

Б. Лоусон Р. Шарп

ИЗУЧАЕМ
HTML5

Библиотека специалиста

2-е издание



B. Lawson, R. Sharp

INTRODUCING HTML5

Second Edition



New
Riders

VOICES THAT MATTER™

Б. Лоусон, Р. Шарп

ИЗУЧАЕМ HTML5

Библиотека специалиста

2-е издание



Москва · Санкт-Петербург · Нижний Новгород · Воронеж

Ростов-на-Дону · Екатеринбург · Самара · Новосибирск

Киев · Харьков · Минск

2012

ББК 32.988.02-018

УДК 004.438.5

Л81

Лоусон Б., Шарп Р.

Л81 Изучаем HTML5. Библиотека специалиста. 2-е изд. — СПб.: Питер, 2012. — 304 с.: ил.

ISBN 978-5-459-01156-2

Эта книга, существенно обновленная и дополненная во втором издании, посвящена изучению нового стандарта веб-программирования HTML5 и использованию новых функциональных возможностей, предоставляемых веб-разработчикам. Написанная опытными программистами, давно работающими с HTML5, книга расскажет, как приступить к освоению этого языка программирования и адаптации веб-проектов для того, чтобы в полной мере воспользоваться преимуществами нового стандарта. Авторы не заставляют вас читать избыточно подробные спецификации языка, а учат применять HTML5 на конкретных примерах.

С помощью этой книги вы узнаете, каковы новые семантические и структуры HTML5, как использовать мультимедийные элементы HTML5 для размещения аудио- и видеоданных, как разрабатывать «умные» веб-формы и совершенствовать приложения, использующие географическую привязку, с помощью API геолокации.

Новое издание книги дополнено более подробной информацией о работе с аудио, видео и графикой, о функциях геолокации, автономном режиме хранения данных, применении HTML5 в старых браузерах.

ББК 32.988.02-018

УДК 004.438.5

Права на издание получены по соглашению с New Riders Publishing. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-0321784421 англ.

ISBN 978-5-459-01156-2

© Remy Sharp and Bruce Lawson

© Перевод на русский язык ООО Издательство «Питер», 2012

© Издание на русском языке, оформление ООО Издательство «Питер», 2012

КРАТКОЕ ОГЛАВЛЕНИЕ

Введение	13
Глава 1. Основная структура	23
Глава 2. Текст	45
Глава 3. Формы	99
Глава 4. Видео и аудио	127
Глава 5. Холст	161
Глава 6. Хранение данных	187
Глава 7. Автономная работа	221
Глава 8. Перетаскивание	239
Глава 9. Геолокация	253
Глава 10. Сообщения и фоновые вычисления	265
Глава 11. Работа в режиме реального времени	281
Глава 12. Заполнение: латаем старые браузеры и учим их понимать HTML 5	291

ОГЛАВЛЕНИЕ

Введение	13
Для кого эта книга?	15
Чем не является эта книга	15
Что мы имеем в виду под HTML5	16
Кто? Что? Когда? Почему? Краткая история HTML5	16
Как HTML5 чуть не канул в бывшность	16
Принципы, скрывающиеся за HTML5	19
Глава 1. Основная структура	23
<head>	25
Использование новых структурных элементов HTML5	29
Оформление HTML5-документа с помощью CSS	33
Создание CSS для Internet Explorer 6, 7, 8	34
Другие проблемы устаревших браузеров	35
Когда используются новые структурные элементы HTML5	36
<header>	36
<nav>	38
<footer>	41
<article>	42
В чем же суть?	42
Заключение	43
Глава 2. Текст	45
Создание структуры основных областей контента	47
Кое-что интересное об элементах <header> и <footer>	51
Добавление сообщений блога и комментариев	53
Работа со структурами HTML5	54
<nav>, <aside> и разделы без заголовка	57
<hgroup>	57
Структурированные корневые элементы	58
Создание стилей для заголовков в HTML5	59
Алгоритм определения структуры и доступность	60
В чем различие между <article> и <section>?	61
Анализ www.guardian.co.uk	66
Что такое WAI-ARIA	72
ARIA-структура документа и роли	73
Как совмещать ARIA и HTML	74
Ресурсы ARIA	74

Еще больше новых структур!	75
Микроданные	75
API DOM Microdata	78
<aside>	79
<bdi>	79
<details>	81
<figure>	81
<mark>	84
<ruby>, <rp>, <rt>	84
<wbr>	85
Переопределенные элементы	85
<address>	85
<cite>	86
<dl>	87
, <i>	87
<hr>	88
	88
<s>	88
<small>	89
, 	89
<u>	90
Удаленные элементы	90
Глобальные атрибуты	90
accesskey	90
contenteditable	91
data-* (пользовательские атрибуты данных)	91
draggable	93
hidden	93
id	93
itemscope, itemprop, itemtype, itemref, itemid	93
role, aria-*	93
spellcheck	94
tabindex (=1)	94
Атрибуты, удаленные из спецификации	95
<table border=...>	95
<table summary=...>	95
	96
Свойства, не рассматриваемые в этой книге	96
<embed>	96
<keygen>	96
<menu>, <command>	97
<style scoped>	97
Заключение	97
Глава 3. Формы	99
Мы ❤ HTML, и теперь он тоже нас ❤	101
Новые типы ввода данных	101
Тип ввода данных email	103
Тип ввода данных URL	103
Тип ввода данных date	103

Тип ввода данных time	104
Тип ввода данных datetime	104
Тип ввода данных month	104
Тип ввода данных week	105
Тип ввода данных number	105
Тип ввода данных range	106
Тип ввода данных search	106
Тип ввода данных tel	106
Тип ввода данных color	107
Новые атрибуты	108
Атрибут list и <datalist>	108
Атрибут autofocus	110
Атрибут placeholder	110
Атрибут required	110
Атрибут multiple	110
Атрибут pattern	111
Атрибут autocomplete	112
Атрибуты min и max	112
Атрибут step	112
Атрибут form	113
Элементы <progress>, <meter>	113
Как собрать все это вместе	114
Форма комментария блога	114
Ползунок со скриптом для вывода	115
Обратная совместимость с устаревшими браузерами	117
Оформление новых полей формы и сообщений об ошибках	118
Переопределение умолчаний браузера	120
Использование JavaScript для DIY-валидации	122
Как инициировать валидацию элемента	122
Валидность элементов	123
Как избежать валидации	123
Событие «всякий раз, когда что-то меняется»	124
Заключение	126
Глава 4. Видео и аудио	127
Родная мультимедиа: почему, что и как?	129
<video>: зачем нужен элемент <video>?	129
Для чего не подходит мультимедиа HTML5	130
Анатомия элементов video и audio	131
Автвоспроизведение (autoplay)	132
Элементы управления (controls)	132
Изображение (poster)	134
Беззвучный режим (muted)	134
Высота, ширина (height, width)	134
Повтор (loop)	134
Предварительная загрузка (preload)	135
Источник (src)	135
Кодеки: ужас-ужас	136
Множественные элементы <source>	137
Видео для устаревших браузеров	139
Видеофайлы разных форматов и карманные устройства	141

Пользовательские элементы управления	142
Воспроизведение, пауза — и все это на одной кнопке	144
Богатые событиями элементы медиа	145
Отслеживаем ход загрузки	146
Когда медиафайл готов к воспроизведению	147
Гонка за воспроизведением видео	149
Обходные пути	149
Перемотка вперед, замедленное и обратное воспроизведение	152
Полноэкранное видео	153
Возможности управления мультимедиа	153
Синхронизация медиатреков	156
Видеоконференции, дополненная реальность	157
Заключение	159
Глава 5. Холст	161
Самое главное о холсте	165
Рисование с градиентами и узорами	166
Рисование путей	169
Использование трансформеров: замаскированные пиксели	172
Захват изображений	173
Работа с отдельными пикселями	177
Запись в файл	180
Анимированные изображения на холсте	181
Сохранение и восстановление условий рисования	183
Визуализация текста	184
Достигимость в пределах элемента холста	185
Заключение	186
Глава 6. Хранение данных	187
Варианты хранения	190
Веб-хранилище	191
15-секундный урок	192
События хранилища	198
Базы данных SQL для веб	202
IndexedDB	211
Заключение	220
Глава 7. Автономная работа	221
Выдерните вилку из розетки: работаем автономно	223
Манифест кэша	224
Подробнее о разделах Network и Fallback	227
Добавление содержимого в раздел FALLBACK	227
Использование «белого списка»	228
Как правильно подавать манифест	229
Процессы между браузером и сервером	230
applicationCache	232
Советы по отладке	233
Определение наличия подключения с помощью манифеста	236
Очистка кэша	237
Заключение	238

Глава 8. Перетаскивание	239
Начинаем перетаскивать	241
Совместимость перетаскиваемых данных	245
Перетаскивание данных в другие приложения	245
Как перетаскивать любой элемент	247
Добавление пользовательского значка перетаскивания	248
Доступность	249
Заключение	251
Глава 9. Геолокация	253
Пришпильте посетителя к карте	255
Методы API	257
Ты попался: обработчик успешного исполнения	258
Вычисляем скорость	260
Адрес 404: обработчик ошибок	260
Настройка геолокации	262
Заключение	263
Глава 10. Сообщения и фоновые вычисления	265
Крошечный чатик с помощью API Messaging	267
Отправка сообщений между доменами	267
А можно ли отправлять что-нибудь еще кроме строк?	269
Разбиение на потоки с помощью фоновых вычислений	269
Зачем использовать фоновое вычисление?	269
Резюме	280
Глава 11. Работа в режиме реального времени	281
Веб-сокеты: работа с потоковыми данными	283
Сокеты решают проблему задержек в приложениях реального времени	284
API веб-сокетов прост	284
Больше, чем прослушивание сокета	286
События, отправляемые сервером	287
Простой сервер EventSource	289
Поддержка реализации	290
Заключение	290
Глава 12. Заполнение: латаем старые браузеры и учим их понимать HTML 5	291
Знакомство с заполнителями	293
Распознавание возможностей	294
Распознавание свойств	294
Распознавание новых функций	295
Распознавание любых объектов без помощи JavaScript	296
Производительность распознавания возможностей и заполнения	296
Что распознать невозможно	297
Где найти заполнители	298
Рабочий пример с использованием Modernizr	298
Резюме	300
И напоследок	300

БЛАГОДАРНОСТИ

Я хочу сказать огромное спасибо моему соавтору-ставшему-другом Реми Шарпу и другу-ставшему-беспрощальным-научным-редактором Патрику Лауке: *il miglior fabbro*. Сотрудники издательства New Riders Майкл Нолан, Маргарет Андерсон, Гретхен Дикстра и Ян Сеймур заслуживают медали за свое терпение и тяжелую работу.

Моя благодарность команде Opera Developer Relations Team, в частности редактору dev.opera.com Крису Миллсу за то, что разрешили использовать в книге некоторые материалы, которые я создавал для них. Спасибо Дэниэлю Дэвису за описание `<ruby>`, Швитанку Дикситу за проверку части черновиков и Дэвиду Стори за его знания о веб-стандартах, которыми он щедро делился со мной. Благодарю бывшего члена команды Генни Свон за ее поддержку и лимонный пирог. Где-то еще, в других отделах компании Opera, команда по спецификациям, состоящая из Джеймса Грэма, Лэчлан Хант, Филипа Ягенштеда, Анни Ван Кестерен и Саймона Питерса, проверяла главы и отвечала на каждый из моих 45 763 тупых вопросов, не теряя при этом оптимизма и чувства юмора. Ничто в этой книге не представляет субъективные убеждения компании Opera Software ASA.

Ян Хиксон также дал ответы на множество моих вопросов, а доктора HTML5 (<http://www.html5doctor.com>) обеспечили постоянную поддержку и помогали разбираться в сути вещей.

Отдельная благодарность Ричарду Ишида за то, что объяснил мне тонкости `<bdi>` и позволил привести его объяснение в книге, а также Аарону Ланину. Горячий поцелуй Робину Берйону и команде Mozilla Developer Center, позволившим цитировать их.

Спасибо Гезу Лемону и великому Стиву Фолкнеру за советы относительно WAI-ARIA. Спасибо Денису Бодро, Адриану Хиггинботаму, Пратику Парелу, Грегори Дж. Росмайта и Леони Уотсон за помощь с программой экранного доступа.

Благодарю Стюарта Лангриджа за совместные вечера за рюмкой чая, аморальную поддержку и идею дать книге рабочее название «Утопия HTML5». Мистер Лучшее Ругательство Прошлой Недели подарил мне немало минут здорового смеха. Спасибо, кто бы ты ни был.

Спасибо Джону Оллсоппу, Тантеку Челику, Кристиану Хейлманну, Джону Фолиоту, Джереми Кейту, Мэтту Мэю и Эрику Мейеру за беседы о будущем разметки. Посвященные мультимедиа публикации в блоге Сильвии Пфайфер стали бесценным подспорьем для понимания темы и написания этой книги.

Стю Робсон бросил вызов IE6, чтобы сделать снимок экрана для главы 1; Теренс Эден сделал снимки экрана устройства BlackBerry для главы 3; Джуллия Гослинг сфотографировала волшебные усы Реми, сделанные с помощью HTML5, для главы 4; а Джейк Смит дал бесценные отзывы о первых набросках книги. Наконец, самую большую благодарность я хотел бы выразить тысячам студентов, участников конференций и пользователей Twitter за их вопросы и комментарии.

Этой книгой я хотел бы почтить память моих бабушек, Марджори Уайтхед (8 марта 1917–28 апреля 2010) и Элси Лоусон (6 июня 1920–29 августа 2010).

Я посвящаю свою книгу Нонгау, Марине и Джеймсу, без которых моя жизнь была бы монохромной.

— Брюс Лоусон

Промадное спасибо Брюсу за возможность поучаствовать в написании этой книги — не получив его приглашение, начало 2010 года я провел бы в жалобах на плохую погоду вместо кропотливого труда. И раз уж на то пошло, хочу поблагодарить Криса Миллса за то, что порекомендовал меня Брюсу.

Роберту Ниману, моему техническому редактору: ты всегда был рядом, когда меня требовалось немножко подстегнуть. Благодаря тебе мой код JavaScript стал таким надежным и красивым. А также спасибо Патрику Лауке, который помог обрести нужную форму не только моему коду, но и моим навыкам владения английским языком.

Кроме того, я хочу сказать спасибо брайтонским кафе Coffee@33 и Café Délice за то, что позволили проводить так много времени, работая над книгой и попивая кофе.

Я обязан упомянуть сообщество любителей цифровых технологий Брайтона и моих новых друзей, которые сохраняли мой рассудок и нужную степень безрассудства в течение последних лет, пока я работал в одиночестве. Дэнни Хоуп, Джош Расселл, Анна Дебенхам — спасибо за то, что оставались моими коллегами на расстоянии.

Джереми Кейт — благодаря тебе я научился восхвалять и поносить HTML5 и делиться идеями. Спасибо, что поддержал меня, когда речь зашла о публикации моих мыслей. И не меньшая благодарность Джессике, которая позволяла нам болтать на технические темы за бокалом пива!

Спасибо HTML5 Doctors и, в частности, Ричу Кларку за приглашение поделиться опытом — и всей команде в целом за публикацию великолепных материалов.

Спасибо всему каналу #jquery-ot за помощь в отладке, за то, что слушали, когда мне нужно было выговориться в трудный момент, и вообще, за то, что к вам всегда можно обратиться за советом — от моих кошечек в деле разработки на JavaScript пользы мало.

Спасибо каналу #whatwg за помощь с неправильно понятной спецификацией — вы наставили меня на путь истинный. Особая благодарность Анни Ван Кестерен, у которой, похоже, есть ответ на любой мой вопрос — даже если некоторые из них спрятаны в местах, до которых мне еще только предстоит добраться.

Спасибо всем организаторам конференций, которые приглашали меня выступать, и всем посетителям, которые приходили послушать мою болтовню. Спасибо моим читателям в Twitter, которые помогали мне находить ответы на важные вопросы и подталкивали нас с Брюсом к скорейшему завершению работы над этой книгой. Я выложился по полной, и если вы найдете здесь какую-нибудь ошибочную или устаревшую информацию, [вините Брюса](#) бегите покупать следующее издание. ;-)

Моей жене, Джули: спасибо за то, что поддерживала меня все эти годы. Я не знаю, чем заслужил такое счастье, но честно — без тебя я не был бы тем человеком, которым являюсь сейчас.

Наконец, я хочу посвятить эту книгу Тие. Моя девочка. Большую часть этой книги я написал, пока ты готовилась прийти к нам. Я представлял, как ты увидишь эту книгу и почувствуешь одновременно гордость, и смущение. Этого не произойдет, и хотя ты покинула нас, ты всегда будешь с нами, и мы всегда будем помнить о тебе.

— Реми Шарп

ВВЕДЕНИЕ



Добро пожаловать на второй сезон шоу Реми и Брюса. С момента публикации первого издания книги в июле 2010 года многое изменилось: поддержка HTML5 стала значительно масштабнее; появился долгожданный браузер Internet Explorer 9; разработчики Google Chrome объявили о прекращении поддержки видео в формате H.264; разработчики Opera экспериментировали с потоковой передачей видео с веб-камеры пользователя через браузер, а лихорадка HTML5 превратилась в истерию HTML5, и теперь любую новую технику и технологию все, включая клиентов, боссов и журналистов, не думая, относят на счет HTML5.

Все эти изменения и многое другое обсуждается в нашем блестящем новеньком втором издании. Вы найдете здесь совершенно новую главу 12, в которой обсуждаются реальности внедрения новых технологий в старые браузеры. Кроме того, мы исправили несколько ошибок, выловили несколько опечаток, переписали часть малопонятных разделов и добавили, по меньшей мере, одну шутку.

Мы разработчики, и мы развлекаемся с HTML5 с Рождества 2008 года — экспериментируем, участвуем в рассылке и всячески пытаемся помочь привести этот язык в порядок и изучить его.

Так как мы разработчики, нас интересует процесс построения разных вещей. Поэтому в центре внимания этой книги находятся проблемы, которые HTML5 в состоянии решить, а не формальное исследование языка. Кроме того, следует отметить, что хотя Брюс работает в компании Opera Software, которая первой начала проверять в действии идеи, позже ставшие основой HTML5, он не является участником команды, разрабатывающей спецификацию; он вносит вклад как автор, использующий данный язык разметки.

Для кого эта книга?

При том что никаких знаний HTML5 от вас не требуется, мы все же предполагаем, что вы — достаточно опытный разработчик (X)HTML, знакомый с идеями семантической разметки. Неважно, с какими типами документов вы лучше знакомы — HTML или XHTML, но вы должны приходить в восторг от написания любой строгой разметки.

Хотя вы не обязаны быть JavaScript-ниндзя, вы должны понимать, насколько стремительно возрастает важность этого языка для современной веб-разработки, а термины вроде DOM и API не должны вызывать у вас желание в ужасе забросить эту книгу подальше и быстро убежать.

Вы все еще здесь? Хорошо.

Чем не является эта книга

Эта книга — не справочник. Мы не будем перечислять все элементы и API в линейном порядке, подробно останавливаясь на каждом из них и затем двигаясь дальше. Эта работа уже выполнена в спецификации: там описаны все умопомрачительные, душепитательные, но абсолютно необходимые детали.

Но спецификация не пытается научить тому, как нужно использовать эти элементы и API, и тому, как они работают в контексте друг друга. Мы будем создавать код для различных

примеров, по ходу обсуждая новые темы, и возвращаться к этим примерам, когда нам будет что обсудить.

Вы также поймете — из названия и того факта, что вы можете спокойно держать эту книгу в руках без помощи электропогрузчика, — что эта книга не содержит исчерпывающего описания. Истолковать спецификацию, которая в печатном виде занимает более 700 страниц (сравните: первая спецификация HTML умещалась на трех), в книге среднего размера можно разве что с помощью какой-нибудь фантастической технологии (наподобие ТАРДИС), что было бы здорово, или микроскопических шрифтов, что уже не здорово.

ЧТО МЫ ИМЕЕМ В ВИДУ ПОД HTML5

Возможно, это звучит глупо, но среди экспертов по стандартам все более популярным становится стремление сваливать все новые и интересные веб-технологии в общую кучу под названием HTML5. Так, например, SVG (Scalable Vector Graphics, масштабируемая векторная графика) описывается как «одна из технологий семейства HTML5», хотя это независимая *графическая* спецификация W3C, созданная десять лет назад.

Еще большая путаница возникает потому, что официальная спецификация W3C чем-то похожа на амебу: кусочки отрываются от нее и становятся отдельными спецификациями, как, например, Web Sockets (сокеты) или Web Storage (хранилище), хотя они и созданы той же группой разработчиков.

Так что в этой книге мы говорим об «HTML5 и связанных с ним спецификациях, созданных WHATWG» (расшифровку этой потрясающей аббревиатуры вы узнаете чуть позже). И мы также привели на вечеринку «еще одного гостя» — Геолокацию, — которая не имеет никакого отношения к нашему определению HTML5, но мы ее добавили просто потому, что она, на самом деле, великолепная, мы от нее в полном восторге, и она является частью «новой волны потрясающих технологий для создания веб-приложений».

КТО? ЧТО? КОГДА? ПОЧЕМУ? КРАТКАЯ ИСТОРИЯ HTML5

Исторические разделы в компьютерных книгах обычно всех раздражают. Не обязательно знать, что такое ARPANET или как зародился HTTP, чтобы понять, как написать новый язык.

И тем не менее представлять себе, как возник HTML5, необходимо — это поможет вам понять, почему некоторые свойства HTML5 выглядят именно так, и, я надеюсь, предвосхитить некоторые вопросы типа: «WTF? Какого черта? Почему они сделали это *так?*» (или, по крайней мере, спокойно на них отреагировать).

КАК HTML5 ЧУТЬ НЕ КАНУЛ В БЕЗВЕСТНОСТЬ

В 1998 году W3C решила больше не заниматься разработкой HTML. Они считали (как и ваши авторы), что будущее за XML. Поэтому HTML был заморожен в версии 4.01. Затем

была выпущена спецификация под названием XHTML 1.0, которая представляла собой XML-версию HTML, использующую правила синтаксиса XML — заключение атрибутов в кавычки, закрытие некоторых тегов (при наличии самозакрывающихся тегов) и т. п. Были разработаны два стандарта (вообще-то три, если вас волнует HTML **Frames**, но я надеюсь, что не волнует — потому что в HTML5 его нет). Первый, XHTML Transitional, был создан, чтобы помочь разработчикам плавно перейти к использованию второго — золотого стандарта XHTML Strict.

Сначала все было нормально — целое поколение разработчиков (или, по крайней мере, профессионалов) привыкло думать в терминах допустимого, хорошо структурированного кода. Но позже началась работа над спецификацией XHTML 2.0, которая предвещала революцию в языке, поскольку ее разработчики пытались улучшить логику и общую организацию языка за счет несоблюдения принципа обратной совместимости.

Однако небольшая группа разработчиков из компании Opera не разделяла того убеждения, что будущее Сети — это XML. Эти люди по своей собственной инициативе начали работу над экспериментальной спецификацией, в которую были добавлены расширения для HTML-форм без нарушения обратной совместимости. В конечном итоге эта спецификация превратилась в Web Forms 2.0, а позже была включена в спецификацию HTML5. К группе вскоре присоединились разработчики из Mozilla, и они все вместе во главе с Яном «Хикси» Хиксоном продолжили работу над спецификацией, неофициально сотрудничая с компанией Apple, «подбадривавшей их, стоя в сторонке», в составе нескольких разработчиков, которые называли себя WHATWG (Web Hypertext Application Technology Working Group, «рабочая группа по разработке гипертекстовых приложений для Интернета» — <http://www.whatwg.org>). Об этом сотрудничестве свидетельствуют сведения об авторских правах, указанные в версии спецификации WHATWG: «© Copyright 2004–2011 Apple Computer, Inc., Mozilla Foundation и Opera Software ASA» (кстати, лицензия разрешает вам ее использовать, копировать и создавать свои версии).

Хиксон перешел из Opera в Google, где он стал полноправным разработчиком HTML5 (что затем стало называться Web Applications 1.0).

В 2006 разработчики W3C решили, что они были чересчур оптимистичны, полагая, что весь мир перейдет на XML (и, естественно, к XHTML 2.0): «Важно развивать HTML шаг за шагом. Попытка заставить мир одномоментно перейти на XML, включая кавычки вокруг атрибутов, закрытие пустых тегов и пространства имен, провалилась», — признал Тим Бернерс-Ли.

Воссозданная HTML Working Group (рабочая группа HTML) проголосовала за использование спецификации WHATWG Web Applications в качестве основы новой версии HTML, и так начался процесс тщательной работы над спецификацией, аналог которой параллельно разрабатывала W3C (в руководстве процессом участвовали Сэм Руби из IBM и Крис Уилсон из Microsoft, а позднее — Руби, Пол Коттон из Microsoft и Мацей Сташовяк из Apple) и WHATWG во главе с ее бессменным редактором Хиксоном.

Этот процесс был крайне необычным в нескольких отношениях. Странной казалась его открытость; кто угодно мог подписать на рассылку WHATWG и помочь в разработке спецификации. Каждое письмо читал лично Хиксон или кто-либо из самых активных членов группы WHATWG (в которую входили такие светила, как создатель JavaScript и Mozilla

СТО Брэндан Айк, разработчик архитектуры Safari и WebKit Дэвид Хаятт и изобретатель CSS и Opera СТО Хокон Виум Ли).

В ПОИСКАХ СПЕЦИФИКАЦИЙ

Поскольку спецификацию HTML5 разрабатывают одновременно W3C и WHATWG, существует несколько версий. Версии WHATWG можно считать «инкубаторами идей».

<http://www.w3.org/TR/html5/> — это официальная версия W3C, а <http://dev.w3.org/html5/spec/> — последняя рабочая версия, которая еще может измениться.

Группа WHATWG решила отказаться от номеров версий, поэтому пятерка в названии пропала; это теперь просто «HTML — живой стандарт». С этим источником жизни вы можете ознакомиться на сайте <http://whatwg.org/html>, но будьте осторожны: здесь вы найдете чрезвычайно экспериментальные идеи. Не думайте, что если о чем-то говорится в этом документе, то это хоть где-то реализовано или хотя бы детально продумано. Но в этой спецификации есть полезные примечания по поводу реализации свойств в различных браузерах.

Существует однастраничная версия полной спецификации WHATWG под названием «Web Applications 1.0», которая включает все, что было добавлено WHATWG (<http://www.whatwg.org/specs/web-apps/current-work/complete.html>), но она может «подвесить» ваш браузер, так как она очень объемная и содержит много скриптов.

Множество спецификаций представляют собой алгоритмы, предназначенные для тех, кто реализует HTML в своих продуктах (например, для разработчиков браузеров). Мы всегда пользуемся самой полезной версией для Сети, расположенной по адресу <http://developers.whatwg.org>, в которой нет заумных вещей для разработчиков, а остальные красиво оформлены с помощью CSS авторства Бена Шварца. Но там тоже есть экспериментальные штуки.

Запутались? На http://wiki.whatwg.org/wiki/FAQ#What_are_the_various_versions_of_the_spec.3F можно найти список и описание различных версий.

Геолокация не относится к спецификации WHATWG и потому хранится отдельно: <http://www.w3.org/TR/geolocation-API/>.

Хорошие идеи реализовывались, плохие отвергались независимо от того, кто подал их, какую сторону они представляли и где впервые прозвучали. Помимо этого, множество хороших идей было позаимствовано из Twitter, блогов и IRC.

В 2009 году W3C прекратила работу над XHTML 2.0 и направила все ресурсы на HTML5. Тогда стало ясно, что HTML5 победил в битве мыслителей: чистота дизайна, даже за счет нарушения обратной совместимости, против практичности и «неразрушения Сети». Тот факт, что в рабочие группы HTML5 входили представители самых разных производителей браузеров, тоже оказался важным. Если производители не хотели внедрять какую-то часть спецификации (например, Microsoft выступала против реализации элемента `<dialog>`, а Mozilla — против `<bb>`), они этого не делали. Хиксон сказал следующее: «Реальность такова, что производители браузеров обладают решающим правом накладывать вето на все, что есть в спецификации, так как если они этого не реализуют, спецификацию можно будет отнести разве что к беллетристике». Многим участникам это показалось в высшей степени неприятным: они жаловались, что производители браузеров захватили «нашу Сеть», и в чем-то были правы.

Справедливости ради следует сказать, что отношения между W3C и WHATWG были не такими уж гладкими, как могли бы быть. W3C исповедует подход, основанный на всеобщем

согласии, тогда как Хиксон продолжал вести себя так, как он делал это в WHATWG, — как доброжелательный диктатор (и многие не одобрили бы использования слова *доброжелательный* в таком контексте). Ясно, что у Хиксона было очень четкое понимание того, как нужно разрабатывать язык.

ПРИНЦИПЫ, СКРЫВАЮЩИЕСЯ ЗА HTML5

За HTML5 скрывается ряд утвержденных принципов разработки (<http://www.w3.org/TR/html-design-principles/>). У HTML5 есть три основных цели:

- установить особенности современных браузеров, в которых возможна интероперабельность;
- впервые определить обработку ошибок;
- доработать язык для упрощения создания веб-приложений.

Не разрушать существующие веб-страницы

Многие современные методы разработки основываются на недокументированных (или, как минимум, не указанных в спецификации) свойствах, которые с течением времени встраивались в браузеры. Например, XMLHttpRequest (XHR) обеспечивает работу несметного числа сайтов на основе Ajax. Эта технология была изобретена компанией Microsoft, затем переработана и встроена во все другие браузеры, но никогда не была определена как стандарт (до тех пор, пока Анне ван Кестерен из Opera наконец не определила его как часть WHATWG). Подумать только, такая жизненно необходимая часть многих сайтов реализована как побочный продукт! Поэтому одной из первых задач HTML5 было создание документации для того, что еще не задокументировано, чтобы тем самым повысить интероперабельность, избавляя авторов Сети и разработчиков браузеров от работы «вслепую».

Также необходимо было однозначно определить, как браузеры и другие агенты пользователей должны поступать с неправильной разметкой. В мире XML это не было проблемой; в XML используется «драконовская обработка ошибок», при которой браузер должен остановить процесс отображения при обнаружении ошибки. Одной из причин повсеместного распространения и успеха Сети было (по моему мнению) то, что даже для плохого кода вероятность, что он будет отображаться в некоторых или всех браузерах, была ненулевой. Барьер, который необходимо было преодолеть для публикации в Сети, был очень низким, но каждый браузер был вправе решать, как он будет отображать плохой код. Простой код, такой как `<i>Hello mum!</i>` (обратите внимание на неправильно расположенные закрывающие теги), создает разные объектные модели документов (DOM) в разных браузерах. Из-за этого одни и те же таблицы стилей могут отображаться совершенно по-разному, а написание JavaScript-кода, работающего во всех браузерах, может стать гораздо более сложной задачей. Непротиворечивые объектные модели документов (DOM) настолько важны для HTML5, что сам язык определен в терминах DOM.

В интересах повышения интероперабельности важно, чтобы обработка ошибок была одинаковой во всех браузерах и таким образом создавала абсолютно такую же DOM даже для невалидного HTML. Чтобы это было так, кто-то должен был это определить. Как мы

уже сказали, спецификация HTML5 занимает более 700 печатных страниц, но только 300 из них имеют отношение к разработчикам сайтов (то есть к нам с вами); остальная часть нацелена на разработчиков браузеров и в ней *точно* описывается, как проводить разбор разметки, в том числе неправильной.

Веб-приложения

Растущее число сайтов в Сети представляют собой то, что мы будем называть веб-приложениями; это значит, что они имитируют настольные приложения, а не традиционные статические документы с текстом, картинками и ссылками, которые составляют большую часть Сети. Сюда относятся, например, размещенные в Сети текстовые редакторы, инструменты для обработки фотографий, сайты с картами и т. д. Интенсивно используя JavaScript, эти сайты исчерпали все возможности HTML 5. В HTML5 заданы новые DOM API для событий сервера, рисования, видео, операций перетаскивания и т. п. Эти новые интерфейсы, которые HTML делает доступными для JavaScript с помощью объектов DOM, упрощают процесс написания таких приложений с использованием жестко заданных стандартов, а не едва задокументированных приемов.

Еще более важной является необходимость открытого стандарта (бесплатного для использования и реализации), который мог бы соперничать с частными патентованными стандартами, такими как Adobe Flash или Microsoft Silverlight. Независимо от того, что вы думаете по поводу этих конкретных технологий и компаний, мы считаем, что Сеть — слишком важная платформа, и поэтому общество, коммерция и коммуникация не должны оказаться в руках одного производителя. Насколько иначе проходила бы эпоха Возрождения, если бы Кекстон сохранил патент и монополию на создание типографий?

Не разрушайте Сеть

В Сети существует тьма-тьмущая веб-страниц, и они обязательно должны и дальше правильно отображаться. Так что HTML5 — это, по сути, расширенный вариант HTML 5, и в нем все еще определены правила работы браузеров с устаревшей разметкой (например, с ``, `<center>` и другими тегами представления), поскольку их используют миллионы веб-страниц. Но так как они являются устаревшими, они не должны использоваться. Для веб-разработчиков актуальна семантическая разметка, хотя у каждого читателя появится свое мнение о том, достаточно ли в HTML5 семантики или в нем слишком много элементов. И еще: однозначные правила парсинга, описанные в спецификации HTML5, должны обеспечить интероперабельность работы древних веб-страниц даже тогда, когда для всех документов будет использоваться встроенный в каждый браузер парсер HTML5.

Как насчет XML?

HTML5 не является XML-языком (и это даже не SGML-язык, если это вам о чем-то говорит). Он *обязан* обрабатываться как `text/html`. Однако если вам все же требуется использовать XML, существует XML-сериализация под названием XHTML5. В ней возможны все те же свойства, но (вполне логично) она требует более строгого синтаксиса (если вы привыкли к XHTML, то вы увидите, что они ничем друг от друга не отличаются). Это *должен*

быть правильно построенный XML, и для него *должен* использоваться MIME-тип XML, даже если IE8 и предшественники не могут его обрабатывать (он предлагает выполнить загрузку вместо того, чтобы отобразить документ). Вот почему в этой книге мы используем синтаксис HTML, а не XHTML.

ПОДДЕРЖКА HTML5

HTML5 развивается очень быстро. Спецификация W3C вышла на финальный уровень «last call» в мае 2011 года, однако браузеры начали внедрять поддержку HTML5 (и особенно API) задолго до этого. Поддержка HTML5 будет постепенно улучшаться по мере того, как в браузерах начнут реализовываться специальные возможности, поэтому на встречающуюся время от времени в книге фразу «это поддерживается только в браузере X» можно будет обращать все меньше внимания — и это прекрасно.

Новые браузерные возможности превосходны, и уже создан ряд веб-сайтов, которые по задумке создателей должны «тестировать поддержку HTML5 в браузерах». Создатели большинства таких сайтов просто смешивают в кучу все подряд спецификации; они выявляют HTML5, связанные спецификации из WHATWG, такие как Web Workers, а затем, опьяниенные громкими словами, хватаются за WebGL, SVG, W3C File API, медиазапросы и даже некоторые запатентованные хитрые штучки Apple — в этот момент им обычно перестает хватать воздуха, и они срочно отправляются в постель, чтобы передохнуть пару часиков.

Не следует обращать на эти сайты слишком много внимания. Их системы оценки ни на чем не основаны, а определения HTML5 не имеют смысла и только вводят в заблуждение.

Как точно заметил Патрик Лауке, наш технический редактор: «HTML5 — это не гонка. Смысл не в том, что первый браузер, реализующий эту спецификацию, завоюет Интернет. Основная задача разработки спецификаций — сделать так, чтобы все браузеры единообразно поддерживали один и тот же набор возможностей».

Если вам интересно, на какой стадии находится поддержка новых потрясающих веб-технологий, то рекомендуем добавить в закладки сайт <http://caniuse.com> Алексиса Девериа.

Давайте перейдем к делу

Итак, вот и закончился урок истории с примесью философии. Вот почему спецификация HTML5 иногда умышленно вступает в противоречия с другими спецификациями — ради обратной совместимости она часто описывает то, что браузеры реально делают, а не то, что, по мнению RFC, они должны делать. И именно поэтому HTML5 часто напоминает клудж или компромисс — так и есть. Это та цена, которую мы платим за интероперабельную открытую сеть, и тогда ее авторы восклицают: «Ура практичности!»

Пристегнули ремень?

Поехали!

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу электронной почты comp@piter.com (издательство «Питер», компьютерная редакция).

На веб-сайте издательства (www.piter.com) вы можете скачать архив с дополнительными материалами к этой книге.

Мы будем рады узнать ваше мнение!

На веб-сайте издательства <http://www.piter.com> вы найдете подробную информацию о наших книгах.

Глава 1

Основная структура

Брюс Лоусон



Хотя большинство возможностей HTML5 связано с новыми API, многое, тем не менее, будет интересно как тем, кто занимается версткой, так и тем, кто отвечает за написание скриптов. В HTML5 добавлены 30 элементов с новой семантикой, которые можно использовать при создании традиционных «статических» веб-страниц, а также ряд элементов управления формами, позволяющих полностью отказаться от необходимости валидации форм с помощью JavaScript.

Давайте сразу обратимся к коду. В этой главе мы попытаемся превратить текущую разметку обычной страницы, использующую элементы `<div>`, в семантически организованную систему. Новые структурные элементы HTML5, такие как `<nav>`, `<header>`, `<footer>`, `<aside>`, `<article>` и др., позволяют выделить определенные типы содержимого. Мы посмотрим, как эти элементы работают, и поговорим об однозначности структуры HTML5-документа и его «семантичности».

<HEAD>

Итак, тип документа:

```
<!DOCTYPE html>
```

Готово. И никаких громоздких строк, которые даже самым опытным веб-разработчикам приходится вырезать и вставлять. Никаких URL-адресов. Никаких номеров версий. Ничего лишнего. Такое объявление типа скорее напоминает не правило, а заклинание: оно предназначено для тех браузеров, которым для перехода в стандартный режим требуется указание типа документа, а такой код является одновременно и кратким, и надежным. Мы записали его в верхнем регистре — для совместимости как с HTML, так и с XML, и я рекомендую вам поступать так же.

Далее определим кодировку символов документа. Если этого не сделать, появляется неясная, но реальная угроза безопасности (см. <http://code.google.com/p/doctype/wiki/ArticleUtf7>). Кодировка должна быть указана в первых 512 байтах документа; лучше всего использовать UTF-8 (если нет серьезных оснований для выбора другой кодировки).

```
<!DOCTYPE html>
<meta charset=utf-8>
```

Посмотрите внимательно на тег `<meta>`. Если вы привыкли к XHTML, то заметите три странности. Во-первых, этот тег гораздо короче (раньше он бы выглядел так: `<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">`). Старый вариант остается допустимым, однако новый использовать все же предпочтительнее, поскольку он везде работает и его проще напечатать.

Вы также заметите, что я не использовал кавычки для атрибута `charset="utf-8"`. Кроме того, я не закрыл тег `<meta charset=utf-8 />`.

Поскольку HTML5 не является XML-языком, всего этого делать не нужно. Однако это и не запрещено. Все варианты, приведенные ниже, одинаково допустимы.

```
<META CHARSET=UTF-8>
<META CHARSET=UTF-8 />
<META CHARSET="UTF-8">
```

```
<META CHARSET="UTF-8" />
<meta charset=utf-8>
<meta charset=utf-8 />
<meTa CHARset="utf-8">
<meTa CHARset="utf-8" />
```

Откуда такая свобода в синтаксисе? Ответ прост: для браузеров синтаксис XHTML-документов типа `text/html` никогда не был важен — его проверкой занимался валидатор XHTML. Поэтому, если бы синтаксис HTML5 требовал соблюдения каких-то конкретных правил, набор этих правил был бы полностью произвольным, а при их несоблюдении страница, будучи невалидной, все равно правильно отображалась бы во всех браузерах. Синтаксис HTML5 совершенно все равно, какой синтаксис вы используете.

ВЫБЕРИТЕ СТИЛЬ И ПРИДЕРЖИВАЙТЕСЬ ЕГО

Несмотря на то что в HTML5 допустимы все приведенные выше варианты, стоит все же отдавать предпочтение какому-то одному из них. Использование разного синтаксиса, как правило, оказывается настоящим кошмаром, особенно если в создании проекта принимает участие большая группа разработчиков.

Лучше всего выбрать стиль, который наиболее удобен для вас, и придерживаться его. При этом неважно, что вы выберете: Реми следует правилам синтаксиса XHTML, а Брюс предпочитает строчные буквы и минимизацию атрибутов (`controls` вместо `controls="controls"`) и использует кавычки для атрибутов только при необходимости, например при добавлении двух классов к элементу (`<div class=important>`, но `<div class="important logged-in">`). В этой книге вы найдете примеры обоих стилей, поскольку каждый из ее авторов привык работать в наиболее комфортном для него режиме. Поэтому вам следует научиться читать код, написанный в этих двух стилях.

Так как вы только начинаете работать с HTML5, вы абсолютно свободны в выборе своего собственного стиля — но сделав этот выбор, оставайтесь верным ему.

Поскольку речь здесь идет о невероятной свободе в синтаксисе (в сравнении с XHTML), давайте попробуем нарушить все существующие правила и после задания заголовка документа перейдем сразу к контенту.

```
<!DOCTYPE html>
<meta charset=utf-8>
<title>Interesting blog</title>
<p>Today I drank coffee for breakfast.
- 14 hours later, I went to bed.</p>
```

Как ни странно, этот увлекательный блог прекрасно проходит валидацию, хотя в нем нет ни тега `<html>`, ни тега `<head>`, ни тега `<body>` (рис. 1.1).

Перед нами, возможно, один из тех моментов «WTF? Какого черта?», о которых говорилось во введении. Эти три элемента (авторам XHTML лучше сесть) являются совершенно необязательными, поскольку браузеры в любом случае считают, что они есть. Чтобы найти этому подтверждение, заглянем «под капот» браузера с помощью Opera Dragonfly (рис. 1.2).

То же самое можно сделать с помощью Internet Explorer 6 Developer Tools (рис. 1.3).

The document is valid HTML5 + ARIA + SVG 1.1 + MathML 2.0
(subject to the utter previewness of this service).

Source

```

1. <!DOCTYPE html> -
2. <meta charset=utf-8> -
3. <title>Interesting blog</title> -
4. <p>Today I drank coffee for breakfast. 14 hours later, I went to bed.</p>

```

Рис. 1.1. Потрясающий факт: документ без тегов `<head>`, `<body>` и `<html>` успешно проходит валидацию

Такое поведение браузеров и привело к тому, что в HTML5 эти теги не требуются. Однако если вы все же не будете их использовать, это может сбить с толку некоторых ваших коллег. Кроме того, если вы планируете использовать AppCache (см. главу 7), то элемент `<html>` обязательно должен присутствовать в разметке. Также здесь удобно задавать основной язык документа:

```
<html lang=en>
```

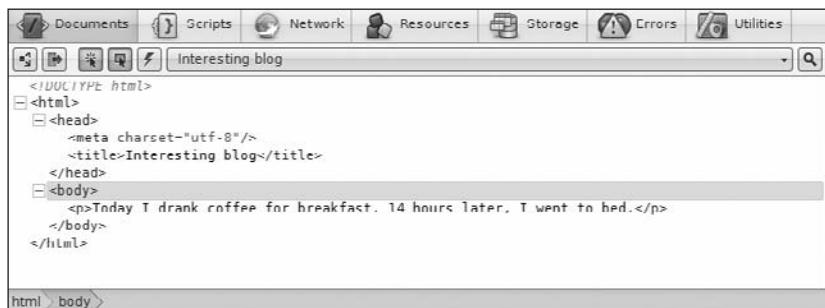


Рис. 1.2. С помощью отладчика Opera Dragonfly мы видим, что браузеры сами добавляют недостающие элементы

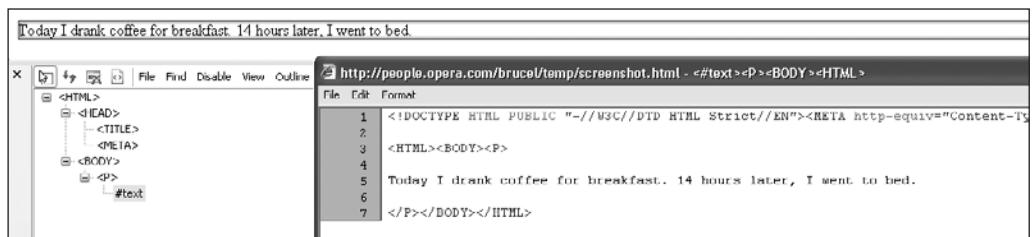


Рис. 1.3. Internet Explorer 8, как и все другие браузеры, добавляет недостающие элементы в DOM (однако IE, похоже, меняет местами элементы `<title>` и `<meta>`)

Пользователи с нарушениями зрения могут работать с вашим веб-сайтом с помощью программы чтения экрана — она произносит текст на странице синтезированным голосом. Когда программа чтения экрана встречает строку «six», она может «произнести» ее

по-разному, в зависимости от того, считается основным языком страницы английский или французский. Подобные программы пытаются угадать язык содержимого, но гораздо лучше указывать его явно, как я сделал выше.

Браузеру IE версии 8 и более ранних для применения CSS для оформления новых элементов HTML5 требуется элемент `<body>`, поэтому о данном элементе также не следует забывать.

Для максимальной надежности добавим эти необязательные элементы, чтобы добиться по возможности минимальной обработки HTML5-страницы.

```
<!DOCTYPE html>
<html lang=en>
<head>
<meta charset=utf-8>
<title>Interesting blog</title>
</head>
<body>
<p>Today I drank coffee for breakfast. 14 hours later,
- I went to bed.</p>
</body>
</html>
```

ВАЛИДАЦИЯ ВСЕ ЕЩЕ ИМЕЕТ ЗНАЧЕНИЕ?

При таком свободном синтаксисе мы можем опускать подразумеваемые теги (такие как `<html>`, `<head>` и `<body>`). Что гораздо важнее, поскольку HTML5 создает согласованную объектную модель документа (DOM) для каждого случая плохой разметки, вам будет простиительно задуматься над тем, имеет ли валидация все еще какое-то значение. Мы тоже задаем себе этот вопрос.

Мы пришли к выводу, что валидация всегда была лишь инструментом обеспечения качества, средством для достижения цели, а не самой целью.

Целью же является семантическая разметка: мы пытаемся добиться того, чтобы выбор элементов определялся значением содержимого, а не его внешним представлением. Можно создать веб-страницу в полном соответствии с правилами, используя только таблицы, элементы `div` и `span`, но такая разметка не будет иметь отношения к семантике. И наоборот, один незакодированный амперсанд может сделать прекрасно структурированную и семантически богатую веб-страницу невалидной, но при этом она останется семантической.

Когда мы руководим группами разработчиков, мы требуем, чтобы код обязательно проходил валидацию перед любой проверкой, не говоря уже о создании «живого» кода. Это отличный способ убедиться в том, что ваш код работает правильно. Ведь несмотря на то, что браузеры создают объектную модель документов (DOM) и для плохой разметки, это может быть не та модель, которая вам нужна.

Кроме того, парсеры HTML5 еще не добавлены во все существующие браузеры, поэтому забота о правильности страниц является залогом предсказуемого поведения CSS и JavaScript.

Мы используем валидаторы <http://validator.w3.org> и <http://html5.validator.nu/>. И мы рассчитываем на то, что в дальнейшем валидаторы будут развиваться: появятся такие возможности, как усиление требований к синтаксису, чтобы при желании вы могли получать уведомления о несоответствии кода, к примеру, стандартам XHTML, хотя спецификация этого не требует. Хорошее впечатление производит <http://lint.brihten.com>, но мы не можем быть уверены, что он использует актуальные процедуры валидации.

ИСПОЛЬЗОВАНИЕ НОВЫХ СТРУКТУРНЫХ ЭЛЕМЕНТОВ HTML5

В 2004 году главный редактор спецификации HTML5 Ян Хиксон проиндексировал с помощью Google 1 млрд веб-страниц, чтобы посмотреть, из чего сделана «реальная» сеть. Одна из статей, в которых впоследствии были опубликованы результаты (<http://code.google.com/intl/ru-RU/webstats/2005-12/classes.html>), содержит список самых часто используемых имен классов в этих HTML-документах.

