Secrets of the JavaScript Ninja

John Resig

MANNING

Secrets of the JavaScript Ninja

JOHN RESIG
BEAR BIBEAULT

MANNING

Manning Publications Co.
20 Baldwin Road
PO Box 261
Shelter Island, NY 11964

Секреты JavaScript ниндзя

ДЖОН РЕЗИГ
БЕЭР БИБО

Москва • Санкт-Петербург • Киев
2013

К 32.973.26-018.2.75
Р34
К 681.3.07

Издательский дом "Вильямс"

Зав. редакцией *С.Н. Тригуб*

Перевод с английского и редакция *И.В. Берштейна*

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

Резиг, Джон, **Бибо**, Беэр.
Секреты JavaScript ниндзя : Пер. с англ. – М. : ООО "И.Д. Вильямс", 2013. – 416 с. : ил. – Парал. тит. англ.

ISBN 978-5-8459-1843-7 (рус.)

ББК 32.973.26-018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.
Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в любой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства Manning Publication, Co.

Authorized translation from the English language edition published by Manning Publications Co, Copyright © 2013. All rights reserved.
No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.
Russian language edition is published by Williams Publishing House according to the Agreement R&I Enterprises International, Copyright © 2013

Научно-популярное издание

Джон Резиг, Беэр Бибо

Секреты JavaScript ниндзя

Литературный редактор *И.А. Попова*
Верстка *М.А. Удалов*
Художественный редактор *В.Г. Павлютин*
Корректор *Л.А. Гордиенко*

Подписано в печать 23.07.2013. Формат 70x100/16.
Гарнитура Times. Печать офсетная.
Усл. печ. л. 33,5. Уч.-изд. л. 34,8.
Тираж 1500 экз. Заказ № 3708.

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

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

978-5-8459-1843-7 (рус.)
978-1-93-398869-6 (англ.)

© Издательский дом "Вильямс", 2013
© by Manning Publications Co., 2013

Оглавление

Часть I. Подготовка к обучению 23

Глава 1. Введение в искусство программирования на JavaScript 25

Глава 2. Вооружение средствами тестирования и отладки 35

Часть II. Обучение ученика 53

Глава 3. Функции как основа основ 55

Глава 4. Обращение с функциями 87

Глава 5. Сближение с замыканиями 117

Глава 6. Объектно-ориентированное программирование с помощью прототипов 151

Глава 7. Овладение регулярными выражениями 187

Глава 8. Укрощение потоков и таймеров 213

Часть III. Обучение кандидата в мастера 231

Глава 9. Вычисление кода во время выполнения 233

Глава 10. Операторы `with` 257

Глава 11. Стратегии разработки кросс-браузерного кода 271

Глава 12. Обращение с атрибутами, свойствами и CSS 295

Часть IV. Обучение мастера 331

Глава 13. Особенности обработки событий 333

Глава 14. Манипулирование моделью DOM 377

Глава 15. Механизмы CSS-селекторов 395

Предметный указатель 412

Содержание

Предисловие 13

Благодарности 15

Об этой книге 16
- Кому адресована книга 16
- Структура книги 17
- Условные обозначения, принятые в книге 19
- Загружаемый исходный код 19
- Как связаться с авторами 20
- Об иллюстрации на обложке книги 20

Об авторах 21

От издательства 22

Часть I. Подготовка к обучению 23

Глава 1. Введение в искусство программирования на JavaScript 25

Избранные библиотеки JavaScript 26
Общее представление о языке JavaScript 27
Соображения по поводу кросс-браузерной разработки 28
Передовые методики 31
- Передовая методика тестирования 31
- Передовая методика анализа производительности 32

Резюме 33

Глава 2. Вооружение средствами тестирования и отладки 35

Отладка кода 36
- Регистрация результатов 36
- Точки прерывания 38

Формирование тестов 39
Среды тестирования 42
- QUnit 44
- YUI Test 44
- JsUnit 44
- Новейшие среды блочного тестирования 45

Основы построения тестовых наборов 45
- Утверждение 45
- Группы тестов 46
- Асинхронное тестирование 48

Резюме 50

Часть II. Обучение ученика 53

Глава 3. Функции как основа основ 55

Главное отличие JavaScript как языка функционального программирования 56
- Особое значение функционального характера JavaScript 57
- Сортировка по алгоритму сравнения 62

Объявление функций 64
- Определение областей действия и функции 68

Вызов функций 72
- От аргументов к параметрам функций 73
- Вызов в виде функции 74
- Вызов в виде метода 75
- Вызов в виде конструктора 77
- Вызов с помощью методов `apply()` и `call()` 80

Резюме 84

Глава 4. Обращение с функциями 87

Анонимные функции 87
Рекурсия 90
- Рекурсия в именованных функциях 90
- Рекурсия в методах 92
- Проблема пропадающих ссылок 93
- Встраиваемые именованные функции 95
- Свойство `callee` 97

Особенности применения функций как объектов 98
- Сохранение функций 99
- Самозапоминающиеся функции 100
- Имитация методов обработки массивов 103

Списки аргументов переменной длины 104
- Предоставление переменного числа аргументов с помощью метода `apply()` 105
- Перегрузка функций 106

Проверка объектов на функции 114
Резюме 116

Глава 5. Сближение с замыканиями 117

Принцип действия замыканий 118
Применение замыканий на практике 122
Привязка контекста функций 127
Частичное применение функций 132
Переопределение поведения функции 135
Немедленно вызываемые функции 141
Резюме 149

Глава 6. Объектно-ориентированное программирование с помощью прототипов 151

Получение экземпляров объектов и прототипы 152
- Получение экземпляров объектов 152
- Типизация объектов через конструкторы 159

Вопросы производительности 300
Затруднения кросс-браузерного характера, возникающие при доступе к атрибутам 304
Расширение идентификатора или имени модели DOM 304
Нормализация URL 306
Атрибут стилевого оформления 308
Атрибут типа 308
Трудности определения индекса перехода по табуляции 309
Имена узлов 310
Трудности обращения с атрибутами стилевого оформления 310
Местонахождение стилей 311
Именование свойств стилевого оформления 313
Свойство стилевого оформления `float` 314
Преобразование значений, указываемых в пикселях 315
Указание размеров по высоте и ширине 316
Манипулирование непрозрачностью 321
Хождение по цветовому кругу 322
Извлечение вычисленных стилей 325
Резюме 330

Часть IV. Обучение мастера 331

Глава 13. Особенности обработки событий 333

Привязка и отвязка обработчиков событий 334
Объект типа `Event` 338
Управление обработкой событий 342
Централизованное хранение связанной информации 342
Управления обработчиками событий 346
Инициирование событий 355
Специальные события 357
Всплывание и делегирование 361
Делегирование событий родительскому элементу 362
Обходной прием для устранения отличий в браузерах 363
Событие готовности документа 372
Резюме 375

Глава 14. Манипулирование моделью DOM 377

Вставка HTML-разметки 378
Преобразование из формата HTML в DOM 379
Выполнение сценариев 385
Клонирование элементов разметки 387
Удаление элементов разметки 389
Текстовое содержимое 390
Установка текста 392
Получение текста 393
Резюме 393

Глава 15. Механизмы CSS-селекторов 395

Прикладной интерфейс Selectors API по стандарту W3C 397
Применение XPath для поиска элементов 400
Реализация чистой модели DOM 401
Синтаксический анализ селектора 404
Поиск элементов разметки 405
Фильтрация результатов поиска 406
Рекурсирование и объединение результатов 407
Восходящий механизм селекторов 408
Резюме 410

Предметный указатель 412

Предисловие

Приступая к написанию этой книги в начале 2008 года, я видел в этом насущную необходимость, поскольку в имевшейся тогда литературе отсутствовало подробное изложение самых важных составляющих языка программирования JavaScript (функций, замыканий и прототипов), а также приемов написания кросс-браузерного кода. К сожалению, ситуация с тех пор, как ни странно, не стала лучше.

Все больше сил и энергии вкладывается в разработку новых технологий, в том числе берущих свое начало от стандарта HTML5 или новых версий ECMAScript. Но нет никакого смысла осваивать новые технологии или применять самые современные библиотеки, не имея надлежащего представления об основных характеристиках языка JavaScript. Несмотря на самое светлое будущее разработки браузеров, главная задача в настоящее время — обеспечить работоспособность прикладного кода в большинстве браузеров и для большей части потенциальных пользователей.

Несмотря на то что эта книга писалась долго, она, к счастью, не утратила своей актуальности. Мой соавтор, Беэр Бибо, внес в нее существенные коррективы. Он постарался сделать так, чтобы материал книги еще долго оставался актуальным.

Долгое написание этой книги объясняется тем, что для материала последних ее глав, посвященных разработке кросс-браузерного кода, мне необходимо было приобрести известный опыт. Мои представления о разработке кросс-браузерного кода на практике опираются в основном на мою работу над библиотекой jQuery для JavaScript. Работая над материалом последних глав, посвященных разработке кросс-браузерного кода, я осознал, что большую часть базового кода библиотеки jQuery можно было бы написать иначе, оптимизировать и сделать ее способной управлять большим числом браузеров.

Вероятно, самые значительные изменения, внесенные в jQuery вследствие написания этой книги, связаны с полным пересмотром базового кода этой библиотеки: от организации пассивного прослушивания сети на уровне отдельных браузеров до обнаружения доступных средств. Благодаря этому применение библиотеки jQuery стало практически неограниченным, исключая необходимость учитывать, что в браузерах всегда будут присутствовать характерные программные ошибки или отсутствовать отдельные средства.

В результате этих изменений в jQuery предусмотрены многие усовершенствования, произведенные в браузерах за два прошедших года: выпуск браузера Chrome компанией Google; широкое распространение пользовательских агентов по мере роста популярности мобильных вычислений; острое соперничество среди компаний Mozilla, Google и Apple за повышение производительности их браузеров; а также значительные усовершенствования браузера Internet Explorer, на которые, наконец-то, решилась корпорация Microsoft. Теперь уже нельзя допускать, что один и тот же механизм визуализации (например, WebKit или Trident в Internet Explorer) будет всегда действовать одинаково. Коренные перемены происходят довольно быстро и распространяются среди постоянно растущего числа пользователей.

С помощью способов и приемов, описываемых в этой книге, кросс-браузерные возможности jQuery обеспечивают довольно прочную гарантию того, что код, написанный средствами jQuery, будет работать в наибольшем числе сред браузеров. Благодаря этой гарантии библиотека jQuery нашла широкое распространение за последние четыре года. По данным, опубликованным на веб-сайте `BuiltWith.com`, она теперь применяется на 58% из 10 тысяч самых широко посещаемых в Интернете веб-сайтов.

Средства JavaScript, в том числе вычисление кода, противоречивые операторы `with` и таймеры, относительно постоянны и продолжают использоваться самыми интересными способами. В настоящее время на основе JavaScript построен или скомпилирован целый ряд активно применяемых языков программирования, в том числе CoffeeScript и Processing.js. Но для того чтобы они действовали эффективно, требуется сложный синтаксический анализ языковых конструкций, вычисление кода и манипулирование областью действия. Несмотря на то что динамическое вычисление кода пользуется недоброй славой в силу его сложности и потенциальных нарушений безопасности, без него стало бы просто невозможным появление языка программирования CoffeeScript, который, в свою очередь, повлиял на составление спецификации языка ECMAScript.

Я лично пользуюсь всеми этими средствами до сих пор в своей работе в Академии Хана. Динамическое вычисление кода в браузере является весьма эффективным средством, позволяя создавать встраиваемые в браузеры среды программирования и внедрять такие необычные приемы, как внесение кода во время фактического выполнения. Все это возбуждает дополнительный интерес к изучению программирования на компьютере и открывает новые возможности, недоступные для традиционных средств обучения.

Будущее разработки браузеров остается весьма прочным и, главным образом, благодаря средствам, встроенным в JavaScript, и прикладным интерфейсам API самих браузеров. Имея основательное представление о самых важных составляющих языка JavaScript, а также желание писать код, способный работать во многих браузерах, вы будете в состоянии разрабатывать изящный, быстродействующий и повсеместно применяемый код.

ДЖОН РЕЗИГ

Благодарности

Количество людей, принявших участие в работе над этой книгой, способно многих удивить. Свой плодотворный вклад в книгу, которую вы держите в руках или читаете на экране в электронном виде, внесли многие одаренные люди.

Сотрудники издательства Manning Publications неутомимо потрудились, чтобы добиться такого качества издания, на которое надеялись авторы книги, за что мы им искренне благодарны. Ведь без них эта книга не увидела бы свет. Мы выражаем благодарность не только издателю Маржан Бэйс (Marjan Bace) и главному редактору Майку Стивенсу (Mike Stephens), но и следующим сотрудникам издательства: Джеффу Блейлу (Jeff Bleiel), Дугласу Паднику (Douglas Pudnick), Себастьяну Стирлингу (Sebastian Stirling), Адреа Каухер (Andrea Kaucher), Карен Тегтмайер (Karen Tegtmayer), Кэйти Теннат (Katie Tennant), Меган Йоки (Megan Yockey), Дотти Марсико (Dottie Marsico), Мэри Пирджиз (Mary Piergies), Энди Кэроллу (Andy Carroll), Мелоди Долаб (Melody Dolab), Тиффани Тейлор (Tiffany Taylor), Деннису Далиннику (Dennis Dalinnik), Гэбриэлу Добреску (Gabriel Dobrescu), а также Рону Томичу (Ron Tomich).

Невозможно выразить словами нашу признательность коллегам, рецензировавшим книгу и помогавшим нам довести ее до состояния полной готовности к печати: от вылавливания простых опечаток до исправления ошибок в терминах и коде и вплоть до организации книги по отдельным главам. Каждый из них внимательно просмотрел рукопись книги, прежде чем она была сдана в печать. За этот нелегкий труд мы хотели бы поблагодарить Алессандро Галло (Alessandro Gallo), Андре Робержа (André Roberge), Остина Кинга (Austin King), Остина Циглера (Austin Ziegler) Чэда Дэвиса (Chad Davis), Чарльза Е. Логстона (Charles E. Logston), Криса Грэя (Chris Gray), Кристофера Хаупта (Christopher Haupt), Крэйга Ланкастера (Craig Lancaster), Кэртиса Миллера (Curtis Miller), Даниэля Бретуа (Daniel Bretoi), Дэвида Веддера (David Vedder), Эрика Арвидссона (Erik Arvidsson), Гленна Стокола (Glenn Stokol), Грэга Дональда (Greg Donald) Джеймса Хэтуэя (James Hatheway), Джареда Хирша (Jared Hirsch), Джима Руза (Jim Roos), Джо Литтона (Joe Litton), Йохана Линка (Johannes Link), Джона Польсона (John Paulson), Джошуа Хейера (Joshua Heyer), Джулио Гвиджарро (Julio Guijarro), Курта Янга (Kurt Jung), Лоика Саймона (Lo c Simon), Нила Микса (Neil Mix), Роберта Хансона (Robert Hanson), Скотта Сойета (Scott Sauyet), Стюарта Каборна (Stuart Caborn), а также Тони Ниманна (Tony Niemann).

Особая благодарность выражается Валентину Креттазу (Valentin Crettaz), научному редактору книги. Помимо проверки каждого примера кода в нескольких средах, он внес неоценимый вклад в уточнение текста рукописи и обнаружение пропущенной информации с учетом последних изменений, внесенных в поддержку JavaScript и HTML5 в браузерах. Не меньшая благодарность выражается также Берту Бэйтсу (Bert Bates), давшему отзыв о книге и неоценимые предложения по поводу ее улучшения. Бесконечные часы общения с ним по Skype, безусловно, окупились сторицей.

Джон Резиг

Мне хотелось бы поблагодарить моих родителей за их постоянную моральную и материальную поддержку в течение многих лет. Они предоставили все необходимые ресурсы и средства, чтобы пробудить во мне интерес к программированию, и с тех пор постоянно поощряли мои занятия в данной области.

Беэр Бибо

За участие в работе над этой, уже пятой для меня книгой я хотел бы, как всегда, поблагодарить своих коллег, участников и организаторов форума JavaRanch (`www.javaranch.com`). Ведь без личного участия в этом форуме я бы никогда не начал писать книги, и поэтому я искренне благодарю Пола Уитона (Paul Wheaton) и Кэти Сьерра (Kathy Sierra) за добрые напутствия, а также моральную поддержку следующих коллег, хотя и не только их одних: Эрика Паскарелло (Eric Pascarello), Эрнста Фридмана Хилла (Ernest Friedman Hill), Эндрю Монкхауза (Andrew Monkhouse), Джина Боярски (Jeanne Boyarsky), Берта Бэйтса (Bert Bates), а также Макса Хабиби (Max Habibi).

Сердечная признательность выражается моему другу Джею и моим собакам, Медвежонку и Козмо, за то, что они терпели мое присутствие и редко наделялись должным вниманием с моей стороны во время работы над этой книгой. И наконец, мне хотелось бы поблагодарить своего соавтора Джона Резига, без которого эта книга не состоялась бы.

Об этой книге

Особое значение JavaScript было очевидно не всегда, но теперь оно несомненно. Веб-приложения служат для того, чтобы предоставить пользователям функционально богатый и удобный интерфейс, но без JavaScript в Интернете можно лишь показывать фотографии кошек. Теперь, как никогда прежде, разработчикам веб-приложений требуются прочные знания и навыки программирования на языке JavaScript, приводящем в действие эти приложения.

Но как и завтрак с апельсиновым соком, язык JavaScript полезен не только для браузеров. Преодолев узкие границы применения в браузерах, этот язык программирования теперь применяется на серверах в таких механизмах, как Rhino и V8, а также в интегрированных средах вроде Node.js. И хотя эта книга посвящена главным образом применению JavaScript в веб-приложениях, основы этого языка программирования, представленные в части II, выходят далеко за пределы применимости при разработке веб-приложений. С увеличением числа разработчиков, пользующихся JavaScript, теперь, как никогда прежде, стало очень важно знать твердо основы этого языка программирования, чтобы владеть им в совершенстве.

Кому адресована книга

Эта книга рассчитана на тех, кто уже знаком с JavaScript. Если же вы только начинаете изучать язык JavaScript или знаете лишь дюжину его операторов из фрагментов кода, найденных в Интернете, значит, эта книга пока еще не для вас. Книга адресована разработчикам веб-приложений, уже владеющим основами программирования на JavaScript. А это означает, что вы должны знать основную структуру операторов JavaScript и уметь составлять из них простые страничные сценарии. Для этого совсем не обязательно иметь большой опыт программирования на JavaScript. Этот опыт можно приобрести, проработав материал данной книги. Но и полным невеждой в JavaScript также нельзя быть.

Для чтения этой книги требуются также знания HTML и CSS. Опять же эти знания и опыт совсем не обязательно должны быть основательными, тем не менее вы должны владеть основами компоновки веб-страниц. Если же вам требуется материал для предварительной подготовки к чтению этой книги, рекомендуем приобрести одно из следующих популярных изданий по JavaScript и разработке веб-приложений: *JavaScript: The Definitive Guide* Дэвида Флэнагана (David Flanagan; в русском переводе книга вышла под названием *JavaScript. Подробное руководство*, *JavaScript: The Good Parts* Дугласа Крокфорда (Douglas Crockford: в русском переводе книга вышла под названием *JavaScript. Сильные стороны*) или *Head First JavaScript* Майкла Моррисона (Michael Morrison; в русском переводе книга вышла под названием *Изучаем JavaScript*)

Структура книги

Эта книга организована так, чтобы вы смогли пройти весь курс обучения от ученика до мастера программирования на JavaScript. В части I представлены основной предмет книги и ряд инструментальных средств для изучения материала остальных частей. В части II главное внимание уделяется основам JavaScript, т.е. тем языковым средствам, которые воспринимаются как вполне естественные, но еще нужно знать, каким образом они действуют. Возможно, это самая важная часть книги, и даже если вы осилите только ее, то и тогда получите намного более основательное представление о JavaScript. В части III основы, представленные в части II, рассматриваются более углубленно в связи с решением трудных задач, которые перед нами ставят браузеры. И в завершающей книгу части IV расширенные языковые возможности рассматриваются на основе уроков, извлеченных из создания таких усовершенствованных библиотек JavaScript, как jQuery.

А теперь перейдем к краткому изложению отдельных глав. В главе 1 представлены те трудности, которые приходится преодолевать разработчикам современных веб-приложений. В ней рассматриваются основные трудности, которые влечет за собой широкое распространение браузеров, а также апробированные передовые методики, которым необходимо следовать при разработке веб-приложений, включая тестирование и анализ производительности.

В главе 2 обсуждаются вопросы тестирования, а также текущее состояние процедур и инструментальных средств тестирования. В ней представлен также небольшой, но эффективный принцип утверждения, который будет в исключительном порядке применяться в остальной части книги с целью убедиться в том, что код выполняет свои функции, а иногда и в том, что он их не выполняет!

Вооружившись упомянутыми выше инструментальными средствами, можно перейти к главе 3, с которой начинается рассмотрение основ языка и, в частности, с понятия *функции* в том виде, в каком оно определяется в JavaScript. На первый взгляд, основное внимание следовало бы уделить понятию *объекта*, тем не менее нужно иметь прежде всего ясное представление о функции, поскольку JavaScript – язык функционального программирования. Ведь именно с понятия функции начинается наше постепенное превращение из обычных учеников в мастеров программирования на JavaScript!

Не завершив до конца рассмотрение функций, основы, усвоенные в главе 3, можно применить уже в главе 4 для решения задач, возникающих при разработке веб-приложений. В этой главе понятие рекурсии рассматривается не только ради нее са-

уществовавших на момент написания книги, в том числе Internet Explorer 9, Firefox, afari и Google Chrome.

Как связаться с авторами

Авторы и издательство Manning Publications приглашают читателей на форум, посвященный этой книге, где они могут разместить свои комментарии к книге, задать опросы технического характера и получить помощь от авторов и других пользователей JavaScript. Для доступа и подписки на форум введите в окне своего браузера адрес ww.manning.com/SecretsoftheJavaScriptNinja и щелкните на ссылке Author nline (Автор в оперативном режиме). На открывшейся странице появятся сведения том, как войти на форум после регистрации, о видах доступной помощи, а также правилах ведения форума.

Об иллюстрации на обложке книги

Фигура воина на обложке книги взята с гравюры на дереве "Актер *но* в роли самурая" еизвестного японского художника середины XIX века. *Но* — это форма музыкальной рамы в классическом японском театре, исполняющейся начиная с XIV века. Многие ерсонажи подобных драм играют в масках, причем мужские и женские роли исполяют только мужчины. Самурай — это героическая фигура японской истории, нередко редставленная в театре и изобразительном искусстве с большим мастерством, пышнотью наряда и суровостью настоящего воина.

Самураи и ниндзя были воинами, отличавшимися в японском военном искусстве своим мастерством, смелостью и хитростью. Самураи относились к военной элите, были орошо образованы, умели читать и писать, а также искусно воевать. Они были связаны трогим кодексом чести под названием бусидо, что означает путь воина. Этот кодекс ести передавался изустно из поколения в поколение, начиная с X века. Самураев наирали из аристократов и высших слоев японского общества аналогично европейским ыцарям. Они шли на битву в строгом боевом порядке в сложных доспехах и ярких деждах, чтобы произвести впечатление и устрашить противника. А ниндзя отличались в большей степени своим боевым искусством, чем общественным положением ли образованностью. Одевались они в черное, закрывая свое лицо, выполняли боеые задания в одиночку или небольшими группами, нападая на противника скрытно, с итрыми уловками и применяя любую тактику для достижения успеха. Единственным равилом их поведения была скрытность.

Иллюстрация на обложке книги отобрана с трех японских гравюр, которые многие оды принадлежат главному редактору издательства Manning Publications. И когда авторы искали походящее изображение ниндзя для обложки этой книги, их внимание прилекла поразительная гравюра самурая, которую они отобрали в качестве иллюстрации лагодаря ее сложной детализации, ярким краскам и выразительному изображению суового воина, готового нанести удар и победить.

В наше время, когда одну компьютерную книгу трудно отличить от другой, издательство Manning Publications славится своей изобретательностью и инициативностью в формлении обложек издаваемых более двадцати лет книг, где наглядно проявляется се многообразие мирового изобразительного искусства. И характерным тому примером служит обложка этой книги.

Об авторах

Джон Резиг является деканом факультета вычислительной техники в Академии Хана и создателем библиотеки jQuery для JavaScript. По данным, опубликованным на веб-сайте BuiltWith.com, эта библиотека в настоящее время применяется на 58% из 10 тысяч самых широко посещаемых в Интернете веб-сайтов, а также на десятках миллионов других сайтов. Следовательно, она относится к числу самых массовых технологий, предназначенных для построения веб-сайтов, а возможно, и самых распространенных за все время технологий программирования.

Джон создал также целый ряд утилит и проектов с открытым кодом, в том числе Processing.js (переносимую на JavaScript версию языка Processing для обработки данных), QUnit (тестовый набор для тестирования кода JavaScript) и TestSwarm (платформу для распределенного тестирования кода JavaScript).

В настоящее время Джон работает над совершенствованием системы образования в области вычислительной техники в Академии Хана, где он разрабатывает учебный план и инструментальные средства для обучения людей программированию независимо от их возраста. Основная цель Академии Хана — создать отличные учебные ресурсы, свободно доступные для всех желающих. Джон работает не только для того, чтобы обучать других программированию, но и с целью зажечь первую искру, возбуждающую интерес к программированию у всякого, написавшего свою первую программу. Проживает он в нью-йоркском районе Бруклин и в свободное от работы время увлекается изучением укиё-э — японской гравюры на дереве.

Беэр Бибо занимается программированием более трех десятилетий, начиная с программы, написанной для игры в крестики-нолики на супер-ЭВМ Control Data Cyber через телетайп на скорости 100 бод. Имея два диплома инженера-электроника, Беэр занимался разработкой антенн и аналогичной аппаратуры, но, начиная с первой работы в корпорации Digital Equipment Corporation, его всегда больше увлекало программирование.

Беэр выполнял проекты на заказ для таких компаний, как Lightbridge Inc., BMC Software, Dragon Systems, Works.com, и многих других коммерческих организаций. Он даже служил в армии США, обучая

солдат-пехотинцев боевому искусству подрывать танки, что очень пригодилось ему впоследствии на совещаниях разработчиков программного обеспечения, где в обстановке бурных дискуссий обсуждались насущные задачи текущих проектов. В настоящее время Беэр работает в должности разработчика архитектуры программного обеспечения в компании, занимающей ведущее место среди поставщиков бытовых шлюзовых устройств и телевизионных абонентских приставок.

Беэр является автором целого ряда других книг, вышедших в издательстве Manning Publications: *jQuery in Action* (первое и второе издания), *Ajax in Practice (Ajax на практике*, пер. с англ., ИД "Вильямс", 2008 г.) и *Prototype and Scriptaculous in Action* (AJAX: библиотеки Prototype и Scriptaculous в действии, пер. с англ., ИД "Вильямс", 2008 г.). Кроме того, он был научным рецензентом многих книг по разработке веб-приложений из серии *Head First*, вышедших в издательстве O'Reilly Publishing: *Head First Ajax*, *Head Rush Ajax* и *Head First Servlets and JSP*.

В свободное от работы за компьютером время Беэр любит готовить сытные обеды, о чем свидетельствует размер его брюк, снимать на фото- и видеокамеру, ездить на своем мотоцикле марки Yamaha V-Star и носить рубашки с отпечатанным рисунком в тропическом стиле. Он работает и проживает в городе Остин, шт. Техас, который очень любит, за исключением совершенно бесшабашных водителей.

От издательства

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

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

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

Наши электронные адреса:

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

Наши почтовые адреса:

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

Часть I

Подготовка к обучению

В этой части подготавливается почва для обучения искусству программирования на JavaScript. Из главы 1 вы узнаете цели авторов этой книги. В ней закладываются основы той среды, в которой авторы привыкли программировать на JavaScript. Из главы 2 вы узнаете, почему так важно выполнять тестирование кода. В ней сначала дается краткий обзор некоторых инструментальных средств тестирования. А затем будут разработаны необычайно простые инструментальные средства тестирования, которыми вам предстоит воспользоваться при последующем обучении. Завершив чтение этой части книги, вы будете готовы приступить к обучению искусству программирования на JavaScript.

Введение в искусство программирования на JavaScript

1

В этой главе...

- Назначение и структура книги
- Краткий обзор рассматриваемых библиотек
- Пояснение передовых методов программирования на JavaScript
- Авторская разработка кросс-браузерного кода
- Примеры применения тестового набора

Если вы читаете эту книгу, то, скорее всего, знаете, что создать эффективный кросс-браузерный код на JavaScript не так-то просто. Помимо обычных трудностей, возникающих при написании чистого кода, приходится преодолевать нелепые сложности и препятствия, которые ставят сами браузеры. Для борьбы с подобными трудностями разработчики кода на JavaScript нередко прибегают к построению набора типичных, неоднократно используемых функций в виде библиотеки JavaScript. Такие библиотеки зачастую отличаются своим содержимым и сложностью, но для них всегда остается одна общая характерная особенность: они должны быть просты в употреблении, построены с наименьшими издержками и пригодны для всех видов целевых браузеров.

Очевидно, что для правильного построения собственного кода требуется ясное представление о том, как устроены и сопровождаются самые лучшие библиотеки JavaScript. Поэтому назначение этой книги — раскрыть приемы и секреты, инкапсулированные этими первоклассными кодовыми базами в единый ресурс. В этой книге будут, в частности, рассмотрены способы построения распространенных библиотек JavaScript. Поэтому ознакомимся с этими библиотеками поближе.

Избранные библиотеки JavaScript

В этой книге главное внимание уделяется способам и практическим приемам, применяемым для создания современных библиотек JavaScript. И основной библиотекой, которую мы будем здесь рассматривать, безусловно, является, jQuery, поскольку она повсеместно применяется в современной практике программирования на JavaScript, занимая ведущее положение среди прочих библиотек.

Библиотека jQuery (`http://jquery.com`) создана и выпущена Джоном Резигом в январе 2006 года. Она распространяет применение CSS-селекторов для согласования с содержимым модели DOM. Помимо прочих функциональных возможностей, она предоставляет средства для манипулирования моделью DOM, обработки Ajax-запросов и событий, а также анимации.

Эта библиотека занимает в настоящее время господствующее положение среди всех библиотек JavaScript, применяемых на сотнях тысяч веб-сайтов, где с ними постоянно сталкиваются миллионы пользователей. Благодаря ее интенсивному применению и критическим отзывам пользователей эта библиотека была с годами уточнена и усовершенствована до современного вида оптимальной кодовой базы.

Помимо примеров кода из библиотеки jQuery, в этой книге будут также рассмотрены методики, реализованные в следующих библиотеках.

- Prototype (`http://prototypejs.org`). Представляет собой прообраз всех современных библиотек JavaScript. Создана и выпущена Сэмом Стивенсоном (Sam Stephenson) в 2005 году. Включает в себя функциональные возможности DOM, Ajax и обработки событий, а также приемы объектно-ориентированного, аспектно-ориентированного и функционального программирования.
- Yahoo UI (`http://developer.yahoo.com/yui`). Являясь результатом собственной разработки интегрированной среды в компании Yahoo, предана гласности в феврале 2006 года. Включает в себя функциональные возможности DOM, Ajax, обработки событий и анимации, а также целый ряд предварительно построенных виджетов (мини-приложений типа календаря, сетки, меню-гармошки и прочего).
- base2 (`http://code.google.com/p/base2`). Создана Дином Эдвардсом (Dean Edwards) и выпущена в марте 2007 года для поддержки функциональных возможностей DOM и обработки событий. Претендует на известность своими попытками реализовать различные спецификации стандарта W3C (Консорциума Всемирной паутины) в универсальном, кросс-браузерном виде.

Все упомянутые выше библиотеки грамотно построены, чтобы всесторонне охватывать те проблемные области, для которых они разработаны. Именно поэтому они и служат удобным основанием для дальнейшего анализа. Правильное представление об основополагающей конструкции этих кодовых баз позволяет лучше понять, как следует строить крупную первоклассную библиотеку на JavaScript. Но рассматриваемые здесь способы и приемы пригодны не только для построения крупных библиотек. Их можно применять во всех случаях программирования на JavaScript независимо от объема кода.

Структуру библиотеки JavaScript можно разделить на три части.

- Расширенное применение языка программирования JavaScript.
- Тщательное построение кросс-браузерного кода.
- Применение целого ряда передовых методик, связывающих все это вместе.

Мы будем подробно анализировать эти три части каждой из упомянутых выше библиотек, чтобы дать полное представление о том, как создавать собственные эффективные кодовые базы на JavaScript.

Общее представление о языке JavaScript

По мере своего профессионального роста большинство программирующих на JavaScript так или иначе доходят до того момента, когда они начинают активно пользоваться в своем коде различными языковыми элементами, включая объекты, обычными и даже анонимными встраиваемыми функциями, если, конечно, они следят за современными тенденциями в программировании. Но их навыки программирования, как правило, редко выходят за рамки самого основного уровня. Кроме того, они обычно очень плохо представляют себе назначение и реализацию *замыканий* в JavaScript. А ведь это понятие помогает окончательно уяснить особое значение функций для данного языка программирования.

В основу JavaScript положена тесная взаимосвязь между объектами, функциями и замыканиями (рис. 1.1). Ясное представление о сильной взаимосвязи между этими понятиями позволяет заметно усовершенствовать навыки программирования на JavaScript, закладывая основы для разработки приложений любого типа.

Рис. 1.1. В основу JavaScript положена тесная взаимосвязь между объектами, функциями и замыканиями

Многие разработчики веб-приложений на JavaScript, особенно имеющие навыки объектно-ориентированного программирования, могут уделять основное внимание объектам, недостаточно представляя тот вклад, который функции и замыкания вносят в общую картину. Помимо этих основополагающих понятий, имеются еще два средства, все еще сильно недооцененные в практике программирования на JavaScript: таймеры и регулярные выражения. Оба средства находят применение буквально в каждой кодовой базе на JavaScript, но их потенциал раскрывается полностью не всегда из-за недостаточно ясного понимания их характера. В частности, принцип действия таймеров в браузере для многих является полной загадкой, а ведь ясное представление о том, как они работают, позволяет решать такие сложные задачи программирования, как, например, организация длительных вычислений и плавной анимации. Кроме того, ясное представление о принципе действия регулярных выражений дает возможность получать фрагменты очень простого и эффективного кода, которые в противном случае зачастую оказываются довольно сложными.

Еще одной важной вехой на пути к более углубленному пониманию особенностей языка JavaScript является рассмотрение оператора `with` в главе 10 и метода `eval()` в главе 9. Оба эти языковые средства превращены в общее место, неправильно применяются и откровенно порицаются большинством программирующих на JavaScript.

Примечание

Тем, кто внимательно следит за современными тенденциями в разработке веб-приложений, должно быть известно, что оба эти языковые средства признаны противоречивыми, а их применение предполагается ограничить или вообще не рекомендовать в последующих версиях JavaScript. Но если эти языковые средства встретятся вам в существующем коде, то их назначение все же нужно знать, даже если вы и не собираетесь пользоваться ими в новом коде.

Но если посмотреть на образцы труда лучших программистов, то нетрудно заметить, что при правильном применении оба упомянутых языковых средства позволяют создавать весьма изящные фрагменты кода, которые нельзя получить иначе. Ведь по большому счету эти средства можно даже применять для упражнений в метапрограммировании, вылепливая из заготовки JavaScript какую угодно форму.

Умение благоразумно пользоваться расширенными языковыми понятиями и средствами должно, без сомнения, благотворно сказаться на качестве создаваемого кода. А оттачивание навыков умелого владения ими доводит наше понимание языка до такого уровня, на котором нам оказывается по плечу разработка практически любого типа приложения на JavaScript. И это послужит нам прочным основанием для продвижения вперед, начиная с написания надежного кросс-браузерного кода.

Соображения по поводу кросс-браузерной разработки

Совершенствование навыков программирования на JavaScript открывает широкие горизонты, особенно теперь, когда применение JavaScript вышло за пределы браузеров и достигло серверов в таких механизмах, как Rhino и V8, а также библиотеках типа Node.js. Но когда дело доходит до разработки браузерных приложений на JavaScript, а именно этому предмету посвящена данная книга, то рано или поздно приходится сталкиваться с самими *браузерами* и весьма неприятными вопросами их несовместимости.

В идеальном случае все браузеры должны работать безошибочно и поддерживать на постоянной основе веб-стандарты, но всем нам хорошо известно, что в действительности дело обстоит совсем иначе. Качество браузеров за последнее время значительно улучшилось, тем не менее им по-прежнему присущи программные ошибки, отсутствие необходимых прикладных интерфейсов API и характерные особенности работы, которые приходится учитывать при разработке веб-приложений. Выработать комплексную стратегию разрешения затруднений, связанных с браузерами, и хорошо знать их отличия и особенности работы не менее важно, если не важнее, чем грамотно программировать на JavaScript.

При написании браузерных приложений или применяемых в них библиотек JavaScript очень важно правильно выбрать поддержку конкретных браузеров. Разумеется, хотелось бы поддерживать все имеющиеся браузеры, но ограничения, накладываемые на ресурсы разработки и тестирования, диктуют совершенно иной подход. Так как же решить, что именно и на каком уровне следует поддерживать?

Для ответа на этот вопрос мы можем позаимствовать старый подход, принятый в веб-службах Yahoo! и называемый *классификацией поддержки браузеров*. При таком подходе мы создаем матрицу поддержки браузеров, которая служит в качестве моментального снимка, отражающего важность браузера и его платформы для наших потребностей. В таком матричном представлении целевые платформы указываются по одной оси, а

браузеры — по другой. Затем в отдельных ячейках матрицы каждой комбинации браузера и платформы присваивается определенный класс (от A до F, хотя для этой цели может быть использована и другая система классификации в зависимости от конкретных потребностей). Гипотетический пример подобной матрицы приведен в табл. 1.1.

Таблица 1.1. Гипотетический пример матрицы поддержки браузеров

	Windows	Mac OS X	Linux	iOS	Android
IE 6		Отсутствует	Отсутствует	Отсутствует	Отсутствует
IE 7, 8		Отсутствует	Отсутствует	Отсутствует	Отсутствует
IE 9		Отсутствует	Отсутствует	Отсутствует	Отсутствует
Firefox				Отсутствует	
Chrome					
Safari			Отсутствует		Отсутствует
Opera					

Обратите внимание на то, что в данной матрице заполнены не все ячейки. Присваивание классов конкретной комбинации платформы и браузера полностью зависит как от требований выполняемого проекта, так и от других важных факторов, в том числе состава целевой аудитории. Применяя подобный подход, можно разработать систему классификации, в которой определяется важность поддержки платформы и браузера, объединив эту информацию со стоимостью такой поддержки, чтобы выявить оптимальный состав поддерживаемых браузеров. Более подробно этот вопрос будет рассматриваться в главе 11.

В повседневной практике программирования нецелесообразно разрабатывать приложения, рассчитанные сразу на большое число платформ и браузеров, поэтому имеет смысл взвесить затраты и преимущества поддержки различных браузеров. В любом подобного рода анализе необходимо принимать во внимание многие факторы. Ниже перечислены самые главные из них.

- Ожидания и потребности целевой аудитории.
- Доля браузера на рынке.
- Затраты труда на поддержку браузера.

Первый фактор носит довольно субъективный характер и определяется в рамках конкретного проекта. С другой стороны, долю браузера на рынке можно зачастую определить с достаточной степенью точности на основании имеющейся информации. А приблизительную оценку затрат труда на поддержку каждого браузера можно произвести, учитывая функциональные возможности браузеров и соблюдение в них современных веб-стандартов.

На рис. 1.2 приведен пример диаграммы, наглядно представляющей сведения об использовании браузеров (получены с веб-сайта StatCounter за август 2012 года) и индивидуальные оценки стоимости затрат на разработку приложений для самых распространенных браузеров. Составление такой диаграммы зависимости затрат от преимуществ поддержки браузеров позволяет сразу же оценить, куда именно следует направить усилия, чтобы получить наибольшую отдачу от вложенных средств. Из этой диаграммы можно сделать следующие выводы.

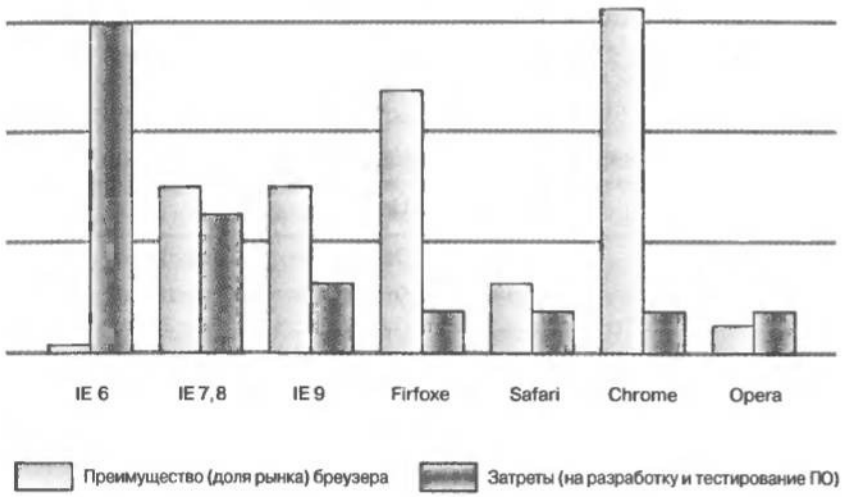

Рис. 1.2. Анализ затрат в сравнении с преимуществами поддержки различных браузеров для настольных систем показывает, куда именно следует направить усилия разработчиков

- Несмотря на то что для поддержки браузеров Internet Explorer 7 и 8 требуется намного больше затрат труда, чем на браузеры, в которых соблюдаются веб-стандарты, им по-прежнему принадлежит значительная доля рынка. Следовательно, дополнительные затраты труда на их поддержку вполне оправданны, если их пользователи относятся к важной целевой аудитории разрабатываемого приложения.
- Благодаря тому что в браузере IE 9 сделаны крупные шаги в сторону соблюдения веб-стандартов, поддерживать его стало легче, чем предыдущие версии. К тому же он постепенно отвоевывает свою долю на рынке.
- Поддержка браузеров Firefox и Chrome не вызывает сомнений, поскольку им принадлежит значительная доля рынка, да и саму поддержку организовать нетрудно.
- Несмотря на то что браузеру Safari принадлежит относительно малая доля рынка, он все же заслуживает поддержки, поскольку в нем соблюдаются веб-стандарты, что существенно снижает затраты на его поддержку. (Существует эмпирическое правило: если веб-приложение работает в браузере Chrome, то оно, скорее всего, будет работать и в браузере Safari, кроме патологических случаев.)
- Несмотря на то что для поддержки браузера Opera требуется не больше затрат труда, чем на поддержку браузера Safari, в настольных системах это может оказаться невыгодным, поскольку ему принадлежит очень малая доля рынка. Но если разработка ведется для мобильных платформ, то здесь браузеру Opera принадлежит значительная доля рынка, как показано на рис. 1.3.
- О поддержке браузера IE 6 и речи быть не может, как следует из материала, приведенного по адресу `www.ie6countdown.com`.

Стоимость затрат на кросс-браузерную разработку может в значительной степени зависеть от квалификации и опыта самих разработчиков. И эта книга призвана повысить уровень квалификации тех читателей, которые в этом кровно заинтересованы.

Поэтому перейдем к рассмотрению наилучших в настоящее время, апробированных на практике, передовых методик.

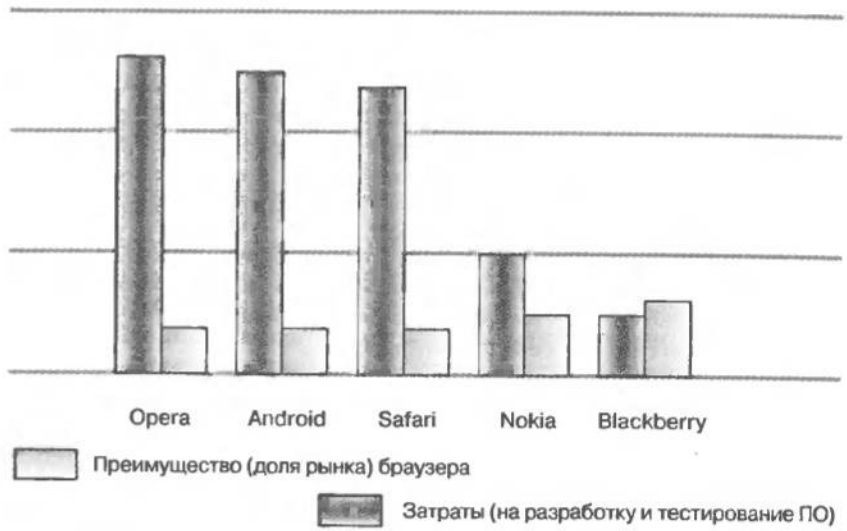

Рис. 1.3. На мобильных платформах, где затраты на поддержку браузеров распределяются довольно равномерно, весь анализ сводится к статистике использования браузеров

Передовые методики

Конечно, иметь хорошие навыки программирования на JavaScript и солидный опыт авторской разработки кросс-браузерного кода никогда не помешает, но это еще не все. Для того чтобы стать по-настоящему квалифицированным разработчиком приложений на JavaScript, необходимо также освоить наилучшие образцы, приемы и способы, накопленные в программировании предыдущими поколениями разработчиков для написания качественного кода. Эти образцы, приемы и способы называются *передовыми методиками* и подробно рассматриваются в главе 2. Помимо совершенного владения языком программирования, они включают в себя следующие элементы.

- Тестирование.
- Анализ производительности.
- Отладка программ.

При написании кода очень важно придерживаться этих методик, что в кросс-браузерной разработке зачастую себя вполне оправдывает. Рассмотрим две из этих передовых методик.

Передовая методика тестирования

В примерах, приведенных в этой книге, мы будем активно пользоваться целым рядом способов тестирования, которые послужат не только для проверки правильности кода, выполняемого в этих примерах, но и в качестве образца для тестирования кода вообще. Основным инструментальным средством, которым мы будем пользоваться для

тестирования кода, служит функция `assert()`, назначение которой — утвердить, что исходная предпосылка истинна или ложна. Так выглядит общая форма этой функции:

```
assert(условие, сообщение);
```

где в качестве первого параметра указывается `условие`, которое должно быть истинно, а в качестве второго параметра — `сообщение`, выводимое в противном случае.

Рассмотрим следующий пример:

```
assert(a == 1, "Disaster! a is not 1!");
```

Если значение переменной a не равно **1**, утверждение не выполняется и выводится предупреждающее об этом сообщение.

Следует иметь в виду, что функция `assert()` не относится к собственным языковым средствам JavaScript, хотя в других языках, например в Java, подобные средства тестирования предоставляются. Поэтому нам придется реализовать эту функцию самостоятельно. О том, как это сделать, речь пойдет в главе 2.

Передовая методика анализа производительности

Другим, не менее важным практическим приемом является анализ производительности. Механизмы JavaScript, действующие в браузерах, претерпели значительные усовершенствования с точки зрения производительности самого языка JavaScript, но это совсем не означает, что можно писать неаккуратный и неэффективный код.

Приведенным ниже фрагментом кода мы будем пользоваться далее в этой книге для сбора данных о производительности.

```
start = new Date().getTime();
for (var n = 0; n < maxCount; n++) {
  /* выполнить измеряемую операцию */
}
elapsed = new Date().getTime() - start;
assert(true,"Measured time: " + elapsed);
```

В приведенном выше фрагменте исполнение измеряемого кода заключается в вилку между временными метками сбора данных: одной до выполнения кода и другой после него. Отличие в этих временных метках покажут, как долго выполняется код. Этим же способом можно сравнить альтернативные варианты выполнения кода.

Как видите, код в данном примере выполняется многократно, и для этой цели служит переменная `maxCount`. Надежно измерить однократное выполнение кода очень трудно, поскольку оно происходит очень быстро, поэтому для получения измеримых величин код приходится выполнять неоднократно. Зачастую количество повторений кода исчисляется десятками, сотнями тысяч или даже миллионами в зависимости от характера измеряемого кода. Подходящее значение переменной `maxCount` можно подобрать методом проб и ошибок.

Рассмотренные выше и другие передовые методики, которые будут представлены далее в книге, в значительной мере способствуют усовершенствованию разработки веб-приложений на JavaScript. Для разработки таких приложений с ограниченными ресурсами, предоставляемыми браузером, просто необходимы прочные и полные навыки программирования, особенно если учесть постоянное усложнение функциональных возможностей и совместимости браузеров.

Резюме

Подведем краткий итог тому, что вы узнали из этой главы.

- Кросс-браузерная разработка веб-приложений — дело непростое, намного более трудное, чем многие думают.
- Для того чтобы преуспеть в ней, нужно не только овладеть в совершенстве языком JavaScript, но и хорошо знать особенности работы браузеров, их сильные и слабые стороны и проблемы несовместимости, а также придерживаться стандартных передовых методик.
- Несмотря на все трудности разработки на JavaScript, есть смельчаки, которые уже прошли этот тернистый путь. К их числу относятся разработчики библиотек JavaScript. Их знаниями и опытом, продемонстрированными при построении кодовых баз, мы можем воспользоваться, чтобы эффективно пополнить свой арсенал приемов и навыков разработки, подняв их до уровня мирового класса.

Изучение подобного опыта, безусловно, будет полезным, поучительным и просветительным, поэтому просто грех им не воспользоваться!

2 Вооружение средствами тестирования и отладки

В этой главе...

- Инструментальные средства для отладки кода JavaScript
- Методы формирования тестов
- Построение набора тестов
- Тестирование асинхронных операций

Построение эффективного набора тестов для разрабатываемого кода имеет непреходящее значение, поэтому рассмотрим эту тему, прежде чем переходить к обсуждению любых вопросов программирования. Надежная методика тестирования требуется для *всего* кода, но она особенно важна в тех случаях, когда внешние факторы могут оказывать влияние на выполнение кода, а именно с такой ситуацией нам приходится сталкиваться при разработке кросс-браузерных приложений на JavaScript.

И дело не только в том, что над одной кодовой базой одновременно может работать несколько разработчиков, в результате чего образуются разорванные части прикладного интерфейса API (это типичные трудности, с которыми приходится иметь дело всем программистам), но и в том, что разрабатываемый код приходится проверять на совместимость со всеми поддерживаемыми браузерами.

Вопросы разработки кросс-браузерных приложений более углубленно будут рассматриваться в главе 11, где речь пойдет о стратегиях написания кросс-браузерного кода. А до тех пор очень важно выяснить вопросы и определить стратегии тестирования, поскольку нам придется пользоваться ими в остальной части книги.

В этой главе мы рассмотрим некоторые инструментальные средства и методики отладки кода JavaScript, формирования тестов на основании результатов отладки и построения набора тестов для надежного выполнения последних.

Отладка кода

Вы, вероятно, помните, что отладка кода JavaScript раньше означала обращение к функции `alert()` для проверки значений переменных. За последние годы возможности отладки кода JavaScript значительно расширились и не в последнюю очередь благодаря широкому распространению расширения Firebug браузера Firefox для веб-разработки. Аналогичные средства веб-разработки имеются теперь и для других основных браузеров.

- Firebug – широко распространенное расширение браузера Firefox для разработчиков (`http://firebug.ru`).
- Средства разработчика IE – включены в состав браузера Internet Explorer 8 и более поздней версии.
- Opera Dragonfly – входит в состав браузера Opera 9.5 или более поздней версии, а также работает с мобильными версиями Opera.
- Средства разработчика WebKit – включены в состав браузера Safari 3 и значительно усовершенствованы в версии Safari 4, а теперь доступны для браузера Chrome.

Отладка кода JavaScript делится на две важные составляющие: регистрация результатов и точки прерывания. И то и другое полезно для отладки кода в одной и той же ситуации, но под разным углом зрения. Это дает возможность выяснить, что же на самом деле происходит в конкретной строке кода. Рассмотрим сначала регистрацию результатов.

Регистрация результатов

Операторы регистрации (в качестве примера можно указать на применение метода `console.log()` в браузерах Firebug, Safari, Chrome, Internet Explorer и последних версиях Opera) являются частью кода (пускай и временно), и поэтому они полезны для отладки кросс-браузерного кода. Вызовы операторов регистрации результатов можно делать непосредственно в отлаживаемом коде, получая отладочную информацию из сообщений, выводимых на консоли всех современных браузеров.

По сравнению со старым методом вывода предупреждения на странице процесс регистрации на консолях браузеров значительно усовершенствовался. Теперь все операторы могут быть сосредоточены на одной консоли и далее просмотрены в любой удобный момент, не мешая нормальному ходу выполнения программы, чего нельзя было добиться раньше с помощью функции `alert()`.

Так, если требуется выяснить значение переменной x в конкретный момент выполнения кода, для этого достаточно написать следующие строки кода:

```
var x = 213;
console.log(x);
```

Результат выполнения этого фрагмента кода с оператором регистрации в браузере Chrome при активизированной консоли JavaScript приведен на рис. 2.1.

В прежних версиях браузера Opera регистрация результатов производилась особым способом: путем реализации оригинального метода `postError()`. Поэтому если требуется регистрация результатов в прежних версиях данного браузера, то, поступив благоразумно, можно реализовать метод регистрации на более высоком уровне, сделав его пригодным для всех браузеров, как показано в листинге 2.1.

Рис. 2.1. Регистрация результатов позволяет видеть фактическое состояние кода при его выполнении

Листинг 2.1. Простой метод отладки, работающий во всех современных браузерах

```
function log() {
  try {
    console.log.apply(console, arguments);
  }
  catch(e) {
    try {
     opera.postError.apply(opera, arguments);
    }
    catch(e){
      alert(Array.prototype.join.call(arguments, " "));
    }
  }
}
```

❶ Попытаться зарегистрировать сообщение самым обычным методом

❷ Перехватить любые сбои в регистрации

❸ Попытаться зарегистрировать так, как это делается в Опера

❹ Выдать предупреждение, если ничто другое не срабатывает

Совет

Для любопытствующих более полный вариант кода из листинга 2.1 имеется по адресу `http://patik.com/blog/complete-cross-browser-console-log/`.

В примере кода из листинга 2.1 сначала предпринимается попытка зарегистрировать сообщение методом, работоспособным в большинстве современных браузеров ❶. Если этого не удастся сделать, будет сгенерировано исключение, которое перехватывается ❷, после чего можно попытаться зарегистрировать сообщение оригинальным для браузера Опера методом ❸. Если же оба эти метода не срабатывают, то приходится обращаться к старому способу выдачи предупреждений ❹.

Примечание

В коде из листинга 2.1 применяются методы `apply()` и `call()` из конструктора объектов типа `Function()` в JavaScript для передачи аргументов из *исходной* функции в функцию регистрации. Эти вспомогательные методы предназначены для точного управления вызовами функций JavaScript и более подробно рассматриваются в главе 3.

Регистрация результатов, безусловно, очень удобна для просмотра состояния кода при его выполнении, но иногда требуется остановить действие и проанализировать его. Именно для этой цели и служат точки прерывания.

Точки прерывания

Точки прерывания не так просты, как операторы регистрации, но они дают заметное преимущество – останавливать выполнение программы или сценария, а заодно и работу браузера, на конкретной строке кода. Это позволяет тщательно проанализировать все обычно недоступные состояния, переменные, контекст и цепочку областей действия. Допустим, имеется страница, на которой применяется новый метод `log()`, как показано в листинге 2.2.

Листинг 2.2. Простая страница, на которой применяется специальный метод `log()`

```
<!DOCTYPE html>
<html>
  <head>
    <title>Listing 2.2</title>
    <script type="text/javascript" src="log.js"></script>
    <script type="text/javascript">
      var x = 213;
      log(x);                    ❶ Строка, где происходит прерывание
    </script>
  </head>
  <body>
  </body>
</html>
```

Если установить точку прерывания средствами Firebug в строке кода, обозначенной меткой ❶ в листинге 2.2 (для этого достаточно щелкнуть на номере строки на полях экрана сценария – Script), а затем обновить страницу для выполнения кода, то отладчик остановит его выполнение на данной строке и выведет на экран результат, приведенный на рис. 2.2.

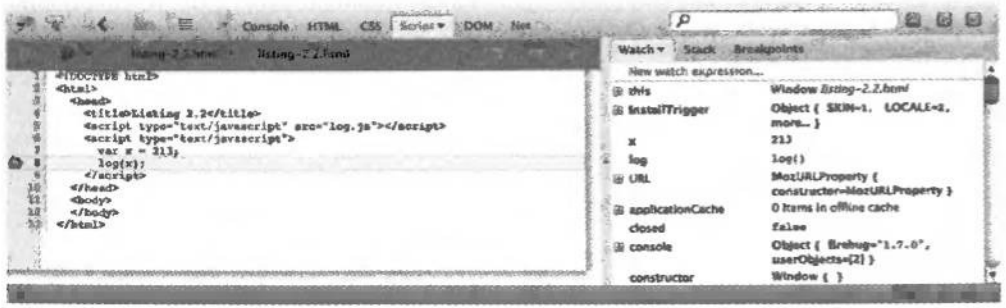

Рис. 2.2. Точки прерывания позволяют остановить выполнение кода на конкретной строке, чтобы проанализировать его состояние

Обратите внимание на то, что на правой панели отображается состояние, в котором находится исполняемый код, включая и значение переменной x. Отладчик прерывает выполнение кода на строке *перед* непосредственным выполнением прерываемой строки кода. В рассматриваемом здесь примере это строка, в которой должен быть выполнен вызов метода `log()`.

Если попытаться отладить программу новым методом, то можно было бы *войти* в этот метод и посмотреть, что же в нем происходит. Так, если щелкнуть на кнопке Step Into (Шаг внутрь): крайняя слева кнопка с золотистой стрелкой), отладчик выполнит код до первой строки кода данного метода, и на экране появится результат, приведенный на рис. 2.3. Обратите внимание, насколько изменилось отображаемое состояние, что дает возможность проанализировать новое состояние, в котором оказался выполняемый метод `log()`.

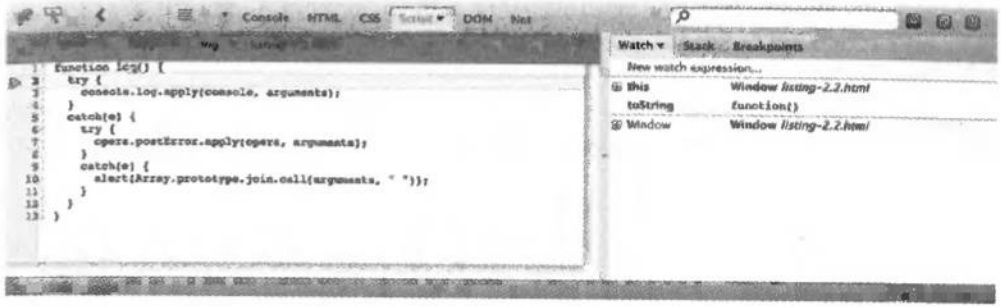

Рис. 2.3. Войдя в отлаживаемый метод, можно увидеть новое состояние, в котором он выполняется

Функционирование любого полноценного отладчика со средствами установки точек прерывания сильно зависит от среды браузера, в которой он выполняется. Именно по этой причине и были созданы упоминавшиеся выше инструментальные средства разработки, функциональные возможности которых иначе были бы просто недоступны. Для сообщества разработчиков стало большим благом и облегчением то обстоятельство, что создатели основных браузеров присоединились наконец к созданию эффективных сервисных программ специально для отладки.

Помимо вполне очевидной цели обнаруживать и устранять программные ошибки, отладка кода служит еще одной весьма полезной цели. Она способствует выработке передовых методик формирования эффективных контрольных примеров для тестирования кода.

Формирование тестов

Известный американский поэт Роберт Фрост писал, что если изгородь хороша, то и соседи окажутся хорошими, но и в разработке веб-приложений в частности и любой области программирования вообще хорошие тесты способствуют написанию хорошего кода. Подчеркнем особое значение слова *хорошие*. Ведь вполне возможно создать обширный тестовый набор, который на самом деле ничем не помогает написанию качественного кода, если тесты построены неудачно.

Хорошие тесты обладают тремя важными свойствами.

- **Повторяемость** – результаты тестирования должны легко воспроизводиться. Повторно выполняемые тесты должны всегда давать одни и те же результаты. Если результаты тестирования не поддаются определению, то как отличить достоверные результаты от недостоверных? Кроме того, повторяемость тестов

гарантирует, что они не зависят от таких внешних факторов, как нагрузка на сеть или ЦП.

- **Простота** — тесты должны быть нацелены на что-нибудь одно. Из теста следует исключить как можно больше разметки в коде HTML, стилевого оформления по таблицам CSS или сценариев JavaScript, но не нарушая исходный контрольный пример. Чем больше элементов исключается из теста, тем больше вероятность того, что на контрольный пример будет оказывать влияние только конкретный тестируемый код.
- **Независимость** — тесты должны выполняться обособленно. Результаты выполнения одного теста не должны зависеть от другого теста. Тесты следует разделять на как можно более мелкие блоки, что помогает точнее определить источник программной ошибки, когда она возникает.

Тесты можно построить самыми разными способами. В целом их можно разделить на две основные разновидности: *деконструктивные контрольные примеры* и *конструктивные контрольные примеры*.

- **Деконструктивные контрольные примеры.** Создаются для того, чтобы свести код (путем деконструирования) к отдельной проблеме, исключая все, что к ней не относится. Благодаря этому удается достичь перечисленных выше свойств тестов. Начиная с целого сайта и последовательно исключая лишнюю разметку, стилевое оформление по таблицам CSS и сценарии JavaScript, можно в конечном итоге прийти к уменьшенному варианту контрольного примера, воспроизводящему искомую проблему.
- **Конструктивные контрольные примеры.** Начинаются с известного, упрощенного примера, сложность которого наращивается (путем конструирования) до тех пор, пока не будет воспроизведена искомая программная ошибка. Для такого тестирования требуется пара простых тестовых файлов, на основании которых строятся новые тесты, а также способ формирования этих тестов из чистового варианта проверяемого кода.

Рассмотрим пример конструктивного тестирования. Создание упрощенных контрольных примеров в конечном итоге приводит к появлению нескольких HTML-файлов, в которые уже включены минимальные функциональные возможности. Для различного функционального назначения могут даже иметься разные начальные файлы: один — для манипулирования моделью DOM, другой — для тестирования средств Ajax, третий — для анимации и т.д. Так, в листинге 2.3 приведен простой контрольный пример модели DOM, используемый для тестирования библиотеки jQuery.

Листинг 2.3. Упрощенный контрольный пример модели DOM для тестирования библиотеки jQuery

```
<script src="dist/jquery.js"></script>
<script>
  $(document).ready(function(){
    $("#test").append("test");
  });
</script>
<style>
  #test { width: 100px; height: 100px; background: red; }
```



```
</style>
<div id="test"></div>
```

Для формирования теста из чистового варианта проверяемой кодовой базы мы можем воспользоваться небольшим сценарием командного процессора, чтобы проверить библиотеку, получить копию контрольного примера и построить тестовый набор, как показано ниже.

```
#!/bin/sh
# Проверить свежую копию библиотеки jQuery
git clone git://github.com/jquery/jquery.git $1
# Скопировать фиктивный контрольный пример в файл
cp $2.html $1/index.html
# Создать копию тестового набора для библиотеки jQuery
cd $1 && make
```

После сохранения в файле `gen.sh` приведенный выше сценарий можно выполнить из командной строки следующим образом:

```
./gen.sh mytest dom
```

В итоге контрольный пример модели DOM будет извлечен из файла `dom.html` в хранилище типа Git.

С другой стороны, можно воспользоваться готовой службой, предназначенной для создания простых контрольных примеров. Одна из таких служб, JSBin (`http://jsbin.com`), представляет собой простое инструментальное средство для построения контрольных примеров, которые затем становятся доступными по особому URL. Имеется даже возможность включать в такие примеры копии некоторых наиболее распространенных библиотек JavaScript. Пример сеанса работы со службой JSBin приведен на рис. 2.4.

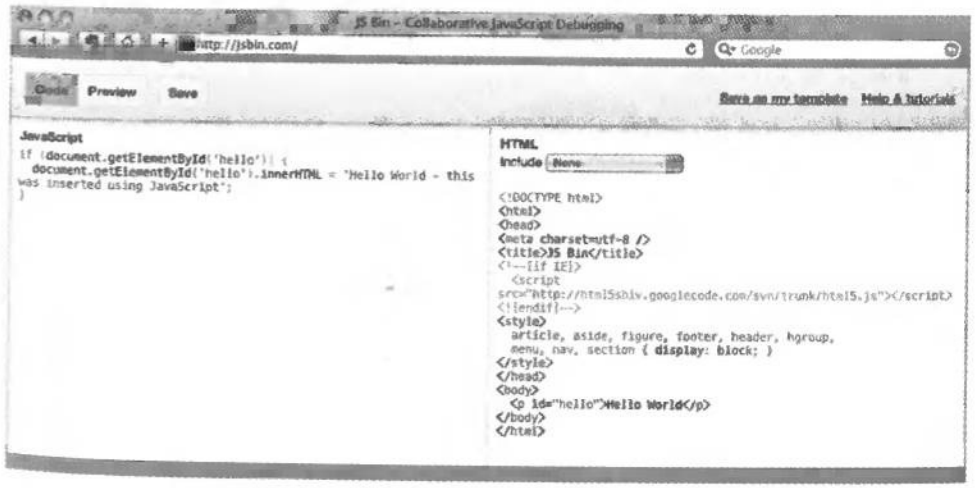

Рис. 2.4. Моментальный снимок экрана действующей веб-службы JSBin

Имея в своем распоряжении подходящие инструментальные средства и зная, как свести контрольный пример к самому простому случаю, мы можем начать построение на его основе тестового набора, чтобы упростить неоднократно повторяющееся выполнение тестов. Ниже поясняется, как это делается.

Среды тестирования

Тестовый набор должен служить в качестве основополагающей стадии процесса разработки, поэтому очень важно выбрать такой набор, который лучше всего подходит для вашего стиля программирования и кодовой базы. Тестовый набор JavaScript должен служить единственной цели: отображать результаты выполнения тестов, чтобы упростить отделение тестов, которые успешно прошли, от тех, что не прошли. И достичь этой цели, не заботясь ни о чем другом, кроме создания тестов и организации их в наборы, помогают тестовые среды.

Существует целый ряд свойств, для которых может потребоваться подходящая среда блочного тестирования, написанная на JavaScript. К их числу относятся следующие.

- Имитация поведения браузера (его реакции на действия мышью, нажатия клавиш и т.д.).
- Управление тестами в диалоговом режиме (приостановка и возобновление тестов).
- Управление асинхронными тестами по истечении времени ожидания.
- Фильтрация тестов для их выполнения.

Результаты неофициального исследования, пытавшегося выяснить, какими именно средами тестирования кода JavaScript пользуются разработчики в своей повседневной работе, проливают некоторый свет на рассматриваемый здесь вопрос. Так, круговая диаграмма, приведенная на рис. 2.5, наглядно показывает неутешительный факт, что большинство опрошенных в ходе данного исследования вообще не тестируют код. Нетрудно представить, что в действительности доля в процентах тех, кто вообще не тестирует код, еще выше.

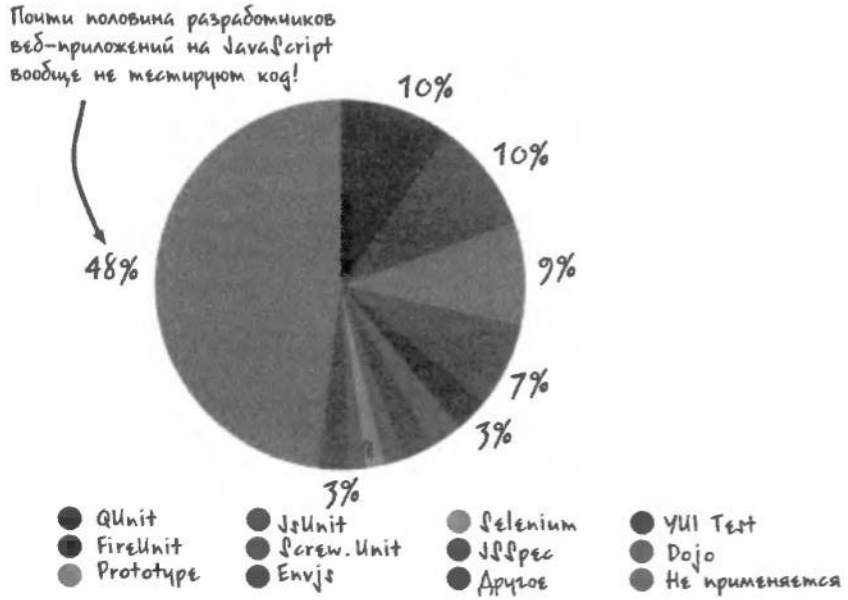

Рис. 2.5. Неутешительный факт: довольно большая доля разработчиков веб-приложений на JavaScript вообще не тестирует код!

Примечание

Интересующиеся могут ознакомиться с исходными результатами опроса в рамках упомянутого выше исследования по адресу `http://spreadsheets.google.com/pub?key=ry8NZN4-KtaolRcwae-9Ljw&output=html`.

Еще одним откровением упомянутого выше исследования стало то обстоятельство, то подавляющее большинство авторов сценариев, которые все-таки тестируют код, пользуются одним из следующих четырех инструментальных средств: JsUnit, QUnit, Selenium или YUI Test. Десять самых распространенных инструментальных средств тестирования приведены на рис. 2.6.

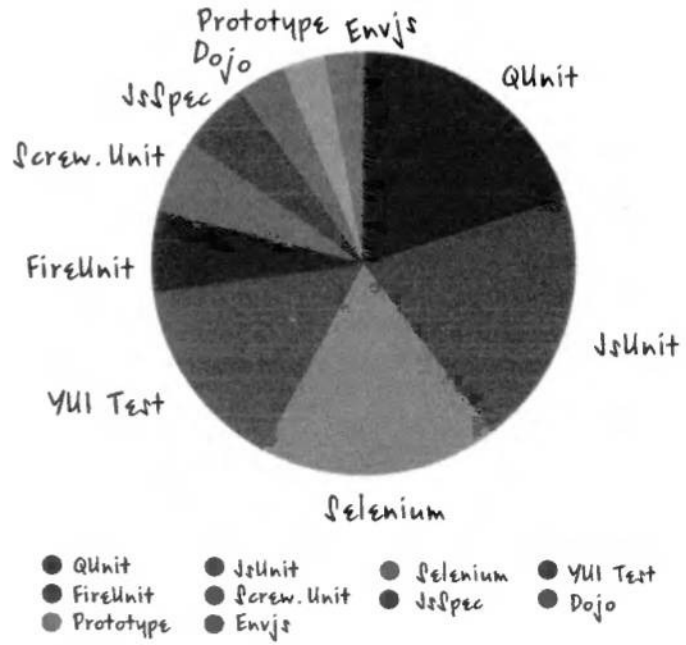

Рис. 2.6. Большинство грамотных разработчиков предпочитают пользоваться всего лишь десятком инструментальных средств тестирования

Результаты данного исследования приводят к любопытному выводу: на данный момент отсутствует какая-нибудь одна наиболее предпочтительная среда тестирования. Но еще любопытнее заметная доля других, одноразовых сред тестирования, которыми пользуется относительно малое число разработчиков (см. рис. 2.6).

Следует, однако, заметить, что написать среду тестирования сызнова не так уж и трудно. Помимо прочего, это позволит лучше понять назначение среды тестирования. Такое упражнение особенно интересно тем, что при написании среды тестирования приходится иметь дело только с JavaScript, не касаясь вопросов кросс-браузерной разработки. И даже если вы попытаетесь сымитировать события в браузере, то в добрый час! Впрочем, речь об этом пойдет в главе 13.

В соответствии с результатами, приведенными на рис. 2.6, ряд разработчиков пришли к тому же самому выводу и написали большое количество одноразовых сред тестирования, отвечающих их насущным потребностям. И хотя написать оригиналь-

ную среду блочного тестирования вполне возможно, вы, скорее всего, воспользуетесь одной из готовых, универсальных сред. Как правило, среды блочного тестирования кода JavaScript состоят из нескольких основных компонентов: модуля выполнения тестов, группирований тестов и утверждений. Некоторые из них предоставляют также возможность выполнять тесты в асинхронном режиме. Рассмотрим вкратце самые распространенные среды блочного тестирования.

QUnit

QUnit – это среда блочного тестирования, первоначально построенная для тестирования библиотеки jQuery. Но с тех пор ее функциональные возможности значительно расширились, и теперь она является автономной средой блочного тестирования. Среда QUnit служит главным образом в качестве простого решения задачи блочного тестирования, предоставляя минимальный, но простой в употреблении прикладной интерфейс API.

Ниже перечислены отличительные особенности QUnit.

- Простой прикладной интерфейс API.
- Поддержка асинхронного тестирования.
- Блочное тестирование, не ограничивающееся только библиотекой jQuery и используемым ее кодом.
- Особая пригодность для регрессионного тестирования.

Дополнительные сведения о jQuery можно найти по адресу `http://qunitjs.com`.

YUI Test

YUI Test – это среда тестирования, построенная, разработанная и выпущенная компанией Yahoo! в октябре 2008 года. Она была полностью переписана в 2009 году согласованно с выпуском библиотеки YUI 3. В YUI Test предоставляется впечатляющее разнообразие средств и функциональных возможностей, способных охватить любые контрольные примеры, требующиеся для блочного тестирования кодовой базы пользователя.

Ниже перечислены отличительные особенности YUI Test.

- Обширные и полные функциональные возможности блочного тестирования.
- Поддержка асинхронного тестирования.
- Хорошая имитация событий.

Дополнительные сведения о YUI Test можно найти по адресу `http://developer.yahoo.com/yui/3/test/`.

JsUnit

JsUnit – это версия распространенной среды тестирования Java JUnit, перенесенная на JavaScript. И хотя JsUnit относится к числу самых распространенных сред тестирования кода JavaScript, в то же время это одна из самых старых сред (как с точки зрения возраста кодовой базы, так и качества). За последнее время эта среда нечасто обновлялась, и поэтому для тестирования кода, предназначенного для работы со всеми

современными браузерами, JsUnit, возможно, окажется не самым лучшим выборомДополнительные сведения о JsUnit можно найти по адресу `www.jsunit.net/`.

Новейшие среды блочного тестирования

Согласно сведениям, опубликованным на главной странице веб-сайта, посвященного библиотеке JUnit, разработчики из компании Pivotal Labs сосредоточили свои усилия на построении новой среды тестирования под названием Jasmine. Подробнее об этом можно узнать по адресу `http://pivotal.github.com/jasmine/`.

Еще одна новейшая среда тестирования, о которой следует знать, называется TestSwarm. Это распределенная среда тестирования с непрерывной интеграцией первоначально разработана Джоном Резигом и теперь входит в состав виртуальной экспериментальной лаборатории Mozilla Labs (`https://github.com/jquery/testswarm/wiki`). Рассмотрим далее процесс создания тестовых наборов.

Основы построения тестовых наборов

Основное назначение тестового набора – сгруппировать в одном месте все отдельные тесты, которые могут иметься для кодовой базы, чтобы выполнять их совместно. Благодаря этому тесты могут неоднократно выполняться из единого легко доступного источника.

Для того чтобы лучше понять принцип действия тестового набора, целесообразно рассмотреть процесс его построения. Как ни странно, наборы для тестирования кода JavaScript строятся довольно просто. В частности, вполне работоспособный тестовый набор можно построить, написав около 40 строк кода.

В этой связи невольно возникает вопрос: а зачем вообще строить новый тестовый набор? Ведь, как правило, нет никакой необходимости писать на JavaScript собственный тестовый набор, поскольку существует целый ряд качественных тестовых наборов, среди которых можно найти наиболее подходящий, как было показано ранее. Но само построение тестового набора может дать неплохой практический опыт, особенно при изучении принципа действия асинхронного тестирования.

Утверждение

В основу среды блочного тестирования положен метод утверждения, называемый `assert()`. Этот метод обычно принимает значение, предпосылка которого утверждается, а также описание цели утверждения. Если значение вычисляется как истинное, утверждение проходит проверку, а иначе оно считается ложным. Соответствующее сообщение регистрируется с подходящей меткой прохождения или непрохождения проверки. Простой пример реализации такого принципа приведен в листинге 2.4.

Функция `assert()` на удивление проста ❶. В ней сначала создается элемент разметки списка `<li>`, содержащий описание цели утверждения, затем этому элементу присваивается имя класса `pass` или `fail` в зависимости от значения параметра утверждения (`value`) и, наконец, новый элемент разметки присоединяется к списку в теле документа ❷.

Листинг 2.4. Простой пример реализации утверждения на JavaScript

```
<html>
<head>
```

```
<title>Test Suite</title>
<script>

function assert( value, desc ) {
  var li = document.createElement("li");
  li.className = value ? "pass" : "fail";
  li.appendChild( document.createTextNode( desc ) );
  document.getElementById("results").appendChild( li );
}

window.onload = function(){
  assert( true, "The test suite is running." );
  assert(false, "Fail!");
};
</script>

<style>
  #results li.pass { color: green; }
  #results li.fail { color: red; }
</style>
</head>
<body>
  <ul id="results"></ul>
</body>
</html>
```

❶ Определить метод assert()

❷ Выполнить тесты, используя утверждения

❸ Задать стили оформления результатов

❹ Сохранить результаты

Тестовый набор состоит из двух простых тестов: один из них всегда проходит, а другой — не проходит ❷. Правила стилевого оформления для классов `pass` и `fail` наглядно показывают в цвете факт прохождения или непрохождения тестов ❸.

Функция `assert()` довольно проста, но она может послужить неплохим стандартным блоком для последующей разработки. А мы будем пользоваться ею как методом на протяжении всей книги для тестирования различных фрагментов кода, проверяя их целостность.

Группы тестов

Простые утверждения, безусловно, полезны, но их истинный потенциал раскрывается, когда они объединяются вместе в контексте тестирования, образуя *группы тестов*. При выполнении блочного тестирования группа тестов, скорее всего, будет представлять совокупность утверждений, связанных с отдельным методом из прикладного интерфейса API или приложения. Если бы выполнялась разработка на основе поведения, то в группе тестов были бы собраны утверждения по отдельным задачам. Но в любом случае реализация, по существу, остается одинаковой.

В рассматриваемом здесь простом примере тестового набора формируется группа тестов, в которой отдельные утверждения вводятся в результаты. И если любое утверждение оказывается ложным, то и вся группа помечается как не проходящая тесты. Результат выполнения тестов из листинга 2.5 остается довольно простым, хотя на практике было бы полезно организовать на некотором уровне динамическое управление группами тестов, сокращая или расширяя и фильтруя их, если они содержат тесты, которые не проходят.

Листинг 2.5. Реализация группирования тестов

```
<html>
  <head>
    <title>Test Suite</title>
      <script>

        (function() {
          var results;
          this.assert = function assert(value, desc) {
            var li = document.createElement("li");
            li.className = value ? "pass" : "fail";
            li.appendChild(document.createTextNode(desc));
            results.appendChild(li);
            if (!value) {
              li.parentNode.parentNode.className = "fail";
            }
            return li;
          };
          this.test = function test(name, fn) {
            results = document.getElementById("results");
            results = assert(true, name).appendChild(
              document.createElement("ul"));
            fn();
          };
        })();

        window.onload = function() {
          test("A test.", function() {
            assert(true, "First assertion completed");
            assert(true, "Second assertion completed");
            assert(true, "Third assertion completed");
          });
          test("Another test.", function() {
            assert(true, "First test completed");
            assert(false, "Second test failed");
            assert(true, "Third assertion completed");
          });
          test("A third test.", function() {
            assert(null, "fail");
            assert(5, "pass")
          });
        };
      </script>
      <style>
        #results li.pass { color: green; }
        #results li.fail { color: red; }
      </style>
  </head>
  <body>
    <ul id="results">></ul>
```

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

Как следует из листинга 2.5, реализация группы тестов не сильно отличается от элементарной регистрации утверждений. Главное отличие состоит во включении в код переменной `results`, содержащей ссылку на текущую группу тестов (благодаря этому регистрируемые утверждения вводятся правильно). Помимо простого тестирования кода, еще одним важным вопросом организации среды тестирования является обработка асинхронных операций.

Асинхронное тестирование

Неприятным и трудным препятствием, на которое наталкивается большинство разработчиков при написании тестового набора на JavaScript, служит обработка асинхронных тестов. Это такие тесты, результаты которых возвращаются через неопределенный промежуток времени, как, например, при тестировании средств Ajax или анимации.

Зачастую обработка асинхронных тестов организуется намного более сложным с технической точки зрения способом, чем требуется на самом деле. Для обработки асинхронных тестов достаточно выполнить следующие действия.

1. Утверждения, которые опираются на ту же самую асинхронную операцию, должны быть объединены в единую группу тестов.
2. Каждая группа тестов должна быть помещена в очередь на выполнение после предыдущих групп тестов.

Таким образом, каждая группа тестов может выполняться асинхронно. Пример реализации такого подхода к обработке асинхронных тестов приведен в листинге 2.6.

Листинг 2.6. Простой пример асинхронного тестового набора

```
<html>
  <head>
    <title>Test Suite</title>
      <script>
        (function() {
          var queue = [], paused = false, results;
          this.test = function(name, fn) {
            queue.push(function() {
              results = document.getElementById("results");
              results = assert(true, name).appendChild(
                document.createElement("ul"));
              fn();
            });
            runTest();
          };
          this.pause = function() {
           paused = true;
          };
          this.resume = function() {
            paused = false;
            setTimeout(runTest, 1);
          };
```



```
          function runTest() {
            if (!paused && queue.length) {
              queue.shift()();
              if (!paused) {
                resume();
              }
            }
          }
          this.assert = function assert(value, desc) {
            var li = document.createElement("li");
            li.className = value ? "pass" : "fail";
            li.appendChild(document.createTextNode(desc));
            results.appendChild(li);
            if (!value) {
              li.parentNode.parentNode.className = "fail";
            }
            return li;
          };
        })();
        window.onload = function() {
          test("Async Test #1", function() {
            pause();
            setTimeout(function() {
               assert(true, "First test completed");
              resume();
            }, 1000);
          });
          test("Async Test #2", function() {
            pause();
            setTimeout(function() {
              assert(true, "Second test completed");
              resume();
            }, 1000);
          });
        };
      </script>
       <style>
         #results li.pass {
           color: green;
         }
         #results li.fail {
           color: red;
         }
      </style>
  </head>
  <body>
    <ul id="results"></ul>
  </body>
</html>
```

Для разделения функциональных возможностей, представленных в листинге 2.6, имеются три общедоступные функции: тестирования — `test()`, приостановки — `pause()` и возобновления — `resume()`. У этих трех функций следующие возможности.

- Функция `test(fn)` принимает переданную ей функцию, которая содержит ряд утверждений, выполняющихся как синхронно, так и асинхронно, помещает ее в очередь и ожидает выполнения.
- Функция `pause()` должна вызываться из функции `test()` и давать тестовому набору команду на приостановку выполнения тестов до тех пор, пока не будет обработана предыдущая группа тестов.
- Функция `resume()` возобновляет тесты и начинает выполнение следующей группы тестов с короткой задержкой, которая специально вводится во избежание длительного выполнения кодовых блоков.

Еще одна внутренняя функция `runTest()` вызывается всякий раз, когда тест помещается в очередь и удаляется из нее. Эта функция проверяет, возобновлено ли в настоящий момент выполнение тестового набора и присутствуют ли в очереди какие-нибудь тесты. Если они присутствуют, то данная функция удаляет очередной тест из очереди и пытается выполнить его. Кроме того, по завершении обработки группы тестов она проверяет, приостановлено ли выполнение тестового набора. И если оно не приостановлено в настоящий момент, а значит, в группе тестов выполнялись только асинхронные тесты, то данная функция начнет выполнение следующей группы тестов. Более подробно особенности отложенного выполнения поясняются в главе 8, посвященной таймерам, где углубленно исследуется механизм задержки выполнения кода JavaScript.

Резюме

В этой главе был рассмотрен ряд основных способов, имеющих отношение к отладке кода JavaScript и построению простых контрольных примеров по результатам отладки.

- Сначала в ней было показано, как пользоваться регистрацией результатов для наблюдения за действиями отлаживаемого кода при его выполнении, а по ходу дела был реализован служебный метод, с помощью которого можно успешно регистрировать информацию как в современных, так и в устаревших браузерах независимо от имеющихся у них отличий.
- Затем были рассмотрены особенности применения точек прерывания для остановки выполнения кода в определенном месте, что дает возможность проанализировать состояние, в котором находится исполняемый код.
- Далее в главе обсуждались вопросы формирования качественных тестов с учетом их свойств: *повторяемости*, *простоты* и *независимости*. Помимо этого, были рассмотрены две разновидности тестирования: *деконструктивного* и *конструктивного*.
- Кроме того, в этой главе были представлены данные о том, как программирующие на JavaScript пользуются тестированием, а также сделан краткий обзор существующих сред тестирования, которые можно изучить и принять на вооружение, если требуется формализованная среда тестирования.

- На основании изложенного выше далее было рассмотрено понятие утверждения и реализован простой метод утверждения, которым мы будем нередко пользоваться в остальной части книги для проверки правильности выполнения кода.
- И наконец, в этой главе было показано, как строится простой тестовый набор для обработки асинхронных тестов. Все рассмотренные здесь способы и средства тестирования и отладки кода послужат прочным основанием для разработки веб-приложений на JavaScript.

Итак, снарядившись всем необходимым, вы можете приступать к обучению. Сделайте короткий перерыв, чтобы ступить на поприще обучения, где первый урок может оказаться совсем не по тому предмету, который вы предполагали изучить!

Часть II

Обучение ученика

Итак, вы мысленно настроились на обучение и вооружились элементарными инструментальными средствами тестирования, разработанными в предыдущей части книги. Теперь вы готовы к изучению самых основ того арсенала средств, который доступен для вас в JavaScript.

Из главы 3 вы узнаете все о самом важном и основополагающем понятии JavaScript, которым является не объект, а *функция*. В этой главе поясняются причины, по которым ясное представление о функциях JavaScript является ключом к разгадке секретов этого языка программирования.

В главе 4 будет продолжено углубленное исследование функций. Ведь они настолько важны, что им стоит посвятить не одну главу книги. А в этой главе будет показано, как пользоваться функциями для разрешения трудных задач и затруднений, с которыми приходится сталкиваться разработчикам веб-приложений.

В главе 5 изучение функций переходит на следующий уровень сложности. В ней рассматриваются замыкания — одно из самых превратно понимаемых (а порой и неизвестных) понятий языка JavaScript.

Глава 6 посвящена изучению объектов с особым упором на то, как образец объекта определяется его прототипом. Из этой главы вы узнаете, как пользоваться объектно-ориентированными средствами JavaScript.

Далее обучение приобретает более углубленный характер, который проявляется в доскональном изучении регулярных выражений в главе 7. Применяя регулярные выражения в JavaScript, вы научитесь сокращать крупные фрагменты кода, сводя их к дюжине операторов.

Ваше ученичество завершается уроками из главы 8, посвященной таймерам, включая и уроки по модели однопоточной обработки, применяемой в JavaScript. По ходу дела вы научитесь владеть и пользоваться ею с наибольшей выгодой, чтобы она не взяла верх над вами.

3

Функции как основа основ

В этой главе...

- Краткий обзор особой роли функций
- Рассмотрение функций в качестве объектов высшего порядка
- Вызов функций из браузера
- Объявление функций
- Присваивание функциям параметров
- Контекст в функциях

Перейдя к чтению этой части, посвященной основам JavaScript, вы, вероятно, будете несколько удивлены тем, что в ней сначала рассматриваются *функции*, а не объекты. Разумеется, мы уделим достаточно внимания и объектам (особенно в главе 6), но, по существу, мастер программирования на JavaScript отличается от середнячка, главным образом, ясным представлением о том, что JavaScript – это язык *функционального программирования*. От понимания этой особенности JavaScript зависит уровень сложности всего кода, который вам предстоит вообще написать на этом языке.

Если вы читаете эту книгу, то уже не относитесь к новичкам и, следовательно, в достаточной степени владеете основами обращения с объектами, а расширенные представления о них вы можете почерпнуть из главы 6. Но ясное представление о роли функций в JavaScript – это единственное и самое главное оружие, которым вы можете владеть. Именно поэтому эта и две последующие главы посвящены подробному исследованию роли функций в JavaScript.

Самое главное, что функции в JavaScript относятся к категории *объектов высшего порядка*. Они могут интерпретироваться в JavaScript как любые другие объекты и сосуществовать с ними. На них, как и на обычные типы данных в JavaScript, можно ссылаться

в переменных, объявлять в литералах и даже передавать в качестве параметров другим функциям.

Тот факт, что функции интерпретируются в JavaScript как объекты высшего порядка, имеет значение на самых разных уровнях, но главное преимущество проявляется в краткости кода. Забегая немного вперед и не вдаваясь в подробный анализ, который будет сделан далее в этой главе, рассмотрим для сравнения следующий код, написанный на Java для выполнения сортировки коллекции:

```
Arrays.sort(values,new Comparator<Integer>(){
  public int compare(Integer value1, Integer value2) {
    return value2 - value1;
  }
});
```

а также эквивалентный ему код JavaScript, написанный с применением функционального подхода к программированию:

```
values.sort(function(value1,value2){ return value2 - value1; });
```

Если такая запись вам не совсем понятна, пусть это вас не смущает — к концу главы вы вполне овладеете приемами написания такого краткого кода. А до тех пор нам просто хотелось бы наглядно продемонстрировать те преимущества, которые дает ясное представление о JavaScript как о языке функционального программирования.

В этой главе основное внимание уделяется роли функций в JavaScript. Излагаемый в ней материал послужит вам прочным основанием для написания кода JavaScript на таком уровне, которым бы гордился любой мастер программирования.

Главное отличие JavaScript как языка функционального программирования

Вам, вероятно, не раз приходилось слышать жалобы коллег по поводу их неприязни к JavaScript. Можно с уверенностью сказать, что в девяти случаях из десяти (или еще чаще) это прямое следствие попытки жалующегося воспользоваться JavaScript как другим языком, который он знает лучше. В результате его постигает сильное разочарование, что это *не тот* язык. Нечто подобное нередко случается с теми, кто переходит на JavaScript с таких языков, как Java, которые определенно не относятся к языкам функционального программирования, но освоены разработчиком до знакомства с JavaScript.

Хуже того, разработчиков часто вводит в заблуждение само название языка *Java*Script. Не вникая в предысторию выбора такого названия, следует заметить, что разработчики, вероятно, имели бы менее предвзятое представление о языке JavaScript, если бы сохранилось его первоначальное название *LiveScript* или же если бы ему было присвоено другое название, в меньшей степени вводящее в заблуждение. Ведь как гласит старая шутка, наглядно показанная на рис. 3.1, JavaScript имеет такое же отношение к Java, как и гамбургер к ветчине.

Совет

Подробнее о том, как язык JavaScript получил свое нынешнее название, можно узнать по следующим адресам: `http://ru.wikipedia.org/wiki/JavaScript#History`, `http://web.archive.org/web/20070916144913/http://wp.netscape.com/newsref/pr/newsrelease67.html` и

`http://stackoverflow.com/questions/2018731/why-is-javascript-called-javascript-since-it-has-nothing-to-do-with-java`. Ознакомившись с материалами, приведенными по указанным ссылкам, вы обнаружите, что создатели JavaScript намеревались определить этот язык в качестве *дополнения* к Java, а не чего-то, имеющего общие с Java характеристики.

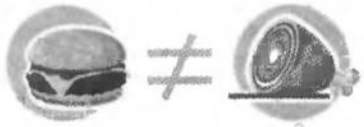

Рис. 3.1. JavaScript имеет такое же отношение к Java, как и гамбургер к ветчине. И то и другое приятно на вкус, а общее у них только сходство в названии: ham в английском слове hamburger означает — ветчина, хотя оно на самом деле происходит от названия немецкого города Гамбург

Как гамбургеры и ветчина относятся к мясной пище, так и JavaScript и Java относятся к языкам программирования с синтаксисом, созданным под влиянием языка C. Но помимо этого, у них нет ничего общего, и они в корне отличаются друг от друга.

Примечание

Еще один фактор, оказывающий влияние на неверное первоначальное представление разработчиков о JavaScript, состоит в том, что большинство из них знакомятся с JavaScript в браузере. Вместо того чтобы реагировать на JavaScript как на *язык программирования*, они, вероятно, исходят из привязок JavaScript к прикладному интерфейсу DOM API. А ведь DOM API — далеко не самый дружественный прикладной программный интерфейс, хотя обвинять в этом JavaScript не приходится.

Прежде чем перейти к рассмотрению ключевого понятия и роли функций в JavaScript, выясним причины, по которым столь важен функциональный характер JavaScript. И особенно это касается написания кода для браузеров.

Особое значение функционального характера JavaScript

Если у вас имеется некоторый опыт написания сценариев для браузеров, вам должно быть известно все, о чем речь пойдет в этом разделе. Но мы все равно обсудим особенности JavaScript, чтобы дальше говорить на общем и понятном для всех языке.

Одна из причин, по которым функции и функциональные понятия имеют особое значение в JavaScript, заключается в том, что функция является основным модульным исполняемым блоком. За исключением встраиваемого сценария, выполняющегося при вычислении разметки веб-документа, весь код сценария, который требуется написать для веб-страниц, обычно размещается в функции.

Примечание

Некогда встраиваемый сценарий применялся для повышения динамичности веб-страниц с помощью функции `document.write()`. А ныне функция

`document.write()` считается ископаемым, и пользоваться ею не рекомендуется. Для создания динамичных веб-страниц теперь имеются более совершенные способы, в том числе построение по шаблону на стороне сервера, манипулирование моделью DOM на стороне клиента или оптимальное сочетание того и другого.

Большая часть кода JavaScript выполняется в результате вызова функции, и поэтому наличие функций в качестве универсальных и эффективных конструкций дает немалые удобства и свободу действий при написании кода. В остальной части этой главы мы обсудим, каким образом характер функций как объектов высшего порядка можно использовать с наибольшей выгодой.

Мы уже не раз использовали в этой главе термин *объект высшего порядка*. Этот термин обозначает очень важное понятие, поэтому поясним сначала, что же оно обозначает.

Функции как объекты высшего порядка

В JavaScript объекты обладают определенными свойствами.

- Их можно создавать с помощью литералов.
- Их можно присваивать переменным, элементам массива и свойствам других объектов.
- Их можно передавать в качестве аргументов функциям.
- Их можно возвращать в качестве значений из функций.
- Они могут обладать свойствами, которые допускается создавать и присваивать динамически.

Функции в JavaScript обладают всеми этими свойствами, а следовательно, их можно рассматривать в данном языке как и любой другой объект. Именно поэтому функции и называются *объектами высшего порядка*. Но помимо того что к функциям следует относиться как и к другим типам объектов, они обладают особым свойством: их можно *вызывать*. Зачастую вызов функции осуществляется *асинхронно*, поэтому рассмотрим подробнее, как и почему это происходит.

Цикл ожидания событий в браузере

Если у вас имеется некоторый опыт программирования графических пользовательских интерфейсов для настольных приложений, вам должно быть известно, что они создаются по одному и тому же следующему образцу.

- Подготовить пользовательский интерфейс.
- Войти в цикл ожидания событий.
- Вызвать обработчики (или так называемые *приемники*) этих событий.

Программирование для браузеров отличается лишь тем, что код веб-приложения не отвечает за выполнение цикла ожидания и диспетчеризацию событий, поскольку браузер делает это автоматически. В обязанности разработчика входит настройка обработчиков на различные события, которые могут наступить в браузере. Эти события помещаются в очередь (список, действующий по принципу "первым пришел — первым обслужен") по мере их наступления, а браузер производит их диспетчеризацию, вызывая любые установленные для них обработчики.

Такие события обычно наступают в произвольные моменты времени и в непредсказуемом порядке. Именно поэтому их обработка, а следовательно, и вызов обрабатывающих функций происходят *асинхронно*. Среди прочих могут наступить следующие события.

- События в браузере, в том числе по окончании загрузки страницы или перед ее выгрузкой.
- События в сети, в том числе как реакция на Ajax-запрос.
- События, инициируемые пользователем, когда он, например, щелкает кнопкой мыши, перемещает мышь или нажимает клавиши на клавиатуре.
- События, инициируемые таймером, в том числе по истечении времени ожидания или заданного промежутка времени.

Большая часть кода веб-приложений выполняется в результате наступления подобных событий. Рассмотрим в качестве примера следующий фрагмент кода:

```
function startup(){
  /* сделать что-нибудь полезное */
}
window.onload = startup;
```

В этом фрагменте кода устанавливается функция, которая должна служить в качестве обработки события `load`, наступающего по окончании загрузки страницы. Установочный оператор выполняется как часть встраиваемого сценария, при условии, что он находится на верхнем уровне, а не в теле любой другой функции. Но самое замечательное, что действия, предусмотренные в *теле* функции, не будут выполнены до тех пор, пока браузер не завершит загрузку страницы и не инициирует событие `load`.

На самом деле приведенный выше фрагмент кода можно упростить до одной строки, как показано ниже.

```
window.onload = function() { /* сделать что-нибудь полезное */ };
```

(Если приведенная выше запись не совсем понятна вам, не смущайтесь — она будет подробно разъяснена далее в этой главе.)

Ненавязчивый JavaScript

Возможно, способ присваивания функции (по имени или иначе) свойству `onload` экземпляра объекта `window` совсем не похож на тот, которым вы обычно устанавливаете обработчик события, наступающего по окончании загрузки страницы. Вам, вероятно, привычнее пользоваться атрибутом `onload` дескриптора `<body>`. И хотя в любом случае достигается один тот же результат, тем не менее мастера программирования на JavaScript отдают предпочтение первому способу, используя свойство `onload` экземпляра объекта `window`, поскольку данный способ соответствует распространенной методике программирования под названием ***ненавязчивый JavaScript***.

Напомним, что с появлением вложенных таблиц стилей (CSS) данные стилевого оформления веб-страниц стали выноситься за пределы разметки документов. И вряд ли кто-нибудь станет спорить, что отделение стилевого оформления от структуры веб-документов было неудачной идеей. По тому же самому принципу действует и ненавязчивый JavaScript, вынося сценарии за пределы разметки документов.

В итоге страницы содержат три основные, изящно разделенные составляющие: структуру, стиль и режим работы. В частности, структура определяется в разметке документа, стиль — в элементах разметки `<style>` или внешних таблицах стилей, а режим работы — в блоках элементов разметки `<script>` или

в файлах внешних сценариев. В примерах, приведенных в этой книге, вы не обнаружите ни одного сценария, встроенного в разметку документа, кроме тех случаев, когда это имеет особый смысл или значительно упрощает пример.

Следует заметить, что цикл ожидания событий в браузере носит *однопоточный* характер. Каждое событие обрабатывается в том порядке, в каком оно помещается в очередь событий. Такая очередь организуется в виде списка, действующего по принципу "первым пришел — первым обслужен", а для программирующих со стажем — по более понятному для них принципу "силосной ямы". Таким образом, каждое событие обрабатывается в *свою* очередь, а все остальные события должны ждать до тех пор, пока не наступит *их* очередь. И ни при каких обстоятельствах два обработчика событий не могут одновременно выполняться в отдельных потоках.

В качестве аналогии рассмотрим очередь клиентов в банке. Все клиенты становятся в один ряд и должны ждать своей очереди на обслуживание банковскими служащими. Но ведь в JavaScript открыто только одно кассовое окно! И через него обслуживается только один клиент, который почему-то считает, что все планирование своих финансов на очередной бюджетный год он может сделать прямо у кассового окна, не считаясь со временем других клиентов, стоящих в очереди, и тем самым стопоря всю работу банка.

Более подробно такая модель выполнения, а также способы преодоления ее недостатков рассматриваются в главе 8. На рис. 3.2 приведена сильно упрощенная блок-схема данного процесса.

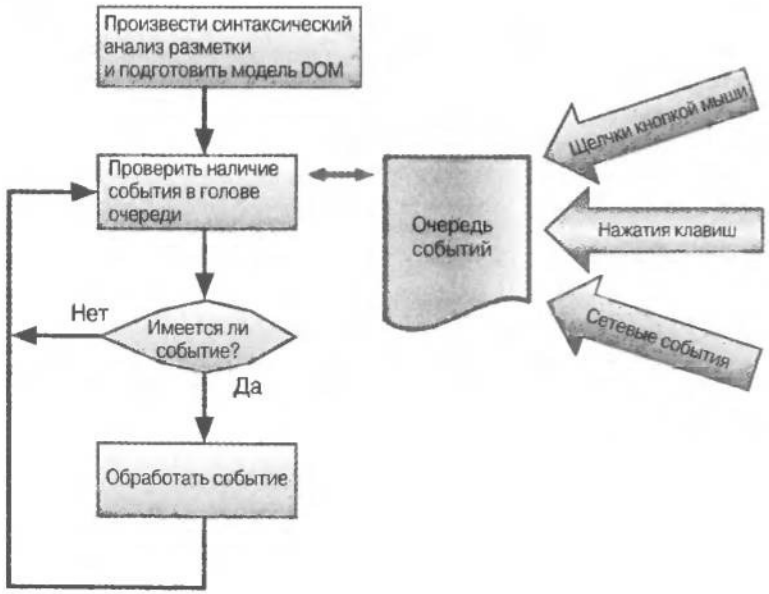

Рис. 3.2. Упрощенное представление процесса обработки, организуемого в цикле ожидания событий в браузере, где каждое событие обрабатывается по очереди в одном потоке

Рассматриваемый здесь принцип является центральным для страничных сценариев JavaScript. Он еще не раз будет встречаться в примерах, приведенных в этой книге, а суть его состоит в следующем: код подготавливается заранее для последующего

выполнения. За исключением встраиваемого установочного кода, в подавляющем большинстве код, размещаемый на веб-странице, должен выполняться в результате наступления определенного события, как показано в прямоугольнике "Обработать событие" на рис. 3.2.

Следует особо подчеркнуть, что механизм браузера, помещающий события в очередь, является внешним по отношению к рассматриваемой здесь модели цикла ожидания событий. Обработка, необходимая для определения момента наступления событий и размещения их в очереди, *не* происходит в потоке, где события обрабатываются.

Например, когда конечный пользователь перемещает курсор мыши по веб-странице, браузер обнаруживает эти действия, помещая целый ряд событий `mousemove` в очередь. Эти события затем обнаруживаются в цикле ожидания событий, где для их обработки запускаются любые обработчики, установленные для данного типа событий. Такие обработчики событий служат характерными примерами более общего понятия *функций обратного вызова*. Поэтому рассмотрим это очень важное понятие более подробно.

Принцип обратного вызова

Всякий раз, когда функция устанавливается для последующего вызова, будь то из браузера или из другого кода, это означает, что, по существу, подготавливается *обратный вызов*. Своим происхождением этот термин обязан тому факту, что функция устанавливается для последующего "обратного ее вызова" из какого-нибудь другого кода в подходящий момент выполнения. Обратные вызовы являются важной составляющей эффективного использования JavaScript, а их применение будет показано далее на конкретном практическом примере. Но это непростой пример, поэтому, прежде чем переходить к нему, внимательно разберем принцип обратного вызова в простейшей его форме.

Как будет показано далее в этой книге, функции обратного вызова широко используются в качестве обработчиков событий, но это лишь один пример их применения. Обратные вызовы можно даже внедрять в свой код. Ниже приведен совершенно бесполезный пример одной функции, принимающей в качестве своего параметра ссылку на другую функцию для ее последующего вызова. Тем не менее этот пример наглядно демонстрирует принцип действия обратного вызова.

```
function useless(callback) { return callback(); }
```

Какой бы бесполезной эта функция ни оказалась, она все же помогает ясно понять, каким образом одна функция сначала передается другой функции в качестве аргумента, а затем вызывается через переданный параметр. Рассматриваемую здесь бесполезную функцию можно проверить с помощью следующего кода:

```
var text = 'Domo arigato!';
assert(useless(function(){ return text; }) === text,
      "The useless function works! " + text);
```

В этом фрагменте кода функция тестирования `assert()`, подготовленная нами в предыдущей главе, используется для проверки того факта, что функция обратного вызова действительно вызывается и возвращает предполагаемое значение, которое, в свою очередь, оказывается бесполезным (`"Domo arigato!"` по-японски означает "Большое спасибо!"). Результат тестирования бесполезной функции приведен на рис. 3.3.

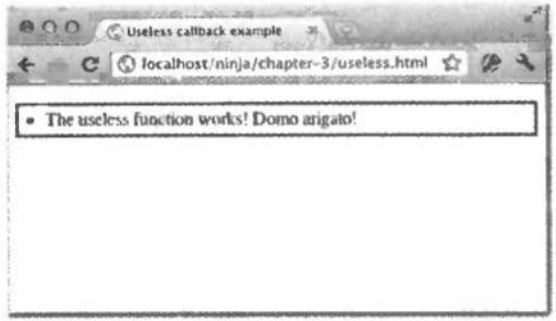

Рис. 3.3. Бесполезная функция мало на что пригодна, тем не менее она наглядно демонстрирует, что функции можно передавать и затем вызывать в любой момент

Итак, мы рассмотрели самый простой пример организации обратного вызова. Но он оказался таковым именно потому, что функциональный характер JavaScript позволяет обращаться с функциями как с объектами высшего порядка. А теперь перейдем к рассмотрению более сложного и полезного примера и сравним его с применением обратных вызовов в языке нефункционального программирования.

Сортировка по алгоритму сравнения

Практически всегда, когда имеется коллекция данных, ее нужно каким-то образом отсортировать. А для выполнения простейших операций сортировки, как оказывается, вполне подходит обратный вызов. Допустим, имеется следующий массив чисел, расположенных в произвольном порядке: **213, 16, 2058, 54, 10, 1965, 57, 9**. Такой порядок расположения чисел может оказаться вполне подходящим, но рано или поздно числа, скорее всего, придется отсортировать в каком-нибудь другом порядке.

В обоих языках, Java и JavaScript, предоставляются простые средства для сортировки массивов по возрастающей. Ниже показано, как это делается в коде Java.

```
Integer[] values = { 213, 16, 2058, 54, 10, 1965, 57, 9 };
Arrays.sort(values);
```

А вот вариант той же самой операции сортировки в коде JavaScript:

```
var values = [ 213, 16, 2058, 54, 10, 1965, 57, 9 ];
values.sort();
```

Примечание

На самом деле мы не имеем ничего против Java. Ведь это отличный язык программирования. Мы вполне оправданно выбрали Java лишь потому, что это характерный пример языка *без* возможностей для функционального программирования. К тому же этот язык знаком многим разработчикам, переходящим на JavaScript.

В реализациях сортировки на этих языках имеются некоторые, хотя и минимальные отличия. В частности, для этой цели в Java предоставляется служебный класс со статическим методом, тогда как в JavaScript – метод для самого массива, но оба подхода к сортировке просты и понятны. Когда же дело доходит до более сложной сортировки,

чем по возрастающей или по убывающей, то отличия в ее реализациях на обоих языках становятся более заметными и даже разительными.

Для того чтобы отсортировать числовые значения в *любом* требующемся порядке, в обоих языках предоставляется алгоритм сравнения, который предписывает алгоритму сортировки порядок расположения значений. Вместо того чтобы просто предоставить алгоритму сортировки возможность самому выяснять, в каком именно порядке должны следовать числовые значения, мы создадим функцию, выполняющую сравнение, обеспечив алгоритму сортировки доступ к ней в форме обратного вызова. Следовательно, она будет вызываться из этого алгоритма всякий раз, когда возникнет потребность в сравнении. Этот принцип одинаков для обоих языков, но его реализация существенно разнится.

В языке нефункционального программирования Java методы не могут существовать самостоятельно и передаваться в качестве аргументов другим методам. Вместо этого они должны быть объявлены в качестве членов объекта, экземпляр которого можно получить и передать другому методу. Поэтому у метода `Arrays.sort()` имеется перегружаемый вариант, который принимает объект, содержащий метод сравнения, обратно вызываемый всякий раз, когда требуется выполнить сравнение. Этот объект и его метод соответствуют известному формату, поскольку Java – строго типизированный язык, а это означает, что нам потребуется определить интерфейс. В данном случае в библиотеке Java предоставляется следующий интерфейс, хотя, как правило, его придется определять самостоятельно:

```
public interface Comparator<T> {
  int compare(T t, T t1);
  boolean equals(Object o);
}
```

Начинающий программировать на Java мог бы создать конкретный класс, реализующий этот интерфейс, но для целей изящного сравнения мы допустим наличие более высокого уровня владения языком Java и воспользуемся встроенной реализацией анонимного интерфейса. В приведенном ниже коде показано, как можно воспользоваться статическим методом `Arrays.sort()` для сортировки числовых значений по убывающей.

```
Arrays.sort(values, new Comparator<Integer>(){
  public int compare(Integer value1, Integer value2) {
    return value2 - value1;
  }
});
```

Метод `compare()` из встраиваемой реализации интерфейса `Comparator` должен возвращать отрицательное число, если порядок следования передаваемых ему числовых значений следует изменить на обратный, положительное число, если этот порядок требуется сохранить, и нуль, если сравниваемые значения равны. Таким образом, простое вычитание числовых значений дает желаемое возвращаемое значение для сортировки массива по убывающей. В результате выполнения приведенного выше кода получается массив, отсортированный следующим образом:

```
2058, 1965, 213, 57, 54, 16, 10, 9
```

Это была не самая трудная по своему характеру задача, но для ее решения потребовалось немало синтаксиса, особенно если включить в код объявление требующегося

интерфейса. Многословность такого подхода к программированию становится еще более очевидной при рассмотрении равнозначного кода JavaScript, в котором выгодно используются преимущества функциональных возможностей JavaScript:

```
var values = [ 213, 16, 2058, 54, 10, 1965, 57, 9 ];
values.sort();
```

В этом коде нет ни интерфейсов, ни лишних объектов. В одной строке просто объявляется встраиваемая анонимная функция, которая передается непосредственно методу `sort()` сортируемого массива.

Функциональные отличия JavaScript позволяют нам сначала создать функцию в виде автономного объекта, а затем передать ее в качестве аргумента методу, воспринимающему ее как параметр. И все это можно проделать с функцией таким же образом, как и с объектом любого другого типа, благодаря тому, что она относится к категории объектов высшего порядка. Ничего подобного и близко не допускается в таких языках нефункционального программирования, как Java.

Примечание

Вполне вероятно, что возможности для функционального программирования будут внедрены в версии Java 8 в виде лямбда-выражений, а до тех пор Java остается языком нефункционального программирования. Если же вам нравится программировать на Java, но требуются средства функционального программирования, попробуйте освоить Groovy. Этот язык поддерживает динамическую компиляцию в виртуальной машине JVM и обладает возможностями функционального программирования в Java-подобном стиле. В последнее время он находит все большее распространение благодаря интегрированной среде Grails для разработки веб-приложений.

К числу самых важных свойств языка JavaScript относится возможность создавать функции в любом месте кода, где появляется выражение. Помимо возможности сделать код более компактным и простым для понимания (благодаря расположению объявления функции рядом с тем местом, где она применяется), это свойство языка позволяет также избежать засорения глобального пространства имен ненужными именами, когда отсутствует обращение к функции из многих мест в коде.

Но каким бы способом ни объявлялись функции (подробнее об этом — в следующем разделе), на них можно ссылаться как на обычные значения и использовать в качестве стандартных блоков, полагаемых в основу построения библиотек повторно используемого кода. Ясное представление о том, как действуют функции, в том числе и анонимные, на самом элементарном уровне, позволяет радикально усовершенствовать навыки написания ясного, краткого и повторно используемого кода.

А теперь перейдем к более подробному рассмотрению того порядка, в котором объявляются и вызываются функции. На первый взгляд, в объявлении и вызове функций нет ничего особенного, но на самом деле в этих операциях происходит немало такого, что нужно и полезно знать.

Объявление функций

Функции в JavaScript объявляются с помощью *функционального литерала*, создающего значение функции таким же образом, как и числовой литерал — числовое значение. Напомним, что функции как объекты высшего порядка являются значениями, которыми можно пользоваться в языке точно так же, как и любыми другими типами значений, в том числе строковыми и числовыми. И сознательно или бессознательно, но вам приходилось делать это все время, программируя на JavaScript.

Функциональные литералы состоят из следующих четырех частей.

1. Ключевое слово `function`.
2. *Необязательное* имя, которое может быть достоверным идентификатором JavaScript, если оно указывается.
3. Разделяемый запятыми список имен параметров, заключенный в круглые скобки. Имена параметров должны быть достоверными идентификаторами, а их список может быть пустым. Круглые скобки следует указывать всегда, даже если список параметров пуст.
4. Тело функции, состоящее из последовательного ряда операторов JavaScript, заключенных в фигурные скобки.

Тело функции может быть пустым, но фигурные скобки должны быть указаны всегда. А тот факт, что имя функции указывать необязательно, может удивить некоторых разработчиков, но в предыдущем разделе мы уже подробно рассматривали некоторые примеры применения подобных *анонимных функций*. Если обращаться к функции по имени нет никакой нужды, то можно и не указывать ее имя. (Это сродни следующей шутке о котах: зачем давать коту имя, если он все равно не отзывается на него?)

Когда функция именуется, ее имя остается действительным в той области действия, в которой она объявлена. Кроме того, если именованная функция объявляется на самом верхнем уровне, то на основании ее имени создается свойство в объекте `window`, ссылающемся на эту функцию. И наконец, у всех функций имеется свойство `name`, в котором имя функции хранится в виде символьной строки. Но и безымянные функции обладают свойством `name`, в котором устанавливается пустая символьная строка.

Сказанное выше уместно подтвердить непосредственно в коде. В частности, мы можем написать тесты, подтверждающие истинность всего, что было сказано выше о функциях. Рассмотрим в качестве примера код, приведенный в листинге 3.1.

Листинг 3.1. Подтверждение порядка объявления функций

```
<script type="text/javascript">

  function isNimble(){ return true; }

  assert(typeof window.isNimble === "function",
         "isNimble() defined");
  assert(isNimble.name === "isNimble",
         "isNimble() has a name");

  var canFly = function(){ return true; };

  assert(typeof window.canFly === "function",
         "canFly() defined");
  assert(canFly.name === "",
         "canFly() has no name");
```

1. Объявить имя функции. Имя функции доступно в текущей области действия и неявно вводится как свойство объекта `window`
2. В первом тесте утверждается, что свойство объекта `window` установлено, а во втором — что свойство `name` функции записано
3. Создать анонимную функцию, присваиваемую переменной `canFly`. Эта переменная является свойством объекта `window`, а свойство `name` функции пусто
4. Проверить, ссылается ли переменная на анонимную функцию и установлена ли в свойстве `name` пустая строка, а не пустое значение

```
window.isDeadly = function(){ return true; };

assert(typeof window.isDeadly === "function",
       "isDeadly() defined");

function outer(){
  assert(typeof inner === "function",
         "inner() in scope before declaration");
  function inner(){}
  assert(typeof inner === "function",
         "inner() in scope after declaration");
  assert(window.inner === undefined,
         "inner() not in global scope");
}

outer();
assert(window.inner === undefined,
       "inner() still not in global scope");

window.wieldsSword = function swingsSword() { return true; };

assert(window.wieldsSword.name === 'swingsSword',
       "wieldSword's real name is swingsSword");

</script>
```

❺ Создать анонимную функцию со ссылкой из свойства объекта window

❻ Проверить, делается ли в свойстве объекта window ссылка на функцию. Здесь можно также проверить, пусто ли свойство name данной функции

❼ Определить одну функцию внутри другой. Проверить, что к функции inner() можно обращаться до и после ее объявления и что для нее не создано никакого глобального имени

❽ Проверить, что к функции outer() можно обращаться в глобальной области действия, а к функции inner() — нельзя

Переменная, которой присваивается функция, никак не связана с ее именем, а контролируется это именованием функции в ее литерале

На тестовой странице, рассматриваемой здесь в качестве примера, функции объявлены в глобальной области действия тремя разными способами.

