим простой редактор текста со сложным форматированием. Вслед за Microsoft в Mozilla 1.3 был также реализован практически идентичный прикладной программный интерфейс, который в конечном итоге был формализован в виде спецификации Midas Specification (*http://www.mozilla.org/editor/midas-spec.html*). Другие броузеры тоже последовали в этом направлении, хотя и с некоторыми незначительными отклонениями. В любом случае, самое тяжелое – это сначала сделать документ редактируемым, а потом вызвать JavaScript-функцию `execCommand`, чтобы создать фактическую разметку. Следуя спецификации Midas Specification, поставленную задачу могли бы решить следующие строки программного кода:

```
// Сделать узел редактируемым... это может быть div с установленной
// шириной и высотой
document.getElementById("foo").contentDocument.designMode="on";
```

```
/* Выделить некоторый текст... */

// Придать выделенному тексту курсивное начертание.
// Никаких дополнительных аргументов не требуется.
editableDocument.execCommand("Italic", false, null);
```

Представьте теперь, что мы могли бы использовать весь арсенал команд манипулирования содержимым с помощью функции `execCommand`, стандартизовать различия между броузерами, скомпоновать удобную инструментальную панель, добавить привлекательную стилизацию и оформить все это в виде переносимого виджета. В действительности именно это и делает диджит `Editor` из библиотеки Dijit. Виджет `Editor` обеспечивает очень большое разнообразие возможностей, среди которых некоторые на первый взгляд могут показаться лишними, тем не менее освоить базовый набор возможностей совсем несложно. В примере 15.12 демонстрируется использование виджета `Editor` в разметке – с применением простого стиля и парой кнопок для взаимодействия с ним.

Вообще без применения стиля виджет `Editor` не имеет границ, занимает всю ширину контейнера и по умолчанию имеет высоту 300 пикселей. Применение стиля в данном случае просто обеспечивает цвет фона и корректирует высоту виджета `Editor`, чтобы она оказалась немного меньше высоты контейнера и содержимое редактора не выходило за пределы видимого прямоугольника и не «наезжало» на кнопки.

*Пример 15.12. Пример типичного использования диджита Editor*

```
<div style="margin:5px;background:#eee; height: 400px; width:525px">
    <div id="editor" height="375px" dojoType="dijit.Editor">
        When shall we three meet again?<br>
        In thunder, lightning, or in rain?
    </div>
</div>
<button dojoType="dijit.form.Button">Save
    <script type="dojo/method" event="onClick" args="evt">
        /* Сохраните содержимое редактора любым способом по вашему выбору */
        console.log(dijit.byId("editor").getValue());
    </script>
</button>
<button dojoType="dijit.form.Button">Clear
    <script type="dojo/method" event="onClick" args="evt">
        dijit.byId("editor").replaceValue("");
    </script>
</button>
```

Вам определенно стоит потратить некоторое время, чтобы поэкспериментировать с диджитом `Editor` и самим убедиться в возможности использовать все его функциональные возможности при очень незначительных усилиях, без чего будет трудно поверить, что это правда. Об-

ратите внимание, что Editor отображает простую разметку HTML, поэтому при сохранении и восстановлении содержимого не придется выполнять промежуточное преобразование. Потом, когда вы будете готовы приступить к более подробному ознакомлению с возможностями виджета Editor, просмотрите список его особенностей, что приводится в табл. 15.14.

Прикладной программный интерфейс диджита Editor является самым сложным в библиотеке Dijit, и во время работы над этой книгой предпринимались серьезные меры по его рефакторингу с целью упрощения. По этой причине следующая таблица содержит лишь малую часть прикладного программного интерфейса, включающую наиболее полезные составляющие. Полный перечень свойств и методов вы найдете в файле с исходными текстами, если вам *действительно* захочется разобраться с диджитом Editor.

*Таблица 15.14. Малая часть прикладного программного интерфейса диджита Editor*

| Имя | Тип | Комментарий |
|---|---|---|
| focusOnLoad | Boolean | Указывает, должен ли Editor получить фокус ввода после загрузки страницы. |
| height | String | Начальная высота диджита Editor. Значение по умолчанию: 300px. |
| inheritWidth | Boolean | Если имеет значение true, наследует ширину родительского элемента, в противном случае простирается по всей ширине. Значение по умолчанию: false. |
| minHeight | String | Минимально допустимая высота. Значение по умолчанию: 1em. |
| name | String | Если определено, содержимое редактора сохраняется, когда пользователь покидает страницу, и восстанавливается при возвращении. |
| plugins | Array | Модули расширения, которые должны загружаться с базовым редактором. По умолчанию подключаются такие общие значения, как bold, italic, underline. |
| extraPlugins | Array | Некоторые модули расширения, которые должны загружаться в дополнение к базовым модулям. |
| getValue() | Function | Возвращает значение виджета Editor. |
| setValue(/*String*/val) | Function | Устанавливает значение виджета Editor. |
| undo() | Function | Отменяет предыдущее действие. |

| Имя | Тип | Комментарий |
|---|---|---|
| `onDisplayChanged(/*Event*/evt)` | Function | Подключает обработчик этого события для выполнения действий при изменении отображения. |
| `close()` | Function | Закрывает диджит `Editor` и сериализует содержимое редактора в его узел. |
| `contentPreFilters` | Array | Функции, которые при необходимости могут применяться к тексту в процессе десериализации до того, как он будет преобразован в дерево DOM. |
| `contentDomPreFilters` | Array | Функции, которые при необходимости могут применяться к дереву DOM в процессе десериализации до того, как он будет загружен для редактирования. |
| `contentDomPostFilters` | Array | Функции, которые при необходимости могут применяться к дереву DOM перед его сериализацией. |
| `contentDomFilters` | Array | Функции, которые при необходимости могут применяться к тексту перед его сериализацией. |
| `execCommand()` | Function | Выполняет команду форматирования текста. Ведет себя как обычная функция `execCommand`, но учитывает расхождения в реализациях различных броузеров. |

## Архитектура диджита Editor

Жизненный цикл диджита `Editor` состоит из трех основных этапов, как показано на рис. 15.6. В следующем списке перечислены все эти этапы и дается краткое описание каждого из них:

*Десериализация содержимого*

На этапе загрузки производится загрузка текстового содержимого из узла DOM, преобразование его в дерево DOM и отображение, чтобы пользователь мог приступить к редактированию. К текстовому потоку и к дереву DOM могут применяться последовательности функций JavaScript, чтобы в случае необходимости отфильтровать и преобразовать содержимое. В качестве типичного примера фильтра можно привести функцию, выполняющую преобразование символов завершения строки в простом текстовом документе в теги `<br>`, чтобы текст корректно отображался в редакторе HTML.

*Взаимодействие с содержимым*

Этап взаимодействия ничем не отличается от редактирования в любом другом редакторе, поддерживающем сложное форматирование текста. На протяжении этого этапа могут производиться разнооб-

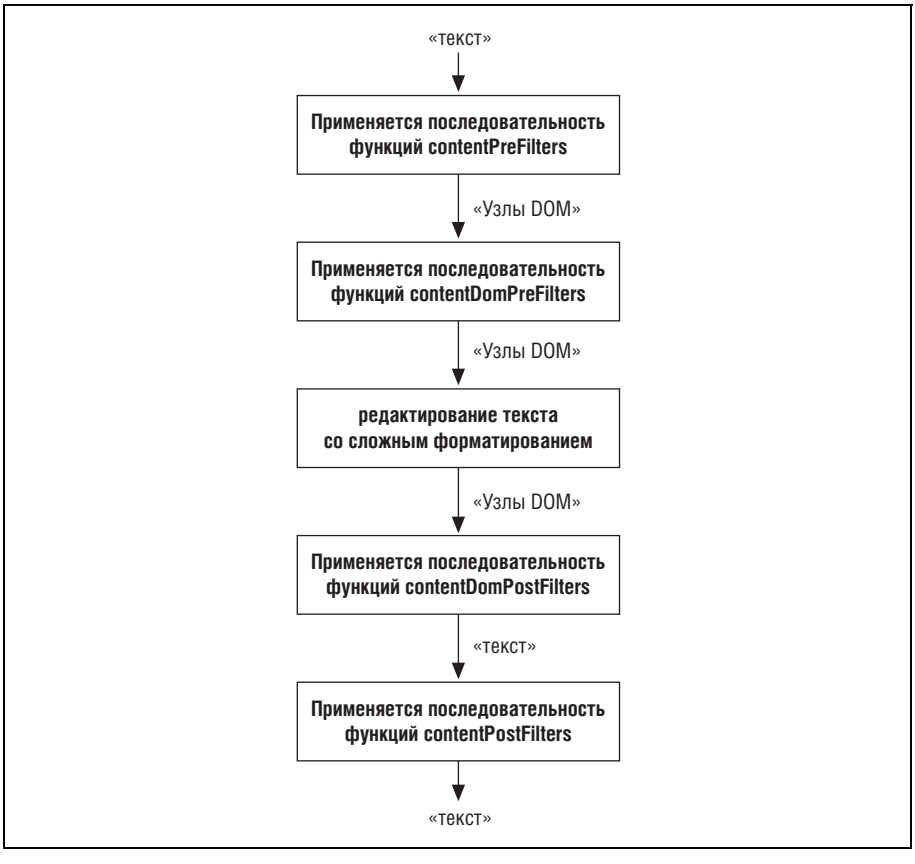



***Рис. 15.6.*** *Основные этапы, поддерживаемые архитектурой диджита Editor*

разные операции, такие как форматирование текста, которые сохраняются в стеке отмены либо через определенные интервалы времени, либо всякий раз, когда изменяется отображение.

*Сериализация содержимого*

По окончании редактирования, когда вызывается метод `close` диджита `Editor`, содержимое сериализуется из дерева DOM обратно в поток текста, который затем записывается в исходный узел. В этот момент обработчик события может выполнить отправку содержимого на сервер для сохранения. Как и на этапе десериализации, к содержимому могут применяться последовательности функций JavaScript.

## Модули расширения диджита Editor

Несмотря на то что диджит `Editor` предоставляет множество очень полезных особенностей, тем не менее рано или поздно вам потребуется

интегрировать в него ваши собственные возможности. Расширяемая архитектура диджита – это ваш пропуск к созданию всего необходимого. Модули расширения – это всего лишь средство инкапсуляции дополнительных функциональных возможностей, которые, несмотря на их востребованность, могут не входить в состав основных компонентов. Модули расширения могут, например, обрабатывать некоторые специальные комбинации клавиш или предоставлять какие-либо пункты меню, реализующие команды автоматизации части рабочего процесса.

Добавление модулей расширения в диджит `Editor` выполняется очень просто; возможно, вы этого еще не поняли, но все, что имеется на панели инструментов и что, на ваш взгляд, является встроенными функциональными возможностями, на самом деле представлено модулями расширения, имеющими говорящие сами за себя названия:.

| | |
|---|---|
| `undo` | `justifyLeft` |
| `redo` | `justifyRight` |
| `cut` | `delete` |
| `copy` | `selectAll` |
| `paste` | `removeFormat` |
| `insertOrderedList` | `bold` |
| `insertUnorderedList` | `italic` |
| `indent` | `underline` |
| `outdent` | `strikethrough` |
| `justifyCenter` | `subscript` |
| `justifyFull` | `superscript` |

Подключить модули расширения можно с помощью атрибута `plugins` или `extraPlugins`, в качестве значения которых следует указать список допустимых модулей расширения, не забыв предварительно загрузить их с помощью функции `dojo.require`. По умолчанию атрибут `plugins` содержит все элементы панели инструментов, которые можно увидеть в конфигурации по умолчанию, но если переопределить этот список и указать, например, такое значение атрибута: `plugins="['bold','italic']"`, то на панели инструментов будут присутствовать только те элементы, что перечислены в атрибуте `plugins`. С помощью атрибута `extraPlugins` можно подключить дополнительные модули расширения помимо тех, что уже перечислены в атрибуте `plugins`.

В состав инструментария входят несколько предварительно настроенных пакетов, имена которых часто используются в качестве значения атрибута `extraPlugins`. Они находятся в каталоге *dijit/_editor/plugins* и включают следующие модули расширения:

`AlwaysShowToolbar`

В случае необходимости раздвигает панель инструментов, чтобы на ней можно было отобразить элементы управления в несколько рядов, с целью обеспечить их постоянную видимость. (Если изменить размер окна так, что оно станет меньше ширины панели инструментов, по умолчанию появится горизонтальная полоса прокрутки и будет отображаться только та часть содержимого панели инструментов, которая обычно должна быть видима.) Чтобы подключить этот модуль расширения, необходимо указать значение `dijit._editor.plugins.AlwaysShowToolbar` в атрибуте `plugins` или `extraPlugins`.

`EnterKeyHandling`

Обеспечивает средство обработки нажатия клавиши Enter, позволяющее обойти различия между броузерами. Например, можно реализовать добавление серии тегов параграфов, окружающих новый текст, тега разрыва строки, набора тегов `DIV` или ничего не добавлять, запретив тем самым обработку клавиши Enter. Чтобы подключить этот модуль расширения, необходимо указать значение `dijit._editor.plugins.EnterKeyHandling` в атрибуте `plugins` или `extraPlugins`.

Расширяемая архитектура диджита `Editor` требует некоторой доработки, и до сих пор продолжаются обсуждения путей ее улучшения. Определенные успехи в этом направлении уже достигнуты, и вы можете следить за ходом работы по адресу *http://trac.dojotoolkit.org/ticket/5707*. Проще говоря, если вам потребуется создать собственный модуль расширения, то, пока архитектура модулей расширения не станет более стабильной, вам придется сначала исследовать исходный программный код в файле *Editor.js*

Кроме того, не забывайте о необходимости загружать используемые модули расширения с помощью функции `dojo.require`, поскольку в настоящее время поддержка модулей расширения не выполняет автоматических действий по определению и загрузке требуемых модулей расширения.

В настоящее время порядок обработки клавиши Enter определяется атрибутом `blockNodeForEnter` модуля расширения `EnterKeyHandling`, который по умолчанию имеет значение `'P'`. В настоящее время не существует лучшего способа изменить такой порядок вещей иначе, чем расширив прототип этого модуля расширения и переопределив его, как показано ниже:

```
dojo.addOnLoad(function() {
    dojo.extend(dijit._editor.plugins.EnterKeyHandling, {
        blockNodeForEnter : "div" // или "br" или "empty"
    });
});
```

FontChoice

Реализует кнопку с диалогом выбора имени шрифта, размера шрифта и формата блока. Аргументами для атрибута plugins или extraPlugins могут быть fontName, fontSize или formatBlock.

LinkDialog

Реализует кнопку с диалогом для ввода гиперссылки и отображаемого значения. Аргументом для атрибута plugins или extraPlugins может быть createLink.

TextColor

Реализует возможность указать цвет текста или цвет фона для фрагмента текста. Аргументами для атрибута plugins или extraPlugins могут быть foreColor или hiliteColor.

ToggleDir

Реализует средство привлечения HTML-атрибута dir (независимо от направления письма в остальной части страницы), благодаря чему содержимое в диджите Editor может вводиться в направлении слева направо или справа налево. Аргументом для атрибута plugins или extraPlugins может быть toggleDir.

Чтобы немного прояснить порядок подключения модулей расширения, далее в табл. 15.15 приводятся различные фрагменты разметки, в которых создается диджит Editor.

*Таблица 15.15. Различные подходы к созданию редактора*

| Разметка | Результат |
|---|---|
| `<div dojoType="dijit.Editor">` | Создает диджит Editor с панелью инструментов по умолчанию. |
| `<div dojoType="dijit.Editor" plugins="['bold', 'italic']">` | Создает диджит Editor с панелью инструментов, на которой имеются только кнопки выбора курсивного и жирного стиля шрифта. |
| `<div dojoType="dijit.Editor" extraPlugins="['hiliteColor']">` | Создает диджит Editor с панелью инструментов по умолчанию и с дополнительной кнопкой подсвечивания текста – предполагается, что была выполнена инструкция dojo.require("dijit._editor.plugins.TextColor"). |
| `<div dojoType="dijit.Editor" plugins="['bold', 'italic']" extraPlugins="['fontName']">` | Создает диджит Editor с панелью инструментов, на которой имеются кнопки выбора курсивного и жирного стиля шрифта, а также инструмент выбора шрифта (предполагается, что была выполнена инструкция dojo.require("dijit._editor.plugins.FontChoice")). Обратите внимание, что тот же самый результат можно получить, включив все три модуля расширения в атрибут plugin. |

# В заключение

По прочтении этой главы вы должны:

- Понимать, как вписываются прикладные диджиты общего назначения в общую архитектуру библиотеки Dijit, и правильно оценивать особую роль, которую они играют в создании ощущений у пользователя
- Уметь создавать прикладные диджиты как в разметке, так и в программном коде JavaScript
- Понимать основные отличия между диджитами `Tooltip` и `TooltipDialog`
- Уметь использовать диджит `Editor` с целью обеспечения возможности редактирования текста, имеющего сложное форматирование
- Уметь использовать диджиты `Toolbar` и `Menu` для предоставления пользователю средств управления приложением
- Уметь встраивать диджит `TooltipDialog` в `DropDownButton`
- Уметь использовать диджит ProgressBar как в детерминированном, так и в недетерминированном режимах для индикации хода выполнения длительной операции
- Уметь использовать диджит `Dialog` для вывода модальных предупреждений и встраивать в него произвольное содержимое для организации взаимодействия с пользователем
- Уметь использовать диджит `InlineEditBox`, чтобы снабдить пользователя возможностью *текст*, который выглядит как обычная разметка
- Уметь использовать диджит `Tree` для интерактивного отображения иерархической информации

В следующей главе будут рассматриваться инструменты сборки, тестирования и вопросы подготовки приложения к эксплуатации.

# 16

# Инструменты сборки, тестирования и вопросы подготовки к выпуску в эксплуатацию

После того как вы завершили разработку приложения на основе инструментария Dojo, наступает момент ввода в эксплуатацию. Библиотека Util содержит потрясающие инструменты сборки и платформу тестирования, которые помогут вам подготовить приложение к эксплуатации. В состав библиотеки Util входят те же самые инструменты сборки, которые используются для подготовки каждого официального выпуска Dojo, а также платформа модульного тестирования Dojo Objective Harness (DOH), которая позволяет провести некоторый автоматический контроль качества до того, как ваше приложение выйдет в мир.

## Сборка

Для установки приложения принципиально важными являются минимальность количества файлов JavaScript и минимальность числа синхронных запросов к серверу. Разница между загрузкой большого числа отдельных файлов ресурсов посредством синхронных запросов, выполняемых функцией `dojo.require`, и парой обращений к серверу существенно меняет ситуацию с точки зрения скорости загрузки страниц.

Инструменты сборки, входящие в состав Dojo, позволяют легко решить эту, казалось бы, такую сложную задачу. Проще говоря, инструменты сборки автоматизируют решение следующих задач:

- Объединение множества модулей в единый файл JavaScript, который называется слоем
- Внедрение строк шаблонов в файлы JavaScript, содержащие слои, чтобы устранить необходимость загружать шаблоны отдельно

- Применение ShrinkSafe – инструмента сжатия, основанного на Rhino, с целью *уменьшить* размеры слоев путем удаления пробелов, символов перевода строки, комментариев и сокращения имен переменных
- Копирование всех файлов «сборок» в отдельный каталог, откуда они могут копироваться и развертываться на веб-сервере

Единственное, что еще не сказано об инструментах сборки, – это то, что эти инструменты не включаются в каталог *util* официальной версии инструментария. Чтобы получить их, необходимо загрузить версию с исходными текстами (имя файла дистрибутива с исходными текстами включает в базовую часть дополнение *-src*) или просто забрать исходные тексты из репозитория Subversion. В главе 1 описывается, как получить исходные тексты из репозитория, но фактически все сводится к тому, что вы указываете клиентской программе адрес репозитория Dojo и ждете, пока не будет загружено содержимое главной ветви репозитория или определенной версии.

В любом случае вы обнаружите, что после этого каталог *util* содержит ряд дополнительных подкаталогов. Один из них, каталог *buildscripts*, содержит то, что мы ищем.

## Несколько слов о Rhino

Rhino – это интерпретатор JavaScript, полностью написанный на языке Java и названный в честь носорога, изображенного на обложке известной книги Дэвида Флэнагана (David Flanagan) «JavaScript: The Definitive Guide» (O'Reilly).[1] В конце 1990 годов проект Rhino, начатый компанией Netscape, задумывался как проект с закрытыми исходными текстами, но после передачи фонду Mozilla его исходные тексты были открыты. Принцип действия интерпретатора Rhino основан на преобразовании сценариев JavaScript в классы Java и разрабатывался Rhino для встраивания в приложения.

Еще один интерпретатор JavaScript – SpiderMonkey, написанный на языке C, – также разрабатывался компанией Netscape и позднее также был передан фонду Mozilla. Подобно Rhino интерпретатор SpiderMonkey является встраиваемой технологией. Он используется в различных популярных приложениях, включая Firefox и Yahoo! Widgets (прежнее название Konfabulator).