Совсем недавно, в 2009 году, поисковая система Opera MAMA просмотрела атрибуты классов в 2 148 723 случайно выбранных URL, а также атрибуты `id` различных элементов (результаты индексирования Google не содержат аналогичных данных) в 1 806 424 URL (см. табл. 1 и табл. 2).

Таблица 1.1. Имена классов

Популярность	Значение	Частота
1	footer	179 528
2	menu	146 637
3	style1	138 308
4	msonormal	123 374
5	text	122 911
6	content	113 951
7	title	91 957
8	style2	89 851
9	header	89 247
10	copyright	86 979
11	button	81 503
12	main	69 620
13	style3	69 349
14	small	68 995
15	nav	68 634
16	clear	68 571
17	search	59 802
18	style4	56 032
19	logo	48 831
20	body	48 052

Если исключить из рассмотрения классы, явно относящиеся к оформлению, мы можем получить представление о структурах, которые авторы пытаются реализовать при создании веб-страниц.

Таблица 1.2. Имена id

Популярность	Значение	Частота
1	footer	288 061
2	content	228 661
3	header	223 726
4	logo	121 352
5	container	119 877
6	main	106 327
7	table1	101 677
8	menu	96 161
9	layer1	93 920
10	autonumber1	77 350
11	search	74 887
12	nav	72 057
13	wrapper	66 730
14	top	66 615
15	table2	57 934
16	layer2	56 823
17	sidebar	52 416
18	image1	48 922
19	banner	44 592
20	navigation	43 664

Точно так же, как в HTML 5 обнаруживаются характерные черты раннего поколения Сети — мира ученых и инженеров (именно поэтому в нем есть такие элементы, как `<kbd>`, `<samp>` и `<var>`), HTML5 служит отражением состояния Сети в период его разработки: добавлено 30 новых элементов, и выбор многих из них основан на данных вышеприведенного списка имен классов и `id`, поскольку это именно то, что разработчики на самом деле используют.

Пока нас интересуют скорее практические цели, чем философские размышления, давайте будем их использовать. В качестве примера здесь приводится разметка главной страницы блога, выполненная в стиле HTML 5 с использованием семантически нейтрального элемента `<div>`.

```
<div id="header">
  <h1>My interesting life</h1>
</div>
<div id="sidebar">
  <h2>Menu</h2>
  <ul>
```

```

<li><a href="last-week.html">Last week</a></li>
    <li><a href="archive.html">Archives</a></li>
</ul>
</div>
<div class="post">
    <h2>Yesterday</h2>
    <p>Today I drank coffee for breakfast. 14 hours later,
       - I went to bed.</p>
</div>
<div class="post">
    <h2>Tuesday</h2>
    <p>Ran out of coffee, so had orange juice for breakfast.
       - It was from concentrate.</p>
</div>
<div id="footer">
    <p><small> This is copyright by Bruce Sharp. Contact me to
       - negotiate the movie rights.</small></p>
</div>
```

Применив простые таблицы стилей, зададим оформление.

```
#sidebar {float:left; width:20%;}
.post {float:right; width:79%;}
#footer {clear:both;}
```

Схематическое изображение нашей страницы показано на рис. 1.4.

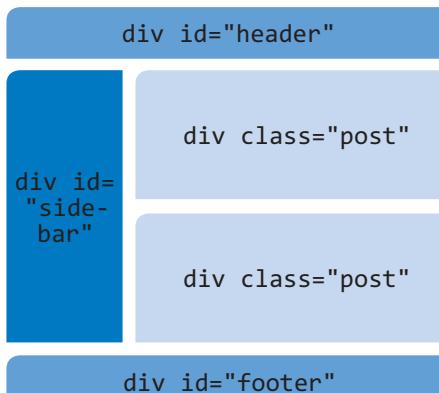


Рис. 1.4. Структура нашего блога в HTML 5

Хотя в этой разметке нет ничего совсем уж неправильного (и она будет прекрасно работать в новом мире HTML5), почти вся структура документа оказывается неизвестной браузеру, так как единственный HTML-элемент, который мы можем использовать для выделения отдельных фрагментов страницы, — это семантически нейтральный `<div>` (определенный в HTML 5 как «универсальный механизм для оформления структуры документов»).

Но если все отображается правильно, что плохого в данном коде? Зачем использовать дополнительные элементы и добавлять семантику?

Представим себе «умный» браузер, в котором с помощью специального сочетания клавиш можно быстро переходить к панели навигации страницы. Вопрос в следующем: как он узнает, куда переходит? Одни разработчики используют `<div class="menu">`, другие — `class="nav", class="navigation", class="links"` или всевозможные их эквиваленты на разных языках. Данные из приведенных выше таблиц Opera МАМА наводят на мысль о том, что `menu`, `nav`, `sidebar` и `navigation` можно было бы считать синонимами, однако гарантии нет: на веб-сайте ресторана `<div class="menu">` может использоваться не для навигации, а для определения списка блюд.

НУ ПОЧЕМУ ЖЕ ЗДЕСЬ НЕТ ЭЛЕМЕНТА <CONTENT>?

Довольно просто представить себе, как работало бы наше гипотетическое сочетание клавиш «переход к панели навигации», однако гораздо чаще пользователю требуется быстрый переход к основной области контента. Некоторые дизайнеры для удобства работы с сайтом добавляют вверху страницы ссылку «skip links», которая позволяет программам экранного доступа обойти элементы навигации. Правда же, было бы здорово, если бы в браузерах предусматривалось сочетание клавиш, позволяющее сразу же перескочить к основному содержимому?

Но ведь в HTML5 нет элемента `<content>`, к которому должен осуществляться переход! Как в таком случае браузер узнает, где начинается основная область контента?

На самом деле определить это очень просто: для этого можно использовать «алгоритм Скуби-Ду» (как я его называю). Всегда очевидно, что в костюме привидения скрывается угрюмый привратник заброшенного парка развлечений, просто потому, что он единственный персонаж серии, который не является Фредом, Дафной, Велмой, Шэгги и Скуби. Точно так же первый фрагмент контента, не заключенный в теги `<header>`, `<nav>`, `<aside>` или `<footer>`, является признаком начала основной области контента, и при этом не важно, находится ли он внутри элемента `<article>` или `<div>` или же является прямым потомком элемента `<body>`.

Это было бы удобно для пользователей программ экранного доступа, а также для производителей мобильных устройств, которые бы заставляли браузеры сразу переходить к основному содержимому страниц.

Если же вас не оставляет тоска по несуществующему элементу `<content>`, используйте WAI-ARIA и добавьте `role=main` к тегу, обрамляющему основной контент (более подробно о WAI-ARIA рассказывается в главе 2). Кстати, так вы заодно добавите якорь для стилизации и сможете использовать селекторы атрибутов CSS (недоступны в IE6), например, `div[role=main]{float:right;}`.

В HTML5 есть новые элементы, которые однозначно задают структурные элементы страницы. Используя некоторые из них, попробуем переписать нашу разметку.

```
<header>
  <h1>My interesting life</h1>
</header>
<nav>
  <h2>Menu</h2>
  <ul>
    <li><a href="last-week.html">Last week</a></li>
    <li><a href="archive.html">Archives</a></li>
  </ul>
</nav>
```

```

<article>
  <h2>Yesterday</h2>
  <p>Today I drank coffee for breakfast. 14 hours later,
    - I went to bed.</p>
</article>
<article>
  <h2>Tuesday</h2>
  <p>Ran out of coffee, so had orange juice for breakfast.
    - It was from concentrate.</p>
</article>

<footer>
  <p><small>This is copyright by Bruce Sharp. Contact me to
    - negotiate the movie rights.</small></p>
</footer>

```

Другое схематическое изображение нашей страницы (с использованием HTML5) показано на рис. 1.5.

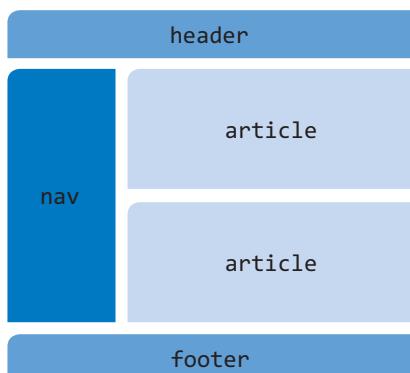


Рис. 1.5. HTML5-структура нашего блога

Перед тем как перейти к обсуждению значения и сферы применения этих новых элементов, давайте создадим оформление для основных частей страницы.

ОФОРМЛЕНИЕ HTML5-ДОКУМЕНТА С ПОМОЩЬЮ CSS

Во всех браузерах (за исключением одного) создавать стили для этих элементов очень просто: CSS можно применять к произвольному элементу, поскольку, как сказано в спецификации, CSS — это «язык таблиц стилей, позволяющий авторам и пользователям добавлять оформление... к структурированным документам (таким как HTML-документы и XML-приложения)», а в XML-приложениях могут использоваться любые элементы.

Таким образом, с помощью CSS можно сделать элемент `<nav>` плавающим, добавить границы для элементов `<header>` и `<footer>` и оформить поля и отступы для `<article>`, причем это почти так же просто, как и оформление элементов `<div>`.

Хотя новые HTML5-элементы можно использовать уже сейчас, они не всегда понятны более старым версиям браузеров. В таком случае они рассматриваются, как неизвестные элементы, и никак особо не обрабатываются.

Необычным может показаться то, что в CSS все элементы по умолчанию являются строчковыми (`display:inline`), поэтому, если просто задать ширину и высоту для структурного элемента (как мы это обычно делаем для элементов `<div>`), в устаревших браузерах он будет отображаться неправильно, и нам придется явно указать, что элемент является блочным (`display:block`). В нынешних браузерах до сих пор используетсяrudиментарная встроенная таблица стилей, переопределяющая строковую стилизацию по умолчанию для элементов, являющихся, *по нашему мнению*, исконно блочными (пример такой таблицы стилей можно найти на <http://www.w3.org/TR/CSS2/sample.html>). В старых браузерах нет правил превращения новых элементов HTML, таких как `<header>`, `<footer>`, `<nav>` и `<article>`, в блоковые (`display:block`), поэтому для них способ отображения должен быть задан вручную в таблице стилей. Для современных браузеров эти строки кода будут безвредными, а заодно послужат резервным вариантом для старых браузеров, которые, как показывает практика, могут использоваться еще довольно долго.

Итак, чтобы оформить наш HTML5-документ в соответствии с дизайном HTML 5, нам понадобятся следующие стили.

```
header, nav, footer, article {display:block;}
nav {float:left; width:20%;}
article {float:right; width:79%;}
footer {clear:both;}
```

И вот на свет появляется прекрасная HTML5-страница. Во всех браузерах, кроме одного.

Создание CSS для Internet Explorer 6, 7, 8

В устаревших (но, к сожалению, не исчезнувших с лица земли) версиях браузера Internet Explorer стили CSS правильно применяются к элементам HTML 5, которые IE поддерживает, но любые новые элементы HTML5, незнакомые браузеру, остаются неоформленными. Это выглядит... несимпатично.

Обмануть старый недобрый IE и заставить его применять стили CSS к элементам HTML5 можно с помощью JavaScript. Почему? Это загадочное явление, и если мы раскроем вам секрет, нам придется вас убить. (На самом деле мы просто этого не знаем.) Если добавить следующий код в элемент `head` документа, IE таинственным образом применит стили к этим элементам (при условии, что в разметке есть элемент `<body>`).

```
<script>
  document.createElement('header');
  document.createElement('nav');
  document.createElement('article');
  document.createElement('footer');
</script>
```

Функцию `document.createElement()` нужно применить к каждому элементу только один раз (независимо от того, сколько раз он встречается на странице).

Помните, что хотя HTML5 не требует использования элемента `<body>`, этого нельзя сказать о такой невероятной смеси из Internet Explorer 8 (и более ранних версий), CSS, HTML5 и JavaScript. IE9 работает аналогично остальным браузерам и не требует помощи JavaScript.

ПРИМЕЧАНИЕ

Элемент `<script>` теперь не требует объявления типа скрипта — JavaScript выбирается по умолчанию. Это верно и для устаревших браузеров, так что вы можете спокойно использовать такой синтаксис.

Хотя для других браузеров такой JavaScript-код не требуется, он будет для них безвредным. Но для оптимизации скорости загрузки страницы можно заключить этот код в условные комментарии (см. <http://dev.opera.com/articles/view/supporting-ie-with-conditional-comments/>), и тогда он будет загружаться и выполняться только в IE.

СЦЕНАРИЙ ВКЛЮЧЕНИЯ

В качестве альтернативного варианта можно воспользоваться крохотным сценарием включения HTML5 авторства Реми, который вы найдете на сайте <http://remysharp.com/2009/01/07/html5-enabling-script/>. Он в один прием выполняет необходимую настройку для всех новых элементов, а также содержит дополнительную утилиту Джона Нила IE Print Protector (<http://www.iecss.com/print-protector>), которая гарантирует, что стили будут правильно применяться к элементам HTML5 не только при отображении, но и при печати документов в IE.

Если у пользователя отключен JavaScript (самостоятельно или в соответствии с корпоративной политикой безопасности), он сможет получить доступ к содержимому, но при этом страница будет не полностью оформленной. Однако это не всегда является решающим фактором. (У пользователя древней версии IE без JavaScript и так создается достаточно унылое впечатление от Сети, так что ваш веб-сайт вряд ли покажется ему самым уродливым.) Саймону Питерсу удалось показать, что, зная, как выглядит DOM, можно создать оформление для HTML5-документа и без JavaScript, однако этот метод ограничен и не очень-то удобен. (См. «Добавление стилей к HTML5-разметке без использования скриптов» (<http://blog.whatwg.org/styling-ie-noscript>)).

ДРУГИЕ ПРОБЛЕМЫ УСТАРЕВШИХ БРАУЗЕРОВ

При создании стилей для HTML5-документов возникают другие проблемы, связанные с устаревшими браузерами. В старых версиях Firefox (до версии 3) и Camino (до версии 2) существует ошибка, о которой подробно рассказывается в статье по адресу <http://html5doctor.com/how-to-get-html5-working-in-ie-and-firefox-2/>.

Мы не собираемся приводить здесь исчерпывающий список таких особенностей; это временные проблемы, которые, как мы надеемся, будут решены с появлением новых версий.

Когда используются новые структурные элементы HTML5

Мы уже использовали эти элементы для написания разметки нашей страницы и даже добавили к ним стили. И хотя о том, как они используются, можно догадаться по названиям, пришло время узнать об этом более подробно.

<HEADER>

В приведенном ранее примере, как и на большинстве сайтов, `header` является первым элементом на странице и содержит название сайта, логотипы, ссылки на главную страницу и т. д. В спецификации говорится следующее:

Элемент `header` представляет собой набор предварительных данных и навигационных средств... Обратите внимание: предполагается, что элемент `header` должен содержать заголовок раздела (элементы `h1-h6` или `hgroup`), но это не обязательно. Элемент `header` может также использоваться для обрамления списка содержимого раздела, формы поиска или любых необходимых логотипов.

Давайте проанализируем это описание. Во-первых, элемент `header` не является обязательным; в приведенном выше примере он избыточен, поскольку включает только элемент `<h1>`. Но его значение — объединять «предварительные данные и навигационные средства», так что приведем более жизненный пример.

```
<header>
  <a href="/"><img src=logo.png alt="home"></a>
  <h1>My interesting blog</h1>
</header>
```

На многих сайтах используется заголовок и подзаголовок. Чтобы скрыть подзаголовок от алгоритма определения структуры документа (и таким образом объединить заголовок и подзаголовок в одну логическую единицу; см. главу 2), основной заголовок и подзаголовок можно сгруппировать с помощью нового элемента `<hgroup>`.

```
<header>
  <a href="/"><img src=logo.png alt="home"></a>
  <hgroup>
    <h1>My interesting blog</h1>
    <h2>Tedium, dullness and monotony</h2>
  </hgroup>
</header>
```

Элемент `header` может также содержать панель навигации. Это может оказаться полезным при использовании панели глобальной навигации, особенно для сайтов, основанных на шаблоне, где весь элемент `<header>` может быть добавлен из файла шаблона. Таким образом, например, устроена горизонтальная панель навигации на сайте <http://www.thaicookery.co.uk/> (рис. 1.6).

```
<header>
  <hgroup>
    <h1>Thai Cookery School</h1>
    <h2>Learn authentic Thai cookery in your own home.</h2>
  </hgroup>
  <nav>
    <ul>
      <li>Home</li>
      <li><a href="courses.html">Cookery Courses</a></li>
      <li><a href="contact.html">Contact</a></li>
    </ul>
  </nav>
</header>
```



Рис. 1.6. Элемент `<header>` сайта <http://www.thaicookery.co.uk/>

Конечно же, элемент `<nav>` не обязательно должен находиться внутри тега `<header>`. Разметку для сайта Thai cookery можно написать и по-другому, поместив основную панель навигации вне этого элемента.

```
<header>
  <hgroup>
    <h1>Thai Cookery School</h1>
    <h2>Learn authentic Thai cookery in your own home.</h2>
  </hgroup>
</header>
<nav>
  <ul>
    <li>Home</li>
    <li><a href="courses.html">Cookery Courses</a></li>
    <li><a href="contact.html">Contact</a></li>
  </ul>
</nav>
```

Выбор в значительной степени зависит от того, считаете ли вы, что панель навигации по сайту является частью элемента `<header>`, и от практических соображений, касающихся простоты написания стилей. Возьмем в качестве примера мой собственный сайт, в котором

слева расположена очень длинная панель навигации по сайту, которая может оказаться длиннее основного контента. Если в таком случае добавить `<nav>` в `<header>`, будет довольно трудно разместить основной контент справа и добавить футер, поэтому я поместил панель навигации вне элемента `<header>` на одном уровне с `<body>`, как в предыдущем примере (рис. 1.7).

Обратите внимание на то, что сейчас мы занимаемся созданием основного элемента `header` страницы; на самом деле, таких элементов может быть несколько — об этом рассказывается в главе 2.



Рис. 1.7. Типичная страница с панелью навигации по сайту вне элемента `<header>`

<NAV>

Элемент `<nav>` используется для создания панели навигации. Панель навигации состоит из ссылок, позволяющих перемещаться по странице (так, например, элемент оглавления, расположенного в начале статьи, связан с определенным местом на данной странице) или по сайту. Однако не любой набор ссылок является элементом `<nav>`; к навигации, например, не относится список спонсорских ссылок. То же самое касается и списка результатов поиска — он является частью основного содержимого страницы.

Как и в случае с элементами `<header>` и `<footer>` (и со всеми новыми элементами), на странице может быть не один элемент `<nav>`. Вы можете создать панель глобальной навигации в теге `<header>`, а затем использовать элемент `<nav>` для оглавления и списка ссылок на похожие статьи в конце страницы.

Содержимым элемента `<nav>`, скорее всего, будет набор ссылок, для которого в разметке будет использоваться неупорядоченный список (это стало традицией после появления фундаментальной статьи Марка Ньюхауса «Дизайн CSS: Укroщение списков» (<http://www.alistapart.com/articles/taminglists/>)) или упорядоченный список (в случае «хлебных крошек»). Помните, что `<nav>` не заменяет элементы `` и ``, а обрамляет их. Поэтому устаревшие браузеры, не знакомые с этим элементом, отобразят панель навигации просто как обычный список.

<NAV>: ИСПОЛЬЗОВАТЬ ИЛИ НЕТ?

Раньше я страдал «навигационизмом» — стремлением окружать любые ссылки на другие части сайта элементами <nav>.

Однако я излечился от этого порока, задумавшись, кто же выигрывает от использования элемента <nav>. Чуть раньше мы рассуждали о возможности добавления ссылки, которая позволяла бы пользователям технологий чтения экрана быстро переходить к навигационным меню. Если на странице содержится несколько десятков элементов <nav>, то среди них станет трудно находить наиболее важные. Поэтому теперь я выступаю за специальную разметку только для наиболее важных навигационных панелей, например, охватывающих навигацию всего сайта (или раздела), а также для оглавлений очень длинных страниц.

Хорошее эмпирическое правило можно сформулировать так: используйте элемент <nav> только в том случае, если к ссылкам, которые вы собираетесь обрамить этим элементом, можно смело добавлять заголовок «Навигация». Если они настолько важны, что заслуживают заголовка (неважно, требуется ли такой заголовок в действительности в концепции дизайна), значит, они достаточно важны для добавления <nav>.

Как говорит спецификация, «Не все группы ссылок на странице требуется обрамлять элементом <nav> — он предназначен для разделов, состоящих из крупных навигационных блоков».

И наоборот, в спецификации предлагается не использовать элемент <nav> для «юридических» ссылок (авторское право, контакты, свобода информации, политика конфиденциальности и т. д.), которые часто оказываются запрятанными в футере: «Часто в футерах содержится краткий список ссылок на различные страницы сайта (такие как пользовательское соглашение, главная страница и страница с информацией об авторских правах). Для этих случаев достаточно именно элемента <footer>; элемент <nav> может использоваться, но обычно это лишнее».

Мы с этим не согласны. Игнорируйте спецификацию и используйте <nav> для футеров. Многие сайты добавляют ссылку на информацию о доступности, позволяющую получить сведения, например, о том, как запросить данные в другом формате. Тем, кому требуются такие сведения, будет гораздо удобнее работать с сайтом, если агенты пользователей позволят им сразу переходить к элементам <nav>.

Хотя использование списка — достаточно разумная идея (и она дает вам большую свободу в создании таблиц стилей), это не является обязательным. Следующий код абсолютно допустим.

```
<nav>
  <p><a href="/">Home</a></p>
  <p><a href="/about">About</a></p>
</nav>
```

В панели навигации можно использовать заголовки.

```
<nav>
  <h2>Pages</h2>
  <ul>
    <li><a href="/about">About me</a></li>
    <li><a href="/news">News</a></li>
  </ul>
```

```
<h2>Categories</h2>
<ul>
    <li><a href="/happy">Happy Pirates</a></li>
    <li><a href="/angry">Angry Pirates</a></li>
</ul>
</nav>
```

Объединение `<nav>` и других элементов в боковой панели

На многих сайтах можно увидеть боковую панель, в которой находится несколько навигационных блоков, а также контент, не имеющий отношения к навигации. Возьмем для примера мой сайт <http://www.brucelawson.co.uk/> (рис. 1.8).

В боковой панели, расположенной слева от основного контента, содержится одна навигационная панель с несколькими списками: страницы, категории, архивы, новые комментарии. В первом издании этой книги я рекомендовал размечать подобные списки как последовательность следующих друг за другом элементов `<nav>`;

с тех пор мое мнение изменилось, и теперь я обрамляю весь набор списков одним всеобъемлющим `<nav>`. (Если на боковой панели вашего сайта присутствуют два или более важных навигационных блоков, не связанных между собой, то, разумеется, для них следует создавать отдельные элементы `<nav>`.)

Вся основная панель навигации моего сайта содержится в элементе `<aside>`, который «может использоваться для создания типографских эффектов (например, врезок и боковых панелей), для рекламы, для групп элементов `<nav>` и для другого контента, который должен располагаться отдельно от основного» (<http://dev.w3.org/html5/spec/semantics.html#the-aside-element>).



Рис. 1.8. Боковая панель моего блога, на которой (когда-то были) смешаны в одну кучу элементы навигации, выходные данные и фотографии симпатичных мужчин

```
<aside>
<nav>
<h2>Pages</h2>
<ul> .. </ul>
<h2>Categories</h2>
<ul> .. </ul>

<h2>Recent comments</h2>
<ul> ... </ul>
</nav>

<section>
<h2>blah blah</h2>
<a href="...">Web hosting by LovelyHost</a>

<p>Powered by <a href="...">WordPress</a></p>
<p><a href="...">Entries (RSS)</a> and <a href="...">
- Comments (RSS)</a></p>
</section>
</aside>
```

ПРИМЕЧАНИЕ

Прежде чем вы с презрением отбросите эту книгу, возмущившись моей привычкой менять мнение, вспомните непреложную истину: не существует Единственно Верного Способа™ выполнять разметку содержимого. HTML — это общий язык, там нет миллиона элементов, описывающих любые случайности и возможности (хотя иногда так может показаться)!

Обратите внимание на то, что для раздела «Blah Blah» элемент `<nav>` не используется, так как ссылка на мой веб-хост, моя фотография и две RSS-ссылки не являются «разделами, состоящими из навигационных блоков» (так в спецификации определен `<nav>`). Для него используется элемент `<section>`, благодаря чему заголовки боковой панели оказываются на одном уровне в алгоритме определения структуры документа (более подробно об этом рассказывается в главе 2).

`<FOOTER>`

В спецификации говорится, что элемент `<footer>` используется для создания «футера для расположенного уровнем выше структурированного контента или структурированного корневого элемента». («Структурированный контент» — это элементы `article`, `aside`, `nav`, `section`; «структурные корневые элементы» — это `blockquote`, `body`, `details`, `fieldset`, `figure` и `td`.)

Обратите внимание, что, как и в случае с элементом `header`, на странице может быть несколько элементов `footer`; мы вернемся к этому вопросу в главе 2. А пока на нашей странице всего один футер, который является непосредственным потомком элемента `body`. Как сказано в спецификации, «если в качестве расположенного уровнем выше структурированного контента или структурированного корневого элемента выступает элемент `body`, футер применяется ко всей странице».

И далее: «элемент `footer` обычно содержит сведения о соответствующем разделе — информацию об авторе, ссылки на похожие документы, сведения об авторских правах и т. п.»

В нашем футтере тоже есть сведения об авторских правах, которые мы заключили в тег `<small>`. В HTML5 этот элемент был переопределён; раньше он относился к оформлению, а в HTML5 у него появилась семантика, и теперь он используется для примечаний или «разделов, написанных мелким шрифтом»: «мелкий шрифт обычно характерен для отказов, предупреждений, правовых ограничений или авторских прав. Мелкий шрифт также иногда используется для указания авторства или для соблюдения условий лицензирования».

В футтере иногда содержится не только упоминание об авторских правах. Там могут быть ссылки на политику конфиденциальности, информацию о доступности (зачем же ее прятать?) и прочие подобные ссылки. Несмотря на указания спецификации, их следует объединять в элемент `<nav>` (см. предыдущий раздел).

В спецификации говорится, что «на некоторых сайтах используется так называемый «большой футер» — футтер, включающий множество материалов, таких как изображения, ссылки на другие статьи, ссылки на страницы обратной связи, специальные предложения... в определенном смысле, это целая «заглавная страница» в одном футтере». Для обрамления всей этой информации предлагается внутри элемента `<footer>` использовать элемент `<nav>`.

Рассматривая возможность добавления «большого футера», задумайтесь, нужно ли вообще помещать такие ссылки в `<nav>` — от навигационизма может быть не так просто избавиться. Также спросите себя, являются ли вообще эти ссылки частью `<footer>`? Может быть, лучше поместить их на один уровень с футером, но в другой элемент, а именно `<aside>`?

<ARTICLE>

Основная область контента главной страницы этого блога содержит несколько записей. Для каждой из них мы используем элемент `<article>`. Он определяется так: «Элемент `<article>` представляет собой самодостаточный компонент страницы, документа, приложения, сайта — в принципе, автономный и предназначенный для многократного использования и свободного распространения, например, при синдицировании. Это может быть публикация на форуме, статья в журнале или газете, запись в блоге, комментарий пользователя, интерактивный виджет или гаджет и любой другой независимый элемент содержимого». Запись блога, руководство, заметка новостей, комикс или видеозапись с расшифровкой полностью соответствуют этому определению. Не так очевидно то, что данное описание также относится к отдельным сообщениям в почтовом веб-клиенте, к картам и веб-виджетам многократного использования. Под «статьей» (так переводится название `<article>`) не стоит всегда понимать газетную статью — вспомните, например, определение «статья расходов», обозначающее отдельный элемент в наборе. Обратите внимание на то, что, как и в случае с `<nav>`, заголовок должен располагаться внутри элемента, так что следующий код некорректен.

```
<h1>My article</h1>
<article>
  <p>Blah blah</p>
</article>
```

Правильный вариант выглядит так.

```
<article>
  <h1>My article</h1>
  <p>Blah blah</p>
</article>
```

У элемента `<article>` есть и другие интересные особенности, которые (как вы, наверняка, догадались) мы рассмотрим в следующей главе.

В ЧЕМ ЖЕ СУТЬ?

Один мой очень мудрый друг, Робин Берджон, написал: «Наверное, любой человек в сетевом сообществе подтвердит, что „семантика — это чудо, и от нее всем сплошная благодать“. Наверное, это правда. Но если копнуть поглубже, становится очевидно, что лишь немногие способны четко сформулировать, почему это так.

Наиболее общий ответ звучит как «потому что позволяет переориентировать содержимое». Отлично, но переориентировать на что? Например, если вы собираетесь выводить свои веб-страницы на маленькие экраны (это одна из форм переориентирования), то `<nav>`

и `<footer>` помогут отделить служебную информацию от фактического содержимого и отбросить ее. Однако если вы изучаете юридические вопросы, то ковыряние внутри `<footer>` с помощью какой-то эвристической логики вряд ли поможет...

Мне кажется, что HTML должен добавлять только элементы, которые либо обнажают функциональность, которая в противном случае не имела бы смысла (например, `<canvas>`), либо предоставляют семантику, помогающую переориентировать содержимое в целях упрощения его просмотра в разных браузерах» (<http://www.alistapart.com/comments/semanticsinhtml5?page=2#12>).

Как предлагает Робин, устройства с небольшими экранами могут отбрасывать области, не содержащие важного контента (или сразу же переводить фокус на контент). Определенное прикосновение или смахивание может переводить фокус к панели навигации, к заголовку или футеру. Поисковый механизм может присваивать ссылкам в футере меньший вес по сравнению с ссылками в навигационной панели. Наверняка в будущем возникнет множество вариантов использования, о которых мы пока что не догадываемся — однако можно с уверенностью сказать, что они будут зависеть от недвусмысленного определения значения контента, а в этом и заключается смысл семантической разметки.

ЗАКЛЮЧЕНИЕ

В этой главе мы познакомились с HTML5 и его типом документа (DOCTYPE). С помощью элементов `<header>`, `<footer>`, `<nav>`, `<aside>` и `<article>` мы создали структурированную веб-страницу, предоставляющую браузерам семантически более богатую структуру, чем набор лишенных смысла универсальных элементов `<div>`, которые были единственным вариантом в HTML 5; мы также создали оформление для новых элементов с помощью CSS.

Мы узнали о достаточно мягких правилах синтаксиса, позволяющих свободно использовать строчные и прописные буквы, кавычки и минимизацию атрибутов, а также опускать подразумеваемые элементы вроде `head`/`body` и стандартные определения наподобие `type="text/javascript"` и `"text/css"` в тегах `<script>` и `<style>`, а также узнали, как укрощать чудовищные старые версии IE. Неплохо для одной главы, не правда ли?

Глава 2

Текст

Брюс Лоусон



Теперь, когда вы написали разметку для основных частей страницы с помощью HTML5 и увидели, как создается структура документа, пришло время заглянуть глубже и познакомиться с различными способами организации контента.

Для этого вы создадите HTML5-разметку для стандартного блога. Мы выбрали именно такой тип сайта потому, что более чем у 70% профессиональных веб-разработчиков есть свой блог (<http://www.aneventapart.com/alaSurvey2008/>) и каждому хотя бы раз в жизни приходилось видеть блог. Кроме того, это превосходный прототип для современных сайтов с заголовками, футерами, боковыми панелями, множественными панелями навигации и формой — элементами, использующимися как в блогах, так и в новостных сайтах и сайтах-брошюрах (где вместо новостей указывается информация о продуктах). Затем мы перейдем к изучению конкретного случая — рассмотрим реальный веб-сайт и решим, где можно было бы применить новые структуры, а также взглянем на новые элементы и глобальные атрибуты.

Создание структуры основных областей контента

Посмотрите на основную область контента одного из блогов (рис. 2.1). Она может состоять из нескольких статей, каждая из которых содержит метаданные и непосредственно текст статьи.

Hello Mr Last Week!
January 24th, 2010

Dear iMrLastWeek, I do hope you are well.

All my love,

Bruce

Posted in Chums | Edit | No Comments »

Cheese. I love it. Do you?
January 24th, 2010

When in the middle east, I recommend buying yours from Cheeses of Nazareth. A marvellous emporium.

Posted in Cheese | Edit | No Comments »

Hello Remy!
January 24th, 2010

The better looking author of this book sends greetings!

Posted in Chums | Edit | No Comments »

Hello Mum!
January 24th, 2010

Posted in Chums | Edit | No Comments »

Рис. 2.1. Несколько статей в типичном блоге

Вот пример стандартной разметки (упрощенной по сравнению с темой WordPrint, использующейся по умолчанию):

```
<div class="post">
<h2>Memoirs of a Parisian lion-tamer</h2>
<small>January 24th, 2010</small>
<div class="entry">
    <p>Claude Bottom's poignant autobiography is this
       - summer's must-read.</p>
</div>
<p class="postmetadata">Posted in <a href="/?cat=3">
   - Books category</a> | <a href="/?p=34#respond">
   - No Comments</a></p>
</div>
```

Такая разметка в общем-то не является неправильной (хотя сомнения вызывает использование в HTML 5 элемента представления `<small>` для оформления даты). Она будет нормально работать во всех браузерах, но кроме заголовка сообщения блога в документе нет ни одной реальной структуры — только семантически нейтральные элементы `<div>` и параграфы.

HTML 5 предлагает нам использовать для разметки контента обобщенные структуры. К примеру, `<div>` — это обобщенный «блок», который сообщает браузеру что-то вроде «вот тут есть что-то, и оно рассматривается как нечто целое», однако этот элемент не имеет значения; никакой семантики, выходящей за рамки выражения «оно рассматривается как нечто целое», здесь нет. Там, где это возможно, мы заменим обобщенные блоки новыми HTML5-элементами; если в каком-то случае мы не найдем подходящего нового элемента, будем использовать `<div>` так, как мы делали это в HTML 5.

Давайте сначала сосредоточимся на отдельной статье. Как уже говорилось в главе 1, внешний элемент `<div class="post">` можно заменить на `<article>`, однако это далеко не все. HTML5-элементы `<header>` и `<footer>` могут использоваться на странице несколько раз; в каждом таком случае они будут относиться к тому разделу, в котором находятся.

Заголовок и время добавления статьи относятся к «предварительным данным» и, следовательно, к сфере применения элемента `<header>`, не так ли? Аналогичным образом для метаданных, которые в настоящий момент заключены в параграф со значением `class="postmetadata"`, лучше всего использовать `<footer>`, который, как утверждает спецификация, «обычно содержит сведения о соответствующем разделе — информацию об авторе, ссылки на похожие документы, сведения об авторских правах и т. п.».

ПРИМЕЧАНИЕ

Если не указано иное, все цитаты с описаниями элементов взяты из спецификации HTML5 в той редакции, которая существовала на момент написания соответствующей главы.

Схематически новая структура статьи блога показана на рис. 2.2.

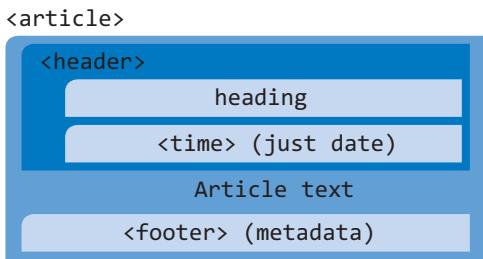


Рис. 2.2. Отдельная статья блога с использованием новых структур HTML5

```

<article>
  <header>
    <h2>Memoirs of a Parisian lion-tamer</h2>
    <time datetime=2010-01-24>January 24th,
      - 2010</time>
  </header>
  <p>Claude Bottom's poignant autobiography is this
    - summer's must-read.</p>
  <footer>
    Posted in <a href=>/?cat=3</a> >Books category</a>.
    - <a href=>/?p=34#respond</a>>No Comments</a>
  </footer>
</article>

```

Давайте посмотрим на этот код повнимательнее.

Элемент `<time>`

`<time>` — это новый элемент, позволяющий однозначно кодировать даты и время (для понимания машинами) и отображать их на экране в привычном для нас виде. Понять, для чего это нужно, довольно просто: браузер сможет добавлять предстоящие события в календарь пользователя; агрегаторы контента смогут создавать визуальные временные шкалы с указанием событий; браузеры, локализованные в Таиланде, смогут отображать даты по буддийскому календарю, то есть значение года будет на 543 больше, чем для западных стран. В спецификации говорится, что «элемент `<time>` обозначает либо время по 24-часовой шкале, либо точную дату по пролептическому григорианскому календарю с возможным указанием времени и смещения часового пояса».

Считываемая часть элемента `<time>` обычно заключена в его атрибут `datetime`, а в качестве контента элемента указывается то, что будет видеть конечный пользователь.

```

<time datetime=2009-11-13>13 November 2009</time>
<time datetime=2009-11-13>13th November last
  - year</time>
<time datetime=2010-11-13>Bruce's 21st birthday</time>
<time datetime=2010-11-13T020:00Z>8PM on my birthday</time>
<time datetime=20:00>8 PM</time>

```

Если считываемый формат совпадает с тем, что должен видеть конечный пользователь, вам не нужно добавлять атрибут `datetime`. В таком случае браузеры найдут нужный контент и распознают его.

СЧИТЫВАЕМЫЕ ДАТЫ И ВРЕМЯ

Чтобы дата могла считываться автоматически, она должна быть в формате YYYY-MM-DD; время, которое также может быть указано, задается в формате НН:ММ с добавлением разделяющего префикса «T». Если вы хотите добавить секунды, отделяйте их от минут с помощью двоеточия. Доли секунды добавляются после точки.

Как вы видели выше, можно указывать только время по 24-часовой шкале (без даты).

Если вы указываете и время, и дату, необходимо указать часовой пояс: используйте «Z» для Всемирного координированного времени (UTC) или задайте смещение относительно UTC в часах и минутах с префиксом «+» или «-».

Итак, вот что мы получаем: «1979-10-14T12:00:00.001-04:00» означает одну миллисекунду после полудня 14 октября 1979 года по летнему восточному времени (UTC–4 часа).

```
<time>20:00</time>
```

На Reddit.com в текстовом содержимом элемента относительное время ("8 часов назад") объединяется с атрибутом `title`, и точное время в удобном для восприятия формате отображается при наведении указателя мыши:

```
<p>Submitted <time title="Sun Jul 3 02:15:49 2011 GMT"
- datetime="2011-07-03T02:15:49.881631+00:00">8 hours</time>
- ago</p>
```

Единственная проблема, связанная с элементом `<time>`, состоит в том, что дата по пролетическому григорианскому календарю должна быть положительной — иными словами, нельзя указать дату до Рождества Христова. Также нельзя использовать неполные даты, такие как «июль 1904». Это существенно ограничивает использование этого элемента для сайтов музеев, исторических сайтов и энциклопедий, генеалогических деревьев и т. п., где точные даты могут быть попросту неизвестны.

Консорциум разработчиков поисковых систем, включая Bing, Google и Yahoo!, запустил инициативный проект под названием schema.org, цель которого — создать и обеспечить поддержку набора распространенных шаблонов разметки. Этот непрерывно растущий набор схем использует микроданные HTML5 (HTML5 Microdata; см. далее в этой главе):

...способами, признанными крупнейшими разработчиками поисковых машин. Механизмы поиска, в том числе Bing, Google и Yahoo!, используют возможности этой разметки для улучшения внешнего вида списков результатов поиска, благодаря чему пользователям становится проще находить нужные веб-страницы (<http://www.schema.org>).

Обратите внимание, что schema.org применяет элемент `<time>` для отображения дат, однако основывается на полном формате даты ISO 8601, а не на сокращенном формате HTML5. Поэтому для них «2011-09» — это также допустимая дата, описывающая любой день в сентябре 2011 года. Схожим образом, с помощью префикса P (что означает period — «период») можно определять промежутки времени:

```
<time itemprop="cookTime" datetime="PT1H30M">1.5 hours</time>
```

ПРИМЕЧАНИЕ

Сейчас, когда второе издание только отправляется в печать, Рабочая группа обсуждает вариант удаления элемента `<time>` из HTML, а на замену ему предлагается создать более общий — и, следовательно, менее полезный — элемент `<data>`. Надеемся, что этого не произойдет, но прежде чем использовать старый элемент, загляните на <http://introducinghtml5.com> и проверьте, не появилось ли там новых описаний ошибок. Или откройте спецификацию!

Мы убедились, что ни один из этих примеров, принадлежащих schema.org, не содержит атрибутов даты и времени, которые были бы способны пройти валидацию HTML5. Это ставит ответственных разработчиков в затруднительное положение: чем следует руководствоваться в первую очередь — личными убеждениями в значимости правильного кода или ощутимыми коммерческими преимуществами, которые дает хорошее понимание вашего контента поисковыми механизмами?

Глупо заставлять разработчиков делать такой выбор. Остается только надеяться, что рабочая группа образумится и ослабит ограничения, накладываемые на элемент `<time>`.

Атрибут `pubdate`

С помощью булева атрибута `pubdate` можно указать, что данный элемент `<time>` обозначает дату публикации элемента `<article>` или всего контента `<body>`.

У вас может возникнуть логичный вопрос: зачем этот атрибут вообще нужен? Почему нельзя считать, что любой `<time>` внутри тега `<header>` элемента `<article>` является датой его публикации?

Рассмотрим следующий пример.

```
<article>
  <header>
    <h1>Come to my party on <time datetime=2010-12-01>1
      ~ December</time></h1>
    <p>Published on <time datetime=2010-06-20 pubdate>20
      ~ June 2010</time></p>
  </header>
  <p>I'm throwing a party at Dr Einstein's Cabaret
    ~ Roller-disco Bierkeller Pizza-parlour-a-gogo. Do come
    ~ and dance to Rusty Trombone's Swingin' Brass Band.
    ~ (Formal dress and lewd hat required.)</p>
</article>
```

Здесь в элементе `<header>` вы видите две даты: дату вечеринки и дату публикации статьи. Атрибут `pubdate` снимает эту неоднозначность. Кстати, приходите, только не пейте слишком много.

КОЕ-ЧТО ИНТЕРЕСНОЕ ОБ ЭЛЕМЕНТАХ `<HEADER>` И `<FOOTER>`

Вот главный сюрприз, с которым мы сталкиваемся при создании новой разметки для нашей статьи: дело в том, что каждая статья может иметь свои собственные элементы `<header>`

и `<footer>`. Это значит, что помимо всей страницы собственные заголовки и футеры могут быть также и у статей. И для них можно создать отдельные таблицы стилей: `body>header` и `body>footer` отсылают к основным заголовкам и футерам (предполагается, что они являются прямыми потомками `<body>`), а `article>header` и `article>footer` отсылают к внутренним структурам.

Чтобы стили правильно отображались в старых версиях IE, можно воспользоваться одной особенностью этого браузера: задать стили для основного заголовка и футера, а затем определить их для заголовка и футера статьи.

```
header {display:block; color:red; text-align:right;}
- /*page header */
article header {color:blue; text-align:center;}
- /*article header */
```

Обратите внимание на то, что до сих пор вы еще не создавали якорей для привязки стилей CSS с помощью атрибутов `class` и `id`.

Использование нескольких элементов `<footer>` внутри одного элемента

В спецификации говорится: «футеры не обязательно должны располагаться в конце раздела, однако чаще всего это так»; согласно спецификации, элемент может иметь два или более футеров. Упрощенная версия примера, приведенного в ней, выглядит так:

```
<body>
  <footer><a href="/">Back to index...</a></footer>
  <h1>Lorem ipsum</h1>
  <p>Lorem ipsum</p>
  <footer><a href="/">Back to index...</a></footer>
</body>
```

Причина этого в следующем: предполагается, что эти элементы не являются элементами представления. Если футер «Back to index» расположен внизу статьи, а вы хотите, чтобы он находился также и вверху, для этих одинаковых фрагментов контента следует использовать одинаковые элементы (независимо от того, где они расположены).

Элементы `<footer>` внутри `<blockquote>`

Интересно, что элемент `<blockquote>` также может иметь футер; это очень удобный способ указать источник так, чтобы он был не только однозначно привязан к цитате, но и правильно отображен на экране (в HTML 5 у элемента `<blockquote>` может быть атрибут `cite`, который, теоретически, предназначен как раз для решения этой задачи — он позволяет авторам добавлять ссылки на источники; однако браузеры не понимают его, и пользователь попросту не видит отсылки к источнику).

```
<blockquote>
  Thou look'st like antichrist, in that lewd hat.
  <footer>Ananias <cite>Scene 4.3, <a href="http://
    - www.gutenberg.org/files/4081/4081-h/4081-h.htm">The
      - Alchemist</a></cite> (Ben Jonson)</footer>
</blockquote>
```

С момента выхода первого издания этой книги мы постоянно встречали рекомендацию избегать использования `<footer>` внутри `<blockquote>` для указания источника цитаты, так как, согласно спецификации, «содержимое `blockquote` должно представлять собой цитату из другого источника», а ссылка на автора — это уже не цитата. Однако это нелогично; зачастую мы «причесываем» цитаты, добавляя многоточия, незаметно исправляя орфографию и т. п. — строго говоря, это тоже не цитирование. Во многих веб-публикациях ссылка на автора помещена внутрь цитаты, что на данный момент спецификацией запрещается (см. исследование моего друга и доктора HTML Оли Стадхолма на <http://oli.jp/2011/blockquote>). Таким образом, мы позволили себе считать, что подобный запрет вкрадся в спецификацию ошибочно.

ДОБАВЛЕНИЕ СООБЩЕНИЙ БЛОГА И КОММЕНТАРИЕВ

Итак, у вас есть страница с заголовком, футером, панелью навигации, основной областью контента с несколькими статьями (сообщениями блога), у каждой из которых есть свой заголовок и футер. Так, подождите... что же это за блог без комментариев?

В спецификации рекомендуется использовать для этого вложенные элементы `<article>`: «если вы имеете дело с вложенными элементами `<article>`, внутренние элементы `<article>` представляют собой статьи, в общем-то, связанные с содержимым внешнего элемента `<article>`. К примеру, в записи блога на сайте, поддерживающем добавление комментариев пользователями, для комментариев могут использоваться элементы `article`, вложенные в `<article>` записи блога».

Давайте так и сделаем. Обратите внимание на то, что комментарии блога обычно располагаются в хронологическом порядке и содержат дополнительные сведения (например, имя автора и URL), которые, по сути, являются заголовочной информацией. Схематически это выглядит примерно так, как показано на рис. 2.3.

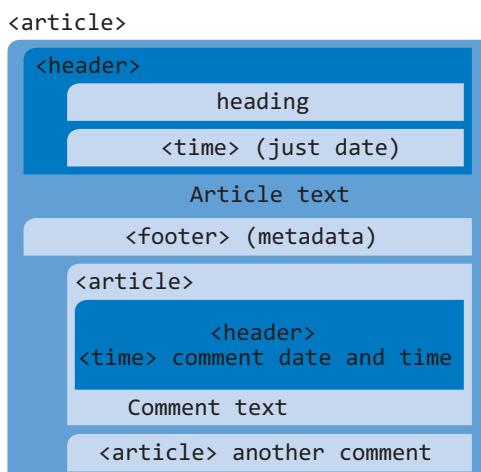


Рис. 2.3. Структура сообщения блога, где комментарии добавлены с помощью вложенных элементов `<article>`

Код выглядит так, как можно было ожидать; комментарии в нем выделены.

```

<article>
  <header>
    <h1>Come to my party on <time datetime=
      - 2010-12-01>1 December</time></h1>
    <p>Published on <time datetime=2010-06-20 pubdate>
      - 20 June 2010</time></p>
  </header>
  <p>I'm throwing a party at Dr Einstein's Cabaret
    - Roller-disco Bierkeller Pizza-parlour-a-gogo. Do come
    - and dance to Rusty Trombone's Swingin' Brass Band.
    - (Formal dress and lewd hat required.)</p>
  <footer>Published in the Parrtay!! category by Bruce
  - </footer>

  <article> <!-- comment -->
    <header>
      Comment from <a href="http://remysharp.com">Remy
      - Sharp</a> at <time datetime="2010-05-01T08:45Z">
      - 8.45 on 1 May 2010</time>
    </header>
    <p>I'll be there. I very much enjoy a bit of Rusty
      - Trombone.</p>
  </article> <!-- end comment -->

  <article> <!-- comment -->
    <header>
      Comment from <a href="http://splintered.co.uk">Patrick
      - Lauke</a> at <time datetime="2010-05-02T10:45Z">10.45
      - on 2 May 2010</time>
    </header>
    <p>Sorry mate. Am off to Bath to see TubaGirl.</p>
  </article> <!-- end comment -->

</article> <!-- end blogpost -->
```

РАБОТА СО СТРУКТУРАМИ HTML5

Некоторые приложения для обработки текста позволяют просматривать структуру документа. На рис. 2.4 показана эта глава в режиме структуры Microsoft Word 2007.