- Функция `isNimble()` объявляется как именованная ❶. Это едва ли не самый распространенный способ объявления функции, известный большинству разработчиков. Но ваше представление о нем постепенно изменится по ходу чтения книги.
- Анонимная функция создается и присваивается глобальной переменной `canFly` ❸. Благодаря функциональному характеру JavaScript функцию можно вызвать по ссылке на эту переменную следующим образом: `canFly()`. И это *почти* функционально (без всякого каламбура), хотя и не полностью равнозначно объявлению именованной функции `canFly`. Главное отличие заключается в том, что свойство `name` анонимной функции содержит пустую символьную строку `" "`, а не строку `"canFly"`.
- Еще одна анонимная функция объявляется и присваивается свойству `isDeadly` объекта `window` ❺. И в этом случае функцию можно вызвать через данное свойство одним из следующих двух способов: `window.isDeadly()` или просто `isDeadly()`. Это опять же *почти* функционально равнозначно объявлению именованной функции `isDeadly`.

В разных местах рассматриваемого здесь примера кода были расположены утверждения, проверяющие истинность всего сказанного выше о функциях. А результаты выполнения тестов приведены на рис. 3.4.

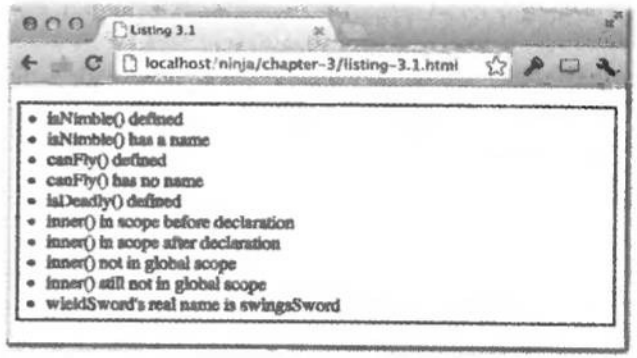

Рис. 3.4. Результаты выполнения тестовой страницы подтверждают истинность всего, что было сказано ранее о функциях!

Таким образом, упомянутые выше тесты подтверждают следующее.

- Конструкция `window.isNimble` определяется как функция. Этим подтверждается, что именованные функции вводятся как свойства в объект `window` ❷.
- У именованной функции `isNimble()` имеется свойство `name`, содержащее символьную строку `"isNimble"` ❷.
- Конструкция `window.canFly` определяется как функция. Этим подтверждается, что глобальные переменные (даже те, что содержат функции) в конечном итоге оказываются в объекте `window` ❸.
- У анонимной функции, присвоенной переменной `canFly`, имеется свойство `name`, содержащее пустую символьную строку ❹.
- Конструкция `window.isDeadly` определяется как функция ❺.

Примечание

Это далеко не полный тестовый набор, чтобы проверить все, что было сказано до сих пор о функциях. Подумайте, как расширить этот тестовый набор, чтобы подтвердить высказанные предположения относительно объявляемых функций.

Затем наступает черед для проверки неглобальных функций. С этой целью создается внешняя функция `outer()`, в которой проверяются утверждения относительно функций, объявляемых в неглобальной области действия ❼. Далее объявляется внутренняя функция `inner()`, но прежде утверждается, что функция находится в своей области действия. Этим проверяется утверждение, что функция доступна в той области действия, где она объявлена, даже если ссылка на нее делается с упреждением.

После этого объявляется внутренняя функция и проверяется, что она находится в области действия внешней функции, а не в глобальной области действия. И наконец, выполняется внутренний тест и еще раз утверждается, что внутренняя функция не выходит за свои пределы в глобальную область действия ❽.

Рассмотренные здесь понятия очень важны, поскольку они закладывают основание для именования, последовательности выполнения и структуры функционального кода. И они начинают устанавливать рамки, в которых возможности функционального программирования используются с наибольшей выгодой.

Утверждение, сделанное относительно внутренней функции, а именно: ссылку на внутреннюю функцию допускается с упреждением делать во внешней функции ❸, может вызвать недоуменный вопрос: "Когда объявляется функция, то в какой именно области действия она доступна?" Это вполне уместный вопрос, а ответ на него дается ниже.

Определение областей действия и функции

При объявлении функции необходимо позаботиться не только о той области действия, в которой данная функция доступна, но и о тех областях действия, которые образует *сама* функция, а также о тех объявлениях в теле функции, на которые оказывают влияние эти области действия. В JavaScript области действия ведут себя несколько иначе, чем в большинстве других языков программирования, на синтаксис которых оказал влияние язык C. К их числу относятся языки, в которых для ограничения кодового блока употребляются фигурные скобки { и }. В большинстве таких языков программирования каждый кодовый блок образует свою область действия, а в JavaScript это происходит по-другому.

В JavaScript области действия определяются *функциями*, а не кодовыми блоками. Область действия объявления, сделанного в пределах кодового блока, не оканчивается вместе с этим блоком, как это происходит в других языках программирования. Рассмотрим для примера следующий фрагмент кода:

```
if (window) {
  var x = 213;
}
alert(x);
```

В большинстве других языков программирования область действия объявленной переменной x должна завершиться в конце кодового блока, образуемого условным оператором `if`, и поэтому функция `alert()` получит неопределенное значение. Но если попытаться выполнить приведенный выше фрагмент кода на веб-странице, то появится предупреждающее сообщение со значением 213, поскольку области действия в JavaScript не завершаются по окончании кодовых блоков.

Казалось бы, все достаточно просто, но в правилах определения областей действия имеется ряд тонкостей, которые проявляются в зависимости от того, что именно объявляется. Некоторые из этих тонкостей могут даже оказаться неожиданными.

- Область действия объявляемых переменных распространяется от места их объявления и до окончания функции, в которой они объявлены, независимо от степени вложенности кодовых блоков.
- Область действия именованных функций полностью находится в пределах той функции, в которой они объявлены, независимо от степени вложенности кодовых блоков. (Некоторые называют этот механизм *подъемным*.)
- Для областей действия объявлений глобальный контекст служит в качестве одной большой функции, охватывающей весь код на странице.

И в этом случае за подтверждением всего сказанного выше обратимся непосредственно к коду. В качестве примера рассмотрим следующий фрагмент кода:


```
function outer(){

  var a = 1;

  function inner(){ /* ничего не делает */ }

  var b = 2;

  if (a == 1) {
    var c = 3;
  }

}

outer();
```

В этом фрагменте кода объявляются пять элементов: внешняя функция `outer()`, внутренняя функция `inner()`, а также три числовые переменные, `a`, `b`, и `c`, во внешней функции. Для того чтобы проверить, находятся ли различные элементы в области их действия, а еще важнее – не выходят ли они за ее пределы, распределим блок тестов по всему этому коду. Это будет один и тот же блок тестов, располагаемых в стратегических важных местах кода: по одному на каждое объявление проверяемых элементов. В каждом тесте (кроме первого, который вообще не является тестом, но служит лишь обычной меткой для удобства чтения кода и результатов его выполнения) утверждается, что один из элементов объявлен в области своего действия. Ниже приведен этот блок тестов.

```
assert(true,"some descriptive text");
assert(typeof outer==='function',
      "outer() is in scope");
assert(typeof inner==='function',
      "inner() is in scope");
assert(typeof a==='number',
      "a is in scope");
assert(typeof b==='number',
      "b is in scope");
assert(typeof c==='number',
      "c is in scope");
```

Следует заметить, что во многих случаях некоторые из приведенных выше тестов не проходят. В обычных условиях следует ожидать, что сделанные в них утверждения будут всегда проходить проверку, но в данном коде, который служит лишь для демонстрации, это вполне отвечает нашим целям – показать, где тесты проходят и где они не проходят, что непосредственно указывает, находится ли проверяемый элемент в области своего действия или нет.

В листинге 3.2 показан весь собранный для наших целей код, исключая повторяющийся тест, чтобы за деревьями был виден лес. (В тех местах, где тестовый код удален, присутствует комментарий `/* здесь следует тестовый код */`, чтобы было понятно, где именно должен находиться тестовый код в файле конкретной страницы.)

Листинг 3.2. Наблюдение за поведением областей действия объявлений

```
<script type="text/javascript">

  assert(true,"|----- BEFORE OUTER -----|");
  /* здесь следует тестовый код */

  function outer(){
  assert(true,"|----- INSIDE OUTER, BEFORE a -----|");
  /* здесь следует тестовый код */

  var a = 1;

  assert(true,"|----- INSIDE OUTER, AFTER a -----|");
  /* здесь следует тестовый код */

  function inner(){ /* ничего не делает */ }

  var b = 2;

  assert(true,"|----- INSIDE OUTER, AFTER inner() AND b -----|");
  /* здесь следует тестовый код */

  if (a == 1) {
    var c = 3;
    assert(true,"|----- INSIDE OUTER, INSIDE if -----|");
    /* здесь следует тестовый код */
  }

  assert(true,"|----- INSIDE OUTER, OUTSIDE if -----|");
  /* здесь следует тестовый код */
  }

  outer();

  assert(true,"|----- AFTER OUTER -----|");
  /* здесь следует тестовый код */

</script>
```

Выполнить тестовый блок, прежде чем определять что-нибудь. Во всех тестах утверждается, что каждый элемент находится в области своего действия, поэтому не пройдут все тесты, кроме тестов тех элементов, ссылаться на которые можно с упреждением. На данный момент в области своего действия оказывается только функция outer(). Для того чтобы убедиться в этом, см. рис. 3.5, а еще лучше выполните код в браузере, чтобы не пролистывать страницы

Выполнить блок тестов в функции outer(), прежде чем объявлять что-нибудь. Функция outer() по-прежнему находится в области своего действия, как и определенная в ней функция inner(). Ссылаться с упреждением можно на функции, но не на объявляемые переменные, поэтому все остальные тесты не проходят

Выполнить блок тестов в функции outer(), но после объявления переменной. Результаты тестирования показывают, что в данный момент переменная a введена в область своего действия

Выполнить тестовый код после объявления функции inner() и переменной b. Результаты тестирования показывают, что переменная b введена в область своего действия. Факт объявления функции inner() в данный момент остается спорным. Область ее действия простирается к началу содержащей ее функции, а объявление, безусловно, не исключает ее из области действия

Выполнить тесты в блоке условного оператора if после объявления в нем переменной c. Результаты тестирования показывают, что в данный момент все элементы находятся в области своего действия

Выполнить тестовый код в функции outer(), но после завершения блока условного оператора if. Результаты тестирования показывают, что все элементы находятся в области их действия (даже переменная c), хотя блок условного оператора if, в котором она объявлена, завершен. В отличие от других блочных языков, область действия переменных в JavaScript простирается от места их объявления до конца функции, пересекая границы любых блоков

Выполнить тесты в глобальной области действия после объявления функции outer(). Опять же лишь функция outer() оказывается в области своего действия, так как область действия всего, что объявлено в ней, ограничивается ее пределами

Результаты выполнения приведенного выше кода представлены на рис. 3.5.

Как и следовало ожидать, многие тесты не проходят, поскольку не все элементы находятся в области своего действия на каждом месте расположения блока тестов. Обратите особое внимание на доступность (*подъем*) объявления функции `inner()` в пределах всей функции `outer()`, тогда как числовые переменные a, b и c доступны только от места их объявления и до конца функции `outer()`. Это явно указывает на то, что ссылки на объявленные функции можно делать с упреждением в пределах их области действия, а на переменные этого делать нельзя.

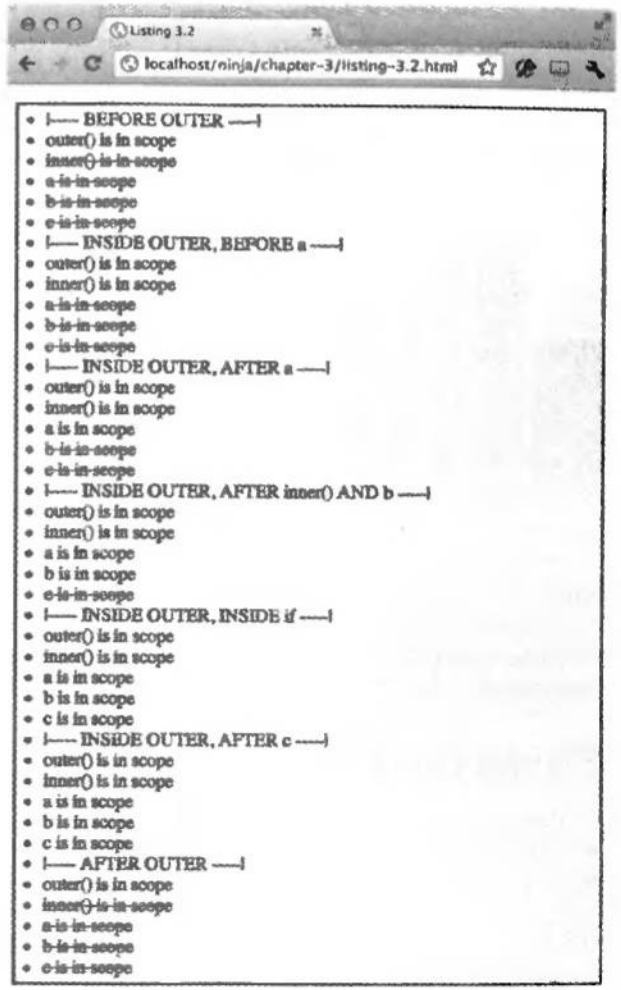

Рис. 3.5. Результаты тестирования областей действия объявляемых элементов наглядно показывают, где эти элементы оказываются в области своего действия и где они выходят за ее пределы

Обратите также особое внимание на то, что окончание блока условного оператора `if`, в котором объявлена переменная c, не прерывает область ее действия. Переменная c, несмотря на то что она вложена в этот кодовый блок, остается доступной от места ее объявления и до конца функции `outer()`, как, впрочем, и остальные переменные, не определенные в данном блоке. Области действия различных объявленных элементов графически показаны на рис. 3.6.

Информация к размышлению

Теперь, когда вы поближе ознакомились с областью действия элементов кода JavaScript, попробуйте найти ответ на следующий вопрос: почему бы не создать одну функцию, которая содержала и вызывала бы весь блок тестов по мере надобности, вместо того чтобы неоднократно вырезать и вставлять тесты в нужных местах кода? Ответа на этот вопрос мы не даем, так что думайте сами.

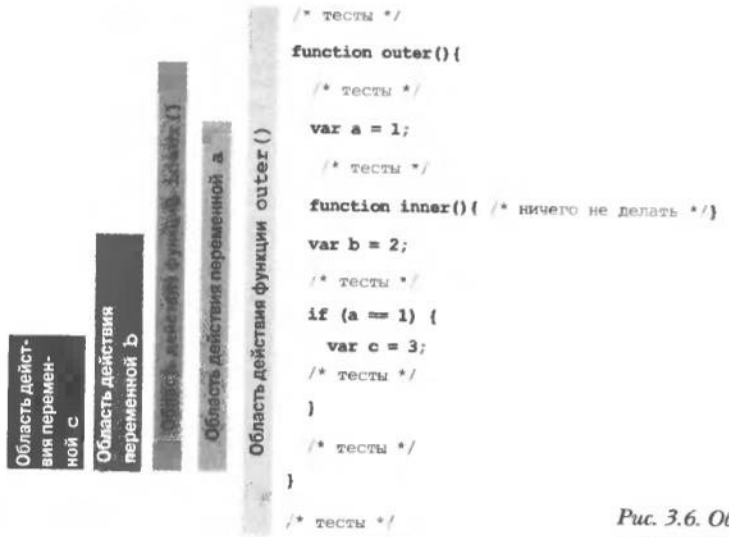

Рис. 3.6. Область действия каждого объявленного элемента кода зависит не только от места его объявления, но и от вида самого элемента, будь то переменная или функция

Итак, рассмотрев порядок объявления функций, перейдем к обсуждению способов их вызова.

Вызов функций

Вам, вероятно, не раз приходилось вызывать функции в коде JavaScript, но задумывались ли вы над тем, что в действительности происходит при вызове функции? В этом разделе мы рассмотрим различные способы вызова функций.

Оказывается, что способ вызова функции существенно влияет на порядок выполнения кода в ней, и особенно на установку параметра `this`. Это отличие имеет намного большее значение, чем кажется на первый взгляд. Поэтому мы подробно рассмотрим его в этом разделе, чтобы выгодно воспользоваться им в остальной части книги и тем самым помочь вам повысить свои навыки программирования на JavaScript до уровня мастера. Итак, имеются четыре разных способа вызова функции, каждому из которых присущи свои особенности.

- **В виде функции.** В этом случае функция вызывается непосредственно.
- **В виде метода.** В этом случае вызов привязывается к объекту, допуская объектно-ориентированное программирование.
- **В виде конструктора.** В этом случае создается новый объект.
- **С помощью методов `apply()` и `call()`.** Это более сложный способ, поэтому мы рассмотрим его, когда до него дойдет черед.

Во всех перечисленных выше способах, кроме последнего, оператор вызова функции представляет собой пару круглых скобок, следующих после выражения, в котором вычисляется ссылка на функцию. Любые аргументы, передаваемые функции, заключаются в эту пару круглых скобок и разделяются запятыми. Ниже приведена эта общая форма вызова функции.


```
выражение(аргумент 1, аргумент 2);
```

Прежде чем перейти к более подробному обсуждению упомянутых выше четырех способов вызова функций, рассмотрим, что же происходит с аргументами, которые передаются вызываемой функции.

От аргументов к параметрам функций

Когда в вызове функции предоставляется список аргументов, эти аргументы присваиваются параметрам, задаваемым в объявлении функции в том порядке, в каком они указаны: первый аргумент присваивается первому параметру, второй аргумент второму параметру и т.д. Если же число аргументов отличается от числа параметров, то никакой ошибки не происходит. В JavaScript подобная ситуация разрешается следующим образом.

- Если в вызове функции предоставляется больше аргументов, чем задано параметров, "лишние" аргументы просто не присваиваются именам параметров. Допустим, имеется функция, объявленная следующим образом:

  ```
  function whatever(a,b,c) { ... }
  ```

- Если при ее вызове указать `whatever(1,2,3,4,5)`, то аргументы 1, 2 и 3 будут присвоены параметрам `a`, `b` и `c` соответственно. В то же время аргументы **4** и **5** не будут присвоены ни одному из параметров данной функции. Несколько ниже будет показано, что даже если аргументы не присваиваются именам параметров, они по-прежнему остаются доступными.
- Если же у функции имеется больше параметров, чем передано ей аргументов, то параметрам без соответствующих аргументов присваивается значение `undefined` (не определено). Так, если при вызове функции `whatever(a,b,c)` указать `whatever(1)`, ее параметру будет присвоено значение **1**, а параметрам `b` и `c` — значение `undefined`.

Любопытно, что при вызове всех функций передаются два неявных параметра: `arguments` и `this`. Под словом *неявный* здесь подразумевается, что такие параметры не перечисляются явно в сигнатуре функции, но по умолчанию передаются ей и находятся в области ее действия. К ним можно обращаться в самой функции таким же образом, как и к любым другим явно именованным параметрам. Рассмотрим каждый из этих неявных параметров по очереди.

Параметр `arguments`

Этот параметр представляет собой коллекцию всех аргументов, передаваемых функции. У такой коллекции имеется свойство `length`, содержащее количество аргументов, а значения отдельных аргументов могут быть получены путем индексирования массива. Так, по индексу `arguments[2]` из массива будет извлечен третий параметр функции.

Следует, однако, иметь в виду, что обращаться к параметру `arguments` все же не рекомендуется. Вы можете неверно принять его за массив, поскольку у него имеется свойство `length`, а его элементы могут быть извлечены, как из массива. Более того, к ним можно по очереди обращаться в цикле `for`. Тем не менее это не типичный для JavaScript массив, и если вы попытаетесь применить к параметру `arguments` методы обработки массивов, то вас постигнет полное разочарование. Поэтому воспринимайте

параметр `arguments` просто как похожую на массив конструкцию и старайтесь пользоваться им как можно реже. Еще более любопытным оказывается рассматриваемый ниже параметр `this`.

Параметр `this`

Всякий раз, когда вызывается функция, помимо параметров, представляющих аргументы, явно указанные при вызове функции, ей также передается неявный параметр `this`. Этот параметр ссылается на объект, который неявно связан с вызываемой функцией и называется *контекстом функции*.

Те, кто перешел на JavaScript из таких языков объектно-ориентированного программирования, как Java, могут превратно трактовать понятие контекста функции, считая, что параметр `this` указывает на экземпляр объекта того класса, в котором определен данный метод. Но не следует забывать, что вызов функции в виде *метода* — это лишь один из четырех способов ее вызова. На самом деле то, на что указывает параметр `this`, определяется, в отличие от Java, не тем, как функция объявляется, а тем, как она *вызывается*. Вследствие этого параметр `this` было бы точнее назвать контекстом *вызова*, но нас, конечно, никто об этом не спрашивал.

Далее мы рассмотрим все четыре способа вызова функций и покажем, что они отличаются, главным образом, тем, как значение параметра `this` определяется в каждом способе вызова функций. После этого мы снова вернемся к контекстам функций, чтобы более основательно исследовать их. Поэтому не особенно тревожьтесь, если ясное представление о контексте функции и параметре `this` не сложилось у вас окончательно. Мы еще обсудим их более подробно, а до тех пор поясним, как вызываются функции.

Вызов в виде функции

Что же означает вызов в виде функции? Разумеется, функции вызываются как *функции*, и было бы неразумно думать иначе. Но в действительности термин "вызов в виде функции" выбран для того, чтобы каким-то образом провести различие между данным способом и другими механизмами вызова функций. Таким образом, если функция не вызывается в виде метода, конструктора или с помощью метода `apply()` или `call()`, это просто означает, что она вызывается "в виде функции".

Подобного рода вызов происходит в том случае, когда функция вызывается с помощью оператора `()`, а выражение, к которому этот оператор применяется, не ссылается на функцию как на свойство объекта. (В последнем случае функция вызывалась бы как метод, но об этом способе вызова функций речь пойдет несколько позже.) Ниже приведены простые примеры вызова в виде функции.

```
function ninja(){};
ninja();

var samurai = function(){};
samurai();
```

При подобном способе вызова контекст функции оказывается глобальным, а следовательно, им становится объект `window`. Мы пока что воздержимся от написания каких-либо тестов, подтверждающих данный факт, поскольку сделать это было бы намного интереснее в сравнении с чем-нибудь другим. На самом деле способ вызова функции в виде функции является частным случаем ее вызова в виде метода, рассматриваемого следующим по очереди. Но в силу того, что объект `window` неявно оказывается

"владельцем" функции, обычно принято считать, что это отдельный механизм, которым вы, вероятно, не раз пользовались, не особенно задумываясь о том, что происходит внутри этого механизма. Итак, рассмотрим далее, в чем же суть способа вызова функций в виде метода.

Вызов в виде метода

Когда функция присваивается свойству объекта и вызов происходит по ссылке на функцию с помощью этого свойства, функция вызывается в виде *метода* данного объекта. Ниже приведен пример такого вызова.

```
var o = {};
o.whatever = function(){};
o.whatever();
```

Что же из этого следует? Функция в данном случае называется "методом", но что в этом любопытного или полезного? Если у вас имеется некоторый опыт объектно-ориентированного программирования, то вы, вероятно, вспомните, что объект, к которому относится вызываемый метод, доступен в теле метода по оператору `this`. То же самое происходит и здесь. Когда функция вызывается как *метод* объекта, этот объект становится контекстом функции и доступен в ней через параметр `this`. И это одно из основных средств для написания на JavaScript объектно-ориентированного кода. (Другим средством является рассматриваемый далее конструктор.)

Сравним этот способ вызова с вызовом в виде функции, при котором функция определяется в контексте объекта `window` и вызывается без необходимости ссылаться на этот объект. Вызов в виде метода осуществляется таким же образом, за исключением того, что ссылка на объект делается *явно*, а не подразумевается как неявная ссылка на объект `window`. При вызове в виде функции последняя принадлежит объекту `window`, который становится контекстом функции. Аналогичным образом объект `o` в приведенном выше примере служит контекстом функции `whatever()`. Оба сравниваемых способа вызова функции, по существу, одинаковы, несмотря на указанные внешние отличия.

Рассмотрим тестовый код, приведенный в листинге 3.3, с целью продемонстрировать сходство и отличие способов вызова в виде функции и метода.

Листинг 3.3. Код, демонстрирующий сходство и отличие способов вызова в виде функции и метода

```
<script type="text/javascript">

  function creep(){ return this; }

  assert(creep() === window,
         "Creeping in the window");

  var sneak = creep;

  assert(sneak() === window,
         "Sneaking in the window");
```

❶ Определить функцию, возвращающую свой контекст. Это позволит исследовать контекст вне функции после ее вызова

❷ Проверить вызов в виде функции и объект `window` в качестве ее контекста (глобальной области действия). Как показано на рис. 3.7, этот тест проходит

❸ Создать ссылку на ту же самую функцию в переменной `sneak`

❹ Вызвать функцию, используя переменную `sneak`. Несмотря на использование переменной, вызов по-прежнему делается в виде функции, а ее контекстом остается объект `window`

```
var ninja1 = {
  skulk: creep
};

assert(ninja1.skulk() === ninja1,
       "The 1st ninja is skulking");

var ninja2 = {
  skulk: creep
};

assert(ninja2.skulk() === ninja2,
       "The 2nd ninja is skulking");

</script>
```

❺ Создать в переменной ninja1 объект со свойством skulk, ссылающимся на исходную функцию creep()

❻ Вызвать функцию через свойство skulk, т.е. в виде метода объекта ninja1. Теперь контекстом функции становится объект ninja1, а это уже объектно-ориентированный подход

❼ Создать еще один объект, ninja2, со свойством skulk, ссылающимся на функцию creep()

Вызвать функцию в виде метода объекта ninja2, который теперь становится ее контекстом

Как показано на рис. 3.7, все тестовые утверждения проходят успешно.

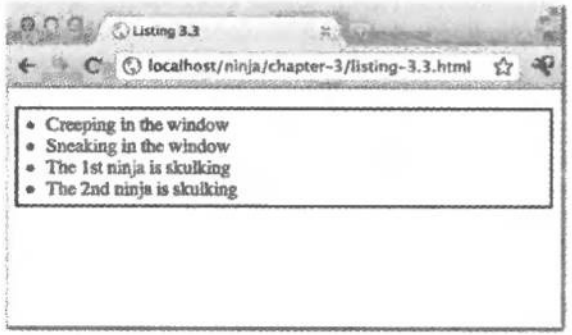

Рис. 3.7. Одна и та же функция, вызываемая разными способами, может служить в качестве обычной функции или метода

В рассматриваемом здесь тесте устанавливается единственная функция `creep()` ❶, используемая в остальной части проверяемого кода. Ее единственное назначение — возвратить свой контекст, чтобы во внешней функции можно было определить контекст вызываемой функции. Ведь другого способа выяснить его не существует.

Когда функция вызывается по имени, ее вызов осуществляется в виде функции. Следовательно, контекстом данной функции должен стать глобальный контекст, т.е. объект `window`. Для того чтобы убедиться в этом, делается соответствующее утверждение ❷, и, как показано на рис. 3.7, это утверждение успешно проходит.

Далее в переменной `sneak` создается ссылка на функцию ❸. Следует, однако, иметь в виду, что при этом создается не второй экземпляр функции, а только ссылка на ту же самую функцию. Как вам теперь уже известно, функции являются объектами высшего порядка.

Когда функция вызывается через переменную, ее вызов осуществляется в виде метода. К такому приему обычно прибегают потому, что оператор вызова функции можно применить к любому выражению, вычисление которого приводит к функции. И в этом случае, как и следовало ожидать, контекстом функции становится объект `window` ❹.

Далее код немного усложняется. В переменной `ninja1` определяется объект со свойством `skulk`, получающим ссылку на функцию `creep()` ❺. Следовательно, можно сказать, что для объекта создан *метод* под названием `skulk`. Но это совсем не означает, что функция `creep()` *становится* методом объекта `ninja1`. Как было показано выше, `creep()` — это независимая функция, которую можно вызвать самыми разными способами.

Как пояснялось ранее, при вызове функции по ссылке на метод следует ожидать, что контекстом вызываемой функции станет объект этого метода (в данном случае `ninja1`), что и утверждается в тесте ❻. И, как показано на рис. 3.7, этот факт, к счастью, подтверждается!

Данная конкретная возможность очень важна для программирования на JavaScript в объектно-ориентированном стиле. Это означает, что в любом методе можно воспользоваться параметром `this` для ссылки на объект, владеющий данным методом. И это один из основополагающих принципов объектно-ориентированного программирования. Для того чтобы данное положение стало более понятным, его проверка продолжена далее созданием еще одного объекта под названием `ninja2` с таким же самым свойством `skulk`, ссылающимся на функцию `creep()` ❼. После вызова метода для этого объекта совершенно правильно утверждается, что контекстом данной функции становится объект `ninja2`.

Следует заметить, что во всех рассмотренных выше примерах вызовов *одной и той же* функции ее контекст менялся в зависимости от того, как функция *вызывалась*, а не как она объявлялась. Например, оба объекта, `ninja1` и `ninja2`, обладают одним и тем же общим экземпляром функции, но при выполнении этой функции она имеет доступ и может выполнять операции над тем объектом, через который вызывается метод. Это означает, что для выполнения одной и той же обработки по разным объектам не нужно создавать отдельные копии функции, что составляет один из основных принципов объектно-ориентированного программирования.

Безусловно, это довольно эффективная возможность, но ее использование в данном примере имеет свои ограничения. Прежде всего, создав два объекта `ninja1` и `ninja2`, мы получили возможность воспользоваться одной и той же функцией в качестве метода каждого из них. Но при этом нам пришлось немного повторить код, чтобы установить отдельные объекты и их методы `skulk`.

Впрочем, это обстоятельство не должно вас обескураживать, ведь в JavaScript предоставляются механизмы, позволяющие создавать объекты по одному шаблону намного проще, чем в рассмотренном здесь примере. Все эти механизмы будут обсуждаться более подробно в главе 6, а до тех пор рассмотрим ту часть подобного механизма, которая относится к вызову функций. Речь далее пойдет о *конструкторе*.

Вызов в виде конструктора

В функции, которую предполагается использовать в качестве конструктора, нет ничего особенного. Функции-конструкторы объявляются таким же образом, как и любые другие функции, а отличаются они лишь способом своего вызова. В частности, для вызова функции в виде *конструктора* перед оператором ее вызова указывается ключевое слово `new`. Для примера обратимся к функции `creep()`, упоминавшейся в предыдущем разделе.

```
function creep(){ return this; }
```

Если функцию `creep()` требуется вызвать в виде конструктора, для этого достаточно написать следующую строку кода:

```
new creep();
```

Но даже если вызвать функцию `creep()` как конструктор, она окажется не совсем пригодной для применения в качестве конструктора. Попробуем выяснить причины этого, обсудив особенности конструкторов.

Сильные стороны конструкторов

Вызов функции в виде конструктора считается весьма эффективным языковым средством JavaScript, поскольку при вызове конструктора производятся следующие действия.

- Создается новый пустой объект.
- Этот объект передается конструктору в качестве параметра `this`, а следовательно, он становится контекстом функции данного конструктора.
- В отсутствие какого-либо явно возвращаемого значения новый объект возвращается в качестве значения конструктора.

И наконец, рассмотрим последнюю причину, по которой функцию `creep()` не стоит вызывать в виде конструктора. Назначение конструктора — создать новый объект, установить и возвратить его в качестве значения конструктора. Все, что мешает достижению этой цели, не годится для функций, предназначенных для применения в качестве конструкторов. В листинге 3.4 представлен пример более подходящей для этой цели функции. Эта функция устанавливает объекты скрывающихся ниндзя из листинга 3.3 в более краткой форме.

Листинг 3.4. Применение конструктора для установки общих объектов

```
<script type="text/javascript">

  function Ninja() {
    this.skulk = function() { return this; };
  }

var ninja1 = new Ninja();
var ninja2 = new Ninja();

assert(ninja1.skulk() === ninja1,
       "The 1st ninja is skulking");
assert(ninja2.skulk() === ninja2,
       "The 2nd ninja is skulking");

</script>
```

❶ Определить конструктор, создающий свойство skulk любого объекта, становящегося контекстом функции. И в этом случае метод возвращает контекст, чтобы проверить его вне функции

❷ Создать два объекта, вызвав конструктор с оператором new. Вновь созданные объекты обозначаются как ninja1 и ninja2

❸ Проверить методы сконструированных объектов. Каждый из них должен возвращать собственный сконструированный объект

Результат тестирования приведенного выше кода представлен на рис. 3.8.

В данном примере создается функция `Ninja()` ❶, предназначенная для конструирования объектов скрывающихся ниндзя. При ее вызове с ключевым словом `new` создается экземпляр пустого объекта, который передается функции в качестве параметра `this`. Конструктор создает для данного объекта свойство `skulk`, которому присваивается функция. В итоге это свойство превращается в метод вновь созданного объекта, выполняющий ту же самую операцию, что и упоминавшаяся ранее функция `creep()`, возвращая контекст функции, который можно проверить вне самой функции.

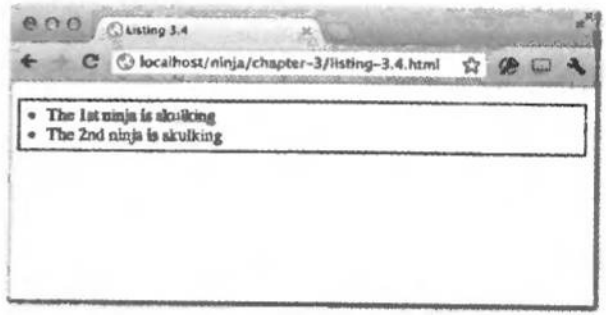

Рис. 3.8. Конструкторы позволяют без особых хлопот создавать многие объекты по одному и тому же шаблону

После определения конструктора создаются два новых объекта `Ninja` путем вызова этого конструктора дважды ❷. Следует иметь в виду, что значения, возвращаемые в результате вызовов конструктора, хранятся в переменных, которые превращаются в ссылки на вновь созданные объекты `Ninja`. Далее выполняются те же самые тесты, что и в листинге 3.3, чтобы проверить, что каждый вызов метода производится по предполагаемому объекту ❸.

Функции, предназначенные для применения в качестве конструкторов, обычно программируются иначе, чем другие функции. Ниже будет показано, как это делается.

Соображения по поводу программирования функций-конструкторов

Как пояснялось выше, назначение конструктора — инициализировать исходными условиями новый объект, который будет создан при вызове функции. А поскольку такие функции *можно* вызывать как обычные функции или даже присваивать их свойствам объектов для вызова в виде методов, то они, как правило, не совсем подходят для применения в качестве конструкторов. Например, функцию `Ninja()` вполне допустимо вызвать следующим образом:

```
var whatever = Ninja();
```

Но в итоге создается свойство `skulk` объекта `window`, который возвращается и сохраняется в переменной `whatever`. Очевидно, что польза от такой операции не особенно велика.

Функции-конструкторы обычно программируются и применяются совсем иначе, чем другие функции, а пользы от них зачастую немного, если только они не вызываются как конструкторы. Поэтому были выработаны определенные условные обозначения имен, чтобы как-то отличать функции-конструкторы от обычных функций и методов. И вы, возможно, уже обратили на это внимание.

Функции и методы обычно именуются глаголами, описывающими их назначение, начиная с прописной буквы (например, `skulk()`, `creep()`, `sneak()`, `doSomethingWonderful()` и т.д.). А конструкторы обычно именуются существительными, описывающими конструируемый объект, начиная с прописной буквы (например, `Ninja()`, `Samurai()`, `Ronin()`, `KungFuPanda()` и т.д.)

Нетрудно заметить, что конструктор намного упрощает создание многих объектов, соответствующих одному шаблону, не повторяя многократно один и тот же код. В этом случае общий код пишется лишь один раз в теле самого конструктора. Более подробно о применении конструкторов и других механизмов объектно-ориентированного программирования, предоставляемых в JavaScript, речь пойдет в главе 6. Эти языковые средства значительно упрощают установку шаблонов объектов.

Но мы еще не завершили рассмотрение способов вызова функций. Ведь в JavaScript имеется еще один способ, предоставляющий немало возможностей для тщательного контроля над вызовом функций. И этот способ будет рассмотрен ниже.

Вызов с помощью методов `apply()` и `call()`

Как было показано ранее, главное отличие в способах вызова функций заключается в том, какой именно объект становится контекстом функции, на который ссылается параметр `this`, неявно передаваемый выполняющейся функции. Для методов это объект, который ими владеет, для функций верхнего уровня — всегда объект `window` (иными словами, метод объекта `window`), а для конструкторов — экземпляр вновь созданного объекта.

Но что, если требуется установить такой объект явным образом? Для того чтобы выяснить, зачем вообще такая возможность нужна, нам придется забежать немного вперед и принять во внимание следующее обстоятельство: когда вызывается обработчик событий, контекстом функции становится связанный с событием объект. Более подробно обработка событий будет рассматриваться в главе 13, а до тех пор достаточно сказать, что связанным является такой объект, для которого устанавливается обработчик событий.

Именно это обычно и требуется, хотя и не всегда. Например, при вызове функции в виде метода может возникнуть потребность принудительно установить в качестве ее контекста объект, владеющий этим методом, а не объект, связанный с событием. Подобная ситуация будет рассмотрена в главе 13, а до тех пор можно лишь сказать, что сделать такое вполне возможно.

Применение методов `apply()` и `call()`

В JavaScript предоставляются средства для вызова функции и явного указания любого объекта, который должен служить в качестве контекста функции. Это делается с помощью одного из двух методов, существующих для каждой функции: `apply()` и `call()`. И мы не оговорились: именно методов функций. Ведь функции — это объекты высшего порядка, создаваемые, между прочим, с помощью конструктора `Function()`. Следовательно, у них могут быть свойства и методы, как и у объектов любого другого типа.

Для вызова функции с помощью метода `apply()` последнему передаются два параметра: объект, применяемый в качестве контекста функции, а также массив значений, используемых в качестве аргументов вызываемой функции. Аналогичным образом используется и метод `call()`, за исключением того, что аргументы передаются функции в списке, а не в массиве. В листинге 3.5 демонстрируется, как эти методы действуют в коде.

Результаты выполнения приведенного выше кода представлены на рис. 3.9.

Листинг 3.5. Применение методов `apply()` и `call()` для предоставления контекста функции

```
<script type="text/javascript">

  function juggle() {
    var result = 0;
    for (var n = 0; n < arguments.length; n++) {
      result += arguments[n];
    }
    this.result = result;
  }

  var ninja1 = {};
  var ninja2 = {};

  juggle.apply(ninja1,[1,2,3,4]);

  juggle.call(ninja2,5,6,7,8);

  assert(ninja1.result === 10,"juggled via apply");
  assert(ninja2.result === 26,"juggled via call");

</script>
```

❶ Определить функцию
❷ Просуммировать аргументы
❸ Сохранить результат в контексте
❹ Подготовить тестируемый объект
❺ Применить функцию
❻ Вызвать функцию
❼ Проверить ожидаемые результаты

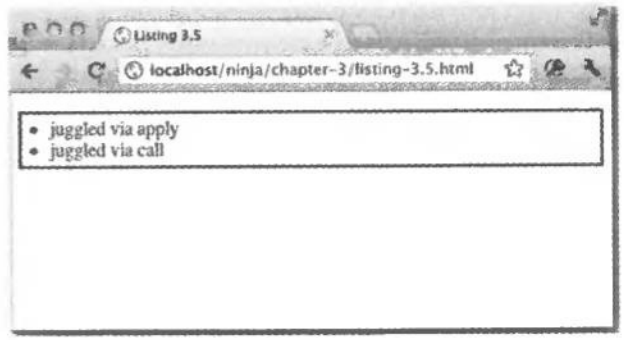

Рис. 3.9. Методы `apply()` *и* `call()` *позволяют установить в качестве контекста функции любой избранный объект*

В рассматриваемом здесь примере сначала устанавливается функция `juggle()` ❶, в которой жонглирование определяется как суммирование всех аргументов ❷ и последующее их сохранение в свойстве `result` объекта, служащего в качестве контекста данной функции ❸. Возможно, это и не совсем убедительное определение жонглирования, тем не менее оно позволит выяснить, были ли аргументы переданы функции правильно и какой именно объект стал ее контекстом.

Далее устанавливаются два объекта, которые станут контекстами функции ❹. Первый из них передается методу `apply()` данной функции вместе с массивом аргументов ❺, а второй — методу `call()` вместе с целым рядом аргументов ❻. И наконец, выполняется тестирование ❼!

Сначала проверяется, получил ли объект ninja1, переданный через метод apply(), свойство result как результат суммирования значений всех аргументов. Затем аналогичным образом проверяется объект ninja2, переданный через метод call(). Результаты, приведенные на рис. 3.9, показывают, что оба теста прошли успешно. Это означает, что нам удалось указать произвольно выбранные объекты, которые способны служить в качестве контекста для вызываемой функции.

Такой способ вызова функций может оказаться очень удобным в тех случаях, когда обычный контекст функции целесообразно приспособить под свои нужды, выбрав для него свой собственный объект. И особенно удобным это может оказаться при обращении к функциям обратного вызова.

Принудительная установка контекста функции при обратных вызовах

Рассмотрим конкретный пример принудительной установки специально избранного объекта в качестве контекста функции. С этой целью создадим простую функцию, которая будет выполнять определенную операцию над каждым элементом массива. В императивном программировании массив зачастую передается методу, а для обращения к элементам массива организуется цикл for, в котором над каждым элементом выполняется требующаяся операция, как показано ниже.

```
function(collection) {
  for (var n = 0; n < collection.length; n++) {
    /* сделать что-нибудь с элементом массива collection[n] */
  }
}
```

С другой стороны, в функциональном программировании обычно создается функция для выполнения операции над одним элементом массива. И этой функции в качестве аргумента передается по очереди каждый элемент массива:

```
function(item){
  /* сделать что-нибудь с элементом */
}
```

Отличие заключается в уровне мышления, на котором функции мыслятся как стандартные блоки программы, а не императивные операторы. На первый взгляд это кажется слишком далеким от практики, поскольку мы лишь переносим цикл for на другой уровень. Но не следует спешить с выводами, поскольку мы еще не рассмотрели данный пример до конца.

Для упрощения стиля функционального программирования во многих распространенных библиотеках JavaScript предоставляется функция, реализующая цикл for-each и вызывающая функцию обратного вызова для каждого элемента массива. Зачастую это более краткий и предпочтительный стиль по сравнению с использованием традиционного оператора цикла for для тех, кто знаком с функциональным программированием. Его организационные преимущества станут еще более очевидными (с точки зрения повторного использования кода), как только мы рассмотрим замыкания в главе 5. Такая итеративная функция могла бы передавать текущий элемент массива в качестве параметра функции обратного вызова, но чаще всего текущий элемент массива делается контекстом функции обратного вызова.

Примечание

В версии JavaScript 1.6 для экземпляров объектов типа Array был определен метод forEach(). И теперь его можно встретить во многих современных браузерах.

Рассмотрим пример построения собственного (упрощенного) варианта подобной функции. Ее исходный код приведен в листинге 3.6.

Листинг 3.6. Построение функции, реализующей цикл for-each, с целью продемонстрировать, каким образом устанавливается контекст функции

```
<script type="text/javascript">

  function forEach(list,callback) {
    for (var n = 0; n < list.length; n++) {
      callback.call(list[n],n);
    }
  }

  var weapons = ['shuriken','katana','nunchucks'];

  forEach(
    weapons,
    function(index){
      assert(this == weapons [index],
             "Got the expected value of " + weapons [index]);
    }
  );

</script>
```

❶ Определить функцию типа for-each

❷ Вызвать функцию обратного вызова

❸ Подготовить тестируемый объект

❹ Проверить функцию

Рассматриваемая здесь итеративная функция отличается простой сигнатурой, принимая в качестве первого аргумента массив объектов для циклического обращения к ним, а в качестве второго аргумента – функцию обратного вызова ❶. В этой функции организуется циклическое обращение к элементам массива, в ходе которого для каждого элемента массива вызывается функция обратного вызова ❷. С этой целью используется метод call() функции обратного вызова, которому передается текущий элемент массива в качестве первого аргумента, а параметр цикла (он же индекс итерации) – в качестве второго аргумента. Это *должно* привести к тому, что текущий элемент массива становится контекстом функции, а индекс итерации передается функции обратного вызова в качестве единственного аргумента.

Далее выполняется тест. С этой целью сначала подготавливается простой массив ❸, а затем вызывается функция forEach(), которой передается тестируемый массив и функция обратного вызова, где, собственно, и проверяется, устанавливается ли предполагаемый элемент массива в качестве контекста функции всякий раз, когда вызывается функция обратного вызова ❹. Как показано на рис. 3.10, рассматриваемая здесь функция действует безупречно.

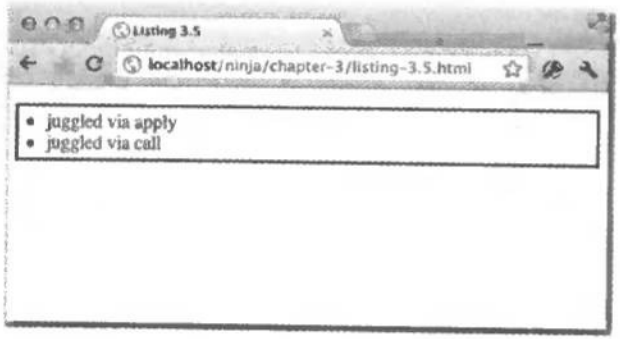

Рис. 3.10. Как показывают результаты тестирования, любой избранный объект можно сделать контекстом функции обратного вызова

Для окончательной реализации подобной функции предстоит еще немало сделать. В частности, решить следующие вопросы: что, если первый аргумент не является массивом, а второй – функцией, и как предоставить автору веб-страницы возможность прервать цикл в любой момент? В качестве упражнения вы можете усовершенствовать данную функцию, разрешив эти вопросы. Кроме того, вы можете дополнить ее возможностью передавать произвольное число аргументов функции обратного вызова, помимо индекса итерации.

Но если методы `apply()` и `call()` делают практически одно и то же, то как же выбрать среди них наиболее подходящий для применения? Общий ответ на этот вопрос такой же, как и на многие подобные вопросы: применять следует тот метод, который повышает ясность кода. А более практический ответ состоит в следующем: применять следует тот метод, который делает более удобным обращение с аргументами. Так, если имеется ряд не связанных друг с другом значений, заданных в переменных или литералах, метод `call()` позволяет указать их непосредственно в списке своих аргументов. Но если значения аргументов уже находятся в массиве или же если их удобнее собрать в массив, то лучше выбрать метод `apply()`.

Резюме

В этой главе были рассмотрены самые разные, примечательные особенности работы функций в JavaScript. Несмотря на повсеместное применение функций, правильное понимание внутреннего механизма их действия имеет решающее значение для написания доброкачественного кода на JavaScript.

В частности, из этой главы вы узнали следующее.

- Написание сложного кода опирается на изучение JavaScript как языка функционального программирования.
- Функции являются объектами высшего порядка и интерпретируются таким же образом, как и любые другие объекты в JavaScript. Подобно объектам любого другого типа, их можно:
  - создавать с помощью литералов;
  - присваивать переменным или свойствам;

  - передавать в качестве параметров;
  - возвращать в качестве результатов выполнения других функций;
  - наделять свойствами и методами.
- У каждого объекта имеются свои особенности, которыми он отличается от остальных объектов. Для функций это способность быть вызванными.
- Функции создаются с помощью литералов с необязательным указанием имени.
- Функции можно вызывать из браузера в течение срока действия веб-страницы, и, в частности, для обработки разных видов событий.
- Область действия элементов, объявляемых в функциях, оказывается в JavaScript иной, чем в большинстве других языков программирования. В частности:
  - Область действия переменных в функции простирается от места их объявления и до конца функции, пересекая границы кодовых блоков.
  - Внутренние именованные функции доступны повсюду в охватывающей внешней функции, даже по ссылкам с упреждением (в этом случае действует подъемный механизм).
- Список параметров функции может отличаться по длине от фактического списка ее аргументов. В частности:
  - Параметры, которым не присвоены значения, вычисляются как имеющие значение `undefined`.
  - Лишние аргументы просто не привязываются к именам параметров.
- При каждом вызове функции ей передаются два параметра:
  - `arguments` – коллекция фактически переданных аргументов;
  - `this` – ссылка на объект, служащий в качестве контекста функции.
- Функции можно вызывать разными способами, а механизм их вызова определяет значение контекста функции. В частности, при вызове функции:
  - в виде обычной функции ее контекстом становится глобальный объект (`window`);
  - в виде метода контекстом функции становится объект, владеющий этим методом;
  - в виде конструктора контекстом функции становится вновь созданный объект, размещаемый в оперативной памяти;
  - с помощью методов `apply()` и `call()` этой функции ее контекстом может стать любой избранный объект.

Итак, в этой главе были рассмотрены основные детали механизма, приводящего функции в действие. А в следующей главе будет показано, как применять на практике знания, полученные о функциях.

Обращение с функциями

4

В этой главе...

- Особое значение анонимных функций
- Способы обращения к функциям для вызова, включая рекурсию
- Сохранение ссылок на функции
- Применение контекста функции для достижения поставленной цели
- Обращение со списками аргументов переменной длины
- Выяснение, является ли объект функцией

В предыдущей главе основное внимание было сосредоточено на том, как функции интерпретируются в JavaScript в качестве объектов высшего порядка и каким образом это способствует выработке функционального стиля программирования. А в этой главе будет показано, как пользоваться функциями для разрешения различных затруднений, которые могут возникнуть в процессе авторской разработки веб-приложений.

Примеры, приведенные в этой главе, специально выбраны для раскрытия секретов, которые помогут вам по-настоящему понять сущность функций в JavaScript. Многие из этих примеров довольно просты, но они наглядно демонстрируют важные понятия, широко применяемые при разрешении затруднений, которые могут возникнуть в реальных проектах разработки программного обеспечения. Итак, попробуем, без лишних слов, применить на практике знания, полученные ранее о функциях JavaScript, обращаясь с ними как с могущественным оружием, которым они на самом деле являются.

Анонимные функции

Возможно, вам не приходилось иметь дело с анонимными функциями до того, как они были представлены в предыдущей главе, но они относятся к тем ключевым понятиям,

которые следует непременно знать, чтобы мастерски программировать на JavaScript. Они являются очень важным и логическим языковым средством, созданным под большим влиянием таких языков функционального программирования, как Scheme.

Как правило, анонимные функции применяются в тех случаях, когда требуется создать функцию, чтобы воспользоваться ею впоследствии, например, сохранить в переменной, установить в качестве метода объекта или использовать ее как функцию обратного вызова (в частности, для обработки события или времени ожидания). Во всех подобных случаях функция совсем не обязательно должна иметь имя для последующей ссылки. В остальной части этой книги будет представлено немало примеров применения анонимных функций, поэтому не особенно тревожьтесь, если данное понятие пока еще кажется вам чуждым и не совсем понятным.

Если вы перешли на JavaScript из строго типизированных языков объектно-ориентированного программирования, то у вас может сложиться представление, что функции и методы жестко определяются перед употреблением, всегда доступны и обязательно именуются для последующей ссылки, т.е. являются чем-то конкретным и устойчивым. Но в языках функционального программирования, в том числе и в JavaScript, функции являются чем-то более эфемерным, поскольку они зачастую определяются по мере надобности и также часто отвергаются.

В листинге 4.1 приведены некоторые примеры объявления анонимных функций.

Листинг 4.1. Типичные примеры применения анонимных функций

```
script type="text/javascript">

  window.onload =
    function(){ assert(true, 'power!'); };

  var ninja = {
    shout: function(){
      assert(true,"Ninja");
    }
  };

  ninja.shout();

  setTimeout(
  function(){ assert(true,'Forever!'); },
  500);

</script>
```

❶ Установить анонимную функцию в качестве обработчика событий. Нет нужды создавать именованную функцию только для ссылки на нее в данном месте

❷ Создать функцию для применения в качестве метода объекта ninja. Для вызова функции послужит свойство shout, поэтому имя ей не потребуется

❸ Передать функцию обратного вызова методу setTimeout(), чтобы вызывать ее по истечении времени ожидания. И в этом случае именовать функцию не нужно

В примере кода из листинга 4.1 выполняются следующие действия. Сначала устанавливается функция, предназначенная в качестве обработчика события, связанного с загрузкой ❶. Эта функция будет вызываться не напрямую, а из механизма обработки событий. То же самое можно было бы сделать следующим образом:

```
function bootMeUp(){ assert(true, 'power!'); };
window.onload = bootMeUp;
```

Но зачем создавать функцию на отдельном уровне, да еще и с именем, которое вообще не понадобится? Далее объявляется анонимная функция в качестве свойства

объекта ❷. Как пояснялось в предыдущей главе, функция в этом случае становится методом объекта, который можно затем вызывать по ссылке на свойство данного объекта.

Еще одно интересное применение анонимной функции, очень похожее на упоминавшееся в предыдущей главе, состоит в организации с ее помощью обратного вызова, передаваемого другой функции при ее вызове. В данном примере анонимная функция передается в качестве аргумента методу `setTimeout()` объекта `window` и вызывается по истечении времени ожидания в течение половины секунды ❸.

На рис. 4.1 приведены результаты выполнения кода из рассматриваемого здесь примера по истечении одной или двух секунд. Обратите внимание на то, что во всех упомянутых выше случаях имя функции совсем не требуется для ее применения после объявления. Обратите также внимание на применение в очередной раз функции `assert()` с проверяемым условием `true` в качестве средства вывода для ленивых. В конце концов, мы написали код, так почему бы им не воспользоваться?

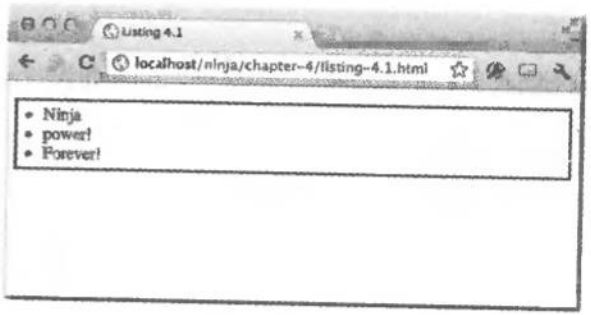

Рис. 4.1. Анонимные функции можно вызывать в разные моменты времени, несмотря на то, что у них нет имени

Примечание

Некоторые могут подумать, что, присвоив анонимную функцию свойству `shout`, мы тем самым дали этой функции имя, но это неверная мысль. Именем `shout` называется *свойство*, а не сама функция. Для того чтобы убедиться в этом, достаточно проверить свойство `name` анонимной функции. Если еще раз просмотреть результаты выполнения кода из листинга 3.1, приведенные на рис. 3.4, то можно заметить, что анонимные функции не обладают именами, в отличие от именованных функций. Их свойство `name` содержит пустую символьную строку.

Мы еще не раз вернемся к рассмотрению анонимных функций в остальной части этой книги, поскольку мастерское владение JavaScript опирается на его использование в качестве языка функционального программирования. В связи с этим мы будем часто прибегать к стилю функционального программирования во всех последующих примерах кода. Функциональное программирование сосредоточено на небольших функциях, обычно лишенных побочных эффектов и служащих в качестве основных стандартных блоков прикладного кода. По ходу дела мы покажем, что такой стиль программирования имеет существенное значение для успешной разработки веб-приложений.

Помимо стремления не засорять глобальное пространство имен ненужными именами функций, следует стараться создавать много мелких функций, которыми можно вполне обойтись вместо крупных функций, изобилующих императивными операто-

рами. Функциональное программирование на основе анонимных функций позволяет решать немало трудных задач, возникающих при разработке веб-приложений на JavaScript. Поэтому в остальной части этой главы мы будем часто обращаться к анонимным функциям, рассматривая различные способы их употребления. И начнем мы с рекурсии.

Рекурсия

Понятие *рекурсии* вам должно быть знакомо из прежнего опыта программирования. Рекурсия происходит всякий раз, когда функция вызывает самое себя или другую функцию, которая, в свою очередь, вызывает исходную функцию из любого места в дереве вызовов.

Рекурсия — очень полезный прием для разработки любых приложений. На первый взгляд, рекурсия оказывается наиболее полезной в тех приложениях, где выполняется немало математических операций. В самом деле, многие математические формулы рекурсивны по своему характеру. Но рекурсия оказывается полезной и для реализации таких конструкций, как деревья перебора. Именно такие конструкции можно чаще всего встретить в веб-приложениях. Кроме того, рекурсию можно применять для выработки более глубокого понимания принципа действия функций в JavaScript. Рассмотрим сначала рекурсию в ее простейшей форме.

Рекурсия в именованных функциях

Существует немало распространенных примеров рекурсивных функций. К их числу относится проверка на палиндром, которая среди рекурсивных способов занимает такое же место, как и вывод предложения "Здравствуй, мир!" в качестве первого упражнения при изучении любого языка программирования. Нерекурсивное определение палиндрома гласит, что это выражение, которое одинаково читается в любом направлении. Мы можем воспользоваться им для реализации функции, создающей обратную копию символьной строки и сравнивающую ее с исходной строкой. Но копирование символьной строки — далеко не самое изящное решение по разным причинам, и не в последнюю очередь потому, что в этом случае приходится выделять оперативную память под вновь создаваемую символьную строку.

Используя более математическое по своему характеру определение палиндрома, можно прийти к более изящному решению. Это определение приведено ниже.

1. Строка, состоящая из одного символа или вообще без символов, является палиндромом.
2. Любая другая символьная строка является палиндромом, если первый и последний ее символы одинаковы, а остальная часть строки, кроме этих символов, оказывается палиндромом.

Реализация такого определения в коде выглядит следующим образом:

```
function isPalindrome(text) {
  if (text.length <= 1) return true;
  if (text.charAt(0) != text.charAt(text.length - 1)) return false;
  return isPalindrome(text.substr(1,text.length - 2));
}
```


Следует заметить, что новое определение палиндрома и его реализация в коде имеет характер *рекурсии*, поскольку оно используется для того, чтобы выяснить, содержит ли символьная строка палиндром. А реализуется рекурсивная функция довольно просто: рекурсивный вызов делается по имени функции в последней строке ее кода.

Информация к размышлению

В рассматриваемой здесь рекурсивной функции значение `null` или `undefined` параметра `text` не обрабатывается. Как организовать их обработку и что следует возвращать в подобных случаях? Являются ли несуществующие символьные строки палиндромными?

Более любопытная и менее ясная ситуация возникает, когда приходится иметь дело с анонимными функциями, но мы дойдем и до этого. А до тех пор рассмотрим очень простой пример рекурсии, на основании которого будут построены другие примеры по ходу обсуждения темы рекурсии.

Ниндзя пользовались системой сигналов, чтобы оповещать друг друга, и зачастую для прикрытия они употребляли естественные звуки. Попробуем наделить наших ниндзя способностью стрекотать, как сверчок, используя определенное количество стрекотаний для кодирования сообщений. Реализацию такого алгоритма начнем с применения рекурсии по имени функции, как показано в листинге 4.2.

Листинг 4.2. Реализация рекурсивного алгоритма стрекотания в именованной функции

```
<script type="text/javascript">

  function chirp(n) {
    return n > 1 ? chirp(n - 1) + "-chirp" : "chirp";
  }

  assert(chirp(3) == "chirp-chirp-chirp",
         "Calling the named function comes naturally.");

</script>
```

❶ Объявить рекурсивную функцию стрекотания, вызывающую себя по имени до тех пор, пока она не определит свое завершение

❷ Подтвердить, что ниндзя может стрекотать, как задумано

В приведенном выше коде сначала объявляется функция `chirp()`, реализующая рекурсию, вызывая самое себя по имени ❶, как это было сделано в примере с палиндромом. А в следующем затем тесте проверяется, действует ли данная функция так, как и предполагалось ❷.

О рекурсии

Функция, приведенная в листинге 4.2, удовлетворяет следующим двум критериям рекурсии: ссылке на самое себя и сходимости к моменту завершения. Эта функция явным образом вызывает самое себя, а следовательно, соблюдается первый критерий. Значение параметра `n` уменьшается на каждом шаге итерации, и поэтому рано или поздно достигается единичное или меньшее значение, на котором рекурсия прекращается, а следовательно, соблюдается и второй критерий. Следует, однако, заметить, что якобы рекурсивная функция, которая не сходится к моменту своего завершения, более известна как бесконечный цикл!

Итак, мы выяснили, каким образом рекурсия действует в именованной функции. Но что, если ее требуется организовать в анонимных функциях? Об этом и пойдет речь в следующем разделе.

Рекурсия в методах

В предыдущем разделе предполагалось наделить наших ниндзя способностью стрекотать, как сверчок, но этого так и не было сделано. Нам удалось лишь создать автономную функцию для реализации алгоритма стрекотания. Попробуем исправить этот недочет, объявив рекурсивную функцию в виде метода объекта `ninja`. Но сделать это будет труднее, поскольку рекурсивная функция становится анонимной и присваиваемой свойству объекта, как следует из листинга 4.3.

Листинг 4.3. Рекурсия в методе объекта

```
<script type="text/javascript">

  var ninja = {
    chirp: function(n) {
      return n > 1 ? ninja.chirp(n - 1) + "-chirp" : "chirp";
    }
  };

assert(ninja.chirp(3) == "chirp-chirp-chirp",
      "An object property isn't too confusing, either.");

</script>
```

❶ Объявить рекурсивную функцию стрекотания как свойство объекта `ninja`. Теперь метод нужно вызвать из самого себя, используя ссылку на метод данного объекта

В приведенном выше тестовом коде рекурсивная функция определена как анонимная, а ссылка на нее делается посредством свойства `chirp` объекта `ninja` ❶. В своем теле эта функция вызывается рекурсивно по следующей ссылке на свойство объекта: `ninja.chirp()`. На эту функцию нельзя ссылаться непосредственно по ее имени, как это делалось в листинге 4.2, поскольку она анонимная и не имеет имени. Возникающая при этом взаимосвязь приведена на рис. 4.2.

Рис. 4.2. Рекурсивная функция теперь становится методом, ссылающимся на самого себя через свойство `chirp` объекта `ninja`

На первый взгляд все выглядит прекрасно, но поскольку в рассматриваемом здесь примере мы полагаемся на *косвенную* ссылку на функцию (через свойство `chirp` объекта `ninja`), то, образно выражаясь, мы рискуем ходить по тонкому льду. А для мастера своего дела это неблагоразумно. Проанализируем причины, по которым мы можем оказаться в полынье.

Проблема пропадающих ссылок

Пример кода, приведенный в листинге 4.3, был построен на том, что у нас имелась ссылка на функцию, вызываемую рекурсивно в свойстве объекта. Но в отличие от конкретного имени функции, такие ссылки могут оказаться временными, и поэтому, полагаясь на них, мы рискуем провалиться под лед.

Внесем изменения в предыдущий пример, введя новый объект, скажем, `samurai`, который также ссылается на анонимную рекурсивную функцию в объекте `ninja`. Соответствующий код приведен в листинге 4.4.

Листинг 4.4. Рекурсия по отсутствующей ссылке на функцию

```
<script type="text/javascript">

  var ninja = {
    chirp: function(n) {
      return n > 1 ? ninja.chirp(n - 1) + "-chirp" : "chirp";
    }
  };

  var samurai = { chirp: ninja.chirp };

  ninja = {};
  try {
    assert(samurai.chirp(3) == "chirp-chirp-chirp",
          "Is this going to work?");
  }
  catch(e){
    assert(false,
          "Uh, this isn't good! Where'd ninja.chirp go?");
  }

</script>
```

❶ Создать метод `chirp()` для объекта `samurai`, ссылаясь на имеющийся одноименный метод объекта `ninja`. Зачем повторять уже написанный однажды код?

❷ Переопределить объект `ninja` так, чтобы у него не было свойств. Это означает, что свойство `chirp` исчезнет!

❸ Проверить, все ли работает по-прежнему. Подсказка: не работает!

Теперь можно лучше видеть, насколько быстро в коде возникает шаткая ситуация. В частности, ссылка на функцию стрекотания копируется в объект `samurai` ❶, в результате чего оба свойства, `ninja.chirp` и `samurai.chirp`, ссылаются на одну и ту же анонимную функцию. Блок-схема образующихся при этом взаимосвязей приведена на рис. 4.3. В части А, похожей на рис. 4.2, показаны конструкции после создания объекта `ninja`, а в части В — они же после создания объекта `samurai`.

В данный момент никаких затруднений пока еще не возникает, поскольку ссылки на функции нередко делаются из многих мест. Но опасность попасть в западню существует из-за того, что функция является рекурсивной и использует ссылку `ninja.chirp` для вызова самой себя независимо от того, вызывается ли она как метод объекта `ninja` или же объекта `samurai`.

Так что же произойдет, если удалить объект `ninja` и оставить объект `samurai`, свалив на него всю ответственность за обращение к анонимной функции? Для проверки этой ситуации объект `ninja` переопределяется далее в рассматриваемом здесь коде как пустой ❷ (см. часть С на рис. 4.3). Анонимная функция по-прежнему существует, и к ней можно обращаться через свойство `samurai.chirp`, тогда как свойство `ninja.chirp`

уже отсутствует. А поскольку эта функция рекурсивно вызывает сама себя через отсутствующую теперь ссылку, то при ее вызове возникает ошибочная и весьма неприятная ситуация ❸.

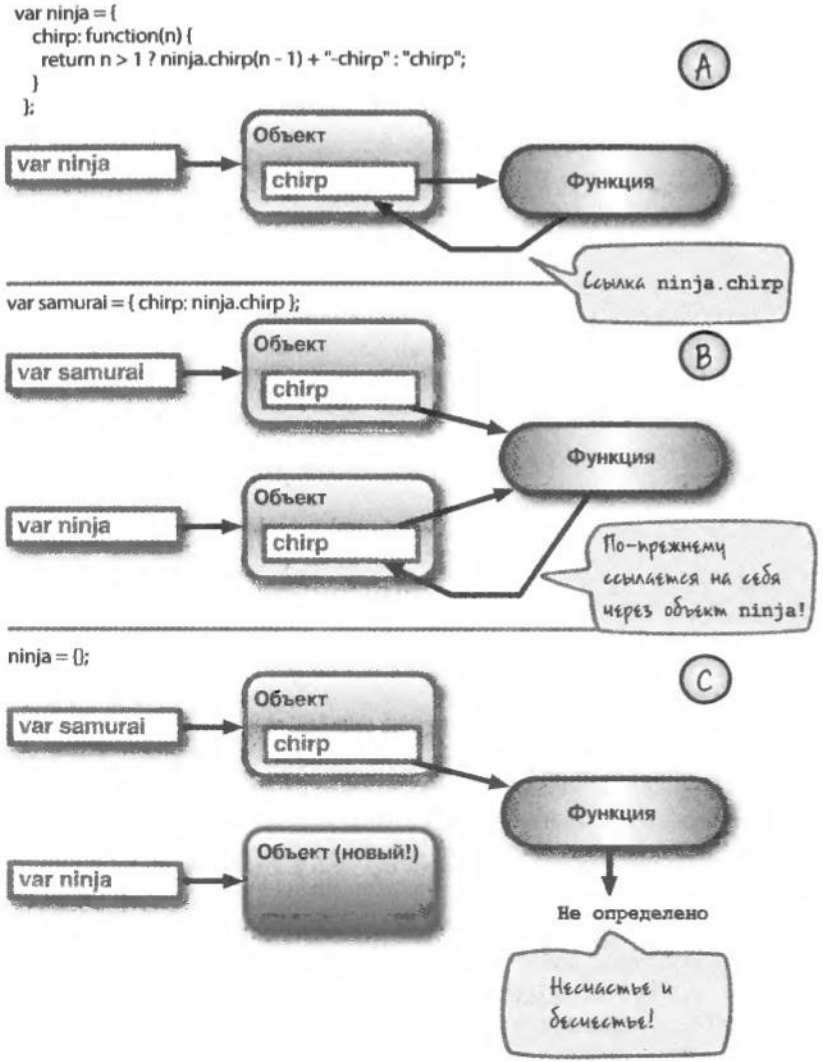

Рис. 4.3. Два объекта содержат ссылку на одну и ту же функцию, но функция ссылается на самое себя только через один из объектов. Шаткая, как тонкий лед, ситуация!

Это затруднение можно разрешить, исправив небрежно сделанное первоначальное определение рекурсивной функции. Вместо того чтобы явно ссылаться на объект `ninja` в анонимной функции, следовало бы воспользоваться контекстом функции (через параметр `this`) следующим образом:

```
var ninja = {
  chirp: function(n) {
    return n > 1 ? this.chirp(n - 1) + "-chirp" : "chirp";
  }
};
```


Напомним, что когда функция вызывается в виде метода, контекстом функции становится объект, через который этот метод вызывался. Так, если функция вызывается как метод `ninja.chirp()`, ее параметр `this` содержит ссылку на объект `ninja`, но когда она вызывается как метод `samurai.chirp()`, ее параметр `this` содержит ссылку на объект `samurai`, и все хорошо. Благодаря использованию контекста функции (через параметр `this`) метод `chirp()` становится намного более надежным. Именно так его и следовало объявлять с самого начала. Казалось бы, затруднение разрешено, но не все так просто.

Встраиваемые именованные функции

Решение, к которому мы пришли в предыдущем разделе, идеально подходит в том случае, когда функции используются как методы объектов. В действительности рассмотренный выше прием, состоящий в использовании контекста функции для ссылки на "владеющий" методом объект, независимо от того, является ли этот метод рекурсивным или нет, считается весьма распространенным и широко принятым. И более подробно он рассматривается в главе 6.

Но теперь возникает другое затруднение. Упомянутое выше решение опирается на тот факт, что функция превращается в метод `chirp()` любого объекта, в котором он определен. Но что, если свойства объекта имеют разные имена или одна из ссылок на функцию даже не является свойством объекта? Данное решение подходит для конкретного случая, когда функция используется как метод, а имя свойства для данного метода остается одинаковым в любом случае. В связи с этим возникает вопрос: можно ли выработать более общее решение?

Рассмотрим другой подход: что, если присвоить *имя* анонимной функции? На первый взгляд такое решение может показаться безрассудным. Ведь если мы собираемся использовать функцию как метод, то зачем еще и присваивать ей собственное имя? Напомним, что при объявлении функционального литерала имя функции указывать не обязательно, и оно опускалось для всех функций, кроме тех, что относятся к самому верхнему уровню. Но оказывается, что в присваивании имени *любому* функциональному литералу нет ничего дурного, даже если они объявляются как функции обратного вызова или методы.

Функции, которые перестают быть анонимными, лучше было бы назвать *встраиваемыми*, чем анонимными именованными, чтобы избежать оксюморона (сочетания противоположных по значению слов). Рассмотрим применение таких функций на примере кода, приведенного а листинге 4.5.

Листинг 4.5. Применение встраиваемой функции в рекурсивном стиле

```
<script type="text/javascript">

var ninja = {
  chirp: function signal(n) {
    return n > 1 ? signal(n - 1) + "-chirp" : "chirp";
  }
};
```

❶ Объявить именованную встраиваемую функцию

```
assert(ninja.chirp(3) == "chirp-chirp-chirp",
      "Works as we would expect it to!");

  var samurai = { chirp: ninja.chirp };

  ninja = {};

assert(samurai.chirp(3) == "chirp-chirp-chirp",
       "The method correctly calls itself.");

</script>
```

❷ Проверить, работает ли функция должным образом

❸ Создать новый объект

❹ Удалить объект ninja, как и в предыдущем примере

❺ Проверить, работает ли функция по-прежнему. И она таки работает!

Сначала в приведенном выше коде встраиваемой функции присваивается имя `signal` ❶, которое используется для рекурсивной ссылки в теле самой функции, а затем проверяется, действует ли по-прежнему ее вызов в виде метода объекта `ninja` ❷. Как и прежде, ссылка на функцию копируется в свойство `samurai.chirp` ❸, а исходный объект `ninja` удаляется ❹.

После проверки вызова функции в виде метода объекта `samurai` ❺ обнаруживается, что все по-прежнему работает нормально, поскольку удаление свойства `chirp` объекта `ninja` не оказало никакого влияния на имя, присвоенное встраиваемой функции и используемое для рекурсивного вызова. Подобное именование встраиваемой функции можно расширить еще больше, распространяя его на обычное присваивание функций переменным. Хотя это может привести к неожиданным результатам, как показано в листинге 4.6.

Листинг 4.6. Проверка идентичности встраиваемой функции

```
<script type="text/javascript">

  var ninja = function myNinja(){
    assert(ninja == myNinja,
           "This function is named two things at once!");
  };

  ninja();

assert(typeof myNinja == "undefined",
      "But myNinja isn't defined outside of the function.");

</script>
```

❶ Объявить именованную встраиваемую функцию и присвоить ее переменной

❷ Проверить, равнозначны ли оба имени во встраиваемой функции

❸ Вызвать функцию для выполнения внутреннего теста

❹ Проверить, недоступно ли имя встраиваемой функции за ее пределами

Приведенный выше пример выявляет самую важную особенность встраиваемых функций: несмотря на то, что встраиваемые функции можно именовать, их имена доступны *только* в самих функциях. Об этом свидетельствуют правила определения областей действия, рассматривавшиеся в главе 3. Имена встраиваемых функций действуют в какой-то степени подобно именам переменных, а область их действия ограничивается той функцией, в которой они объявлены.

Сначала в приведенном выше примере кода была объявлена встраиваемая функция с именем `myNinja` ❶, а затем выполнен внутренний тест, чтобы проверить, означают ли одно и то же имя функции и ссылка, по которой эта функция присваивается переменной ❷. Этот тест выполняется при вызове встраиваемой функции ❸. Далее было проверено, недоступно ли имя данной функции за ее пределами ❹. Как и ожидалось, при выполнении кода этот тест проходит.

Примечание

Именно поэтому функции самого верхнего уровня создаются как методы объекта `window`. Без свойств объекта `window` ссылаться на функции было бы просто невозможно.

Таким образом, на именование встраиваемых функций накладываются определенные ограничения, хотя оно и представляет средства для четкого действия рекурсивных ссылок в подобных функциях. Безусловно, такой подход вносит в код большую ясность, чем применение контекста функции через параметр `this`. А теперь попробуем выяснить, имеются ли другие способы, которыми можно было бы воспользоваться на практике.

Свойство `callee`

Рассмотрим еще один способ организации рекурсии, представив попутно очередное понятие, имеющее отношение к функциям. Это свойство `callee` параметра `arguments`. С этой целью обратимся к примеру кода, приведенного в листинге 4.7.

Предупреждение

Свойство `callee` относится к числу новых языковых средств, которые предполагается внедрить в грядущей версии JavaScript, тогда как стандарт ECMAScript 5 запрещает применение этого свойства в "строгом" режиме. Свойство `callee` можно применять в современных браузерах, хотя его будущее неопределенно. Поэтому применять его в новом коде нецелесообразно. Тем не менее свойство `callee` рассматривается здесь, поскольку оно может встретиться в уже имеющемся коде.

Листинг 4.7. Применение свойства `arguments.callee` для ссылки на вызывающую функцию

```
<script type="text/javascript">

  var ninja = {
    chirp: function(n) {
      return n > 1 ? arguments.callee(n - 1) + "-chirp" : "chirp";
    }
  };

assert(ninja.chirp(3) == "chirp-chirp-chirp",
      "arguments.callee is the function itself.");

</script>
```

Сослаться на функцию через свойство arguments.callee

Проверить, можно ли стрекотать сколько угодно!

Как пояснялось в разделе главы 3, посвященном вызову функций, параметр `arguments` неявно передается каждой функции. У этого параметра имеется также свойство `callee`, ссылающееся на функцию, которая выполняется в настоящий момент.

Это свойство может служить надежным способом, постоянно обеспечивающим доступ к самой функции. Далее в этой главе, а также в последующей главе 5, посвященной замыканиям, мы обсудим более подробно, чего можно добиться с помощью этого конкретного свойства.

Рассмотренные выше различные способы обращения к функциям совместно дают немало преимуществ, когда дело доходит до написания сложного кода. Ведь они предоставляют различные средства для обращения к функциям, не прибегая к жестко кодируемым и хрупким зависимостям, в том числе от имен переменных и свойств. Следующей вехой на пути к изучению функций является представление о том, каким образом объектно-ориентированный характер функций в JavaScript может оказать помощь в повышении качества кода.

Особенности применения функций как объектов

Как уже не раз подчеркивалось в этой главе, функции в JavaScript не похожи на функции во многих других языках программирования. В JavaScript функции наделяются многими возможностями, среди которых далеко не последнее место принадлежит интерпретации функций как объектов высшего порядка. Как было показано ранее, функции можно наделять свойствами и методами, присваивать переменным и свойствам и вообще пользоваться ими в полной мере как обычными объектами. Но у них имеется одна замечательная и очень сильная сторона: их можно вызывать.

В этом разделе мы рассмотрим ряд способов, позволяющих выгодно воспользоваться сходством функций с объектами других типов. Но сначала напомним ряд ключевых положений, которые будут использоваться в дальнейшем. И начнем мы с присваивания функций переменным, как показано в приведенном ниже примере кода.

```
var obj = {};
var fn = function(){};
assert(obj && fn, "Both the object and function exist.");
```

Функцию можно присвоить переменной точно так же, как и любой другой объект. Это же относится и к присваиванию функций свойствам объектов для создания методов.

Примечание

Не забывайте ставить точку с запятой после определений `function(){}` функций. Точки с запятой принято ставить в конце всех операторов, и особенно после операторов присваивания переменным. Это же относится к анонимным функциям. Ведь правильно расставленные точки с запятой дают больше возможностей для эффективного применения способов сжатия кода.

Еще одна не совсем обычная возможность состоит в том, что к функциям можно присоединять свойства, как и к любым другим объектам. Ниже приведены характерные тому примеры.

```
var obj = {};
var fn = function(){};
obj.prop = "hitsuke (distraction)";
fn.prop = "tanuki (climbing)";
```


Этой особенностью функций можно по-разному воспользоваться в библиотечном или обычном коде страничных сценариев. И это особенно важно, когда дело касается таких предметов, как управление функциями обратного вызова для обработки событий. Рассмотрим две более любопытные возможности, которые открывает данная особенность функций: сначала сохранение функций в коллекциях, а затем методику под названием "запоминание".

Сохранение функций

Иногда возникает потребность сохранить в коллекции связанные вместе, но однозначные функции. Характерным тому примером служит управление функциями обратного вызова для обработки событий, более подробно рассматриваемое в главе 13. Но при вводе функций в коллекцию может возникнуть затруднение, связанное с тем, что нужно отделить те функции, которые являются новыми для коллекции и поэтому должны быть в нее введены, от тех функций, которые уже находятся в коллекции и поэтому не должны в нее вводиться.

Очевидно, хотя и наивно предположить, что все функции можно сначала сохранить в массиве, а затем организовать циклическое обращение к элементам этого массива, чтобы проверить, не дублируются ли функции. К сожалению, это малоэффективный способ, который не годится для мастера, стремящегося к тому, чтобы программа не просто работала, а работала надежно и эффективно. Поэтому для достижения поставленной цели на подходящем уровне сложности можно воспользоваться свойствами функций, как показано в листинге 4.8.

Листинг 4.8. Сохранение однозначных функций в коллекции

```
<script type="text/javascript">

  var store = {
    nextId: 1,

    cache: {},

    add: function(fn) {
      if (!fn.id) {
        fn.id = store.nextId++;
        return !!(store.cache[fn.id] = fn);
      }
    }
  };

  function ninja(){}

  assert(store.add(ninja),
         "Function was safely added.");
  assert(!store.add(ninja),
         "But it was only added once.");

</script>
```

1. Отследить следующий доступный для присваивания идентификатор
2. Создать объект в качестве кеша для хранения функций
3. Добавить в кеш только однозначные функции
4. Проверить, все ли работает должным образом

В приведенном выше примере кода сначала создается объект, который присваивается переменной `store` для дальнейшего хранения однозначного набора функций. У этого объекта имеются два свойства данных: одно — для хранения следующего доступного значения идентификатора `id` ❶, а другое — для кеширования сохраняемых функций ❷. Функции добавляются в кеш с помощью метода `add()` ❸.

В методе `add()` сначала проверяется, введено ли свойство `id` в функцию, и если оно введено, то предполагается, что функция уже обработана, и поэтому она просто игнорируется. В противном случае свойство `id` присваивается функции и попутно инкрементируется свойство `nextId`, а сама функция вводится как свойство в объект `cache`, используя значение `id` в качестве имени свойства. Затем возвращается логическое значение `true`, которое вычисляется сложным путем преобразования функции в ее логический эквивалент. Благодаря этому после вызова функции `add()` уведомляется, что функция введена в коллекцию.

Совет

Конструкция `!!` позволяет очень просто преобразовать любое выражение на языке JavaScript в его логический эквивалент. Например: `!!"he shot me down" === true` и `!!0 === false`. В листинге 4.8 функция в конечном итоге преобразуется в свой логический эквивалент, который всегда равен `true`. (Безусловно, логическое значение `true` можно было бы закодировать жестко, но тогда у нас не было бы удобной возможности представить языковую конструкцию `!!`.)

Если выполнить рассматриваемый здесь код на веб-странице в браузере, то при попытке в ходе тестирования ввести функцию `ninja()` в коллекцию дважды ❹ она вводится лишь один раз, как показано на рис. 4.4.

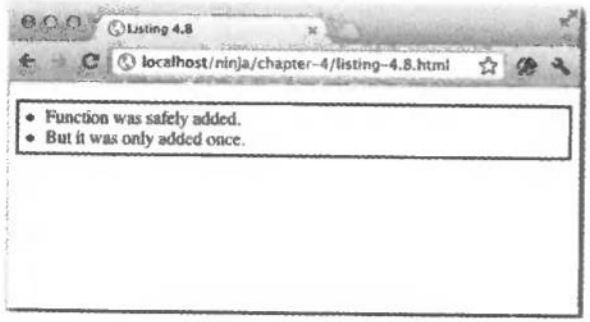

Рис. 4.4. Функцию можно отслеживать, наделяя ее свойством

Еще один полезный прием, который рекомендуется взять на вооружение, используя свойства функции, состоит в том, чтобы предоставить функции возможность видоизменяться. Таким приемом можно было бы воспользоваться для запоминания вычисленных ранее значений, экономя время при последующих вычислениях.

Самозапоминающиеся функции

Запоминание представляет собой процесс построения функции, способной запоминать свои вычисленные ранее значения. Это дает возможность заметно повысить производительность, избегая повторения лишний раз сложных вычислений, которые уже

были произведены. Этот прием будет рассмотрен сначала в контексте сохранения результатов затратных вычислений, а затем на более практическом примере сохранения списка найденных элементов модели DOM.

Сохранение результатов затратных вычислений

В качестве элементарного примера рассмотрим простой (и разумеется, не особенно эффективный) алгоритм вычисления простых чисел. И хотя это довольно простой пример, тем не менее он демонстрирует прием, который можно вполне применить в сложных и затратных вычислениях, например, для получения хеш-кода символьной строки по алгоритму шифрования MD5. Такие вычисления слишком сложны, чтобы продемонстрировать их здесь на конкретных примерах.

Внешне самозапоминающаяся функция практически ничем не будет отличаться от любой обычной функции, но в ее теле будет скрытно построен кеш, в котором она будет сохранять результаты выполняемых ею вычислений. Исходный код этой функции приведен в листинге 4.9.

Листинг 4.9. Запоминание вычисленных ранее значений

```
<script type="text/javascript">

  function isPrime(value) {
    if (!isPrime.anwers) isPrime.answers = {};      ❶ Создать кеш
    if (isPrime.answers[value] != null) {
      return isPrime.answers[value];                ❷ Проверить кешированные значения
    }
    var prime = value != 1; // единица не может быть простым числом
    for (var i = 2; i < value; i++) {
      if (value % i == 0) {
        prime = false;
        break;
      }
    }
    return isPrime.answers[value] = prime;          ❸ Сохранить вычисленное значение
  }

  assert(isPrime(5), "5 is prime!");
  assert(isPrime.answers[5], "The answer was cached!");   ❹ Проверить, все ли работает

</script>
```

Сначала в функции `isPrime()` проверяется, создано ли свойство `answers`, которое предполагается использовать в качестве кеша. И если это свойство отсутствует, то оно создается ❶. Создание пустого кеша происходит лишь при первом вызове функции, после чего он уже существует.

Затем проверяется, запомнен ли в кеше `answers` результат вычисления переданного значения ❷. Сохранение результатов вычислений в виде положительных или отрицательных ответов (`true` или `false`) будет осуществляться в этом кеше с использованием значения в качестве ключа свойства. Если в кеше найден ответ, он просто возвращается.

Если же в кеше не найдено ни одного значения, то выполняются вычисления с целью определить, является ли значение простым числом (для больших значений такая операция может оказаться весьма затратной), а результат сохраняется в кеше при возврате из функции ❸. Как показывают простые тесты ❹, запоминание работает успешно!

У такого подхода имеются два главных преимущества.

- Конечный пользователь запрашивает вычисленное ранее значение, выгодно используя преимущества вызова функции.
- Механизм запоминания действует слаженно и незаметно. Ни конечному пользователю, ни автору веб-страницы не нужно делать какие-либо специальные запросы или другие лишние инициализирующие операции, чтобы привести этот механизм в действие.

Но у данного подхода имеются также свои недостатки, которые могут даже перевесить его преимущества.

- Любого рода кеширование, безусловно, приносит в жертву расходование оперативной памяти в пользу повышения производительности.
- Пуристы могут заявить, что кеширование не следует смешивать с бизнес-логикой, поскольку функция или метод должны делать что-нибудь одно, и делать это хорошо.
- Проверить загрузку или измерить производительность такого алгоритма нелегко.

Рассмотрим еще один похожий пример.

Запоминание элементов модели DOM

Запрос на ряд элементов модели DOM по имени дескриптора относится к числу довольно распространенных операций, хотя эта операция и не отличается особой производительностью Но мы можем воспользоваться сильными сторонами функций в отношении запоминания, построив кеш для хранения отобранного ряда элементов. Обратимся к следующему примеру:

```
function getElements(name) {
  if (!getElements.cache) getElements.cache = {};
  return getElements.cache[name] =
    getElements.cache[name] ||
    document.getElementsByTagName(name);
}
```

Код запоминания (кеширования) довольно прост и не слишком усложняет общий процесс запроса. Но если произвести анализ производительности функции, то можно обнаружить, что даже на таком простом уровне кеширования производительность возрастает в 5 раз, как показано в табл. 4.1. Иметь в своем распоряжении такие сильные стороны функций совсем не плохо!

Таблица 4.1. Влияние запоминания на производительность кода*

Версия кода	Среднее	Минимальное	Максимальное	Число прогонов
Без кеширования	16,7	18	19	10
С кешированием	3,2	3	4	10

**Время указано в мс для 100 тыс. итераций в копии браузера Chrome 17.*

Даже приведенные выше простые примеры показывают, насколько полезными оказываются свойства функций. Они позволяют сохранять состояние и кешировать данные в одном, надежно скрытом месте, что дает несомненные преимущества не только в отношении организации, но и повышения производительности, не обращаясь к внешней памяти и не засоряя область действия объектами кеширования. Мы еще вернемся к механизму запоминания в последующих главах, когда его применение окажется более уместным.

Наделение функций свойствами, как и любых других объектов в JavaScript, — далеко не единственная сильная сторона функций. Немало сильных сторон функций связано с их контекстом, как будет показано далее на конкретном примере.

Имитация методов обработки массивов

Иногда требуется создать объект, содержащий коллекцию данных. Если бы дело ограничивалось только коллекцией данных, то для этой цели можно было бы просто воспользоваться массивом. Но в некоторых случаях в качестве состояния требуется сохранять не только саму коллекцию, но и определенные метаданные, связанные с элементами коллекции. С этой целью можно было бы, в частности, формировать новый массив всякий раз, когда требовалось бы создать новый вариант подобного объекта, вводя в него свойства и методы обработки метаданных. Напомним, что свойства и методы можно вводить в объекты, в том числе и массивы, сколько угодно. Но, как правило, такой способ оказывается медленным, не говоря уже о его трудоемкости.

Рассмотрим способ воспользоваться обычным объектом и наделить его нужными функциональными возможностями. Методы обращения с коллекциями уже имеются для объекта типа `Array` (функции-конструктора), но оказывается, что мы можем внедрить их в свои собственные объекты. Характерный тому пример приведен в листинге 4.10.

Листинг 4.10. Имитация методов обработки массивов

```
<body>

  <input id="first"/>
  <input id="second"/>

  <script type="text/javascript">

    var elems = {

      length: 0,

      add: function(elem){
        Array.prototype.push.call(this, elem);
      },

      gather: function(id){
        this.add(document.getElementById(id));
      }
    };
```

❶ Сохранить число элементов. Если имитируется массив, то нужно где-то хранить число сохраняемых его элементов

❷ Реализовать метод для ввода элементов в коллекцию. В прототипе Array для этого уже имеется метод, так почему бы не воспользоваться им вместо того, чтобы изобретать колесо?

❸ Реализовать метод gather() для поиска элементов по их значениям id и ввода их в коллекцию

```
    elems.gather("first");
    assert(elems.length == 1 && elems[0].nodeType,
           "Verify that we have an element in our stash");

    elems.gather("second");
    assert(elems.length == 2 && elems[1].nodeType,
          "Verify the other insertion");
  </script>
</body>
```

❹ Проверить методы gather() и add()

В данном примере создается обычный объект, который наделяется имитируемыми функциональными возможностями массива. Сначала в нем определяется свойство `length` для записи количества сохраняемых элементов, как в настоящем массиве ❶. Затем определяется метод `add()` для ввода элемента в конце имитируемого массива ❷. Вместо того чтобы писать код самостоятельно, мы воспользовались собственным методом обращения к массивам в JavaScript: `Array.prototype.push()`. (Часть `prototype` данной ссылки на метод `push()` пусть вас не смущает — подробнее о ней речь пойдет в главе 6. А до тех пор ее следует рассматривать как свойство, где конструкторы хранят свои методы.)

Как правило, метод `Array.prototype.push()` оперирует собственным массивом через свой контекст функции. Но в данном случае в качестве его контекста мы устанавливаем свой объект, вызывая метод `call()` и принудительно делая этот объект контекстом метода `push()`. Этот метод инкрементирует свойство `length`, считая его свойством массива, и тем самым вводит нумерованное свойство в объект, ссылающийся на переданный элемент. В какой-то степени это действие носит подрывной характер, но оно наглядно показывает, что можно делать с переменчивыми контекстами объектов.

В методе `add()` предполагается получить ссылку на элемент, передаваемый для сохранения. И хотя такая ссылка иногда имеется в наличии, чаще всего она отсутствует, и поэтому приходится определять служебный метод `gather()`, находящий элемент по его значению `id` и вводящий его на хранение ❸. И наконец, выполняются два теста, в каждом из которых элемент вводится в объект с помощью метода `gather()`, а затем проверяется, правильно ли настроено свойство `length` и введены ли элементы в нужных местах ❹.

Поведение, продемонстрированное в этом разделе, граничит с беззаконием. Тем не менее оно не только раскрывает истинный потенциал, который таит в себе податливый характер контекстов функций, но и служит превосходным основанием для дальнейшего обсуждения тех трудностей, которые вызывает обращение с аргументами функций.

Списки аргументов переменной длины

В целом JavaScript — очень гибкий язык программирования. Именно этой гибкости он, главным образом, обязан своим нынешним положением. К числу гибких и эффективных языковых средств JavaScript относится способность функций принимать произвольное число аргументов. Подобное удобство дает разработчикам больше возможностей для управления процессом написания своих функций, а следовательно, и приложений.

Рассмотрим ряд простых примеров, наглядно демонстрирующих, как пользоваться с выгодой для себя списками аргументов переменной длины. Эти примеры позволят выяснить следующее.

- Предоставление нескольких аргументов функциям, способным принимать любое их количество.
- Применение списков аргументов переменной длины для реализации перегрузки функций.
- Правильное применение свойства `length` списка аргументов.

В JavaScript отсутствует перегрузка функций, хотя вы, вероятно, уже привыкли к этому средству языков объектно-ориентированного программирования. Поэтому гибкость списка аргументов служит ключом к получению преимуществ, аналогичных тем, которые дает перегрузка в других языках. Рассмотрим сначала применение метода `apply()` для передачи переменного числа аргументов.

Предоставление переменного числа аргументов с помощью метода `apply()`

В любом языке программирования можно найти операции, которые вполне очевидны с точки зрения программирующих, но упущены разработчиками языка по каким-то совершенно непонятным причинам. И в этом отношении JavaScript не является исключением. К числу таких очевидных упущений относятся операции нахождения наибольшего и наименьшего значений в массиве. Казалось бы, эти операции выполняются настолько часто, что должны быть непременно включены в состав JavaScript, но в действительности наиболее близкие к ним операции выполняют методы `min()` и `max()` из класса `Math`.

На первый взгляд, эти методы позволяют разрешить данное затруднение, но при более тщательном исследовании оказывается, что каждый из них предполагает получить список аргумент переменной длины, а не массив. И как досадно не иметь возможности предоставлять и то и другое. Это, например, означает, что вызовы метода `Math.max()` можно было бы написать следующим образом:

```
var biggest = Math.max(1,2);
var biggest = Math.max(1,2,3);
var biggest = Math.max(1,2,3,4);
var biggest = Math.max(1,2,3,4,5,6,7,8,9,10,2058);
```

Что же касается массивов, то написать нечто, подобное приведенному ниже, уже нельзя.

```
var biggest = Math.max(list[0],list[1],list[2]);
```

Если длина массива заранее неизвестна, то откуда знать, сколько аргументов следует передать? Но даже если длина массива известна, такое решение вряд ли может считаться удовлетворительным. Прежде чем отказаться от метода `Math.max()` и прибегнуть к циклическому обращению к массиву вручную, чтобы определить минимальное и максимальное значения, попробуем все же найти простой и надежный способ для использования массива в качестве списка аргументов переменной длины. И такой способ состоит в применении метода `apply()`.

Напомним, что методы `call()` и `apply()` доступны для *всех* функций — даже встроенных в JavaScript, как было показано ранее в примере имитации методов массива. Поэтому попробуем выгодно воспользоваться доступностью этих методов в функциях обследования массивов, как показано в листинге 4.11.

Листинг 4.11. Обобщенные функции min() и max() для массивов

```
<script type="text/javascript">
  function smallest(array){
    return Math.min.apply(Math, array);
  }

  function largest(array){
    return Math.max.apply(Math, array);
  }

  assert(smallest([0, 1, 2, 3]) == 0,
         "Located the smallest value.");
  assert(largest([0, 1, 2, 3]) == 3,
         "Located the largest value.");
</script>
```

❶ Реализовать функцию для нахождения наименьшего значения

❷ Реализовать функцию для нахождения наибольшего значения

❸ Проверить реализации обеих функций

В приведенном выше коде определяются две функции: одна – для нахождения наименьшего значения в массиве ❶, а другая – для нахождения наибольшего значения в массиве ❷. Обратите внимание на то, что в обеих функциях метод `apply()` служит для предоставления методам из класса `Math` значений из массивов, передаваемых в качестве списков аргументов.

При вызове функции `smallest()` передается массив `[0,1,2,3]`, как это делается в тестах ❸. Это, в свою очередь, приводит к вызову метода `Math.min()`, что функционально равнозначно следующему вызову:

```
Math.min(0,1,2,3);
```

Следует также заметить, что в качестве контекста указывается объект типа `Math`. Делать это совсем не обязательно, поскольку методы `min()` и `max()` будут действовать независимо от того, что именно передается в качестве контекста. Тем не менее ничто не мешает соблюдать аккуратность в данной ситуации.

Итак, мы выяснили, как пользоваться списками аргументов переменной длины при вызове функций. А теперь рассмотрим, как объявлять свои функции, чтобы принимать эти списки.

Перегрузка функций

В разделе главы 3, посвященном вызову функций, был представлен параметр `arguments`, неявно передаваемый всем функциям. Рассмотрим теперь этот важный параметр более подробно. Итак, он неявно передается всем функциям, что позволяет обрабатывать в них любое число передаваемых аргументов. Даже если определить только некоторое число аргументов, через параметр `arguments` все равно будут доступны *все* передаваемые аргументы. Обратимся к краткому примеру, наглядно демонстрирующему применение этого удобного средства для реализации эффективной перегрузки функций.

Обнаружение и обход списка аргументов

В других, в большей степени объектно-ориентированных языках программирования перегрузка методов обычно осуществляется путем объявления отдельных реализаций методов под одним и тем же именем, но с разными списками параметров. Совсем

иначе дело обстоит в JavaScript, где функции "перегружаются" в единственной реализации, видоизменяющей свое поведение путем проверки количества и характера передаваемых аргументов. Посмотрим, как этого можно добиться.

В листинге 4.12 приведен пример кода, где свойства нескольких объектов объединяются в одном корневом объекте. Этот код может послужить важным подспорьем для осуществления принципа наследования. (Более подробно об этом речь пойдет в главе 6 при обсуждении прототипов объектов.)

Листинг 4.12. Обход списка аргументов переменной длины

```
<script type="text/javascript">

  function merge(root){
    for (var i = 1; i < arguments.length; i++) {
      for (var key in arguments[i]) {
        root[key] = arguments[i][key];
      }
    }
    return root;
  }

  var merged = merge(
    {name: "Batou"},
    {city: "Niihama"});

  assert(merged.name == "Batou",
         "The original name is intact.");
  assert(merged.city == "Niihama",
         "And the city has been copied over.");
</script>
```

❶ Реализовать функцию merge()

Вызвать реализованную функцию

Проверить правильность ее работы

Первое, что можно заметить в данной реализации функции `merge()`, – это объявление единственного параметра `root` в ее сигнатуре ❶. Но это совсем не означает, что данную функцию можно вызывать только с единственным параметром. На самом деле функцию `merge()` можно вызывать как с любым числом параметров, так и вообще без них.

В JavaScript отсутствует правило, предписывающее передавать одно и то же число аргументов функции, поскольку в ее объявлении указаны конкретные параметры. Успешное обращение с этими аргументами (или их отсутствием) полностью зависит от определения самой функции, но в JavaScript подобные ограничения не накладываются. Тот факт, что функция объявлена с единственным параметром `root`, означает, что только к одному из всех аргументов, которые могут быть переданы функции, допускается обращаться по имени. В данном случае это первый аргумент.

Совет

Для того чтобы проверить, был ли передан аргумент, соответствующий именованному параметру, достаточно воспользоваться выражением `paramname === undefined`. В результате его вычисления получается логическое значение `true`, если соответствующий аргумент отсутствует.

Итак, первый переданный аргумент может быть доступен через параметр `root`, но как получить доступ к остальным аргументам, которые могли быть переданы функции?

Разумеется, через параметр `arguments`, ссылающийся на коллекцию всех переданных аргументов. Напомним, что в рассматриваемом здесь примере предпринимается попытка объединить свойства любого объекта, передаваемого в качестве второго *n*-го аргумента, в объекте, передаваемом в качестве первого аргумента (`root`). С этой целью организуется циклическое обращение к списку аргументов, начиная с индекса **1**, чтобы исключить первый аргумент. На каждом шаге цикла, где происходит обращение к объекту, передаваемому функции, осуществляется перебор всех свойств этого объекта и копирование любых обнаруженных свойств в объект `root`.

Совет

Если вам еще не приходилось иметь дело с оператором цикла `for-in`, то имейте в виду, что он просто организует циклическое обращение ко всем свойствам объекта, устанавливая имя свойства (`key`) в качестве элемента итерации.

Теперь должно быть очевидно, что возможность доступа и обхода коллекции `arguments` является эффективным механизмом для создания сложных и логически развитых методов. С помощью такого механизма можно проверять аргументы, передаваемые любой функции, чтобы гибко оперировать ими, даже если заранее неизвестно, что именно должно быть передано функции.

В таких библиотеках, как jQuery UI, перегрузка функций применяется довольно широко. Рассмотрим в качестве примера метод `dialog()` для создания виджета пользовательского интерфейса и управления им как плавающим диалоговым окном. Этот же метод используется для создания диалогового окна и выполнения операций в нем. В частности, для создания диалогового окна достаточно сделать следующий вызов:

```
$("#myDialog").dialog({ caption: "This is a dialog" });
```

Этот же метод используется для выполнения таких операций, как, например, открытие диалогового окна:

```
$("#myDialog").dialog("open");
```

На самом деле порядок действий в методе `dialog()` определяется в результате тщательной проверки того, что ему передано.

Рассмотрим еще один пример, в котором применение параметра `arguments` не так очевидно, как в примере из листинга 4.12.

Разбиение списка аргументов

В качестве следующего примера построим функцию, перемножающую первый аргумент на самый большой по значению из всех остальных аргументов. Такая функция вряд ли найдет полезное применение, но послужит примером, демонстрирующим дополнительные способы обращения с аргументами в функции. На первый взгляд, эта функция реализуется довольно просто: извлечь первый аргумент и перемножить его на результат применения уже знакомой нам функции `Math.max()` к значениям остальных аргументов. Но поскольку нам требуется передать только часть массива, начиная со второго его элемента, в списке аргументов функции `Math.max()`, то воспользуемся стандартным методом `slice()` для обработки массивов, чтобы сформировать новый массив, опустив в нем первый элемент из старого массива. Итак, напишем код для данного примера, как показано в листинге 4.13.

Листинг 4.13. Разбиение списка аргументов

```
<script type="text/javascript">

  function multiMax(multi){
    return multi * Math.max.apply(Math, arguments.slice(1));
  }

  assert(multiMax(3, 1, 2, 3) == 9, "3*3=9 (First arg, by largest.)");

</script>
```

Но при выполнении приведенного выше сценария получается довольно неожиданный результат, как показано на рис. 4.5. В чем же дело? По-видимому, не все так просто, как казалось на первый взгляд. Ранее в этой главе указывалось, что параметр `arguments` ссылается не на подлинный массив. И хотя он очень похож на настоящий массив (его элементы можно, в частности, перебрать в цикле `for`), тем не менее ему недостает всех основных методов обработки массивов, в том числе и очень удобного метода `slice()`.

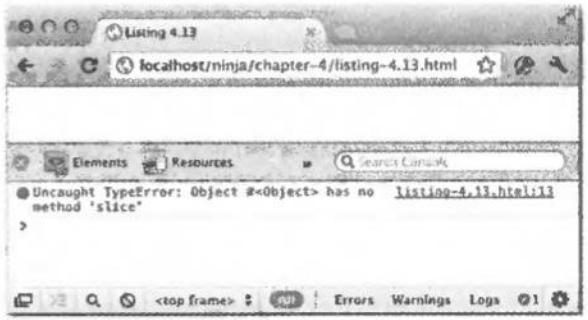

Рис. 4.5. Что-то неладно в датском королевстве, как и в коде этого сценария!

По желанию можно, конечно, создать ряд собственных методов разбиения аргументов в качестве специального набора инструментов или сформировать подлинный массив вручную, скопировав в него значения аргументов. Но оба эти подхода кажутся неуклюжими и избыточными. Тем более, что искомые функциональные возможности уже имеются у класса `Array`.

Прежде чем копировать данные или создавать специальные средства для разбиения аргументов, обратимся еще раз к коду из листинга 4.10, где мы обманным путем заставили метод из класса `Array` трактовать ненастоящий массив как подлинный. Воспользуемся этим приемом и перепишем код рассматриваемого здесь примера так, как показано в листинге 4.14.

Листинг 4.14. Разбиение списка аргументов — на этот раз успешно

```
<script type="text/javascript">

  function multiMax(multi){
    return multi * Math.max.apply(Math,
      Array.prototype.slice.call(arguments, 1));
  }
```

Заставить метод `slice()` обманным путем обрабатывать список аргументов, который не является экземпляром объекта типа `Array`

```
  assert(multiMax(3, 1, 2, 3) == 9,
         "3*3=9 (First arg, by largest.)");
</script>
```

В данном примере использован тот же самый прием, что и в коде из листинга 4.10, чтобы заставить метод `slice()` из класса `Array` трактовать "массив", на который ссылается параметр `arguments`, как подлинный, даже если он и не настоящий. Итак, выяснив, как обращаться с параметром `arguments`, рассмотрим ряд способов перегрузки функций на основании того, что нам уже известно.

Способы перегрузки функций

Что касается перегрузки функций — способа определения функции, выполняющей разные действия в зависимости от того, что ей передано, нетрудно себе представить, что подобную функцию несложно реализовать, используя рассмотренные до сих пор механизмы проверки списка аргументов и последующего выполнения действий в блоках условных операторов `if-then` и `else-if`. И зачастую такой подход оказывается вполне пригодным, особенно если действия, которые требуется выполнить, довольно просты.

Но как только дело начинает приобретать более сложный оборот, длинные функции с подобными кодовыми блоками очень быстро становятся громоздкими и неуклюжими. Поэтому в остальной части этого раздела мы рассмотрим способ, позволяющий создавать несколько функций с одинаковым именем, но отличающиеся друг от друга числом передаваемых им аргументов. Но самое главное, что они могут быть написаны как совершенно отдельные анонимные функции, а не монолитными блоками условных операторов `if-then-else-if`. Этот способ основывается на малоизвестном свойстве функций, которое мы рассмотрим в первую очередь.

Свойство `length` функции

У всех функций имеется интересное, но малоизвестное свойство `length`, дающее представление о том, каким образом функция была объявлена. Это свойство не следует путать со свойством `length` параметра `arguments`. Оно точно указывает количество параметров, заданных при объявлении функции.

Так, если объявить функцию с единственным формальным параметром, ее свойство `length` будет иметь значение `1`. Рассмотрим в качестве примера следующий фрагмент кода:

```
function makeNinja(name){}
function makeSamurai(name, rank){}
assert(makeNinja.length == 1, "Only expecting a single argument");
assert(makeSamurai.length == 2, "Two arguments expected");
```

Следовательно, об аргументах функции можно выяснить следующее:

- Количество именованных параметров, с которыми функция была объявлена, исходя из ее свойства `length`.
- Количество аргументов, переданных при вызове функции, исходя из свойства `arguments.length`.

Покажем, как воспользоваться свойством `length` для построения функции, с помощью которой можно создавать перегружаемые функции, отличающиеся количеством аргументов.

Перегрузка функций по числу аргументов

Перезагрузить конкретную функцию в зависимости от ее аргументов можно разными способами. Один из распространенных подходов к решению этой задачи состоит в том, чтобы выполнять разные операции в зависимости от типа передаваемых аргументов, а другой подход — в том, чтобы переключить порядок выполнения функции в зависимости от наличия или отсутствия определенных параметров. Еще один подход основывается на *числе* переданных аргументов. Именно этот подход и будет рассмотрен ниже.

Допустим, требуется иметь метод для объекта, выполняющий разные операции в зависимости от числа аргументов. Если бы для этой цели потребовалось написать длинную монолитную функцию, ее исходный код выглядел бы следующим образом:

```
var ninja = {

  whatever: function() {
        switch (arguments.length) {
          case 0:
            /* сделать что-нибудь одно */
            break;
          case 1:
            /* сделать что-нибудь другое */
            break;
          case 2:
            /* сделать что-нибудь третье */
            break;
        // и так далее ...
        }
    }
}
```

При таком подходе в каждой ветви оператора `case` выполняется другая операция в зависимости от числа аргументов, получаемых через параметр `arguments`. Но такой подход не очень изящный и, конечно, не красит настоящего мастера. Поэтому рассмотрим другой подход. Допустим, требуется ввести перегружаемый метод, используя синтаксис в следующих строках кода:

```
var ninja = {};
addMethod(ninja,'whatever',function(){ /* сделать что-нибудь одно */ });
addMethod(ninja,'whatever',function(a){ /* сделать что-нибудь другое */ });
addMethod(ninja,'whatever',function(a,b){ /* сделать что-нибудь третье */ });
```

В этом фрагменте кода сначала создается объект, а затем в него вводятся методы под одним и тем же именем `whatever`, но отличающиеся своими функциями в каждом случае перегрузки. Обратите внимание на то, что для каждой перегрузки фактически создается отдельная анонимная функция. И в конечном итоге код получается изящным и аккуратным.

Но ведь функция `addMethod()` не существует, а следовательно, ее придется создать самостоятельно. Свое оружие нужно держать всегда наготове, чтобы выходить с честью из подобных затруднительных положений. В листинге 4.15 приведен исходный код функции `addMethod()` для перегрузки методов, вводимых в объект.

Листинг 4.15. Функция перегрузки методов

```
function addMethod(object, name, fn) {

  var old = object[name];
  object[name] = function(){
    if (fn.length == arguments.length)
      return fn.apply(this, arguments)
    else if (typeof old == 'function')
      return old.apply(this, arguments);
  };
}
```

❶ Сохранить предыдущую функцию, так как ее придется вызвать, если число параметров и аргументов в переданной функции не совпадает

❷ Создать новую анонимную функцию, которая становится методом

❸ Вызвать переданную функцию, если число ее параметров и аргументов совпадает

❹ Вызвать предыдущую функцию, если число параметров и аргументов в переданной функции не совпадает

Функция `addMethod()` принимает три следующих аргумента.

- Объект, к которому привязывается метод.
- Имя свойства, с которым связывается метод.
- Объявление привязываемого метода.

Еще раз приведем пример применения данной функции в коде.

```
var ninja = {};
addMethod(ninja,'whatever',function(){ /* сделать что-нибудь одно */ });
addMethod(ninja,'whatever',function(a){ /* сделать что-нибудь другое */ });
addMethod(ninja,'whatever',function(a,b){ /* сделать что-нибудь третье */ });
```

При первом вызове функции `addMethod()` создается новая анонимная функция, которая обращается к функции, передаваемой в качестве параметра `fn`, если она вызывается со списком аргументов нулевой длины. А поскольку объект `ninja` пока еще новый, то для него ранее не установлено ни одного метода.

При втором вызове функции `addMethod()` сохраняется ссылка на анонимную функцию, сохраненную при предыдущем вызове в переменной `old` ❶, после чего создается еще одна анонимная функция, становящаяся методом ❷. В этом новом методе проверяется, равно ли **1** число переданных аргументов, и если это именно так, то вызывается функция, передаваемая в качестве параметра `fn` ❸. В противном случае вызывается функция, хранящаяся в переменной `old` ❹, где, напомним, проверяется отсутствие (нулевое число) параметров и вызывается вариант функции `fn` без параметров.

При третьем вызове функции `addMethod()` передается функция `fn` с двумя аргументами и весь процесс повторяется снова: создание еще одной анонимной функции, становящейся методом, вызов функции `fn` с двумя параметрами, если ей переданы два аргумента, а иначе обращение к ранее созданной функции с одним аргументом. В итоге функции как бы накладываются друг на друга слоями, причем в каждом слое проверяется совпадение числа параметров и аргументов, и если это число не совпадает, то происходит обращение к функции, созданной в предыдущем слое.

Доступ внутренней анонимной функции к параметрам `old` и `fn` организуется с помощью ловкого приема, в котором применяется понятие *замыкания*, подробнее рассматриваемое в следующей главе. А до тех пор следует лишь сказать, что когда внутренняя анонимная функция выполняется, она получает доступ к текущим значениям параметров `old` и `fn`. В листинге 4.16 производится проверка новой функции.

Если загрузить веб-страницу с приведенным выше сценарием, то все тесты пройдут успешно, как показано на рис. 4.6. Для проверки функции перегрузки методов сначала

определяется базовый объект, содержащий некоторые тестовые данные, состоящие из Ф.И.О. известных мастеров программирования на JavaScript. Затем к этому объекту привязываются три метода с одинаковым именем find. Назначение всех трех методов — найти мастеров программирования на JavaScript по критериям, передаваемым этим методам.

Листинг 4.16. Проверка функции addMethod()

```
<script type="text/javascript">

  var ninjas = {
    values: ["Dean Edwards", "Sam Stephenson", "Alex Russell"]
  };

  addMethod(ninjas, "find", function(){
    return this.values;
  });

  addMethod(ninjas, "find", function(name){
    var ret = [];
      for (var i = 0; i < this.values.length; i++)
        if (this.values[i].indexOf(name) == 0)
          ret.push(this.values[i]);
      return ret;
  });

  addMethod(ninjas, "find", function(first, last){
    var ret = [];
    for (var i = 0; i < this.values.length; i++)
      if (this.values[i] == (first + " " + last))
        ret.push(this.values[i]);
    return ret;
  });

  assert(ninjas.find().length == 3,
        "Found all ninjas");
  assert(ninjas.find("Sam").length == 1,
        "Found ninja by first name");
  assert(ninjas.find("Dean", "Edwards").length == 1,
         "Found ninja by first and last name");
  assert(ninjas.find("Alex", "Russell", "Jr") == null,
        "Found nothing");

</script>
```

❶ Объявить объект как базовый, предварительно загруженный тестовыми данными

❷ Привязать к базовому объекту метод без аргументов

❸ Привязать к базовому объекту метод с одним аргументом

❹ Привязать к базовому объекту метод с двумя аргументами

Проверить привязанные методы

Далее объявляются и привязываются следующие три варианта метода `find()`.

- Один метод, не ожидающий аргументов и возвращающий Ф.И.О. всех мастеров ❷.
- Другой метод, ожидающий один аргумент и возвращающий Ф.И.О. любых мастеров, имена которых начинаются с передаваемого ему текста ❸.
- Третий метод, ожидающий два аргумента и возвращающий Ф.И.О. любых мастеров, имена и фамилии которых совпадают с передаваемыми ему символьными строками ❹.

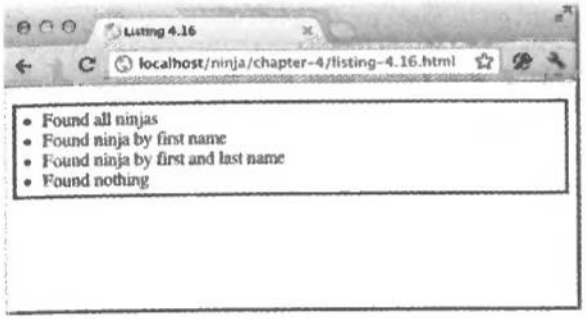

Рис. 4.6. Мастера программирования на JavaScript найдены с помощью методов, перегружаемых под одинаковым именем find()

Особое изящество данного способа проявляется в том, что привязываемые методы на самом деле не сохраняются ни в одной из типичных структур данных. Напротив, все они хранятся как ссылки в замыканиях, более подробно рассматриваемых в следующей главе. Следует, однако, указать на следующие предостережения, которые необходимо принимать во внимание, пользуясь этим конкретным способом:

- Перегрузка подходит только для разного числа аргументов. Она не различает их по типу, имени или иному критерию, но именно это зачастую и требуется.
- Методам, перезагружаемым подобным способом, присущи дополнительные издержки на вызов функций. Это обстоятельство следует непременно принимать во внимание в тех случаях, когда требуется обеспечить высокую производительность.

Тем не менее рассмотренная выше функция служит хорошим примером, наглядно демонстрирующим некоторые приемы функционального программирования, а также дающим возможность представить свойство `length` функций. До сих пор в этой главе речь шла об интерпретации функций в качестве объектов высшего порядка. А теперь рассмотрим функции под другим углом зрения, обратив внимание на возможность проверить, является ли объект функцией.

Проверка объектов на функции

В заключение обзора роли функций в JavaScript покажем, как выяснить, является ли конкретный объект экземпляром функции, т.е. тем, что можно вызвать. Казалось бы, это несложная задача, если не принимать во внимание вопросы кросс-браузерной совместимости. Для ее решения, как правило, достаточно воспользоваться оператором `typeof`, как в приведенном ниже фрагменте кода.

```
function ninja(){}

assert(typeof ninja == "function",
       "Functions have a type of function");
```

Это типичный способ проверки объекта на функцию. Он всегда действует, если проверяемый объект действительно является функцией. Но ведь имеются и другие случаи, когда подобная проверка может дать ложноположительные результаты, о которых следует знать.

- **Браузер Firefox.** Выполнение проверки типа объекта с помощью оператора `typeof` в элементе разметки `<object>` в коде HTML дает неточный результат `"function"` вместо ожидаемого результата `"object"`.

- **Браузер Internet Explorer.** При попытке выяснить тип функции, являвшейся частью другого, уже не существующего окна (например, встраиваемого фрейма), сообщается, что ее тип неизвестен (`"unknown"`).
- **Браузер Safari.** Объект `NodeList` модели DOM считается функцией. Поэтому `typeof document.body.childNodes == "function"`.