---

[1] Дэвид Флэнаган «JavaScript. Подробное руководство». – Пер. с англ. – СПб.: Символ-Плюс, 2008.

На сайте *http://svnbook.red-bean.com/* вы найдете неофициальное руководство к Subversion[1], доступное в нескольких форматах. Если сейчас потратите чуть-чуть времени и добавите этот ценный ресурс в свои закладки, это поможет вам не терять времени впоследствии.

Чтобы вы могли воспользоваться инструментами сборки, у вас должна быть установлена виртуальная машина Java 1.4.2 или более поздней версии, которую можно получить на сайте *http://java.sun.com* (это обусловлено тем, что инструмент ShrinkSafe основан на интерпретаторе Rhino, который, в свою очередь, написан на языке Java). Но не волнуйтесь: чтобы использовать ShrinkSafe, от вас не потребуется знание языка Java. Инструмент ShrinkSafe поставляется в виде единственного файла *jar* (выполняемый архив Java), который можно рассматривать как обычный исполняемый файл.

## Запуск сборки

Главным сценарием, с запуска которого начинается сборка, является *buildscripts/build.sh* (в Windows – *build.bat*). Он в действительности всего лишь запускает *jar*-файл Rhino, который выполняет всю основную работу, основываясь на выбранном профиле (подробнее об этом будет рассказываться чуть ниже). Однако, поскольку файл *jar* – это всего лишь обычный исполняемый файл, его легко можно задействовать в таких инструментах сборки, как *Make* или *ant*, на определенном этапе процесса сборки приложения. Такая возможность особенно удобна, когда серверные компоненты пишутся на языках компилирующего типа.

Запуск соответствующего сценария или *jar*-файла без параметров командной строки приводит к выводу на экран внушительного списка доступных параметров. В табл. 16.1 приводится список параметров, полученный непосредственно на устройстве стандартного вывода.

*Таблица 16.1. Параметры сценария сборки*

| Параметр | Описание |
|---|---|
| `xdScopeArgs` | Если используется параметр `loader=xdomain`, то значение этого параметра будет использовано в качестве аргумента функции, определяющей модуль в файле *.xd.js*. Это позволяет подключать к странице более одной версии одного и того же модуля. Подробности смотрите в описании `djConfig.scopeMap`. |
| `cssOptimize` | Определяет порядок оптимизации файлов CSS. Если указано значение `comments`, то из файла будут удалены комментарии и символы перевода строки. Если указано значение `comments.keepLines`, то из файла будут удалены комментарии, но символы перевода строки останутся. В любом случае будут встроены инструкции `@import`. |

[1] При корректных настройках языка в броузере автоматически открывается перевод руководства на русский язык. – *Прим. перев.*

*Таблица 16.1 (продолжение)*

| Параметр | Описание |
|---|---|
| releaseName | Название выпуска. Внутри каталога *releaseDir* будет создан подкаталог с этим именем. Значение по умолчанию: *dojo*. |
| localeList | Набор языковых настроек, которые будут использоваться для поддержки интернационализации. Значение по умолчанию: cs, de-de, en-gb, en-us, es-es, fr-fr, hu, it-it, ja-jp, ko-kr, pl, pt-br, ru, zh-tw, zh-cn. |
| releaseDir | Каталог верхнего уровня, куда помещаются собранные выпуски. Внутрь этого каталога помещаются каталоги *releaseName*. Значение по умолчанию: *../../release/*. |
| copyTests | Включает или отключает копирование файлов тестов. Значение по умолчанию: true. |
| symbol | Присваивает глобальные имена всем функциям, чтобы все анонимные функции обнаруживались отладчиками (особенно важно в случае IE, который не пытается выводить имена функций из контекста их определения). Допустимыми значениями являются long и short. Если указано значение short, то файл *symboltables.txt* будет сгенерирован для каждого каталога модуля в выпуске для отображения коротких имен в более описательные имена. |
| action | Действия, выполняемые в ходе сборки. Может быть списком значений, разделенных запятыми, например: action=clean, release. В качестве действий допускается указывать clean и release. Значение по умолчанию: help. |
| internStrings | Включает или выключает режим внедрения файлов шаблонов в виджеты. Значение по умолчанию: true. |
| scopeMap | Изменяет используемые по умолчанию имена областей видимости *dojo, dijit* и *dojox* на что-то другое. Удобно, когда вы хотите включить Dojo в библиотеку JS, но при этом хотите создать самостоятельную библиотеку, не использующую внешних ссылок на *dojo/dijit/dojox*. Значение определяется в форме строки, не содержащей пробелов, и напоминает значение djConfig.scopeMap (обратите внимание на обратные слеши ниже – они необходимы для экранирования символов, имеющих специальное значение в командной оболочке): `scopeMap: [[\"dojo\",\"mydojo\"],[\"dijit\",\"mydijit\"], [\"dojox\",\"mydojox\"]]` |
| optimize | Определяет порядок оптимизации файлов модулей. Если указано значение comments, из программного кода будут удалены все комментарии. Если указано значение shrinksafe, файлы будут сжаты компрессором Dojo и будут удалены символы перевода строки. Если указано значение shrinksafe.keepLines, файлы будут сжаты компрессором Dojo, но символы перевода строки будут оставлены на месте. Если указано значение packer, будет задействован упаковщик Packer Дина Эдвардса (Dean Edwards) (подробности по адресу *http://dean.edwards.name/packer/*). |

| Параметр | Описание |
|---|---|
| loader | Тип используемого загрузчика dojo. Допустимыми значениями являются default (значение по умолчанию) и xdomain. |
| log | Включает подробный режим протоколирования. Список допустимых целочисленных значений вы найдете в файле *util/buildtools/jslib/logger.js*. Значение по умолчанию: 0. |
| profileFile | Путь к файлу профиля. Этот параметр используется в том случае, если ваш профиль находится за пределами каталога с профилями. Не используйте параметр profile, если вы собираетесь определить значение profileFile. |
| xdDojoPath | Если применяется параметр loader=xdomain, то значение этого параметра будет использоваться в вызове dojo.registerModulePath() для *dojo*, *dijit* и *dojox*. Значением параметра *xdDojoPath* должно быть имя каталога, содержащего подкаталоги *dojo*, *dijit* и *dojox*, и не должно завершаться символом слеша. Например: *http://www.example.com/path/to/dojo*. |
| version | Версия сборки будет дополнена этой строкой. Значение по умолчанию: 0.0.0.dev. |
| profile | Имя профиля, используемого для сборки. Это должна быть первая часть имени файла профиля в каталоге *profiles/*. Например, чтобы задействовать профиль *base.profile.js*, следует указать profile=base (значение по умолчанию). |
| layerOptimize | Определяет порядок оптимизации файлов уровней. Если указано значение comments, из программного кода будут удалены все комментарии. Если указано значение shrinksafe, файлы будут сжаты компрессором Dojo и будут удалены символы перевода строки. Если указано значение shrinksafe.keepLines, файлы будут сжаты компрессором Dojo, но символы перевода строки будут оставлены на месте. Если указано значение packer, будет задействован упаковщик Packer Дина Эдвардса (Dean Edwards). Значение по умолчанию: shrinksafe. |
| xdDojoScopeName | Если применяется параметр loader=xdomain, то значение этого параметра будет использоваться вместо значения dojo (значение по умолчанию) в вызове функции dojo._xdResourceLoaded(), который завершает файлы *.xd.js*. Это позволит переместить dojo в другое пространство имен, но будет препятствовать загрузке версии XDomain с этим именем пространства имен. |
| cssImportIgnore | Вы можете использовать cssOptimize=comments, чтобы вынудить процедуру вставки инструкций @import игнорировать набор файлов. Этот параметр должен содержать список имен файлов CSS, разделенных запятыми, которые следует игнорировать. Имена файлов должны соответствовать строковым значениям, используемым в инструкциях @import. |

| Параметр | Описание |
|---|---|
| buildLayers | Список имен слоев для сборки, разделенных запятыми. Этот параметр позволяет ограничиться сборкой только указанных слоев. Этот параметр может использоваться для ускорения разработки, когда процедура сборки сопровождается циклами тестирования слоев. Если у вас возникнут проблемы с этим параметром, попробуйте убрать его и выполнить полную сборку с параметром action=clean,release. Этот параметр предполагает, что полная сборка выпуска производилась хотя бы один раз. |
| scopeDjConfig | Встраивает определение массива djConfig в собранный файл *dojo.js*, что может быть удобно, когда при создании собственной сборки необходимо иметь объект djConfig, локальный по отношению к данной версии, который не оказывал бы влияния на глобальный объект djConfig, объявляемый в странице. Значение этого параметра должно быть строкой, которая будет похожа на литерал объекта JavaScript после помещения ее в файл с исходными текстами. Кроме того, это может быть полезно в ситуациях, когда необходимо применять Dojo в составе самостоятельной библиотеки JavaScript, не использующей внешних ссылок на *dojo*, *dijit* и *dojox*. Например:<br>`scopeDjConfig={isDebug:true,scopeMap:[[\"dojo\",\"mydojo\"], [\"dijit\",\"mydijit\"], [\"dojox\",\"mydojox\"]]}`<br>Обратите внимание на обратные слеши – они необходимы для экранирования символов, имеющих специальное значение, когда значение вводится в командной строке. |

Несмотря на такое многообразие параметров, процедура сборки на практике очень проста и требует небольшого числа из перечисленных параметров. Но сначала нам потребуется определить профиль.

## Профили сборки

*Профиль* – это файл с параметрами настройки для сборки, имя которого указывается в параметре profile или profileFile. Основная функция профиля заключается в том, чтобы точно определить, какие ресурсы Dojo должны объединяться в самостоятельный файл JavaScript, известный также как *слой*. Основное правило при этом – каждой странице вашего приложения должен соответствовать свой собственный слой. Преимущество слоя состоит в том, что это самый обычный файл JavaScript, который можно подключать непосредственно в заголовке страницы, и загружать все, чем вы этот файл наполнили, посредством единственного синхронного запроса к серверу – по крайней мере, идея в этом. В соответствии с соглашениями библиотека Base всегда так интенсивно используется, что ее обычно оставляют в ее собственном файле *dojo.js*; поэтому приложению обычно приходится выполнять два

синхронных запроса: один, чтобы загрузить библиотеку Base, и второй, чтобы загрузить созданный вами слой.

## Настройка профиля

Предположим, что приложение состоит из трех различных страниц. В этом случае вам может потребоваться три файла слоев и одна копия библиотеки Base.

Если вы по какой-то причине хотите включить свои собственные модули в состав файла *dojo.js*, который обычно содержит только библиотеку Base, вы можете назвать слой *dojo.js*. Однако часто предпочтительнее держать библиотеку Base в виде отдельного файла, потому что она может использоваться всеми страницами приложения и при этом она кэшируется броузером.

Физически профиль – это простой файл, содержащий объект в формате JSON. В примере 16.1 показано содержимое профиля, объединяющего несколько диджитов форм, которые явно подключаются к странице с помощью вызова функции `dojo.required`. Все неявные зависимости отслеживаются автоматически. Как всегда при использовании функции `dojo.require`, вы просто определяете, что будет использоваться, а неявные зависимости будут удовлетворяться автоматически.

*Пример 16.1. Простой профиль сборки*

```
dependencies ={
    layers: [
        {
            name: "form.js",
            dependencies: [
                "dijit.form.Button",
                "dijit.form.Form",
                "dijit.form.ValidationTextBox"
            ]
        }
    ],
    prefixes: [
         [ "dijit", "../dijit" ]
    ]
};
```

Предположим, что этот профиль находится в файле *util/buildscripts/profiles/form.profile.js* и вы работаете в командной оболочке Bash; тогда следующая команда, выполненная в каталоге *util/buildscripts*, запустит процесс сборки. Обратите внимание, что параметр `profile` подразумевает имя файла в виде *<имя профиля>.profile.js*, и потому ему передается только часть *<имя профиля>*:

```
bash build.sh profile=form action=release
```

Если вы не хотите сохранять файл *util/buildscripts/profiles/form.profile.js*, то вместо параметра `profile` можно использовать параметр `profileFile`.

После запуска команды на экране должна появиться информация, свидетельствующая о том, что сборка началась и что производится встраивание строк из файлов шаблонов в файлы JavaScript. Результатом сборки будет каталог с собранным выпуском, содержащий подкаталоги *dojo*, *dijit* и *util*. В каталоге *dojo* вы найдете то, что и ожидалось найти; особого внимания заслуживают четыре файла:

- Сжатая и несжатая версии библиотеки Base – файлы *dojo.js* и *dojo.js.uncompressed.js*
- Сжатая и несжатая версии слоя *form* – файлы *form.js* и *form.js.uncompressed.js* (загляните внутрь каждого из них, чтобы убедиться в этом).

Но как быть, если вам потребуются ресурсы, не включенные в состав файла слоя? Нет проблем – если ресурсы не указаны в профиле, они будут запрошены у сервера соответствующими инструкциями `dojo.require`. Предположим, что вы взяли весь каталог с выпуском и поместили его в какой-либо каталог на своем сервере; тогда инструкции `dojo.require`, загружающие ресурсы, не включенные в состав слоя, будут работать как обычно, но при этом будет затрачиваться некоторое время на получение требуемых ресурсов с сервера.

Попытки загрузить библиотеку Base и ресурсы, включенные в слой, с помощью инструкций `dojo.require`, не будут приводить к выполнению запросов к серверу, потому что они уже доступны локально. Однако для загрузки ресурсов, не включенных в слой, будут выполняться синхронные HTTP-запросы (рис. 16.1).

Хотя обычно лучше оказывается включить в слой все ресурсы, используемые сборкой, но в некоторых ситуациях может оказаться предпочтительнее прибегнуть к механизму отложенной загрузки. Обычно в подобных ситуациях приходится выбирать между «достаточно небольшим» объемом данных, загружаемых первоначально, и стоимостью синхронной загрузки с помощью `dojo.require`, производимой позднее.

Если вы допустили опечатку или по ошибке определили зависимость, которой не существует, инструмент ShrinkSafe все равно сможет завершить сборку, несмотря на то, что некоторые зависимости не могут быть удовлетворены. Например, если по ошибке вместо `dijit.form.Button` вы укажете `dijit.Button`, сборка скорее всего завершится успешно и вы можете даже не заметить, что ресурс `dijit.form.Button` не был включен в состав слоя, потому что инструкция `dojo.require("dijit.form.Button")` извлечет его с сервера и в вашем приложении не будет наблюдаться никаких отклонений от нормы.

Всегда полезно дважды проверить свою сборку, заглядывая на вкладку Net (сеть) в отладчике Firebug, чтобы убедиться, что все необходимое действительно включено в состав сборки.

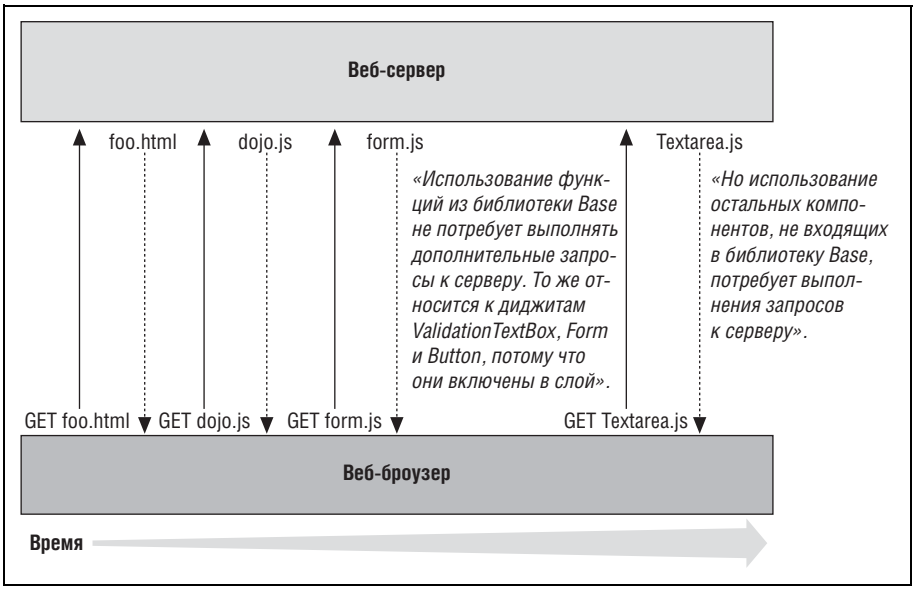



***Рис. 16.1.*** *Концептуальная схема запросов, демонстрирующая, как выполняется загрузка различных файлов JavaScript*

## Настройка профиля (более грамотный вариант)

Немного более грамотный вариант настройки профиля по сравнению с тем, что обсуждался выше, заключается в создании собственного модуля, который не делает ничего, кроме подключения всех ресурсов, ранее помещавшихся в слой с помощью файла профиля. После этого модуль просто включается в файл профиля как единственная зависимость слоя.

Для начала в примере 16.2 показано, как может выглядеть такой модуль. Предположим, что модуль имеет имя `dtdg.page1` и оформлен в виде файла *dtdg/page1.js*.

*Пример 16.2. Модуль, используемый при более грамотном способе настройки профиля*

```
dojo.provide("dtdg.page1");

dojo.require("dijit.form.Form");

dojo.require("dijit.form.Button");

dojo.require("dijit.form.ValidationTextBox");
```

Теперь в профиле достаточно будет указать только ваш модуль, так как все остальные зависимости, указанные в нем, будут отслежены и удовлетворены автоматически. В примере 16.3 приводится измененный профиль, где предполагается, что подкаталог с модулем находится в одном каталоге с подкаталогом *util*.

*Пример 16.3. Измененный профиль сборки*

```
dependencies ={
    layers: [
        {
            name: "form.js",
            dependencies: [
                "custom.page1"
            ]
        }
    ],
    prefixes: [
         [ "custom", "../custom" ]
    ]
};
```

Наконец, ваша страница могла бы содержать следующий тег SCRIPT, который загружает модуль вместе с библиотекой Base:

```
<script type="text/javascript"
djConfig="baseUrl: './',modulePaths: {custom:'path/to/custom/page1.js'},
    require: ['custom.page1']"
src="scripts/dojo.js"></script>
```

## Стандартный профиль сборки

Обратите внимание, что каталог *util/buildscripts/profiles* содержит несколько примеров профилей сборки, среди которых имеется файл *standard.profile.js*, содержащий слои стандартной сборки Dojo. С помощью стандартного профиля выполняется сборка библиотеки Base, основного слоя Dijit, содержащего наиболее часто используемые механизмы, применяемые практически во всех приложениях, где задействуются диджиты, а также сборка ряда других полезных слоев. Примечательно, что любой профиль в файле *standard.profile.js* должен быть доступен через AOL CDN. Например, чтобы получить базовый профиль Dijit, можно просто выполнить следующую инструкцию:

```
dojo.require("dijit.dijit");
```

Не забывайте при этом, что первой в теге SCRIPT должна подключаться библиотека Base (*dojo.xd.js*), поэтому все остальные теги SCRIPT, выполняющие подключение любых дополнительных слоев, должны стоять после тега, подключающего библиотеку Base.

## Средства оптимизации в ShrinkSafe и другие часто используемые параметры

При подготовке к эксплуатации практически любого приложения вам потребуется применить инструмент ShrinkSafe для уменьшения объема, занимаемого программным кодом. Несмотря на то что предыдущие примеры создают оптимизированные сборки, т.е. наряду с внедрением строк шаблонов они минимизируют размеры файлов *dojo.js* и *form.js*, тем не менее ShrinkSafe позволит минимизировать каждый файл в сборке.

Вспомните, что объем, передаваемый «по кабелю», начинает обретать смысл, когда речь заходит о производительности с точки зрения пропускной способности. Файлы могут передаваться в сеть с тем же размером, какой они имеют на стороне сервера, однако большинство веб-серверов способны выполнять сжатие с помощью утилиты *gzip*, если броузер окажется в состоянии обрабатывать такие файлы. Инструмент ShrinkSafe минимизирует файлы за счет удаления таких элементов, как лишние пробелы, комментарии и т. д.; кроме того, размеры файлов могут быть уменьшены еще больше за счет устранения повторяющихся общедоступных символов, таких как `dojo`, `dijit` и ваших собственных лексем.

*Минимизация* файла – это уменьшение его размера за счет удаления таких элементов, как запятые, пробелы, символы перевода строки и других. *Сжатие* – это алгоритмическая манипуляция, которая позволяет уменьшить размер файла за счет поиска повторяющихся лексем и создания эквивалентного файла, в котором повторяющиеся лексемы замещаются более короткими аналогами. Более подробно о сжатии по алгоритму *gzip* вы найдете по адресу *http://en.wikipedia.org/wiki/Gzip*.[1]

Еще одна примечательная особенность ShrinkSafe заключается в том, что этот инструмент никогда не изменяет общедоступный прикладной программный интерфейс в противовес некоторым инструментам JavaScript, которые пытаются шифровать программный JavaScript, применяя регулярные выражения или другую замысловатую логику для «защиты» сценариев. Вообще говоря, попытки защитить программный код JavaScript лишены практически какого-либо смысла. Поскольку JavaScript – это язык интерпретирующего типа, у пользователя вашего приложения практически наверняка будет возможность получить доступ к вашему исходному программному коду, а чтобы с помощью отладчика превратить защищенный сценарий в более читабельную форму, требуется приложить не так много усилий.