В HTML5 есть строгий алгоритм определения структуры документа, который позволяет агентам пользователей строить похожие структуры для веб-страниц. Как и в случае с программами обработки текста, это может понадобиться для того, чтобы пользователь мог быстро получить общее представление о странице (и, кроме того, нет причин считать, что веб-страницы не являются по своей сути приложениями для обработки текста). Еще одна область применения этого алгоритма — синдикация: вставляя часть содержимого

в другое место, вы можете быть уверены, что на новой странице контент сохранит свою логическую структуру.

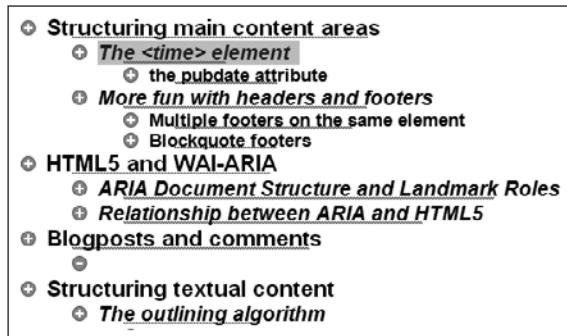


Рис. 2.4. Режим структуры Microsoft Word 2007

Должен сразу предупредить: в этой части главы вы найдете достаточно много информации, а новые возможности, описанные здесь, еще не реализованы ни в одном браузере (правда, существует JavaScript-реализация: <http://code.google.com/p/h5o/> — по этому адресу вы найдете расширение для Chrome, а расширение для Opera можно загрузить с <https://addons.opera.com/addons/extensions/details/html5-outliner>). Для правильного использования элементов `<section>`, `<aside>` и `<div>` вы должны хорошо представлять себе, что такая структура документа.

Перед тем как мы двинемся дальше, важно познакомиться с еще одним понятием, отличающим HTML 5 от HTML 4: некоторые HTML5-элементы — `<article>`, `<section>`, `<nav>` и `<aside>` — являются *структурированным контентом* и служат началом новых разделов в структуре. Чтобы было понятнее, рассмотрим простой код:

```

<h1>Hello</h1>
<div>
  <h1>World</h1>
</div>
  
```

Чтобы продемонстрировать работу алгоритма, я использую сетевую утилиту <http://gsnedders.html5.org/outliner/>, поскольку этот алгоритм не встроен ни в один современный браузер. Этот код генерирует ту структуру, которую и следовало ожидать (рис. 2.5).

На рис. 2.6 показано, как изменилась структура после замены семантически нейтрального элемента `<div>` на `<article>`, который является структурированным контентом.

```

<h1>Hello</h1>
<article>
  <h1>World</h1>
</article>
  
```

1. Hello
2. World

Рис. 2.5. Простая структура

1. Hello
1. World

Рис. 2.6. Структура документа после замены `<div>` на `<article>`

Как вы видите, появление структурированного контента изменило иерархию так, что элемент `<article>` оказался «ниже» остального контента. Проще говоря, `<h1>`, расположенный внутри статьи, логически является `<h2>`, поскольку `<article>` начинает новый раздел. Использование `<section>`, `<nav>` и `<aside>` вместо `<article>` будет иметь такой же эффект, так как все эти элементы являются структурированным контентом.

На самом деле, неважно, какой уровень заголовка вы здесь используете; алгоритм определения структуры учитывает вложения и относительные уровни, так что следующий код будет иметь точно такую же структуру, как и предыдущий (см. рис. 2.6).

```
<h3>Hello</h3>
<article>
  <h6>World</h6>
</article>
```

Вы, возможно, скажете: «Ну и что? Зачем все это нужно?» Что ж, во-первых, это означает, что вы не ограничены шестью уровнями заголовков, как это было в HTML 4. Заголовочный элемент внутри семи уровней элементов `<article>`, `<section>`, `<nav>` или `<aside>` (или любой их комбинации) логически является элементом `<h7>` (правда, использование такого количества уровней может быть оправдано, только если вы создаете разметку правовых документов или других ужасных вещей; иначе вам следует пересмотреть содержимое вашего документа).

Еще одно преимущество структурированного контента — синдициация. Предположим, что Реми добавил такую статью:

```
<article>
  <h1>What I did on my holiday</h1>
  <p>I went to Narnia. I was bitten by
    - a trilobite. Then I
    - came home.</p>
</article>
```

ОТ TBL '91 ДО XHTML2 И HTML5

Поражает новизной, не так ли? Вообще-то, нет. Идея о том, что логика заголовков должна зависеть от уровней вложенности разделов `<section>`, впервые появилась еще в 1991 году и была изложена самим сэром Тимом Бернерс-Ли:

«Я бы предпочел, чтобы вместо определения заголовков с помощью `<H1>`, `<H2>` и т. д. мы использовали вложенный элемент `<SECTION>..</SECTION>`, а обобщенный тег `<H>..</H>` на любом уровне внутри разделов создавал бы заголовок соответствующей величины».

По каким-то причинам в HTML эта идея не прижилась. Однако ее возродили в XHTML2 и еще раз в HTML5, но без элемента `<h>` — для обеспечения обратной совместимости вместо него применяются заголовки `<h1>`–`<h6>`.

Предположим, вы работаете в крупной интернет-газете и хотите добавить эту историю. Когда волшебная программа для синдициации добавит ее в ваш шаблон, получится такой код.

```
<h1>The Monotonous Times</h1>
<section>
  <h2>Breaking news</h2>
```

```

<article>
    <h1>What I did on my holiday</h1>
    <p>I went to Narnia. I was bitten by
       - a trilobite. Then I came home.</p>
</article>
..
</section>
```

Очевидно, что в иерархии заголовок «Breaking news» находится выше, чем заголовок сообщения блога Реми, но из-за несогласованности шаблона Реми и вашего шаблона элемент `<h2>` оказывается важнее элемента `<h1>`.

Однако, если мы посмотрим на схему, мы увидим, что все находится на своих местах (рис. 2.7). Отлично!

1. The Monotonous Times
 1. Breaking news
 1. What I did on my holiday

Рис. 2.7. Алгоритм строит правильную структуру и при синдикиации

<NAV>, <ASIDE> И РАЗДЕЛЫ БЕЗ ЗАГОЛОВКА

А сейчас я немного расскажу об инструментах проверки структуры, поскольку, скорее всего, просмотр структуры документа в процессе его разработки станет для вас привычкой, так же как валидация кода и проверка его работы в разных браузерах.

Обычно, если приложение обнаруживает структурированный контент без заголовка, оно сообщает об этом.

```

<article>
    <p>I have no heading</p>
</article>
```

В таком случае вы получите структуру «Untitled Section». Это особенно полезно для элементов `<section>` и `<article>`, которые, как правило, должны начинаться с заголовка.

Внутри элементов `<nav>` и `<aside>`, наоборот, может не быть заголовка. Для блоков `<nav>`, таких как «Популярные сообщения» или «Недавние комментарии», использование заголовков может быть и оправданным, но вы вряд ли захотите добавлять к панели глобальной навигации явно лишний заголовок «Панель навигации».

Таким образом, если вы видите «Untitled Section», это не значит, что вы обязательно должны добавить недостающий заголовок. Считайте это предупреждением, а не ошибкой.

<HGROUP>

Иногда помимо заголовка используется подзаголовок или слоган: на сайте Slashdot — «News for nerds. News that matters», на dev.Opera — «Follow the standards. Break the rules», на Metafilter — «community weblog». Как следует писать разметку для таких слоганов? В HTML 5 можно было сделать это так:

```

<h1>Metafilter</h1>
<p>community weblog</p>
```

Но это не очень правильный вариант, поскольку слоган — это в каком-то смысле тоже заголовок. Еще один способ выглядел бы так:

```
<h1>Metafilter</h1>
<h2>community weblog</h2>
```

Но в таком случае для всех остальных заголовков сайта придется использовать теги `<h3>–<h6>`, так как в иерархии они будут находиться ниже слогана. В HTML5 подзаголовок можно разметить как заголовок, но так, чтобы он не отображался в структуре документа.

```
<hgroup>
  <h1>Metafilter</h1>
  <h2>community weblog</h2>
</hgroup>
```

Структура этого кода показана на рис. 2.8.

1. Metafilter

Рис. 2.8. Мы видим, что частью структуры является только заголовок Metafilter

По поводу `<hgroup>` в спецификации говорится следующее: «для удобства получения информации о структуре документа и других похожих сведений текст элемента `<hgroup>` определяется как текст заголовка самого высокого уровня (`h1–h6`) из числа потомков этого элемента `hgroup`».

Таким образом, в структуре документа со следующим кодом будет отображаться только «Remy Sharp», поскольку это заголовок самого высокого уровня внутри группы.

```
<article>
  <hgroup>
    <h2>Get the beers in! Here comes</h2>
    <h1>Remy Sharp!</h1>
  </hgroup>
</article>
```

ПРИМЕЧАНИЕ

В этом примере внутри `<carticle>` мы не использовали элемент `<header>`. `<header>` — это элемент, который группирует предварительные данные; поскольку здесь у нас есть только заголовки, уже объединенные в `<hgroup>`, еще одно объединение было бы хотя и не ошибочным, но явно излишним.

СТРУКТУРИРОВАННЫЕ КОРНЕВЫЕ ЭЛЕМЕНТЫ

Некоторые элементы — `<blockquote>`, `<body>`, `<details>`, `<fieldset>`, `<figure>` и `<td>` — называются *структурированными корневыми элементами*. Они могут иметь свою собственную структуру, однако их внутренние разделы и заголовки не учитываются при построении структуры своих предков. Почему так происходит? Если, например, в теге `<blockquote>` вы цитируете несколько разделов какой-либо статьи, эти разделы не имеют отношения к общей структуре документа.

В следующем примере содержимое `<blockqoute>` не входит в структуру документа (рис. 2.9).

```

<h1>Unicorns and butterflies</h1>
<nav>
    <h2>Main nav</h2>
    ...
</nav>
<article>
    <h2>Fairies love rainbows!</h2>
    <p>According to Mr Snuggles the fluffy kitten, fairies
        - like:</p>
    <blockquote>
        <h3>Pretty dainty things</h3>
        <p>Fairies love rainbows, ribbons,
            - and ballet shoes</p>
        <h3>Weaponry</h3>
        <p>Fairies favour Kalashnikovs, flick
            - knives, and
            - depleted uranium missiles</p>
    </blockquote>
</article>

```

1. Unicorns and butterflies
 1. Main nav
 2. Fairies love rainbows!

Рис. 2.9. Структура не включает содержимое структурированного корневого элемента

СОЗДАНИЕ СТИЛЕЙ ДЛЯ ЗАГОЛОВКОВ В HTML5

Все эти премудрости представляют серьезную проблему для авторов CSS. Рассмотрим следующий код.

```

<article><section><h1>...</h1></section></article>
<article><article><h1>...</h1></article></article>
<section><section><h1>...</h1></section></section>
<section><aside><h1>...</h1></aside></section>
<h3>...</h3>

```

Поскольку эти элементы потенциально могут относиться к одному логическому уровню, вполне возможно, что вам захочется применить к ним одинаковое оформление (а возможно и нет). Это может привести к добавлению огромных блоков правил в таблицы стилей. Уже некоторое время ведутся разговоры о новом псевдоклассе или псевдоэлементе CSS `:heading(n)` — его добавление было бы возможным (потому что браузер «знает» об уровне заголовка благодаря алгоритму определения структуры) и существенно облегчило бы написание стилей.

```

*:heading(1) {font-size: 2.5em;} /* логический <h1> */
*:heading(2) {font-size: 2em;} /* логический <h2> */

```

Однако на момент написания этой книги это всего лишь чудесная мечта. В качестве временного решения проблемы Mozilla пытается внедрить в ночные сборки Firefox новый механизм объединения селекторов под названием :moz-any(), который позволяет использовать новую форму краткой записи — см. <http://hacks.mozilla.org/2010/05/moz-any-selector-grouping/>.

По этой причине вы, возможно, будете использовать только элементы <h1>, поскольку так будет проще создать оформление, а всю остальную работу возьмет на себя алгоритм определения структуры. Ведь в спецификации говорится: «разделы могут содержать заголовки любого уровня, однако разработчикам настоятельно рекомендуется использовать либо только элементы <h1>, либо заголовки того уровня, который соответствует уровню вложенности раздела». Но делать этого (пока) не стоит, поскольку при этом снижается доступность.

АЛГОРИТМ ОПРЕДЕЛЕНИЯ СТРУКТУРЫ И ДОСТУПНОСТЬ

Недавнее исследование WebAim показало, что 57% пользователей программ экранного доступа впервые знакомятся с содержимым объемного сайта, ориентируясь по его структуре заголовков (с полными результатами исследования можно ознакомиться на <http://www.webaim.org/projects/screenreadersurvey3/>). Эти люди используют иерархию заголовков как для того, чтобы составить мысленное представление о документе (то есть определить его структуру), так и для того, чтобы перемещаться по его содержимому. В большинстве программ экранного доступа для перемещения по заголовкам предусмотрены специальные сочетания клавиш. К примеру, JAWS (кстати, согласно исследованию, этот пакет применяется наиболее часто) использует клавишу Н для перехода от заголовка к заголовку, клавишу 1 для перехода к следующему <h1>, клавишу 2 для перехода к следующему <h2> и т. д.

ПРИМЕЧАНИЕ

Посмотрите замечательное видео под названием «Значение HTML-заголовков для доступности» (*Importance of HTML Headings for Accessibility*) на http://www.youtube.com/watch?v=AmUPhEVWu_E. В нем показано, как слепой эксперт по доступности веб-контента перемещается по сайту с помощью JAWS. В идеальном мире любой профессиональный дизайнер или разработчик обязан был бы посмотреть и понять это видео. Но, к сожалению, это не идеальный мир.

В настоящее время при построении внутренней модели страницы ни один браузер не учитывает всех сложных правил, упомянутых выше, и, следовательно, не может передать эту модель программе экранного доступа или вспомогательной технологии. Поэтому использование только элементов <h1> делает невозможной навигацию по сайту и тем самым снижает доступность вашей страницы.

Поэтому мы советуем следовать указаниям, приведенным в спецификации: «используйте заголовки того уровня, который соответствует уровню вложенности раздела». Иными словами, следите за тем, чтобы на ваших страницах иерархия заголовков оставалась правильной даже без учета новых HTML5-элементов. Это также упростит написание таблиц стилей.

По поводу случаев, когда статьи добавляются на сайт с другого сайта и логический порядок уровней может быть нарушен, я скажу вот что: абсолютно ненаучный опрос пользователей программ экранного доступа, проведенный сайтом Twitter, показал, что плохо вложенные заголовки разделов лучше, чем использование заголовков одного уровня (что, в свою очередь, лучше, чем полное их отсутствие).

Сделайте все, что в ваших силах, и ждите, пока браузеры и зависящие от них программы экранного доступа внедрят алгоритм определения структуры. Как я уже сказал, это не идеальный мир.

В ЧЕМ РАЗЛИЧИЕ МЕЖДУ <ARTICLE> И <SECTION>?

Этот вопрос нам регулярно задают на htm5doctor.com.

Статья (`article`) — это независимый, обособленный фрагмент дискретного контента: например, сообщение блога или новостное сообщение. В веб-приложении элемент `<article>` может использоваться для отдельных электронных сообщений в почтовой программе или репортажей в клиенте для чтения RSS-лент, поскольку каждое электронное письмо или репортаж, являясь компонентом приложения, может быть использовано независимо от него. Услышав слово «статья», не нужно думать о газетной или журнальной статье — вспомните определение «статья расходов», в котором это слово обозначает отдельный элемент в некотором наборе.

`<article>`

Рассмотрим статью блога/новостей из реального мира.

```
<article>
<h1>Bruce Lawson Is World's Sexiest Man</h1>
<p>Legions of lovely ladies voted luscious lothario Lawson
- as the World's Sexiest Man today.</p>
<h2>Second-sexiest man concedes defeat</h2>
<p>Remington Sharp, JavaScript glamourpuss and Brighton
- roister-doister, was gracious in defeat. "It's cool
- being the second sexiest man when number one is Awesome
- Lawson" he said from his swimming pool-sized jacuzzi full
- of supermodels.</p>
</article>
```

Она может быть добавлена на какой-либо сайт (с помощью RSS или других средств), и она понятна без дополнительных уточнений. Точно так же, как вы можете добавлять не целое сообщение, а только его часть, интригующая цитата из статьи остается статьей.

```
<article>
<a href=full-story.html>
    <h1>Bruce Lawson is World's Sexiest Man</h1>
    <p><img src=bruce.png alt="bruce lawson">Legions of lovely
    - ladies voted luscious lothario Lawson as the World's
    - Sexiest Man today.</p>
    <p>Read more</p>
</a>
</article>
```

Как мы уже знаем, комментарии к сообщениям блога — это элементы `<article>` внутри родительских элементов `<article>`. Такое вложение может использоваться и в других ситуациях, например при добавлении расшифровки к видеозаписи.

```
<article>
<h1>Stars celebrate Bruce Lawson</h1>
<video>...</video>

<article class=transcript>
<h1>Transcript</h1>
  <p>Supermodel #1: "He's so hunky!"</p>
  <p>Supermodel #2: "He's a snogtabulous bundle of gorgeous
    - manhood! And I saw him first, so hands off!"</p>
</article>

</article>
```

«БЛОЧНЫЕ» ССЫЛКИ

Внимательно посмотрев на этот пример, вы заметите, что «блочные» элементы можно заключать в ссылки. В спецификации HTML 5 это не разрешено, поэтому вам пришлось бы заключать заголовок, абзац с интригующей цитатой и фразу «read more» в отдельные элементы ссылки, указывающие на один и тот же адрес.

Однако выясняется, что все браузеры очень слаженно и с большим энтузиазмом поддерживают обрамление ссылками разнообразных блочных элементов (нет, если честно, им все же требуется пара подсказок — вспомните <http://mattwilcox.net/sandbox/html5-block-anchor/test.html>). Только старая спецификация HTML шла против всех и не допускала этого, а в HTML5 одна ссылка может окружать целый блок `<article>`. Так как браузеры уже умеют работать со ссылками вокруг блочных элементов, и мы имеем дело с примером, в котором их как раз и нужно использовать, у нас нет причин сохранять ошибки в структуре.

Давно стало хорошей традицией (для обеспечения хорошей доступности содержимого и удобства его использования) заранее загружать важную информацию в самом начале ссылок. Теперь, когда ссылки могут охватывать еще более крупные фрагменты содержимого, это приобрело дополнительную значимость (спасибо Стиву Фолкнеру за статью <http://www.pacielllogroup.com/blog/2011/06/html5-accessibility-chops-block-links>).

Сама расшифровка представляет собой законченный фрагмент контента, несмотря на то что он относится к видеозаписи, расположенной во внешнем элементе `<article>`. В спецификации говорится следующее: «если вы имеете дело сложенными элементами `article`, внутренние элементы `article` представляют собой статьи, в общем-то, связанные с содержимым внешнего элемента `article`».

`<section>`

Элемент `<section>` — «раздел», — напротив, не является «компонентом страницы, документа, приложения или сайта, предназначенным к многократному использованию и свободному распространению». Это либо способ разделения страницы на несколько тематических областей, либо способ разделения статьи на (иначе не скажешь) разделы.

ПРИМЕЧАНИЕ

Элемент `<section>` обычно начинается с заголовка, содержащего вводную информацию. Исключением может быть раздел, в котором заголовок добавляется с помощью JavaScript.

Если вы не собираетесь добавлять заголовок или же вам просто нужен элемент-контейнер для создания оформления, лучше использовать `<div>`.

Давайте рассмотрим инструкции с предыдущей работы Реми во внебродвейской постановке «Волшебника из страны Оз» в разметке HTML 5.

```
<h1>Rules for Munchkins</h1>
<h2>Yellow Brick Road</h2>
    <p>It is vital that Dorothy follows it—so no selling
    ~ bricks as "souvenirs"</p>
<h2>Fan Club uniforms</h2>
    <p>All Munchkins are obliged to wear their "I'm a friend
    ~ of Dorothy!" t-shirt when representing the club</p>
    <p><strong>Vital caveat about the information above:
    ~ does not apply on the first Thursday of the month.
    ~ </strong></p>
```

Возникает вопрос: параграф «Vital caveat about the information above» относится ко всей статье (то есть к тому, что расположено ниже `<h1>`) или же только к информации, расположенной ниже ближайшего к нему `<h2>` («Fan Club uniforms»)? В HTML 5 этот параграф будет относиться к `<h2>`, и не существует простого способа это изменить. В HTML5 элемент `<section>` снимает эту неоднозначность (именно это мы, веб-разработчики, понимаем под термином «семантический»).

```
<article>
<h1>Rules for Munchkins</h1>

<section>
    <h2>Yellow Brick Road</h2>
    <p>It is vital that Dorothy follows it—so no selling
    ~ bricks as "souvenirs"</p>
</section>

<section>
    <h2>Fan Club uniforms</h2>
    <p>All Munchkins are obliged to wear their "I'm a friend
    ~ of Dorothy!" t-shirt when representing the club</p>
</section>

    <p><strong>Vital caveat about the information above:
    ~ does not apply on the first Thursday of the month.
    ~ </strong></p>
</article>
```

Схематическое изображение этой статьи показано на рис. 2.10.

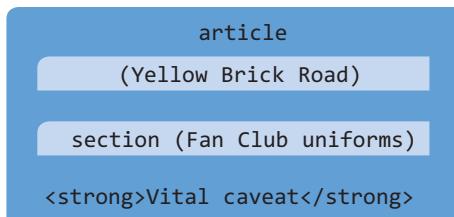


Рис. 2.10. Теперь видно, что «Vital caveat...» относится ко всей статье

Если бы этот параграф находился внутри последнего раздела, он бы однозначно относился только к этому разделу (рис. 2.11).

```

<article>
...
<section>
    <h2>Fan Club uniforms</h2>
    <p>All Munchkins are obliged to wear their "I'm a friend
    - of Dorothy!" t-shirt when representing the club</p>
    <p><strong>Vital caveat about the information above:
    - does not apply on the first Thursday of the month
    - </strong></p>
</section>
</article>
    
```

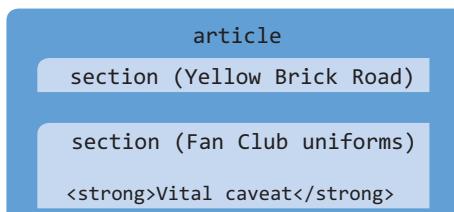


Рис. 2.11. Элемент <section> снимает любую неоднозначность

Было бы некорректно разделять эту статью на логические фрагменты с помощью вложенных элементов <article>, так как эти фрагменты не являются самостоятельными дискретными единицами.

Отлично. Итак, мы увидели, что можно использовать <article> внутри <article> и <section> внутри <article>. Но еще можно использовать <article> внутри <section>. Как же дело обстоит в таком случае?

<article> внутри <section>

Представьте себе, что ваша основная область контента разделена на два блока: в одном расположены статьи о ламах, в другом — о корнеплодах. Вот такой у меня контент.

Вы не обязаны размечать статьи о ламах отдельно от статей о корнеплодах, но вы хотите показать, что эти две группы тематически отличаются друг от друга. Возможно, именно из-за этих отличий вы захотите поместить их в отдельные колонки или с помощью CSS и JavaScript создать интерфейс с вкладками.

В HTML 5 мы бы использовали нашего хорошего, но семантически бедного друга `<div>`. В HTML5 мы используем `<section>`, который, как и `<article>`, запускает алгоритм определения структуры (`<div>` этого не делает, так как не обладает специфическим структурным значением).

```

<section>
<h1>Articles about llamas</h1>

<article>
<h2>The daily llama: Buddhism and South American camelids
- </h2>
<p>blah blah</p>
</article>

<article>
<h2>Shh! Do not alarm a llama</h2>
<p>blah blah</p>
</article>

</section>

<section>
<h1>Articles about root vegetables</h1>

<article>
<h2>Carrots: the orange miracle</h2>
<p>blah blah</p>
</article>

<article>
<h2>Eat more Swedes (the vegetables, not the people)</h2>
<p>blah blah</p>
</article>

</section>

```

Почему мы не используем вместо двух элементов `<section>` элементы `<article>`? Потому что в этом примере каждый элемент `<section>` — это набор независимых единиц, каждая из которых может быть добавлена на другую страницу, — но вы вряд ли станете добавлять на другую страницу весь этот набор.

Обратите внимание на то, что элемент `<section>` не обязан содержать много элементов `<article>`; он может состоять из нескольких параграфов с различными сведениями, такими как лицензирование Creative Commons, биография автора или предупреждение об авторском праве. В нашем примере каждая статья может содержать подстатьи или разделы (об этом уже говорилось ранее) или и то и другое.

Эстель Вейль приводит хорошую аналогию в статье на <http://www.standardista.com/html5-section-v-article>: «Представьте себе газету. В газете есть разделы. Вы видите раздел спорта, раздел недвижимости, раздел «Дом и сад» и т. д. Каждый из этих разделов, в свою очередь, содержит статьи. А некоторые из статей поделены на подразделы.

ПРИМЕЧАНИЕ

На извечный вопрос «какой же структурный элемент мне использовать?!» с легкостью отвечает документ «Удивительная диаграмма элементов HTML5, в которых все постоянно путаются, от докторов HTML5» по адресу <http://html5doctor.com/wp-content/uploads/HTML5Doctor-sectioning-flowchart.pdf>.

Иными словами, можно создавать родительские элементы `<section>` с вложенными элементами `<article>`, в которых есть один ли несколько элементов `<section>`. Не все страницы и документы должны быть устроены именно так, но это абсолютно допустимый и корректный способ вложения элементов».

АНАЛИЗ WWW.GUARDIAN.CO.UK

Давайте продолжим тему газет и посмотрим на настоящий сайт. Попробуем определить, где следует использовать новые структуры. На рис. 2.12 показан скриншот страницы моей любимой газеты The Guardian на <http://www.guardian.co.uk>. Давайте посмотрим, как можно создать такую страницу с помощью HTML5.

Хочу обратить ваше внимание на следующее: далее рассказывается о том, как бы я создавал разметку для этой страницы; вы могли бы выбрать другие структуры, и это нормально. Нет «единственно верного способа» сделать это; все зависит частично от того, как вы будете использовать контент — собираетесь ли вы передавать статьи в другие издания или же добавлять их из базы данных, чтобы каждая статья отображалась на нескольких страницах с разной иерархией заголовков.

Достаточно легко заметить название бренда и другое вводное содержание внутри элемента `<header>`, который также включает две структуры `<nav>`, то есть две панели навигации по сайту (рис. 2.13).

Я решил использовать два отдельных элемента `<nav>`, так как верхняя навигационная лента («News, Sport, Comment.») относится ко всему сайту, а вторая («News, UK, World.») — только к одному разделу. В качестве альтернативного структурного подхода можно было бы создать один элемент `<nav>` в заголовке и навигацию по разделу поместить в список второго уровня внутри навигации по сайту:

```
<nav>
<ul>
<li><a href=...>News</a>
    <ul>
        <li><a href=...>News</a></li>
        <li><a href=...>UK</a></li>
        <li><a href=...>World</a></li>
    ...
    </ul>
</li>
...
</ul>
</nav>
```

Sign in | Register | Mobile version | **A** Text larger | smaller | Go to: guardian.co.uk home | **Go**
 Newspaper website of the year | 31 March 2010 | Last updated less than one minute ago | Webfeed | Search | guardian.co.uk | **Search**
 Weather | London |  7°C |  4°C

guardian.co.uk

[News](#) | [Sport](#) | [Comment](#) | [Culture](#) | [Business](#) | [Money](#) | [Life & style](#) | [Travel](#) | [Environment](#) | [TV](#) | [Blogs](#) | [Video](#) | [Community](#) | [Jobs](#)
[News](#) | [UK](#) | [World](#) | [United States](#) | [Politics](#) | [Media](#) | [Education](#) | [Society](#) | [Science](#) | [Technology](#) | [Football](#) | [Guardian](#) | [Observer](#)
Breaking news: LATEST: Four schoolchildren seriously injured in coach crash in South Lanarkshire, Strathclyde police said. More details soon ...

Brown likely to stay as PM in hung parliament
 MPs will get 18-day window to form government under emergency plan drawn up by Whitehall
 36 comments
 LIVE blog: All the pre-election buildup
 Gordon Brown to pledge new curb on immigration
 PM to pick TV debate position due to poor eyesight

LIVE: latest severe weather updates
 Last updated one minute ago

 Join Adam Gabbatt as blizzards across Northern Ireland and Scotland black out 40,000 homes and disrupt travel
 4 comments
 Snow cuts power and transport
 In pictures: Spring snow in Scotland

Network Rail begins strike legal fight
 Rail operator serves papers on union but insists 'constructive talks' are continuing between the two sides
 BA warns of cancellations as staff return to work

Congress pressed over Israel stance

 Aipac circulates letter calling for end to bitter public confrontation between White House and Netanyahu
 Obama and Sarkozy unite on Iran – and hamburgers
 White House allows Virginia offshore drilling

Ofcom orders Sky Sports price cut
 Last updated one minute ago
 Media regulator rules BSkyB must make 23.4% cut to wholesale price it charges rivals such as Virgin and BT
 30 comments

Isas 'leave UK savers £3bn short'
 Last updated three minutes ago
 Consumer Focus cites 'unfair changes to cash Isa rates and transfer delays in super-complaint to watchdog
 22 comments

Farc release hostage after 12 years

 Colombian guerrillas hand over 32-year-old soldier Pablo Emilio Moncayo, abducted in 1997

Climate research 'secrecy' criticised
 Leaked Climate Research Unit emails show scientists withheld information – but inquiry blames university
 26 comments

Serbians 'sorry' for Srebrenica killing

England holds its breath ... Rooney ankle injury caps bleak night for United at Bayern


guardianjobs
 Search all jobs | **Go**
 Upload your CV
 Get jobs by email
 Join our career forums

Online dating

 Find your Soulmate with the Guardian's dating site
 Join Guardian Soulmates

Jacques Brel & Georges Brassens CD boxsets

 Order both sets (ten original albums) for just £12.95
 Find 1,000s of offers

Today's paper
 The Guardian
 Comment and debate
 Editorials, letters and corrections
 G2 features
 Obituaries
 Sport
 SocietyGuardian

Zeitgeist
 Today's hot topics

On this site
 A-Z
 Audio & podcasts
 Cartoons
 Community
 Corrections
 Crossword
 Datablog

Trending
 Obama and Sarkozy
 Election 2010
 Moscow bombings
 Mephedrone

Best of guardian.co.uk
Is Britain Broken?

 Is the Tory mantra an accurate description of life today, or glib jargon with no grounding in reality? **Amelia Gentleman** investigates
 43 comments

Notes and queries

 Which came first, orange the colour or orange the fruit? And when Blur were better than Oasis...

Tech Weekly live: Personal privacy

 Aleks Krotoski and her guests talk about privacy, surveillance and online censorship in a special live edition of our podcast
 1 comment

How to do business in China

 Business podcast: John Kay, author of *Obligivity*, joins us for a discussion on the Treasury and China

Latest multimedia


Рис. 2.12. Главная страница сайта газеты Guardian

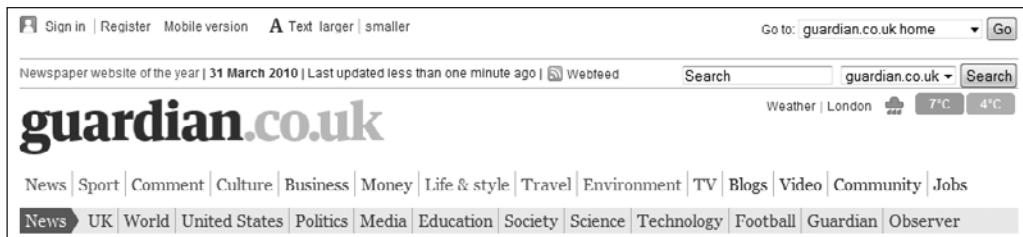


Рис. 2.13. Название бренда на главной странице сайта газеты The Guardian

Сразу после заголовка расположена область под названием «Breaking news» («Свежие новости») и «телеграфная лента» с текстом. Каждая сводка является ссылкой на более подробное описание (рис. 2.14).

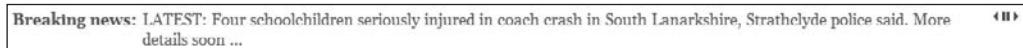


Рис. 2.14. Область «Breaking news» на сайте газеты The Guardian

Если не считать эффекта «телеграфной ленты», добавленного с помощью JavaScript, раздел «Breaking news» представляет собой простой список ссылок на другие страницы. Так что для него подходит элемент `<nav>`. Не важно, что он горизонтальный и заголовок расположен на той же строке; CSS с этим разберется.

```
<nav>
<h2>Breaking news</h2>
<ul>
  <li><a href=...>Four schoolchildren injured...</a></li>
  <li><a href=...>Terrible thing happens to someone</a></li>
  ...
</ul>
</nav>
```

Хотя визуально эта область кажется тесно связанной с заголовком, это все же не вводное содержание и не глобальная панель навигации. Различие здесь очень тонкое, но я считаю, что ссылки на комментарии и страницы, посвященные телевидению и спорту, являются частью глобальной навигации, тогда как перемещение между новостными сообщениями на сайте новостей — это «упрощенный» вариант навигации на более низком уровне. Таким образом, для этой области будет использоваться элемент `<nav>`, не являющийся частью элемента `<header>`, а следующий после него.

Справа от основной области контента расположена еще одна панель навигации (рис. 2.15). Как мы уже знаем из главы 1, такая боковая панель может состоять из навигационной панели и другого содержимого, не относящегося к навигации. Для группировки всего этого вместе мы используем элемент `<aside>`.

```
<aside>
<nav>
  <h2>guardianjobs</h2>
  <form role=search ... > ... </form>
  <ul>
```

```

<li><a href=...>Upload your CV</a></li>
...
</ul>
<h2>Online Dating</h2>
<ul>
...
</ul>
...
</nav>
<section>
<h2>Sponsored Features</h2>
<section>
</aside>
```

Наша навигационная панель представляет собой один элемент `<nav>`, содержащий несколько неупорядоченных списков — каждый с собственным заголовком («Jobs», «Dating», «CD box sets», «Today's paper» и т. д.) — на сероголубом фоне с красной границей наверху.

ПРИМЕЧАНИЕ

Обратите внимание на то, что я добавил в `<nav>` форму поиска (хотя и поместил ее за пределами элементов ``); я считаю, что формы поиска не менее важны, чем собственно навигация. ARIA-атрибуту `role` я присвоил значение, соответствующее функциям формы.

Однако вопреки собственному совету из первого издания этой книги, я не стал обрамлять каждый список отдельным элементом `<nav>`, так как поставил целью помочь пользователям программ чтения экрана быстро обнаруживать навигационную панель. Множество разрозненных `<nav>`, сконцентрированных в одной области страницы, затруднили бы восприятие информации такими пользователями. Однако это всего лишь мои предположения; как только вспомогательные технологии начнут поддерживать `<nav>`, все это нужно будет тщательно протестировать на живых пользователях.

Раздел «Sponsored Features» я не поместил в элемент `<nav>`, поскольку это не панель навигации; предполагается, что его основная цель — реклама. С другой стороны, нам никто не запрещает определить для его заголовка такой же стиль, какой мы решили добавить к заголовкам внутри `<nav>`, если того требует дизайн страницы.

А теперь посмотрим на основную область контента (рис. 2.16).

Неудивительно, что основная область главной страницы сайта газеты The Guardian целиком состоит из новостных статей. Обратите внимание на то, что здесь нет основного заголовка, объединяющего самые важные статьи (такого как «Top stories» («Главные новости»)), иначе все это можно было бы поместить

Рис. 2.15. Боковая навигационная панель справа на главной странице The Guardian



в тер `<section>`. В нашем случае мы имеем дело с простым списком статей (`<article>`). Поскольку `<section>` для этого не подходит, при необходимости (например, для создания оформления) можно объединить статьи с помощью семантически нейтрального элемента `<div>`.

Brown likely to stay as PM in hung parliament

MPs will get 18-day window to form government under emergency plan drawn up by Whitehall

36 comments

- LIVE blog: All the pre-election buildup
- Gordon Brown to pledge new curb on immigration
- PM to pick TV debate position due to poor eyesight

LIVE: latest severe weather updates

Last updated one minute ago

Join Adam Gabbatt as blizzards across Northern Ireland and Scotland black out 40,000 homes and disrupt travel

4 comments

- Snow cuts power and transport
- In pictures: Spring snow in Scotland

Network Rail begins strike legal fight

Rail operator serves papers on union but insists 'constructive talks' are continuing between the two sides

- BA warns of cancellations as staff return to work

Congress pressed over Israel stance

Aipac circulates letter calling for end to bitter public confrontation between White House and Netanyahu

- Obama and Sarkozy unite on Iran – and hamburgers
- White House allows Virginia offshore drilling

Trending

- Obama and Sarkozy
- Election 2010
- Moscow bombings
- Mephedrone

Best of guardian.co.uk

Is Britain Broken?

Is the Tory mantra an accurate description of life today, or glib jargon with no grounding in reality? **Amelia Gentleman** Investigates

43 comments

Campaigns and investigations

Climate wars: Hacked emails investigation

Katine project in Uganda

Notes and queries

Which came first, orange the colour or orange the fruit? And when *Blur* were better than *Oasis* ...

Рис. 2.16. Основная область контента на сайте газеты The Guardian

Еще на главной странице есть интересная статья, состоящая главным образом из фотографии, по всей вероятности, потому что из всех фотографий эта является самой яркой и запоминающейся (рис. 2.17).

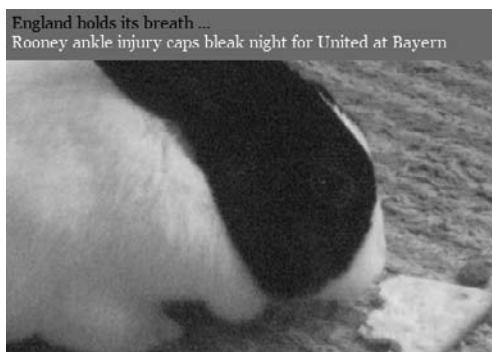


Рис. 2.17. Интересная фотография

Для этой статьи мы будем использовать элемент `<article>`, правда, нам понадобится дополнительный атрибут `<class>` или `<id>`, чтобы создать особое оформление.

Под интересной статьей есть несколько разделов, которые не относятся к основным новостям (на главной странице в них указывается только общая информация, и они являются в первую очередь ссылками на другие страницы, но все же не ссылками в чистом виде). Каждый такой раздел имеет заголовок («Best of guardian.co.uk», «Latest multimedia», «What you're saying»), а затем следует несколько статей. Поэтому логично было бы использовать здесь элементы `<article>` внутри `<section>`.

```
<section>
<h2>Best of guardian.co.uk</h2>
<article>
<h3>Is Britain broken?</h3>
...
</article>

<article>
<h3>Notes and queries</h3>
...
</article>

<article>
<h3>Tech Weekly live: Personal privacy</h3>
...
</article>
...
</section>

<section>
<h2>Latest multimedia</h2>
...
</section>
```

На сайте (но не на скриншоте) есть еще несколько блоков `<nav>` («Trending», «Campaign and Investigations») и «большой футер». Последний, как обсуждалось в главе 1, оформляется с помощью нескольких блоков `<nav>` во всю ширину страницы, расположенных снаружи «реального» `<footer>`, в котором содержатся сведения о конфиденциальности, сроках и условиях использования, а также доступности.

Итак, леди и джентльмены, перед вами HTML5-версия www.guardian.co.uk. Решение любой задачи, выходящей за рамки банальной разметки (а в нашем случае это именно так), допускает несколько разных подходов. Это нормально. HTML — это обобщенный язык, поэтому в нем не предусмотрены элементы для каждого специфического случая. Выбирайте наиболее подходящий и не забывайте о необходимости единообразно оформлять схожие материалы на всех страницах сайта.

ЧТО ТАКОЕ WAI-ARIA

WAI-ARIA (Web Accessibility Initiative's Accessible Rich Internet Applications) — отдельная спецификация, которая «заполняет дыры» в HTML 5 (или любом другом языке разметки) для повышения доступности приложений и веб-страниц.

Представьте себе, что вы написали скрипт для реализации ползунка. Так как в HTML 5 нет встроенного ползунка, с помощью нескольких HTML-элементов (элемента `<input>` и изображений) и JavaScript-кода вы создаете то, что выглядит и функционирует как ползунок. Но в таком случае не существует способа сообщить операционной системе, что этот виджет исполняет роль ползунка, и передать сведения о его текущем состоянии и значении. А если система не располагает этой крайне необходимой информацией, вспомогательные технологии (например, программы экранного доступа) не смогут донести ее до пользователя.

ARIA представляет собой вариант решения этой проблемы путем добавления целого ряда новых атрибутов, которые могут быть понятны браузерам и вспомогательным технологиям.

ПРИМЕЧАНИЕ

Если вы будете использовать новые ARIA-атрибуты, ваши HTML 5-страницы не будут проходить валидацию. Если в остальном ваша разметка построена правильно, не обращайте на это внимания — доступность важнее валидности. Атрибуты ARIA не создают в DOM неразберихи и не вызывают ошибок в браузерах — как и правила CSS, они попросту игнорируются, если браузер их не понимает.

Итак, используя ужасный древний HTML, можно — теоретически — добавить ARIA в такой код.

```
<font size="+5" color="red">I should be a heading</font>
```

В результате получится следующее.

```
<font size="+5" color="red" role="heading" aria-level="2">
- I should be a heading</font>
```

Такой код сообщает агенту пользователя, что этот текст является заголовком уровня 2. Конечно же, это абсурд, так как в HTML уже есть абсолютно допустимый и семантический способ создания такой структуры:

```
<h2>I AM a heading</h2>
```

Разработчик может по невнимательности не добавить необходимые ARIA-атрибуты, в то время как использование правильного элемента `<h2>` само по себе подразумевает «заголовочность» и включает сведения об уровне, поэтому такой синтаксис намного надежнее. ARIA — не панацея и не карт-бланш для тех, кто хочет пренебречь правилами разметки и построить сайт с помощью одних только элементов `<div>` и ``. По возможности создавайте правильную разметку и используйте ARIA только в тех ситуациях, когда не существует другого способа отразить семантику элементов (как в случае с ползунком в HTML 5).

В спецификации ARIA говорится: «мы надеемся, что с течением времени базовые языки будут развиваться, и в результате семантическими станут даже те объекты, которые сейчас требуют использования WAI-ARIA. Когда для определенной возможности будет разрабо-

тана «родная» семантика, разработчик сможет использовать ее и полностью отказаться от WAI-ARIA».

Поэтому, например, к HTML5-элементу `<nav>` не нужно добавлять атрибут `aria-role=navigation`, так как он должен быть встроенным (в идеальном мире). Однако HTML5 вышел относительно недавно, тогда как ARIA уже поддерживается многими вспомогательными технологиями. Так что будет не лишним использовать встроенный элемент *вместе с* данными ARIA — это поможет пользователям вспомогательных технологий. Поэтому HTML5-валидатор рассчитан на HTML5 и ARIA (в то время как валидаторы HTML 5 в случае обнаружения ARIA-атрибутов выдают сообщение об ошибке, поскольку HTML 5 был выпущен раньше ARIA).

ARIA-СТРУКТУРА ДОКУМЕНТА И РОЛИ

В WAI-ARIA определен ряд ролей, которые сообщают вспомогательным технологиям сведения о структуре документа и предназначении основных его частей. Ниже приведен список некоторых из них:

- application;
- article;
- banner;
- complementary;
- contentinfo;
- document;
- form;
- heading;
- main;
- navigation;
- search.

На рис. 2.18 показано, как выглядит простая страница с точки зрения ARIA.

Очевидно, что некоторые из них совпадают с HTML5-элементами: `<article>`, `<form>`, `<header>`, `<nav>`.

Для других ролей такого явного взаимооднозначного соответствия нет. К примеру, `role=banner` «обычно содержит логотип или фирменный знак спонсора сайта, а также специфический для данного сайта инструмент поиска». Баннер обычно располагается вверху страницы и занимает всю ее ширину». Поначалу это напоминает элемент `<header>`, но как мы уже знаем, на странице может быть несколько заголовков. Так что атрибут `role=banner` может применяться только к «заголовку страницы».

Точно так же `contentinfo` определяется как «большая заметная область, которая содержит информацию о родительском документе».

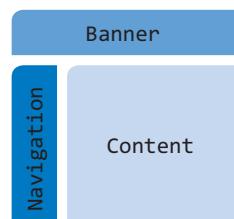


Рис. 2.18.
Простая страница с заголовком, боковой панелью и областью основного содержимого: роли ARIA

В качестве такой информации могут быть указаны, к примеру, сведения об авторском праве и ссылки на заявление о конфиденциальности». Это напоминает элемент `<footer>`, но точнее было бы назвать его «футер страницы», так как другие футеры под это описание не подходят.

Атрибут `role=main` определяет «основную область контента» страницы. В главе 1 мы говорили о том, как алгоритмически вычислить эту область, но поскольку вспомогательные технологии могут уже сейчас использовать ARIA, разумно добавлять этот атрибут к элементу, обрамляющему ваш основной контент. В браузерах, поддерживающих селекторы атрибутов, его можно использовать также и для создания оформления с помощью CSS.

```
div[role=main] {color:red; background-color:yellow;
- font-family: "Comic Sans MS", cursive; ... }
```

Наконец-то: доступность и превосходная типографика в совершенной гармонии друг с другом.

КАК СОВМЕЩАТЬ ARIA И HTML

Мы советуем вам придерживаться следующей точки зрения: использование ARIA во всех случаях, где это уместно, является временной мерой, которая улучшает доступность и не влияет на валидность (но обратите внимание на «Замечания по поводу программ экранного доступа», приведенные ниже). Однако в данной книге мы этого не делаем (поскольку мы пытаемся научить вас HTML5, а не ARIA). Небольшой модуль по адресу <http://github.com/yatil/accessifyhtml5.js> умеет добавлять роли в большинстве наиболее распространенных и обобщенных сценариях использования HTML5, но, к сожалению, он не способен как по мановению волшебной палочки сделать доступными для считывания вспомогательными технологиями тяжелые веб-приложения на базе AJAX.

РЕСУРСЫ ARIA

В спецификациях HTML5 и ARIA есть полезная перекрестная ссылка <http://dev.w3.org/html5/spec/embedded-content-0.html#annotations-for-assistive-technology-products-aria>. Стив Фолкнер из The Paciello Group создал список ARIA-информации, которая не встроена в HTML5 (<http://www.pacielligroup.com/blog/?p=585>).

Мое личное мнение звучит так: если вы следуете правилам спецификации, неспособность браузера или программы экранного доступа адекватно воспринимать контент — не ваша проблема. Но это только мое личное мнение. Вы можете считать иначе; кроме того, в определенных ситуациях вам может потребоваться упростить контент, чтобы разметка соответствовала возможностям этих программ. Конечно, к тому времени, как вы прочитаете эту книгу, часть проблем уже может быть решена. А пока вы сами должны знать своих пользователей и требования в вашей области.

Общую информацию о WAI-ARIA можно найти в статье Геза Лемона «Введение в WAI-ARIA» («Introduction to WAI-ARIA») на <http://dev.opera.com/articles/view/introduction-to-wai-aria/> и в блоге The Paciello Group (<http://www.pacielligroup.com/blog/>). Также я рекомендую две книги: «Universal Design for Web Applications» Венди Чисхолм и Мэтта Мэя (O'Reilly) и «Designing with Progressive Enhancement: Building the Web that Works for Everyone» Тодда

Паркера и др. (New Riders). В них вы найдете полезные практические рекомендации по использованию ARIA.

Спецификацию ARIA можно найти на <http://www.w3.org/WAI/PF/aria/>.

ЗАМЕЧАНИЯ ПО ПОВОДУ ПРОГРАММ ЭКРАННОГО ДОСТУПА

Хьюстон, у нас проблемы.