Для того чтобы устранить препятствия в коде, учитывая эти особые случаи, требуется найти решение, которое было бы работоспособным во всех целевых браузерах, позволяя выяснить, насколько правильно отдельные функции (или иные типы объектов) сообщают о себе. И здесь открывается обширное поле для исследования, но, к сожалению, почти все встречающиеся способы заводят в тупик. Например, известно, что у функций имеются методы `apply()` и `call()`, но эти методы отсутствуют у проблематичных функций для браузера Internet Explorer. Впрочем, имеется один вполне надежный способ, состоящий в том, чтобы преобразовать функцию в символьную строку и определить ее тип по упорядоченному в последовательную форму значению, как показано в приведенном ниже коде.

```
function isFunction(fn) {
  return Object.prototype.toString.call(fn) === "[object Function]";
}
```

И даже такая проверка неидеальна. Тем не менее во всех случаях, подобных упомянутым выше, она пройдет успешно, давая правильное значение для дальнейшей обработки.

Примечание

Подробнее о том, что собой представляет свойство `prototype` и как оно действует, речь пойдет в главе 6. А пока достаточно сказать, что это очень важная составляющая функции-конструктора, определяющая, какие именно свойства и методы войдут в состав конструируемого объекта.

Но из всего сказанного выше имеется, как всегда, одно примечательное исключение. О типе методов обработки элементов из модели DOM в браузере Internet Explorer сообщается как об объекте следующим образом: `typeof domNode.getAttribute == "object"` и `typeof inputElem.focus == "object"`. Поэтому рассматриваемый здесь конкретный способ не охватывает данный случай.

Для правильной реализации функции `isFunction()` требуется особый прием: организовать доступ к внутреннему методу `toString()` свойства `Object.prototype`. По умолчанию этот метод возвращает символьную строку с внутренним представлением объекта (например, `Function` или `String`). Поэтому данный метод можно вызвать для любого объекта, чтобы получить доступ к его подлинному типу. Хотя подобный прием выходит далеко за рамки одной только проверки, является ли объект функцией. С его помощью можно также выяснить, является ли объект символьной строкой (`String`), регулярным выражением (`RegExp`), датой (`Date`) или объектом другого типа.

Причина, по которой не стоит прибегать к непосредственному вызову метода `fn.toString()`, носит двоякий характер.

- Для отдельных объектов, скорее всего, имеются свои собственные реализации метода `toString()`.
- Для большинство типов объектов в JavaScript уже предварительно задан метод `toString()`, перегружающий метод, предоставляемый по ссылке `Object.prototype`.

Непосредственный доступ по ссылке `Object.prototype` гарантирует, что в итоге будет получен исходный, но не перегружаемый вариант метода `toString()`, а следовательно, именно те сведения, которые и требуются. И это лишь один небольшой, но характерный пример преодоления скрытых препятствий, которые стоят на пути к составлению кросс-браузерных сценариев. Писать код, надежно работающий во многих браузерах, не так-то просто, но это просто необходимый навык для всякого, стремящегося разрабатывать устойчивые и работоспособные веб-приложения. Более подробно методики кросс-браузерной разработки веб-приложений будут рассматриваться в главе 11, где этим вопросам уделяется основное внимание.

Резюме

В этой главе знания, полученные о функциях в главе 3, были применены для разрешения целого ряда затруднений, которые могут возникнуть в процессе разработки веб-приложений. В частности, в ней было рассмотрено следующее.

- Анонимные функции, позволяющие создавать более мелкие исполняемые блоки, чем крупные функции, наполненные императивными операторами.
- Рекурсивные функции, на примере которых были продемонстрированы различные способы ссылки на функции:
  - по имени;
  - в виде метода (по имени свойства объекта);
  - по имени встраиваемой функции;
  - через свойство `callee` параметра `arguments`.
- Свойства функций, которые могут служить для хранения любой используемой информации, включая:
  - хранение одних функций в свойствах других функций для последующей ссылки и вызова;
  - применение свойств функций для создания кеша с целью запоминания.
- Умелое управление контекстом функции, передаваемым ей при вызове, которое позволяет обманным путем заставлять методы оперировать не предназначенными для них объектами. Такой обходной прием может оказаться полезным для применения методов, уже имеющихся для объектов типа `Array` и `Math`, в целях обработки конкретных данных.
- Перегрузка функций — способность функций выполнять различные операции в зависимости от передаваемых им аргументов. С этой целью можно проверить список аргументов, предоставляемых параметром `arguments`, и выяснить, что нужно делать в зависимости от типа или числа переданных аргументов.
- Проверка объекта, является ли он экземпляром функции, по результату `"function"` выполнения оператора `typeof`. Но это делается не без учета вопросов кросс-браузерной совместимости.

В одном из примеров кода, а точнее — в листинге 4.15, было широко использовано понятие *замыкания*, которое определяет, какие именно значения данных доступны для функции при ее выполнении. В следующей главе это очень важное понятие будет рассмотрено достаточно подробно.

5 Сближение с замыканиями

В этой главе...

- Принцип действия замыканий
- Применение замыканий для упрощения разработки веб-приложений
- Повышение быстродействия кода с помощью замыканий
- Разрешение типичных затруднений, возникающих при определении области действия, с помощью замыканий

Замыкания являются определяющим языковым средством JavaScript, тесно связанным с функциями, подробно рассмотренными в предыдущих главах. Несмотря на то что многие авторы веб-страниц обходятся при написании страничных сценариев без ясного представления о преимуществах замыканий, последние не только помогают сократить объем и уменьшить сложность сценариев, необходимых для оснащения страниц более развитыми средствами, но и позволяют делать то, что просто невозможно или слишком сложно осуществить без них. Внедрение замыканий и способы написания с их помощью кода окончательно определило перспективу JavaScript как языка программирования.

По традиции замыкания всегда считались средством языков исключительно функционального программирования, и поэтому особенно приветствовалось их внедрение в практику массовой разработки приложений. Теперь замыкания проникли во многие библиотеки JavaScript и другие развитые кодовые базы кода благодаря их способности существенно упрощать сложные операции. В этой главе мы всесторонне исследуем понятие замыканий и покажем, как правильно пользоваться ими с целью довести написание страничных сценариев до самого высокого уровня мастерства.

Принцип действия замыканий

Коротко говоря, *замыкание* — это область действия, которая создается при объявлении функции и позволяет ей получать доступ и манипулировать внешними по отношению к ней переменными. Иными словами, замыкания предоставляют функции доступ ко всем переменным и другим функциям, которые находятся в области действия при объявлении самой функции.

На первый взгляд понятие замыкание кажется вполне интуитивным, но не следует забывать, что объявленная функция может быть вызвана в любой последующий момент, даже *после* завершения области действия, в которой она была объявлена. Это понятие, вероятно, лучше всего пояснить на примере конкретного кода. Поэтому начнем с небольшого примера кода, приведенного в листинге 5.1.

Листинг 5.1. Простое замыкание

```
<script type="text/javascript">

  var outerValue = 'ninja';                          ❶ Объявить переменную в глобальной области действия

  function outerFunction() {
    assert(outerValue == "ninja","I can see the ninja.");   ❷ Объявить функцию в глобальной области действия
  }

  outerFunction();                                   ❸ Выполнить функцию

</script>
```

В приведенном выше простом примере кода переменная ❶ и функция ❷ объявляются в одной и той же области действия (в данном случае в глобальной). После этого функция выполняется. Как показано на рис. 5.1, переменная `outerValue` доступна для данной функции. Такой код вам, вероятно, приходилось писать не раз, даже не осознавая, что вы невольно создавали замыкание!

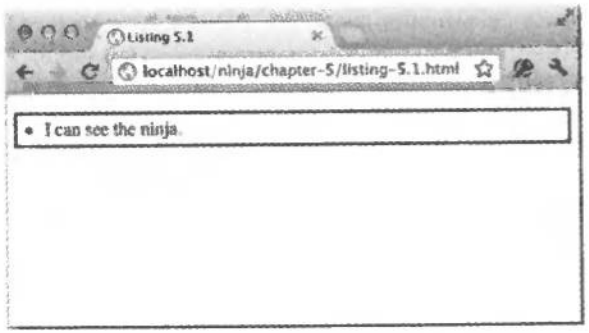

Рис. 5.1. Данная функция обнаружила ниндзя, скрывавшегося в пределах простой видимости

Приведенный выше пример, по-видимому, не особенно впечатляет и даже не удивляет. Внешние переменная и функция объявлены в глобальной области действия, которая вообще не завершается (до тех пор, пока страница загружена), и поэтому не удивительно, что переменная существует в данной области действия и по-прежнему доступна для функции. Но несмотря на наличие замыкания, его преимущества в данном примере пока еще неясны.

Усложним немного пример, как показано в листинге 5.2.

Листинг 5.2. Непростое замыкание

```
<script type="text/javascript">

  var outerValue = 'ninja';                          Объявить пустую переменную, чтобы воспользоваться ею позднее. Разумное именование позволяет лучше понять назначение элементов кода

  var later;

  function outerFunction() {                         Объявить переменную в функции. Область действия этой переменной ограничивается функцией, за пределами которой она недоступна
    var innerValue = 'samurai';

    function innerFunction() {                       Объявить внутреннюю функцию во внешней функции. При объявлении этой функции переменная innerValue оказывается в области ее действия
      assert(outerValue,"I can see the ninja.");
      assert(innerValue,"I can see the samurai.");
    }

    later = innerFunction;                           Сохранить ссылку на внутреннюю функцию в переменной later. Эта переменная находится в глобальной области действия, что дает возможность вызвать в дальнейшем данную функцию

  }

  outerFunction();                                   Вызвать внешнюю функцию и, как следствие, объявить внутреннюю функцию, а также присвоить ссылку на нее переменной later

  later();                                           Вызвать внутреннюю функцию через переменную later. Ее нельзя вызвать непосредственно, так как область ее действия (а также переменной innerValue) не выходит за пределы внешней функции

</script>
```

Проанализируем код из функции `innerFunction()` в приведенном выше примере и попробуем спрогнозировать, что при этом может произойти. Первое утверждение, безусловно, должно пройти, поскольку переменная `outerValue` находится в глобальной области действия и доступна повсюду. А что же второе утверждение?

Внутренняя функция выполняется *после* внешней благодаря копированию ссылки на нее в глобальную ссылку, сохраняемую в переменой `later`. Когда выполняется внутренняя функция, область действия во внешней функции уже завершена и недоступна на момент вызова этой функции по ссылке в переменной `later`. Следовательно, можно было бы с полной уверенностью предположить, что второе утверждение не пройдет, поскольку значение переменной `innerValue` не определено (`undefined`). Тем не менее тест проходит успешно, как показано на рис. 5.2.

Как же такое возможно? Каким чудом переменная `innerValue` все еще доступна и существует при выполнении внутренней функции после того, как область действия, в которой она была создана, уже давно завершена? Ответ, разумеется, следует искать в замыканиях.

При объявлении внутренней функции `innerFunction()` во внешней функции было определено ее объявление, а также образовано замыкание, охватывающее не только эту функцию, но и все переменные, находящиеся в области ее действия *на момент объявления*. Когда же функция `innerFunction()` выполняется, даже *после*

фактического завершения области действия, в которой она была объявлена, она по-прежнему имеет доступ к этой исходной области действия через свое замыкание, как показано на рис. 5.3.

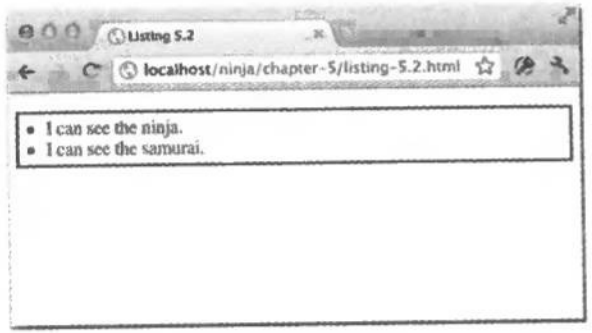

Рис. 5.2. Несмотря на попытку скрыться во внутренней функции, самурай выслежен!

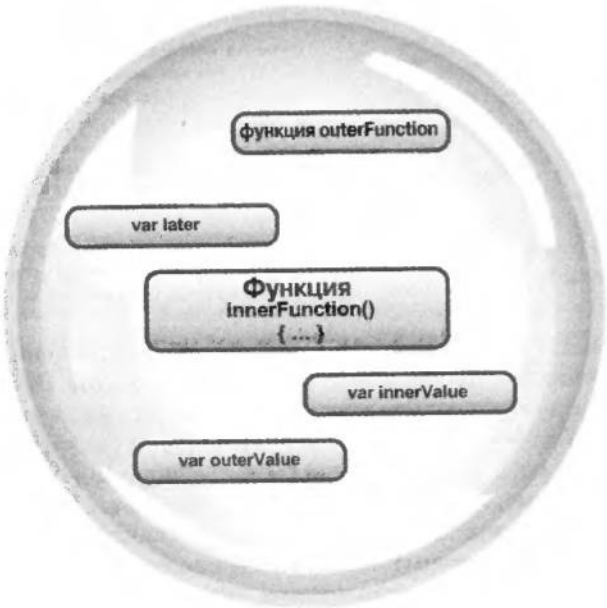

Рис. 5.3. Подобно защитной оболочке, замыкание функции `innerFunction()` *сохраняет переменные в области ее действия от "сборки мусора" до тех пор, пока существует сама функция*

В этом, собственно, и состоит принцип действия замыканий. Они образуют "защитную оболочку" вокруг функции и переменных, находящихся в области действия на момент объявления данной функции, и благодаря этому у нее имеется все необходимое для выполнения. Эта "оболочка", охватывающая как саму функцию, так и ее переменные, остается до тех пор, пока существует сама функция.

А теперь усложним еще больше данный пример, добавив несколько функций, чтобы исследовать дополнительные особенности замыканий. Эти дополнения выделены полужирным в коде из листинга 5.3.

Листинг 5.3. Что еще доступно в замыканиях

```
<script type="text/javascript">

  var outerValue = 'ninja';
  var later;

  function outerFunction() {
    var innerValue = 'samurai';

    function innerFunction(paramValue) {
      assert(outerValue,"Inner can see the ninja.");
      assert(innerValue,"Inner can see the samurai.");
      assert(paramValue,"Inner can see the wakizashi.");
      assert(tooLate,"Inner can see the ronin.");
    }

    later = innerFunction;
  }

  assert(!tooLate,"Outer can't see the ronin.");

  var tooLate = 'ronin';

  outerFunction();
   later('wakizashi');

</script>
```

❶ Параметр, введенный во внутреннюю функцию

❷ Проверить, доступен ли параметр функции и включены ли в замыкание переменные, объявленные после функции. Любопытно, что при этом произойдет?

❸ Найти значение переменной later в той же самой области действия. Интересно, пройдет ли этот тест?

❹ Объявить переменную после объявления внутренней функции

❺ Вызвать внутреннюю функцию, чтобы выполнить содержащийся в ней тест. Попробуйте предсказать его результаты

Преодолевая неизвестность, проанализируем, что же происходит в приведенном выше примере кода. По сравнению с кодом из предыдущего примера в этот код внесен целый ряд интересных дополнений. Так, во внутреннюю функцию добавлен параметр ❶, и поэтому при ее вызове через переменную `later` этой функции передается значение ❺. Кроме того, была добавлена переменная, объявленная после объявления внешней функции ❹. При выполнении тестов во внутренней ❷ и внешней ❸ функциях получаются результаты, приведенные на рис. 5.4.

Данный пример показывает три другие интересные особенности замыканий.

- Параметры функций включаются в ее замыкание. (Это кажется совершенно очевидным, но теперь мы сами убедились в этом.)
- Все переменные, находящиеся в области действия внешней функции, даже те, что объявлены *после* ее объявления, включаются в ее замыкание.
- Переменные, еще не определенные в той же самой области действия, недоступны по ссылке с упреждением.

Вторая и третья особенности замыканий объясняют, почему переменная `tooLate` доступна во внутреннем замыкании и недоступна во внешнем. Следует, однако, иметь

в виду, что за такую возможность хранить информацию и ссылаться на нее приходится нести прямые затраты, хотя и не вся эта информация доступна по ссылке непосредственно, поскольку не существует какого-то определенного объекта "замыкания", в котором она хранилась бы. Не следует также забывать, что каждая функция, получающая непосредственный доступ к информации посредством замыкания, связана напрямую с тем местом, где эта информация хранится. Следовательно, замыкания не только очень полезны, но и требуют определенных издержек. Всю эту информацию приходится хранить в оперативной памяти до тех пор, пока для механизма JavaScript не станет очевидно, что она больше не нужна и может быть благополучно собрана как "мусор", или же до тех пор, пока страница не выгрузится.

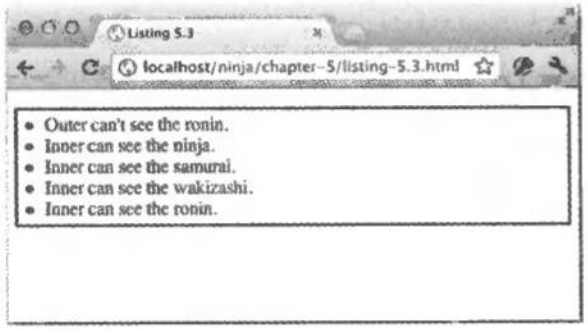

Рис. 5.4. Оказывается, что изнутри видно не дальше, чем снаружи!

Применение замыканий на практике

Итак, мы рассмотрели, что собой представляют замыкания и как они действуют — по крайней мере, на высоком уровне. А теперь покажем, как они применяются на практике при разработке веб-приложений.

Частные переменные

Замыкания чаще всего применяются для инкапсуляции некоторой информации в виде определенного рода "частной переменной" — иными словами, для ограничения области действия подобных переменных. Объектно-ориентированный код, написанный на JavaScript, не может иметь традиционные частные переменные (свойства объектов, недоступные для применения извне). Но, используя понятие замыкания, мы можем все-таки добиться похожего и вполне приемлемого результата, как показано в листинге 5.4.

Листинг 5.4. Приближенное представление частных переменных с помощью замыканий

```
<script type="text/javascript">

  function Ninja() {

    var feints = 0;
```

❶ Определить конструктор для объекта типа Ninja

❷ Объявить переменную в функции (конструкторе). Область действия этой переменной не выходит за пределы конструктора, поэтому это "частная" переменная. Она послужит для подсчета количества ложных выпадов, сделанных ниндзя


```
    this.getFeints = function(){
      return feints;
    };

    this.feint = function(){
      feints++;
    };
  }

  var ninja = new Ninja();

  ninja.feint();

  assert(ninja.getFeints() == 1,
         "We're able to access the internal feint count.");

  assert(ninja.feints === undefined,
         "And the private data is inaccessible to us.");

</script>
```

❸ Создать метод доступа к подсчету ложных выпадов. Переменная feints недоступна за пределами конструктора, поэтому ее значение доступно только для чтения

❹ Объявить метод для инкрементирования значения переменной feints. Это частная переменная, и поэтому ее значение никто не может изменить без разрешения. Посторонним предоставляется ограниченный доступ к ней через методы

❺ Протестировать код, построив сначала экземпляр объекта типа Ninja

❻ Вызвать метод feint(), в котором наращивается подсчет количества ложных выпадов со стороны ниндзя

❼ Проверить, можно ли получить прямой доступ к переменной

❽ Показать, что значение переменной feints увеличилось на 1, несмотря на отсутствие прямого доступа к ней. Воздействовать на значение переменной feints все-таки можно, поскольку она включена в замыкание (защитную оболочку), несмотря на то что конструктор, в котором она объявлена, завершился вместе со своей областью действия. Это замыкание образовано при объявлении метода feint() и доступно для него

В примере кода из листинга 5.4 создается функция, служащая в качестве конструктора ❶. Применение функции в качестве конструктора рассматривалось в предыдущей главе и будет продолжено в главе 6. А до тех пор достаточно напомнить, что если указать ключевое слово new при обращении к функции ❺, то будет создан экземпляр нового объекта, а функция — вызвана с новым объектом в качестве ее контекста, служа для этого объекта в качестве конструктора. Следовательно, ссылка this в теле функции указывает на полученный экземпляр нового объекта.

В самом конструкторе определяется переменная feints для хранения состояния ❷. Правила определения области действия переменных в JavaScript ограничивают доступность этой переменной *в пределах* конструктора. Для того чтобы получить доступ к значению данной переменной из кода за пределами области ее действия, в рассматриваемом здесь коде определяется *метод доступа* getFeints() ❸, с помощью которого можно читать, но не записывать значение в частной переменной. (Методы доступа нередко называются методами получения и просто "получателями".)

Далее реализуется метод feint() для полноценного контроля над значением переменной feints ❹. В реальном приложении это может быть какой-нибудь бизнес-метод, а в данном случае он просто инкрементирует значение переменной feints. После установки конструктора сначала он вызывается с ключевым словом new ❺, а затем и метод feint() ❻.

Как показывают тесты ❼ и ❽, с помощью метода доступа можно получить значение частной переменной, но не прямой доступ к ней. Это, по существу, не дает возможности вносить неконтролируемые изменения в значение переменной, как если бы она была полноценной частной переменной из языка объектно-ориентированного программирования. Данная ситуация наглядно иллюстрируется на рис. 5.5.

Благодаря этому состояние объекта типа Ninja можно сохранять в методе, не предоставляя прямой доступ к нему пользователю этого метода. Ведь переменная feints

доступна для внутренних методов через из замыкания, но не для кода, находящегося за пределами конструктора. Данный пример раскрывает объектно-ориентированный характер JavaScript, который будет более углубленно исследован в следующей главе. А до тех пор уделим основное внимание другому типичному примеру применения замыканий.

Рис. 5.5. Сокрытие переменной в конструкторе делает ее недоступной во внешней области действия, но она по-прежнему существует в замыкании

Обратные вызовы и таймеры

Еще одно полезное применение замыкания находят, когда приходится иметь дело с обратными вызовами и таймерами. В обоих случаях функция вызывается впоследствии, и внутри нее приходится обрабатывать некоторые внешние данные. Замыкания обеспечивают интуитивный способ доступа к данным, особенно если требуется избежать создания дополнительных переменных только для хранения этих данных. Рассмотрим простой пример Ajax-запроса, формируемого с помощью библиотеки jQuery для JavaScript, как показано в листинге 5.5.

Листинг 5.5. Применение замыканий в обратном вызове для Ajax-запроса

```
<div id="testSubject"></div>

<button type="button" id="testButton">Go!</button>

<script type="text/javascript">
  jQuery('#testButton').click(function(){
```

❶ Установить обработчик событий от щелчков кнопкой мыши на проверяемой кнопке. Эта функция передается методу click() и вызывается всякий раз, когда производится щелчок на кнопке.


```
    var elem$ = jQuery("#testSubject");

    elem$.html("Loading...");

    jQuery.ajax({
      url: "test.html",
      success: function(html){
        assert(elem$,
               "We can see elem$, via the closure for this callback.");
        elem$.html(html);
      }
    });
  });
</script>
```

❷ Объявить переменную elem$, чтобы хранить в ней ссылку на элемент разметки , определенный в самом начале кода

❸ Предварительно загрузить в элемент разметки текст, уведомляющий пользователей о происходящем

❹ Определить в списке аргументов, передаваемых методу ajax() из библиотеки jQuery, функцию обратного вызова при возврате из сервера ответа на Ajax-запрос. Текст ответа передается функции обратного вызова в качестве параметра html, вводимого в элемент разметки через переменную elem$ в замыкании

Несмотря на всю краткость примера кода из листинга 5.5, в нем происходит немало интересного. Начинается он с пустого элемента разметки веб-страницы, в который требуется загрузить текст "Loading..." (Идет загрузка) ❸ после щелчка на кнопке ❶. Между тем выполняется Ajax-запрос, по которому с сервера доставляется новое содержимое для загрузки в элемент разметки после возврата ответа на данный запрос.

Ссылаться на элемент разметки приходится дважды: первый раз – для его предварительной загрузки, а второй раз – для загрузки в него содержимого, доставляемого с сервера всякий раз, когда от него приходит ответ на запрос. Конечно, ссылку на элемент разметки можно было бы каждый раз находить, но ведь нам приходится действовать экономно, чтобы добиться нужной производительности, и поэтому мы ищем эту ссылку лишь один раз и, найдя, сохраняем ее в переменной elem$ ❷.

Совет

Применение знака $ в качестве суффикса или префикса принято в библиотеке jQuery для указания на то, что переменная содержит ссылку на объект jQuery.

Среди аргументов, передаваемых методу ajax() из библиотеки jQuery, определяется анонимная функция, служащая для обратного вызова по запросу ❹. В этой функции обратного вызова делается ссылка на переменную elem$ через замыкание, чтобы заполнить элемент разметки текстом ответа на запрос. Как отмечалось выше, несмотря на всю краткость кода из данного примера, в нем происходит немало поучительного. В частности, вам нужно сначала разобраться, почему функция обратного вызова может иметь доступ к переменной elem$. По желанию вы можете загрузить код из данного примера в окно браузера и установить точку прерывания на функции обратного вызова, чтобы выяснить, что же находится в области ее действия.

А теперь обратимся к чуть более сложному примеру кода, приведенного в листинге 5.6. В этом коде создается простая анимация.

Листинг 5.6. Применение замыкания при обратном вызове в интервале работы таймера

```
<div id="box"> ボックス </div>

<script type="text/javascript">
```

❶ Создать элемент, который предполагается оживить

```
function animateIt(elementId) {

  var elem = document.getElementById(elementId);
  var tick = 0;

  var timer = setInterval(function(){
    if (tick < 100) {
      elem.style.left = elem.style.top = tick + "px";
      tick++;
    }
    else {
      clearInterval(timer);
      assert(tick == 100,
             "Tick accessed via a closure.");
      assert(elem,
             "Element also accessed via a closure.");
      assert(timer,
             "Timer reference also obtained via a closure.");
    }
  }, 10);
}

animateIt('box');

</script>
```

❷ Получить ссылку на этот элемент в функции animateIt()

❸ Установить счетчик для отслеживания тактов анимации

❹ Создать и запустить таймер интервалов с функцией обратного вызова через каждые 10 мс. Положение элемента регулируется через каждые 100 тактов анимации

Остановить таймер после 100 тактов анимации и выполнить тесты, чтобы подтвердить доступность всех переменных, требующихся для анимации

Запустить анимацию, как только все будет готово!

Загрузив в окно браузера код из примера, приведенного в листинге 5.6, можно наблюдать анимацию, а результат ее выполнения представлен на рис. 5.6. В отношении этого кода следует особо заметить, что единственная анонимная функция ❹ используется в нем для анимации целевого элемента ❶. Для управления процессом анимации доступ к трем переменным в этой функции осуществляется через замыкание.

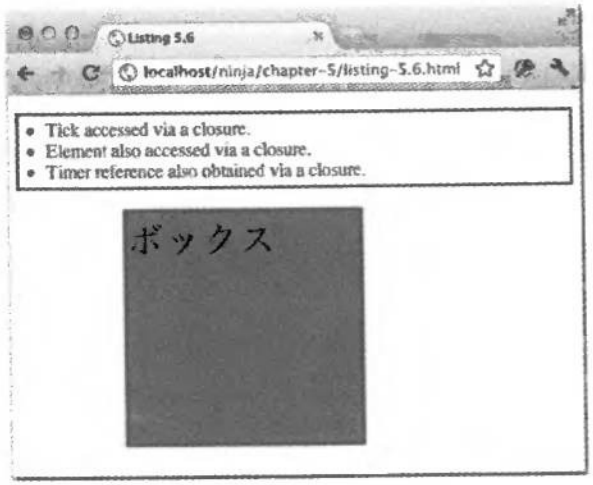

Рис. 5.6. Для отслеживания тактов анимации можно воспользоваться замыканиями

Все три переменные (ссылка на элемент модели DOM ❷, счетчик тактов анимации ❸ и ссылка на таймер ❹) должны сохраняться на протяжении тактов анимации. Кроме того, они не должны находиться в глобальной области действия. Но почему? Ведь код из рассматриваемого здесь примера будет по-прежнему работать нормально, если переместить переменные из функции `animateIt()` в глобальную область действия. К чему все эти строгие предостережения против засорения глобальной области действия?

Попробуйте сначала переместить переменные в глобальную область действия и убедитесь в том, что код из данного примера остается по-прежнему работоспособным. Затем видоизмените этот код для анимации двух элементов, добавив еще один элемент с однозначным идентификатором `id` и вызвав функцию `animateIt()` с этим идентификатором сразу же после ее вызова с идентификатором исходного элемента. Затруднение тотчас станет очевидным. Если оставить переменные в глобальной области действия, то для *каждого* вида анимации потребуется набор из трех переменных, иначе они будут перешагивать друг через друга при попытке использовать один и тот же набор переменных для отслеживания нескольких состояний.

Благодаря тому что переменные определяются в *теле* функции и становятся доступными через замыкания для обратных вызовов таймера, каждый вид анимации получает свою частную "оболочку" переменных, как показано на рис. 5.7. Без замыканий организовать одновременное выполнение нескольких действий, будь то обработка событий, анимация или даже формирование и обработка Ajax-запросов, было бы невероятно трудно. В этом, собственно, и состоит главная причина для внедрения замыканий в JavaScript!

Рассматриваемый здесь пример проясняет еще одну важную особенность замыканий. Значения переменных анимации не только доступны с момента, когда образовано замыкание, но и могут быть обновлены в замыкании до тех пор, пока функция выполняется в нем. Иными словами, замыкание — это не просто моментальный снимок состояния области действия в тот момент, когда образовано замыкание, но и активная инкапсуляция состояния, которое может быть видоизменено до тех пор, пока существует замыкание.

Данный пример особенно подходит для демонстрации возможности получать удивительно интуитивный и краткий код, используя принцип замыкания. Включая переменные в функцию `animateIt()`, мы образуем неявное замыкание, не прибегая к сложным синтаксическим конструкциям. Итак, разобрав примеры применения замыканий в различных функциях обратного вызова, перейдем к рассмотрению других возможностей их использования, начиная с приспосабливания с их помощью контекста функции под свои нужды.

Привязка контекста функций

В предыдущей главе при обсуждении контекста функции было показано, как методы `call()` и `apply()` могут быть использованы для манипулирования самим контекстом функции. И хотя такое манипулирование может оказаться очень полезным, оно все же таит в себе потенциальную опасность для объектно-ориентированного кода. Обратимся к примеру из листинга 5.7, где функция, служащая в качестве метода объекта, привязывается к элементу модели DOM как к приемнику событий.

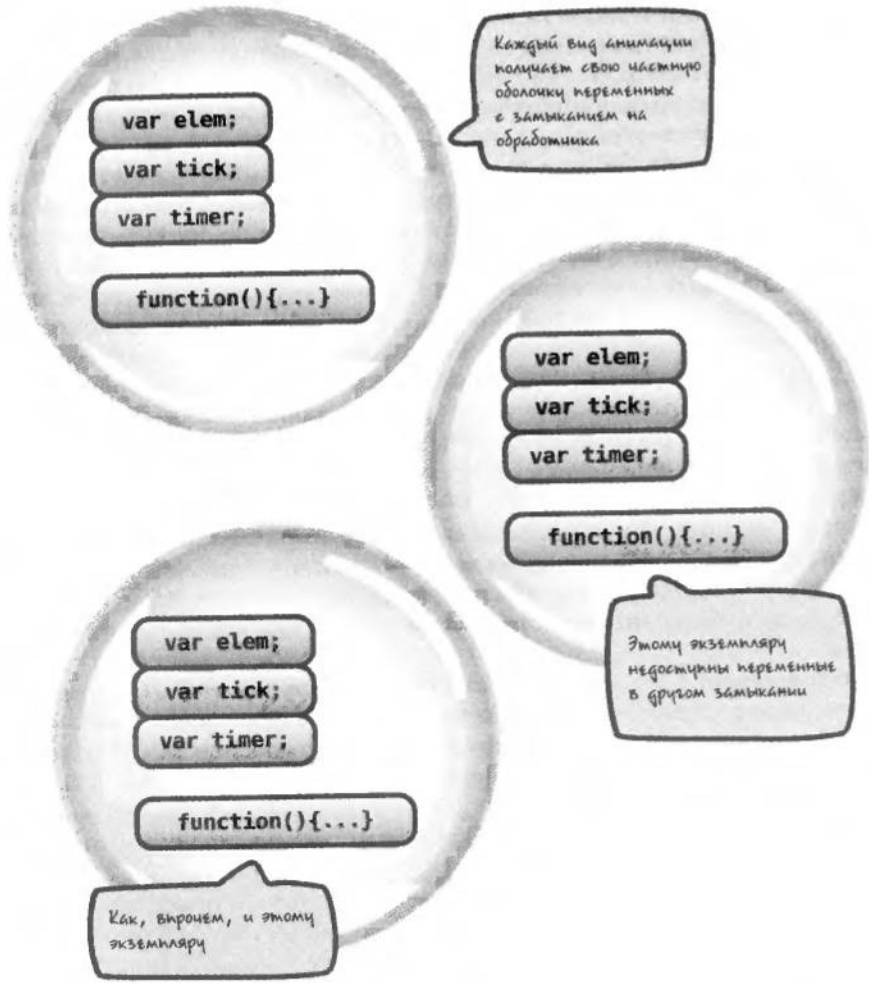

Рис. 5.7. Образуя несколько замыканий, можно организовать одновременное выполнение многих действий

Листинг 5.7. Привязка конкретного контекста к функции

```
<button id="test">Click Me!</button>

<script type="text/javascript">

  var button = {

    clicked: false,

    click: function(){
      this.clicked = true;
```

❶ Создать элемент разметки кнопки, чтобы назначить для него обработчик событий

❷ Определить объект для хранения состояния кнопки, чтобы с его помощью отслеживать, произведен ли щелчок на кнопке

❸ Объявить метод в качестве обработчика событий от щелчков на кнопке. Этот метод принадлежит объекту, что используется в функции для получения ссылки на объект


```
    assert(button.clicked,"The button has been clicked");
  }
};

var elem = document.getElementById("test");
elem.addEventListener("click",button.click,false);

</script>
```

❹ В самом методе проверяется, правильно ли было изменено состояние кнопки после щелчка на ней

❺ Установить обработчик событий от щелчков на кнопке

В данном примере создается кнопка ❶ и требуется выяснить, был ли вообще произведен на ней щелчок кнопкой мыши. Для хранения данных состояния (нажатой или отпущенной) кнопки создается вспомогательный объект `button` ❷, в котором определяется также метод в качестве обработчика событий ❸, запускающегося после щелчка на кнопке. В этом методе сначала устанавливается логическое значение `true` в свойстве `clicked`, а затем проверяется, правильно ли записано состояние кнопки во вспомогательном объекте ❹. А сам этот метод далее устанавливается в качестве обработчика событий от щелчков на кнопке ❺. Если загрузить код из данного примера в окно браузера и щелкнуть на кнопке, то появится не совсем понятный результат, приведенный на рис. 5.8: перечеркнутый текст явно указывает на то, что тест не прошел.

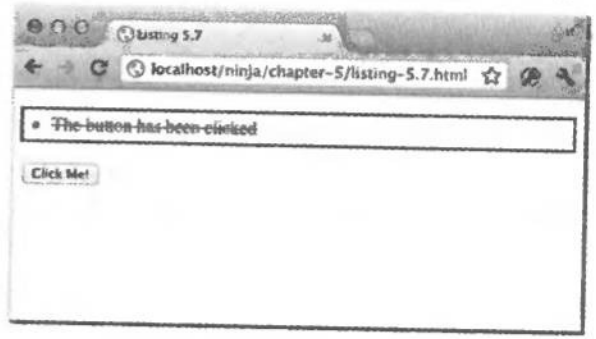

Рис. 5.8. Почему тест не прошел и куда девалось измененное состояние кнопки?

Тест кода из листинга 5.7 не проходит потому, что контекст функции `click()` *не* ссылается, как предполагалось, на объект `button`. Как пояснялось в главе 3, если вызвать функцию по следующей ссылке:

```
button.click()
```

то ее контекстом *действительно* станет объект кнопки. Но в данном примере система обработки событий в браузере определяет в качестве контекста вызываемой функции целевой элемент события. А это приводит к тому, что контекстом функции `click()` становится элемент разметки `<button>`, а не объект `button`. В итоге состояние `click` щелчка на кнопке попадает не в тот объект!

Установка по умолчанию целевого элемента в качестве контекста функции при вызове обработчика событий считается вполне допустимой, и ее можно и нужно учитывать в большинстве ситуаций. Но в данном случае она служит препятствием на пути к поставленной цели. Правда, его нетрудно устранить с помощью замыканий. В частности, мы можем сделать так, чтобы всегда устанавливался нужный контекст вызываемой

функции, используя для этой цели анонимные функции, метод `apply()` и замыкания в определенном сочетании. Рассмотрим в качестве примера код из листинга 5.8, который является обновленным вариантом кода из листинга 5.7 с дополнениями, выделенными полужирным. В этом коде контекст функции приспосабливается под конкретные нужды.

Листинг 5.8. Привязка конкретного контекста к обработчику событий

```
<script type="text/javascript">

  function bind(context,name){
    return function(){
      return context[name].apply(context,arguments);
    };
  }

  var button = {
    clicked: false,
    click: function(){
      this.clicked = true;
      assert(button.clicked,"The button has been clicked");
      console.log(this);
    }
  };

  var elem = document.getElementById("test");
  elem.addEventListener("click",bind(button,"click"),false);

</script>
```

❶ Определить функцию "привязки", заключающую в себя один вызов метода объекта внутри другого

❷ Использовать функцию привязки, чтобы установить объект кнопки в качестве контекста для обработчика событий

Весь секрет в приведенном выше коде кроется в добавлении функции `bind()` ❶. Эта функция служит для создания и возврата новой анонимной функции, вызывающей исходную функцию с помощью метода `apply()`, чтобы принудительно установить любой требующийся объект в качестве контекста данной функции. В данном случае это любой объект, передаваемый функции `bind()` в качестве первого аргумента. Этот контекст запоминается вместе с именем вызываемого метода, служащего в качестве конечной функции, через замыкание анонимной функции, включающее в себя параметры, передаваемые функции `bind()`.

В дальнейшем, когда дело доходит до установки обработчика событий, для его указания служит функция `bind()` вместо прямой ссылки `button.click` ❷. В итоге обертывающая анонимная функция становится обработчиком событий. И когда производится щелчок на кнопке, вызывается анонимная функция, которая, в свою очередь, вызывает функцию `click()`, принудительно устанавливая объект `button` в качестве контекста данной функции. Образующиеся при этом взаимосвязи приведены на рис. 5.9.

Данная конкретная реализация функции привязки основывается на том предположении, что будет использован метод существующего объект (т.е. функция, присоединенная к нему в качестве его свойства) и что этот объект должен стать контекстом данной функции. Исходя из этого предположения, функции `bind()` требуются только две порции информации: ссылка на объект, содержащий метод, а также имя данного метода.

Рис. 5.9. Анонимная функция служит в качестве заместителя настоящего обработчика событий, который определяется через параметры, привязываемые к замыканию

Функция `bind()` является упрощенным вариантом метода из распространенной библиотеки Prototype для JavaScript, стимулирующей к написанию кода в ясном и классическом объектно-ориентированном стиле. Исходный вариант метода из библиотеки Prototype выглядит приблизительно так, как показано в листинге 5.9.

Листинг 5.9. Пример кода, реализующего привязку контекста функции в библиотеке Prototype

```
Function.prototype.bind = function(){
  var fn = this, args = Array.prototype.slice.call(arguments),
    object = args.shift();

  return function(){
    return fn.apply(object,
      args.concat(Array.prototype.slice.call(arguments)));
  };
};

var myObject = {};
function myFunction(){
  return this == myObject;
}
```

❶ Добавить метод `bind()` ко всем функциям через его прототип, более подробно рассматриваемый в следующей главе

```
assert( !myFunction(), "Context is not set yet" );

var aFunction = myFunction.bind(myObject)
assert( aFunction(), "Context is set properly" );
```

Этот метод очень похож на функцию, реализованную в листинге 5.8, но у него имеются два важных дополнения. Прежде всего, он присоединяется ко всем функциям, а не представляется как глобально доступная функция ❶. С этой целью он вводится в качестве свойства `prototype` в класс `Function` языка JavaScript. Более подробно прототипы будут рассматриваться в главе 6, а до тех пор достаточно сказать, что их следует трактовать как центральные образцы для типов данных в JavaScript (в данном случае для всех функций).

Функция, привязываемая как метод ко всем функциям (через прототип), используется в коде следующим образом: `var boundFunction = myFunction.bind(myObject)`. С помощью данного метода можно также привязать аргументы к анонимной функции. Благодаря этому удается предварительно задать некоторые аргументы в форме частичного применения функций, рассматриваемого в следующем разделе.

Следует особо подчеркнуть, что метод `bind()` из библиотеки Prototype (или рассмотренный выше вариант его реализации) ни в коем случае не предназначен для замены таких методов, как `apply()` или `call()`. Напомним, что основная цель привязки контекста — управление им для отложенного выполнения через анонимную функцию и замыкание. Это важное отличие делает методы `apply()` и `call()` особенно полезными для организации обратных вызовов отложенного выполнения в обработчиках событий и таймерах.

Примечание

Начиная с версии JavaScript 1.8.5 для функций определен собственный метод `bind()`.

А теперь вернемся к предварительно заданным аргументам функции, которые упоминались выше. Они будут рассмотрены в следующем разделе вместе с понятием частичного применения функций.

Частичное применение функций

Частичное применение функции представляет собой особенно интересный способ задания аргументов функции перед ее выполнением. По существу, при частичном применении функции возвращается новая функция с предварительно заданными аргументами, которую можно затем вызвать. Это своего рода замещение функций, когда одна из них заменяет другую, вызывая ее во время своего выполнения, было использовано как специальный прием в предыдущем разделе для привязки конкретных контекстов к вызываемым функциям. А в этом разделе тот же самый прием будет использован по-другому.

Способ предварительного заполнения первых нескольких аргументов функции (а также возврата новой функции) обычно называется *каррированием*. Как обычно, за разъяснениями лучше всего обратиться к конкретным примерам. Но прежде чем перейти непосредственно к реализации каррирования, рассмотрим, как им лучше воспользоваться. Допустим, требуется разложить на составляющие символьную строку,

разделенную запятыми, исключая лишние пробелы. Это нетрудно сделать с помощью метода `split()` из класса `String`, снабдив его подходящим регулярным выражением, как показано ниже.

```
var elements = "val1,val2,val3".split(/,\s*/);
```

Примечание

Если у вас нет достаточной практики обращения с регулярными выражениями, то поясним, что приведенное выше выражение просто означает совпадение с запятой и любым последующим количеством пробелов. Приобрести необходимые навыки обращения с регулярными выражениями вы сможете, проработав материал главы 7.

Но постоянно помнить и набирать такое регулярное выражение неудобно. Поэтому реализуем метод `csv()`, который будет делать это за нас, используя каррирование, как показано в листинге 5.10.

Листинг 5.10. Частичное применение аргументов к собственной функции

```
String.prototype.csv = String.prototype.split.partial(/,\s*/);  ❶ Создать новую функцию в виде метода из класса String

var results = ("Mugan, Jin, Fuu").csv();  ❷ Вызвать каррированную функцию

assert(results[0]=="Mugan" &&
       results[1]=="Jin" &&
       results[2]=="Fuu",
       "The text values were split properly");  ❸ Проверить результаты
```

В коде из листинга 5.10 используется метод `split()` из класса `String` и пока еще не реализованная функция `partial()`, рассматриваемая далее в листинге 5.12, для предварительного заполнения регулярного выражения, по которому разделяется исходная символьная строка ❶. В итоге получается новая функция `csv()`, которую можно вызвать в любой момент для преобразования списка разделенных запятой значений в массив, не обращаясь непосредственно к регулярным выражениям ❷. На рис. 5.10 приведены результаты выполнения теста в браузере ❸. Таким образом, исходный замысел реализован так, как и было задумано. Если бы все задуманное так же легко воплощалось в жизнь в повседневной практике разработки!

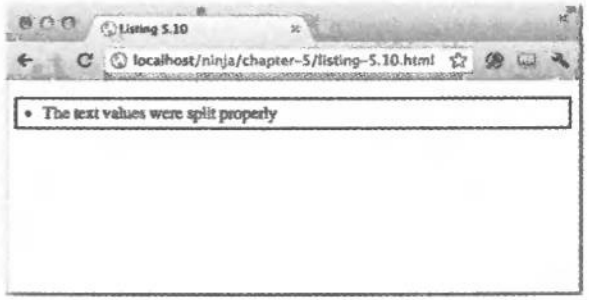

Рис. 5.10. Функция разложения символьной строки, разделенной запятыми, работает! Остается лишь реализовать ее полностью

Принимая все это во внимание, покажем, как частично применяемая или каррированная функция более или менее похоже реализуется в библиотеке Prototype. Ее исходный код приведен в листинге 5.11.

Листинг 5.11. Пример каррированной функции, предварительно заполняющей первые указанные аргументы

```
Function.prototype.curry = function() {
  var fn = this,
      args = Array.prototype.slice.call(arguments);
  return function() {
    return fn.apply(this, args.concat(
      Array.prototype.slice.call(arguments)));
  };
};
```

❶ Запомнить функцию и предварительно заполнить ее аргументами переменные, которые будут перехвачены в замыкании

❷ Создать анонимную каррированную функцию

Это еще один неплохой пример применения замыкания с целью запомнить состояние. В данном случае требуется запомнить расширяемую функцию (параметр `this` вообще не включается ни в одно из замыканий, поскольку при каждом вызове функции предоставляется отдельный параметр `this`), а также предварительно заполняемые аргументы ❶ и передать их вновь построенной функции ❷. Предварительно заполненные и новые аргументы сцепляются с новыми аргументами и передаются все вместе этой новой функции. В итоге получается метод, позволяющий предварительно заполнять аргументы, предоставляя в то же время новую, более простую функцию, которой можно затем воспользоваться.

Такой способ частичного применения функции, конечно, очень удобен, но мы можем добиться еще большего. Допустим, нам требуется заполнить *любой* аргумент, недостающий в заданной функции, а не только первые ее аргументы. Реализации подобного способа частичного применения функции существуют и в других языках программирования, но именно в Javascript его одним из первых продемонстрировал Оливер Стил (Oliver Steele) на примере своей библиотеки `Functional.js` (`http://osteele.com/sources/javascript/functional`). Пример одной из возможных реализаций этого способа приведен в листинге 5.12. (Именно этой реализацией мы и воспользовались, чтобы сделать работоспособным код из листинга 5.10.)

Листинг 5.12. Пример более сложного варианта частичного применения функции

```
Function.prototype.partial = function(){
  var fn = this, args = Array.prototype.slice.call(arguments);
  return function(){
    var arg = 0;
    for ( var i = 0; i < args.length && arg < arguments.length; i++ )
      if ( args[i] === undefined )
        args[i] = arguments[arg++];
      return fn.apply(this, args);
  };
};
```

Такая реализация, в сущности, очень похожа на метод `curry()` из библиотеки Prototype, но у нее имеются некоторые важные отличия. В частности, при вызове метода `partial()` пользователь может указать недостающие аргументы, которые

должны быть заполнены впоследствии, обозначив их как неопределенные значения (`undefined`). И для этого придется расширить возможности применяемого способа объединения аргументов. По существу, это означает, что необходимо организовать циклическое обращение ко всем передаваемым аргументам, выявить среди них соответствующие пробелы (неопределенные значения) и заполнить их указанными недостающими аргументами.

Выше уже демонстрировался пример построения функции, раскладывающей символьную строку на составляющие. Но теперь мы рассмотрим другие способы применения этих новых функциональных возможностей. Прежде всего, мы можем создать функцию, способную легко задерживать выполнение кода:

```
var delay = setTimeout.partial(undefined, 10);
  delay(function(){
  assert( true,
  "A call to this function will be delayed 10 ms." );
});
```

В этом фрагменте кода создается новая функция `delay()`, которой можно в любой момент передать другую функцию, чтобы вызвать ее асинхронно (спустя 10 миллисекунд). Мы могли бы также создать простую функцию для привязки событий, как показано ниже.

```
var bindClick = document.body.addEventListener
  .partial("click", undefined, false);

bindClick(function(){
  assert( true,
    " Click event bound via curried function." );
});
```

Подобным способом можно построить простые вспомогательные методы привязки событий для библиотеки. В итоге должен получиться более простой прикладной интерфейс API, где конечный пользователь будет избавлен от неудобств, которые доставляют ненужные аргументы, поскольку их указание фактически сведется к единственному вызову частично применяемой функции.

До сих пор мы пользовались замыканиями для упрощения кода, продемонстрировав тем самым некоторые сильные стороны функционального программирования на JavaScript. А теперь продолжим исследование замыканий, чтобы применять их для еще большего упрощения и наделения кода более развитой логикой работы.

Переопределение поведения функции

Положительным следствием столь полного контроля над функциями в JavaScript является тот факт, что мы можем полностью манипулировать их внутренним поведением без ведома пользователя. Этого можно, в частности, добиться двумя способами: видоизменив существующее поведение функций, не прибегая к замыканиям, или же получив новые самоизменяющиеся функции из уже существующих статических функций.

В главе 4 было представлено понятие запоминания. А теперь рассмотрим его под другим углом зрения.

Запоминание

Как пояснялось в главе 4, *запоминание* – это процесс построения функции, способной запоминать свои рассчитанные ранее результаты. Как было показано в той же главе, принцип запоминания очень просто реализуется в существующей функции. Но ведь функции, которые требуется оптимизировать, доступны далеко не всегда.

Рассмотрим в качестве примера метод `memoized()` из листинга 5.13, с помощью которого можно запомнить значения, возвращаемые существующей функции. В этом методе задействованы только функции, но не замыкания, как станет ясно из дальнейшего.

Листинг 5.13. Метод запоминания для функций

```
<script type="text/javascript">

  Function.prototype.memoized = function(key){
    this._values = this._values || {};
    return this._values[key] !== undefined ?
      this._values[key] :
      this._values[key] = this.apply(this, arguments);
  };

  function isPrime(num) {
    var prime = num != 1;
    for (var i = 2; i < num; i++) {
      if (num % i == 0) {
        prime = false;
        break;
      }
    }
    return prime;
  }

  assert(isPrime.memoized(5),
         "The function works; 5 is prime.");
  assert(isPrime._values[5],
         "The answer has been cached.");

</script>
```

❶ Проверить сначала, существует ли свойство _values, а иначе создать его объект, чтобы сохранить в нем кешированные значения

❷ Проверить при вызове функции с ключом, имеется ли для него кешированное значение. Если имеется, то возвратить его, а иначе сохранить вычисленное значение для последующих вызовов функции

❸ Вычислить простые числа в качестве теста

❹ Проверить, возвращает ли функция правильное значение и кешируется ли оно

В приведенном выше коде используется функция `isPrime()`, рассматривавшаяся в предыдущей главе ❸. Она по-прежнему действует неловко и медленно, выбирая для запоминания число, претендующее называться простым.

У нас нет достаточных средств, чтобы проникнуть внутрь существующей функции, но мы можем без особого труда ввести новые методы в функцию или вообще во *все* функции через свойства `prototype`. В данном примере новый метод `memoized()` вводится во все функции, что дает нам возможность заключать функции в оболочку и присоединять свойства, связанные с самой функцией. В частности, мы можем создать информационный массив (кеш) для хранения всех предварительно вычисленных значений. Рассмотрим, как это делается в данном примере.

Перед тем как выполнять любые вычисления или извлекать значения, мы должны убедиться в том, что информационный массив существует и что он присоединен к самой родительской функции. Это делается с помощью следующей простой операции укороченного вычисления ❶:


```
this._values = this._values || {};
```

Если свойство `_values` уже существует, то достаточно еще раз сохранить ссылку на это свойство функции, а иначе – создать новый информационный массив (первоначально пустой объект) и сохранить ссылку на него в свойстве `_values`. При вызове функции через метод `memoized()` проверяется, хранится ли значение в информационном массиве ❷. Если это значение хранится в нем, то оно возвращается. В противном случае значение вычисляется и сохраняется в информационном массиве для любых последующих вызовов данной функции.

Любопытно, что в рассматриваемом здесь коде вычисление и сохранение значения выполняется за один раз. Значение вычисляется при вызове метода `.apply()` для родительской функции и сохраняется непосредственно в информационном массиве. Но оператор `this` заключен в операторе `return`, а это означает, что результирующее значение также возвращается из родительской функции. Следовательно, вся последовательность событий (вычисление, сохранение и возврат значения) происходит в пределах единственной логической единицы кода.

Результаты тестирования ❹ рассматриваемого здесь кода показывают, что значения можно сначала вычислять, а затем кешировать. Но недостаток такого подхода состоит в том, что там где, вызывается функция `isPrime()`, нужно не забыть вызвать ее с помощью метода `memoized()`, чтобы пожинать плоды запоминания, а это никуда не годится.

Имея в своем распоряжении метод запоминания для контроля над входящими и выходящими значениями из существующей функции, рассмотрим возможность применить замыкания для получения новой функции, способной автоматически запоминать все свои вызовы. Таким образом, там, где эта функция вызывается, уже не нужно будет помнить, что ее следует непременно вызвать с помощью метода `memoized()`. Код, реализующий данный подход, выделен в листинге 5.14 полужирным.

Листинг 5.14. Способ запоминания функций с помощью замыканий

```
<script type="text/javascript">

  Function.prototype.memoized = function(key){
    this._values = this._values || {};
    return this._values[key] !== undefined ?
      this._values[key] :
      this._values[key] = this.apply(this, arguments);
  };

  Function.prototype.memoize = function(){
    var fn = this;
    return function(){
      return fn.memoized.apply( fn, arguments );
    };
  };

  var isPrime = (function(num) {
    var prime = num != 1;
```

❶ Перенести контекст функции в замыкание, присвоив его переменной, иначе контекст потеряется и вообще не войдем в замыкание

❷ Заключить исходную функцию в оболочку функции запоминания

```
    for (var i = 2; i < num; i++) {
      if (num % i == 0) {
        prime = false;
        break;
      }
    }
    return prime;
  }).memoize();

  assert(isPrime(17),"17 is prime");

</script>
```

Функция вызывается как обычно. Там, где она вызывается, уже не нужно беспокоиться о дополнительном запоминании

Код из листинга 5.14 основывается на коде из предыдущего примера, где был создан метод `memoized()`, а в данном коде добавлен еще один, новый метод `memoize()`. Он возвращает функцию, заключающую в оболочку исходную функцию с применяемым методом `memoized()`. Благодаря этому всегда возвращается запомненный вариант исходной функции ❷, а там, где она вызывается, уже не нужно применять метод `memoized()`.

Следует иметь в виду, что в методе `memoize()` путем копирования контекста функции в переменную образуется замыкание для запоминания исходной функции, которая получается из контекста и которую требуется запомнить ❶. И это довольно распространенный прием. Ведь у каждой функции имеется свой контекст, и поэтому контексты вообще не входят в замыкание. Но значения из контекста могут стать частью замыкания, если установить ссылку на значение в переменной. Запомнив исходную функцию, мы можем возвратить новую функцию, которая будет всегда вызывать метод `memoized()`. Благодаря этому мы получаем прямой доступ к запомненному экземпляру функции.

В листинге 5.14 продемонстрирован сравнительно необычный пример определения новой функции `isPrime()`. В частности, нам нужно было сделать так, чтобы функция `isPrime()` всегда запоминалась, а для этого нам пришлось построить временную функцию, результаты выполнения которой не должны запоминаться. Эта анонимная функция, определяющая простые числа, немедленно запоминается, а в итоге получается новая функция, которая присваивается переменной `isPrime`. Более подробно данная конструкция будет рассмотрена далее в этой главе. А в данном случае определить простое число без запоминания не удастся. Ведь существует единственная функция `isPrime()`, которая полностью инкапсулирует исходную функцию, скрытую в замыкании.

Пример из листинга 5.14 наглядно демонстрирует сокрытие исходных функциональных возможностей посредством замыкания. Такой прием может оказаться особенно полезным с точки зрения разработки, но в то же время ущербным. Ведь если скрывать в коде слишком много, то он перестанет быть расширяемым, что явно желательно. Впрочем, этому нередко препятствует неизбежная потребность вносить впоследствии изменения в код. Более подробно данный вопрос будет обсуждаться далее в этой книге.

Заключение функций в оболочку

Заключение функций в оболочку — это способ инкапсуляции логики самой функции и одновременная ее перезапись с новыми или расширенными функциональными возможностями. Такой прием лучше всего подходит для тех ситуаций, когда требуется пе

реопределить некоторое предыдущее поведение функции, оставив в то же время ряд других видов ее поведения в качестве определенных вариантов ее выполнения.

Заключение функций в оболочку обычно применяется для реализации фрагментов кросс-браузерного кода в тех случаях, когда приходится принимать во внимание отсутствие каких-нибудь функциональных возможностей в браузере. В качестве примера рассмотрим прием, позволяющий обойти программную ошибку в реализации доступа к атрибутам заголовка в браузере Opera. В библиотеке Prototype заключение функций в оболочку применяется как средство обойти эту программную ошибку.

Вместо крупного блока условных операторов `if-else` (это далеко не самый изящный прием, не особенно подходящий для разделения ответственности) в функции `readAttribute()` из библиотеки Prototype старый метод полностью переопределяется заключением в оболочку, тогда как остальные функциональные возможности оставляются исходной функции. Рассмотрим этот прием подробнее. Сначала создается обертывающая функция, позволяющая заключать другие функции в оболочку, а затем эта функция используется с целью создать оболочку для функции `readAttribute()` из библиотеки Prototype, как показано в листинге 5.15.

Листинг 5.15. Заключение старой функции в оболочку с новыми функциональными возможностями

```
function wrap(object, method, wrapper){

  var fn = object[method];

  return object[method] = function(){
    return wrapper.apply(this, [ fn.bind(this) ].concat(
      Array.prototype.slice.call(arguments)));
  };
}

if (Prototype.Browser.Opera) {

  wrap(Element.Methods, "readAttribute",
       function(orig, elem, attr){
    return attr == "title" ?
        elem.title :
        orig(elem, attr);
  });
}
```

Определить обобщенную обертывающую функцию, принимающую в качестве параметров объект, метод которого заключается в оболочку, имя этого метода, а также функцию, которая должна выполняться вместо исходного метода

Запомнить исходную функцию, чтобы в дальнейшем на нее можно было при желании сослаться через замыкание

Заключить в оболочку исходную функцию, создав новую функцию, вызывающую другую функцию, передаваемую в качестве оболочки. В новой функции эта функция-оболочка вызывается с помощью метода `apply()`, принудительно устанавливая объект в виде контекста с помощью функции `bind()` и передавая в качестве аргументов исходный метод с первоначальными аргументами

Использовать функцию `wrap()` для подстановки новых функциональных возможностей, если значение аргумента `attr` равно `"title"`, а иначе — исходную функцию

Использовать механизм из библиотеки Prototype для обнаружения браузера, чтобы выяснить, требуется ли заключить функцию в оболочку. Этот код взят из библиотеки Prototype, поэтому логично использовать в нем ее собственные ресурсы

Рассмотрим более углубленно, каким образом действует функция `wrap()`. Ей передается базовый объект, имя заключаемого в оболочку метода из этого объекта, а также новая обертывающая функция. Прежде всего, в данной функции сохраняется ссылка на исходный метод в переменной `fn` ❶, чтобы обратиться к ней в дальнейшем через замыкание анонимной функции, которую предполагается создать.

Затем происходит переопределение метода новой анонимной функцией ❷. Эта новая функция выполняет функцию-оболочку `wrapper()`, доступную через замыкание, передавая ей заполненный список аргументов. В данном случае необходимо, чтобы первым аргументом была исходная, переопределяемая функция. С этой целью создается массив, содержащий ссылку на исходную функцию, контекст которой привязывается с помощью метода `bind()` из листинга 5.8, чтобы соблюсти такой же контекст, как и у функции-оболочки. В этот же массив вводятся исходные аргументы. Как пояснялось в главе 3, этот массив используется в методе `apply()` в качестве списка аргументов.

В библиотеке Prototype функция `wrap()` служит для переопределения существующего метода (в данном случае — `readAttribute()`), заменяя его новой функцией ❶. Но эта новая функция по-прежнему имеет доступ к исходным функциональным возможностям в виде аргумента `original`, предоставляемого методом. Это означает, что функция может быть благополучно переопределена без потери своих функциональных возможностей. Применение замыкания, образуемого анонимной обертывающей функцией, наглядно показано на рис. 5.11.

Рис. 5.11. Благодаря замыканию анонимной обертывающей функции доступна не только исходная функция, но и передаваемая функция-оболочка

В конечном итоге получается повторно используемая функция `wrap()` для переопределения существующих функциональных возможностей методов объекта ненавязчивым образом. И все это становится возможным благодаря эффективному применению замыканий. А теперь рассмотрим нередко используемую синтаксическую конструкцию, которая на первый взгляд выглядит не совсем обычно, но в то же время играет очень важную роль в функциональном программировании.

Немедленно вызываемые функции

Ниже приведена очень важная языковая конструкция, применяемая в передовом функциональном программировании на JavaScript и опирающаяся на эффективное использование замыканий.

```
(function(){})()
```

Этот небольшой фрагмент кода необыкновенно универсален и придает языку JavaScript невиданную ранее эффективность. Но поскольку синтаксис этой языковой конструкции не совсем привычен, разберем ее детально. Не обращая внимания на содержимое первых круглых скобок, рассмотрим сначала конструкцию `(...)()`. Как известно, любая функция вызывается с помощью синтаксической конструкции `functionName()`, но вместо имени функции в не ней можно указать *любое* выражение, ссылающееся на экземпляр функции. Именно поэтому функцию можно вызвать, указав имя ссылающейся на нее переменной, как показано ниже.

```
var someFunction = function(){ ... };
result = someFunction();
```

Если оператор, а в данном случае это оператор вызова функции `()`, требуется применить ко всему выражению, такое выражение, как и любое другое, заключается в круглые скобки. Благодаря этому выражения `(3 + 4) * 5` и `3 + (4 * 5)` отличаются друг от друга. И это означает, что первая пара круглых скобок в конструкции `(...)()` лишь устанавливает разграничители, в которые заключается выражение, тогда как вторая пара выполняет роль оператора. Поэтому приведенный выше фрагмент кода можно изменить на совершенно законных основаниях, заключив в круглые скобки выражение, ссылающееся на функцию, как показано ниже.

```
var someFunction = function(){ ... };
result = (someFunction)();
```

Таким образом, все, что находится в первых круглых скобках, следует рассматривать как ссылку на выполняемую функцию. И хотя первые круглые скобки в последнем примере кода на самом деле *не* требуются, тем не менее такой синтаксис совершенно допустим. Если же вместо имени переменной ввести в первых круглых скобках непосредственно анонимную функцию, опустив ради краткости ее тело, то в итоге получится следующая синтаксическая конструкция:

```
(function(){...})();
```

А если пойти еще дальше, введя и тело функции, то данная синтаксическая конструкция развернется в следующую:

```
(function(){
  оператор-1;
  оператор-2;
...
  оператор-n;
})();
```

В конечном итоге получается выражение, выполняющее в одном операторе следующие действия:

- создание экземпляра функции;
- выполнение функции;
- удаление функции из-за отсутствия ссылок на нее по завершении оператора.

Не следует также забывать, что у такой функции, как и у любой другой, может быть замыкание, а это обеспечивает доступ ко всем внешним переменным и параметрам, которые находятся в той же самой области действия, что и оператор, в течение короткого срока действия данной функции. И как станет ясно в дальнейшем, эта простая конструкция, называемая *немедленно вызываемой функцией*, оказывается необычайно эффективной. Ее подробное рассмотрение мы начнем с того, как область действия взаимодействует с немедленно вызываемыми функциями.

Временная область действия и частные переменные

Используя немедленно вызываемые функции, мы можем приступить к построению довольно интересных замкнутых конструкций для наших целей. Такие функции выполняются немедленно, но, как и во всех остальных функциях, все их внутренние переменные остаются в их теле как в ограниченной области действия. Этим обстоятельством мы можем воспользоваться для создания временной области действия, в пределах которой можно сохранять определенное состояние. Исследуем, каким образом функционируют подобные временные и замкнутые области действия.

Примечание

Следует иметь в виду, что в JavaScript область действия переменных ограничивается той функцией, в которой они определены. Поэтому, создавая временную функцию, можно выгодно воспользоваться этим обстоятельством, чтобы сформировать временную область действия для переменных.

Создание замкнутой области действия

Рассмотрим следующий фрагмент кода:

```
(function(){
  var numClicks = 0;

  document.addEventListener("click", function(){
    alert( ++numClicks );
  }, false);
})();
```

Приведенная выше немедленно вызываемая функция выполняется немедленно (отсюда и ее название), поэтому и обработчик сразу же привязывается к событиям от щелчков кнопкой мыши. Следует также подчеркнуть, что для обработчика событий образуется замыкание, включающее в себя переменную `numClicks`. Благодаря этому переменная `numClicks` сохраняется вместе с обработчиком и становится доступной по ссылке, но *только* для него и нигде больше.

Это один из самых распространенных способов применения немедленно вызываемых функций в качестве простых, замкнутых оболочек. Все переменные, необходимые для нормальной работы таких функций, перехватываются в замыкании, но они не доступны нигде больше. А как насчет модульности? Но не следует забывать, что это все-таки функции, а следовательно, им можно найти и более интересное применение, как в приведенном ниже примере.


```
document.addEventListener("click", (function(){
  var numClicks = 0;

  return function(){
    alert( ++numClicks );
  };
})(), false);
```

Это, без сомнения, более сложный вариант предыдущего примера. И в этом случае создается немедленно вызываемая функция, хотя на сей раз из нее возвращается значение, а именно: функция, служащая в качестве обработчика событий. Возвращаемое значение передается методу `addEventListener()`, поскольку это такое же выражение, как и любое другое. Но внутренняя функция по-прежнему получает нужную ей переменную `numClicks` через свое замыкание.

Это совсем другой подход к трактовке области действия. В большинстве языков программирования область действия отдельных элементов определяется тем кодовым блоком, в котором они находятся. А в JavaScript область действия переменных определяется *замыканием*, в котором они находятся. Более того, с помощью рассматриваемой здесь простой языковой конструкции (немедленно вызываемых функций) мы можем теперь разделить область действия переменных на блоки, подблоки и уровни. Разделение области действия кода на такие единицы, как аргументы в вызове функции, обладает необыкновенными потенциальными возможностями, демонстрируя подлинную гибкость языка JavaScript.

Соблюдение имен в области действия через параметры

До сих пор рассматривались немедленно вызываемые функции, которым вообще не передается никаких параметров. Но ведь они подобны всем остальным функциям, и поэтому при вызове такой функции ей можно передать аргументы, а она, как и любая другая функция, обратится к этим аргументам по именам своих параметров. Рассмотрим следующий пример:

```
(function(what){ alert(what); })('Hi there!');
```

Более практический пример применения этой конструкции можно найти на веб-страницах, где используются средства из разных библиотек, например jQuery и Prototype. Так, имя `jQuery` внедрено в глобальной области действия библиотеки jQuery в качестве имени ее основной функции, а имя `$` — в качестве псевдонима этой функции. Но имя `$` довольно распространено в библиотеках JavaScript, и, в частности, оно применяется в библиотеке Prototype. В связи с этим в библиотеке jQuery поддерживается способ, позволяющий перейти к использованию имени `$` в любой другой библиотеке именно так, как это в ней принято, а реализуется он в функции `jQuery.noConflict()`. Поэтому на таких веб-страницах приходится использовать имя `jQuery` для ссылки на библиотеку jQuery, а имя **$** — для ссылки на библиотеку Prototype.

Мы привыкли пользоваться именем для ссылки на библиотеку jQuery и хотели бы делать это, не беспокоясь о том, что происходит в остальной части страницы. И это особенно справедливо для кода, повторно используемого на многих страницах, о компоновке и характере которых мы даже не подозреваем. Используя немедленно вызываемые функции, мы можем назначить имя **$** *обратно* для ссылки на библиотеку jQuery

внутри оболочки, образованной немедленно вызываемой функцией. С этой целью рассмотрим пример кода, приведенного в листинге 5.16.

Листинг 5.16. Соблюдение имени в охватываемой области действия

```
<body>

  <img src="../images/ninja-with-pole.png">

  <script type="text/javascript">

    $ = function(){ alert('not jQuery!'); };

    (function($){
      $('img').on('click',function(event){
        $(event.target).addClass('clickedOn');
      })

    })(jQuery);

  </script>

</body>
```

❶ Переопределить имя $ для ссылки на что-нибудь другое, а не на jQuery

❷ Немедленно вызываемая функция ожидает единственный параметр, которым является имя $. В этой функции данный параметр переопределяет любое использование имени $ в области действия более высокого уровня

❸ В теле функции имя $ используется так, как будто оно все еще назначено для jQuery. В функции используется не только имя $, но и обработчик событий. И хотя он вызывается намного позже, параметр $ привязан к нему посредством замыкания

❹ При обращении к немедленно вызываемой функции имя jQuery передается в качестве единственного аргумента для параметра $

В данном примере сначала переопределяется имя $ для ссылки на что-нибудь другое, а не на библиотеку `jQuery` ❶. Это может также произойти в результате включения на веб-странице средств из библиотеки Prototype, любой другой библиотеки или кода, присваивающего себе имя $. Но поскольку имя $ требуется нам для ссылки `jQuery` на одноименную библиотеку в данном фрагменте кода, то мы определяем немедленно вызываемую функцию с единственным параметром **$** ❷. В теле этой функции параметр **$** получит приоритет над глобальной переменной **$**. И все, что ни будет передано функции, станет доступно по ссылке из параметра **$** в *теле* самой функции. Так, если немедленно вызываемой функции передается имя **`jQuery`** ❹, оно становится значением параметра **$** в теле этой функции.

Следует иметь в виду, что параметр **$** становится частью замыкания любых функций, создаваемых в теле немедленно вызываемой функции, включая и обработчик событий, передаваемый методу `on()` из библиотеки jQuery ❸. И хотя обработчик событий, скорее всего, будет запущен намного позже выполнения и удаления немедленно вызываемой функции, тем не менее параметр **$** может быть использован в нем для ссылки `jQuery` на одноименную библиотеку.

Рассмотренным здесь приемом пользуются многие авторы модулей, подключаемых к библиотеке jQuery, если их код включается на тех веб-страницах, которые они не оформляли. Допускать, что имя $ ссылается на библиотеку jQuery, ненадежно, поэтому они могут включать код подключаемого модуля в тело немедленно вызываемой функции, чтобы благополучно пользоваться именем $ для ссылки на библиотеку jQuery. И прежде чем переходить к другой теме, рассмотрим еще один пример кода из библиотеки Prototype.

Сохранение удобочитаемости кода с помощью коротких имен

Нередко во фрагменте кода делаются частые ссылки на объект. Если ссылки длинные и сложные, то при многократном их повторении код становится трудно читаемым, а такой код никому не нужен. В качестве наивного выхода из этого затруднительного положения можно было бы присвоить ссылку на переменную с коротким именем следующим образом:

```
var short = Some.long.reference.to.something;
```

Несмотря на то что в последующем коде короткое имя `short` заменяет длинное имя `Some.long.reference.to.something`, вместе с тем новое имя без особой нужды вводится в текущую область действия, чего следует всячески избегать. Вместо этого настоящий мастер функционального программирования может воспользоваться немедленно вызываемой функцией, чтобы ввести короткое имя в *ограниченную* область действия. Ниже приведен пример того, как это сделано в библиотеке Prototype для JavaScript.

```
(function(v) {
  Object.extend(v, {
    href: v._getAttr,
    src: v._getAttr,
    type: v._getAttr,
    action: v._getAttrNode,
    disabled: v._flag,
    checked: v._flag,
    readonly: v._flag,
    multiple: v._flag,
    onload: v._getEv,
    onunload: v._getEv,
    onclick: v._getEv,
    ...
  });
})(Element._attributeTranslations.read.values);
```

В данном случае объект расширяется целым рядом свойств и методов из библиотеки Prototype. В приведенном выше коде можно было бы создать временную переменную для структуры данных `Element._attributeTranslations.read.values`, но вместо этого была выбрана ее передача в качестве первого аргумента немедленно вызываемой функции. Это означает, что первый аргумент (и параметр `v`) будет теперь служить вместо прежнего длинного имени ссылкой на приведенную выше структуру данных, оставаясь в то же время в области действия немедленно вызываемой функции.

Нетрудно заметить, что код становится более удобочитаемым благодаря ссылке по короткому имени `v` вместо длинного имени `Element.attributeTranslations.read.values`. Такая возможность создавать временные переменные в пределах заданной области действия особенно удобна для циклического обращения, о котором речь пойдет ниже.

Циклы

Одно из самых полезных применений немедленно вызываемых функций заключается в том, с их помощью можно устранить одну неприятную особенность циклов и замыканий. Примером тому служит типичный фрагмент проблематичного кода, приведенный в листинге 5.17.

Листинг 5.17. Код, в котором итератор не выполняет свои функции в замыкании

```
<body>

  <div>DIV 0</div>
  <div>DIV 1</div>

  <script type="text/javascript">

    var divs = document.getElementsByTagName("div");

    for (var i = 0; i < divs.length; i++) {
      divs[i].addEventListener("click", function() {
        alert("divs #" + i + " was clicked.");
      }, false);
    }

  </script>
</body>
```

Составить список из всех элементов разметки страницы (в данном случае из двух)

Каждый обработчик должен сообщить порядковый номер элемента DIV, но, как показано ниже, этого не происходит!

В приведенном выше коде должен выводиться порядковый номер каждого элемента разметки после щелчка на нем. Но если загрузить веб-страницу с этим кодом и щелкнуть на метке DIV 0, то на экране появится результат, представленный на рис. 5.12.

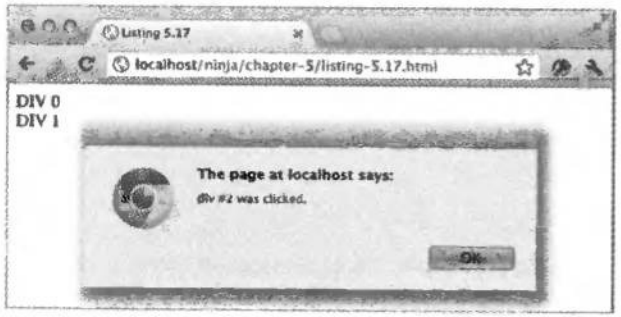

Рис. 5.12. Где произошла ошибка? Почему порядковый номер DIV 0 оказывается равным 2?

В коде из листинга 5.17 возникает затруднение, типичное для замыканий и циклического обращения, а именно: охватываемая переменная (в данном случае i) обновляется *после* привязки функции. А это означает, что каждая привязанная функция-обработчик будет всегда предупреждать лишь о последнем сохраненном значении переменной i (в данном случае это значение 2). И это происходит потому, что в замыканиях сохраняются только *ссылки* на включаемые в них переменные, а не их конкретные значения в тот момент, когда они создаются.

Незнание этого очень важного отличия сбивает с толку многих разработчиков. Но эту неприятную особенность замыкания можно преодолеть с помощью другого замыкания и немедленно вызываемой функции, вышибая, так сказать, клин клином, как показано в листинге 5.18, где изменения в коде выделены полужирным.

Листинг 5.18. Применение немедленно вызываемой функции для правильного обращения с итератором

```
<div>DIV 0</div>
<div>DIV 1</div>

<script type="text/javascript">
  var div = document.getElementsByTagName("div");

  for (var i = 0; i < div.length; i++) (function(n){
    div[n].addEventListener("click", function(){
      alert("div #" + n + " was clicked.");
    }, false);
  })(i);
</script>
```

Используя функцию немедленного вызова в качестве тела цикла for вместо предыдущего кодового блока, теперь можно добиться правильного соблюдения порядковых номеров элементов, передавая порядковый номер функции немедленного вызова, а следовательно, включая его в замыкание внутренней функции. Это означает, что в пределах области действия на каждом шаге цикла for переменная i определяется заново, и благодаря этому замыканию обработчика событий от щелчков кнопкой мыши передается именно то значение, которое и предполагается получить. После загрузки веб-страницы с обновленным кодом на экране появляется именно такой результат, какой и следовало ожидать (рис. 5.13).

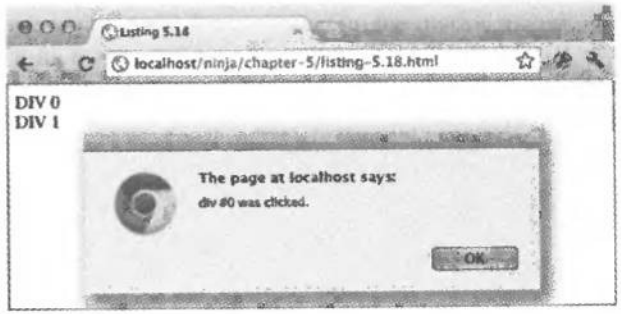

Рис. 5.13. Вот теперь совсем другое дело: каждому элементу разметки страницы присваивается правильный порядковый номер!

Данный пример наглядно показывает, что область действия переменных и значений можно точно контролировать, используя функции немедленного вызова и замыкания. А теперь рассмотрим, чем это может помочь при разработке библиотек JavaScript.

Заключение библиотеки в оболочку

Еще один важный пример точного контроля области действия, предоставляемой замыканиями и функциями немедленного вызова, имеет большое значение для разработки библиотек JavaScript. При разработке библиотеки очень важно не загромождать глобальное пространство имен ненужными переменными, особенно теми, которые используются лишь временно. И для этой цели особенно полезными оказываются за-

мыкания и функции обратного вызова. В частности, они позволяют сохранить закрытой как можно большую часть библиотеки и только выборочно вводить переменные в глобальное пространство имен. Такой подход обстоятельно воплощается в библиотеке jQuery, которая полностью объемлет все свои функциональные возможности и вводит переменные только по мере надобности, как, например, переменную `jQuery` в приведенном ниже фрагменте кода.

```
(function(){
  var jQuery = window.jQuery = function(){
  // инициализировать
  };

  // ...
})();
```

Обратите внимание на два намеренно сделанных присваивания в приведенном выше фрагменте кода. Сначала конструктор jQuery присваивается (в качестве анонимной функции) объекту типа `window.jQuery` с целью ввести его как глобальную переменную. Но это не гарантирует, что данная переменная будет оставаться в таком состоянии, поскольку вполне возможно, что она может быть изменена или удалена во внешнем, неподконтрольном нам коде. Во избежание этого она присваивается локальной переменной `jQuery`, чтобы сделать ее доступной в области действия функции немедленного вызова.

Это означает, что мы можем постоянно пользоваться именем `jQuery` в данной библиотеке, тогда как внешне с глобальной переменной может произойти все что угодно. А поскольку все функции и переменные, требующиеся в библиотеке, аккуратно инкапсулируются, то в результате конечный пользователь получает немало удобств в их применении так, как ему требуется. Но это далеко не единственный способ такого определения переменной. Ниже приведен другой способ.

```
var jQuery = (function(){
  function jQuery(){
  // инициализировать
  }

  // ...
  return jQuery;
})();
```

Этот код дает такой же результат, как и предыдущий код, но построен он иначе. В данном случае функция `jQuery` определяется в анонимной области действия, где она свободно используется, после чего происходит возврат к глобальной области действия, где эта функция присваивается переменной, которая также называется `jQuery`. Зачастую предпочтение следует отдавать именно такому способу, если экспортируется лишь единственная переменная, поскольку результат выполнения функции в данном случае более очевиден. В конечном счете выбор конкретных форматов и структур остается за разработчиком. И это хорошо, поскольку язык JavaScript предоставляет все свои средства и функциональные возможности, чтобы разработчик составил структуру своего приложения именно так, ему нужно.

Резюме

В этой главе были рассмотрены замыкания — одно из главных понятий функционального программирования на JavaScript.

- После изложения основ было показано, как замыкания реализуются и применяются в приложениях. На целом ряде примеров было продемонстрировано, насколько полезными могут быть замыкания, в том числе при определении частных переменных, в обратных вызовах и таймерах.
- Затем в этой главе был рассмотрен целый ряд понятий, расширяющих возможности языка JavaScript с помощью замыканий. К их числу относится соблюдение контекста функции, частичное применение функций и переопределение поведения функции. После этого были углубленно исследованы функции немедленного вызова, которые, как выяснилось, позволяют очень точно контролировать область действия переменных.
- В целом же ясное и полное представление о замыканиях становится бесценным достоянием для разработки сложных приложений на JavaScript и помогает разрешать целый ряд распространенных затруднений, с которыми приходится сталкиваться разработчикам.

В примерах кода, представленных в этой главе, было частично использовано понятие *прототипов*. И теперь настало время рассмотреть прототипы более основательно. Итак, сделав короткий перерыв, чтобы усвоить полученные в этой главе знания, перейдите к чтению следующей главы.

Объектно-ориентированное программирование с помощью прототипов

6

В этой главе...

- Применение функций в качестве конструкторов
- Исследование прототипов
- Расширение объектов с помощью прототипов
- Избежание типичных скрытых препятствий
- Построение классов с помощью наследования

Итак, выяснив, каким образом функции в JavaScript представлены в качестве объектов высшего порядка, мы можем перейти к рассмотрению другого важного свойства функций: их *прототипов*. Те, кто хотя бы немного знаком с прототипами в JavaScript, могут посчитать, что они тесно связаны с объектами, но и в этом случае речь идет не столько об объектах, сколько о функциях. Прототипы представляют собой удобный способ для определения типов объектов, но в то же время они фактически являются свойством функций.

В JavaScript прототипы широко применяются в качестве удобного средства для определения свойств и функциональных возможностей, автоматически применяемых к экземплярам объектов. Как только свойства прототипов определены, они становятся свойствами получаемых экземпляров объектов, служа в качестве своего рода образца для создания сложных объектов.

Иными словами, прототипы служат той же цели, что и классы в обычных языках объектно-ориентированного программирования. И действительно, в JavaScript прототипы применяются главным образом для написания кода в классическом объектно-ориентированном стиле с наследованием. Итак, приступим к их исследованию.

Получение экземпляров объектов и прототипы

У всех функций имеется свойство `prototype`, которое по умолчанию содержит пустой объект. Это свойство не служит никакой цели до тех пор, пока функция не будет использована в качестве *конструктора*. Как было показано в главе 3, с помощью ключевого слова `new` функция вызывается как конструктор с вновь созданным экземпляром пустого объекта в качестве ее контекста. А поскольку конструкторы чаще всего применяются для получения экземпляров объектов, то выясним сначала, как это на самом деле происходит.

Получение экземпляров объектов

Создать новый объект проще всего с помощью оператора, аналогичного приведенному ниже.

```
var o = {};
```

В этой строке кода создается новый пустой объект, который можно затем заполнить свойствами, используя операторы присваивания, как показано ниже.

```
var o = {};
o.name = 'Saito';
o.occupation = 'marksman';
o.cyberizationLevel = 20;
```

Но тем, кто имеет опыт объектно-ориентированного программирования, может недоставать инкапсуляции и структурирования, сопутствующих понятию конструктора класса как функции, служащей для инициализации объекта в известное исходное состояние. Ведь если бы потребовалось создать несколько экземпляров объекта одного и того же типа, то присваивание ему многих свойств по отдельности оказалось бы не только трудоемкой, но и не лишенной ошибок операцией. Для этой цели требуются средства, позволяющие объединить свойства и методы для объектов класса в одном месте.

И такой механизм в JavaScript предоставляется, хотя он и заметно отличается по своей форме от аналогичных механизмов в других языках программирования. Как и в языках объектно-ориентированного программирования вроде Java и C++, в JavaScript применяется оператор `new` для получения экземпляров новых объектов через конструкторы, но в то же время в JavaScript отсутствует определение класса как таковое. Вместо этого оператор `new` применяется к функции-конструктору, как пояснялось в главах 3 и 4, инициируя тем самым создание вновь размещаемого объекта.

В предыдущих главах не пояснялось, что прототип служит в качестве своего рода образца при получении экземпляров новых объектов. Поэтому рассмотрим далее, как это происходит на практике.

Прототипы в качестве образцов для объектов

Обратимся к простому примеру применения функции как с оператором `new`, так и без него, чтобы показать, каким образом свойство `prototype` предоставляет нужные свойства для экземпляра объекта. Исходный код данного примера приведен в листинге 6.1.

Листинг 6.1. Создание экземпляра объекта с прототипным методом

```
<script type="text/javascript">

  function Ninja(){}                                  ❶ Определить функцию, которая ничего не делает и не возвращает

  Ninja.prototype.swingSword = function(){            ❷ Ввести метод в прототип функции
    return true;
  };

  var ninja1 = Ninja();                               ❸ Вызвать функцию как таковую. Тестирование должно подтвердить, что вообще ничего не происходит
  assert(ninja1 === undefined,
        "No instance of Ninja created.");

  var ninja2 = new Ninja();                           ❹ Вызвать функцию как конструктор. Тестирование должно подтвердить, что новый экземпляр объекта не только создается, но и обладает методом из прототипа функции
  assert(ninja2 &&
        ninja2.swingSword &&
        ninja2.swingSword(),
        "Instance exists and method is callable." );

</script>
```

В приведенном выше коде сначала определяется функция `Ninja()` ❶, которая, по-видимому, ничего особенного не делает, но вызывается следующими двумя способами: как обычная функция ❸ и как конструктор ❹. После создания этой функции в ее свойство `prototype` вводится метод `swingSword()` ❷. Затем данная функция проверяется в ходе тестирования.

Сначала функция `Ninja()` вызывается как обычно ❸, сохраняя свой результат в переменной `ninja1`. Анализируя тело этой функции ❶, можно заметить, что она вообще не возвращает значение, и поэтому значение переменной `ninja1`, скорее всего, должно быть неопределенным (`undefined`), что и утверждается в тесте. Поэтому от функции `Ninja()`, какой бы простой она ни была, мало проку.

Затем функция `Ninja()` вызывается как *конструктор* через оператор `new`, и в этом случае все происходит совершенно иначе. Данная функция вызывается снова, но на этот раз создает вновь размещаемый объект, становящийся ее контекстом. Результат выполнения оператора `new` возвращается в виде ссылки на вновь созданный объект. И далее проверяется, содержит ли переменная `ninja2` ссылку на вновь созданный объект, а сам объект — метод `swingSword()`, который можно вызвать впоследствии.

Таким образом, прототип функции служит своего рода образцом для нового объекта, когда функция вызывается как конструктор. Результаты тестирования рассматриваемого здесь кода приведены на рис. 6.1.

Обратите внимание на то, что в конструкторе не было явно сделано ничего такого, что привело к упомянутому выше результату. Метод `swingSword()` присоединяется к новому объекту простым вводом в свойство `prototype` этого конструктора. Обратите также внимание на то, что метод *присоединяется*, а *не* добавляется к объекту. Выясним, почему это происходит именно так, а не иначе.

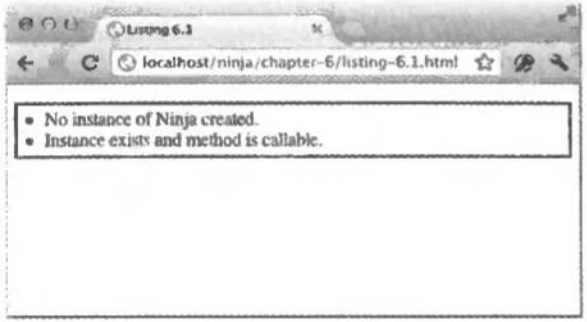

Рис. 6.1. Прототип позволяет предварительно задавать свойства, в том числе и методы, автоматически применяемые к новым экземплярам объекта

Свойства экземпляра объекта

Когда функция вызывается как конструктор через оператор `new`, в качестве ее контекста устанавливается новый экземпляр объекта. Это означает, что, помимо присоединения свойств через прототип, имеется также возможность инициализировать значения в функции-конструкторе через параметр `this`. Рассмотрим пример создания свойств такого экземпляра объекта в коде, приведенном в листинге 6.2.

Листинг 6.2. Наблюдение за старшинством операций инициализации

```
<script type="text/javascript">

  function Ninja(){

    this.swung = false;

    this.swingSword = function(){
      return !this.swung;
    };
  }

  Ninja.prototype.swingSword = function(){
    return this.swung;
  };

  var ninja = new Ninja();
  assert(ninja.swingSword(),
         "Called the instance method, not the prototype method.");

</script>
```

Создать переменную экземпляра, инициализировав ее логическим значением false

❶ Создать метод экземпляра, возвращающий обратное значение переменной экземпляра swung

❷ Определить метод прототипа с тем же самым именем, что и у метода экземпляра. Какой же из них получит старшинство?

❸ Построить экземпляр объекта типа Ninja и утвердить, что метод этого экземпляра переопределит одноименный метод прототипа. Пройдет ли этот тест?

Приведенный выше код очень похож на код из предыдущего примера в том отношении, что метод определяется путем ввода в свойство `prototype` конструктора ❷. Но точно так же именуемый метод вводится и в самой функции-конструкторе ❶. Оба метода определяются таким образом, чтобы возвращать противоположные результаты, по которым можно было бы различить вызываемый метод.

Примечание

В реальном коде этого делать не рекомендуется. Но в данном примере это делается лишь для того, чтобы продемонстрировать старшинство инициализаторов.

После загрузки страницы в браузер и выполнения теста ❸ оказывается, что тест проходит! Это наглядно показывает, что члены экземпляра объекта, создаваемые в конструкторе, получают приоритет над свойствами одноименного экземпляра, определяемого в прототипе. Старшинство операций инициализации имеет большое значение и соблюдается следующим образом.

1. Свойства привязываются к экземпляру объекта из прототипа.
2. Свойства добавляются к экземпляру объекта в функции-конструкторе.

Операции привязки в конструкторе всегда получают приоритет над аналогичными операциями в прототипе. А поскольку контекст, определяемый оператором `this` в конструкторе, ссылается на собственный экземпляр, то операции инициализации можно выполнять в самом конструкторе сколько угодно. А теперь рассмотрим подробнее взаимосвязь свойств экземпляра с прототипами, выяснив, каким образом в JavaScript увязываются ссылки на свойства объекта.

Увязывание ссылок

Очень важно разобраться, каким же образом в JavaScript увязываются ссылки и как в течение этого процесса вступает в действие свойство `prototype`. Из предыдущих примеров можно было бы сделать следующий вывод: когда новый объект создается и передается конструктору, свойства прототипа этого конструктора копируются в объект. При этом, конечно, следовало бы учитывать, что значение, присваиваемое свойству в теле конструктора, переопределяет соответствующее значение в прототипе. Но оказывается, что если бы все происходило именно так, то некоторые виды поведения просто не имели бы никакого смысла.

Если допустить, что значения просто копируются из прототипа в объект, то любые изменения, внесенные в прототип *после* построения объекта, не будут отражены в самом объекте. Попробуем реорганизовать немного код, как показано в листинге 6.3, чтобы посмотреть, к чему это приведет.

Листинг 6.3. Наблюдение за характером изменений в прототипе

```
<script type="text/javascript">

  function Ninja(){
    this.swung = true;
  }

  var ninja = new Ninja();

  Ninja.prototype.swingSword = function(){
    return this.swung;
  };

  assert( ninja.swingSword(), "Method exists, even out of order." );

<script>
```

❶ Определить конструктор, создающий объект типа Ninja с единственным свойством, принимающим логическое значение

❷ Получить экземпляр объекта типа Ninja, вызвав функцию-конструктор через оператор new

❸ Ввести метод в прототип после создания объекта

❹ Проверить, существует ли метод в объекте

В данном примере кода сначала определяется конструктор ❶, который затем используется для создания экземпляра объекта ❷. После того как будет создан экземпляр объекта, в прототип добавляется метод ❸. Далее выполняется тест, чтобы проверить, произошли ли изменения, внесенные в прототип после создания объекта. Этот тест проходит ❹, свидетельствуя о том, что сделанное в нем утверждение оказалось истинным, как показано на рис. 6.2. Очевидно, что при создании объекта можно выполнять не только простое копирование свойств.

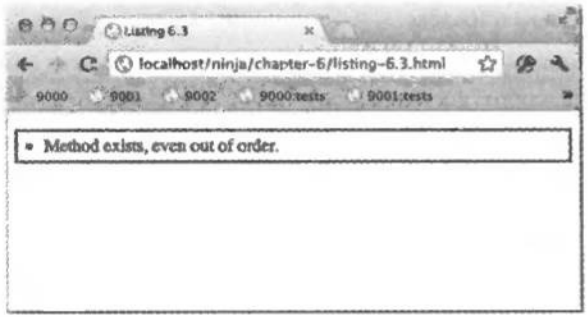

Рис. 6.2. Как подтверждает тест, изменения, вносимые в прототип, действительно происходят!

На самом деле свойства из прототипа никуда не копируются, а вместо этого сам прототип присоединяется к построенному объекту, и к нему происходит обращение в процессе увязывания ссылок, делаемых на свойства объекта. Ниже дается краткий обзор данного процесса.

1. Когда делается ссылка на свойство объекта, в самом объекте проверяется существование данного свойства. Если оно существует, то выбирается, а иначе...
2. Прототип, связанный с объектом, обнаруживается и проверяется на наличие искомого свойства. Если оно существует, то выбирается, а иначе...
3. Значение не определено (undefined).

Как будет показано далее в этой главе, на самом деле данный процесс сложнее, чем описано выше. Но и этого описания должно быть достаточно для его понимания. Для большей наглядности на рис. 6.3 приведена блок-схема, демонстрирующая данный процесс.

У каждого объекта в JavaScript имеется неявное свойство `constructor`, ссылающееся на конструктор, использовавшийся для создания объекта. А поскольку прототип является свойством конструктора, то каждый объект может найти свой прототип. Обратимся к примеру, приведенному на рис. 6.4, где показан моментальный снимок консоли JavaScript в браузере Chrome после загрузки кода из листинга 6.3.

Набрав на консоли ссылку `ninja.constructor`, можно заметить, что она, как и ожидалось, делается на функцию `Ninja()`, поскольку объект `ninja` был создан с помощью этой функции в качестве конструктора. Более глубокая ссылка `ninja.constructor.prototype.swingSword` обнаруживает порядок доступа к свойствам прототипа из экземпляра объекта. Этим объясняется, почему внесенные в прототип изменения происходят после того, как объект построен. Прототип активно присоединяется к объекту, и любые ссылки на свойства объекта увязываются, если требуется, с помощью прототипа *во время ссылки*.

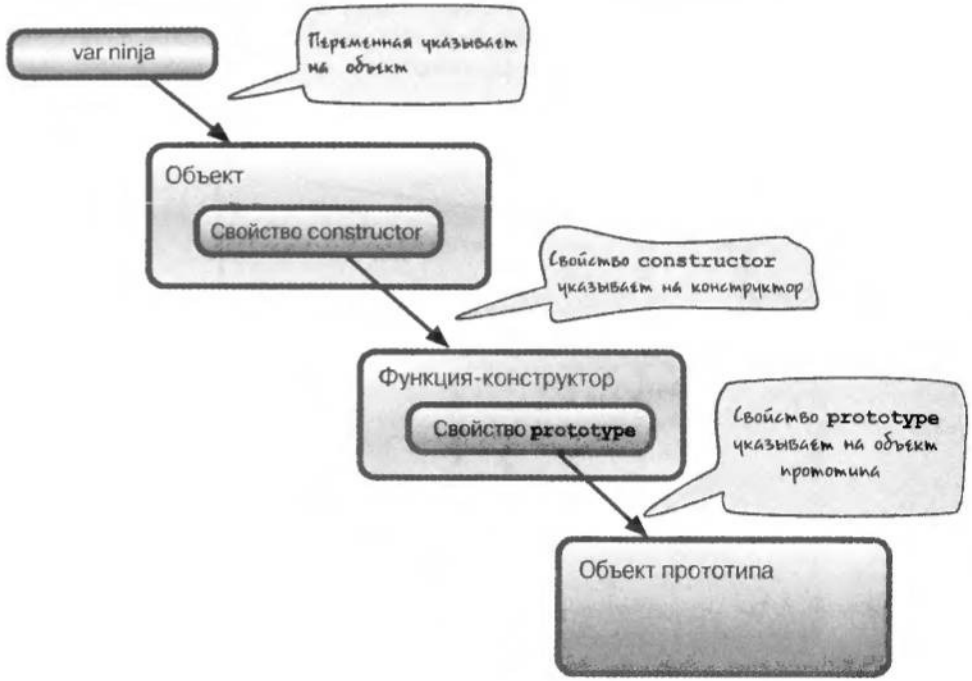

Рис. 6.3. Объекты связаны со своими конструкторами, которые, в свою очередь, связаны с прототипами объектов, создаваемых конструктором

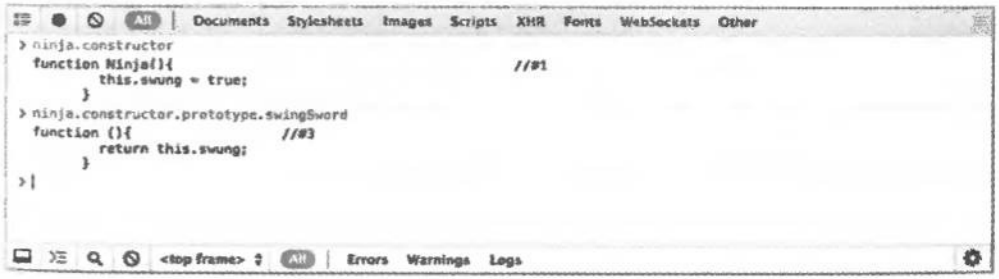

Рис. 6.4. При анализе структуры объекта обнаруживается путь к его прототипу

Такие плавные и активные обновления обладают невероятным потенциалом и расширяемостью до такой степени, которую обычно не удается найти в других языках программирования. Допуская эти активные обновления, вполне возможно построить функциональный каркас, который пользовали могут расширять дополнительными функциональными возможностями, даже после того, как экземпляры объектов уже получены. Возникающие при этом взаимосвязи приведены на рис. 6.5.

Как показано на рис. 6.5, у объекта, доступного по ссылке из переменной `ninja`, имеются свойства `member1` и `member2`. Ссылка на любое из этих свойств разрешается наличием самих этих свойств. Если же свойство, на которое делается ссылка, отсутствует у объекта, а в данном случае – это свойство `member3`, то его поиск осуществляется в прототипе конструктора. Так, ссылка на свойство `member4` приведет к неопределен-

ному результату (undefined), поскольку оно отсутствует везде. Прежде чем переходить к рассмотрению следующего вопроса, обратимся к еще одному примеру, являющемуся вариацией на ту же тему, как показано в листинге 6.4.

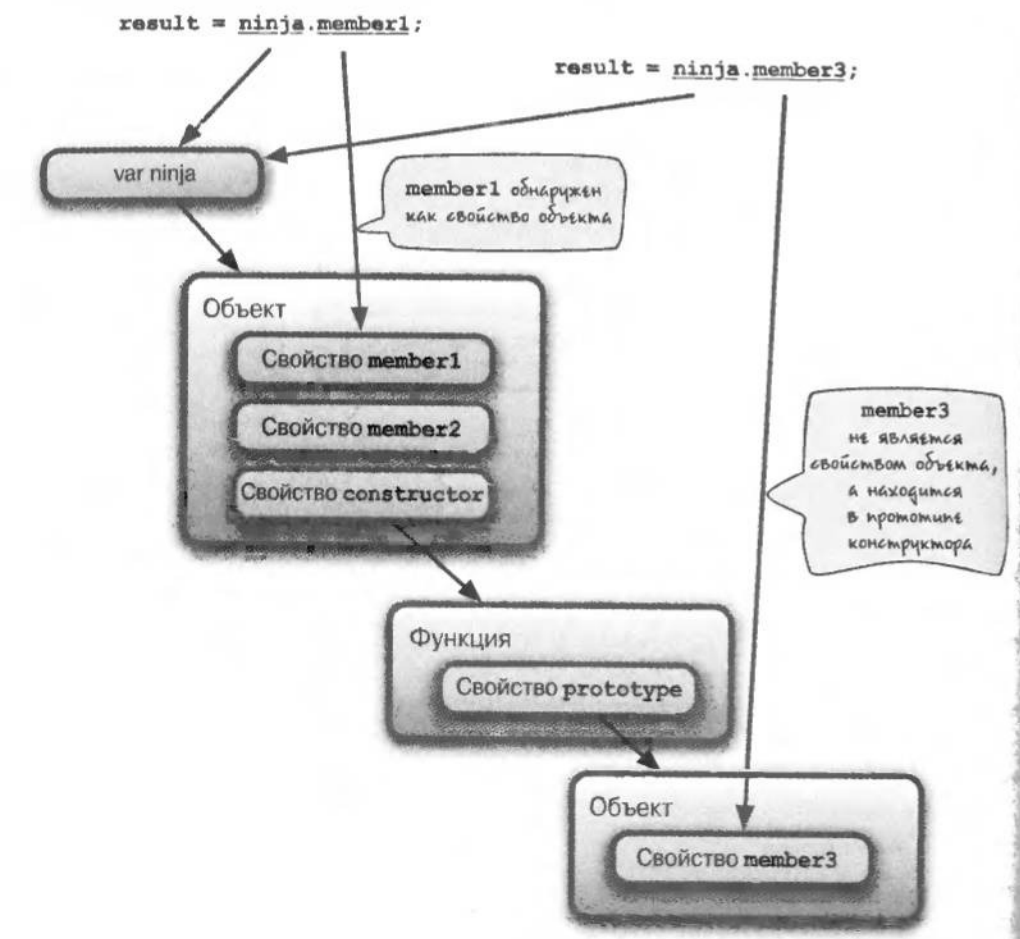

Рис. 6.5. Ссылки на свойства сначала обнаруживаются в самом объекте, и если они не найдены, то анализируется прототип конструктора

Листинг 6.4. Дополнительное наблюдение за изменениями в прототипе

```
<script type="text/javascript">

  function Ninja(){
    this.swung = true;
    this.swingSword = function(){
    return !this.swung;
    };
  }
```

❶ Определить метод экземпляра с тем же именем, что и у метода прототипа


```
  var ninja = new Ninja();

  Ninja.prototype.swingSword = function(){
    return this.swung;
  };

  assert(ninja.swingSword(),
        "Called the instance method, not the prototype method.");

</script>
```

❷ Определить прототипный метод с тем же именем, что и у метода экземпляра

❸ Проверить, какой из методов одержит верх

В данном примере снова вводится метод экземпляра ❶ с тем же именем, что и у прототипного метода ❷, как это уже делалось в коде примера из листинга 6.2. В том примере метод экземпляра получил приоритет над прототипным методом. Но теперь прототипный метод вводится после выполнения конструктора. Какой же из методов одержит верх на этот раз?

Как показывает тестирование ❸, даже если прототипный метод вводится *после* метода экземпляра, старшинство имеет метод экземпляра. И в этом есть свой смысл. Ведь обращение к прототипу происходит только в том случае, если ссылка на свойство в самом объекте не достигает цели. А поскольку свойство `swingSword` непосредственно принадлежит объекту, то прототипный вариант не вступает в действие, даже если он оказался самым последним вариантом созданного метода.

Дело в том, что ссылки на свойства разрешаются *сначала* в объекте, и по умолчанию обращение к прототипу осуществляется лишь в том случае, если эти ссылки не достигают цели. Итак, выяснив, каким образом экземпляры объектов получаются через функции-конструкторы, перейдем к более подробному рассмотрению характера самих объектов.

Типизация объектов через конструкторы

Безусловно, знать, каким образом прототип используется в JavaScript при увязывании ссылок на свойства, очень важно. Но не менее полезно знать, какая именно функция построила экземпляр объекта. Как было показано ранее, конструктор объекта доступен через свойство `constructor`. Сделать обратную ссылку на конструктор можно в любой момент, используя его даже в форме контроля типов, как показано в примере кода из листинга 6.5.

Листинг 6.5. Анализ типа экземпляра объекта и его конструктора

```
<script type="text/javascript">

  function Ninja(){}

  var ninja = new Ninja();

  assert(typeof ninja == "object",
        "The type of the instance is object.");

assert(ninja instanceof Ninja,
      "instanceof identifies the constructor." );
```

❶ Проверить тип `ninja`, используя оператор `typeof`. Это позволит лишь выяснить, что `ninja` — это объект, но не более того

❷ Проверить тип `ninja`, используя оператор `instanceof`. Это позволит также узнать, что объект построен конструктором типа `Ninja`

```
assert(ninja.constructor == Ninja,
      "The ninja object was created by the Ninja function.");

</script>
```

Проверить тип ninja, используя ссылку на конструктор. Это позволит получить конкретную ссылку на функцию-конструктор

В примере кода из листинга 6.5 сначала определяется конструктор, и с его помощью создается экземпляр объекта. Затем проверяется тип экземпляра этого объекта с помощью оператора `typeof` ❶. Это не особенно помогает, поскольку все экземпляры будут распознаваться как объекты, а в результате всегда возвращается значение `"object"`. Намного более интересный результат дает оператор `instanceof` ❷, который действительно помогает выяснить, был ли экземпляр объекта создан с помощью конкретной функции-конструктора.

Помимо этого, можно воспользоваться свойством `constructor`, которое, как известно, вводится во все экземпляры объектов, в качестве обратной ссылки на исходную функцию, создавшую экземпляр объекта. С помощью этого свойства можно проверить происхождение экземпляра объекта подобно тому, как это делается с помощью оператора `instanceof`. А поскольку это всего лишь обратная ссылка на исходный конструктор, то по ней можно получить новый экземпляр объекта типа `Ninja`, как показано в примере кода из листинга 6.6.

Листинг 6.6. Получение нового экземпляра объекта по ссылке на конструктор

```
<script type="text/javascript">

  function Ninja(){}

  var ninja = new Ninja();

  var ninja2 = new ninja.constructor();

  assert(ninja2 instanceof Ninja, "It's a Ninja!");

  assert(ninja !== ninja2, "But not the same Ninja!");

<script>
```

❶ Построить второй объект типа Ninja из первого

❷ Убедиться в том, что новый экземпляр объекта относится к типу Ninja

❸ Это не один и тот же объект, но два разных его экземпляра

В данном примере кода сначала определяется конструктор, и с его помощью создается экземпляр объекта. Затем свойство `constructor` вновь созданного экземпляра объекта используется для построения второго его экземпляра ❶. Как показывает тестирование ❷, несмотря на то, что построен второй экземпляр объекта типа `Ninja`, переменная уже не ссылается на тот же самый экземпляр ❸.

В примере кода из листинга 6.6 особое внимание обращает на себя тот факт, что экземпляр объекта можно получить, не обращаясь даже к исходной функции-конструктору. Ссылкой на этот конструктор можно пользоваться совершенно скрытно, даже если он уже не находится в текущей области действия.

Примечание

Свойство `constructor` объекта можно изменять, хотя очевидных или особых причин для этого не существует (разве что сделать это со злым умыслом). Ведь основное его назначение — уведомлять об источнике построения объекта.

Если же свойство `constructor` перезаписывается, его исходное значение просто теряется.

Все это, конечно, очень хорошо и даже полезно знать, но мы затронули лишь малую часть тех огромных потенциальных возможностей, которые предоставляют нам прототипы. И самое интересное нас еще ждет впереди.

Наследование и цепочка прототипов

Оператор `instanceof` можно также использовать с выгодой для себя, реализуя форму наследования объектов. Но для этого нужно ясно понимать, каким образом в JavaScript действует наследование и какая в этом роль принадлежит *цепочке прототипов*. Рассмотрим пример кода, приведенный в листинге 6.7, где предпринимается попытка ввести наследование в экземпляр объекта.

Листинг 6.7. Попытка добиться наследования с помощью прототипов

```
<script type="text/javascript">

  function Person(){}
  Person.prototype.dance = function(){};

  function Ninja(){}

  Ninja.prototype = { dance: Person.prototype.dance };

  var ninja = new Ninja();
  assert(ninja instanceof Ninja,
        "ninja receives functionality from the Ninja prototype");
  assert( ninja instanceof Person, "... and the Person prototype");
  assert( ninja instanceof Object, "... and the Object prototype");

</script>
```

❶ Определить объект танцующего человека (типа Person) через его конструктор и прототип

❷ Определить объект ниндзя типа Ninja

❸ Попытаться сделать ниндзя танцующим человеком, скопировав метод танца (dance) из прототипа Person

Прототип функции является обычным объектом, и поэтому существует несколько способов копирования его функциональных возможностей, в том числе свойств и методов, чтобы осуществить наследование. В приведенном выше примере кода сначала определяется объект типа `Person` ❶, а затем объект типа `Ninja` ❷. А поскольку объект типа `Ninja` явно определяет человека (в данном случае ниндзя), то можно сделать попытку добиться того, чтобы он унаследовал свойства объекта типа `Person`, определяющего человека вообще. И такая попытка предпринимается путем копирования свойства `dance` из метода прототипа `Person` в одноименное свойство прототипа `Ninja` ❸.

Как показывает тестирование рассматриваемого здесь кода (рис. 6.6), объект типа `Ninja` нельзя сделать объектом типа `Person`. И хотя ниндзя можно научить танцевать, как и всякого человека, объект типа `Ninja` все равно нельзя сделать объектом типа `Person`, скопировав соответствующее свойство. Ведь это не наследование, а только копирование.

Такой подход является крупной и старой ошибкой. Хотя и ущерба от него не особенно много, поскольку он предполагает лишь ручное копирование каждого свойства в отдельности из прототипа одного объекта (`Person`) в прототип другого объекта (`Ninja`).

Но это не имеет никакого отношения к наследованию. Поэтому продолжим исследование данного вопроса дальше

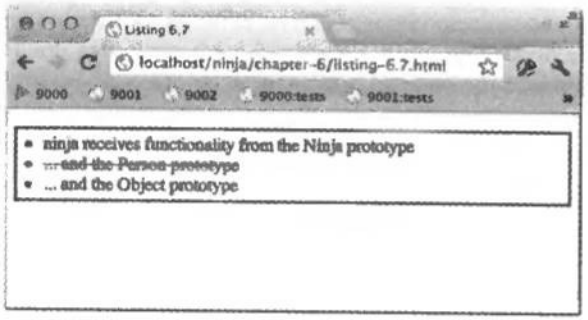

Рис. 6.6. Объект типа `Ninja` на самом деле не является объектом типа `Person`. Поэтому особой радости танцы не приносят!

Примечание

Любопытно отметить, что все объекты неявно являются экземплярами объекта типа `Object`. Для того чтобы убедиться в этом, попробуйте выполнить оператор `console.log({}.constructor)` в отладчике браузера.

На самом деле нам требуется образовать *цепочку прототипов*, чтобы определить ниндзя (объект тип `Ninja`) как человека (объект типа `Person`), человека (объект типа `Person`) — как млекопитающее (объект типа `Mammal`), а млекопитающее (объект типа `Mammal`) — как животное (объект типа `Animal`), и так далее до самого объекта типа `Object`. Такую цепочку прототипов лучше всего создать, используя экземпляр одного объекта в качестве прототипа другого объекта, как показано ниже.

```
SubClass.prototype = new SuperClass();
```

Например:

```
Ninja.prototype = new Person();
```

В этом случае цепочка прототипов сохраняется, поскольку прототип экземпляра класса `SubClass` будет экземпляром класса `SuperClass`, у которого имеется прототип со всеми свойствами объекта типа `SuperClass`, а у того, в свою очередь, — прототип, указывающий на экземпляр *его* суперкласса, и т.д. Попробуем применить такой способ создания цепочки прототипов, внеся незначительные изменения в пример кода из листинга 6.7, как выделено полужирным в листинге 6.8.

Листинг 6.8. Осуществление наследования с помощью прототипов

```
<script type="text/javascript">

  function Person(){}
  Person.prototype.dance = function(){};

  function Ninja(){}

  Ninja.prototype = new Person();
```

❶ Сделать объект типа `Ninja` объектом типа `Person`, превратив прототип `Ninja` в экземпляр объекта типа `Person`


```
  var ninja = new Ninja();
  assert(ninja instanceof Ninja,
        "ninja receives functionality from the Ninja prototype");
  assert(ninja instanceof Person, "... and the Person prototype");
  assert(ninja instanceof Object, "... and the Object prototype");
  assert(typeof ninja.dance == "function", "... and can dance!")

</script>
```

Единственное изменение, внесенное в код рассматриваемого здесь примера, состоит в использовании экземпляра объекта типа `Person` в качестве прототипа для объекта типа `Ninja` ❶. В итоге все тесты проходят успешно, как показано на рис. 6.7. Самые важные следствия из этого примера состоят в том, что, выполняя оператор `instanceof`, можно выяснить, наследует ли функция какие-нибудь функциональные возможности любого объекта по своей цепочке прототипов.

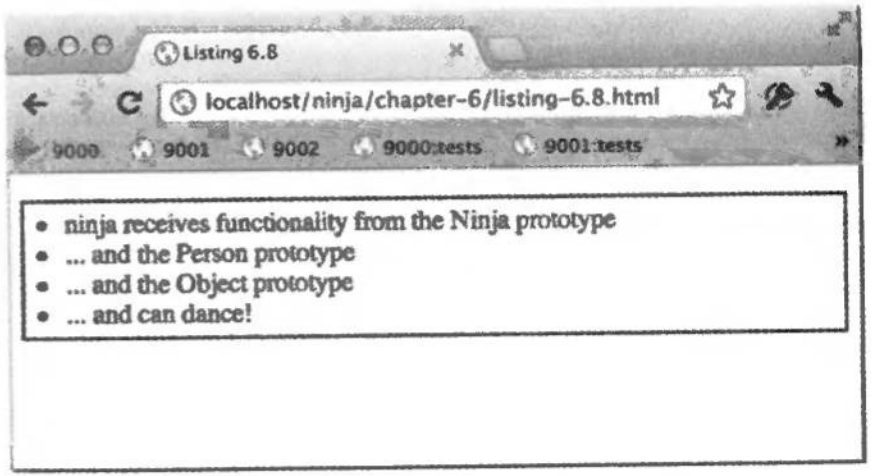

Рис. 6.7. Теперь объект типа `Ninja` стал объектом типа `Person`. На радостях можно и сплясать победный танец!

Примечание

Не рекомендуется пользоваться следующим способом: `Ninja.prototype = Person.prototype;`, т.е. применяя прототип `Person` непосредственно как прототип `Ninja`. Ведь в этом случае любые изменения в прототипе `Ninja` обусловят изменения в прототипе `Person`, поскольку они представляют один и тот же объект. А это может привести к нежелательным побочным эффектам.

Еще один побочный эффект от такого наследования прототипов состоит в том, что все наследуемые прототипы функций продолжают и далее активное обновление. На рис. 6.8 показано, каким образом цепочка прототипов применяется в рассматриваемом здесь примере. Следует особо подчеркнуть, что у данного объекта имеются также свойства, наследуемые от прототипа `Object`.