---

[1] На русском языке см. *http://ru.wikipedia.org/wiki/Gzip*, хотя эта страница содержит меньше информации, чем англоязычная. – *Прим. перев.*

Инструмент ShrinkSafe создавался не только для применения в Dojo – чтобы использовать преимущества сжатия, вы можете применять его к любым файлам JavaScript, воспользовавшись демонстрационной версией по адресу *http://shrinksafe.dojotoolkit.org/*. Пользователи OS X могут получить версию инструмента по адресу *http://dojotoolkit.org/downloads*, а пользователи других платформ могут получить отдельный *jar*-файл Rhino по адресу: *http://svn.dojotoolkit.org/dojo/trunk/buildscripts/lib/custom_rhino.jar*.

Проще говоря, ShrinkSafe уменьшает размеры файлов без изменения общедоступных имен. Фактически, если вы заглянете в файл *form.js*, созданный в предыдущих примерах, то сможете убедиться, что ShrinkSafe удалил все комментарии и/или лишние пробельные символы, включая символы перевода строки, и заменил все частные имена более короткими. Обратите внимание, что замена всех имен более короткими, не несущими смысла версиями, опрвдана только для кодирования и при этом совсем не полезна при отладке.

Давайте изменим существующий профиль:

- Добавим минимизацию всех файлов в выпуске с помощью параметра `optimize="shrinksafe"`
- Определим текст примечания, которое должно находиться в начале каждого минимизированного файла JavaScript, входящего в состав дополнительного (вымышленного) модуля `foo`, сохранив этот текст в файле *CUSTOM_FILE_NOTICE.txt*
- Определим текст примечания, которое должно находиться в начале окончательного файла *form.js*, воспользовавшись тем же самым файлом *CUSTOM_FILE_NOTICE.txt*
- Определим собственное имя каталога для выпуска с помощью параметра `releaseName="form"`
- Определим собственный номер версии сборки с помощью параметра `version="0.1.0."`

Ниже дается содержимое измененного файла профиля *form.profile.js*, приводившегося выше в примере 16.1. Обратите внимание, что примечания должны оформляться в стиле комментариев JavaScript. Относительный путь к файлам примечаний определяется от каталога *util/buildscripts*, но можно указать и абсолютный путь:

```
dependencies ={
    layers: [
        {
            copyrightFile : "CUSTOM_LAYER_NOTICE.txt",
            name: "form.js",
            dependencies: [
                "dijit.form.Button",
                "dijit.form.Form",
```

```
                "dijit.form.ValidationTextBox"
            ]
        }
    ],
    prefixes: [
        [ "dijit", "../dijit" ],
        [ "foo", "../foo", "CUSTOM_FILE_NOTICE.txt" ]
    ]
};
```

Команда запуска сборки в этом случае получилась достаточно простой; она помещает результаты в каталог *release/form*, расположенный по соседству с каталогом *dojo*, содержащим исходные тексты:

```
bash build.sh profile=form action=release optimize=shrinksafe
releaseName=form
version=0.1.0
```

Чтобы задействовать получившуюся версию библиотеки в своем приложении, достаточно просто указать пути к сжатым файлам *dojo.js* и *form.js* в тегах `SCRIPT` в заголовке страницы, как показано ниже.

## Создание сборок для Rhino

Хотя формальным адресатом сборок Dojo являются броузеры, тем не менее систему сборки можно использовать также для создания собственных сборок Dojo, которые будут использоваться непосредственно в Rhino. Данная потребность может появиться, если вы собираетесь вызывать программный код JavaScript из классов Java или если вы применяете такой инструмент, как Helma (*http://dev.helma.org*), который дает возможность использовать сценарии JavaScript на стороне сервера.

Для этого в профиле сборки нужно только указать `hostenvType="rhino"`. Если вам потребуется воспользоваться платформой модульного тестирования DOH, чтобы протестировать сборку для Rhino, вы должны также включить дополнительный префикс *shrinksafe*. Ниже приводится пример профиля, используемого с целью создания сборки для Rhino:

```
hostenvType = "rhino";

dependencies = {
    layers : [];
    prefixes : [
        ["dojox", "../dojox"],
        ["shrinksafe", "../util/shrinksafe"]
    ]
};
```

При этом файл *dojo.js* должен подключаться первым, потому что *form.js* зависит от него:

```
<html>
    <head><title>Fun With Forms!</title>

        <!-- подключить таблицы стилей и прочее -->
        <script type="text/javascript"
                path="relative/path/to/form/dojo.js"></script>
        <script type="text/javascript"
                path="relative/path/to/form/form.js"></script>
    </head>
    <!-- остальная часть страницы -->
```

Вот и все. В этом случае приложению потребуется выполнить всего два синхронных запроса для загрузки в страницу программного кода JavaScript (в который уже включены все необходимые шаблоны). Доступ к остальным ресурсам, подключаемым к сборке через список `prefixes`, будет обеспечиваться стандартными инструкциями `dojo.require`.

Если вы твердо уверены, что никакие другие дополнительные ресурсы JavaScript, помимо *dojo.js* и вашего файла слоя, вам не потребуются, вы можете подключить только отдельные ресурсы из структуры каталогов с выпуском. Однако при этом вам придется проделать дополнительную работу по выявлению зависимостей со встроенными темами CSS, такими как *tundra*, потому что некоторые таблицы стилей могут использовать относительные пути и относительные URL в инструкциях `@import`.

Исследование содержимого вкладки Net в отладчике Firebug может оказать существенную помощь в определении зависимостей, которые следует поместить в каталог с выпуском, но имейте в виду, что Firebug может не отображать ошибки с кодом 404 (страница не найдена), порождаемые инструкциями `@import`, которые используются в таблицах стилей.

# Платформа тестирования Dojo (DOH)

Автоматизированное тестирование веб-приложений находит все более широкое применение из-за увеличивающегося объема программного кода и сложности большинства современных активных Интернет-приложений. Платформа тестирования DOH (Dojo Objective Harness) использует инструментарий Dojo для своих внутренних нужд, но она может применяться не только для тестирования программного кода Dojo – как и ShrinkSafe, она может использоваться для создания модульных тестов, выполняющих проверку любого программного кода JavaScript, хотя при этом никакие функции манипуляции с деревом DOM или функции, характерные для определенных типов броузеров, доступны не будут.

Платформа тестирования DOH предоставляет три простые конструкции проверки, которые имеют большое значение для автоматизации процесса тестирования. Каждая из этих конструкций доступна через глобальный объект `doh`, предоставляемый платформой:

- `doh.assertEqual(expected, actual)`
- `doh.assertTrue(condition)`
- `doh.assertFalse(condition)`

Прежде чем углубляться в изучение платформы DOH, чтобы получить некоторое представление о ее возможностях, рассмотрим пример простейшего теста, который можно запустить из командной строки с помощью Rhino. Тест, представленный ниже, демонстрирует, что при использовании платформы DOH можно запускать отдельные тесты с помощью обычных объектов Function, а также через специальные тестовые контексты. *Тестовый контекст* (*test fixture*) – это не более чем способ дополнить тест операциями инициализации и освобождения ресурсов.

## Средства тестирования Rhino без использования Dojo

А теперь перед вами такое средство тестирования. Обратите внимание, что в тесте не используются какие-либо средства из инструментария Dojo – в нем используется только объект `doh`. В частности, в этом примере используется функция `doh.register`, первый аргумент которой определяет имя модуля (файл JavaScript, находящийся в одном каталоге с подкаталогом *util*), а второй содержит список функций тестирования или тестовых контекстов:

```
doh.register("testMe", [

    // тестовый контекст, который выполняется успешно
    {
        name : "fooTest",
        setUp : function() {},
        runTest : function(t) { t.assertTrue(1); },
        tearDown : function() {}
    },

    // тестовый контекст, который терпит неудачу
    {
        name : "barTest",
        setUp : function() { this.bar="bar"},
        runTest : function(t) { t.assertEqual(this.bar, "b"+"a"+"rr"); },
        tearDown : function() {delete this.bar;}
    },

    // отдельная функция, которая выполняется успешно
    function baz() {doh.assertFalse(0)}
]);
```

Если считать, что этот тест сохранен в файле с именем *testMe.js* и находится по соседству с каталогом *util*, то этот тест можно было бы выполнить, запустив из каталога *util/doh* приведенную ниже команду (Обратите внимание: хотя в данном примере используется наш собственный *jar*-файл Rhino, созданный с применением инструментов сборки, тем не менее можно было бы использовать любой достаточно новый *jar*-файл Rhino.):

```
java -jar ../shrinksafe/custom_rhino.jar runner.js dojoUrl="../../dojo/
dojo.js" testModule=testMe
```

Эта команда просто сообщает исполняемому *jar*-файлу Rhino, что с помощью файла JavaScript *runner.js* (компонент DOH) необходимо запустить модуль `testMe`, используя указанную версию библиотеки Base. Хотя инструментарий Dojo и не привлекается к тестированию, тем не менее платформа DOH использует библиотеку Base для внутренних нужд, поэтому необходимо указать путь к ней.

Теперь, когда вы увидели платформу DOH в действии, можно познакомиться с содержимым табл. 16.2, где приводится перечень дополнительных функций, доступных в объекте `doh`.

*Таблица 16.2. Функции модуля doh*

| Функция | Комментарий |
|---|---|
| `registerTest(/*String*/group, /* Function \|\| Object */ test)` | Добавляет тест или объект контекста в указанную группу тестов. |
| `registerTests(/*String*/group, /*Array*/ tests)` | Автоматически регистрирует группу тестов, включенных в массив `tests`. |
| `registerTestNs(/*String*/group, /*Object*/ns)` | Добавляет функции, включенные в объект `ns`, в коллекцию, которая должна представлять группу тестов. Функции, чьи имена начинаются с символа подчеркивания, не включаются в группу, потому что начальный символ подчеркивания обычно свидетельствует о том, что данное имя является частным. |
| `register(/* ...*/)` | Применяет соответствующую функцию регистрации, анализируя аргументы и определяя, какой функции регистрации они соответствуют. |
| `assertEqual(expected, actual)` | Используется для проверки равенства двух значений. |
| `assertTrue(/*Boolean*/condition)` | Используется для проверки истинности значения. |
| `assertFalse(/*Boolean*/ condition)` | Используется для проверки неистинности значения. |

| Функция | Комментарий |
|---|---|
| `is(expected, actual)` | Короткая форма имени `assertEqual`. |
| `t(/*Boolean*/condition)` | Короткая форма имени `assertTrue`. |
| `f(/*Boolean*/condition)` | Короткая форма имени `assertFalse`. |
| `registerGroup(/*String*/ group, /*Array\|\|Function\|\|Object*/tests, /*Function*/ setUp, /*Function*/tearDown)` | Добавляет всю группу тестов, представленную аргументом `tests`, в группу `group`. Использует специальные функции `setUp` и `tearDown`, если они указаны. |
| `run()` | Используется для запуска тестов программным способом. |
| `runGroup(/*String*/groupName)` | Используется для запуска группы тестов программным способом. |
| `pause` | Может использоваться для программной приостановки запущенного теста. Возобновить работу приостановленного теста можно с помощью функции `run()`. |
| `togglePaused` | Может использоваться для выполнения последовательности приостановок и возобновлений тестов программным способом. |

Кроме того, следует заметить, что файл *runner.js* может принимать дополнительные параметры, перечисленные в табл. 16.3.

*Таблица 16.3. Параметры для сценария runner.js*

| Функция | Комментарий |
|---|---|
| `dojoUrl` | Путь к файлу *dojo.js*. |
| `testUrl` | Путь к тестируемому файлу. |
| `testModule` | Список имен модулей тестов, разделенных запятыми, которые должны быть выполнены, например: `foo.bar,foo.baz`. |

## Средства тестирования Rhino с использованием Dojo

Несмотря на то что платформа DOH может использоваться без инструментария Dojo, тем не менее есть вероятность, что вам потребуется использовать Dojo вместе с Rhino. Библиотека Core включает несколько замечательных примеров, которые можно запускать при помощи *runner.js* без дополнительных аргументов. Значения по умолчанию определяют, что тесты находятся в каталоге *dojo/tests* и используется версия библиотеки Base в файле *dojo/dojo.js*.

Если заглянуть в любой файл теста, находящийся в библиотеке Core, можно увидеть, что они устроены вполне очевидным образом. Каждый файл начинается с инструкции `dojo.provide`, которая определяет имя модуля тестов, затем выполняется подключение тестируемых ресурсов, и далее следует серия функций `register`, создающих контексты для тестов.

Предположим, что имеется некоторый модуль `foo.bar`, находящийся в файле */tmp/foo/bar.js*, и что имеется файл с тестами */tmp/testBar.js*. Содержимое обоих файлов JavaScript приводится ниже.

Сначала файл *testBar.js*:

```
/* dojo.provide объявляет модуль с тестами, как любой другой модуль */
dojo.provide("testBar");

/* Вам может потребоваться добавить пути к модулям, если они находятся
   за пределами корневого каталога dojo */
dojo.registerModulePath("foo.bar", "/tmp/foo/bar");

/* с помощью dojo.require подключается все, что будет необходимо */
dojo.require("foo.bar");

/* регистрация модуля */
doh.register("testBar", [

    function() { doh.t(alwaysReturnsTrue()); },
    function() { doh.f(alwaysReturnsFalse()); },
    function() { doh.is(alwaysReturnsOdd()%2, 1); },
    function() { doh.is(alwaysReturnsOdd()%2, 1); },
    function() { doh.is(alwaysReturnsOdd()%2, 1); },
    {
        name : "BazFixture",
        setUp : function() {this.baz = new Baz;},
        runTest : function() {doh.is(this.baz.talk(), "hello");},
        tearDown : function() {delete this.baz;}
    }
]);
```

А теперь содержимое файла *foo/bar.js* модуля `foo.bar`:

```
/* Коллекция малополезных функций */
dojo.provide("foo.bar");

function alwaysReturnsTrue() {
    return true;
}

function alwaysReturnsFalse() {
    return false;
}

function alwaysReturnsOdd() {
    return Math.floor(Math.random()*10)*2-1;
}
```

```
// Взгляните: здесь есть даже определение "класса"
dojo.declare("Baz", null, {
    talk : function() {
        return "hello";
    }
});
```

Выполнить тестирование можно с помощью следующей команды, запускаемой из каталога *util/buildscripts*:

```
java -jar ../shrinksafe/custom_rhino.jar runner.js dojoUrl=../../dojo/dojo.js
testUrl=/tmp/testBar.js
```

Особое внимание обратите на то, что в файле с тестами перед подключением модуля `foo.bar` явно регистрируется путь к нему. Этот дополнительный шаг необходим, когда ресурсы располагаются за пределами корневого каталога dojo, чтобы имелась возможность отыскать ваш модуль.

Если все идет, как планировалось, вы увидите сообщение, указывающее, что все тесты были успешно или неуспешно пройдены. Регистрация группы тестов, совместно использующих одни и те же операции по инициализации и освобождению ресурсов, выполняется точно так же, за исключением того, что вместо функции `doh.register` (или ее более специализированной версии) следует использовать функцию `doh.registerGroup`.

Если вам требуется более полный контроль над выполнением тестов, вы можете приостанавливать и возобновлять тестирование программно, внеся следующие изменения в файл *testBar.js*:

```
/* загрузить dojo.js и runner.js */
load("/usr/local/dojo/dojo.js");
load("/usr/local/dojo/util/doh/runner.js");

/* dojo.provide объявляет модуль с тестами, как любой другой модуль */
dojo.provide("testBar");

/* Вам может потребоваться добавить пути к модулям, если они находятся
   за пределами корневого каталога dojo */
dojo.registerModulePath("foo.bar", "/tmp/foo/bar");

/* с помощью dojo.require подключается все, что будет необходимо */
dojo.require("foo.bar");

/* регистрация модуля */
doh.register("testBar", [

    function() { doh.t(alwaysReturnsTrue()); },
    function() { doh.f(alwaysReturnsFalse()); },
    function() { doh.is(alwaysReturnsOdd()%2, 1); },
    function() { doh.is(alwaysReturnsOdd()%2, 1); },
    function() { doh.is(alwaysReturnsOdd()%2, 1); },
    {
        name : "BazFixture",
```

```
        setUp : function() {this.baz = new Baz;},
        runTest : function() {doh.is(this.baz.talk(), "hello");},
        tearDown : function() {delete this.baz;}
    }
]);

doh.run();

/* можно приостанавливать и перезапускать по мере необходимости... */
```

Хотя мы не использовали тот факт, что `testBar` является модулем, который объявляется инструкцией `dojo.provide`, тем не менее вы легко можете объединить коллекции тестов с помощью инструкций `dojo.require`, как и в любом другом модуле, который объявляет себя.

Хотя с помощью Rhino можно выполнять асинхронные тесты, тем не менее в следующем разделе описываются приемы асинхронного тестирования в броузере, потому что они особенно полезны для проведения тестирования сетевого ввода-вывода и таких событий, как анимация.

# Тестирование в броузере

Проведение тестирования из Rhino чрезвычайно полезно, однако кроме этого платформа DOH обеспечивает возможность, позволяющую автоматизировать проведение тестирования в окне броузера. Для этого тест оформляется как обычная страница HTML, после чего она загружается в механизм запуска тестов DOH посредством строки запроса в URL механизма запуска. Механизм запуска анализирует строку запроса, извлекает параметры настройки, такие как `testUrl`, и использует их для загрузки тестовой страницы во фрейм.

Конечно, вам ничто не мешает запустить в броузере свой тест без привлечения платформы DOH, но в этом случае вы лишаетесь удобного визуального представления и необязательных восклицаний Гомера Симпсона, а результаты тестирования будут выводиться как в обычной консоли.

## Пример теста, выполняемого в броузере

Ниже приводится пример теста, оформленного в виде обычной страницы HTML. Обратите внимание, что в примере используется локальная версия инструментария Dojo, потому что начиная с версии 1.1 платформа DOH не распространяется в составе версии AOL CDN:

```
<html>
    <head><title>Fun with DOH!</title>

        <script
            type="text/javascript"
            src="local/path/to/dojo/dojo.js">
        </script>
```

```
        <script type="text/javascript">
            dojo.require("doh.runner");
            dojo.addOnLoad(function( ) {
                doh.register("fooTest", [
                    function foo( ) {
                        var bar = [];
                        bar.push(1);
                        bar.push(2);
                        bar.push(3);

                        doh.is(bar.indexOf(1), 0); //операция не переносима!
                    }
                ]);

                doh.run( );
            });
        </script>
    </head>
    <body></body>
</html>
```

## Пример асинхронного теста, выполняемого в броузере

Особую ценность практически любого набора тестов для веб-приложений представляет возможность проверить наступление асинхронных событий, таких как завершение воспроизведения анимационных эффектов, получение ответа от сервера и т. д. В примере 16.4 демонстрируется, как с помощью платформы DOH можно создать асинхронный тест. Основу теста составляет применение объекта `doh.Deferred` (практически идентичен объекту `dojo.Deferred`, но имеет некоторые особенности), реализованного внутри платформы DOH и не имеющего внешних зависимостей. Если вам потребуется быстро вспомнить, что такое объекты `Deferred`, вернитесь к главе 4, где они рассматриваются достаточно подробно.

Прежде чем перейти к рассмотрению фактического программного кода, ознакомьтесь с типичным шаблоном проведения асинхронного тестирования:

- Создать объект `doh.Deferred`, который будет использоваться для проверки результатов асинхронной функции (которая возвращает объект `dojo.Defekred`)
- Вызвать асинхронную функцию, возвращающую объект `dojo.Deferred`, и сохранить ссылку на него в переменной
- Добавить функции обратного вызова и обработки ошибок в объект `dojo.Deferred`, которые просто передают результаты асинхронной функции собственным функциям обратного вызова и обработки ошибок объекта `doh.Deferred`

*Пример 16.4. Шаблон асинхронного теста*

```
doh.register("foo", [

    function( ) {
        var dohDfd = new doh.Deferred();
        var expectedResult = "baz";

        var dojoDfd = asynchronousBarFunction();
        dojoDfd.addBoth(function(response, io) {

            //используется ссылка на объект dohDfd...
            if (response == expectedResult) {
                dohDfd.callback(true);
            }
            else {
                dohDfd.errback(new Error( /* ... */));
            }
        });

        //...и обратно возвращается объект dohDfd
        return dohDfd;
    }
]);
```

В зависимости от накладываемых ограничений можно было бы явно указать значение параметра `timeout`, чтобы убедиться, что асинхронные операции, предусматривающие предельное время ожидания, выполняются в соответствии с вашими критериями тестирования. В любом случае, главный вывод из этого примера заключается в том, что выполнить асинхронное тестирование совсем несложно – абстракция объекта `Deferred` упрощает решение этой задачи и освобождает вас для решения других проблем.