В 2007 году меня очень беспокоил тот факт, что производители программ экранного доступа не принимают участия в создании спецификации HTML5, и я написал в W3C письмо с просьбой пригласить этих производителей подключиться к работе. В 2009 я спросил разработчика HTML Яна Хиксона, отреагировал ли кто-нибудь на это предложение. Он ответил: «Несколько производителей отреагировали. Но они, к сожалению, сказали, что у них нет на это времени. Правда, после этого компания Apple стала активно заниматься встроенным в Mac OS X программным обеспечением для экранного доступа, и сейчас Apple активно поддерживает с нами связь. Так что, по крайней мере, один производитель вовлечен в этот процесс».

Недавнее исследование (<http://www.accessibleculture.org/research/html5-aria-2011/>) показывает, что старые версии двух самых популярных на рынке программ экранного доступа не могут правильно обрабатывать контент, для разметки которого используются одновременно HTML5 и ARIA (ха-ха), или элементы `<nav>` внутри `<header>`. Однако не все такие программы работают неправильно; VoiceOver компании Apple не пропускает контент, JAWS 12 избавилась от ошибки в версиях 10 и 11, а свободная программа экранного доступа NVDA (<http://www.nvda-project.org/>) читает весь контент и позволяет перемещаться по структурным ARIA-элементам страницы.

ЕЩЕ БОЛЬШЕ НОВЫХ СТРУКТУР!

Пока вы еще ничего толком не узнали. Вообще-то, это неправда: вы знаете уже довольно много. И раз у нас еще остались силы, давайте познакомимся с другими новыми HTML5-элементами и обсудим некоторые отличия от HTML 5. Мы рассмотрим глобальные атрибуты, которые могут быть применены к любому элементу, и скажем «привет» новым HTML5-свойствам, которые останутся за рамками этой книги.

МИКРОДАННЫЕ

Микроданные (microdata) позволяют добавлять семантику содержимому, не используя дополнительные элементы HTML. Здесь наблюдается определенное сходство с RDFa и микроформатами, но все намного проще (хотя и не все с этим согласны).

Микроданные включают пять атрибутов, которые можно использовать с любыми элементами HTML. Наиболее важные из них — `itemscope`, `itemtype` и `itemprop`.

`itemscope` определяет «хват» определенного элемента; он говорит: «этот контейнер охватывает одну вещь и больше ничего».

Например, блок `<div>`, описывающий эту книгу:

```
<div itemscope>
I love that Introducing HTML5 by Bruce Lawson and Remy Sharp
- (ISBN 0321784421)
</div>
```

КАТЕГОРИИ ЭЛЕМЕНТОВ И МОДЕЛИ КОНТЕНТА HTML5

В HTML 5 элементы делились на «блочные» (block-level) и «строковые» (inline). В HTML5 эти названия больше не используются, так как относятся исключительно к оформлению — они дают понять, как браузеры будут отображать содержимое этих элементов, основываясь на своих таблицах стилей по умолчанию. Никакие элементы HTML нельзя назвать традиционно «блочными» или «строковыми».

По умолчанию в CSS любой элемент определяется как `display:inline`, если только их не определяют таблицы стилей браузера по умолчанию или связанный с разметкой великолепный дизайн величайшего и талантливейшего дизайнера, которым вы, несомненно, являетесь. (Не стоит краснеть — я всего лишь говорю правду!)

В HTML5 мы обнаруживаем множество новых моделей контента, включая `phrasing` (в первом приближении — аналог строковых элементов) и `flow` (более-менее имитирующую блочные элементы). Некоторые элементы (`<a>`, `<ins>`, ``) можно с равным успехом отнести как к одному, так и ко второму классу. Кроме того, различают заголовочное содержимое (`<h1>...<h6>`) и элементы секционирования, такие как `<article>`, `<section>`, `<nav>` и `<aside>`.

Также существует встраиваемый контент (контент, импортирующий в документ содержимое с другого ресурса, или контент из другого словаря, вставленного в документ, например `<audio>`, `<canvas>`, `<embed>`, `<iframe>`, ``, `<math>`, `<object>`, `<svg>` и `<video>`), интерактивный (`<a>`, `<audio>` [если присутствует атрибут `controls`], `<button>`, `<details>`, `<embed>`, `<iframe>` и `` [если присутствует атрибут `usemap`], `<input>` [если атрибут `type` находится не в состоянии `hidden`], `<keygen>`, `<label>` и `<menu>` [если атрибут `type` находится в состоянии `toolbar`], `<object>` [если присутствует атрибут `usemap`], `<select>`, `<textarea>` и `<video>` [если присутствует элемент `controls`]), метаданные и другие виды.

Но не стоит паниковать раньше времени от такого изобилия. Все это интуитивно понятно: за исключением того, что `<a>` теперь ведет себя как `<ins>` и `` и может быть «строковым» или «блочным» (если выражаться древними идиомами HTML 5), в терминах стилизации вы не заметите никаких различий. Тем более никакой разницы не будет, если вы станете использовать HTML5 shiv (<http://code.google.com/p/html5shiv/>) — сценарий, помогающий старым браузерам распознавать новые элементы HTML5 в ожидании времен, когда представления по умолчанию будут добавлены в таблицы стилей всех браузеров.

`itemtype` позволяет указать словарь, для того чтобы парсер или программа-обходчик знал, какой тип информации я размещаю (требуется абсолютный URL-адрес):

```
<div itemscope itemtype="http://schema.org/Book">
I love that Introducing HTML5 by Bruce Lawson and Remy Sharp
- (ISBN 0321784421)
</div>
```

Используя `itemprop`, можно назначить свойства контента:

```
<div itemscope itemtype="http://schema.org/Book">
I love that <span itemprop="name">Introducing HTML5</span>
by <span itemprop="author"> Bruce Lawson</span> and
<span itemprop="author">Remy Sharp</span>
(ISBN <span itemprop="isbn">0321784421</span>)
</div>
```

Обратите внимание, что я использовал свойства `name` (название книги), `isbn` (код ISBN) и дважды `author` — указав Брюса и еще какого-то парня.

Фактическое значение, которое будет связано с `itemprop`, зависит от того, какому элементу принадлежит этот атрибут. Чаще всего выбирается текстовое содержимое элемента, за исключением следующих случаев:

- если у элемента также есть атрибут `itemscope`, значением считается то, что создается данным элементом;
- если речь идет об элементе `<meta>`, значением считается значение атрибута `content` данного элемента;
- если это элемент `<audio>`, `<embed>`, `<iiframe>`, ``, `<source>`, `<track>` или `<video>`, в качестве значения выбирается содержимое атрибута `src` данного элемента, разрешенное в абсолютный URL-адрес;
- если это элемент `<a>`, `<area>` или `<link>`, также выбирается атрибут `src` элемента, разрешенный в абсолютный URL-адрес;
- если это элемент `<object>`, в значение помещается атрибут `data` элемента, разрешенный в абсолютный URL-адрес;
- для элемента `<time>` с атрибутом `datetime` значением считается значение атрибута `datetime`.

Элементы микроданных могут быть вложенными, поэтому информацию об авторах можно дать в форме электронных визиток vCard. В `span`, содержащий имя автора, мы вставляем `itemprop` данного автора, представляющий его взаимоотношение со словарем книги, а `span` более высокого уровня мы определяем как контейнер, содержащий данные о человеке, поэтому добавляем `itemscope` и `itemtype`, указывающие на словарь hCard:

```
<div itemscope itemtype="http://schema.org/Book">
I love that <span itemprop="name">Introducing HTML5</span>
    By
    <span itemprop="author" itemscope itemtype=
        "http://microformats.org/profile/hcard">
        <span itemprop="fn">Bruce Lawson</span>
    </span>
    and
    <span itemprop="author" itemscope itemtype=
        "http://microformats.org/profile/hcard">
        <span itemprop="fn">Remy Sharp</span>
    </span>
    (ISBN <span itemprop="isbn">0321784421</span>)
</div>
```

ПРИМЕЧАНИЕ

Это несерьезный пример микроданных, который не дает никакой дополнительной информации о наших авторах, помимо их имен; цель данного примера — продемонстрировать возможность вкладывания элементов один в другой. Намного более полный пример вы найдете в спецификации по адресу <http://www.whatwg.org/specs/web-apps/current-work/multipage/microdata.html#mdvocabs>.

Большинство сценариев применения микроданных реализовываются ненамного сложнее. Однако есть еще два атрибута, о которых также следует упомянуть для полноты информации.

itemref

Иногда возникает необходимость перечислить дополнительные элементы, чтобы пользовательский агент мог считать их и найти пары из имени и значения. Это приходится делать для элементов, не являющихся потомками элемента с атрибутом `itemscope`. Внутри элемента с атрибутом `itemscope` можно через пробел перечислить уникальные маркеры (чувствительные к регистру), соответствующие идентификаторам элементов на той же странице.

itemid

Если есть желание, то можно также использовать атрибут `itemid`. Это глобальный уникальный идентификатор — глобальный не только для вашего веб-сайта, но и для всей сети. Он может представлять, например, ISBN или URL или вообще все что угодно, если только оно гарантированно уникально. Делая так, вы словно распыляете над своим сайтом волшебную пыльцу Семантической Сети: программы-обходчики и агрегаторы контента «узнают», что на вашем сайте говорится о тех же вещах, что и на каких-то других страницах, потому что на них встречаются одинаковые значения `itemid`. (Правда, на момент написания этой главы в словарях микроданных, опубликованных Bing, Google и Yahoo! на schema.org, `itemid` вообще не используется.)

Решив использовать `itemid`, вы должны также указать словарь, поддерживающий глобальные идентификаторы: «атрибут `itemid` можно добавлять только к элементам, имеющим два других атрибута, `itemscope` и `itemtype`. Его нельзя добавлять к элементу с атрибутом `itemscope`, атрибут `itemtype` которого указывает на словарь, не поддерживающий глобальные идентификаторы (определяется спецификацией словаря)».

Более подробную информацию о микроданных вы найдете в следующих источниках:

- «Расширяем HTML5 — микроданные» («Extending HTML5 — Microdata») Оли Стадхолм (<http://html5doctor.com/microdata>);
- «Учебник по микроданным» («Microdata Tutorial») Таба Аткинса младшего (<http://www.xanthis.com/blog/b4570>);
- «Использование нескольких словарей в микроданных» («Using Multiple Vocabularies in Microdata») Джени Теннисон (<http://www.jenitennison.com/blog/node/161>).

API DOM MICRODATA

С микроданными также связан свой API объектной модели документа с микроданными, позволяющий манипулировать объектами и свойствами. Метод `document.getItems()` возвращает для указанной страницы список узлов, содержащий объекты микроданных. Если запускать его без аргумента, метод собирает все объекты на странице; если передать ему URL, связанный с атрибутом `itemtype`, то метод вернет только объекты с данным значением атрибута `itemtype`.

На момент написания этой главы только предварительные версии Opera 12 поддерживают этот API DOM.

<ASIDE>

В главе 1 мы использовали `<aside>` для добавления боковой панели. Этот элемент используется для «раздела страницы, который косвенно связан с окружающим контентом, но который должен располагаться отдельно». В типографике такие разделы обычно оформляются в виде врезок. Этот элемент может использоваться для создания типографских эффектов (например, врезок и боковых панелей), для рекламы, для групп элементов `<nav>` и для другого контента, который должен располагаться отдельно от основного».

Например, внутри `<article>` элемент `<aside>` — отличное место для информации, имеющей косвенное отношение к статье, или цитат, но (скажу, забегая вперед) не для панели глобальной навигации по сайту.

По умолчанию `<aside>` имеет ARIA-роль `note`, но это можно изменить, добавив `role="complementary"` или (если он оформляет форму поиска) `role="search"`.

<BDI>

В спецификации говорится, что `<bdi>` «представляет фрагмент текста, который необходимо изолировать от окружающего контента в целях двунаправленного форматирования текста». Чего?! Я попросил Ричарда Ишида (@r12a в Twitter — добавьте его в свой список!), директора по локализации в W3C, объяснить, что же тут имеется в виду, и он в своей бескрайней доброте написал ради этого отдельную публикацию в блоге (<http://rishida.net/blog/?p=564>), которую я, с его разрешения, привожу здесь в сжатом виде.

Спецификация HTML5 содержит набор новых возможностей, поддерживающих вывод текста на веб-страницах в обоих направлениях. В языки с написанием справа налево — такие как арабский, иврит, персидский, тана, урду и т. д. — очень часто примешиваются слова и фразы на английском или каком-то другом языке с написанием слева направо. Результатом такой смеси становится двунаправленный текст или *bidi-текст* (от слова *bidirectional*).

HTML 5.01 благодаря помощи алгоритма Unicode Bidirectional достаточно хорошо справляется с двунаправленным текстом. Тем не менее остается еще ряд нерешенных проблем, связанных с встраиванием текста, полученного от пользователя или запрошенного из хранилища.

В следующем примере названия ресторанов добавляются на страницу из базы данных. Символы иврита показаны с помощью кодов ASCII:

```
<p>Aroma - 3 reviews</p>
<p>PURPLE PIZZA - 5 reviews</p>
```

На рис. 2.19 и 2.20 показано, что мы бы ожидали увидеть и что мы видим на самом деле соответственно.

Проблема возникает из-за того, что, по мнению браузера, символы «-5» — это тоже часть текста на иврите. Так утверждает алгоритм Unicode Bidi Algorithm, который, кстати, обычно правильно распознает текст. Но не в этом случае.



Рис. 2.19. Вот такой мы бы хотели видеть нашу страницу



Рис. 2.20. А так наша bidi-страница выглядит в действительности. Обратите внимание, что цифра 5 отделена от слова reviews. Текст теперь совершенно потерял смысл

Вопрос в том, как исправить ситуацию. Нам поможет трюк с элементом `<bdi>` — нужно окружить этим тегом текст, который требуется изолировать от остального контента (`bdi` расшифровывается как `bidi-isolate`, то есть «дву направленная изоляция»).

```
<p><bdi>Aroma</bdi> - 3 reviews</p>
<p><bdi>PURPLE PIZZA</bdi> - 5 reviews</p>
```

Теперь bidi-алгоритм считает текст на иврите и символы «-5» отдельными фрагментами контента и выстраивает их согласно общему направлению текста (в данном случае слева направо).

Вы наверняка обратили внимание, что в предыдущем примере тег `<bdi>` окружает также текст «`Aroma`». Разумеется, в этом нет необходимости, но и вреда также не причинит. Кстати, это помогает при написании PHP-сценариев, например чего-то вроде этого:

```
foreach $restaurant echo "<bdi>".$restaurant['name']."
- </bdi> - %1 reviews";
```

Это решение позволяет обработать все названия, поступающие из базы данных, независимо от того, какому языку они принадлежат.

Использование атрибута dir с bdi

С элементом `<bdi>` можно использовать атрибут `dir`, позволяющий задавать базовое направление. Когда речь идет о простых строках текста, вроде PURPLE PIZZA, особой необходимости в нем нет. Тем не менее, если `<bdi>` содержит двунаправленный текст, лучше дополнительно указать базовое направление.

Во времена, предшествующие появлению HTML5, атрибут `dir` можно было использовать только с `ltr` и `rtl`. Проблема в том, что в ситуациях, подобных описанной выше, когда строки извлекаются из базы данных или вводятся пользователем, вы можете не знать, с чем имеете дело в каждый конкретный момент.

Вот почему в HTML5 для атрибута `dir` добавили новое значение `auto`, которое по умолчанию считается установленным для элемента `bdi`. Значение `auto` приказывает браузеру проверять в элементе первый символ со строгим контролем типов и на его основе определять базовое направление элемента. Если выясняется, что это символ из иврита (или арабского языка и т. д.), то элементу присваивается направление `rtl`. А если это символ латинского алфавита, то выбирается направление `ltr`.

Почти всегда результат получается правильным и ожидаемым, ошибки возникают крайне редко.

Обратите внимание, что данная возможность пока нигде не реализована, но я счел необходимым добавить информацию о ней, так как сведений о `bdi` также не очень много.

<DETAILS>

Мне очень нравится элемент `<details>`. Его прелесть в том, что он по природе своей поддерживает очень распространенное поведение — область, которая может разворачиваться и сворачиваться, — избавляя нас от необходимости прибегать к помощи различных сценариев JavaScript (или подтягивания целой библиотеки jQuery, — а я это наблюдал на таком количестве сайтов, что мне уже даже не смешно).

```
<details>
  <summary>Photograph details</summary>
  <p>Photograph taken on <time datetime=2009-12-25>Xmas
  - Day 09</time> with a Canon IXUSi.</p>
  <p><small>Copyright Bruce Lawson,
  - <address>bruce@brucelawson.co.uk</address></small>.etc.</p>
  - </details>
```

Содержимое внутреннего элемента `<summary>` действует как элемент управления: при его активации мышью или клавиатурой остальное содержимое `<details>` сворачивается или разворачивается. Если элемент `<summary>` не указан, браузер сам добавляет соответствующий текст, например «`details`» (детали) или локализованный вариант. Обычно браузеры также добавляют специальный значок, указывающий на то, что текст можно развернуть (к примеру, стрелку вниз).

У `<details>` есть необязательный атрибут `open`: с его помощью можно указать, что при загрузке страницы текст элемента будет развернут.

```
<details open>
```

ПРИМЕЧАНИЕ

Кстати, этот элемент не обязан содержать только текстовую разметку — внутри может располагаться форма регистрации, поясняющий видеофайл, таблица исходных данных диаграммы или описание структуры таблицы для пользователей вспомогательных технологий, людей с пониженной обучаемостью или тех, кто (как я) просто не понимает чисел.

На момент написания этой главы только Google Chrome 12 поддерживает `<details>`. Но используйте его с осторожностью и тщательно тестируйте, так как он может вызывать определенные проблемы с обращением к содержимому: данным элементом невозможно управлять с клавиатуры, и пользователю обязательно требуется мышь. Надеюсь, в будущих версиях Chrome этот недостаток будет исправлен.

<FIGURE>

Я всегда испытывал ощущение какой-то семантической неаккуратности, добавляя к рисунку название для того, чтобы дать пояснение или указать автора, потому что раньше сделать это можно было только с помощью текста, встраиваемого в окружающий контент, причем явно привязать его к изображению было невозможно. Раньше для этого не было специ-

альных структур. Возможно, это немного странно, но именно поэтому я так рад появлению элемента `<figure>`, который заключает в себя изображение (или видео, или фрагмент кода, или цитату) и его заголовок, обрамляемый тегом `<figcaption>`.

```
<figure>
<img src=welcome.jpg>
<figcaption>
Bruce and Remy welcome questions
<small>Photo &copy; Bruce's mum</small>
</figcaption>
</figure>
```

Добавив к этой разметке стили, можно создать изящное оформление (рис. 2.21).



Рис. 2.21. Элементы `<figure>` и `<figcaption>` плюс немного шикарных CSS3-эффектов

Обратите внимание, что у изображения отсутствует атрибут `alt`. В первом издании я добавлял пустой атрибут `alt=""`, но это было некорректно.

Если текст внутри тега `figcaption` говорит все, что вашим зрителям необходимо знать об иллюстрации («Пиппа Миддлтон и Реми Шарп позируют для фотографов на красной дорожке перед премьерой фильма по роману Джейн Остин «Гордость и ECMAScript»), не стоит дублировать его в атрибуте `alt`, так как дублирующийся контент очень быстро начинает раздражать.

В первом издании этой книги, для того чтобы избежать дублирования контента, я также добавлял `alt=""` к элементу ``. Однако если атрибут `alt` изображения пуст, то подразумевается, что с ним связана ARIA-роль `role=presentation` (для того чтобы подчеркнуть презентационное предназначение). Таким образом, элемент исключается из дерева элементов страницы, используемого программами чтения экрана (да, никакие пользовательские агенты пока что этого не делают, но, в целом, по плану должно быть так).

Поскольку изображение выполняет не только презентационную функцию, но и играет определенную роль в контенте (в противном случае, вы не выделяли бы его как отдельную иллюстрацию и не присваивали бы ему заголовок), не следует оставлять атрибут `alt` пустым. Лучше вообще не использовать его.

Стив Фолкнер создал весьма полезный документ под названием «HTML5: техники представления полезных альтернатив тексту» («HTML5: Techniques for providing useful text

`alternatives») — это всего лишь первый публикуемый черновик, поэтому вероятность изменений очень, очень велика, — в котором он пишет, в частности, следующее:`

Обстоятельства, в которых недопустимо использовать пустой или равный `null` атрибут `alt`: изображение находится внутри элемента `figure` и с ним связан заголовок, определенный с помощью элемента `figcaption`. (<http://dev.w3.org/html5/alt-techniques/>)

С другой стороны, вам может показаться, что в приведенном выше примере неплохо было бы добавить текст в атрибут `alt` (``), так как в противном случае пользователи программ чтения экрана не поймут шутку.

Пока браузеры не научились «понимать» элемент `figure` и делать все это автоматически, в качестве бонуса для пользователей таких программ было бы полезно (хотя и не обязательно) добавлять ARIA-атрибуты, связывающие изображение с заголовком.

В случае отсутствия заполненного текстом атрибута `alt` используйте `aria-labelledby` и привязывайте `id` элемента `figcaption` к элементу `img`:

```
<figure>
<img src=welcome.jpg aria-labelledby=figcap219>
<figcaption id=figcap219>
Pippa Middleton and Remy Sharp pose on the red carpet at
- the premier of the Jane Austen movie <cite>Pride and
- ECMAScript</cite>
</figcaption></figure>
```

Если же `alt` с текстом есть, используйте `aria-describedby`:

```
<figure>
<img src=welcome.jpg
alt="Bruce and Remy glower menacingly into the camera"
aria-describedby=figcap219>
<figcaption id=figcap219>
Bruce and Remy welcome questions
<small>Photo © Bruce's mum</small>
</figcaption></figure>
```

HTML5 И АЛЬТЕРНАТИВНЫЙ ТЕКСТ ИЗОБРАЖЕНИЙ

Очень часто улицы наших городов наполняются рыданиями и жалобами на то, что в определенных обстоятельствах валидатор не пинает вас за отсутствие атрибута `alt` у элемента `` (хотя я сам кого угодно пну, если еще хоть раз услышу выражение «тег `alt`»):

- когда есть `<meta name=generator>`, отсутствие `alt` не считается ошибкой;
- когда есть `title`, отсутствие `alt` не считается ошибкой;
- когда есть `figcaption`, отсутствие `alt` не считается ошибкой.

Я рекомендую в 99,99% случаев добавлять атрибут `alt` к изображениям, а для чисто декоративных рисунков оставлять этот атрибут пустым. Редкими исключениями будут изображения внутри `<figure>`, о которых мы поговорили выше. Если назначение изображения недвусмысленно сформулировано в `<figcaption>`, вообще не используйте `alt`. Оставшиеся 0,01% относятся к написанию шаблонов для автоматически генерируемых веб-страниц с импортированием изображений, для которых значение `alt` получить невозможно, например, к автоматическому захвату изображений из видеопотока веб-камеры (в такой ситуации используйте в заголовке `<meta name=generator>`).

<MARK>

Элемент `<mark>` позволяет выполнять действие, аналогичное использованию маркера, когда мы отмечаем определенные слова в печатном тексте. Это не то же самое, что выделение — для этого используется ``. Но если у вас уже есть текст и вы хотите вывести на передний план фрагмент, не выделенный в исходном тексте, можно использовать `<mark>`, а затем в таблице стилей задать для него курсивный шрифт или желтый фон. При печати вы часто будете видеть фразы вроде «*my italics*» или «*emphasis added*».

В спецификации также сказано: «если этот элемент используется в основном тексте документа, он обозначает часть документа, которая была выделена по причине ее вероятной значимости для того, чем в данный момент занимается пользователь».

Приведу пример: на моем сайте используется адаптированная версия скрипта Searchhi Стюарта Лэнгриджа (<http://www.kryogenix.org/code/browser/searchhi/>), который в случае, если на страницу ссылается поисковый движок, обходит DOM и заключает каждое слово из поискового запроса в элемент `<mark>`, который оформляется приятным розовым цветом. Было бы неправильно использовать для этого элементы `` или ``, так как здесь мы имеем дело не с эмфатическим выделением — которое изменило бы смысл содержимого страницы, — а с выделением слов, релевантных для того, чем в настоящий момент занимается пользователь: просмотра информации об определенном поисковом слове.

<RUBY>, <RP>, <RT>

`<ruby>` — полезное дополнение для тех, кто пишет контент на азиатских языках. Дэниэл Дэвис написал очень полезную статью «HTML5-элемент `<ruby>` в словах из одного слога и меньше» («The HTML5 `<ruby>` element in words of one syllable or less»; <http://my.opera.com/tagawa/blog/the-html5-ruby-element-in-words-of-one-syllable-or-less>), в которой он объясняет, как работает этот элемент (а также связанные теги `<rt>` и `<rp>`), на примере японского языка (с разрешения автора привожу отрывок из его статьи).

В любом фрагменте японского текста (баннер, статья, правовой документ и т. д.) используется несколько систем письма: кандзи, хирагана и катакана. Часто бывает так, что пользователи не умеют читать кандзи в основном потому, что знаки кандзи могут иметь несколько вариантов произношения. Один из примеров нерегулярного или неоднозначного произношения таких знаков — имена людей и названия мест.

- 日 может произноситься как «nichi», «hi» или «ka».
- 本 может произноситься как «hon» или «moto».
- 日本 может произноситься как «nihon» или «nippon».

Чтобы текст был более понятным для читателя, иногда над знаком кандзи пишется его произношение, и при этом используется хирагана. По-японски это называется *фуригана*, а по-английски — *ruby* (от названия мелкого шрифта размером 5,5 пт, использующегося в британской типографской традиции для подобного рода аннотаций). Такой шрифт часто используется в газетах и книгах, но на сайтах он встречается редко, поскольку не так просто разместить мелкий текст над обычным в пределах одной строки. Элемент `<ruby>` создан для того, чтобы решить эту проблему.

Согласно текущей спецификации HTML5, элемент `<ruby>` — это строковый элемент, содержащий слово (или знак), которое вы хотите уточнить. Это может выглядеть так:

```
<ruby>日本</ruby>
```

Сам по себе такой код не делает ничего. Поэтому мы добавим произношение либо к каждому знаку, либо (как в нашем примере, и я рекомендую вам поступать именно так) к целому слову. Для этого будем использовать тег `<rt>`, который обозначает текст `<ruby>`.

```
<ruby>日本<rt>にほん</rt></ruby>
```

Если оставить это в таком виде, браузеры, поддерживающие этот элемент, отобразят произношение над текстом кандзи, но остальные браузеры проигнорируют теги и отобразят и текст, и произношение на одной строке в линейном порядке. Чтобы решить эту проблему, добавим скобки, обрамленные специальным тегом `<rp>`, который разумно скрывает эти скобки в браузерах, поддерживающих элемент `ruby`. Это означает, что вы можете писать произношение в скобках, которые будут отображаться в браузерах, не поддерживающих `ruby`, тогда как остальные браузеры отобразят произношение без скобок над основным текстом (рис. 2.22).

```
<ruby>日本<rp>(</rp><rt>にほん</rt><rp>)</rp></ruby>
```

日本	日本 (にほん)
supporting browser	non-supporting browser

Рис. 2.22. В браузерах, поддерживающих `ruby`, текст-транскрипция отображается над основным текстом. В браузерах, не поддерживающих `ruby`, текст-транскрипция отображается в скобках и следует после основного текста

<WBR>

В Netscape 4 теперь уже (к великой нашей радости) стандартизованный в HTML5 элемент `<wbr>` сообщает браузеру, что тот может (но не обязан) вставить в данном месте разрыв строки, если он испытывает необходимость добавить его где-то здесь.

ПЕРЕОПРЕДЕЛЕННЫЕ ЭЛЕМЕНТЫ

В HTML5 не только добавлены новые элементы, но и переопределены некоторые уже существующие. И вот перед нами наши старые друзья: кто-то полностью изменился, а кто-то просто довел до совершенства свою прическу.

<ADDRESS>

Элемент `<address>` используется для указания контактной информации автора текущей статьи или документа, а не как универсальный элемент для почтовых адресов.

Новым в отношении этого элемента является то, что теперь на странице может быть несколько элементов `<address>` — по одному в каждом `<article>`. Информация об авторе,

привязанная к определенному `<article>`, не относится к вложенным элементам `<article>`. Так что сообщение блога внутри `<article>` может содержать элемент `<address>` для его автора, а каждый комментарий (который, как вы помните, является вложенным `<article>`) может содержать `<address>` того, кто его оставил.

Мы читатели информационного хайвея, и, вероятно, все мы используем для связи с авторами электронные средства. Таким образом, в качестве контактной информации может быть указан адрес электронной почты, почтовый адрес и т. д. Для разметки могут быть использованы микроформаты, RDFa или микроданные (рис. 2.23).

```
<address>
<a href="http://introducinghtml5.com">
<img src=qr.png alt="">
Bruce Lawson, Remy Sharp</a>
</address>
```

Между прочим, форматирование адресов (а также добавление разрывов строк в стихотворениях и примерах кода) — это одна из немногих оставшихся причин использования элемента `
`:

```
<address>
Dunhackin<br>
123 Standards Boulevard<br>
Semanticchester<br>
UK<br>
</address>
```

<CITE>

В HTML 5 элемент `<cite>` можно было использовать для обозначения имени говорящего:

```
As <cite>Harry S. Truman</cite> said, <Q lang="en-us">
- The buck stops here.</Q>
```

В HTML5 это запрещено: «имя человека не является названием произведения — даже если это выдающийся человек, — и таким образом, этот элемент не должен использоваться для разметки имен людей».

Просто бред! Из-за этого правила существующий контент, написанный в соответствии с правилами HTML 5, недопустим с точки зрения HTML5. Правда, валидаторы все равно никогда не смогут признать этот код недопустимым, поскольку не существует способа автоматически отличить имя «Harry S. Truman» от одноименной биографии.

В статье «Провокация бунта» («Incite a riot») Джереми Кейт написал: «Присоединяйтесь к кампании гражданского неповиновения ненужным, запрещающим и приводящим к обратной несовместимости изменениям в правилах использования элемента `cite`» (<http://24ways.org/2009/incite-a-riot>).

Я согласен с ним. Используйте `<cite>` для имен, если хотите.



Bruce Lawson, Remy Sharp

Рис. 2.23. Элемент `<address>`, содержащий QR-код в качестве контактной информации

<DL>

В HTML 5 `<dl>` обозначал список определений, состоящий из слова и нескольких его определений. Все было очень четко и понятно, но потом на спецификацию опустился туман, в котором мы моментально заблудились. Если конкретно, в спецификации также предлагается использовать `<dl>` для разметки диалогов, и даже приводятся примеры соответствующего кода. Таким образом, этот элемент часто неправильно использовался для разметки произвольных пар имя/значение независимо от того, определяет ли одно другое.

В HTML5 сфера применения этого элемента расширена: теперь это «связанный список, состоящий из нуля или более групп имя/значение... Группы имя/значение могут представлять собой слова с определениями, разделы метаданных со значениями и др.». Ниже приведен пример списка книг из коллекции Реми, в котором используются элементы `<dt>` и `<dd>` для указания названия и автора (или авторов).

```
<dl>
  <dt>History of French plastic sandals</dt>
  <dd>Phillipe Philoppe</dd>
  <dt>J-Lo's plastic surgery: a profile</dt>
  <dd>Hugh Jarce</dd>
  <dt>The Orpheus and Eurydice myth</dt>
  <dd>Helen Bach</dd>
  <dt>The Proctologist and the Dentist</dt>
  <dd>Ben Dover</dd>
  <dd>Phil McCavity</dd>
</dl>
```

, <i>

`` означает эмфатическое выделение, которое немного изменяет значение предложения. Ответ на вопрос «Did you say you live in Paris?» мог бы выглядеть так:

```
<p>No, my <em>name</em> is Paris. I live in <em>Troy</em>.  
- Cloth-ears.</p>
```

Если вы хотите использовать несколько степеней важности, добавьте вложенные элементы ``, чтобы сделать содержимое более выделенным.

В спецификации говорится, что элемент `<i>` «представляет собой небольшой отрезок текста, который произносится другим голосом или с другим настроением — не так, как обычно; это может быть сложный адрес, технический термин, идиоматическое выражение на другом языке, мысль, имя корабля или другой текст, который в типографике выделяется курсивом».

Ниже приведены несколько примеров использования элемента `<i>`, где `` был бы неуместен.

```
<p>The <i>Titanic</i> sails at dawn.</p>
<p>The design needs a bit more <i lang=fr>ooh la la</i>.</p>
<p>You, sir, deserve a jolly good kick up the <i>gluteus maximus</i>!</p>
```

<HR>

Элемент `<hr>` теперь не зависит от среды и обозначает «тематический разрыв на уровне параграфа». На HTML5doctor.com есть очень меткий комментарий: «Это эквивалент обозначения «***», которое часто используется в рассказах и эссе». Мы хотели отменить его как историческую редкость, когда доктор Оли Стадхольм написал, что «японские дизайнеры достаточно часто используют `<hr>` для обособления разделов. Обычно они скрыты благодаря CSS, но видны в сHTML-браузерах для мобильных телефонов, поддерживающих только базовые правила CSS и не отображающих визуальное оформление (и вместе с ним визуальное обособление разделов)».

Если только большую долю вашей аудитории не составляют пользователи таких телефонов, мы рекомендуем использовать вместо этого структурированный контент и заголовки вместе с изящными разделителями CSS, а о `<hr>` забыть. Таким образом, на ваших страницах уменьшится объем разметки; кроме того, трудно сделать стиль `<hr>` одинаковым во всех браузерах.

Определение нашего старого доброго друга, упорядоченного списка, не поменялось, однако он получил два новых атрибута.

В HTML 5 использование атрибута `start` для элемента `` было нежелательным, поскольку он считался атрибутом представления. К счастью, HTML5 отменил это неверное решение. Если вы хотите начать нумерацию списка не с одного, а с пяти, используйте такой код:

```
<ol start=5>
```

Интересное нововведение, еще не реализованное ни в одном браузере, — это атрибут `reversed`. Рассмотрим следующий пример.

```
<h3>Top five dreamy mega-hunks</h3>
<ol reversed>
  <li>Brad Pitt</li>
  <li>George Clooney</li>
  <li>Orlando Bloom</li>
  <li>Remy Sharp</li>
  <li>Bruce Lawson</li>
</ol>
```

Такой код создает список, в котором нумерация ведется в обратном направлении: от пяти (мистер Питт) к одному (я). Простите, Брэд, Джордж и Орландо, но что вы знаете об HTML5?

<S>

В HTML 5.01 элементы `<strike>` и `<s>` использовались для оформления текста, перечеркнутого прямым горизонтальным отрезком. В HTML5 элемент `<s>` сохранился, и он предназначен для предоставления содержимого, которое теперь считается ошибочным или

не относящимся к теме документа и, следовательно, «вычеркнутого» из текста. Например, с помощью данного элемента на ценнике со специальным предложением можно оформить старую цену.

```
<p>Photograph of Remy Sharp in mankini. <s>&pound;100</s>
- Now: 12 pence.</p>
```

<SMALL>

Элемент `<small>` был полностью переопределён. Из простого универсального элемента представления, делающего текст мелким, он превратился в «мелкий шрифт», который «обычно характерен для отказов, предупреждений, правовых ограничений или авторских прав. Мелкий шрифт также иногда используется для указания авторства или для соблюдения условий лицензирования».

Возможно, вы даже не заметите последствий такого переопределения, так как браузеры, скорее всего, продолжат всего лишь уменьшать размер шрифта, как и раньше. Однако согласно новой семантике, `<small>` теперь также относится к той части радиообъявлений, которую дикторы очень быстро тараторят в самом конце, поэтому программа чтения экрана с настройками акустического воспроизведения по умолчанию может попытаться имитировать это.

Если вся страница посвящена «юридическим» вопросам, не используйте `<small>`. В таком случае юридический текст — *и есть* основной контент страницы, так что нет необходимости выделять его с помощью этого элемента. Он используется только для коротких отрезков текста. `<small>` не имеет отношения к `` и ``.

**, **

Элемент `` выделяет фрагменты контента с высокой важностью, однако в отличие от `` не меняет смысла предложения. Рассмотрим пример:

```
<p><strong>Warning! This banana is
- dangerous.</strong></p>
```

Можно использовать вложенные элементы ``, чтобы придать тексту особую важность.

Элемент `` «представляет собой отрезок контента, к которому внимание привлекается по каким-то чисто утилитарным причинам — ему не придается дополнительная значимость, и он не «произносится» другим голосом или с другим настроением. Это могут быть ключевые слова в аннотации, названия продуктов в обзоре, слова в повествительном наклонении в интерактивном текстовом приложении или вступительные слова в начале статьи».

Например:

```
<p>Remy never forgot his fifth birthday—feasting on
- <b>powdered toast</b> and the joy of opening his gift:
- a <b>Log from Blammo!</b>.</p>
```

<U>

Элемент `<u>` также когда-то использовался исключительно для оформления, а теперь приобрел Новое! Улучшенное! семантическое значение в свежепричесанном определении, которое, тем не менее, не кажется особенно полезным.

Спецификация говорит, что он «представляет фрагмент текста с неявным, хотя и явно оформленным текстовым примечанием. Например, это может быть имя собственное в тексте на китайском языке или текст с грамматической ошибкой».

УДАЛЕННЫЕ ЭЛЕМЕНТЫ

Некоторые элементы, знакомые вам по HTML 5, были признаны полностью устаревшими в HTML5: `<applet>` (вместо него используйте `<embed>`), `<big>`, `<blink>`, `<center>`, `` и `<marquee>`. Они не пройдут валидацию, и разработчикам не следует их использовать. Также были отменены фреймы (но `<iframe>` остался). Туда им и дорога.

Конечно же, HTML5-браузеры должны продолжать отображать эти милые почившие элементы, так как «на воле» их все еще очень много. Но вы должны избегать их, как будто это тарантулы, зомби, тигры-людоеды, сэндвичи с плутонием или песни Селин Дион.

ГЛОБАЛЬНЫЕ АТРИБУТЫ

В HTML5 добавлено также несколько глобальных атрибутов, которые могут быть применены к любому элементу. О них мы поговорим в этой главе.

ACCESSKEY

Атрибут `accesskey` позволяет разработчику указать клавишное сочетание, активирующее элемент или переводящее на него фокус. Он был добавлен еще в HTML 5 для повышения уровня доступности содержимого. Однако использовали его редко — во-первых, из-за проблем с обнаружением, а во-вторых, из-за того, что большинство возможных комбинаций конфликтует с клавишными сочетаниями в программах чтения экрана и подобных вспомогательных средствах.

Так как HTML5 предназначен для веб-приложений, а продвинутые пользователи отдают предпочтение клавишным сочетаниям, а не щелчкам мышью, атрибут `accesskey` решили не удалять из HTML5 и его можно использовать с любым элементом.

Для предотвращения конфликтов с другими приложениями и собственными клавишными сочетаниями браузера теперь в атрибуте `accesskey` допускается указание нескольких альтернативных сочетаний. В спецификации приводится такой пример:

```
<input type="search" name="q" accesskey="s 0">
```

Там дается следующее разъяснение: «с полем поиска связываются две клавиши доступа: `s` и `0` (в указанном порядке). Пользовательский агент на устройстве с полноразмерной клавиатурой может выбрать клавишное сочетание `Ctrl+Alt+S`, тогда как пользовательский агент на небольшом устройстве только с цифровой клавиатурой может установить в качестве клавишного сочетания всего лишь клавишу `0` без дополнений».

CONTENTEDITABLE

Разработанный Microsoft атрибут `contenteditable` был затем переосмыслен и реализован другими браузерами и теперь официально является частью HTML5.

Для браузеров `contenteditable` означает две вещи. Во-первых, пользователи могут редактировать содержимое элементов, к которым добавлен этот атрибут: этот элемент должен быть выделяемым, и браузер должен отображать курсор для выбора места, которое будет отредактировано. Во-вторых, текст можно сделать полужирным, изменить шрифт, добавить списки, заголовки и т. д. `contenteditable` является булевым атрибутом, поэтому в качестве его значения можно задать `true` или `false`. Хотя заглавные буквы в разметке уже не имеют никакого значения, атрибут DOM (если бы вы задавали его программным способом с помощью JavaScript) требует написания `contentEditable` (обратите внимание на заглавную Е). В DOM также есть свойство `isContentEditable`, которое позволяет определить, является ли элемент редактируемым, поскольку флагок `contenteditable` мог быть унаследован от родительского элемента.

Чтобы сделать весь документ редактируемым, можно использовать `document.designMode = 'on'` (обратите внимание — не '`true`'). Однако для этого необходим JavaScript — эквивалентного атрибута, который можно было бы задать в коде HTML, не существует.

Наконец, к любому контенту, выбранному (то есть выделенному) пользователем, может быть применен ряд команд, например `document.execCommand('bold')`. Стандартные сочетания клавиш, позволяющие делать текст полужирным или курсивным (такие как `CTRL+B` и `CTRL+I` соответственно в Windows и Linux), также затрагивают DOM, добавляя вокруг редактируемого элемента теги `` или `<i>`.

Если вы используете `contenteditable` для формы CMS, вы захотите в определенный момент сохранять изменения на сервере. Для этого не существует отдельного API-метода, но поскольку изменения пользователя меняют DOM, для сохранения в CMS достаточно отправить на сервер `innerHTML` редактируемого элемента (или весь документ, если вы находитесь в режиме `designMode`).

DATA-* (ПОЛЬЗОВАТЕЛЬСКИЕ АТРИБУТЫ ДАННЫХ)

HTML5 допускает наличие у любых элементов пользовательских атрибутов данных, которые можно использовать для передачи данных локальным сценариям. Раньше, чтобы хранить специальные данные в разметке, авторам приходилось прибегать к неприятным решениям вроде применения классов использования: `<input class="spaceship shields-5 lives-3 energy-75">`. В этом случае скрипту приходилось тратить время на сбор имен классов, таких как `shields-5`, разбиение их у знака-разделителя (в данном случае его роль играет дефис) и извлечение значения — все это слишком неопрятно. Кроме того, многие считают, что такое использование атрибута `class` недопустимо, ведь он служит (согласно HTML 5.01) в качестве якоря для стилизации или для «обычной обработки пользовательскими агентами».

В своей книге 2007 года «PPK на JavaScript» (New Riders, ISBN 0321423305) Петер-Пауль Кох объясняет, как это делается и почему на некоторых страницах HTML 5 он решил использовать специальные атрибуты. Таким образом он делает JavaScript более стройным и про-

стым для написания, но также провоцирует техническую невалидность страниц. Так как для передачи пар «имя/значение» скриптом гораздо проще использовать `data-shields=5`, HTML5 вводит и стандартизует эту полезную, приближенную к реальности практику и предоставляет нам простой и стандартизованный API для доступа и манипулирования этими пользовательскими атрибутами.

Когда атрибуты `data-*` будут полностью поддерживаться браузерами, JavaScript сможет получать доступ к свойствам при помощи `element.dataset.foo` (где атрибут `data-foo` содержит значение).

В настоящее время такое поведение поддерживается во всех браузерах, за исключением Internet Explorer, но и в этом случае вам может помочь плагин со страницы <http://gist.github.com/362081>.

В ином случае скрипты могут получать доступ к значениям при помощи методов `get/setAttribute`. Преимущество свойства `dataset` по отношению к `setAttribute` состоит в том, что ему намного проще присвоить числовые значения. Например, вам необходимо извлечь все значения, хранящиеся в атрибутах `data-*`. Код, написанный с использованием встроенной функциональности, получается простым и понятным (данний пример не оптимизирован):

```
var values = [];
for (var key in element.dataset) {
    values.push(element.dataset[key]);
}
```

Однако современный вариант кода, хотя и дает такой же результат, не так понятен и доступен неопытным разработчикам и людям, плохо знакомым с JavaScript и DOM:

```
var attributes = el.attributes,
    values = [];
for (var i = 0; i < attributes.length; i++) {
    if (attributes[i].name.indexOf('data-') === 0) {
        values.push(attributes[i].nodeValue);
    }
}
```

ПРИМЕЧАНИЕ

Пользовательские атрибуты данных предназначены только для передачи информации внутренним сценариям сайта в случаях, когда нет никаких других подходящих атрибутов или элементов. Спецификация утверждает: «Эти атрибуты не предназначены для использования приложениями, независимыми от сайта, на котором используются атрибуты». Таким образом, данные атрибуты не могут передавать информацию программам-обходчикам или парсерам сторонних производителей. Это работа для микроформатов, микроданных и RDFa.

В браузерах, где это реализовано полностью, установка атрибута `dataset` приводит к автоматической установке атрибута `content` для элемента, что дает возможность быстрее и удобнее устанавливать пользовательские данные. Таким образом, вместо того чтобы применять

```
element.setAttribute('data-author', 'Remy and Bruce');
```

можно просто-напросто выполнить

```
element.dataset.author = 'Remy and Bruce';
```

Такой синтаксис означает автоматическую установку атрибута в узле DOM и настройку свойства `dataset.name`.

DRAGGABLE

Атрибут `draggable` указывает на то, что элемент можно перетаскивать с помощью drag-and-drop API (см. главу 8).

HIDDEN

Этот атрибут аналогичен `aria-hidden`: он сообщает браузеру о том, что контент этого элемента ни в коем случае не должен отображаться. Это позволяет скрывать контент, но все время держать его «наготове». К примеру, как-нибудь потом можно с помощью JavaScript удалить этот атрибут, и элемент «вернется в реальность».

Я процитирую спецификацию (чтобы не заниматься ее перефразированием): «Атрибут `hidden` не должен использоваться для контента, который может быть отображен в другом представлении. Например, некорректно использовать `hidden` для того, чтобы скрыть панели в диалоговом окне с вкладками, поскольку интерфейс с вкладками — это уже избыточное представление, так как все элементы управления формой можно уместить на одной большой странице с ползунком. Точно так же неправильно делать элемент скрытым только от одного представления: если что-то скрыто, оно должно быть скрыто во всех представлениях, включая, к примеру, программы экранного доступа».

Даже если вы знаете, что позже «откроете» контент с помощью какого-нибудь скрипта, к скрытому элементу нужно относиться так, как будто его на самом деле нет. Не добавляйте ссылки, указывающие на скрытый контент, и не привязывайте к нему другие элементы с помощью `aria-describedby` или `aria-labelledby`.

ID

Не думаю, что кому-то из вас действительно требуются пояснения насчет нашего старого доброго знакомого `id`. Однако теперь значение `id` может начинаться с цифры (раньше такая роскошь была доступна только для атрибута `class`). Как говорят ребята на четверть века меня моложе, крутотень!

ITEMSCOPE, ITEMPROP, ITEMTYPE, ITEMREF, ITEMID

Эти атрибуты относятся к спецификации микроданных.

ROLE, ARIA-*

Как вы уже поняли, HTML5 рассматривает WAI-ARIA как разрешенные дополнения к языку — в том смысле, что они успешно проходят валидацию.

SPELLCHECK

Этот булев атрибут заставляет браузер проверять правописание и грамматику элемента — в частности, это относится к `<input>` и `<textarea>`, но и к любому другому элементу, так как атрибут `contenteditable` может быть у любого элемента. Если атрибут `spellcheck` отсутствует, «выбирается состояние, соответствующее поведению элемента по умолчанию — возможно, исходя из состояния `spellcheck` родительского элемента».

TABINDEX (=-1)

В основе атрибута `tabindex` лежит достаточно старый принцип, позволяющий задавать порядок приоритетов при перемещении по странице с помощью клавиатуры (обычно используется клавиша Tab, но в некоторых браузерах — главным образом, Opera — предусмотрены другие сочетания клавиш).

Этот атрибут был достаточно популярен, когда сайты создавались на основе таблиц с высокой степенью вложенности: порядок следования элементов в них зачастую значительно отличался от визуальной компоновки и логической последовательности элементов. Но поскольку никто, за исключением adeptов WYSIWYG, не использовал таблицы с тех пор, как Моцарт отправился охотиться на стегозавров, сейчас это почти не нужно. По умолчанию порядок перемещения определяется порядком элементов в разметке, поэтому правильно построенный и структурированный документ не требует дополнительных указаний, касающихся приоритетности перемещения.

Однако у `tabindex` есть полезный побочный эффект. Обычно с помощью клавиатуры можно перемещаться только к ссылкам, формам и растровым изображениям. Добавление `tabindex` создает такую возможность и для других элементов. Таким образом, применение JavaScript-функции `focus()` позволит перемещать фокус браузера на них. Однако в таком случае эти элементы смогут быть выбраны и с помощью клавиатуры, что иногда нежелательно.