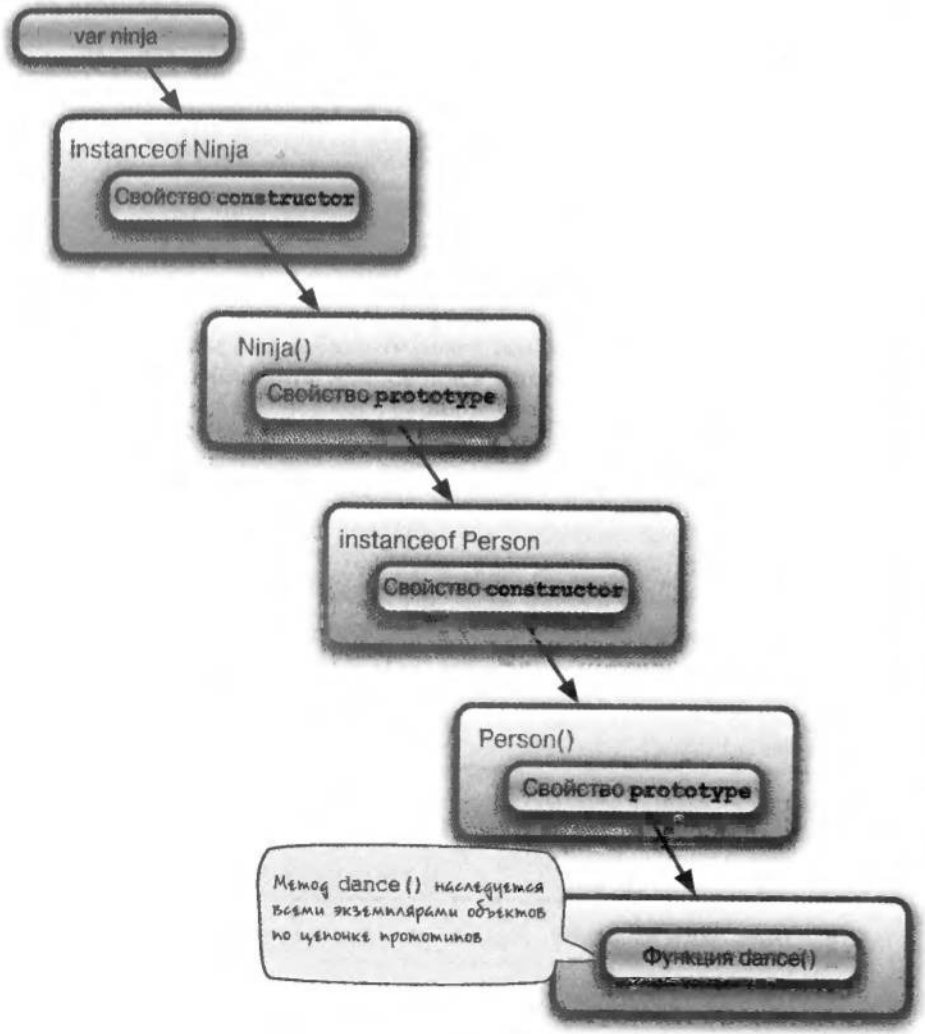

Рис. 6.8. Цепочка прототипов, по которой свойства увязываются для наследования объектом танцующего ниндзя

Следует заметить, что конструкторы всех объектов собственных классов JavaScript, включая `Object`, `Array`, `String`, `Number`, `RegExp` и `Function`, обладают свойствами прототипов, которые подлежат манипулированию и расширению. И в этом есть свой смысл, поскольку каждый из конструкторов упомянутых объектов сам является функцией. И в конечном итоге это дает невероятно эффективное средство для усовершенствования языка программирования. Ведь с его помощью можно существенно расширить функциональные возможности *самого* языка, внедрив новые или недостающие его элементы.

Примечание

Как и все самые передовые методики, рассмотренный здесь способ представляет собой палку о двух концах. Неплохая сводка по данному вопросу имеется в блоге *Perfection Kills* (Совершенствование убивает) по адресу `http://perfectionkills.com/extending-built-in-native-objects-evil-or-not/`.

Такое усовершенствование JavaScript как языка программирования может принести немалую пользу в одной из будущих версий. Например, в версии JavaScript 1.6 внедрен ряд очень удобных вспомогательных методов, в том числе и для обработки массивов. В одном из таких методов, `forEach()`, организуется циклическое обращение к свойствам массива с вызовом функции для каждого свойства, что может оказаться особенно удобным в тех случаях, когда требуется подключить другие функциональные возможности, не меняя общую структуру циклического обращения.

И хотя данный метод уже появился в большинстве современных браузеров, он пока еще недоступен во всех наиболее употребительных браузерах, где для этого, возможно, потребуется соответствующая поддержка. Если же реализовать подобные функциональные возможности в старых браузерах, то можно уже не опасаться за работоспособность остальной части прикладного кода. В листинге 6.9 приведена одна из возможных реализаций метода `forEach()`, которой можно было бы воспользоваться для восполнения пробелов, существующих в старых браузерах.

Листинг 6.9. Реализация метода `forEach()` в версии JavaScript 1.6 с перспективой на будущее

```
<script type="text/javascript">

  if (!Array.prototype.forEach) {

    Array.prototype.forEach = function(callback, context) {
      for (var i = 0; i < this.length; i++) {
        callback.call(context || null, this[i], i, this);
      }
    };
  }

  ["a", "b", "c"].forEach(function(value, index, array) {
    assert(value,
           "Is in position " + index + " out of " +
          (array.length - 1));
  });

</script>
```

❶ Проверить, существует ли уже метод. Его не придется переопределять в тех браузерах, где он предоставляется автоматически

❷ Ввести метод в прототип класса `Array`. После этого метод становится доступным для всех массивов

❸ Воспользоваться функцией обратного вызова для обращения к каждому элементу массива

Проверить данную реализацию

Прежде чем использовать данную реализацию метода `forEach()`, следует проверить, определен ли он уже в классе `Array` ❶, и если он определен, то пропустить всю остальную часть кода. Благодаря этому код данного метода становится совместимым на перспективу. Ведь если он выполняется в той среде, где метод `forEach()` определен, то предпочтение отдается собственному методу.

Если же окажется, что метод `forEach()` не существует, он вводится в прототип класса `Array` ❷ простым перебором массива в традиционном цикле `for` и обратным вызовом метода `call()` для каждого элемента массива ❸. При обратном вызове этому методу передаются следующие значения: элемент массива, индекс и исходный массив. Следует иметь в виду, что выражение `context || null` предотвращает передачу возможного значения `undefined` методу `call()`.

Все встроенные объекты, в том числе и типа `Array`, содержат прототипы, поэтому у нас имеется возможность расширять язык JavaScript по мере необходимости. Не следует, однако, забывать, что внедрение новых средств и методов в собственные методы чревато теми же опасностями, что и введение новых переменных в глобальную область действия. Ведь на самом деле существует лишь единственный экземпляр собственного объекта, а следовательно, имеется большая вероятность возникновения конфликтов в присваивании имен.

Следует также иметь в виду, что реализация языковых средств для собственных прототипов на перспективу, как в приведенном выше примере метода `forEach()`, может и не совпасть с их окончательной реализацией в новой версии языка программирования, приводя к осложнениям, когда функциональные возможности новой версии будут поддерживаться в самом браузере. Поэтому браться за это дело нужно, лишь тщательно взвесив все аргументы за и против.

Итак, мы показали, что прототипами можно пользоваться для расширения функциональных возможностей собственных объектов в JavaScript. А теперь обратим свое внимание на модель DOM.

Прототипы HTML-разметки элементов модели DOM

В современных браузерах Internet Explorer 9, Firefox, Safari и Opera все элементы модели DOM наследуют от конструктора класса `HTMLElement`. Благодаря доступности прототипа класса `HTMLElement` в браузерах появляется возможность для расширения любого выбранного узла HTML-разметки, как показано в примере кода из листинга 6.10.

Листинг 6.10. Добавление нового метода ко всем элементам HTML-разметки с помощью прототипа класса `HTMLElement`

```
<div id="parent">
  <div id="a">I'm going to be removed.</div>
  <div id="b">Me too!</div>
</div>

<script type="text/javascript">
  HTMLElement.prototype.remove = function() {
    if (this.parentNode)
      this.parentNode.removeChild(this);
  };

  var a = document.getElementById("a");
  a.parentNode.removeChild(a);
  document.getElementById("b").remove();

  assert(!document.getElementById("a"),"a is gone.");
  assert(!document.getElementById("b"),"b is gone too.");
</script>
```

❶ Ввести новый метод во все элементы, добавив его к прототипу HTMLElement

❷ Реализовать это в коде по-старому

❸ Реализовать это в коде по-новому

Сначала в приведенном выше примере кода новый метод `remove()` вводится во все элементы модели DOM путем расширения прототипа конструктора базового класса `HTMLElement` ❶. Затем для сравнения элемент `a` удаляется собственными средствами ❷, а элемент `b` — с помощью нового метода ❸. В обоих случаях утверждается, что элементы удаляются из модели DOM.

Совет

Подробнее о данной конкретной возможности можно узнать из спецификации HTML 5 по следующему адресу:

`http://www.whatwg.org/specs/web-apps/current-work/multipage/section-elements.html`

К числу тех библиотек, где такая возможность интенсивно используется, относится Prototype. В этой библиотеке существующие элементы модели DOM дополняются всеми видами функциональных возможностей, включая среди прочего вставку HTML-разметки и манипулирование стилевым оформлением по таблицам CSS. Но самое главное, что, работая с подобными прототипами, не следует рассчитывать на их присутствие в версиях браузера Internet Explorer, предшествовавших версии 8. Если же Internet Explorer не является целевой платформой, то они могут сослужить неплохую службу.

В связи с применением прототипов для HTML-разметки нередко возникает еще один противоречивый вопрос: можно ли получать экземпляры элементов HTML-разметки непосредственно из их функции-конструктора, как в следующем случае:

```
var elem = new HTMLElement();
```

Но такой код вообще не работоспособен. Ведь даже если браузеры и раскроют конструктор и прототип корневого класса, они могут выборочно исключить всякую возможность вызывать конструктор (по-видимому, для того чтобы ограничить создание элемента только для внутреннего употребления).

Если бы не скрытые препятствия, которые стоят на пути внедрения упомянутых выше средств введения методов в элементы модели DOM с точки зрения совместимости со старыми браузерами, то преимущества получения чистого кода оказались бы весьма значительными и заслуживали бы внимательного исследования в подходящих ситуациях.

Примечание

У рассмотренного выше способа имеются свои противники. Они вполне обоснованно считают, что видоизменение отдельных элементов модели DOM носит слишком навязчивый характер и способно внести нестабильность в работу веб-страницы, поскольку изменения, вносимые в подобные элементы, могут невольно помешать нормальному функционированию других, ничего не подозревающих об этом компонентов страницы. Поэтому если вы выберете этот способ, то внедрять его следует весьма осмотрительно. Методы обычно вводятся довольно безболезненно, но изменять порядок выполнения уже имеющегося кода нужно очень аккуратно.

А теперь перейдем непосредственно к обсуждению скрытых препятствий, которых следует избегать, обращаясь с прототипами.

Скрытые препятствия

Как и во всем остальном, в JavaScript имеется целый ряд скрытых препятствий, связанных с прототипами, получением экземпляров объектов и наследованием. Одни из этих препятствий можно обойти, но большинство из них требуют от разработчика умерить свой пыл и трезво оценить ситуацию. Итак, рассмотрим некоторые из подобных препятствий.

Расширение прототипа класса `Object`

Вероятно, самой грубой ошибкой в обращении с прототипами было бы расширение собственного прототипа `Object.prototype`. Препятствие в данном случае состоит в том, что при расширении этого прототипа *все* объекты получают соответствующие дополнительные свойства. Это особенно досадно, поскольку при повторении уже существующих свойств объекта появляются новые, вызывая всевозможные виды неожиданного поведения.

Проиллюстрируем это на конкретном примере из листинга 6.11. Допустим, требуется сделать что-нибудь безобидное на первый взгляд, например, ввести в класс `Object` метод `keys()`, который должен возвращать массив всех имен (ключей) свойств, имеющихся у объекта.

Листинг 6.11. Неожиданное поведение, к которому приводит добавление дополнительных свойств в прототип класса `Object`

```
<script type="text/javascript">

  Object.prototype.keys = function() {
    var keys = [];
    for (var p in this) keys.push(p);
    return keys;
 };

  var obj = { a: 1, b: 2, c: 3 };

  assert(obj.keys().length == 3,
        "There are three properties in this object.");

</script>
```

❶ Определить новый метод в прототипе класса Object

❷ Создать объект, который должен служить предметом тестирования

❸ Протестировать новый метод, проверив длину массива

Сначала в приведенном выше примере кода определяется новый метод ❶, в котором просто перебираются свойства и накапливаются ключи в возвращаемом массиве. Затем определяется предмет тестирования из трех свойств ❷ и проверяется, что в результате должен быть получен массив из трех элементов ❸. Но, как показано на рис. 6.9, тест не проходит.

Непрохождение теста объясняется тем, что добавлением метода `keys()` в класс `Object` на самом деле было введено *еще одно* свойство, которое появится во всех объектах, что и не было учтено при составлении теста. Это свойство оказывает влияние на все объекты, и поэтому оно должно быть непременно учтено как дополнительное в любом коде. Но это может нарушить нормальную работу кода, основанного на вполне благоразумных исходных представлениях авторов веб-страниц. Очевидно, что подобная ситуация неприемлема, и поэтому ее следует всячески избегать! Но ведь это может сделать кто-нибудь *другой*, внеся ошибку в код. Что же можно с этим поделать? Оказывается, данное препятствие можно обойти, чтобы защитить свой код от вмешательства несведущих посторонних – пускай даже и с добрыми намерениями.

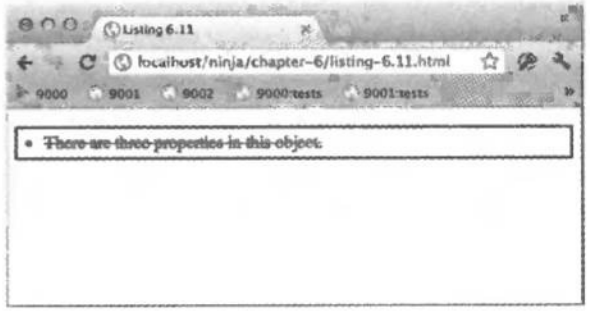

Рис. 6.9. Похоже, что нарушено основополагающее исходное представление об объектах!

В JavaScript предоставляется метод `hasOwnProperty()`, с помощью которого можно выявить, определены ли свойства в экземпляре объекта или они импортированы из прототипа. Рассмотрим применение этого метода в примере кода из листинга 6.12, где внесенные изменения выделены полужирным по сравнению с примером кода из листинга 6.11.

Листинг 6.12. Укрощение строптивых расширений прототипа класса `Object` с помощью метода `hasOwnProperty()`

```
<script type="text/javascript">

  Object.prototype.keys = function() {
    var keys = [];
    for (var i in this)
      if (this.hasOwnProperty(i)) keys.push(i);
    return keys;
  };

  var obj = { a: 1, b: 2, c: 3 };

  assert(obj.keys().length == 3,
        "There are three properties in this object.");

</script>
```

❶ Игнорировать свойства из прототипа, пропустив их с помощью метода hasOwnProperty()

❷ Проверить метод, подсчитав элементы массива

В переопределенном методе из приведенного выше примера кода теперь игнорируются свойства, не определенные в экземпляре объекта ❶. Поэтому на этот раз тест проходит ❷.

Но если рассматриваемое здесь препятствие можно обойти, то это совсем не означает, что таким обходным приемом следует злоупотреблять, что может стать лишней обузой для пользователей кода. Циклическое обращение к свойствам объекта весьма распространено в практике разработки приложений на JavaScript, и поэтому разработ-

чики нередко пользуются методом `hasOwnProperty()` в своем коде, хотя многие авторы веб-страниц даже не подозревают о его существовании.

Как правило, следует пользоваться подобными обходными приемами, чтобы защитить свой код от возможных нарушений. Но не стоит полагаться на то, что и другие разработчики будут аналогичным образом защищать свой код. А теперь рассмотрим еще одно скрытое препятствие, на которое можно невольно натолкнуться, обращаясь с прототипами.

Расширение прототипа класса `Number`

Большинство собственных прототипов, кроме класса `Object`, как пояснялось в предыдущем разделе, можно благополучно расширять. Но еще одним проблематичным в этом отношении является прототип класса `Number`. Результат может оказаться довольно запутанным, в зависимости от того, как числа и свойства чисел интерпретируются механизмом JavaScript, что и демонстрирует пример кода из листинга 6.13.

Листинг 6.13. Добавление метода в прототип класса `Number`

```
<script type="text/javascript">

  Number.prototype.add = function(num){
    return this + num;
  };

  var n = 5;
  assert(n.add(3) == 8,
        "It works when the number is in a variable.");

  assert((5).add(3) == 8,
        "Also works if a number is wrapped in parentheses.");

  assert(5.add(3) == 8, "What about a simple literal?");

</script>
```

❶ Определить новый метод для прототипа Number

❷ Проверить метод, используя переменную

❸ Проверить метод, используя формат выражения

❹ Проверить метод, используя формат литерала. Должны ли пройти все эти тесты?

Сначала в приведенном выше примере кода определяется новый метод `add()` для класса `Number`, который выбирает значение своего аргумента, складывает его с числовым значением и возвращает результат ❶. Затем новый метод проверяется в различных форматах числа:

- в переменной ❷;
- в выражении ❸;
- непосредственно в числовом литерале ❹.

Но если попытаться загрузить веб-страницу с рассматриваемым здесь кодом сценария в окно браузера, то она вообще не загрузится, как показано на рис. 6.10. Оказывается, что синтаксический анализатор не в состоянии обработать контрольный пример с литералом.

Преодолеть рассматриваемое здесь препятствие будет нелегко, поскольку логика его появления может оказаться довольно запутанной. Ведь до сих пор существуют библиотеки, в том числе Prototype, в которые включаются функциональные возможности

прототипа класса `Number` и в которых просто оговариваются условия их применения. Разумеется, данное препятствие можно обойти при наличии тщательно выверенной документации на библиотеку и прилагаемого к ней учебного материала.

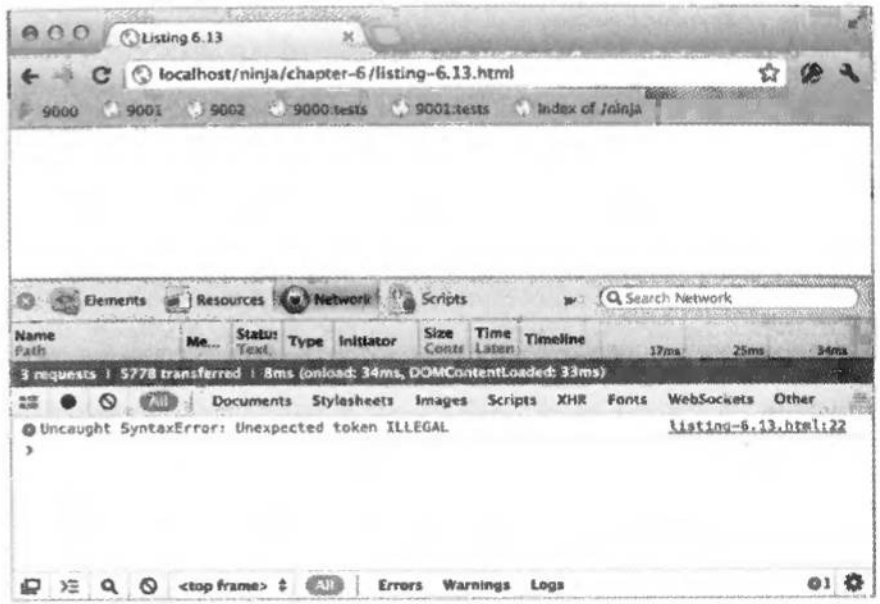

Рис. 6.10. Если тесты вообще не загружаются, значит, имеется серьезное препятствие

Как правило, с прототипом класса `Number` лучше не возиться, если в этом нет особой необходимости. А теперь рассмотрим ряд вопросов, которые могут возникнуть при подклассификации, а не расширении объектов собственных классов.

Подклассификация объектов собственных классов

Еще одно скрытое препятствие возникает при подклассификации объектов собственных классов. Единственным классом, который довольно просто разделяется на подклассы, является класс `Object`, поскольку это корень, с которого начинается вся цепочка прототипов. Но как только возникает потребность в подклассификации объектов других собственных классов, ситуация сразу же становится не столь ясной. Так, при подклассификации объектов типа `Array` все, казалось бы, происходит именно так, как и следовало ожидать, но рассмотрим этот случай на конкретном примере кода из листинг 6.14.

Листинг 6.14. Подклассификация объекта типа `Array`

```
<script type="text/javascript">

  function MyArray() {}

  MyArray.prototype = new Array();

  var mine = new MyArray();
```

```
  mine.push(1, 2, 3);

assert(mine.length == 3,
       "All the items are in our sub-classed array.");
  assert(mine instanceof Array,
        "Verify that we implement Array functionality.");

</script>
```

В данном примере кода подклассификация объекта типа `Array` осуществляется с помощью создаваемого нового конструктора `MyArray()`, и все действует замечательно и прекрасно до тех пор, пока не будет предпринята попытка загрузить веб-страницу с кодом данного сценария в окно браузера Internet Explorer. Оказывается, что свойство `length`, тесно связанное с числовыми индексами массивов в объекте типа `Array`, настолько специфично, что соответствующая реализация функциональных возможностей такого объекта для Internet Explorer недостаточно хорошо реагирует на операции с этим свойством.

Примечание

Подробнее этот вопрос освещается в связи со стандартом ECMAScript 5 в блоге *Perfection Kills* по адресу `http://perfectionkills.com/how-ecmascript-5-still-does-not-allow-to-subclass-an-array/`.

В подобных случаях более правильный подход состоит в том, чтобы реализовать функциональные возможности объектов собственных классов по частям вместо того, чтобы подклассифицировать их полностью. Рассмотрим такой подход на примере кода из листинга 6.15.

Листинг 6.15. Имитация функциональных возможностей объекта типа `Array` без подлинной подклассификации

```
<script type="text/javascript">

  function MyArray() {}
  MyArray.prototype.length = 0;

  (function() {
    var methods = ['push', 'pop', 'shift', 'unshift',
      'slice', 'splice', 'join'];

    for (var i = 0; i < methods.length; i++) (function(name) {
      MyArray.prototype[ name ] = function() {
        return Array.prototype[ name ].apply(this, arguments);
      };

    })(methods[i]);
  })();

  var mine = new MyArray();
  mine.push(1, 2, 3);
  assert(mine.length == 3,
        "All the items are on our sub-classed array.");
```

❶ Определить новый "класс" с прототипным свойством length

❷ Скопировать избранные функциональные возможности массива

Проверить новый "класс"


```
  assert(!(mine instanceof Array),
        "We aren't subclassing Array, though.");

</script>
```

Сначала в приведенном выше примере кода определяется конструктор для нового класса `MyArray`, который наделяется собственным свойством `length` ❶. Затем используется функция немедленного вызова для ввода методов, избранных из класса `Array`, в новый класс с помощью метода `apply()`, рассматривавшегося в главе 4, вместо того чтобы пытаться подклассифицировать класс `Array`, что, как пояснялось выше, подходит не для всех браузеров. Обратите внимание на то, что массив с именами методов обеспечивает необходимый порядок действий и простоту расширения.

Самостоятельно приходится реализовывать только свойство `length`, поскольку оно должно оставаться изменяемым, а такая возможность в браузере Internet Explorer не поддерживается. Рассмотрим далее возможные пути разрешения типичных затруднений, которые могут возникнуть у тех, кто попытается воспользоваться разработанным прикладным кодом.

Препятствия при получении экземпляров объектов

Как отмечалось ранее, функции могут выполнять двоякую роль, служа в качестве обычных функций и конструкторов. Вследствие этого пользователям разработанного прикладного кода не всегда понятно конкретное назначение функций. Обратимся сначала к простому примеру кода, приведенного в листинге 6.16, чтобы выяснить, к чему может привести неверное толкование роли функций.

Листинг 6.16. Результат исключения оператора `new` из вызова функции

```
<script type="text/javascript">

  function User(first, last){
    this.name = first + " " + last;
  }

    var user = User("Ichigo", "Kurosaki");

  assert(user, "User instantiated");

  assert(user.name == "Ichigo Kurosaki",
        "User name correctly assigned");

</script>
```

❶ Определить класс User со свойством name

❷ Создать тестовый объект пользователя, передав образец его имени

❸ Проверить, был ли получен экземпляр тестового объекта

❹ Проверить, правильно ли конструктор присвоил имя пользователя

Сначала в приведенном выше примере кода определяется класс `User` ❶, конструктор которого принимает в качестве аргументов имя и фамилию пользователя, сцепляя их в полное имя, которое сохраняется в свойстве `name`. На самом деле этот "класс" нельзя считать таким же настоящим, как и тот, что определяется в других языках объектно-ориентированного программирования, но именно так его принято называть, и поэтому мы не будем отступать от данного правила. Затем создается экземпляр объекта данного

класса, сохраняемый в переменной `user` ❷, а далее проверяется, был ли получен экземпляр этого объекта ❸ и насколько правильно действовал при этом конструктор ❹.

Но если попытаться выполнить рассматриваемый здесь код, то результат получится совершенно неверным, как показано на рис. 6.11. Оказывается, первый тест не проходит. А это означает, что экземпляр объекта вообще не был получен, и поэтому при выполнении второго теста возникает серьезная ошибка.

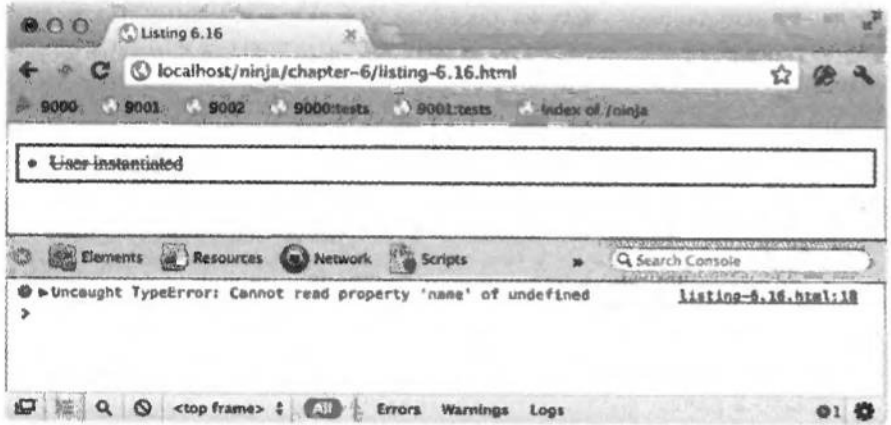

Рис. 6.11. Экземпляр объекта не был вообще получен

Беглого взгляда на рассматриваемый здесь код совершенно недостаточно, чтобы понять, что функция `User()` предназначена для вызова с помощью оператора `new` или что на это не было явно указано в коде по недосмотру или оплошности. Но в любом случае отсутствие оператора `new` привело к тому, что функция была вызвана как обычно, а не как конструктор, т.е. без получения нового экземпляра объекта. Неопытный пользователь может легко попасться на этом, пытаясь вызвать функцию без оператора `new`, что приведет к совершенно неожиданным результатам (например, значение переменной `user` может оказаться неопределенным).

Примечание

Вы, вероятно, обратили уже внимание на то, что с самого начала в примерах кода, приведенных в этой книге, имена одних функций начинаются со строчной буквы, а других — с прописной. Как отмечалось в предыдущих главах, это повсеместно принятые условные обозначения, где имена функций в роли конструкторов начинаются в прописной буквы, а имена функций в роли обычных функций — со строчной. Более того, имена конструкторов, как правило, обозначаются существительными, указывая на класс, который они конструируют (`Ninja` — ниндзя, `Samurai` — самурай, `Tachikoma` — тачикома (колесно-шагающий робот-танкетка) и т.д.), тогда как имена обычных функций обозначаются глаголами или парами глаголов и существительных, указывающих на то, что они делают (`throwShuriken` — бросить шурикен (метательное оружие скрытого действия), `swingSword` — взмахнуть мечом, `hideBehindAPlant` — скрыться за растением).

Помимо неожиданных результатов, к которым приводит вызов функции в качестве конструктора, для чего она совершенно не предназначена, возможны также едва заметные побочные эффекты, в том числе засорение текущей области действия (зачастую глобального пространства имен). А это, в свою очередь, может привести к еще более неожиданным результатам. В качестве примера рассмотрим код, приведенный в листинге 6.17.

Листинг 6.17. Неожиданное внедрение переменной в глобальное пространство имен

```
<script type="text/javascript">

  function User(first, last){
    this.name = first + " " + last;
  }

  var name = "Rukia";

  var user = User("Ichigo", "Kurosaki");

  assert(name == "Rukia",
         "Name was set to Rukia.");

</script>
```

❶ Создать глобальную переменную

❷ Снова вызвать конструктор неправильно

❸ Проверить глобальную переменную

Приведенный выше код похож на код из предыдущего примера, за исключением того, что на этот раз глобальная переменная `name` объявляется в глобальном пространстве имен ❶. Как и в предыдущем примере, в этом коде возникает та же самая ошибка из-за отсутствия оператора `new` по недосмотру или оплошности ❷. Но на этот раз отсутствует и тест, выявляющий эту ошибку. Вместо этого выполняется тест, выявляющий, что значение глобальной переменной `name` перезаписано ❸. Именно поэтому он и не проходит!

Для того чтобы выяснить причину подобной ошибки, проанализируем код конструктора. Когда конструктор вызывается как таковой, контекстом вызываемой функции становится вновь размещаемый объект. Но каким будет этот контекст, если вызвать функцию как обычно? Как упоминалось в главе 3, контекстом функции в этом случае становится глобальная область действия, а это означает, что ссылка `this.name` делается не на свойство `name` вновь размещаемого объекта, а на переменную `name`, находящуюся в глобальной области действия.

Отладка такого кода превращается в сущий кошмар. Разработчик может вновь обратиться к переменной `name`, даже не осознавая того, что произошла ошибка из-за неверного применения функции `User()`. И в этом случае ему придется иметь дело с весьма неприятным и неопределенным изъяном в коде: бесконтрольным изменением значений переменных.

Но настоящим мастерам программирования на JavaScript совсем не безразличны нужды конечных пользователей, поэтому попробуем найти выход из создавшегося затруднительного положения. В частности, нам нужно найти какой-нибудь способ, чтобы выяснить, вызывается ли неправильно функция, которую предполагается использовать в качестве конструктора. С этой целью обратимся к примеру кода, приведенного в листинге 6.18.

Листинг 6.18. Выяснение, вызывается ли функция как конструктор

```
<script type="text/javascript">

  function Test() {
    return this instanceof arguments.callee;
  }

  assert(!Test(), "We didn't instantiate, so it returns false.");
  assert(new Test(), "We did instantiate, returning true.");

</script>
```

Напомним ряд положений.

- Обратиться к функции, выполняющейся в настоящий момент, можно по ссылке `arguments.callee`, как пояснялось в главе 4.
- Контекстом обычной функции становится глобальная область действия, если только кто-нибудь не изменит это положение своими действиями.
- Оператор `instanceof`, применяемый к построенному объекту, позволяет выявить его конструктор.

Принимая во внимание эти положения, мы можем прийти к выводу, что результатом вычисления следующего выражения:

```
this instanceof arguments.callee
```

оказывается логическое значение `true`, если это выражение вычисляется в теле конструктора, а иначе — логическое значение `false`, если оно вычисляется в теле обычной функции.

Это означает, что в теле функции, которую предполагается вызывать как конструктор, можно проверить, вызывается ли она кем-нибудь без оператора `new`. Замечательно, но что с этим делать дальше? Конечно, можно было бы выдать пользователю сообщение об ошибке, посоветовав ему поступить в следующий раз правильно. Но настоящий мастер должен попытаться найти более изящный выход из этого затруднительного положения. Попробуем устранить ошибку сами, не адресуя ее пользователю, но внеся соответствующие изменения в код конструктора класса `User`, как выделено полужирным в листинге 6.19.

Листинг 16.19. Устранение ошибки вместо пользователя, вызывающего функцию

```
<script type="text/javascript">

  function User(first, last) {
    if (!(this instanceof arguments.callee)) {
      return new User(first,last);
    }
    this.name = first + " " + last;
  }

var name = "Rukia";

var user = User("Ichigo", "Kurosaki");
```

❶ Устранить ошибку, если окажется, что функция вызвана неверно, вызвав ее правильно

❷ Вызвать конструктор правильно


```
assert(name == "Rukia","Name was set to Rukia.");
assert(user instanceof User, "User instantiated");
assert(user.name == "Ichigo Kurosaki",
      "User name correctly assigned");

</script>
```

❸ Проверить правильность исправлений

В приведенном выше примере получается экземпляр объекта типа `User` ❶, который возвращается как результат выполнения функции. Для этого используется выражение, составленное в примере кода из листинга 6.18 с целью выяснить, вызвал ли пользователь функцию правильно. В итоге экземпляр объекта типа `User` получается независимо от того, вызывает ли пользователь функцию как обычно ❷ или как-то иначе, что и проверяется в последующих тестах ❸. Таким образом, прикладной код становится удобным в использовании! Но прежде чем похвалить себя за находчивость, поразмыслим над тем, насколько такой подход можно вообще считать правильным, принимая во внимание следующие соображения.

- Как отмечалось в главе 4, свойство `callee` станет не рекомендованным к применению в последующих версиях JavaScript и вообще запрещенным в строгом режиме. Поэтому рассмотренный выше обходной прием можно будет применять только в тех средах, где строгий режим вообще не предполагается соблюдать. В связи с этим возникает следующий вопрос: имеются ли какие-нибудь причины не соблюдать строгий режим?
- Является ли такой прием действительно передовой методикой программирования? Конечно, это изящный пример, но его доброкачественность можно оспорить.
- Можно ли с полной уверенностью сказать, что намерения пользователя хорошо известны? Не действуем ли мы слишком самоуверенно?

Все эти соображения должен непременно принимать во внимание настоящий мастер своего дела. Не следует забывать, что остроумный способ, позволяющий сделать что-нибудь, придумать можно, но не всегда *нужно*. Итак, рассмотрев скрытые препятствия, которые таит в себе обращение с прототипами, перейдем к обсуждению возможностей использовать вновь раскрытый потенциал прототипов для написания кода, в большей степени похожего на класс.

Код, похожий на класс

То, что JavaScript позволяет организовать наследование в определенной форме с помощью прототипов, само по себе замечательно, но общим желанием многих разработчиков, особенно с опытом программирования в классическом объектно-ориентированном стиле, является упрощение или абстракция системы наследования в JavaScript до более знакомого им уровня. А это неизбежно приводит к такому понятию, как классы. Но в JavaScript классическая поддержка наследования не реализована.

Как правило, разработчики жаждут следующего.

- Системы, упрощающей синтаксис построения новых функций-конструкторов и прототипов.
- Простого способа наследования прототипов.
- Удобного доступа к методам, переопределяемым прототипом функции.

В целом ряде уже существующих библиотек JavaScript имитируется классическая форма наследования, и в этом отношении в лучшую сторону отличаются две библиотеки: base2 и Prototype. Несмотря на то что каждая из этих библиотек содержит целый ряд развитых средств, объектно-ориентированное ядро составляет очень важную их часть. Попробуем разобраться в том синтаксисе, который они предлагают, чтобы с его помощью сделать код в большей степени похожим на классический объектно-ориентированный. В листинге 6.20 приведен пример синтаксиса, позволяющего достичь упомянутых выше целей.

Листинг 6.20. Пример синтаксиса, осуществляющего наследование в стиле, похожем на классический объектно-ориентированный

```
<script type="text/javascript">

  var Person = Object.subClass({
    init: function(isDancing) {
      this.dancing = isDancing;
    },

    dance: function() {
      return this.dancing;
    }
  });

  var Ninja = Person.subClass({
    init: function() {
      this._super(false);
    },
    dance: function() {
      // здесь следует конкретный код для класса Ninja
      return this._super();
    },
    swingSword: function() {
      return true;
    }
  });

  var person = new Person(true);
  assert(person.dance(),
        "The person is dancing.");

  var ninja = new Ninja();
  assert(ninja.swingSword(),
        "The sword is swinging.");
  assert(!ninja.dance(),
        "The ninja is not dancing.");

  assert(person instanceof Person,
        "Person is a Person.");
  assert(ninja instanceof Ninja &&
        ninja instanceof Person,
        "Ninja is a Ninja and a Person.");
</script>
```

❶ Создать подкласс Person как производный от класса Object, используя метод subclass(), который будет реализован далее

❷ Создать подкласс Ninja как производный от класса Person

❸ Вызвать конструктор суперкласса

Проверить класс Person, создав экземпляр его объекта и выяснив, содержит ли он метод танца

Проверить класс Ninja, создав экземпляр его объекта и выяснив, содержит ли он метод взмаха мечом и унаследованный метод танца

Проверить иерархию классов с помощью операторов instanceof

К приведенному выше примеру кода уместно сделать следующие примечания.

- Создание нового класса осуществляется путем вызова метода `subClass()` из функции-конструктора, существующей для суперкласса. В данном примере класс `Person` создается как производный от класса `Object` ❶, а класс `Ninja` — от класса `Person` ❷.
- Создание конструктора должно быть простым. С этой целью в предлагаемом синтаксисе для каждого класса (в данном случае `Person` и `Ninja`) предоставляется метод `init()`.
- Все созданные классы в конечном итоге наследуют от единого родительского класса `Object`. Так, если требуется создать совершенно новый класс, он должен стать подклассом, производным от класса `Object`, или же наследующим от него классом по иерархии классов, полностью имитирующей текущую систему прототипов.
- Самое сложное в рассматриваемом здесь синтаксисе — разрешить доступ к переопределяемым методам, правильно установив их контекст. В данном случае это делается с помощью метода `this._super()`, вызывающего исходные методы `init()` и `dance()` из суперкласса `Person` ❸.

Предложить синтаксис для того, чтобы осуществить конкретную схему наследования, несложно. Намного труднее его реализовать. В коде из листинга 6.21 понятие классов реализуется в виде структуры, сохраняя простое наследование и позволяя вызывать суперметод. Следует, однако, иметь в виду, что это совсем не простой код. Но если вы стремитесь стать настоящим мастером программирования на JavaScript, то, не теряя присутствия духа, уделите достаточно времени, чтобы разобраться в этом коде основательно.

Для того чтобы немного упростить вашу задачу, код, реализующий наследование, полностью представлен в листинге 6.21, чтобы было лучше видно, каким образом все его части сходятся вместе. А в последующих подразделах этот код подробно рассматривается по частям.

Листинг 6.21. Метод подклассификации

```
(function() {
  var initializing = false,
    superPattern =
      /xyz/.test(function() { xyz; }) ? /\b_super\b/ : /.*/;

  Object.subClass = function(properties) {
    var _super = this.prototype;

    initializing = true;
    var proto = new this();
    initializing = false;

    for (var name in properties) {

      proto[name] = typeof properties[name] == "function" &&
                    typeof _super[name] == "function" &&
                    superPattern.test(properties[name]) ?
```

Это замысловатое регулярное выражение определяет, возможна ли сериализация функций. Далее поясняется, что это означает ❶

❷ Ввести метод subClass() в класс Object

❸ Получить экземпляр суперкласса

❹ Скопировать свойства в прототип

```
        (function(name, fn) {
          return function() {
            var tmp = this._super;

            this._super = _super[name];

            var ret = fn.apply(this, arguments);
            this._super = tmp;

            return ret;
          };
        })(name, properties[name]) :
        properties[name];
  }

  function Class() {
    // все конструирование фактически выполняется в методе init()
    if (!initializing && this.init)
      this.init.apply(this, arguments);
    }

    Class.prototype = proto;

    Class.constructor = Class;

    Class.subClass = arguments.callee;

    return Class;
  };
})();
```

❺ Определить переопределяемую функции

Создать фиктивный конструктор класса

Заполнить прототип класса

Переопределить ссылку на конструктор

Сделать класс расширяемым

К двум самым важным частям данного примера реализации наследования относятся метод инициализации и суперметод. Имея ясное представление о тех целях, которые достигаются в этих частях рассматриваемого здесь кода, можно лучше понять, каким образом наследование реализуется в целом. Но было бы опрометчиво переходить к рассмотрению данного кода сразу с средины, поэтому будем продвигаться постепенно от самого начала кода и до конца. А начнем мы с понятия, которое вам, возможно, еще незнакомо.

Проверка на возможность сериализации функций

К сожалению, код, с которого начинается рассматриваемый здесь пример реализации наследования, имеет весьма таинственный вид и может смутить многих. Далее в коде данного примера потребуется выяснить, поддерживается ли сериализация функций в браузере. Но составленный для этой цели тест имеет довольно сложный синтаксис, поэтому мы пока что отставим его в сторону и попробуем сохранить результат тестирования таким образом, чтобы не усложнять и без того непростой последующий код.

Сериализация функции — это действие, состоящее в получении исходного текста из отдельно взятой функции. Эта операция потребуется нам в дальнейшем для того, чтобы

проверить, имеется ли в теле функции конкретная интересующая нас ссылка. В большинстве современных браузеров эта операция выполняется над функцией с помощью ее метода `toString()`. Как правило, сериализация функции производится при ее использовании в том контексте, в котором ожидается символьная строка, что автоматически приводит к вызову ее метода `toString()`. Именно это обстоятельство используется в рассматриваемом здесь коде, чтобы проверить, работоспособен ли механизм сериализации функции.

После установки логического значения `false` в переменной `initializing` (ниже будет пояснено, зачем это делается) работоспособность механизма сериализации функции проверяется в следующем выражении ❶:

```
/xyz/.test(function() { xyz; })
```

В этом выражении создается функция, содержащая текст **"xyz"** и передающая его методу `test()` регулярного выражения, проверяющему наличие символьной строки **"xyz"**. Если сериализация функции произведена правильно, т.е. в методе `test()` предполагается получить символьную строку, запускающую на выполнение метод `toString()` данной функции, то в результате будет получено логическое значение `true`.

С помощью данного текстового выражения устанавливается регулярное выражение, которое будет использоваться далее в рассматриваемом здесь коде. И делается это следующим образом:

```
superPattern = /xyz/.test(function() { xyz; }) ? /\b_super\b/ : /.*/;
```

В приведенной выше строке кода устанавливается переменная `superPattern`, которая будет в дальнейшем использоваться с целью проверить, содержит ли функция символьную строку **"_super"**. Это можно сделать лишь в том случае, если сериализация функций поддерживается, поэтому в тех браузерах, где она не допускается, подставляется шаблон для совпадения с любыми символами. Результат этой проверки будет использован в дальнейшем, но она производится теперь, чтобы не вставлять данное выражение в последующий код, синтаксис которого и без того довольно сложный.

Примечание

Более подробно регулярные выражения будут рассматриваться в следующей главе.

А теперь перейдем непосредственно к реализации метода подклассификации.

Инициализация подклассов

Итак, всего готово для объявления метода, который будет осуществлять подклассификацию суперкласса ❷. И делается это в следующем фрагменте кода:

```
Object.subClass = function(properties) {
var _super = this.prototype;
```

В этом фрагменте кода в класс `Object` вводится метод `subClass()`, принимающий единственный аргумент, которым должен быть набор свойств, добавляемых в подкласс.

Для того чтобы сымитировать наследование с помощью прототипа функции, применяется рассматривавшийся ранее способ создания экземпляра суперкласса и последующего его присваивания прототипу. Без учета предыдущей реализации этот способ выглядел бы в коде следующим образом:

```
function Person(){}
function Ninja(){}
Ninja.prototype = new Person();
assert((new Ninja()) instanceof Person,
      "Ninjas are people too!");
```

Недостаток этого фрагмента кода заключается в том, что нам на самом деле нужны лишь те преимущества, которые дает проверка экземпляра объекта с помощью оператора `instanceof`, но не все затраты на получение экземпляра объекта `Person` и выполнение его конструктора. Во избежание этого в рассматриваемом здесь коде имеется переменная `initializing`, в которой устанавливается логическое значение `true` всякий раз, когда требуется инициализировать класс с единственной целью использовать его для прототипа. Следовательно, когда наступает момент для построения экземпляра объекта, мы можем проверить, что не находимся в режиме инициализации, и соответственно выполнить или пропустить метод `init()`, как показано ниже.

```
if (!initializing && this.init)
  this.init.apply(this, arguments);
```

Следует особо подчеркнуть, что метод `init()` может выполнять всякие виды кода запуска, требующего затрат на соединение с сервером, создание элементов DOM и т.д. Но мы можем обойтись и без ненужного и затратного кода запуска, просто создав экземпляр объекта, чтобы он служил в качестве прототипа.

Далее требуется скопировать в экземпляр прототипа любые характерные для подкласса свойства, переданные методу подклассификации. Но сделать это не так просто, как кажется на первый взгляд.

Сохранение суперметодов

В большинстве языков программирования, поддерживающих наследование, сохраняется возможность доступа к переопределяемому методу. Это очень удобно, поскольку в одних случаях требуется полностью заменить функциональные возможности метода, а в других — только расширить их. В рассматриваемой здесь конкретной реализации создается новый временный метод `_super()`, который доступен только из подклассифицированного метода и ссылается на исходный метод в суперклассе.

Как было показано в примере кода из листинга 6.20, когда потребовалось вызвать конструктор суперкласса, для этой цели служил следующий код (некоторые его части опущены ради краткости):

```
var Person = Object.subClass({
  init: function(isDancing){
    this.dancing = isDancing;
  }
});

var Ninja = Person.subclass({
  init: function(){
    this._super(false);
  }
});
```

В конструкторе класса `Ninja` вызывается конструктор класса `Person`, которому передается соответствующее значение. Это избавляет от необходимости копировать

код, поскольку можно выгодно воспользоваться уже имеющимся в суперклассе кодом, который выполняет нужные действия. В коде из листинга 6.21 подобные функциональные возможности реализуются поэтапно. Для того чтобы расширить подкласс набором свойств объекта, передаваемым методу `subClass()`, достаточно объединить свойства суперкласса с передаваемыми свойствами.

Сначала нужно создать экземпляр суперкласса, чтобы использовать его в качестве прототипа ❸. И делается это в следующем фрагменте кода:

```
initializing = true;
var proto = new this();
initializing = false;
```

Обратите внимание на то, что код инициализации "защищается" присваиванием соответствующего значения переменной `initializing`, как пояснялось в предыдущем разделе.

Теперь нужно соединить передаваемые свойства с объектом `proto` (прототипом прототипа, если угодно) ❹. Если бы нам были безразличны функции суперкласса, эта задача оказалась бы для нас несложной, как показано ниже.

```
for (var name in properties) proto[name] = properties[name];
```

Но функции суперкласса нам *не* безразличны, и поэтому приведенный выше код годится для всех свойств, кроме тех функций, в которых требуется вызывать их эквиваленты из суперкласса. Когда функция переопределяется той, которая будет вызывать ее по ссылке `_super`, функцию подкласса необходимо заключить в оболочку, где ссылка на функцию суперкласса определяется в свойстве `_super`.

Но прежде чем сделать это, необходимо выявить условие, при котором функция подкласса должна быть заключена в оболочку. Для этого можно воспользоваться следующим условным выражением:

```
typeof properties[name] == "function" &&
typeof _super[name] == "function" &&
superPattern.test(properties[name])
```

Это выражение состоит из операторов, проверяющих следующие условия.

- Является ли свойство подкласса функцией?
- Является ли свойство суперкласса функцией?
- Содержит ли функция подкласса ссылку на метод `_super()`?

Выполнять любую другую операцию, кроме копирования значения свойства, следует лишь в том случае, если все три условия истинны (`true`). Для того чтобы проверить, вызывает ли функция свой эквивалент из суперкласса, используется шаблон рассмотренного ранее регулярного выражения наряду с сериализацией функции. Если же условное выражение указывает на то, что функцию следует заключить в оболочку, с этой целью свойству подкласса присваивается приведенная ниже функция немедленного вызова ❺.

```
(function(name, fn) {
  return function() {
    var tmp = this._super;

    this._super = _super[name];
```

```
    var ret = fn.apply(this, arguments);
    this._super = tmp;

    return ret;
  };
})(name, properties[name])
```

Эта функция немедленного вызова создает и возвращает новую функцию, заключающую в оболочку функцию подкласса, и в то же время делает доступной функцию суперкласса через свойство `_super`. Сначала в данной функции для порядка сохраняется старая ссылка `this._super`, независимо от того, существует ли она вообще, а по завершении эта ссылка восстанавливается. Это полезно сделать на тот случай, если уже существует переменная с таким же самым именем, чтобы не удалить ее случайно.

Далее создается новый метод `_super`, который служит лишь ссылкой на метод, существующий в прототипе суперкласса. Правда, здесь не нужно вносить никаких дополнительных изменений или переопределять область действия, поскольку контекст функции установится автоматически, если это свойство текущего объекта (ссылка `this` всегда делается на экземпляр текущего объекта, а не на суперкласс). И наконец, вызывается исходный метод, который делает то, что ему положено (возможно, даже используя метод `_super`), после чего восстанавливается исходное состояние ссылки на метод `_super` и происходит возврат из функции.

Имеются и другие способы, с помощью которых можно добиться аналогичного результата. В некоторых реализациях суперметод привязывается к самому методу, доступному через свойство `arguments.callee`, но рассмотренный здесь способ обеспечивает наилучшее сочетание простоты и удобства применения.

Резюме

Внедрение в JavaScript средств объектно-ориентированного программирования с помощью прототипов функций и прототипного наследования может стать большим благом для тех разработчиков, которые предпочитают писать код в объектно-ориентированном стиле. Благодаря объектно-ориентированному характеру код приложений на JavaScript становится более структурированным, управляемым, ясным и качественным.

В этой главе было показано, каким образом свойство `prototype` функций позволяет придать коду JavaScript объектно-ориентированный характер.

- Сначала в ней была подробно рассмотрена сущность самого свойства `prototype` и его роль в тех случаях, когда функция в сочетании с оператором `new` превращается в конструктор. На конкретных примерах наблюдалось поведение функций в роли конструкторов и его отличия от непосредственного вызова функции.
- Затем было показано, каким образом определяется тип объекта и создающий его конструктор.
- Далее было рассмотрено объектно-ориентированное понятие наследования и показано, как пользоваться цепочкой прототипов для осуществления наследования в коде JavaScript.
- После этого были рассмотрены типичные скрытые препятствия и западни, в которые можно опрометчиво попасть, расширяя класс `Object` и другие собственные классы JavaScript. Попутно были даны рекомендации, как обойти эти скрытые препятствия и избежать затруднений, возникающих из-за неправильного использования конструкторов при получении экземпляров объектов.
- И наконец, в этой главе был предложен синтаксис, который может быть использован для реализации подклассификации объектов в JavaScript, а в качестве примера был создан метод, в котором этот синтаксис применяется. (Этот пример не рассчитан на малодушных!)

Благодаря тому что прототипы позволяют расширять функциональные возможности вплоть до наследования, они предоставляют универсальную платформу для разработки в перспективе. В последнем примере кода, приведенном в этой главе, применялись регулярные выражения. Эти нередко упускаемые из виду весьма эффективные языковые средства JavaScript будут подробно рассмотрены в следующей главе.

7 Овладение регулярными выражениями

В этой главе...

- Основные положения о регулярных выражениях
- Компилирование регулярных выражений
- Фиксация результатов с помощью регулярных выражений
- Часто встречающиеся задачи и их решения с помощью регулярных выражений

Без регулярных выражений трудно обойтись в разработке современного программного обеспечения. И хотя многие разработчики могут счастливо прожить, пренебрегая регулярными выражениями, некоторые задачи программирования на JavaScript нельзя решить изящно без регулярных выражений. Безусловно, одни и те же задачи можно решать по-разному. Но зачастую полстраницы кода можно заменить одной строкой, применяя регулярные выражения надлежащим образом. Поэтому всякий, стремящийся стать настоящим мастером программирования на JavaScript, должен включить регулярные выражения в арсенал своих средств.

Регулярные выражения упрощают процесс разбиения символьных строк на отдельные части и поиска информации. В какую бы из основных библиотек JavaScript ни заглянуть, везде можно обнаружить, что регулярным выражениям отводится главенствующая роль в решении следующих задач.

- Манипулирование узлами HTML.
- Обнаружение частичных селекторов в выражениях CSS-селекторов.
- Определение имени конкретного класса у элемента.
- Извлечение непрозрачности из свойства `filter` в браузере Internet Explorer.
- И многое другое...

Совет

Для овладения регулярными выражениями требуется немалая практика. Оперативно поупражняться на примерах регулярных выражений можно, например, на веб-сайте *JS Bin* (`jsbin.com`). Еще один полезный веб-сайт, посвященный проверке регулярных выражений, находится по адресу `www.regexplanet.com/advanced/javascript/index.html`.

Итак, начнем рассмотрение регулярных выражений с простого примера.

Достоинства регулярных выражений

Допустим, требуется убедиться в том, что символьная строка, введенная в форме, заполняемой посетителем веб-сайта, соответствует принятому в США формату почтового кода из девяти цифр. Всем жителям США известно, что почтовое ведомство этой страны не любит шуток и категорически настаивает на том, чтобы почтовый код (или так называемый почтовый индекс) соответствовал следующему конкретному формату:

99999-9999

где **9** — десятичная цифра. Этот формат состоит из пяти десятичных цифр, дефиса и еще четырех десятичных цифр. Если указать на конверте с письмом или упаковке с посылкой почтовый индекс в другом формате, такое почтовое отправление затеряется где-то в недрах отделения ручной сортировки почты, и никто не знает, когда оно снова появится на свет.

Итак, создадим функцию, которая будет проверять переданную ей символьную строку на соответствие формату почтового индекса, принятому в США. Для этого можно было бы прибегнуть к сравнению каждого символа в строке, но вряд ли настоящий мастер посчитает такое решение изящным, поскольку оно подразумевает частое и ненужное повторение кода. Вместо этого рассмотрим решение, приведенное в примере кода из листинга 7.1.

Листинг 7.1. Проверка строки по конкретному шаблону

```
function isThisAZipCode(candidate) {

  if (typeof candidate !== "string" ||
     candidate.length != 10) return false;
  for (var n = 0; n < candidate.length; n++) {
    var c = candidate[n];
    switch (n) {
      case 0: case 1: case 2: case 3: case 4:
      case 6: case 7: case 8: case 9:
        if (c < '0' || c > '9') return false;
        break;
      case 5:
        if (c != '-') return false;
        break;
    }
  }
  return true;
}
```

Укороченные логические операторы, очевидно, не подходят

Выполнить тесты по индексу символа в строке

Если все прошло успешно, то и хорошо!

В приведенном выше примере кода выгодно используется тот факт, что в зависимости от положения символа в строке производятся только две разные проверки. А во время выполнения кода по-прежнему приходится выполнять до девяти сравнений, хотя код для каждого сравнения достаточно написать лишь один раз. Тем не менее следует ли считать изящным такое решение? Безусловно, оно более изящно, чем грубый неитеративный подход, но все-таки предполагает написание слишком большого объема кода для реализации столь простой проверки.

А теперь рассмотрим еще одно решение, приведенное ниже.

```
function isThisAZipCode(candidate) {
  return /^\d{5}-\d{4}$/.test(candidate);
}
```

Как видите, это намного более изящное и краткое решение, если не считать довольно таинственного вида кода в теле функции. В то же время данный пример наглядно демонстрирует потенциал, скрывающийся в регулярных выражениях, как в подводной части айсберга. Конечно, синтаксис регулярного выражения обычно выглядит так, как будто оно было набрано косой лапой на клавиатуре. Но не смущайтесь — мы сделаем краткий обзор регулярных выражений, прежде чем перейти к их мастерскому применению в разработке веб-приложений.

Основные положения о регулярных выражениях

Как бы нам ни хотелось, но мы не можем позволить себе пространное изложение материала по регулярным выражениям в силу ограниченности места в книге. Впрочем, на эту тему имеется довольно обширная литература. В частности, рекомендуем две книги: *Mastering Regular Expressions* Джеффри Е.Ф. Фридла (Jeffrey E.F. Friedl; издательство O'Reilly; в русском переводе книга вышла под названием *Регулярные выражения*), а также *Regular Expressions Cookbook* Яна Гойвертца и Стивена Левитана (Jan Goyvaerts, Steven Levithan; издательство O'Reilly). Но мы все же постараемся ниже осветить все самое главное, что касается регулярных выражений. Итак, приступим.

Назначение регулярных выражений

Термин *регулярное выражение* появился в математике в середине XX века, когда американский математик Стивен Клини (Stephen Kleene) занимался описанием моделей автоматов (абстрактных вычислительных устройств) как регулярных множеств. Но это объяснение вряд ли поможет лучше понять назначение регулярных выражений, поэтому сформулируем его проще: регулярное выражение — это способ обозначить *шаблон* для сопоставления с текстовыми строками. Само регулярное выражение состоит из членов и операторов, позволяющих определять такие шаблоны. Ниже будет показано, что собой представляют члены и операторы регулярного выражения.

В JavaScript, как и в большинстве других языков объектно-ориентированного программирования, регулярное выражение может быть создано двумя способами: с помощью литерала регулярного выражения и построения экземпляра объекта типа `RegExp`. Так, если требуется создать очень простое регулярное выражение для точного сопоставления с символьной строкой `"test"`, это можно сделать с помощью литерала регулярного выражения следующим образом:

```
var pattern = /test/;
```

Такое регулярное выражение с косыми чертами выглядит не совсем обычно, но литералы регулярных выражений заключаются в косые черты точно так же, как и строковые литералы в кавычки. С другой стороны, можно построить экземпляр объекта типа RegExp, передав ему регулярное выражение в качестве символьной строки, как показано ниже.

```
var pattern = new RegExp("test");
```

В обоих случаях создается одно и то же регулярное выражение, сохраняемое в переменной pattern.

Если регулярное выражение заранее известно во время разработки, то предпочтение отдается синтаксису литерала, тогда как синтаксис конструктора применяется при построении регулярного выражения во время выполнения кода динамически в символьной строке. Предпочтение, отдаваемое синтаксису литерала, над синтаксисом конструктора объясняется тем, что символ обратной косой черты играет очень важную роль в регулярных выражениях, как станет ясно в дальнейшем. Но ведь символ обратной косой черты выполняет также роль знака переключения кода в строковых литералах, и поэтому для обозначения этого символа в строковом литерале его приходится указывать в виде двойной обратной косой черты (\\). Именно поэтому регулярные выражения, которые и без того обладают не очень понятным синтаксисом, могут иметь совершенно необычный вид, когда они обозначаются в символьных строках.

Помимо самого выражения, в регулярном выражении могут быть указаны следующие три флажка:

- i — делает регулярное выражение не зависящим от регистра, поэтому регулярное выражение /test/i совпадает не только с символьной строкой "test", но и со строками "Test", "TEST", "tEsT" и т.д.
- g — допускает совпадение со всеми экземплярами шаблона, в отличие устанавливаемого по умолчанию "локального" совпадения только с первым экземпляром. Подробнее об этом речь пойдет далее.
- m — допускает сопоставление со многими строками, которые могут быть получены из значения элемента разметки textarea.

Эти флажки могут присоединяться в конце литерала (например, /test/ig) или же передаваться в символьной строке в качестве второго параметра конструктора типа RegExp (например, new RegExp("test", "ig")). Простое совпадение с символьной строкой "test" (пускай даже точное и без учета регистра) не особенно интересно. Ведь такую проверку можно выполнить простым сравнением символьных строк. Поэтому рассмотрим члены и операторы, которые наделяют регулярные выражения огромным потенциалом для сопоставления с более сложными шаблонами.

Члены и операторы

Регулярные выражения, как и большинство других известных вам выражений, состоят из членов и операторов, уточняющих эти члены. В последующих подразделах мы рассмотрим эти члены и операторы и покажем, как пользоваться ими для обозначения шаблонов.

Точное совпадение

Любой символ, не являющийся специальным или оператором, представляет собой символ, который должен буквально присутствовать в выражении. Например, в упоминавшемся выше примере регулярного выражения /test/ имеются четыре члена, представляющих символы, которые должны буквально присутствовать в символьной строке, чтобы она точно совпала с шаблоном, обозначенным в данном регулярном выражении. Расположение символов друг за другом неявно указывает на операцию, которая означает "за одним следует другое". Таким образом, регулярное выражение /test/ означает, что за символом t следует символ e, за ним — символ s, а затем — символ t.

Совпадение с классом символов

Зачастую требуется совпадение не с одним конкретным символом, а с любым символом из конечного набора. Такое условие можно задать с помощью оператора над множеством, называемого также *оператором класса символов*, для чего набор сопоставляемых символов заключается в квадратные скобки. Например, выражение [abc] означает, что требуется совпадение с любым из символов a, b или c. Следует, однако, иметь в виду, что совпадение в данном примере будет происходить только с одним символом в проверяемой строке, несмотря на то, что выражение фактически состоит из пяти символов.

Но иногда требуется совпадение со всеми символами, *кроме* указанных в конечном наборе. Для этого достаточно указать знак вставки (^) сразу после открывающей квадратной скобки в операторе над множеством:

```
[^abc]
```

И в этом случае совершенно меняется смысл регулярного выражения. Теперь оно означает совпадение с любыми символами, *кроме* символов a, b или c.

Имеется еще одна бесценная разновидность операции над множеством, позволяющая указывать диапазон значений. Так, если бы требовалось совпадение с любой строчной буквой в пределах от a до m, для этой цели можно было бы составить выражение [abcdefghijklm]. Но то же самое условие можно выразить более кратко, как показано ниже.

```
[a-m]
```

Дефис в данном выражении обозначает пределы от a до m включительно, т.е. в сопоставляемый набор символов входят буквально все указанные выше строчные буквы.

Экранирование

Не все символы представляют свой буквальный эквивалент. Разумеется, все буквенно-цифровые символы представляют самих себя, но, как будет показано ниже, специальные знаки, например, денежной единицы ($) и точки (.), обозначают совпадение с каким-то другим символом, а не с самими собой, или же операторы, уточняющие предыдущий член выражения. В представленных ранее примерах регулярных выражений вам уже встречались знаки [,], - и ^, представляющие не самих себя, а нечто другое.

Как же указать, что требуется совпадение с самим специальным знаком, например [, $ или ^? Для этой цели в регулярном выражении служит знак обратной косой черты, "экранирующий" любой следующий за ним символ, делая его тем самым буквально сопоставляемым членом данного выражения. Следовательно, последовательность символов \[обозначает буквальное совпадение со знаком [, а не открытие выражения с классом символов, а двойная обратная косая черта (\\) — совпадение с единственным знаком обратной косой черты.

Начало и конец сопоставлений

Нередко требуется, чтобы сопоставление с шаблоном происходило в начале, а возможно, и в конце символьной строки. Так, если знак вставки указывается первым символом в регулярном выражении, сопоставление с шаблоном привязывается к началу символьной строки. Например, в регулярном выражении `/^test/` совпадение произойдет лишь в том случае, если подстрока `"test"` окажется в начале проверяемой символьной строки. (Следует заметить, что это своего рода перегрузка знака **^**, поскольку он используется также для исключения набора символов из сопоставления.)

Аналогично знак денежной единицы (**$**) обозначает, что сопоставляемый шаблон должен появиться в конце символьной строки: **`/test$/`**. Если же в выражении используются оба знака, **^** и **$**, это означает, что указанный шаблон должен охватывать всю проверяемую символьную строку: `/^test$/`.

Повторяющиеся экземпляры

Если требуется сопоставление с четырьмя подряд символами **a**, то шаблон для него можно выразить как `/aaaa/`. Но что, если требуется совпадение с *любым* количеством одного и того же символа? Для указания количества различных вариантов повторений в регулярных выражениях предоставляются следующие средства.

- Если требуется задать символ как необязательный (иными словами, он может присутствовать один раз или вообще отсутствовать в проверяемой на совпадение строке), после него следует указать знак вопроса (**?**). Например, регулярное выражение `/t?est/` обеспечивает совпадение как со строкой `"test"`, так и со строкой `"est"`.
- Если требуется, чтобы символ присутствовал однократно или многократно в проверяемой на совпадение строке, после него следует указать знак "плюс" (**+**). Например, регулярное выражение `/t+est/` обеспечивает совпадение со строками `"test"`, `"ttest"` и `"tttest"`, но не со строкой `"est"`.
- Если требуется, чтобы символ присутствовал многократно или вообще отсутствовал в проверяемой на совпадение строке, после него следует указать знак звездочки (*****). Например, регулярное выражение `/t*est/` обеспечивает совпадение со строками `"test"`, `"ttest"`, `"tttest"` *и* `"est"`.
- Если требуется задать фиксированное число повторений символа в проверяемой на совпадение строке, это число следует указать в фигурных скобках после данного символа. Например, регулярное выражение `/a{4}/` означает проверку на совпадение с четырьмя подряд символами a.
- Если требуется задать число повторений символа в определенных пределах, эти пределы следует указать через запятую в фигурных скобках после проверяемого символа. Например, регулярное выражение `/a{4,10}/` обеспечивает совпадение с любой строкой, в которой следует подряд от четырех до десяти символов **a**.
- Если требуется задать число повторений символа в расширяемых пределах, эти пределы следует указать в фигурных скобках после проверяемого символа, опустив второе числовое значение, но оставив запятую. Например, регулярное выражение `/a{4,}/` обеспечивает совпадение с любой строкой, в которой следуют подряд четыре и больше символов a.

Любые из перечисленных выше операторов повторения могут быть как *поглощающими*, так и *непоглощающими*. По умолчанию эти операторы поглощающие, т.е. они охватывают все символы, которые могут совпасть с сопоставляемым шаблоном. Если же указать оператор с последующим знаком вопроса (**?**), т.е. перегрузить оператор **?**, как, например, `a+?`, такая операция окажется непоглощающей. Это означает, что она будет охватывать лишь столько символов, сколько окажется достаточно для совпадения.

Так, если требуется проверить совпадение с символьной строкой `"aaa"`, то регулярное выражение `/a+/` обеспечит совпадение со всеми тремя символами a, тогда как непоглощающее выражение `/a+?/` — только с одним символом a. Ведь для удовлетворения условия, задаваемого членом `a+` такого выражения, достаточно совпадения с единственным символом a.

Предопределенные классы символов

Имеются символы, которые требуется проверить на совпадение, но которые невозможно указать буквально. К их числу относятся такие управляющие символы, как возврат каретки. Имеются также классы символов, которые часто проверяются на совпадение. К их числу относятся наборы десятичных цифр или пробелов. Для представления подобных символов или общеупотребительных классов символов в синтаксисе регулярных выражений предоставляется целый ряд предопределенных членов, с помощью которых можно проверить на совпадение управляющие символы. Это избавляет от необходимости прибегать к оператору класса символов для сопоставления с общеупотребительными наборами символов в регулярных выражениях.

Все эти члены, а также отдельные управляющие символы или наборы символов, которые они представляют, перечислены в табл. 7.1.

Таблица 7.1. Члены, обозначающие управляющие символы и предопределенные классы символов

Предопределенный член	Сопоставление
`\t`	Горизонтальная табуляция
`\b`	Возврат на одну позицию
`\v`	Вертикальная табуляция
`\f`	Перевод страницы
`\r`	Возврат каретки
`\n`	Перевод строки
`\cA:\cZ`	Управляющие символы
`\x0000:\xFFFF`	Шестнадцатеричные значения символов в уникоде
`\x00:\xFF`	Шестнадцатеричные значения символов в коде ASCII
`.`	Любой символ, кроме перевода строки (`\n`)
`\d`	Любая десятичная цифра, что эквивалентно выражению `[0-9]`
`\D`	Любой символ, кроме десятичной цифры, что эквивалентно выражению `[^0-9]`
`\w`	Любой буквенно-цифровой символ, включая и знак подчеркивания, что эквивалентно выражению `[A-Za-z0-9_]`
`\W`	Любой символ, кроме буквенно-цифровых, включая и знак подчеркивания, что эквивалентно выражению `[^A-Za-z0-9_]`

Окончание табл. 7.1

Предопределенный член	Сопоставление
\s	Любой символ пробела (собственно пробела, табуляции, перевода строки, перевода страницы и т.д.)
\S	Любой символ, кроме пробела
\b	Граница слова
\B	Не граница слова, а его внутренняя часть

Упомянутые выше преопределенные наборы символов помогают придать регулярным выражениям менее таинственный и запутанный вид.

Группирование

В приведенных до сих пор примерах было показано, что операторы, например + или *, оказывают воздействие только на предшествующий член регулярного выражения. Но если требуется применить оператор к группе членов, то для этого можно воспользоваться круглыми скобками, заключив в них группы членов регулярного выражения таким же образом, как это делается в любом математическом выражении. Например, регулярное выражение `/(ab)+/` обеспечивает совпадение с одним последовательным вхождением подстроки `"ab"` или больше.

Когда часть регулярного выражения заключается в круглые скобки как отдельная группа, она служит также для так называемой *фиксации*. Имеются самые разные виды фиксации, более подробно рассматриваемые далее в главе.

Чередование (логическое сложение)

Выбор альтернативных вариантов обозначается с помощью знака вертикальной черты (`|`). Например, регулярное выражение `/a|b/` обеспечивает совпадение с символом `a` или `b`, а регулярное выражение `/(ab)+|(cd)+/` — с одним или более вхождением подстроки `"ab"` или `"cd"`.

Обратные ссылки

К числу самых сложных членов, которые можно обозначить в регулярных выражениях, относятся обратные ссылки на *фиксации*, определяемые в этих выражениях. Подробнее о фиксациях речь пойдет далее в главе, а до тех пор достаточно сказать, что их следует рассматривать как части проверяемой символьной строки, успешно совпавшие с членами регулярного выражения. Такие члены обозначаются обратной косой чертой и номером фиксации, на которую делается ссылка, начиная с `1`, например `\1`, `\2` и т.д.

В качестве примера рассмотрим регулярное выражение `/^([dtn])a\1/`, которое обеспечивает совпадение с символьной строкой, начинающейся с любого из символов `d`, `t` или `n`, за которыми следует символ `a` и далее любой символ, совпадающий с первой фиксацией. Последнее очень важно уяснить! Ведь это не одно и то же, что и выражение `/[dtn]a[dtn]/`. Символ, следующий после символа `a`, не может быть любым из символов `d`, `t` или `n`, но должен быть каким угодно из тех символов, которые инициируют совпадение с первым символом. Следовательно, символ, который совпадет с членом `\1`, неизвестен до самого момента вычисления данного регулярного выражения.

Характерным примером полезного применения обратных ссылок в регулярных выражениях может служить проверка на совпадение с элементами разметки XML-документов. Рассмотрим следующий пример:

```
/<(\w+)>(.+)<\/\1>/
```

С помощью такого регулярного выражения можно проверить на совпадение такие простые элементы разметки, как, например, `<strong>все что угодно</strong>`. Подобная проверка оказалась бы невозможной без указания обратной ссылки. Ведь заранее неизвестно, какой именно закрывающий дескриптор совпадет с открывающим.

Совет

Приведенный выше материал можно сравнить с головокружительным ускоренным курсом по регулярным выражениям. Если они все еще кружат вам голову и вы чувствуете, что вам трудно будет усвоить последующий материал этой главы, настоятельно рекомендуется обратиться к дополнительной литературе, упоминавшейся в начале главы.

А теперь, рассмотрев основные положения о регулярных выражениях, перейдем к их благоразумному применению непосредственно в коде.

Компиляция регулярных выражений

Регулярные выражения проходят многочисленные стадии выполнения, поэтому ясное представление о том, что происходит на каждой стадии этого процесса, способствует написанию оптимизированного кода на JavaScript. К наиболее примечательным среди них относятся стадии компиляции и выполнения. В частности, компиляция происходит всякий раз, когда регулярное выражение определяется впервые, а выполнение — когда скомпилированное регулярное выражение используется для сопоставления шаблонов с символьной строкой.

Во время компиляции механизм JavaScript выполняет синтаксический анализ регулярного выражения и преобразует его во внутреннее представление, каким бы оно ни было. Стадия синтаксического анализа и преобразования должна происходить всякий раз, когда в коде JavaScript встречается регулярное выражение, если только браузер не выполнил его оптимизацию.

Зачастую браузеры способны *сами* определить, используются ли идентичные регулярные выражения, чтобы кешировать результаты компиляции конкретного повторяющегося выражения. Но это характерно далеко не для всех браузеров. Так, если речь идет о сложных выражениях, заметных улучшений в быстродействии можно добиться путем предварительного определения, а следовательно, и предварительной компиляции регулярного выражения для последующего его применения.

Как следует из приведенного выше краткого обзора регулярных выражений, в JavaScript они могут быть скомпилированы двумя способами: с помощью литерала или конструктора. Обратимся к конкретному примеру, приведенному в листинге 7.2.

Листинг 7.2. Два способа создания скомпилированного регулярного выражения

```
<script type="text/javascript">

  var re1 = /test/i;
```

Создать регулярное выражение с помощью литерала

```
  var re2 = new RegExp("test", "i");

  assert(re1.toString() == "/test/i",
        "Verify the contents of the expression.");
  assert(re1.test("TesT"), "Yes, it's case-insensitive.");
  assert(re2.test("TesT"), "This one is too.");
  assert(re1.toString() == re2.toString(),
        "The regular expressions are equal.");
assert(re1 != re2, "But they are different objects.");

</script>
```

Создать регулярное выражение с помощью конструктора

В приведенном выше примере кода оба регулярных выражения оказываются в конечном итоге в скомпилированном состоянии. Если заменить каждую ссылку на переменную `re1` литералом `/test/i`, то одно и то же выражение будет компилироваться снова и снова. Поэтому однократная компиляция регулярного выражения и последующее его сохранение в переменной может стать важной мерой для оптимизации кода.

Обратите внимание на то, что у каждого регулярного выражения имеется разное объектное представление. Всякий раз, когда регулярное выражение определяется, а следовательно, и компилируется, создается также новый объект регулярного выражения. Отличие регулярных выражений от других примитивных типов, включая строковые и числовые, в том и состоит, что их компиляция всегда дает однозначный результат.

Особое значение имеет применение конструктора `new RegExp(...)` для создания нового регулярного выражения. Такой способ позволяет составлять и компилировать выражение из символьной строки, динамически формируемой во время выполнения. И это может быть очень удобно для построения сложных выражений, чтобы пользоваться ими неоднократно.

Допустим, в документе требуется определить элементы с именем определенного класса, неизвестным до времени компиляции. Имена элементов могут быть связаны с разными классами и храниться в неудобном формате, разделенными пробелами в символьной строке. Это представляет удобную возможность для компиляции регулярного выражения во время выполнения, как показано в листинге 7.3.

Листинг 7.3. Компиляция регулярного выражения для последующего применения

```
<div class="samurai ninja"></div>
<div class="ninja samurai"></div>
<div></div>
<span class="samurai ninja ronin"></span>

<script>
  function findClassInElements(className, type) {

    var elems =
      document.getElementsByTagName(type || "*");

    var regex =
      new RegExp("(^|\\s)" + className + "(\\s|$)");

  var results = [];
```

❶ Создать объекты для тестирования из различных элементов разметки с разными именами классов

❷ Собрать элементы по типу

❸ Скомпилировать регулярное выражение, используя переданное имя класса

❹ Сохранить результат


```
  for (var i = 0, length = elems.length; i < length; i++)
    if (regex.test(elems[i].className)) {
      results.push(elems[i]);
    }
    return results;
  }

assert(findClassInElements("ninja", "div").length == 2,
      "The right amount of div ninjas was found.");
assert(findClassInElements("ninja", "span").length == 1,
      "The right amount of span ninjas was found.");
assert(findClassInElements("ninja").length == 3,
      "The right amount of ninjas was found.");
</script>
```

❺ Проверить регулярное выражение на совпадения

Из примера кода, приведенного в листинге 7.3, можно извлечь немало полезных уроков. Сначала в данном примере устанавливается ряд элементов разметки `<div>` и `<span>` с именами классов в разных сочетаниях, чтобы служить в качестве объектов для тестирования ❶. Затем определяется функция для проверки имен классов, принимающая в качестве своих аргументов имя проверяемого класса, а также тип элемента разметки, в котором он проверяется.

Далее собираются все элементы указанного типа ❷ и составляется регулярное выражение ❸. Обратите внимание на применение конструктора `new RegExp()` для компилирования регулярного выражения на основании имени класса, передаваемого функции. Это экземпляр объекта, в котором нельзя использовать литерал регулярного выражения, поскольку искомое имя класса заранее неизвестно.

Данное регулярное выражение составляется, а следовательно, и компилируется, лишь один раз, чтобы исключить частые обращения к его повторной компиляции. А поскольку содержимое выражения носит динамический характер (и зависит от входящего аргумента `className`), то, обращаясь с выражением подобным образом, можно добиться значительного выигрыша в производительности.

Само регулярное выражение обеспечивает совпадение с началом символьной строки или символом пробела, затем с целевым именем класса и далее с символом пробела или концом строки. Обратите особое внимание в нем на использование двойной обратной косой черты (`\\`) для экранирования этого знака в выражении `\\s`. При создании регулярных выражений с помощью литералов обратная косая черта указывается в членах выражения только один раз. Но поскольку этот знак записывается в символьной строке, его необходимо экранировать, указывая дважды. Об этой особенности не следует забывать при составлении регулярных выражений в символьных строках, а не литералах. Как только регулярное выражение будет скомпилировано, совпадающие элементы ❹ сразу же собираются с его помощью в методе `test()` ❺.

Предварительное составление и компиляция регулярных выражений с целью их повторного использования (и выполнения) настоятельно рекомендуется как эффективный способ повышения производительности, которым не стоит пренебрегать. Ведь этот способ приносит заметные выгоды практически во всех случаях применения сложных регулярных выражений.

В начале этого раздела упоминалось о том, что круглые скобки служат в регулярных выражениях не только для группирования их членов и применения к ним операторов, но и для так называемых *фиксаций*. Теперь настало время рассмотреть это понятие более подробно, что и будет сделано в следующем разделе.

Фиксация совпадающих частей

Польза от применения регулярных выражений становится более очевидной, когда происходит *фиксация* полученных результатов для последующей их обработки тем или иным способом. Первым очевидным шагом на этом пути может стать простая проверка на совпадение символьной строки с шаблоном. И зачастую этого оказывается достаточно, хотя во многих случаях полезно также выяснить, *что* именно совпало в результате подобной проверки.

Выполнение простых фиксаций

Допустим, требуется извлечь значение из сложной символьной строки. Характерным примером такой ситуации может служить порядок указания значений непрозрачности для совместимости с прежними версиями в браузере Internet Explorer. Вместо обычного правила `opacity` с числовым значением непрозрачности, установленного в других браузерах, в Internet Explorer 8 и прежних версиях этого браузера применяется следующее правило:

```
filter:alpha(opacity=50);
```

В примере кода из листинга 7.4 значение извлекается из приведенной выше символьной строки фильтрации.

Листинг 7.4. Простая функция для фиксации встраиваемого значения

```
<div id="opacity"
    style="opacity:0.5;filter:alpha(opacity=50);">
</div>

<script type="text/javascript">
  function getOpacity(elem) {
    var filter = elem.style.filter;
    return filter ?
      filter.indexOf("opacity=") >= 0 ?
        (parseFloat(filter.match(/opacity=([^)]+)/)[1]) / 100) + "" :
        "" :
      elem.style.opacity;
  }

  window.onload = function() {
    assert(
          getOpacity(document.getElementById("opacity")) == "0.5",
          "The opacity of the element has been obtained.");
  };
</script>
```

❶ Определить объект для тестирования

❷ Выяснить, что следует возвратить

Сначала в приведенном выше примере кода определяется элемент разметки, в котором указываются оба стиля непрозрачности: один — для браузеров, соблюдающих стандарты, а другой — для прежних версий Internet Explorer. Именно этот элемент и послужит объектом для тестирования ❶. Затем создается функция, возвращающая устанавливаемое по стандарту значение непрозрачности в пределах от **0,0** до **1,0**, независимо от того, как оно было изначально определено.

Код выявления непрозрачности путем синтаксического анализа регулярного выражения может показаться, на первый взгляд, немного запутанным ❷, но если разобрать его по частям, то он окажется вполне логично составленным. Прежде всего необходимо выяснить, имеется ли вообще свойство `filter` для синтаксического анализа. Если оно отсутствует, в таком случае можно попытаться получить доступ к свойству стиля `opacity`. А если свойство `filter` присутствует, то следует проверить, содержит ли оно искомую строку со значением непрозрачности. С этой целью вызывается функция `indexOf()`.

После этого можно, наконец, перейти непосредственно к извлечению искомого значения непрозрачности. В частности, метод `match()` с регулярным выражением возвращает массив зафиксированных значений, если совпадение обнаружено, или же пустое значение `null`, если оно не обнаружено. В данном случае можно не сомневаться, что совпадение произойдет, поскольку этот факт уже был выяснен при вызове функции `indexOf()`. Массив, возвращаемый методом `match()`, содержит результат полного совпадения в первом своем элементе, а в каждом последующем элементе — остальные зафиксированные результаты.

Итак, в первом элементе массива будет храниться полностью совпавшая символьная строка `"filter:alpha(opacity=50)"`, а в следующем элементе — значение 50. Напомним, что в регулярном выражении фиксации определяются круглыми скобками. Следовательно, при совпадении значение непрозрачности будет находиться в массиве на позиции `[1]`, поскольку единственная фиксация, указанная в данном регулярном выражении, была сделана в круглых скобках после первой части `opacity=`.

В данном примере были использованы локальное регулярное выражение и метод `match()`. Но совсем другое дело, когда используются глобальные регулярные выражения. Рассмотрим далее, как это происходит.

Проверка на совпадение с помощью глобальных регулярных выражений

Как было показано в предыдущем разделе, если локальное регулярное выражение (без глобального флажка `g`) используется в методе `match()`, вызываемом для объекта типа `String`, то из этого метода возвращается массив, содержащий полностью совпавшую строку наряду с любыми другими результатами совпадений, зафиксированными в ходе данной операции. Но если предоставить методу `match()` глобальное регулярное выражение (с глобальным флажком `g`), то будет возвращен совсем другой результат. Это будет по-прежнему массив, но он будет содержать результаты глобальных совпадений. Ведь глобальное регулярное выражение обеспечивает все возможные варианты совпадений в проверяемой символьной строке, а не только первое совпадение. И в этом случае результаты, зафиксированные при каждом совпадении, не возвращаются. Покажем, как это происходит, на примере кода, приведенном в листинге 7.5.

Листинг 7.5. Отличия глобального и локального поиска на совпадение

```
<script type="text/javascript">

  var html = "<div class='test'><b>Hello</b> <i>world!</i></div>";

  var results = html.match(/<(\/?)(\w+)([^>]*?)>/);

  assert(results[0] == "<div class='test'>", "The entire match.");
```

❶ Проверить на совпадение с помощью локального регулярного выражения

```
  assert(results[1] == "", "The (missing) slash.");
  assert(results[2] == "div", "The tag name.");
  assert(results[3] == " class='test'", "The attributes.");

  var all = html.match(/<(\/?)(\w+)([^>]*?)>/g);

  assert(all[0] == "<div class='test'>", "Opening div tag.");
  assert(all[1] == "<b>", "Opening b tag.");
  assert(all[2] == "</b>", "Closing b tag.");
  assert(all[3] == "<i>", "Opening i tag.");
  assert(all[4] == "</i>", "Closing i tag.");
  assert(all[5] == "</div>", "Closing div tag.");

</script>
```

❷ Проверить на совпадение с помощью глобального регулярного выражения

Как следует из приведенного выше примера кода, при локальном совпадении ❶ возвращается единственный совпавший экземпляр и зафиксированные при этом результаты, а при глобальном совпадении ❷ – список совпадений. Если же имеют значение фиксации, их можно восстановить, выполняя глобальный поиск совпадений и вызывая для этой цели метод `exec()` с регулярным выражением всякий раз, когда требуется возвратить очередную порцию совпавших данных. Типичный тому пример приведен в коде из листинга 7.6.

Листинг 7.6. Глобальный поиск и фиксация результатов с помощью метода exec()

```
<script type="text/javascript">

  var html = "<div class='test'><b>Hello</b> <i>world!</i></div>";
  var tag = /<(\/?)(\w+)([^>]*?)>/g, match;
  var num = 0;

  while ((match = tag.exec(html)) !== null) {
    assert(match.length == 4,
           "Every match finds each tag and 3 captures.");
    num++;
  }

  assert(num == 6, "3 opening and 3 closing tags found.");

</script>
```

❶ Многократно вызывать метод exec()

В приведенном выше примере кода метод `exec()` вызывается многократно ❶, благодаря чему сохраняется состояние из его предыдущего вызова. Таким образом, при каждом последующем вызове данного метода происходит переход к следующему глобальному совпадению. После каждого вызова метода `exec()` возвращается очередное совпадение *и* его фиксации.

С помощью метода `match()` или `exec()` можно всегда обнаружить точные совпадения и фиксации искомых результатов. Но этого оказывается недостаточно, когда возникает потребность в обратной ссылке на сами фиксации.

Ссылки на фиксации

Существуют два способа обратной ссылки на отдельные части зафиксированных результатов совпадения. Первым способом это делается внутри совпадения, а вторым – в строке замены (там, где это, конечно, возможно). Вернемся к примеру из листинга 7.6, где обнаруживается совпадение с открывающим или закрывающим дескриптором HTML. Видоизменим его таким образом, чтобы обнаруживалось также совпадение с внутренним содержимым дескриптора, как показано в листинге 7.7.

Листинг 7.7. Применение обратных ссылок для проверки на совпадение с содержимым дескриптора HTML

```
<script type="text/javascript">

  var html = "<b class='hello'>Hello</b> <i>world!</i>";

  var pattern = /<(\w+)([^>]*)>(.*?)<\/\1>/g;

  var match = pattern.exec(html);

  assert(match[0] == "<b class='hello'>Hello</b>",
         "The entire tag, start to finish.");
  assert(match[1] == "b", "The tag name.");
  assert(match[2] == " class='hello'", "The tag attributes.");
  assert(match[3] == "Hello", "The contents of the tag.");

  match = pattern.exec(html);

  assert(match[0] == "<i>world!</i>",
         "The entire tag, start to finish.");
  assert(match[1] == "i", "The tag name.");
  assert(match[2] == "", "The tag attributes.");
  assert(match[3] == "world!", "The contents of the tag.");

</script>
```

Использовать обратную ссылку

Сопоставить шаблон с проверяемой строкой

Проверить различные фиксации по определенному шаблону

В примере кода из листинга 7.7 член `\1` используется в регулярном выражении для обратной ссылки на первую фиксацию в этом выражении (в данном случае это имя дескриптора). Используя эту информацию, можно обнаружить совпадение с соответствующим закрывающим дескриптором по обратной ссылке на исходный дескриптор, конечно, при условии, что в текущем дескрипторе отсутствуют другие дескрипторы с таким же самым именем. Поэтому данный пример едва ли можно считать исчерпывающим.

Кроме того, ссылки на фиксации в заменяющей строке можно получить с помощью метода `replace()`. Вместо кодов обратной ссылки, как в предыдущем примере, в данном случае используется синтаксис `$1`, `$2`, `$3` и так далее для ссылки на каждую фиксацию по ее номеру, как показано ниже.

```
assert("fontFamily".replace(/([A-Z])/g, "-$1").toLowerCase() ==
       "font-family", "Convert the camelCase into dashed notation.");
```

В этом фрагменте кода ссылка ($1) на первое зафиксированное значение (в данном случае прописную букву F) делается в *строке замены*. Это позволяет указывать строку замены, даже не зная, каким будет ее значение до самого момента совпадения. Такой эффективный прием стоит взять на вооружение!

Наличие ссылок на фиксации в регулярных выражениях способствует существенному упрощению и удобочитаемости кода. Выразительный характер таких ссылок позволяет сделать краткими и ясными операторы, которые в противном случае получились бы довольно запутанными, неясными и длинными.

Группируемые фиксации и выражения заключаются в круглые скобки, поэтому процессору регулярных выражений неизвестно, какие именно скобки используются для группирования членов регулярного выражения и какие из них служат для обозначения фиксаций. Все они интерпретируются как группы и фиксации, что может привести к фиксации большего объема информации, чем требуется на самом деле. Как же поступать в подобных случаях? Ответ на этот вопрос дается в следующем разделе.

Нефиксируемые группы

Как отмечалось ранее, круглые скобки имеют двойное назначение: они не только группируют члены регулярного выражения для выполнения операций, но и обозначают фиксации. Обычно это не вызывает особых затруднений, но если в регулярном выражении происходит частое группирование, то в конечном итоге может быть зафиксировано много лишней информации. А это, в свою очередь, затрудняет сортировку зафиксированных результатов. В качестве примера рассмотрим следующее регулярное выражение:

```
var pattern = /((ninja-)+)sword/;
```

В данном примере преследуется цель составить такое регулярное выражение, которое должно приводить к появлению префикса "ninja-" один или больше раз перед словом "sword", а также к фиксации всего префикса в целом. Для этого в данном регулярном выражении требуются два ряда круглых скобок.

- Внешние скобки обозначают фиксацию всего, что предшествует символьной строке "sword".
- Внутренние скобки группируют префикс "ninja-" для применения оператора +.

Все это верно, но в результате возникает не одна предполагаемая фиксация, а больше из-за наличия внутренних группирующих скобок. Для того чтобы указать, что круглые скобки не должны приводить к фиксации, в синтаксисе регулярных выражений предусмотрено обозначение ?:, которое делается сразу же после открывающей круглой скобки. Это обозначение называется *пассивным подвыражением*.

Следовательно, если заменить упомянутое выше регулярное выражение на следующее:

```
var pattern = /((?:ninja-)+)sword/;
```

то к фиксации результатов приведут только внешние круглые скобки. А внутренние круглые скобки в этом выражении будут преобразованы в пассивное подвыражение. Для того чтобы убедиться в этом на практике, рассмотрим пример кода, приведенный в листинге 7.8.

Листинг 7.8. Группирование без фиксации

```
<script type="text/javascript">

  var pattern = /((?:ninja-)+)sword/;
  var ninjas = "ninja-ninja-sword".match(pattern);

  assert(ninjas.length == 2,"Only one capture was returned.");
  assert(ninjas[1] == "ninja-ninja-",
        "Matched both words, without any extra capture.");

</script>
```

❶ Использовать пассивное подвыражение

Результаты выполнения тестов в приведенном выше примере кода показывают, что пассивное подвыражение ❶ препятствует появлению лишних фиксаций. Поэтому при всякой возможности следует стремиться использовать нефиксируемые (т.е. пассивные) группы вместо фиксации, когда она не требуется, чтобы облегчить работу механизму обработки регулярных выражений по запоминанию и возврату фиксаций. Ведь если зафиксированные результаты не нужны, то зачем их запрашивать! Правда, за это приходится платить усложнением и без того запутанных регулярных выражений.

А теперь обратим внимание на еще один способ раскрытия истинного потенциала регулярных выражений. И состоит он в применении функций вместе с методом replace() из класса String.

Замена с помощью функций

Метод replace() из класса String довольно эффективен и универсален, в чем мы уже успели убедиться ранее при обсуждении фиксации. Когда регулярное выражение предоставляется методу replace() в качестве первого аргумента, это приводит к замене на результат совпадения (или *совпадений*, если регулярное выражение является глобальным) по шаблону, а не на фиксированную символьную строку.

Допустим, в символьной строке требуется заменить все символы верхнего регистра на букву "X". Для этого можно было бы составить приведенное ниже регулярное выражение, которое дало бы в результате символьную строку "XXXXXfg".

```
"ABCDEfg".replace(/[A-Z]/g,"X")
```

Это, конечно, замечательно, но, вероятно, наиболее сильной стороной метода replace() является возможность предоставить в качестве заменяющего значения функцию, а не фиксированную символьную строку. Когда заменяющее значение (второй аргумент данного метода) оказывается функцией, она вызывается для каждого обнаруженного совпадения с переменным списком параметров приводимого ниже содержания (напомним, что при глобальном поиске в исходной символьной строке обнаруживаются все экземпляры совпадения с шаблоном). А значение, возвращаемое данной функцией, служит в качестве заменяющего.

- Полный текст совпадения.
- Фиксации совпадений по каждому параметру.
- Индекс совпадения в исходной символьной строке.
- Исходная символьная строка.

Это дает немалую свободу действий для определения содержимого заменяющей символьной строки во время выполнения, причем бо́льшая часть этой информации зависит от характера искомого совпадения. В листинге 7.9 приведен пример кода, в котором функция предоставляет заменяющее значение в динамическом режиме для преобразования символьной строки со словами, разделенными тире, в эквивалентное им смешанное написание.

Листинг 7.9. Преобразование написанной через тире символьной строки в смешанное написание

```
<script type="text/javascript">

  function upper(all,letter) { return letter.toUpperCase(); }

  assert("border-bottom-width".replace(/-(\w)/g,upper)
         == "borderBottomWidth",
       "Camel cased a hyphenated string.");

</script>
```

Преобразовать в верхний регистр

Сопоставить с символами, написанными через тире

В приведенном выше примере кода предоставляется глобальное регулярное выражение, обеспечивающее совпадение с любым символом, которому предшествует знак тире. А фиксация в этом выражении обозначает совпавший символ (без тире). Всякий раз, когда вызывается функция `upper()`, а в данном случае это происходит дважды, ей передается полностью совпавшая символьная строка в качестве первого аргумента, а также фиксация (только один раз для этого регулярного выражения) в качестве второго аргумента. Остальные аргументы не важны, поэтому они и не указываются.

При первом вызове функции `upper()` ей передаются аргументы `"-b"` и `"b"`, а при втором вызове — аргументы `"-w"` и `"w"`. В каждом случае зафиксированная строчная буква преобразуется в прописную и возвращается в качестве заменяющей символьной строки. И в конечном итоге подстрока `"-b"` заменяется на `"B"`, а подстрока `"-w"` — на `"W"`.

Глобальное регулярное выражение обусловливает выполнение данной функции замены всякий раз, когда происходит совпадение в исходной символьной строке. Поэтому рассмотренный здесь способ можно даже расширить за пределы обычной замены, чтобы использовать его в качестве средства для прохождения символьных строк и своего рода альтернативы типичному применению метода `exec()` в цикле `while`, как было показано ранее в этой главе.

Допустим, требуется преобразовать строку запроса `"foo=1&foo=2&blah=a&blah=b&foo=3"` в альтернативный формат `"foo=1,2,3&blah=a,b"`, в большей степени подходящий преследуемым целям. Для решения этой задачи можно воспользоваться регулярным выражением и методом `replace()` в коде, который получается особенно кратким, как показано в листинге 7.10.

Листинг 7.10. Способ сжатия строки запроса

```
<script type="text/javascript">

  function compress(source) {
    var keys = {};
    source.replace(
      /([^=&]+)=([^&]*)/g,
      function(full, key, value) {
        keys[key] =
          (keys[key] ? keys[key] + "," : "") + value;
       return "";
      }
    );

    var result = [];
    for (var key in keys) {
      result.push(key + "=" + keys[key]);
    }

    return result.join("&");
  }

  assert(compress("foo=1&foo=2&blah=a&blah=b&foo=3") ==
        "foo=1,2,3&blah=a,b",
        "Compression is OK!");

</script>
```

❶ Сохранить расположенные ключи

❷ Извлечь данные о ключе и значении

❸ Собрать данные о ключе

❹ Соединить результаты знаком &

Самое любопытное в коде из листинга 7.10 состоит в том, что метод `replace()` замены символьной строки служит скорее в качестве средства прохождения этой строки для обнаружения отдельных значений, а не механизма поиска и замены. Такой прием преследует две цели: передать функцию в качестве аргумента заменяющего значения при вызове метода `replace()` и просто использовать ее для поиска вместо возврата значения.

Сначала в рассматриваемом здесь примере кода объявляется информационный массив для хранения ключей и значений, обнаруживаемых в исходной строке запроса ❶. Затем для исходной строки вызывается метод `replace()` ❷, которому передается регулярное выражение для поиска на совпадение пар "ключ–значение" и фиксации ключа и значения, а также функция, принимающая в качестве своих аргументов результат полного совпадения, зафиксированный ключ и зафиксированное значение. Эти зафиксированные величины сохраняются в информационном массиве для последующего обращения к ним. Следует заметить, что из этой функции возвращается пустая символьная строка, поскольку в данном случае используются побочные эффекты, а не сам результат замены в исходной строке, который особого значения не имеет.

После возврата из метода `replace()` объявляется обычный массив, в котором будут накапливаться результаты, и далее организуется циклическое обращение к обнаруженным ключам, в ходе которого каждый ключ вводится в данный массив ❸. И наконец, отдельные результаты, хранящиеся в массиве, соединяются разграничительным знаком & и возвращается полученный результат ❹.

Подобным способом метод `replace()` из класса `String` можно использовать в качестве самостоятельного механизма поиска в символьных строках, добиваясь результата не только быстро, но и просто и эффективно. Потенциальные возможности такого способа трудно переоценить, особенно если учесть, что для этого требуется совсем немного кода.

На самом деле регулярные выражения могут оказывать заметное влияние на порядок написания сценариев для веб-страниц. Покажем теперь, как применить знания,

полученные о регулярных выражениях, для решения типичных задач, которые могут возникнуть при разработке веб-приложений.

Решение типичных задач с помощью регулярных выражений

В связи с различными способами выражения на языке JavaScript постоянно возникает ряд затруднений, но их разрешение не всегда оказывается очевидным. И здесь на помощь может прийти мастерское владение регулярными выражениями. В этом разделе будут рассмотрены некоторые типичные задачи, которые можно решить с помощью одного или двух регулярных выражений.

Обрезка символьных строк

Потребность в удалении лишних пробелов в начале и конце символьной строки возникает нередко, но до последнего времени необходимые для этого средства в классе `String` отсутствовали. Почти в каждой библиотеке JavaScript предоставляется реализация обрезки символьных строк для прежних версий браузеров, где отсутствует поддержка метода `String.trim()`. Поэтому чаще всего применяется подход, аналогичный приведенному в коде из листинга 7.11.

Листинг 7.11. Общее решение задачи удаления лишних пробелов из символьной строки

```
<script type="text/javascript">

  function trim(str) {
    return (str || "").replace(/^\s+|\s+$/g, "");
  }

  assert(trim(" #id div.class ") == "#id div.class",
        "Extra whitespace trimmed from a selector string.");

</script>
```

Обрезка символьной строки без организации цикла

Обратите внимание на то, что в коде приведенного выше примера нет никаких циклов! Вместо циклического обращения к символам строки для определения тех из них, которые требуется удалить, строка обрезается в результате единственного вызова метода `replace()` с регулярным выражением, обеспечивающим совпадение с символами пробела в начале или конце строки.

Стивен Левитан, один из авторов книги *Regular Expressions Cookbook* (издательство O'Reilly, 2009 г.), упоминавшейся в начале этой главы, провел обширные исследования по данному вопросу, найдя целый ряд альтернативных его решений. Подробнее о них можно узнать, посетив его блог *Flagrant Badassery* (Обалденная крутизна) по адресу `http://blog.stevenlevithan.com/archives/faster-trim-javascript`. Следует, однако, заметить, что его контрольные примеры пригодны для очень крупных документов, что, безусловно, является крайним случаем для большинства приложений.

Два из этих решений вызывают особый интерес. Первое из них реализуется с помощью регулярных выражений, но без операторов `\s+` и `|`, как показано в листинге 7.12.

Листинг 7.12. Альтернативный способ обрезки символьной строки путем двойной замены

```
<script type="text/javascript">

  function trim(str) {
    return str.replace(/^\s\s*/, '')
              .replace(/\s\s*$/, '');
  }

  assert(trim(" #id div.class ") == "#id div.class",
        "Extra whitespace trimmed from a selector string.");

</script>
```

Обрезать, используя две замены

В данной реализации функции обрезки символьной строки выполняются две замены: одна — для начального пробела, другая — для конечного. Во втором решении лишние пробелы в конце символьной строки удаляются вручную и даже не предпринимается попытка сделать это с помощью регулярного выражения, как показано в листинге 7.13.

Листинг 7.13. Способ обрезки символьной строки усечением ее конца вручную

```
<script type="text/javascript">

  function trim(str) {
    var str = str.replace(/^\s\s*/, ''),
        ws = /\s/,
        i = str.length;
    while (ws.test(str.charAt(--i)));
    return str.slice(0, i + 1);
  }

  assert(trim(" #id div.class ") == "#id div.class",
        "Extra whitespace trimmed from a selector string.");

</script>
```

Обрезать, используя регулярное выражение и усечение

В данной реализации функции обрезки символьной строки начальный пробел обрезается с помощью регулярного выражения, а конечный пробел — в ходе операции усечения. Если сравнить производительность всех этих реализаций функции обрезки `trim()` на примерах коротких и длинных символьных строк типа документа, то отличия сразу же бросятся в глаза. В табл. 7.2 приведено для сравнения время (в миллисекундах) выполнения функции `trim()` в трех упомянутых выше реализациях.

Таблица 7.2. Сравнение производительности функции `trim()` в трех ее реализациях

Реализация обрезки символьной строки	Короткая строка	Документ
Листинг 7.11	8,7 мс	2075,8 мс
Листинг 7.12	8,5 мс	3706,7 мс
Листинг 7.13	13,8 мс	169,4 мс

Такое сравнение позволяет легко выяснить, какая именно реализация функции обрезки символьной строки оказывается наиболее пригодной для расширения. Несмотря на то что реализация в коде из листинга 7.13 уступает по производительности двум другим реализациям при обрезке коротких символьных строк, она оставляет их далеко позади при обрезке довольно длинных строк типа документов. В конечном счете выбор той или иной реализации зависит от конкретной ситуации, в которой должна выполняться обрезка символьной строки. В большинстве библиотек используется первое из рассмотренных выше решений. Оно лучше всего подходит для обрезки коротких символьных строк и, по-видимому, является наиболее надежным с точки зрения совместимости с устаревшими версиями браузеров. А теперь перейдем к следующей типичной задаче.

Совпадение концов строк

При выполнении поиска нередко требуется, чтобы член . (точка), который обычно совпадает с любым символом, кроме перевода строки, охватывал бы и символы новой строки. Для этой цели в реализации регулярных выражений на других языках программирования зачастую включается дополнительный флажок, а в JavaScript этого сделать нельзя. Поэтому рассмотрим два способа возмещения этого явного упущения в JavaScript, как показано в листинге 7.14.

Листинг 7.14. Совпадение *всех* символов, в том числе и конца строки

```
<script type="text/javascript">

  var html = "<b>Hello</b>\n<i>world!</i>";

  assert(/.*/.exec(html)[0] === "<b>Hello</b>",
         "A normal capture doesn't handle endlines.");
  assert(/[\S\s]*/.exec(html)[0] ===
        "<b>Hello</b>\n<i>world!</i>",
        "Matching everything with a character set.");
  assert(/(?:.|\s)*/.exec(html)[0] ===
        "<b>Hello</b>\n<i>world!</i>",
        "Using a non-capturing group to match everything.");

</script>
```

❶ Определить объект для тестирования

❷ Показать, что совпадения с символами новой строки не произошло

❸ Использовать сопоставление с пробелами для совпадения со всеми символами

❹ Использовать нередование для совпадения со всеми символами

Сначала в данном примере определяется символьная строка, служащая в качестве объекта тестирования и содержащая знак перевода строки ❶. Затем предпринимается попытка обнаружить разными способами совпадение со всеми символами в строке. В первом тесте совпадение со знаками перевода строки оператором точки (.) не обнаруживается ❷.

Но настоящие мастера своего дела не отступают перед трудностями, поэтому в следующем тесте для проверки используется альтернативный вариант регулярного выражения, /[\S\s]*/, в котором определяется класс символов для совпадения с любыми символами, которые *не* являются пробелами, а также с любыми символами, которые *относятся* к пробельным ❸. Такое сочетание охватывает весь набор символов.

Еще одна попытка предпринимается в следующем тесте, где используется регулярное выражение /(?:.|\s)*/, обеспечивающее с помощью оператора точки (.) совпадение со всеми символами, кроме перевода строки, а также с любыми пробельными символами, включая и перевод строки ❹. Такое сочетание охватывает весь набор символов, в том числе и знак перевода строки. Обратите внимание на использование в данном случае пассивного подвыражения, исключающего любые нежелательные фиксации.

Очевидно, что самым оптимальным оказывается решение с помощью регулярного выражения /[\S\s]*/ благодаря его простоте и подразумеваемым преимуществам в быстродействии. А теперь сделаем смелый шаг для выхода на мировую арену.

Уникод

Нередко в выражении требуется обнаружить совпадение буквенно-цифровых символов для последующего применения (например, селектор идентификатора в реализации механизма CSS-селекторов). Но было бы недальновидно предполагать наличие среди буквенно-цифровых символов только букв английского алфавита. В этой связи целесообразно расширить набор символов до уникода (Unicode), чтобы поддерживать явным образом многие языки мира, не охватываемые традиционным набором буквенно-цифровых символов, как показано в листинге 7.15.

Листинг 7.15. Проверка на совпадение с символами уникода

```
<script type="text/javascript">

  var text ="\u5FCD\u8005\u30D1\u30EF\u30FC";

  var matchAll =
    /[\w\u0080-\uFFFF_-]+/;

  assert((text).match(matchAll),
        "Our regexp matches unicode!");

</script>
```

Сопоставить со всеми символами, в том числе и в уникоде

В приведенном выше примере кода проверка на совпадение охватывает все символы уникода, для чего создается класс символов, включающий член \w для сопоставления со всеми обычными символами с кодом до **128** в десятичной системе и **0x80** в шестнадцатеричной системе счисления, а также диапазон, охватывающий весь набор символов уникода с десятичным кодом свыше **128** (или **0x80** в шестнадцатеричной форме). Начиная с десятичного кода **128**, обнаруживаются некоторые символы из верхней половины набора в коде ASCII и все символы в уникоде.

Проницательные читатели могли заметить, что введением в регулярное выражение всего диапазона символов уникода свыше кода **\u0080** в данном примере обеспечивается совпадение не только с буквенными символами, но и со всеми знаками препинания и другими специальными символами уникода (например, стрелками). В этом нет никакой погрешности, поскольку основное назначение данного примера — показать, как вообще осуществляется проверка на совпадение с символами уникода. Если же у вас имеется какой-то определенный диапазон символов для сопоставления, вы можете воспользоваться данным примером, чтобы ввести любой нужный вам диапазон в класс символов. И в заключение нашего исследования потенциальных возможностей регулярных выражений рассмотрим еще одну весьма распространенную задачу.

Экранированные символы

Авторы веб-страниц часто пользуются именами, совпадающими с идентификаторами программ, присваивая значения `id` элементам разметки страницы, поскольку так принято. Но ведь значения `id` могут содержать не только обычные символы, но и знаки препинания. В частности, разработчик может присвоить элементу разметки значение `id`, равное `form:update`.

Реализуя, например, механизм CSS-селекторов, разработчик библиотеки может организовать поддержку знаков препинания, применяя экранирование символов. Это дает пользователю возможность указывать сложные имена, не согласующиеся с принятыми условными обозначениями. Рассмотрим для примера код, приведенный в листинге 7.16.

Листинг 7.16. Проверка на совпадение с экранированными символами в CSS-селекторе

```
<script type="text/javascript">

  var pattern = /^((\w+)|(\\.))+$/;

  var tests = [
    "formUpdate",
    "form\\.update\\.whatever",
    "form\\:update",
    "\\f\\o\\r\\m\\u\\p\\d\\a\\t\\e",
    "form:update"
  ];

  for (var n = 0; n < tests.length; n++) {
    assert(pattern.test(tests[n]),
           tests[n] + " is a valid identifier" );
  }

</script>
```

Это регулярное выражение обеспечивает совпадение с любой последовательностью, состоящей из буквенных символов, обратной косой черты и любого символа или того и другого

Установить различные тестируемые объекты. Все они должны пройти тест, кроме последнего, где не удается экранировать специальный знак (:)

Проверить все тестируемые объекты

В данном конкретном выражении обнаруживается совпадение с последовательностью буквенных символов или последовательностью символов обратной косой черты и любых других символов.

Резюме

Подведем итог тому, что было рассмотрено в этой главе.

- Регулярные выражения являются весьма эффективным средством, все чаще проникающим буквально во все аспекты разработки современных веб-приложений на JavaScript, обеспечивая любую проверку на совпадение в зависимости от их конкретного применения. Хорошо разбираясь в основных принципах составления и применения регулярных выражений, рассмотренных в этой главе, любой разработчик должен уверенно чувствовать себя, преодолевая трудности при написании тех фрагментов кода, где можно выгодно воспользоваться регулярными выражениями.

- Сначала в этой главе были рассмотрены различные члены и операторы и продемонстрированы способы их объединения при составлении регулярных выражений для сопоставления с шаблоном.
- Затем было показано, каким образом осуществляется предварительная компиляция регулярных выражений и какие огромные преимущества в производительности это дает по сравнению с необходимостью перекомпилировать регулярные выражения всякий раз, когда это требуется.
- Далее было пояснено, как пользоваться регулярным выражением для проверки символьной строки на совпадение с шаблоном, представленным в этом выражении, а самое главное – как фиксировать отдельные совпавшие части исходной символьной строки.
- После этого было продемонстрировано, как пользоваться такими полезными методами обработки регулярных выражений, как `exec()`, а также ориентированными на них методами `match()` и `replace()` из класса `String`. Попутно были разъяснены отличия локальных регулярных выражений от глобальных.
- Кроме того, было показано, как пользоваться зафиксированными результатами сопоставления с шаблоном для обратной ссылки и заменяющими строками и как исключать лишние фиксации с помощью пассивных подвыражений.
- Далее был рассмотрен механизм, обеспечивающий динамическое определение в функции заменяющей строки. И в заключение этой главы были представлены решения типичных задач с помощью регулярных выражений, включая обрезку символьных строк и проверку на совпадение со знаками перевода строки и символами уникода.

Итак, регулярные выражения относятся к тем эффективным средствам, которые должен взять на вооружений всякий, стремящийся стать мастером программирования на JavaScript. В самом начале главы 3 упоминалось о цикле ожидания событий и подчеркивалось, что в JavaScript все обратные вызовы для обработки событий происходят по очереди в одном потоке. А в следующей главе мы продолжим исследование организации поточной обработки в JavaScript, подробно обсудив ее влияние на таймеры и интервалы.

8 Укрощение потоков и таймеров

В этой главе...

- Организация поточной обработки в JavaScript
- Исследование порядка выполнения таймеров
- Обработка крупных заданий с помощью таймеров
- Управление анимацией с помощью таймеров
- Усовершенствование тестирования с помощью таймеров

Таймеры относятся к недостаточно ясно понимаемым и зачастую недооцененным средствам JavaScript, которые дают немало преимуществ в разработке сложных приложений, если они используются надлежащим образом. Следует, однако, иметь в виду, что таймеры рассматриваются здесь как средство, доступное в JavaScript, хотя сами они *не* относятся непосредственно к JavaScript. Напротив, таймеры предоставляются как часть объектов и методов, которая становится доступной в веб-браузере. Это означает, что если применять JavaScript в другой среде, отличающейся от браузеров, то таймеры, скорее всего, будут в ней отсутствовать. Поэтому в такой среде придется реализовать собственную версию таймеров с помощью подходящих для этой цели инструментальных средств, как, например, Threads в Rhino.

Таймеры позволяют асинхронно задерживать выполнение определенного фрагмента кода на заданное количество миллисекунд. По своему характеру JavaScript является языком однопоточного программирования (т.е. одновременно допускается выполнение только одного фрагмента кода JavaScript), но таймеры позволяют обойти это ограничение, чтобы выполнить код косвенным путем. В этой главе будет показано, как это осуществляется на практике.

Примечание

По мере широкого распространения стандарта HTML5 многое в отношении поточной обработки изменится, хотя в современных браузерах этого пока еще не происходит. Именно поэтому очень важно ясно понимать, каким образом поточная обработка и таймеры поддерживаются в современных браузерах.

Принцип действия таймеров и поточной обработки

В силу явной полезности таймеров очень важно понимать принцип их действия хотя бы на самом элементарном уровне. На первый взгляд, таймеры порой ведут себя неочевидно из-за того, что они выполняются в одном потоке, тогда как многие программирующие привыкли к тому, что таймеры действуют в многопоточной среде. Последствия ограничений на однопоточную обработку в JavaScript будут исследованы несколько ниже, а сначала рассмотрим методы, которыми можно воспользоваться для построения таймеров и манипулирования ими.

Установка и очистка таймеров

Для создания таймеров в JavaScript предоставляются два метода и столько же методов для их очистки или удаления. Все эти методы относятся к объекту `window`, служащему в качестве глобального контекста. В табл. 8.1 приводится их краткое описание.

Таблица 8.1. Методы манипулирования таймерами в JavaScript (все они относятся к объекту `window`)

Метод	Формат	Описание
`setTimeout()`	`id = setTimeout(fn,delay)`	Инициирует таймер, выполняющий передаваемую ему функцию обратного вызова по истечении времени задержки. Возвращает значение, однозначно обозначающее таймер
`clearTimeout()`	`clearTimeout(id)`	Удаляет (очищает) таймер, обозначаемый передаваемым значением, если таймер еще не запущен
`setInterval()`	`id = setInterval(fn,delay)`	Инициирует таймер на постоянное выполнение передаваемой функции обратного вызова через указанный интервал задержки вплоть до отмены. Возвращает значение, однозначно обозначающее таймер
`clearInterval()`	`clearInterval(id)`	Удаляет (очищает) интервальный таймер, обозначаемый передаваемым значением

Упомянутые выше методы позволяют устанавливать и очищать таймеры, которые запускаются однократно или периодически через указанные интервалы времени. На практике в большинстве браузеров допускается очищать таймер одним из методов — `clearTimeout()` или `clearInterval()`. Тем не менее ради ясности кода эти методы рекомендуется использовать согласованными парами.

Что касается применения таймеров в JavaScript, то следует особо подчеркнуть, что задержка срабатывания таймера не гарантируется. И объясняется это главным образом характером организации поточной обработки в JavaScript. Поэтому рассмотрим, как эта обработка осуществляется.

Выполнение таймера в потоке исполнения

В браузере весь код JavaScript пока еще выполняется в одном и только одном потоке, если, конечно, в будущем не будет предусмотрено иное. Неизбежным следствием этого является то обстоятельство, что обработчики асинхронных событий, в том числе интерфейсных и таймеры, выполняются лишь после того, как ничего больше не останется для выполнения. Это означает, что обработчики событий должны выстраиваться в очередь на выполнение, как только появится для этого удобный момент, и что ни один из обработчиков не может прервать выполнение другого. Это положение наглядно демонстрируется на временной диаграмме, приведенной на рис. 8.1.

Рис. 8.1. Временная диаграмма, наглядно демонстрирующая, каким образом в одном потоке выполняется основной код и обработчики событий

Из временной диаграммы на рис. 8.1 можно извлечь немало полезного, но только ясное представление о том, что именно она демонстрирует, позволит правильно понять принцип действия асинхронного выполнения кода JavaScript. Эта диаграмма одномерна и развертывает слева направо время выполнения кода, указываемое в миллисекундах. В отдельных прямоугольниках указаны исполняемые блоки кода JavaScript, а их ширина соответствует времени выполнения этих кодовых блоков. В частности, первый блок основного кода JavaScript выполняется около 18 мс, блок обработки события от щелчка кнопкой мыши — около 10 мс и т.д.

Вследствие того что JavaScript является языком однопоточного программирования, блоки его кода могут выполняться лишь по очереди. Поэтому каждый блок исполняемого кода блокирует обработку других асинхронно наступающих событий. А это означает,

что когда наступает асинхронное событие (вроде щелчка кнопкой мыши, срабатывания таймера или завершения запроса типа `XMLHttpRequest`), оно ставится в очередь на последующую обработку, как только освободится поток. Конкретный способ организации такой очереди обработки событий на самом деле зависит от типа браузера, поэтому здесь она рассматривается в несколько упрощенном виде. Хотя и этого должно быть достаточно, чтобы понять сам принцип организации поточной обработки в JavaScript.

Начиная с нулевого момента времени, в течение выполнения первого блока кода JavaScript, которое занимает около 18 мс, происходит ряд важных событий, перечисленных ниже.

- В момент времени **0** мс запускается таймер блокировки по времени с задержкой срабатывания на **10** мс, а также интервальный таймер с такой же точно задержкой срабатывания.
- В момент времени **6** мс производится щелчок кнопкой мыши.
- В момент времени **10** мс срабатывает таймер блокировки по времени и оканчивается первый интервал времени.

Если в текущий момент код не выполняется, то в обычных условиях можно ожидать, что обработчик событий от щелчков кнопкой мыши начнет выполняться сразу же в момент времени **6** мс, а обработчики обоих таймеров — по истечении времени ожидания и их срабатывания в момент времени 10 мс. Но ни один из этих обработчиков не может начать свое выполнение, поскольку не завершился еще исходный блок кода. В силу однопоточного характера JavaScript эти обработчики становятся в очередь на выполнение в следующий доступный момент времени.

По завершении исходного кодового блока в момент времени **18** мс в очереди на выполнение находятся три кодовых блока: обработчик событий от щелчков кнопкой мыши, обработчик блокировки по времени и вызываемый первый раз обработчик интервалов времени. Предполагается, что очередь в браузере организуется по принципу "первым пришел — первым обслужен", но не следует забывать, что в браузере может быть выбран и более сложный алгоритм. Это означает, что первым из ожидающих в очереди начнет выполнение обработчик событий от щелчков кнопкой мыши, на что может уйти до 10 мс.