# Вопросы производительности

В этом разделе рассматриваются некоторые из наиболее удобных путей, которыми вы можете следовать при разработке внешнего интерфейса. Значительный объем справочной информации о способах повышения производительности вы найдете в книге Стива Содерса (Steve Souders) «High Performance Web Sites: Essential Knowledge for Front-End Engineers» (O'Reilly). Эта книга читается очень легко и полностью соответствует своему названию. Большая ее часть доступна по адресу: *http://developer.yahoo.com/performance/rules.html*.

Несмотря на то что качественного программного кода JavaScript вполне достаточно, чтобы получить высокопроизводительное веб-приложение, тем не менее существует еще несколько моментов, о которых желательно знать, когда подходит время ввода приложения в эксплуатацию. Вопрос оптимизации производительности веб-приложений мог бы стать темой отдельной книги, тем не менее ниже приводятся неко-

торые из наиболее удобных подручных средств, которыми вы можете воспользоваться:

*Инструменты сборки Dojo*

Инструменты сборки Dojo позволяют решить ряд основных задач, а их использование не требует приложения значительных усилий. В процессе сборки выполняется минимизация исходных файлов, за счет чего уменьшается общий объем данных, а благодаря объединению нескольких файлов JavaScript в слои и внедрению строк шаблонов везде, где это возможно, значительно уменьшаются задержки HTTP.

*Отложенная загрузка*

В этой главе много говорилось о преимуществах применения инструментов сборки для создания минимального числа файлов слоев, используемых приложением, однако, будут встречаться ситуации, когда предпочтение следует отдать отложенной загрузке. Например, если вы определили, что пользователи очень редко используют некоторую особенность, требующую включения в слой значительного объема программного кода JavaScript, то вы можете не включать этот программный код в слой, а подключать его с помощью `dojo.require`.

Еще одно замечание по поводу отложенной загрузки заключается в грамотном использовании виджетов компоновки для загрузки содержимого в ходе выполнения. Например, первоначально можно загружать содержимое только для видимой вкладки в виджете `TabContainer`, а содержимое для других вкладок загружать по требованию; подождать достаточно продолжительное время, чтобы дать возможность загрузиться остальной части страницы, и только потом загружать содержимое остальных вкладок. Типичным средством отложенной загрузки содержимого является диджит `ContentPane`.

*Настройка веб-сервера*

Настройте веб-сервер так, чтобы он посылал заголовки `Expires` с датой в далеком будущем, что позволит веб-броузерам активно кэшировать файлы JavaScript и другое статическое содержимое, и воспользуйтесь преимуществами таких часто используемых параметров, как использование сжатия *gzip*.

*Максимальный объем статического содержимого*

Статическое содержимое передается очень быстро, поэтому чем больший объем будет занимать статическое содержимое, тем меньше времени веб-сервер будет тратить на ответ. Максимизируйте использование статических файлов HTML, насколько это возможно, заполняя отдельные участки через cookies или посредством запросов XHR. Например, если страница регистрации среди сотен байтов статической информации содержит незначительный объем информации, конкретной для каждого пользователя, создайте статическую

страницу и используйте сценарий JavaScript для асинхронного извлечения тех небольших объемов, которые потребуются для заполнения, вместо того, чтобы динамически создавать всю страницу.

*Профилирование*

Если страница работает медленно или ее производительность меняется после загрузки, воспользуйтесь профилировщиком отладчика Firebug, чтобы выявить участки программного кода JavaSript, на работу которых затрачивается основное время, и подумайте об оптимизации выявленных функций.

## Преимущества использования сборок XDomain

Хотя это и не очевидно, но если вы решите создать и использовать в своем приложении сборку XDomain, вы получите множество преимуществ:

- Вы сможете развернуть инструментарий Dojo на выделенной машине и использовать его в различных приложениях независимо от того, находятся ли они в том же самом домене сети.
- Инструкции `dojo.require`, выполняемые во время загрузки страницы, удовлетворяют зависимости не синхронно, а асинхронно (в случае, когда сборка создана с параметрами по умолчанию), что может ускорить загрузку страницы, так как запросы выполняются в неблокирующем режиме.
- Некоторые броузеры, такие как IE, по умолчанию позволяют держать открытыми не более двух подключений на поддомен, поэтому использование сборки XDomain удваивает число потенциальных подключений в приложении – два для Dojo и два для всего остального в локальном домене.
- Если вы обслуживаете несколько приложений, использующих сборку XDomain, общая задержка, обусловленная работой протокола HTTP, наверняка уменьшится, потому что увеличится объем информации, кэшируемой броузерами.

## Не занимайтесь оптимизацией преждевременно

И последнее предупреждение: не занимайтесь оптимизацией преждевременно – никогда не занимайтесь оптимизацией вслепую, основываясь лишь на догадках и предположениях. Всегда используйте информацию, доступную для анализа, такую как информация профилировщика или файлы журналов сервера. Именно в отношении оптимизации наши инстинкты часто могут обманываться. И помните: Firebug – ваш друг.

# В заключение

После прочтения этой главы вы должны:

- Уметь использовать инструменты сборки Dojo с целью создания объединенных, сжатых слоев для вашего веб-приложения
- Знать некоторые параметры настройки, наиболее часто используемые для создания собственных сборок
- Знать, что практически всегда *dojo.js* остается существовать в виде отдельного файла JavaScript, – он не встраивается в слой
- Уметь использовать платформу DOH для создания модульных тестов, выполняющих тестирование функций JavaScript
- Поближе познакомиться с Rhino и понимать роль, которую он играет в инструментах сборки и DOH
- Знать, что ShrinkSafe и DOH, хотя и являются важными компонентами инструментария, тем не менее не имеют прямого отношения к инструментарию Dojo и могут использоваться отдельно от него
- Знать некоторые удобные пути к достижению высокой производительности вашего веб-приложения

# A

# Учебник по работе с отладчиком Firebug

Если вы занимаетесь разработкой веб-приложений, то расширение Firebug (*http://www.getfirebug.com*) для броузера Firefox компании Mozilla (*http://www.getfirefox.com*) – это инструмент, не заметить который было бы для вас непростительной ошибкой. Это особенно верно для тех, кто работает с высокопроизводительным инструментарием JavaScript, потому что Firebug является самым важным инструментом, способным облегчить усилия, прилагаемые для отладки. Это приложение последовательно освещает многие ключевые особенности отладчика Firebug, знакомя вас с самым лучшим способом отладки веб-приложений (или исследования страниц просто для удовольствия).

Это приложение – скорее набор сведений, предназначенный для того, чтобы привлечь внимание к Firebug, чем полномасштабное учебное руководство.

## Установка

Установка Firebug, как и установка любых других расширений для броузера Firefox, выполняется чрезвычайно просто. Откройте страницу *http://www.getfirebug.com* и щелкните на кнопке с предложением установить Firebug. Обратите внимание, что перед вами на экране в верхней части окна может появиться предупреждение, препятствующее установке; в этом случае вам может потребоваться щелкнуть на кнопке Edit Options (Изменить), чтобы подтвердить желание выполнить установку. Как только Firefox будет перезапущен после установки, в меню Tools (Инструменты) должен появиться пункт Firebug и значок Firebug в правом нижнем углу окна броузера, как показано на рис. A.1.

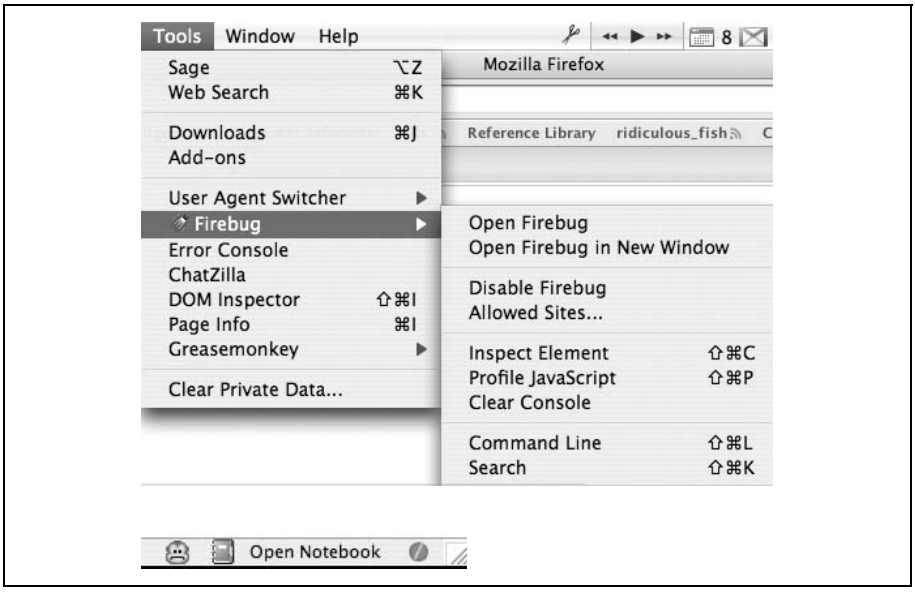



***Рис. А.1.*** *Вверху: сразу после установки расширения Firebug в меню «Tools» («Инструменты») появляется дополнительное подменю, содержащее несколько стандартных пунктов; внизу: в дополнение к пункту в меню «Tools» («Инструменты») в правом нижнем углу окна броузера появляется значок*

## Включать или не включать?

Прежде чем углубляться в изучение потрясающих особенностей Firebug, позволяющих манипулировать деревом DOM и исследовать программный код JavaScript, давайте потратим пару минут на обзор некоторых способов настройки Firebug на автоматическое включение и выключение его для определенных веб-сайтов. Об этой возможности часто забывают, но она очень удобна, потому что позволяет настроить Firebug на использование только для разработки. Например, можно по умолчанию оставлять Firebug отключенным, потому что отладчик существенно замедляет работу веб-приложений, таких как Gmail, активно использующих JavaScript. Однако при этом можно определить список адресов URL разрабатываемых вами веб-приложений, для которых Firebug всегда должен включаться, чтобы вам не приходилось вручную включать и выключать его при одновременной работе с несколькими страницами в нескольких вкладках броузера. Если у вас возникнет потребность использовать Firebug при работе с большим числом сайтов, просто добавьте их в список разрешенных сайтов.

Щелчок правой кнопкой мыши на значке Firebug в правом нижнем углу окна броузера приводит к появлению контекстного меню с пунктами его настройки. В частности, в меню имеются следующие пункты, как видно из рис. А.2.

*Disable Firebug (запретить Firebug)*

Отключает Firebug до тех пор, пока либо в этом пункте меню не будет снята галочка, либо не будет выполнен переход на сайт, находящийся в списке, созданном в результате выбора пункта меню «Allowed Sites...» (разрешенные сайты).

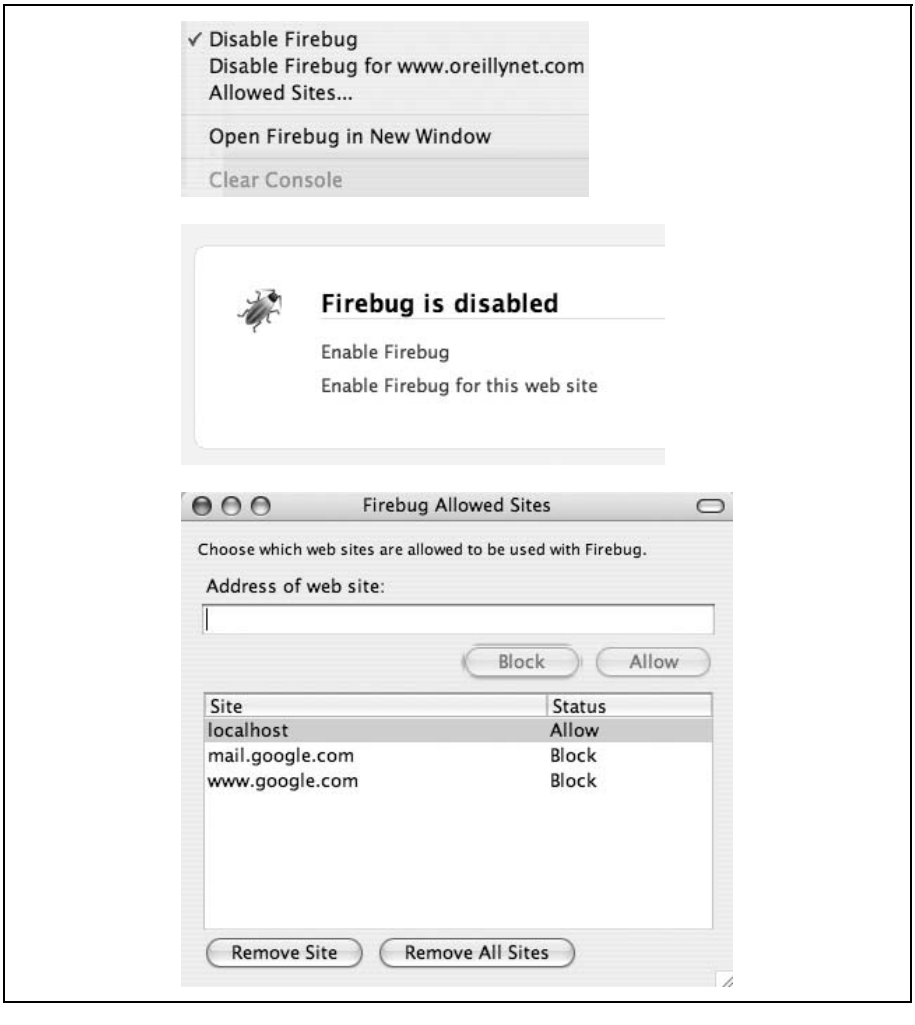



***Рис. А.2.*** *Различные параметры настройки Firebug*

*Disable Firebug for... (запретить Firebug для...)*

Отключает Firebug для текущего сайта и добавляет его в список запрещенных сайтов. Этот пункт меню удобно использовать, когда по умолчанию Firebug включен, но его необходимо отключать для определенных сайтов.

*Allowed Sites... (разрешенные сайты...)*

Открывает диалог «Firebug Allowed Sites» (разрешенные сайты для Firebug), где можно определить собственные списки разрешенных и запрещенных сайтов.

*Open Firebug in New Window (открыть Firebug в новом окне)*

Открывает Firebug в отдельном окне, что очень удобно, когда в основном окне недостаточно места для панели, которая по умолчанию открывается в нижней части окна броузера.

## Более интересные сведения

Теперь, когда была выполнена настройка Firebug для работы с определенными сайтами, можно приступать к рассмотрению некоторых замечательных его особенностей, о которых так много говорится. Начальная страница сайта O'Reilly Network (*http://www.oreillynet.com*) вполне подойдет в качестве отправной точки в наших исследованиях.

Перейдя на страницу *http://www.oreillynet.com*, активируйте Firebug и обратите внимание, что значок в правом нижнем углу окна броузера, ранее окрашенный в серые тона, превратился в зеленый кружочек с галочкой, как показано на рис. А.3. После этого вы можете заметить некоторую активность в панели, расположенной ниже линии вкладок, когда выбрана вкладка «Console» (консоль) при обработке запросов GET. Наведение указателя мыши на гиперссылку вызывает появление подсказки, указывающей местоположение сценария на сервере.

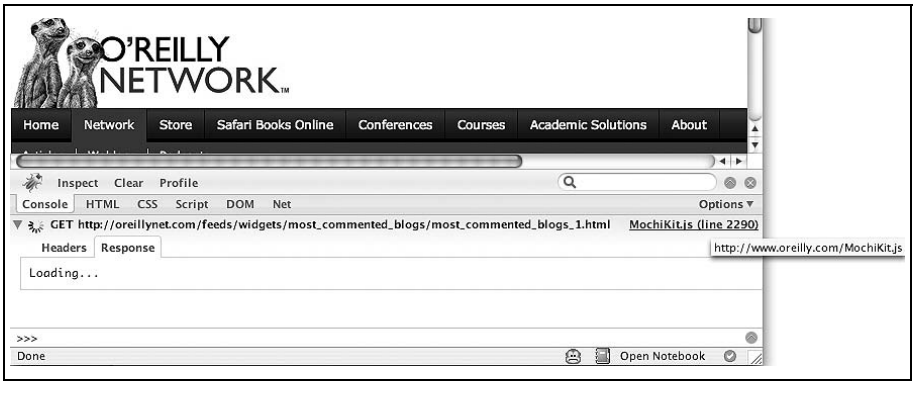



***Рис. А.3.*** *Firebug предоставляет информацию о выполняемых действиях, ценную для отладки (и исследования)*

Мы поочередно рассмотрим все вкладки, которые вы можете наблюдать в интерфейсе Firebug: Console (консоль), HTML, CSS, Script (сценарий), DOM и Net (сеть). Как будет показано далее, Firebug обладает феноменальным количеством функциональных возможностей, упакованных в то, что называют «простым» расширением. К тому же, Firebug обладает очень удобным и интуитивно понятным интерфейсом, работать с которым становится очень легко почти сразу после начала работы с ним.

# Inspect

Обратите внимание на красочный значок Firebug в левом верхнем углу панели Firebug. Помимо художественного оформления, он также выполняет функции кнопки, щелчком на которой открывается меню, очень напоминающее подменю Firebug в меню Tools (Инструменты). Правее этой кнопки располагаются другие кнопки: Inspect (обзор), Clear (очистить) и Profile (профилировать). Начнем с кнопки Inspect (обзор).

Кнопка Inspect (обзор) позволяет быстро отыскать местонахождение любого элемента в дереве DOM, двигая указателем мыши в окне броузера, что очень удобно, когда вы пытаетесь исследовать определенную часть сложной схемы компоновки или отыскать иголку в стоге сена. Щелкните на кнопке Inspect (обзор), чтобы она зафиксировалась в нажатом состоянии, как показано на рис. А.4. Главное меню всегда содержит элементы главного меню и всегда находится в левом верхнем углу. Кнопка Inspect (обзор) всегда находится рядом независимо от выбранной вкладки, что очень удобно, так как это одна из тех функций, которую вы наверняка будете использовать наиболее часто. Обратите внимание, как при этом Firebug переключается на вкладку HTML и при перемещении указателя мыши над различными элементами страницы отображает соответствующий код HTML и CSS. Код разметки HTML, соответствующий элементу, находящемуся под указателем мыши, подсвечивается во вкладке HTML, чтобы вам было проще исследовать как сам элемент, так и его окружение.

Щелчок на элементе страницы приводит к тому, что соответствующее ему содержимое подсвечивается во вкладке HTML, а прокручивание прекращается, чтобы вы могли пользоваться мышью, не теряя найденное место. Как только вы нашли интересующий вас элемент DOM, с ним можно выполнить некоторые действия во вкладке HTML: отредактировать содержимое узла, добавить атрибуты, удалить атрибуты, определить стили CSS, применяемые к этому узлу, и т. д.

Уделите некоторое время, чтобы поближе познакомиться с функцией обзора, исследовав различные аспекты главной страницы O'Reilly Network. В частности, достаточно интересно попробовать изменять элементы в дереве DOM и понаблюдать за эффектами, которые оказывают

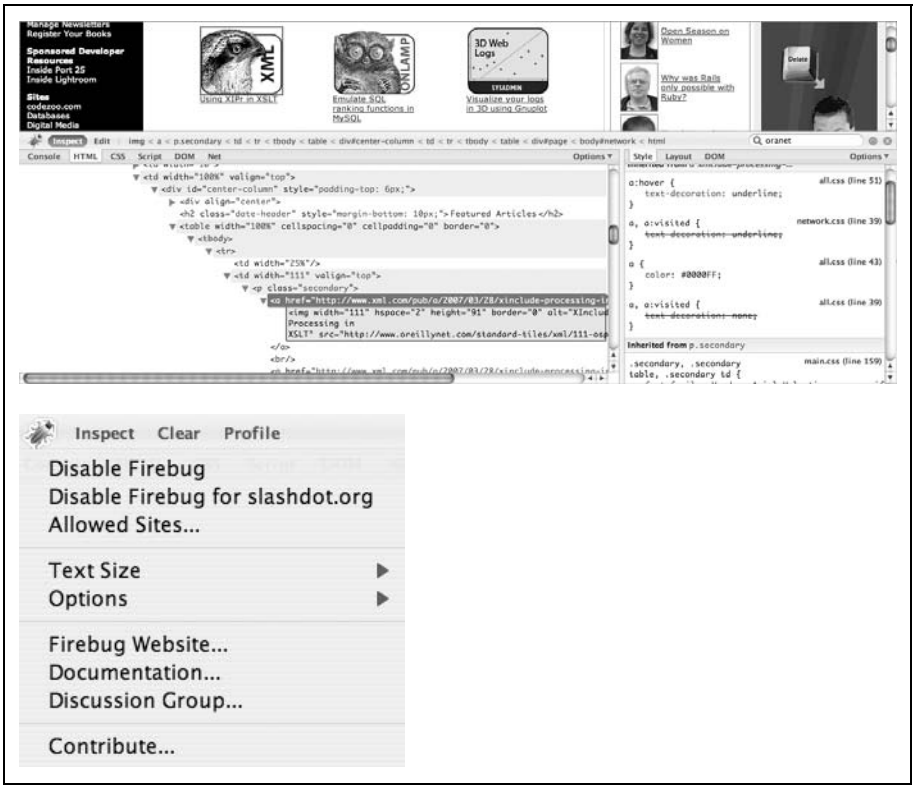



***Рис. А.4.*** *Кнопка «Inspect» (обзор) в действии*

эти изменения, прямо в окне броузера. Например, на рис. А.5 демонстрируется результат изменения ширины рисунка в заголовке на главной странице. Чтобы изменить атрибуты любого тега и увидеть, что из этого получится, достаточно просто щелкнуть на разметке HTML. Можете также попробовать добавить новые атрибуты, выбрав требуемый элемент DOM и щелкнув на кнопке Edit (редактировать), которая появляется рядом с кнопкой Inspect (обзор) в режиме обзора.