С использованием отрицательного целого значения (общепринятого `tabindex="-1"`) можно переходить к элементу программно, «но не нельзя сделать этого с помощью последовательной навигации».

Этот атрибут полезен в борьбе с ошибкой Internet Explorer, из-за которой пользователи программ экранного доступа в некоторых случаях не могут перейти к элементам вроде заголовков, которые являлись объектами внутренних ссылок страницы, и информация становилась недоступной (см. <http://www.juicystudio.com/article/ie-keyboard-navigation.php>). В HTML 5 значение "-1" было недопустимым для этого атрибута, а сам атрибут был недопустим для всех элементов, кроме полей форм и ссылок. Но сейчас, когда он работает в браузерах и решает реальную проблему, HTML5 разрешает использовать его везде. Ура!

Атрибуты, удаленные из спецификации

<TABLE BORDER=...>

Я не сомневаюсь, что такие крутые специалисты, как вы, ни за что не станут создавать макет страницы на основе таблиц. Но на случай, если вам придется заняться поддержкой или обновлением подобных древних монстров, сообщу, что у атрибута `border` осталось только два допустимых значения: пустая строка и «1». Они всего лишь сообщают пользовательским агентам, что данная таблица предназначена для разметки. Однако намного лучше делать это с помощью новомодного ARIA-атрибута `role=presentation`, интересная особенность которого заключается в том, что потомки элемента не наследуют значение данного атрибута — поэтому таблицу можно пометить как презентационную, однако в глазах программ чтения экрана и других пользовательских агентов, понимающих ARIA, ее содержимое (например, форма внутри одной из ячеек) презентационным считаться не будет.

Разумеется, для создания красочных рамок вокруг таблиц данных следует использовать CSS. Этот атрибут необходим лишь для обратной совместимости.

<TABLE SUMMARY=...>

В предыдущих версиях HTML у тега `<table>` мог быть атрибут `summary`, который не влиял на визуальное представление, а предназначался исключительно «для пользовательских агентов, которые ориентировались на него при воспроизведении содержимого невизуальным образом, например с помощью речи или шрифта Брайля». С новой спецификацией это не согласуется. Одна из причин, почему это было сделано, заключается в том, что невидимые данные могут рассогласоваться с визуальными данными, которые они описывают. Например, долго не спавший разработчик обновляет данные в таблице, но забывает обновить соответствующую скрытую сводку (атрибут `summary`) — просто потому, что не замечает ее, окидывая взглядом страницу и проверяя, все ли в порядке. Неправильная сводка по табличным данным — это куда хуже, чем отсутствие сводки в принципе.

Звучало такое мнение, что следует потребовать от пользовательских агентов визуально отображать сводку по таблице. К сожалению для WCAG 1, исходные рекомендации по доступности содержимого в обязательном порядке требовали наличия у таблиц атрибута `summary`, поэтому в сети существует огромное множество таблиц с очень полезной сводкой «*This is a layout table*» («Это таблица для поддержки макета»). Вряд ли сеть станет лучше и привлекательнее, если подобный текст везде окажется навиду.

Мне кажется, что если структура веб-сайта настолько сложна, что требует отдельных сводок для помощи пользователям с нарушениями зрения, то и всем остальным пользователям, даже могущим самостоятельно прочитать текст, такая информация тоже не помешает. Поэтому спецификация предлагает несколько вариантов визуального представления подобных сведений: вокруг таблицы, в заголовке таблицы, в элементе `<details>`, рядом с таблицей в том же `<figure>`, рядом с таблицей в `<figcaption>` или просто в обычном тексте.

Атрибут `longdesc` использовался крайне редко, по большей части с изображениями, которые ссылались на отдельную страницу, где приводилось детальное описание данного изображения. Из HTML5 его удалили — разработчики его почти не применяли, а те, кто все же использовал, очень часто делали это неправильно. Тем не менее его считают весьма полезным пользователи программ чтения экрана, 60% которых утверждают, что данный атрибут «достаточно» или «очень» полезен (<http://webaim.org/projects/screenreadersurvey3/#longdesc>). Не существует сравнимого метода, позволяющего предоставить такую же форму расширенного описания изображения, поэтому для выполнения этой задачи вам придется использовать другие механизмы (например, элемент `<details>`).

СВОЙСТВА, НЕ РАССМАТРИВАЕМЫЕ В ЭТОЙ КНИГЕ

Для полноты картины я расскажу о нескольких самых интересных свойствах HTML5, которые по причинам ограниченного объема или недостаточной реализации больше не будут обсуждаться в этой книге.

<EMBED>

Элемент `<embed>` широко известен и используется уже многие годы, но он всегда был «вне закона», потому что никогда не проходил валидацию. Однако так же, как и другой разбойник, Робин Гуд, он имел очень сильную поддержку, так как выполнял полезную функцию: это единственный способ добиться стабильной работы плагинов (например, Flash) во всех браузерах, что и объясняет его невероятно широкое распространение (см. статистику за 2008 год на <http://dev.opera.com/articles/view/mama-plug-ins/>). Поэтому нет никаких причин запрещать его валидацию. HTML5 решил облагородить эту «коровью тропу» и наконец включил `<embed>` в официальную спецификацию языка.

Но погодите-ка секундочку. Разве HTML5 не должен заменить чем-то все эти технологии, основанные на плагинах? Вопреки сенсационным заголовкам авторства некоторых журналистов, HTML5 не сможет волшебным образом за одну ночь избавить мир от плагинов, поэтому теперь мы можем смело встраивать их в код HTML5, не навлекая на себя гнев валидатора.

<KEYGEN>

Этот элемент, который уже обладает хорошей браузерной поддержкой (за исключением того самого большого слона IE), используется в тех ситуациях, когда форма должна отослать открытый ключ. Если вы не знаете, что такое шифрование с открытым ключом, посмотрите http://en.wikipedia.org/wiki/Public_key.

Если вы все еще ничего не понимаете, вам не стоит использовать этот элемент.

<MENU>, <COMMAND>

Это потрясающие элементы: они позволяют создавать для вашего приложения панели инструментов и контекстные меню со значками и соответствующими командами, при активации которых выполняются скрипты. Они не сравнятся даже с целой охапкой стихов Лу Рида! Но так как ни один браузер их еще не поддерживает, мы не будем о них больше говорить.

<STYLE SCOPED>

Атрибут `scoped`, примененный к элементу `<style>`, заставляет браузер применить стили к элементу, внутри которого расположен `<style scoped>`, и его потомкам. Таким образом, его можно обнаружить и у элементов внутри `<body>` документа, а не только в `<head>`, за пределы которого элемент `style` раньше выходить не смел. Это позволяет локализовать стилевое оформление внутри HTML-документа. Так, например, элемент `<article>`, содержащий блок `<style scoped>`, сохранит свое оформление при синдинкации.

Однако ни один браузер его пока что не поддерживает.

ЗАКЛЮЧЕНИЕ

Ух, интересная поездка, не так ли? Вы увидели много новых структур, новых элементов и узнали об изменениях в старых элементах. Если вы внимательно изучали наши примеры разметки, вы также познакомились с любимым оружием фей, так что если вы гоблин или орк, будьте осторожны.

HTML5 позволяет размечать обычные структуры с помощью узкоспециализированных элементов, а не пустых элементов `<div>` и ``. Последние остаются важной частью языка, но, как и в HTML 5, использовать их следует только тогда, когда не удается найти более подходящих элементов, которых у вас теперь целый арсенал. Вы также узнали, что у некоторых элементов есть встроенные роли, которые облегчают работу вспомогательных технологий. Однако пока мы находимся на переходном этапе и браузеры (и, что важнее, программы экранного доступа) обладают недостаточной поддержкой этих встроенных ролей, вы можете (и это законно и допустимо) добавлять дополнительную ARIA-информацию.

Возможно, все это достаточно трудно, но поверьте мне на слово: по мере того как вы будете использовать эти структуры, вы станете понимать их все лучше и лучше. Единственный способ познакомиться с ними — начать их использовать, так что включайтесь в работу!

Глава 3

ФОРМЫ

Брюс Лоусон



Одна из проблем с формами в HTML 5 состоит в том, что поля формы — это всего лишь пустые ячейки для заполнения, не более того. На сервере, конечно же, требуется валидация данных, но для создания осмысленного интерфейса приходится дублировать эту проверку в браузере с помощью JavaScript. Учитывая, что формы используются практически на каждой веб-странице — формы поиска, регистрации, добавления комментария и т. д., — было бы здорово, если бы в браузеры была включена валидация тех типов данных, которые чаще всего вводятся пользователем.

Как вы уже догадались, формы HTML5 позволяют делать именно это.

Мы ❤ HTML, И ТЕПЕРЬ ОН ТОЖЕ НАС ❤

HTML5 позволяет создавать формы гораздо быстрее. Кроме того, в HTML5 появился ряд новых приятных возможностей — например, к уже существующим типам HTTP-запросов, которые могут отправлять формы (`get` и `post`), было добавлено два новых: `update` и `delete`. Однако самое потрясающее обновление, которое оценят не только разработчики, но и их боссы и клиенты, заключается в том, что теперь жизнь каждого из них значительно упростилась — новые типы ввода данных поддерживают особые пользовательские интерфейсы и, самое главное, обладают встроенными возможностями сообщений об ошибках.

Со временем проверка для основных типов данных с помощью JavaScript станет ненужной, однако сейчас не стоит посыпать ее нафталином — JavaScript будет необходим до наступления светлого будущего, когда у каждого будет HTML5-браузер (или ваш начальник решит, что пользователям древних браузеров придется ограничиться проверкой форм только со стороны сервера). В главе 12 мы познакомим вас с методологией под названием «заполнение» (`polyfill`), которая гарантирует, что старые браузеры (и только старые браузеры) получат помощь от JavaScript, в то время как вы будете создавать современный код, согласующийся со стандартом.

ПРИМЕЧАНИЕ

Эти усовершенствования форм пока что реализованы далеко не везде. Наиболее широко их поддержка внедрена в Opera, на втором месте браузеры WebKit и Firefox. На момент написания этой главы кое-что поддерживается даже в Internet Explorer 10 Platform Preview 2. Золотые времена настают!

Новые типы ввода данных

Новые поля форм были основой спецификации, которая превратилась в HTML5, и именно они представляют собой воплощение идеи обратно совместимого расширения. Расширения в данном случае — это в основном новые значения атрибута `type` элемента `input`. В HTML 5 говорится, что браузеры по умолчанию должны использовать `<input type=text>`, если вы не указали другое или неизвестное значение атрибута `type`. Таким образом, устаревшие версии, не распознавающие новые расширения, обратятся к значению по умолчанию и предложат пользователю обычное текстовое поле ввода. Эту ситуацию можно распознать

с помощью сценария и по необходимости заполнение, для того чтобы старые браузеры имитировали новое поведение.

Спецификация не накладывает никаких ограничений на то, как браузеры должны оформлять новые типы ввода данных, сообщать об ошибках и т. д. Различные браузеры и устройства будут использовать различные пользовательские интерфейсы; сравните, к примеру, как выглядит поле выбора в Safari для ПК и iPhone (рис. 3.1).

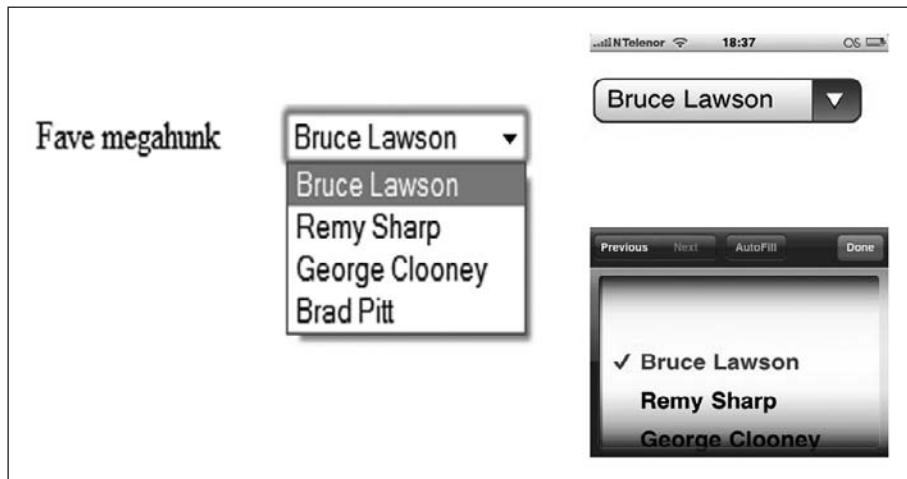


Рис. 3.1. Одно и то же поле выбора в Safari/Windows (слева) и Safari/iPhone

Точно так же не существует единых правил относительно того, как браузер будет сообщать об ошибках. Рисунок 3.2 иллюстрирует ошибку «необходимо заполнить текстовое поле перед отправкой данных» в Opera, Firefox и Google Chrome. Кроме того, вы видите ту же ошибку в локализованной японской версии Opera. Так как сообщения являются частью браузера, они автоматически локализуются, что означает меньше работы для разработчика и лучшее впечатление от работы с браузером у пользователя.

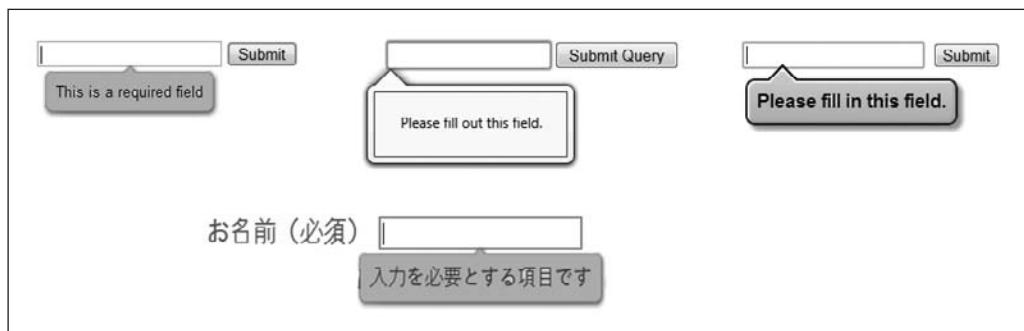


Рис. 3.2. Автоматически сгенерированные сообщения об ошибке в Opera, Firefox, Chrome и Japanese Opera (внизу)

ТИП ВВОДА ДАННЫХ EMAIL

Строка `<input type=email>` сообщает браузеру, что форма не должна быть отправлена, пока пользователь не ввел нечто, похожее на правильный адрес электронной почты, то есть браузер не проверяет, существует ли такой адрес, а только определяет правильность его формата. Пользователь может отправить форму с незаполненным полем, только если к этому полю не добавлен атрибут `required` (это верно для всех типов ввода данных).

Атрибут `multiple` означает, что в качестве значения поля может быть указан список из допустимых электронных адресов, разделенных запятыми. Это, конечно, не значит, что пользователь должен вводить эти адреса вручную; браузер может открывать специальное окно со списком контактов пользователя из почтового клиента (из памяти телефона), где нужные адреса можно отметить флагками, а затем самостоятельно незаметно от пользователя конструировать список.

Современные браузеры пока что не такие услужливые, но поскольку данный тип однозначно определен и легко считывается и распознается компьютером, браузеры теперь понимают, чего добивается разработчик, и могут, вероятно, предоставить более тесно связанный с контекстом пользовательский интерфейс. Например, экспериментальное дополнение Firefox Contacts (<http://mozilla-labs.com/blog/2010/03/contacts-in-the-browser>) формирует список контактов из различных источников, чтобы при заполнении поля `<input type=email>` пользователь мог выбрать в нем нужный адрес. Он также делает эту информацию доступной для скриптов сайта с помощью рабочей версии W3C Contacts API (<http://www.w3.org/2009/dap/contacts/>).

ТИП ВВОДА ДАННЫХ URL

Строка `<input type=url>` заставляет браузер проверять, что пользователь ввел правильный URL-адрес. Браузер может помочь пользователю это сделать — Opera, например, автоматически добавляет `http://` в начало URL в случаях, когда пользователь не указывает протокол (то есть не добавляет в начале `http://` или `ftp://` или какое-то еще общепринятое обозначение протокола). (URL не обязательно должен относиться к веб-узлу; страница может быть, например, сетевым HTML-редактором, в котором пользователям иногда требуется псевдопротокол `tel:`).

ТИП ВВОДА ДАННЫХ DATE

Тип `date` — один из моих любимых. Все мы видели веб-страницы, в которых пользователю необходимо ввести дату полета, концерта и т. д. Поскольку не очень понятно, как это делать (в формате DDMM-YYYY, или MM-DD-YYYY, или DD-MMM-YY?), разработчики с помощью JavaScript создают виджеты для выбора даты, которые от сайта к сайту значительно различаются внешним видом, удобством использования и доступностью.

`<input type=date>` решает эту проблему, открывая доступ к встроенному в браузер виджету выбора даты. Opera, к примеру, открывает виджет с календарем (рис. 3.3).

На смартфонах BlackBerry в BlackBerry Device Software версии 5.0 в качестве элемента управления для ввода даты использовался тот же Java-компонент, что и в приложении «Календарь» BlackBerry (хотя он не встроен в приложение «Календарь») (рис. 3.4).



Рис. 3.3. Opera 10.5 открывает виджет с календарем



Рис. 3.4. Так визуализируется <input type=date> в браузере BlackBerry

Безусловно, это только начало. Вполне возможно, что в будущем браузер сможет не только выводить блестящий новенький виджет выбора даты, но и делать что-то намного более интеллектуальное, например вызывать встроенное приложение «Календарь», чтобы вы могли выбрать дату исходя из вашего расписания. Главное — теперь браузер может понимать, что вы хотите ввести. Раньше виджеты выбора даты были с точки зрения браузера не более чем элементами `<div>`, `` и ссылками с большим количеством JavaScript-кода, отвечающего за поведение. Теперь браузер знает, что вы собираетесь ввести дату и время, и может предложить вам усовершенствованные элементы управления и обеспечить интеграцию с другими сведениями о дате/времени.

ТИП ВВОДА ДАННЫХ TIME

`<input type=time>` позволяет вводить время по 24-часовой шкале и проверяет правильность введенных данных. И снова выбор пользовательского интерфейса остается за браузером — это может быть просто ввод чисел с выводом ошибки в случае, если количество часов превышает 24 или количество минут превышает 59, или же он может быть разработан более детально: например, в виде циферблата, стрелки которого можно перетаскивать курсором мыши. Интерфейс может также учитывать смещение часового пояса.

ТИП ВВОДА ДАННЫХ DATETIME

Типы `date` и `time`, с которыми мы только что познакомились, можно объединять с помощью `<input type=datetime>` для проверки точной даты и времени. Поле ввода местного времени и даты работает так же, как и `datetime`, за исключением того, что пользователь не может добавить (или изменить) смещение часового пояса.

ТИП ВВОДА ДАННЫХ MONTH

`<input type=month>` позволяет вводить месяц и выполняет проверку введенных данных. Хотя внутри эта информация хранится в виде числа от 1 до 12, браузер может предложить пользователю выбрать месяц по названию. Вы могли бы, например, использовать поле выбора из 12 вариантов (январь–декабрь), но такой вариант не универсален с точки зрения выбора языка. Если вы будете использовать тип ввода данных HTML5 `month`, французская

локализация браузера может, к примеру, в раскрывающемся списке заменить «Январь» на «Janvier». Такие тонкости — в большей степени забота браузера и в меньшей степени — вида. И это абсолютно нормально.

ТИП ВВОДА ДАННЫХ WEEK

`<input type=week>` позволяет вводить номер недели и выполняет проверку введенных данных. Хотя это может выглядеть как простое поле, позволяющее пользователю вводить число, на практике все оказывается сложнее: в некоторых годах 53 недели. Поэтому для седьмой недели 2010 года используется формат 2010-W07.

Opera позволяет открыть пользовательский интерфейс выбора даты и помещает в поле ввода номер недели, соответствующий любой выбранной дате, а не дате в формате YYYY-MM-DD (рис. 3.5).

Рис. 3.5. `<input type=week>` в Opera

ТИП ВВОДА ДАННЫХ NUMBER

Как это ни удивительно, но тип ввода данных `number` выдает ошибку, если пользователь не ввел численные символы. Он не предназначен для ввода номеров телефона, так как их часто указывают с пробелами, скобками, значком плюса, дефисами и т. п.; используйте для телефонов `<input type=tel>`.

Он прекрасно работает с атрибутами `min`, `max` и `step`. В Opera и Chrome он выглядит как счетчик, значения которого можно ограничить сверху и снизу, а шаг задается с помощью атрибута `step`; при этом пользователь может ввести число с клавиатуры (рис. 3.6). Стрелки счетчика находятся за пределами поля ввода в Opera, но внутри этого поля в Chrome. В спецификации ничего не говорится относительно того, как должен выглядеть пользовательский интерфейс этих новых элементов управления.

Рис. 3.6. Способ визуализации `<input type=number>` в Opera (слева) и Chrome (справа)

В настольных браузерах, нелагающих для числовых данных особый пользовательский интерфейс, возникает неприятная ситуация. В настоящее время ввод буквенных симво-

лов в поля `type=number` в Opera и Chrome не приводит к ошибке валидации. По-моему, такое поведение не согласуется с логикой. Подобное случается из-за того, что браузер не занимается проверкой введенных данных — он даже не заменяет текущее значение поля введенными пользователем нецифровыми символами. К сожалению, пользовательский интерфейс предполагает, что вы ввели буквы и они были одобрены.

ТИП ВВОДА ДАННЫХ RANGE

`<input type=range>` отображается в виде ползунка. На рис. 3.7 показано, как это выглядит в Google Chrome.



Рис. 3.7. Визуализация `<input type=range>` в Chrome

Раньше ползунки приходилось имитировать и для этого требовалось перехватывать входные данные и использовать JavaScript и изображения для стрелок. Так как они не были встроены в браузеры, обеспечение удобного управления с клавиатуры требовало особой аккуратности и дополнительного кода. Теперь, когда ползунки встроены в HTML, с разработчика снимается ответственность, и за счет этого уменьшается объем кода и повышается доступность для пользователей клавиатуры.

Чтобы подробнее узнать об этом, посмотрите пример в разделе «Как собрать все это вместе» далее в этой главе. Данный тип прекрасно работает с атрибутами `min`, `max` и `step` (см. далее).

ТИП ВВОДА ДАННЫХ SEARCH

Этому типу ввода данных требуется поисковый термин. В Safari есть уникальный атрибут, не предусмотренный в спецификации, который добавляет историю недавних запросов (с помощью `results=n`). Разница между типами ввода данных `search` и `text` чисто стилистическая; в Safari на Mac `search` оформляется тем стилем, который используется по умолчанию в операционной системе (с закругленными углами), что тем не менее можно изменить с помощью специального CSS-кода (спасибо Уилфреду Нэсу).

```
input[type="search"] {-webkit-appearance: textfield;}
```

ТИП ВВОДА ДАННЫХ TEL

Тип `tel` используется для ввода телефонного номера. Особой проверки не производится; более того, не требуется вводить только цифры — многие номера часто пишутся с использованием других знаков, например +44 (0) 208 123 1234.

Поскольку мобильные телефоны «знают» свой собственный номер, мы надеемся, что в большинстве мобильных телефонов появится функция автозаполнения таких полей. Пока это нигде не реализовано, но iPhone умеет открывать экран ввода телефонного номера (рис. 3.8).



Рис. 3.8. Клавиатура iPhone для заполнения полей типа `<input type=tel>`

ТИП ВВОДА ДАННЫХ COLOR

`<input type=color>` позволяет пользователю ввести значение цвета, выбрав нужный вариант в палитре. Пока это реализовано только в BlackBerry (рис. 3.9) и Opera.



Рис. 3.9. Визуализация `<input type=color>` на устройстве BlackBerry

НЕ ЗАБЫВАЙТЕ ОБ АТРИБУТЕ NAME!

Конечно, теперь валидация на клиентской стороне встроена в браузер, но это не означает, что можно окончательно расслабиться. Не забывайте присваивать полям ввода (и группам переключателей) уникальные значения в атрибуте name, так как именно они позволяют получать введенные данные, переданные формой, на серверной стороне. Старые версии Opera требуют этого еще до выполнения валидации HTML5, и спецификация говорит ровно о том же.

Так как старые версии IE иногда путают id и name при использовании `getElementById`, мы рекомендуем задавать одно и то же уникальное значение для атрибутов id и name каждого поля. Это делает формы более доступными:

```
<label for=f-email>Email address</label>
<input id=f-email name=f-email type=email>
```

Новые атрибуты

Помимо новых типов ввода данных, в HTML5 определен ряд новых атрибутов для элемента `<input>`, отвечающих за его поведение и ограничения: `autocomplete`, `min`, `max`, `multiple`, `pattern` и `step`. Есть еще один атрибут — `list`; он присоединяется к новому элементу, позволяя использовать новый метод ввода данных.

Атрибут `list` и `<datalist>`

Сочетание элемента `<input>` с атрибутом `list` и элементом `<datalist>` дает комбинированное поле — комбинацию раскрывающегося списка и текстового поля из одной строки. Оно позволяет пользователям вводить свой собственный текст, если их не устраивают предложенные в списке варианты.

Список создается внутри нового элемента `<datalist>`, `id` которого указывается в качестве значения атрибута `list`.

```
<input id=form-person-title type=text list=mylist>
  <datalist id=mylist>
    <option label=Mr value=Mr>
    <option label=Mrs value=Mrs>
    <option label=Prof value="Mad Professor">
  </datalist>
```

Сам `<datalist>` не отображается, но появляется в виде значений поля, похожего на поле выбора.

В предыдущем примере `type=text` делает возможным ввод произвольного текста; но `<datalist>` можно использовать и с другими типами ввода данных, например `url` и `email`.

Многие спрашивают, почему пара `<input>/<datalist>` не объединена в один элемент, как, например, `<select>`. На самом деле это связано с обратной совместимостью: в устаревших браузерах пара `<input>/<datalist>` превращается в `<input type=text>`, благодаря чему пользователь может хотя бы что-то ввести, а вы уже можете дополнить реализацию с помощью JavaScript.

Джереми Кейт приводит отличный пример подобной обратной совместимости в статье по адресу <http://adactio.com/journal/4272/> (воспроизвожу ее с его разрешения):

```
<label for="source">How did you hear about us?</label>
<datalist id="sources">
  <select name="source">
    <option>please choose...</option>
    <option value="Television">Television</option>
    <option value="Radio">Radio</option>
    <option value="Newspaper">Newspaper</option>
    <option>Other</option>
  </select>
  If other, please specify:
</datalist>
<input id="source" name="source" list="sources">
```

Обратите внимание, что мы обрамили элементы `<option>` дополнительным тегом `<select>`, благодаря чему визуализация содержимого `datalist` имитирует разметку традиционных элементов выбора `dropdown` (раскрывающийся список). Браузеры, понимающие элемент `<datalist>`, проигнорируют все лишнее, сосредоточившись на элементах `<option>` — для них вложенный `<select>` попросту невидим. Текст «*If other, please specify*» также игнорируется. С другой стороны, не поддерживающие данную функциональность браузеры не увидят элемент `<datalist>` и покажут то, что они воспринимают как стандартный элемент `<select>`. Также они отобразят текст «*If other...*» и поле ввода, с которым связан `datalist`. Другими словами, браузеры, поддерживающие `<datalist>`, считают каждый `<option>` частью списка данных (`datalist`) и кроме них ничего не видят. Браузеры, не поддерживающие `<datalist>`, считают все `<option>` частью `<select>` и видят дополнительный текст «*If other, please specify*»; элемент ввода, связанный со списком данных через атрибут `list`, превращается в простое текстовое поле ввода (рис. 3.10).

Рис. 3.10. Визуализация `<datalist>` в Opera (вверху) и упрощенный вариант в Safari (внизу)

Это отличный шаблон, который наверняка станет частью вашего ежедневного арсенала инструментов для кодирования форм — если только Рабочая группа не примет решение отобрать его у нас (см. примечание)!

ПРИМЕЧАНИЕ

Рабочая группа (Working Group) рассматривает вариант устранения данного метода плавного ухудшения характеристик через разметку, приводя не слишком убедительные обоснования — мол, веб-разработчики не так часто его применяют, к тому же его трудно описать в спецификации и реализовать (что делать с элементами `<script>` внутри `<datalist>`?).

Если такое произойдет, мы обязательно упомянем об этом на <http://www.introducinghtml.com>, а вам придется полагаться на помощь сценариев, добавляющих поддержку `<datalist>` в старые браузеры (через плавное ухудшение характеристик). Это будет ужасно.

Как и я сам, это не самое симпатичное в мире создание, однако оно работает (правда, в IE10 Platform Preview 2 вы в результате получаете комбинацию из `select + input`) и хорошо демонстрирует плавный перенос новых возможностей на ограниченные старые платформы.

АТРИБУТ autofocus

Булев атрибут `autofocus` — способ задания установки фокуса на определенном элементе управления формы во время загрузки страницы. Раньше разработчику требовалось писать для этого JavaScript-код, используя `control.focus()`. Новый способ позволяет делать достаточно разумные вещи, например не устанавливать фокус на элементе, если пользователь в это время вводит данные в другом месте (проблема, типичная для старых JavaScript-скриптов, реализующих эту возможность).

На странице должно быть только одно такое поле ввода. Помните об удобстве использования и применяйте этот атрибут с осторожностью. Мы рекомендуем добавлять его только там, где поле формы служит для выполнения основной функции страницы — как, например, форма поиска.

АТРИБУТ placeholder

Достаточно популярным у разработчиков является такой прием: изначально в поле ввода добавляется текст-подсказка, которая исчезает при установке фокуса на этом поле, а затем (при установке фокуса на другом элементе) снова появляется. Раньше для этого требовался JavaScript. Но теперь такое поведение можно задать с помощью атрибута `placeholder`. В спецификации говорится: «Для добавления более подробной подсказки или совета лучше использовать атрибут `title`».

Обычно для оформления подсказок используется более светлый оттенок, чем для обычного текста, который пользователь вводит в данном поле. Для стилизации можно использовать `::-webkit-input-placeholder`, `:-moz-placeholder` и `-ms-input-placeholder`. Чтобы обеспечить совместимость с будущими версиями браузеров, также добавляйте `-o-input-placeholder`. Это экспериментальный формат, а не часть официальной спецификации CSS.

Важно также помнить, что атрибут `placeholder` не заменяет элемент `<label>`.

АТРИБУТ required

Новый атрибут `required` можно использовать для `<textarea>` и большинства полей ввода (кроме случаев, когда атрибут `type` принимает значения `hidden`, `image` или задает кнопочный тип, такой как `submit`). Браузер не разрешит пользователю отправить форму, если обязательные поля будут пустыми, и вернет ошибку.

Также мы рекомендуем добавлять к таким полям ARIA-атрибут `aria-required`, который будет полезен для вспомогательных технологий (ARIA подробно обсуждалась в главе 2).

АТРИБУТ multiple

Тип `<input type=file>` не является новым в HTML5, но теперь благодаря атрибуту `multiple` пользователь может загружать несколько файлов.

`<input type=file multiple>`

В HTML 5 так было делать нельзя, поэтому разработчикам для достижения того же эффекта приходилось прибегать к помощи Java-апплетов и Flash.

Этот атрибут можно добавлять и к другим типам ввода данных: например, `<input type=email multiple>` позволяет вводить несколько адресов электронной почты.

АТРИБУТ PATTERN

Некоторые типы ввода данных, о которых говорилось выше, — `email`, `number`, `url` и т. д. — на самом деле являются «встроенными» регулярными выражениями, так как браузер просто проверяет, что введенные данные выглядят так, как должны.

А что если вы хотите использовать другой шаблон? Атрибут `pattern` позволяет вам самим создавать регулярные выражения, которым должны соответствовать вводимые данные. Так что если пользователь должен ввести одну цифру и три заглавных символа алфавита, регулярное выражение будет содержать одно число `[0-9]` и три буквы `[A-Z]{3}`, каждая из которых должна быть в верхнем регистре; тогда для элемента ввода будет использоваться такой код.

```
<input pattern="[0-9][A-Z]{3}" name=part
  - title="A part number is a digit followed by three
  - uppercase letters.">
```

Можно, кстати, добавить `placeholder="1ABC"` или другую короткую подсказку.

В спецификации говорится, что регулярные выражения атрибута `pattern` строятся в соответствии с синтаксисом регулярных выражений в JavaScript за исключением того, что в них подразумевается наличие `(: ?` в начале и `)$` в конце.

Так что если вы привыкли работать с регулярными выражениями, вы, скорее всего, уже знаете, что нужно делать. Если нет, перед вами открывается интереснейший мир регулярных выражений.

В Интернете вы почти всегда сможете найти то регулярное выражение JavaScript, которое вам нужно. Но с регулярными выражениями на самом деле довольно легко работать, если смотреть на это проще.

Например, для поля ввода ZIP-кода в формате 99999 или 99999-9999 (считая, что 9 соответствует любой цифре) можно использовать такой код.

```
<input pattern="[0-9]{5}(\-[0-9]{4})?" title="A zip code in
  - the format of 99999 or 99999-9999">
```

Это регулярное выражение ищет последовательности из пяти цифр с необязательным суффиксом — дефисом и еще одной последовательностью из четырех цифр.

Можно расширить наш шаблон так, чтобы он позволял вводить почтовые коды Соединенного Королевства (используя упрощенный вариант).

```
<input required pattern="[0-9]{5}(\-[0-9]{4})?|[a-zA-Z]
  - {1,2}\d{1,2}\s?\d[a-zA-Z]{1,2}" name=part title="A valid
  - zip code or UK postcode">
```

Теперь наше регулярное выражение стало гораздо более сложным и поэтому проверить его в большой форме на веб-странице будет не очень просто. Но так как здесь используется тот же синтаксис, что и в JavaScript, мы можем выполнить проверку в консоли браузера (например, в Firebug или Opera Dragonfly), используя чистый JavaScript. Приведенный ниже пример — тест, определяющий правильность проверки почтового кода Соединенного

Королевства; в нем используется JavaScript-метод `test`. Обратите внимание на то, что я добавил `^(:?: в начало и)$` в конец тестов (об этом говорится в спецификации HTML5).

```
/^(?:[a-zA-Z]{1,2}\d{1,2}\s?\d[a-zA-Z]{1,2})$/.test
¬ ("bn14 8px")
> true
/^(:?[a-zA-Z]{1,2}\d{1,2}\s?\d[a-zA-Z]{1,2})$/.test
¬ ("bn149 8px")
> false
```

Эти результаты верны, поскольку «`bn149`» не соответствует стандарту почтовых кодов (и, безусловно, недопустим с точки зрения нашего надуманного примера). Наконец, стоит отметить, что атрибут `pattern` восприимчив к регистру, и так как у нас нет возможности переключить его в режим *невосприимчивости*, в этом примере нам приходится явно указывать символы в нижнем и верхнем регистре (отсюда и запись `[a-zA-Z]`).

АТРИБУТ AUTOCOMPLETE

В большинстве браузеров есть свои функции автозаполнения, работу которых позволяет контролировать новый атрибут `autocomplete`. Хотя впервые стандартизирован он именно в спецификации HTML5, сама идея не нова — в IE5 уже существовала подобная нестандартная возможность.

По умолчанию поле ввода наследует состояние автозаполнения формы, к которой оно относится. Для форм автозаполнение по умолчанию включено.

Если атрибут `autocomplete` элемента формы включен, поле может автоматически заполняться.

Я приведу фрагмент спецификации, описывающий состояние `off` (его можно отнести к разряду черного юмора): «Состояние `off` можно использовать либо когда вводимые данные строго конфиденциальны (например, если это код активации ядерного оружия), либо когда эти данные никогда не будут повторно использоваться (например, разовый ключ для входа в систему банка), и пользователь должен будет каждый раз вводить их явно».

АТРИБУТЫ MIN И MAX

Как мы уже видели, когда говорили о `<input type=number>`, эти атрибуты позволяют ограничивать диапазон вводимых значений; форма не будет отправлена, если значение меньше `min` или больше `max`. Их можно использовать и с другими типами ввода данных — например, поле `<input type=date min=2010-01-01 max=2010-12-31>` примет только дату, относящуюся к 2010 году. Часто бывает полезно делать так, чтобы при создании HTML-кода сервер присваивал атрибуту `min` дату сегодняшнего дня, и тогда вводить можно было бы только последующие даты (например, на сайте бронирования авиабилетов), или же чтобы сервер присваивал это значение атрибуту `max` (например, для поля ввода даты рождения).

АТРИБУТ STEP

Атрибут `step` отвечает за степень подробности вводимых данных. Если вы хотите, чтобы пользователь ввел число в процентах (между 0 и 100) с точностью до 5, можно использовать такой код:

```
<input type=number min=0 max=100 step=5>
```

Тогда счетчик будет увеличиваться с шагом 5.

В примере с элементом управления для ввода времени вы можете использовать `step=any`. Это позволит пользователю выбрать любое время с любой точностью (например, с точностью до тысячных долей секунды или более); тогда как обычно такие элементы управления требуют ввода времени с точностью до одной минуты.

```
<input name=favtime type=time step=any>
```

АТРИБУТ FORM

По традиции элементы управления формы должны располагаться внутри элемента `<form>`. Если бы по какой-либо причине — связанной с дизайном, оформлением или личными вкусами — разработчики решили поместить форму в одно, а некоторые связанные с ней элементы в другое место страницы, им пришлось бы (в худшем случае) обрамлять элемент `<form>` всю страницу.

Однако — дивный новый мир! — в HTML5 некоторые элементы, которые ранее обязательны должны были находиться внутри элемента `form` (`<button>`, `<fieldset>`, `<input>`, `<label>`, `<select>`, `<textarea>` и `<object>`), а также новые элементы, такие как `<keygen>`, `<meter>`, `<output>` и `<progress>`), теперь могут располагаться в любом месте страницы, а с формой их нужно связывать с помощью атрибута `form`, указывающего на `id` их формы-владельца.

Рассмотрим следующий пример:

```
<form id=foo>
<input type="text">
...
</form>
<textarea form=foo></textarea>
```

Элемент `<input>` принадлежит форме `foo`, так как он находится внутри нее, и у него нет атрибута `form`, определяющего владельца. Элемент `<textarea>` находится за пределом формы, также принадлежит ей, так как атрибут `form` указывает на `id` формы-владельца. Это обеспечивает большую гибкость стилизации, позволяя создавать элементы, оторванные от родительских форм как визуально, так и стилистически.

Атрибут `form` поддерживается в Opera, Firefox и Chrome. Так как кодовая база у Safari такая же, как и у Chrome, в этом браузере данный атрибут также должен получить поддержку в ближайшем будущем. В настоящее время IE атрибут `form` не поддерживает, и на момент написания этой главы о добавлении его поддержки в версию IE10 не сообщается.

ЭЛЕМЕНТЫ <PROGRESS>, <METER>

Элемент `<progress>` применяется для представления «индикатора выполнения», сообщающего, какой процент задачи (например, загрузки файла) уже выполнен. У него два атрибута: `max`, определяющий общий объем работы, и `value`, определяющий размер выполненной доли. Единицы измерения произвольные, их указывать не нужно:

```
<progress value=5 max=20>5</progress>
```

В поддерживающих данную функциональность браузерах он заменяется изображением индикатора хода выполнения (рис. 3.11).



Рис. 3.11. Визуализация элемента `<progress>` в Chrome (слева) и в Opera (справа).

В Chrome с изображением связана небольшая анимация; в Opera оно статичное

Элемент `<meter>` очень похож на `<progress>` (и многие справедливо задаются вопросом, действительно ли существует необходимость в двух отдельных элементах). В спецификации утверждается, что `<meter>` «представляет скалярную меру в пределах известного диапазона или дробную величину; например, объем занятого пространства на диске, релевантность результата запроса или долю населения, имеющего право голоса, которая проголосовала за определенного кандидата».

Он принимает следующие атрибуты, значением которых может быть число с плавающей запятой:

- `value` — «измеренное» значение, отображаемое счетчиком;
- `min` — нижняя граница диапазона счетчика;
- `low` — точка, обозначающая верхнюю границу «нижнего» сегмента счетчика;
- `high` — точка, обозначающая нижнюю границу «верхнего» сегмента счетчика;
- `max` — верхняя граница диапазона счетчика;
- `optimum` — точка, обозначающая «оптимальную» позицию счетчика.

Обязательный атрибут только один, `value`. Если атрибуты `min` и `max` отсутствуют, то подразумевается диапазон от 0 до 1.

Opera и Chrome (два браузера, поддерживающие `<progress>` и `<meter>` на момент написания этой главы) окрашивают счетчик иным цветом, если значение `value` находится в диапазоне между `low` и `high`.

В старых браузерах в качестве альтернативного варианта отображается текстовое содержимое элементов.

КАК СОБРАТЬ ВСЕ ЭТО ВМЕСТЕ

Достаточно сложно разобраться, к каким типам ввода данных относятся те или иные атрибуты, особенно когда вы знакомитесь со всеми ними одновременно (как в данном случае). Но когда вы начнете их использовать, все встанет на свои места. К примеру, ясно, что для `<textarea>` нельзя использовать `min` и `max`, потому что это бессмысленно, а `required` — можно.

ФОРМА КОММЕНТАРИЯ БЛОГА

Давайте рассмотрим классический пример формы, с которой знаком каждый. Практически во всех блогах есть раздел комментариев, в котором есть поля для имени пользователя, оставившего комментарий (обязательное), адреса его электронной почты (обязательное), URL (не обязательное) и самого комментария (обязательное). Если бы мы создавали валидацию вручную, нам потребовалось бы написать много JavaScript-кода.

Впрочем, в HTML5 требуется всего лишь несколько новых типов форм. Кроме того, мы добавим кнопку отправки формы — в настоящее время в Opera процесс валидации полей происходит только после отправки формы.

```
<form>
  <label for=form-name>Name</label>
  <input name=form-name id=form-name type=text required>
  <label for=form-email>Email</label>
  <input name=form-email id=form-email type=email required>
  <label for=form-url>URL</label>
  <input name=form-url id=form-url type=url>
  <label for=form-comment>Comment</label>
  <textarea name=form-comment id=form-comment required>
  </textarea>
  <input type=submit>
</form>
```

Раз, два — и готово! У нас есть превосходная форма для отправки комментария, проверяющая введенные данные, и никакого JavaScript!

ПОЛЗУНОК СО СКРИПТОМ ДЛЯ ВЫВОДА

Мы уже познакомились с `<input type=range>` ранее в этой главе. Обратите внимание, что по умолчанию браузеры отображают ползунок, но не выводят для него минимальное, максимальное и текущее значение. Давайте напишем код для следующего случая: пусть нам нужно показать пользователю диапазон допустимых значений в виде ползунка, причем мы хотим добиться автоматического вывода значений верхней и нижней границы и динамического вывода текущего значения ползунка.

Ползунок будет перемещаться от 1 до 11, как и все нормальные элементы управления (будь то гитарный усилитель или что-либо другое), с шагом 1, и поскольку это значение используется по умолчанию, мы можем вообще не добавлять атрибут `step`.

```
<input type=range min=1 max=11 name=tap>
```

Чтобы отобразить значения верхней и нижней границы, будем использовать генерируемый контент (который не работает для ползунков в браузерах WebKit).

```
input[type=range]::before {content: attr(min);}
input[type=range]::after {content: attr(max);}
```

Этот код отобразит нужные значения и создаст для них оформление с помощью CSS. В качестве примера (рис. 3.12) приведем результат работы следующего кода.

```
input[type=range]{width:500px; color:red; font-family:
- cursive; font-size:2em;}
```



Рис. 3.12. Визуализация `<input type=range>` и генерируемые значения `min` и `max` в Opera

Чтобы отобразить текущее значение ползунка, мы будем использовать новый элемент `<output>`.

Элемент `<output>`

Элемент `<output>` предназначен для вывода результатов вычислений (или чего-то другого) с помощью скрипта. Он может принадлежать какой-нибудь форме, и тогда он должен либо располагаться внутри нее, либо ссылаться на нее через атрибут `form`. Для удобства взаимодействия с пользователем, к форме могут быть также привязаны новые элементы `<progress>` и `<meter>`, о которых я также рассказал чуть выше.

Чтобы привязать этот элемент к ползунку, будем использовать имя ползунка (`name=tab`) и событие `oninput`. Когда форма, которой принадлежит `<output>`, получит вводимые данные (это произойдет при перемещении ползунка), мы сможем отобразить значение этих данных.

```
// в этом примере подразумевается, что переменная output содержит
// элемент output из DOM
// а переменная slider содержит элемент input range из DOM
<form oninput="output.value=slider.value">
  <input id=slider type=range min=1 max=5 value=5>
  <output id=output>5</output>
</form>
```

Фактическое содержимое элемента `<output>` (в нашем случае «5») будет отображаться только в качестве начального значения до тех пор, пока ползунок не будет перемещен. В этом простом примере мы напрямую помещаем значение «5» в разметку — это такое же начальное значение, как `type=range`. Можно было бы, конечно, использовать сценарий, выполняющий `onload` и программным образом заполняющий элемент `output` соответствующим введенным значением.

ПРИМЕЧАНИЕ

В первом издании нашей книги мы использовали событие `onforminput`, а не `oninput`, но оно устарело. Интересующиеся историей разметки могут почитать об этом и о многом другом на странице <http://www.useragentman.com/blog/2011/05/10/is-onforminput-deprecated-in-html5-forms-and-why-you-should-i-care-anyways>.

Использование WAI-ARIA для временного обеспечения доступности

Хотя мы говорили о том, что `<input type=range>` снимает с разработчика ответственность за обеспечение доступности, это будет верно только тогда, когда HTML5 будет обладать широкой поддержкой, а вспомогательные технологии будут понимать этот новый тип ввода.

Сейчас, когда мы находимся на переходном этапе, при использовании HTML5-ползунков вам следует также добавлять кое-какую информацию WAI-ARIA (на данном этапе придется дублировать некоторые сведения).

```
<input id=tab
       name=tab
       type=range
       min=1
       max=11
```

```
value=0
aria-valuemin=1
aria-valuemax=11
aria-valuenow=0>
```

ПРИМЕЧАНИЕ

В первом издании этой книги я добавлял атрибут `role=slider`, который сообщал вспомогательным технологиям, каким образом следует сопоставлять данный элемент управления с элементами управления операционной системы. Я ошибался или, по крайней мере, бежал впереди паровоза. В идеальном мире с полной поддержкой браузерами (однажды, дорогой читатель, однажды это произойдет) это работало бы превосходно. Но пока что не стоит явно делать такие объявления; делайте это в коде JavaScript после проверки, поддерживается ли `input type=range` (см. раздел «Обратная совместимость с устаревшими браузерами» ниже). В браузерах, не поддерживающих `type=range`, такое явное объявление добавляет роль ползунка обычному полю для ввода текста, что совершенно неверно и лишь сбьет с толку пользователя программы чтения экрана.

При изменении положения ползунка значение `aria-valuenow` нужно обновить с помощью JavaScript. В таком случае нужно обращаться к событию `change` ползунка; но в нашем примере мы будем просто использовать атрибут `onchange`. К сожалению, для правильного обновления значения `aria-valuenow` мы не можем использовать синтаксис этого свойства — нам придется обновлять атрибут DOM.

```
<input id=“tap”
name=“tap”
type=“range”
min=1
max=11
value=0
aria-valuemin=1
aria-valuemax=11
aria-valuenow=0
onchange=“this.setAttribute(‘aria-valuenow’,
- this.value)”>
```

Такой код будет обновлять значение атрибута `aria-valuenow`; это можно проверить, проанализировав элемент с помощью DOM Inspector.

ОБРАТНАЯ СОВМЕСТИМОСТЬ С УСТАРЕВШИМИ БРАУЗЕРАМИ

Остается главный вопрос: что мы можем сделать для устаревших браузеров? Ответ звучит так: пока не отправляйте на пенсию вашу старую JavaScript-валидацию и модные сборщики данных DHTML; оставьте их в качестве запасного варианта и выполните проверку возможностей.

Как мы видели раньше, встречая неподдерживаемый тип, браузеры прибегают к альтернативному варианту и используют `type=text`. Таким образом, устаревший браузер, встретив

`input type=email`, просто заменит это на `input type=text`. Данное изменение распространяется и на DOM, поэтому, проверив тип элемента ввода, мы можем программным образом определить, поддерживает ли браузер новые крутые элементы, и в противном случае выбрать подходящий обходной путь.