Во время выполнения обработчика блокировки по времени наступает момент времени **20** мс, когда истекает второй интервал. И в этот момент обработчик интервалов снова не может быть выполнен, поскольку поток занят выполнением обработчика блокировки по времени. Но на этот раз экземпляр функции обратного вызова уже находится в очереди, ожидая выполнения, и поэтому вызов этой функции пропускается. Следовательно, браузер не будет ставить в очередь больше одного экземпляра обработчика для одного и того же интервала.

Обработчик событий от щелчков кнопкой мыши завершает свою работу в момент времени 28 мс, и тогда, а не в момент времени **10** мс, как предполагалось, запускается на выполнение ожидающий обработчик блокировки по времени. Именно это имелось в виду, когда ранее утверждалось, что заданное время задержки будет учитываться при определении момента запуска обработчика на выполнение.

В момент времени **30** мс в очередной (третий) раз оканчивается интервал времени, но и на этот раз дополнительный экземпляр его обработчика не вводится в очередь, поскольку он уже там находится. А в момент времени **34** мс обработчик блокировки по времени завершает свою работу, и начинается выполнение ожидающего своей очереди обработчика интервалов времени, на что требуется 6 мс. И пока это происходит,

интервал истекает в очередной (четвертый) раз в момент времени **40** мс, что приводит к постановке вызова обработчика интервалов в очередь. По завершении его первого вызова в момент времени **42** мс начинает выполняться ожидающий своей очереди обработчик.

На этот раз обработчик интервалов времени завершает свою работу в момент времени **47** мс, т.е. раньше, чем следующий интервал истечет в момент времени **50** мс. Следовательно, по истечении интервала в пятый раз его обработчик уже не придется ставить в очередь, но можно запустить на выполнение, как только истечет данный интервал.

Из всего сказанного выше можно сделать следующий важный вывод: в силу однопоточного характера обработки в JavaScript одновременно может быть выполнен только один исполняемый блок, и поэтому нельзя гарантировать, что обработчики таймеров будут выполняться именно в тот момент, в какой и предполагалось. И это особенно касается обработчиков интервалов времени. Как следует из рассмотренного выше примера, из пяти запланированных запусков обработчика интервалов в моменты времени **10**, **20**, **30**, **40** и **50** мс были выполнены только три в моменты времени **35**, **42** и **50** мс. Как видите, интервалы времени требуют иного обращения, чем блокировки по времени. Поэтому рассмотрим подробнее их отличия.

Отличия интервалов от блокировок по времени

На первый взгляд интервал кажется похожим на блокировку по времени, поскольку и то и другое периодически повторяется. Но их отличия оказываются более глубокими. Для того чтобы проиллюстрировать эти отличия, а заодно и отличия в применении методов `setTimeout()` и `setInterval()`, обратимся к примеру кода, приведенному в листинге 8.1.

Листинг 8.1. Два способа создания повторяющихся таймеров

```
<script type="text/javascript">

  setTimeout(function repeatMe() {
    /* Длинный кодовый блок... */
    setTimeout(repeatMe, 10);
  }, 10);

  setInterval(function() {
    /* Длинный кодовый блок... */
  }, 10);

</script>
```

❶ Установить блокировку по времени с повторением через каждые 10 мс

❷ Установить интервал с перезапуском через каждые 20 мс

Оба фрагмента кода из листинга 8.1 могут показаться функционально равнозначными, но на самом деле это не так. В частности, вариант кода с методом `setTimeout()` ❶ будет всегда выполняться с задержкой как минимум на **10** мс после выполнения предыдущего обратного вызова (эта задержка может оказаться и большей, но не меньше **10** мс), тогда как попытка выполнить вариант кода с методом `setInterval()` ❷ будет предприниматься через каждые **10** мс независимо от того, когда был выполнен последний обратный вызов.

Как следует из примера, рассматривавшегося в предыдущем разделе, точно гарантировать момент обратного вызова обработчика блокировки по времени нельзя. Вместо перезапуска через каждые 10 мс подобно интервалу установка блокировки по времени на 10 мс повторяется после того, как будет выполнена предыдущая ее обработка.

Итак, подведем краткие итоги всему сказанному выше.

- У механизма JavaScript имеется только один поток, что вынуждает асинхронные события ожидать своей очереди на обработку.
- Если немедленное выполнение таймера блокируется, оно задерживается до следующего момента, удобного для его выполнения и наступающего не позднее указанной задержки.
- Интервалы времени могут обрабатываться по очереди без задержки, если они резервируются в достаточной степени, а в очереди размещается не больше одного экземпляра одного и того же обработчика интервалов времени.
- Методы `setTimeout()` и `setInterval()` принципиально отличаются способом определения частоты их запуска на выполнение.

Все это очень важно знать для правильного обращения с таймерами и интервалами. Ведь ясное понимание того, как механизм JavaScript обрабатывает асинхронный код, особенно при большом количестве асинхронных событий, что зачастую и случается на типичной веб-странице со сценарием, имеет решающее значение для написания эффективного прикладного кода.

В примерах, приведенных в этом разделе, использовались очень малые величины задержек, но даже при задержке на **10** мс нам удалось выяснить немало важных особенностей организации поточной обработки в JavaScript. А теперь попробуем выяснить, насколько оптимистичными могут быть эти величины, обратив особое внимание на степень детализации, с которой можно их указывать.

Минимальная задержка таймера и надежность

Если установка задержек таймера в секундах, минутах, часах или любого другого требуемого интервала времени кажется вполне очевидной, то этого нельзя сказать о самой минимальной задержке таймера, которую можно выбрать. В определенный момент браузер оказывается просто не в состоянии обеспечить достаточно высокую разрешающую способность таймеров, чтобы обрабатывать их задержки надлежащим образом. Ведь они сами зависят от временных характеристик операционной системы.

Еще несколько лет назад указывать задержки короче 10 мс было несерьезно и слишком оптимистично. Но за последнее время было сделано немало для усовершенствования поддержки сценариев JavaScript в браузерах, поэтому мы можем проверить, до какой степени допускается уменьшать устанавливаемые задержки. Итак, начнем с интервального таймера, установив для него задержку **1** мс и измерив фактическую задержку между интервалами вызова обработчика в течение первых 100 моментов срабатывания этого таймера. Результаты этих измерений приведены на рис. 8.2 и 8.3 в виде диаграмм, демонстрирующих, сколько раз из 100 попыток срабатывания было достигнута величина каждого интервала.

Под Mac OS X средняя величина задержки в браузере Firefox составила около **4** мс, хотя ее разброс оказался настолько большим, что на один интервал пришлась задержка **22** мс. Большее постоянство в этом отношении проявил браузер Chrome, средняя величина задержки в котором также составила около **4-5** мс, тогда как в браузере Safari она оказалась заметно большей, достигнув **10** мс. Самым быстродействующим оказался браузер Opera 11, где задержка на 56 интервалах из 100 составила заданную величину **1** мс.

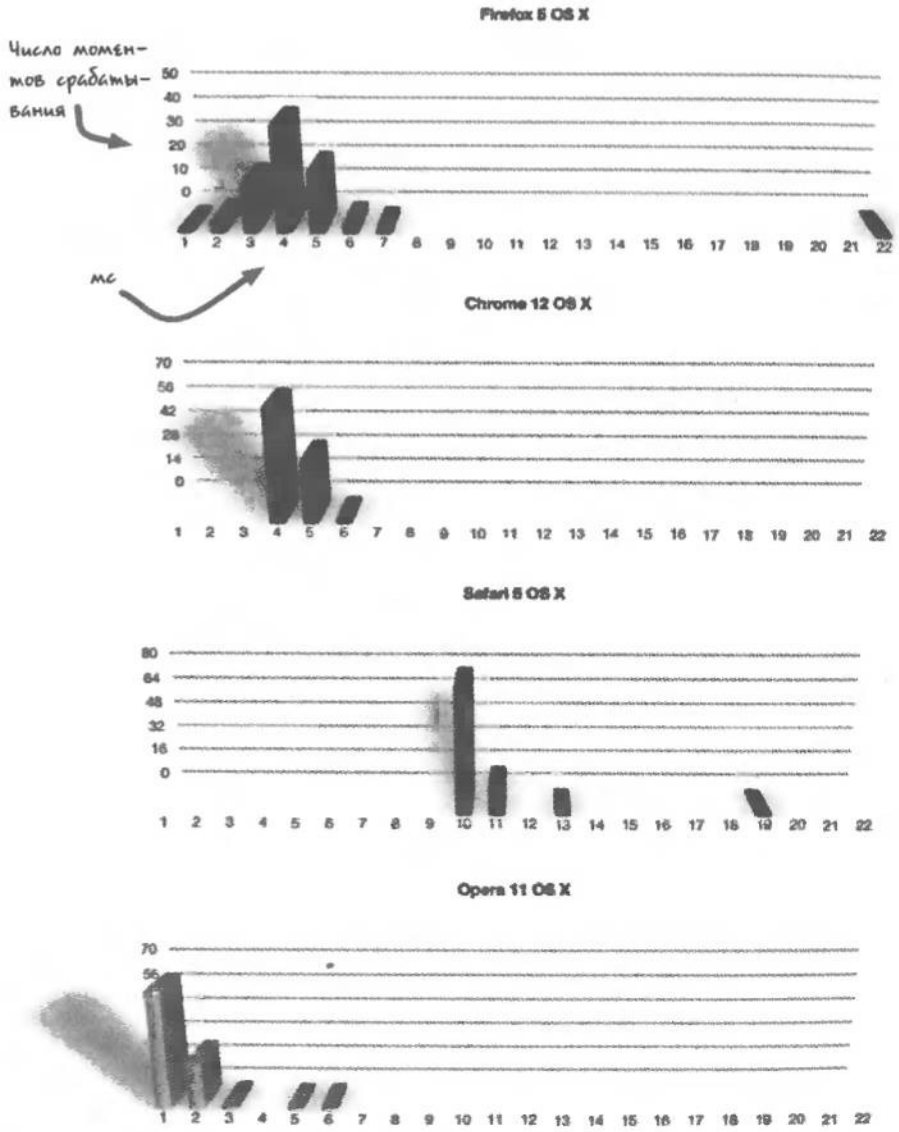

*Рис. 8.2. Производительность интервального таймера, измеренная в браузерах под Mac OS X, показывает, что степень детализации величины задержки в одних браузерах оказалась довольно близкой к **1** мс, тогда как в других – не такой близкой*

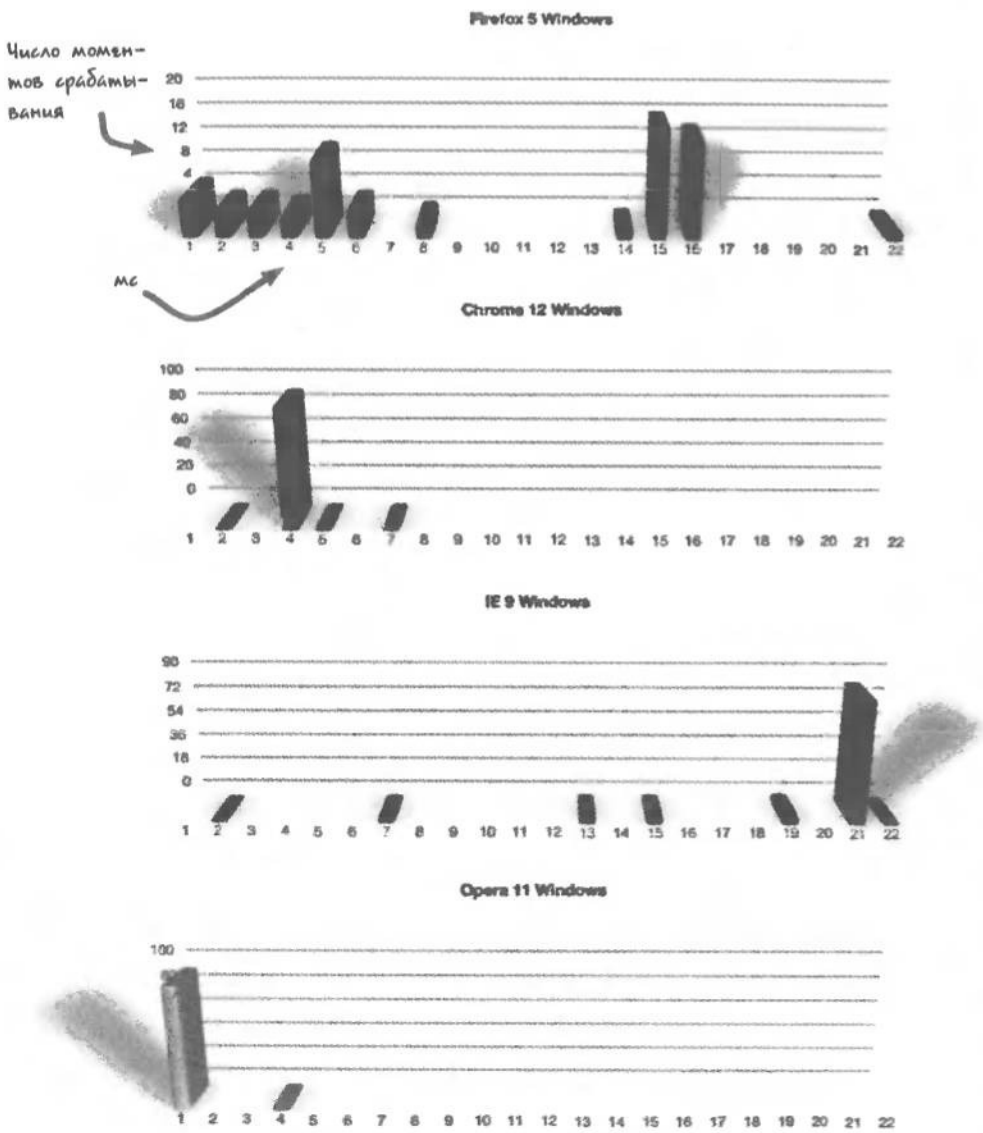

Рис. 8.3. Производительность интервального таймера, измеренная в браузерах под Windows, в целом сравнима с их производительностью под Mac OS X

Под Windows браузер Firefox снова проявил себя как самый непостоянный, а разброс результатов оказался настолько большим, что не дает ясной картины о средней величине задержки. В браузере Chrome средняя величина задержки составила **4** мс, тогда как в Internet Explorer – **9** мс, хотя и с незначительным отклонением до **21** мс. Браузер Opera снова одержал верх, обеспечив заданную величину задержки **1** мс во всех интервалах, кроме одного.

Примечание

Эти испытания были проведены на переносном компьютере MacBook Pro следующей конфигурации: процессор Intel Core 2 Duo с тактовой частотой 2,5 МГц, ОЗУ на 4 Гбайт и ОС Mac OS X 10.6.7, а также на переносном ПК такой конфигурации: процессор Intel Core Quad Q9550 с тактовой частотой 2,83 МГц, ОЗУ на 4 Гбайт и ОС Windows 7.

Из приведенных выше результатов испытаний можно сделать вывод, что современные браузеры в целом пока еще неспособны надежно и устойчиво достигать степени детализации величины интервальных задержек на уровне **1** мс. Хотя некоторые из них довольно близко приблизились к этому уровню.

В этих испытаниях была исходно установлена величина задержки **1** мс, хотя можно было бы указать и нулевую задержку в качестве наименее возможной. Впрочем, здесь имеется одна загвоздка: браузер Internet Explorer затрудняется обработать вызов метода `setInterval()` с нулевой задержкой. Всякий раз, когда такой вызов делается, обратный вызов по истечении заданного интервала выполняется только один раз, как если бы вместо метода `setInterval()` вызывался метод `setTimeout()`.

Из приведенных выше результатов можно сделать и ряд других выводов. Наиболее важный из них лишь подкрепляет то, что было выяснено ранее: браузеры не гарантируют точное соблюдение установленного интервала задержки. И хотя имеется возможность затребовать конкретные величины задержки, точность их соблюдения не всегда гарантируется, и особенно это касается малых величин задержки. Все это необходимо принимать во внимание, применяя таймеры при разработке веб-приложений. Если отличие в величинах задержки **10** и **15** мс имеет решающее значение или же если требуется еще более мелкая их детализация, то, возможно, имеет смысл пересмотреть принятый подход в связи с тем, что браузеры просто неспособны обеспечить согласование по времени именно на таком уровне.

Примечание

Как правило, для передачи данных функциям обратного вызова при срабатывании таймеров и истечении интервалов времени применяются замыкания. Но механизмы современных браузеров WebKit, Mozilla и Opera, кроме Internet Explorer любой версии, позволяют также передавать дополнительные аргументы при установочном вызове. Например, в результате вызова метода `setTimeout(callback,interval,arg1,arg2,arg3)` аргументы `arg1`, `arg2`, `arg3` будут переданы функции обратного вызова по истечении блокировки по времени.

Имея в виду все сказанное выше, покажем, как воспользоваться правильными представлениями о таймерах, чтобы избежать некоторых скрытых препятствий, связанных с производительностью.

Затратная по вычислениям обработка

Вероятно, самым трудным для преодоления скрытым препятствием при разработке сложного приложения на JavaScript является однопоточный характер программирования на этом языке. Вследствие этого взаимодействие исполняемого кода JavaScript с пользователем в лучшем случае замедляется, а в худшем – код вообще перестает реаги-

ровать на действия пользователя. В итоге браузер зависает и все обновления воспроизводимой страницы задерживаются на время выполнения кода JavaScript.

В силу этого обстоятельства разделение всех сложных операций продолжительностью более нескольких сотен миллисекунд на поддающиеся управлению части становится насущной потребностью. Кроме того, некоторые браузеры выводят диалоговое окно, предупреждающее пользователя о том, что сценарий перестал "реагировать", если он выполнялся безостановочно не менее 5 секунд. Такая особенность характерна для браузеров Firefox и Opera. Аналогично безостановочное выполнение любого сценария в течение более 5 секунд на платформе iPhone приводит к его удалению, причем без всяких предупреждений.

Когда родня встречается по тому или иному поводу, среди собравшихся нередко находится какой-нибудь особенно разговорчивый родственник, который без умолку рассказывает всем одни и те же семейные истории. И если кто-нибудь не остановит его и не ввернет во время словечко, общий разговор перестанет быть приятным для всех, разумеется, кроме самого этого словоохотливого родственника. Аналогичная ситуация возникает и с кодом, выполнение которого занимает все время обработки, в результате чего интерфейс приложения слабо реагирует на действия пользователя, что, естественно, никуда не годится. Тем не менее не исключены и такие ситуации, в которых приходится обрабатывать большие массивы данных, как, например, при манипулировании тысячами элементов модели DOM.

И в подобных случаях особенно полезными оказываются таймеры. Ведь они способны эффективно задерживать выполнение фрагмента кода JavaScript до более подходящего момента, а также разбивать отдельные фрагменты кода на более мелкие части, выполнение которых не должно привести к зависанию браузера. Принимая все это во внимание, обычные циклы и интенсивно выполняющиеся операции можно преобразовать в неблокирующие операции. В качестве примера рассмотрим код из листинга 8.2, где выполнение задания, скорее всего, займет немало времени.

Листинг 8.2. Длительное задание

```
<table><tbody></tbody></table>

<script type="text/javascript">

  var tbody = document.getElementsByTagName("tbody")[0];

  for (var i = 0; i < 20000; i++) {
    var tr = document.createElement("tr");
    for (var t = 0; t < 6; t++) {
      var td = document.createElement("td");
      td.appendChild(document.createTextNode(i + "," + t));
      tr.appendChild(td);
    }

    tbody.appendChild(tr);
  }

</script>
```

Найти элемент разметки <tbody>, для которого предполагается создать крупное множество строк

Создать 20 тыс. строк как крупное множество

Создать отдельную строку

Создать по шесть ячеек на строку с текстовым узлом в каждой

Присоединить новую строку к ее родительскому элементу

В данном примере создается в общем 240 тыс. узлов модели DOM, заполняющих таблицу большим числом ячеек. Это невероятно затратная по вычислениям операция, которая, скорее всего, приведет к зависанию браузера, препятствуя нормальному взаимодействию с пользователем. Ее можно сравнить со словоохотливым родственником, которого только и слышно в общем разговоре на семейном собрании. В подобной ситуации требуется каким-то образом заставить разговорчивого родственника замолчать на некоторое время, чтобы дать другим родственникам возможность присоединиться к разговору. По аналогии в код вводятся таймеры, чтобы прервать выполнение длительного задания, как показано в листинге 8.3.

Листинг 8.3. Прерывание длительного задания с помощью таймера

```
<script type="text/javascript">

  var rowCount = 20000;
  var divideInto = 4;
  var chunkSize = rowCount/divideInto;
  var iteration = 0;

  var table = document.getElementsByTagName("tbody")[0];

  setTimeout(function generateRows(){
    var base = (chunkSize) * iteration;
    for (var i = 0; i < chunkSize; i++) {
      var tr = document.createElement("tr");
      for (var t = 0; t < 6; t++) {
        var td = document.createElement("td");
        td.appendChild(
          document.createTextNode((i + base) + "," + t +
                                  "," + iteration));
        tr.appendChild(td);
      }
      table.appendChild(tr);
    }
    iteration++;
    if (iteration < divideInto)
      setTimeout(generateRows,0);
  },0);

</script>
```

❶ Подготовить исходные данные

❷ Продолжить вычисление с прежнего места

❸ Запланировать следующую стадию вычислений

В данном примере видоизмененного кода длительная операция разделена на чеыре более мелких операции, в каждой их которых создается своя доля узлов модели DOM. Эти операции с намного меньшей вероятностью приведут к прерыванию нормальной работы браузера. Рассматриваемый здесь код организован таким образом, что значения данных, управляющие всей операцией, накапливаются в легко настраиваемых переменных ❶ на тот случай, если потребуется разбить операцию, скажем, на десять частей, а не на четыре. Кроме того, для отслеживания того места, на котором была остановлена предыдущая стадия вычислений, требуется немного математических расчетов ❷. А следующая стадия вычислений планируется автоматически до тех пор, пока они не завершатся полностью ❸.

Но самое замечательное, что для применения рассматриваемого здесь нового асинхронного подхода к выполнению длительного задания требуется совсем немного кода. Затратив еще немного труда, можно организовать контроль за ходом операции, обеспечить правильный порядок ее выполнения и запланировать его по частям. А в остальном основная часть кода мало чем отличается от первоначального варианта из предыдущего листинга.

С точки зрения удобств для пользователя рассматриваемый здесь подход к выполнению длительного задания отличается от исходного еще и тем, что продолжительное прерывание работы браузера теперь заменяется четырьмя наглядными обновлениями страницы, хотя их может быть и больше. Пытаясь выполнить более мелкие фрагменты кода как можно быстрее, браузер будет также воспроизводить изменения в модели DOM после каждого цикла работы таймера. А в первоначальном варианте рассматриваемого здесь кода требовалось ждать единого массового обновления. Чаще всего такие обновления происходят незаметно для пользователя, но об этом все же не следует забывать. И поэтому нужно стремиться к тому, чтобы любой код, внедряемый на веб-странице, не прерывал ощутимо нормальную работу браузера.

Данный способ особенно пригодился одному из авторов книги при разработке веб-приложения для составления учебного плана занятий студентов в колледже. Первоначальное приложение имело типичный общий шлюзовой интерфейс CGI для связи клиента с сервером, где учебные планы составлялись и отправлялись обратно клиенту. Но это приложение пришлось переделать, чтобы перенести составление учебных планов на сторону клиента. На рис. 8.4 приведен вид экрана для составления учебных планов.

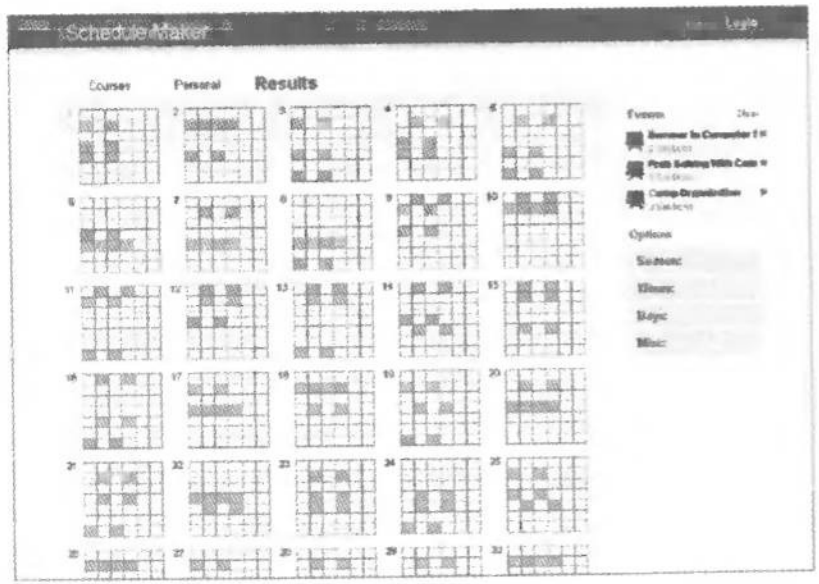

Рис. 8.4. Веб-приложение для составления учебного плана занятий со всеми расчетами на стороне клиента

Операции по составлению учебных планов были достаточно затратными по вычислениям, поскольку для получения правильного результата приходилось вносить тысячи изменений в учебные планы. Возникавшие в итоге затруднения в отношении производительности удалось разрешить, разделив группы плановых вычислений на вполне осязаемые фрагменты, после выполнения которых пользовательский интерфейс соответственно обновлялся. В конечном итоге пользователь получил в свое распоряжение удобный, практичный и быстро реагирующий на его действия интерфейс.

Просто удивительно, насколько полезным может быть рассматриваемый здесь способ. Он находит широкое применение в таких длительных процессах, как тестовые наборы, о которых речь пойдет в конце этой главы. Но самое главное, что данный способ демонстрирует, насколько просто обойти ограничения, накладываемые средой браузера, пользуясь языковыми средствами JavaScript, чтобы обеспечить удобство взаимодействия с конечным пользователем. Но и у роз имеются шипы. Ведь обращаться с большим количеством таймеров не так-то просто. Поэтому попробуем далее найти подходящий выход из этого затруднительного положения.

Центральное управление таймерами

При использовании большого количества таймеров возникает проблема управления ими. Это особенно актуально для анимации, когда приходится одновременно манипулировать большим количеством свойств и нужно найти способ как-то справиться со всем этим.

Управление многими таймерами затруднительно по целому ряду причин. Ведь нужно не только сохранять ссылки на многие интервальные таймеры, а рано или поздно их придется удалить, хотя это затруднение можно разрешить с помощью замыканий, но и вмешиваться в нормальную работу браузера. Как было показано ранее, если исключить чрезмерно долгое выполнение операций при вызове любого обработчика таймеров, то можно предотвратить блокирование других операций. Хотя при этом необходимо учитывать и другие особенности работы браузеров. К их числу относится "сборка мусора".

Одновременный запуск большого количества таймеров, скорее всего, приведет к инициированию задания на "сборку мусора" в браузере. Это примерно означает, что браузер попытается навести порядок в оперативной памяти, удалив из нее ненужные переменные, объекты и прочее. Обращение с таймерами вызывает особые трудности потому, что управление ими обычно осуществляется вне обычного хода работы однопоточного механизма JavaScript, т.е. из других потоков браузера.

Одни браузеры лучше справляются с данной ситуацией, чем другие, и от этого зависит продолжительность циклов "сборки мусора". Об этом можно судить по тому, насколько плавно анимация воспроизводится в одних браузерах и прерывисто – в других. Устранить подобные недостатки помогает сокращение числа одновременно действующих таймеров. Именно поэтому во всех современных механизмах анимации применяется способ, называемый *центральным управлением таймерами*.

Наличие центрального управления таймерами доставляет немало удобств и возможностей, включая следующие.

- Достаточно, чтобы на каждой странице одновременно действовал только один таймер.
- Работу таймеров можно по желанию приостанавливать и возобновлять.
- Процесс удаления функций обратного вызова существенно упрощается.

Рассмотрим пример, в котором данный способ применяется для управления несколькими функциями, осуществляющими анимацию отдельных свойств. Для этой цели в данном примере создается специальная конструкция, позволяющая управлять

функциями-обработчиками с помощью единственного таймера, как показано в листинге 8.4.

Листинг 8.4. Центральное управление таймерами для манипулирования несколькими обработчиками

```
<script type="text/javascript">

  var timers = {                          ❶ Объявить объект для управления таймером

    timerID: 0,
    timers: [],                           ❷ Записать состояние

    add: function(fn) {                   ❸ Создать функцию для ввода обработчиков
      this.timers.push(fn);
    },

    start: function() {                   ❹ Создать функцию для запуска таймера
      if (this.timerID) return;
      (function runNext() {
        if (timers.timers.length > 0) {
          for (var i = 0; i < timers.timers.length; i++) {
            if (timers.timers[i]() === false) {
              timers.timers.splice(i,1);
              i--;
            }
          }
          timers.timerID = setTimeout(runNext, 0);
        }
      })();
    },

    stop: function() {                    Создать функцию для остановки таймера
      clearTimeout(this.timerID);
      this.timerID = 0;
    }
  };

</script>
```

В примере кода из листинга 8.4 создана структура центрального управления ❶, где можно вводить новые функции обратного вызова таймеров и начинать либо останавливать их выполнение. Кроме того, функции обратного вызова способны удалять себя сами в любой момент, просто возвращая логическое значение `false`, что намного проще сделать, чем с помощью типичного вызова метода `clearTimeout()`. А теперь проанализируем более подробно код из данного примера.

Сначала все функции обратного вызова сохраняются в массиве `timers` наряду с идентификатором любого текущего таймера ❷. Эти переменные составляют единственное состояние, которое требуется поддерживать в данной конструкции таймера. Метод `add()` принимает обработчик обратного вызова и просто вводит его в массив `timers` ❸.

Но самое главное начинается с метода `start()`, где сначала проверяется, действует ли уже какой-нибудь таймер ❹. В частности, проверяется наличие значения в свойстве `timerID` из объекта управления таймером. И если ни один из таймеров не действует, то сразу же выполняется немедленно вызываемая функция, запускающая центральный таймер.

Если зарегистрированы какие-нибудь обработчики, то в теле немедленно вызываемой функции организуется цикл для выполнения каждого обработчика. Если же обработчик возвращает логическое значение `false`, то он удаляется из массива обработчиков и далее планируется следующий такт анимации.

С помощью такой конструкции сначала формируется элемент разметки анимации, как показано ниже.

```
<div id="box">Hello!</div>
```

А затем начинается анимация в следующем фрагменте кода:

```
var box = document.getElementById("box"), x = 0, y = 20;

timers.add(function() {
  box.style.left = x + "px";
  if (++x > 50) return false;
});

timers.add(function() {
  box.style.top = y + "px";
  y += 2;
  if (y > 120) return false;
});

timers.start();
```

В этом фрагменте кода сначала получается ссылка на элемент, а затем вводится один обработчик для перемещения элемента по горизонтали и другой — по вертикали. И после этого начинается сама анимация. Результат завершения анимации приведен на рис. 8.5.

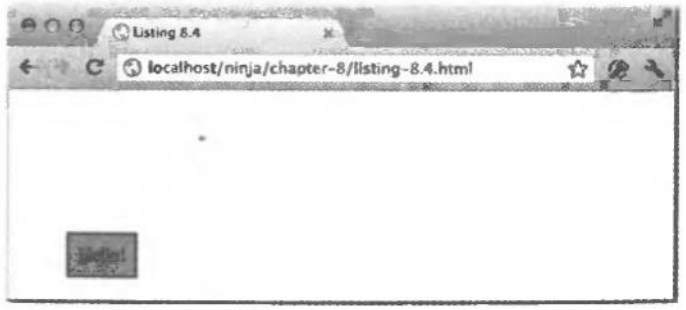

Рис. 8.5. После выполнения нескольких обработчиков анимации оживляемый элемент был перемещен вниз по странице

Следует иметь в виду, что такая организация таймеров гарантирует, что функции обратного вызова будут всегда выполняться в том порядке, в каком они вводятся. А ведь это не всегда гарантируется обычными таймерами, поскольку браузер может выбрать

выполнение одного таймера прежде другого. Такая организация таймеров имеет решающее значение для крупных приложений и вообще для любого вида анимации на JavaScript. Наличие готового решения, безусловно, способствует последующей разработке веб-приложений и особенно при создании различных видов анимации. Помимо анимации, центральное управление таймерами помогает также организовать асинхронное тестирование кода. Ниже будет показано, как это делается.

Асинхронное тестирование

Еще одна ситуация, в которой оказывается полезным центральное управление таймерами, возникает при организации асинхронного тестирования. Главная трудность здесь в том, что, когда требуется выполнить тестирование действий, которые не могут быть завершены немедленно (например, внутри таймера или запроса типа XMLHttpRequest), тестовый набор приходится организовывать таким образом, чтобы он действовал совершенно асинхронно.

Как было показано на целом ряде примеров в предыдущих главах, тесты можно легко выполнять, постепенно доходя до каждого из них, и, как правило, этого оказывается достаточно. Но если требуется асинхронное тестирование, то все тесты придется разделить и затем обращаться с ними по отдельности, как показано в листинге 8.5. Код в этом листинге может показаться вам уже знакомым.

Листинг 8.5. Простой набор асинхронных тестов

```
<script type="text/javascript">

  (function() {

    var queue = [], paused = false;
      this.test = function(fn) {
      queue.push(fn);
      runTest();
    };

    this.pause = function() {
    paused = true;
    };

    this.resume = function() {
      paused = false;
      setTimeout(runTest, 1);
    };

    function runTest() {
      if (!paused && queue.length) {
        queue.shift()();
        if (!paused) resume();
      }
    }
  })();

</script>
```

Сохранить состояние

Определить функцию регистрации

Определить функцию для приостановки тестирования

Определить функцию для возобновления тестирования

Выполнить тесты

В примере кода из листинга 8.5 обращает на себя внимание, прежде всего, тот факт, что каждая функция, передаваемая функции `test()`, будет содержать не более одного асинхронного теста. Асинхронность последнего определяется функциями `pause()` и `resume()`, вызываемыми до и после асинхронного события. На самом деле рассматриваемый здесь код служит лишь одной цели: поддерживать определенный порядок выполнения функций, содержащих асинхронное поведение. И хотя его совсем не обязательно использовать только для выполнения контрольных примеров, он особенно полезен именно для этой цели.

А теперь проанализируем код, необходимый для реализации асинхронного поведения. В этом отношении он очень похож на код из листинга 8.4. Большая часть функциональных возможностей данного кода содержится в функциях `resume()` и `runTest()`. Они действуют аналогично методу `start()` из предыдущего примера, но в данном случае им приходится иметь дело с очередью данных. Для этого функция, ожидающая своей очереди, извлекается из нее и затем выполняется, а иначе действие интервала времени полностью прекращается. Следует, однако, иметь в виду, что код обработки очереди действует совершенно асинхронно и в пределах заданного интервала времени. Благодаря именно этому последующее выполнение кода гарантируется после вызова функции `pause()`.

Рассматриваемый здесь небольшой фрагмент кода вынуждает тестовый набор действовать совершенно асинхронно, поддерживая в то же время порядок выполнения тестов, что может иметь решающее значение для некоторых тестовых наборов, если результаты их выполнения носят деструктивный характер, оказывая отрицательное влияние на другие тесты. Правда, данный пример ясно показывает, что для ввода надежного асинхронного тестирования в существующий тестовый набор не требуется много кода, если пользоваться таймерами эффективно.

Резюме

Эта глава была посвящена применению таймеров в коде JavaScript, что само по себе очень интересно! Подведем итог тому, что было рассмотрено в этой главе.

- Реализовать эти, на первый взгляд, простые средства на самом деле не так легко. Но разобравшись в особенностях работы таймеров, можно научиться эффективно применять их на практике.
- Совершенно очевидно, что наибольшую пользу таймеры приносят в сложных приложениях, включая следующее.
  - Затратный по вычислениям код.
  - Анимация.
  - Асинхронные тестовые наборы.
- Благодаря простоте своего применения (особенно вместе с замыканиями) таймеры позволяют легко выходить даже из самых трудных ситуаций.

Итак, мы рассмотрели целый ряд средств и способов, которыми можно пользоваться при написании сложного кода, держа в то же время его сложность под полным контролем. В следующей главе мы уделим основное внимание тому, каким образом в JavaScript производятся вычисления во время выполнения и как поставить их потенциальные возможности себе на службу.

Часть III

Обучение кандидата в мастера

Итак, усвоив основы мастерства программирования на JavaScript и пройдя курс ученичества, вы можете теперь перейти к чтению данной части книги, где поясняется, как эти основы можно применить на практике выживания в среде браузеров, которая нередко оказывается недружелюбной к разработчикам веб-приложений. Представленные здесь способы выхода из затруднительных положений, в которые браузеры нередко ставят разработчиков веб-приложений, основываются на знаниях и опыте самых лучших мастеров программирования на JavaScript.

В главе 9 мы сразу же приступим к рассмотрению особенностей вычисления кода – передового способа, который самые опытные программирующие на JavaScript обычно приберегают про запас. Но и вы можете пополнить им свой арсенал.

В главе 10 мы обсудим оператор `with` – едва ли не самую противоречивую языковую конструкцию JavaScript. И хотя его употребление в новом коде не рекомендуется, тем не менее вы можете воспользоваться им в любом унаследованном коде, с которым вам придется иметь дело.

Из главы 11 вы узнаете, как решать вопросы разработки кросс-браузерного кода и с честью пройти это суровое испытание. Курс обучения кандидата в мастера завершается главой 12, где подробно рассматриваются атрибуты, свойства объектов и связанные с ними темы, в том числе стили и таблицы CSS. Проработав материал главы 12, вы, вероятно, будете жаждать большего. И такая возможность вам представится в следующей, четвертой части книги, где продолжается посвящение в тайны мастерства программирования на JavaScript.

9 Вычисление кода во время выполнения

В этой главе...

- Особенности вычисления кода
- Различные способы вычисления кода
- Применение вычисления кода в приложениях
- Декомпиляция функций
- Организация пространства имен
- Уплотнение и запутывание кода

К числу многих сильных сторон, отличающих JavaScript от других языков программирования, относится возможность динамически интерпретировать и исполнять фрагменты кода во время выполнения. Вычисление кода — это одновременно и самое развитое, и самое неверно используемое языковое средство JavaScript. Поэтому правильное представление о тех ситуациях, когда его можно и *должно* использовать (наряду с наилучшими способами применения), дает вполне определенные преимущества при разработке современных веб-приложений на JavaScript.

В этой главе исследуются различные способы интерпретации кода во время выполнения и в тех случаях, когда это эффективное средство способно усовершенствовать качество самого кода. В ней будут рассмотрены различные механизмы, предоставляемые в JavaScript для вычисления кода во время выполнения, а также показано, каким образом вычисление кода во время выполнения может быть использовано в различных, представляющих особый интерес ситуациях, которые могут возникнуть при разработке веб-приложений. Но прежде всего следует выяснить, как же организовать вычисление кода во время выполнения.

Механизмы вычисления кода

В JavaScript имеются самые разные механизмы вычисления кода. У каждого из них имеются свои преимущества и недостатки, поэтому выбирать их следует тщательно, исходя из контекста, в котором их предполагается использовать. К числу этих механизмов относятся следующие.

- Метод `eval()`.
- Функции-конструкторы.
- Таймеры.
- Элемент разметки `<script>`.

Исследуя каждый из этих механизмов, мы обсудим сначала область действия подобного рода вычисления, а затем надежные практические приемы, которые следует иметь в виду при организации вычисления кода во время выполнения. И начнем мы с самого распространенного способа реализации вычисления кода, которым нередко пользуются авторы веб-страниц.

Вычисление кода с помощью метода `eval()`

Метод `eval()` относится к самым распространенным средствам вычисления кода во время выполнения. Этот метод определяется в глобальной области действия как функция и выполняет в своем текущем контексте код, передаваемый ему в форме символьной строки. А возвращает этот метод результат вычисления последнего выражения.

Основные функциональные возможности

Рассмотрим основные функциональные возможности метода `eval()` с практической точки зрения. От него ожидаются две самые основные функциональные возможности.

- Вычисление кода, передаваемого ему в форме символьной строки.
- Выполнение этого кода в той области действия, в которой вызывается метод `eval()`.

Рассмотрим пример кода, приведенный в листинге 9.1, где предпринимается попытка подтвердить перечисленные выше функциональные возможности метода `eval()`.

Листинг 9.1. Проверка основных функциональных возможностей метода `eval()`

```
<script type="text/javascript">

  assert(eval("5 + 5") === 10,
        "5 and 5 is 10");

  assert(eval("var ninja = 5;") === undefined,
        "no value was returned" );

  assert(ninja === 5, "The variable ninja was created");

  (function(){
    eval("var ninja = 6;");
    assert(ninja === 6,
    "evaluated within the current scope.");
```

❶ Проверить простое выражение

❷ Проверить выражение, не имеющее значения

❸ Проверить побочный эффект

❹ Проверить область действия для вычисления


```
  })();

  assert(window.ninja === 5,
        "the global scope was unaffected");

  assert(ninja === 5,
        "the global scope was unaffected");

</script>
```

❺ Проверить "чистоту" в области действия

В приведенном выше примере кода проверяется целый ряд утверждений относительно функциональных возможностей метода `eval()`. Результаты тестирования этих утверждений приведены на рис. 9.1.

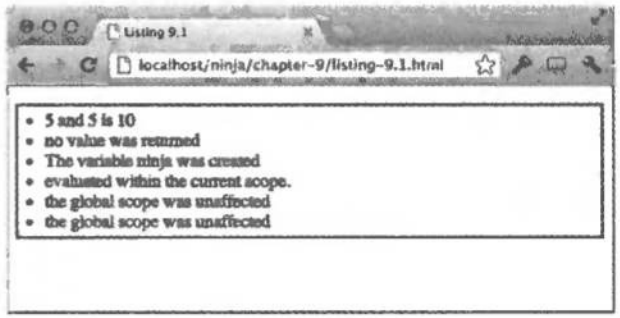

Рис. 9.1. Подтверждение функциональных возможностей метода `eval()` вычислять различные выражения в локальной области действия

Сначала в рассматриваемом здесь коде методу `eval()` передается символьная строка, содержащая простое выражение, и проверяется правильность получаемого результата ❶. Затем предпринимается попытка вычислить оператор присваивания `ninja=5`, не дающий в результате никакого значения, и проверяется, что возврат значения не ожидается ❷. Но ведь этого явно недостаточно для тестирования. В самом деле, никакого результата не ожидалось, но какова причина его отсутствия: вычисление выражения не дало никакого результата, или же просто ничего не произошло? Ответ на этот вопрос требует дополнительного тестирования.

Далее предполагается, что код вычисляется в текущей области действия (в данном случае в глобальной), поэтому в качестве побочного эффекта вычисления ожидается создание переменной `ninja` в глобальной области действия. И действительно, очередной простой тест подтверждает это предположение ❸.

После этого требуется проверить правильность вычисления в глобальной области действия. С этой целью создается немедленно вызываемая функция и в ней вычисляется выражение `var ninja=6;` ❹. А далее выполняется тест на существование этой переменной с предполагаемым значением. Но этого теста явно недостаточно. Ведь неясно, значение переменной `ninja` вычисляется равным **6**, потому что в локальной области действия была создана новая переменная или же потому, что была видоизменена одноименная переменная в глобальной области действия? Поэтому еще один тест подтверждает, что глобальная область действия не была затронута ❺.

Результаты вычисления

Как пояснялось ранее, метод `eval()` возвращает результат вычисления последнего выражения в переданной ему символьной строке. Так, если сделать следующий вызов:

```
eval('3+4;5+6')
```

то в результате будет возвращено значение **11**.

Следует, однако, заметить, что все, что не относится к простым переменным, примитивам или присваиванию, фактически должно быть заключено в круглые скобки. Это необходимо для того, чтобы было возвращено правильное значение. Так, если бы потребовалось создать простой объект, используя метод `eval()`, с этой целью можно было бы попытаться написать следующую строку кода:

```
var o = eval('{ninja: 1}');
```

Но это не дало бы желаемого результата. Поэтому литерал объекта следует *непременно* заключить в круглые скобки, как показано ниже.

```
var o = eval('({ninja: 1})');
```

А теперь составим и выполним ряд дополнительных тестов, как показано в примере кода из листинга 9.2.

Листинг 9.2. Проверка значений, возвращаемых из метода `eval()`

```
<script type="text/javascript">

  var ninja = eval("({name:'Ninja'})");
  assert(ninja != undefined,"the ninja was created");
  assert(ninja.name === "Ninja",
        "and with the expected property");

  var fn = eval("(function(){return 'Ninja';})");
  assert(typeof fn === 'function',
        "the function was created");
  assert(fn() === "Ninja",
        "and returns expected value");

  var ninja2 = eval("{name:'Ninja'}");
  assert(ninja2 != undefined,"ninja2 was created");
  assert(ninja2.name === "Ninja",
        "and with the expected property");

</script>
```

❶ Создать объект из символьной строки, содержащей литерал объекта, и проверить, что создан не только объект, но и ожидаемое его свойство name

❷ Создать функцию из литерала функции в символьной строке и проверить, что функция создана и при ее вызове возвращается ожидаемое значение

Попытаться создать еще один вариант первого теста, опустив круглые скобки. Первый тест проходит (т.е. объект создан), а второй тест не проходит (т.е. ожидаемый объект не создан). С помощью отладчика кода JavaScript можно выяснить, что же на самом деле было создано

В приведенном выше примере кода объект ❶ и функция ❷ оперативно создаются с помощью метода `eval()`. Обратите внимание на то, что в обоих случаях вычисляемые выражения следует непременно заключить в квадратные скобки. В качестве упражнения сделайте копию файла `listing-9.2.html`, удалите круглые скобки и загрузите этот файл в окно браузера. Видите, как далеко вы уже продвинулись!

Если вы попробуете выполнить тесты из данного файла в браузере Internet Explorer или более ранней версии, то, скорее всего, получите довольно неожиданный результат.

В версиях, предшествовавших Internet Explorer 9, возникает серьезное затруднение при выполнении данного конкретного синтаксиса. Поэтому для обеспечения правильного выполнения метода `eval()` при его вызове приходится составлять замысловатое логическое выражение. Ниже приведен пример из библиотеки jQuery, демонстрирующий применение специального приема для создания функции с помощью метода `eval()` в прежних версиях браузера Internet Explorer. А в версии Internet Explorer 9 этот недостаток устранен.

```
var fn = eval("false||function(){return true;}");
assert(fn() === true,
      "The function was created correctly.");
```

Уместно спросить: а зачем вообще создавать функцию именно таким образом? Обычно к такому приему не прибегают. Ведь если заранее известно, какую именно функцию требуется создать, то она определяется одним из способов, подробно рассматривавшихся в главе 3. Но что, если синтаксис функции заранее неизвестен? Ее код можно сгенерировать во время выполнения или получить из другого источника. (Если эта последняя возможность пугает вас, не смущайтесь — вопросов безопасности мы коснемся далее.)

Как и при создании функций в конкретной области действия обычными средствами, функции, создаваемые с помощью метода `eval()`, наследуют замыкание в той же области действия вследствие того, что метод `eval()` выполняется в локальной области действия. Разумеется, если дополнительное замыкание не требуется, то ему можно найти подходящую альтернативу.

Вычисление кода с помощью функции-конструктора

Все функции в JavaScript являются экземплярами объекта типа `Function`. Как пояснялось в главе 3, именованные функции можно создавать с помощью синтаксиса `function имя(...){...}` или же опускать имя, если требуется создать анонимную функцию. Но мы можем не останавливаться на этом, получая экземпляры функций непосредственно с помощью конструктора класса `Function`, как показано ниже.

```
var add = new Function("a", "b", "return a + b;");
assert(add(3,4) === 7, "Function created and working!");
```

Последним аргументом в списке аргументов переменной длины конструктора типа `Function` всегда указывается код, который становится телом самой функции. А все предшествующие ему аргументы будут обозначать имена параметров функции. Поэтому приведенный выше пример кода равнозначен следующему:

```
var add = function(a,b) { return a + b; }
```

Несмотря на то что оба приведенных выше примера кода функционально равнозначны, явное их отличие состоит в том, что при создании функции с помощью конструктора тело функции передается в виде символьной строки во время выполнения. Еще одно важное отличие заключается в том, что при создании функций с помощью конструктора типа `Function` замыкания не образуются. Это может пригодиться в том случае, если требуется избежать дополнительных издержек, связанных с ненужными замыканиями.

Вычисление кода с помощью таймеров

Код может быть вычислен и асинхронно с помощью таймеров. Как пояснялось в главе 8, для этой цели таймеру передается встраиваемая функция или ссылка на функцию при вызове методов `setTimeout()` и `setInterval()`, хотя они могут также принимать символьные строки для вычисления после запуска таймера. В качестве примера ниже приведена строка кода, в которой функция создается с помощью таймера.

```
var tick = window.setTimeout('alert("Hi!")',100);
```

Впрочем, к такому способу приходится прибегать очень редко, поскольку он мало чем отличается от применения упоминавшейся ранее конструкции `new Function()`. Поэтому пользоваться данным способом обычно не рекомендуется, за исключением тех случаев, когда код должен вычисляться из символьной строки во время выполнения.

Вычисление кода в глобальной области действия

При обсуждении метода `eval()` ранее подчеркивалось, что вычисление кода выполняется в той области действия, в которой этот метод вызывается, что и было подтверждено тестом из листинга 9.1. Но зачастую код из символьных строк требуется вычислять в глобальной области действия, несмотря на то, что он может и не находиться в текущей области действия, где происходит его выполнение. Например, в некоторых функциях возникает потребность вычислять код в глобальной области действия, как показано ниже.

```
(function(){
  eval("var test = 5;");
})();

assert(test === 5,
      "Variable created in global scope"); // тест не пройдет!
```

Надеясь на создание переменной `test` в глобальной области действия после выполнения немедленно вызываемой функции, мы будем сильно разочарованы результатом ее тестирования, поскольку тест не пройдет. Дело в том, что область действия для вычисления кода ограничивается телом немедленно вызываемой функции, и там же находится переменная. Эта ситуация наглядно иллюстрируется на рис. 9.2.

В качестве наивного решения можно было бы изменить порядок вычисления кода следующим образом:

```
eval("window.test = 5;");
```

И хотя это действительно приведет к определению переменной в глобальной области действия, тем не менее область, в которой происходит вычисление, не изменится, и поэтому она по-прежнему останется локальной, а не глобальной. В данном простом примере всего лишь присваивается числовой литерал переменной, но этот пример приобретает намного большее значение, когда дело доходит до ссылок на переменные из локальной области действия.

Впрочем, в современных браузерах можно применить специальный прием, чтобы добиться приемлемого результата. Он состоит в том, чтобы вставить динамический дескриптор `<script/>` в документ со сценарием, который требуется выполнить. Андреа Джамарки (Andrea Giammarchi — мастер-самоучка программирования на JavaScript и PHP) разработал этот специальный прием для применения на многих платфор-

мах. С первоначальным вариантом этого приема можно ознакомиться в блоге *Web Reflection* (Веб-отражение) этого автора по адресу `http://webreflection.blogspot.com/2007/08/global-scope-evaluation-and-dom.html`. Адаптированный вариант этого приема приведен в листинге 9.3.

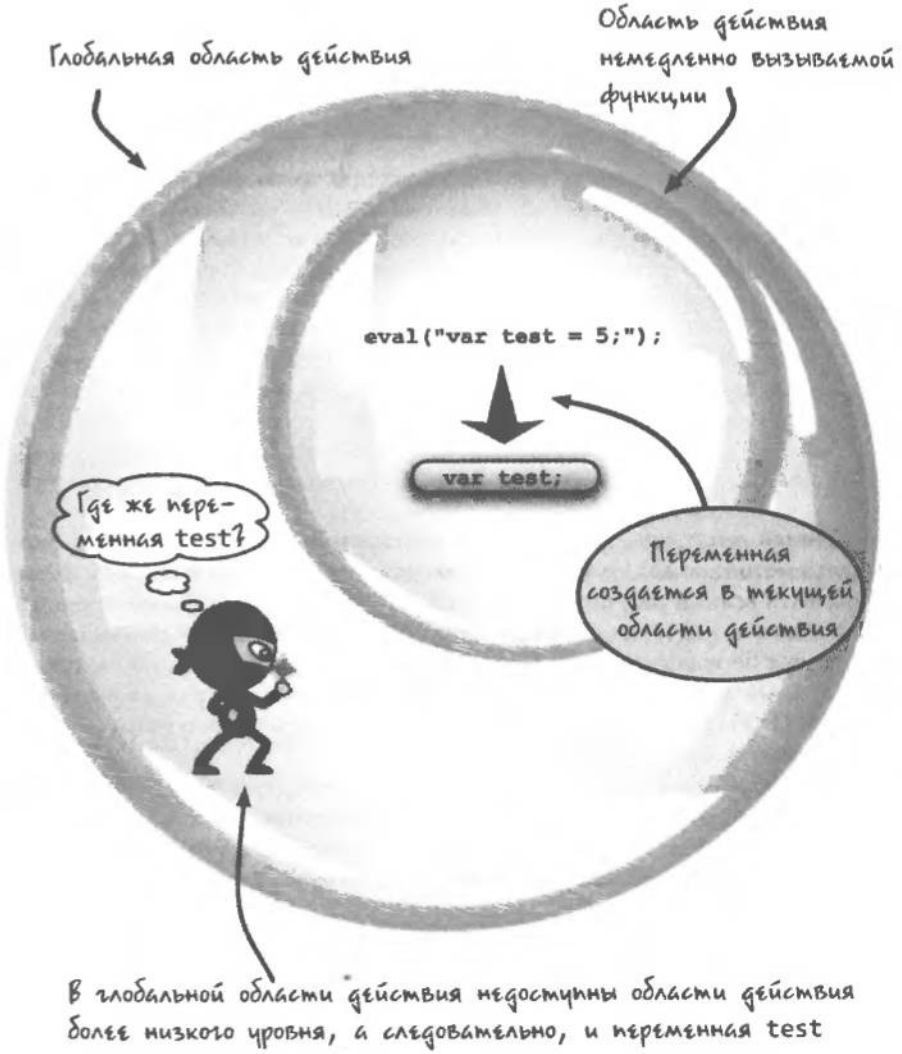

Рис. 9.2. Код вычисляется в теле немедленно вызываемой функции, а следовательно, переменная создается в области действия этой функции и недоступна в других областях действия

Листинг 9.3. Вычисление кода в глобальной области действия

```
<script type="text/javascript">

  function globalEval(data) {
    data = data.replace(/^\s*|\s*$/g, "");
```

❶ Определить глобальную функцию вычисления кода

```
    if (data) {
      var head = document.getElementsByTagName("head")[0] ||
                 document.documentElement;
          script = document.createElement("script");   ❷ Создать узел сценария

      script.type = "text/javascript";
      script.text = data;

      head.appendChild(script);   ❸ Присоединить узел к модели DOM
      head.removeChild(script);   ❹ Удалить узел
    }
  }

  window.onload = function() {
    (function() {
      globalEval("var test = 5;");
    })();

    assert(test === 5, "The code was evaluated globally.");
  };

</script>
```

Приведенный выше код, реализующий рассматриваемый здесь прием, довольно прост. Вместо метода `eval()` в нем определяется функция `globalEval()` ❶, которую можно вызывать всякий раз, когда требуется, чтобы вычисление кода происходило в глобальной области действия. Сначала в данной функции удаляются любые пробелы в начале и конце передаваемой ей символьной строки, для чего составляется регулярное выражение (подробнее об этом см. в главе 7), а затем находится элемент разметки `<head>` модели DOM или самого документа и создается присоединяемый соответственно элемент разметки `<script>` ❷.

Далее устанавливается тип элемента разметки сценария и в его тело загружается символьная строка с вычисляемым кодом. Присоединение элемента разметки сценария к модели DOM в качестве производного от элемента разметки `<head>` ❸ приводит к вычислению сценария в глобальной области действия. И как только сценарий сделает

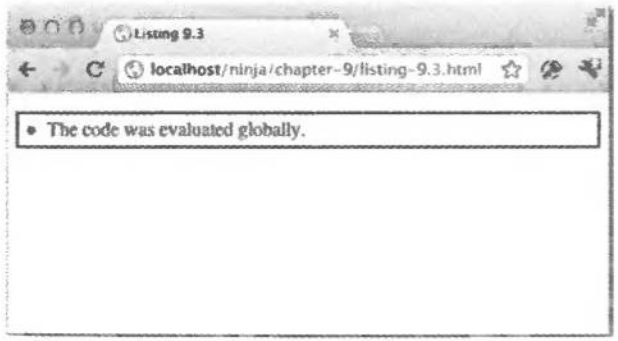

Рис. 9.3. Вычисление кода можно выполнить в глобальном контексте, если произвести незначительные манипуляции с моделью DOM

свое дело, элемент его разметки удаляется без всяких церемоний ❹. Результаты тестирования данного кода приведены на рис. 9.3.

Данный код чаще всего применяется в тех случаях, когда динамически выполняемый код возвращается с сервера. Практически всегда требуется, чтобы подобного рода код выполнялся в глобальной области действия. А для этого необходима описанная выше новая функция. Но можно ли доверять серверу вычисление кода? Попробуем найти ответ на этот вопрос в следующем разделе.

Безопасное вычисление кода

В связи с вычислением кода нередко возникает вопрос, связанный с безопасным выполнением кода JavaScript. Иными словами, возможно ли безопасное выполнение ненадежного кода JavaScript на странице, не нарушая целостность веб-сайта? Кто его знает, что содержит такой код? Ведь некоторые наивные программирующие могут предоставить символьную строку с кодом, который выполняет бесконечный цикл, удаляет нужные элементы из модели DOM или портит ценные данные. А еще хуже, если какой-нибудь злоумышленник намеренно внесет вредоносный код, способный нарушить безопасность веб-сайта. Поэтому ответ на этот вопрос, как правило, отрицательный. Дело в том, что существует немало способов обойти любые преграды и в итоге получить доступ к информации, не предназначенной для разглашения.

Впрочем, есть надежда. В компании Google разработан проект под названием Caja (`http://code.google.com/p/google-caja`). Он представляет собой попытку создать транслятор для JavaScript, преобразующий код JavaScript в более безопасный вид, неуязвимый к атакам со злонамеренной целью. Рассмотрим в качестве примера следующий фрагмент кода:

```
var test = true;
(function(){ var foo = 5; })();
Function.prototype.toString = function(){};
```

Компилятор Caja преобразует этот код в следующий вид:

```
___.loadModule(function (___, IMPORTS___) {
{
  var Function = ___.readImport(IMPORTS___, 'Function');
  var x0___;
  var x1___;
  var x2___;
  var test = true;
  ___.asSimpleFunc(___.primFreeze(___.simpleFunc(function () {
  var foo = 5;
  })))();
  IMPORTS___[ 'yield' ] ((x0___ = (x2___ = Function,
  x2___.prototype_canRead___?
  x2___.prototype: ___.readPub(x2___, 'prototype')),
  x1___ = ___.primFreeze(___.simpleFunc(function () {})),
  x0___.toString_canSet___? (x0___.toString = x1___):
  ___.setPub(x0___, 'toString', x1___)));
}
});
}
```

Обратите внимание на обширное применение встроенных методов и свойств для проверки целостности данных, производимой главным образом во время выполнения. А замысловатые их имена со знаками подчеркивания являются попыткой исключить случайное их совпадение с теми именами, которые могут быть использованы на веб-странице.

Потребность в безопасном выполнении кода JavaScript возникает в связи со стремлением разрабатывать веб-приложения со смешанным, неоднородным содержимым (так называемые *мэшапы*), где можно благополучно встраивать рекламу, не беспокоясь о том, что безопасность пользователя будет поставлена под угрозу. В данной области предстоит еще сделать немало, и ведущая роль в этом принадлежит проекту Caja компании Google.

Итак, мы рассмотрели целый ряд способов преобразования символьной строки в немедленно исполняемый код. А можно ли сделать наоборот, декомпилировав код?

Декомпиляция функций

В большинстве реализаций JavaScript предоставляется возможность декомпилировать уже вычисленный код JavaScript. Как пояснялось в главе 6, такой процесс называется *сериализацией*, хотя для его обозначения употребляется также термин *декомпиляция*. Как правило, термин *декомпиляция* обозначает восстановление исходного кода из ассемблерного или байт-кода, что, очевидно, не подходит для JavaScript. Но помимо сериализации, у которой имеются свои семантические особенности, на самом деле не существует подходящего термина для обозначения данного процесса. Так, термин "развычисление" трудно вообще произнести, поэтому мы будем в дальнейшем пользоваться термином *декомпиляция*, признавая, что он не совсем подходит в данном контексте.

Какой бы сложной ни показалась декомпиляция на первый взгляд, на самом деле она реализуется довольно просто с помощью метода `toString()` для вызываемых функций. Проверим это на очередном примере кода, приведенном в листинге 9.4.

Листинг 9.4. Декомпиляция функции в символьную строку

```
<script type="text/javascript">

  function test(a){ return a + a; }

  assert(test.toString() ===
         "function test(a){ return a + a; }",
         "Function decompiled");

</script>
```

❶ Определить функцию

❷ Проверить результат декомпиляции

В приведенном выше примере кода сначала создается функция `test` ❶, а затем утверждается, что метод `toString()` этой функции возвращает ее исходный текст. Следует, однако, иметь в виду, что значение, возвращаемое методом `toString()`, будет содержать все пробелы из исходного объявления функции, *включая* и ограничители строк. Для целей тестирования в листинге 9.4 использовано простое определение функции в одной строке. Если же сделать копию файла с рассматриваемым здесь кодом и внести изменения в форматирование объявления функции, то тест не пройдет до тех пор, пока проверяемая символьная строка не будет точно согласована с форматом объ-

явления функции. Поэтому непременно учитывайте пробелы и форматирование тела функции, выполняя ее декомпиляцию.

Такому способу декомпиляции можно найти целый ряд применений, особенно при переписывании макрокоманд и кода. Но чаще всего она применяется для считывания аргументов декомпилированной функции в массив именованных аргументов, как это делается, например, в библиотеке Prototype для JavaScript. С помощью декомпиляции можно зачастую проанализировать содержимое функции и определить, какие именно значения предполагается в ней получить. В листинге 9.5 приведен пример упрощенного кода из библиотеки Prototype для выведения имен параметров функции.

Листинг 9.5. Функция для поиска имен аргументов другой функции

```
<script type="text/javascript">

  function argumentNames(fn) {
    var found = /^[\s\(]*function[^(]*\(\s*([^)]*?)\s*\)/
                .exec(fn.toString());
    return found && found[1] ?
           found[1].split(/,\s*/) :
           [];
  }

  assert(argumentNames(function(){}).length === 0,
         "Works on zero-arg functions.");
  assert(argumentNames(function(x){})[0] === "x",
         "Single argument working.");

  var results = argumentNames(function(a,b,c,d,e){});
  assert(results[0] == 'a' &&
         results[1] == 'b' &&
         results[2] == 'c' &&
         results[3] == 'd' &&
         results[4] == 'e',
         "Multiple arguments working!");

</script>
```

❶ Найти список аргументов

❷ Разделить список

❸ Проверить отсутствие аргументов

❹ Проверить наличие единственного аргумента

❺ Проверить наличие нескольких аргументов

Функция в приведенном выше коде состоит всего лишь из нескольких строк кода, но в операторах ее тела используется немало расширенных языковых средств JavaScript. Сначала в данной функции осуществляется декомпиляция передаваемой ей функции, а для извлечения списка разделяемых запятой аргументов составляется регулярное выражение ❶ (подробнее о регулярных выражениях см. в главе 7). Следует иметь в виду, что в методе `exec()` предполагается получить символьную строку, и поэтому приведение аргумента функции к строковому формату с помощью метода `toString()` можно было бы опустить, чтобы этот метод вызывался неявно. Но в данном примере его вызов явно включен в код ради большей ясности.

Затем результат извлечения разделяется на составляющие значения, чтобы подготовить список аргументов к разным видам проверки, в том числе и на отсутствие аргументов ❷. И наконец, проверяется отсутствие аргументов ❸, наличие единственного аргумента ❹ и нескольких аргументов ❺. Соответствующие тесты выполняются так, как и предполагалось, а их результаты приведены на рис. 9.4.

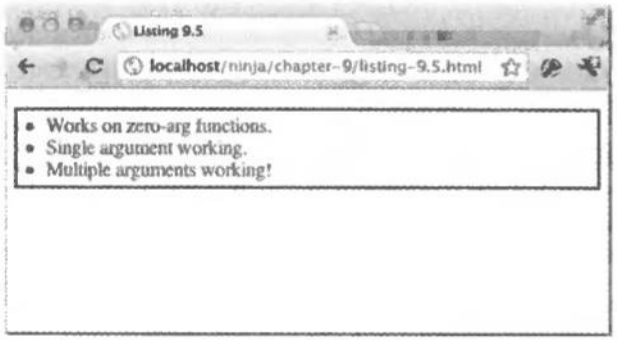

Рис. 9.4. Выполняя декомпиляцию функции, можно осуществить немало интересного и, в частности, вывести имена ее аргументов

Обращаясь с функциями подобным образом, следует принимать во внимание то обстоятельство, что далеко не все браузеры поддерживают декомпиляцию. К числу тех браузеров, которые все-таки поддерживают декомпиляцию, относится Opera Mini. Если этот браузер относится к числу поддерживаемых в веб-приложении, данное обстоятельство следует принять во внимание в том коде, где выполняется декомпиляция функции.

Как подчеркивалось ранее в этой книге и особенно будет акцентировано в последующих ее главах, для того чтобы выявить наличие поддержки конкретной функциональной возможности (в данном случае декомпиляции функций совсем), не обязательно обращаться непосредственно к браузеру. Вместо этого можно прибегнуть к имитации данной функциональной возможности, подробнее рассматриваемой в главе 11, чтобы проверить, поддерживается ли она в браузере. Это можно, в частности, сделать следующим образом:

```
var FUNCTION_DECOMPILATION = /abc(.|\n)*xyz/.test(function(abc){xyz;});
assert(FUNCTION_DECOMPILATION,
      "Function decompilation works in this browser");
```

И в этом случае с помощью регулярного выражения, а такие выражения, к сожалению, недостаточно применяются в практике программирования на JavaScript, функция передается методу `test()`, причем метод `toString()` вызывается неявно, поскольку в методе `test()` предполагается получить символьную строку. А результат сохраняется в переменной для последующего применения или тестирования, как в данном случае.

Итак, мы рассмотрели различные средства вычисления кода во время выполнения. А теперь перейдем к их применению на практике.

Вычисление кода на практике

В начале этой главы были рассмотрены различные способы организации вычисления кода. Им можно найти интересное практическое применение в разрабатываемом коде. Рассмотрим некоторые практические примеры вычисления кода, чтобы дать более полное представление о том, когда и где следует пытаться использовать этот прием в прикладном коде.

Преобразование из формата JSON

Едва ли не чаще всего вычисление кода во время выполнения употребляется для преобразования символьных строк формата JSON в их объектные представления на JavaScript. Данные в формате JSON представлены в виде подмножества языка JavaScript, и поэтому они вполне поддаются вычислению как код JavaScript.

В большинстве современных браузеров поддерживается собственный объект формата JSON с помощью методов `parse()` и `stringify()`, но в целом ряде действующих браузеров этот объект по-прежнему не предоставляется. В отношении таких браузеров следует знать, как обращаться с форматом JSON без объекта `window.JSON`. Но как часто бывает, даже в самые совершенные планы вкрадывается какая-нибудь одна мелкая неувязка, которую следует все же принимать во внимание. В частности, текст, представляющий языковые конструкции, следует заключать в круглые скобки, чтобы правильно вычислить код. Впрочем, сделать это совсем не трудно, как показано в примере кода из листинга 9.6. Нужно лишь не забывать об этом.

Листинг 9.6. Преобразование символьной строки формата JSON в объект JavaScript

```
<script type="text/javascript">

  var json = '{"name":"Ninja"}';

  var object = eval("(" + json + ")");

  assert(object.name === "Ninja",
         "My name is Ninja!");

</script>
```

Определить исходную строку в формате JSON, представляющую объект с единственным свойством

Преобразовать строку формата JSON в объект JavaScript

Проверить, успешно ли выполнено преобразование

Как видите, такое преобразование осуществляется довольно просто, причем в большинстве механизмов JavaScript. Но пользоваться методом `eval()` для синтаксического анализа строк формата JSON следует с одной оговоркой. Чаще всего данные формата JSON поступают с удаленного сервера, а слепо выполнять код, поступающий с удаленного сервера, неосмотрительно, как пояснялось выше.

К числу самых распространенных относится сценарий преобразования из формата JSON, написанный Дугласом Крокфордом (Douglas Crockford), разработавшим спецификацию разметки документов в формате JSON. В этом сценарии выполняется первоначальный синтаксический анализ строки формата JSON, чтобы попытаться воспрепятствовать проникновению любой вредоносной информации. Полный код этого сценария можно найти по следующему адресу: `https://github.com/douglascrockford/JSON-js`.

Этот сценарий выполняет следующие важные операции предварительной подготовки непосредственно к вычислению кода.

- Защищает от некоторых символов уникода, способных вызвать осложнения в некоторых браузерах.
- Защищает от шаблонов, не относящихся к формату JSON, но имеющих злонамеренные цели, включая оператор присваивания и оператор `new`.
- Проверяет наличие только символов, допустимых в формате JSON.

Если данные в формате JSON поступают для вычисления кода из собственных или других надежных источников (прикладного кода или серверов), то и беспокоиться о внесении злонамеренного кода особенно не стоит, хотя неплохо бы и перестраховаться на всякий случай. Но если нет никаких оснований доверять данным формата JSON, предназначенным для вычисления кода, то благоразумнее всего принять такие меры защиты, как в упомянутом выше сценарии, написанном Дугласом Крокфордом.

Тема обращения с ненадежным кодом подробно исследуется в следующих книгах, вышедших в издательстве Manning Publications.

- *Single Page Web Applications* (Одностраничные веб-приложения) Майкла С. Миковского (Michael S. Mikowski) и Джоша К. Пауэлла (Josh C. Powell) (`http://www.manning.com/mikowski/`).
- *Third-Party JavaScript* (Сторонние сценарии на JavaScript) Бена Винегара (Ben Vinegar) и Антона Ковалева (`http://manning.com/vinegar/`).

А теперь перейдем к другому распространенному на практике примеру вычисления кода во время выполнения.

Импорт кода, размещаемого в пространстве имен

Как пояснялось в главе 3, размещение кода в пространстве имен служит в качестве меры против засорения текущего контекста — обычно глобального. И это очень хорошая мера. Но что, если код, размещенный в пространстве имен, требуется намеренно перенести в текущий контекст? Сделать нечто подобное не так-то просто, если учесть, что в JavaScript для этого не предусмотрено никаких средств. Чаще всего для этого приходится выполнять действия, аналогичные приведенным ниже.

```
var DOM = base2.DOM;
var JSON = base2.JSON;
// и т.д.
```

В библиотеке base2 предоставляется очень интересное решение задачи импорта из пространства имен в текущий контекст. Но поскольку автоматизировать эту задачу нельзя, то для ее решения можно прибегнуть к вычислению кода во время выполнения. Всякий раз, когда новый класс или модуль добавляется в пакет base2, формируется символьная строка с исполняемым кодом, который можно вычислить для ввода функции в контекст, как показано в листинге 9.7, где предполагается, что пакет base2 уже загружен.

Листинг 9.7. Проверка работоспособности импорта из пространства имен base2

```
<script type="text/javascript">

  base2.namespace ==
    "var Base=base2.Base;var Package=base2.Package;" +
    "var Abstract=base2.Abstract;var Module=base2.Module;" +
    "var Enumerable=base2.Enumerable;var Map=base2.Map;" +
    "var Collection=base2.Collection;var RegGrp=base2.RegGrp;" +
    "var Undefined=base2.Undefined;var Null=base2.Null;" +
    "var This=base2.This;var True=base2.True;var False=base2.False;" +
    "var assignID=base2.assignID;var detect=base2.detect;" +
    "var global=base2.global;var lang=base2.lang;" +
    "var JavaScript=base2.JavaScript;var JST=base2.JST;" +
```

Определить импортируемые имена


```
    "var JSON=base2.JSON;var IO=base2.IO;var MiniWeb=base2.MiniWeb;" +
    "var DOM=base2.DOM;var JSB=base2.JSB;var code=base2.code;" +
    "var doc=base2.doc;");

  assert(typeof This === "undefined",
         "The This object doesn't exist." );

  eval(base2.namespace);

  assert(typeof This === "function",
         "And now the namespace is imported." );
  assert(typeof Collection === "function",
         "Verifying the namespace import.");

</script>
```

Проверить состояние до импорта и убедиться, что ни одно из вновь определяемых имен не существует

Вычислить импортированный код

Выборочно проверить состояние после импорта и убедиться, что имена импортированы

Это весьма изобретательное решение столь сложной задачи. И хотя его нельзя назвать самым изящным, нам все же придется обходиться тем, что есть, до тех пор, пока не появятся другие реализации в последующих версиях JavaScript. И если уж дело коснулось изобретательности, то нельзя не упомянуть о еще одном практическом примере вычисления кода: упаковке кода JavaScript, о которой речь пойдет ниже.

Уплотнение и запутывание кода JavaScript

В практике разработки веб-приложений нередко возникает потребность каким-то образом доставить клиентский код на сторону клиента. А это означает, что передачу кода нужно сделать как можно менее заметной. С этой целью можно было бы, конечно, написать код максимально лаконично, но тогда он стал бы неудобочитаемым. Поэтому лучше написать код как можно более ясно, а затем сжать его для последующей передачи.

Для этой цели существует широко распространенное инструментальное средство Packer, разработанное Дином Эдвардсом (Dean Edwards). Оно представляет собой отдельный сценарий, сжимающий код JavaScript, предоставляя в итоге файл JavaScript намного меньшего размера, который может сам разворачиваться и затем выполняться. Это инструментальное средство доступно по адресу: `http://dean.edwards.name/packer`.

В результате применения данного средства получается закодированная строка, которая преобразуется в строку кода JavaScript и затем вычисляется с помощью функции `eval()`. Типичный пример сжатого подобным образом кода JavaScript приведен ниже.

```
eval(function(p,a,c,k,e,r){e=function(c){return(c<a?'':e(
  parseInt(c/a)))+((c=c%a)>35?String.fromCharCode(c+29):
  c.toString(36))};if(!''.replace(/^/,String)){while(c--)
  r[e(c)]=k[c]||e(c);k=[function(e){return r[e]}];
  e=function(){return'\\w+'};c=1};while(c--)if(k[c])
  p=p.replace(new RegExp('\\b'+e(c)+'\\b','g'),k[c]);
  return p}(' // ... длинная строка ...
```

Это, конечно, довольно любопытный способ, но ему присущи некоторые существенные недостатки. В частности, издержки на разархивирование сценария при каждой его загрузке довольно ощутимы.

Что касается распространения фрагментов кода JavaScript, то по традиции считается, что чем меньше размер кода в байтах, тем быстрее он загружается. Но это не всегда справедливо. В самом деле, меньший по объему код может загружаться быстрее, но не всегда *вычисляться* быстрее. А если учесть, что код сначала загружается, а затем вычисляется, то возникает очень важный вопрос производительности веб-страниц. И это приводит нас к следующей простой формуле:

```
время загрузки = время передачи + время вычисления
```

Рассмотрим скорость загрузки библиотеки jQuery в следующих трех формах.

- Обычная (без сжатия).
- Минимизированная, получаемая с помощью уплотнителя YUI Compressor от компании Yahoo!, который удаляет пробелы и выполняет ряд других простых операций сжатия кода.
- Упакованная, получаемая с помощью сценария Packer, написанного Дином Эдвардсом и выполняющего массовую перезапись и разуплотнение кода, используя метод `eval()`.

Все три формы располагаются по степени убывания размера файла в следующем порядке: нормальная, минимизированная и упакованная, и правомерно было бы ожидать, что время загрузки в этих формах окажется пропорциональным размеру файла. Но упакованная форма требует дополнительных издержек на разуплотнение и вычисление кода на стороне клиента. Такая распаковка существенно сказывается на времени загрузки. И в конечном счете оказывается, что минимизированная форма кода обрабатывается намного быстрее, чем упакованная, несмотря на то, что размер ее файла заметно больше.

С результатами проведенного исследования библиотеки jQuery, в ходе которого были проанализированы тысячи загрузок файлов, можно ознакомиться по адресу `http://ejohn.org/blog/library-loading-speed`. А в табл. 9.1 сведены результаты сравнительного анализа скоростей загрузки библиотеки jQuery в трех разных формах.

Таблица 9.1. Результаты сравнительного анализа скоростей загрузки библиотеки jQuery в трех разных формах

Форма	Среднее время загрузки (мс)	Число выборок
Нормальная	645,4818	12589
Минимизированная	519.7214	12611
Упакованная	591,6636	12606

Нельзя сказать, что пользоваться кодом, сжатым инструментальным средством Packer, нецелесообразно, особенно если во главу угла ставится производительность.

Но производительность может не всегда быть главной целью. Несмотря на дополнительные издержки, Packer может оказаться весьма ценным инструментальным средством, если преследуется цель *запутать* код. В отличие от серверного кода, который в разумно защищенном веб-приложении совершенно недоступен со стороны клиента, код JavaScript приходится пересылать на сторону клиента для выполнения. Ведь браузер не в состоянии выполнить код, если не получит его.

В те времена, когда самые сложные сценарии на веб-страницах предназначались для выполнения обыкновенных функций вроде динамических подстановок изображений, никого не беспокоило, что код доставлялся на сторону клиента и был доступен для просмотра кем угодно на приемной стороне. Но в настоящее время, когда появились высоко-функциональные Ajax-страницы и так называемые одностраничные приложения, объем и сложность кода может существенно возрасти. И поэтому в некоторых организациях могут весьма скептически отнестись к перспективе выставления кода на всеобщее обозрение.

Запутывание кода, обеспечиваемое такими сценариями, как Packer, может частично удовлетворить запросы подобных организаций, хотя это и не самое безупречное решение. По крайней мере, Packer может служить неплохим примером применения метода `eval()` для осуществления вычисления кода во время выполнения.

Совет

Если вас интересуют уплотнители кода, попробуйте загрузить YUI Compressor по адресу `http://developer.yahoo.com/yui/compressor/` или Closure Compiler по адресу `https://developers.google.com/closure/compiler/`. Кроме того, компания Yahoo! предоставляет интересные сведения о производительности веб-приложений по адресу `http://developer.yahoo.com/performance/rules.html`.

Перейдем далее к рассмотрению еще одной операции над кодом, к которой можно прибегнуть во время выполнения. В следующем разделе речь пойдет о динамическом переписывании кода.

Динамическое переписывание кода

Как пояснялось ранее в этой главе, отдельные функции JavaScript можно декомпилировать, вызывая их метод `toString()`. А это означает, что можно создавать новые функции, извлекая содержимое старой функции, чтобы сообщить ему новые функциональные возможности. Одним из примеров такого обращения с кодом служит библиотека блочного тестирования Screw.Unit (`https://github.com/nkallen/screw-unit`). В этой библиотеке динамически переписывается содержимое существующих тестовых функций для применения тех функций, которые предоставляются библиотекой. В качестве примера ниже приведен типичный тест из библиотеки Screw.Unit.

```
describe("Matchers", function() {
  it("invokes the provided matcher on a call to expect", function() {
    expect(true).to(equal, true);
    expect(true).to_not(equal, false);
  });
});
```

Обратите внимание на методы `describe()`, `it()` и `expect()`, отсутствующие в глобальной области действия. Для преодоления этого ограничения в библиотеке Screw.Unit динамически *переписывается* приведенный выше код, чтобы заключить все функции в нескольких операторах `with(){}`, о которых речь пойдет в главе 10, вставляя содержимое в функции по мере потребности в их выполнении, как показано ниже.

```
var contents = fn.toString().match(/^[^{]*{((.*\n*)*)}/m)[1];
var fn = new Function("matchers", "specifications",
```

```
  "with (specifications) { with (matchers) { " + contents + " } }"
);

fn.call(this, Screw.Matchers, Screw.Specifications);
```

В данном случае вычисление кода используется для того, чтобы упростить взаимодействие с конечным пользователем, которым является составитель тестов, но без отрицательных последствий вроде внедрения многих переменных в глобальной области действия. Рассмотрим далее очередной термин, широко обсуждавшийся в последние несколько лет в среде пишущих серверный код. Речь пойдет об аспектно-ориентированном программировании и его особенностях применительно к написанию сценариев на JavaScript.

Аспектно-ориентированные дескрипторы сценариев

Аспектно-ориентированное программирование (АОП) определяется в Википедии как "парадигма программирования, основанная на идее разделения сквозной функциональности для улучшения разбиения программы на модули". От такого определения голова может пойти кругом. А проще говоря, АОП — это методика внесения и выполнения кода в динамическом режиме для решения таких "сквозных", т.е. взаимно пересекающихся, задач, как регистрация, кеширование, обеспечение безопасности и т.д. Вместо того чтобы обременять код цепочкой операторов регистрации, механизм АОП вводит код регистрации во время выполнения, избавляя программирующего от необходимости заниматься этим во время разработки.

Совет

Подробнее с принципами АОП можно познакомиться, прочитав статью в Википедии по адресу `http://en.wikipedia.org/wiki/Aspect-oriented_programming`[1]. А если вас интересует применение принципов АОП в Java, рекомендуем прочитать книгу *AspectJ in Action* (Практика АОП в Java) Рамниваса Ладдада (Ramnivas Laddad; `www.manning.com/laddad2/`).

Внесение и вычисление кода во время выполнения вполне соответствует теме этой главы. Поэтому покажем, каким образом можно воспользоваться принципами АОП с наибольшей выгодой.

Примечание

Возможно, вы еще не забыли, что в разделе "Переопределение поведения функции" главы 5 рассматривался пример запоминания. Это был характерный пример применения принципов АОП в JavaScript. Таким образом, вы уже пользовались ими, даже не осознавая этого!

Как обсуждалось ранее, дескрипторы сценариев с недействительными атрибутами типов применяются в качестве средства для включения на странице новых фрагментов данных, которых не должен касаться браузер. Этот подход можно расширить дальше, чтобы с его помощью усовершенствовать существующий сценарий JavaScript. Допустим, что по той или иной причине требуется создать новый тип сценария под названием `onload`. А что такое новый *тип* сценария и как это вообще можно сделать?

[1] Эта статья доступна на русском языке после выбора ссылки Русский слева на веб-странице, открывшейся по указанному адресу. — *Примеч. ред.*

Оказывается, что определить типы специальных сценариев совсем не трудно, поскольку браузеры будут игнорировать любой тип сценария, который им непонятен. В частности, браузер можно заставить полностью проигнорировать блок сценария, чтобы использовать его в каких угодно целях, применяя для этого значение нестандартного типа. Так, если требуется создать новый тип `onload`, для этого достаточно указать блок сценария следующим образом:

```
<script type="x/onload"> ... здесь следует специальный сценарий ... </script>
```

Следует иметь в виду, что данном случае **x** обозначает "специальный". Подобные блоки создаются намеренно, чтобы содержать обычный код JavaScript, который будет выполняться всякий раз, когда загружается страница, а не, как обычно, по порядку. В качестве примера рассмотрим код, приведенный в листинге 9.8.

Листинг 9.8. Создание дескриптора типа сценария для выполнения только после загрузки страницы

```
<script type="text/javascript">

  window.onload = function(){
    var scripts = document.getElementsByTagName("script");   ❶ Найти все блоки сценария
    for (var i = 0; i < scripts.length; i++) {
      if (scripts[i].type == "x/onload") {
        globalEval(scripts[i].innerHTML);   ❷ Найти и выполнить блоки типа "x/onload"
      }
    }
  };

</script>

<script type="x/onload">
  assert(true,"Executed on page load");   ❸ Предоставить специальный сценарий
</script>
```

В приведенном выше примере кода предоставляется блок специального сценария ❸, игнорируемый браузером. В обработчике `onload` загрузки страницы обнаруживаются все блоки сценариев ❶. И как только будет найден любой блок сценария специального типа, вызывается функция `globalEval()`, рассматривавшаяся ранее в этой главе и вычисляющая код сценария в глобальном контексте ❷.

Данный пример довольно прост, тем не менее он наглядно демонстрирует способ, который можно успешно применять для решения более сложных и практических задач. В частности, блоки специальных сценариев применяются в методе `tmpl()` из библиотеки jQuery для предоставления шаблонов во время выполнения. Это дает возможность выполнять сценарии по взаимодействию с пользователем или в том случае, когда модель DOM подготовлена для манипулирования, или даже относительно соседних элементов разметки. Таким образом, применение данного способа ограничивается лишь воображением автора веб-страницы. А теперь рассмотрим еще один передовой пример вычисления кода во время выполнения.

Метаязыки и предметно-ориентированные языки

Наиболее значительным примером эффективности вычисления кода может служить реализация других языков программирования в дополнение к языку JavaScript. Код этих так называемых *метаязыков* может быть динамически преобразован в исходный код JavaScript и затем вычислен. Зачастую такие языки предназначаются для решения специальных задач во время разработки, и поэтому для их обозначения был придуман особый термин *предметно-ориентированный язык* (DSL). Особый интерес представляют две разновидности предметно-ориентированных языков, которые рассматриваются ниже.

Processing.js

Это перенесенный вариант языка визуального представления обработки (Processing Visualization Language; подробнее см. по адресу `http://processing.org/`), который обычно реализуется средствами Java. Но в данном случае он перенесен Джоном Резигом на JavaScript для выполнения в элементе `Canvas` (Холст) разметки документов по стандарту HTML 5.

В итоге получился полноценный язык программирования, который обычно позволяет манипулировать визуальным представлением области расположения рисунка. Безусловно, язык Processing.js особенно хорошо приспособлен для решения подобных задач и эффективно переносится на другую платформу. Ниже приведен пример кода Processing.js, реализующего блок сценария типа `"application/processing"`.

```
<script type="application/processing">
class SpinSpots extends Spin {
  float dim;
  SpinSpots(float x, float y, float s, float d) {
    super(x, y, s);
    dim = d;
  }

  void display() {
    noStroke();
    pushMatrix();
    translate(x, y);
    angle += speed;
    rotate(angle);
    ellipse(-dim/2, 0, dim, dim);
    ellipse(dim/2, 0, dim, dim);
    popMatrix();
  }
}
</script>
```

Приведенный выше код Processing.js преобразуется в код JavaScript, который затем вычисляется с помощью вызываемого метода `eval()`. Получающийся в итоге код JavaScript выглядит следующим образом:

```
function SpinSpots() {with(this){
  var __self=this;function superMethod(){
  extendClass(__self,arguments,Spin);
```



```
  this.dim = 0;
  extendClass(this, Spin);
  addMethod(this, 'display', function() {
    noStroke();
    pushMatrix();
    translate(x, y);
    angle += speed;
    rotate(angle);
    ellipse(-dim/2, 0, dim, dim);
    ellipse(dim/2, 0, dim, dim);
    popMatrix();
  });
  if ( arguments.length == 4 ) {
    var x = arguments[0];
    var y = arguments[1];
    var s = arguments[2];
    var d = arguments[3];
    superMethod(x, y, s);
    dim = d;
  }
}}
```

Для того чтобы подробно описать преобразование кода метаязыка в код JavaScript, потребовалась бы отдельная глава, а возможно, и целая книга, поэтому данный вопрос выходит за рамки этой книги. К тому же это довольно непростой и запутанный вопрос, которым интересуются "посвященные", а не простые смертные. Но зачем вообще пользоваться метаязыками? Применение такого метаязыка, как Processing.js, дает следующие преимущества по сравнению с JavaScript.

- Выгоды от использования развитых средств Processing.js, включая классы и наследование.
- Простой прикладной интерфейс API для рисования средствами Processing.js.
- Полная документация и демонстрационные примеры на Processing.js.

Дополнительные сведения о применении метаязыка Processing.js можно найти по адресу `http://ejohn.org/blog/processingjs/`.

Из приведенного выше можно сделать следующий важный вывод: вся усовершенствованная подобным образом обработка оказывается возможной благодаря языковым средствам вычисления кода в JavaScript. Рассмотрим далее еще один интересный проект реализации метаязыка.

Objective-J

Еще одним значительным проектом по переносу языка программирования Objective-C на платформу JavaScript стал Objective-J, разработанный компанией 280 North для программного продукта 280 Slides (формирователя слайд-шоу в оперативном режиме).

У специалистов из компании 280 North накопился большой опыт разработки приложений на платформе Mac OS X, написанных в основном на языке Objective-C. Для того чтобы создать среду, более пригодную для работы, они перенесли язык Objective-C на платформу JavaScript. Помимо тонкого слоя, накладываемого на язык JavaScript,

в Objective-J допускается также сочетание кода JavaScript с кодом Objective-C. J. Ниже приведен характерный тому пример.

```
// DocumentController.j
// Редактор
//
// Создан Франсиско Толмаски.
// Copyright 2005 - 2008, 280 North, Inc. All rights reserved.

import <AppKit/CPDocumentController.j>
import "OpenPanel.j"
import "Themes.j"
import "ThemePanel.j"
import "WelcomePanel.j"

@implementation DocumentController : CPDocumentController
{
  BOOL _applicationHasFinishedLaunching;
}

- (void)applicationDidFinishLaunching:(CPNotification)aNotification
{
  [CPApp runModalForWindow:[[WelcomePanel alloc] init]];
  _applicationHasFinishedLaunching = YES;
}

- (void)newDocument:(id)aSender
{
  if (!_applicationHasFinishedLaunching)
    return [super newDocument:aSender];

  [[ThemePanel sharedThemePanel]
    beginWithInitialSelectedSlideMaster:SaganThemeSlideMaster
      modalDelegate:self
        didEndSelector:@selector(themePanel:didEndWithReturnCode:)
          contextInfo:YES];
}

- (void)themePanel:(ThemePanel)aThemePanel
didEndWithReturnCode:(unsigned)aReturnCode
{
  if (aReturnCode == CPCancelButton)
    return;

  var documents = [self documents],
    count = [documents count];

  while (count--)
    [self removeDocument:documents[0]];

  [super newDocument:self];
}
```


В приложении Objective-J для синтаксического анализа, написанном на JavaScript и динамически преобразующем код Objective-J во время выполнения, применяются облегченные выражения для согласования и интерпретации синтаксиса Objective-C без нарушения существующего синтаксиса JavaScript. В итоге получается символьная строка с кодом JavaScript, которая вычисляется во время выполнения.

Такая реализация дает меньше преимуществ в перспективе, поскольку ее можно применять только в данном конкретном контексте. Тем не менее она сулит определенные выгоды тем пользователям, которые уже знакомы с Objective-C, но стремятся освоить программирование для разработки веб-приложений.

Резюме

В этой главе рассмотрены основы вычисления кода в JavaScript во время выполнения. В частности, в ней обсуждались следующие вопросы.

- Различные механизмы, предоставляемые в JavaScript для вычисления кода из символьных строк во время выполнения. В том числе:
  - метод `eval()`;
  - функции-конструкторы;
  - таймеры;
  - динамические блоки сценариев типа `<script>`.
- Средства, предоставляемые в JavaScript для выполнения обратного процесса — получения символьной строки исходного кода функции с помощью ее метода `toString()`. Этот процесс называется *декомпиляцией функции*.
- Различные практические примеры вычисления кода во время выполнения, включая такие:
  - преобразование из формата JSON;
  - перенос кода определяемых функций из одного пространства имен в другое;
  - уплотнение и запутывание кода JavaScript;
  - динамическое переписывание и внесение кода;
  - и даже создание метаязыков.

Существует вероятность злоупотребления перечисленными выше эффективными языковыми средствами. Тем не менее их потенциальные возможности, позволяющие поставить вычисление кода себе на службу, превращают их в отличные инструментальные средствами, которыми должен овладеть всякий, стремящийся стать мастером программирования на JavaScript.

10 Операторы with

В этой главе...

- Причины противоречивости операторов `with`
- Принцип действия операторов `with`
- Упрощение кода с помощью операторов `with`
- Особо скрытые препятствия на пути применения операторов `with`
- Построение по шаблону с использованием операторов `with`

Оператор `with` является весьма эффективным и зачастую недооцениваемым, хотя и противоречивым языковым средством JavaScript. Он позволяет размещать все свойства объекта в пределах текущей области действия, чтобы обращаться к ним и присваивать им значения, *не* предваряя их ссылкой на объект, к которому они принадлежат. Следует, однако, иметь в виду, что перспективы дальнейшего присутствия данного оператора в JavaScript довольно призрачны. Так, в спецификации ECMAScript 5 запрещается его применение в строгом режиме, причем до такой степени, что оно будет считаться синтаксической ошибкой.

Более того, еще до появления спецификации ECMAScript 5 применение оператора `with` имело своих явных противников и скептиков. К их числу относится Дуглас Крокфорд, известный мастер программирования на JavaScript, изобретатель формата JSON и автор книги *JavaScript: The Good Parts* (JavaScript: сильные стороны). В своей широко известной статье "`with` Statement Considered Harmful" (Оператор `with`, который

следует считать вредным), опубликованной в 2006 году в блоге по библиотеке YUI, он заявляет следующее.

> *"Если вы не в состоянии прочитать программу и не знаете точно, для чего она предназначена, значит, вы не можете быть уверены в том, что она будет работать правильно. Именно поэтому следует избегать применения оператора* `with`*".*
>
> *Дуглас Крокфорд, апрель 2006 г.* (`http://yuiblog.com/blog/2006/04/11/with-statement-considered-harmful/`)

И г-н Крокфорд оказался в этом не одинок. Во многих редакторах исходного кода JavaScript и интегрированных средах разработки (IDE) употребление оператора `with` в коде сопровождается предупреждениями, не рекомендующими пользоваться им. Так зачем его вообще обсуждать? Дело в том, что существует еще немало унаследованного кода, в котором применяется оператор `with`, и вам такой код вполне может встретиться. Поэтому вам нужно хотя бы иметь представление об операторе `with`, даже если вы не собираетесь употреблять его в новом коде. Принимая все это во внимание, зададимся целью разобраться в особенностях применения оператора `with`.

Особенности применения оператора `with`

Оператор `with` создает область действия, в пределах которой ссылаться на свойства указанного объекта можно без специального префикса. Как будет показано далее в главе, оператором `with` можно пользоваться в самых разных целях, включая следующее.

- Сокращение ссылок на объект, находящийся глубоко в иерархии.
- Упрощение тестового кода.
- Раскрытие свойств в качестве ссылок самого верхнего уровня на шаблон.
- И многое другое.

Рассмотрим все это по порядку.

Организация ссылок на свойства объекта в области действия оператора `with`

Прежде всего следует разобраться в принципе действия оператора `with`. С этой целью обратимся к примеру кода, приведенного в листинге 10.1.

Листинг 10.1. Формирование области действия оператора `with` с помощью объекта

```
<script type="text/javascript">

  var use = "other";                    ❶ Определить переменную верхнего уровня

  var katana = {                        ❷ Создать объект
    isSharp: true,
      use: function(){
      this.isSharp = !this.isSharp;
    }
  };
```



```
  with (katana) {                       ❸ Установить область действия оператора with

    assert(use !== "other" && typeof use == "function",   ❹ Выполнить проверку в области действия
           "use is a function from the katana object.");
    assert(this !== katana,
           "context isn't changed; keeps its original value");
  }
                                        ❺ Выполнить проверку вне области действия
  assert(use === "other",
         "outside the with use is unaffected.");
  assert(typeof isSharp === "undefined",
         "outside the with the properties don't exist.");

</script>
```

В приведенном выше примере кода свойства объекта `katana` вводятся в область действия, образованную оператором `with`. В этой области действия на свойства можно ссылаться непосредственно, т.е. без префикса `katana`, как будто они являются переменными и методами самого верхнего уровня. Для того чтобы убедиться в этом, в рассматриваемом здесь коде сначала определяется переменная верхнего уровня `use` ❶. Затем создается встраиваемый объект с одноименным свойством `use` и еще одним свойством `isSharp` ❷. Ссылка на этот объект делается в переменной `katana`.

Самое интересное начинается тогда, когда область действия оператора `with` устанавливается с помощью объекта `katana` ❸. В пределах этой области действия ссылаться на свойства объекта `katana` можно непосредственно, опуская префикс `katana`. Для этого выполняется тест ❹, в котором проверяется, что по имени `use` делается ссылка не на переменную верхнего уровня, а на свойство объекта `katana`, представляющее функцию. Это все равно, что ссылаться по имени `use` на метод `katana.use()`. Как показано на рис. 10.1, тест, утверждающий этот факт, проходит.

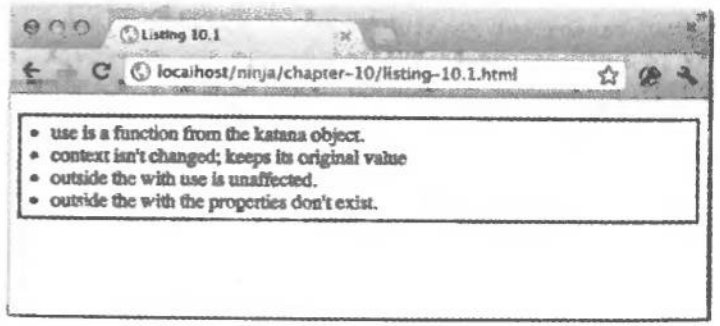

Рис. 10.1. С помощью оператора `with` *можно разрешать простые ссылки на свойства объекта*

Далее тестирование продолжается за пределами области действия оператора `with` ❺. При этом проверяется, что по имени `use` делаются ссылки на переменную верхнего уровня и что свойство `isSharp` уже недоступно. Следует, однако, иметь в виду, что в области действия оператора `with` свойства объекта получают абсолютный приоритет над одноименными переменными, определенными в областях действия более

высокого уровня. И это одна из главных причин, по которым оператор `with` не воспринимается всерьез. Ведь код в области его действия приобретает неоднозначный смысл.

Кроме того, тестирование в данном примере подтвердило, что оператор `with` не оказывает влияния на контекст функции (`this`). Итак, мы рассмотрели возможность непосредственного обращения к свойствам объекта с помощью оператора `with` для чтения из них. А как насчет записи в них?

Присваивание в области действия оператора `with`

Рассмотрим присваивание в области действия оператора `with`. С этой целью обратимся к примеру кода, приведенного в листинге 10.2.

Листинг 10.2. Присваивание в области действия оператора `with`

```
<script type="text/javascript">

  var katana = {                    ❶ Создать объект
    isSharp: true,
    use: function(){
    this.isSharp = !this.isSharp;
   }
  };

  with (katana) {                   ❷ Присвоить значение существующему свойству
    isSharp = false;

    assert(katana.isSharp === false,   ❸ Проверить присваивание
          "properties can be assigned");
    cut = function(){                  ❹ Попытаться создать новое свойство
      isSharp = false;
    };

    assert(typeof katana.cut == "function",   ❺ Проверить присваивание
          "new properties can be created on the scoped object");
    assert(typeof window.cut == "function",
          "new properties are created in the global scope");
  }

</script>
```

В приведенном выше примере кода создается тот же самый объект `katana` со свойствами `use` и `isSharp`, что и в предыдущем примере ❶, и с его помощью снова формируется область действия оператора `with`. Но вместо ссылок на свойства данного объекта на этот раз предпринимается попытка присвоить им некоторые значения. Сначала логическое значение `false` присваивается свойству `isSharp` ❷. Если имя `isSharp` разрешается как ссылка на одноименное свойство объекта `katana`, то можно ожидать, что исходное логическое значение этого свойства изменится с `true` на `false`. Данное свойство проверяется в тесте явным образом ❸, и этот тест проходит, как показано на рис. 10.2. Он лишний раз доказывает, что беспрефиксные ссылки на свойства объектов можно делать как для чтения, так и для присваивания значений.

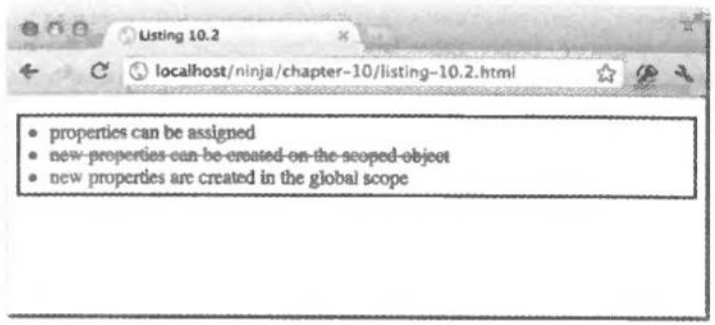

Рис. 10.2. Как показывают результаты тестирования, беспрефиксные ссылки нельзя использовать для создания новых свойство объектов

Затем в рассматриваемом здесь коде предпринимается попытка сделать нечто более сложное, а именно: определить функцию и присвоить ее новому свойству `cut` объекта `katana` ❹. В связи с этим возникает следующий вопрос: в какой именно области действия будет создано новое свойство: объекта `katana`, потому что присваивание осуществляется в области действия оператора `with`, или же в глобальной области действия объекта `window`, которая, как и следует ожидать, находится за пределами действия любого оператора `with`?

Для того чтобы выяснить, что же на самом деле происходит, в рассматриваемом здесь коде выполняются еще два теста ❺, но только один из них проходит. В первом тесте утверждается, что свойство будет создано в области действия объекта `katana`, а во втором, — что оно будет создано в глобальной области действия. На рис. 10.2 наглядно показано, что второй тест проходит, а следовательно, присваивание без ссылок производится в глобальном контексте, а не по отношению к существующему свойству объекта в области действия оператора `with`.

Если бы потребовалось создать новое свойство объекта `katana`, для этого пришлось бы употребить префикс в ссылке на этот объект — даже находясь в области действия оператора `with`, как показано ниже.

```
katana.cut = function(){
  isSharp = false;
};
```

Данный пример наглядно показывает, насколько просто можно обойти ограничения области действия оператора `with`. Но в то же время подобный обходной прием нивелирует прежде всего назначение такой области действия. Тем не менее это лишний раз доказывает, что обращаться с областью действия оператора `with` нужно очень внимательно и аккуратно, поскольку даже элементарная опечатка при наборе имени свойства может привести к необычным и неожиданным результатам. В частности, новая глобальная переменная может быть введена вместо требующегося видоизменения существующего свойства объекта в области действия оператора `with`. Разумеется, это лишь самые общие соображения, предусматривающие тщательное тестирование кода. А что еще следует иметь в виду, употребляя операторы `with`?

Соображения по поводу производительности

Есть еще один важный фактор, который следует иметь в виду, употребляя операторы `with`. Дело в том, что оператор `with` замедляет выполнение любого кода JavaScript,

который он объемлет. И это положение распространяется не только на объекты, взаимодействующие с данным оператором. В качестве примера рассмотрим тесты на производительность, приведенные в листинге 10.3.

Листинг 10.3. Тестирование производительности оператора `with`

```
<script type="text/javascript">

  var ninja = { foo: "bar" },                    ❶ Установить ряд переменных
    value,
    maxCount = 1000000,
    n,
    start,
    elapsed;

  start = new Date().getTime();                  ❷ Проверить без оператора with
  for (n = 0; n < maxCount; n++) {
    value = ninja.foo;
  }
  elapsed = new Date().getTime() - start;
  assert(true,"Without with: " + elapsed);

  start = new Date().getTime();                  ❸ Проверить правильность ссылок
  with(ninja){
    for (n = 0; n < maxCount; n++) {
      value = foo;
    }
  }
  elapsed = new Date().getTime() - start;
  assert(true,"With (with access): " + elapsed);

  start = new Date().getTime();                  ❹ Проверить правильность присваивания
  with(ninja){
    for (n = 0; n < maxCount; n++) {
      foo = n;
    }
  }
  elapsed = new Date().getTime() - start;
  assert(true,"With (with assignment): " + elapsed);

  start = new Date().getTime();                  ❺ Проверить без доступа
  with(ninja){
    for (n = 0; n < maxCount; n++) {
      value = "no test";
    }
  }
  elapsed = new Date().getTime() - start;
  assert(true,"With (without access): " + elapsed);

</script>
```

Для выполнения приведенных выше тестов сначала устанавливается ряд переменных, включая и переменную `ninja`, которая послужит целевым объектом для области

действия оператора `with` ❶. Затем выполняются четыре теста на производительность. В каждом из этих тестов одно и то же действие повторяется миллион раз, после чего отображаются его результаты.

- В первом тесте значение свойства `ninja.foo` присваивается переменной `value` без какого-либо объявления области действия оператора `with` ❷.
- Во втором тесте выполняется такое же присваивание, как и в первом тесте, за исключением того, что присваивание происходит в области действия оператора `with`, а ссылка на свойство `foo` делается без префикса его объекта ❸.
- В третьем тесте свойству `foo` присваивается значение счетчика цикла в области действия оператора `with`, а ссылка на это свойство делается без префикса его объекта ❹.
- И в последнем, четвертом тесте переменой `value` присваивается значение в области действия оператора `with`, но без всякого доступа к объекту `ninja` ❺.

Результаты выполнения этих тестов приведены в табл. 10.1. Все тесты выполнялись в перечисленных браузерах на переносном компьютере MacBook Pro под ОС Mac OS X Lion 10.7.3, с процессором Core i7 на 2,8 МГц и ОЗУ на 8 Мбайт. А тестирование в браузере Internet Explorer выполнялось в ОС Windows 7, работавшей на виртуальной машине Parallels. Все временные характеристики приведены в миллисекундах. Результаты тестирования впечатляют и могут даже удивить. Их временные характеристики не только разительно отличаются по браузерам, что и так понятно с точки зрения производительности JavaScript, но и между собой.

Таблица 10.1. Результаты выполнения тестов на производительность из листинга 10.3. Время указано в миллисекундах

Браузер	Вне области действия оператора `with`	В области действия оператора `with`, ссылка	В области действия оператора `with`, присваивание	В области действия оператора `with`, без доступа к объекту
Chrome 21	87	1456	1395	1282
Safari 5.1	6	264	308	279
Firefox 14	256	717	825	648
Opera 11	13	677	679	623
Internet Explorer 9	13	173	157	139

Независимо от типа браузера, в котором производились тесты, код в областях действия операторов `with` выполнялся значительно медленнее. И это может быть и не так удивительно для тестов, в которых делалась ссылка или присваивалось значение свойству объекта в области действия оператора `with`. Но обратите внимание на временные характеристики в крайнем справа столбце для теста, выполнявшегося вообще без доступа к объекту. Одного лишь факта, что код выполнялся в области действия оператора `with`, оказалось достаточно, чтобы его выполнение замедлилось в 41 раз, несмотря на полное отсутствие доступа к объекту в этой области действия!

Принимая решение воспользоваться любыми удобствами, доставляемыми оператором `with`, следует оценить, насколько приемлемыми окажутся связанные с этим дополнительные издержки. И очевидно, что применение оператора `with` в прикладном коде, где во главу угла ставится производительность, совершенно неуместно.

Практические примеры употребления оператора with

Едва ли не самой распространенной причиной для применения оператора with служит возможность исключить дублирование ссылок на переменные для доступа к свойствам объектов. Эта мера нередко применяется в библиотеках JavaScript для упрощения операторов, которые в противном случае получаются слишком громоздкими и сложными. Ниже приведен ряд примеров кода из основных библиотек JavaScript, начиная с Prototype.

```
Object.extend(String.prototype.escapeHTML, {
  div:  document.createElement('div'),
  text: document.createTextNode('')
  });

with (String.prototype.escapeHTML) div.appendChild(text);
```

В данном примере с помощью оператора with удается избежать указания длинного префикса String.prototype.escapeHTML в ссылках на свойства div и div. Ведь в противном случае в подобных ссылках пришлось бы указывать слишком много префиксов. Но насколько действительно необходимо употреблять оператор with для этой цели? Можно ли достичь ее каким-нибудь другим путем, из тех, что рассматривались ранее в этой книге, не прибегая к области действия оператора with? Рассмотрим следующий фрагмент кода:

```
(function(s){
  s.div.appendChild(s.text);
})(String.prototype.escapeHTML);
```

Это уже знакомая вам немедленно вызываемая функция! В области действия этой функции длинную ссылку String.prototype.escapeHTML можно заменить на короткую ссылку s через параметр функции. И хотя это не одно и то же, что и область действия оператора with, поскольку в данном случае префикс не исключается, а просто заменяется более короткой ссылкой, тем не менее многие опытные разработчики согласятся, что замена сложной ссылки на простую намного эффективнее пассивно-агрессивного способа полного исключения префикса. А если учесть, что оператор with находится по угрозой полного исчезновения из JavaScript, то применение немедленно вызываемых функций позволяет заменять сложные ссылки языковыми конструкциями, которые совершенно понятны и имеют неплохие перспективы для поддержки в будущем.

Ниже приведен другой пример употребления оператора with — на этот раз из библиотеки base2 для JavaScript.

```
with (document.body.style) {
  backgroundRepeat = "no-repeat";
  backgroundImage =
    "url(http://ie7-js.googlecode.com/svn/trunk/lib/blank.gif)";
backgroundAttachment = "fixed";
}
```

В данном фрагменте кода из библиотеки base2 оператор with употребляется как простое средство, исключающее многократное повторение длинного префикса document.body.style. Благодаря этому достигается упрощенное до предела видоизменение объекта стилевого оформления элемента модели DOM. А вот еще один пример из той же библиотеки base2:


```
var Rect = Base.extend({
  constructor: function(left, top, width, height) {
    this.left = left;
    this.top = top;
    this.width = width;
    this.height = height;
    this.right = left + width;
    this.bottom = top + height;
  },

contains: function(x, y) {
  with (this)
    return x >= left && x <= right && y >= top && y <= bottom;
},

toString: function() {
  with (this) return [left, top, width, height].join(",");
  }
});
```

В этом, втором примере из библиотеки base2 оператор with применяется как средство простого доступа к свойствам экземпляра объекта. Как правило, такой код получается намного более длинным, но его можно сократить с помощью оператора with, который в данном случае вносит в код столь необходимую ясность.

И последний, приведенный ниже пример взят из расширения Firebug браузера Firefox для разработчиков.

```
const evalScriptPre = "with(__scope__.vars){ with(__scope__.api){" +
  " with(__scope__.userVars){ with(window){";
const evalScriptPost = "}}}}";
```

Эти строки кода из расширения Firebug служат особенно сложным примером применения оператора with в общедоступном фрагменте кода. В данном случае речь идет о применении оператора with в отладочной части расширения Firebug, где пользователю предоставляется доступ к локальным переменным, прикладному интерфейсу API и глобальному объекту, причем все это доступно из консоли JavaScript. Подобные операции обычно не выполняются в большинстве приложений, но они наглядно демонстрируют потенциальные возможности оператора with для упрощения фрагментов сложного кода на JavaScript.

В приведенном выше примере кода из расширения Firebug особое внимание привлекает двукратное применение оператора with, обеспечивающее предшествование объекта window другим вводимым объектам. Как было показано в примере кода из листинга 10.1, при возникновении конфликта имен объект из области действия оператора with обычно имеет приоритет над значениями из глобальной области действия. Структурирование кода следующим образом:

```
with ( x ) { with ( window ) { ... } }
```

позволяет ввести свойства объекта x с помощью оператора `with`, отдавая в то же время исключительный приоритет глобальным переменным при возникновении конфликта имен. А теперь рассмотрим еще одну область применения оператора `with`.

Импорт кода из пространства имен

Как было показано ранее, оператор `with` чаще всего применяется для упрощения уже существующих операторов, в которых имеются многочисленные ссылки на свойства объекта. Подобное чаще всего происходит в коде, размещаемом в пространстве имен, где одни объекты оказываются внутри других объектов, обеспечивая тем самым организованную структуру кода. Побочным следствием такого подхода становится неприятная необходимость набирать каждый раз вручную имена пространств имен объектов.

В приведенном ниже фрагменте кода обе оператора выполняют одну и ту же операцию, используя библиотеку YUI для JavaScript. Запись первого оператора выглядит как обычно, а в записи второго употребляется оператор `with`.

```
YAHOO.util.Event.on(
  [YAHOO.util.Dom.get('item'), YAHOO.util.Dom.get('otheritem')],
  'click', function(){
    YAHOO.util.Dom.setStyle(this,'color','#c00');
  }
);

with ( YAHOO.util.Dom ) {
  YAHOO.util.Event.on([get('item'), get('otheritem')], 'click',
    function(){ setStyle(this,'color','#c00'); });
}
```

Благодаря вводу единственного оператора `with` код в данном примере существенно упрощается. А теперь попробуем выяснить, как себя проявляет оператор `with` во время тестирования.

Тестирование

При тестировании отдельных функциональных возможностей в тестовом наборе необходимо внимательно следить за некоторыми моментами. Главный из них заключается в синхронизации методов утверждения и выполняемого в настоящий момент контрольного примера. Как правило, это не вызывает особых затруднений, но становится особенно хлопотным, когда приходится иметь дело с асинхронными тестами. Распространенным выходом из этого затруднительного положения является создание центрального объекта, отслеживающего выполнение каждого теста. Модуль выполнения тестов, применяемый в библиотеках Prototype и script.aculo.us, действует именно по этому образцу, предоставляя упомянутый центральный объект в качестве контекста для каждого прогона текста. Сам же этот объект содержит все необходимые методы утверждения, чтобы без особого труда накапливать результаты в одном центральном месте. Ниже приведен характерный тому пример.

```
new Test.Unit.Runner({
  testSliderBasics: function(){with(this){
    var slider = new Control.Slider('handle1', 'track1');
    assertInstanceOf(Control.Slider, slider);
    assertEqual('horizontal', slider.axis);
    assertEqual(false, slider.disabled);
    assertEqual(0, slider.value);
    slider.dispose();
  }},
  // ...
});
```


Обратите внимание на применение оператора `with(this)` в приведенном выше примере теста. Переменная экземпляра содержится во всех методах утверждения (`assertInstanceOf`, `assertEqual` и т.д.). Конечно, вызовы этих методов можно было бы указать и явным образом, как, например, `this.assertEqual`. Но, используя оператор `with(this)` для ввода только тех методов, которые требуется использовать, можно добиться значительного упрощения кода.

Перейдем далее к более продвинутому применению оператора `with`, о котором вы, возможно, и не задумывались. В следующем разделе речь пойдет о построении по шаблону.

Построение по шаблону с помощью оператора `with`

Последним и, вероятно, самым примечательным примером применения оператора `with` служит упрощенная система построения по шаблону. При создании такой системы обычно преследуются следующие цели:

- Предоставить способ как для выполнения встроенного кода, так и для вывода данных.
- Снабдить средствами для кеширования скомпилированных шаблонов.
- А самое главное: обеспечить простой доступ к преобразованным данным.

Для достижения именно последней цели особенно полезным оказывается оператор `with`. Но прежде чем переходить непосредственно к применению оператора `with` в реализации системы построения по шаблону, рассмотрим шаблон, применяемый в подобной системе и представленный в листинге 10.4.

Листинг 10.4. Пример шаблона для формирования HTML-страницы

```
<html>
<head>
  <script type="text/tmpl" id="colors">
    <p>Here's a list of <%= items.length %> items:</p>
    <ul>
      <% for (var i = 0; i < items.length; i++) { %>
        <li style='color:<%= colors[i % colors.length] %>'>
          <%= items[i] %></li>
      <% } %>
    </ul>
      и еще......
```

```
  </script>
  <script type="text/tmpl" id="colors2">
    <p>Here's a list of <%= items.length %> items:</p>
    <ul>
      <% for (var i = 0; i < items.length; i++) {
        print("<li style='color:", colors[i % colors.length], "'>",
          items[i], "</li>");
      } %>
    </ul>
  </script>
  <script type="text/javascript" src="tmpl.js"></script>
  <script type="text/javascript">
    var colorsArray = ['red', 'green', 'blue', 'orange'];

    var items = [];
    for (var i = 0; i < 10000; i++) {
      items.push( "test" );
    }

    function replaceContent(name) {
      document.getElementById('content').innerHTML =
        tmpl(name, {colors: colorsArray, items: items});
    }
  </script>
</head>
<body>
  <input type="button" value="Run Colors"
    onclick="replaceContent('colors')">
  <input type="button" value="Run Colors2"
    onclick="replaceContent('colors2')">
  <p id="content">Replaced Content will go here</p>
</body>
</html>
```

В данном шаблоне применяются специальные ограничители, чтобы отделить встраиваемый код JavaScript (<% и %>) от вычисляемых выражений (<%= и %>). Опытные программирующие JavaScript, вероятно, узнают в них ограничители старого типа, применявшиеся в технологии JSP 1 для построения веб-страниц по шаблону (при переходе на технологию JSP 2 в 2002 году они были заменены на более современные). А теперь рассмотрим пример реализации системы построения по шаблону, приведенный в листинге 10.5.