## Console

Возможно, вы уже обратили внимание, что в панели Firebug на вкладке Console (консоль) присутствует приглашение к вводу команды. Эта командная строка, как показано на рис. А.6, выполнит любой программный код JavaScript, который вы введете. Ее можно использовать для проверки новых идей или быстрого получения ссылок на узлы и манипулирования ими. В частности, вы можете получить ссылку на любой элемент страницы с помощью функции `document.getElementById`.

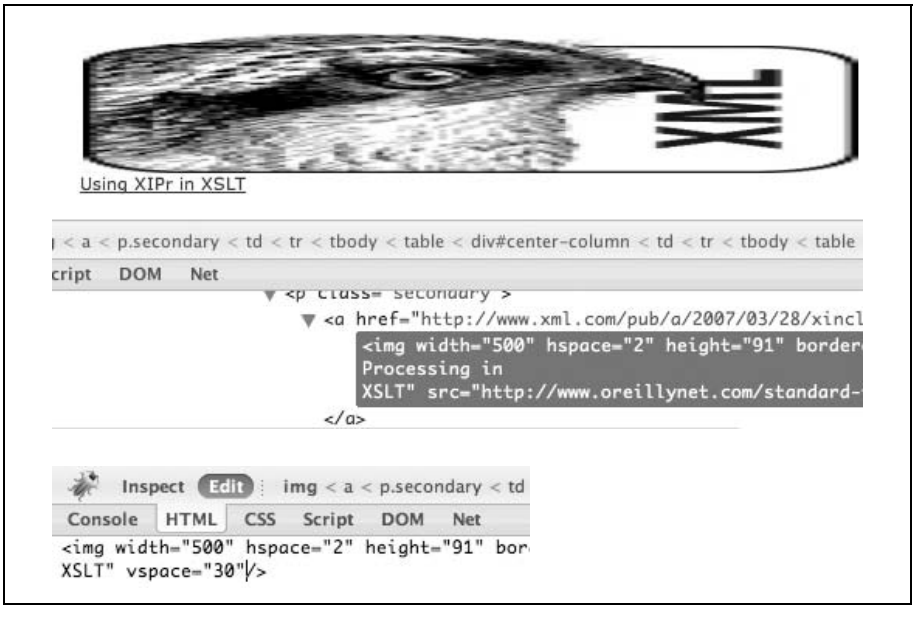



***Рис. А.5.*** *Вы можете использовать Firebug для экспериментов с кодом разметки HTML страницы и сразу же наблюдать получающиеся результаты*

Поэтому, если элемент не имеет значения в атрибуте `id`, вы можете задать значение и двигаться дальше. Кроме того, вы можете воспользоваться встроенной функцией консоли – `console.dir` (точно так же, как в Python), чтобы получить перечень доступных методов. Теперь самое время обратиться к документации Firebug (*http://getfirebug.com/docs.html*), где вы найдете описание консоли, чрезвычайно удобных функций, таких как `dir`, и многое другое.

Еще одна контекстная кнопка видима, когда выбрана вкладка Console (консоль), – это кнопка Profile (профилировать), которая делает именно то, что вы и подумали: выполняет профилирование программного кода JavaScript в странице, как показано на рис. А.7. Щелчок на этой кнопке запускает профилирование, а повторный щелчок останавливает его, и в консоли появляются собранные статистические данные. Существует ли более простой способ получения этой информации? На снимке с экрана, представленном ниже, приводятся некоторые результаты профилирования программного кода JavaScript в странице *http://jobs.oreilly.com*, выполняемого в результате щелчка на панели навигации в левой части страницы.

Особенно примечательно, что инструментарий Dojo тесно интегрирован с Firebug. Во время загрузки в Firefox он обеспечивает доступ к консоли для вывода отладочных сообщений с помощью функции `console.log`. При работе в других броузерах вывод сообщений и средства

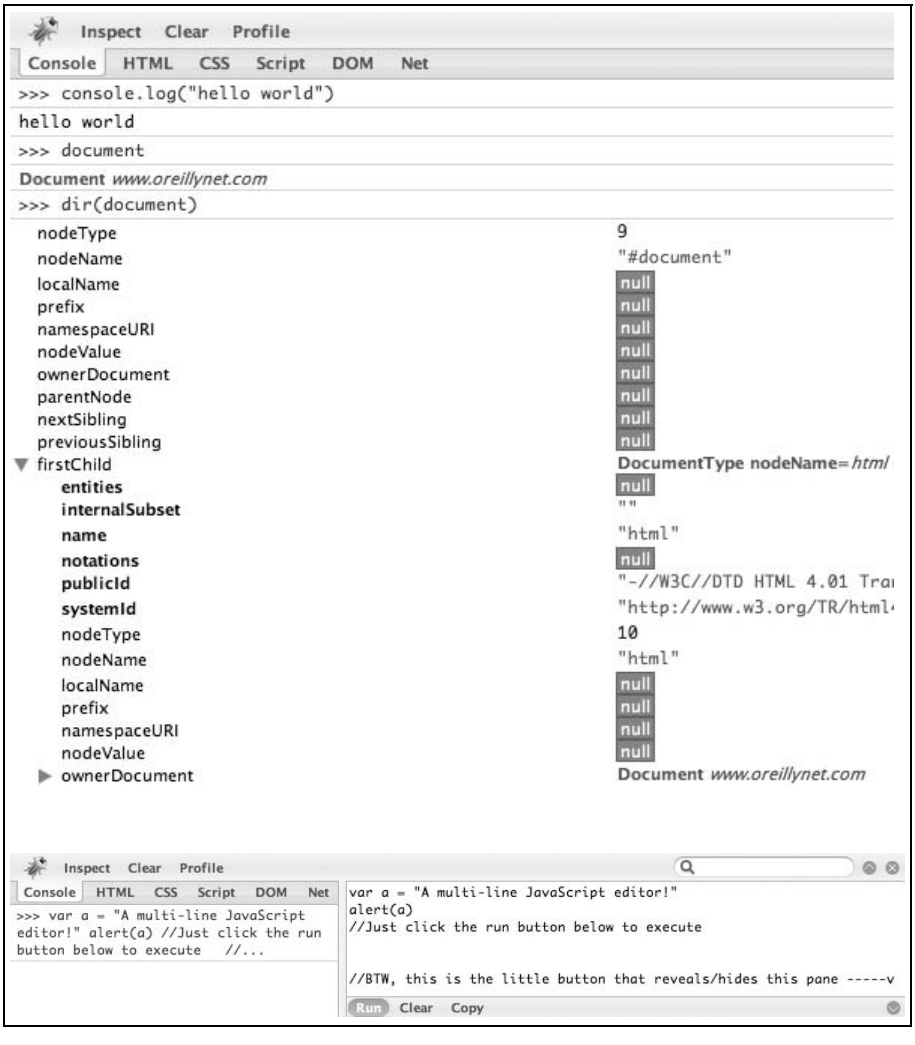



***Рис. А.6.*** *Слева: консоль – это программный интерфейс к любой веб-странице; справа: щелчок на кнопке с изображением «крышки» («^») справа от строки приглашения к вводу открывает многострочный редактор программного кода JavaScript*

взаимодействия с деревом DOM поддерживаются с помощью Firebug Lite (консоль с минимальными возможностями).

Поскольку функция `console.log` очень часто используется для отладки, нелишним будет знать, что вы должны явно выводить элементы, отделяя их от остального вывода, и не использовать, например, операцию конкатенации. Дело в том, что Firebug позволяет просматривать

***Рис. А.7.*** *В состав Firebug входит мощный профилировщик, позволяющий получить разнообразную полезную информацию о производительности программного кода JavaScript в страницах*

элементы по щелчку мыши, но если вы неявно преобразуете их в строку, вы потеряете это преимущество. Например, вы могли бы выводить информацию о переменной с именем `foo` с помощью вызова `console.log("the value of foo is", foo)`, чтобы получить возможность исследовать содержимое `foo`. Если вместо этого вызова вы используете вызов `console.log("the value of foo is" + foo)`, вы получите строку. Для данных базовых типов это может не иметь большого значения, но если переменная `foo` представляет сложный объект, вам наверняка пригодится возможность исследовать его содержимое.

# HTML и CSS

Вам должны понравиться вкладки HTML и CSS, потому что в них отображается то же самое содержимое, что и при нажатой кнопке «Inspect» (обзор). Еще одна ценная особенность состоит в том, что можно щелкнуть правой кнопкой мыши на любом элементе в одной из этих вкладок и получить контекстное меню с набором полезных пунктов. Эти пункты предоставляют возможность прокручивать страницу до выбранного элемента, производить регистрацию событий, связанных с элементом, вносить изменения в элемент и т. д.

Расширенный объем информации, что выводится во вкладке CSS, также выводится во вкладке HTML, если выбрать дополнительную вкладку Style (стиль) в панели справа. Обратите внимание, что вы можете напрямую редактировать стили элементов как во вкладке CSS, как показано на рис. А.8, так и во вкладке HTML. Кроме того, можно щелкнуть на элементе стиля, чтобы запретить его действие в зависимости от того, что требуется в каждой конкретной ситуации. Но и это еще не все: во вкладке Layout (компоновка), расположенной рядом со вкладкой Style (стиль), отображаются активные свойства текущего элемента, определяющие ширину отступов, границ, полей и смещений. Как можно догадаться, вы можете наблюдать влияние этих свойств, изменяя их значения непосредственно в соответствующем графическом представлении.

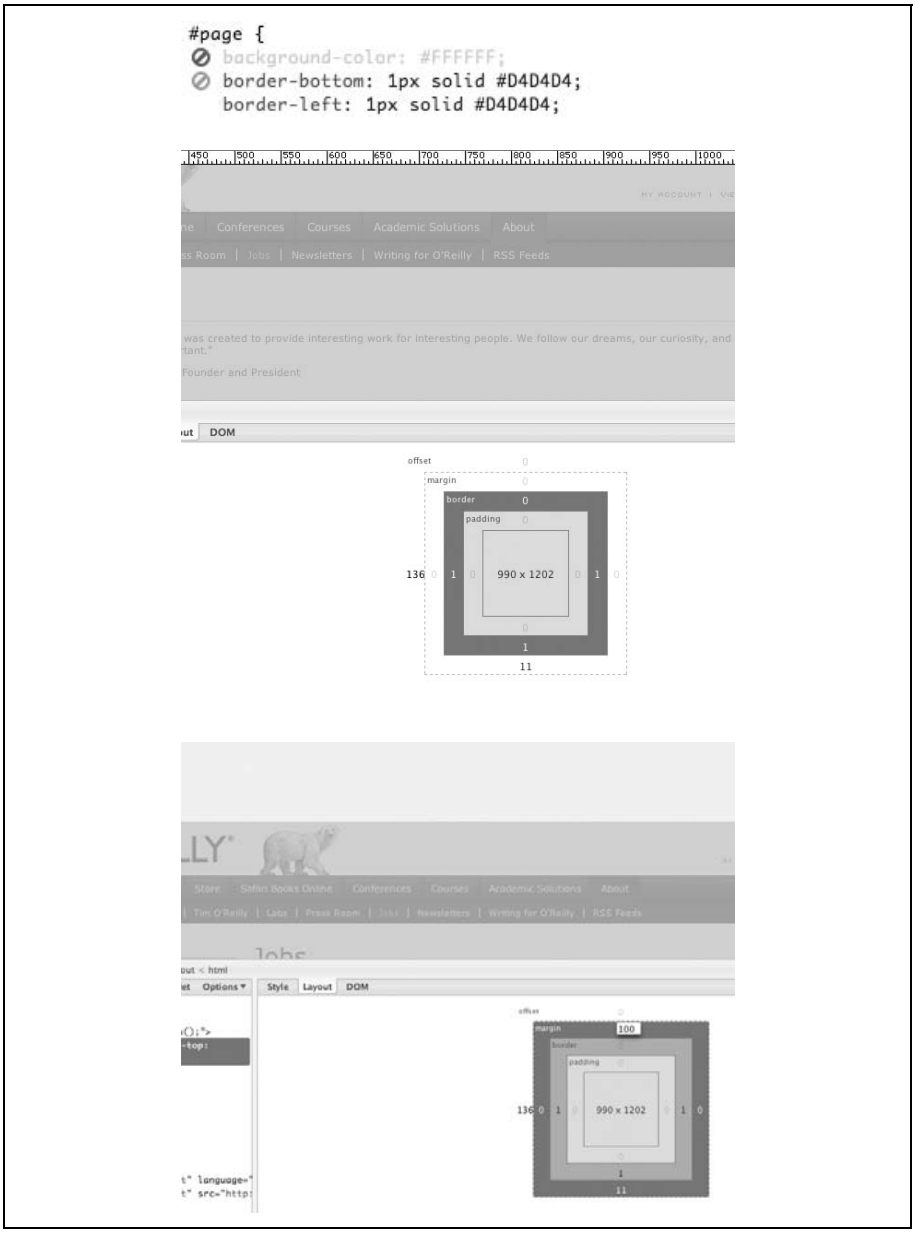



***Рис. А.8.*** *Вверху: отключение элементов стиля; в середине: линейка компонента «Layout» (компоновка) в Firebug; внизу: манипулирование свойством элемента DOM, определяющим ширину поля, или другими подобными свойствами (в этом примере показано, как изменилось верхнее поле, после того как ему было установлено значение 100px)*

## Script и DOM

Во вкладке Script находится мощный отладчик программного кода JavaScript, позволяющий устанавливать точки останова в сценариях и просматривать значения переменных в процессе выполнения. В отличие от некоторых других отладчиков, с которыми вам, возможно, приходилось сталкиваться ранее, отладчик JavaScript в Firebug весьма прост в освоении. Чтобы установить точку останова, достаточно просто загрузить интересующий сценарий с помощью контекстного меню, расположенного чуть выше вкладки Script (сценарий), и щелкнуть на номере строки, где требуется установить точку останова. В этом месте должен появиться маленький красный кружок, подтверждающий установку точки останова. Вы можете щелкнуть на красном кружочке, чтобы удалить точку останова, или щелкнуть на любой другой строке, чтобы поставить еще одну точку останова. Кроме того, во вкладке Watch (наблюдать) можно ввести имя переменной или выражение с ее участием, чтобы наблюдать за тем, как изменяется ее значение между двумя точками останова. Разве существует более простой способ отладки JavaScript?

Название вкладки DOM, которая отображена на рис. А.9, говорит само за себя. Фактически в этой вкладке выводится та же самая информация, что и во вкладке HTML, но только в виде дерева, что в зависимости от ситуации может упростить исследование элементов страницы и манипулирование ими.

## Net

К настоящему моменту вы познакомились с большинством мощных и полезных возможностей Firebug. Однако «на сладкое» осталась еще вкладка Net (сеть), которая показана на рис. А.10. В основном вкладка Net (сеть) предоставляет полную информацию, которая потребуется вам для оценки производительности, поскольку она имеет отношение к загрузке компонентов страницы и для удобства логически объединяет компоненты различных типов в категории, такие как CSS, JavaScript, изображения и т. д. Особое место занимает категория, содержащая статистическую информацию о запросах XHR (AJAX). Одним словом, вкладка Net (сеть) предоставляет ценную информацию, которая позволит выявить узкие места с точки зрения производительности и определить время загрузки содержимого. Для проектов с большим объемом динамического содержимого эта вкладка равноценна сетевому профилировщику.

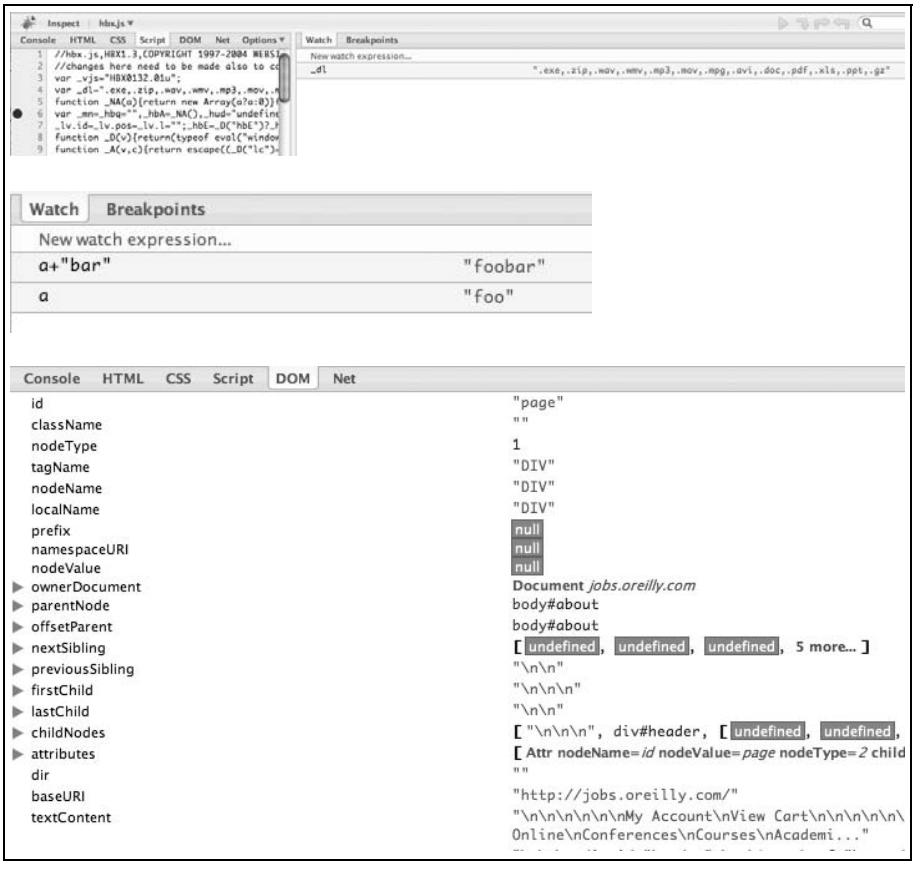



***Рис. А.9.*** *Вверху: установка точек останова в отладчике программного кода JavaScript; в середине: наблюдение за значениями переменных и даже за значениями введенных вами выражений; внизу: вкладка «DOM» позволяет исследовать элементы DOM страницы и манипулировать ими, используя традиционное древовидное представление*

# Идите и исследуйте

Мы рассмотрели наиболее значимые особенности отладчика Firebug, но лучший способ овладеть им состоит в том, чтобы потратить некоторое время на исследование веб-страниц с его помощью. Отправьтесь на любой сайт с не очень сложной разметкой и проведите некоторое время, чтобы выяснить, как разработчики сводят разные компоненты воедино. Помимо удовольствия, получаемого от работы детектива, вы попутно узнаете массу интересного. После этого, когда в следующий раз вам понадобится быстро отладить результаты своего труда, вы смо-

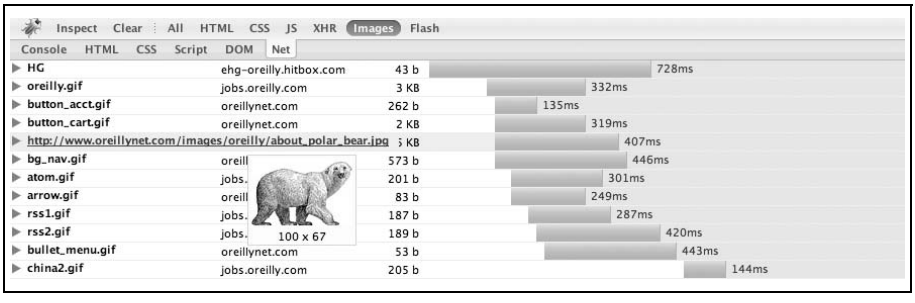



***Рис. А.10.*** *Вкладка «Net» (сеть) в отладчике Firebug распределяет загружаемое содержимое страницы по категориям и показывает соответствующие им времена загрузки*

жете использовать свои навыки владения отладчиком Firebug и быстро исправить ситуацию, не пролив ни капли пота. (Напомню еще раз, что это приложение не является исчерпывающим руководством по работе с отладчиком Firebug, – оно дает лишь начальные сведения, чтобы дать вам почувствовать, сколько времени вам удастся сэкономить, если научиться пользоваться им.)

# B

# Краткий обзор DojoX

Библиотека DojoX – это каноническое место размещения экспериментальных и специализированных расширений инструментария Dojo. В отличие от остальной части книги, где подробно описываются библиотеки Base, Core, Dijit и Util, в этом небольшом приложении дается лишь краткий обзор библиотеки DojoX, заслуживающей отдельной книги, которая могла бы быть в два раза толще этой. Я надеюсь, что после ознакомления с основной частью инструментария применение чего-либо из библиотеки DojoX не должно представлять сложностей.