Например, чтобы определить, поддерживается ли `<input type=email>`, вы создаете новый `<input type=email>` с помощью JavaScript, но не добавляете его на страницу. Далее вы просите ваш новый элемент сообщить, какой у него тип. Если в ответ вы получаете «`email`», браузер поддерживает эту возможность — и тогда пусть он делает все сам без JavaScript-валидации. Если вы получаете в ответ «`text`», это значит, что было выбрано значение по умолчанию и эта возможность не поддерживается. Так что в идеале ваш код должен загружать альтернативную библиотеку валидации, используя метод загрузки *по требованию* (*lazy load*).

```
var i = document.createElement("input");
i.setAttribute("type", "email");
return i.type !== "text";
Aтрибуты тоже можно проверять.
return 'autofocus' in document.createElement("input");
```

Итак, что вам это дает? Во-первых, вы создаете удобные в использовании и доступные формы, предоставляя простые механизмы ввода (такие, как сборщики данных) и проверяя пользовательские данные еще до того, как будет отправлен первый запрос на сервер. Во-вторых, вы экономите ресурсы, прибегая к помощи встроенных возможностей браузера (если они уже понимают валидацию на клиентской стороне и новые типы и атрибуты HTML5) или, в случае устаревших браузеров, изящно подтягивая поддержку традиционных библиотек JavaScript.

Методологию и обсуждение добавления такой поддержки в старые браузеры вы найдете в главе 12.

ОФОРМЛЕНИЕ НОВЫХ ПОЛЕЙ ФОРМЫ И СООБЩЕНИЙ О ОШИБКАХ

Каждый раз, когда мы рассказываем о наших новых логичных полях форм на конференциях, кто-то спрашивает, как создавать стили для таких полей, а также для сообщений об ошибках.

Практически ко всем новым элементам управления можно добавлять базовое оформление: шрифты, цвета и т. п. Однако с некоторыми элементами управления современный код CSS совместим плохо. Например, как понять, к чему относится `color` в случае `type=range` и какой стиль меняет `background-color`? На что воздействует `border-radius`? Как поменять цвет «трека», по которому перемещается ползунок?

Естественный выбор разработчика, желающего добавить новые функции CSS для стилизации полей формы, — это модуль с манящим названием CSS Basic User Interface Module (<http://www.w3.org/TR/css3-ui/>). Он существует примерно с 2004 года, и теперь, когда браузеры окончательно освоились с этими возможностями, обновлением спецификации занимается Тантек Челик.

Уже сейчас можно воплотить отдельные полезные идеи — в частности, псевдокласс `:invalid` применяется для стилизации поля формы для случая ввода недопустимого содержимого. Таким образом, пользователь моментально получает ответ от формы, еще даже не нажав на кнопку отправки информации. Следующий код CSS окрашивает границу поля формы с недопустимыми данными в красный цвет, а в случае, когда данные допустимые, — в зеленый.

```
input:invalid {border:2px solid red;}
input:valid { border: 2px solid green; }
```

К сожалению, возникают сразу несколько проблем удобства использования. Например, данные в поле считаются недопустимыми в момент загрузки страницы — просто потому, что там требуются данные, а поле пустое. Следовательно, применяется стиль «`invalid`», что пугает пользователя. Значение в поле `input type=email` считается недопустимым все то время, пока пользователь вводит какие-то символы, не содержащие «@», потому что лишь после появления «@» можно сказать, что он действительно ввел адрес электронной почты.

Великий Патрик Лауке (один из технических редакторов нашей книги) предлагает для смягчения этой проблемы добавлять `:focus`, то есть применять стиль «`invalid`» только тогда, когда пользователь взаимодействует с данным конкретным полем:

```
input:focus:invalid {border:2px solid red;}
input:focus:valid {border: 2px solid green;}
```

Такой вариант работает, но только для поля, на котором установлен фокус ввода. Нам требуется нечто большее: мы хотим, чтобы недопустимые поля оформлялись особенным образом только после того, как пользователь введет туда какие-то данные.

Из-за всех этих проблем в Firefox решили использовать альтернативный псевдокласс под названием `-moz-ui-invalid`, обеспечивающий пользователям намного более удобный интерфейс. Как рассказывают разработчики Mozilla Developer Center, «если данные внутри элемента управления считаются допустимыми в момент, когда пользователь начинает работать с ним, то стилизация поменяется лишь после перевода фокуса на другой элемент управления. Однако если пользователь попытается исправить значение, ранее помеченное как недопустимое, то стилизация поменяется сразу же, как только значение станет допустимым. К элементам, заполнение которых обязательно, псевдокласс применяется только в случае редактирования пользователем, или если пользователь попытается отправить форму с неотредактированным допустимым значением».

Среди других имеющихся псевдоклассов можно также перечислить `:in-range`, `:out-of-range` и `:indeterminate` (последний применяется к числовым полям ввода, для которых заданы значения `min` и `max`, но которые в данный момент пусты).

Пока мы дожидаемся полностью описанного в спецификации и одобренного способа единственнообразной стилизации всех объектов во всех браузерах, можно применять различные трюки и фокусы, которые разработчики изобретают независимо друг от друга для своих продуктов. Например, мы уже познакомились с `ui-invalid` от Mozilla. WebKit предлагает способы стилизации облачков с сообщениями об ошибках стилизации (см. рис. 3.2):

- `::-webkit-validation-bubble{}`
- `::-webkit-validation-bubble-top-outer-arrow{}`
- `::-webkit-validation-bubble-top-inner-arrow{}`
- `::-webkit-validation-bubble-message{}`

В настоящий момент существует не так много методов единообразной настройки внешнего вида и поведения форм HTML во всех браузерах. Но это не обязательно плохо. Специалисты по брендингу, конечно, расстроятся, если текст-подсказка в поле не будет оформлен в корпоративном фиолетово-оранжевом стиле. Но зато вы выигрываете в удобстве использования и доступности. Хотя иногда очень хочется самому создать стили для содержимого полей формы, лучше, по возможности, оставлять их оформление наиболее близким к тому, что браузер использует по умолчанию, — и не важно, что на это скажут специалисты по брендингу. Ползунок и поле выбора даты будут выглядеть в браузере одинаково на разных сайтах, и благодаря этому они будут более понятны пользователю. Гораздо лучше, когда поле выбора даты на сайте X выглядит и работает так же, как на сайте Y или на сайте Z.

И наконец, в большинстве случаев использование встроенных элементов управления вместо их имитации с помощью JavaScript делает ваши формы более доступными для пользователей, пользующихся вспомогательными технологиями.

ПЕРЕОПРЕДЕЛЕНИЕ УМОЛЧАНИЙ БРАУЗЕРА

Встроенные сообщения об ошибках прекрасны, но что если вы хотите изменить эти сообщения в соответствии с требованиями заказчика? Что если сегодня день подражания пиратам? Может быть, я хочу, чтобы все сообщения проверки были написаны на пиратском языке.

Это возможно — нужно всего лишь немного JavaScript-кода, основанного на методе `setCustomValidity`. Однако при задании уникального сообщения само поле становится невалидным, поэтому приходится применять обходное решение: сначала передавать этому методу пустую строку, удаляя любые пользовательские ошибки, вручную выполнять валидацию в коде, а затем уже задавать собственное сообщение об ошибке, которое в итоге увидит пользователь.

Итак, пусть мы не хотим, чтобы на экране отображался такой текст.

humptydumpty is not a legal e-mail address

Тогда мы изменим валидацию так, чтобы сообщение соответствовало правилам «традиционной» речи пиратов.

humptydumpty be not a legal e-mail address

Метод `setCustomValidity` позволяет задать ваше собственное сообщение валидации.

```
<!DOCTYPE html>
<title>custom validity</title>
<form>
<input type=email id=foo name=foo>
<input type=submit>
</form>
<script>
var email = document.getElementById('foo');
email.form.onsubmit = function () {
    // сбросить любые ранее установленные специальные сообщения
```

```

    - об ошибках – пусть выполнится логика валидации браузера
email.setCustomValidity('');
// теперь, когда браузер проверил, является ли введенное значение
- допустимым адресом email, вставим специальное сообщение на случай,
- если валидация пройдена не будет (то есть это недопустимый
- адрес email)
if (!email.validity.valid) {
    email.setCustomValidity(email.value + " be not a legal
    - email address");
}
};

</script>

```

На рис. 3.13 показано, как специальное сообщение валидации отображается в браузере.



Рис. 3.13. Так в Opera выглядят стандартное сообщение об ошибке в адресе (слева) и наше специальное сообщение для дна «разговариваем как злобные пираты» (справа)

К сожалению, на сегодняшний день только Opera в полной мере поддерживает эту возможность. В действительности, неясно даже, должно ли событие `submit` срабатывать при недопустимом содержимом поля. Может быть, нужно прослушивать событие `invalid` для элемента? Но если мы задаем специальное сообщение об ошибке, поле остается в состоянии «недопустимо» даже после того, как ошибка исправлена. Таким образом, проверка на допустимость повторяется, и на второй раз специальное сообщение удаляется — но для этого пользователю приходится *дважды* нажимать на кнопку отправки данных, хотя он всего лишь исправил ошибку.

Получается, что единственный правильный способ задания специального сообщения об ошибке заключается в том, чтобы проверять валидность данных в поле при любом нажатии клавиши. Лично мне кажется странной необходимость опрашивать элемент `input`, однако, в каком-то смысле, он работает так же, как псевдоселектор CSS `:invalid`. Итак, мы можем отредактировать предыдущий пример кода:

```

<!DOCTYPE html>
<title>custom validity</title>
<form>
<input type=email id=foo name=foo>
<input type=submit>
</form>
<script>
var email = document.getElementById('foo');
email.oninput = function () {
    // сбросить любые ранее установленные специальные сообщения
    - об ошибках – пусть выполнится логика валидации браузера
    email.setCustomValidity('');
}

```

```
// теперь, когда браузер проверил, является ли введенное значение
// допустимым адресом email, вставим специальное сообщение на случай,
// если валидация пройдена не будет (то есть это недопустимый
// адрес email)
if (!email.validity.valid) {
    email.setCustomValidity(email.value + " be not a legal
    // email address");
}
};

</script>
```

Однако если вас не устраивает то, как работают специальные сообщения валидации, вы можете задать поведение вашей валидации так, чтобы оно соответствовало вашему приложению. Когда вы запускаете `setCustomValidity`, метод создает DOM-атрибут `validationMessage`, доступный только для чтения. Это можно использовать для управления валидацией, о чём мы и поговорим в следующем разделе.

Кстати, если вы предпочтёте писать код разметки, ориентированный на одного производителя браузера, то можете переопределить сообщения об ошибках в Firefox, применяя уникальный атрибут `x-moz-errormessage`:

```
<input type=email x-moz-errormessage="Please specify a
valid email address.">
```

ИСПОЛЬЗОВАНИЕ JAVASCRIPT ДЛЯ DIY-ВАЛИДАЦИИ

Мы узнали, как настраивать сообщения об ошибках валидации, которые выводятся на экран для пользователя, применяя для этого немного JavaScript-кода. Однако существует возможность с помощью JavaScript управлять всей валидацией и всеми сообщениями для пользователя.

С помощью API JavaScript Web Forms мы можем регулировать то, как результат валидации будет представлен пользователю, но мы также можем передать весь фактический код валидации в ведение API новых форм. Также API позволяет точно определить, почему конкретное поле формы не проходит валидацию.

КАК ИНИЦИИРОВАТЬ ВАЛИДАЦИЮ ЭЛЕМЕНТА

Все элементы форм и элементы ввода данных (включая `<select>` и `<textarea>`) содержат в узле DOM метод `checkValidity`. Ничего страшного, если вы думаете, что именно этот метод нужно использовать для переопределения процессов валидации и обратной связи, использующихся в браузере по умолчанию.

Метод `checkValidity` возвращает значение `true` или `false` в зависимости от успешности выполнения проверок валидации, но в то же время он заставляет браузер выполнять эту проверку и при необходимости выводить сообщения об ошибках. Если вы хотите самостоятельно управлять отображением результатов валидации, вам не стоит использовать этот метод.

ВАЛИДНОСТЬ ЭЛЕМЕНТОВ

У отдельных полей формы помимо метода `checkValidity` есть DOM-атрибут, который возвращает объект `ValidityState`. У объекта валидности есть несколько атрибутов состояния, но самым простым и самым важным из них является атрибут `valid`. Проверяя его значение с помощью JavaScript, вы можете управлять специальной системой валидации с обратной связью.

Если мы подключимся к обработчику события отправки нашей формы, мы сможем вручную просматривать все поля ввода и проверять атрибут валидности. Но что если у поля *нет* правил валидации? Вам повезло: в API есть атрибут `willValidate`, с помощью которого мы можем узнать, следует ли проверять это конкретное поле. Приведу пример (слегка надуманный):

```
var email = document.getElementById('email');
if (email.willValidate) {
    if (!email.validity.valid) {
        alert("Yarr, ye old email be invalid");
    }
}
```

ПРИМЕЧАНИЕ

Следует обратить внимание на то, что у элементов `<fieldset>` тоже есть атрибут `validity`, но он, к сожалению, ничего не делает: атрибут `validity` всегда имеет значение `true`. Для элемента `<fieldset>` вы также можете вызвать метод `checkValidity`, но в современных браузерах, поддерживающих добавление собственной валидации, снова ничего не произойдет.

Если у вас есть состояние валидации отдельного поля формы, вы можете получить сведения о специально заданных сообщениях, для которых использовался метод `element.validationMessage`, или проверить другие состояния валидации, такие как `valueMissing`, `typeMismatch`, `patternMismatch`, `tooLong`, `rangeUnderflow`, `rangeOverflow`, `stepMismatch` и `customError`.

Стоит заметить, что очень важно не забывать об отключении встроенных возможностей валидации браузера. Добавляя атрибут `novalidate` к элементу формы, как вы увидите далее, вы отключаете вывод сообщений о валидации, но вам все так же доступен API JavaScript, и вы можете спокойно проверять состояние валидности полей. Это означает, что в ваших руках по-прежнему находится полный контроль над процессом отображения сообщений об ошибках.

КАК ИЗБЕЖАТЬ ВАЛИДАЦИИ

Последний вопрос, на который мы должны ответить: Что если вы хотите отправить форму, но *не* хотите, чтобы браузер проводил ее валидацию? Такое тоже возможно. Но зачем вам это может понадобиться? Что если у вас есть многоуровневая форма для регистрации или отправки каких-либо данных? В длинных формах удобно выделять уровни (так, например,

делает eBay, когда вы что-то продаете). Также пользователю может понадобиться сохранить состояние заявки, даже если форма еще не заполнена или заполнена неправильно.

Существует два способа отменить валидацию. Это можно сделать применительно к отдельному элементу управления или к целой форме. Атрибут `novalidate`, который может быть задан только для элемента формы, отменяет валидацию данного конкретного поля. Как я уже говорил в предыдущем разделе, если вы желаете отключить встроенные сообщения об ошибках (то есть маленькие облачка с текстом, появляющиеся под элементами ввода), но все так же иметь полный доступ к API JavaScript, который возвращает значения `true` и `false` для состояний валидации поля, — это ваш выбор.

Второй метод — `formnovalidate`, который используется уже сейчас, — можно применять к отдельным элементам ввода данных и кнопочным элементам (хотя, возможно, это имеет смысл только для `type="submit"` и `type="button"`). Атрибут `formnovalidate` позволяет отправить форму без выполнения валидации, которая требуется для полей формы. Приведенный ниже фрагмент кода создает для каждого элемента `<fieldset>` кнопку «`save session`», благодаря чему пользователь может каждый раз сохранять введенные данные, и валидация не выполняется до тех пор, пока не нажата заключительная кнопка отправки.

```
<form>
  <fieldset>
    <legend>Some bits about you</legend>
    <div>
      <label for="email">Email:</label>
      <input id="email" name="email" type="email"
        required />
    </div>
    <div>
      <label for="url">Homepage:</label>
      <input id="url" type="url" name="url" />
    </div>
    <input type="submit" value="save session"
      formnovalidate />
  </fieldset>
</form>
```

С помощью свойства `HTMLFieldSetElement.elements` можно сделать так, чтобы при нажатии кнопки «`save session`» выполнялась JavaScript-валидация только полей, расположенных внутри этого элемента `fieldset` (это свойство является новым в спецификации HTML5, так что, возможно, вам придется воспользоваться `fieldset.getElementsByTagName` и найти все нужные поля формы).

СОБЫТИЕ «ВСЯКИЙ РАЗ, КОГДА ЧТО-ТО МЕНЯЕТСЯ»

Одно практически незаметное изменение, касающееся элемента `<form>`, — добавление нового события `oninput`. На самом деле, это очень полезное событие: оно запускается для элемента формы всякий раз при изменении какого-либо поля. Благодаря этому вам не придется добавлять множество обработчиков `onchange` к каждому элементу управления формой.

ПРИМЕЧАНИЕ

В первом издании этой книги мы говорили о `onforminput`, но с тех пор этот обработчик событий впал в немилость. Однако не пугайтесь, все сказанное в этом разделе все так же имеет силу, так как очень похожий обработчик событий `oninput` входит в спецификацию HTML5 и его можно применять почти так же, как `onforminput`.

Например, если бы мне нужно было создать поле выбора цвета одновременно в модели RGBA и HSLA, мне бы, скорее всего, пришлось добавить приемник событий к каждому ползунку. Но с помощью события `oninputchange` я могу добавить один обработчик событий к форме и пересчитать значения RGBA и HSLA одним методом.

Результат использования одного или нескольких обработчиков событий будет выглядеть почти одинаково. Однако вариант без дублирования обработчиков событий выглядит лучше и аккуратнее.

При передвижении ползунка будут сгенерированы значения RGBA и HSLA, и образец цвета будет обновлен. Ниже приведен JavaScript-код, который для этого потребуется.

```
form.oninput = function () {
    var i = this.length, values = [], value = 0;
    while (i--, value = this[i].value) {
        if (this[i].type == 'range') {
            switch (this[i].name) {
                // значение alpha_channel может быть от 0 до 1
                case 'alpha_channel': values.push(value / 100);
                - break;
                // hue – это простое значение от 0 до 360
                case 'hue': values.push(value); break;
                // default включает процентные значения насыщенности
                - и яркости
                default: values.push(value + '%');
            }
        }
    }
    hsla.value = 'hsla(' + values.reverse().join(', ') + ')';
    preview.style.backgroundColor = hsla.value;
    rgba.value = getComputedStyle(preview, null).
    - backgroundColor;
};
```

В итоге мое поле выбора цвета использует тип ввода данных `range`, новое событие `oninput` и новые элементы вывода для отображения значения (хотя это можно было сделать с помощью `.innerHTML`). Результат показан на рис. 3.14.

Относительно события `oninput` особенно важно отметить, что, в отличие от `onchange`, которое срабатывает, только когда элемент `заполнен` (хотя исторически некоторые браузеры понимали это неправильно и заставляли событие срабатывать во время ввода — удивительно, но в этом списке нет IE!), `oninput` срабатывает, пока пользователь вводит данные в форму, то есть непосредственно в то время, пока изменения вносятся.

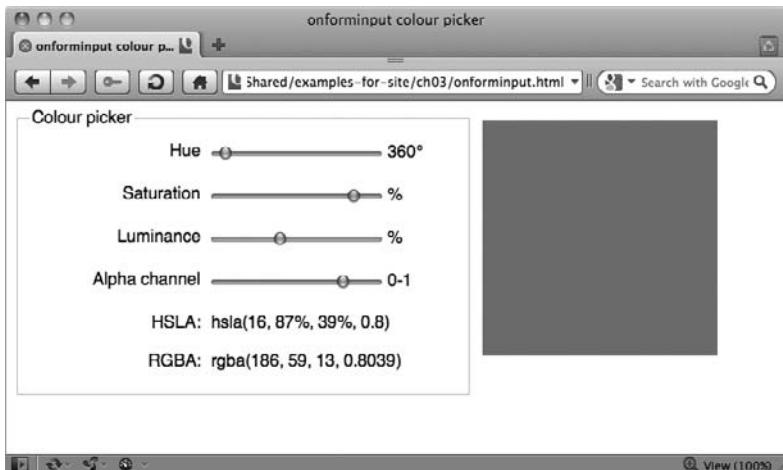


Рис. 3.14. Поле выбора цвета в HSLA с использованием события oninput

ЗАКЛЮЧЕНИЕ

Я надеюсь, вы поняли, что формы HTML5 позволяют разработчикам существенно увеличить производительность, а пользователям — последовательно взаимодействовать с системой. Эти формы предлагают вам огромное множество встроенных возможностей, для которых раньше потребовался бы дополнительный код (например, валидация форм и создание ползунков). Реализация этих возможностей варьируется, она растет в Opera и браузерах WebKit (Safari, Chrome), а в Firefox и IE10 находится на начальной стадии. Отсутствие реализации в IE9 можно компенсировать с помощью JavaScript, поскольку новые свойства устроены по принципу «изящного ухудшения».

А теперь мы перейдем к еще более интересным вопросам.

Глава 4

Видео и аудио

Брюс Лоусон и Реми Шарп



Давным-давно, в далекой-далекой галактике мультимедиа во Всемирной паутине была ограничена проигрыванием MIDI-мелодий и анимированными GIF-файлами. По мере увеличения пропускной способности и усовершенствования технологий сжатия музыка в формате MP3 вытеснила MIDI, и стало появляться привычное нам видео. Разнообразные плееры боролись за место под солнцем — Real Player, Windows Media и т. д., — пока в 2005 году не появился победитель: Adobe Flash, в основном благодаря тому, что его плагин был широко распространен и именно он использовался для YouTube.

HTML5 предлагает конкурирующий открытый стандарт для размещения мультимедийных данных в Сети с оригинальными видео-, аудиоэлементами и API. В этой главе, по большей части, обсуждается элемент `<video>`, так как видео круче, но большинство меток и скриптов применимы и для `<audio>`.

Родная мультимедиа: почему, что и как?

В 2007 году Анне ван Кестерен написала рабочей группе: «У Opera есть несколько внутренних экспериментальных сборок, использующих элемент `<video>`. Элемент работает за счет простого API (пока), похожего на объекты `Audio()`: `play()`, `pause()`, `stop()`. Смысл в том, что он ведет себя как объект (`<object>`), но кроме того, у него есть особая семантика `<video>`, похожая на семантику ``».

API стал более сложным, но первоначальная мысль ван Кестерен и сейчас используется во всех основных браузерах, в том числе Internet Explorer 9.

Элемент `<video>`, очевидно, должен сочетаться с элементом `<audio>`; у них есть множество общих черт. Так что в этой главе мы поговорим и об одном и о другом, а также отметим их сходства и отличия.

<VIDEO>: ЗАЧЕМ НУЖЕН ЭЛЕМЕНТ <VIDEO>?

Раньше, если разработчики хотели разместить видео на веб-странице, им приходилось использовать элемент `<object>`, универсальный контейнер для «внешних объектов». Из-за различий браузеров также приходилось использовать недопустимый ранее элемент `<embed>` и многое дублировать. В результате получался код, который выглядит примерно так:

```
<object width="425" height="344">
<param name="movie" value="http://www.youtube.com/
 - v/9sEI1AUFJKw&hl=en_GB&fs=1"></param>
<param name="allowFullScreen"
value="true"></param>
<param name="allowScriptAccess"
value="always"></param>
<embed src="http://www.youtube.com/
 - v/9sEI1AUFJKw&hl=en_GB&fs=1"
type="application/x-shockwave-flash"
allowScriptAccess="always"
allowFullScreen="true" width="425"
height="344"></embed>
</object>
```

Этот код откровенно ужасен. Что еще хуже, браузеру придется передать видео плагину третьей стороны; и дай бог, чтобы у пользователя была верная версия плагина (или права на его скачивание и установку, или, вообще, навыки установки приложений). И даже после этого есть шанс, что работой плагина будет невозможно управлять с клавиатуры. Возможны и другие проблемы, связанные с передачей информации неродному приложению. Кроме того, плагины часто являются причиной нестабильной работы браузера и могут отпугивать технически не подкованных пользователей, так как им придется загружать и инсталлировать новые версии.

Когда вы добавляете на страницу плагин, под него резервируется область на странице, которая потом передается плагину браузером. Для браузера область, предназначенная для плагина, всегда остается черным ящиком, так как браузер не обрабатывает и не интерпретирует происходящее там.

Обычно это не проблема, но помните, что текст и графика на странице могут пересекаться с областью размещения плагина. Представьте, например, сайт, на котором есть видео и основанное на JavaScript или CSS выпадающее меню, которое должно разворачиваться над фильмом. По умолчанию область, которую занимает плагин, располагается на переднем слое страницы, то есть получается, что элементы меню будут всплывать за фильмом.

Также могут возникнуть сложности, если макет страницы динамически меняется. Если изменяются размеры области плагина, могут возникнуть неприятные сюрпризы (размер проигрываемого фильма не изменится, а свободное место либо останется белым, либо фильм будет обрезан). HTML5 предоставляет стандартизованный способ отображения видео напрямую из браузера без дополнительных плагинов.

Одно из основных достоинств элемента видео в HTML5 — то, что он является полноправным гражданином Сети. Видео больше не отодвигается на задворки `<object>` и не управляется элементом `<embed>`, допустимость которого невозможно проверить.

Итак, теперь элементы `<video>` можно оформлять с помощью кода CSS — например, реализовать изменение размера через переходы (transition) CSS. Видео можно настроить и отобразить в теге `<canvas>` с использованием JavaScript. А самое большое достоинство заключается в том, что теперь мы можем «ковыряться во внутренностях» этого открытого веб-стандартта. Раньше видеоданные находились под замком. С мультимедиа HTML5 информацией можно управлять так, как этого хочется.

ПРИМЕЧАНИЕ

`<embed>` наконец-то стандартизовано в HTML5; этот тег не включался в предыдущие спецификации (X)HTML.

Для чего не подходит мультимедиа HTML5

Несмотря на все достоинства и недостатки HTML5, упоминавшиеся в технической литературе, этот формат не искоренит плагины в один момент, так как некоторые случаи их использования пока не покрываются нововведениями.

ПРИМЕЧАНИЕ

Если вам совсем-совсем не терпится разобраться в тонкостях DRM, зайдите на страницу Генри Сивонена <http://lists.whatwg.org/htdig.cgi/whatwg-whatwg.org/2010-July/027051.html>. Его метод не требует изменений в спецификации.

В частности, в HTML5 не затрагивается вопрос защиты от копирования (что совсем не удивительно, ведь открытость является основой HTML5). Поэтому те пользователи, которым необходимо цифровое управление правами, вероятно, не захотят работать с мультимедиа в формате HTML5, при использовании которого аудио- и видеоданные можно будет загружать на свой компьютер так же просто, как сейчас изображения. Некоторые браузеры предоставляют простой способ доступа к видео (или даже его сохранения) через контекстное меню. Разработчики могут просмотреть источник, найти ссылку на URL-адрес видео и запросто загрузить его. Не то, чтобы мы говорили о том, что цифровое управление правами — штука бессмысленная. Но оно будет только раздражать честных пользователей и создавать лишь небольшие неудобства пиратам.

HTML5 также не помогает с адаптивной потоковой передачей видео. Это процесс, в котором качество видео, загружаемого в браузер, меняется в зависимости от условий среды передачи, для того чтобы пользователю не приходилось ждать слишком долго. Над этим работают, но пока что результат не достигнут окончательно.

Плагины пока что остаются лучшим вариантом передачи видео и аудио через браузер с машины пользователя на веб-страницу типа Daily Mugshot или Chat Roulette, хотя в Chrome, Opera и Firefox есть перспективы использования getUserMedia и WebRTC (см. раздел «Видеоконференции, дополненная реальность» в конце этой главы). Конечно, без Chat Roulette мы бы остались совершенно одинокими в этом мире, но не забывайте и об огромном объеме данных в сети, для визуализации которых нам еще очень долго будут требоваться плагины.

АНАТОМИЯ ЭЛЕМЕНТОВ VIDEO И AUDIO

В самом простом случае для размещения видео на странице в HTML5 потребуется такой код:

```
<video src=turkish.webm></video>
```

Расширение файла .webm используется здесь, чтобы указать на формат видео, закодированного с помощью WebM.

Так же как и при использовании `<object>`, между тегами можно вставить альтернативный код, который сработает в более ранних веб-браузерах, не поддерживающих подобный вариант размещения видео. В этом случае нужно, по крайней мере, предоставить ссылку на видео, тогда пользователь сможет загрузить его к себе на машину и посмотреть через медиаплеер. На рис. 4.1 показан код для современного браузера и для более ранних версий.



Рис. 4.1. Видео HTML5 в современном браузере и сообщение об ошибке в более ранних версиях ПО

```
<h1>Video and legacy browser fallback</h1>
<video src=leverage-a-synergy.webm>
    Download the <a href=leverage-a-synergy.webm>How to
        ~ leverage a synergy video</a>
</video>
```

ПРИМЕЧАНИЕ

Так как основная задача http — это передача ресурсов в веб, сейчас можно просто прописать элемент <video> или <audio>.

Однако пока этот код работать не будет. Вы увидите только первый кадр фильма. Это произойдет потому, что вы ничего не написали о необходимости воспроизведения файла и не указали на то, что браузер должен предоставить элементы управления для воспроизведения или остановки видео.

АВТОВОСПРОИЗВЕДЕНИЕ (AUTOPLAY)

Вы можете указать браузеру на то, что он должен запускать видео или аудио автоматически, но этого делать не следует, так как большинству пользователей такой вариант вряд ли придется по душе (особенно тем, кто использует скринридеры). Те, кто выходит в Интернет через телефон, вероятно, тоже будут не в восторге от подобного положения вещей. Тем не менее вот как это делается:

```
<video src=leverage-a-synergy.webm autoplay>
<!-- альтернативная разметка -->
</video>
```

ЭЛЕМЕНТЫ УПРАВЛЕНИЯ (CONTROLS)

Если вы предоставите пользователю возможность управлять проигрыванием видео при помощи элементов управления, это будет приблизительно в 764 раза лучше, чем автоматическое воспроизведение (рис. 4.2). Можешь использовать более простой вариант JavaScript, чтобы написать свои элементы управления (вернемся к этому позднее) или указать браузеру на то, что он должен предоставить их автоматически:

```
<video src=leverage-a-synergy.webm controls>
</video>
```

Естественно, между элементами управления в разных браузерах могут быть некоторые различия, так как спецификация не диктует внешний вид кнопок. Большинство разработчиков браузеров не пытаются заново изобрести велосипед и предоставляют пользователям привычные элементы управления: воспроизведение и пауза, полоса поиска и регулятор громкости.



Рис. 4.2. Стандартные элементы управления в Firefox (во всех современных браузерах примерно те же)

Обратите внимание на то, что эти элементы управления появляются, когда пользователь наводит на видео курсор или переходит к нему при помощи кнопки Tab (с ее помощью можно также переходить от одного элемента управления к другому). Возможность управления видео с клавиатуры уже является частью плагинов, однако при переходе к клипу от окружающего содержания страницы могут возникнуть проблемы.

Если у элемента `<audio>` есть атрибут `controls`, вы увидите на странице элементы управления. Без этого атрибута вы услышите звук, но на странице вообще ничего не будет отображаться. Конечно, все сохранится в модели DOM, и содержанием можно будет управлять при помощи JavaScript и новых API.

ПРИМЕЧАНИЕ

Уровень доступности элементов управления для пользователей, работающих только на клавиатуре, во всех браузерах разный. В Firefox стрелки вправо и влево по умолчанию позволяют прокрутить содержимое вперед и назад (а если перевести фокус на видео, то нужно использовать стрелки вверх и вниз). Однако визуальных подсказок, сообщающих, на каком элементе находится фокус в данный момент, нет. Элементы управления не отображаются, если в браузере пользователя отключен JavaScript; поэтому, хотя остановить и запустить видео можно и с помощью контекстного меню, программно обнаружить и управлять им может быть сложно.

В Opera, даже если JavaScript выключен, встроенные элементы управления, доступные с клавиатуры, присутствуют на странице всегда — в том числе, если атрибут `controls` не добавлен.

В IE9 уровень доступа с клавиатуры очень высокий. В Chrome и Safari он намного ниже. По мере того как разработчики будут преодолевать трудности первоначального периода, мы надеемся на повышение доступности любых элементов для пользователей клавиатуры.

ИЗОБРАЖЕНИЕ (POSTER)

Атрибут `poster` указывает на то, какое изображение будет использовать браузер, пока загружается видео или пока вы не нажали на кнопку воспроизведения (данный элемент не применяется для `<audio>`). Это избавляет нас от необходимости пользоваться дополнительными ухищрениями, такими как отображение картинки, а затем удаление ее при помощи JavaScript после запуска видео.

Если вы не используете атрибут `poster`, браузер отображает первый кадр фильма, который может не особо хорошо смотреться на странице.

На мобильных устройствах поведение может немного отличаться. Если атрибут `poster` отсутствует, Mobile Safari захватывает первый кадр видео; Opera Mobile экономит пропускную способность и оставляет контейнер пустым.

БЕЗЗВУЧНЫЙ РЕЖИМ (MUTED)

Атрибут `muted` (недавнее добавление к спецификации, что означает очень низкую на сегодняшний день степень поддержки) позволяет по умолчанию отключать звук мультимедийного элемента — пользователь должен самостоятельно включить его, чтобы что-то услышать. Видео (или объявление) воспроизводится автоматически, но беззвучно, для того чтобы не вызывать недовольства пользователей. При желании пользователь может включить звук:

```
<video src="adverts.cgi?kind=video" controls autoplay loop
- muted></video>
```

ВЫСОТА, ШИРИНА (HEIGHT, WIDTH)

Эти атрибуты определяют размер видео в пикселях (они тоже не применяются для `<audio>`). Если вы откажетесь от их использования, браузер возьмет за основу стандартный размер изображения. В том же случае, если данный размер не определен, браузер воспользуется стандартным размером картинки, которая выводится на экран, пока не началось воспроизведение видео. Если не определен ни один из параметров, о которых говорилось выше, размер будет приравнен к 300 пикселям.

Если вы определите только один из атрибутов, браузер подгонит размер, используя коэффициент пропорциональности.

Если вы определили `width` и `height` в диапазоне, который не совпадает с размерами видео, изображение не растянется в соответствии с этими данными, а будет отображаться с черными полосами по краям кадра внутри элемента видео определенного вами размера, сохранив привязку к коэффициенту пропорциональности.

ПОВТОР (LOOP)

Атрибут `loop` также является логическим выражением. Как вы, наверное, догадались, он определяет бесконечный повтор воспроизведения. Пока что он мало где поддерживается, так что не стоит надеяться, что вам удастся добавить на страницу короткий отрывок мелодии и циклически воспроизводить его так, чтобы пользователь не заметил разрывов между

итерациями. Разумеется, ситуация с поддержкой скоро исправится — браузеры, играющие роль медиапроигрывателей, пока что всем в новинку.

ПРЕДВАРИТЕЛЬНАЯ ЗАГРУЗКА (PRELOAD)

Возможно, вы уверены в том, что пользователь захочет активировать медиа (например, он переходит на страницу, на которой нет ничего, кроме этого файла), но вы не хотите использовать атрибут `autoplay`. В этом случае вы можете сделать так, чтобы при загрузке страницы началась буферизация видеофайла, который пользователь активирует при помощи элементов управления.

```
<video src=leverage-a-synergy.ogv controls preload>
</video>
```

Спецификация определяет три значения атрибута `preload`. Если вы просто добавите атрибут в свой код, не определив значение, вы дадите пользовательскому агенту возможность самостоятельно решить, что ему делать. Ведь, например, браузер мобильного телефона может по умолчанию не начинать буферизацию, пока этого не потребовал пользователь. Важно помнить, что веб-разработчик не в состоянии управлять поведением браузера: `preload` играет роль подсказки, а не команды. Браузер сам примет решение, основываясь на том, на какое устройство установлен, каковы условия сети и др.

1. `preload=auto` (или просто `preload`)

Определяя атрибут таким образом, мы предполагаем, что браузер должен начать загрузку файла. Заметьте, это только предположение.

2. `preload=none`

Такой вариант предполагает, что браузер не должен начинать буферизацию, пока пользователь не активирует элементы управления.

3. `preload=metadata`

Этот вариант говорит о том, что браузер должен подгрузить только данные о видео (размер, первый кадр, список воспроизведения, длительность и т. д.) и больше ничего, пока пользователь не активирует элементы управления.

ПРИМЕЧАНИЕ

При условии, что адрес `http` указывает на ресурс потокового воспроизведения в сети, можно просто дать ссылку на него в элементе `<video>` или `<audio>`, и содержимое будет передаваться в потоке.

ИСТОЧНИК (SRC)

Этот атрибут, так же как и в теге ``, указывает на аудио- или видеоресурс, который браузер воспроизведет, если он поддерживает соответствующий формат кодека и контейнера. Ссылка на один исходный файл в атрибуте `src` обычно используется только для быстрого макетирования или для сайтов в интранет-сети, когда вы знаете, какой браузер у пользователя и какие он поддерживает кодеки.

Но так как не все браузеры умеют воспроизводить все существующие форматы, в реальных условиях рекомендуется указывать несколько исходных файлов. Об этом мы поговорим в следующем разделе.

КОДЕКИ: УЖАС-УЖАС

Ранние версии спецификации HTML5 предполагали, что во всех браузерах должна быть обеспечена встроенная поддержка мультимедиа двух типов кодеков: Ogg Vorbis для аудио и Ogg Theora для видео. Vorbis — это кодек, используемый сервисами типа Spotify; он также применяется для записи аудиофрагментов в играх, например Microsoft Halo.

Однако данные кодеки не используются в технических характеристиках HTML5 после того, как против этого выступили Apple и Nokia, так что на данный момент рекомендаций по поводу кодеков нет вовсе. Таким образом, мы сталкиваемся с некоторой проблемой: разные браузеры поддерживают разные форматы, в зависимости от идеологических и коммерческих побуждений.

В настоящее время существует две основные комбинации контейнера и кодека, о которых следует знать всем разработчикам: новый формат WebM (<http://www.webmproject.org>), построенный на базе кодека VP8, купленного компанией Google за 104 миллиона долларов и имеющего открытую лицензию, и вездесущий формат MP4, основанный на патентованном кодеке H.264. Это означает, что в некоторых ситуациях вы обязаны платить владельцам кодека H.264 за публикацию видео, использующих данный кодек. Мы не юристы, поэтому не можем дать конкретных рекомендаций — зайдите на <http://www.mpeгла.com>, и пусть ваши люди побеседуют с людьми тех людей.

Мы добавили кодек Ogg Theora в нашу удобную высокоинформационную таблицу из уважения к истории, но в действительности он может вам пригодиться разве что для реализации поддержки старых версий браузеров с ограниченными возможностями использования элемента `<video>`, таких как Firefox 3.x и Opera 10.x.

ПРИМЕЧАНИЕ

На момент написания этой главы Chrome все еще поддерживает H.264, однако было объявлено о скором прекращении поддержки. Следовательно, вам лучше исходить из предположения, что данный кодек уже не поддерживается.

Таблица 4.1. Поддержка видеокодеков в современных браузерах

	WEBM (кодек VP8)	MP4 (кодек H.264)	OGV (кодек OGG THEORA)
Opera	Да	Нет	Да
Firefox	Да	Нет	Да
Chrome	Да	Да (см. примечание, поддержка будет прекращена)	Да
IE9+	Да (но кодек необходимо устанавливать вручную)	Да	Нет
Safari	Нет	Да	Нет

Поразительно, но два браузера, поддерживающие кодек H.264, принадлежат организации, собирающей авторские отчисления за использование этого кодека (<http://www.mpegl.com/main/programs/AVC/Pages/Licensors.aspx>).

Аналогичная двоякая ситуация существует и в мире аудиокодеков — и также причина проблемы кроется в авторских отчислениях (табл. 4.2).

Таблица 4.2. Поддержка аудиокодеков в современных браузерах

	.OGG/.OGV (кодек VORBIS)	MP3	MP4/M4A (кодек AAC)	WAV
Opera	Да	Нет	Нет	Да
Firefox	Да	Нет	Нет	Да
Chrome	Да	Да	Да	Да
IE9+	Нет	Да	Да	Нет
Safari	Нет	Да	Да	Да

Золотое правило: чтобы все пользователи могли просматривать контент, необходимо на сайте обеспечить поддержку бесплатного формата WebM и видеокодека H.264, а также аудиокодеков Vorbis и MP3. Давайте не будем повторять старые ошибки из серии «Сайт предназначен для браузера Netscape Navigator», связанные с поддержкой форматов.

ПРИМЕЧАНИЕ

Поддержку MP3 в браузер Firefox можно внедрить методом «заполнения» (polyfill). JSmad (<http://jsmad.org>) — это библиотека JavaScript, умеющая налету декодировать MP3 и восстанавливать данные для вывода с помощью API Audio Data (хотя мы сомневаемся, что на слабосильных устройствах производительность окажется достаточно высокой). Изучение подобного API не входит в задачи нашей книги — хотя мы, например, рассмотрели геолокацию, которая также не является частью HTML5 — API отдельных производителей могут слишком широко трактовать формулировки и определения.

МНОЖЕСТВЕННЫЕ ЭЛЕМЕНТЫ <SOURCE>

Чтобы это сделать, нужно закодировать мультимедиа дважды: один раз как WebM и один раз как H.264 для видео, а также Vorbis и MP3 для аудио. А затем необходимо привязать две разные версии файла к элементу медиа.

Раньше, для того чтобы указать источник видео, приходилось использовать синтаксис `<video src="...">`. Это подходит для одного файла, но как сообщить браузеру о существовании нескольких версий видео (с разными вариантами кодирования)? Вместо того чтобы использовать единственный атрибут `src`, вы добавляете внутрь элемента `<audio>` или `<video>` отдельные вложенные элементы `<source>` для каждого варианта кодирования с соответствующими атрибутами `type` и позволяете браузеру подгрузить тот формат, в котором он может отображать данные. Встретив несколько элементов `<source>`, браузер просматривает их в указанном порядке и выбирает первый, который, как он думает, он может воспроизвести (основываясь на значении атрибута `type`, явно сообщающего тип

MIME контейнера и кодек, или если атрибут отсутствует, то основываясь на расширении файла). Заметьте, в этом случае мы не используем атрибут `src` в самом элементе медиа:

```

1 <video controls>
2   <source src=leverage-a-synergy.mp4 type='video/mp4;
      - codecs="avc1.42E01E, mp4a.40.2"'>
3   <source src=leverage-a-synergy.webm type='video/webm;
      - codecs="vp8, vorbis"'>
4 <p>Your browser doesn't support video.
5 Please download the video in <a href=leverage-a-
      - synergy.webm>webM</a> or <a href=leverage-a-
      - synergy.mp4>MP4</a> format.</p>
6 </video>
```

КАКОЙ КОДЕК ЛУЧШЕ?

Вопрос, что «лучше» — WebM или MP4, — неизменно приводит к спору, по сравнению с которым война сторонников Mac и PC кажется тихой беседой в кругу старых друзей.

Обсуждение внутренних характеристик означает дискуссию о типах макроблоков в В-кадрах и фильтрации б-тап для предсказаний отсчета яркостной составляющей с точностью 1/2 пикселя — фактически, спор в стиле «мой потоковый накопитель больше твоего!»

Достаточно будет сказать, что для загрузки видео по сети оба кодека, WebM и MP4, обеспечивают достаточно хорошее качество и степень сжатия. Ogg Theora менее дружелюбен по отношению к пользователям сети.

Настоящие различия кроются в авторских отчислениях и аппаратном ускорении. Некоторым людям приходится платить за добавление видео MP4/H.264 на свой веб-сайт.

Многие микропроцессоры выполняют аппаратное декодирование данных H.264 — вот почему при попытке просмотреть видео на мобильном телефоне его аккумулятор не разряжается за считанные секунды, как было бы, если бы декодирование выполнялось программно. На момент написания этой главы (июль 2011 года, спустя год после открытия исходного кода WebM) микропроцессоры с аппаратным декодированием WebM только-только начинают появляться на рынке.

Первая строка указывает браузеру на то, что в отображаемый контент нужно включить видео и предоставить стандартные элементы управления. Вторая предлагает видео в формате MP4. Мы поставили версию mp4 на первое место, так как в некоторых старых версиях Mobile Safari на iPad есть ошибка — они распознают только первый элемент `<source>`, и если первым указано не видео в формате mp4, оно не воспроизводится. Мы добавляем атрибут `type`, чтобы браузер мог определить, какой тип контейнера используется (при помощи указания типа MIME) и какой кодек использовался для кодировки видео- и аудиопотока. Если вы не определите атрибут `type`, браузер загрузит по небольшой части каждого файла, прежде чем выяснит, что какой-то из типов не поддерживается. На это уйдет время и трафик, а отображение видео начнется позднее.

Обратите внимание на кавычки вокруг параметров: в спецификации рекомендуется использовать `'video/mp4; codecs="avc...'"` (одинарные снаружи и двойные вокруг кодека). Некоторые браузеры спотыкаются, если кавычки используются наоборот. В третьей строке предлагается видео WebM. Стока с кодеками для H.264 и AAC сложнее, чем для WebM, так как для H.264 и AAC есть несколько профилей для разных категорий устройств

и типов соединения. Более высокие профили сильнее нагружают ЦПУ при декодировании, но коэффициент сжатия у них выше, так что на передачу видео уходит меньше трафика. Мы могли бы также предложить видео в формате Ogg для старых версий Firefox и Opera, добавив соответствующее указание после версии WebM. Таким образом, если такая возможность у него есть, браузер выберет более высококачественную версию WebM, а более старые (но поддерживающие элемент `<video>`) браузеры остановятся на Ogg.

Внутри элемента `<video>` есть сообщение о некорректном срабатывании, в котором содержатся ссылки на оба формата (для браузеров, которые не могут реализовать ни один из них, но при условии, что операционная система предлагает возможности просмотра видео). Таким образом, пользователь сможет загрузить файл и посмотреть его в медиаплеере.

Все это время мы говорили о видео HTML5 для пользователей современных браузеров. А что насчет старых, таких как Internet Explorer 8 и еще более ранних?

ВИДЕО ДЛЯ УСТАРЕВШИХ БРАУЗЕРОВ

Старые версии браузеров не смогут проигрывать видео и аудио таким способом. Но если вас не пугает тот факт, что придется полагаться на плагины, вы можете обеспечить пользователям более ранних версий браузеров возможность обрабатывать контент способом, ничуть не уступающим тому, который они используют на данный момент.

Вы не забыли, что содержимое элемента `<video>` может включать разметку; такую как текст или ссылки (см. предыдущий пример)? Мы поместим целый ролик в формате Flash внутрь такого альтернативного содержимого (и, разумеется, добавим обходной путь для бедных пользователей, у которых нет даже Flash). К счастью, нам не придется кодировать наше видео в еще одном формате, таком как FLV (устаревший формат контейнера для видео, принадлежащий Flash); так как Flash (начиная с версии 9) умеет загружать файлы MP4 в качестве внешних ресурсов, можно просто подгрузить такой файл в Flash-проигрыватель. Такая комбинация даст нам надежное решение для Internet Explorer 8 и устаревших версий других браузеров. Вы не сможете различными способами манипулировать видео (мы узнаем много интересного о том, что можно сделать с видео, чуть дальше в этой главе), но, по крайней мере, ваши пользователи смогут просмотреть ролик при любом раскладе.

Код, который нам придется для этого написать, конечно, ужасен. Тем не менее он будет работать в любой версии браузера, если установлен Flash Player, — то есть почти везде. Вы можете ознакомиться с этой остроумной техникой в статье под названием «Видео для всех!», написанной Кроком Каменом (http://camendesign.com/code/video_for_everybody).

В качестве альтернативного варианта видеоролик можно загрузить на специальный сайт, позволяющий хранить видеофайлы, и вставить ссылку на файл между тегами элемента `video`:

```
<video controls>
  <source src=leverage-a-synergy.mp4 type='video/mp4;
  - codecs="avc1.42E01E, mp4a.40.2"'>
  <source src=leverage-a-synergy.webm type='video/webm;
  - codecs="vp8, vorbis"'>
```

```
<embed src="http://www.youtube.com/v/cmtcc94Tv3A&hl=
- en_GB&fs=1&rel=0" type="application/x-shockwave-flash"
- allowscriptaccess="always" allowfullscreen="true"
- width="425" height="344">
</video>
```

ПРИМЕЧАНИЕ

Контент между тегами используется при ошибке загрузки в браузерах, которые не поддерживают элемент `<video>`. Браузер, который хоть как-то обрабатывает видео HTML5, но не может запустить ни один из форматов, указанных в коде, не будет использовать данный контент — пользователи увидят «сломанный» элемент воспроизведения видео. Несколько раз это создавало мне проблемы, но, к сожалению, подтверждающей это видеозаписи не осталось.