Листинг 10.5. Простая система построения по шаблону с использованием оператора `with`

```
(function(){
  var cache = {};

    this.tmpl = function tmpl(str, data){

    var fn = !/\W/.test(str) ?
      cache[str] = cache[str] ||
```

Выяснить, имеется ли шаблон или его нужно загрузить и непременно кешировать результат


```
        tmpl(document.getElementById(str).innerHTML) :

      new Function("obj",
        "var p=[],print=function(){p.push.apply(p,arguments);};" +
        "with(obj){p.push('" +

        str
          .replace(/[\r\t\n]/g, " ")
          .split("<%").join("\t")
          .replace(/((^|%>)[^\t]*)'/g, "$1\r")
          .replace(/\t=(.*?)%>/g, "',$1,'")
          .split("\t").join("');")
          .split("%>").join("p.push('")
          .split("\r").join("\\'")
      + "');}return p.join('');");

    return data ? fn( data ) : fn;
  };
})();

assert( tmpl("Hello, <%= name %>!", {name: "world"}) ==
      "Hello, world!", "Do simple variable inclusion." );

var hello = tmpl("Hello, <%= name %>!");
assert( hello({name: "world"}) == "Hello, world!",
      "Use a pre-compiled template." );
```

Создать функцию, кешируемую и повторно используемую в качестве генератора шаблонов

❶ Ввести данные в локальных переменных, используя область действия оператора with

Преобразовать шаблон в чистый код JavaScript

Предоставить пользователю возможность для элементарного каррирования

Мы не будем глубоко вдаваться в подробности реализации данной системы построения по шаблону. И хотя в ней не применяется ничего такого, чего еще не было рассмотрено в этой книге, тем не менее она построена довольно сложным способом. Но не смущайтесь, если вы пока еще не понимаете до конца, как эта система построена. Важнее понять другое: каким образом область действия оператора `with` используется для предоставления свойств информационного объекта, передаваемого шаблону ❶. Ведь благодаря именно этому ссылки на свойства информационного объекта можно делать в шаблоне, как на переменные самого верхнего уровня.

Какой бы сложной ни была эта система построения по шаблону, она представляет собой черновое решение простой задачи подстановки переменной. Эта система допускает повторное использование, предоставляя пользователю возможность передавать объект, содержащий имена и значения переменных шаблона, которые требуется заполнить, а также элементарные средства доступа к переменным. И все это становится возможным благодаря применению оператора `with`, который упрощает обращение к свойствам по ссылкам в самом шаблоне.

Рассматриваемая здесь система построения по шаблону преобразует предоставляемые ей строки шаблона в массив значений, сцепляя их в конечном итоге вместе. Затем отдельные операторы типа <%=name%> преобразуются в более приемлемый вид name, складываясь во встраиваемый код в процессе построения массива. В итоге получается необыкновенно быстродействующая и эффективная система построения по шаблону.

Кроме того, все шаблоны формируются динамически (т.е. по мере необходимости, поскольку допускается выполнение встраиваемого кода). Для того чтобы упростить повторное использование формируемых шаблонов, можно поместить весь построенный

код шаблона в конструкции `new Function(...)`. В итоге получится функция шаблона, в которую можно активно вставлять нужные данные.

Вся система построения по шаблону сводится вместе с помощью встраиваемых шаблонов. У современных браузеров и поисковых механизмов имеется один крупный недостаток: они не в состоянии правильно интерпретировать элементы разметки `<script/>`, внутри которых указывается непонятный им тип, и поэтому такие элементы полностью игнорируются. А это означает, что мы можем указать сценарии, содержащие наши шаблоны, присвоив им тип `"text/tmpl"` наряду с уникальным идентификатором, а затем воспользоваться данной системой для повторного извлечения шаблонов в дальнейшем. В целом о рассматриваемой здесь системе построения по шаблону можно сказать, что она проста в использовании благодаря возможностям оператора `with`.

Резюме

Из этой главы вы узнали следующее.

- Основное назначение оператора `with` состоит в упрощении сложного кода благодаря тому, что ссылки на свойства объекта можно делать, не ссылаясь на сам объект, которому эти свойства принадлежат. Это дает возможность значительно сократить и сделать более понятным код со многими ссылками на свойства объекта.
- Подобное упрощение распространяется на такие области, как организация пространства имен, тестирование и даже построение систем построения по шаблону. В подтверждение этого в главе были представлены некоторые примеры употребления оператора `with` в ряде распространенных библиотек JavaScript.
- Как и всеми остальными эффективными средствами, оператором `with` следует пользоваться осмотрительно. Применяя его, можно очень легко запутать код, вместо того чтобы прояснить его.
- За все время своего существования оператор `with` заслужил славу весьма противоречивого языкового средства JavaScript. К тому же у него нет будущего, и поэтому его употребления в новом коде следует избегать.

Что касается области действия оператора `with`, то в этой главе даже не упоминалось о какой-либо несовместимости браузеров. Но теперь наступает черед глав, где вопросы совместимости браузеров будут рассматриваться вплотную. Так, в следующей главе мы обсудим способы борьбы с кросс-браузерным сумасшествием, не теряя при этом рассудка.

11 Стратегии разработки кросс-браузерного кода

В этой главе...

- Стратегии разработки повторно используемого кросс-браузерного кода JavaScript
- Анализ характера затруднений, которые приходится разрешать
- Разумный подход к разрешению возникающих затруднений

Всякий, кому приходилось разрабатывать код страничных сценариев на JavaScript, уже через пять минут осознавал наличие целого ряда затруднений, связанных с обеспечением надежной работы кода в поддерживаемых браузерах. Эти затруднения приходится принимать во внимание на самых разных стадиях разработки: от элементарного решения текущих задач, планирования последующих выпусков браузеров и вплоть до повторного использования кода на еще не созданных веб-страницах.

Написание кода для многих браузеров — безусловно, нетривиальная задача, которая требует увязки с избранными методологиями разработки веб-приложений, а также с наличными ресурсами для работы над проектом. Как бы нам ни хотелось, чтобы разрабатываемые веб-страницы идеально функционировали в каждом браузере, который существовал, имеется и еще только появится, суровая реальность вынуждает нас признать, что для разработки в нашем распоряжении имеются ограниченные ресурсы. Поэтому нам приходится тщательно планировать такое применение этих ресурсов, чтобы извлечь из них наибольшую пользу. И начинать следует с тщательного выбора поддерживаемых браузеров.

Выбор поддерживаемых браузеров

Решая, куда направить имеющиеся ограниченные ресурсы, необходимо прежде всего выбрать те основные браузеры, которые должны поддерживаться в прикладном коде. Как и во всех остальных случаях разработки веб-приложений, необходимо очень тщательно отобрать те браузеры, где может быть обеспечено оптимальное восприятие веб-содержимого конечными пользователями. При выборе поддерживаемого браузера во внимание обычно принимаются следующие соображения.

1. Активное тестирование кода в выбранном браузере с помощью тестового набора.
2. Исправление программных ошибок и регрессий, связанных с выбранным браузером.
3. Выбранный браузер должен обладать приемлемой производительностью.

Например, большинство библиотек JavaScript рассчитано на поддержку около дюжины браузеров. При этом подразумеваются предыдущие, текущие и последующие бета-версии (если таковые имеются) браузеров из так называемой "Большой пятерки", а именно:

- Internet Explorer;
- Firefox;
- Safari;
- Chrome;
- Opera.

Это довольно обширный ряд поддерживаемых браузеров, особенно если учесть, что тестирование кода в них приходится выполнять на нескольких платформах и что у таких браузеров, как, например, Internet Explorer, в употреблении одновременно находится несколько версий. Если к разработке библиотек JavaScript вроде jQuery можно привлечь многих специалистов, пускай и на добровольных началах, то у обычного автора веб-страниц такой роскоши нет. Поэтому ему приходится реалистично подходить к выбору поддерживаемых браузеров.

Примечание

Конечно, можно было бы положиться на возможности, заложенные в основные библиотеки JavaScript, чтобы автоматически заручиться поддержкой браузеров. Но в этой книге не предполагается, что вы будете пользоваться библиотекой, и поэтому в ней даются рекомендации по оптимальному выбору браузеров, которые должны поддерживаться в разрабатываемом вами прикладном коде.

Принимая решение относительно выбора поддерживаемых браузеров, целесообразно составить так называемую *матрицу поддержки браузеров*, заполнив ее в соответствии с конкретными поставленными целями. Пример такой матрицы приведен в табл. 11.1, где сделанный выбор (отмечено галочкой) не отражает каких-либо оценок или предпочтений отдельных браузеров. В остальной части этой главы будут даны полезные советы, призванные помочь вам правильно заполнить матрицу поддержки браузеров. Но имейте в виду, что вам, возможно, придется дополнительно разграничить браузеры по платформам, если вы не собираетесь обеспечивать их равноценную поддержку на тех платформах, где они существуют.

Таблица 11.1. Пример матрицы поддержки браузеров —заполняется на основании индивидуальных решений

Версия	Chrome	Firefox	Safari	Internet Explorer	Opera
Предыдущая	✓	✓	✓	✓	✓
Текущая	✓	✓	✓	✓	✓
Бета	✓	✓	×	×	×

Любой фрагмент повторно используемого кода JavaScript, будь то из повсеместно принятой библиотеки JavaScript или страничного сценария, должен быть написан таким образом, чтобы надежно работать в как можно большем числе сред и, главным образом, в тех браузерах и на тех платформах, которым отдают предпочтение пользователи. Для повсеместно применяемых библиотек перечень целевых сред довольно обширен, тогда как для специализированных приложений он может быть сужен. Но при этом очень важно не переусердствовать, чтобы не принести качество в жертву количеству. Принимая ответственное решение относительно поддерживаемых сред, не забывайте следующее правило:

Качество не должно быть принесено в жертву охвату поддерживаемых сред.

В этой главе сначала будут рассмотрены различные ситуации, в которых код JavaScript должен быть написан с учетом кросс-браузерной поддержки, а затем наилучшие методики написания кода с целью предотвратить любые осложнения, которые могут возникнуть в подобных ситуациях. Это поможет вам правильно выбрать те методики, принятие которых стоит затраченного времени, а следовательно, заполнить матрицу поддержки браузеров надлежащим образом.

Самые насущные задачи разработки

При написании любого нетривиального кода возникает масса задач, которые приходится решать на стадии разработки веб-приложений. Среди них можно выделить пять самых насущных и трудных задач, встающих перед разработчиком при написании повторно используемого кода JavaScript (рис. 11.1.)

Рис. 11.1. Пять самых насущных задач разработки используемого кода JavaScript

Ниже перечислены эти пять насущных задач разработки используемого кода JavaScript.

- Программные ошибки в браузерах.
- Устранение программных ошибок в браузерах.
- Отсутствующие средства в браузерах.
- Внешний код.
- Регрессии в браузерах.

Решая каждую из этих задач, необходимо найти золотую середину между временем, которое приходится на это тратить, и выгодой, которая будет получена в конечном итоге. Например, стоит ли тратить дополнительные 40 часов рабочего времени на улучшение поддержки браузера Internet Explorer 6? В конечном счете ответ на этот вопрос каждому разработчику приходится искать самостоятельно, исходя из сложившейся ситуации. И этот ответ может оказаться совершенно иным для веб-приложений, предназначенных для общего доступа к Интернету, чем для приложений, предназначенных для внутреннего употребления сотрудниками организации, привязанными, как цепями, к браузеру Internet Explorer 6 луддитами из отдела информационных технологий!

Анализ предполагаемой аудитории, наличных ресурсов для разработки и график выполнения работ – все эти факторы следует непременно учитывать, принимая ответственное решение. Для разработки приложений полезно усвоить следующую аксиому:

Не забывать прошлое, смотреть в будущее и анализировать настоящее.

Пытаясь разработать повторно используемый код JavaScript, следует принимать во внимание все, что имеет отношение к данному вопросу. В первую очередь необходимо обратить особое внимание на самые распространенные в настоящее время браузеры, затем принять в расчет те изменения, которые могут появиться в последующих версиях браузеров, и, наконец, попытаться сохранить на приемлемом уровне совместимость со старыми версиями браузеров, обеспечив поддержку как можно большего арсенала средств, но не принося в жертву качество, или хотя бы тех средств, которые должны поддерживаться во всех браузерах. В последующих разделах перечисленные выше задачи будут рассмотрены по отдельности, чтобы стали понятнее возникающие трудности и пути их преодоления.

Программные ошибки и отличия в браузерах

Главной заботой разработки повторно используемого кода JavaScript должна стать обработка различных программных ошибок в браузерах и преодоление несоответствий в прикладном интерфейсе API, касающихся поддержки избранных браузеров. Это означает, что любые средства, предоставляемые в прикладном коде, должны быть полностью *проверены* на пригодность к применению во всех этих браузерах.

И достичь этой цели совсем не трудно, как пояснялось в главе 2 и демонстрируется на протяжении всей этой книги: достаточно составить исчерпывающий набор тестов, охватывающий как типичные, так и крайние случаи применения кода. Охватив тестами разрабатываемый прикладной код в достаточной степени, мы можем быть уверены в том, что разрабатываемый код будет надежно работать в избранном ряде поддерживаемых браузеров. А если допустить, что последующие изменения в браузерах не нарушат обратную совместимость, то можно даже надеяться на то, что код будет нормально работать и в последующих версиях этих браузеров.

Далее в этой главе будут рассмотрены конкретные стратегии устранения программных ошибок и отличий в браузерах. Самое трудное во всем этом – реализовать устранение программных ошибок в текущих браузерах таким образом, чтобы противостоять любым устранениям этих ошибок в последующих версиях браузеров.

Устранение программных ошибок в браузерах

Было бы неразумно допустить, что конкретная программная ошибка будет вечно присутствовать в браузере. Ведь в большинстве браузеров программные ошибки в конечном итоге устраняются, и поэтому было бы опрометчиво рассчитывать на постоянное присутствие подобных ошибок в браузерах. В этой связи лучше всего воспользоваться методикой, рассматриваемой далее в главе, чтобы как можно более надежно застраховать любые приемы обхода программных ошибок на будущее.

При написании фрагментов повторно используемого кода JavaScript требуется обеспечить продолжительное его существование. Как и при разработке любого другого компонента веб-сайта (CSS, HTML и т.д.), нежелательно возвращаться назад, чтобы внести изменения в веб-сайт, который не работает в новой версии какого-нибудь браузера.

Чаще всего неполадки в работе веб-сайта возникают из-за неверных допущений в отношении программных ошибок в браузерах. Они сводятся к следующему: конкретные усовершенствования вводятся с целью обойти программные ошибки, вносимые браузером, что приводит к нарушению нормальной работы веб-сайта, когда эти ошибки устраняются в последующих версиях браузера. Данное затруднение зачастую устраняется путем построения фрагментов кода, имитирующего конкретный компонент (подробнее об этом – далее), вместо того, чтобы делать какие-то допущения в отношении исправности браузера.

Вопрос обращения с программными ошибками в браузере, по существу, распадается на два других.

1. Работоспособность разрабатываемого кода будет нарушена, если в браузере будет устранена ошибка.
2. Разработчики браузеров могут быть невольно приучены *не* устранять программные ошибки, опасаясь нарушить работу веб-сайтов.

Интересная ситуация в связи с этим возникла в ходе разработки версии браузера Firefox 3. В эту версию браузера было внесено изменение, вынуждавшее DOM-документ к принятию узлов модели DOM, созданных в другом документе, если их предполагалось вставлять в другой документ (в соответствии со спецификацией модели DOM). Так, приведенный ниже код не должен работать.

```
var node = documentA.createElement("div");
documentB.documentElement.appendChild( node );
```

Для того чтобы он нормально работал, его следовало бы написать так, как показано ниже.

```
var node = documentA.createElement("div");
documentB.adoptNode( node );
documentB.documentElement.appendChild( node );
```

Но поскольку в данной версии браузера Firefox присутствовала программная ошибка, из-за которой первая часть приведенного выше фрагмента кода работала, тогда как на самом деле она не должна была работать, пользователи написали свой код таким

образом, что он зависел от работоспособности данного фрагмента кода. Это привело к тому, что разработчики из компании Mozilla произвели откат своих изменений, опасаясь нарушить работу многочисленных веб-сайтов. Они сами признаются в этом, как следует из примечаний компании Mozilla к данной ошибке в документах (`https://developer.mozilla.org/en-US/docs/DOM/WRONG_DOCUMENT_ERR_note`).

В связи с программными ошибками в браузерах напрашивается следующий важный вывод: если часть функциональных возможностей браузера потенциально ошибочна, то следует вернуться к спецификации. В рассмотренной выше ситуации браузер Internet Explorer действовал более убедительно, генерируя исключение, если узел отсутствовал в документе, и это было правильное поведение. Но пользователи предположили, что в данном случае это была просто ошибка в Internet Explorer, и поэтому на всякий случай написали условный код в качестве резервного варианта. В итоге сложилась такая ситуация, когда пользователи придерживались спецификации только для одного подмножества браузеров и вынужденно отклонялись от нее для всех остальных браузеров.

Программную ошибку в браузере необходимо отличать от неуказанного в спецификации прикладного интерфейса API. В связи с этим целесообразно вернуться к спецификациям браузеров, поскольку они содержат точные нормативы для разработки и усовершенствования кода браузеров. В то же время реализация неуказанного в спецификации прикладного интерфейса API может измениться в любой момент, особенно если будет предпринята попытка указать и затем внести по ходу дела изменения в эту реализацию. Что же касается несогласованности прикладных интерфейсов API, не внесенных в спецификацию, то рекомендуется всегда проверять предполагаемый результат, выполняя дополнительные контрольные примеры имитации компонентов, как поясняется далее в этой главе. Следует всегда иметь в виду изменения, которые могут впоследствии произойти в этих прикладных интерфейсах по мере того, как они примут окончательную, устоявшуюся форму.

Существуют также различия между устранением программных ошибок и изменениями в прикладном интерфейсе API. Если устранение программных ошибок нетрудно предвидеть (они в конечном итоге устраняются в реализации браузера, пусть даже и нескоро), то выявить изменения в прикладном интерфейсе API намного труднее. Изменения в стандартные прикладные интерфейсы API вносятся редко, хотя это и нельзя считать совершенно невозможным. Намного более вероятными оказываются изменения в прикладных интерфейсах API, не внесенных в спецификацию.

Правда, такие изменения редко приводят к массовым нарушениям в работе веб-сайтов. Но если подобные нарушения все же происходят, их, по существу, невозможно выявить заранее, если, конечно, не задаться целью проверить все прикладные интерфейсы API, когда-либо применявшиеся в разрабатываемом коде, хотя связанные с этим издержки просто неоправданны. Любые подобного рода изменения в прикладном интерфейсе API следует интерпретировать таким же образом, как и любую другую регрессию.

Что же касается следующей насущной задачи разработки повторно используемого кода JavaScript, то следует иметь в виду, что прикладной код, как и человек, один в поле не воин. Попробуем выяснить, к каким последствиям это может привести.

Сосуществование с внешним кодом и разметкой

Любой повторно используемый код должен сосуществовать с окружающим его кодом. Предполагается ли выполнение кода на веб-страницах, созданных собственными силами, или на веб-сайтах, разработанных другими, необходимо обеспечить нормальное его сосуществование на странице с любым другим кодом. Но эта задача сродни палке о двух концах: код должен не только сосуществовать с неудачно написанным внешним кодом, но и не оказывать неблагоприятного воздействия на соседний код.

Решение этой насущной задачи во многом зависит от той среды, в которой предполагается использовать разрабатываемый код. Так, если повторно используемый код применяется на ограниченном количестве веб-сайтов, но рассчитан на большое число браузеров, внешний код вряд ли вызовет какие-то особые осложнения, поскольку в этом случае у разработчика имеются все необходимые полномочия и возможности самостоятельно устранять возникающие неполадки.

Совет

Данная насущная задача настолько важна, что ее рассмотрению следует посвятить отдельную книгу. Поэтому, если вас заинтересует этот предмет, рекомендуем прочитать упоминавшуюся ранее книгу *Third-Party JavaScript* Бена Винегара и Антона Ковалева (издательство Manning Publications; `http://manning.com/vinegar/`).

Но если попытаться разработать код для самого широкого применения, то придется каким-то образом обеспечить его надежность и эффективность. Рассмотрим некоторые стратегии для достижения этой цели.

Инкапсуляция кода

Для того чтобы разрабатываемый код не оказывал влияние на другие фрагменты кода на тех веб-страницах, где он загружается, его лучше всего *инкапсулировать*. В словаре термину *инкапсуляция* дается следующее определение: "размещение внутри, как в капсуле", а в программировании: "языковый механизм для ограничения доступа к некоторым компонентам объекта". Но, попросту говоря, инкапсуляция действует по следующему принципу: не суй нос не в свое дело!

Когда код вводится на веб-странице, он должен оставлять за собой минимальный след. На самом деле сохранить такой след совсем не трудно в нескольких глобальных переменных, а еще лучше — в одной. Характерным тому примером может служить библиотека jQuery. В ней вводится одна глобальная переменная (функция) под названием `jQuery` и один псевдоним этой глобальной переменной — `$`. В этой библиотеке поддерживаются даже средства для возврата псевдонима `$` любому другому страничному коду или библиотеке, где он может быть использован.

Практически все операции в jQuery выполняются через функцию `jQuery()`. А любые другие, так называемые *служебные функции*, которые она предоставляет, определяются как ее свойства. (Как пояснялось в главе 3, одни функции совсем не трудно определить как свойства других функций.) Таким образом, имя `jQuery` используется в качестве пространства имен для всех определений данной функции.

Подобную стратегию можно взять на вооружение. Допустим, определяется ряд функций для собственного употребления или для применения другими и все они должны быть сгруппированы в избранном пространстве имен `ninja`. Как и в библиотеке jQuery, для этой цели можно определить глобальную функцию `ninja()`, выполняющую различные операции в зависимости от того, что именно ей передается:

```
var ninja = function(){ /* здесь следует код реализации */ }
```

Определить далее собственные служебные функции, используя эту глобальную функцию в качестве их пространства имен, не представляет большого труда, как показано ниже.

```
ninja.hitsuke = function(){
  /* здесь следует код для отвлечения огнем внимания охраны */ }
```

Если бы потребовалось, чтобы имя `ninja` служило не в качестве функции, а лишь пространства имен, его можно было бы определить следующим образом:

```
var ninja = {};
```

В этом случае создается пустой объект, в котором можно определить свойства и функции, чтобы не вводить их имена в глобальном пространстве имен.

К числу других приемов, которых следует избегать, чтобы сохранить код инкапсулированным, относится видоизменение любых существующих переменных, прототипов функций и даже элементов модели DOM. С одной стороны, любой компонент веб-страницы, который является внешним по отношению к прикладному коду и видоизменяется им, служит потенциальным местом для конфликтов и недоразумений. А с другой стороны, даже если следовать самым лучшим методикам и тщательно инкапсулировать прикладной код, то и тогда нельзя гарантировать, что чужой код будет вести себя также безупречно.

Обращение с не менее типичным кодом

С тех пор как Грейс Хоппер удалила моль с реле на первой в мире ЭВМ Mark I, существует старая поговорка: "Единственный код, который не засасывает, — это код, написанный самостоятельно". Возможно, это и слишком циничная точка зрения, но когда прикладной код сосуществует с другим кодом, не поддающимся контролю, приходится ради перестраховки предполагать самое худшее. Другой код, пусть даже и грамотно написанный, а не ошибочный, может *неумышленно* выполнять такие действия, как, например, видоизменение прототипов функций, свойств объектов и методов обращения к элементам модели DOM. И какими бы благими ни были намерения авторов чужого кода, он может стать западней, в которую легко попасться.

И если в подобных случаях в нашем прикладном коде будет сделано что-нибудь совершенно безобидное, например, использованы массивы JavaScript, то никто не сможет нам поставить в вину следующее простое допущение: массивы JavaScript должны действовать именно так, как им и положено. Но если код на какой-нибудь другой странице изменит порядок действия этих массивов, то наш прикладной код может перестать нормально работать, хотя мы и не совершили никакой оплошности.

К сожалению, существует немало устоявшихся правил поведения в подобных затруднительных ситуациях, но эти правила не мешают предпринять ряд предупредительных мер для выхода из них. И такие меры будут представлены в последующих подразделах.

Исключение внедряемых свойств

В качестве первой предупредительной меры можно научиться избегать свойств, которые могут быть подспудно внедрены в объекты без нашего ведома. Для того чтобы выявить подобные действия, мы можем воспользоваться функцией `hasOwnProperty()`. Эта функция наследуется из класса `Object` всеми классами JavaScript и проверяет, обладает ли объект указанным свойством. Она действует подобно оператору `in` в JavaScript, но с тем важным отличием, что не проверяет цепочку прототипов.

Следовательно, с помощью данной функции можно выявить отличия между свойствами, внедренными путем расширения `Object.prototype`, и теми свойствами, которые установлены непосредственно в объекте. Для того чтобы понаблюдать за поведением данной функции, рассмотрим тесты, приведенные в листинге 11.1.

Листинг 11.1. Применение функции `OwnProperty()` для проверки наследуемых свойств

```
<script type="text/javascript">

  Object.prototype.ronin = "ronin";

  var object = { ninja: 'value' };
  object.samurai = 'samurai';

assert(object.hasOwnProperty('ronin'),"ronin is a property");
assert(object.hasOwnProperty('ninja'),"ninja is a property");
assert(object.hasOwnProperty('samurai'),"samurai is a property");

</script>
```

Установить наследуемое свойство

Установить ненаследуемое свойство

Результаты выполнения приведенных выше тестов представлены на рис. 11.2. Эти результаты явно показывают, что свойство `ronin`, добавленное в прототип класса `Object`, не считается "собственным" свойством созданных объектов. Правда, количество сценариев, где применяется рассматриваемый здесь прием, очень мало, но вред, который они наносят, может оказаться довольно большим, если свойства, вводимые в прототип, нарушают нормальное выполнение кода.

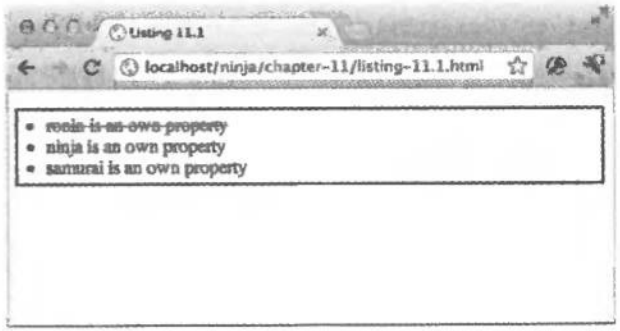

Рис. 11.2. Как показывают результаты тестов, с помощью функции `hasOwnProperty()` можно выявить наследуемые свойства

Особые трудности могут возникнуть при циклическом обращении к свойствам объекта с помощью оператора `for-in`. А учесть подобные осложнения можно, используя функцию `hasOwnProperty()`, чтобы выяснить, следует ли проигнорировать свойство или нет, как показано ниже.

```
for (var p in someObject) {
  if (someObject.hasOwnProperty(p)) {
    // сделать что-нибудь полезное
  }
}
```

В данном фрагменте кода демонстрируется применение функции `hasOwnProperty()` для игнорирования свойств, введенных в прототип объекта.

Совет

В версии JavaScript 1.8.5 внедрен метод `Object.getOwnPropertyNames()`. Подробные сведения о нем можно найти по адресу `https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Object/getOwnPropertyNames`.

Борьба с поглощающими идентификаторами

Большинство браузеров обладают *антисвойством*, которое нельзя назвать программной ошибкой, поскольку его поведение носит совершенно намеренный характер. Оно способно нарушить нормальное выполнение прикладного кода и привести к неожиданному сбою в нем. Это антисвойство обусловливает добавление ссылок на другие элементы разметки с использованием атрибута `id` исходного элемента. А когда атрибут `id` вступает в конфликт со свойствами, которые уже являются частью элемента разметки, может произойти все, что угодно. В качестве примера рассмотрим приведенный ниже фрагмент HTML-разметки простейшей формы, чтобы выявить те неприятности, к которым могут привести так называемые *поглощающие идентификаторы*.

```
<form id="form" action="/conceal">
<input type="text" id="action"/>
<input type="submit" id="submit"/>
</form>
```

В браузерах эта форма вызывается следующим образом:

```
var what = document.getElementById('form').action;
```

Справедливо предположить, что переменной `what` будет присвоено значение атрибута `action` элемента разметки `form`. И в большинстве случаев так и произойдет. Но если проверить содержимое этой переменной, то обнаружится, что она, как ни странно, содержит ссылку на элемент разметки `input#action`! Проведем еще один эксперимент, как показано ниже.

```
document.getElementById('form').submit();
```

Этот оператор должен привести к предъявлению формы, но вместо этого получается следующее сообщение об ошибке в сценарии, где сообщается, что `submit()` не является функцией объекта формы:

```
Uncaught TypeError: Property 'submit' of object #<HTMLFormElement> is not a
function
```

Что же произошло? А все дело в том, что браузеры ввели в элемент разметки `<form>` для каждого из элементов разметки `<input>` ввода данных в форму свойства, ссылающиеся на этот элемент. На первый взгляд это может показаться очень удобным, но по зрелом размышлении оказывается, что имя введенного свойства берется из значения атрибута `id` элементов разметки `<input>`. А если это значение окажется уже использовавшимся свойством элемента разметки формы, например `action` или `submit`, то исходные свойства будут заменены новым свойством.

Таким образом, перед созданием элемента разметки `input#submit` ссылка `form.action` указывает на значение атрибута `action` элемента разметки `<form>`. А впоследствии она указывает на элемент разметки `input#submit`. То же самое происходит и со ссылкой `form.submit`. Возникает запутанная ситуация. Ведь для такого поведения элементы разметки `<input>` должны содержать атрибут `id`. А если у них имеется атрибут `id`, то к ним можно без труда обращаться и не вводя свойства в форму.

Но в любом случае рассматриваемое здесь антисвойство браузеров может вызвать немало озадачивающих осложнений в коде, поэтому его следует непременно иметь в виду, отлаживая прикладной код в браузерах. Когда встречаются свойства, которые были необъяснимым образом преобразованы в нечто совершенно другое, чем предполагалось, причиной тому, скорее всего, являются поглощающие идентификаторы, и поэтому их следует непременно проверить.

Правда, это затруднение можно преодолеть в собственной разметке, исключив из нее простые значения атрибутов `id`, способные вступить в конфликт со стандартными именами свойств. Именно такой прием и рекомендуется применять в дальнейшем. А значения `submit` следует особенно избегать для атрибутов `id` и `name`, поскольку оно служит основным источником неверного и запутанного поведения кода.

Порядок следования таблиц стилей

Зачастую CSS-правила уже имеются к тому времени, когда прикладной код начинает выполняться. Самый лучший способ гарантировать доступность CSS-правил, предоставляемых в таблицах стилей, при выполнении кода JavaScript на странице – указать внешние таблицы стилей перед тем, как включать файлы внешних сценариев. В противном случае возможны неожиданные результаты, поскольку в сценарии будет предпринята попытка получить доступ к неверной информации о стилях. К сожалению, это затруднение не так-то просто устранить только средствами JavaScript, и поэтому его разрешение следует отнести скорее к несовершенству пользовательской документации.

Выше были рассмотрены лишь некоторые и самые простые примеры влияния, и зачастую совершенно неумышленного, внешних факторов на работоспособность разрабатываемого кода. Чаще всего подобные затруднения возникают при попытке пользователей внедрить разрабатываемый код на своих сайтах. В таком случае можно организовать раннюю диагностику подобных затруднений, составив соответствующие тесты для их выявления и устранения. В других случаях подобные затруднения могут возникнуть в процессе интеграции чужого кода на собственных веб-страницах. И можно надеяться, то полезные советы, приведенные в предыдущих подразделах, помогут вам выявить причины возникающих затруднений.

К сожалению, других способов устранения осложнений в связи с подобным внедрением кода, кроме ранней диагностики и принятия предохранительных мер при написании кода, не существует. А теперь перейдем к рассмотрению следующей насущной задачи.

Отсутствующие средства в браузерах

В тех браузерах, которые исключены из матрицы поддержки браузеров, а следовательно, из процесса тестирования кода, зачастую отсутствуют некоторые основные средства, требующиеся для нормальной работы прикладного кода. Могут быть и такие браузеры, у которых нет нужных основных средств, но которые все же требуется поддерживать по тем или иным причинам.

Постепенное снижение функциональных возможностей

Даже если полная поддержка всех браузеров и не предполагается, особенно не прошедших отбор, код все-таки лучше писать осмотрительно, чтобы его функциональные возможности снижались постепенно, или предусмотреть какой-нибудь другой резервный вариант для конечных пользователей, остановивших по тем или иным причинам свой выбор на других браузерах, а не на тех, для которых имеют ресурсы, необходимые для тестирования кода. Следовательно, основная стратегия в подобной ситуации — предоставить пользователям максимум функциональных возможностей и постепенно снижать их в том случае, если нельзя предоставить их полностью. Такая стратегия называется *постепенным снижением функциональных возможностей*.

К реализации постепенного снижения функциональных возможностей следует подходить весьма осмотрительно, тщательно все взвесив. В качестве примера рассмотрим случай, когда браузер способен инициализировать и скрывать целый ряд элементов для перемещения по странице, возможно, надеясь на создание раскрывающихся меню, но код обработки связанных с этим событий не работает. В итоге получается функционирующая наполовину страница, от которой мало проку. В подобных случаях следует непременно предусмотреть в коде резервный вариант для альтернативного режима работы страницы — даже с сокращенными функциональными возможностями.

Обратная совместимость

Более совершенная стратегия состоит в разработке прикладного кода, допускающего обратную совместимость в как можно большей степени, направляя те браузеры, о которых заранее известно, что они не в состоянии отобразить веб-страницу, к альтернативному ее варианту или же на веб-сайт, специально настроенный на браузеры с более скромными функциональными возможностями. Подобная стратегия принята на большинстве веб-сайтов компании Yahoo!, где браузеры разделяют на разные категории по уровням поддержки. Некоторое время спустя составляется "черный список" браузеров, особенно если они занимают очень малую долю рынка (около 0,05%), а пользователи таких браузеров направляются (в зависимости от пользовательского посредника) к версии приложения только в коде HTML, т.е. без стилевого оформления с помощью таблиц CSS и сценариев JavaScript.

Это означает, что разработчики подобных приложений имеют возможность формировать и намечать оптимальное восприятие веб-содержимого обширным кругом (около 99%) пользователей, направляя устаревшие браузеры ко вполне функциональному, хотя и более скромному по восприятию эквиваленту такого содержимого. Ниже перечислены основные положения данной стратегии.

- В отношении восприятия веб-содержимого пользователями старых браузеров не делается никаких предположений. Если браузер больше непригоден для тестирования кода (и занимает пренебрежимо малую долю рынка), он просто обслуживается усеченным и упрощенным вариантом страницы или вообще не поддерживается и оставляется без внимания.
- Всем пользователям современных и устаревших браузеров гарантируется, что страница будет представлена в целостном, ненарушенном виде.
- Допускается, что новые и неизвестные браузеры также должны работать.

Главный недостаток данной стратегии заключается в том, что для поддержки уже устаревших и еще не появившихся браузеров, помимо текущих целевых браузеров и

платформ, требуются дополнительные затраты на разработку. Но, несмотря на затраты, это вполне разумная стратегия, поскольку она допускает относительно долгий срок эксплуатации веб-приложений и лишь самые незначительные их изменения.

Регрессии

Регрессии представляют собой самую трудную задачу, с которой приходится иметь дело при разработке повторно используемого, устойчивого кода JavaScript. Они представляют собой программные ошибки или обратно несовместимые изменения в прикладных интерфейсах API (главным образом, неуказанных в спецификации), вносимые браузерами и поэтому приводящие к нарушению нормальной работы кода самым непредсказуемым образом.

Примечание

Термин *регрессия* употребляется здесь в его классическом определении: работавшее раньше средство больше не функционирует должным образом. И происходит это, как правило, ненамеренно, хотя иногда обусловлено преднамеренными изменениями, нарушающими нормальную работу существующего кода.

Предвосхищение изменений

Некоторые изменения в прикладном интерфейсе API можно предвосхитить, с упреждением обнаружить и соответственно учесть, как показано в примере кода из листинга 11.2. Так, в версии браузера Internet Explorer 9 была внедрена поддержка обработчиков событий второго уровня в модели DOM, привязываемых с помощью метода `addEventListener()`. Поэтому в коде, написанном до появления версии Internet Explorer 9, данное изменение можно было учесть, просто обнаружив объект.

Листинг 11.2. Предвосхищение изменений в прикладном интерфейсе API

```
function bindEvent(element, type, handle) {
  if (element.addEventListener) {
    element.addEventListener(type, handle, false);
  }
  else if (element.attachEvent) {
    element.attachEvent("on" + type, handle);
  }
}
```

❶ Привязать, используя стандартный интерфейс API

❷ Привязать, используя оригинальный интерфейс API

В данном примере код был написан с заделом на будущее, предвидя или хотя бы надеясь на то, что когда-нибудь браузер Internet Explorer будет приведен в соответствие со стандартами модели DOM. Если в браузере поддерживается прикладной интерфейс API, совместимый с этими стандартами, это можно выявить, обнаружив объект, чтобы затем воспользоваться стандартным интерфейсом API ❶. Если же такая поддержка отсутствует, то проверяется наличие метода, оригинального для браузера Internet Explorer, чтобы воспользоваться именно им ❷. В противном случае ничего не делается вообще.

К сожалению, большинство предстоящих изменений в прикладном интерфейсе API или программные ошибки не так-то просто предвидеть. И это еще одно веское основание для тестирования кода, на котором постоянно делается акцент в данной книге. Перед лицом непредсказуемых изменений, способных оказать влияние на код, вся на-

дежда остается только на прилежное тестирование кода в каждом выпуске браузера, чтобы как можно быстрее устранить затруднения, которые могут внести регрессии.

Непредсказуемые программные ошибки

В качестве примера рассмотрим следующую непредсказуемую программную ошибку: в версии браузера Internet Explorer 7 была внедрена основная оболочка в виде объекта типа `XMLHttpRequest`, в которую заключается собственный объект запроса ActiveX. И поэтому буквально во всех библиотеках JavaScript объект типа `XMLHttpRequest` стал по умолчанию применяться для выполнения Ajax-запросов, как это и должно быть, поскольку выбор стандартных прикладных интерфейсов API практически всегда наиболее предпочтителен.

Но при реализации в браузере Internet Explorer была нарушена обработка запросов локальных файлов (веб-сайт, загруженный с рабочего стола, не мог больше запрашивать файлы, используя объект типа `XMLHttpRequest`). ни кому не удавалось выявить эту программную ошибку (и тем более предупредить ее) до тех пор, пока она не вырвалась на свободу и не нарушила нормальную работу многих веб-сайтов. Для устранения этой ошибки было найдено решение, которое состояло в использовании реализации ActiveX, главным образом, для запросов локальных файлов.

Таким образом, наличие хорошего набор тестов и постоянное отслеживание появляющихся выпусков браузеров представляет собой самый лучший способ избежать подобного рода регрессий в будущем. И совсем не обязательно, чтобы это как-то сказалось на привычном цикле разработки приложений, в который уже должна входить стадия тестирования кода. Поэтому выполнение тестов в новых выпусках браузеров следует непременно учитывать, планируя любой цикл разработки.

Сведения о появляющихся выпусках браузеров можно получить из следующих источников:

- Internet Explorer: `http://blogs.msdn.com/ie/`
- Firefox: `http://ftp.mozilla.org/pub/mozilla.org/firefox/nightly/latest-trunk/`
- WebKit (Safari): `http://nightly.webkit.org/`
- Opera: `http://snapshot.opera.com/`
- Chrome: `http://chrome.blogspot.com/`

Такой подход требует прилежания. Ведь заранее неизвестно, когда программные ошибки будут внесены браузером, и поэтому самое лучше — постоянно контролировать работоспособность кода, чтобы предотвратить любые кризисные ситуации, которые могут наступить. Правда, разработчики браузеров делают немало для того, чтобы предотвратить подобного рода регрессии. Так, тестовые наборы из различных библиотек JavaScript интегрированы в основной тестовый набор браузеров Firefox и Opera. Благодаря этому удастся выявить непосредственное влияние появляющихся впоследствии регрессий на эти библиотеки. И хотя это и не позволяет выявить все регрессии во всех браузерах, подобное начинание служит положительным признаком прогресса, наметившегося в отношении разработчиков браузеров к вопросам предотвращения многих осложнений в работе с их программными продуктами.

Итак, мы обсудили особенности тех трудных задач, с которыми приходится имело дело при разработке веб-приложений, а также возможные пути их решения. А теперь перейдем к рассмотрению некоторых стратегий, помогающих решать многие насущные вопросы разработки веб-приложений.

Стратегии реализации кросс-браузерного кода

Знать и понимать те задачи, которые придется решать при разработке кросс-браузерного кода, — это лишь полдела. Но совсем другое дело — выбрать эффективную стратегию для реализации кросс-браузерного кода. Вряд ли найдется такая стратегия, которая оказалась бы пригодной в каждой ситуации, и поэтому большинство вопросов, связанных с кодовой базой, следует решать, сочетая разные стратегии. Ниже будет рассмотрен ряд таких стратегий. И начнем мы с самой простой и не требующей особых хлопот стратегии.

Надежное устранение ошибок в кросс-браузерном коде

К самым простым и надежным способам устранения ошибок в кросс-браузерном коде относятся те, которые обладают следующими важными особенностями.

- Не оказывают никакого отрицательного влияния на работу других браузеров и не проявляют никаких побочных эффектов.
- Не прибегают ни к каким формам выявления браузера или его компонента.

Примеры подобного способа устранения ошибок обычно встречаются редко, но именно к такой тактике следует всегда прибегать при разработке веб-приложений. Обратимся к конкретному примеру. Ниже приведен фрагмент кода, представляющий изменение, внесенное в библиотеку jQuery для нормальной работы с браузером Internet Explorer.

```
// игнорировать отрицательные значения ширины и высоты
if ( (key == 'width' || key == 'height') && parseFloat(value) < 0 )
  value = undefined;
```

В некоторых версиях браузера Internet Explorer при установке отрицательного значения свойств высоты (`height`) или ширины (`width`) в стилевом оформлении веб-страницы генерируется исключение, тогда как все остальные браузеры просто игнорируют вводимые отрицательные значения этих свойств. Для того чтобы обойти данное препятствие, было выбрано решение просто игнорировать все отрицательные значения во *всех* браузерах. Благодаря подобному изменению в коде удалось предотвратить генерирование исключения в браузере Internet Explorer и в то же время избежать отрицательного влияния на остальные браузеры. Такое дополнение оказалось безболезненным, а пользователь получил в свое распоряжение единообразный прикладной интерфейс API. Ведь генерирование неожиданных исключений крайне нежелательно.

Другим примером подобного упрощения в библиотеке jQuery может служить приведенный ниже код манипулирования атрибутами.

```
if ( name == "type" && elem.nodeName.toLowerCase() == "input" &&
  elem.parentNode )
throw "type attribute can't be changed";
```

Браузер Internet Explorer не позволяет манипулировать атрибутом `type` в тех элементах разметки ввода данных, которые уже являются частью модели DOM. Всякая попытка сделать это приведет к генерированию специального исключения. Поэтому в библиотеке jQuery было принято компромиссное решение: просто предотвратить всякие попытки манипулирования атрибутом `type` во вставляемых элементах разметки ввода данных, причем во всех браузерах сразу. В этом случае может быть сгенерировано исключение, но только информационное.

Благодаря такому изменению в коде библиотеки jQuery отпала потребность выявлять конкретный браузер или его компонент, поскольку оно было просто внедрено как средство унификации прикладного интерфейса API для всех браузеров. И хотя упомянутое выше действие по-прежнему вызывает исключение, тем не менее оно генерируется в единообразной для всех браузеров форме. Данное конкретное дополнение носит довольно противоречивый характер. Ведь оно явно ограничивает применение в браузерах тех компонентов библиотеки, на которые подобная программная ошибка не оказывает влияния. Разработчики библиотеки jQuery тщательно взвесили свое решение и посчитали за лучшее сделать так, чтобы прикладной интерфейс API стал унифицированным и работал согласованно, чем давал неожиданные сбои при применении в прикладном кросс-браузерном коде. Не исключено, что с подобными ситуациями придется сталкиваться всем, кто разрабатывает собственные повторно используемые кодовые базы. В этом случае необходимо тщательно проанализировать, насколько подобная ограничительная мера подходит для потенциальной аудитории пользователей.

В отношении подобных изменений в коде не следует забывать о том, что они дают решение, вполне пригодное для всех браузеров и не требующее выявления отдельного браузера или его компонента, а следовательно, на них не будут оказывать никакого влияния какие-либо последующие изменения. Таким образом, следует всегда стремиться к таким оптимальным решениям, даже если они находят редкое и нечастое применение.

Обнаружение объектов

Как обсуждалось ранее, *обнаружение объектов* относится к одному из тех приемов, к которым наиболее часто прибегают при написании кросс-браузерного кода, поскольку этот прием прост и, в общем, довольно эффективен. Он состоит в том, чтобы выявить наличие определенного объекта или его компонента, и если то или другое присутствует, то сделать допущение о наличии у него подразумеваемых функциональных возможностей. (В следующем разделе мы обсудим, как быть в тех случаях, когда подобное допущение не подтверждается.)

Чаще всего обнаружение объектов применяется для выбора среди нескольких прикладных интерфейсов API, представляющих одинаковые функциональные возможности. Например, в коде, приведенном в листинге 11.2 и повторяющемся ниже, обнаружение объектов применяется для выбора подходящих прикладных интерфейсов API, предоставляемых браузером для привязки к событиям.

```
function bindEvent(element, type, handle) {
  if (element.addEventListener) {
    element.addEventListener(type, handle, false);
  }
  else if (element.attachEvent) {
    element.attachEvent("on" + type, handle);
  }
}
```

В данном случае проверяется, существует ли свойство `addEventListener`, и если оно существует, то допускается, что оно является выполняемой функцией, с помощью которой можно привязать к данному элементу приемник событий. А далее проверяются другие прикладные интерфейсы API, например, на наличие свойства `attachEvent`. Обратите внимание на то, что проверка начинается со *стандартного* метода `addEventListener()`, предоставляемого в соответствии со спецификацией модуля W3C DOM Events. И сделано это намеренно.

При всякой возможности следует стремиться выбирать сначала указанный в спецификации способ выполнения определенного действия. Как упоминалось ранее, такой подход помогает создавать прикладной код с перспективой на будущее. Более того, влияние, оказываемое со стороны повсеместно принимаемых библиотек, а также мнение, высказываемое авторитетными специалистами в данной области в социальных сетях и прочих средствах массовой информации, стимулирует разработчиков браузеров работать над внедрением стандартных средств для выполнения действий.

К числу тех важных областей, в которых применяется обнаружение кода, относится выявление средств, предоставляемых той средой браузера, в которой выполняется код. Это дает возможность воспользоваться подобными средствами в коде, а если этого сделать нельзя, то предоставить запасной вариант.

В приведенном ниже фрагменте кода демонстрируется простой пример, в котором обнаружение объекта используется с целью выяснить, следует ли предоставлять полноценный или же сокращенный запасной вариант функциональных возможностей приложения для взаимодействия с пользователем.

```
if ( typeof document !== "undefined" &&
    (document.addEventListener || document.attachEvent) &&
    document.getElementsByTagName && document.getElementById ) {
  // Возможностей прикладного интерфейса API достаточно для построения
  // полноценного варианта приложения
} else {
  // предоставить запасной вариант
}
```

В данном примере кода проверяется следующее:

- Загружен ли документ в браузер.
- Предоставляются ли в браузере средства для привязки обработчиков событий.
- Способен ли браузер обнаруживать элементы разметки по имени дескриптора.
- Способен ли браузер обнаруживать элементы разметки по идентификатору.

Если любая из этих проверок не даст положительного результата, то придется прибегнуть к запасному варианту. И все, что делается в запасном варианте, должно отвечать ожиданиям потребителей прикладного кода, а также предъявляемым к нему требованиям. Для реализации запасного варианта имеются следующие возможности.

- Произвести дополнительное обнаружение объектов и выяснить, как предоставить сокращенный вариант взаимодействия приложения с пользователем, в котором по-прежнему используется некоторый сценарий JavaScript.
- Не выполнять сценарий JavaScript, довольствовавшись одной только HTML-разметкой веб-страницы.
- Направить пользователя к упрощенному варианту веб-сайта (нечто подобное делается, например, с почтой GMail в Google).

Обнаружение объектов требует весьма незначительных издержек на простое выявление объектов и их свойств и относительно просто реализуется. Поэтому данный способ вполне пригоден для предоставления самых элементарных функциональных возможностей в виде запасного варианта как на уровне прикладного интерфейса API, так и на уровне самого приложения. Он может служить в качестве надежной первой линии обороны при авторской разработке повторно используемого кода.

Но что, если предположение о правильности работы прикладного интерфейса API основывается лишь на том, что он *существует*, но в итоге оказывается слишком оптимистичным? Попробуем далее выяснить, как поступить в таком случае.

Имитация компонентов

Еще одним средством, препятствующим регрессиям и в то же время наиболее эффективным для выявления фактов устранения программных ошибок в браузерах, является *имитация компонентов*. В отличие от обнаружения объектов, которое просто выявляет объекты и их свойства, имитация компонентов предполагает полный прогон компонента с целью проверить, что он действует должным образом.

Обнаружение объектов вполне подходит для проверки *существования* компонента, но оно совсем не гарантирует, что *поведение* компонента будет именно таким, как и предполагалось. Но если заранее известны конкретные программные ошибки, то можно очень быстро составить тесты, чтобы проверить, устранена ли программная ошибка в компоненте, а затем написать код для обхода этой ошибки до момента ее устранения в браузере.

Например, при вызове метода `getElementsByTagName("*")` в версии браузера Internet Explorer 8 и более ранних его версиях будут возвращаться как элементы разметки, так и комментарии к ним. Но чтобы определить, произойдет ли это вообще, простого обнаружения объектов окажется явно недостаточно. Как и следовало ожидать, эта программная ошибка была исправлена разработчиками в версии Internet Explorer 9 данного браузера.

Ниже приведен пример кода, в котором имитация компонентов применяется с целью выяснить, будет ли метод `getElementsByTagName()` действовать должны образом.

```
window.findByTagWorksAsExpected = (function(){
  var div = document.createElement("div");
  div.appendChild(document.createComment("test"));
  return div.getElementsByTagName("*").length === 0;
})();
```

В данном примере применяется немедленно вызываемая функция, возвращающая логическое значение `true`, если вызов метода `getElementsByTagName("*")` действует, как и предполагалось, а иначе – логическое значение `false`. В этой тестовой функции выполняются следующие простые действия.

- Создание обособленного элемента разметки `<div>`.
- Ввод узла комментариев в элемент разметки `<div>`.
- Вызов функции, определение количества возвращаемых значений и возврат логического значения `true` или `false` в зависимости от результата.

Но выяснить наличие ошибки – это лишь полдела. Как воспользоваться выясненной ситуацией, чтобы улучшить ее для нормального выполнения прикладного кода? В листинге 11.3 представлен пример, в котором приведенный выше фрагмент кода используется в полезном контексте для обхода программной ошибки.

Листинг 11.3. Применение имитации компонентов на практике для обхода программной ошибки в браузере

```
<!DOCTYPE html>
<html>
  <head>
    <title>Listing 11.3</title>
    <script type="text/javascript" src="../scripts/assert.js"></script>
    <link href="../styles/assert.css" rel="stylesheet" type="text/css">
  </head>
  <body>

    <div><!-- комментарий #1--></div>
    <div><!-- комментарий #2--></div>

    <script type="text/javascript">

      function getAllElements(name) {
        if (!window.findByTagWorksAsExpected) {

          window.findByTagWorksAsExpected = (function(){
            var div = document.createElement("div");
            div.appendChild(document.createComment("test"));
            return div.getElementsByTagName("*").length === 0;
          })();
        }

        var allElements = document.getElementsByTagName('*');

        if (!window.findByTagWorksAsExpected) {
          for (var n = allElements.length - 1; n >= 0; n--) {
            if (allElements[n].nodeType === 1)
              allElements.splice(n,1);
          }
        }

        return allElements;

      }

      var elements = getAllElements();
      var elementCount = elements.length;

      for (var n = 0; n < elementCount; n++) {
        assert(elements[n].nodeType === 1,
               "Node is an element node");
      }
    </script>
  </body>
</html>
```

❶ Проверить, известно ли уже, что браузер работает должным образом

❷ Иначе выяснить, работает ли компонент в браузере как следует, или же он испорчен

❸ Вызвать подозреваемый компонент и сохранить результат

❹ Устранить ошибку в браузере, если она известна

❺ Подготовить код к тестированию

❻ Проверить компонент с обходом ошибки

В приведенном выше примере кода устанавливаются элементы разметки `<div>`, содержащие узлы комментариев, которые в дальнейшем будут использоваться для тестирования. Затем дело доходит до сценария. Непосредственное применение метода `document.getElementsByTagName('*')` вызывает подозрение, и поэтому вместо него определяется альтернативный метод `getAllElements()`. Действие этого метода сводится к вызову метода `document.getElementsByTagName('*')` в тех браузерах,

где он реализован правильно. А в тех браузерах, где он не реализова, используется резервный вариант, дающий правильные результаты.

В методе `getAllElements()` прежде всего используется немедленно вызываемая функция, разработанная ранее с целью выяснить, работает ли проверяемый компонент должным образом ❷. Результат этой проверки сохраняется в переменной, находящейся в области действия объекта `window`, что дает возможность обратиться к ней в дальнейшем и проверить, была ли она уже установлена. Благодаря этому проверка работоспособности компонента путем его имитации производится лишь один раз, хотя она и относительно затратная ❶. После этой проверки осуществляется вызов метода `document.getElementsByTagName('*')`, а результат сохраняется в переменной ❸.

На данном этапе уже имеется узловой список всех элементов, а также известно, имеется или отсутствует ошибка в узле комментариев используемого браузера. Если такая ошибка имеется, то производится обход всех узлов, из числа которых исключаются любые узлы элементов ❹. Этот процесс пропускается для тех браузеров, где подобная ошибка отсутствует. И наконец, новый метод проверяется в действии ❺. При этом утверждается, что возвращаемый узловой список содержит только узлы элементов ❻.

Примечание

Для узлов элементов значение свойства `nodeType` равно `1`, тогда как для узлов комментариев оно равно `8`. В современных браузерах, включая версии Internet Explorer 8 и 9, определяется ряд констант в объекте `Node`, в том числе `Node.ELEMENT_NODE` и `Node.COMMENT_NODE`. Но искомые ошибки возникают в более ранних версиях этого браузера, и поэтому нет никаких оснований полагать, что эти константы существуют, а следовательно, в рассматриваемом здесь коде используются жестко закодированные значения типов узлов. С полным перечнем типов узлов и присвоенных им значений можно ознакомиться по адресу `https://developer.mozilla.org/en/nodeType`.

Данный пример наглядно показывает, что имитация компонента осуществляется в два этапа. Сначала выполняется простой тест, чтобы выяснить, действует ли проверяемый компонент должным образом. При этом очень важно проверить целостность компонента, убедившись в том, что он работает правильно, вместо того, чтобы проверять явным образом наличие программной ошибки. Это обстоятельство очень важно иметь в виду, несмотря на возможные семантические отличия.

Затем результаты тестирования используются для ускорения циклического обращения к элементам массива. Для правильно работающего браузера, возвращающего только элементы, производить проверку элементов на каждом шаге цикла не нужно, и поэтому ее можно пропустить полностью и без потери производительности в тех браузерах, которые работают правильно. Таким образом, имитация компонентов осуществляется по следующему общему принципу: сначала убедиться в том, что проверяемый компонент работает должным образом, а затем предоставить резервный код для неправильно работающих браузеров.

В отношении имитации компонентов очень важно иметь в виду, что это своего рода компромиссное решение. За возможность точно выяснить, как работает отдельный компонент в текущем браузере, и застраховаться на будущее от осложнений в связи с устранением программных ошибок в последующих выпусках браузера приходится расплачиваться дополнительными издержками на производительность. Но такая страховка оказывается просто бесценной, если речь идет о повторно используемых кодовых базах. Имитация

компонентов очень удобна для проверки работоспособности браузеров. Но что делать с теми ошибками в браузерах, которые упорно не желают подвергаться тестированию?

Области непроверяемых ошибок в браузерах

К сожалению, в языке JavaScript и модели DOM имеется ряд проблемных областей, которые невозможно или слишком дорого проверить. Впрочем, подобные ситуации возникают довольно редко. Если же они все-таки возникают, то всегда имеет смысл исследовать их причины более углубленно. В последующих подразделах обсуждаются некоторые известные области ошибок, которые практически невозможно проверить обычными средствами взаимодействия с кодом JavaScript.

Привязка обработчиков событий

К числу самых неприятных погрешностей в работе браузеров относится отсутствие возможности определить, был при привязан обработчик событий. В частности, браузеры не позволяют никоим образом выяснить, была ли какая-нибудь функция привязана к приемнику событий в отдельном элементе. В силу этого из элемента просто невозможно удалить все привязанные обработчики событий, если только ссылки на них не поддерживаются при их создании.

Инициирование событий

Еще одна погрешность не позволяет определить, будет ли инициировано событие. Если еще можно каким-то образом определить, поддерживаются ли в браузере средства привязки событий, как это было уже не раз продемонстрировано ранее в главе, то совершенно *невозможно* выяснить, будет ли событие инициировано в браузере. А это может вызвать осложнения в следующих случаях.

Во-первых, если сценарий загружается динамически после загрузки самой веб-страницы, то в нем может быть предпринята попытка привязать приемник событий, чтобы дождаться загрузки окна, когда на самом деле данное событие уже наступило некоторое время назад. А поскольку определить этот факт уже нельзя, то код будет ожидать своего исполнения до бесконечности.

И во-вторых, если в сценарии предполагается обрабатывать отдельные события от браузера в качестве альтернативы. Например, браузер Internet Explorer предоставляет события типа `mouseenter` и `mouseleave`, чтобы упростить процесс определения того факта, что пользователь входит в пределы элемента или выходит за них. Эти события нередко служат в качестве альтернативы событиям типа `mouseover` и `mouseout`, потому что они действуют чуть более интуитивно, чем стандартные события. Но поскольку нельзя определить, будут ли они инициированы, если это произойдет без предварительной привязки событий и ожидания некоторых связанных с ними действий пользователя, то обработка таких событий в повторно используемом коде становится проблематичной.

Влияние свойств стилевого оформления

Еще одна неприятность связана с определением того влияния, которое изменение свойств стилевого оформления (CSS) оказывает на представление. Целый ряд свойств стилевого оформления оказывает влияние только на визуальное представление отображаемого элемента, не меняя окружающие его элементы или же другие свойства элемента, в том числе его цвет (`color`), цвет фона (`backgroundColor`) или непрозрачность (`opacity`). В силу этого нельзя определить программно, приведут ли изменения в этих

свойствах стилевого оформления к желаемым последствиям. А выявить их влияние можно только при визуальной проверке отображаемой страницы.

Аварийные сбои в браузерах

Тестирование сценария, приводящее к аварийному сбою в браузере, служит еще одним источником неприятностей. Код, приводящий к аварийному сбою в браузере, особенно трудно проверить. В отличие от исключений, которые легко перехватываются и обрабатываются, такой код всегда приводит к аварийному завершению работы браузера.

Например, в предыдущих версиях браузера Safari составление регулярного выражения с наборами символов уникода аналогично приведенному ниже всегда приводило к аварийному завершению работы этого браузера.

```
new RegExp("[\\w\u0128-\uFFFF*_-]+");
```

Дело в том, что проверить работоспособность такого регулярного выражения обычными средствами имитации компонентов нельзя, поскольку это будет всегда приводить к нежелательным результатам в старой версии данного браузера. Кроме того, с программными ошибками, приводящими к аварийным сбоям, очень трудно бороться, поскольку аварийные сбои браузеров для пользователей вообще неприемлемы, хотя в этих браузерах и можно запретить выполнение проблематичных сценариев JavaScript.

Несовместимость прикладных интерфейсов API

Как пояснялось ранее, в библиотеке jQuery запрещается изменять атрибут типа во всех браузерах из-за программной ошибки в браузере Internet Explorer. Такую операцию можно было бы проверить и отменить ее только в браузере Internet Explorer, но это привело бы к тому, что прикладной интерфейс API действовал бы по-разному в различных браузерах. В подобных случаях единственная возможность состоит в том, чтобы предоставить другое решение в обход области трудно преодолимой программной ошибкой.

Помимо областей непроверяемых ошибок и погрешностей, имеются также области, допускающие проверку, хотя произвести ее эффективно не так-то просто. Рассмотрим некоторые из этих областей.

Производительность прикладных интерфейсов API

Иногда отдельные прикладные интерфейсы API работают быстрее или медленнее в различных браузерах. Поэтому при написании надежного повторно используемого кода очень важно выбрать те прикладные интерфейсы API, которые обеспечивают приемлемую производительность. К сожалению, это не всегда очевидно. Но для эффективного анализа производительности отдельного компонента ему обычно требуется предоставить большой объем данных, а сам анализ производится относительно долго. Поэтому организовать такой анализ в коде не удастся таким же образом, как это делается для имитации компонентов.

Проверка правильности формирования Ajax-запросов

Подобная проверка представляет собой еще одну непреодолимую преграду. Как упоминалось выше при рассмотрении регрессий, в браузере Internet Explorer 7 была нарушена обработка запросов локальных файлов через объект типа `XMLHttpRequest`. Для того чтобы проверить, устранена ли эта программная ошибка, пришлось бы выполнить дополнительный запрос после загрузки каждой страницы, где предпринимается попытка подобного запроса. Но это неоптимальное решение. Помимо этого, вместе с библиотекой пришлось бы включить дополнительный файл, единственное назначение которого заключалось бы в существовании только для подобных запросов. И то и другое влечет за собой непомерные издержки, которые, безусловно, не стоят затраченных ресурсов и времени.

Неподдающиеся проверке средства и компоненты, конечно, доставляют немало хлопот, ограничивая возможности для эффективного написания повторно используемого кода JavaScript. Тем не менее можно всегда найти способ обойти эти трудные препятствия. Прежде всего, применяя альтернативные приемы или строя прикладные интерфейсы API таким образом, чтобы разрешить подобные затруднения, можно все же создавать достаточно эффективный код, несмотря на очевидные сложности.

Сокращение допущений

Написание кросс-браузерного кода — это своего рода состязание в допущениях. Но проведя тщательное исследование и применяя авторский подход к разработке, можно сократить число допущений, которые делаются в прикладном коде. Всякое допущение относительно создаваемого кода влечет за собой возникающие впоследствии осложнения.

Так, если допустить, что недостаток или программная ошибка будет присутствовать в отдельном браузере постоянно, такое допущение окажется чрезмерным. Напротив, тестирование на предмет ошибок, как это делалось в примерах, представленных ранее в этой главе, оказывается намного более эффективным. При написании прикладного кода следует всегда стремиться к сокращению числа допущений, по существу, уменьшая вероятность допустить ошибку и невольно создать условия для осложнений, которые могут болезненно отозваться впоследствии.

Чаще всего при написании сценариев JavaScript допускается обнаружение пользовательского посредника. В частности, сначала анализируется пользовательский посредник, предоставляемый браузером (`navigator.userAgent`), и на основании этого делается допущение относительно возможного поведения браузера. К сожалению, строковый анализ большинства пользовательских посредников становится источником большого числа вносимых впоследствии ошибок. Допустить, что программная ошибка будет всегда связана с конкретным браузером, — значит, накликать на себя беду.

Но у допущений имеется один существенный недостаток: исключить их полностью нельзя. В какой-то момент все равно придется допустить, что браузер будет делать именно то, что ему и полагается делать. Поэтому выявление того критического момента, когда это равновесие нарушается, остается полностью в руках разработчика. Именно здесь и проявляется отличие зрелого мастера от зеленого ученика.

В качестве примера рассмотрим еще раз код присоединения события, который уже неоднократно упоминался в этой главе:

```
function bindEvent(element, type, handle) {
  if (element.addEventListener) {
    element.addEventListener(type, handle, false);
  }
  else if (element.attachEvent) {
    element.attachEvent("on" + type, handle);
  }
}
```

Попробуйте сами определить три допущения, сделанные в данном коде. Действуйте смелее, а мы подождем.

(*Слышится мелодия "Опасная тема"...*)

Ну как, удалось? В приведенном выше коде было сделано по меньшей мере три допущения.

1. Проверяемые нами свойства на самом деле являются вызываемыми функциями.
2. Это именно те функции, которые подходят для выполнения предполагаемых действий.
3. Обе функции (или методы) являются единственными возможными средствами для привязки события.

Первым допущением можно было бы благополучно пренебречь, введя проверки, выявляющие, действительно ли анализируемые свойства являются функциями. Но тогда разрешить два остальных допущения было бы намного труднее.

В разрабатываемом коде следует всегда решать, сколько допущений можно себе позволить, исходя из требований к проекту и с учетом целевой аудитории. Зачастую сокращение числа допущений приводит к увеличению размера и сложности кодовой базы. Ничто, конечно, не мешает сократить допущения до разумного числа, но в определенный момент придется остановиться и оценить критическим взглядом, что уже имеется, а затем двигаться дальше, отталкиваясь от этого рубежа. Ведь даже код с наименьшим числом допущений может быть по-прежнему подверженным регрессиям, вносимым браузером.

Резюме

Итак, подведем краткий итог тому, что было рассмотрено в этой главе.

- Разработка кросс-браузерного кода предполагает учет трех факторов.
  - Размер кода. Размер исходного файла должен быть небольшим.
  - Издержки на производительность. Отрицательное влияние на производительность должно быть сведено к приемлемому уровню.
  - Качество прикладного интерфейса API. Предоставляемые прикладные интерфейсы API должны работать единообразно во всех браузерах.
- Магической формулы для определения оптимального сочетания перечисленных выше факторов не существует.
- Оптимальное сочетание факторов разработки каждому разработчику приходится подбирать самостоятельно.
- Правда, умело применяя такие специальные приемы, как обнаружение объектов и имитация компонентов, можно защитить повторно используемый код от любых напастей с самых разных сторон, не принося для этого непомерные жертвы.

В этой главе было уделено немало внимания обсуждению тех трудностей, которыми отличается разработка кросс-браузерного кода. А в следующей главе речь пойдет непосредственно о трудностях, обусловленных разной интерпретацией атрибутов, свойств и стилевого оформления в браузерах.

12 Обращение с атрибутами, свойствами и CSS

В этой главе...

- Представление о свойствах и атрибутах модели DOM
- Обращение с кросс-браузерными атрибутами и стилями
- Обращение со свойствами размерности элементов
- Открытие вычисленных стилей

Большая часть предыдущих глав этой книги, за исключением главы 11, была посвящена языку JavaScript. И хотя у языка JavaScript имеется немало своих особенностей, в которых необходимо хорошо разбираться, ситуация становится еще более запутанной, когда к этому примешивается еще и модель DOM в браузере.

Ясное представление о понятиях модели DOM и их связи с JavaScript имеет большое значение для становления мастера программирования на JavaScript, особенно если учесть, что непростые понятия модели DOM, на первый взгляд, противоречат всякой логике. Одни только понятия атрибутов и свойств модели DOM сбивают с толку многих авторов веб-страниц на JavaScript. Помимо отличий в поведении, имеется ряд областей применения атрибутов и свойств, чреватых программными ошибками и осложнениями кросс-браузерного характера.

Тем не менее понятия атрибутов и свойств очень важны. В частности, атрибуты являются неотъемлемой частью конструкции DOM, а свойства – основными средствами для хранения информации в элементах модели во время выполнения и доступа к этой информации. Рассмотрим следующий краткий пример, наглядно демонстрирующий потенциальный риск запутаться в этих понятиях:

```
<img src="../images/ninja-with-nunchuks.png">

<script type="text/javascript">

  var image = document.getElementsByTagName('img')[0];

  var newSrc = '../images/ninja-with-pole.png';

  image.src = newSrc;

  assert(image.src === newSrc,
        'the image source is now ' + image.src);

  assert(image.getAttribute('src') === '../images/ninja-with-nunchuks.png',
        'the image src attribute is ' + image.getAttribute('src'));

</script>
```

В приведенном выше фрагменте кода создается дескриптор изображения, получается ссылка на него и устанавливается новое значение в его свойстве `src`. На первый взгляд, все очень просто, но далее выполняются два теста, чтобы убедиться в следующем.

- Действительно ли свойство `src` получило переданное ему значение. Ведь если написать выражение `x = 213`, то, безусловно, следует ожидать, что переменная `x` будет содержать значение `213`.
- Остался ли атрибут прежним, поскольку он не был изменен.

Но если загрузить этот код в браузер, то обнаружится, что оба теста не проходят. В часности, свойство `src` не содержит присвоенное ему значение, а нечто вроде следующего:

```
http://localhost/ninja/images/ninja-with-pole.png
```

Этому свойству было присвоено конкретное значение. Следует ли в связи с этим ожидать, что оно будет содержать именно это значение? Еще более странным оказывается результат непрохождения второго теста. Несмотря на то что атрибут элемента разметки `src` не изменялся, его значение все-таки изменилось на следующее:

```
../images/ninja-with-pole.png
```

Что же произошло? Для ответа на данный вопрос в этой главе будут исследованы все загадочные ситуации, в которые браузеры ставят нас в отношении свойств и атрибутов элементов разметки, а также выяснены причины, по которым результаты тестирования прикладного кода оказываются не такими, как предполагалось. И начнем мы с подробного разъяснения самих понятий атрибутов и свойств.

Атрибуты и свойства модели DOM

Доступ к значениям атрибутов элементов разметки можно получить двумя путями: используя традиционные методы модели DOM (например, `getAttribute()` и `setAttribute()`) или же свойства самих объектов DOM, которые соответствуют атрибутам. Например, для того чтобы получить значение атрибута `id` элемента разметки, ссылка на который хранится в переменной `e`, можно воспользоваться одним из следующих способов:


```
e.getAttribute('id')
e.id
```

В любом случае получается значение атрибута `id`. Для того чтобы стало понятнее, каким образом ведут себя значения атрибутов и соответствующих им свойств, обратимся к примеру кода, приведенного в листинге 12.1.

Листинг 12.1. Доступ к значениям атрибутов через методы модели DOM и свойства

```
<div></div>

<script type="text/javascript">

  window.onload = function(){

    var div = document.getElementsByTagName("div")[0];

    div.setAttribute("id","ninja-1");
    assert(div.getAttribute('id') === "ninja-1",
          "Attribute successfully changed");

    div.id = "ninja-2";
    assert(div.id === "ninja-2",
          "Property successfully changed");

    div.id = "ninja-3";
    assert(div.id === "ninja-3",
          "Property successfully changed");
    assert(div.getAttribute('id') === "ninja-3",
          "Attribute successfully changed via property");

    div.setAttribute("id","ninja-4");
    assert(div.id === "ninja-4",
          "Property successfully changed via attribute");
    assert(div.getAttribute('id') === "ninja-4",
          "Attribute successfully changed");
  };

</script>
```

❶ Получить ссылку на элемент

❷ Проверить метод модели DOM

❸ Проверить значение свойства

❹ Проверить свойство и атрибут на соответствие

❺ Еще раз проверить свойство и атрибут на соответствие

В данном примере кода демонстрируется любопытное поведение в отношении свойств и атрибутов элементов разметки. Сначала в нем определяется простой элемент разметки `<div>`, используемый далее в качестве объекта для тестирования. В обработчике событий, вызываемом при загрузке веб-страницы, чтобы убедиться в том, что построение модели DOM завершено, сначала получается ссылка на единственный элемент разметки `<div>` ❶, а затем выполняется ряд тестов.

В первом тесте значение `"ninja-1"` атрибута `id` устанавливается с помощью метода `setAttribute()` ❷. Затем утверждается, что метод `getAttribute()` возвращает то же самое значение данного атрибута. И не должно быть ничего удивительного в том, что этот тест пройдет при загрузке веб-страницы. Аналогично в следующем тесте сначала устанавливается значение `"ninja-2"` свойства `id`, а затем проверяется, что это значение действительно изменилось, как и следовало ожидать.

В следующем тесте уже происходит нечто любопытное ❹. Здесь снова устанавливается новое значение свойства `id` (на этот раз `"ninja-3"`) и проверяется, изменилось ли значение этого свойства. Но затем утверждается также, что измениться должно не только значение свойства, но и значение *атрибута* `id`. И оба утверждения проходят. Из этого теста можно сделать следующий вывод: свойство `id` и атрибут `id` каким-то образом связаны вместе. При изменении значения свойства `id` изменяется также значение атрибута `id`. И в следующем тесте ❺ этот факт подтверждается иначе: при установке значения атрибута изменяется также значение соответствующего свойства.

Но сделанный выше вывод не должен вводить вас в заблуждение относительно того, что свойство и атрибут разделяют общее значение, поскольку это на самом деле не так. Как будет показано далее в этой главе, несмотря на существующую явную связь между атрибутом и свойством, их значения далеко не всегда одинаковы. Некоторое представление об этом уже было дано в начале главы. В отношении свойств и атрибутов необходимо иметь в виду пять следующих особенностей.

- Кросс-браузерное присваивание имен.
- Ограничения на присваивание имен.
- Отличия HTML от XML.
- Поведение специальных атрибутов.
- Производительность.

Рассмотрим каждую из этих особенностей по очереди.

Кросс-браузерное присваивание имен

Что касается присваивания имен атрибутам и свойствам, которые им соответствуют, то имена свойств обычно более постоянны среди браузеров. Так, если требуется получить доступ к свойству по определенному имени в одном браузере, то, скорее всего, это имя останется тем же и в других браузерах. Несмотря на некоторые расхождения, отличия в присваивании имен атрибутам более существенны, чем в присваивании имен свойствам.

Например, если в большинстве браузеров атрибут `class` может быть получен по имени `class`, то в браузере Internet Explorer — только по имени `className`. И объясняется это, скорее всего, тем, что свойство имеет имя `className`, и поэтому в браузере Internet Explorer имя атрибута и свойства согласованы. Обычно согласованность ничего плохого не сулит, но отличия в присваивании имен в браузерах способны привести в отчаяние.

В библиотеках вроде jQuery осуществляется нормализация расхождений в присваивании имен. Это дает пользователям библиотек возможность указывать одно имя независимо от платформы, а все необходимые преобразования выполняются в самой библиотеке незаметно для пользователей. Но и без помощи библиотек необходимо ясно понимать отличия в присваивании имен, чтобы правильно учитывать их при написании прикладного кода.

Ограничения на присваивание имен

Атрибуты передаются методам модели DOM в строковом представлении, и поэтому их можно именовать довольно свободно. А ссылки на имена свойств могут делаться как на идентификаторы посредством записи оператора-точки. На присваивание имен свойствам накладываются определенные ограничения, поскольку они должны соответствовать правилам, установленным для идентификаторов. Кроме того, в именах свойств нельзя использовать некоторые зарезервированные слова.

В спецификации по стандарту ECMAScript, доступной по адресу `http://www.ecma-international.org/publications/standards/Ecma-262.htm`, устанавливается, что некоторые ключевые слова нельзя использовать в качестве имен свойств, и поэтому для них определены альтернативные варианты. Например, атрибут `for` элементов разметки `<label>` представлен свойством `htmlFor`, поскольку слово `for` зарезервировано, тогда как атрибут `class` всех элементов разметки представлен свойством `className`, потому что слово `class` также зарезервировано. Кроме того, имена атрибутов, составленные из нескольких слов, например `readonly`, представлены свойствами, имеющими имена со смешанной формой написания (в данном случае `readOnly`). Другие примеры подобных различий в присваивании имен атрибутам и свойствам приведены в табл. 12.1.

Таблица 12.1. Отличия в именах атрибутов и свойств

Имя атрибута	Имя свойства
`for`	`htmlFor`
`class`	`className`
`readonly`	`readOnly`
`maxlength`	`maxLength`
`cellspacing`	`cellSpacing`
`rowspan`	`rowSpan`
`colspan`	`colSpan`
`tabindex`	`tabIndex`
`cellpadding`	`cellPadding`
`usemap`	`useMap`
`frameborder`	`frameBorder`
`contenteditable`	`contentEditable`

Следует также иметь в виду, что в стандарт HTML5 внедрены новые элементы разметки и атрибуты, которыми, возможно, придется дополнить приведенный выше перечень, когда этот стандарт устоится. К их числу относятся атрибуты `accessKey`, `contextMenu`, `dropZone`, `spellCheck`, `hrefLang`, `dateTime`, `pubDate`, `isMap`, `srcDoc`, `mediaGroup`, `autoComplete`, `noValidate` и `radioGroup`.

Отличия HTML от XML

Понятие свойств, автоматически соответствующих атрибутам, характерно только для модели HTML DOM. А для представления значений атрибутов в элементах модели XML DOM никаких свойств автоматически не создается. Следовательно, для получения значений атрибутов приходится обращаться к методам манипулирования атрибутами в традиционной модели DOM. Но это не такие уж и обременительные издержки, поскольку для XML-документов обычно не характерен такой же длинный перечень ошибок в присваивании имен, какой наблюдается в HTML-документах для атрибутов модели DOM.

Примечание

Если вас смущает термин *XML DOM*, то просто имейте в виду, что он обозначает объектную структуру, создаваемую в оперативной памяти для представления XML-документа таким же образом, как и модель HTML DOM представляет HTML-документ.

Было бы совсем неплохо организовать в коде определенного рода проверку, чтобы выяснить, относится ли элемент (или документ) к числу элементов (или документов) XML, чтобы продолжить выполнение кода соответствующим образом. Ниже приведен пример подобной проверки в теле функции.

```
function isXML(elem) {
  return (elem.ownerDocument ||
    elem.documentElement.nodeName.toLowerCase() !== "html";
}
```

Эта функция возвращает логическое значение `true`, если элемент относится к числу элементов XML, а иначе — логическое значение `false`.

Поведение специальных атрибутов

Далеко не все атрибуты представлены свойствами элементов разметки. И хотя это справедливо в целом для атрибутов, исходно указываемых в модели HTML DOM, *специальные атрибуты*, которые могут размещаться в элементах разметки веб-страниц, не становятся автоматически представленными свойствами этих элементов. Поэтому для доступа к значению специального атрибута придется вызывать методы `getAttribute()` и `setAttribute()` из модели DOM. Если вы не уверены, существует ли свойство для конкретного атрибута, то можете всегда проверить его наличие, и если оно не существует, выбрать резервный вариант. В приведенном ниже примере показано, как это реализуется непосредственно в коде.

```
var value = element.someValue ? element.someValue :
                                element.getAttribute('someValue');
```

Совет

В соответствии со стандартом HTML5 имена всех специальных атрибутов должны указываться с префиксом **`data-`**, чтобы обеспечить их достоверность по спецификации HTML5. Но это рекомендуется делать и в разметке документов по стандарту HTML4, чтобы обеспечить прочный задел на будущее. Кроме того, это неплохое условное обозначение имен, четко отделяющее специальные атрибуты от собственных.

Вопросы производительности

Как правило, непосредственный доступ к свойствам осуществляется быстрее, чем с помощью соответствующих методов манипулирования атрибутами в модели DOM, особенно в браузере Internet Explorer. Попробуем убедиться в этом сами. Как упоминалось в главе 2, для тестирования на производительность нужно измерить продолжительность многократного выполнения отдельной операции, поскольку однократное ее выполнение происходит настолько быстро, что не поддается точному измерению, о чем говорилось в главе 8 при обсуждении таймеров. Именно такой подход и принят для тестирования на производительность в коде из листинга 12.2.

В приведенном выше примере кода выполняется тест на производительность методов `getAttribute()` и `setAttribute()` модели DOM в сравнении с аналогичными операциями чтения и записи соответствующего свойства. Мы прогнали этот тест на

Листинг 12.2. Сравнение методов модели DOM со свойствами по производительности

```
<div id="testSubject"></div>

<script type="text/javascript">

  var count = 5000000;
  var n;
  var begin = new Date();
  var end;
  var testSubject = document.getElementById('testSubject');
  var value;

  for (n = 0; n < count; n++) {
    value = testSubject.getAttribute('id');
  }
  end = new Date();
  assert(true,'Time for DOM method read: ' +
        (end.getTime() - begin.getTime()));

  begin = new Date();
  for (n = 0; n < count; n++) {
    value = testSubject.id;
  }
  end = new Date();
  assert(true,'Time for property read: ' +
        (end.getTime() - begin.getTime()));

  begin = new Date();
  for (n = 0; n < count; n++) {
    testSubject.setAttribute('id','testSubject');
  }
  end = new Date();
  assert(true,'Time for DOM method write: ' +
         (end.getTime() - begin.getTime()));

  begin = new Date();
  for (n = 0; n < count; n++) {
    testSubject.id = 'testSubject';
  }
  end = new Date();
  assert(true,'Time for property write: ' +
        (end.getTime() - begin.getTime()));

</script>
```

Установить переменные заранее

Проверить чтение методом модели DOM

Проверить чтение свойства

Проверить запись методом модели DOM

Проверить запись свойства

нескольких браузерах и подытожили результаты наших испытаний на производительность в табл. 12.2, где все временные характеристики указаны в миллисекундах. Как видите, операции получения и установки свойства практически всегда выполняются быстрее, чем соответствующие методы `getAttribute()` и `setAttribute()` из моде-

ли DOM. А результаты выполнения данного теста в отдельно взятом браузере приведены на рис. 12.1.

Таблица 12.2. Результаты тестов на производительность методов модели DOM в сравнении с непосредственным доступом к свойству

Браузер	Метод `getAttribute()`	Получение свойства	Метод `setAttribute()`	Установка свойства
Internet Explorer 9	3970	940	7667	956
Firefox 14	827	434	1414	1584
Safari 5	268	142	1055	627
Chrome 21	294	159	1140	862
Opera 12	2109	1642	2370	1635

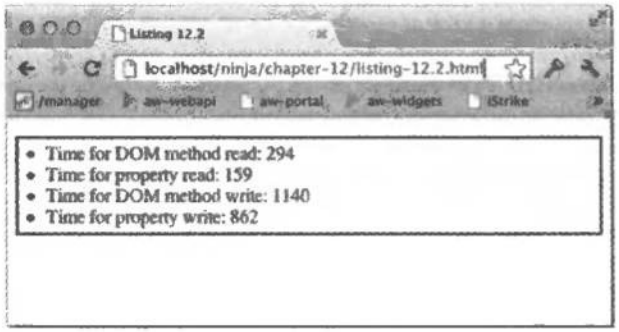

Рис. 12.1. Результаты выполнения теста на производительность в браузере Chrome

Примечание

Большая часть этих испытаний на производительность была проведена на переносном компьютере MacBook Pro 2011 года выпуска с процессором Intel Core i7 на 2,8 ГГц и ОЗУ на 8 Гбайт под ОС Mac OS X Lion. А испытания на производительность в браузере Internet Explorer были проведены на ПК с тем же самым процессором Intel Core i7 на 2,8 ГГц и ОЗУ на 4 Гбайт под ОС Windows 7 (64-разрядной версии).

Если при однократном выполнении операций эти отличия в производительности некритичны, то они становятся явно заметными при многократном выполнении операций, например, в сплошном цикле. Для повышения производительности, возможно, придется реализовать метод, чтобы получать с его помощью доступ к значению свойства, если оно существует. А если свойство отсутствует, то в качестве резервного варианта можно использовать метод из модели DOM. Рассмотрим для примера код, приведенный в листинге 12.3.

Листинг 12.3. Функция для установки и получения значений атрибутов

```
<div id="testSubject"></div>

<script type="text/javascript">
```



```
(function(){

  var translations = {
    "for": "htmlFor",
    "class": "className",
    readonly: "readOnly",
    maxlength: "maxLength",
    cellspacing: "cellSpacing",
    rowspan: "rowSpan",
    colspan: "colSpan",
    tabindex: "tabIndex",
    cellpadding: "cellPadding",
    usemap: "useMap",
    frameborder: "frameBorder",
    contenteditable: "contentEditable"
  };

  window.attr = function(element,name,value) {
    var property = translations[name] || name,
      propertyExists = typeof element[property] !== "undefined";

    if (typeof value !== "undefined") {
      if (propertyExists) {
        element[property] = value;
      }
    else {
      element.setAttribute(name,value);
    }
  }

  return propertyExists ?
    element[property] :
    element.getAttribute(name);
  };
})();

var subject = document.getElementById('testSubject');
assert(attr(subject,'id') === 'testSubject',
      "id value fetched");

assert(attr(subject,'id','other') === 'other',
      "new id value set");
assert(attr(subject,'id') === 'other',
      "new id value fetched");

assert(attr(subject,'data-custom','whatever') === 'whatever',
      "custom attribute set");
assert(attr(subject,'data-custom') === 'whatever',
      "custom attribute fetched");

</script>
```

❶ Образовать локальную область действия

❷ Составить таблицу преобразования

❸ Определить функцию получения и установки

Проверить новую функцию

В данном примере кода не только создается функция получения и установки значений атрибутов и свойств, но и демонстрируется целый ряд важных принципов и приемов, которыми можно пользоваться в прикладном коде. В этой функции требуется преобразовать имена атрибутов в имена свойств по табл. 12.1, для чего составляется специальная таблица преобразования ❷. Но это нужно сделать так, чтобы не засорять данной таблицей глобальное пространство имен. Поэтому она должна быть доступна для функции в ее локальной области действия, но не более того.

С этой целью определение таблицы преобразования и объявление функции заключаются в тело немедленно вызываемой функции, образующей локальную область действия ❶. В итоге таблица преобразования ❷ становится недоступной за пределами немедленно вызываемой функции, но функция получения и установки, которая также определяется в теле немедленно вызываемой функции ❸, получает доступ к этой таблице через свое замыкание. Согласитесь, что это очень изящный прием.

Еще один важный принцип демонстрируется в самой функции `attr()`: она может действовать одновременно как получатель и установщик, анализируя список своих аргументов. Так, если данной функции передается аргумент `value`, она действует как установщик, устанавливая в атрибуте переданное ей значение. А если при вызове данной функции аргумент `value` опускается, а передаются только два первых аргумента, она действует как получатель, извлекая значение из указанного атрибута.

Но в любом случае возвращается значение атрибута, что упрощает применение данной функции в любом из ее режимов работы по цепочке вызовов функций. Следует, однако, иметь в виду, что в рассмотренной здесь реализации данной функции не принимаются во внимание многие затруднения кросс-браузерного характера, препятствующие нормальному доступу к атрибутам. Поэтому выясним далее, в чем же заключаются эти затруднения.

Затруднения кросс-браузерного характера, возникающие при доступе к атрибутам

Как упоминалось ранее, разрешение затруднений кросс-браузерного характера при написании кода может оказаться нелегкой задачей. И при обращении к атрибутам подобных затруднений возникает немало. Поэтому в этом разделе рассматриваются лишь некоторые основные и наиболее часто встречающиеся затруднения, начиная с расширения имени модели DOM.

Расширение идентификатора или имени модели DOM

Самая неприятная программная ошибка, с которой приходится иметь дело при разработке веб-приложений, заключается в неверной реализации исходного кода модели DOM в браузерах. Как пояснялось в предыдущей главе, во всех браузерах из так называемой "Большой пятерки" значения атрибутов `id` или `name` извлекаются из элементов ввода данных в форму и вводятся в родительский элемент разметки `<form>` как свойства для ссылки на эти элементы. Образуемые в итоге свойства, по существу, заменяют любые свойства, которые уже могут присутствовать под тем же самым именем в элементе разметки формы.

Кроме того, в браузере Internet Explorer ссылками на элементы разметки заменяются не только свойства, но и значения атрибутов. Все эти затруднения наглядно демонстрируются в примере кода из листинга 12.4.

Листинг 12.4. Демонстрация силового воздействия браузеров на элементы разметки формы

```
<form id="testForm" action="/">
  <input type="text" id="id"/>
  <input type="text" name="action"/>
</form>

<script type="text/javascript">
  window.onload = function(){

    var form = document.getElementById('testForm');

    assert(form.id === 'testForm',
          "the id property is untouched");
    assert(form.action === '/',
          "the action property is untouched");

    assert(form.getAttribute('id') === 'testForm',
          "the id attribute is untouched");
    assert(form.getAttribute('action') === '/',
          "the action attribute is untouched");
  };
</script>
```

❶ Создать объект для тестирования

❷ Проверить, были ли свойства подавлены

❸ Проверить, были ли свойства искажены

Приведенные выше тесты демонстрируют, каким образом данное прискорбное затруднение способно вызвать потери данных разметки. Сначала в рассматриваемом здесь коде определяется HTML-форма с двумя порожденными элементами ввода данных ❶. Один порожденный элемент имеет атрибут `id` с идентификатором, а другой — атрибут `action` с именем. В первом наборе тестов ❷ утверждается, что свойства `id` и `action` элемента разметки формы должны быть установлены именно так, как это было исходно сделано в разметке HTML-формы. А во втором наборе тестов ❸ утверждается, что значения атрибутов отражают разметку. Но после выполнения теста в браузере Chrome получаются результаты, приведенные на рис. 12.2.

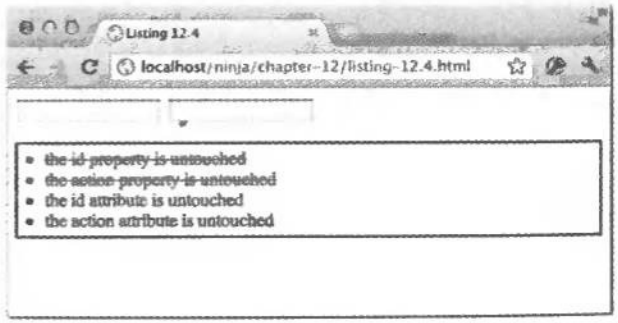

Рис. 12.2. Похоже, что значения свойств в разметке формы были подавлены!

При выполнении этих тестов во всех современных браузерах свойства `id` и `action` были перезаписаны ссылками на элементы ввода данных просто потому, что для этих

элементов были выбраны значения атрибутов id или name. В итоге исходные значения этих свойств исчезли без следа! Во всех браузерах, кроме Internet Explorer, можно получить исходные значения, используя методы из модели DOM для манипулирования атрибутами. А в браузере Internet Explorer заменяются даже эти значения.

Но настоящие мастера не привыкли отступать. Несмотря на то что в браузерах делается все возможное, чтобы избавиться от исходных значений свойств, это препятствие можно все же обойти специальным приемом, получив доступ к исходному узлу модели DOM, представляющему сам атрибут элемента. Ведь этот узел остается незатронутым порчей со стороны браузеров. Для того чтобы извлечь значение атрибута из узла модели DOM, например, атрибута action, можно воспользоваться следующей строкой кода:

```
var actionValue = element.getAttibuteNode("action").nodeValue;
```

В качестве упражнения попробуйте найти возможность воспользоваться данным приемом, чтобы расширить функцию attr(), разработанную в листинге 12.3, чтобы обнаружить момент, когда атрибут разметки формы заменен ссылкой на элементы, а иначе перейти к резервному варианту, чтобы получить значение из узла модели DOM, если он заменен.

Примечание

Если вас интересуют подобные затруднения, возникающие в связи с расширением элементов разметки, рекомендуем опробовать инструментальное средство DOMLint, разработанное Юрием Зайцевым, доступное по адресу http://kangax.github.com/domlint/ и способное анализировать веб-страницы на наличие потенциальных осложнений. А Гарретт Смит (Garrett Smith) написал на эту тему статью "Небезопасные имена для элементов управления HTML-форм" (Unsafe Names for HTML Form Controls), доступную по адресу http://jibbering.com/faq/names/.

Данное затруднение нельзя считать программной ошибкой, поскольку оно является предполагаемым поведением браузера. Тем не менее оно носит разрушительный характер и, безусловно, оказывается излишним, когда ссылки на элементы настолько просто получить с помощью таких методов, как document.getElementById(), и ему подобных. Но это далеко не единственная несуразность, связанная с тем, как браузеры обращаются с атрибутами. Рассмотрим далее еще одну.

Нормализация URL

Для всех современных браузеров характерна одна неприятная особенность, нарушающая принцип наименьшей неожиданности. При доступе к свойству, ссылающемуся на URL (например, href, src или action), этот URL автоматически преобразуется из указанной в полную, каноническую форму URL. О подобной автоматической нормализации URL упоминалось в начале этой главы, а теперь настало время продемонстрировать ее неприятные последствия на конкретном примере, как показано в листинге 12.5.

Листинг 12.5. Демонстрация неприятных последствий автоматической нормализации URL

```
<a href="listing-12.5.html" id="testSubject">Self</a>
<script type="text/javascript">
```



```
var link = document.getElementById('testSubject');

var linkHref = link.getAttributeNode('href').nodeValue;

assert(linkHref === 'listing-12.5.html',
      'link node value is ok');

assert(link.href === 'listing-12.5.html',
      'link property value is ok');

assert(link.getAttribute('href') === linkHref,
    'link attribute not modified');

</script>
```

❶ Получить исходное значение непосредственно из узла

❷ Проверить, совпадает ли исходное значение из узла с указанным в элементе разметки. Этот тест проходит

❸ Проверить, содержит ли свойство href то же самое, т.е. предполагаемое значение. Этот тест не проходит!

❹ Проверить, содержит ли атрибут предполагаемое значение. Этот тест проходит!

В приведенном выше примере кода сначала устанавливается дескриптор привязки с помощью атрибута href для обратной ссылки на ту же самую веб-страницу. Затем получается ссылка на этот элемент разметки для последующего тестирования. Далее применяется прием, рассматривавшийся в предыдущем разделе и состоящий в обращении к исходным узлам модели DOM для выявления исходного значения разметки ❶. Это значение проверяется, чтобы не доверять слепо работоспособности данного приема ❷.

Затем значение свойства href проверяется на совпадение ❸. Этот тест не проходит во всех современных браузерах, поскольку значение данного свойства нормализовано по полной форме URL. И наконец, проверяется, изменилось ли значение атрибута href ❹. Этот тест проходит во всех современных браузерах, кроме старых версий Internet Explorer.

Приведенные выше тесты не только демонстрируют характер рассматриваемого здесь затруднения, но и выявляют способ его преодоления: для того чтобы получить значение атрибута, которое определенно не было видоизменено, достаточно обратиться к узлу модели DOM, где это значение хранится. А для прежних версий браузера Internet Explorer, выпущенных до версии 8, подходит другой обходной прием: оригинальное расширение метода getAttribute() в браузере Internet Explorer. Так, если передать этому методу магическое число 2 в качестве второго аргумента, получаемое в итоге значение атрибута href не будет нормализовано, как показано ниже.

```
var original = link.getAttribute('href',2);
```

Любым из упомянутых выше обходных приемов можно воспользоваться в современных браузерах. Так, обращение к узлам модели DOM подходит для всех браузеров, тогда как любое значение, передаваемое методу getAttribute() в качестве второго аргумента, будет игнорироваться во всех современных браузерах, кроме Internet Explorer. В старых версиях браузера Opera может произойти (без всяких на то причин) аварийный сбой при передаче данному методу второго аргумента, поэтому избегайте этого приема в подобных версиях браузера Opera, если они включены вами в матрицу поддержки браузеров.

Вероятность осложнений в прикладном коде в связи с автоматической нормализацией URL довольно мала, если только в этом коде не возникнет абсолютная необходимость в получении ненормализованных значений. А теперь перейдем к другому затруднению, которое может иметь намного дальше идущие последствия.

Атрибут стилевого оформления

К числу тех атрибутов элементов разметки, которые особенно трудно поддаются установке и получению значений, относится атрибут `style`. У элементов HTML DOM имеется специальное свойство `style`, доступное для получения информации о стилевом оформлении элемента (например, `elem.style.color`). Но если требуется получить исходную символьную строку стилевого оформления, заданную в атрибуте `style` этого элемента, то сделать это будет намного труднее. В качестве примера рассмотрим следующую разметку:

```
<div style='color:red;'></div>
```

Что, если требуется получить исходную символьную строку `color:red;`? Свойство `style` для этого вообще не подходит, поскольку оно устанавливается в объекте, содержащем результаты синтаксического анализа исходной символьной строки. И хотя вызов метода `getAttribute("style")` нормально выполняется в большинстве браузеров, в браузере Internet Explorer этого все же не происходит. Вместо этого в браузере Internet Explorer употребляется свойство `cssText`, которое хранится в объекте `style`. С помощью этого свойства можно получить исходную символьную строку стилевого оформления, например `elem.style.cssText`.

Потребность в непосредственном получении исходного значения атрибута `style` может возникать сравнительно реже, чем в доступе к получающемуся в итоге объекту `style`. Но в браузерах возникает еще одно затруднение, которое может оказать неблагоприятное воздействие на любую веб-страницу, где элементы модели DOM формируются во время выполнения.

Атрибут типа

В версии Internet Explorer 8 и более ранних версиях этого браузера существует еще одно скрытое препятствие, которое связано с атрибутом `type` элементов разметки `<input>` и для устранения которого отсутствует какой-нибудь приемлемый обходной прием. Как только элемент разметки `<input>` вставляется в документ, его атрибут `type` уже не подлежит изменению. В действительности при всякой попытке изменить его в браузере Internet Explorer генерируется соответствующее исключение. В качестве примера рассмотрим код из листинга 12.6, в котором предпринимается попытка изменить тип элемента ввода данных задним числом.

Листинг 12.6. Попытка изменить тип элемента ввода данных после его вставки в документ

```
<form id="testForm" action="/"></form>

<script type="text/javascript">
  window.onload = function(){

    var input = document.createElement('input');

    input.type = 'text';
    assert(input.type == 'text',
          'Input type is text');

    document.getElementById('testForm')
            .appendChild(input);
```

❶ Создать новый элемент ввода данных, установив атрибут type по умолчанию

❷ Установить свойство type и проверить его

❸ Вставить новый элемент ввода данных в модель DOM


```
    input.type = 'hidden';
    assert(input.type == 'hidden',
          'Input type changed to hidden');
  });

</script>
```

❹ Изменить тип элемента ввода данных после его вставки

В приведенном выше примере кода сначала создается новый элемент разметки `<input>` ❶, которому присваивается тип `text`, а затем утверждается, что это присваивание было успешным ❷. Далее новый элемент вводится в модель DOM ❸. После ввода тип этого элемента изменяется на `hidden` и затем утверждается, что данное изменение имело место ❹. Во всех современных браузерах, кроме Internet Explorer, все эти тесты проходят успешно. А в версии Internet Explorer 8 и более ранних версиях этого браузера при попытке присваивания генерируется исключение, и поэтому второй тест вообще не выполняется.

Для преодоления данного скрытого препятствия можно принять две временные меры, хотя сделать это будет непросто.

- Вместо того чтобы пытаться изменить атрибут `type`, можно создать новый элемент разметки `<input>`, скопировать все свойства и атрибуты и заменить исходный элемент вновь созданным. На первый взгляд такое решение кажется достаточно простым, но реализовать его не так-то просто. Во-первых, используя методы в модели DOM второго уровня, нельзя заранее выяснить, были ли установлены для данного элемента какие-нибудь обработчики событий, если только не отслеживать их самостоятельно. И во-вторых, любые ссылки на исходный элемент становятся недействительными.
- Просто пресечь любые попытки изменить значение атрибута `type` во всяком прикладном интерфейсе API, создаваемом для внесения изменений в свойства или атрибуты.

Ни одно из предложенных выше решений нельзя считать полностью удовлетворительным. В библиотеке jQuery применяется второй подход. При этом генерируется информационное исключение, если пользователи попытаются изменить атрибут `type` после вставки элемента ввода данных в документ. Очевидно, что это компромиссное решение, но, по крайней мере, оно позволяет сохранить согласованность пользовательского восприятия веб-содержимого на всех платформах. Правда, рассматриваемое здесь затруднение уже устранено в браузере Internet Explorer 9.

А теперь рассмотрим еще одну неприятную особенность браузеров сбивать разработчиков веб-приложений с толку. И на этот раз она связана с элементами разметки формы.

Трудности определения индекса перехода по табуляции

Еще одним неприятным препятствием для разработки веб-приложений является определение индекса элемента для перехода по табуляции, поскольку не существует общего единодушия относительного того, как он должно происходить. И хотя имеется возможность получить индекс для перехода по табуляции с помощью свойства `tabIndex` или атрибута `"tabindex"` тех элементов, где они явно определены, браузер обычно возвращает нулевое значение для свойства `tabIndex` и пустое значение для атрибута `"tabindex"` тех элементов, где они *не* определены явно. И это, конечно, озна-

чает, что заранее нельзя знать, какой именно индекс перехода по табуляции присвоен тем элементам разметки, где значение этого индекса не задано явным образом.

Это довольно сложный вопрос, но его решение становится особенно важным с точки зрения практичности и доступности веб-приложений. И наконец, рассмотрим еще одно, последнее затруднение, связанное с атрибутами. Хотя на самом деле оно не имеет к никакого отношения к атрибутам.

Имена узлов

Определение имени узла, по существу, не имеет прямого отношения к атрибутам, тем не менее оно вызывает не меньше трудностей. Ранее был рассмотрен целый ряд обходных приемов, опирающихся на обращение к узлам модели DOM. Но оказывается, что определить имя самого узла не так-то просто.

В частности, чувствительность к регистру в имени узла может меняться в зависимости от типа документа. Если это обычный HTML-документ, то свойство nodeName будет возвращать имя элемента, набранное всеми прописными буквами (например, "HTML" или "BODY"). Но если это XML- или XHTML-документ, то свойство nodeName будет возвращать имя, указываемое пользователем прописными, строчным буквами или же в смешанном написании.

Обычное решение для преодоления этого препятствия состоит в том, чтобы нормализовать имя перед любым его сравнением, для чего его написание, как правило, преобразуется в строчные буквы. Допустим, требуется выполнить некоторую операцию только над элементами разметки div> и <ul>. Заранее неизвестно, в каком именно написании будут получены имена узлов: div, DIV или dIv, и поэтому эти имена следует нормализовать, как показано в приведенном ниже фрагменте кода.

```
var all = document.getElementsByTagName("*")[0];

for (var i = 0; i < all.length; i++) {
  var nodeName = all[i].nodeName.toLowerCase();
  if (nodeName === "div" || nodeName === "ul") {
     all[i].className = "found";
  }
}
```

Если точно известно, в каком именно типе документа должен выполняться прикладной код, то можно и не беспокоиться о подобной чувствительности к регистру. Но если приходится писать повторно используемый код, который должен выполняться в любой среде, то лучше предусмотрительно выполнить нормализацию имен узлов.

В этом разделе были рассмотрены затруднения, связанные с атрибутами и свойствами элементов разметки, а также небольшое затруднение, которое вызывает обращение к свойству style. Но это была лишь малая толика тех неприятностей, которые браузеры припасли для разработчиков веб-приложений в отношении стилевого оформления. Поэтому в следующем разделе мы рассмотрим наиболее характерные трудности, которые возникают при обращении с атрибутами стилевого оформления в браузерах.

Трудности обращения с атрибутами стилевого оформления

Аналогично общим атрибутам, получение доступа и установка атрибутов стилевого оформления может вызвать серьезные трудности. Как и для обращения с атрибутами и

свойствами, рассмотренными в предыдущем разделе, в данном случае имеются два подхода: обращаться непосредственно к значению атрибута style или же к образуемому из него свойству элемента разметки.

Из этих двух подходов чаще всего применяется обращение к свойству style элемента разметки, которое содержит не символьную строку, а объект со свойствами, соответствующими значениям стилевого оформления, указанным в разметке элемента. Помимо этого, имеется прикладной интерфейс API для доступа к текущей, вычисленной информации о стилевом оформлении элемента, где *вычисленная информация о стилевом оформлении* по существу означает конкретные стили, которые будут применяться к элементу после вычисления информации о всех наследуемых и применяемых стилях. В этом разделе поясняется, что нужно знать о том, как стилевое оформление интерпретируется в браузерах. И начнем мы с выяснения места, где регистрируется информация о стилевом оформлении.

Местонахождение стилей

Информация о стилевом оформлении, находящаяся в свойстве style элемента модели DOM, первоначально устанавливается, исходя из значения, указанного в атрибуте style разметки данного элемента. Например, в результате операции присваивания style="color:red;" информация о стилевом оформлении размещается в объекте style. А во время выполнения сценария на веб-странице значения свойств могут быть установлены или видоизменены в объекте style, причем эти изменения будут активно воздействовать на порядок отображения элемента.

Многие авторы сценариев с разочарованием обнаруживают, что ни одно из значений элементов <style> разметки страницы или внешних таблиц стилей недоступно в объекте style данного элемента. Но настоящим мастерам не пристало пребывать долго в разочаровании, и поэтому мы попытаемся далее найти способ получить подобную информацию. А до тех пор выясним, каким образом свойство style получает свои значения. С этой целью обратимся к примеру кода, приведенного в листинге 12.7.

Листинг 12.7. Исследование свойства style

```
style>
  div { font-size: 1.8em; border: 0 solid gold; }
</style>

<div style="color:#000;" title="Ninja power!">
  " "忍者パワー
"</div>

<script>

  window.onload = function(){

    var div = document.getElementsByTagName("div")[0];

    assert(div.style.color == 'rgb(0, 0, 0)' ||
           div.style.color == '#000',
           'color was recorded');
```

❶ Объявить страничную таблицу стилей с информацией о размере шрифта и обрамлении

❷ Этот тестируемый элемент должен получить несколько стилей из разных мест, включая его собственный атрибут style и таблицу стилей

❸ Проверить, зарегистрирован ли встраиваемый стиль цвета

```
    assert(div.style.fontSize == '1.8em',
           'fontSize was recorded');

    assert(div.style.boderWidth == '0',
           'borderWidth was recorded');

    div.style.borderWidth = "4px";

    assert(div.style.borderWidth == '4px',
           'borderWidth was replaced');
  };

</script>
```

❹ Проверить, зарегистрирован ли наследуемый стиль размера шрифта

❺ Проверить, зарегистрирован ли наследуемый стиль ширины обрамления

❻ Заменить стиль ширины обрамления

❼ Проверить результат замены стиля

В данном примере кода сначала создается элемент разметки `<style>` для установки внутренней таблицы стилей, значения в которой будут далее применяться к элементам разметки веб-страницы ❶. В этой таблице стилей определяются все элементы разметки `<div>`, текст в которых должен появляться на странице выделенными шрифтом, имеющим размер в 1,8 раза крупнее, чем выбираемый по умолчанию, и обрамленным сплошной золотистой рамкой нулевой ширины. Это означает, что любые элементы, к которым применяется такое стилевое оформление, будут иметь обрамление, которого, впрочем, не будет видно из-за его нулевой ширины. Затем создается элемент разметки **`<div>`** с атрибутом встраиваемого стиля, определяющим черную окраску текста в данном элементе разметки ❷.

Далее начинается тестирование. Как только будет получена ссылка на элемент разметки `<div>`, проверяется, получил ли атрибут `style` свойство `color`, представляющее цвет, присвоенный данному элементу ❸. Следует иметь в виду, что в большинстве браузеров свойство `color` нормализуется в формате RGB, когда его значение устанавливается в свойстве `style`, несмотря на то, что во встраиваемом стиле оно указано в формате `#000`. Именно поэтому оно и проверяется в обоих форматах. Как показано на рис. 12.3, данный тест проходит.

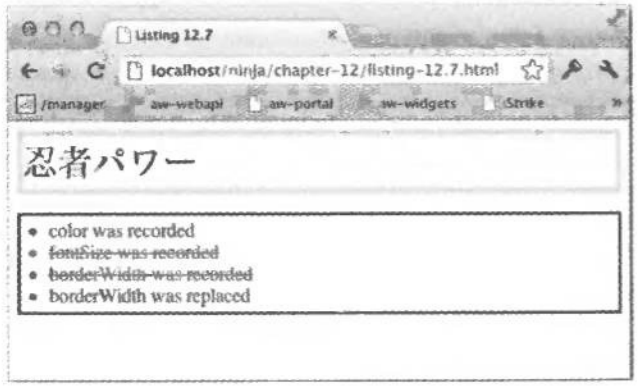

Рис. 12.3. Как показывают результаты тестирования, встраиваемые и присваиваемые стили регистрируются, тогда как наследуемые стили не регистрируются

Предупреждение

Нормализация цвета в браузерах не всегда происходит согласованно. В большинстве браузеров цвета нормализуются в формате RGB, но в некоторых браузерах цвета оставляются указанными как именованные (например, `black`).

Далее проверяется, зарегистрированы ли в объекте `style` размер шрифта ❹ и ширина обрамления ❺, указанные в таблице встраиваемых стилей. И хотя на рис. 12.3 показано, что стиль, определяющий размер шрифта, применен к элементу разметки, его тест тем не менее не проходит. Дело в том, что содержимое объекта `style` не отражает информацию о стилевом оформлении, наследуемую из вложенных таблиц стилей (CSS).

После этого значение свойства `borderWidth` в объекте `style` изменяется путем присваивания на ширину **4** пикселя ❻, а далее проверяется, было ли применено это изменение ❼. Как следует из рис. 12.3, тест данного изменения проходит, и невидимое ранее обрамление теперь окружает элемент разметки страницы. В результате такого присваивания значение свойства `borderWidth` оказывается в свойстве `style` данного элемента разметки, что и подтверждает тест ❼.

Следует иметь в виду, что любые значения, устанавливаемые в свойстве `style` элемента разметки, будут получать приоритет над всем, что наследуется из таблицы стилей, даже если в правиле составления таблицы стилей используется аннотация `!important` (важно!). Возможно, вы обратили внимание на то, что в коде из листинга 12.7 свойство размера шрифта обозначается в таблице CSS как `font-size`, а в сценарии — как `fontSize`. Чем объясняется такое расхождение? Ответ на этот вопрос мы постараемся найти в следующем разделе.

Именование свойств стилевого оформления

С атрибутами стилевого оформления связано относительно немного затруднений кросс-браузерного характера, если речь идет о доступе к значениям, предоставляемым браузером. Тем не менее существуют отличия в именовании стилей в таблицах CSS, сценариях, а также в браузерах.

Имена атрибутов стилевого оформления в таблицах CSS, составленные из нескольких слов, обычно разделяются дефисом, например `font-weight`, `font-size` и `background-color`. Напомним, что в JavaScript имена свойств также *могут* разделяться дефисом, но в таком случае они становятся недоступными через оператор-точку. Например, приведенная ниже строка кода считается вполне допустимой.

```
var color = element.style['font-size'];
```

А следующая строка кода является недопустимой:

```
var color = element.style.font-size;
```

Синтаксический анализатор JavaScript интерпретирует дефис в этой строке кода как оператор вычитания, что, вероятнее всего, приведет к малоутешительным последствиям. Поэтому имена стилей в таблицах CSS обычно приводятся к смешанному написанию, когда они употребляются в качестве имен свойств, чтобы не вынуждать разработчиков веб-страниц пользоваться только общей формой доступа к свойствам. Таким образом, имя `font-size` преобразуется в `fontSize`, имя `background-color` — в `backgroundColor`. Для того чтобы не помнить об этом постоянно, можно написать

простой прикладной интерфейс API, в котором стили устанавливаются и получаются с автоматическим приведением к форме смешанной записи, как показано в листинге 12.8.

Листинг 12.8. Простой способ организации доступа к стилям

```
<div style="color:red;font-size:10px;background-color:#eee;"></div>

<script type="text/javascript">
  function style(element,name,value){
    name = name.replace(/-([a-z])/ig,
                        function(all,letter){
                          return letter.toUpperCase();
                        });

    if (typeof value !== 'undefined') {
      element.style[name] = value;
    }

    return element.style[name];
  }

  window.onload = function(){

    var div = document.getElementsByTagName('div')[0];

    assert(true,style(div,'color'));
    assert(true,style(div,'font-size'));
    assert(true,style(div,'background-color'));
  };
</script>
```

Определить функцию обращения к стилям

Привести имя к смешанному написанию

Установить значение, если оно предоставлено

Возвратить значение

За исключением приведения параметра name к смешанной форме написания, функция `style()` действует в данном примере кода практически так же, как и функция `attr()`, рассматривавшаяся ранее в листинге 12.3, и поэтому мы не будем повторяться, поясняя принцип ее действия. Если же вас смущает вид регулярного выражения, используемого для приведения имен стилей к смешанному написанию, просмотрите еще раз материал главы 7. Следует также заметить, что никакого тестирования функции `style()` на самом деле не производится, несмотря на наличие вызовов метода `assert()` в ее теле. В данном случае утверждение служит для упрощенного вывода результатов на страницу, как показано на рис. 12.4.

В качестве упражнения составьте ряд утверждений для тщательного тестирования этой новой функции. Как упоминалось ранее, существует целый ряд проблемных свойств стилевого оформления, которые по-разному интерпретируются в браузерах. Поэтому ниже будет рассмотрено одно из них.

Свойство стилевого оформления `float`

Одно из самых главных затруднений в области атрибутов и свойств стилевого оформления связано с тем, как обращаться к свойству `float`. В силу того что свойство `float` вступает в конфликт с ключевым словом `float`, встроенным в JavaScript, браузерам приходится предоставлять для него альтернативное имя.

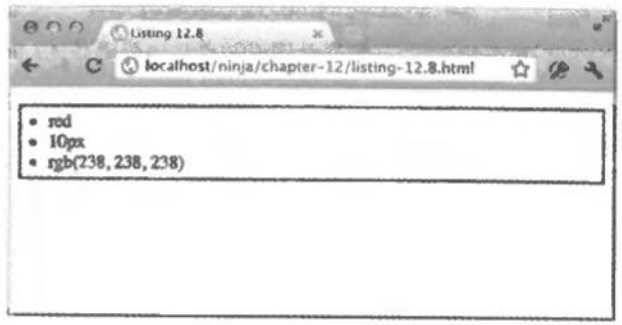

Рис. 12.4. В результате прогона функции `style()` в браузере выясняется, что она способна автоматически привести имя стиля из таблицы CSS к правильному написанию имени свойства стилевого оформления

И как часто бывает в подобных случаях, в браузерах, придерживающихся стандартов, для выхода из этого затруднительного положения был избран один путь, а в браузере Internet Explorer — совсем другой. Практически во всех браузерах обозначение `cssFloat` используется в качестве альтернативного имени данного свойства, тогда как в браузере Internet Explorer — обозначение `styleFloat`, как это ни прискорбно. Для правильной интерпретации подобных различий можно воспользоваться способом преобразования имен стилей, представленным в листинге 12.3, внеся соответствующие изменения в функцию `style()` из листинга 12.8.

Ранее в этой главе было показано, каким образом значения цвета можно преобразовать из одного формата в другой, когда они вводятся в качестве свойства стилевого оформления. А теперь рассмотрим другую ситуацию, возникающую при стилевом оформлении.

Преобразование значений, указываемых в пикселях

При установке значений атрибутов стилевого оформления следует обращать особое внимание на присваивание числовых значений, указываемых в пикселях. Так, в устаревших атрибутах вроде `height` дескриптора `<img>` числовые значения просто указывались в пикселях, а браузеру предоставлялась возможность самому разбираться с единицами измерения задаваемых величин. Если же присваивать значения, указываемые в пикселях, свойствам стилевого оформления, это может доставить немало хлопот.