Автор практически регулярно ведет свою колонку о Dojo на сайте *http://dojotdg.com*, где кроме всего прочего имеется информация о проектах, входящих в состав библиотеки DojoX, поэтому подумайте о том, чтобы добавить ссылку на нее в свой инструмент чтения RSS и быть в курсе событий.

Управление библиотекой DojoX осуществляется на основе проектов, входящих в ее состав, а условия в каждом конкретном проекте могут существенно отличаться от других. Некоторые проекты, такие как *cometd* и *charting*, отличаются высокой стабильностью, в то время как другие находятся лишь в начальной стадии разработки, и библиотека DojoX предоставляет им возможность дальнейшего развития. Единственное общее требование, предъявляемое ко всем проектам в библиотеке DojoX, состоит в том, что в них должен присутствовать файл *README*, содержащий информацию о состоянии проекта, его версии, а также контактную информацию об авторах. Проекты в библиотеке DojoX могут опираться на библиотеки Base, Core или Dijit, но они могут быть и абсолютно независимыми разработками. В отличие от библиотеки Dijit, DojoX не дает официальных гарантий обеспечения доступности или интернационализации, и стиль реализации проектов может существенно изменяться, в отличие от единых проектов, таких как Dijit.

Главное, на что вы должны были обратить внимание при изучении остальной части инструментария, – это его глубина и широта; библиотека DojoX в этом отношении не является исключением. Грубый анализ числа функций (включая встроенные анонимные функции) и инструкций в библиотеках Core, Base и Dijit, полученного с помощью рассмотренных ниже команд, приводится в табл. B.1:

```
grep -rc 'function' * | grep -v \.svn | cut -d : -f 2 | awk '{for (i=1; i<=NF;
i++) s=s+$i}; END{print s}'
```

И

```
grep -rc '\;' * | grep -v \.svn | cut -d : -f 2 | awk '{for (i=1; i<=NF; i++)
s=s+$i}; END{print s}'
```

*Таблица B.1. Грубая оценка числа функций и инструкций (в тысячах) в инструментарии версии 1.1*

| **Base** | | **Core** | | **Dijit** | | **DojoX** | |
|---|---|---|---|---|---|---|---|
| Функций | Инструкций | Функций | Инструкций | Функций | Инструкций | Функций | Инструкций |
| 0,7 | 2,2 | 1,9 | 9,5 | 1,6 | 15,1 | 7,1 | 54 |

Из таблицы видно, какой огромный объем программного кода содержит библиотека DojoX. В любом случае, дополнительная широта и глубина, обеспечиваемые библиотекой DojoX, поражают воображение, и во многих случаях реализации, присутствующие в DojoX, гораздо более интересны, чем в библиотеке Dijit, потому что DojoX, по сути, является передовым краем и средоточием специализированных интересов. То, что в этой книге не удалось охватить библиотеку DojoX, еще не означает, что вы не найдете в ней множество полезных особенностей, которые помогут вам сэкономить массу времени.

Краткий перечень проектов, входящих в библиотеку DojoX, что приводится в табл. B.2, может служить неплохим путеводителем по ресурсам библиотеки DojoX версии 1.1. Библиотека DojoX постоянно изменяется, поэтому лучший способ оставаться в курсе того, что входит в ее состав, заключается в том, чтобы загрузить ночную сборку по адресу *http://archive.dojotoolkit.org/nightly/* и исследовать содержимое файла *README*.

*Таблица B.2. Проекты DojoX*

| **Проект** | **Описание** |
|---|---|
| *analytics* | Механизмы мониторинга системы, которые могут использоваться для регистрации на стороне сервера различных событий, происходящих на стороне клиента, таких как щелчки мышью, периоды бездействия, вызовы функций `console.*` и другие. |
| *av* | Проект реализации поддержки аудио/видео в роликах Flash и QuickTime. |

| Проект | Описание |
|---|---|
| *charting* | Расширенный механизм представления диаграмм, опирающийся на проекты *dojox.gfx* и *dojox.gfx3d*. |
| *collections* | Множество функций, обеспечивающих поддержку таких структур данных, как стеки, множества, очереди, расширенные ассоциативные массивы, массивы и т. д. |
| *color* | Улучшенная поддержка таких цветовых схем, как CMYK и HSL, а также HSV. |
| *cometd* | Реализация протокола Bayeaux, отличающаяся низкой задержкой передачи данных от сервера к клиенту. |
| *data* | Поддержка таких механизмов доступа к данным, как FlickrRestStore, XmlStore, CsvStore и других, которые реализуют интерфейс `dojo.data` и предоставляют дополнительные функции для `dojo.data`. |
| *date* | Реализация операций с датами, таких как форматирование, которые часто присутствуют в других языках программирования и серверных технологиях, например в PHP. |
| *embed* | Средство, упрощающее вставку внешних объектов, для которых обычно требуются теги `OBJECT` и `EMBED`. |
| *dtl* | Проект, целью которого является полная реализация языка шаблонов Django (Django Template Language). |
| *encoding* | Комплект процедур, реализующих распространенные алгоритмы кодирования для нужд шифрования, дайджестов и сжатия. |
| *flash* | Проект, целью которого является распространение возможностей Flash в среду DHTML. |
| *form* | Коллекция удобных виджетов форм, включающих такие функциональные возможности, как проверка пароля, элементы множественного выбора, в которых выбор нескольких элементов производится не с помощью щелчка мыши при нажатой клавише Ctrl, а с помощью флажков. |
| *fx* | Набор анимационных эффектов, расширяющих и дополняющих эффекты в библиотеках Base и Core. |
| *gfx* | Переносимая библиотека двухмерной графики, использующей такие технологии, как VML, SVG и другие, обеспечивающая возможность создания графических изображений – как статических, так и анимированных. |
| *gfx3d* | Переносимая библиотека трехмерной графики. Опирается на возможности, реализованные в библиотеке *gfx*. |
| *grid* | Мощный компонент представления данных в табличном виде, способный воспроизводить любые объемы данных, извлекаемых с помощью механизмов доступа к данным. |
| *highlight* | Механизм подсветки синтаксиса, который обеспечивает возможность подсветки синтаксиса различных языков программирования в содержимом тегов `<CODE>`. |

*Таблица В.2 (продолжение)*

| Проект | Описание |
|---|---|
| *image* | Обеспечивает поддержку обычных операций над изображениями, таких как воспроизведение слайдшоу из последовательности изображений, увеличение, создание эскизов, осветление и т. д. |
| *io* | Поддержка составных функций XHR и механизма прокси на основе XHR IFRAME для выполнения запросов в другие домены с помощью объекта `XMLHttpRequest`. |
| *jsonPath* | Напоминает Xpath, но используется для выполнения запросов к объектам JavaScript. Очень удобно применять при работе со сложными структурами JSON. |
| *lang* | Языковые утилиты, обеспечивающие дополнительные операции с массивами, хешами и расширениями функционального программирования (lambda-выражения). |
| *layout* | Дополнительные виджеты компоновки. |
| *math* | Множество дополнительных математических функций, таких как функции абстрактного определения кривых, вычисления координат. |
| *off* | Обертка вокруг механизма Google Gears, которая обеспечивает возможность функционирования веб-приложений без подключения к сети. |
| *presentation* | Механизм решения разнообразных задач, связанных с отображением чего-либо на экране, например презентаций. |
| *rpc* | Дополнительные компоненты, опирающиеся на `dojo.rpc`, для вызова удаленных процедур. |
| *sketch* | Графический редактор, не зависящий от типа броузера, основанный на модуле `dojox.gfx`. |
| *storage* | Абстракция на языке JavaScript, обеспечивающая ограниченную поддержку механизмов доступа к данным посредством «родных» для броузеров расширений, таких как Flash и Google Gears. |
| *string* | Различные функции для работы со строками. |
| *timing* | Поддержка дополнительных конструкций определения временных интервалов. |
| *uuid* | Реализация вычисления универсально-уникальных идентификаторов (Universally Unique Identifier) в соответствии с RFC 4122. |
| *validate* | Набор функций для решения типичных задач проверки содержимого, таких как адреса электронной почты, номера карточек социального обеспечения и т. д. |

| Проект | Описание |
|---|---|
| *widget* | Набор виджетов, напоминающих те, что можно найти в библиотеке Dijit, включая улучшенный элемент выбора цвета, список с эффектом «рыбий глаз», тостер, мастер, лупа, улучшенные панели с прокруткой и многие другие. |
| *wire* | Прикладной программный интерфейс, предоставляющий упрощенные шаблоны MVC (Model-View-Controller – модель-представление-контроллер) на стороне клиента, реализующий универсальный механизм привязки данных и библиотеку для работы с сетевыми службами. |
| *xml* | Утилиты для работы с форматом XML. |

# Алфавитный указатель

## Специальные символы


#, символический оператор, 169
$=, символический оператор, 169
*, символический оператор, 168
*=, символический оператор, 169
, (запятая)
    завершающие запятые, 303
    как символический оператор, 169
^=, символический оператор, 169
~=, символический оператор, 169
>, символический оператор, 182


## A


AccordionContainer, диджит, 343
    обзор, 444
AJAX (Asynchronous JavaScript and XML – асинхронный JavaScript и XML)
    hitch, функция, 137
    OpenAjax Alliance, 164
    WAI-ARIA, поддержка, 321
    обзор, 128
    поддержка форм, 377
    примеры применения функций XHR, 134
_Animate, класс
    переключение состояния узла, 248
animateProperty, функция
    параметры, 231
    пример, 231
_Animation, класс, 227, 240
    animateProperty, свойство, 184
    gotoPercent, метод, 237
    pause, метод, 237
    play, метод, 226, 237
    status, метод, 237
    stop, метод, 236
    программное управление анимацией, 236
    простые эффекты растворения и проявления, 227
    создание цепочек и комбинирование эффектов, 245
ant, инструмент сборки, 493
ARIA (Accessibility for Rich Internet Applications – стандарт доступности активных Интернет-приложений), 316
array, ключевое слово, 80, 179
Array, объект
    NodeList, класс, 168
    наследование, 304
    поддержка в JavaScript, 75
    поддержка в броузерах Mozilla, 76
    прикладные диджиты, 460
ASCII, набор символов, 403


## B


Base, библиотека, 38
    Color, класс, 251
    Deferred, структура данных, 148
    в зависимостях библиотеки DojoX, 529
    исследование, 60
    клонирование объектов, 94
    манипулирование контекстом объекта, 94
    нормализация событий и клавиатуры, 113
    обзор, 38
    обработка массивов, 76
    обработчики событий, 116
    определение типа, 73
    организация взаимодействий по подписке, 123

поддержка AJAX, 131
поддержка Dijit, 316
поддержка анимации, 226
поиск узлов DOM, 72
примерное число функций, 530
профили сборки, 496, 499
управление исходным программным кодом, 81
утилиты для работы с броузером, 106
утилиты для работы с деревом DOM, 99
утилиты для работы с объектами JavaScript, 90
утилиты для работы со строками, 74
beforeBegin, событие, 237, 245
blur, событие, 387
BorderContainer, диджит, 343
back, функция, 440
design, атрибут, 434
forward, функция, 440
liveSplitters, атрибут, 434
maxSize, атрибут, 434
minSize, атрибут, 434
persist, атрибут, 434
region, атрибут, 434
splitter, атрибут, 434
обзор, 433
buildscripts, каталог, 493
Button, диджит
iconClass, свойство, 409, 457
label, свойство, 409
onClick, событие, 339
onClick, точка расширения, 409
setAttribute, функция, 409
setLabel, метод, 409
showLabel, свойство, 409
и диджит Toolbar, 457
и функция dojo.require, 411
обзор, 341, 409
создание экземпляра, 328

## C

CheckBox, диджит
checked, атрибут, 412
getValue, метод, 414
onChange, точка расширения, 414
setValue, метод, 412
type, атрибут, 412
value, атрибут, 412, 414
обзор, 411
CherryPy, сервер
загрузка, 47
загрузка файлов, 155
исследование объекта Deferred, 142
поддержка форм, 378
типы ответов, отличные от HTML, 158
Color, класс, 251
Color, метод, 252
setColor, метод, 252
toCss, метод, 253
toHex, метод, 253
toRgb, метод, 252
toRgba, метод, 253
toString, метод, 253
дополнительные именованные значения цветов, 255
именованные значения цветов, 254, 255
создание и смешивание цветов, 251
ColorPalette, диджит
defaultTimeout, атрибут, 456
onChange, функция, 457
palette, 457
timeoutChangeRate, атрибут, 456
обзор, 344, 456
ComboBox, диджит
autoComplete, атрибут, 405
close, метод, 405
fetch, метод, 405
fetchItemByIdentity, метод, 405
fetchSelectedItem, метод, 405
getIdentity, метод, 405
getLabel, метод, 405
getValue, метод, 405
hasDownArrow, атрибут, 406
ignoreCase, атрибут, 406
isItemLoaded, метод, 405
item, атрибут, 405
pageSize, атрибут, 405
query, атрибут, 405
queryExpr, атрибут, 406
searchAttr, атрибут, 405
searchDelay, атрибут, 405
store, атрибут, 405
обзор, 341, 401
ComboButton, диджит, 451
и диджит TooltipDialog, 451
и функция dojo.require, 411
обзор, 416
_Contained, класс, 371, 426

_Container, класс, 371
и диджит Menu, 462
и диджиты компоновки, 426
отношения родитель-потомок, 371
ContentPane, диджит, 342
cancel, метод, 431
errorMessage, атрибут, 431
extractContent, атрибут, 430
href, атрибут, 430
isLoaded, атрибут, 431
loadingMessage, атрибут, 431
onContentError, точка расширения, 431
onDownloadEnd, точка расширения, 431
onDownloadError, точка расширения, 431
onDownloadStart, точка расширения, 431
onLoad, точка расширения, 431
onUnload, точка расширения, 431
parseOnLoad, атрибут, 430
preload, атрибут, 430
preventCache, атрибут, 430
refresh, метод, 431
refreshOnShow, атрибут, 430
setContent, метод, 431
setHref, метод, 431
и BorderContainer, 434
и диджит Dialog, 451
обзор, 429
отложенная загрузка, 513
пример, 436
расширение, 432
cookies, 107
Core, библиотека, 39
back, модуль, 109
behavior, модуль, 187
DeferredList, структура данных, 148
dojo.fx, модуль, 226, 240
dojo.rpc, модуль, 161
i18n, модуль, 193
OpenAjax, модуль, 164
string, модуль, 75
в зависимостях библиотеки DojoX, 529
дополнительные именованные значения цветов, 255
парсер, 330
парсеры, 204
передача данных с помощью IFRAME, 154
поддержка Dijit, 316
поддержка JSONP, 152
поддержка интернационализации, 192, 197
поддержка тестирования, 507
примерное число функций, 530
CSS
Color, модуль, 255
HelloWorld, диджит, 362
анимация произвольных свойств, 230
блочная модель, 102
и диджиты, 350
наиболее часто используемые селекторы, 168
нарезка изображений, 460
поддержка RGBA, 253
учебное руководство, 203
CurrencyTextBox, диджит
обзор, 400

## D

Date, объекты (JavaScript), 396
DateTextBox, диджит
datePattern, атрибут, 394
formatLength, атрибут, 393
getDisplayedValue, метод, 396, 397
locale, атрибут, 394
select, атрибут, 394
serialize, точка расширения, 397
setDisplayedValue, метод, 396, 397
strict, атрибут, 394
toString, метод, 397
Григорианский календарь, 393
обзор, 391
Declaration, инструмент, 372
Deferred структура данных
canceller, свойство, 141
CherryPy, примеры, 142
fired, свойство, 141
silentlyCancelled, свойство, 141
Deferred, структура данных
addBoth, функция, 141
addCallback, функция, 141
addCallbacks, функция, 141
addErrback, функция, 141
callback, функция, 141
cancel, функция, 141, 148

errback, функция, 141
асинхронные запросы, 139
DeferredList, структура данных, 148
DHTML, анимация, 184
Dialog, диджит
duration, атрибут, 451
hide, метод, 451
layout, метод, 451
open, атрибут, 451
setContent, метод, 450
show, метод, 451
обзор, 343, 449
Dijit, библиотека, 39
доступность, 319
интернационализация, 192
исследование, 68
обзор, 315
поддержка DojoX, 529
поддержка блочной модели, 106
поддержка доступности, 380
поддержка парсеров, 204
предопределенные темы, 325, 380
примерное число функций, 530
функции прикладного интерфейса, 345
dijit.byId, функция, 328
dijit.form, модуль, 380
Button, диджит, 328, 339, 341, 409, 457
CheckBox, диджит, 411
ComboBox, диджит, 341, 401
ComboButton, диджит, 416, 451
ContentPane, диджит, 464
CurrencyTextBox, диджит, 400
DateTextBox, диджит, 391
DropDownButton, диджит, 415, 451
FilteringSelect, диджит, 341, 406
Form, диджит, 340, 423
HorizontalRule, диджит, 418, 422
HorizontalRuleLabel, диджит, 422
HorizontalRuleLabels, диджит, 418
HorizontalSlider, диджит, 418
MappedTextBox, диджит, 390, 406
MultiSelect, диджит, 342, 407
NumberSpinner, диджит, 334, 341, 399
NumberTextBox, диджит, 399
RadioButton, диджит, 414
RangeBoundTextBox, диджит, 390, 399
SimpleTextarea, диджит, 341, 408
Slider, диджит, 341
Textarea, диджит, 341, 408, 466
TextBox, диджит, 342, 384, 398, 466
TimeTextBox, диджит, 391
ToggleButton, диджит, 410
ValidationTextBox, диджит, 387, 398
VerticalRule, диджит, 421, 422
VerticalRuleLabel, диджит, 422
VerticalRuleLabels, диджит, 421
VerticalSlider, диджит, 420
структура дерева наследования, 380
dijit.layout, модуль
AccordionContainer, диджит, 343, 444
addChild, метод, 372, 427
BorderContainer, диджит, 343, 433
ContentPane, диджит, 342, 429, 451, 513
domNode, свойство, 428
getChildren, метод, 372, 427
getNextSibling, метод, 372, 428
getParent, метод, 372, 427
getPreviousSibling, метод, 372, 428
isLayoutContainer, метод, 427
layout, метод, 427
LayoutContainer, диджит, 433
removeChild, метод, 372, 427
resize, метод, 427, 450
SplitContainer, диджит, 433
StackContainer, диджит, 342, 439
TabContainer, диджит, 342, 429, 441
наследование, 426
отложенная загрузка, 513
поддержка клавиатуры, 428
проблемавидимости и отображения, 445
djConfig, массив, 56
afterOnLoad, параметр, 56
baseUrl, параметр, 56, 86
cacheBust, параметр, 57
debugAtAllCosts, параметр, 57
dojoBlankHtmlUrl, параметр, 57
dojoIframeHistoryUrl, параметр, 57
enableMozDomContentLoaded, параметр, 58
extraLocale, параметр, 58
isDebug, параметр, 58
libraryScriptUri, параметр, 58
locale, параметр, 58
modulePaths, параметр, 58, 90
parseOnLoad, ключ, 222

parseOnLoad, параметр, 59
require, параметр, 59
useXDomain, параметр, 59
xdWaitSeconds, параметр, 59
настройка платформы, 55
djConfig, массив параметров
parseOnLoad, ключ, 331
и парсеры, 331
djConfig, массив с параметрами конфигурации
проверка переводов, 196
DOCTYPE, тег (HTML), 324
document, объект
getElementsByClass, функция, 167
getElementsByClassName, функция, 167
getElementsByTagName, функция, 166, 170
document, объект (JavaScript)
getElementById, функция, 63, 521
DOH (Dojo Objective Harness), платформа
описание, 504
тестирование в броузере, 510
doh, модуль
assertEqual, функция, 505, 506
assertFalse, функция, 505, 506
assertTrue, функция, 505, 506
f, функция, 507
is, функция, 507
pause, функция, 507
register, функция, 506, 509
registerGroup, функция, 507, 509
registerTest, функция, 506
registerTestNs, функция, 506
registerTests, функция, 506
run, функция, 507
runGroup, функция, 507
t, функция, 507
togglePaused, функция, 507
Dojo
добавление в страницы, 61
исследование с помощью Firebug, 60
как получить, 42
легковесный сервер ответов, 47
настройки системы безопасности броузера, 47
обзор архитектуры, 37
отладка с помощью Firebug, 46
подготовка к работе, 42
самонастройка, 38, 52
терминология, 49
dojo,hitch, функция, 137
dojo.addClass, функция, 101, 180
dojo.addOnLoad, функция, 54, 249
анимация свойств CSS, 231
и парсеры, 204
и функция getLocalization, 196
комбинирование анимационных эффектов, 247
координата Z, 210
объединение анимационных эффектов в цепочку, 246
параметры, 53
прикладные диджиты, 454
простые эффекты растворения и проявления, 229
эффект сворачивания узла, 244
dojo.addOnLoad, функция
парсеры, 331
dojo.addOnLoad, функция
события сброса, 222
dojo.addOnUnload, функция, 56
dojo.anim, функция, 236
dojo.animateProperty, функция, 240
beforeBegin, событие, 237
onAnimate, метод, 241
onAnimate, событие, 238
onBegin, событие, 238
onEnd, событие, 238
onPause, событие, 238
onPlay, событие, 238
onStop, событие, 238
параметры, 240
dojo.attr, функция, 101
dojo.back, модуль
addToHistory, функция, 109
init, функция, 109
setInitialState, функция, 109
dojo.behavior, модуль
add, метод, 187, 188, 190
apply, метод, 187, 190
dojo.body, функция, 95
dojo.boxModel, атрибут, 104
dojo.byId, функция, 63, 65, 72, 328
сравнение с функцией dijit.byId, 328
dojo.clone, функция, 94
dojo.colors, модуль, 255
dojo.connect, функция, 65, 183
использование с замыканиями, 120
события от мыши, 113
события сброса, 219