КОДИРУЕМ БЕСПЛАТНОЕ ВИДЕО И АУДИО

В идеале, следует исходить из самого формата источника, а не сжимать уже и без того сжатую версию. Двойное сжатие может серьезно снизить качество продукта на выходе. Если у вас есть сильно сжатая версия видео в формате MP4/H.264, не пытайтесь преобразовать ее в WebM/VP8, а лучше вернитесь к исходному необработанному видеофайлу и выполните сжатие для него. Что касается аудио, в открытом ПО для редактирования аудиофайлов Audacity (<http://audacity.sourceforge.net/>) есть встроенная поддержка экспорта Ogg Vorbis.

Для преобразования видео также есть пара отличных способов. Для WebM на данный момент существует лишь несколько кодировщиков, что не удивительно, учитывая возраст данного кодека. Загляните на <http://www.webmproject.org/tools/>, чтобы ознакомиться с постоянно пополняемым списком инструментов для этой задачи.

Пользователи Windows и Mac могут загрузить Miro Video Converter (<http://www.mirovideo-converter.com/>), который позволяет перетащить файл в окно программы и таким образом преобразовать его в WebM, Theora или H.264, оптимизировав результирующий файл для различных устройств, таких как iPhone, Android Nexus One, PS2 и т. д.

Бесплатная программа VLC (<http://www.videolan.org/vlc/>) может конвертировать файлы в Windows, Mac или Linux.

Те разработчики, кого не пугает команда строка, могут воспользоваться библиотекой с открытым кодом FFmpeg (<http://ffmpeg.org>) — это настоящий монстр конвертации. Все, что вам нужно, это `$ ffmpeg -i video.avi video.webm`.

Процесс конвертации может также быть автоматизирован и привязан к серверу. Например, в среде CMS у вас нет возможности проконтролировать формат, в котором авторы подгружают свои файлы, так что сжатие файлов на сервере представляется весьма осмысленным. ffmpeg можно установить на сервер для того, чтобы проводить массовое преобразование загружаемых файлов (может, организуете собственную альтернативу YouTube?).

Если для вас принципиально важным вопросом является объем памяти и вы готовы делиться своими файлами (аудио и видео) в соответствии с одной из различных CC-лицензий, ознакомьтесь со средством Internet Archive (<http://www.archive.org/create/>), которое конвертирует и сохранит их за вас. Просто создайте пароль и приступайте к загрузке, а затем поставьте ссылку на источник и используйте элемент `<video>` на своей странице.

Еще один вариант конвертации и хранения средствами третьей стороны — это vid.ly. Эта бесплатная служба позволяет загружать до 2 Гбайт видеоданных через веб-сайт, после чего данные конвертируются. Когда пользователи зайдут на сайт, им будет предложен видеоролик, сконвертированный с использованием подходящего кодека, даже если пользователи просматривают сайт с мобильного телефона.

Можно использовать библиотеку HTML5 Media Library (<http://html5media.info>), чтобы исключить элемент `<video>` и автоматически добавить необходимый вариант исправления ошибки, написав одну строку на JavaScript в верхней части страницы.

ВИДЕОФАЙЛЫ РАЗНЫХ ФОРМАТОВ И КАРМАННЫЕ УСТРОЙСТВА

Обычно видеофайлы довольно объемны, а загрузка файла в высоком разрешении на карманные устройства с маленьким экраном едва ли оправдана. Нет никакого смысла в том, чтобы отсыпалить видео в формате HD, предназначенном для больших экранов, на экран смартфона или КПК, и большинство пользователей карманных устройств с радостью согласятся на небольшую потерю качества при условии, что видео будет оперативно загружаться по мобильному соединению. Сжатие видео до размера, подходящего для маленького экрана, может сэкономить трафик. А это, безусловно, не только поднимет настроение обладателю карманного устройства, но и сделает спокойнее жизнь вашего сервера.

HTML5 предлагает атрибут `media` для элемента `<source>`, позволяющий опросить браузер и узнать через него размер экрана (или количество цветов, коэффициент пропорциональности и т. д.). Затем отсылается тот файл, который лучше подходит под характеристики экрана.

Эта функциональность и синтаксис заимствованы из спецификации CSS Media Queries (<http://www.w3.org/TR/css3-mediaqueries/>), но являются частью разметки, так как мы переключаем исходные файлы в зависимости от того, каковы характеристики устройства. В следующем примере браузеру отправляется запрос, допустима ли ширина экрана устройства в 800px, то есть допустим ли широкоформатный режим. Если ответ на запрос положительный, браузер получает файл hi-res.webm; если нет — lo-res.webm:

```
<video controls>
  <source src=hi-res.webm ... media="(min-device-width:
    - 800px)">
  <source src=lo-res.webm>
  ...
</video>
```

ПРИМЕЧАНИЕ

Мы используем `min-device-width`, а не `min-width`. Мобильные браузеры (которые могут сообщать неправильный размер области отображения, для того чтобы веб-страницы лучше выглядели на экране) в таком случае сообщают номинальную ширину физического экрана.

Обратите внимание на то, что вы все еще должны использовать атрибут `type` с параметрами `codecs` и контентом, демонстрирующимся при ошибке воспроизведения, о котором мы говорили ранее. В данном случае мы его не указали, чтобы не перегружать код.

ПОЛЬЗОВАТЕЛЬСКИЕ ЭЛЕМЕНТЫ УПРАВЛЕНИЯ

Одна из самых удачных особенностей элементов медиа `<video>` и `<audio>` — это то, что API JavaScript для них крайне прост. События и методы API одинаковы и для `<audio>`, и для `<video>`. Уяснив это, перейдем к обсуждению более интересного элемента медиа, `<video>`, и работы с ним при помощи JavaScript.

Как вы уже поняли из начала этой главы, Анне ван Кестерен рассказала о новом API и о новых простых методах, таких как `play()`, `pause()` (метода `stop` нет, но можно использовать паузу и вернуться к началу файла), `load()` и `canPlayType()`. На самом деле, этими методами список исчерпывается. Все остальное относится к событиям и атрибутам.

В табл. 4.3 вы найдете список атрибутов, методов и событий для медиаэлементов.

Используя JavaScript и новый API для медиаэлементов, вы можете создавать и организовывать свои собственные элементы управления видеоплеером. Мы на примере покажем вам, как создать простой набор элементов управления, и продемонстрируем их в действии. Наш пример вряд ли будет для вас большим откровением (он далеко не так крут, как сам элемент `<video>`; кроме того, он несколько искусственен), но вы получите хорошее представление о том, что можно сделать при помощи написания скриптов. Самое лучшее в нашем примере то, что пользовательский интерфейс будет построен на основе CSS и HTML. Так что для внесения в него каких-либо изменений потребуются лишь поверхностные знания в области веб-стандартов. Переписывать код внешнего Flash-плеера не будет ни малейшей необходимости.

Среди основных элементов управления видеоплеера, которые мы подкорректируем вручную, будет переключатель воспроизведение/пауза; кроме того, у пользователя будет возможность перемещать бегунок таким образом, чтобы пропускать отдельные части видеофайла, как показано на рис. 4.3.



Рис. 4.3. Наши простые, но уже видоизмененные элементы управления плеером

Таблица 4.3. Атрибуты, методы и события для медиаэлементов

Состояние ошибки	load()
error	canPlayType(type)
Состояние сети	play()
src	pause()
currentSrc	addTrack(label, kind, language)
networkState	
preload	События
buffered	loadstart
Состояние готовности	progress
readyState	suspend
seeking	abort
Элементы управления	error
controls	emptied
volume	stalled
muted	play
Треки	pause
tracks	loadedmetadata
Состояние воспроизведения	loadeddata
currentTime	waiting
startTime	playing
duration	canplay
paused	canplaythrough
defaultPlaybackRate	seeking
playbackRate	seeked
played	timeupdate
seekable	ended
ended	ratechange
autoplay	
loop	
Специфичные для видео	
width	
height	
videoWidth	
videoHeight	
poster	

ПРИМЕЧАНИЕ

В некоторых браузерах, в частности в Opera, встроенные элементы отображаются, даже если JavaScript отключен; в других браузерах ситуации могут быть какими угодно.

Начнем мы с видео со стандартными элементами управления. Затем мы используем JavaScript, чтобы переписать их в соответствии с нашими желаниями. Таким образом, если JavaScript отключен, у пользователя все-таки будет возможность контролировать видео:

```
<video controls>
  <source src="leverage-a-synergy.webm" type="video/webm" />
  <source src="leverage-a-synergy.mp4" type="video/mp4" />
  Your browser doesn't support video.
  Please download the video in <a href="leverage-a-
  - synergy.webm">WebM</a> or <a href="leverage-a-
  - synergy.mp4">MP4</a> format.
</video>
<script>
var video = document.getElementsByTagName('video')[0];
video.removeAttribute('controls');
</script>
```

ВОСПРОИЗВЕДЕНИЕ, ПАУЗА — И ВСЕ ЭТО НА ОДНОЙ КНОПКЕ

Теперь мы хотим, чтобы у нас была возможность воспроизводить и ставить на паузу видео при помощи элементов управления, над которыми мы уже немного потрудились. Мы добавили кнопку, с которой связем обработчик нажатия и, таким образом, реализуем функциональность «воспроизведение/пауза». В моих примерах кода, когда я упоминаю об объекте play, имеется в виду этот элемент button:

```
<button class="play" title="play">▶&#x25BA;</button>
```

▶►
||▐ (twice)

Рис. 4.4. Сущности XML представляют кнопки воспроизведения и паузы

Мы используем ►, геометрическую сущность XML, которая *выглядит как кнопка «воспроизведение»*. Когда мы нажимаем на кнопку, начинается воспроизведение видео, а кнопке приписывается новое значение, ▐, описывающие две вертикальные линии (похожие на кнопку «пауза») — это показано на рис. 4.4.

Для простоты я сделал кнопку при помощи метки, но по мере применения к нашим элементам управления видео прогрессивного усовершенствования все дополнитель-

ные элементы (для воспроизведения, паузы, промотки и т. д.) должны быть описаны при помощи JavaScript.

К переключателю «воспроизведение/пауза» нам нужно добавить еще несколько деталей:

1. Если пользователь щелкает на кнопке переключения и видео стоит на паузе, то необходимо начать воспроизведение. Если воспроизведение завершено и указатель текущего

кадра находится в самом конце видео, нужно сбросить текущее значение времени на 0, то есть перевести указатель на начало видео и запустить воспроизведение.

2. Нужно запрограммировать изменение значения кнопки-переключателя, чтобы, когда пользователь нажимает на нее, она менялась с паузы на воспроизведение или с воспроизведения на паузу.
3. Наконец, мы воспроизводим видео (или ставим его на паузу):

```
playButton.addEventListener('click', function () {
    if (video.paused || video.ended) {
        if (video.ended) {
            video.currentTime = 0;
        }
        this.innerHTML = ''; // &#x2590;&#x2590; doesn't
        - need escaping here
        this.title = 'pause';
        video.play();
    } else {
        this.innerHTML = ''; // &#x25BA;
        this.title = 'play';
        video.pause();
    }
}, false);
```

Проблема этой логики заключается в том, что мы полностью полагаемся на наш скрипт при определении состояния кнопки «воспроизведение/пауза». А что если у пользователя будет возможность каким-то образом воспроизводить и ставить на паузу видео при помощи родных элементов управления (некоторые браузеры позволяют пользователю выбрать данные опции при нажатии правой кнопки мыши)? Кроме того, когда воспроизведение закончится, кнопка «воспроизведение/пауза» будет выводить значок паузы. На самом деле, нам нужно, чтобы элементы управления всегда работали в зависимости от состояния видео.

БОГАТЫЕ СОБЫТИЯМИ ЭЛЕМЕНТЫ МЕДИА

Вокруг элементов медиа есть масса событий: начало воспроизведения, конец загрузки, изменение громкости и т. д. Так что для дальнейшей работы с кнопкой «воспроизведение/пауза» мы беремся за часть скрипта, которая сосредоточена на изменении визуального отображения:

```
playButton.addEventListener('click', function () {
    if (video.ended) {
        video.currentTime = 0;
    }
    if (video.paused) {
        video.play();
    } else {
        video.pause();
    }
}, false);
```

В простейшем коде, если воспроизведение файла закончилось, мы сбрасываем видео на начало, а затем изменяем значение переключателя в зависимости от его текущего состояния. Сама метка на элементе управления изменяется благодаря отдельным (анонимным) функциям, которые мы напрямую добавили к обработчикам событий нашего элемента видео:

```
video.addEventListener('play', function () {
    play.title = 'pause';
    play.innerHTML = '';
}, false);
video.addEventListener('pause', function () {
    play.title = 'play';
    play.innerHTML = '';
}, false);
video.addEventListener('ended', function () {
    this.pause();
}, false);
```

ПРИМЕЧАНИЕ

В этих примерах мы используем API модели DOM второго уровня `addEventListener`, а не `attachEvent`, так как последний используется исключительно в Internet Explorer до восьмой версии. IE9 поддерживает видео, кроме того, он, к счастью, поддерживает еще и стандартный `addEventListener`, так что и в этом браузере наш код будет работать.



Рис. 4.5. Наш специальный прогресс-бар для видео, включая поиск по контенту и текущую позицию воспроизведения

Теперь каждый раз, когда видео проигрывается, ставится на паузу или заканчивается, запускается функция, привязанная к соответствующему событию, которая проверяет, правильный ли вариант кнопки отображается на панели.

Теперь, после того как мы описали воспроизведение и паузу, мы хотим показать пользователю, какая часть видео загрузилась, то есть какая часть может быть воспроизведена. Как вы уже догадались, речь идет о количестве буферизованного видео. Мы также хотим поймать событие, сообщающее, какая часть видео уже была проиграна, чтобы бегунок на панели воспроизведения перемещался в соответствующую область, как показано на рис. 4.5. И наконец, нам необходимо описать событие, которое говорит о том, что можно начать просмотр, то есть у нас достаточно количество буферизованного видео для начала воспроизведения.

ОТСЛЕЖИВАЕМ ХОД ЗАГРУЗКИ

У элемента `media` есть событие `progress`, которое срабатывает, когда файл медиа уже обнаружен, но еще не обработан. При активации этого события мы можем считывать объект `video.seekable`, у которого есть методы `length`, `start()` и `end()`. Мы можем изменить

панель и добавить отображение буферизованного видео (как показано на рис. 4.5 на второй картинке светло-серым цветом), используя следующий код (где переменная `buffer` — это элемент, указывающий на то, в пределах какой части видео можно осуществлять поиск, то есть какая часть файла была загружена):

```
video.addEventListener('progress', updateSeekable, false);
function updateSeekable() {
    var endVal = this.seekable && this.seekable.length ?
        ~this.seekable.end() : 0;
    buffer.style.width = (100 / (this.duration || 1) *
        ~endVal) + '%';
}
```

Код связан с событием `progress`, и когда оно срабатывает, мы видим, какая часть от всего видео может быть воспроизведена на данный момент. Обратите внимание на ключевое слово `this`, которое относится к элементу `video`. Именно оно является контекстом, в рамках которого будет реализовываться функция `updateSeekable`. А атрибут `duration` — это длина медиафайла в секундах.

Однако в Firefox иногда возникает проблема: в предыдущих версиях значение `seekable.length` не совпадало с фактической длительностью видео, а в последней версии (5.0.1) атрибут `seekable` вообще отсутствует. Для того чтобы диапазон поиска по видеофрагменту неискажался, можно включить прослушивание события `progress` и анализировать значение `duration` для нашего элемента `video`:

```
video.addEventListener('durationchange', updateSeekable,
    ~false);
video.addEventListener('progress', updateSeekable, false);
function updateSeekable() {
    buffer.style.width = (100 / (this.duration || 1) *
        (this.seekable && this.seekable.length ? this.seekable.
            ~end() : this.duration)) + '%';
}
```

Неприятно, конечно, что у нас нет надежного способа определения диапазона видеоролика, по которому возможен поиск. Альтернативное решение основывается на использовании свойства `video.buffered`, но нам оно не подходит: мы пытаемся справиться с проблемой в Firefox, а данное значение в настоящий момент в Firefox не возвращает в методе `video.buffered.end()` ничего — следовательно, использовать его невозможно.

Когда медиафайл готов к воспроизведению

Браузер не может начать воспроизведение медиафайла, как только обнаружит его на странице. Видео (или аудио) должно быть загружено и декодировано и только после этого станет возможным его воспроизведение. Как только вся подготовительная работа проведена, элемент медиа запускает событие `canplay`. Обычно именно в это время удаляется индикатор загрузки, и вы инициализируете элементы управления. Наш код для инициализации элементов управления в обычных условиях выглядел бы примерно так:

```
video.addEventListener('canplay', initialiseControls,
    ~false);
```

В нем нет ничего сверхъестественного. Инициализация элементов управления активирует переключатель «воспроизведение/пауза», а на полосе прокрутки сбрасывает указатель текущего кадра.

Однако иногда это событие запускается не сразу (или, по крайней мере, не в тот момент, когда вы этого ожидаете). Время от времени загрузка приостанавливается из-за того, что браузер пытается сохранить слишком большую часть загружаемой информации. Это может быть крайне неприятно, если вы уже ожидаете событие `canplay`, которое не запустится, пока вы не поработаете над элементом медиа. Так что мы лучше займемся наблюдением за событием `loadeddata`.

ПРИМЕЧАНИЕ

События, которые имеют отношение к загрузке, запускаются в следующем порядке: `loadstart`, `durationchange`, `loadedmetadata`, `loadeddata`, `progress`, `canplay`, `canplaythrough`.

Оно говорит нам о том, что загруженные данные уже есть, но пока загрузка не завершена. Это значит, что доступны метаданные (высота, ширина, длительность и т. д.) и часть медиаконтента — но не весь. Такие браузеры, как Firefox, предоставляют пользователю возможность запустить видео, когда сработало событие `loadeddata`, и переходят из состояния ожидания к загрузке оставшейся части медиаконтента, что позволяет воспроизвести видео целиком.

Возможно, вы обнаружите, что в большинстве ситуаций, когда вы создаете собственный интерфейс медиапроигрывателя, вам не приходится ждать загрузки фактических видеоданных — достаточно только метаданных. Это означает, что срабатывает событие `loadedmetadata`: а для того чтобы оно сработало, нужно, чтобы загрузился первый кадр, информация о продолжительности и размере кадра и другие метаданные. Фактически этого достаточно для создания собственного пользовательского интерфейса.

Таким образом, `loadeddata` — это как раз та точка цикла событий, в которой должен быть активирован пользовательский интерфейс:

```
video.addEventListener('loadedmetadata', initialiseControls,
- false);
```

ПРЕДВАРИТЕЛЬНАЯ ЗАГРУЗКА МЕТАДАННЫХ

Элементы медиа также поддерживают атрибут `preload`, который позволяет разработчикам указывать долю видеоролика, после загрузки которой можно отображать страницу. По умолчанию его значение равно `auto`, но можно также использовать `none` и `metadata`. Если значение атрибута равно `none`, то пользователь увидит либо изображение, которое вы выбрали для атрибута `poster`, либо ничего, если вы не указали никакого изображения. Сам медиафайл будет запрошен с сервера только после того, как пользователь попытается воспроизвести медиаэлемент.

Если значение атрибута `preload` равно `metadata`, то браузер загружает необходимые метаданные медиаэлемента. Также он запускает событие `loadedmetadata`, что весьма полезно, если вы прослушиваете именно это событие для настройки собственного интерфейса медиапроигрывателя.

Гонка за воспроизведением видео

До этого я представлял вам элементы видео и аудио как бочки, наполненные свежим благопухающим медом, однако и здесь без ложки дегтя не обошлось. В реализации элемента `video` существует проблема, создающая так называемое «условие состязания».

СОСТЯЗАНИЕ, НУ И ЧТО?

В данной ситуации условие состязания означает, что ожидаемая последовательность срабатывания событий нарушается и события выполняются в непредсказуемом порядке. В частности, события срабатывают до того, как присоединяется код для их обработки.

Проблема (хоть и маловероятная) заключается в том, что браузер может загрузить элемент медиа до того, как у вас будет время привязать приемники событий.

Например, вы прослушиваете событие `loadedmetadata`, для того, чтобы понять, когда видео будет готово к использованию в вашем модном, сделанном собственными руками видеоплеере, однако возможно, что «родной» элемент `video` HTML запустит события *до того*, как будет загружен ваш код JavaScript.

ОБХОДНЫЕ ПУТИ

Для данного условия состязания существует несколько обходных путей — конечно, хорошо бы было их избегать, но, боюсь, они относятся к тем вещам, которые необходимо добавлять в код для собственной защиты.

Обходной путь № 1: делегирование более высокому событию

В данном решении нам необходимо присоединить обработчик события к объекту `window`. Таким образом, обработчик событий *должен быть* выше элемента медиа. Очевидный недостаток данного подхода заключается в том, что элемент сценария находится выше контента, вследствие чего появляется риск блокировки загрузки контента (в целом, рекомендуется добавлять все блоки сценариев в конце документа).

Тем не менее спецификация HTML5 утверждает, что события медиа должны всплыть в DOM на самый верх, вплоть до объекта `window`. Следовательно, когда событие `loadedmetadata` срабатывает на объекте `window`, мы проверяем, где это событие было создано (с помощью свойства `target`), и если это наш элемент, можно запускать код настройки. Обратите внимание, что в представленном ниже примере я проверяю только `nodeName` элемента; вы можете выполнить данный код для всех элементов `audio` или проверить больше свойств в узле DOM, чтобы удостовериться, что нашли правильный.

```
<script>
function audioloaded() {
    // настройка модного проигрывателя
}

window.addEventListener('loadedmetadata', function (event) {
    if (event.target.nodeName === 'AUDIO') {
```

```
// привязка данного контекста к узлу DOM
audioloaded.call(event.target);
},
}, true);

</script>

<audio src="hanson.mp3">
    <p>Если вы видите этот текст, значит, не сможете насладиться
        - прекрасными голосами братьев Хэнсонов.</p>
</audio>
```

Обходной путь № 2: встраиваем повыше

Так реализовывается похожий подход с использованием встроенного обработчика (inline handler):

```
<script>
function audioloaded() {
    // настройка модного проигрывателя
}
</script>

<audio src="hanson.mp3" onloadedmetadata=
    - "audioloaded.call(this)">
    <p>Если вы видите этот текст, значит, не сможете насладиться
        - прекрасными голосами братьев Хэнсонов.</p>
</audio>
```

Обратите внимание, что во встроенном обработчике событий я использую `.call(this)` для привязки ключевого слова `this` к элементу `audio`, для которого сработало событие. Это означает, что ту же функцию будет проще повторно использовать позже, если когда-нибудь (через много лет) разработчики исправят эту ошибку.

Встраивая обработчик событий, мы гарантируем, что обработчик будет присоединен сразу же, как только создастся элемент DOM, то есть он окажется на месте до того, как сработает событие `loadedmetadata`.

Обходной путь № 3: элемент медиа генерируется с помощью JavaScript

Еще один обходной путь подразумевает вставку элемента медиа с помощью JavaScript. Вы создаете элемент медиа, присоединяете обработчики событий, а затем задаете источник и вставляете его в DOM.

Помните: при вставке элемента медиа с использованием JavaScript приходится либо вручную добавлять весь набор элементов `source`, либо распознавать возможности браузера и добавлять атрибут `src` со значением, которое данный браузер поддерживает, например WebM/video для Chrome.

Мне не слишком-то нравится это решение, так как оно подразумевает, что пользователи, в браузерах которых JavaScript отключен, вообще не получают никакого видео или аудио. Несмотря на то что многое, сделанное на базе HTML5, представляет собой «веб-

приложения», что-то говорит мне (и, надеюсь, вам), что это не очень круто, когда на JavaScript приходится полагаться *только* для того, чтобы заставить связанные с видео события работать так, как нам требуется. Даже если вы не так уж похожи на меня (что вполне вероятно), поисковые машины большого брата Google не смогут найти и проиндексировать ваше поразительное видео с котиком, танцующим под братьев Хэнсонов, если ролик будет вставляться на страницу посредством JavaScript-кода. Так что давайте сразу перейдем к моему любимому обходному пути № 4.

Обходной путь № 4: проверка readyState

Это, вероятно, наилучший подход, хотя и немного беспорядочный (по сравнению с одним элементом `video` и обработчиком событий). Мы будем проверять свойство `readyState` для элемента медиа. У обоих элементов, `audio` и `video`, есть данное свойство, и оно может принимать следующие значения:

- `HAVE NOTHING` = 0;
- `HAVE_METADATA` = 1;
- `HAVE_CURRENT_DATA` = 2;
- `HAVE_FUTURE_DATA` = 3;
- `HAVE_ENOUGH_DATA` = 4;

Таким образом, если ваша цель — привязка к событию `loadedmetadata`, делать это нужно только тогда, когда значение `readyState` равно 0. Или, если вы хотите выполнить привязку до того, как будет достаточно данных для воспроизведения, просто выбирайте вариант, когда `readyState` меньше 4.

Наш предыдущий пример можно переписать так:

```
<audio src="hanson.mp3">
    <p>Если вы видите этот текст, значит, не сможете насладиться
       - прекрасными голосами братьев Хэнсонов.</p>
</audio>

<script>
function audioloaded() {
    // настройка модного проигрывателя
}

var audio = document.getElementsByTagName('audio')[0];

if (audio.readyState > 0) {
    audioloaded.call(audio);
} else {
    audio.addEventListener('loadedmetadata', audioloaded,
        false);
}
</script>
```

Таким образом, наш код аккуратно разместится внизу документа, и если JavaScript в браузере будет отключен, аудиоролик все равно будет доступен. В моей книге — только лучшее.

А эту проблему с условием состояния когда-нибудь починят?

Я понимаю, что, технически, эта проблема всегда существовала в браузерах. Например, элемент `image`: если событие `load` срабатывает до того, как присоединяется обработчик события загрузки, ничего не происходит. Такое можно наблюдать, когда изображение уже загружено в кэш — оно загружается слишком быстро, или в среде разработки, где скорость доставки такая, будто файлы подгружает обшившийся энергетиков Супермен. Событие попросту не срабатывает.

У изображений нет состояний `ready`, но есть свойство `complete`. Пока изображение загружается, значение `complete` равно `false`. Как только загрузка завершилась (обратите внимание, что это также означает завершение загрузки с ошибкой — из-за какого-то сбоя), свойство `complete` принимает значение `true`. Таким образом, перед привязкой события `load` можно проверять свойство `complete` и, если оно равно `true`, вручную запускать обработчик события `load`.

Поскольку данная логика для изображений существует уже очень давно, было бы логично ожидать, что ее распространят и на элементы медиа. Поэтому, технически, все вышеперечисленное нельзя считать ошибкой, каким бы странным подобное поведение ни казалось мне и вам!

ПЕРЕМОТКА ВПЕРЕД, ЗАМЕДЛЕННОЕ И ОБРАТНОЕ ВОСПРОИЗВЕДЕНИЕ

В спецификации описан атрибут `playbackRate`. По умолчанию мы предполагаем, что `playbackRate` — это 1, то есть нормальное воспроизведение медиафайла со стандартной скоростью. Если мы увеличим значение этого атрибута, воспроизведение будет ускорено; если уменьшим — замедлено. При отрицательных значениях атрибута видео будет проигрываться в обратном направлении.

Пока не все браузеры поддерживают `playbackRate` (на данный момент только те, которые основаны на Webkit, а также IE9), так что если вам нужна поддержка перемотки вперед и назад, вы можете ее настроить. Для этого нужно будет внести изменения в `currentTime`:

```
function speedup(video, direction) {
    if (direction == undefined) direction = 1; // or -1 for
    // reverse

    if (video.playbackRate != undefined) {
        video.playbackRate = direction == 1 ? 2 : -2;
    } else { // do it manually
        video.setAttribute('data-playbackRate', setInterval
            ((function playbackRate () {
                video.currentTime += direction;

                return playbackRate; // allows us to run the
            // function once and setInterval
            })(), 500));
    }
}
```

```

function playnormal(video) {
    if (video.playbackRate != undefined) {
        video.playbackRate = 1;
    } else { // do it manually
        clearInterval(video.getAttribute('data-playbackRate'));
    }
}

```

Как видно из приведенного примера, если поддерживается `playbackRate`, вы можете оперировать и положительными и отрицательными числами, чтобы контролировать направление воспроизведения. Используя `playbackRate`, вы сможете не только проматывать видео вперед и назад, но и проигрывать его в любом направлении с любой скоростью, в том числе с замедленной (при помощи `video.playbackRate = 0.5`, то есть с замедлением нормальной скорости на 50%).

ПОЛНОЭКРАННОЕ ВИДЕО

В течение длительного времени спецификация запрещала полноэкранное видео. Тем не менее это, очевидно, полезная возможность, поэтому разработчики WebKit предложили собственное решение на базе `WebKitEnterFullscreen()`. API WebKit реализован так, что только пользователь может инициировать переход в полноэкранный режим; как и всплывающие окна, полноэкранный режим не включается сам по себе, а только после действия, такого как щелчок мышью. Единственная альтернатива этому специальному решению от WebKit — растягивание видео по размеру окна браузера. Так как в некоторых браузерах есть возможность разворачивания на полный экран, теоретически, можно просмотреть ваше любимое видео с Брюсом, исполняющим танец живота, в полноэкранном режиме. Однако для этого пользователю придется преодолеть целую полосу препятствий, а нам бы хотелось этого избежать.

В мае 2011 года разработчики WebKit объявили о реализации API для полноэкранного просмотра от Mozilla (<https://wiki.mozilla.org/Gecko:FullScreenAPI>). Этот API позволяет разворачивать на весь экран любой элемент (а не только `<video>`). Можно растягивать игры на базе `<canvas>` или видеовиджеты, встроенные в страницу с помощью `<iframe>`. Также можно добавлять сценарии, чтобы ввод любых символов с клавиатуры работал и в полноэкранном режиме. Это означает, что на базе API `<canvas>` с применением этих возможностей можно создать суперграндиозную игру, которая будет разворачиваться на весь экран, не теряя в поддержке клавиатуры.

Поскольку ребятам из Opera этот подход тоже нравится, наверняка мы в скором времени увидим нечто, напоминающее совместимость. Но до тех пор придется имитировать полноэкранный режим, задавая для элемента видео размеры, соответствующие размеру окна.

Возможности управления мультимедиа

Мы говорили о возможности управления элементом видео с клавиатуры, но как насчет заголовков и субтитров мультимедиа? В конце концов, для аудио или видео нет атрибута `alt`, как для ``. Контент между тегами, срабатывающий в случае ошибки, предназначен только для браузеров, которые не могут справиться с «родным» видео. Все это ничем не поможет

тем людям, которые не могут слышать звук или видеть изображение из-за проблем со здоровьем или по каким-либо иным причинам (например, в шумной обстановке или из-за того, что было необходимо сэкономить трафик), если браузеры корректно отображают медиа. Существует два способа присоединения к мультимедийному содержимому синхронизированной текстовой альтернативы (заголовков, субтитров и т. п.): внутриполосный и внеполосный. *Внутриполосный* метод (*in-band*) означает добавление текстового файла в контейнер мультимедиа; например, файл MP4 в действительности представляет собой контейнер для видео H.264 и аудио AAC и может содержать прочие файлы метаданных, включая субтитры. WebM — это контейнер (основанный на формате открытого стандарта Matroska Media Container) для видео VP8 и аудио Ogg Vorbis. В настоящее время WebM не поддерживает субтитры, а Google ожидает, когда рабочие группы определятся с форматом HTML5: «WHATWG/W3C RFC выпустит рекомендации относительно субтитров и других перекрывающих данных для элемента HTML5 `<video>` в ближайшем будущем. Планируется, что WebM будет отвечать этим рекомендациям». (Разумеется, даже если контейнер может содержать дополнительные метаданные, от медиапроигрывателя или браузера зависит, увидят ли пользователи эту информацию.)

Внеполосные (*out-of-band*) текстовые альтернативы не прячутся внутри контейнера, а содержатся в отдельном файле и связываются с медиафайлом через дочерний элемент `<track>`:

```
<video controls>
<source src=movie.webm>
<source src=movie.mp4>
<track src=english.vtt kind=captions srclang=en>
<track src=french.vtt kind=captions srclang=fr>
<p>Fallback content here with links to download video
- files</p>
</video>
```

В этом примере к видеофайлу привязываются два трэка с заголовками: на английском и на французском языках. В браузерах должен быть определенный механизм пользовательского интерфейса, позволяющий выбирать желаемый трек (также среди внутриполосных треков).

Элемент `<track>` не предполагает никакого определенного формата, но разработчики браузеров, вероятно, будут реализовывать новый формат WebVTT (ранее известный как WebSRT и основанный на формате SRT) (<http://www.whatwg.org/specs/web-apps/current-work/multipage/the-video-element.html#webvtt>).

Данный формат все еще разрабатывается группой WHATWG, которая также принимает во внимание отзывы действительно знающих людей из BBC, Netflix и Google (организаций, обладающих самым большим опытом в деле доставке пользователям видео через сервисы YouTube). Так как процесс пока не завершен, мы не можем заглянуть в детали формата, более того, он наверняка поменяется к тому моменту, как вы возьмете в руки эту книгу.

WebVTT — это всего лишь текстовый файл в кодировке UTF-8, в самой простой реализации выглядящий так:

WEBVTT

```
00:00:11.000 --> 00:00:13.000
Luftputefartøyet mitt er fullt av ål
```

Этот код накладывает заголовок «Luftputefartøyet mitt er fullt av ål» поверх видео через 11 секунд после начала и убирает его на отметке в 13 секунд (а не удерживает на экране в течение 13 секунд).

В настоящее время ни один браузер не поддерживает WebVTT или <track>, однако вы можете воспользоваться помощью парочки заполнителей. Джюлиен Виллеторт (@delphiki) написал Playr (<http://www.delphiki.com/html5/playr/>), легковесный сценарий, добавляющий поддержку этих возможностей во все браузеры, поддерживающие видео HTML5 (рис. 4.6).



Рис. 4.6. Реми декламирует 155-й сонет Шекспира, а субтитры на уэльском языке выводятся с помощью Playr

WebVTT также поддерживает полужирный, курсивные и цветной текст, вертикальное расположение текста для азиатских языков, написание справа налево для таких языков, как арабский и иврит, ruby-аннотации (см. главу 2) и перенос текста из местоположения по умолчанию (например, для того чтобы он не закрывал собой важные детали изображения) — но только если вам нужны все эти возможности.

Этот формат специально старались сделать как можно более простым, что крайне важно для обеспечения доступности содержимого: *если это сложно написать, то люди просто не будут делать этого*, и никакие API во всем мире не сделают видео доступным, если для него нет субтитров.

ПРИМЕЧАНИЕ

Утилита Скотта Уилсона VTT Caption Creator (<http://scottbw.wordpress.com/2011/06/28/creating-subtitles-and-audio-descriptions-with-html5-video/>) позволяет использовать авторские субтитры как автономный HTML-код или виджет W3C.

Стоит также отметить, что наличие простого текста не просто важно, а крайне важно для людей с физическими ограничениями. Текстовые записи читаются поисковыми

механизмами, что крайне радует оптимистичных сторонников поисковых систем. Кроме того, текст можно выделять, копировать, вставлять, менять размер шрифта, настраивать стили с помощью CSS, переводить автоматическими переводчиками, смешивать символы в кучу и творить прочие чудеса. Как сказал Шекспир в 155 сонете: «Если текст выделяется, публика восхищается».

СИНХРОНИЗАЦИЯ МЕДИАТРЕКОВ

HTML5 поддерживает не только добавление, но и синхронизацию разных медиатреков в одном элементе `<audio>` или `<video>`.

Если у вас есть несколько записей спортивного события, сделанных разными камерами с разных углов, то при выборе другого трека (или изменении скорости воспроизведения для просмотра деталей) пользователь ожидает, что все остальные дорожки синхронно изменятся. Таким образом, разные медиафайлы приходится группировать.

Это может стать огромным благом для технологий упрощенного доступа: представьте себе добавление дорожки с сурдопереводом, трека с описанием звуковой составляющей, дублированного аудиотрека и схожих дополнительных или альтернативных дорожек, связанных с основным аудио- и видеотреком.

MEDIAELEMENT.JS, КОРОЛЬ ЗАПОЛНИТЕЛЕЙ

MediaElement.js (<http://www.mediaelementjs.com>) — это встраиваемый модуль авторства Джона Даера (<http://j.hn>), веб-разработчика из даллаской теологической семинарии.

Создание плеера на базе HTML5 куда проще высшей наноматематики. Проблемы возникают при необходимости добавить поддержку видео в старые браузеры, не имеющие «родных» возможностей работы с мультимедиа, или же в браузеры, незнакомые с вашими кодеками.

В большинстве случаев для решения этой проблемы используется отдельный проигрыватель Flash Player. Однако и этот подход может привести к двум видам неприятностей. Во-первых, у вас на руках оказываются два совершенно разных пользовательских интерфейса воспроизведения (один в HTML5, а второй в Flash), которые необходимо оформлять и стилизовать независимо друг от друга. Во-вторых, вы не можете синхронизировать прочие элементы на странице с помощью медиасобытий HTML5, таких как `ended` и `timeupdate`.

MediaElement.js реализует совершенно другой подход. Вместо того чтобы предлагать в качестве альтернативного решения «голый» Flash-проигрыватель, он добавляет специальный проигрыватель, имитирующий весь API Media HTML5. Flash (или Silverlight, в зависимости от того, что установлено на компьютере пользователя) визуализирует медиаэлемент, а затем передает браузеру фальшивые события HTML5. Это означает, что благодаря MediaElement.js даже наш старый знакомый IE6 будет вести себя так, словно с рождения поддерживает `<video>` и `<audio>`. Получается, что вместо поиска альтернативного решения мы дерзко движемся вперед, опережая время.

В мобильных системах (Android, iOS, WP7) MediaElement.js полагается на пользовательский интерфейс операционной системы. В настольных системах этот сценарий поддерживает все современные браузеры со встроенной функциональностью HTML5, а также выводит на новый уровень устаревшие браузеры. Кроме того, он «впрыскивает» вакцину поддержки неподдерживаемых кодеков, используя для этого встраиваемые модули. Это обеспечивает возможность воспроизведения MP4, Ogg и WebM, а также WMV, и FLV, и MP3.

MediaElement.js также поддерживает многоязыковые субтитры и навигацию по главам с помощью элементов `<track>` и WebVTT. Плагины для Wordpress, Drupal и BlogEngine.net упрощают развертывание и использование видео на этих платформах до уровня «дважды два четыре».

Достойный соперник, с честью занимающий второе место, — это проигрыватель LeanBack (http://dev.mennerich.name/showroom/html5_video/) с заполнением на базе WebVTT, независимый от внешних библиотек и предлагающий превосходную поддержку клавиатуры.

Того же результата можно добиться с помощью JavaScript или же явно объявить атрибут `mediagroup` для элемента `<audio>` или `<video>`:

```
<div>
  <video src="movie.webm" autoplay controls
    - mediagroup=movie></video>
  <video src="signing.webm" autoplay
    - mediagroup=movie></video>
</div>
```

ПРИМЕЧАНИЕ

25 августа 2011 года Федеральная комиссия связи США выпустила FCC 11-126, требующий от определенных телевизионных и видеосетей предоставлять для определенных телевизионных программ описание видео.

Добавление описания ключевых визуальных элементов во время естественных пауз диалога программы — это прекрасная возможность применить `mediagroup` и связанный API.

Это все великолепно, но очень ново, поэтому мы не будем углубляться в детали: спецификация постоянно меняется, а реализаций пока нет.

ВИДЕОКОНФЕРЕНЦИИ, ДОПОЛНЕННАЯ РЕАЛЬНОСТЬ

Как мы уже упоминали ранее, когда-то веб-страницы могли обращаться к камере или микрофону устройства только через встраиваемые модули. HTML5 предоставляет способ обращаться к этим устройствам прямо из JavaScript, применяя для этого API под названием `getUserMedia`. (В устаревших ресурсах вы можете встретить упоминание элемента `<device>`. Он не был описан в спецификации, а данную концепцию вынесли в чистый API.)

Экспериментальная сборка Opera Mobile для Android дает первое представление о том, каким станет Сеть, когда эта возможность получит широкое распространение. Здесь камера присоединяется к элементу `<video>` с помощью JavaScript: сначала выполняется распознавание поддержки `getUserMedia`, а затем, в случае положительного ответа, поток, поступающий с камеры, устанавливается в качестве значения `src` элемента `<video>`:

```
<!DOCTYPE html>
<h1>Простое демо для тестирования веб-камеры</h1>
<video autoplay></video>
<script type="text/javascript">
var video = document.getElementsByTagName('video')[0],
    heading = document.getElementsByTagName('h1')[0];
```

```

if(navigator.getUserMedia) {
    navigator.getUserMedia('video', successCallback,
    ~ errorCallback);
    function successCallback( stream ) {
        video.src = stream;
    }
    function errorCallback( error ) {
        heading.textContent =
            "Произошла ошибка: [КОД " + error.code + "]";
    }
} else {
    heading.textContent =
        "Потоковая передача с веб-камеры в этом
        ~ браузере не поддерживается!";
}
</script>

```

Сделав это, вы можете манипулировать видеосодержимым по своему усмотрению. Рич Тиббетт написал демонстрационную программу, которая копирует видео на холст (обеспечивая, таким образом, доступ к пиксельным данным), изучает пиксельную информацию, распознает лицо и пририсовывает ему усы — и все это на JavaScript (рис. 4.7).



Рис. 4.7. Реми Шарп и его волшебные HTML5-усы (фотография Джуллии Гослинг)

Норвежский разработчик Тригв Ли создал демонстрационную программу на базе `getUserMedia`, которая отправляет изображения с телефона Android с экспериментальной сборкой Opera Mobile через веб-сокеты (см. главу 10) прямо на настольный компьютер. Исходный код и видео, иллюстрирующее процесс, вы найдете на <https://github.com/trygve-lie/demos-html5-realtime>.

Очевидно, что открытый доступ веб-сайтов к веб-камерам может привести к нарушению тайны личной жизни, поэтому пользователи должны будут самостоятельно включать дан-

ную возможность, так же, как это делается для геолокации. Тем не менее это не техническая проблема — этим должен заниматься пользовательский интерфейс.

Зарываясь в концепцию еще глубже, стоит упомянуть API Peer-to-Peer, который в данный момент разрабатывается для HTML. Он позволит привязывать камеру и микрофон устройства к элементам `<video>` и `<audio>` в чужом браузере, что означает возможность проведения видеоконференций.

В мае 2011 года Google представила WebRTC, основанную на спецификации HTML5 открытую технологию передачи голоса и видео в Сети. WebRTC использует VP8 (видеокодек из WebM) и два аудиокодека, оптимизированных для обработки голоса и устранения шума и эха. Узкополосный голосовой кодек называется iLBC, а адаптивный широкополосный кодек носит название iSAC (см. <http://sites.google.com/site/webrtc/>).

Как говорится на веб-сайте проекта, «Надеемся скоро увидеть поддержку WebRTC в Firefox, Opera и Chrome!»

ЗАКЛЮЧЕНИЕ

Вы узнали о том, как HTML5 предоставляет вам первую адекватную альтернативу сторонним плагинам. Из-за отсутствия совместимых кодеков на данный момент разместить видео на странице и заставить его работать в любом браузере гораздо сложнее, чем использовать плагины.

С другой стороны, видео и аудио теперь считаются стандартными элементами, которые поддерживаются браузерами (а не загадочными плагинами), для них предлагается эффективный API и их очень просто контролировать при помощи JavaScript. Разработчики могут с легкостью писать собственные элементы управления, имея в арсенале только поверхностные знания о веб-стандартах, или манипулировать видео по своему желанию при помощи нескольких строк кода. Чтобы подстраховаться в случае, если браузер не сможет воспроизвести видео, мы рекомендуем добавить ссылки для того, чтобы загрузить файлы вне элемента `<video>`.

Уже существует несколько готовых доступных скриптов, которые позволяют вам легко оптимизировать взаимодействие ваших страниц с HTML5, и у вас нет необходимости писать весь этот код самим. jPlayer (<http://www.jplayer.org>) — это аудиоплеер jQuery audio со свободной лицензией, который «опускается» до Flash в устаревших браузерах, может быть изменен при помощи CSS и расширяется для добавления возможности работы с плейлистами. Что касается видео, вы уже познакомились с Playr, MediaElement.js и LeanBack Player, которые я считаю наилучшими, однако существует и множество других проигрывателей. Полезную диаграмму сравнения видеоплееров вы найдете на странице <http://praegnanz.de/html5video/>.

Организация доступа к видео при помощи JavaScript — это не просто написание новых плееров. В следующей главе вы узнаете, как манипулировать стандартными элементами медиа, что обеспечит несколько замечательных эффектов — по крайней мере, вы научитесь бросаться головами через весь экран, а что может быть удивительнее?

Глава 5

Холст

Реми Шарп



Если элемент `<video>` — это лицо HTML5, то элемент `<canvas>` — определенно его изюминка. Это одна из самых объемных частей спецификации HTML5. Более того, API холста, *контекст рисования 2D*, вынесен в отдельный документ, хотя сам элемент `<canvas>` все еще является частью официальной спецификации HTML5.

Элемент `<canvas>` предоставляет интерфейс прикладного программирования для двумерного рисования: линии, заливки, изображения, текст и т. п. Этот API уже используется во множестве ситуаций, включая (интерактивные) фоны веб-сайтов, элементы навигации, графические инструменты, полноценные приложения, игры и симуляторы. Кто бы мог предположить, что разработчики будут вдохновляться сделанными на базе холста играми про Супер Марио! Представить подобную функциональность несложно, если вспомнить версию MS Paint, которая входила в состав Windows 95. В действительности, Paint можно полностью воспроизвести с помощью элемента `<canvas>`, как показано на рис. 5.1. По всему Интернету появляются приложения, грозящие превратиться в полнофункциональные программы векторной графики (рис. 5.2) (хотя, возможно, лучшим выбором был бы формат Scalable Vector Graphics (SVG) — см. врезку «Холст и SVG: в каком случае что использовать» далее в этой главе). Так как эти приложения базируются на технологии Open Web, они работают в браузерах на самых разных устройствах. Приложение Harmony, окно которого показано на рис. 5.3, работает даже на мобильных устройствах, таких как телефоны iPhone и Android.

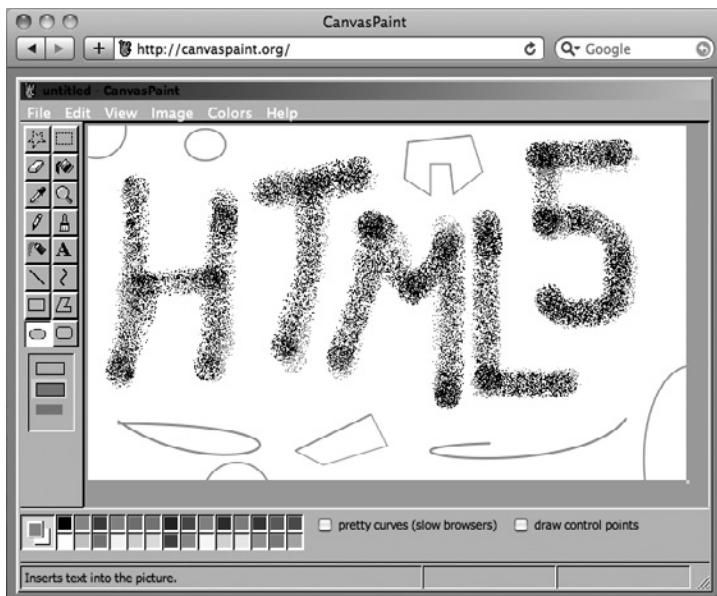


Рис. 5.1. Приложение MS Paint, воспроизведенное при помощи элемента `<canvas>`

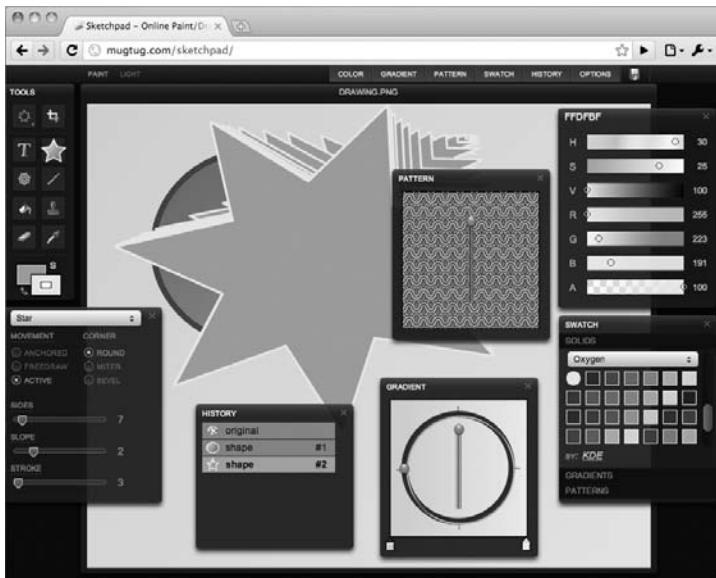


Рис. 5.2. На основе элемента <canvas> создаются и более сложные графические приложения



Рис. 5.3. Демонстрационная версия приложения Harmony в неизменном состоянии работает даже в мобильных браузерах

API 2D настолько обширен, что я предвижу появление целых томов, посвященных исключительно этому интерфейсу. Поскольку в моем распоряжении лишь одна глава, я коснусь только основ. Однако я также собираюсь продемонстрировать несколько интересных вещиц, таких как захват кадров видео и обработка отдельных пикселов изображения внутри холста. Я даже покажу вам, как экспорттировать файлы изображений. Также мы с вами создадим анимированное изображение, пробудив ностальгию по эпохе программирования на BASIC.