При установке числового значения в свойстве стилевого оформления необходимо указывать одинаковые единицы измерения для надежной работы прикладного кода во всех браузерах. Допустим, требуется установить значение 10 пикселей в свойстве `height` стилевого оформления элемента разметки веб-страницы. Для этой цели можно воспользоваться одним из следующих двух способов, обеспечивающих надежную работу прикладного кода во всех браузерах:

```
element.style.height = "10px";
element.style.height = 10 + "px";
```

А приведенный ниже способ нельзя считать надежным.

```
element.style.height = 10;
```

Казалось бы, достаточно добавить немного логики в функцию `style()` из листинга 12.8, чтобы присоединить обозначение единиц измерения "px" к числовому значению, передаваемому данной функции, но не все так просто! Далеко не все числовые значения представлены в пикселях! Существует целый ряд свойств стилевого оформления, принимающих числовые значения в других единицах измерения. К их числу относятся следующие свойства:

- `z-index`
- `font-weight`
- `opacity`
- `zoom`
- `line-height`

Для указания числовых значений, присваиваемых этим (и любым другим мыслимым) свойствам, вы можете соответственно расширить функцию `style()` из листинга 12.8, чтобы автоматически интерпретировать значения, указываемые в других единицах измерения, кроме пикселей.

Кроме того, для чтения значений, указываемых в пикселях, из атрибута стилевого оформления следует пользоваться методом `parseFloat()`, чтобы в любом случае получить предполагаемое значение. А теперь перейдем к рассмотрению ряда важных свойств стилевого оформления, обращение с которыми может вызвать определенные трудности.

Указание размеров по высоте и ширине

Обращение со свойствами стилевого оформления вроде `height` и `width` вызывает особые трудности, поскольку они принимают значение `auto`, если оно не указано специально. Это означает, что размеры элемента разметки веб-страницы устанавливаются автоматически, исходя из его содержимого. В итоге свойствами `height` и `width` нельзя воспользоваться для получения точных значений, если только не указать их значения явным образом в символьной строке соответствующего атрибута стилевого оформления.

Правда, свойства `offsetHeight` и `offsetWidth` обеспечивают довольно надежный способ доступа к правильному значению высоты и ширины элемента. Но не следует забывать, что значения, присваиваемые этим двум свойствам, включают в себя заполнение элемента разметки. Такая информация зачастую весьма желательна, если требуется расположить один элемент над другим. Но иногда информацию о размерах элементов требуется получить как с указанием границ и заполнения, так и без них.

Следует, однако, иметь в виду, что на веб-сайтах с повышенной интерактивностью элементы разметки веб-страниц могут находиться некоторое время в неотображаемом состоянии, когда значение свойства `display` стилевого оформления установлено равным `none`. А когда такой элемент не отображается, то у него отсутствуют размеры. И в результате любой попытки извлечь значения свойств `offsetWidth` и `offsetHeight` неотображаемого элемента разметки будет получено нулевое значение.

Если требуется получить нескрытые размеры скрываемых подобным образом элементов, то с этой целью можно воспользоваться специальным приемом, чтобы раскрыть элемент разметки на мгновение, извлечь из него значение, а затем скрыть его снова. Разумеется, это нужно сделать совершенно незаметно, выполнив следующую последовательность действий.

1. Изменить значение свойства `display` на `block`.
2. Установить значение `hidden` свойства `visibility`.
3. Установить значение `absolute` свойства `position`.
4. Извлечь значения размеров.
5. Восстановить исходные значения упомянутых выше свойств.

Благодаря изменению значения свойства `display` на `block` появляется возможность извлечь конкретные значения свойств `offsetHeight` и `offsetWidth`, но в таком случае элемент разметки включается в отображаемую часть веб-страницы, а следовательно, он становится видимым. А для того чтобы сделать его невидимым, устанавливается значение `hidden` свойства `visibility`. Но здесь, как всегда, возникает следующая загвоздка: на месте неотображаемого элемента разметки веб-страницы появляется крупная прореха. Поэтому свойству `position` присваивается значение `absolute`, чтобы исключить его из обычной последовательности отображения содержимого веб-страницы. Реализовать такой прием оказывается намного проще, чем описать его суть, что и демонстрирует пример кода из листинга 12.9.

Листинг 12.9. Извлечение размеров скрытых элементов разметки

```
<div>
  Lorem ipsum dolor sit amet, consectetur adipiscing elit.
  Suspendisse congue facilisis dignissim. Fusce sodales,
  odio commodo accumsan commodo, lacus odio aliquet purus,
  <img src="../images/ninja-with-pole.png" id="withPole" />
  <img src="../images/ninja-with-shuriken.png"
  id="withShuriken" style="display:none" />
  vel rhoncus elit sem quis libero. Cum sociis natoque
  penatibus et magnis dis parturient montes, nascetur
  ridiculus mus. In hac habitasse platea dictumst. Donec
  adipiscing urna ut nibh vestibulum vitae mattis leo
  rutrum. Etiam a lectus ut nunc mattis laoreet at
  placerat nulla. Aenean tincidunt lorem eu dolor commodo
  ornare.
</div>

<script type="text/javascript">

  (function(){                                  ❶ Создать локальную область действия

    var PROPERTIES = {                          ❷ Определить целевые свойства
      position: "absolute",
      visibility: "hidden",
      display: "block"
    };

    window.getDimensions = function(element) {  ❸ Создать новую функцию
      var previous = {};                        ❹ Запомнить установки
      for (var key in PROPERTIES) {
        previous[key] = element.style[key];
        element.style[key] = PROPERTIES[key];   ❺ Заменить установки
      }
```

```
    var result = {
      width: element.offsetWidth,
      height: element.offsetHeight
    };

    for (key in PROPERTIES) {
      element.style[key] = previous[key];
    }
    return result;
  };
})();

window.onload = function() {

  setTimeout(function(){

    var withPole = document.getElementById('withPole'),
        withShuriken = document.getElementById('withShuriken');

    assert(withPole.offsetWidth == 41,
           "Pole image width fetched; actual: " +
           withPole.offsetWidth + ", expected: 41");
    assert(withPole.offsetHeight == 48,
           "Pole image height fetched: actual: " +
           withPole.offsetHeight + ", expected 48");

    assert(withShuriken.offsetWidth == 36,
           "Shuriken image width fetched; actual: " +
           withShuriken.offsetWidth + ", expected: 36");
    assert(withShuriken.offsetHeight == 48,
           "Shuriken image height fetched: actual: " +
           withShuriken.offsetHeight + ", expected 48");

    var dimensions = getDimensions(withShuriken);

    assert(dimensions.width == 36,
           "Shuriken image width fetched; actual: " +
           dimensions.width + ", expected: 36");
    assert(dimensions.height == 48,
           "Shuriken image height fetched: actual: " +
           dimensions.height + ", expected 48");
  },3000);
}

</script>
```

❻ Извлечь размеры

❼ Восстановить установки

❽ Проверить видимые элементы

❾ Проверить скрытые элементы

❿ Использовать новую функцию

⓫ Еще раз проверить элементы

Приведенный выше листинг довольно длинный, но большую его часть занимает тестовый код, тогда как реализация новой функции для извлечения размеров элемента разметки – лишь около десятка строк кода. Рассмотрим данный пример кода по частям. Сначала в нем устанавливается ряд элементов разметки для последующего тестирования. В частности, элемент разметки `<div>` содержит фрагмент текста, заверстанный

вокруг двух изображений, и выровненный по левому краю, для чего используются стили из внешней таблицы стилей. А элементы разметки обоих изображений устанавливаются в качестве объектов для последующего тестирования. С этой целью один из них сделан видимым, а другой – невидимым.

Перед выполнением сценария элементы разметки веб-страницы выглядят так, как показано на рис. 12.5. Если не скрыть второе изображение ниндзя, оно появится рядом с первым.

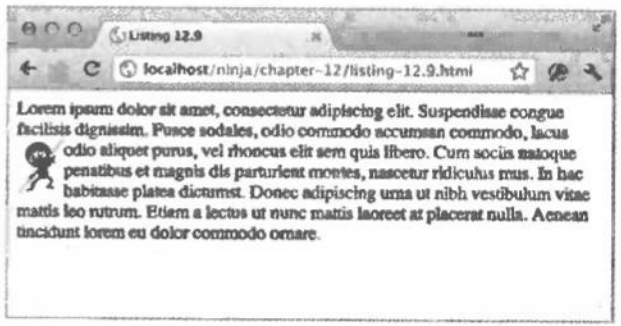

Рис. 12.5. Для тестирования рассматриваемого здесь приема выборки размеров элементов используются два изображения: одно – видимое, другое – скрытое

Затем определяется новая функция. Для хранения важной информации предполагается использовать информационный массив, поэтому, как и в коде из листинга 12.3, объявление локальной переменной и новой функции заключается в теле немедленно вызываемой функции, чтобы образовать локальную область действия и замыкание ❶. Локальный информационный массив, предназначенный для хранения свойств, которыми предполагается в дальнейшем манипулировать, определяется и заполняется тремя свойствами и значениями для замены ❷.

Далее объявляется новая функция для выборки размеров элемента разметки ❸. В качестве аргумента эта функция принимает элемент разметки, размеры которого требуется выбрать для последующей обработки. В теле этой функции сначала создается информационный массив `previous` ❹, в котором будут храниться предыдущие значения свойств стилевого оформления, чтобы их можно было заменить, а затем и восстановить. С этой целью организуется циклическое обращение к каждому предыдущему свойству и их замена новыми значениями ❺.

Сделав все это, можно далее перейти к выборке размеров элемента разметки, который теперь невидим и имеет абсолютное положение в отображаемой компоновке веб-страницы. Выбранные размеры этого элемента сохраняются в информационном массиве, присвоенном локальной переменной `result` ❻. После извлечения нужной информации следы этой скрытной операции тщательно заметаются путем восстановления исходных значений свойств стилевого оформления ❼, в результате чего возвращается информационный массив, содержащий свойства `width` и `height`. Все это выглядит красиво в теории, а как оно будет работать на практике? Попробуем найти ответ и на этот вопрос.

В обработчике событий, связанных с загрузкой веб-страницы, тесты выполняются по обратному вызову таймера, срабатывающего через 3 секунды. А зачем это нужно? Подобным образом в обработчике событий гарантируется, что тест не будет выпол-

няться до тех пор, пока не станет известно, что модель DOM построена. А таймер позволяет наблюдать отображаемое содержимое веб-страницы по ходу выполнения теста, исключая сбои в отображении на время манипулирования свойствами скрытого элемента. Ведь если нормальное отображение так или иначе нарушится при выполнении функции выборки размеров скрытого элемента, то рассматриваемый здесь прием никуда не годится.

В функции обратного вызова таймера сначала получается ссылка на объекты тестирования (оба изображения), а затем утверждается, что размеры видимого изображения могут быть получены из свойств `offsetHeight` и `offsetWidth` ❽. Соответствующие тесты проходят, как показано на рис. 12.6.

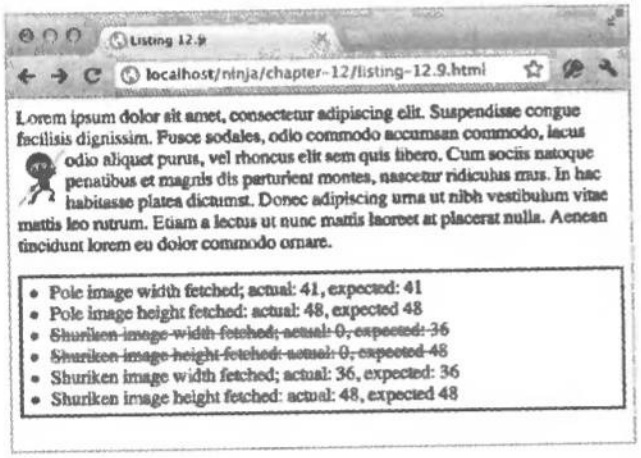

Рис. 12.6. Временно корректируя свойства стилевого оформления скрытых элементов разметки, можно успешно выбирать их размеры

Далее тот же самый тест выполняется для скрытого элемента разметки ❾. В этом тесте неверно утверждается, что в свойствах `offsetHeight` и `offsetWidth` будут доступны прежние значения. И нет ничего удивительного в том, что этот тест не проходит. Ведь мы уже убедились в этом ранее. После этого новая функция вызывается для скрытого изображения ❿, а результаты этой операции проверяются еще раз ⓫. На этот раз тест проходит, как показано на рис. 12.6. Наблюдая за отображением содержимого веб-страницы по ходу выполнения теста (напомним, что выполнение теста задерживается на 3 секунды после загрузки страницы и построения модели DOM), можно заметить, что отображение никоим образом не нарушается подспудными манипуляциями свойствами стилевого оформления скрытого элемента разметки.

Совет

Проверка свойств `offsetWidth` и `offsetHeight` стилевого оформления на обнуление может послужить невероятно эффективным средством для определения видимости элемента разметки.

Свойства стилевого оформления размеров элементов разметки — далеко не единственные подобные свойства, вызывающие серьезные трудности при разработке веб-приложений. Рассмотрим далее особенности обращения со свойством непрозрачности.

Манипулирование непрозрачностью

Непрозрачность — это особый случай, и поэтому свойство стилевого оформления `opacity` требует особого обращения в разных браузерах. Во всех современных браузерах, включая и Internet Explorer 9, свойство `opacity` поддерживается как собственное, тогда как в предыдущих версиях браузера Internet Explorer применяется собственный фильтр альфа-канала (`alpha`), определяющий степень непрозрачности изображения. В силу этого стили непрозрачности зачастую указываются в таблице стилей (или непосредственно в атрибуте `style`) следующим образом:

```
opacity: 0.5;
filter: alpha(opacity=50);
```

В стандартном стиле значения непрозрачности элемента разметки указываются в пределах от `0,0` до `1,0`, тогда как в фильтре альфа-канала — в процентах от `0` до `100`. Так, в приведенных выше правилах стилевого оформления указывается степень непрозрачности `50%`.

Допустим, имеется элемент разметки, непрозрачность которого определяется обоими упомянутыми выше способами, как показано ниже.

```
<div style="opacity:0.5;filter:alpha(opacity=50);">Hello</div>
```

При попытке выбрать оба значения непрозрачности, возникает затруднение двоякого характера.

- Помимо фильтра `alpha`, имеются и другие типы фильтров, например преобразований. В связи с этим приходится иметь дело со многими типами фильтров, не полагаясь на то, что непрозрачность всегда будет обозначаться только с помощью фильтра `alpha`.
- Несмотря на то что в версиях браузера Internet Explorer, предшествовавших версии 9, свойство `opacity` не поддерживается, указанное в нем значение будет возвращаться при обращении к этому свойству по ссылке `style.opacity`, даже если оно полностью игнорируется браузером.

Последнее обстоятельство затрудняет возможность определить в прикладном коде, имеется ли в браузере собственная поддержка свойства `opacity`. Но и это затруднение по плечу преодолеть настоящим мастерам, если внимательно разобраться в причинах, по которым браузеры упорно пытаются сбить нас с толку в отношении непрозрачности.

Оказывается, что те браузеры, в которых поддерживается свойство `opacity`, всегда нормализуют значение непрозрачности до величины меньше `1,0` и с начальным нулем на позиции первой десятичной цифры. Так, если в свойстве `opacity` задано значение `,5`, браузер с собственной поддержкой данного свойства возвратит значение `0,5`, тогда как браузер без такой поддержки просто оставит значение в его первоначальном виде, т.е. `,5`. Следовательно, для того чтобы выяснить, имеется ли в браузере собственная поддержка свойства `opacity`, можно воспользоваться имитацией компонентов, рассматривавшейся в главе 11. В качестве примера рассмотрим код, приведенный в листинге 12.10.

Листинг 12.10. Выяснение факта поддержки в браузере свойства непрозрачности

```
<img src="../images/ninja-with-nunchuks.png" style="opacity:.5;">

<script type="text/javascript">
```

```
var div = document.createElement("div");
div.setAttribute('style','opacity:.5');
var OPACITY_SUPPORTED = div.style.opacity === "0.5";

assert(OPACITY_SUPPORTED,
       "Opacity is supported.");

</script>
```

❶ Проверить, поддерживается ли свойство непрозрачности

❷ Отобразить результаты

В данном примере кода сначала определяется элемент разметки изображения со значением `,5` свойства `opacity`. Этот элемент предназначен не для применения в прикладном коде, а лишь для наглядного представления факта учета или игнорирования свойства непрозрачности в браузере.

Затем следует тест, где создается неприсоединяемый элемент разметки ❶, который расширяется атрибутом стилевого оформления со значением `,5` свойства `opacity`. Далее это значение считывается обратно и проверяется, сохранилось ли оно в исходном виде или же было изменено на значение `0,5`. В первом случае свойство непрозрачности не поддерживается в браузере, а во втором — поддерживается. И наконец, составляется утверждение поддержки с помощью соответствующей переменной. Таким образом, тест проходит в тех браузерах, где имеется собственная поддержка свойства непрозрачности, и не проходит там, где она отсутствует. На рис. 12.7 приведен результат загрузки данного теста ❷ в браузер Chrome 17 (*вверху*) и браузер Internet Explorer 7 (*внизу*).

Воспользовавшись приобретенными знаниями и навыками, попробуйте создать функцию `getOpacity(element)` по типу функции `getDimensions()` из листинга 12.9. Эта функция должна возвращать значение непрозрачности передаваемого ей элемента разметки в пределах от `0,0` до `1,0` независимо от используемой платформы.

Совет

При создании подобной функции вам может пригодиться регулярное выражение для выявления конкретного значения фильтра непрозрачности альфа-канала, а для приведения этого значения к нужной форме — метод `window.parseFloat()`. Если же такое значение не удастся выявить, то в качестве резервного следует возвратить значение `1,0`, поскольку оно используется по умолчанию для обозначения непрозрачности.

А теперь обратимся к другим проблематичным свойствам стилевого оформления. Они досаждают тем, что их значения могут принимать множество равнозначных форм.

Хождение по цветовому кругу

Как было показано ранее в этой главе, значения цвета могут быть представлены в самых разных форматах. Это обстоятельство несколько затрудняет обработку значений цвета, извлекаемых из свойства `style`. В какой-то степени разработчики веб-приложений становятся заложниками тех форматов цвета, которые выбираются авторами веб-страниц, а следовательно, и тех преобразований, которые выполняются над этими форматами в браузерах.

При доступе к свойствам цвета с помощью различных методов получения вычисленной информации о стилевом оформлении возникают некоторые расхождения в тех

форматах, в которых значения цвета возвращаются из различных браузеров. Вследствие этого любые попытки получить доступ к полезным составляющим цвета (скорее всего, это будет информация о каналах красного, зеленого и синего основных цветов) требуют дополнительных усилий. Существуют различные форматы, в которых цвета могут быть представлены в современных браузерах. Все эти форматы сведены в табл. 12.3.

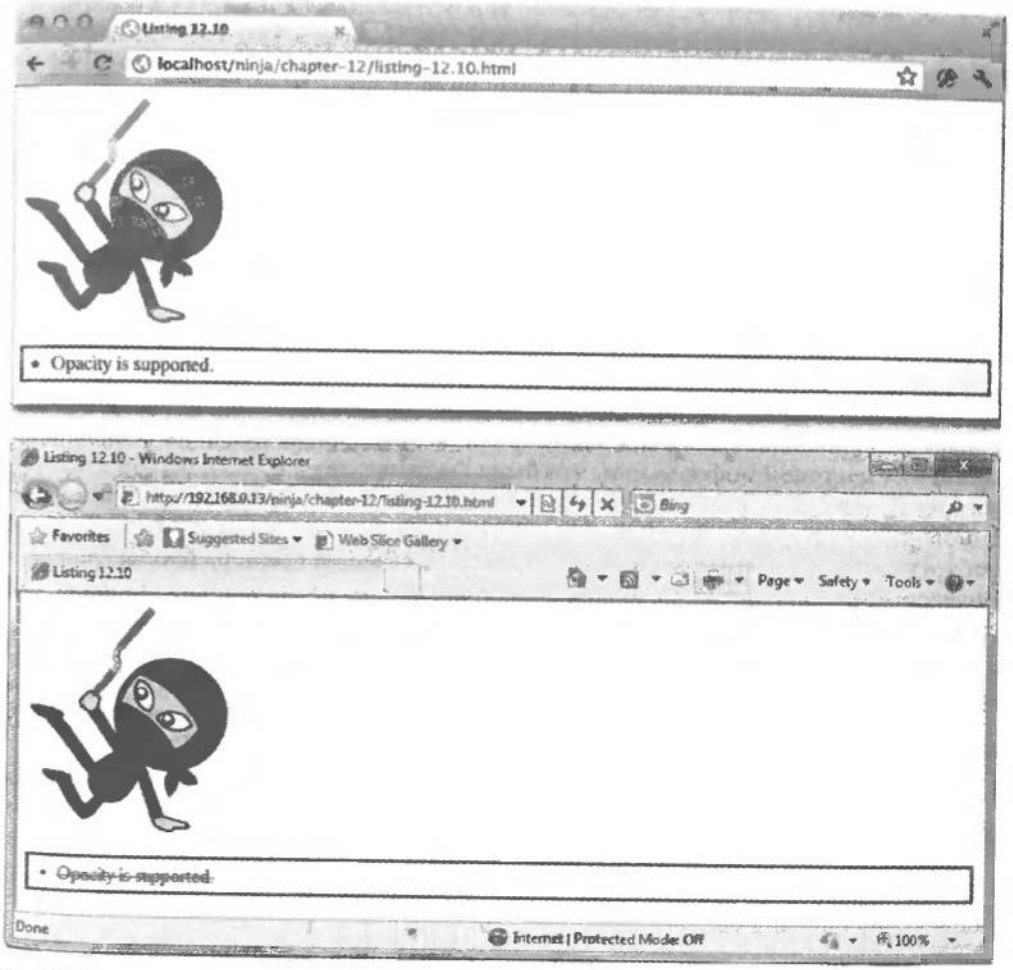

Рис. 12.7. Визуальные ориентиры и результаты тестирования наглядно показывают, что свойство непрозрачности поддерживается в браузере Chrome, но не поддерживается в версиях браузера Internet Explorer, предшествовавших версии 9

Таблица. 12.3. Форматы представления значений цвета по стандарту CSS3

Формат	Описание
`keyword`	Любое распознаваемое в HTML ключевое слово, обозначающее цвет (`red`, `green`, `maroon` и т.д.); расширенные в SVG ключевые слова для цвета (`bisque`, `chocolate`, `darkred` и т.д.) или ключевое слово `transparent`, что равнозначно формату `rgba(0,0,0,0)`, — см. ниже

Окончание табл. 12.3

Формат	Описание
#rgb	Краткая шестнадцатеричная форма представления значений основных цветов RGB (красного, зеленого, синего), где каждая составляющая цвета представлена шестнадцатеричным значением от 0 до f
#rrggbb	Длинная шестнадцатеричная форма представления значений основных цветов RGB (красного, зеленого, синего), где каждая составляющая цвета представлена шестнадцатеричным значением от 00 до ff
rgb(r,g,b)	Форма RGB представления цвета, где каждая составляющая цвета представлена десятичным значением от 0 до 255 или от 0% до 100%
rgba(r,g,b,a)	Форма RGB представления цвета с альфа-каналом, где величины непрозрачности в альфа-канале находятся в пределах от 0,0 (полная прозрачность) до 1,0 (полная непрозрачность)
hsl(h,s,l)	Форма HSL представления цвета в виде значений оттенка, насыщенности и яркости. Значения оттенка находятся в пределах от 0 до 360 (угол поворота на цветовом круге), а значения насыщенности и яркости — в пределах от 0% до 100%
hsla(h,s,l)	Форма HSL представления цвета с альфа-каналом

Как следует из табл. 12.3, автору веб-страницы предоставляется немало удобств для выражения цветовой информации, что было бы замечательно, если бы эта информация преобразовывалась браузерами в значения цвета, размещаемые в свойстве style в единообразном формате. К сожалению, на практике этого не происходит, что и вызывает трудности обращения со свойствами цвета. В качестве примера рассмотрим тест, приведенный в листинге 12.11, чтобы выяснить, какие неприятности нам готовят браузеры в отношении цвета.

Листинг 12.11. Выяснение способа форматирования цветовой информации в браузере

```
<div style="background-color:darkslateblue"> </div>
<div style="background-color:#369"> </div>
<div style="background-color:#123456"> </div>
div style="background-color:rg6b(44,88,168)"> </div>
<div style="background-color:rgba(44,88,166,0.5)"> </div>
<div style="background-color:hsl(120,100%,25%)"> </div>
<div style="background-color:hsla(120,100%,25%,0.5)"> </div>

<script type="text/javascript">

  var divs = document.getElementsByTagName('div');

  for (var n = 0; n < divs.length; n++) {
    assert(true,divs[n].style.backgroundColor);
  }

</script>
```

❶ Создать элементы разметки с атрибутами цвета

❷ Собрать элементы разметки

❸ Отобразить цветовую информацию

В приведенном выше примере кода сначала создается ряд элементов разметки со свойствами стилевого оформления цвета фона, выражаемого в семи различных фор-

матах ❶. Затем собираются ссылки на эти элементы разметки ❷ и организуется циклическое обращение к получаемой в итоге коллекции, в ходе которого отображается значение, хранящееся в свойстве style.backgroundColor каждого элемента разметки ❸. Этим наглядно показывается, каким образом браузер, в котором выполняется рассматриваемый здесь тест, форматирует цветовую информацию для различных методов, с помощью которых она указывается. Как показано на рис. 12.8, форматы, в которых представлена цветовая информация, интерпретируются в браузерах по-разному.

В силу столь разительных отличий в интерпретации цветовой информации в браузерах мы не будем занимать здесь место для разработки функции getColor(element, property), предоставив вам сделать это самостоятельно. Ведь теперь в вашем распоряжении имеются все необходимые для этого инструментальные средства. К тому же эта задача не столько трудная, сколько длительная. В качестве аргументов element и property данная функция должна принимать элемент разметки и свойство цвета (например, color или backgroundcolor), а возвращать — ключевое слово цвета, информационный массив, содержащий свойства red, green, blue и alpha, либо информационный массив, содержащий свойства hue, lightness, saturation и alpha. Опираясь на свои знания и навыки составления регулярных выражений, приобретенные в главе 7, а также на примеры функций getDimensions() и getOpacity(), разработанных ранее в этой главе, вы должны быть в состоянии справиться с такой задачей.

Усложнение задачи

Если вас не пугают трудности, вы можете также организовать преобразование значений цвета из формата HSL в формат RGB, используя формулу, представленную в статье Википедии по адресу http://en.wikipedia.org/wiki/HSL_and_HSV#Converting_to_RGB.[1]

Очевидно, что обработка значений цвета — задача не новая, поскольку ее приходилось решать и прежде. С одним из ее решений можно ознакомиться, проанализировав исходный код подключаемого модуля jQuery Color, написанного Блэром Митчелмором (Blair Mitchelmore; http://plugins.jquery.com/project/color).

До сих пор рассматривались в основном затруднения, которые приходится преодолевать, когда дело доходит до обращения к свойству style элемента разметки. Но, как отмечалось ранее, это свойство не содержит никакой информации о стилевом оформлении, наследуемом из тех таблиц стилей, которые находятся в области действия данного элемента разметки. А ведь зачастую оказывается очень удобно знать полностью *вычисленный стиль*, примененный к элементу разметки. Поэтому попробуем далее выяснить, можно ли каким-то образом получить этот стиль.

Извлечение вычисленных стилей

В любой момент времени *вычисленный стиль* элемента разметки представляет собой определенное сочетание стилей, применяемых к данному элементу с помощью таблиц стилей, его атрибута style и любых манипуляций со свойством style в сценарии. Стандартным прикладным интерфейсом API, определенным в спецификации W3C и реализованным во всех браузерах, включая и Internet Explorer 9, служит window метод .getComputedStyle(). Этот метод принимает элемент разметки, стили которого

[1] Эта статья доступна на русском языке после выбора ссылки Русский слева на веб-странице, открывшейся по указанному адресу. — *Примеч. ред.*

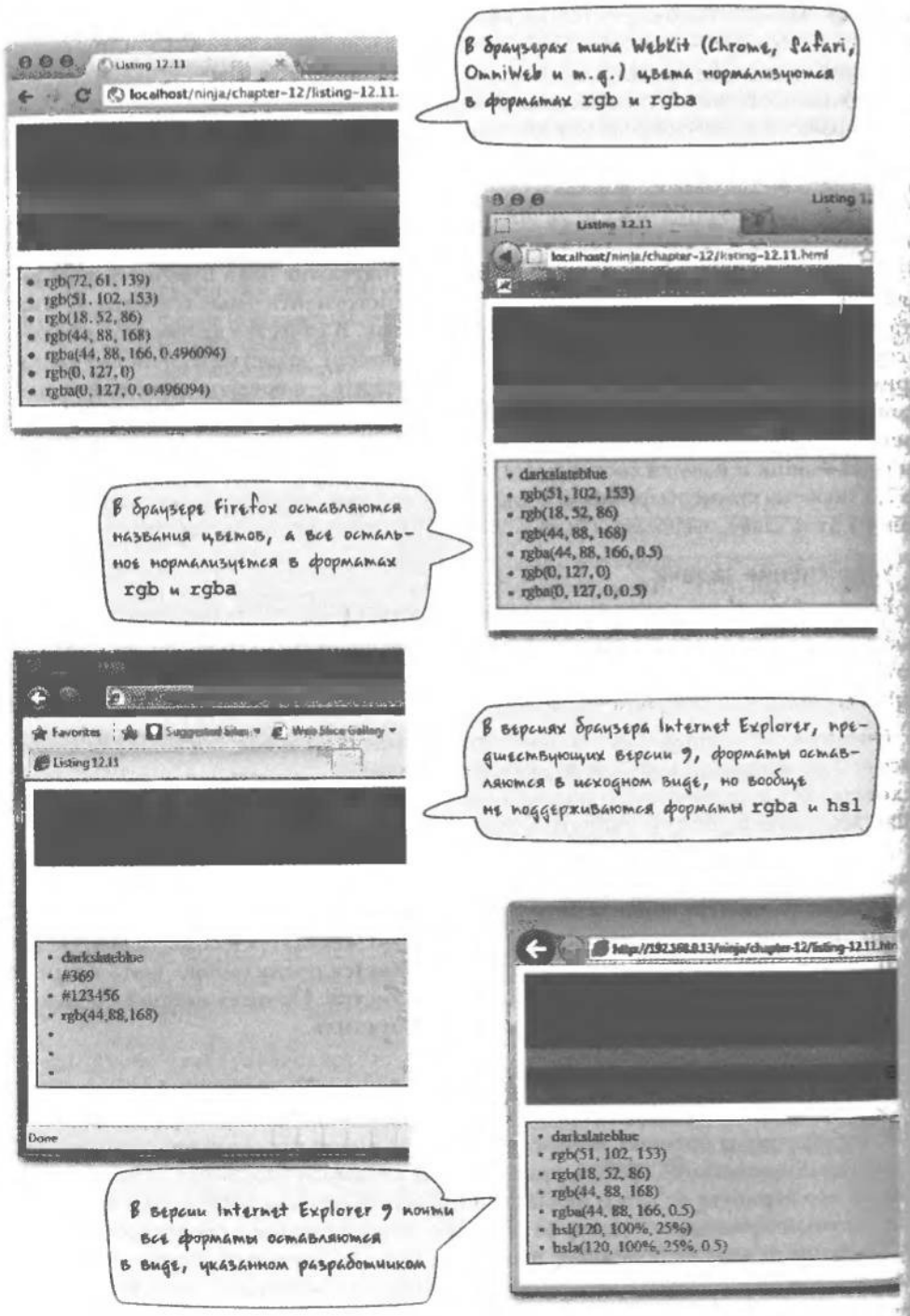

Рис. 12.8. В разных браузерах форматы представления цветовой информации интерпретируются по-разному!

должны быть вычислены, и возвращает интерфейс, посредством которого можно делать запросы к свойствам данного элемента. Так, для выборки вычисленного стиля конкретного свойства стилевого оформления в возвращаемом интерфейсе предоставляется метод `getPropertyValue()`. В отличие от свойств объекта `style` элемента разметки, метод `getPropertyValue()` принимает имена свойств стилевого оформления в формате CSS (например, `font-size` и `background-color`), а не в смешанном написании.

В версиях браузера Internet Explorer, предшествующих версии 9, применяется оригинальный способ доступа к вычисленному стилю элемента разметки: свойство `currentStyle` присоединяется ко всем элементам разметки и ведет себя почти так же, как и свойство `style`, за исключением того, что оно предоставляет актуальную, вычисленную информацию о стилевом оформлении. Этой информации оказывается достаточно для написания функции `fetchComputedStyle()`, которая должна получать вычисленное значение любого свойства стилевого оформления элемента разметки.

Вновь создаваемую функцию можно было бы назвать и `getComputedProperty()`. В листинге 12.12 приведен код, реализующий функцию `fetchComputedStyle()`. В ней применяются стандартные средства для обработки вычисленных стилей, если они доступны, а иначе — оригинальный метод в качестве запасного варианта.

Листинг 12.12. Извлечение вычисленных стилей

```
<style type="text/css">                                   ❶ Определить таблицу стилей
  div {
    background-color: #ffc; display: inline; font-size: 1.8em;
    border: 1px solid crimson; color: green;
  }
</style>                                                  Создать объект для тестирования ❷

<div style="color:crimson;" id="testSubject" title="Ninja power!">
  " "忍者パワー
</div>

<script type="text/javascript">
                                                          ❸ Определить новую функцию
  function fetchComputedStyle(element,property) {

    if (window.getComputedStyle) {
                                                          ❹ Получить интерфейс
      var computedStyles = window.getComputedStyle(element);

      if (computedStyles) {                               ❺ Извлечь значение стиля
        property = property.replace(/([A-Z])/g,'-$1').toLowerCase();
        return computedStyles.getPropertyValue(property);
      }
    }
                                                          ❻ Использовать оригинальные средства
    else if (element.currentStyle) {
      property = property.replace(
        /-([a-z])/ig,
        function(all,letter){ return letter.toUpperCase(); });
      return element.currentStyle[property];
```

```
    }
  }

  window.onload = function(){

    var div = document.getElementsByTagName("div")[0];

    assert(true,
          "background-color: " +
          fetchComputedStyle(div,'background-color'));
    assert(true,
          "display: " +
          fetchComputedStyle(div,'display'));
    assert(true,
          "font-size: " +
          fetchComputedStyle(div,'fontSize'));
    assert(true,
          "color: " +
          fetchComputedStyle(div,'color'));
     assert(true,
           "border-top-color: " +
           fetchComputedStyle(div,'borderTopColor'));
    assert(true,
          "border-top-width: " +
          fetchComputedStyle(div,'border-top-width'));
  };

</script>
```

❼ Отобразить результаты

Для того чтобы проверить вновь созданную функцию, в приведенном выше примере кода устанавливается элемент разметки, в котором указывается информация о его стилевом оформлении ❷, а также внешняя таблица стилей, предоставляющая правила стилевого оформления, которые должны применяться к данному элементу разметки ❶. При этом предполагается, что вычисленные стили должны стать результатом как непосредственного, так и внешнего стилевого оформления данного элемента разметки.

Затем определяется новая функция, принимающая в качестве своих аргументов элемент разметки и свойство стилевого оформления, для которого требуется найти вычисленное значение ❸. Ради особого удобства многословные имена свойств могут указываться в одном из двух форматов: через дефис или в смешанном написании. Иными словами, данная функция должна принимать свойство стилевого оформления как под именем `backgroundColor`, так и под именем `background-color`. Ниже поясняется, как этого добиться.

Прежде всего требуется проверить, доступны ли стандартные средства, а они обычно доступны во всех браузерах, кроме прежних версий браузера Internet Explorer. Если стандартные средства доступны, то далее получается интерфейс для обработки вычисленного стиля. Этот интерфейс сохраняется в переменной для последующего обращения к нему ❹. И это требуется сделать потому, что заранее неизвестно, насколько затратной может оказаться подобная операция, а следовательно, было бы желательно избегать ее повторения без особой надобности.

Если интерфейс будет получен успешно, а особых оснований для иного исхода не существует, хотя и бдительность терять нельзя, то из данного интерфейса вызывается метод `getPropertyValue()` с целью получить значение вычисленного стиля ❺. Но прежде необходимо привести имя свойства стилевого оформления к написанию через дефис или к смешанному написанию. Так, в методе `getPropertyValue()` предполагается получить имя в написании через дефис, и поэтому для вставки дефиса перед каждой прописной буквой в имени свойства и преобразования всего имени в строчные буквы вызывается метод `replace()` из класса `String` с простым, но искусным регулярным выражением.

Если же обнаружится, что стандартный метод недоступен, то проверяется, имеется ли оригинальное для браузера Internet Explorer свойство `currentStyle`. И если оно имеется, то в имени этого свойства все строчные буквы, которым предшествует дефис, заменяются прописными, а сам дефис удаляется, чтобы привести написание имени через дефис к смешанному написанию, после чего возвращается значение данного свойства ❻. Но в любом случае, если произойдет что-нибудь непредвиденное, то никакого значения вообще не возвращается.

Для проверки вновь созданной функции делается целый ряд ее вызовов, в которых ей передаются различные имена стилей в разных форматах, а результаты тестирования выводятся на экран ❼, как показано на рис. 12.9. Следует иметь в виду, что стили извлекаются независимо от того, были ли они явным образом объявлены в элементе разметки или унаследованы из таблицы стилей. Кроме того, значение свойства `color`, указанного как в таблице стилей, так и непосредственно в элементе разметки, возвращается явно заданным. Стили, задаваемые в атрибуте `style` элемента разметки, всегда получают приоритет над наследуемыми стилями, даже если они и отмечены аннотацией `!important` в таблице стилей.

Обращаясь со свойствами стилевого оформления, следует также иметь в виду *сочетание* свойств. В таблицах CSS допускается краткая запись сочетания свойств, например, свойств обрамления. Вместо того чтобы указывать цвета, ширину и стили обрамления

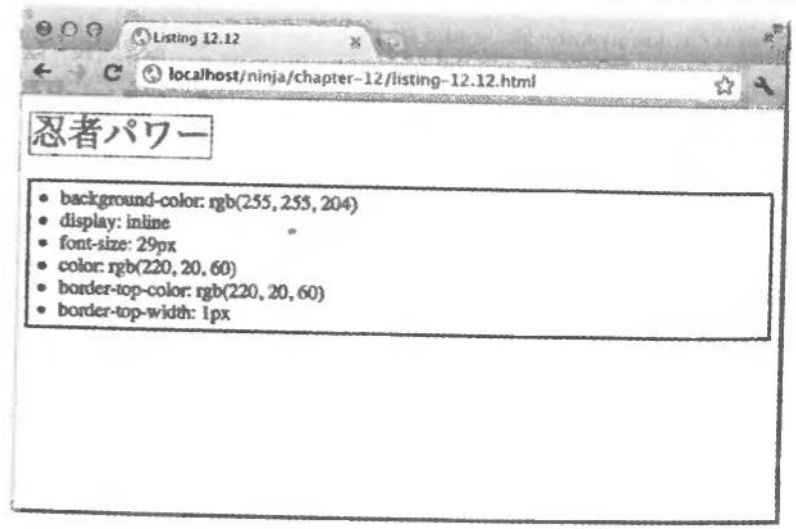

Рис. 12.9. Вычисленные стили включают в себя все стили, указанные в элементе разметки, а также те, что наследуются из таблиц стилей

по отдельности для всех четырех рамок, можно записать следующее правило их стилевого оформления:

```
border: 1px solid crimson;
```

Именно это правило и было использовано в листинге 12.12. Оно позволяет сэкономить на наборе стилей, но не следует забывать, что стили нужно извлекать из отдельных свойств более низкого уровня. В частности, нельзя извлечь стиль `border`, но можно извлечь стиль `border-top-color` и `border-top-width`, что и было сделано в приведенном выше примере. Такой способ может доставить немного хлопот, особенно если всем четырем стилям присвоены одинаковые значения. Впрочем, и это препятствие нетрудно обойти, как пояснялось ранее.

Резюме

Что касается кросс-браузерной совместимости, то получение и установка атрибутов, свойств и стилей в элементах модели DOM может оказаться не самой трудной частью разработки веб-приложений на JavaScript, хотя и здесь имеется немалая доля трудностей. Правда, эти трудности можно преодолеть с учетом кросс-браузерной совместимости и не прибегая к обнаружению браузеров. Ниже перечислены основные вопросы, рассмотренные в этой главе.

- Значения атрибутов устанавливаются исходя из тех атрибутов, которые находятся в элементе разметки.
- Извлекаемые значения атрибутов могут оказаться одинаковыми, но иногда они форматируются иначе, чем указано в исходной разметке.
- Свойства, представляющие значения атрибутов, создаются в элементах разметки.
- Обозначения этих свойств могут отличаться как от исходных имен атрибутов, так и в браузерах, а их значения — от значений атрибутов или исходной разметки по своему формату.
- В крайнем случае нужное значение можно извлечь из исходной разметки, обратившись к исходным узлам атрибутов в модели DOM и получив его оттуда.
- Обращаться со свойствами обычно эффективнее, чем пользоваться методами модели DOM для манипулирования атрибутами.
- В версиях браузера Internet Explorer, предшествующих версии 9, не разрешается изменять атрибут `type` элементов разметки `<input>`, как только такой элемент станет частью модели DOM.
- Атрибут `style` ставит особо трудные препятствия и не содержит вычисленный стиль для элемента разметки.
- Вычисленные стили могут извлекаться из объекта `window` с помощью стандартного прикладного интерфейса API в большинстве современных браузеров или же через оригинальное свойство в версиях браузера Internet Explorer, предшествующих версии 9.

В этой главе были рассмотрены затруднения, возникающие в связи с отличиями в реализации средств обращения со свойствами и атрибутами в браузерах, что доставляет немало хлопот при разработке веб-приложений. Но, вероятно, ни одна область разработки не доставляет столько хлопот кросс-браузерного характера, как обработка событий. Именно этой теме и будет посвящена следующая глава.

Часть IV

Обучение мастера

Если вы дошли в своем обучении до этой стадии, то своим мастерством уже выделяетесь среди прочих программирующих на JavaScript. Но если вам требуется еще более доскональное обучение, то в этой части вы найдете немало других, глубоко скрытых секретов программирования на JavaScript. Материал глав этой части не рассчитан на малодушных и новичков, поскольку его усвоение требует определенной подготовки, а излагается он более углубленно и в ускоренном темпе по сравнению с предыдущими главами. Вам предстоит восполнить пробелы в своих знаниях и навыках, углубившись в неизведанные ранее области, хотя на этом пути вас ожидают серьезные препятствия.

Главы этой части написаны с точки зрения разработчиков распространенных библиотек JavaScript. Их материал дает некоторое представление о решениях и методиках, применяемых для реализации самых сложных частей этих библиотек.

В главе 13 основное внимание уделяется кросс-браузерной обработке событий, которая относится едва ли не к самым трудным препятствиям, которые ставят браузеры перед разработчиками веб-приложений. А в главе 14 рассматриваются приемы манипулирования моделью DOM.

И наконец, в главе 15 рассматриваются механизмы CSS-селекторов. Усвоение этой темы требует немалых знаний, даже если написание подобного механизма сызнова не входит в ваши планы просвещения.

Итак, приведите свой арсенал и снаряжение в боевую готовность. Ведь данная стадия обучения станет для вас настоящим испытанием.

13 Особенности обработки событий

В этой главе...

- Причины, по которым обработка событий вызывает трудности
- Способы привязки и отвязки событий
- Инициирование событий
- Применение специальных событий
- Всплывание и делегирование событий

Управление событиями в модели DOM *должно* быть относительно простым, но даже из того, что данной теме посвящена целая глава, нетрудно догадаться, что это на самом деле совсем не так. Несмотря на то что для управления событиями во всех браузерах предоставляются относительно устойчивые прикладные интерфейсы API, в них применяются совершенно разные подходы для реализации такого управления на практике. И помимо трудностей, обусловленных отличиями в браузерах, средств, предоставляемых самими браузерами, зачастую оказывается недостаточно для решения большинства задач обработки событий в сложных приложениях.

В силу этих недостатков в библиотеках JavaScript приходится едва ли не дублировать прикладные интерфейсы API, существующие в браузерах для обработки событий. В этой книге не предполагается, что вы собираетесь писать собственную библиотеку. Тем не менее вам будет очень полезно знать, каким образом обработка событий реализуется в любой библиотеке, которой вы собираетесь воспользоваться, а самое главное — проникнуть в секреты реализации обработки событий в библиотеках.

Все, кто дочитал книгу до этой главы, скорее всего, знакомы с типичным применением модели обработки событий по стандарту DOM Level 0, где обработчики событий устанавливаются через свойства или атрибуты элементов разметки документа. Так,

если в коде игнорируются принципы ненавязчивого JavaScript, установка обработчика событий в элементе разметки <body> может выглядеть следующим образом:

```
<body onload="doSomething()">
```

А если для реализации поведения (обработки событий) в коде не соблюдается структура разметки, то установка обработчика событий может выглядеть так, как показано ниже.

```
window.onload = doSomething;
```

В обоих случаях применяется модель обработки событий по стандарту DOM Level 0. Но на события в стандарте DOM Level 0 накладываются суровые ограничения, что делает их непригодными для повторно используемого кода или страниц любого уровня сложности. В модели обработки событий по стандарту DOM Level 2 предоставляется более надежный прикладной интерфейс API, но и его применение проблематично, поскольку он недоступен в версиях браузера Internet Explorer, предшествующих версии 9. И как уже упоминалось выше, ему недостает целого ряда очень нужных средств.

Мы не будем рассматривать здесь модель обработки событий по стандарту DOM Level 0 из-за того, что она практически бесполезна, а сосредоточим основное внимание на модели по стандарту DOM Level 2. (Для любопытствующих попутно заметим, что модель обработки событий не была внедрена по стандарту DOM Level 1.) В этой главе мы постараемся благополучно пройти минное поле обработки событий и поясним, как уцелеть в той враждебной обстановке, в которую нас нередко ставят браузеры.

Привязка и отвязка обработчиков событий

В модели обработки событий по стандарту DOM Level 2 привязка и отвязка обработчиков событий осуществляется с помощью стандартных методов `addEventListener()` и `removeEventListener()` во всех современных браузерах, совместимых с моделью DOM, а в устаревших версиях браузера Internet Explorer, предшествующих версии 9, — с помощью метода `detachEvent()`.

Ради простоты далее в этой главе модель обработки событий по стандарту DOM Level 2 будет просто называться *моделью DOM*, тогда как аналогичная модель для устаревших версий браузера Internet Explorer — *моделью IE*. Первая из них доступна во всех современных версиях браузеров из так называемой "Большой пятерки", а последняя — во всех версиях браузера Internet Explorer, предшествующих версии 9. Обе модели обработки событий по большей части действуют одинаково, за одним существенным исключением: в модели IE не предоставляются средства для приема или так называемого "прослушивания" событий на стадии их фиксации, но поддерживается только стадия всплывания событий в процессе их обработки.

Примечание

Для тех, кто незнаком с моделью обработки событий по стандарту DOM Level 2, вкратце поясним, что события распространяются от их инициатора вверх к корневому узлу модели DOM на *стадии всплывания*, а затем они перемещаются вниз по дереву к инициатору на *стадии фиксации*.

Кроме того, в реализации модели IE контекст в привязанном обработчике событий не устанавливается должным образом, в результате чего обращение по ссылке

`this` в обработчике событий происходит к глобальному контексту (объекту `window`) вместо целевого элемента. Более того, в модели IE данные о событиях не передаются их обработчику, но вносятся в глобальный контекст, т.е. в объект `window`.

Это означает, что для обработки событий приходится принимать меры, характерные для конкретного браузера. К их числу относятся приведенные ниже.

- Привязка обработчика событий.
- Отвязка обработчика событий.
- Получение данных о событии.
- Получение инициатора событий.

Обнаружение браузера для принятия тех или иных мер на каждой стадии обработки событий вряд ли сделает код надежным и повторно используемым. Поэтому попробуем выяснить, что можно предпринять для создания общего набора прикладных интерфейсов API, чтобы навести элементарный порядок в обработке событий. С этой целью рассмотрим сначала пути преодоления препятствий, которые возникают в связи с применением многих прикладных интерфейсов API и тем, что контекст не устанавливается в модели IE, как показано в коде из листинга 13.1.

Листинг 13.1. Предоставление надлежащего контекста во время привязки обработчиков событий

```
<script type="text/javascript">

  if (document.addEventListener) {          ❶ Проверить наличие модели DOM

    this.addEvent = function (elem, type, fn) {   ❷ Создать функцию привязки, используя модель DOM
      elem.addEventListener(type, fn, false);
      return fn;
    };

    this.removeEvent = function (elem, type, fn) {   ❸ Создать функцию отвязки, используя модель DOM
      elem.removeEventListener(type, fn, false);
    };

  }
  else if (document.attachEvent) {          ❹ Проверить наличие модели IE

    this.addEvent = function (elem, type, fn) {   ❺ Создать функцию привязки, используя модель IE
      var bound = function () {
        return fn.apply(elem, arguments);
      };
      elem.attachEvent("on" + type, bound);
      return bound;
    };

    this.removeEvent = function (elem, type, fn) {   ❻ Создать функцию отвязки, используя модель IE
      elem.detachEvent("on" + type, fn);
    };

  }

</script>
```

В приведенном выше примере кода в глобальный контекст вводятся два метода: `addEvent()` и `removeEvent()`. Их реализация приспосабливается к той среде, в которой выполняется сценарий. Если присутствует модель DOM, то используется именно она, а иначе – модель IE (разумеется, при ее наличии). Если же обе модели отсутствуют, то и методы не создаются. А реализуются они очень просто. После проверки наличия модели DOM ❶ определяются тонкие оболочки, в которые заключаются стандартные методы модели DOM: одна – для привязки обработчиков событий ❷, другая – для отвязки обработчиков событий ❸.

Следует иметь в виду, что функция привязки обработчиков событий возвращает установленный обработчик событий в качестве своего значения (о важности этого обстоятельства речь пойдет несколько ниже) и передает логическое значение `false` в качестве третьего аргумента методам обработки событий из прикладного интерфейса API модели DOM. Таким образом, они обозначаются как обработчики *всплывающих* событий. А поскольку они предназначены для кросс-браузерных сред, то в рассматриваемых здесь функциях стадия фиксации событий не поддерживается.

Если модель DOM отсутствует, то проверяется наличие модели IE ❹. И если она присутствует, то с ее помощью определяются две функции привязки и отвязки обработчиков событий. Функция отвязки определяется очень просто в качестве тонкой оболочки для стандартного метода модели IE ❻. Совсем иначе определяется функция привязки ❺.

Напомним, что, помимо построения единообразного интерфейса API, одной из основных причин определения рассматриваемых здесь функций было устранение недостатка, связанного с тем, что в контексте обработчика событий не устанавливается их инициатор. Поэтому в функции привязки вместо простой передачи обработчика событий (в виде параметра `fn`) методу модели последний заключается сначала в оболочку анонимной функции, которая, в свою очередь, вызывает обработчик событий, но использует для этой цели метод `apply()`, чтобы принудительно установить целевой элемент события в качестве контекста. Затем данная обертывающая функция передается методу модели в качестве обработчика событий. Таким образом, при наступлении события заключенная в оболочку функция его обработки будет вызываться с надлежащим контекстом. Как и все остальные функции, данная функция возвращает обработчик событий в качестве своего значения, хотя на этот раз возвращается оболочка, а не функция, передаваемая в виде параметра `fn`.

Возврат функции важен потому, что для последующей отвязки обработчика событий необходимо передать ссылку на функцию, установленную в качестве обработчика событий, соответствующего методу модели. В данном случае это обертывающая функция, хранящаяся в переменной `bound`. Рассмотрим весь этот механизм в действии на кратком примере теста, приведенном в листинге 13.2. Этот тест требует вмешательства со стороны пользователя, и поэтому в нем отсутствуют утверждения. Вместо этого организуется взаимодействие со страницей и наблюдаются результаты.

Листинг 13.2. Проверка прикладного интерфейса API для привязки обработчиков событий

```
addEvent(window, "load", function () {      ❶ Установить обработчик событий загрузки

  var elems = document.getElementsByTagName("div");   ❷ Извлечь элементы для тестирования

  for (var i = 0; i < elems.length; i++) (function (elem) {

    var handler = addEvent(elem, "click", function () {   ❸ Установить обработчики событий для тестирования
      this.style.backgroundColor =
        this.style.backgroundColor=='' ? 'green' : '';
      removeEvent(elem, "click", handler);   ❹ Отвязать обработчики событий
    });
  })(elems[i]);
});
```


Прежде чем выполнять рассматриваемый здесь тест, необходимо подождать до тех пор, пока не загрузится модель DOM, чтобы выбрать подходящий для тестирования прикладной интерфейс API и установить остальную часть теста в качестве обработчика событий загрузки ❶. Если функция привязки не работает, тест вообще не пройдет. В обработчике событий загрузки извлекаются ссылки на все элементы разметки `<div>`, находящиеся на странице, чтобы служить в качеств объектов для тестирования ❷. А далее организуется циклическое обращение к получающейся в итоге коллекции элементов разметки.

Для каждого целевого элемента вызывается функция `addEvent()`, устанавливающая для него обработчик событий и сохраняющая возвращаемую ссылку на функцию в переменной `handler` ❸. Это делается с целью установить ссылку в замыкании для обработчика событий, чтобы в дальнейшем функция обработки событий ссылалась на самое себя. Следует заметить, что в данном случае нельзя полагаться на свойство `callee`, поскольку заранее известно, что при обработке событий с помощью модели IE возвращаемая функция будет отличаться от передаваемой.

В обработчике событий от щелчков кнопкой мыши ссылка на целевой элемент делается через параметр `this`, подтверждающий, что контекст установлен правильно, а также определяется, был ли задан цвет фона этого элемента. Если цвет фона не задан, то устанавливается зеленый цвет, в противном случае он не устанавливается. И если все это оставить так, как есть, то после каждого последующего щелчка кнопкой мыши на данном элементе разметки цвет фона будет либо отсутствовать, либо становиться зеленым. Но этого нельзя так оставить. Поэтому еще до завершения обработчика событий функция `removeEvent()` и переменная `handler`, привязанная к ней через замыкание, используются для удаления этого обработчика событий ❹. Таким образом, обработчик событий, будучи однажды запущен, больше не запускается на выполнение.

Если добавить приведенные ниже элементы в разметку веб-страницы, исключив применение в них цвета фона из таблиц стилей, то можно ожидать, что после щелчка на каждом элементе разметки `<div>` он будет выделен зеленым цветом, а в результате последующих щелчков цвет фона не изменится.

```
<div title="Click me"> 私をクリック </div>
<div title="but only once"> 一度だけ </div>
```

Если загрузить веб-страницу в окно браузера и выполнить рассматриваемый здесь ручной тест, то окажется, что проверяемые функции работают должным образом. На рис. 13.1 показано состояние веб-страницы после ее загрузки в окно браузера Chrome и после щелчка на первом элементе ее разметки несколько раз подряд, вообще не затрагивая второй элемент разметки. А на рис. 13.2 показано состояние той же самой веб-страницы после ее загрузки и выполнения аналогичных действий в окне браузера Internet Explorer 8, где модель DOM не поддерживается.

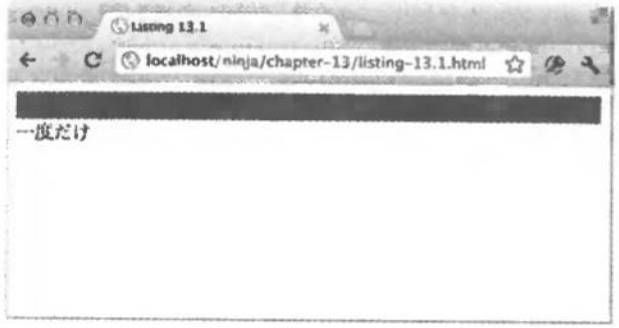

Рис. 13.1. Как показывает этот ручной тест, единообразный прикладной интерфейс API позволяет привязывать и отвязывать события

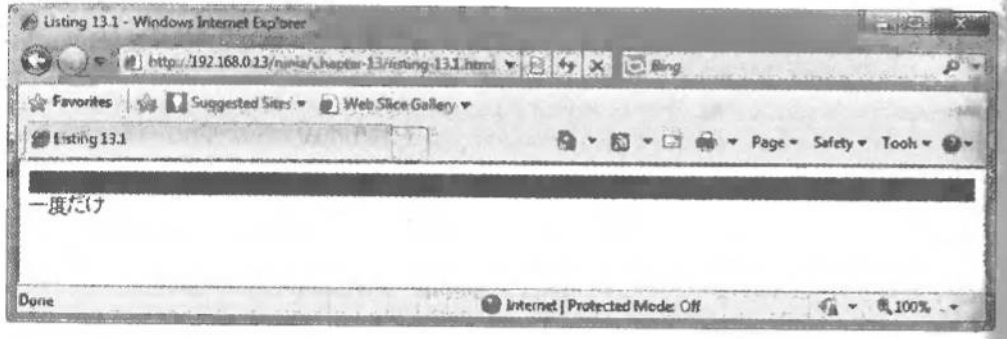

Рис. 13.2. Привязка и отвязка событий действует и в прежней версии браузера Internet Explorer

Итак, начало положено совсем не плохое. Но главный недостаток построенного прикладного интерфейса API состоит в том, что его пользователи должны аккуратно регистрировать ссылку на обработчик событий, когда она возвращается из функции `addEvent()`, поскольку для устаревших версий браузера Internet Explorer обработчик приходится заключать в оболочку. В противном случае обработчик событий нельзя будет привязать в дальнейшем. Еще один недостаток данного интерфейса заключается в том, что в нем не разрешается затруднение, связанное с доступом к данным о событии. Таким образом, желаемая цель пока еще не достигнута, несмотря на произведенные усовершенствования. Попробуем выяснить, можно ли добиться большего.

Объект типа `Event`

Как отмечалось ранее, модель IE, которой приходится пользоваться для обработки событий в устаревших браузерах, имеет целый ряд отличий от модели DOM. К их числу относится порядок доступа к экземпляру объекта типа `Event` из обработчиков событий. В модели DOM этот объект передается обработчику событий в качестве первого аргумента, тогда как в модели IE он извлекается из свойства `event`, размещаемого в глобальном контексте (`window.event`).

Хуже того, содержимое экземпляра объекта типа `Event` отличается в обеих моделях. Единственный приемлемый выход из этого затруднительного положения состоит

в создании *нового* объекта, имитирующего собственный объект события в браузере, с последующей нормализацией его свойств в соответствии с моделью DOM. А нельзя ли просто видоизменить существующий объект? К сожалению, сделать это невозможно, поскольку в данном объекте имеется немало свойств, которые нельзя перезаписывать.

Еще одно преимущество клонирования объекта события состоит в том, что оно разрешает затруднение, обусловленное тем, что по модели IE объект сохраняется в глобальном контексте. И как только наступит новое событие, объект любого другого предыдущего события удаляется. Перенос свойств события в новый объект, срок действия которого можно контролировать, разрешает любые потенциальные затруднения подобного характера. В качестве примера попробуем создать функцию для нормализации событий, как показано в листинге 13.3.

Листинг 13.3. Функция, нормализующая экземпляр объекта события

```
<script type="text/javascript">

  function fixEvent(event) {

    function returnTrue() { return true; }
    function returnFalse() { return false; }

    if (!event || !event.stopPropagation) {
      var old = event || window.event;

      // клонировать старый объект, чтобы видоизменить значения его свойств
      event = {};
      for (var prop in old) {
        event[prop] = old[prop];
      }

      // В этом элементе произошло событие
      if (!event.target) {
        event.target = event.srcElement || document;
      }

      // выяснить, с каким еще элементом связано данное событие
      event.relatedTarget = event.fromElement === event.target ?
          event.toElement :
          event.fromElement;

      // остановить действие исходного браузера
      event.preventDefault = function () {
        event.returnValue = false;
        event.isDefaultPrevented = returnTrue;
      };

      event.isDefaultPrevented = returnFalse;

      // остановить всплывание события
      event.stopPropagation = function () {
        event.cancelBubble = true;
```

❶ Предварительно определить часто используемые функции

❷ Проверить, требуется ли нормализация

❸ Клонировать существующие свойства

```
      event.isPropagationStopped = returnTrue;
    };

    event.isPropagationStopped = returnFalse;

    // остановить всплывание события и выполнение других обработчиков
    event.stopImmediatePropagation = function () {
      this.isImmediatePropagationStopped = returnTrue;
      this.stopPropagation();
    };

    event.isImmediatePropagationStopped = returnFalse;

    // определить положение курсора мыши
    if (event.clientX != null) {
      var doc = document.documentElement, body = document.body;

      event.pageX = event.clientX +
        (doc && doc.scrollLeft || body && body.scrollLeft || 0) -
        (doc && doc.clientLeft || body && body.clientLeft || 0);
      event.pageY = event.clientY +
        (doc && doc.scrollTop || body && body.scrollTop || 0) -
        (doc && doc.clientTop || body && body.clientTop || 0);
    }

    // обработать нажатия клавиш
    event.which = event.charCode || event.keyCode;

    // выявить кнопку мыши, на которой был произведен щелчок:
    // 0 == левая; 1 == средняя; 2 == правая
    if (event.button != null) {
      event.button = (event.button & 1 ? 0 :
          (event.button & 4 ? 1 :
          (event.button & 2 ? 2 : 0)));
    }
  }
  return event;      ❹ Возвратить нормализованный экземпляр объекта
}

</script>
```

Несмотря на то что листинг приведенного выше примера кода получился довольно длинным, большая часть выполняемых в нем действий довольно проста, поэтому мы не будем анализировать его построчно, но укажем лишь на самые важные моменты. По существу, назначение функции, создаваемой в данном примере кода, состоит в том, чтобы взять экземпляр объекта типа Event и проверить, соответствует ли он модели DOM. И если он не соответствует данной модели, то делается все возможное, чтобы добиться подобного соответствия. Подробнее с определением объекта типа Event в модели DOM можно ознакомиться на веб-сайте консорциума W3C по адресу http://www.w3.org/TR/DOM-Level-2-Events/events.html#Events-interface.

Прежде всего в рассматриваемой здесь функции определяются две другие функции ❶. Напомним, что это допускается делать в JavaScript, причем область действия таких функций ограничивается их родительской функцией, а это избавляет от необходимости заботиться о засорении глобального пространства имен. Эти функции должны часто возвращать логическое значение true или false соответственно по ходу выполнения кода нормализации, поскольку пользоваться ими намного удобнее, чем избыточными функциональными литералами.

Затем проверяется, требуется ли сделать что-нибудь для нормализации ❷. Если экземпляр объекта не существует (в данном случае предполагается, что событие определено в глобальном контексте) или же если он все-таки существует, но стандартное свойство stopPropagation отсутствует, то предполагается, что нормализация нужна. В таком случае выбирается копия существующего события (та, что была передана, или же та, что находится в глобальном контексте) и сохраняется в переменной old. В противном случае происходит переход в конец данной функции и возврат из нее существующего события ❹.

Если требуется нормализация, то сначала создается новый пустой объект, который будет служить для представления нормализованного события, а затем в него копируются все существующие свойства из старого события ❸. После это выполняется сама нормализация с целью устранить любые расхождения между объектом типа Event, определяемом в модели DOM по стандарту консорциума W3C, и объектом, предоставляемым в модели IE. Ниже перечислены некоторые важные свойства объекта из модели DOM, которые нормализуются в ходе рассматриваемого здесь процесса.

- target. Обозначает первоначальный источник события. В модели IE сведения о нем хранятся в свойстве srcElement.
- relatedTarget. Применяется для события, связанного с другим элементом разметки (например, события mouseover или mouseout). В модели IE ему эквивалентны свойства toElement и fromElement.
- preventDefault. Препятствует действию исходного браузера. Для достижения аналогичного результата в модели IE необходимо установить логическое значение false в свойстве returnValue.
- stopPropagation. Останавливает всплывание события далее вверх по дереву. Для достижения аналогичного результата в модели IE необходимо установить логическое значение true в свойстве cancelBubble.
- pageX и pageY. Предоставляют данные о положении курсора мыши относительно всего документа. Они отсутствуют в модели IE, но могут быть легко сдублированы с помощью других данных. Так, свойства clientX и clientY предоставляют сведения о положении курсора мыши относительно окна, свойства scrollTop и scrollLeft — данные о состоянии прокрутки относительно документа, а свойства clientTop и clientLeft — данные о смещении самого документа. Из сочетания этих трех свойств получаются окончательные значения свойств pageX и pageY.
- which. Эквивалентно коду нажатой клавиши при наступлении события от клавиатуры. Оно может быть сдублировано путем доступа к свойствам charCode и keyCode в модели IE.
- button. Обозначает кнопку мыши, которой пользователь щелкнул при наступлении события от мыши. В модели IE для этой цели используется битовая

маска, где значение **1** обозначает щелчок левой кнопкой мыши, значение **2** — щелчок правой кнопкой мыши, значение **4** — щелчок средней кнопкой мыши. Поэтому эти значения необходимо преобразовать в значения **0**, **1** и **2** соответственно в модели DOM.

Еще одним источником полезных сведений об объекте типа `Event` в модели DOM и его кросс-браузерных возможностях служат таблицы совместимости режимов, которые приводятся на веб-сайте QuirksMode.

- Совместимость объекта типа `Event`: `http://www.quirksmode.org/dom/w3c_events.html`.
- Совместимость положения курсора мыши: `http://www.quirksmode.org/dom/w3c_cssom.html#mousepos`.

Кроме того, свойства объектов событий от клавиатуры и мыши обсуждаются в отличном документе "Сумасшествие JavaScript" (JavaScript Madness).

- События от клавиатуры: `http://unixpapa.com/js/key.html`.
- События от мыши: `http://unixpapa.com/js/mouse.html`.

Итак, в нашем распоряжении теперь имеются средства для нормализации экземпляра объекта типа `Event`. Попробуем далее выяснить, что можно сделать для достижения допустимого предела управления процессом привязки.

Управление обработкой событий

По целому ряду причин было бы целесообразно не привязывать обработчики событий непосредственно к элементам разметки. Если воспользоваться вместо этого промежуточным обработчиком событий, сохранив все обработчики в отдельном объекте, то тем самым можно повысить уровень управления процессом обработки событий. Среди прочего, это даст возможность выполнить следующее.

- Нормализовать контекст обработчиков событий.
- Упорядочить свойства объектов типа `Event`.
- Выполнить "сборку мусора" среди привязанных обработчиков событий.
- Запустить или удалить некоторые обработчики событий с помощью фильтра.
- Отвязать все события определенного типа.
- Клонировать обработчики событий.

Для того чтобы добиться всех этих преимуществ, потребуется доступ ко всему списку обработчиков событий, привязанных к элементу разметки. Поэтому целесообразно избегать непосредственной привязки событий и производить ее самостоятельно. Рассмотрим далее, как это делается.

Централизованное хранение связанной информации

Один из самых лучших способов управления обработчиками событий, связанными с элементом модели DOM, состоит в том, чтобы присвоить сначала каждому обрабатываемому элементу однозначный идентификатор, который не следует путать с идентификатором `id` в самой модели DOM, а затем сохранить все связанные с ним данные в одном, централизованном объекте. На первый взгляд, более естественным было

бы хранить данные в каждом элементе в отдельности. Тем не менее централизованное их хранение поможет избежать потенциальных утечек памяти в браузере Internet Explorer, которому присуща потеря памяти при определенных обстоятельствах. (Так, присоединение к элементу модели DOM функций, имеющих замыкание на узел этой модели, может привести в браузере Internet Explorer к неудачному исходу при попытке восстановить память после ухода со страницы.)

Попробуем организовать централизованное хранение информации, связанной с конкретными элементами модели DOM. Соответствующий пример кода приведен в листинге 13.4.

Листинг 13.4. Реализация центрального объекта для хранения данных об элементах модели DOM

```
<div title="Ninja Power!"> 忍者パワー！ </div>
<div title="Secrets"> 秘密 </div>

<script type="text/javascript">
  (function () {

    var cache = {},                          ❶ Установить локальное место хранения
        guidCounter = 1,
        expando = "data" + (new Date).getTime();

    this.getData = function (elem) {         ❷ Определить функцию getData()
      var guid = elem[expando];
      if (!guid) {
        guid = elem[expando] = guidCounter++;
        cache[guid] = {};
      }
      return cache[guid];
    };

    this.removeData = function (elem) {      ❸ Определить функцию removeData()
      var guid = elem[expando];
      if (!guid) return;
      delete cache[guid];
      try {
        delete elem[expando];
      }
      catch (e) {
        if (elem.removeAttribute) {
          elem.removeAttribute(expando);
        }
      }
    };
  })();

  var elems = document.getElementsByTagName('div');   ❹ Извлечь объекты для тестирования

  for (var n = 0; n < elems.length; n++) {            ❺ Присвоить связанные данные
    getData(elems[n]).ninja = elems[n].title;
```

```
  }

  for (var n = 0; n < elems.length; n++) {
    assert(getData(elems[n]).ninja === elems[n].title,
           "Stored data is " + getData(elems[n]).ninja);
  }

  for (var n = 0; n < elems.length; n++) {
    removeData(elems[n]);
    assert(getData(elems[n]).ninja === undefined,
           "Stored data has been destroyed.")
  }
</script>
```

❻ Проверить, были ли данные сохранены

❼ Проверить, были ли данные удалены

В данном примере кода устанавливаются две обобщенные функции, `getData()` и `removeData()`, для извлечения блока данных из элемента модели DOM и удаления этого блока соответственно, когда он больше не требуется. Для того чтобы не засорять глобальную область действия, потребуется ряд переменных, и поэтому они устанавливаются в немедленно вызываемой функции. И хотя эти переменные объявляются в области действия немедленно вызываемой функции, тем не менее они доступны для обеих функций обработки данных через их замыкания. (Как упоминалось в главе 5, замыкания играют главную роль во многих операциях, которые приходится реализовывать в коде JavaScript.)

В теле немедленно вызываемой функции устанавливаются три переменные ❶.

- `cache`. Содержит объект, в котором должны храниться данные, связываемые с элементами.
- `guidCounter`. Содержит текущий счетчик для формирования идентификаторов GUID элементов разметки.
- `expando`. Содержит имя свойства, которое должно быть введено в каждый элемент разметки для хранения его идентификатора GUID. Это имя формируется с использованием текущей метки времени, чтобы предотвратить любые потенциальные конфликты с определяемыми пользователем расширенными свойствами.

Далее определяется функция `getData()` ❷. Прежде всего в этой функции предпринимается попытка извлечь любой идентификатор GUID, который уже был присвоен элементу разметки при предыдущем вызове данной функции. Если это первый ее вызов для данного элемента, то его идентификатор GUID пока еще не существует, и поэтому он создается сызнова, увеличивая всякий раз счетчик на единицу, а затем присваивается элементу с использованием имени свойства, хранящегося в переменной `expando`. Кроме того, создается новый пустой объект, связанный с идентификатором GUID и сохраняемый в переменной `cache`.

Независимо от того, формируются ли кешируемые данные сызнова или нет, они возвращаются в виде значения из данной функции. А там, где эта функция вызывается, можно свободно ввести в кеш любые данные следующим образом:

```
var elemData = getData(element);
elemData.someName = 213;
elemData.someOtherName = 2058;
```


Функции также относятся к данным, и поэтому их можно косвенным образом связать с элементом разметки, как показано ниже.

```
elemData.someFunction = function(x){ /* сделать что-нибудь */ }
```

После установки функции `getData()` определяется функция `removeData()`, с помощью которой можно замести в событии все следы данных, если они больше не нужны ❸. В функции `removeData()` предпринимается попытка получить идентификатор GUID передаваемого ей элемента разметки, и если такой идентификатор отсутствует, то происходит немедленный возврат из данной функции. Отсутствие идентификатора GUID означает, что элемент не был им оснащен с помощью функции `getData()` или данные уже удалены.

Затем связанный блок данных удаляется из кеша и предпринимается попытка удалить расширенное свойство. При определенных обстоятельствах эта операция может завершиться неудачно, и в этом случае перехватывается ошибка и предпринимается попытка удалить атрибут, созданный от имени расширенного свойства. Таким образом, заметаются все следы, оставшиеся от действий, предпринятых в функции `getData()`: кешированный блок данных и расширенное свойство, размещенное в элементах разметки.

Реализовать все это в коде нетрудно, но нужно еще убедиться в его работоспособности. С этой целью два элемента разметки `<div>` устанавливаются в качестве объектов для тестирования, причем у каждого из них свой особый атрибут `title`. Затем получаются ссылки на эти элементы разметки ❹ и организуется циклическое обращение к ним, в ходе которого для каждого элемента разметки создается элемент данных, содержащий значение атрибута `title` из элемента разметки и получающий имя `ninja` ❺. После этого снова организуется циклическое обращение к элементам разметки, в ходе которого проверяется, связан ли с ними соответствующий элемент данных `ninja`, содержащий то же самое значение, что и у его атрибута `title` ❻. И наконец, организуется еще одно циклическое обращение к элементам разметки, в ходе которого вызывается функция `removeData()` для каждого элемента разметки и проверяется отсутствие данных ❼. Как показано на рис. 13.3, все эти тесты проходят.

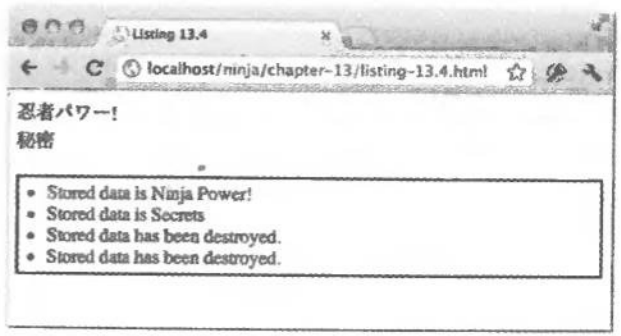

Рис. 13.3. Как показывают простые тесты, данные, связанные с элементом разметки, можно сохранять, даже не храня их в самом элементе разметки

Рассмотренные выше функции могут оказаться очень полезными и помимо управления обработчиками событий. С их помощью можно присоединить любого вида данные к элементу разметки. Но эти функции были созданы со специальной целью:

связать данные обработки событий с элементами разметки. А теперь воспользуемся этими функциями, чтобы создать отдельный набор функций для привязки и отвязки обработчиков событий от элементов разметки.

Управления обработчиками событий

Для того чтобы добиться полного управления процессом обработки событий, необходимо создать ряд функций, охватывающих привязку и отвязку событий. Благодаря этому можно построить единообразную в наибольшей степени модель обработки событий на всех платформах. И начать следует с привязки обработчиков событий.

Привязка обработчиков событий

Создав функцию для привязки событий, а не только для непосредственной привязки обработчиков событий, можно отслеживать обработчики, полностью контролируя процесс обработки событий. Такая функция должна устанавливать другую функцию в качестве обработчика, т.е. привязывать его, а также удалять другую функцию в качестве обработчика, т.е. отвязывать его. А в помощь ей потребуется ряд служебных функций. Итак, начнем с привязки обработчиков событий с помощью функции `addEvent()`, как показано в листинге 13.5.

Листинг 13.5. Функция для привязки обработчиков событий путем их отслеживания

```
(function(){

  var nextGuid = 1;

  this.addEvent = function (elem, type, fn) {

    var data = getData(elem);                    ❶ Получить связанный блок данных

    if (!data.handlers) data.handlers = {};      ❷ Создать объект для хранения обработчиков

    if (!data.handlers[type])
    data.handlers[type] = [];                    ❸ Создать массив по типу данных

    if (!fn.guid) fn.guid = nextGuid++;          ❹ Отменить оснащенные функции
    data.handlers[type].push(fn);                ❺ Добавить обработчик в список

    if (!data.dispatcher) {                      ❻ Создать диспетчер обработчиков
      data.disabled = false;
      data.dispatcher = function (event) {

        if (data.disabled) return;
        event = fixEvent(event);

        var handlers = data.handlers[event.type];
        if (handlers) {
          for (var n = 0; n < handlers.length; n++) {
            handlers[n].call(elem, event);
          }
```



```
        }
      };
    }

    if (data.handlers[type].length == 1) {       ❽ Зарегистрировать диспетчер
      if (document.addEventListener) {
        elem.addEventListener(type, data.dispatcher, false);
      }
      else if (document.attachEvent) {
        elem.attachEvent("on" + type, data.dispatcher);
      }
    }
  };
})();
```

На первый взгляд приведенный выше код функции кажется довольно сложным, но на самом деле он состоит из довольно простых фрагментов. Рассмотрим его по частям.

Прежде всего, в коде рассматриваемой здесь функции применяется обычный прием, когда все необходимое определяется в немедленно вызываемой функции, поскольку в данном случае требуется локальное хранилище, которое не следует путать с хранилищем по стандарту HTML5. В качестве такого хранилища служит текущий счетчик значений GUID в переменной `nextGuid`. А значения GUID должны служить в качестве однозначных меток подобно тому, как они применялись в листинге 13.4 и как поясняется ниже. Затем определяется функция `addEvent()`, принимаются элемент разметки, к которому привязывается обработчик событий, тип события и сам обработчик.

Сразу же после входа в тело данной функции блок данных, связанный с элементом разметки, извлекается и сохраняется в переменной `data` ❶, для чего служат функции, определенные ранее в листинге 13.4. А делается это по двум причинам.

- Ссылка на этот блок данных делается неоднократно, поэтому использование переменной сокращает последующие ссылки.
- С получением блока данных могут быть связаны определенные издержки, поэтому их лучше понести лишь один раз.

В данном случае требуется добиться как можно более высокой степени управления процессом привязки (и последующей отвязки) и поэтому лучше создать собственный вспомогательный обработчик событий, вместо того чтобы вводить для той же цели обработчик, передаваемый данной функции, непосредственно в элемент разметки. Этот вспомогательный обработчик будет зарегистрирован в браузере, чтобы отслеживать привязанные обработчики событий и запускать их на выполнение по мере надобности.

Вспомогательный обработчик будет далее называться *диспетчером*, чтобы каким-то образом отличать его от привязываемых обработчиков событий, передаваемых пользователями данной функции. Этот диспетчер будет создан ближе к концу данной функции, но прежде необходимо организовать хранилище для отслеживания привязываемых обработчиков событий. Такое хранилище будет создаваться динамически, т.е. по мере надобности в нем, а не заранее выделенным в памяти. В самом деле, зачем создавать массив для хранения обработчиков событий `mouseover`, если ни один из них вообще не будет привязан?

Обработчики событий должны быть связаны с тем элементом разметки, к которому они привязываются, через его блок данных, который получается из переменной `data`. Поэтому далее проверяется, содержит ли этот блок данных свойство `handlers`. И если он не содержит данное свойство, то создается соответствующий объект ❷. А при последующих вызовах рассматриваемой здесь функции для того же самого элемента разметки обнаружится, что данный объект уже существует, и поэтому его не нужно создавать.

В данном объекте создаются массивы, в которых будут храниться ссылки на выполняемые обработчики по типу событий. Но, как пояснялось выше, эти массивы будут распределяться в памяти только по мере надобности. Так, если объект `handlers` не содержит свойство, обозначаемое параметром `type`, передаваемым данной функции, то оно создается ❸. Благодаря этому события сохраняются в отдельных массивах по их типам, но это должны быть только типы событий, для которых имеются привязанные обработчики. Такое использование ресурсов следует считать вполне благоразумным.

Далее требуется пометить функции, обращение к которым происходит от имени той части кода, откуда вызывается рассматриваемая здесь функция (это делается по причинам, которые будут рассмотрены далее, когда речь пойдет о функции отвязки). С этой целью свойство `guid` добавляется в передаваемую функцию и увеличивается текущий счетчик ❹. И в этом случае проверяется, что данная операция выполняется лишь один раз для каждой функции, поскольку функция может быть привязана в качестве обработчика неоднократно по желанию автора веб-страницы.

На данном этапе уже известно, что имеется объект `handlers` и что он содержит массив, в котором отслеживаются обработчики событий передаваемого типа, и поэтому передаваемый обработчик событий помещается в конце данного массива ❺. По существу, это единственное действие в рассматриваемой здесь функции, выполнение которого гарантируется всякий раз, когда эта функция вызывается.

Теперь дело дошло до функции диспетчера. Когда эта функция вызывается в первый раз, никакого диспетчера вообще не существует. Но ведь и требуется только *один* диспетчер. Поэтому сначала проверяется его наличие, и если он отсутствует, то создается ❻.

В функции диспетчера, которая вызывается всякий раз, когда происходит привязка события, сначала проверяется установлен ли признак `disabled`. И если данный признак установлен, что на этом действие функции диспетчера завершается. (Далее в этой главе будет показано, при каких именно обстоятельствах требуется на время отменить диспетчеризацию событий.) Затем вызывается функция `fixEvent()`, определенная в листинге 13.3. В этой функции происходит циклическое обращение к обработчикам, зарегистрированным в массиве по типу события, обозначаемого экземпляром объекта типа `Event`. Каждый из этих обработчиков событий вызывается, и при этом ему предоставляется элемент разметки в качестве контекста функции, а также объект типа `Event` в качестве единственного аргумента ❼.

И наконец, проверяется, был ли создан первый обработчик событий данного типа. И если это подтверждается, то устанавливается делегат в качестве обработчика событий данного типа. Для этой цели используются подходящие средства того браузера, в котором выполняется прикладной код ❽.

Совет

Проверку на первый обработчик событий данного типа можно было бы перенести в условное выражение, в котором создание массива обработчиков событий проверяется ранее в рассмотренной выше функции ❸. Но код данной функции составлен именно так, как показано в листинге 13.5, ради удобства его пояснения, т.е. сначала в нем создаются конструкции данных, а затем делегат, в котором они используются. В выходном коде было бы благоразумнее совместить обе упомянутые проверки, чтобы не производить их лишний раз.

В конечном итоге функции, передаваемые рассмотренной выше функции, вообще не устанавливаются как обработчики событий. Напротив, они сохраняются и вызываются делегатом, когда наступает событие, а настоящим обработчиком оказывается сам делегат. Это дает возможность убедиться в том, что независимо от конкретной платформы должно происходить следующее.

- Экземпляр объекта типа `Event` нормализован.
- Целевой элемент разметки установлен в качестве контекста функции.
- Экземпляр объекта типа `Event` передан обработчику событий в качестве единственного аргумента.
- Обработчики событий всегда выполняются в том порядке, в котором они привязываются.

Таким уровнем управления процессом обработки событий, достигаемым с помощью рассмотренного здесь подхода, может по праву гордиться всякий, считающий себя мастером программирования на JavaScript.

Приборка за собой

Итак, в нашем распоряжении имеется функция для привязки событий, и теперь нам потребуется функция для их отвязки. А поскольку мы отказались от непосредственной привязки обработчиков событий, выбрав полное управление процессом обработки событий с помощью делегата в качестве единого обработчика, то не можем уже полагаться на функции отвязки, предоставляемые браузерами. Следовательно, нам придется создать их самостоятельно.

Помимо отвязки обработчиков событий, необходимо также аккуратно прибрать после себя. В рассмотренной выше функции привязки было уделено немало внимания тому, чтобы *не* выделять память без особой надобности, и поэтому было бы неразумно отнестись нерадиво к восстановлению памяти, которая высвобождается в результате отвязки. Оказывается, что уборку за собой нужно произвести не в одном месте, а следовательно, ее целесообразно выделить в отдельную функцию, как показано в листинге 13.6.

Листинг 13.6. Очистка конструкций обработчиков событий

```
function tidyUp(elem, type) {

  function isEmpty(object) {                ← ❶ Обнаружить пустой объект
    for (var prop in object) {
      return false;
    }
    return true;
  }

  var data = getData(elem);
                                            ❷ Проверить типы обработчиков
  if (data.handlers[type].length === 0) {
```

```
    delete data.handlers[type];

    if (document.removeEventListener) {
      elem.removeEventListener(type, data.dispatcher, false);
    }
    else if (document.detachEvent) {
      elem.detachEvent("on" + type, data.dispatcher);
    }
  }

  if (isEmpty(data.handlers)) {
    delete data.handlers;
   delete data.dispatcher;
  }

  if (isEmpty(data)) {
    removeData(elem);
  }
}
```

❸ Проверить наличие любых обработчиков

❹ Проверить, нужны ли данные вообще

В приведенном выше примере кода создается функция `tidyUp()`, принимающая элемент разметки и тип события в качестве своих аргументов. В этой функции далее проверяется наличие любых обработчиков событий данного типа. И если они отсутствуют, то освобождается как можно больше ненужной больше памяти. Это надежная мера, поскольку, как было показано ранее на примере функции `addEvent()`, если память для хранения данных потребуется снова, она будет выделена под хранилище по мере надобности.

Необходимо также проверить, имеются ли у объекта какие-нибудь свойства, хранящиеся в разных местах, или же он пуст. А поскольку в JavaScript для этой цели отсутствует специальный оператор, то подобную проверку придется организовать самостоятельно в виде отдельной функции `isEmpty()` ❶. Эта функция будет использоваться только в функции `tidyUp()`, и поэтому она объявляется в теле данной функции, чтобы максимально ограничить область ее действия.

Сначала очищаемый из памяти блок данных, связанных с элементом разметки, извлекается и сохраняется в переменной `data` для последующей ссылки. Затем начинается проверка всего, что требуется очистить из памяти. Прежде всего проверяется, пуст ли массив обработчиков событий передаваемого типа ❷. Если он пуст, то больше не нужен и может быть удален из памяти. Кроме того, отвязывается делегат, зарегистрированный в браузере, поскольку обработчики событий данного типа отсутствуют и этот делегат больше не нужен.

После удаления одного из массивов обработчиков событий указанного типа вполне может оказаться, что это был единственный такой массив. Следовательно, после него может остаться пустой объект `handlers`. Этот факт проверяется далее ❸, и если объект `handlers` действительно пуст, то он за ненадобностью удаляется, а вместе с ним и делегат, который также не нужен.

И наконец, проверяется, оказался ли в результате всех предыдущих удалений ненужным элемент разметки, связанный с блоком данных ❹. И если этот элемент действительно не нужен и бесполезен, то он очищается из памяти. Вот, собственно, и все, что требуется для того, чтобы убрать после себя, освободив неиспользуемую память.

Отвязка обработчиков событий

А теперь, когда в нашем распоряжении имеются все необходимые средства, чтобы прибрать за собой, мы можем приступить к разработке функции для отвязки обработчиков событий, привязанных с помощью функции `addEvent()`. Ради наибольшего удобства там, где эта функция вызывается, предоставляются следующие возможности.

- Отвязка всех событий, привязанных к отдельному элементу разметки.
- Отвязка всех событий определенного типа от элемента разметки.
- Отвязка отдельного обработчика событий от элемента разметки.

Все эти возможности обеспечиваются с помощью списка аргументов переменной длины. Чем больше информации будет предоставлено при вызове функции отвязки, тем более конкретной окажется сама операция отвязки.

Например, для удаления всех привязанных событий из элемента разметки можно было бы написать следующую строку кода:

```
removeEvent(element)
```

а для удаления все привязанных событий конкретного типа – такую строку кода:

```
removeEvent(element, "click");
```

И наконец, для удаления конкретного экземпляра обработчика событий подошла бы следующая строка кода:

```
removeEvent(element, "click", handler);
```

В последнем случае предполагается предварительное сохранение ссылки на исходный обработчик событий. Функция отвязки, предназначенная для выполнения всех перечисленных выше действий, приведена в листинге 13.7.

Листинг 13.7. Функция для отвязки обработчиков событий

```
this.removeEvent = function (elem, type, fn) {

  var data = getData(elem);

  if (!data.handlers) return;

  var removeType = function(t){
    data.handlers[t] = [];
    tidyUp(elem,t);
  };

  if (!type) {
    for (var t in data.handlers) removeType(t);
    return;
  }

  var handlers = data.handlers[type];
  if (!handlers) return;

  if (!fn) {
    removeType(type);
```

❶ Объявить функцию

❷ Извлечь связанный элемент данных

❸ Завершить функцию, если не требуется никаких действий

❹ Установить служебную функцию

❺ Удалить все привязанные обработчики

❻ Найти все обработчики заданного типа

❼ Удалить все обработчики заданного типа

```
    return;
  }

  if (fn.guid) {
    for (var n = 0; n < handlers.length; n++) {
      if (handlers[n].guid === fn.guid) {
        handlers.splice(n--, 1);
      }
    }
  }

  tidyUp(elem, type);

};
```

❽ Удалить один привязанный обработчик

Сначала в приведенном выше примере кода определяется сигнатура функции отвязки со следующими тремя параметрами: элемент разметки, тип события и функция обработчика событий ❶. Как пояснялось выше, там, где эта функция вызывается, два последних ее аргумента можно опустить за ненадобностью. Затем получается блок данных, связанных с элементом разметки, передаваемым данной функции ❷.

Совет

В рассматриваемой здесь функции предусматривается список аргументов переменной длины, и поэтому было бы целесообразно проверить наличие такого списка. Подумайте, как реализовать подобную проверку в качестве дополнительной возможности?

Как только будет получен блок данных, проверяется наличие любых привязанных обработчиков событий, и если они отсутствуют, то данная функция сразу же завершается ❸. Следует заметить, что для этой цели не потребовалась проверка содержимого объекта `handlers`, чтобы выяснить, пуст ли он или содержит пустой список обработчиков событий, поскольку все необходимые операции по освобождению памяти выполняются в разработанной ранее функции из листинга 13.6. Благодаря этому намного упрощается код рассматриваемой здесь функции.

По завершении предыдущей проверки становится известно, что придется удалить обработчики, привязанные по типу событий, т.е. по всем типам событий, если параметр `type` опущен, или по конкретному типу, если он указан. В любом случае обработчики придется удалить по типу событий не в одном месте, поэтому во избежание повторения кода далее определяется служебная функция ❺, которая удаляет все обработчики событий указанного ей типа `t`, заменяя массив обработчиков событий пустым массивом, а затем вызывая функцию `tidyUp()` для данного типа событий.

После определения этой функции проверяется, опущен ли параметр `type` ❻. И если он опущен, то удаляются все обработчики событий всех типов для данного элемента разметки. По завершении этой операции происходит возврат из рассматриваемой здесь функции, поскольку никаких действий по отвязке обработчиков событий выполнять больше не нужно.

На данной стадии выполнения рассматриваемой здесь функции должно быть уже ясно, что ей передан тип событий для удаления всех обработчиков, если аргумент `fn` опущен, или конкретного обработчика событий данного типа. Поэтому ради упрощения кода список обработчиков событий данного типа выбирается и сохраняется далее в переменной `handlers` ❻. Если такой список отсутствует, то происходит возврат из данной функций. Если же аргумент `fn` опущен, что вызывается служебная функция для удаления всех обработчиков указанного типа и возврат из рассматриваемой здесь функции ❼.

Примечание

Функция `removeEvent()` завершается операторами `return` в самых разных местах кода, приведенного в листинге 13.7. Некоторым разработчикам не нравится подобный стиль программирования, и поэтому они предпочитают употреблять единственный оператор `return`, управляя ходом выполнения кода с помощью глубоко вложенных условных операторов. Если вы относитесь к их числу, то можете переписать код данной функции, чтобы употребить в ней единственный оператор `return` или организовать подразумеваемый возврат из нее.

Неудачное завершение всех предыдущий проверок, означает, что рассматриваемой здесь функции передан конкретный обработчик событий для его удаления. Но если это не конкретный привязанный обработчик, то и не стоит его искать. Поэтому далее проверяется, введено ли свойство `guid` в функцию, что могло произойти при ее передаче функции `addEvent()`, а иначе ничего не делается. Если же речь идет о конкретном привязанном обработчике, то его поиск осуществляется в списке обработчиков событий, откуда удаляются все обнаруженные его экземпляры, поскольку они могут встретиться неоднократно ❽. И, как обычно, перед возвратом из данной функции осуществляется очистка и освобождение памяти.

Испытание функций на "дым"

Рассмотрим далее простое испытание разработанных ранее функций привязки и отвязки на "дым". Как и прежде, в коде соответствующего теста из листинга 13.8 компонуется небольшая веб-страница, предполагающая взаимодействие с ней в ручном режиме для выполнения простого и наглядного теста.

Примечание

Термин *испытание на "дым"* означает поверхностное тестирование работоспособности основных функций. Оно совсем не строгое и доскональное, а лишь позволяет убедиться в работоспособности тестируемого объекта в целом. Этот термин появился в конце XIX века, когда трубы испытывались на утечку пропусканием сквозь них дыма. Позднее он стал применяться и в электронике для обозначения первого испытания новой схемы, для чего ее подключали к источнику питания и смотрели, не сгорит и задымится ли она сразу. Если дым из схемы не повалит, значит, можно считать, что она в целом работоспособна!

Листинг 13.8. Испытание функций привязки и отвязки на "дым"

```
<script type="text/javascript">

  addEvent(window, "load", function () {

    var subjects = document.getElementsByTagName("div");
```

❶ Привязать обработчик события загрузки страницы

❷ Собрать объекты для тестирования

```
    for (var i = 0; i < subjects.length; i++) (function (elem) {

      addEvent(elem, "mouseover", function handler(e) {
        this.style.backgroundColor = "red";
      });

      addEvent(elem, "click", function handler(e) {
        this.style.backgroundColor = "green";
        removeEvent(elem, "click", handler);
      });
    })(subjects[i]);
  });

</script>

<div id="testSubject1" title="Click once"> 一度クリックします </div>
<div id="testSubject2" title="mouse over"> マウス </div>
<div id="testSubject3" title="many times"> 何度も </div>
```

❸ Привязать события от мыши

❹ Привязать события от щелчков кнопкой мыши

В приведенном выше простом тесте привязываются три разных типа событий, а отвязывается только одно из них. Сначала устанавливается обработчик событий `load` загрузки веб-страницы ❶, а после блока сценария определяются три объекта для тестирования (элементы разметки `<div>`). Поэтому выполнение остальной части сценария задерживается на время загрузки модели DOM. Как только наступит событие загрузки веб-страницы, в его обработчике собираются все элементы разметки `<div>` и организуется циклическое обращение к ним ❷. Для каждого из них устанавливаются следующие обработчики событий.

- Обработчик событий `mouseover`, выделяющий элемент разметки красным цветом ❸.
- Обработчик событий `click`, выделяющий элемент разметки зеленым цветом, а затем отвязывающийся, поэтому каждый элемент разметки реагирует на щелчок кнопкой мыши только *один* раз ❹.

После загрузки веб-страницы в браузер выполняются следующие действия.

1. Курсор мыши наводится на элементы разметки и далее наблюдается, все ли они становятся красными. Этим проверяется правильность привязки и активизации события `mouseover`.
2. Производится щелчок кнопкой мыши на элементе разметки и наблюдается, становится ли он зеленым. Этим проверяется правильность привязки и активизации события `click`. На рис. 13.4. приведен вид веб-страницы на данной стадии испытания на "дым".
3. Курсор мыши наводится на тот элемент разметки, на котором был произведен щелчок кнопкой мыши, и наблюдается, становится ли он красным, что и происходит благодаря обработчику событий `mouseover`. Затем на этом элементе разметки снова производится щелчок кнопкой мыши.
4. Если бы обработчики событий от щелчков кнопкой мыши были отвязаны правильно, они бы не запускались, а следовательно, элемент разметки не становился бы зеленым снова, но оставался бы красным. Как показывают наблюдения за испытанием на "дым", именно так и происходит.

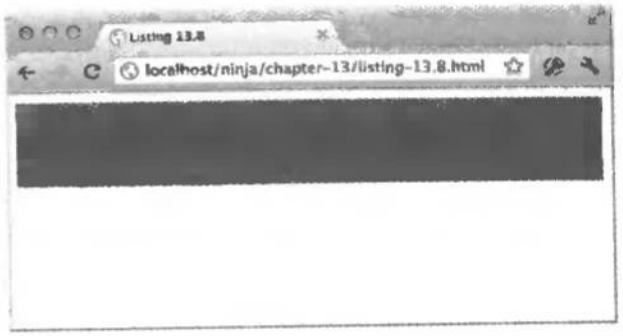

Рис. 13.4. Как показывают результаты испытания на "дым", по крайней мере некоторые из основных компонентов испытуемых функций привязки и отвязки действуют правильно

Разумеется, такое испытание на "дым" совсем не строгое. В качестве упражнения попробуйте составить ряд утверждений, чтобы автоматизировать процесс тестирования функций привязки и отвязки в полном объеме.

Поощрение

В файл `events.js`, загружаемый среди прочих примеров исходного кода к данной книге (подробнее об этом — в предисловии), включена удобная функция `proxy()`. Ею можно воспользоваться для изменения контекста функции обработчика событий при его запуске на нечто отличающееся от целевого события. Подобный специальный прием рассматривался в разделе "Особенности применения функций как объектов" главы 4.

Итак, мы убедились в возможности добиться полного контроля над привязкой и отвязкой событий. Рассмотрим далее, какие еще чудеса можно творить над событиями, как по мановению волшебной палочки.

Инициирование событий

В нормальных условиях события инициируются, когда происходят какие-нибудь действия со стороны пользователя, действия в браузере или в сети. Но иногда требуется инициировать одну и ту же реакцию на определенное действие под управлением сценария. (Ниже будет показано, что это не единственное желательное поведение, но требуется также обработка специальных событий.) Например, обработчик событий от щелчков кнопкой мыши может запускаться не только тогда, когда пользователь щелкнет на кнопке, но и в том случае, если произойдет какое-нибудь другое действие, в ответ на которое должен быть выполнен сценарий.

Конечно, для реализации такой возможности можно было бы, не мудрствуя лукаво, повторить код, но настоящий мастер должен найти более изящное решение. В частности, можно было бы заключить общий код обработки событий в тело именованной функции и вызывать ее из любого места. Но такое решение не лишено недостатков, связанных с нерациональным использованием пространства имен и усложнением кодовой базы. Кроме того, в подобных случаях вместо вызова функции лучше сымитировать событие. Таким образом, возможность запускать обработчики событий на выполнение без "настоящего" события сулит немало преимуществ.

Для запуска функции обработчика событий на выполнение необходимо обеспечить целый ряд условий.

- Запустить на выполнение привязанный обработчик событий для целевого элемента разметки.
- Создать условия для всплывания события вверх по модели DOM, чтобы попутно запускались любые другие привязанные обработчики событий.
- Создать условия для инициирования действия над целевым элементом, если таковое у него имеется.

В листинге 13.9 представлен пример кода функции, которая все это делает. При этом предполагается, что в данной функции используются разработанные ранее функции привязки и отвязки событий.

Листинг 13.9. Инициирование всплывающего события для элемента разметки

```
function triggerEvent(elem, event) {

  var elemData = getData(elem),
    parent = elem.parentNode || elem.ownerDocument;

  if (typeof event === "string") {
    event = { type:event, target:elem };
  }
  event = fixEvent(event);

  if (elemData.dispatcher) {
    elemData.dispatcher.call(elem, event);
  }

  if (parent && !event.isPropagationStopped()) {
    triggerEvent(parent, event);
  }
  else if (!parent && !event.isDefaultPrevented()) {

    var targetData = getData(event.target);

    if (event.target[event.type]) {

      targetData.disabled = true;

      event.target[event.type]();

      targetData.disabled = false;

    }
  }
}
```

Извлечь из элемента данные и ссылку на родительский элемент (для всплывания)

Если имя события передано в символьной строке, сформировать из него событие

Нормализовать свойства события

❶ Если переданный элемент содержит диспетчер, выполнить установленные обработчики событий

❷ Рекурсивно вызвать функцию для всплывания события вверх по модели DOM, если оно не остановлено явно

❸ Инициировать действие по умолчанию при достижении вершины модели DOM, если это не запрещено

❹ Проверить, имеется у целевого элемента действие по умолчанию для данного события

❺ Временно отменить диспетчеризацию действий в целевом элементе, так как обработчик уже выполнен

❻ Выполнить любое действие по умолчанию

Возобновить диспетчеризацию событий

Рассматриваемая здесь функция `triggerEvent()` принимает следующие два параметра:

- элемент, для которого инициируется событие;
- инициируемое событие.

Последний параметр может быть событийным объектом или символьной строкой, содержащей тип события.

Для того чтобы инициировать событие, необходимо пройти от инициатора события к вершине модели DOM, выполняя любые обнаруженные по пути обработчики событий ❶. Как только будет достигнут элемент документа, всплывание события завершится. После этого можно выполнить действие по умолчанию для типа события в целевом элементе, если оно у него имеется ❻.

Следует иметь в виду, что в процессе всплывания события проверяется, не остановилось ли его распространение ❷. И прежде чем выполнять любое действие по умолчанию, проверяется также, не было ли оно запрещено ❸. Кроме того, диспетчеризация событий отменяется на время выполнения действия по умолчанию ❺, чтобы исключить риск повторного выполнения, поскольку обработчики событий уже запущены.

Для того чтобы инициировать в браузере действие по умолчанию, вызывается соответствующий метод для целевого элемента разметки. Так, если инициируется событие фокуса, то проверяется, имеется ли у элемента разметки, исходно считающегося целевым для этого события, метод `focus()`, который затем и выполняется ❹. Возможность инициировать события под управлением сценария сама по себе очень удобна, но в то же время она неявно допускает также инициирование специальных событий. Поэтому рассмотрим далее, что собой представляют специальные события.

Специальные события

Возникало ли у вас когда-нибудь страстное желание инициировать свое собственное специальное событие? Допустим, вам требуется выполнить действие, но инициировать его при различных условиях из разных частей кода, возможно, даже из кода, находящегося в общих файлах сценариев.

Начинающий программист повторил бы в этом случае код там, где это требуется. Более опытный программист создал бы глобальную функцию и организовал бы ее вызов в нужном месте. А настоящий мастер программирования на JavaScript воспользуется для этой цели специальными событиями. Обсудим далее основания для такого решения.

Слабое связывание

Рассмотрим ситуацию, в которой операции должны выполняться из общего кода, причем в этом коде должно быть известно, когда следует реагировать на некоторое условие. Если пойти по пути создания глобальной функции, то на всех веб-страницах, где используется общий код, придется определить эту функцию с фиксированным именем, что является существенным недостатком такого подхода.

А что, если требуется выполнить несколько действий при наступлении инициирующего условия? Допустить несколько уведомлений было бы и тяжело, и совершенно ненужно. Все эти недостатки проявляются в результате *сильного связывания*, при котором в коде, обнаруживающем условие, должны быть известны подробности о коде, который будет реагировать на это условие. С другой стороны, *слабое связывание* возникает в том случае, если в коде, инициирующем условие, ничего неизвестно о коде, который будет реагировать на это условие, или даже о самой возможности реагирования на него.

К числу преимуществ обработчиков событий относится возможность установить их столько, сколько потребуется, причем все они будут действовать совершенно независимо друг от друга. Именно поэтому обработка событий служит характерным примером слабого связывания. Когда инициируется событие от щелчка на кнопке, в коде, инициирующем это событие, ничего неизвестно о тех обработчиках событий данного типа, которые установлены на веб-странице, или даже о том, имеются ли они вообще. Вместо этого событие от щелчка на кнопке помещается браузером в очередь событий (подробнее об этом см. в главе 3), а инициатору данного события совершенно безразлично, что с ним произойдет дальше. Если для события от щелчка на кнопке установлены соответствующие обработчики, то они будут в конечном итоге вызваны по отдельности и совершенно независимо друг от друга.

О слабом связывании можно было бы сказать еще немало. Что же касается рассматриваемой здесь ситуации с общим кодом, то когда в нем обнаруживается привлекающее внимание условие, формируется определенного рода сигнал, уведомляющий о следующем: произошло нечто любопытное и все, кого это заинтересует, могут отреагировать на него, а данному коду нет до того никакого дела. Вместо того чтобы изобретать собственную систему сигнализации, можно воспользоваться для данной цели кодом обработки событий, уже разработанным ранее в этой главе. Обратимся к конкретному примеру.

Пример обработки Ajax-запроса

Допустим, имеется некоторый общий код, автоматически выполняющий Ajax-запрос. Веб-страницы, на которых будет использоваться этот код по требованию, должны уведомляться о том, когда этот запрос начинается и когда завершается. И на каждой веб-странице должны находиться собственные средства для реагирования на подобные "события". Например, на одной странице требуется отобразить анимационное изображение вертушки в формате GIF, когда начнется Ajax-запрос, а по его завершении скрыть ее, чтобы дать пользователю наглядное представление о том, что данный запрос обрабатывается.

Если представить себе начальное условие как событие `ajax-start`, а конечное условие — как событие `ajax-complete`, то, казалось бы, нетрудно установить на веб-странице обработчики этих событий, отображающие и скрывающие изображение вертушки в нужный момент, как показано ниже.

```
var body = document.getElementsByTagName('body')[0];

addEvent(body, 'ajax-start', function(e){
  document.getElementById('whirlyThing').style.display = 'inline-block';
});

addEvent(body, 'ajax-complete', function(e){
  document.getElementById('whirlyThing').style.display = 'none';
});
```

К сожалению, эти события на самом деле не существуют. Но ведь мы разработали ранее код для ввода обработчиков событий, а также код для имитации их запуска. Поэтому мы можем вполне воспользоваться этим кодом и для имитации специальных событий, даже если браузерам ничего не известно о типах специальных событий.

Инициирование специальных событий

Специальные события — это способ имитации (для пользователя общего кода) восприятия настоящего события, не прибегая к базовой поддержке событий в браузерах. Ранее мы уже проделали определенную работу по организации поддержки кросс-браузерных событий. И оказывается, что поддержка специальных событий нами уже реализована!

Для поддержки специальных событий нам не нужно ничего менять в коде, уже написанном для функций `addEvent()`, `removeEvent()` и `triggerEvent()`. Ведь функционально нет никаких отличий между настоящим событием, инициируемым в браузере, и событием, которое на самом деле не существует, а только инициируется вручную. В листинге 13.10 приведен пример кода, в котором инициируется специальное событие.

Листинг 13.10. Инициирование специальных событий

```
<!DOCTYPE html>
<html>
  <head>
    <title>Listing 13.10</title>
    <meta charset="utf-8">
    <script type="text/javascript" src="data.js"></script>
    <script type="text/javascript" src="fixup.js"></script>
    <script type="text/javascript" src="events.js"></script>
    <script type="text/javascript" src="trigger.js"></script>
    <style type="text/css">
      #whirlyThing { display: none; }
    </style>

    <script type="text/javascript" src="ajaxy-operation.js"></script>

    <script type="text/javascript">

      addEvent(window, 'load', function(){

        var button = document.getElementById('clickMe');
        addEvent(button, 'click', function(){
          performAjaxOperation(this);
        });

        var body = document.getElementsByTagName('body')[0];

        addEvent(body, 'ajax-start', function(e){
          document.getElementById('whirlyThing')
            .style.display = 'inline-block';
        });

        addEvent(body, 'ajax-complete', function(e){
          document.getElementById('whirlyThing')
            .style.display = 'none';
        });
```

❶ Добавить обработчик событий от щелчков кнопкой мыши к кнопке, инициирующей Ajax-операцию в течение 5 секунд. Этому обработчику ничего неизвестно об изображении вертушки

❷ Установить в элементе разметки <body> обработчик специального события ajaxstart, приводящего к показу изображения вертушки. Какая-либо связь с кодом, реагирующим на щелчок на кнопке, отсутствует

❸ Установить в элементе разметки <body> обработчик специального события ajaxcomplete, приводящего к сокрытию изображения вертушки. И здесь какая-либо связь отсутствует

```
    });
  </script>
</head>
<body>

  <button type="button" id="clickMe">Start</button>

  <img id="whirlyThing" src="whirly-thing.gif" />

</body>
</html>
```

Создать кнопку, выбираемую щелчком ❶

Определить изображение вертушки, которое должно показываться только при выполнении Ajax-операции ❺

В приведенном выше примере кода специальные события бегло проверяются по следующему критерию: анимационное изображение ❺ должно показываться во время выполнения Ajax-операции. Эта операция инициируется щелчком ❶ на кнопке ❶. Обработчик события `ajax-start` ❷, как, впрочем, и обработчик события `ajax-complete` ❸, устанавливается без всякой связи с кодом, инициирующим оба события. В обработчиках этих специальных событий показывается и скрывается анимационное изображение вертушки ❺ соответственно.

Следует иметь в виду, что всем трем обработчикам событий ничего неизвестно о существовании друг друга. В частности, обработчик событий от щелчков на кнопке не несет никакой ответственности за показ и сокрытие анимационного изображения вертушки. Сама же Ajax-операция имитируется в приведенном ниже фрагменте кода.

```
function performAjaxOperation(target) {

  triggerEvent(target, 'ajax-start');

  window.setTimeout(function(){
    triggerEvent(target, 'ajax-complete');
  },5000);

}
```

Приведенная выше функция инициирует событие `ajax-start`, имитируя Ajax-запрос. А выбор кнопки в качестве первоначального инициатора данного события происходит произвольно. Обработчики событий устанавливаются в элементе разметки `<body>` (т.е. в обычном месте), а все события в конечном итоге всплывают вверх, достигая этого элемента разметки, и тогда запускается на выполнение соответствующий обработчик.

Далее в рассматриваемой здесь функции делается задержка по времени на 5 секунд, чтобы сымитировать обработку Ajax-запроса в течение этого периода времени. По истечении установленной задержки по времени имитируется возврат ответа на запрос и инициируется событие `ajax-complete`, чтобы обозначить завершение Ajax-операции. На рис. 13.5 показано, каким образом ход выполнения данной операции наглядно представлен в окне браузера.

Обратите внимание на высокую степень развязывания в данном примере. Общему коду выполнения Ajax-операции ничего неизвестно о том, что предполагается сделать в коде на веб-странице при наступлении специальных событий, и даже о том, существует ли этот код на странице вообще. А код на странице составлен в небольшие независимые друг от друга обработчики событий. В свою очередь, коду на странице ничего неизвестно о том, что делается в общем коде. Он просто реагирует на события, которые могут наступить, а могут и не наступить.

Такая степень развязывания помогает сделать код модульным, упрощает его написание и заметно облегчает его отладку, если в нем обнаружатся какие-нибудь ошибки. Она упрощает также совместное использование отдельных фрагментов кода и их перенос без опасения нарушить имеющуюся связь между ними. Развязывание дает неоспоримые преимущества, когда в прикладном коде используются специальные события, а также позволяет разрабатывать приложения в намного более выразительном и гибком стиле. Следует, однако, иметь в виду, что код, рассмотренный в этом разделе, служит характерным примером не только развязывания, но и *делегирования*.

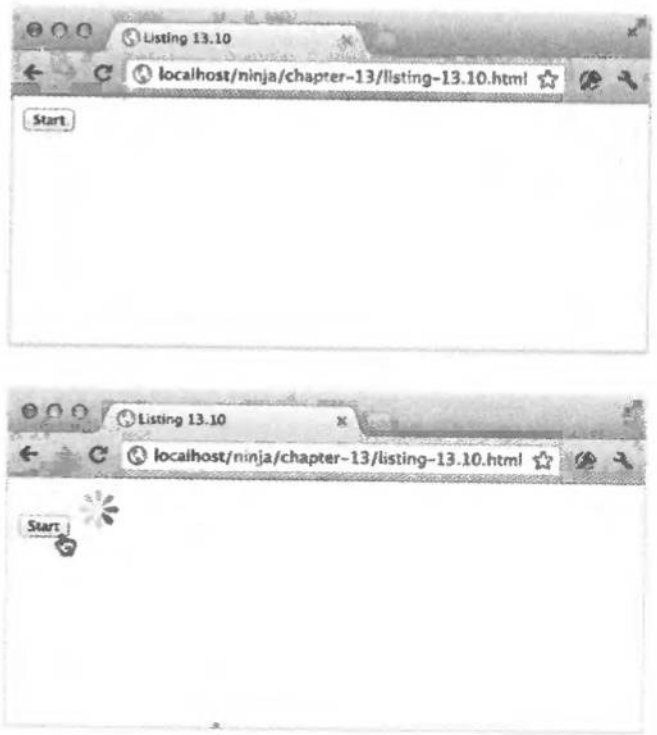

Рис. 13.5. Специальные события позволяют запускать на выполнение код их обработки без всякой связи с их инициатором

Всплывание и делегирование

Проще говоря, *делегирование* – это действие по установке обработчиков событий на более высоких уровнях модели DOM, чем представляющие интерес элементы. Напомним, что обработчики событий устанавливались в предыдущем примере кода в элементе разметки `<body>`, чтобы показывать и скрывать анимационное изображение, несмотря на то, что само изображение было спрятано в недрах модели DOM.

И это был пример делегирования полномочий над изображением родительскому элементу (в данном случае элементу разметки `<body>`). Но делегирование полномо-

чий ограничивалось в нем лишь специальными дескрипторами или даже элементом разметки <body>. А теперь рассмотрим возможность делегирования полномочий при использовании обычных типов событий и элементов разметки.

Делегирование событий родительскому элементу

Допустим, требуется наглядно показать, щелкнул ли пользователь кнопкой мыши на ячейке таблицы, отобразив сначала каждую ячейку белым цветом фона, а затем изменив этот цвет на желтый, как только она будет выбрана щелчком кнопкой мыши. С этой целью можно было бы организовать циклическое обращение ко всем ячейкам и установить в каждой из них обработчик событий, изменяющий свойство цвета фона следующим образом:

```
var cells = document.getElementsByTagName('td');

for (var n = 0; n < cells.length; n++) {
  addEvent(cells[n], 'click', function(){
    this.style.backgroundColor = 'yellow';
  });
}
```

Безусловно, такой прием вполне работоспособен, но вряд ли его можно считать изящным. Ведь тот же самый обработчик событий придется устанавливать в десятках, а возможно, и сотнях ячеек таблицы фактически для выполнения *одного и того* же действия.

Намного более изящный подход состоит в установке единственного обработчика событий на уровень выше, чем ячейки таблицы, чтобы он обрабатывал все события, используя механизм их всплывания, предоставляемый браузером. Известно, что все ячейки являются потомками объемлющей их таблицы и что по ссылке `event.target` можно обратиться к элементу, на котором был произведен щелчок кнопкой мыши. Но намного изысканнее было бы *делегировать* таблице обработку событий в ее ячейках следующим образом:

```
var table = document.getElementById('#someTable');

addEvent(table, 'click', function(event){
  if (event.target.tagName.toLowerCase() == 'td')
  event.target.style.backgroundColor = 'yellow';
});
```

В данном случае устанавливается единственный обработчик событий, легко справляющийся с задачей изменения цвета фона во всех ячейках таблицы, выбранных щелчком кнопкой мыши. Очевидно, что это намного более изящный и эффективный прием. Делегирование событий считается одним из самых лучших способов для разработки высокопроизводительных и масштабируемых веб-приложений.

Всплывание событий является единственным способом, доступным во всех браузерах, тогда как *фиксация* событий недоступна в версиях браузера Internet Explorer, предшествующих версии 9. Поэтому очень важно обеспечить делегирование событий для элементов, являющихся родителями тех элементов, которые являются инициаторами событий. Подобным образом гарантируется, что события будут в конечном итоге всплывать вверх, достигая того элемента, которому делегирована их обработка. На первый взгляд такой подход кажется вполне логичным и достаточно простым. Но как его осуществление будет выглядеть на практике? Попробуем найти ответ на этот вопрос.

Обходной прием для устранения отличий в браузерах

К сожалению, реализация в различных браузерах механизма всплывания событий типа `submit`, `change`, `focus` и `blur` вызывает серьезные трудности. И если мы хотим воплотить в жизнь делегирование событий, то должны найти какой-нибудь обходной прием для устранения этих отличий. Прежде всего, события `submit` и `change` не всплывают во всех устаревших версиях браузера Internet Explorer, тогда как в браузерах, совместимых с моделью DOM по стандарту консорциума W3C, механизм всплывания событий реализован более или менее согласованно. Поэтому воспользуемся, как и прежде, специальным приемом, позволяющим выяснить, существует ли препятствие, и если оно существует, то изящно обойти его. В данном случае требуется выяснить, способно ли событие всплывать вверх, достигая родительского элемента.

Один из примеров кода, в котором обнаруживается всплывание событий, приведен в листинге 13.11 и написан Юрием Зайцевым. (Более подробно этот код описан в блоге *Perfection Kills* (Совершенствование убивает) этого автора по адресу `http://perfectionkills.com/detecting-event-support-without-browser-sniffing/`.)