события, возникающие при перетаскивании, 207, 209
dojo.connectPublisher, функция, 126
dojo.contentBox, функция, 104, 105
dojo.cookie, функция, 108
dojo.cookie.isSupported, функция, 108
dojo.coord, функция, 181
dojo.coords, функция, 105
dojo.currency, модуль
format, функция, 201
parse, функция, 201
и диджит CurrencyTextBox, 400
параметры форматирования, 200
фунции форматирования, 201
dojo.data, модуль
Identity, интерфейс, 263, 268
ItemFileReadStore, поддержка, 272
ItemFileWriteStore, поддержка, 272, 280
Notification, интерфейс, 263, 271
Read, интерфейс, 263
Write, интерфейс, 263, 269
десериализация типов данных, 288
и диджит Tree, 469, 474
обзор интерфейсов, 262
отклик на события, 271
поддержка ItemFileReadStore, 403
сериализация типов данных, 288
dojo.date, модуль
add, функция, 198
compare, функция, 198
datePattern, атрибут, 393
difference, функция, 198
formatLength, атрибут, 393
getDaysInMonth, функция, 198
getTimezoneName, функция, 198
isLeapYear, функция, 198
timePattern, атрибут, 393
Григорианский календарь, 393
обзор, 197
dojo.declare
основной шаблон создания класса, 299
dojo.declare, функция
constructor, функция, 299, 353, 354
extend, функция, 90
mixin, функция, 90
postscript, функция, 299, 353
preamble, функция, 299, 353
имитация классов, 52, 297
поддержка Dijit, 317, 348
создание объектов функций, 52
dojo.DeferredList, сигнатура, 148
dojo.delegate, функция, 98
dojo.disconnect, функция, 66, 116
обработчики событий, 116
dojo.dnd, модуль
destroy, функция, 215
Selector, класс, 223
Source, класс, 215, 222
Target, класс, 215, 218
ограничение перемещений, 211
поддержка механизма перетащил и бросил, 478
простые перемещения, 203
dojo.doc, объект, 95
dojo.every, функция, 77, 80
dojo.extend, функция, 92, 294
dojo.fadeIn, функция, 227, 240, 247
dojo.fadeOut, функция, 227, 240, 247
dojo.filter, функция, 78, 80
dojo.fixEvent, функция, 114
dojo.forEach, функция, 77, 80
dojo.formToJson, функция, 149, 150
dojo.formToObject, функция, 149
dojo.formToQuery, функция, 149, 150
dojo.fx, модуль
chain, функция, 245
combine, функция, 245
slideTo, функция, 242, 247
Toggler, класс, 248
wipeIn, функция, 243
wipeOut, функция, 243
переходные функции, 230
поддержка анимации, 226, 240
dojo.fx.slideTo, функция, 242
dojo.global, объект, 95
dojo.hasAttr, функция, 101
dojo.hasClass, функция, 101
dojo.hitch, функция, 97
dojo.i18n, модуль, 193, 196
dojo.i18n.getLocalization, функция, 196
dojo.indexOf, функция, 76
dojo.io.frame.send, функция, 155
dojo.io.iframe.create, функция, 160
dojo.isAIR, свойство, 107
dojo.isAlien, функция, 73
dojo.isArray, функция, 73
dojo.isArrayLike, функция, 73
dojo.isDescendant, функция, 99
dojo.isFF, свойство, 107
dojo.isFunction, функция, 73

dojo.isIE, свойство, 107
dojo.isKhtml, свойство, 107
dojo.isMozilla, свойство, 107
dojo.isObject, функция, 73
dojo.isOpera, свойство, 107
dojo.isQuirks, свойство, 107
dojo.isSafari, свойство, 107
dojo.isString, функция, 73
dojo.js, файл
  модуль Base, 38
  проблемы сборки, 497
dojo.keys, объект, 115
dojo.lastIndexOf, функция, 76
dojo.locale, объект, 193
dojo.map, функция, 78, 80
dojo.marginBox, функция, 105, 213
dojo.mixin, функция, 91
  и функция dojo.extend, 92
  пример наследования, 295
dojo.number, модуль, 199
  и диджит NumberTextBox, 399
  параметры форматирования, 199
  фунции форматирования, 199
dojo.objectToQuery, функция, 150
dojo.partial, функция, 95
dojo.place, функция, 102, 181
dojo.provide, функция
  управление исходным программным кодом, 81
dojo.publish, функция, 123
dojo.query, функция, 67
  манипулирование узлами, 168
  поддержка парсеров, 333
dojo.queryToObject, функция, 150
dojo.rawXhrPost, функция, 133
dojo.rawXhrPut, функция, 133
dojo.registerModulePath, функция, 86
dojo.removeAttr, функция, 101
dojo.removeClass, функция, 101, 180
dojo.require, функция, 53
  dojo.fx, модуль, 247
  и виджеты форм, 411
  и прикладные диджиты, 452
  поддержак анимации, 240
  поддержка Dijit, 345
  поддержка анимации, 184
  поддержка парсеров, 331
  прикладные диджиты, 459
  управление исходным программным кодом, 81
dojo.rpc, модуль
  JsonpService, конструктор, 162
  JsonService, конструктор, 162
  RpcService, конструктор, 162
dojo.setObject, функция, 82
dojo.setSelectable, функция, 100
dojo.some, функция, 77, 80
dojo.stopEvent, функция, 120
dojo.string, модуль, 75
  pad, функция, 75
  substitute, функция, 75
  обзор, 75
  поддержка Dijit, 358
dojo.style, функция, 100, 101, 180
dojo.subscribe, функция
  организация взаимодействий по подписке, 123
  события сброса, 219
  события, возникающие при перетаскивании, 207, 209
dojo.toggleClass, функция, 101
dojo.toJson, функция, 285
dojo.trim, функция, 74
dojoType, атрибут
  поддержка Dijit, 318, 319
  поддержка парсеров, 204, 217, 333
  проверка правильности документа, 324
DojoX, библиотека, 40
  обзор, 529
  поддерживаемые проекты, 530
  примерное число функций, 530
dojo.xd.js, файл
  загрузка версии XDomain, 55, 63
  пример CDN, 45
dojo.xhr, функция, 133, 136
dojo.xhrDelete, функция, 133
dojo.xhrGet, функция, 133
  и межсайтовый скриптинг, 142, 143
  изменение предельного времени ожидания, 145
  пример, 136
dojo.xhrPost, функция, 133
dojo.xhrPut, функция, 133
DOM события
  onclick, 328, 329
  методы и события диджитов, 327
  обработчики событий, 116
  особенности объекта, описывающего поведение, 188
  пакетная обработка, 182
  распространение событий, 119

события от мыши, 113, 182
точки событий, 358
часто используемые свойства, 114
DOM узлы
behavior, модуль, 187
dojo.query, функция, 168
Nodelist, класс, 174
onclick, событие, 328, 329
диджиты, 327
изменение стилей, 100
переключение состояния, 248
поиск, 72
прикладные диджиты, 450
события от мыши, 113
DOM утилиты
appendChild, 99
removeChild, 99
изменение стилей узлов, 100
селективность текста, 99
DropDownButton, диджит, 451
iconClass, атрибут, 452
и диджит TooltipDialog, 451
и функция dojo.require, 411
обзор, 415
DTD (Document Type Definition – определение типа документа), 324

## E

Editor, диджит
close, функция, 485
contentDomFilters, атрибут, 485
contentDomPostFilters, атрибут, 485
contentDomPreFilters, атрибут, 485
contentPreFilters, атрибут, 485
execCommand, функция, 485
extraPlugins, атрибут, 484
focusOnLoad, атрибут, 484
getValue, функция, 484
height, атрибут, 484
inheritWidth, атрибут, 484
minHeight, атрибут, 484
onDisplayChanged, функция, 485
plugins, атрибут, 484
setValue, функция, 484
undo, функция, 484
обзор, 345, 482
расширяемая архитектура, 486
Event, объект
altKey, свойство, 115
bubbles, свойство, 114
cancelable, свойство, 114
clientX, свойство, 115
clientY, свойство, 115
ctrlKey, свойство, 115
currentTarget, свойство, 115
metaKey, свойство, 115
screenX, свойство, 115
screenY, свойство, 115
shiftKey, свойство, 115
target, свойство, 115
type, свойство, 115
extend, функция, 92

## F

filter, функция, 78
FilteringSelect, диджит
getDisplayedValue, метод, 406
getValue, метод, 406
labelAttr, атрибут, 407
labelFunc, обработчик события, 407
labelType, атрибут, 407
setDisplayedValue, метод, 406
setValue, метод, 406
обзор, 341, 406
Firebug Lite, 47, 523
Firebug, расширение (Firefox)
Console, вкладка, 521
console.log, функция, 46, 522
CSS, вкладка, 524
DOM, вкладка, 526
HTML, вкладка, 524
Inspect, кнопка, 520
Layout, вкладка, 524
Net, вкладка, 504, 526
Profile, кнопка, 522
Script, вкладка, 526
Style, вкладка, 524
вопросы производительности, 514
где загрузить, 47
исследование Dojo, 60
настройка, 517
поддержка отладки, 46
установка, 516
Firefox, броузер
getElementsByClassName, функция, 167
где загрузить, 47
и завершающие запятые, 303
поддержка RGBA, 253
поддержка SpiderMonkey, 492

Flickr, источник данных, 153
forEach, функция, 77
ForestStoreModel, класс
  onAddToRoot, функция, 477
  onLeaveRoot, функция, 477
  rootId, атрибут, 473
  rootLabel, атрибут, 473
  обзор, 469, 472, 477
Form, диджит
  getValues, метод, 424
  isValid, метод, 424
  onSubmit, точка расширения, 424
  reset, метод, 424
  setValues, метод, 424
  submit, метод, 424
  validate, метод, 424
  обзор, 340, 423
form, тег (HTML)
  action, атрибут, 379
  disabled, атрибут, 381
  enctype, атрибут, 379
  method, атрибут, 379
  name, атрибут, 379
  onsubmit, атрибут, 379
  tabIndex, атрибут, 321
  tabindex, атрибут, 379, 381
  элементы управления, 378
_FormWidget, класс, 381
  alt, атрибут, 381
  disabled, атрибут, 381
  focus, метод, 382
  forWaiValuenow, точка расширения, 383
  intermediateChanges, атрибут, 382
  isFocusable, метод, 382
  name, атрибут, 381
  onChange, точка расширения, 383
  readOnly, атрибут, 382
  setAttribute, метод, 382
  tabIndex, атрибут, 381
  type, атрибут, 381
  value, атрибут, 381
  и диджит Button, 409
  и диджит Textarea, 408

## G

Gecko, механизм отображения, 107
getViewport, 346
gzip, алгоритм сжатия, 501, 513

## H

HelloWorld, диджит
  Declaration, инструмент, 372
  внедрение шаблона, 366
  изменение шаблона), 364
  обработка событий в диджитах, 369
  первая попытка, 361
  передача параметров, 367
Helma, инструмент, 503
HorizontalRule, диджит, 418, 422
  container, атрибут, 422
  count, атрибут, 422
  ruleStyle, атрибут, 422
HorizontalRuleLabel, диджит, 422
  constraints, атрибут, 423
  getLabels, метод, 423
  labels, атрибут, 422
  labelStyle, атрибут, 422
  maximum, атрибут, 423
  minimum, атрибут, 422
  numericMargin, атрибут, 422
HorizontalRuleLabels, диджит, 418
HorizontalSlider, диджит
  clickSelect, атрибут, 422
  container, атрибут, 419
  decrement, функция, 422
  discreteValues, атрибут, 421
  increment, функция, 422
  maximum, атрибут, 421
  minimum, атрибут, 421
  pageIncrement, атрибут, 422
  showButtons, атрибут, 421
  slideDuration, атрибут, 422
  обзор, 418
HSL, формат представления цвета, 255
HSLA, формат представления цвета, 255
HTML
  HelloWorld, диджит, 361
  UTF-8, кодировка символов, 401
  обеспечение доступности, 320
  поддержка форм, 149, 376
  проверка DOCTYPE, 324
  статическое содержимое, 513
HTTP DELETE, метод, 131, 136
HTTP GET, метод, 131, 136, 155
HTTP POST, метод, 131, 136, 155
HTTP PUT, метод, 131, 136

## I

Identity, интерфейс
fetchItemByIdentity, функция, 268, 276
getFeatures, функция, 268
getIdentity, функция, 268
getIdentityAttributes, функция, 268
ItemFileReadStore, поддержка, 272
ItemFileWriteStore, поддержка, 272, 280
обзор, 268
IEC (International Electrotechnical Commission – международная электротехническая комиссия), 402
IFRAME, передача данных, 154
IMG, тег (HTML), 320
index, ключевое слово, 80, 179
InlineEditBox, диджит
autoSave, атрибут, 467
buttonCancel, атрибут, 467
buttonSave, атрибут, 467
cancel, функция, 468
editing, атрибут, 467
editor, атрибут, 467
editorParams, атрибут, 467
enableSave, функция, 468
noValueIndicator, атрибут, 468
onChange, функция, 468
renderAsHtml, атрибут, 467
save, функция, 468
setDisabled, функция, 468
setValue, функция, 468
value, атрибут, 467
width, атрибут, 467
обзор, 344, 466
input, элемент
поддержка текстовых полей ввода, 383
input, тег (HTML)
и тег script, 412
поддержка кнопок, 412
Internet Explorer, броузер
document.getElementById, функция, 64
обслуживание кнопки «Назад», 110
ограничение на число соединений, 514
проблемы доступности, 320
режим разработки (design mode), 482
тестирование Dojo, 46
ISO (International Organization for Standardization – международная организация по стандартизации), 402
ISO-4217, стандарт, 400
item, ключевое слово, 80, 179
ItemFileReadStore
_type, атрибут, 288
_value, атрибут, 288
и диджит ComboBox, 403
функциональные возможности, 272
ItemFileReadStore, механизм
и диджит Tree, 470
обзор, 264
ItemFileWriteStore
_saveCustom, 285
serialize, функция, 289
ItemFileWriteStore, функциональные возможности, 280

## J

jar файлы, 493
Java, язык программирования, 291
JavaScript
clone, функция, 94
execCommand, функция, 482
extend, функция, 92
HelloWorld, диджит, 363
prototype, свойство, 51
setInterval, функция, 454
setTimeout, функция, 139, 455
асинхронные запросы, 139
и Java, 291
наследование, поддержка, 293
определение методов в разметке, 338
определение типа, 73
организация взаимодействий по подписке, 123
поддержка ShrinkSafe, 501
поддержка делегирования, 98
управление cookie, 108
управление исходным программным кодом, 81
утилиты для работы со строками, 74
JSON (JavaScript Object Notetion – представление объектов JavaScript
обзор, 130
JSONP (JSON with Padding – JSON с дополнением)
межсайтовый скриптинг, 142
обзор, 150

управление функциями обратного вызова, 152
JsonpService, конструктор, 162

## K

Konfabulator, 492

## L

LayoutContainer, диджит, 433

## M

Make, инструмент сборки, 493
map, функция, 78
MappedTextBox, диджит
и диджит FilteringSelect, 406
обзор, 390
Menu, диджит
addChild, функция, 463
bindDomNode, функция, 463
contextMenuForWindow, атрибут, 463
getChildren, функция, 463
onCancel, функция, 464
onClick, функция, 463
onExecute, функция, 464
onItemHover, функция, 463
onItemUnhover, функция, 463
parentMenu, атрибут, 463
popupDelay, атрибут, 463
removeChild, функция, 463
targetNodeIds, атрибут, 463
unBindDomNode, функция, 463
обзор, 343, 459
MenuItem, диджит
disabled, атрибут, 464
iconClass, атрибут, 464
label, атрибут, 464
onClick, функция, 464
setDisabled, функция, 464
обзор, 416, 459
meta, тег (HTML), 401
Midas Specification, 482
mixin, функция, 91
MochiKit, веб-сайт, 139
Moveable, конструктор
ограничение перемещений, 211
простые перемещения, 203
Mover, объект, 206
Mozilla, броузер
Midas Specification, 482
Rhino, интерпретатор JavaScript, 492, 503
SpiderMonkey, интерпретатор JavaScript, 492
MultiSelect, диджит
addSelected, метод, 407
getSelected, метод, 407
invertSelection, метод, 407
setValue, метод, 407
size, атрибут, 407
обзор, 342, 407

## N

name, атрибут, 484
new, оператор, 91
функции-конструкторы, 51
Nihilo, тема (Dijit), 326
nls, каталог, 193, 194
NodeList, класс
addClass, метод, 175, 180
addContent, метод, 175, 181
adopt, метод, 176, 182
anim, функция, 185
animateProperties, функция, 185
concat, метод, 175
connect, метод, 176, 183
coords, метод, 176, 181
every, метод, 174
fadeIn, функция, 184
fadeOut, функция, 184
filter, метод, 175, 179
forEach, метод, 175, 183
indexOf, метод, 174
instantiate, метод, 176
lastIndexOf, метод, 174
map, метод, 175
orphan, метод, 176, 182
place, метод, 175
removeClass, метод, 175, 180
slice, метод, 175
slideTo, функция, 185
some, метод, 175
splice, метод, 175
style, метод, 175, 180
wipeIn, функция, 184
wipeOut, функция, 184
передача результатов по цепочке, 177
создание расширений, 185

Notification, интерфейс
getFeatures, функция, 271
ItemFileWriteStore, поддержка, 280, 286
onDelete, функция, 272
onNew, функция, 271
onSet, функция, 271
и диджит ComboBox, 405
обзор, 271
NumberSpinner диджит
определение методов в разметке, 338
NumberSpinner, диджит
defaultTimeout, атрибут, 400
largeDelta, атрибут, 400
max, ограничение, 400
min, ограничение, 400
smallDelta, атрибут, 400
timeoutChangeRate, атрибут, 400
обзор, 341, 399
создание в разметке, 334
создание программным способом, 335
NumberTextBox, диджит
max, ограничение, 399
min, ограничение, 399
pattern, ограничение, 399
places, ограничение, 399
type, ограничение, 399
наследование, 399
обзор, 399

## O

Object, тип
prototype, свойство, 307
наследование, 51, 304
onAnimate, событие, 238
onBegin, событие, 238
onblur, событие, 183
onChange, событие, 414
onClick, событие, 183
dojo.connect, функция, 114
поддержка диджитами, 328, 329
onDndCancel, событие, 221
onDndDrop, событие, 221
onDndSourceOver, событие, 220
onDndStart, событие, 220
onEnd, событие, 238, 245
onFirstMove, событие, 209
onFocus, событие, 183
onKeydown, событие, 114, 183
onKeypress, событие, 114, 183
onKeyup, событие, 114, 183
onMousedown, событие, 114, 183, 224
onMouseenter, событие, 114, 183
onMouseleave, событие, 114, 183
onMousemove, событие, 114, 183, 224
onMouseout, событие, 114, 183, 370
onMouseover, событие, 114, 183, 370
onMouseup, событие, 114, 183, 224
onMove, событие, 210
onMoved, событие, 210
onMoveStart, событие, 209
onMoveStop, событие, 209
onMoving, событие, 210
onOutEvent, событие, 224
onOverEvent, событие, 224
onPause, событие, 238
onPlay, событие, 238
onStop, событие, 238
onSubmit, событие, 379
OpenAjax Alliance, консорциум производителей, 83, 164
OpenAjax Hub, 123, 164

## P

parseOnLoad:true, директива, 204
PopupMenuItem, диджит
disabled, атрибут, 464
iconClass, атрибут, 464
label, атрибут, 464
onClick, функция, 464
setDisabled, функция, 464
обзор, 459
ProgressBar, диджит
indeterminate, атрибут, 455
maximum, атрибут, 455
onChange, точка расширения, 455
places, атрибут, 455
progress, атрибут, 455
update, функция, 455
обзор, 344, 453
отображение, 320
Python, язык программирования, 47, 303

## R

RadioButton, диджит
обзор, 414
RangeBoundTextBox, диджит
обзор, 390

Read, интерфейс
close, функция, 267
containsValue, функция, 265
fetch, функция, 265
getAttributes, функция, 264
getFeatures, функция, 267
getLabel, функция, 267
getLabelAttributes, функция, 267
getValue, функция, 264, 277, 284
getValues, функция, 264, 284
hasAttribute, функция, 264
isItem, функция, 265
isItemLoaded, функция, 265
ItemFileReadStore, поддержка, 272
ItemFileWriteStore, поддержка, 280
loadItem, функция, 265
и диджит ComboBox, 405
обзор, 264
readings.js, файлы, 194, 196
README, файлы
поддержка CherryPy, 47
требования к проектам в DojoX, 529
RFC 3066, 193
RGB, формат представления цвета, 251
RGBA, формат представления цвета, 253
Rhino, интерпретатор JavaScript (Mozilla), 492, 503
поддержка тестирования, 505
RPC (Remote Procedure Call – вызов удаленных процедур), 161
RpcService, конструктор, 162
runner.js, файл, 507

## S

script, тег (HTML)
и тег input, 412
парсинг виджетов, 333
поддержка JSONP, 150
поддержка массива djConfig, 56
профили сборки, 500
Section 508, поправка к закону США о реабилитации, 319
select, элемент (HTML), 401, 407
Selector, интерфейс класса
getSelectedNodes, метод, 223
Selector, интерфейс класса deleteSelectedNodes, метод, 223
Selector, интерфейс класса destroy, метод, 223
Selector, интерфейс класса insertNodes, метод, 223
Selector, интерфейс класса onMouseDown, событие, 224
Selector, интерфейс класса onMouseMove, событие, 224
Selector, интерфейс класса onMouseUp, событие, 224
Selector, интерфейс класса onOutEvent, событие, 224
Selector, интерфейс класса onOverEvent, событие, 224
Selector, интерфейс класса selectAll, метод, 223
Selector, интерфейс класса selectNone, метод, 223
Selector, класс, 223
Shape, класс, 293, 304
ShrinkSafe
и Rhino, интерпретатор JavaScript, 493
оптимизация сборок, 501
SimpleTextarea, диджит
обзор, 341, 408
Slider, диджит
обзор, 341
SMD (Simple Method Description – простое описание методов), 161
Soria, тема (Dijit), 326
Source, класс, 222
сброс объектов, 215, 222
собственные аватары, 218
SpiderMonkey, интерпретатор JavaScript, 492
SplitContainer, диджит, 433
StackContainer, диджит, 342
closeable, атрибут, 440
closeChild, функция, 440
doLayout, атрибут, 440
onClose, точка расширения, 440
selectChild, функция, 440
selected, атрибут, 440
selectedChildWidget, атрибут, 440
title, атрибут, 440
обзор, 439
standard.profile.js, файл, 500
Subversion, репозиторий, 493

## T

TabContainer, диджит, 342, 429
closeable, атрибут, 443
onClose, точка расширения, 443
tabPosition, атрибут, 443
title, атрибут, 443
и ContentPane, 429
обзор, 441
Target, класс, 215, 218
_Templated, класс, 355, 358
buildRendering, метод, 358, 359
containerNode, свойство, 360
dojoAttachPoint, атрибут, 358
templatePath, свойство, 359, 364
templateString, свойство, 359, 364, 366
widgetsInTemplate, свойство, 359
Textarea, диджит
getValue, метод, 408
onChange, точка расширения, 408
setValue, метод, 408
и диджит InlineEditBox, 466
обзор, 341, 408
textarea, элемент (HTML), 158, 408
TextBox, диджит
format, точка расширения, 384
getValue, метод, 385
lowercase, атрибут, 384
maxLength, атрибут, 384
parse, точка раширения, 384
propercase, атрибут, 384
setValue, метод, 385
trim, атрибут, 384
uppercase, атрибут, 384
и диджит InlineEditBox, 466
наследование, 398
обзор, 342, 384
this, ключевое слово, 338
TimeTextBox, диджит
clickableIncrement, атрибут, 394
formatLength, атрибут, 395
getDisplayedValue, метод, 396, 397
locale, атрибут, 395
selector, атрибут, 395
serialize, точка расширения, 397
setDisplayedValue, метод, 396, 397
timePattern, атрибут, 395
toString, метод, 397
visibleIncrement, атрибут, 394
visibleRange, атрибут, 395
Григорианский календарь, 393
обзор, 391
TitlePane, диджит
duration, атрибут, 465
open, атрибут, 465
setContent, функция, 465
setTitle, функция, 465
title, атрибут, 465
toggle, функция, 465
обзор, 344, 464
ToggleButton, диджит
checked, атрибут, 410
onChange, точка расширения, 410
setAttribute, метод, 410
и функция dojo.require, 411
обзор, 410
Toggler, класс, 248
Toolbar, диджит
addChild, метод, 459
getChildren, метод, 459
removeChild, метод, 459
обзор, 343, 457
Tooltip, диджит
connectId, атрибут, 448
label, атрибут, 449
showDelay, атрибут, 449
обзор, 344, 447
TooltipDialog, диджит
обзор, 344, 451
Tree, диджит
checkAcceptance, метод, 479
checkItemAcceptance, метод, 479
childrenAttr, атрибут, 475
dndController, атрибут, 478
ForestStoreModel, класс, 469
itemCreator, метод, 479
model, атрибут, 475
onClick, точка расширения, 474, 475
onDndCancel, метод, 479
onDndDrop, метод, 479
openOnClick, атрибут, 475
persist, атрибут, 475
query, атрибут, 475
showRoot, атрибут, 475
_TreeNode, класс, 469
TreeStoreModel, класс, 469
доступность, 468
обзор, 345, 468
поддержка операции «перетащил и бросил», 478
простое дерево, пример, 469

простой лес, пример, 471
реакция на события от мыши, 474
_TreeNode, класс, 469
TreeStoreModel, класс
destroyRecursive, функция, 477
getChildren, функция, 476
getIdentity, функция, 476
getLabel, функция, 476
getRoot, функция, 476
mayHaveChildren, функция, 476
newItem, функция, 476
onChange, функция, 477
onChildrenChange, функция, 477
pasteItem, функция, 476
обзор, 469, 470, 476
Tundra, тема (Dijit), 325, 380
Twisted, веб-сайт, 139
typeof, оператор, 74

## U

UCS (Universal Character Set – универсальный набор символов), стандарт, 402
UTF-8, кодировка символов, 401
Util, библиотека, 41
инструменты сборки, 491
повышение производительности, 197
поддержка тестирования, 491
пример HelloWorld, 366

## V

ValidationTextBox, диджит
constraints, атрибут, 388
displayMessage, точка расширения, 389
invalidMessage, атрибут, 388
isValid, метод, 388
promtMessage, атрибут, 388
regExp, атрибут, 388
regExpGen, атрибут, 388
required, атрибут, 388
tooltipPosition, атрибут, 388
validator, точка расширения, 389
и диджит ComboBox, 401
наследование, 398, 401
обзор, 387
VerticalRule, диджит, 421, 422
container, атрибут, 422
count, атрибут, 422
ruleStyle, атрибут, 422
VerticalRuleLabel, диджит, 422
constraints, атрибут, 423
getLabels, метод, 423
labels, атрибут, 422
labelStyle, атрибут, 422
maximum, атрибут, 423
minimum, атрибут, 422
numericMargin, атрибут, 422
VerticalRuleLabels, диджит, 421
VerticalSlider, диджит
clickSelect, атрибут, 422
decrement, функция, 422
discreteValues, атрибут, 421
increment, функция, 422
maximum, атрибут, 421
minimum, атрибут, 421
pageIncrement, атрибут, 422
showButtons, атрибут, 421
slideDuration, атрибут, 422
обзор, 420

## W

W3C
WAI-ARIA, 321
нормализация событий, 114
WAI-ARIA, 321
WebKit, веб-сайт, 167
_Widget, класс, 52
buildRendering, метод, 354, 364
create, метод, 428
destroyRecursive, метод, 356
domNode, свойство, 357, 381
id, свойство, 357
lang, свойство, 357
postCreate, метод, 355
postMixInProperties, метод, 354, 365
startup, метод, 355
uninitialize, метод, 356
и диджиты компоновки, 426
и наследование, 348
методы управления жизненным циклом, 351
window, объект, 94
Write, интерфейс
deleteItem, функция, 270
getFeatures, функция, 269
hasAttribute, функция, 281
isDirty, функция, 270, 281
ItemFileWriteStore, поддержка, 280
newItem, функция, 269

revert, функция, 270
save, функция, 270, 282
setValue, функция, 270, 280
setValues, функция, 270
unsetAttribute, функция, 270
обзор, 269

## X

XDomain
преимущества, 514
XMLHttpRequest (XHR)
многоцелевая функция выполнения запросов, 136
XMLHttpRequest (XHR), объект, 129
асинхронные запросы, 139
настройка платформы, 55
поддержка форм, 377
политика одного источника, 142, 150
примеры применения функций XHR, 134

## Y

Yahoo!, 492

## Z

Zulu, универсальное время, 393

## А

аватары, 218
анимация
вычисление цвета, 251
переключение состояния узла, 248
поддержка в NodeList, 184
поддержка в модуле dojo.fx, 226
поддержка операции перетащил и бросил, 249
программное управление, 236
произвольных свойств CSS, 230
простые эффекты растворения и проявления, 227
создание цепочек и комбинирование эффектов, 245
эффект свертывания, 243
анонимные функции, 122
аргументы функции в виде строк, 80
асинхронные взаимодействия, структура данных Deferred, 139
ассоциативные массивы, 51
атрибуты
манипулирование, 101
определение, 262
поддержка Dijit, 323

## Б

безопасность
межсайтовый скриптинг, 142
настройка броузера, 47
блок рамки, 104
блок содержимого, 104
букмарклеты (bookmarklet), 61

## В

веб-броузеры
и завершающие запятые, 303
настройка системы безопасности броузера для запуска Dojo на локальной системе, 47
определение типа, 73
поддержка редактирования, 482
проблемы доступности, 320
сжатие gzip, 501
тестирование, 510
веб-порталы, 123
взаимодействие с сервером
асинхронные запросы, 139
взаимодействие со сбрасываемыми объектами, 223
взаимодействия по подписке, 123
взаимодействия с сервером
Deferred, примеры, 143
межсайтовый скриптинг, 150
поддержка с помощью модуля Core IO, 152
утилиты для работы с формами и HTTP, 149
виджеты, 52
организация взаимодействий по подписке, 123
парсинг, 204, 330
вопросы производительности, 512
врезки схема, 435
вызов удаленных процедур, 161

## Г

генераторы псевдослучайных чисел, 84
гонка за ресурсами, 54
Григорианский календарь, 393

грубое определение типа, 74, 295
грязные (dirty) элементы, 281

## Д

делегирование, 98, 295
деревья, 469
десериализация
содержимого, 485
типов данных, 288
диджиты (виджеты Dojo), 52, 350
byId, функция, 328, 346, 414
byNode, функция, 346
class, атрибут, 325
constraints, атрибут, 336
_Contained, класс, 371, 426
_Container, класс, 371, 426, 462
Declaration, инструмент, 372
dir, атрибут, 325
dojoType, атрибут, 332
domNode, атрибут, 328, 330
getEnclosingWidget, функция, 346
getViewport, функция, 346
HelloWorld, диджит, 360
id, атрибут, 325
jsId, атрибут, 334
lang, атрибут, 325
onClick, событие, 328, 329
registry, свойство, 346
style, атрибут, 325
_Templated, класс, 355, 358
title, атрибут, 325
диджиты приложения, 343
и CSS, 350
и HTML, 350
и JavaScript, 350
и узлы DOM, 327
методы и события DOM, 327
методы управления жизненным циклом, 351
общие атрибуты, 325
объект Function, 335
определение, 327, 348
особенности HTML, 339
парсинг, 330
соединения в тексте разметки, 123
диджиты приложения, 343
динамическое связывание, 98
доступность
и диджит Tree, 468
поддержка Dijit, 319, 380

## З

заголовка схема, 435
задержка
и версия XDomain, 514
и нарезка изображений, 460
замыкания
анонимные функции, 122
пример работы с координатой z, 210
реализация при использовании функции forEach, 78
с функцией dojo.connect, 120
запрос дочерних элементов, 278
закон США о реабилитации (Rehabilitation Act) от 1973 года, 319
значения атрибутов, 262, 279
золоте сечение, 84

## И

иерархии в формате JSON, 273
импортирование модулей и ресурсов, 81
инкапсуляция, 317
инструментальный набор
определение, 49
инструменты сборки, 513
вопросы производительности, 512
загрузка, 492
интернационализация
Dijit, библиотека, 192
модулей, 193
исходный программный код
управление с помощью модулей, 81

## К

карринг, прием, 96
клавиатуры события
поддержка в NodeList, 182
классы
имитация, 52, 296
ограничения JavaScript, 52
определение, 52
основной шаблон создания, 299
парсинг виджетов, 331
кнопка Назад, 109
кодировка символов, 402
комбинатор дочернего узла, 172
комментарии, удаление, 492, 501
константы, события от клавиатуры, 115
конструкторы
определение, 51

контекст
манипулирование контекстом объекта, 94
координата Z, 210

## Л

Леонардо из Пизы, 84
лес, 469, 471

## М

межсайтовый скриптинг (cross-site scripting), 142, 150
методы
класса NodeList, 174
определение, 51
расширение прототипов объектов, 92
унаследованные, 305
минимизация, 501
модули
импортирование, 81
интернационализация, 193
определение, 49, 81
пример, 84
пример модуля волшебного джинна, 87
управление исходным программным кодом, 81
модули расширения (Editor)
AlwaysShowToolbar, 488
EnterKeyHandling, 488
FontChoice, 489
LinkDialog, 489
TextColor, 489
обзор, 486
определение, 486
модули расширения (Editor)ToggleDir, 489
монотонные функции, 234
мыши события
поддержка в NodeList, 182

## Н

наблюдение за состоянием, 172
нарезка изображений, 460
наследование
Object, тип, 51
в JavaScript, 293
диджиты компоновки, 426
диджиты форм, 380
и класс _Widget, 348
имитация классов, 296
ошибки при использовании механизма наследования на базе прототипов, 304
примеры, 300
независимость, 317
непрозрачность, 254
неявное отображение типов, 288

## О

обработка массивов, 75
аргументы в стиле функции в виде строк, 179
аргументы функции в виде строк, 80
обход элементов, 77
поиск местоположения элементов, 76
преобразование элементов, 78
проверка элементов на соответствие условию, 77
обработчики событий
распространение событий, 119
соединения в тексте разметки, 123
объекты
клонирование, 94
манипулирование контекстом, 94
определение, 51
определение типа, 73
грубое определение типа, 74
определение типа броузера, 107
осиротевшие (orphaned) элементы, 182
откат (rollback), 282
отладка, поддержка Firebug, 46, 526
отложенная загрузка, 441, 498, 513

## П

параметры
HelloWorld, пример, 367
передача параметров по частям, 95
парсеры, 330
и атрибут dojoType, 204, 217
обзор, 204
первоклассные объекты, 50
передача репрезентативного состояния (Representational State Transfer, REST), 131
переключение состояния узла, 248
переменные
определение типа, 73
сокращение имен, 492

перетащил и бросил
взаимодействие со сбрасываемыми объектами, 223
зона сброса, 218
координата Z, 210
обзор, 202
обзор операции сброса, 215
ограничение перемещений, 211
поддержка анимации, 249
простые перемещения, 203
собственные аватары, 218
события сброса, 219
события, возникающие при перетаскивании, 207, 209
портлеты, 123
привязки (bundle), 194
прикладные диджиты
ColorPalette, диджит, 344, 456
Dialog, диджит, 343, 449
Editor, диджит, 345, 482
InlineEditBox, диджит, 344, 466
Menu, диджит, 343, 459
MenuItem, диджит, 416, 459
PopupMenuItem, диджит, 459
ProgressBar, 320
ProgressBar, диджит, 344, 453
TitlePane, диджит, 344, 464
Toolbar, диджит, 343, 457
Tooltip, диджит, 344, 447
TooltipDialog, диджит, 344, 451
Tree, диджит, 345, 468
пробельные символы
усечение, 74
проблемы доступности, 320
Firefox, броузер, 320
проверка
DOCTYPE, 324
содержимого формы, 379
элементов на соответствие условию, 77
программное управление анимацией, 236
прозрачность, 254
простое описание методов (Simple Method Description, SMD), 161
пространства имен
ввод в действие, 55
определение, 50
прототипов цепочки, 307
профили сборки
ShrinkSafe, оптимизация, 501
standard.profile.js, файл, 500
настройка, 497
определение, 496
создание собственных модулей, 499

## Р

распространение событий, 119
режим обратной совместимости, 107
режим разработки (design mode), 482
ресурсы
импортирование, 81
определение, 50, 81
Римский календарь, 393
Ричардсон, Леонард (Leonard Richardson), 131
Руби, Сэм (Sam Ruby), 131

## С

самодостаточность, 317
самонастройка
djConfig, массив, 55
dojo.addOnLoad, функция, 54
обзор, 52
свойства
добавление к объектам, 91
определение, 51
расширение прототипов объектов, 92
событий DOM, 114
упрощенный синтаксис, 235
селекторы CSS, 166
сериализация
содержимого, 486
типов данных, 288
сжатие
gzip, 501, 513
определение, 501
поддержка ShrinkSafe, 501
символы перевода строки, удаление, 492
системы пакетов, 50
скольжение узлов, 242
скриптинг
межсайтовый, 142, 150
слои, 496
события от клавиатуры
нормализация, 113
поддерживаемые константы, 115

события от мыши
и диджит Tree, 474
нормализация, 113
события DOM, 113
содержимое
взаимодействие, 485
десериализация, 485
сериализация, 486
статическое, 513
соединения
однократные, 121
поддержка Dijit, 123
установка в цикле, 121
создание цепочек и комбинирование эффектов, 245
составление цепочек
из функций обратного вызова, 139
специальные эффекты
переключение состояния узла, 248
пример операции перетащил и бросил, 250
свертывание, 243
скольжение, 242
создание цепочек и комбинирование эффектов, 245
специальный режим, 104
среднее гринвичское время (Greenwich Mean Time, GMT), 393
срок действия, cookie, 108
ссылки в формате JSON, 273
статическое содержимое, 513
сценарии сборки
action, параметр, 494
buildLayers, параметр, 496
copyTests, параметр, 494
cssImportIgnore, параметр, 495
cssOptimize, параметр, 493
internStrings, параметр, 494
layerOptimize, параметр, 495
loader, параметр, 495
localeList, параметр, 494
log, параметр, 495
optimize, параметр, 494
profile, параметр, 495, 498
releaseDir, параметр, 494
releaseName, параметр, 494
scopeDjConfig, параметр, 496
scopeMap, параметр, 494
symbol, параметр, 494
version, параметр, 495
xdDojoPath, параметр, 495
xdDojoScopeName, параметр, 495
xdScopeArgs, параметр, 493

## Т

тестирование
в броузерах, 46, 510
поддержка DOH, 504
поддержка в библиотеке Core, 507
тем, 326
тестовый контекст, 505
типы данных, сериализация и десериализация, 288
точечная нотация имен объектов, 82
точки расширения, 323

## У

универсальное скоординированное время (Coordinated Universal Time, UTC), 393
управление соединениями, 183
устройства чтения с экрана, 321, 380

## Ф

Фибоначчи, последовательность чисел, 84, 86
формы
обзор, 376
определение, 376
поддержка AJAX, 377
поддержка CherryPy, 378
поддержка HTML, 149, 324, 376
поддержка объекта XHR, 377
проверка, 324
проверка содержимого, 379
функции
constructor, 319
анонимные, 51, 122
конструкторы, 51
монотонные, 234
определение, 51
функции обработки ошибок
возвращаемые значения, 145
поддержка в DeferredList, 148
функции обратного вызова
возвращаемое значение, 134
возвращаемые значения, 145
и функция hitch, 137
поддержка JSONP, 151

поддержка в DeferredList, 148
составление цепочек, 139
функции-конструкторы
определение, 51, 293
поддержка Dijit, 319

## Ц

цвета
создание и смешивание, 251
цепочки
анимационных эффектов, 245
обработки списков NodeList, 177
прототипов, 307
цепочки прототипов, 98

## Э

элементы, 182
обход, 77
поиск местоположения, 76
преобразование, 78
проверка на соответствие условию, 77
элементы (dojo.data)
грязные (dirty), 281
доступ, 264
запрос дочерних элементов, 278
извлечение по идентификатору, 268, 276
извлечение по произвольному критерию, 277
изменение, 269, 280
определение, 262
создание, 269, 283
удаление, 269, 283
эффект свертывания, 243
эффект скольжения, 242

## Ю

Юлианский календарь, 393
Юникод (Unicode), стандарт, 402

По договору между издательством «Символ-Плюс» и Интернет-магазином «Books.Ru – Книги России» единственный легальный способ получения данного файла с книгой ISBN 978-5-93286-151-6, название «Dojo. Подробное руководство» – покупка в Интернет-магазине «Books.Ru – Книги России». Если Вы получили данный файл каким-либо другим образом, Вы нарушили международное законодательство и законодательство Российской Федерации об охране авторского права. Вам необходимо удалить данный файл, а также сообщить издательству «Символ-Плюс» (piracy@symbol.ru), где именно Вы получили данный файл.