Листинг 13.11. Код, обнаруживающий всплывание событий и первоначально написанный Юрием Зайцевым

```
function isEventSupported(eventName) {

  var element = document.createElement('div'),
      isSupported;

  eventName = 'on' + eventName;
  isSupported = (eventName in element);

  if (!isSupported) {
    element.setAttribute(eventName, 'return;');
    isSupported = typeof element[eventName] == 'function';
  }

  element = null;

  return isSupported;
}
```

❶ Создать новый элемент разметки для тестирования. В дальнейшем он будет удален

❷ Проверить, поддерживается ли событие, убедившись в наличии свойства, поддерживающего это событие, в элементе

❸ Если простой прием не принесет успеха, создать атрибут обработки событий и проверить, насколько он подходит

Удалить временный элемент независимо от результата

Рассмотрим реализацию приема, позволяющего обнаружить всплывание событий, в приведенном выше коде. Сначала в этом коде проверяется, существует ли свойство `ontype` в элементе разметки , где `type` обозначает тип события ❷. Этот элемент разметки выбирается потому, что для таких элементов характерно большое разнообразие типов всплывающих событий, которые достигают их, в том числе событий `change` и `submit`. Но рассчитывать особенно на элемент разметки , уже находящийся на веб-странице, нельзя, и тем более не стоит вмешиваться в чужой элемент разметки. Поэтому для выяснения факта всплывания событий создается временный элемент разметки ❶.

Если упомянутая выше быстрая и простая проверка не приносит успеха, то далее следует более основательная попытка проверить наличие свойства `ontype` ❸. Если

же это свойство отсутствует, то создается атрибут ontype, которому присваивается немного кода, а далее проверяется, известно ли данному элементу, как преобразовать этот код в функцию. И если это известно, значит, ему известно также, как интерпретировать конкретное событие после всплывания. А теперь воспользуемся рассмотренным здесь кодом в качестве основания для реализации правильно функционирующего механизма всплывания событий во всех браузерах.

Всплывание событий submit

Событие submit относится к небольшому числу тех событий, которые не всплывают в устаревших версиях браузера Internet Explorer. Правда, это событие проще всего сымитировать. Событие submit может быть инициировано двумя способами.

- Привести в действие элемент ввода данных или кнопки типа submit или же элемент ввода данных типа image. Такие элементы приводятся в действие щелчком кнопкой мыши или нажатием клавиши <Enter> либо пробела при наведении на них курсора.
- Нажать клавишу <Enter>, установив курсор в поле ввода текста или пароля.

Зная об этих двух способах инициирования события submit, можно совместить с ним два инициируемых события, click и keypress, которые обычно всплывают. Примем пока что подход, который состоит в создании специальных функций для привязки и отвязки событий submit. Если выяснится, что события submit необходимо обрабатывать особым способом из-за отсутствия надлежащей поддержки со стороны браузеров, то придется установить совмещение событий, а иначе — осуществить привязку и отвязку обычным образом, как показано в примере кода из листинга 13.12.

Листинг 13.12. Совмещение события submit с событием click или keypress

```
script type="text/javascript">

  (function(){

    var isSubmitEventSupported = isEventSupported("submit");

    function isInForm(elem) {
      var parent = elem.parentNode;
      while (parent) {
        if (parent.nodeName.toLowerCase() === "form") {
          return true;
        }
        parent = parent.parentNode;
      }
      return false;
    }

    function triggerSubmitOnClick(e) {
      var type = e.target.type;
      if ((type === "submit" || type === "image") &&
           isInForm(e.target)) {
        return triggerEvent(this,"submit");
      }
    }
```

❶ Определить служебную функцию, чтобы проверить, присутствует ли переданный элемент в форме

❷ Предопределить обработчик событий click, чтобы выяснить, следует ли совмещать событие submit с событием click, и в этом случае запустить обработчик


```
    function triggerSubmitOnKey(e) {
      var type = e.target.type;
      if ((type === "text" || type === "password") &&
           isInForm(e.target) && e.keyCode === 13) {
        return triggerEvent(this,"submit");
      }
    }

    this.addSubmit = function (elem, fn) {

      addEvent(elem, "submit", fn);
      if (isSubmitEventSupported) return;

      // Здесь нужно добавить дополнительные обработчики событий,
      // если окажется, что это не форма.
      // Добавить обработчики событий только для первой привязки.
      if (elem.nodeName.toLowerCase() !== "form" &&
          getData(elem).handlers.submit.length === 1) {
        addEvent(elem, "click", triggerSubmitOnClick);
        addEvent(elem, "keypress", triggerSubmitOnKey);
      }
    };

    this.removeSubmit = function (elem, fn) {

      removeEvent(elem, "submit", fn);
      if (isEventSupported("submit")) return;

      var data = getData(elem);

        if (elem.nodeName.toLowerCase() !== "form" &&
            !data || !data.events || !data.events.submit) {
          removeEvent(elem, "click", triggerSubmitOnClick);
          removeEvent(elem, "keypress", triggerSubmitOnKey);
        }
    };
  })();

</script>
```

❸ Предопределить обработчик событий keypress, чтобы выяснить, следует ли совмещать событие submit с событием keypress, и в этом случае запустить обработчик

❹ Создать специальную функцию для привязки событий submit

❺ Привязать обработчик событий submit, как обычно, и выйти из функции, если в браузере имеется достаточная поддержка

❻ Если это не форма, но первый привязываемый обработчик событий submit, установить обработчики для совмещения с событиями click и keypress

❼ Создать специальную функцию для отвязки событий submit

❽ Отвязать обработчик событий submit, как обычно, и выйти из функции, если в браузере имеется достаточная поддержка

❾ Если это не форма, но последний отвязываемый обработчик событий submit, удалить обработчики для совмещения с событиями click и keypress

Сначала в приведенном выше коде определяется немедленно вызываемая функция (этот прием должен быть уже вам знаком), чтобы создать автономную среду для данного кода. Но прежде чем приступать к реализации специальной поддержки событий submit, необходимо определить ряд компонентов, которые потребуются в дальнейшем. Итак, сначала нужно выяснить, находится ли элемент разметки в ряде мест внутри формы, для чего определяется функция isInForm() ❶. Она просто обходит родительское дерево данного элемента, чтобы выявить один из его родителей в форме.

Затем определяются две функции, которые в дальнейшем будут служить в качестве обработчиков событий click и keypress. Первая из этих функций ❷ инициирует событие submit, если элемент разметки присутствует в форме, а целевой элемент имеет семантику предъявления формы, т.е. имеет тип submit или является элемен-

том ввода изображений. А вторая функция ❸ инициирует событие submit, если нажата клавиша <Enter> и целевой элемент присутствует в форме и является элементом ввода текста или пароля.

Как только эти функции обработки событий будут определены, можно приступать к написанию функций привязки и отвязки. В частности, функция привязки addSubmit() ❹ устанавливает обработчик событий submit, как обычно, используя функцию addEvent() ❺, а затем из нее происходит немедленный возврат, если в браузере имеется надлежащая поддержка механизма всплывания событий submit. В противном случае проверяется, не происходит ли привязка к форме (в этом случае для всплывания событий submit нет никаких препятствий) и привязывается ли первый обработчик событий submit ❻. Если же всплывание событий submit не поддерживается в браузере, то устанавливаются обработчики для совмещения с событиями click и keypress.

Аналогичным образом действует и функция отвязки removeSubmit() ❼. Событие submit отвязывается, как обычно, и происходит немедленный возврат из данной функции, если в браузере имеется соответствующая поддержка механизма всплывания событий submit ❽. В противном случае отвязываются обработчики для совмещения с событиями click и keypress, если целевой элемент не является формой и отвязывается последний обработчик событий submit ❻. Подобным способом можно преодолеть препятствия, возникающие при всплывании других событий в модели DOM, в том числе событий change.

Примечание

Описанная выше логика обработки событий submit было реализована в виде отдельных функций, в которых применяется функция addEvent(), чтобы основное внимание было легче сосредоточить на написании кода, требующегося для обработки событий submit. Но очевидно, что разделение логики на отдельные функции не очень удобно для вызывающей части прикладного кода. Поэтому всю эту логику следовало бы разместить в теле функции addEvent(), чтобы обработка событий submit происходила автоматически и незаметно для вызывающей части прикладного кода. Подумайте, как реализовать такую возможность в функции addEvent()?

Всплывание событий change

Событие change также относится к числу тех событий, которые не всплывают в устаревших версиях браузера Internet Explorer. К сожалению, реализовать его всплывание намного сложнее, чем для события submit. Для этого придется привязать ряд следующих разнотипных событий.

- Событие focusout для проверки значения после выхода из элемента разметки формы.
- События click и keydown для проверки значения сразу же после его изменения.
- Событие beforeactivate для получения предыдущего значения перед установкой нового.

В листинге 13.13 приведен пример реализации специальных функций, привязывающих и отвязывающих обработчики событий change путем их совмещения со всеми предыдущими событиями.

Листинг 13.13. Реализация кросс-браузерного всплывания события change

```
<script type="text/javascript">

  (function(){

    this.addChange = function (elem, fn) {

      addEvent(elem, "change", fn);
      if (isEventSupported("change")) return;

      if (getData(elem).events.change.length === 1) {
        addEvent(elem, "focusout", triggerChangeIfValueChanged);
        addEvent(elem, "click", triggerChangeOnClick);
        addEvent(elem, "keydown", triggerChangeOnKeyDown);
        addEvent(elem, "beforeactivate", triggerChangeOnBefore);
      }
    };

    this.removeChange = function (elem, fn) {

      removeEvent(elem, "change", fn);
      if (isEventSupported("change")) return;

      var data = getData(elem);
      if (!data || !data.events || !data.events.submit) {
        addEvent(elem, "focusout", triggerChangeIfValueChanged);
        addEvent(elem, "click", triggerChangeOnClick);
        addEvent(elem, "keydown", triggerChangeOnKeyDown);
        addEvent(elem, "beforeactivate", triggerChangeOnBefore);
      }
    };

    function triggerChangeOnClick(e) {
      var type = e.target.type;
      if (type === "radio" || type === "checkbox" ||
          e.target.nodeName.toLowerCase() === "select") {
        return triggerChangeIfValueChanged.call(this, e);
      }
    }

    function triggerChangeOnKeyDown(e) {
      var type = e.target.type,
          key = e.keyCode;
      if (key === 13 && e.target.nodeName.toLowerCase() !== "textarea" ||
          key === 32 && (type === "checkbox" || type === "radio") ||
          type === "select-multiple") {
        return triggerChangeIfValueChanged.call(this, e);
      }
    }
```

Определить специальную функцию для привязки событий change

Добавить обработчик, как обычно, и выйти, если в браузерах имеется надлежащая поддержка

Совместить с другими событиями в первом привязываемом обработчике событий change

Определить специальную функцию для отвязки событий change

Удалить обработчик, как обычно, если в браузерах имеется надлежащая поддержка

Удалить совмещения, если отвязывается последний обработчик событий change

Обработчик совмещаемых событий click

Обработчик совмещаемых событий keydown

```
  function triggerChangeOnBefore(e) {
    getData(e.target)._change_data = getVal(e.target);
  }

  function getVal(elem) {
    var type = elem.type,
        val = elem.value;
    if (type === "radio" || type === "checkbox") {
       val = elem.checked;
    } else if (type === "select-multiple") {
      val = "";
      if (elem.selectedIndex > -1) {
        for (var i = 0; i < elem.options.length; i++) {
          val += "-" + elem.options[i].selected;
        }
      }
    } else if (elem.nodeName.toLowerCase() === "select") {
      val = elem.selectedIndex;
    }
    return val;
  }

  function triggerChangeIfValueChanged(e) {
    var elem = e.target, data, val;
    var formElems = /textarea|input|select/i;
    if (!formElems.test(elem.nodeName) || elem.readOnly) {
      return;
    }
    data = getData(elem)._change_data;
    val = getVal(elem);
    if (e.type !== "focusout" || elem.type !== "radio") {
      getData(elem)._change_data = val;
    }
    if (data === undefined || val === data) {
      return;
    }
    if (data != null || val) {
      return triggerEvent(elem, "change");
    }
  }
 })();

</script>
```

Обработчик совмещаемых событий beforeactivate, где сохраняется значение элемента для грядущего события focusout

Служебная функция для извлечения значения передаваемого ей элемента

Обработчик совмещаемых событий focusout, который запускается, если изменилось значение элемента

Большая часть этого кода написана в том же духе, что и код из листинга 13.12, поэтому мы не будем анализировать его подробно. Единственное его отличие заключается в том, что в нем обрабатывается больше событий. Это отличие проявляется, главным образом, в коде функций `getVal()` и `triggerChangeIfValueChanged()`.

В частности, функция `getVal()` возвращает значение, представляющее собой сериализованный вариант состояния передаваемого ей элемента разметки формы. При наступлении любого события `beforeactivate` это значение будет сохраняться для последующего использования в свойстве `_change_data` объекта, находящегося в данном элементе. А функция `triggerChangeIfValueChanged()` отвечает за обнаружение фактических изменений, произошедших в промежутке между сохранением предыдущего значения элемента и установкой нового, а также за инициирование события `change`, если эти значения отличаются.

Помимо обнаружения изменений, произошедших после наступления события `focusout` (при выходе из элемента разметки формы), проверяется также, была ли нажата клавиша <Enter> на любом элементе разметки формы, кроме элемента `textarea`, или же клавиша пробела на флажке либо кнопке-переключателе. Кроме того, проверяется, был ли произведен щелчок кнопкой мыши на флажке, кнопке-переключателе или элементе выбора, поскольку эти действия также повлекут за собой наступление события `change`.

В рассматриваемом здесь коде выполняется немало операций по поддержке механизма всплывания событий, которая должна быть присуща самим браузерам. Но потребность в таком коде отпадет лишь тогда, когда устаревшие версии браузера Internet Explorer канут в прошлое.

Реализация обработки событий `focusin` и `focusout`

События `focusin` и `focusout` относятся к числу оригинальных событий, внедренных в браузере Internet Explorer в качестве эквивалентов событий `focus` и `blur`. Эти события наступают в любом элементе разметки или производном от него элементе при наведении или отведении от него курсора. Следовательно, эти события относятся к разряду фиксирующих, а не всплывающих.

События `focus` и `blur` достойны внимания, потому что они не всплывают, как предписывается рекомендациями консорциума W3C для модели DOM, что и реализовано во всех браузерах. Оказывается, что намного проще реализовать аналоги событий `focus` и `blur` во всех браузерах, чем пытаться обойти намерения браузеров следовать стандартам и обеспечить всплывание этих событий.

Самый лучший способ реализовать обработку событий `focus` и `blur` — видоизменить уже имеющуюся функцию `addEvent()`, чтобы преобразовать их типы путем непосредственной подстановки, как показано ниже.

```
if (document.addEventListener) {
  elem.addEventListener(
    type === "focusin" ? "focus" :
      type === "focusout" ? "blur" : type,
    data.handler, type === "focusin" || type === "focusout");
}
else if (document.attachEvent) {
  elem.attachEvent("on" + type, data.handler);
}
```

А затем видоизменить функцию `removeEvent()`, чтобы отвязать эти события тем же способом, как показано ниже.

```
if (document.removeEventListener) {
  elem.removeEventListener(
    type === "focusin" ? "focus" :
      type === "focusout" ? "blur" : type,
    data.handler, type === "focusin" || type === "focusout");
}
```

```
else if (document.detachEvent) {
  elem.detachEvent("on" + type, data.handler);
}
```

В конечном итоге организуется поддержка обработки нестандартных событий focusin и focusout во всех браузерах. Естественно, что специальную логику обработки этих событий желательно отделить от внутренней логики функций addEvent() и removeEvent(). В этом случае можно было бы реализовать их в некоторой форме расширяемости, чтобы заменить собственные механизмы привязки и отвязки, предоставляемые браузерами для особых типов событий.

Дополнительные сведения о кросс-браузерной обработке событий focus и blur можно почерпнуть из блога на веб-сайте QuirksMode по адресу http://www.quirksmode.org/blog/archives/2008/04/delegating_the.html. Имеются и другие нестандартные типы событий, которые следовало бы рассмотреть, что и будет сделано ниже.

Реализация обработки событий mouseenter и mouseleave

События mouseenter и mouseleave относятся к числу специальных событий, внедренных в браузере Internet Explorer для упрощения процесса, в ходе которого выясняется, находится ли курсор мыши в пределах элемента разметки или за его пределами. Как правило, обработка стандартных событий mouseover и mouseout обеспечивается в браузерах, но зачастую они не предоставляют достаточных для этого средств. Дело в том, что событие инициируется в браузерах при перемещении курсора не только между родительским и порожденными элементами разметки, но и между самими порожденными элементами. Это типичное условие для срабатывания механизма всплывания событий, но при реализации элементов пользовательского интерфейса возникает серьезное затруднение, когда требуется просто выяснить, находится ли курсор по-прежнему в пределах родительского элемента разметки. При этом совсем не обязательно уведомлять, переместился ли курсор к его порожденному элементу. Эта ситуация наглядно показана на рис. 13.6.

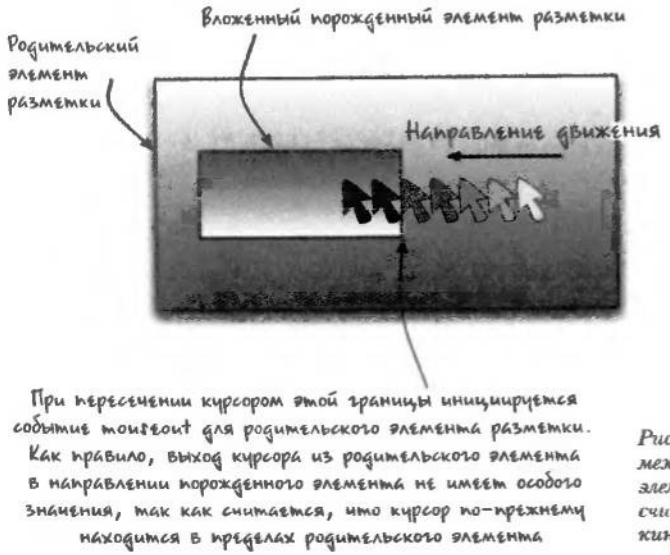

Рис. 13.6. Если курсор пересек границу между родительским и порожденным элементами разметки, то следует ли считать, что курсор действительно покинул пределы родительского элемента?

Когда курсор мыши пересекает границу между родительским и порожденным элементами разметки, инициируется событие mouseout, несмотря на то, что курсор может считаться по-прежнему находящимся в пределах родительского элемента. Аналогично событие mouseover инициируется, когда покидаются пределы порожденного элемента разметки. И в этой ситуации особенно удобными оказываются события mouseenter и mouseleave. Они инициируются только для главного элемента разметки, к которому они привязаны, уведомляя лишь о том, что курсор фактически покинул пределы родительского элемента разметки. А поскольку обработка этих полезных событий в настоящее время реализована только в браузере Internet Explorer, то ее придется полностью воспроизвести для всех остальных браузеров. В листинге 13.14. приведен пример кода, в котором реализуется функция hover(), обеспечивающая поддержку обработки событий mouseenter и mouseleave во всех браузерах.

Листинг 13.14. Дополнительная поддержка обработки событий mouseenter и mouseleave во всех браузерах

```
<script>

  (function() {

    if (isEventSupported("mouseenter")) {

      this.hover = function (elem, fn) {
        addEvent(elem, "mouseenter", function () {
          fn.call(elem, "mouseenter");
        });

        addEvent(elem, "mouseleave", function () {
          fn.call(elem, "mouseleave");
        });
      };
    }
    else {

      this.hover = function (elem, fn) {
        addEvent(elem, "mouseover", function (e) {
          withinElement(this, e, "mouseenter", fn);
        });

        addEvent(elem, "mouseout", function (e) {
          withinElement(this, e, "mouseleave", fn);
        });
      };
    }

    function withinElement(elem, event, type, handle) {

      var parent = event.relatedTarget;

      while (parent && parent != elem) {
        try {
```

Проверить, имеется ли в браузере собственная поддержка обработки событий mouseenter и mouseleave

Ввести обработчики, вызывающие обработчик для тех браузеров, где эти события поддерживаются

Обработать события mouseover и mouseout в тех браузерах, где они не поддерживаются, используя обработчик, в котором выявляется, следует ли запускать нужный обработчик событий

Внутренний обработчик, запускающий исходный обработчик событий для имитации нестандартного поведения

Получить элемент, в который входит или покидает курсор

Продвинуться вверх до тех пор, пока не будет достигнута вершина модели DOM или элемент, на который наведен курсор

```
        parent = parent.parentNode;
      }
      catch (e) {
        break;
      }
    }
    if (parent != elem) {
      handle.call(elem, type);
    }
  }
})();

</script
```

Если возникнет ошибка, снимать обработку завершенной (такое может произойти с элементами разметки XUL в браузере Firefox)

Запустить обработчик событий, если курсор не покидает наведенный элемент или не входит в него

Большая часть логики обработки событий `mouseenter` и `mouseleave` находится в функции `withinElement()`, которая устанавливается в качестве обработчика событий `mouseenter` и `mouseleave` в тех браузерах, где обработка нестандартных событий не поддерживается. В этой функции проверяется содержимое свойства `relatedTarget` события, которое становится элементом разметки, в который входит курсор для наступления событий `mouseout`, или же элементом разметки, который покидает курсор для наступления событий `mouseover`. Но в любом случае элемент, связанный с наведенным элементом, игнорируется, если он вложен в него. В противном случае считается, что это наведенный элемент, который покидает курсор или в который входит курсор, и поэтому запускается обработчик соответствующих событий.

И в завершение этой главы следует упомянуть еще об одном событии, которое может оказаться полезным при разработке веб-приложений. Обсудим его вкратце в следующем разделе.

Событие готовности документа

И последнее событие из рассматриваемых в этой главе относится к типу готовности документа и реализовано как событие `DOMContentLoaded` в браузерах, совместимых с моделью DOM по стандарту консорциума W3C. Это событие наступает, как только загружается весь DOM-документ, указывая на то, что он готов к обходу и манипулированию. Такое событие стало неотъемлемой частью многих современных интегрированных сред, допуская ненавязчивое наслоение кода. Оно наступает перед отображением веб-страницы и не ожидает загрузки остальных ресурсов, способных задержать инициирование события `load`.

Кросс-браузерная обработка подобных событий опять же затруднена из-за необходимости поддерживать версии браузера Internet Explorer, предшествующие версии 9. В браузерах, совместимых со стандартами консорциума W3C, инициировать событие `DOMContentLoaded` совсем не трудно, когда DOM-документ готов к употреблению. Но в устаревших версиях браузера Internet Explorer приходится применять комплексный подход, чтобы получить уведомление о готовности DOM-документа.

Для этой цели можно, в частности, воспользоваться специальным приемом, разработанным Диего Перини (Diego Perini) и описанным по адресу `http://javascript.nwbox.com/IEContentLoaded/`. Этот прием состоит в том, чтобы попытаться прокрутить документ в крайнее левое положение, которое является вполне естественным

для него. Такая попытка окажется безуспешной, если документ еще не загружен. Но если предпринимать ее непрерывно, используя таймер, чтобы не заблокировать цикл ожидания событий, то в конечном итоге можно выяснить, что DOM-документ загружен и готов к употреблению, как только данная попытка окажется успешной.

Второй путь комплексного решения рассматриваемой здесь задачи в устаревших версиях браузера Internet Explorer состоит в приеме события `onreadystatechange` для документа. Это событие действует менее устойчиво, чем событие `doScroll` в первом способе. Оно всегда наступает после загрузки DOM-документа, но иногда и намного позже, хотя и всегда перед событием `load`, наступающим при окончательной загрузке документа в окно браузера. Тем не менее такой способ может стать неплохой поддержкой для обработки событий готовности документа в устаревших версиях браузера Internet Explorer. По крайней мере, он позволяет убедиться в готовности документа к употреблению до наступления события `load`.

И третий путь состоит в анализе свойства `document.readyState`. Это свойство доступно во всех браузерах, и в нем регистрируется полнота загрузки DOM-документа на текущий момент. Анализируя его, можно определить момент, когда будет достигнуто состояние полной загрузки страницы. Длительные задержки при загрузке, особенно в браузере Internet Explorer, могут стать причиной того, что свойство `readyState` слишком рано уведомит о полной загрузке документа. Именно поэтому нельзя полагаться только на данное свойство. Тем не менее проверка свойства `readyState` в процессе загрузки документа позволит избежать ненужной привязки событий, если DOM-документ уже находится в состоянии готовности к употреблению.

Примечание

Подробнее о состоянии документа можно узнать из документации на браузер Mozilla Firefox по адресу `https://developer.mozilla.org/en-US/docs/DOM/document.readyState`.

Рассмотрим пример, в котором обработка событий готовности документа реализуется тремя описанными выше способами. Код этого примера приведен в листинге 13.15.

Листинг 13.15. Реализация кросс-браузерной обработки событий готовности DOM-документа

```
script type="text/javascript">

  (function () {

    var isReady = false,
      contentLoadedHandler;

    function ready() {
      if (!isReady) {
        triggerEvent(document, "ready");
        isReady = true;
      }
    }

    if (document.readyState === "complete") {
      ready();
```

Начать с допущения, что документ не готов

Определить функцию, запускающую обработчик событий готовности документа лишь один раз, а при последующих вызовах ничего не делать

Если документ готов на данный момент, запустить обработчик

```
  }

  if (document.addEventListener) {
    contentLoadedHandler = function () {
      document.removeEventListener(
        "DOMContentLoaded", contentLoadedHandler, false);
      ready();
    };

    document.addEventListener(
        "DOMContentLoaded", contentLoadedHandler, false);
  }
  else if (document.attachEvent) {
    contentLoadedHandler = function () {
      if (document.readyState === "complete") {
        document.detachEvent(
          "onreadystatechange", contentLoadedHandler);
        ready();
      }
    };

    document.attachEvent(
        "onreadystatechange", contentLoadedHandler);

    var toplevel = false;
    try {
      toplevel = window.frameElement == null;
    }
    catch (e) {
    }

    if (document.documentElement.doScroll && toplevel) {
      doScrollCheck();
    }
  }

  function doScrollCheck() {
    if (isReady) return;
    try {
      document.documentElement.doScroll("left");
    }
    catch (error) {
      setTimeout(doScrollCheck, 1);
      return;
    }
    ready();
  }
})();

</script>
```

Создать для браузеров, действующих по стандартам W3C, обработчик событий DOMContentLoaded, который запускает обработчик событий готовности документа и удаляет себя

Установить только что созданный обработчик событий DOMContentLoaded

Создать для модели IE обработчик, запускающий обработчик событий готовности документа, если свойство readyState содержит состояние полной загрузки документа, а затем удаляющий себя

Установить предыдущий обработчик событий onreadystatechange. Он, скорее всего, запустится слишком поздно, но надежно для встраиваемого фрейма

Выполнить проверку на прокрутку, если это не встраиваемый фрейм

Определить функцию для проверки на прокрутку, в которой предпринимается попытка выполнить прокрутку документа до тех пор, пока она не окажется успешной

Реализовав обработку событий готовности документа, мы можем считать, что в нашем распоряжении уже имеются все необходимые инструментальные средства для построения завершенной системы обработки событий по модели DOM. И теперь самое время отметить свои достижения любимым напитком.

Резюме

В этой главе рассмотрена вся система обработки событий по модели DOM, которую трудно назвать простой. Да и модель IE для обработки событий в устаревших версиях браузера Internet Explorer вызывает серьезные трудности, которые приходится каким-то образом преодолевать, хотя эти версии вряд ли будут поддерживаться через несколько лет. Но подобные трудности характерны не только для браузера Internet Explorer. Ведь даже в тех браузерах, где соблюдаются стандарты консорциума W3C, недостает расширяемости собственных прикладных интерфейсов API, а следовательно, и в этом случае придется как-то преодолевать возникающие препятствия, усовершенствуя систему обработки событий, чтобы прийти к решению, наиболее универсальному для всех браузеров. В этой главе были рассмотрены следующие вопросы.

- В браузерах применяются три модели обработки событий, которые, вероятнее всего, требуют поддержки в прикладном коде.
  - Модель DOM Level 0, которая больше всего известна, но непригодна для надежного управления событиями.
  - Модель DOM Level 2, являющаяся стандартом консорциума W3C, но ей недостает многих средств, необходимых для создания полноценной системы управления событиями.
  - Модель IE, являющаяся оригинальной для браузера Internet Explorer, но обладающая еще меньшим набором средств, чем у модели DOM Level 2, хотя именно ею приходится пользоваться в устаревших версиях браузера Internet Explorer.
- К недостаткам модели IE относится отсутствие подходящего контекста в обработчиках событий. Поэтому для нормализации событий в этой главе был разработан ряд функций привязки и отвязки.
- Еще один недостаток заключается в отличии данных о событиях в моделях DOM Level 2 и IE. И для устранения подобных отличий в этой главе разработана функция, приводящая экземпляры событийных объектов к согласованному виду на всех платформах.
- Для хранения данных, относящихся к отдельным элементам разметки, не прибегая к глобальному хранилищу, требуются соответствующие средства. Поэтому в этой главе был разработан способ для заполнения данными элементов разметки. И хотя этот способ был в конечном итоге использован для хранения данных в процессе обработки событий, тем не менее он носит универсальный характер и может служить многим другим целям.
- Функции привязки и отвязки событий были усовершенствованы с целью использовать средства хранения данных для отслеживания обработчиков событий всех типов в любом элементе разметки.

- К числу самых важных средств, введенных в систему обработки событий, относится инициирование событий под управлением сценария. И хотя это средство удобно само по себе, оно открывает немало других полезных возможностей, в том числе формирование и инициирование специальных событий.
- Формирование и инициирование специальных событий позволяет внедрить принцип свободного связывания, чтобы делать на веб-странице практически все, что угодно. Благодаря этому значительно упрощается построение независимых модульных компонентов.
- В этой главе было также показано, как изящно и эффективно свести к минимуму объем кода, необходимого для создания и установки обработчиков событий, делегируя обработку событий родителям целевого объекта.
- Далее в этой главе основное внимание было сосредоточено на отличиях в браузерах и разработаны способы для реализации следующих возможностей.
  - Всплывание событий `submit` подобно другим событиям.
  - Всплывание событий `change` подобно другим событиям.
  - Обработка событий `focusin` и `focusout` во всех браузерах.
  - Обработка событий `mouseenter` и `mouseleave` во всех браузерах.
- И наконец, в главе был разработан обработчик событий готовности документа, который запускается во всех браузерах, чтобы знать, когда DOM-документ будет загружен и готов для манипулирования, прежде чем наступит событие `load`.

Теперь у вас имеется достаточно знаний и навыков для реализации полноценной и полезной системы обработки событий по модели DOM, способной преодолевать даже самые трудные препятствия, которые ставят модели обработки событий в браузерах. Но трудности обращения с браузерами на этом не заканчиваются. Манипулирование самой моделью DOM доставляет немало хлопот и неприятностей. О том, как справиться с ними, речь пойдет в следующей главе.

14 Манипулирование моделью DOM

В этой главе...

- Вставка HTML-разметки на странице
- Клонирование элементов разметки
- Удаление элементов разметки
- Манипулирование текстом в элементах разметки

Если открыть любую библиотеку JavaScript, то в ней, как ни странно, можно обнаружить немало фрагментов длинного и сложного кода для выполнения простых операций в модели DOM. Даже такие, на первый взгляд, простые операции, как клонирование или удаление узла (им соответствуют методы `cloneNode()` и `removeChild()` в модели DOM), реализуются относительно сложно. И в связи с этим возникают следующие два вопроса.

- Почему этот код такой сложный?
- Зачем разбираться в нем, если библиотека возьмет на себя все хлопоты по выполнению нужных операций?

Самой побудительной причиной для оптимизации такого кода служит *производительность*. Имея ясное представление о том, как в библиотеках реализуется модификация модели DOM, вы сможете написать код, который будет работать лучше и быстрее, обращаясь к библиотеке. А кроме того, у вас появится возможность применить в своем коде специальные приемы из библиотеки.

Большинство тех, кто пользуется библиотеками, вряд ли удивят следующие два обстоятельства: в библиотеках не только устраняется больше кросс-браузерных несоответствий, чем в типичном прикладном коде, но и зачастую их код выполняется бы-

стрее. В побудительных причинах для повышения производительности кода библиотек нет ничего удивительного, поскольку их разработчики постоянно учитывают самые последние дополнения в браузерах. Для этой цели в библиотеках применяются самые лучшие способы и приемы, позволяющие создавать наиболее производительный код.

Например, для вставки фрагментов HTML-разметки на веб-странице в библиотеках используются *фрагменты документа* или метод `createContextualFragment()`. Но ни один из этих приемов не применяется в современной разработке веб-приложений, хотя оба позволяют вставлять элементы разметки на веб-странице намного быстрее, чем это делают самые известные методы, в том числе и `createElement()`.

Еще одна возможность для повышения производительности лежит в области управления памятью. Можно с относительной уверенностью сказать, что большинство разработчиков редко обращают внимание на использование памяти в своих веб-приложениях. Совсем иначе обстоит дело в библиотеках JavaScript, где приходится принимать во внимание использование памяти, чтобы исключить излишнее дублирование ресурсов. В примерах кода, приведенных в этой главе, будут представлены многие приемы и способы, с помощью которых можно сократить потребление оперативной памяти в веб-приложениях.

Итак, в этой главе будут рассмотрены все осложнения кросс-браузерного характера, которые возникают в коде модификации модели DOM, а также возможности повышения его производительности. Ясное представление о том, как повышение производительности достигается на практике, позволит вам разрабатывать веб-приложения, которые будут работать быстрее, чем обычно. Ниже перечислены некоторые полезные ресурсы для дальнейшего изучения рассматриваемых здесь вопросов.

- Новейший и весьма распространенный метод `range.createContextualFragment()`, хотя и не вошедший еще в библиотеку jQuery: `https://developer.mozilla.org/en/DOM/range.createContextualFragment`.
- Библиотека `metamorph.js`, в которой реализуется манипулирование моделью DOM и которая достойна всяческого внимания: `https://github.com/tomhuda/metamorph.js/blob/master/lib/metamorph.js`.

А теперь перейдем от слов к делу. Засучив рукава, углубимся в изучение особенностей манипулирования моделью DOM.

Вставка HTML-разметки

В этой главе рассматривается эффективный способ вставки HTML-разметки в произвольном месте документа. Этот конкретный способ выбран потому, что он зачастую применяется для решения следующих задач.

- Вставка произвольной HTML-разметки на странице, ввода шаблонов и манипулирования ими на стороне клиента.
- Извлечение и вставка HTML-содержимого, получаемого с сервера.

Решение этих задач вызывает определенные технические трудности, особенно в сравнении с построением объектно-ориентированной конструкции DOM прикладного интерфейса API, которая реализуется проще, хотя и требует более высокого уровня абстракции, чем вставка HTML-содержимого.

В прикладном интерфейсе API уже существует метод для вставки произвольных строк формата HTML. Он был внедрен в браузере Internet Explorer и теперь включен в спецификацию HTML 5, разработанную консорциумом W3C. Речь идет о методе `insertAdjacentHTML()`, пригодном для обращения ко всем элементам модели HTML DOM (подробнее об этом см. по адресу http://www.w3.org/TR/2011/WD-html5-20110525/apis-in-html-documents.html#insertadjacenthtml). Пользоваться этим методом очень просто, а удобную для усвоения документацию на него можно найти по адресу https://developer.mozilla.org/en-US/docs/DOM/element.insertAdjacentHTML.

Но дело в том, что нельзя особенно полагаться на этот метод из прикладного интерфейса API, организуя поддержку всего ряда браузеров. Несмотря на то что он широко применяется во всех современных браузерах, вряд ли можно рассчитывать на его поддержку в некоторых устаревших браузерах, поскольку он внедрен относительно недавно. Так, его реализация в прежних версиях браузера Internet Explorer страдает многими ошибками и работает только с некоторыми из всех имеющихся элементов разметки. Но даже если бы вы могли позволить себе роскошь поддерживать только самые последние и лучшие версии браузеров, вам все равно не помешало бы знать, каким образом осуществляется вставка HTML-разметки, чтобы пополнить свой арсенал средств настоящего мастера программирования на JavaScript.

По этим причинам нам придется реализовывать прикладной интерфейс API для манипулирования моделью DOM сызнова. Его реализация разделяется на несколько стадий.

1. Преобразование произвольной, но действительной строки формата HTML/XHTML в структуру DOM.
2. Вставка этой структуры DOM в произвольном месте документа как можно более эффективным образом.
3. Выполнение любых встраиваемых сценариев, присутствующих в исходной строке.

В результате выполнения всех этих трех стадий пользователь получает в свое распоряжение изящный прикладной интерфейс API для вставки HTML-содержимого в документ. Итак, приступим к его реализации.

Преобразование из формата HTML в DOM

В преобразовании строки формата HTML в структуру DOM нет ничего таинственного. Для этой цели служит известное инструментальное средство: свойство `innerHTML` элементов модели DOM. Процесс преобразования выполняется в несколько этапов.

- Проверка достоверности HTML/XHTML-содержимого строки формата HTML или хотя бы его подгонка под достоверность.
- Заключение строки в объемлющую разметку, как того требуют правила в браузерах.
- Вставка HTML-строки с помощью свойства `innerHTML` в фиктивный элемент модели DOM.
- Извлечение узлов модели DOM в обратном порядке.

Эти этапы сами по себе не очень сложны, за исключением самой вставки, таящей в себе некоторые скрытые препятствия. Тем не менее они вполне преодолимы. Рассмотрим все эти этапы по очереди.

Предварительная обработка исходной строки формата HTML/XHTML

Прежде всего необходимо привести исходное HTML-содержимое в соответствующий порядок. И выполнение этого первого этапа зависит от конкретных производственных потребностей и контекста. Так, в конструкции библиотеки jQuery очень важна поддержка таких элементов стилевого оформления XML, как "<table/>".

Подобные самозамыкающиеся элементы стилевого оформления XML на самом деле пригодны только для небольшого подмножества элементов HTML-разметки. И всякая попытка использовать этот синтаксис как-то иначе приводит к осложнениям в таких браузерах, как Internet Explorer. Для преобразования таких элементов разметки, как "<table/>", в элементы "<table></table>", единообразно обрабатываемые во всех браузерах, можно произвести быстрый предварительный синтаксический анализ HTML-строки, как показано в листинге 14.1.

Листинг 14.1. Проверка правильности интерпретации самозамыкающихся элементов стилевого оформления

```
<script type="text/javascript">

  var tags =
    /^(abbr|br|col|img|input|link|meta|param|hr|area|embed)$/i;
  function convert(html) {
    return html.replace(/(<(\w+)[^>]*?)\/>/g, function (all, front, tag) {
      return tags.test(tag) ?
          all :
          front + "></" + tag + ">";
    });
  }

  assert(convert("<a/>") === "<a></a>", "Check anchor conversion.");
  assert(convert("<hr/>") === "<hr/>", "Check hr conversion.");

</script>
```

Воспользоваться регулярным выражением для совпадения с именем дескриптора любых элементов, которыми можно пренебречь

Функция, в которой регулярные выражения используются для преобразования самозамыкающихся дескрипторов в обычную форму

Произвести, как всегда, тестирование кода!

По завершении первого этапа необходимо выяснить, требуется ли заключить новые элементы в оболочку.

Заключение HTML-содержимого в оболочку

Итак, у нас имеется теперь исходная HTML-строка, но прежде чем вставлять ее на странице, придется выполнить еще один дополнительный этап. Целый ряд элементов HTML должен быть заключен в некоторый контейнер перед тем, как их вставлять. Например, элемент разметки <option> должен быть заключен в элемент разметки <select>. Эту задачу можно решить двумя путями, и оба требуют составления определенной схемы соответствия проблематичных элементов разметки их контейнерам.

- Строку можно вставить с помощью свойства innerHTML непосредственно в конкретный родительский элемент, построенный ранее методом createElement(). И хотя такой способ вполне подходит в некоторых случаях, его универсальная пригодность для всех браузеров совсем не гарантируется.

- Строку можно заключить в подходящую разметку, а затем вставить непосредственно в любой контейнерный элемент (например, div). Это более надежный, но и трудоемкий способ.

Второй способ предпочтительнее первого. Он требует написания очень малого количества специфического для браузера кода, в отличие от первого способа, который предполагает написание, главным образом, специфического для браузера кода. Проблематичные элементы, которые требуется заключить в конкретные контейнерные элементы, вполне поддаются управлению и могут быть разделены на семь различных групп. В приведенном ниже перечне многоточие (...) обозначает место, где должны быть вставлены элементы.

- Элементы разметки <option> и <optgroup> должны содержаться в блоке <select multiple="multiple">...</select>.
- Элемент разметки <legend> должен содержаться в блоке <fieldset>...</fieldset>.
- Элементы разметки <thead>, <tbody>, <tfoot>, <colgroup> и <caption> должны содержаться в блоке <table>...</table>.
- Элемент разметки <tr> должен содержаться в блоке <table><thead>...</thead></table>, <table><tbody>...</tbody></table> или <table><tfoot>...</tfoot></table>.
- Элементы разметки <td> и <th> должны содержаться в блоке <table><tbody><tr>...</tr></tbody></table>.
- Элемент разметки <col> должен содержаться в блоке <table><tbody></tbody><colgroup>...</colgroup></table>.
- Элементы разметки <link> и <script> должны содержаться в блоке ... .

Приведенная выше схема соответствия не требует особых пояснений, за исключением следующих моментов.

- Вместо элемента немножественного выбора используется элемент разметки <select> с атрибутом multiple, поскольку в нем не производится автоматическая проверка любых вариантов выбора, которые в нем размещены, тогда как в элементе одиночного выбора автоматически проверяется первый вариант выбора.
- В элемент разметки <col> входит дополнительный элемент <tbody>, без которого элемент разметки <colgroup> сформируется неправильно.
- Блок, в который должны быть заключены элементы разметки <link> и <script>, внешне выглядит не совсем обычно. Ведь иначе элементы разметки <link> и <script> не могут быть сформированы в браузере Internet Explorer с помощью свойства innerHTML, если только не соблюдаются следующие условия: оба они содержатся в другом элементе, а также имеется соседний узел.

Итак, приведя элементы в соответствие с требованиями заключить их в оболочку, перейдем к их формированию.

Формирование узлов модели DOM

Приведенная выше схема соответствия элементов разметки их контейнерам предоставляет достаточно информации для формирования HTML-содержимого, которое требуется вставить в элемент модели DOM. В примере кода из листинга 14.2 показано, каким образом на основании этой информации формируются узлы модели DOM.

Листинг 14.2. Формирование узлов модели DOM из исходной разметки

```
<script type="text/javascript">

  function getNodes(htmlString, doc) {

    var map = {
      "<td":[3,"<table><tbody><tr>","</tr></tbody></table>"],
      "<th":[3,"<table><tbody><tr>","</tr></tbody></table>"],
      "<tr":[2,"<table><thead>","</thead></table>"],
      "<option":[1,"<select multiple='multiple'>","</select>"],
      "<optgroup":[1,"<select multiple='multiple'>","</select>"],
      "<legend":[1,"<fieldset>","</fieldset>"],
      "<thead":[1,"<table>","</table>"],
      "<tbody":[1,"<table>","</table>"],
      "<tfoot":[1,"<table>","</table>"],
      "<colgroup":[1,"<table>","</table>"],
      "<caption":[1,"<table>","</table>"],
      "<col":[2,"<table><tbody></tbody><colgroup>","</colgroup></table>"],
      "<link":[3,"<div></div><div>","</div>"]
    };

    var tagName = htmlString.match(/<\w+/),
      mapEntry = tagName ? map[tagName0]] : null;
        if (!mapEntry) mapEntry = [0, " ". " " ];

    var div = (doc || document).createElement("div");

    div.innerHTML = mapEntry[1] + htmlString + mapEntry[2];

    while (mapEntry[0]--) div = div.lastChild;

    return div.childNodes;
  }

  assert(getNodes("<td>test</td><td>test2</td>").length === 2,
        "Get two nodes back from the method.");
  assert(getNodes("<td>test</td>")[0].nodeName === "TD",
        "Verify that we're getting the right node.");

</script>
```

❶ Схема соответствия типов элементов и родительских контейнеров для них. У каждого элемента имеется глубина на один новый узел, открывая и закрывая HTML-разметку для родительских элементов

❷ Воспользоваться регулярным выражением для совпадения с открывающей узловой скобкой и именем дескриптора вставляемого элемента разметки

❸ Если элемент имеется в схеме соответствия, извлечь его, а иначе построить фиктивный элемент с пустой "родительской" разметкой и нулевой глубиной

❹ Создать элемент разметки , а в нем — новые узлы. Для этого используется передаваемый документ, если он существует, а иначе — текущий документ

❺ Заключить входящую разметку в родительские элементы по схеме соответствия и вставить ее в качестве внутреннего HTML-содержимого во вновь созданный элемент разметки

❻ Пройти вновь созданное дерево на глубину, определенную по схеме соответствия. Это должен быть родительский элемент нужного узла, созданного из разметки

❼ Возвратить вновь созданный элемент разметки

Прежде чем возвращать сформированный ряд узлов, необходимо обратить внимание на две программные ошибки в браузере Internet Explorer. Первая из них состоит в том, что браузер Internet Explorer вводит элемент `<tbody>` в пустом элементе `<table>`. Для устранения этой ошибки достаточно проверить, намечается ли создать пустой элемент `<table>`, а затем удалить любые порожденные узлы. А вторая ошибка состоит в том, что в браузере Internet Explorer все начальные пробелы удаляются из символьной строки, передаваемой свойству `innerHTML`. Не следует забывать, что в HTML-разметке пробелы игнорируются, и поэтому они обычно не принимаются браузерами во внимание при воспроизведении документа. Эту ошибку можно исправить, проверив, является ли первый сформированный узел текстовым и содержит ли он начальные пробелы. И если он их не содержит, то необходимо создать новый текстовый узел и заполнить его пробелами явным образом.

Итак, у нас имеются установленные узлы модели DOM. И теперь мы можем приступить к их вставке в документ.

Вставка в документ

Сформировав конкретные узлы модели DOM, мы можем наконец перейти непосредственно к их вставке в документ. Для этого придется предпринять определенные шаги, что и будет сделано далее. У нас уже имеется массив элементов, которые требуется вставить в любых местах документа. Поэтому мы можем попытаться сократить число необходимых для этого операций.

Это можно сделать с помощью *фрагментов* модели DOM — части спецификации модели DOM по стандарту W3C, поддерживаемой во всех браузерах. Это полезное средство предоставляет контейнер для хранения целой коллекции узлов DOM. Помимо того, что такая возможность сама по себе полезна, она дает следующие преимущества: фрагмент можно вставлять и клонировать в единой операции вместо того, чтобы вставлять и клонировать каждый узел по отдельности. Благодаря этому значительно сокращается количество операций, которые требуется выполнить на странице.

Прежде чем воспользоваться этим механизмом в коде, вернемся к коду функции `getNodes()` из листинга 14.2 и отредактируем его немного, чтобы применить в нем фрагменты модели DOM. Эти изменения минимальны и состоят в добавлении параметра `fragment` в список параметров данной функции, как показано ниже:

```
function getNodes(htmlString, doc, fragment) {
```

Если этот параметр передается данной функции, то предполагается, что им должен быть фрагмент модели DOM, в который требуется вставить узлы для последующего применения. Для того чтобы вставить узлы в передаваемый фрагмент модели DOM, достаточно добавить следующий фрагмент кода непосредственно перед оператором возврата из функции:

```
if (fragment) {
  while (div.firstChild) {
    fragment.appendChild(div.firstChild);
  }
}
```

А теперь применим данный механизм на практике. В примере кода, приведенном в листинге 14.3 и взятом из библиотеки jQuery, предполагается, что обновленная функция `getNodes()` находится в области своего действия, а фрагмент создается и передается этой функции для преобразования входящей HTML-строки в элемент модели

DOM. И этот элемент модели DOM автоматически присоединяется к вновь созданному фрагменту.

Листинг 14.3. Вставка фрагмента DOM в нескольких местах документа

```
<div id="test"><b>Hello</b>, I'm a ninja! </div>
<div id="test2"></div>

<script type="text/javascript">

  window.onload = function(){
    function insert(elems, args, callback){
      if ( elems.length ) {
        var doc = elems[0].ownerDocument || elems[0],
          fragment = doc.createDocumentFragment(),
          scripts = getNodes( args, doc, fragment ),
          first = fragment.firstChild;

        if ( first ) {
          for ( var i = 0; elems[i]; i++ ) {
            callback.call( root(elems[i], first),
            i > 0 ? fragment.cloneNode(true) : fragment );
          }
        }
      }
    }

    var divs = document.getElementsByTagName("div");

    insert(divs, ["<b>Name:</b>"], function(fragment){
      this.appendChild( fragment );
    });

    insert(divs, ["<span>First</span> <span>Last</span>"],
    function(fragment){
      this.parentNode.insertBefore( fragment, this );
    });
  };

</script>
```

Следует также обратить внимание на следующее: если данный элемент разметки вставляется в нескольких местах документа, соответствующий фрагмент придется клонировать неоднократно. И если бы не фрагменты, то нам пришлось бы всякий раз клонировать каждый узел в отдельности вместо целого фрагмента сразу. И наконец, следует принять во внимание еще одно, хотя не столь существенное обстоятельство. Когда авторы веб-страниц пытаются вставить строку таблицы непосредственно в элемент разметки `<table>`, они, как правило, предполагают вставить строку прямо в элемент разметки `<tbody>`, находящийся в элементе `<table>`. Для того чтобы учесть данное обстоятельство, можно написать простую функцию приведения в соответствие, как показано в листинге 14.4.

Листинг 14.4. Выявление конкретного места для вставки элемента разметки

```
<script type="text/javascript">

  function root(elem, cur) {
    return elem.nodeName.toLowerCase() === "table" &&
      cur.nodeName.toLowerCase() === "tr" ?
      (elem.getElementsByTagName("tbody")[0] ||
        elem.appendChild(elem.ownerDocument.createElement("tbody"))) :
      elem;
  }

</script>
```

Итак, теперь имеется возможность интуитивно формировать и вставлять произвольные элементы модели DOM. А как насчет элементов разметки сценария, вставляемых в исходную строку? Ответить на этот вопрос мы попытаемся в следующем разделе.

Выполнение сценариев

Помимо самой вставки HTML-содержимого в документ, нередко требуется выполнение встраиваемых элементов разметки сценария. Такая потребность возникает главным образом в тех случаях, когда фрагмент HTML-разметки возвращается с сервера в виде ответа на Ajax-запрос, а кроме него имеется еще и сценарий, который нужно выполнить.

Самый лучший способ обращения со встраиваемыми сценариями обычно состоит в их извлечении из структуры модели DOM перед непосредственной вставкой в документ. Поэтому функция, преобразующая HTML-содержимое в узел модели DOM, должна принять окончательный вид, аналогичный коду из библиотеки jQuery, приведенному в листинге 14.5.

Листинг 14.5. Собирание сценариев

```
for ( var i = 0; ret[i]; i++ ) {
  if ( jQuery.nodeName( ret[i], "script" ) &&
      (!ret[i].type ||
        ret[i].type.toLowerCase() === "text/javascript") ) {
    scripts.push( ret[i].parentNode ?
      ret[i].parentNode.removeChild( ret[i] ) :
      ret[i] );
  } else if ( ret[i].nodeType === 1 ) {
    ret.splice.apply( ret, [i + 1, 0].concat(
      jQuery.makeArray(ret[i].getElementsByTagName("script"))));
  }
}
```

В приведенном выше примере кода используются два массива: `ret` (содержит все сформированные узлы модели DOM) и `scripts` (заполняется всеми сценариями из данного фрагмента по порядку их следования в документе). Кроме того, в этом коде удаляются только те сценарии, которые обычно выполняются как сценарии JavaScript, где тип явно не указывается в атрибуте `type` или задается как `'text/javascript'`.

После вставки структуры модели DOM в документ содержимое массива scripts извлекается и вычисляется. И для этого достаточно сделать некоторые перестановки, не особенно усложняя код. Но после этого наступает самая трудная стадия рассматриваемого здесь процесса.

Вычисление глобального кода

Когда сценарии встраиваются для исполнения, предполагается, что они будут выполняться в глобальном контексте. Это означает, что если в сценарии определена переменная, то она должна стать глобальной. Это же относится и к любым функциям. Стандартные методы для вычисления кода в лучшем случае неустойчивы. Поэтому единственный надежный способ выполнения кода в глобальной области действия состоит в том, чтобы создать новый элемент разметки сценария, вставить в него код, который требуется выполнить по сценарию, а затем быстро вставить и удалить сценарий из документа. Этот прием уже обсуждался в разделе "Механизмы вычисления кода" главы 9. Он вынуждает браузер выполнять внутреннее содержимое элемента разметки сценария в глобальной области действия. В листинге 14.6 приведен фрагмент кода для глобального вычисления сценария из библиотеки jQuery.

Листинг 14.6. Вычисление сценария в глобальной области действия

```
<script type="text/javascript">

  function globalEval(data) {
    data = data.replace(/^\s+|\s+$/g, "");

    if (data) {
      var head = document.getElementsByTagName("head")[0] ||
              document.documentElement,
          script = document.createElement("script");

      script.type = "text/javascript";
      script.text = data;

      head.insertBefore(script, head.firstChild);
      head.removeChild(script);
    }
  }

</script>
```

Применяя подобный прием, нетрудно организовать вычисление кода в элементе разметки сценария. Его можно даже дополнить простым кодом для динамической загрузки сценария, если имеется ссылка на внешний URL, а затем выполнить и этот код, как показано в листинге 14.7.

Листинг 14.7. Вычисление кода сценария, даже если он и удаленный

```
<script type="text/javascript">

  function evalScript(elem) {
    if (elem.src)
      jQuery.ajax({
        url:elem.src,
        async:false,
        dataType:"script"
      });
    else
      jQuery.globalEval(elem.text || "");

    if (elem.parentNode)
      elem.parentNode.removeChild(elem);
  }

</script>
```


Следует иметь в виду, что после вычисления кода сценария он удаляется из модели DOM. То же самое было сделано ранее, когда элемент разметки сценария был удален перед вставкой в документ. Это делается для того, чтобы исключить случайное выполнение сценария повторно (например, когда присоединяемый к документу сценарий вызывается рекурсивно).

Итак, вы пополнили свой арсенал средств программирования на JavaScript новыми приемами ввода элементов в модель DOM. Обсудим далее возможности для копирования новых элементов из уже существующих.

Клонирование элементов разметки

Элементы разметки довольно просто клонируются (с помощью метода cloneNode() из модели DOM) во всех браузерах, кроме Internet Explorer. Если в устаревших версиях браузера Internet Explorer происходят в определенном сочетании три вида поведения, то в итоге создаются не очень благоприятные условия для клонирования элементов разметки.

Во-первых, при клонировании браузер Internet Explorer копирует все обработчики событий в клонируемый элемент разметки. Кроме того, переносятся любые специально расширенные свойства, присоединяемые к элементу. Такое поведение проверяется с помощью простого теста в jQuery, как показано в листинге 14.8.

Листинг 14.8. Выявление факта копирования обработчиков событий в клонируемый элемент разметки

```
<script type="text/javascript">

  var div = document.createElement("div");

  if (div.attachEvent && div.fireEvent) {
    div.attachEvent("onclick", function () {
      jquery.support.noCloneEvent = false;
      div.detachEvent("onclick", arguments.callee);
    });
    div.cloneNode(true).fireEvent("onclick");
  }

</script>
```

При клонировании узла ни один из привязываемых к нему обработчиков событий не должен копироваться, что имеет место в браузере Internet Explorer

Во-вторых, вполне очевидным шагом избежать этого было бы удаление обработчика событий из клонированного элемента разметки. Но любопытно, что если сделать это в браузере Internet Explorer, то обработчик событий будет также удален из исходного элемента. Естественно, что любые попытки удалить специально расширенные свойства из клонируемого элемента приведут к тому, что они будут также удалены из исходного элемента.

И в-третьих, в качестве выхода из этого положения можно попытаться клонировать только сам элемент разметки, вставить его в другой элемент, а затем считать содержимое свойства innerHTML этого элемента и преобразовать его обратно в узел модели DOM. Это многоэтапный процесс, но в результате его выполнения получается чистый клонированный элемент. Правда, придется преодолеть еще одну программную ошибку в браузере Internet Explorer: свойство innerHTML (а по существу, и свойство outerHTML) элемента разметки не всегда правильно отражает состояние его атрибутов. Это особенно проявляется при динамическом изменении атрибутов имен у элементов ввода данных — новое значение атрибута оказывается не представленным в свойстве innerHTML.

Необходимо сделать еще одну оговорку: у элементов модели XML DOM свойство innerHTML отсутствует, и поэтому приходится прибегать к традиционному вызову метода cloneNode(). Правда, приемники событий встречаются в элементах модели XML DOM крайне редко.

Таким образом, для преодоления описанных выше недостатков в работе браузера Internet Explorer приходится действовать окольным путем. Вместо быстрого вызова метода cloneNode() придется выполнить сначала сериализацию содержимого свойства innerHTML, извлекаемого из узла модели DOM, а затем подменить его любыми атрибутами, которые не были перенесены. Степень такой подмены путем так называемых "обезьяньих исправлений" определяется самим разработчиком. Соответствующий пример кода приведен в листинге 14.9.

Листинг 14.9. Фрагмент кода из библиотеки jQuery для клонирования элементов разметки

```
<script type="text/javascript">

  function clone() {
    var ret = this.map(function () {
      if (!jQuery.support.noCloneEvent && !jQuery.isXMLDoc(this)) {
        var clone = this.cloneNode(true),
          container = document.createElement("div");
        container.appendChild(clone);
        return jQuery.clean([container.innerHTML])[0];
      }
      else
        return this.cloneNode(true);
    });

    var clone = ret.find("*").andSelf().each(function () {
      if (this[ expando ] !== undefined)
        this[ expando ] = null;
    });

    return ret;
```



```
  }

</script>
```

Обратите внимание на то, что в приведенном выше примере кода используется метод jQuery.clean() из библиотеки jQuery. Этот метод преобразует HTML-строку в структуру модели DOM, как пояснялось ранее.

Итак, мы выяснили, каким образом новые элементы разметки добавляются или копируются. А теперь обсудим, как избавляться от них.

Удаление элементов разметки

Удаление элемента из модели DOM *должно* быть таким же несложным, как и вызов метода removeChild(), но на самом деле это не так. Для этого придется выполнить немало операций предварительной очистки, прежде чем появится сама возможность удалить элемент из модели DOM. Обычно очистка элемента перед его удалением из модели DOM производится в два этапа.

Прежде всего необходимо очистить элемент разметки от любых привязанных к нему обработчиков событий. Если интегрированная среда спроектирована грамотно, то она должна привязывать обработчики событий к элементу поочередно, а следовательно, очистка от них элемента разметки должна быть не сложнее простого удаления конкретной функции обработчика событий. Именно таким образом и была построена система обработки событий в главе 13. Данный этап очень важен, поскольку браузер Internet Explorer будет отбирать лишние ресурсы памяти, если удаляемая функция обработки событий ссылается на любые элементы модели DOM.

Второй этап очистки состоит в удалении любых внешних данных, связанных с элементом разметки. Как пояснялось в главе 13, системе обработки событий требуется надежный способ для связывания фрагментов данных с элементом разметки, не присоединяя данные непосредственно в качестве расширяемого свойства. Эти данные лучше очистить, чтобы они не занимали лишнюю память.

Оба упомянутых выше этапа необходимо выполнить по отношению к удаляемому элементу разметки и всем его порожденным элементам. Ведь все порожденные элементы также удаляются, хотя это и не совсем очевидно. В качестве примера в листинге 14.10 приведен соответствующий код из библиотеки jQuery.

Листинг 14.10. Функция из библиотеки jQuery для удаления элемента разметки

```
<script type="text/javascript">

  function remove() {

    jQuery("*", this).add([this]).each(function () {

      jQuery.event.remove(this);

      jQuery.removeData(this);
    });

    if (this.parentNode)
      this.parentNode.removeChild(this);
```

Обойти дерево, найдя все порожденные и удаляемые элементы

Удалить все привязанные элементы

Удалить присоединенные данные

Удалить элемент, если он отсутствует в модели DOM

```
  }

</script>
```

После предварительной очистки можно приступать непосредственно к удалению элемента из модели DOM. Большинство браузеров отлично справляются с задачей удаления элемента разметки со страницы, за исключением браузера Internet Explorer, как пояснялось ранее. При удалении каждого элемента со страницы в отдельности занимаемая им память не освобождается до тех пор, пока не будет оставлена сама страница. А это означает, что долговременные страницы, на которых удаляется немало элементов, занимают намного больше памяти в браузере Internet Explorer, чем требуется.

Существует одно вполне работоспособное, хотя и частичное разрешение данного затруднения. У браузера Internet Explorer имеется собственное свойство `outerHTML`, которое предоставляет строковое HTML-представление элемента разметки. Как бы там ни было, но свойство `outerHTML` выполняет роль не только получателя, но и установщика, Так, если выполнить следующую строку кода:

```
outerHTML = "";
```

то элемент разметки будет полностью удален из памяти браузера Internet Explorer вместо вызова метода `removeChild()`. И такой шаг можно сделать в дополнение к вызову метода `removeChild()`. В листинге 14.11 приведен соответствующий пример кода.

Листинг 14.11. Установка свойства `outerHTML` с целью освободить побольше памяти в браузере Internet Explorer

```
if (this.parentNode)
  this.parentNode.removeChild(this);

if (typeof this.outerHTML !== "undefined")
  this.outerHTML = "";
```

Удалить элемент разметки, если он присутствует в модели DOM

Следует иметь в виду, что данный способ не гарантирует *полное* освобождение памяти, занимаемой удаляемым элементом разметки. Но можно с уверенностью сказать, что он позволяет освободить значительно больше памяти, чем это удается сделать обычным средствами. А это уже неплохо хотя бы для начала.

Не следует, однако, забывать, что всякий раз, когда элемент разметки удаляется со страницы, приходится проходить все три описанных выше этапа данного процесса. К ним относится очистка содержимого удаляемого элемента, замена содержимого этого элемента текстом или HTML-содержимым или же непосредственная замена самого элемента. Для того чтобы избежать дополнительных осложнений при использовании памяти, модель DOM следует поддерживать в чистоте и порядке.

Итак, мы рассмотрели все, что касается манипулирования элементами разметки в коде HTML. Но ведь веб-страница состоит не только из этих элементов. На ней имеется также текст. Именно о нем и пойдет речь в следующем разделе.

Текстовое содержимое

Обращаться с текстом намного проще, чем с элементами разметки в коде HTML, особенно при наличии соответствующих встроенных методов, надежно работающих во всех браузерах. Но и здесь не обходится без программных ошибок в браузерах, которые нужно как-то преодолевать, что делает все эти методы и соответствующие прикладные интерфейсы API ненужными и устаревшими по ходу разработки приложений. Как правило, обработка текстового содержимого выполняется по двум распространенным сценариям.

- Извлечение текстового содержимого из элемента разметки.
- Установка текстового содержимого элемента разметки.

В браузерах, совместимых со стандартами консорциума W3C, предоставляется свойство `textContent` для элементов модели DOM. Доступ к содержимому этого свойства позволяет получить текстовое содержимое элемента, а также его непосредственных потомков и порожденных узлов. А в устаревших версиях браузера Internet Explorer имеется свое собственное свойство `innerText`, выполняющее ту же самую роль, что и свойство `textContent`. (На всякий случай в некоторых браузерах типа WebKit также поддерживается свойство `textContent`.) В листинге 14.12 приведены примеры применения обоих свойств в прикладном коде.

Листинг 14.12. Применение свойств `.innerText` и `.textContent`

```
<div id="test"><b>Hello</b>, I'm a ninja!</div>

<script type="text/javascript">

  window.onload = function () {
    var b = document.getElementById("test");
    var text = b.textContent || b.innerText;

    assert(text === "Hello, I'm a ninja!",
          "Examine the text contents of an element.");
    assert(b.childNodes.length === 2,
          "An element and a text node exist.");

    if (typeof b.textContent !== "undefined") {
      b.textContent = "Some new text";
    }
    else {
      b.innerText = "Some new text";
    }

    text = b.textContent || b.innerText;

    assert(text === "Some new text", "Set a new text value.");
    assert(b.childNodes.length === 1,
          "Only one text node exists now.");
  };

</script>
```

Следует иметь в виду, что при установке свойств `innerText` и `textContent` исходная структура элемента разметки удаляется. Поэтому применение этих свойств таит в себе определенный ряд скрытых препятствий, несмотря на всю их полезность. Прежде всего это связано с отсутствием гарантий восстановления памяти после удале-

ния элемента разметки, как уже обсуждалось ранее. Кроме того, кросс-браузерная обработка пробелов в этих свойствах никуда не годится. И к тому же оказывается, что ни один из браузеров неспособен возвращать согласованные результаты.

Поэтому если сохранение пробелов (особенно конечных) не имеет особого значения, то свойствами `innerText` и `textContent` можно пользоваться без всяких ограничений для доступа к текстовому значению элемента разметки. А для установки такого значения придется найти альтернативное решение.

Установка текста

Установка текстового значения разделяется на два этапа.

- Освобождение содержимого элемента разметки.
- Вставка на его место нового текстового содержимого.

Освобождение содержимого осуществляется достаточно просто, как следует из примера кода, приведенного в листинге 14.10. А для вставки нового текстового содержимого каждую вставляемую текстовую строку придется экранировать надлежащим образом. Вставка HTML-содержимого отличается от вставки текста тем, что в тексте может присутствовать ряд символов, которые имеют специальное назначение в HTML-разметке, и поэтому они должны быть непременно экранированы. Без этого символ `'<'` в HTML-разметке превратится, например, в символы `'<'`.

Правда, для решения этой задачи можно воспользоваться встроенным методом `createTextNode()`, пригодным для обращения к DOM-документам, как показано в листинге 14.13.

Листинг 14.13. Установка текстового содержимого документа

```
<div id="test"><b>Hello</b>, I'm a ninja!</div>

<script type="text/javascript">
  window.onload = function () {
    var b = document.getElementById("test");

    while (b.firstChild)
      b.removeChild(b.firstChild);

    b.appendChild(document.createTextNode("Some new text"));

    var text = b.textContent || b.innerText;

    assert(text === "Some new text", "Set a new text value.");
    assert(b.childNodes.length === 1,
           "Only one text nodes exists now.");
  };

</script>
```

Вызов этого метода можно заменить вызовом избранного метода `empty()`

Вставить экранированный текстовый узел

Итак, мы рассмотрели установку текста. А теперь перейдем к получению текста.

Получение текста

Для получения *точного* текстового значения из элемента разметки придется игнорировать результаты, полученные из свойства `innerText` или `textContent`. Самое распространенное затруднение связано с удалением лишних конечных пробелов из результата, возвращаемого из этих свойств. Поэтому для получения точного результата необходимо выбрать все значения из текстового узла вручную. В листинге 14.14 приведено одно из возможных решений данной задачи с помощью рекурсии.

Листинг 14.14. Получение текстового содержимого элемента разметки

```
<div id="test"><b>Hello</b>, I'm a ninja!</div>

<script>

  window.onload = function () {
    function getText(elem) {
      var text = "";

      for (var i = 0; i < elem.childNodes.length; i++) {
        var cur = elem.childNodes[i];

        if (cur.nodeType === 3)
          text += cur.nodeValue;

        else if (cur.nodeType === 1)
          text += getText(cur);
      }
      return text;
    }

    var b = document.getElementById("test");
    var text = getText(b);

    assert(text === "Hello, I'm a ninja!",
           "Examine the text contents of an element.");
    assert(b.childNodes.length === 2,
           "An element and a text node exist.");
  };

</script>
```

Тип текстовых узлов равен 3

Если это элемент, то рекурсию следует продолжить

Если в разрабатываемых веб-приложениях можно обойтись без удаления лишних пробелов, то рекомендуется пользоваться свойствами `innerText` или `textContent`, так как они намного упрощают процесс разработки. Но было бы неплохо предусмотреть и запасной вариант на тот случай, если эти свойства не подойдут.

Резюме

В этой главе был проведен всесторонний анализ наилучших путей решения трудных задач, связанных с манипулированием моделью DOM. И хотя в современных браузерах

предоставляются новые возможности для манипулирования моделью DOM, умение делать это вручную имеет решающее значение для обеспечения поддержки устаревших браузеров и повышения производительности.

Несмотря на то что решение задач манипулирования моделью DOM может оказаться более простым, чем представляется на первый взгляд, возникающие при этом трудности кросс-браузерного характера способны существенно усложнить конкретную реализацию в коде. Но приложив немного усилий, все-таки можно прийти к единому решению, подходящему для большинства браузеров. Именно к этому и следует всегда стремиться.

В этой главе были рассмотрены следующие вопросы:

- Применение регулярных выражений, подробно рассмотренных в главе 7, дает возможность привести фрагмент HTML-разметки к правильно сформированным синтаксическим конструкциям, которые можно проанализировать.
- Вставка фрагмента HTML-разметки в свойство `innerHTML` временного элемента представляет собой простой и быстрый способ преобразования символьной строки с HTML-разметкой в элементы модели DOM.
- Некоторые элементы разметки, например табличные, приходится заключать в какие-то другие контейнерные элементы, чтобы создавать их надлежащим образом.
- Элементы сценариев во фрагментах HTML-разметки могут выполняться в глобальной области действия с помощью способов, рассмотренных в главе 9, посвященной вычислению кода.
- В устаревших версиях браузера Internet Explorer возникают серьезные трудности при клонировании узлов, поскольку они копируют много *лишнего*, включая обработчики событий и расширенные свойства.
- Удаляя элементы из модели DOM, следует благоразумно управлять памятью, особенно при создании долговременных веб-страниц.

В этой главе было показано, каким образом создаются, клонируются и удаляются элементы разметки. А как их находить? В следующей главе мы обсудим последний предмет обучения мастера программирования на JavaScript: обнаружение элементов разметки с помощью CSS-селекторов.

15 Механизмы CSS-селекторов

В этой главе...

- Поддержка CSS-селекторов в современных браузерах
- Стратегии построения механизма селекторов
- Применение прикладного интерфейса API по стандарту W3C
- Краткие сведения о языке запросов XPath
- Построение механизма DOM-селекторов

Приятно осознавать, что профессиональные разработчики веб-приложений наконец-то дождались появления во всех браузерах прикладного интерфейса Selectors API, разработанного консорциумом W3C. Этот прикладной интерфейс имеет два уровня (Level 1 и Level 2) и предоставляет методы `querySelectorAll()` и `querySelector()` наряду с другими полезными средствами для организации в прикладном коде очень быстрого обхода элементов модели DOM в более или менее кросс-браузерном виде.

Примечание

Подробнее об этом прикладном интерфейсе API можно узнать, перейдя на страницы веб-сайта консорциума W3C по следующим адресам: для уровня Level 1 — `www.w3.org/TR/selectors-api/`, для уровня Level 2 — `www.w3.org/TR/selectors-api2/`.

В связи с этим возникает вполне закономерный вопрос: если прикладной интерфейс Selectors API по стандарту консорциума W3C внедрен практически во всех браузерах, то зачем вообще обсуждать вопросы реализации механизма CSS-селекторов в чистом виде на JavaScript?

Разумеется, внедрение стандартного прикладного интерфейса API — дело благое, но в то же время реализация прикладного интерфейса Selectors API по стандарту W3C в большинстве браузеров (по крайней мере, по состоянию на вторую половину 2012 года) вынудила привести в соответствие со стандартами JavaScript/DOM уже существующие внутренние механизмы CSS-селекторов. А для этого пришлось отказаться от целого ряда приятных мелочей, с которыми обычно связывается понятие хорошего прикладного интерфейса API. Так, в методах нельзя воспользоваться уже построенными кешами модели DOM, они не выводят подробные сообщения об ошибках и не в состоянии обеспечивать расширяемость в какой-либо форме. Впрочем, все эти факторы учтены при реализации механизмов CSS-селекторов в распространенных библиотеках JavaScript. Поэтому для повышения производительности в них применяются кеши модели DOM, обеспечиваются дополнительные уровни уведомления об ошибках и высокая степень расширяемости.

Совет

Под пышным термином *механизм CSS-селекторов* обычно подразумеваются функциональные средства, приводящие ряд элементов модели DOM в соответствие с заданным выражением CSS-селектора. Например, все элементы, содержащие класс `ninja`, могут быть собраны с помощью выражения селектора `.ninja`.

Несмотря на все сказанное выше, по-прежнему возникает резонный вопрос: зачем разбираться в принципе действия механизма CSS-селекторов, реализуемого в чистом виде на JavaScript? Дело в том, что ясное представление о принципе действия такого механизма позволяет добиться поразительных успехов в повышении производительности. Это дает возможность не только реализовывать в прикладном коде более совершенные алгоритмы обхода дерева модели DOM для ускорения поиска нужных элементов, но и научиться приспосабливать CSS-селекторы к особенностям работы их механизмов, делая эти селекторы еще более производительными.

Механизмы CSS-селекторов являются неотъемлемой частью современной разработки веб-приложений, и поэтому ясное представление о том, как ускорить их работу, служит важным подспорьем в процессе разработки. Ведь в конечном счете все, что требуется реализовать в страничных сценариях, зачастую приходится делать по следующему образцу.

1. Найти элементы в модели DOM.
2. Сделать что-нибудь с ними или же с их помощью.

За исключением нового прикладного интерфейса Selectors API, поиск элементов в модели никогда не был сильной стороной JavaScript в браузерах. Методы, доступные для обнаружения элементов разметки, обладали весьма ограниченными средствами для поиска этих элементов по значениям идентификаторов и именам дескрипторов. Поэтому все, что удастся сделать для упрощения первой стадии поиска элементов, позволит сосредоточить основное внимание на более интересной второй стадии операции над найденными элементами. В современных механизмах реализуются CSS3-селекторы, как определено в стандарте консорциума W3C и поясняется на соответствующей веб-странице по адресу `www.w3.org/TR/css3-selectors/`.

Существуют три основных средства реализации механизма CSS-селекторов.

1. Прикладной интерфейс Selectors API по стандарту W3C, реализованный в большинстве браузеров.
2. Язык XPath запросов к элементам модели DOM, встроенный в большинство современных браузеров.
3. Чистая модель DOM. Этот оплот механизмов CSS-селекторов допускает постепенное сокращение функциональных возможностей в отсутствие двух других средств.

Каждое из этих средств и соответствующие стратегии будут подробно рассмотрены в этой главе. Это позволит принять взвешенное решение относительно реализации механизма CSS-селекторов в JavaScript или хотя бы понять принцип его действия. И начнем мы с рассмотрения прикладного интерфейса Selectors API по стандарту W3C.

Прикладной интерфейс Selectors API по стандарту W3C

Интерфейс Selectors API по стандарту W3C относится к сравнительно новым видам прикладных программных интерфейсов API и предназначен для упрощения той работы, которая требуется для реализации полноценного механизма CSS-селекторов во всех основных браузерах, включая Safari 3, Firefox 3.1, Internet Explorer 8, Chrome (с первой же версии) и Opera 10. В реализациях этого интерфейса обычно поддерживаются все селекторы, реализуемые в CSS-механизме браузера. Поэтому если в браузере полностью поддерживается стандарт CSS 3, то это обстоятельство непременно будет отражено и в самой реализации прикладного интерфейса Selectors API.

В этом прикладном интерфейсе API предоставляется целый ряд полезных методов, два из которых реализованы в современных браузерах и перечислены ниже.

- Метод `querySelector()`. Принимает строку CSS-селектора и возвращает первый найденный элемент или же пустое значение (`null`), если искомый элемент не найден.
- Метод `querySelectorAll()`. Принимает строку CSS-селектора и возвращает статический объект типа `NodeList` со всеми элементами, обнаруживаемыми селектором.

Оба метода пригодны для обращения ко всем элементам, документам и фрагментам модели DOM. В листинге 15.1 приведены два примера применения этих методов из прикладного интерфейса Selectors API.

Листинг 15.1. Применение прикладного интерфейса Selectors API в прикладном коде

```
<div id="test">
  <b>Hello</b>, I'm a ninja!
</div>
<div id="test2"></div>

<script type="text/javascript">

  window.onload = function () {
    var divs = document.querySelectorAll("body > div");
    assert(divs.length === 2, "Two divs found using a CSS selector.");
```

Найти элементы разметки , порожденные от элемента разметки <body>

```
    var b = document.getElementById("test")
        .querySelector("b:only-child");
    assert(b,
          "The bold element was found relative to another element.");
  };

</script>
```

Найти только порожденные элементы разметки полужирным!

Единственное скрытое препятствие, стоящее на пути применения в настоящее время прикладного интерфейса Selectors API по стандарту W3C состоит в том, что оно ограничивается поддержкой CSS-селекторов в браузерах, а не в различных реализациях, которые были сначала осуществлены в библиотеках JavaScript. Это наглядно показывает приведенный в листинге 15.2 пример реализации правил сопоставления с запросами, укорененными в элементах разметки, где метод `querySelector()` или `querySelectorAll()` вызывается относительно элемента разметки.

Листинг 15.2. Запросы, укорененные в элементах разметки

```
div id="test">
  <b>Hello</b>, I'm a ninja!
</div>

<script type="text/javascript">
  window.onload = function () {
    var b = document.getElementById("test").querySelector("div b");
    assert(b, "Only the last part of the selector matters.");
  };

</script>
```

Следует заметить, что при выполнении укорененного в элементе запроса селектор проверяет только наличие своей завершающей части в элементе разметки. Такой подход может показаться не совсем логичным. Ведь если проанализировать код из листинга 15.2, то можно заметить, что в нем отсутствуют элементы разметки `<div>` с атрибутом `id = test`, хотя именно это и пытается проверить селектор.

Но поскольку большинство пользователей ожидают совсем иного от механизма CSS-селекторов, придется прибегнуть к обходному приему. А состоит он в том, чтобы добавить новый атрибут `id` в укорененный элемент разметки и тем самым соблюсти его контекст, как показано в листинге 15.3.

Листинг 15.3. Соблюдение контекста укорененного элемента

```
<div id="test">
  <b>Hello</b>, I'm a ninja!
</div>

<script type="text/javascript">
  (function() {

    var count = 1;

    this.rootedQuerySelectorAll = function (elem, query) {
```

Привязать переменную count к функции rootedQuerySelectorAll() в немедленно вызываемой функции

Определить функцию в глобальном контексте


```
      var oldID = elem.id;
      elem.id = "rooted" + (count++);

      try {
        return elem.querySelectorAll("#" + elem.id + " " + query);
      }
      catch (e) {
        throw e;
      }
      finally {
        elem.id = oldID;
      }
    };
  })();

  window.onload = function () {
    var b = rootedQuerySelectorAll(
        document.getElementById("test"), "div b");
    assert(b.length === 0, "The selector is now rooted properly.");
  };

</script>
```

Запомнить исходный идентификатор, чтобы восстановить его в дальнейшем

Присвоить однозначно формируемое значение временного идентификатора

Восстановить исходный идентификатор в блоке `finally`, чтобы исключить его изменение обходным путем

В приведенном выше примере кода следует обратить внимание на два важных момента. Сначала элементу необходимо присвоить однозначный идентификатор и восстановить старый, чтобы исключить любые конфликты в конечном результате при построении селектора. А затем данный идентификатор указывается перед селектором (в форме `"#id "`, где `id` – однозначно формируемое значение).

Как правило, достаточно было бы удалить идентификатор `id` и возвратить результат из запроса, но дело в том, что методы из интерфейса Selectors API способны генерировать исключения, которые чаще всего связаны с синтаксисом селекторов или неподдерживаемыми селекторами. В силу этого операция, выполняемая селектором, заключается в блоке `try/catch`. Но поскольку идентификатор `id` требуется восстановить, то этот блок можно дополнить блоком `finally`. И здесь проявляется следующая весьма любопытная особенность языка: будет ли в операторе `try` возвращаться значение или же генерироваться исключение в операторе `catch`, код в операторе `finally` будет всегда выполняться после любого из этих операторов, но перед возвратом значения из функции. Подобным образом можно всегда проверить правильность восстановления идентификатора.

Интерфейс Selectors API относится к числу самых многообещающих прикладных интерфейсов API, появившихся за последнее время. У него достаточно потенциальных возможностей, чтобы заменить большую часть библиотек JavaScript простым методом. Но это станет возможным лишь после того, как поддерживающие его браузеры займут господствующее положение на рынке и будут полностью (или хотя бы по большей части) соответствовать стандарту CSS3. А теперь обратимся к другому способу реализации механизма CSS-селекторов, ориентированного в большей степени на языковые средства XML.

Применение XPath для поиска элементов

Единой альтернативой применению интерфейса Selectors API в тех браузерах, где он не поддерживается, служит язык запросов XPath, предназначенный для поиска узлов в DOM-документе. Это намного более эффективное средство, чем традиционные CSS-селекторы. В большинстве современных браузеров (Firefox, Safari 3+, Opera 9+, Chrome) предоставляются реализованные в той или иной мере средства языка XPath для работы с HTML-ориентированными DOM-документами. А в браузере Explorer, начиная с версии 6, языковые средства XPath поддерживаются для работы с XML-документами, но не с HTML-документами, хотя именно это чаще всего и требуется.

Характерной особенностью выражений XPath является быстрота их выполнения, несмотря на их сложность. При реализации механизма селекторов с помощью чистой модели DOM постоянно возникает противоречие в связи с тем, что браузер способен масштабировать все операции в сценарии JavaScript и модели DOM. Но в то же время выражения XPath не отличаются особой простотой.

Существует некоторый, хотя и не совсем определенный предел, при котором предпочтение следует отдавать не выражениям XPath, а операциям с чистой моделью DOM. И если такой предел может быть выявлен программно, то оказывается, что поиск элементов разметки по атрибуту `id` и простым дескрипторным селекторам (`<div>`) всегда выполняется быстрее в коде чистой модели DOM, в том числе с помощью методов `getElementById()` и `getElementsByTagName()`. Но освоив выражения XPath и найдя их удобными в применении, хотя и только в современных браузерах, где они поддерживаются, можно просто воспользоваться методом, приведенным в листинге 15.4 из библиотеки Prototype, и полностью пренебречь всем остальным, что касается построения механизма CSS-селекторов.

Листинг 15.4. Метод из библиотеки Prototype, выполняющий выражение XPath для обращения к HTML-документу

```
if ( typeof document.evaluate === "function" ) {
  function getElementsByXPath(expression, parentElement) {
    var results = [];
    var query = document.evaluate(expression,
      parentElement || document,
      null, XPathResult.ORDERED_NODE_SNAPSHOT_TYPE, null);
    for (var i = 0, length = query.snapshotLength; i < length; i++)
      results.push(query.snapshotItem(i));
    return results;
  }
}
```

Конечно, было бы удобно пользоваться только языковыми средствами XPath повсеместно, но, к сожалению, это просто нереально. Несмотря на все свои функциональные возможности и ориентированность на разработчиков, язык XPath слишком сложен по сравнению с CSS-селекторами для составления выражений. Здесь недостаточно места для раскрытия всех языковых средств XPath, поэтому далее будут рассмотрены лишь наиболее типичные выражения и их соответствие CSS-селекторам, как показано в табл. 15.1.

Таблица 11.1. Соответствие CSS-селекторов и выражений XPath

Цель поиска	XPath	CSS 3
Все элементы	`//*`	`*`
Все родительские элементы	`//p`	`p`
Все порожденные элементы	`//p/*`	`p > *`
Элемент по идентификатору	`//*[@id='foo']`	`#foo`
Элемент по классу	`//*[contains(concat(" ", @class, ""), " foo ")]`	`.foo`
Элемент с атрибутом	`//*[@title]`	`*[title]`
Первый потомок всех родителей	`//p/*[0]`	`p > *:first-child`
Все родители потомка	`//p[a]`	Невозможно
Следующий элемент	`//p/following-sibling::*[0]`	`p + *`

Применение выражений XPath аналогично построению механизма селекторов на основе чистой модели DOM, где синтаксический анализ селекторов выполняется с помощью регулярных выражений, но за одним существенным исключением: получающиеся в результате части CSS-селектора должны быть преобразованы в соответствующие выражения XPath и затем выполнены.

Но добиться этого не так-то просто, поскольку код в итоге получается таким же крупным, как и при обычной реализации механизма CSS-селекторов с помощью чистой модели DOM. Поэтому многие разработчики просто отказываются от реализации механизма селекторов языковыми средствами XPath, чтобы сделать менее сложным получающийся в итоге механизм. В этой связи целесообразно взвесить преимущества механизма селекторов, построенного на основе XPath, в отношении производительности, принимая особенно во внимание конкурентные возможности интерфейса Selectors API, и тот размер кода, который из этого вытекает. А теперь перейдем к самому трудоемкому способу построения механизма CSS-селекторов.

Реализация чистой модели DOM

В основу каждого механизма CSS-селекторов положена реализация чистой модели DOM. Попросту говоря, она заключается в синтаксическом анализе CSS-селекторов и применении существующих методов из модели DOM, например, `getElementById()` или `getElementsByTagName()`, для поиска соответствующих элементов.

Совет

К числу методов, доступных по спецификации HTML5, добавлен метод `getElementsByClassName()`.

Наличие реализации чистой модели DOM для построения механизма CSS-селекторов имеет значение по ряду следующих причин.

1. **Версии браузера Explorer 6 и 7.** Если в версии Internet Explorer 8 имеется поддержка метода `querySelectorAll()`, то в отсутствие поддержки XPath и прикладного интерфейса Selectors API в версиях 6 и 7 реализация чистой модели DOM становится просто необходимой.
2. **Обратная совместимость.** Если требуется постепенное сокращение функциональных возможностей в коде ради сохранения работоспособности в тех браузерах, где отсутствует поддержка прикладного интерфейса Selectors API или языка XPath, как, например, в Safari 2, то для этой цели придется реализовать чистую модель DOM в той или иной форме.
3. **Быстродействие.** Реализация чистой модели DOM повышает быстродействие целого ряда селекторов, например, при поиске элементов по идентификатору.
4. **Полный охват.** Не все браузеры поддерживают одни и те же CSS3-селекторы. Если же требуется поддержка полного (или хотя бы самого общего) набора селекторов во всех браузерах, то такую поддержку придется реализовать вручную, засучив рукава.

Принимая во внимание приведенные выше причины, рассмотрим два возможных способа реализации механизма CSS-селекторов: нисходящий (сверху вниз) и восходящий (снизу вверх).

В нисходящем механизме синтаксический анализ CSS-селектора производится слева направо последовательным сопоставлением элементов документа с каждым дополнительным сегментом селектора. Такой механизм можно обнаружить в большинстве современных библиотек JavaScript, и, как правило, он является наиболее предпочтительным средством для поиска элементов на странице.

Рассмотрим следующий простой пример разметки:

```
<body>
<div></div>
<div class="ninja">
  <span>Please </span><a href="/ninja"><span>Click me!</span></a>
</div>
</body>
```

Если требуется выбрать элемент разметки `<span>`, содержащий текст `"Click me!"` (Щелкни на мне!), то для этого можно воспользоваться приведенным ниже селектором.

```
div.ninja a span
```

Нисходящий способ применения этого селектора в модели DOM наглядно представлен на рис. 15.1.

В первом члене выражения, `div.ninja`, селектора выявляется поддерево в пределах документа ❶. В этом поддереве применяется следующий член, `a`, чтобы выявить поддерево, укорененное в элементе привязки ❷. И наконец, в члене `span` выявляется целевой узел ❸. Следует, однако, иметь в виду, что данный пример сильно упрощен. В действительности на каждой стадии можно выявить не одно, а множество поддеревьев.

При разработке нисходящего механизма селекторов необходимо принимать во внимание следующее.

- Результаты должны быть представлены в том порядке, в каком они получены из документа.

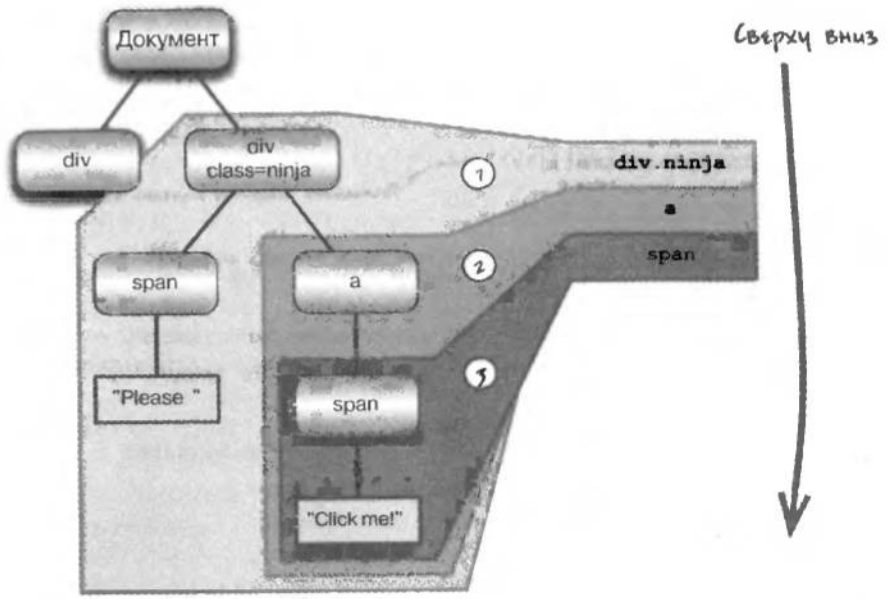

Рис. 15.1. Нисходящие механизмы селекторов действуют, начиная с вершины документа и обнаруживая поддеревья, совпадающие с членами в выражении селектора

- Результаты должны быть однозначными (дублирование возвращаемых элементов не допускается).

В силу этих ограничений разработка нисходящего механизма селекторов может быть сильно затруднена. В качестве примера рассмотрим упрощенный вариант реализации нисходящего механизма селекторов, ограничившись поиском элементов разметки по именам их дескрипторов, как показано в листинге 15.5.

Листинг 15.5. Упрощенный вариант нисходящего механизма селекторов

```
<div>
  <div>
    <span>Span</span>
  </div>
</div>

<script type="text/javascript">

  window.onload = function(){
    function find(selector, root){

      root = root || document;

      var parts = selector.split(" "),
          query = parts[0],
          rest = parts.slice(1).join(" "),
          elems = root.getElementsByTagName(query),
          results = [];
```

Начать сверху документа, если корень не предоставляется

Разделить селектор пробелами, извлечь первый член, собрать остаток, найти элемент, совпадающий с первым членом, а затем инициализировать массив, чтобы сохранить в нем результаты

```
    for (var i = 0; i < elems.length; i++) {
      if (rest) {
        results = results.concat(find(rest, elems[i]));
      }
      else {
        results.push(elems[i]);
      }
    }
    return results;
  };

  var divs = find("div");
  assert(divs.length === 2, "Correct number of divs found.");

  var divs = find("div", document.body);
  assert(divs.length === 2,
        "Correct number of divs found in body.");

  var divs = find("body div");
  assert(divs.length === 2,
        "Correct number of divs found in body.");

  var spans = find("div span");
  assert(spans.length === 2, "A duplicate span was found.");
};

</script>
```

Вызвать метод find() рекурсивно до тех пор, пока не будут употреблены все селекторы

Разместить найденные элементы в массиве results

Возвратить список совпавших элементов

В коде из приведенного выше примера реализован простой нисходящий механизм селекторов, способный обнаруживать элементы только по имени дескриптора. Этот механизм можно разделить на несколько частей по выполняемым функциям: синтаксический анализ селектора, поиск элементов, фильтрация, рекурсирование и объединение результатов. Рассмотрим каждую из этих частей по очереди.

Синтаксический анализ селектора

В рассматриваемом здесь простом примере синтаксический анализ сводится к преобразованию обыкновенного CSS-селектора, состоящего из имен дескрипторов (`"div span"`), в массив строк (`["div", "span"]`). В стандарты CSS 2 и 3 была внедрена возможность находить элементы по атрибуту или значению атрибута, а это позволяет вводить дополнительные пробелы в большинстве селекторов, что заметно упрощает разбиение селектора на части при его синтаксическом анализе.

Для полноценной реализации механизма селекторов потребуется целый ряд правил синтаксического анализа, чтобы обрабатывать любые выражения, которые могут встретиться, — чаще всего в форме регулярных выражений. В листинге 15.6 приведен пример кода, в котором демонстрируется более надежный вариант реализации синтаксического анализатора с помощью регулярного выражения. Такой анализатор способен разбивать селектор на отдельные части и фиксировать их, разделяя запятыми по мере необходимости.

Листинг 15.6. Разбиение CSS-селектора на части с помощью регулярного выражения

```
<script type="text/javascript">

  var selector = "div.class > span:not(:first-child) a[href]"

  var chunker = /((?:\([^\)]+\)|\[[^\]]+\]|[^ ,\(\[]+)+)(\s*,\s*)?/g;

  var parts = [];

  chunker.lastIndex = 0;

  while ((m = chunker.exec(selector)) !== null) {
    parts.push(m[1]);

    if (m[2]) {
      extra = RegExp.rightContext;
      break;
    }
  }

  assert(parts.length == 4,
        "Our selector is broken into 4 unique parts.");
  assert(parts[0] === "div.class", "div selector");
  assert(parts[1] === ">", "child selector");
  assert(parts[2] === "span:not(:first-child)", "span selector");
  assert(parts[3] === "a[href]", "a selector");

</script>
```

Установить в исходное положение регулярное выражение, разбивающее селектор на части

Собрать селектор по частям

Остановиться, когда встретится запятая

Очевидно, что разбиение селектора на части — это лишь полдела. Для каждого вида поддерживаемого выражения потребуются дополнительные правила синтаксического анализа. В большинстве реальных механизмов селекторов содержится таблица соответствия регулярных выражений и функций. Когда происходит совпадение с частью селектора, вызывается на выполнение соответствующая функция. Здесь недостаточно места для подробного рассмотрения подобных выражений. Если вас действительно интересует их построение, рекомендуем внимательно проанализировать ту часть исходного кода из библиотеки jQuery или другой избранной вами библиотеки, где реализуется синтаксический анализ селекторов.

Далее требуется найти элементы разметки, совпадающие с синтаксически проанализированным выражением. Эта стадия работы механизма селекторов рассматривается в следующем разделе.

Поиск элементов разметки

Поиск нужных элементов разметки на странице — это задача, которую можно решать по-разному. Конкретное решение и применяемые приемы во многом зависят от того, какие именно селекторы поддерживаются и какие возможности имеются для этого в браузере. Но между этими условиями существует немало очевидных взаимосвязей.

Рассмотрим метод `getElementById()`. Он доступен только для корневого узла HTML-документов, обнаруживая на странице первый элемент разметки с указанным

идентификатором, и удобен для поиска по идентификатору с помощью CSS-селектора типа "#id". А в браузерах Internet Explorer и Opera этот метод обнаруживает на странице первый элемент с именем, аналогичным указанному в параметре name. Если же элементы требуется найти только по идентификатору id, то придется произвести дополнительную проверку, чтобы исключить из результатов лишние элементы разметки, услужливо найденные этим методом.

Если же требуется найти *все* элементы разметки, совпадающие с конкретным идентификатором id, что характерно для CSS-селекторов, хотя в HTML-документах допускается только один идентификатор id на страницу, то придется обойти все элементы разметки, чтобы найти среди них имеющие нужный идентификатор id. С другой стороны, можно воспользоваться выражением document.all["id"], возвращающим массив всех элементов, совпадающих с идентификатором id в тех браузерах, где это свойство поддерживается, а именно в Internet Explorer, Opera и Safari.

Метод getElementsByTagName() выполняет вполне очевидную операцию, находя элементы, совпадающие с конкретным именем дескриптора. Но у него имеется и другое назначение: поиск в документе всех элементов или же отдельного элемента по имени дескриптора *. Это особенно удобно для обращения с селекторами, основанными на атрибутах и не предоставляющими конкретное имя дескриптора, например .class или [attr]. Следует, однако, иметь в виду, что при поиске комментариев к элементам разметки по критерию * в браузере Internet Explorer, помимо узлов элементов, будут также возвращаться узлы комментариев (последние имеют имя дескриптора ! в этом браузере и поэтому возвращаются). Следовательно, для исключения узлов комментариев из результатов поиска придется организовать элементарную фильтрацию.

Метод getElementsByName() служит единственной цели: находить все элементы разметки по указанному имени, например, элементы ввода данных, имеющие имя, указываемое в атрибуте name. К их числу относятся элементы разметки формы. Следовательно, данный метод особенно подходит для реализации единственного селектора [name=*имя*].

Метод getElementsByClassName() относится к числу новых методов, реализуемых в браузерах по спецификации HTML5. Он и находит элементы разметки, исходя из содержимого их атрибута class. Оказывается, что данный метод существенно ускоряет выполнение кода выборки класса.

Существуют и другие способы поиска и выборки элементов разметки, но приведенные выше методы, как правило, относятся к основным инструментальным средствам поиска на странице. Используя результаты, возвращаемые этими методами, можно организовать дальнейшую обработку, как будет пояснено далее.

Фильтрация результатов поиска

CSS-выражение обычно состоит из нескольких отдельных частей. Например, выражение div.class[id] состоит из трех частей, выполняющих поиск всех элементов разметки с именем class и атрибутом id.

Прежде всего необходимо выявить наличие корневого селектора. Так, если обнаруживается используемый селектор div, то с помощью метода getElementsByTagName() можно сразу же выбрать все элементы разметки на странице. Затем нужно отфильтровать полученные результаты, чтобы оставить только те, что содержат указанное имя class и атрибут id. Такой процесс фильтрации характерен для большинства реализаций механизмов селекторов. Содержимое фильтров имеет в основном

отношение к атрибутам или положению элемента относительно родственных с ним элементов. Ниже перечислены наиболее употребительные способы фильтрации результатов поиска.

- **Фильтрация атрибутов.** Применяется для доступа к атрибутам модели DOM (как правило, с помощью метода getAttribute()) и проверки их значений. Фильтрация классов (по критерию .class) является подмножеством такого поведения и означает доступ к атрибуту className и проверку его значения.
- **Фильтрация по положению.** Для таких селекторов, как nth-child(even) или :last-child, применяется определенное сочетание свойств, доступных для родительского элемента. В тех браузерах, где поддерживается свойство children, в частности в Internet Explorer, Safari, Opera и Firefox 3.1, получается список всех порожденных элементов. А во всех браузерах, где поддерживается свойство childNodes, получается список порожденных узлов, включая текстовые узлы, комментарии и т.д. С помощью обоих указанных свойств можно организовать все формы фильтрации элементов по положению.

Создание функции фильтрации преследует двоякую цель: предоставить пользователю простой способ проверки интересующих его элементов разметки, а также организовать быструю проверку элементов на совпадение с конкретным селектором. А теперь обратимся к инструментальным средствам для уточнения результатов.

Рекурсирование и объединение результатов

Как следует из листинга 15.1, механизму селекторов требуется рекурсирование (т.е. обнаружение порожденных элементов) и объединение полученных результатов. Но рассматриваемая здесь реализация механизма селекторов слишком проста и позволяет получать в итоге два элемента разметки вместо единственного. Поэтому придется внедрить дополнительную проверку, чтобы убедиться в том, что возвращаемый массив элементов разметки содержит только однозначные результаты. Большинство самых известных реализаций механизмов селекторов обладает некоторыми средствами для соблюдения подобной однозначности.

К сожалению, не существует простого способа определить однозначность элемента модели DOM. Поэтому приходится идти на присвоение элементам временных идентификаторов, чтобы проверить, встречались ли они прежде, как показано в листинге 15.7.

Листинг 15.7. Поиск однозначных элементов в массиве

```
<div id="test">
  <b>Hello</b>, I'm a ninja!
</div>
<div id="test2"></div>

<script type="text/javascript">

  (function(){

    var run = 0;

    this.unique = function(array) {
```

Установить нужные объекты для тестирования

Определить функцию unique() в теле немедленно вызываемой функции, чтобы образовать замыкание, включающее переменную run, недоступную извне

Принять массив элементов разметки и возвратить новый массив, содержащий только однозначные элементы из исходного массива

```
    var ret = [];

    run++;

    for (var i = 0, length = array.length; i < length; i++) {
      var elem = array[i];

      if (elem.uniqueID !== run) {
        elem.uniqueID = run;
        ret.push(array[i]);
      }
    }
    return ret;
  };
})();

window.onload = function(){
  var divs = unique(document.getElementsByTagName("div"));
  assert(divs.length === 2, "No duplicates removed.");

  var body = unique([document.body, document.body]);
  assert(body.length === 1, "body duplicate removed.");
};

</script>
```

Отследить стадию просмотра. Всякий раз, когда инкрементируется значение переменной run, вызывается функция и для проверки на однозначность используется единственное значение идентификатора

Просмотреть массив, скопировав еще не просмотренные элементы и отметив их на будущее как просмотренные

Возвратить итоговый массив, содержащий только ссылки на однозначные элементы разметки

Проверить! Первый тест не должен выявить никаких удаленных дубликатов, поскольку они не были переданы, а второй тест должен свести переданные дубликаты к единственному элементу

В функции `unique()` из приведенного выше примера кода вводится расширяемое свойство во все элементы разметки, находящиеся в массиве, благодаря чему можно проверить, встречались ли они прежде. Таким образом, в итоговом массиве окажутся только однозначные, не повторяющиеся элементы разметки. Подобный прием встречается в разных вариантах во всех библиотеках.

Более подробно особенности присоединения свойств к узлам модели DOM рассматриваются в главе 13, посвященной обработке событий. Затруднение, разрешаемое в данной функции, возникло в результате применения нисходящего подхода к реализации механизма селекторов. А теперь рассмотрим вкратце другой, восходящий подход.

Восходящий механизм селекторов

Если нет желания выявлять однозначные элементы разметки, то можно выбрать другую разновидность механизма CSS-селекторов, где ничего подобного не требуется. Восходящий механизм селекторов действует в противоположном нисходящему механизму направлении. Так, если задан селектор `div span`, то по нему сначала будут найдены все элементы разметки `<span>`, а затем по каждому найденному элементу будет произведен обход всех его родительских элементов, чтобы найти родительский элемент разметки `<div>`. Именно такая конструкция механизма селекторов соответствует принципу, по которому действует большинство подобных механизмов в браузерах.

Тем не менее восходящий механизм селекторов не нашел такого распространения, как нисходящий. И хотя он вполне пригоден для простых (и порожденных) селекторов, обход родительских элементов разметки в таком механизме требует значительных

затрат и недостаточно хорошо масштабируется. Но все эти недостатки возмещает простота, которую обеспечивает данный механизм.

Конструкция восходящего механизма селекторов довольно проста. Сначала осуществляется поиск последнего выражения в CSS-селекторе, а затем выбираются подходящие элементы разметки, как в нисходящем механизме, но в данном случае они находятся по последнему, а не по первому элементу. После этого все операции выполняются для последовательной фильтрации, в ходе которой удаляются все лишние элементы разметки, как показано в листинге 15.8.

Листинг 15.8. Простой восходящий механизм селекторов

```
<div>
  <div>
    <span>Span</span>
  </div>
</div>

<script type="text/javascript">

  window.onload = function(){
    function find(selector, root){
      root = root || document;

      var parts = selector.split(" "),
          query = parts[parts.length - 1],
          rest = parts.slice(0,-1).join(""),
          elems = root.getElementsByTagName(query),
          results = [];

      for (var i = 0; i < elems.length; i++) {
        if (rest) {
          var parent = elems[i].parentNode;
          while (parent && parent.nodeName != rest) {
          parent = parent.parentNode;
          }

          if (parent) {
            results.push(elems[i]);
          }
        } else {
          results.push(elems[i]);
        }
      }

      return results;
    };

    var divs = find("div");
    assert(divs.length === 2, "Correct number of divs found.");

    var divs = find("div", document.body);
```

```
    assert(divs.length === 2,
          "Correct number of divs found in body.");

    var divs = find("body div");
    assert(divs.length === 2,
          "Correct number of divs found in body.");

    var spans = find("div span");
    assert(spans.length === 1, "No duplicate span was found.");
  };

</script>
```

В примере кода из листинга 15.8 представлена конструкция простого восходящего механизма селекторов. Следует, однако, иметь в виду, что он осуществляет поиск и выборку лишь на один родительский элемент в глубину, а для более углубленного поиска в документе придется отслеживать состояние текущего уровня углубления. И этого можно добиться с помощью двух массивов состояния: массива возвращаемых элементов разметки (некоторые из них заданы как неопределенные (undefined), поскольку они не совпадают с полученными результатами), а также массива элементов разметки, совпадающих с проверяемым в настоящий момент родительским элементом.

Как упоминалось ранее, процессу дополнительной проверки родительского элемента недостает масшабируемости, присущей нисходящему механизму селекторов. Но, с другой стороны, отпадает необходимость в особом методе или функции для получения правильного результата, и в этом есть свое несомненное преимущество.

Резюме

Создаваемые в JavaScript механизмы CSS-селекторов являются весьма эффективными инструментальными средствами для поиска и выборки практически любых элементов модели DOM на веб-странице с помощью обыкновенного синтаксиса селекторов. Существует немало особенностей, препятствующих полноценной реализации механизма селектора, не говоря уже о недостатке инструментальных средств, помогающих преодолеть эти препятствия, хотя положение в данной области быстро меняется к лучшему.

В этой главе были рассмотрены следующие вопросы.

- В современных браузерах реализуются прикладные интерфейсы API по стандартам консорциума W3C для выборки элементов разметки, но они пока еще далеки от совершенства.
- Механизмы селекторов приходится по-прежнему создавать самостоятельно, главным образом, из соображений производительности.
- Создать механизм селекторов можно одним из трех способов.
  - Воспользоваться прикладными интерфейсами API, разработанными по стандартам консорциума W3C.
  - Воспользоваться языковыми средствами XPath.
  - Организовать самостоятельно обход узлов модели DOM для достижения оптимальной производительности.

- Наибольшее распространение получил нисходящий подход к созданию механизмов селекторов, но он требует выполнения дополнительных операций редактирования получаемых результатов, в том числе для обеспечения однозначности обнаруженных элементов разметки.
- Восходящий подход к созданию механизмов селекторов исключает потребность в подобных операциях, но ему присущи недостатки в отношении производительности и масштабируемости.

По мере внедрения прикладного интерфейса Selector API по стандарту консорциума W3C в современных браузерах потребность учитывать особенности реализации механизмов селекторов постепенно отпадет. Но для многих разработчиков веб-приложений это время вряд ли наступит очень скоро.

Предметный указатель

D

Document object model. См. DOM

Б

Библиотеки JavaScript
  заключение в оболочку 147
  имитация классической формы наследования 178
  особенности 25
  разновидности 26
  структура 26
Борьба с поглощающими идентификаторами 280

В

Выбор поддерживаемых браузеров 272
Вычисление кода
  безопасное 241
  в глобальной области действия 238, 386
  динамическое переписывание 249
  запутывание 249
  импорт из пространства имен 246
  механизмы 234
  преобразование из формата JSON 245
  реализация метаязыков 252
  специальные сценарии 251
  с помощью
    метода eval() 234
    таймеров 238
    функции-конструктора 237
  уплотнение 247

З

Замыкания
  доступ к частным переменным 122
  обратные вызовы таймеров 124
  определение 118
  особенности 121
  применение 122
  принцип действия 118
  сокрытие функциональных возможностей 138
Запоминание
  возвращаемых значений 136
  вызовов функций 137
  вычисленных значений 102
  определение 100
  элементов модели DOM 102
Запутывание кода, назначение 249

И

Имитация
  методов обработки массивов 103
  наследования в библиотеках JavaScript 178
  перегрузки функций 106
  частных переменных 122
Инкапсуляция кода 277
Исключение внедряемых свойств 278

К

Каррирование
  назначение 132
  применение 133
Конструкторы
  назначение 78
  операции привязки, приоритетность 155
  сильные стороны 78
Кросс-браузерная разработка
  главные факторы 29
  имитация компонентов 288
  надежное устранения ошибок в коде 285
  насущные задачи 273
  обнаружение объектов 286
  передовые методики
    анализа производительности 32
    основные элементы 31
    тестирования 31
  поддержка браузеров, оценка затрат 29
  сокращение допущений 293

М

Метаязыки
  Objective-J 253
  Processing.js 252
  назначение 252
Механизмы
  CSS-селекторов
    восходящие, реализация 408
    нисходящие, реализация 402
    определение 396
    основные средства реализации 396
    поиск элементов разметки 405
    прикладной интерфейс Selectors API 397

    рекурсирование и объединение результатов 407
    синтаксический анализ 404
    фильтрация результатов поиска, способы 406
    чистая модель DOM, реализация 401
    язык запросов XPath 400
  АОП 250
  всплывания событий, реализация 363
  вычисления кода 234
  подъемные 68
Модель DOM
  атрибуты и свойства
    вопросы производительности 300
    доступ к значениям, способы 296
    затруднения кросс-браузерного характера 304, 310
    назначение 295
    отличия HTML от XML 299
    поведение специальных атрибутов 300
    присваивание имен, особенности 298
    трудности стилевого оформления 311, 322
  вставка HTML-разметки
    в документ 383
    выполнение сценариев 385
    заключение в оболочку 380
    преобразование из формата HTML в DOM 379
    прикладной интерфейс API, реализация 379
    способ 378
    формирование узлов 382
  клонирование элементов разметки, особенности 387
  обработка текста
    в браузерах, особенности 390
    получение текста 393
    установка текста 392
  удаление элементов разметки, особенности 389
  фрагменты
    преимущества 383
    применение 383
  чистая, реализация 401

Н

Наследование
  имитация классической формы 178
  реализация 180
Непроверяемые ошибки в браузерах, области 291, 293

О

Области действия
  временные и замкнутые, создание 142
  определение
    особенности 68
    правила 68
  поведение, проверка 70
  разделение 143
Обработка ошибок в браузерах 274
Обратная совместимость, соблюдение 282
Обратные вызовы
  применение 61
  принцип действия 61
Объектная модель документа. См. DOM
Объекты
  window
    как контекст функций 74
    свойства 65
  как контекст функций 80
  наследование 161
  получение экземпляров 152
  принудительная установка в качестве контекста функций 82
  проверка на функции 114
  свойства
    constructor 156, 160
    основные 58
    экземпляров 154
  собственных классов, подклассификация 171
  типа Event
    нормализация экземпляров 339
    особенности обращения 338
    свойства 341
    таблицы совместимости режимов 342
  типизация через конструкторы 159
Оператор with
  область действия
    ограничения 261
    присваивание 260
    ссылки на свойства 258
  особенности применения 258
  примеры применения 264, 270
  проверка на производительность 261
Отладка кода JavaScript
  доступные средства 36
  операторы регистрации 36
  регистрация результатов 36
  точки прерывания 38

П

Поддержка браузеров
  классификация 28
  матрица 28, 272
  оценка затрат 29
Постепенное снижение функциональных возможностей 282
Построение по шаблону
  назначение 267
  с помощью оператора with 267
Программирование
  аспектно-ориентированное
    определение 250
    применение принципов 250
  в объектно-ориентированном стиле 77
  в функциональном стиле 82, 89
  функциональное
    на основе анонимных функций 90
    особенности 89
    преимущества 82
Прототипы
  HTML-разметки 167
  в качестве образцов для объектов 152
  как свойства функций 151
  назначение 151
  наследование по цепочке 163
  скрытые препятствия и их преодоление 168, 177
  увязывание ссылок 156
  цепочка, создание 162

Р

Ранняя диагностика таблиц стилей 281
Регрессии
  меры борьбы 283
  определение 283
  трудности преодоления 283
Регулярные выражения
  группирование
    без фиксации 202
    членов 202
  замена с помощью функций 203
  компиляция и выполнение 195
  локальные и глобальные 199
  назначение 187, 189
  обратные ссылки 195
  обрезка символьных строк 206
  пассивные подвыражения 202
  поглощающие и непоглощающие операторы 193
  построение, способы 189
  проверка на совпадение
    с концами строк 208
    с символами юникода 209
    с экранированными символами 210
  скрытый потенциал 189
  фиксация
    назначение 198
    обратные ссылки 201
    простая 198
  флажки 190
  члены и операторы 190, 194
Рекурсия
  в именованных функциях 90
  в методах 92
  назначение 90
  определение 90
  основные критерии 91

С

Связывание
  сильное 357
  слабое 357
События
  change
    всплывание 366
    привязка других событий 366
  focusin и focusout, реализация обработки 369
  mouseenter и mouseleave, реализация обработки 370
  submit
    всплывание 364
    инициирование 364
    совмещение с событием click или keypress 364
  всплывание и фиксация 362
  готовности документа
    наступление 372
    реализация обработки, комплексный подход 372
  делегирование
    назначение 361
    родительскому элементу 362
  диспетчеризация 348
  инициирование
    порядок действий 355
    условия 356
  обработка
    модели 333, 334
    особенности управления 342, 346
    очистка памяти 349

    принимаемые меры 335
  отвязка обработчиков 351
  порядок обработки по очереди 60
  привязка обработчиков 346
  разновидности 59
  специальные
    имитация 358
    инициирование 359
Списки аргументов переменной длины
  обнаружение и обход 107
  применение 104
Среды блочного тестирования
  Jasmine, особенности 45
  JsUnit, особенности 44
  QUnit, особенности 44
  TestSwarm, особенности 45
  YUI Test, особенности 44
  применение на практике 42
  свойства 42
Стилевое оформление
  вычисленные стили
    извлечение 327
    назначение 325
  именование свойств 313
  манипулирование непрозрачностью 321
  местонахождение стилей 311
  обращение к свойству float 314
  преобразование значений, указываемых в пикселях 315
  расхождения в форматах цвета 322
  самозамыкающиеся элементы 380
  сочетание свойств 329
  указание размеров по высоте и ширине 316

Т

Таймеры
  выполнение, особенности 215
  методы манипулирования 214
  минимальная задержка 218
  назначение 213
  отличия интервалов от блокировок по времени 217
  прерывание длительного задания 222
  принцип действия 214
  установка и очистка 214
  центральное управление 225
Тестирование
  асинхронное
    организация с помощью таймеров 228
    особенности 48
    осуществление с помощью оператора with 266
    порядок действий 48
  группы тестов
    назначение 46
    формирование и управление 46
  инструментальные средства 43
  надежная методика, необходимость 35
  основные свойства тестов 39
  построение тестов
    контрольные примеры 40
    обработка асинхронных тестов 48
  службы создания контрольных примеров 41
  тестовые наборы
    назначение 42
    построение 45
  утверждения, принцип и реализация 45

У

Уплотнение кода, назначение 248
Условные обозначения имен 174

Ф

Функции
  assert(), назначение 32
  hasOwnProperty(), назначение 278
  анонимные
    объявление 88
    применение 88
  аргументы
    порядок присваивания 73
    разбиение списка 108
  встраиваемые, применение 95
  вызов
    асинхронный 59
    в виде
      конструктора 77
      метода 75
      функции 74
    с помощью методов apply() и call() 80
    способы 72
  декомпиляция 242
  заключение в оболочку 138
  испытание на дым 353
  как объекты высшего порядка 58, 98
  конструкторы, особенности 79
  контекст
    определение 74
    привязка 127
    установка 80
  наделение свойствами 98

немедленно вызываемые
конструкция 141
применение 142, 146
неявные параметры
arguments 73, 106
this 74
области действия, определение 68
обратного вызова
применение 61
установка контекста 82
объявление, особенности 64
перегрузка
по числу аргументов 111
способы 110
присваивание переменным и свойствам 98
рекурсивные, реализация 91
самозапоминающиеся, применение 101
свойства
arguments.callee 97
length 110
name 65
prototype 152
преимущества 103
сериализация 180
сохранение 99
частичное применение 132

Ц

Цикл ожидания событий в браузере
выполнение 58
однопоточный характер 60

Я

Язык JavaScript
взаимосвязь между объектами, функциями и замыканиями 27
конструкторы, особенности 78
литералы функциональные, состав 65
методы
apply(), применение 80, 105
Array.prototype.push(), применение 104
assert(), назначение 45
call(), применение 80
eval(), назначение 27, 234
exec(), применение 200
forEach(), применение 165
hasOwnProperty(), применение 169
match(), применение 199
replace(), применение 203
доступа, применение 123
ненавязчивый, понятие 59
области действия, определение 68
обратные вызовы, принцип действия 61
объекты, основные свойства 58
однопоточный характер обработки 215
операторы
instanceof, применение 160
new, применение 152
typeof, применение 160
with, особенности 27, 258
отличия от Java 57, 63
перечень решаемых задач 187
регулярные выражения, применение 27
таймеры, назначение 27
форма наследования 177
функциональный характер 57