САМОЕ ГЛАВНОЕ О ХОЛСТЕ

«Hello world» любого эксперимента с холстом — это добавление элемента `<canvas>` на страницу. По умолчанию холст невидим, а его размер — 300 пикселов в ширину и 150 пикселов в высоту.

```
<!DOCTYPE html>
<title>canvas hello world</title>
<canvas></canvas>
```

Теперь, когда элемент холста на экране, чтобы начать рисовать в контексте 2D, нам понадобится JavaScript:

```
var ctx = document.querySelector('canvas').
  getContext('2d');
```

Добавив контекст, мы получили доступ ко всей функциональности API и можем рисовать в свое удовольствие. Добавьте на холст простую фигуру (рис. 5.4):

```
ctx.fillRect(10, 20, 50, 50);
```

`fillRect` принимает четыре аргумента: координаты `x` и `y`, высоту и ширину. Координаты `x` и `y` отсчитываются от верхнего левого угла прямоугольника. Как показано на рис. 5.4, цвет фигуры по умолчанию — черный. Давайте добавим цвет и нарисуем вокруг прямоугольника контур, чтобы получить холст, как на рис. 5.5:

```
ctx.fillStyle = 'rgb(0, 255, 0)';
ctx.fillRect(10, 20, 50, 50); // залитый прямоугольник
ctx.strokeStyle = 'rgb(0, 182, 0)';
ctx.lineWidth = 5;
ctx.strokeRect(9, 19, 52, 52); // обводка
```



Рис. 5.4. Залитый цветом прямоугольник с настройками холста по умолчанию



Рис. 5.5. Используем стили заливки и прямоугольную обводку

В предыдущем фрагменте кода мы рисуем две фигуры: первую с помощью `fillRect`, а вторую с помощью `strokeRect`, предварительно задавая цвет и стиль контекста 2D. Настраивать внешний вид фигур необходимо до вывода на экран; в противном случае

будет использован стандартный черный цвет. Помимо цветов CSS (например, RGB, hex, RGBA и т. д.), методы `fillStyle` и `strokeStyle` поддерживают градиенты и узоры, сгенерированные в API 2D.

КАК НАСЧЕТ ПОДДЕРЖКИ БРАУЗЕРАМИ?

Поддержка элемента `<canvas>` в браузерах реализована достаточно хорошо; холст поддерживается в последних версиях четырех из пяти наиболее распространенных браузеров (и, если уж на то пошло, в старых версиях холст также работает неплохо). Чаще всего, конечно, задают вопрос, работает ли это в Internet Explorer.

В версиях IE, которые элемент `<canvas>` не поддерживает (IE8 и более ранние), его можно имитировать с помощью пары хитростей. Например, применить решение FlashCanvas — оно выглядит наиболее многообещающим. В качестве резервного средства оно полагается на Flash, но должно успешно считывать весь код, относящийся к холсту, и преобразовывать его в графический слой Flash: <http://flashcanvas.net>.

Существует еще похожий метод на базе Silverlight и библиотеки под названием html5canvas (<http://blogs.msdn.com/delay/archive/2009/08/24/using-one-platform-to-build-another-html-5-s-canvas-tag-implemented-using-silverlight.aspx>). Также можно воспользоваться библиотекой excanvas (<http://code.google.com/p/explorercanvas/>), которая преобразует API холста в Microsoft VML.

Эти две библиотеки охватывают не всю функциональность API 2D, только большинство наиболее часто используемых методов. В сети есть примеры, демонстрирующие реализацию практических задач с помощью этих библиотек. Веб-сообщество считает наиболее приемлемым метод заполнения с помощью FlashCanvas. Мне чудится тонкая ирония в том, что нам (снова) приходится полагаться на Flash для реализации технологии, которая, по идеи, должна его заменить. Но что делать — такова Сеть!

Стоит также отметить и хорошенько запомнить, что перечисленные методы заполнения не сравнятся в производительности с «родным» элементом `<canvas>`. Не могу подтвердить свои слова научными диаграммами, но мне кажется, что FlashCanvas должен работать быстрее остальных. Тем не менее «лучше остальных» не означает такой же уровень производительности, особенно если рассматривать браузер с аппаратным ускорением визуализации холста, таким как реализован в IE9.

ПРИМЕЧАНИЕ

`querySelector` и `querySelectorAll` — это новые методы объектной модели документов (Document Object Model, DOM), которые принимают селектор CSS и возвращают соответствующие элементы. Доступный во всех новейших браузерах, `querySelector` возвращает первый найденный узел DOM, тогда как `querySelectorAll` возвращает объект NodeList. Пройдя по содержимому этого объекта, вы можете выбрать необходимые элементы.

РИСОВАНИЕ С ГРАДИЕНТАМИ И УЗОРАМИ

Используя объект контекста, можно создать линейный градиент, радиальный градиент или заполнение узором, и с помощью метода `fillStyle` сделать его стилем заполнения фигур. Линейные и радиальные градиенты работают так же, как градиенты CSS — вы указываете только начальную точку и ограничивающие цвета.

Создавая узор, вы указываете источник изображения, а затем определяете, каким образом рисунок должен повторяться — аналогично созданию фонового изображения в CSS. Но что делает метод `createPattern` действительно интересным, так это то, что в качестве источника может применяться изображение, другой холст или элемент `<video>` (хотя на момент написания этой главы использование видео в качестве источника узора еще не реализовано). Определить простой градиент очень просто — вы сделаете это быстрее, чем откроете Photoshop:

```
var canvas = document.querySelector('canvas'),
    ctx = canvas.getContext('2d'),
    gradient = ctx.createLinearGradient(0, 0, 0, canvas.
        height);
gradient.addColorStop(0, '#fff');
gradient.addColorStop(1, '#000');
ctx.fillStyle = gradient;
ctx.fillRect(0, 0, canvas.width, canvas.height);
```

В предыдущем коде объект контекста 2D используется для создания объекта линейного градиента, для которого затем определяются ограничивающие цвета. В качестве аргументов передаются начальная точка градиента (x_1 и y_1) и конечная точка градиента (x_2 и y_2). В этом примере я создаю градиент, начинающийся в верхнем левом углу и заканчивающийся в нижнем левом углу. Таким образом, у меня получается вертикальный градиент (рис. 5.6).

Радиальный градиент создается аналогично. Единственное отличие в том, что в дополнение к координатам методу `createRadialGradient` нужно также передать радиус.

```
var canvas = document.querySelector('canvas'),
    ctx = canvas.getContext('2d'),
    gradient = ctx.createRadialGradient(canvas.width/2,
        canvas.height/2, 0,
        canvas.width/2, canvas.height/2, 150);
gradient.addColorStop(0, '#fff');
gradient.addColorStop(1, '#000');
ctx.fillStyle = gradient;
ctx.fillRect(0, 0, canvas.width, canvas.height);
```

Единственное отличие в том, градиент какого вида вы создаете. В этом примере я установил начальную точку градиента в центр холста, а начальный радиус задал равным нулю. Конечный радиус равен 150, однако обратите внимание, что центр конечной окружности градиента находится там же, где центр начальной окружности: `canvas.width/2, canvas.height/2`. Вот почему мой градиент выглядит так аккуратно (рис. 5.7).



Рис. 5.6. Вертикальный градиент на элементе холста



Рис. 5.7. Начальная и конечная точки радиального градиента совпадают, но конечный радиус намного больше, благодаря чему создается гладкий рисунок в форме круга

С узорами еще проще. Вам нужен только источник; элемент источника передается методу `createPattern`, а результат используется в методе `fillStyle`. Помнить нужно только о том, что для правильного захвата изображений и видео элемент источника должен сначала полностью загрузиться.

Чтобы создать эффект, показанный на рис. 5.8 (копии изображений, уложенные плиткой по всей площади холста), сначала растяните холст по размеру окна. Затем динамически создайте изображение и, когда сработает событие загрузки, примените изображение в качестве источника повторяющегося узора:

```
var canvas = document.querySelector('canvas'),
    img = document.createElement('img'),
    ctx = canvas.getContext('2d');
canvas.width = window.innerWidth;
canvas.height = window.innerHeight;
img.onload = function () {
    ctx.fillStyle = ctx.createPattern(this, 'repeat');
    ctx.fillRect(0, 0, canvas.width, canvas.height);
};
img.src = 'remysharp_avatar.jpg';
```

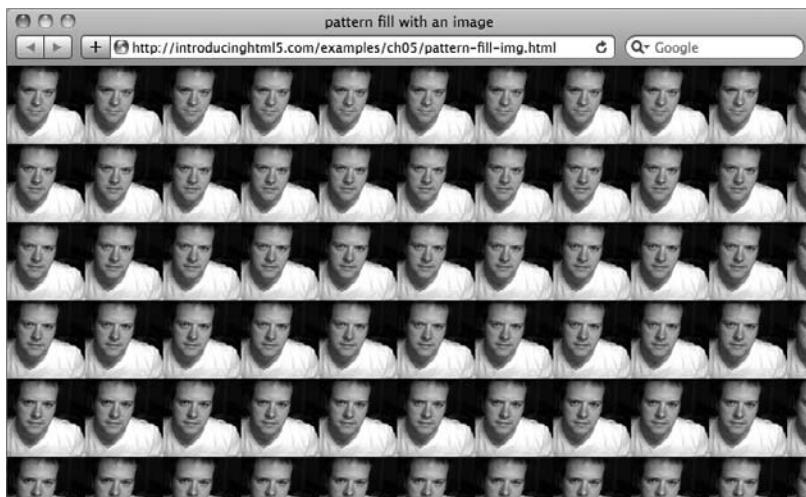


Рис. 5.8. Укладываем изображение на холсте с помощью метода `createPattern`

ОТ ГРАДУСОВ К РАДИАНАМ

Все методы `arc`, `bezier` и `quadratic` работают с радианами, поэтому тем из вас, кто привык делать вычисления в градусах, придется преобразовывать их в радианы. Вот как это сделать на JavaScript:

```
var radians = degrees * Math.PI / 180;
```

Во время рисования вы будете часто передавать методам значение в 360 градусов, что соответствует `Math.PI * 2`, и 180 градусов, то есть `Math.PI`.

В этом примере я создаю изображение на лету с помощью `document.createElement`, и только когда срабатывает событие `onload`, я продолжаю строить узор для заливки. Перед тем как что-то делать с изображением, нужно обязательно дождаться завершения загрузки данных.

Когда изображение загружено, я могу определить стиль заполнения `fillStyle` с помощью метода `createPattern`. В вызове метода — `createPattern(this, 'repeat')` — `this` относится к изображению, которое заставило сработать событие загрузки, однако в качестве источника можно указать, например, любой другой холст. Стока '`repeat`' соответствует синтаксису `background-repeat` в CSS; `repeat-x`, `repeat-y` и `no-repeat` тоже будут работать в этом методе.

Если вы измените размер элемента `canvas` в коде CSS, то при этом он просто растянется. Фактически с пикселями ничего не случится — задействуется только узел DOM, как видно на рис. 5.9. Если нарисовать что-то на холсте и поменять свойства `height` и `width` элемента `canvas` (скажем, вас не устраивает размер по умолчанию 300×150 пикселов), содержимое холста будет стерто, а состояние стилей заливки, обводки, толщины линии и т. п. сбросится. Фактически это обнуление холста — трюк или проблема, в зависимости от того, как на это посмотреть.

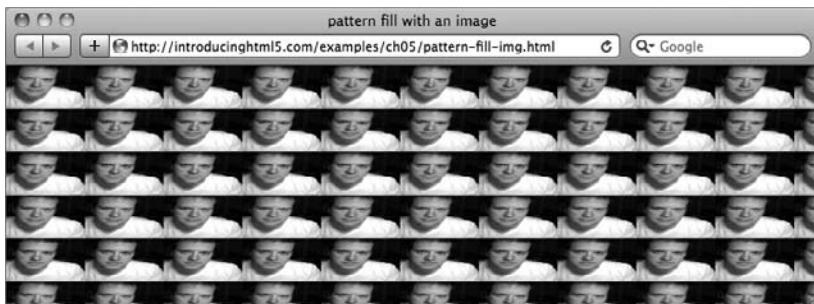


Рис. 5.9. Если растянуть холст после того, как рисование закончилось, то содержимое холста тоже будет растянуто

РИСОВАНИЕ ПУТЕЙ

Внутри API 2D есть API `path` — API путей, позволяющий перемещаться по холсту и рисовать линии и фигуры. На рис. 5.10 показан человечек, нарисованный с помощью API путей. Я не собираюсь во всех подробностях рассказывать вам о коде для рисования этого шедевра, остановлюсь только на ключевых моментах и поговорю о методах. Для того чтобы нарисовать человечка типа «палка-палка-огуречик», нужно указать координаты `x` и `y` на холсте для каждой составляющей его фигуры и палочки. Например, следующий код рисует голову:

```
ctx.beginPath();
ctx.arc(100, 50, 30, 0, Math.PI*2, true); // голова
ctx.fill();
```

Под головой я имею в виду круг со сплошной заливкой. Я задал координаты `x` и `y`, равные 100 и 50 соответственно, и радиус 30 пикселов. Следующие два аргумента — это начальная

и конечная точки дуги в радианах. В данном примере мне нужен полный круг, поэтому я начинаю рисовать его с нуля и заканчиваю на значении `Math.PI*2`, что соответствует 360 градусам. Наконец, последний аргумент — это направление рисования дуги: по часовой стрелке или против часовой стрелки. В данном случае это неважно, но аргумент обязательный, поэтому какое-то значение все равно должно присутствовать.

Итак, голова нарисована, и настал черед лица. Улыбка и глаза будут красными. Перед тем как переходить к лицу, нужно снова использовать метод `beginPath`. На рис. 5.11 показано, что получится, если забыть о нем. Предыдущая дуга сольется с новым путем, с помощью которого я рисую лицо. Это неправильно — для рта мне нужна новая отдельная дуга. Убрать отрезок, соединяющий очертания головы со ртом, можно при помощи `moveTo` — это как поднять карандаш над холстом и начать рисовать в другом месте, — однако при этом вокруг головы все равно появится цветной контур, который мне не нужен.

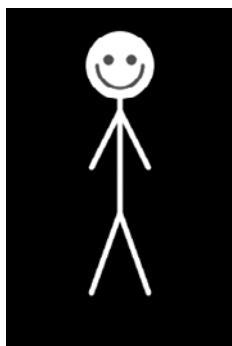


Рис. 5.10. Человечек, нарисованный с помощью API путей

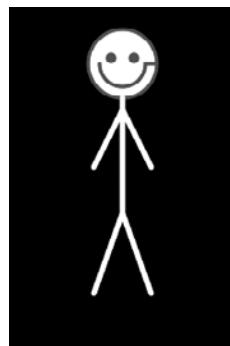


Рис. 5.11. Пример ошибки на рисунке, вызванной соединением разных путей

```
ctx.beginPath();
// рисую улыбку
ctx.strokeStyle = '#c00';
ctx.lineWidth = 3;
ctx.arc(100, 50, 20, 0, Math.PI, false);
ctx.stroke();
```

Предыдущий код рисует симпатичную полуокружность новым цветом и новой ширины. Для головы я использовал метод `fill`, но для лица мне нужен метод `stroke`, иначе вместо линии получится залитая фигура. Теперь перейдем к глазам:

```
ctx.beginPath();
ctx.fillStyle = '#c00';
// начало левого глаза
ctx.arc(90, 45, 3, 0, Math.PI*2, true);
ctx.fill();
ctx.moveTo(113, 45);
// рисую правый глаз
ctx.arc(110, 45, 3, 0, Math.PI*2, true);
ctx.fill();
ctx.stroke(); // более толстые линии для глаз
```

Я снова начал новый путь, то есть перешел к дуге левого глаза, не используя `moveTo` (так же, как перешел от головы к улыбке). Однако после того как я нарисовал и залил первую дугу (получив в итоге круг), я *поднял* карандаш методом `moveTo(113, 45)` и перешел к рисованию правого глаза. Обратите внимание, что в методе `moveTo` координата `x` равна 113 — это координата `x` второй дуги плюс ее радиус. Таким образом, начальная точка второй дуги оказалась в точности там, куда я перенес карандаш. В конце с помощью метода `stroke` я делаю контуры глаз немного толще.

Дальше в коде я переношу начальную точку в разные места на холсте, рисую линии и в итоге получаю этого прекрасного человечка-огуречка.

Есть и другие методы рисования путей, которые в этой главе мы рассматривать не будем. Однако вы можете использовать `quadraticCurveTo`, `bezierCurveTo`, `arcTo`, `rect`, `clip` и `isPointInPath` для более точного рисования линий и фигур.

ХОЛСТ И SVG: В КАКОМ СЛУЧАЕ ЧТО ИСПОЛЬЗОВАТЬ

Оба API для рисования — Canvas и SVG — прекрасны, но предназначены для разных целей. Оцените их преимущества и недостатки, чтобы выбрать инструмент, наиболее подходящий для вашей задачи. SVG — это API с отображением с задержкой, а 2D canvas — API с немедленным отображением.

SVG основывается на векторной графике, поэтому намного лучше масштабируется; холст дает растровое изображение, поэтому не масштабируется, просто сжимаются и растягиваются. SVG строит дерево, представляющее текущее состояние всех объектов на экране (аналогично дереву DOM, описывающему текущий документ). Именно поэтому он называется API с отображением с задержкой. Такое дерево предоставляет отличную возможность для построения интерактивных приложений. Привязываясь к различным объектам дерева, вы прослушиваете щелчки или ожидаете события прикосновения, например, для распознавания действий пользователя в играх. Код SVG можно писать вручную, так как это всего лишь обычный XML — а все новейшие браузеры (за исключением, как ни странно, браузеров Android WebKit) полностью поддерживают данную технологию, как в случае внешней привязки файлов с кодом, так и внутри HTML5. Но если вам не слишком по нраву сражаться с XML, настольные инструменты, такие как Adobe Illustrator и Inkscape, позволяют импортировать и экспортить рисунки SVG, что значительно упрощает жизнь.

Если вы еще не до конца уверены, хороший ли это инструмент, попробуйте поэкспериментировать с Raphaël, библиотекой JavaScript Дмитрия Барановского (<http://raphaeljs.com>). В ней используется только SVG, и она способна создавать очень впечатляющие рисунки и анимацию.

Холст, по сути, представляет собой набор пикселов и хорошо подходит для создания анимированных изображений и приложений, сконцентрированных вокруг JavaScript. По сравнению с SVG, это более низкоуровневый API, то есть он лучше подходит для ситуаций, когда движения мыши отслеживать не требуется. Так как дерева состояния холста здесь нет, невозможно привязать обработчик событий к объектам на холсте — приходится вычислять позицию, где произошло событие мыши, и сохранять все координаты нарисованных объектов в памяти. Но поскольку холст — очень JavaScript-центричный интерфейс, события клавиатуры можно обрабатывать на уровне документа. Наконец, при работе с холстом вы оперируете отдельными пикселями, что прекрасно иллюстрируют снимки экрана в начале главы, следовательно, API canvas превосходно подходит для пиксельного рисования.

У каждой из этих технологий есть сильные и слабые стороны. Ваша задача как разработчика — проанализировать требования создаваемого вами приложения и выбрать правильный инструмент. Удачи!

ИСПОЛЬЗОВАНИЕ ТРАНСФОРМЕРОВ: ЗАМАСКИРОВАННЫЕ ПИКСЕЛЫ

Можно перемещать карандаш по холсту с помощью таких методов, как `moveTo`, можно рисовать формы и линии, но у API `canvas` есть и еще одна интересная возможность — благодаря трансформациям он позволяет менять холст *под* карандашом.

Методы трансформации включают вращение, масштабирование, преобразжение и трансляцию (все они схожи со своими эквивалентами в CSS).

На рис. 5.12 вы видите нарисованную спираль; моя цель — заставить ее вращаться вокруг своей оси, создавая псевдогипнотический эффект. Ради простоты кода мне не хотелось бы менять функцию рисования спирали, положение спирали на экране, начальные точки или какие-либо другие характеристики. Для того чтобы сделать код понятным и управляемым, я буду менять не спираль, а холст. Спираль будет оставаться неизменной, а холст на каждом шаге будет поворачиваться на небольшой угол.

Метод `rotate` поворачивает холст вокруг координаты $(0, 0)$, то есть по умолчанию — вокруг верхнего левого угла. Меня это не устраивает, потому что при таком раскладе спираль будет уезжать с экрана, словно она находится на конце маятника. У меня холст должен вращаться вокруг центра спирали, которую я помешу в центр холста. Следовательно, я должен вращать холст вокруг его центра.

Поможет мне в этом метод `translate`. Он перемещает координату $(0, 0)$ в новое место. На рис. 5.13 вы видите точку и координаты, которые я передавал методу `translate`. При каждом выполнении `translate` превращает указанные координаты в точку $(0, 0)$. Обратите внимание, что этот метод не поворачивает и не перемещает холст, хотя зрителю может показаться именно так; он изменяет базовую систему координат, на которой основываются все функции рисования.

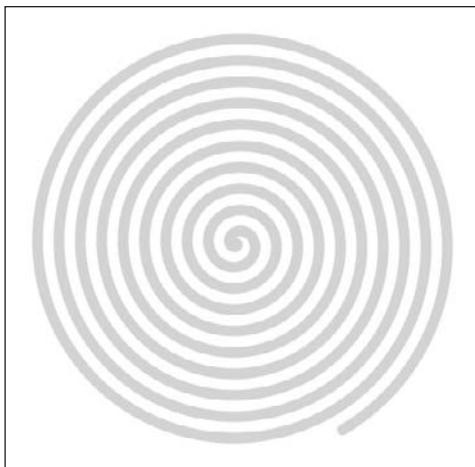


Рис. 5.12. Анимированная спираль вращается, вращается и никак не остановится

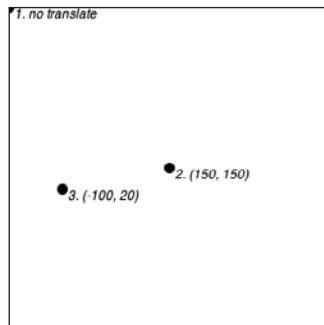


Рис. 5.13. Пример перемещения начала координат методом `translate`

Теперь, чтобы реализовать задуманное, я должен инициализировать холст при помощи метода `translate`, а затем использовать `setInterval` для перерисовки спирали (обратите

внимание, что `drawSpiral` написана мной; это не какая-то стандартная функция, которая строит путь для спирали и визуализирует ее на холсте с помощью последовательности вызовов `stroke`):

```
ctx.translate(ctx.canvas.width/2,
- ctx.canvas.height/2);
drawSpiral(); // сложная спиральная математика

setInterval(function () {
    ctx.clearRect(-ctx.canvas.width/2,
    -ctx.canvas.height/2,
        ctx.canvas.width,
        - ctx.canvas.height);
    ctx.rotate(Math.PI / 180 * 0.5) // 1/2
    - градуса
    drawSpiral();
}, 10);
```

Единственная сложность, с которой мне приходится сталкиваться, — это очистка холста. В обычной ситуации я бы просто применял `clearRect(0, 0, width, height)`, но так как метод `translate` перенес точку `(0, 0)` в центр экрана, мне приходится вручную указывать верхний левый угол, как видно в предыдущем фрагменте кода.

ЗАХВАТ ИЗОБРАЖЕНИЙ

Помимо рисования линий и форм, вы также можете копировать изображения — фотографии, видео и другие элементы холста — из других источников. Я уже демонстрировал, как использовать изображение в качестве источника для заливки `createPattern`. Но можно и просто поместить изображение на холст. Или не просто, а кадрировав его или применив другие корректировки.

```
var ctx = document.getElementById('mycanvas').
getContext('2d'),
img = new Image();

img.onload = function () {
    ctx.canvas.height = 500;
    ctx.canvas.width = 500;
    ctx.drawImage(this, 10, 10, 100, 100, 0, 0, 500, 500);
};
img.src = 'bruce-and-remy-promo-pics.jpg';
```

СОВЕТ

Все контексты рисования 2D включают обратную ссылку на элемент холста, на котором они рисуют объекты. Это означает, что вам не приходится передавать функциям две переменные, можно просто передать контекст и получить обратную ссылку на элемент холста, а затем поменять высоту, ширину или получить URL данных.

Представленный выше код — это простой пример динамического создания изображения налету. Как только оно загружается, я могу вывести его часть на холст. Как вы узнаете через мгновение, есть несколько способов применения метода `drawImage`; здесь я вырезал квадрат со стороной 10 пикселов, увеличил его до квадрата со стороной 100 пикселов и растянул на квадратный холст со стороной 500 пикселов.

Захват изображения из элемента `<video>` предлагает несколько интересных возможностей. В сети есть множество демонстрационных проектов, включая динамическую вставку содержимого в видео, замену зеленого фона и распознавание лиц — все это делается с помощью холста и видео и программируется на знакомом JavaScript.

Захват и рисование происходят в методе `drawImage`, которому требуется ссылка на источник (изображение, видео или холст), целевые координаты (координаты точки, обозначающей верхний левый угол изображения, которое будет нарисовано на холсте) и несколько вспомогательных аргументов для обрезки и масштабирования изображения:

- `drawImage(image, dx, dy)`
- `drawImage(image, dx, dy, dw, dh)`
- `drawImage(image, sx, sy, sw, sh, dx, dy, dw, dh)`

Здесь координаты, начинающиеся с `d`, — это координаты итогового изображения, а `s`-координаты обозначают положение источника. Например, я хочу взять синергетические видео Брюса из четвертой главы и соорудить наверху своей веб-страницы ленту из картинок, на которых Брюс колотит банан. Для этого мне понадобится кадрировать и масштабировать видео с помощью метода `drawImage`.

Вот какие компоненты мне потребуются:

- холст, размещенный наверху страницы;
- скрытый элемент `<video>`, который будет воспроизводить видеоклип о синергизме;
- способ, как закольцевать видео;
- метод для захвата содержимого видео и переноса его на холст.

Элемент `video` должен быть скрыт, так как я не хочу, чтобы посетители видели источник, из которого я захватываю изображения для холста. Мне нужно просто незаметно копировать кадры и переносить их на холст.

Меня интересует исключительно фрагмент, где Брюс разбивает банан деревянным молотком (это часть между отметками времени 0:49 и 0:52). Следовательно, я должен сообщить элементу `video`, что воспроизводиться должна только эта часть. Атрибутов содержимого, которые позволили бы это сделать, нет, поэтому я просто присвою `currentTime` значение 49, то есть перенесу время начала на сорок девятую секунду. Затем в событии `timeupdate`, если текущее время превысит 52 секунды, я снова присвою `currentTime` значение 49. Таким образом, воспроизводиться будет только содержимое между 49 и 52 секундами. Поскольку некоторые браузеры задерживают данные, а свойство `video.seekable` не поддерживает, в этом примере я собираюсь использовать таймер:

```
var jumpTimer = setInterval(function () {
  try {
    // если данные недоступны, установка
    // currentTime вернет ошибку
```

```

        video.currentTime = start;
        clearInterval(jumpTimer);
        video.play();
    } catch (e) {}
}, 100);

video.addEventListener('timeupdate', function () {
    if (this.currentTime > 52)
        this.currentTime = 49;
}, false);

```

В предыдущем коде я пытаюсь установить значение `video.currentTime`; если это делается, пока видеоданные не готовы, JavaScript возвращает ошибку. Если возвращается ошибка, код не доходит до `clearInterval`, однако, если он исполняется успешно, `setInterval` очищается, и видео воспроизводится.

Теперь, когда закольцованное видео загружено, можно начинать захватывать кадры из элемента `video`. Для рисования холста я *мог бы* использовать событие `timeupdate`, но мне прекрасно известно, что результат будет гораздо хуже того, который даст рисование согласно собственному таймеру. Возможно, причина кроется в том, что браузер тратит все силы на сложную работу по визуализации элемента `<video>`. Разделив обязанности, мы дадим браузеру возможность вздохнуть свободно.

После того как событие `loadeddata` сработает, показывая, что видео загружено, я инициализирую холст, ширина которого будет совпадать с шириной окна (в противном случае изображение будет выглядеть растянутым, как вы видели на рис. 5.9). Затем я выключаю звук видеофрагмента (для того чтобы эффект не был слишком навязчивым!) и определяю, какая из сторон изображения короче. Это понадобится для кадрирования — в итоге я хочу получить квадратное изображение и выложить холст его копиями как плиткой:

```

video.addEventListener('loadeddata', function () {
    var size = 78; // размер эскиза
    canvas.width = window.innerWidth;
    video.volume = 0;
    shortestEdge = video.videoHeight > video.videoWidth ?
        video.videoWidth :
        video.videoHeight;

    // запускаю цикл рисования
    setInterval(function () {
        for (var i = 0, w = canvas.width; i < w; i += size) {
            // список аргументов разбит на несколько
            // строк
            ctx.drawImage(
                video,
                (video.videoWidth - shortestEdge)/2, // sx
                (video.videoHeight - shortestEdge)/2, // sy
                shortestEdge, // sw
                shortestEdge, // sh
                i, // dx
                0, // dy

```

```

    size, // dh
    size / dy
);
}
}, 67); // 67 – это приблизительно 15 кадров в секунду
}, false);

```

Главное волшебство происходит внутри метода `setInterval`, который срабатывает каждую 0,067 долю секунды (так как в JavaScript секунды измеряются в тысячах миллисекунд, а $1000/15 =$ приблизительно 67, то мы получаем примерно 15 кадров в секунду). Этого должно быть достаточно для имитации воспроизведения видео. После входа в `setInterval` запускается цикл по ширине холста, чтобы заполнить его верхнюю часть копиями нашего изображения.

Соответствие аргументов метода `drawImage` демонстрируется на рис. 5.14.

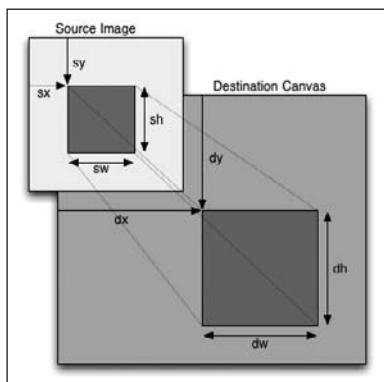


Рис. 5.14. Визуальное представление аргументов `drawImage`

Используя простое кадрирование по высоте и ширине и определяя короткую сторону, я с легкостью масштабирую изображение, предоставляя всю самую сложную работу элементу `<canvas>`. Результат: Брюс колотит банан по всей верхней кромке моего сайта (рис. 5.15).

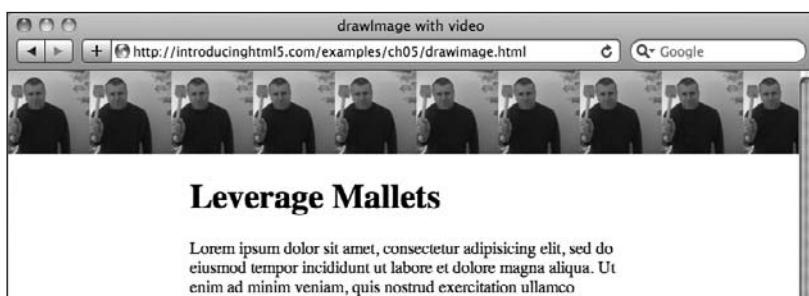


Рис. 5.15. Анимированный баннер в верхней части окна сделан с помощью холста и видеофрагмента

РАБОТА С ОТДЕЛЬНЫМИ ПИКСЕЛАМИ

Одна из самых впечатляющих возможностей API холста — возможность опрашивать отдельные пиксели. Альтернативная векторная технология рисования, SVG, такого не умеет, так как не предназначена для работы с пикселями. Работая с холстом, вы можете извлечь из объекта контекста 2D любой пикセル, разбитый на четыре цветовых канала: красный, зеленый, синий и альфа-канал прозрачности (red, green, blue, alpha transparency — rgba). Например:

```
var ctx = document.querySelector('canvas').
  getContext('2d'),
  img = document.createElement('img');

// прежде чем считывать данные, нужно дождаться окончания загрузки
// изображения
img.onload = function () {
  ctx.drawImage(img, 0, 0);
  var pixels = ctx.getImageData(0, 0, img.width,
    img.height);
};

};
```

ПРИМЕЧАНИЕ

Для того чтобы в методе `drawImage` использовалось другое изображение, оно должно передаваться по протоколу http (а не указываться как файл в локальной системе).

Переменная `pixels` относится к типу `CanvasPixelArray` и включает свойства `height`, `width` и `data`. Свойство `data` представляет собой массив данных пикселов, составленный следующим образом:

```
[ r1, g1, b1, a1, r2, g2, b2, a2, r3, g3, b3, a3, ... ]
```

Здесь `r1, g1, b1, a1` составляют первый пиксель, `r2, g2, b2, a2` — второй и т. д. `data.length` — это число пикселов, захваченных с помощью `getImageData` (в предыдущем примере оно равно числу пикселов изображения), умноженное на 4, так как для описания каждого пикселя используется 4 канала. Обратите внимание, что в `CanvasPixelArray` пиксели выделенной области перечислены от верхнего левого угла к нижнему правому построчно.

Имея доступ к подобным данным, изображения можно обрабатывать на уровне пикселов. Например, создавать пользовательские фильтры для приложений, таких как графические редакторы, показанные на рис. 5.2, или сканировать изображения с целью поиска определенных цветовых диапазонов. Можно даже написать веб-приложение для распознавания лиц на фотографиях.

Пол Руже и Тристан Нито из Mozilla в начале 2009 года продемонстрировали приложение (рис. 5.16), позволяющее вставлять в видеофрагмент (элемент `<video>`) динамическое содержимое — статическое или анимированное изображение, текст, другое видео. Когда на холст выводится очередной кадр видеофрагмента, приложение считывает пиксельные данные и ищет непрерывный блок белых пикселов (со значением цветовых каналов 255, 255, 255). Этот блок считается якорной точкой для вывода на холст другого визуально-

го элемента. На рис. 5.16 вы видите, как один элемент холста динамически вставляется в другой. Поиграть с этим приложением можно на веб-сайте по адресу <http://people.mozilla.com/~prouget/demos/DynamicContentInjection/play.xhtml>.

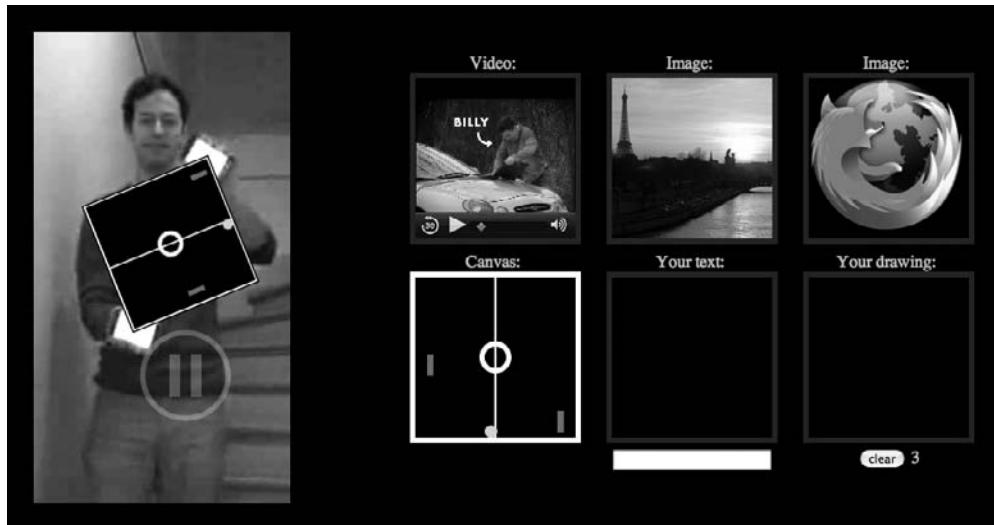


Рис. 5.16. Сканирование видео в поисках яркого пятна, поверх которого вставляется динамическое содержимое

В следующем примере кода я загружаю на холст изображение и инвертирую все пиксели, создавая курьезную рентгеновскую версию нашей с Брюсом фотографии (рис. 5.17):

```
var ctx = document.querySelector('canvas').  
    getContext('2d'),  
    img = document.createElement('img');  
  
// нужно дождаться, пока изображение загрузится  
img.onload = function () {  
    ctx.canvas.width = img.width;  
    ctx.canvas.height = img.height;  
    ctx.drawImage(img, 0, 0);  
    var pixels = ctx.getImageData(0, 0, img.width,  
        img.height);  
  
    for (var i = 0, n = pixels.data.length; i < n; i += 4) {  
        pixels.data[i+0] = 255 - pixels.data[i+0]; // красный  
        pixels.data[i+1] = 255 - pixels.data[i+2]; // зеленый  
        pixels.data[i+2] = 255 - pixels.data[i+1]; // синий  
        // i + 3 – это альфа-канал, который нам не нужен  
    }  
    ctx.putImageData(pixels, 0, 0);  
};  
img.src = 'authors.jpg';
```

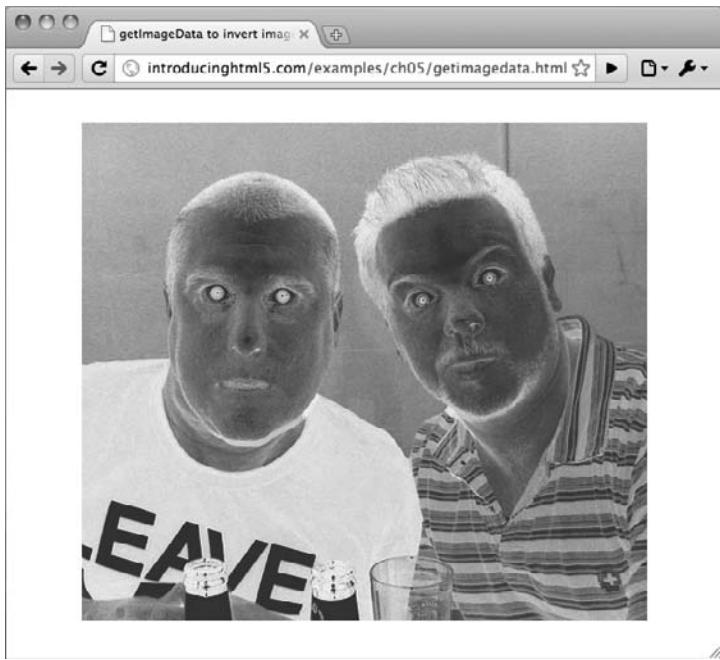


Рис. 5.17. Если бы вы загнали Брюса и Реми в рентгеновский аппарат, то на выходе получили бы такую странную картинку

Что происходит в этом листинге: я дожидаюсь, пока изображение полностью загрузится, а затем копирую его на холст. После того как изображение выводится на холст, я сразу же считаю данные пикселов, чтобы применить к ним нужную корректировку (инверсию).

В цикле `for` я использую выражение `i += 4`, чтобы пройти по всем пикселям, но не по каждому каналу каждого пикселя. Присваивая битам цветовых каналов значение (255 – текущее значение), я получаю инвертированные цвета.

Наконец, выполнив корректировку, я вывожу содержимое переменной `pixels` на холст с помощью метода `putImageData`. Для этого я передаю методу объект `CanvasPixelArray` и координаты `x` и `y` начальной точки.

ПРИМЕЧАНИЕ

У элемента `<canvas>` есть обеспечивающий безопасность внутренний флаг соответствия источников, который по умолчанию равен `true`. Значение этого флага меняется на `false`, если на холст выводится изображение или видео, источник которого не совпадает с источником документа-владельца холста. То же самое относится к использованию в качестве источника изображения другого холста, у которого флаг соответствия источников уже равен `false`. Если значение флага равно `false`, то применить метод `getImageData` или `toDataURL` невозможно, даже если изменить размер холста или нарисовать что-нибудь на нем после того, как флагу было присвоено значение `false`.

ЗАПИСЬ В ФАЙЛ

Вы создали произведение искусства, сравнимое по красоте с Сикстинской Мадонной? Хотите экспортить свое творение и поместить на рабочий стол? Сохранить во множестве различных форматов? Без проблем! Все это предусмотрено в функциональности холста.

Элемент `<canvas>` (но *не* контекст 2D) поддерживает экспортование текущего состояния холста в URL данных.

ЧТО ТАКОЕ URL ДАННЫХ?

Большинство браузеров поддерживают возможность считывания ресурсов, таких как изображения, в кодировке Base64. Веб-приложения, подобные Gmail, используют закодированные в Base64 изображения в коде CSS; это уменьшает число запросов к серверу (хотя файл CSS становится больше из-за встраиваемых в него данных изображения). Схема URL выглядит примерно так:

`data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAAAoAAAAK... и т. д.`

Адрес начинается со слова `data`, затем указан тип `mime`, кодировка — `base64`, а после этого идут необработанные данные (объем которых приблизительно на 30% больше размера исходного изображения). Необработанные (или сырье) данные — это то, что экспортится из элемента холста. Браузеры умеют декодировать их, превращая в реальные ресурсы (к сожалению, это не относится к IE7 и предыдущим версиям IE). Помимо этого, IE8 поддерживает URL данных длиной только до 32 Кбайт — и не стоит об этом забывать!

Экспортовать содержимое очень просто. У элемента холста есть метод `toDataURL`, при вызове которого вы указываете нужный формат изображения. Спецификация холста требует только поддержки формата PNG, однако в браузерах может быть реализована поддержка и других типов. Например, Safari поддерживает GIF, PNG и JPG. При попытке получить URL данных неподдерживаемого формата TIFF вы получаете только букву «A», повторенную множество раз, и никакого `data:<тип mime>`. Opera поддерживает только PNG, причем при запросе данных JPG и GIF возвращает также данные в формате PNG, игнорируя требуемый формат файла. Старые версии Firefox (на Mac) поддерживают только PNG, а для всех остальных типов данных возвращают ошибку (что, на мой взгляд, все же чрезмерно жестоко). Какой урок вы должны извлечь из этого? Получив URL данных, проверьте, что он начинается с `data:<нужный тип mime>`, чтобы удостовериться, что вы действительно получили то, что запрашивали.

В следующем примере кода мы создаем простейший (уровня Hello, world!) рисунок и сразу же сохраняем его в формате PNG, перенаправляя браузер на сформированный URL данных:

```
var ctx = document.querySelector('canvas').
  getContext('2d');
ctx.fillStyle = 'rgb(0, 0, 255)';
ctx.fillRect(0, 0, ctx.canvas.width, ctx.canvas.height);
ctx.fillStyle = 'rgb(0, 255, 0)';
ctx.fillRect(10, 20, 50, 50); // маленький квадратик
window.location = ctx.canvas.toDataURL('image/png');
```

Наконец, `toDataURL` также принимает необязательный второй аргумент, который доступен только в случае, когда поддерживается экспортование типа `image/jpg` с возможностью выбора уровня качества изображения. Допустимые значения аргумента — от 0.0 до 1, где

единица обозначает высочайшее качество. Будьте осторожны, так как уровень качества влияет на размер итогового изображения и, как в нашем случае, на объем данных Base64, который генерирует метод `toDataURL`.

АНИМИРОВАННЫЕ ИЗОБРАЖЕНИЯ НА ХОЛСТЕ

В этой главе мы уже касались простейшей анимации, но я хотел бы подробнее остановиться на некоторых аспектах.

БИБЛИОТЕКА JAVASCRIPT PROCESSING

Как вы вскоре увидите, перемещаться по холstu, рисуя линии и заливая формы, — это целая проблема, однако уже появились библиотеки, значительно упрощающие работу с холстом. Одна из таких библиотек называется `processing.js` (<http://processingjs.org/>). Она написана автором jQuery, Джоном Резигом.

Вообще-то, первоначально она не предназначалась для облегчения рисования. Эта библиотека интерпретирует на JavaScript язык Processing, с помощью которого вы, в свою очередь, рисуете на элементе холста. `processing.js` — это великолепный инструмент визуализации, позволяющий создавать намного более сложные рисунки и использовать комплексные процедуры анимации в API рисования 2D.

Простая анимация — это, в сущности, попеременная очистка текущего состояния холста и рисование новой картинки. Так как речь идет о холсте — естественном API рисования, — то все происходит очень просто и быстро. Я покажу вам пример, в котором голова Брюса будет скакать по всей поверхности холста. Этот пример основан на выдающемся учебнике по работе с холстом авторства Билла Милла. Я всего лишь усовершенствовал урок, заставив двигаться лицо Брюса вместо простого черного круга.

Код, результат исполнения которого показан на рис. 5.18, относительно прост. Он разбит на следующие этапы:

- 1) инициализировать холст и объект;
- 2) очистить холст;
- 3) нарисовать на холсте мяч.



Рис. 5.18. Скрасить скучные часы ожидания вам поможет голова Брюса, скакущая по холсту и вращающаяся вокруг своей оси

Для того чтобы добавить примеру изящества, я заставлю лицо Брюса не только скакать по экрану, но и вращаться. Для этого мне нужно будет периодически поворачивать холст, как мы уже делали ранее в этой главе.

Поскольку лицо Брюса должно вращаться, лучше сделать это с помощью *еще одного* холста, чтобы не засорять вращениями и трансляцией главный холст. Таким образом, я разобью одну сложную задачу на две простых: фотография Брюса будет вращаться на одном холсте, а на втором я буду вычислять положение изображения и выводить его на экран.

```
var ctx = document.querySelector('canvas').  
  ~ getContext("2d"),  
    ballctx,  
    x = 100, // произвольная начальная точка  
    y = 50,  
    dx = 2,  
    dy = 4,  
    width = ctx.canvas.width,  
    height = ctx.canvas.height;  
  
// загрузка изображения  
ballImg = document.createElement('img');  
ballImg.src = 'bruce-ball.png';  
  
// после загрузки изображение может начинать  
// прыгать  
ballImg.onload = function () {  
    var ball = document.createElement('canvas');  
    ball.height = 50;  
    ball.width = 50;  
  
    ballctx = ball.getContext('2d');  
    // транслируем центр для правильного вращения  
    ballctx.translate(25, 25);  
  
    setInterval(draw, 10);  
};  
  
function draw() {  
    ctx.clearRect(0, 0, width, height);  
  
    ballctx.rotate(Math.PI/180*5); // 5 градусов  
  
    // рисуем в положении 0,0  
    ballctx.drawImage(ballImg, 0, 0,  
        ~ ballImg.width,  
        ~ ballImg.height, -25, -25, 50, 50);  
  
    // копирование источника после поворота  
    ctx.drawImage(ballctx.canvas, x, y);  
  
    if (x + dx > width || x + dx < 0)
```

```

dx = -dx;
if (y + dy > height || y + dy < 0)
    dy = -dy;

x += dx;
y += dy;
}

```

Все действие происходит в функции `draw`, но только после того, как я заканчиваю подготовку. Подготовка заключается в том, что я динамически создаю холст с мячиком, но не помещаю его в DOM. Затем этот холст транслируется, поэтому лицо Брюса вращается в центре холста. Я все так же могу использовать контекст 2D этого «незакрепленного» холста, и я явно задаю высоту и ширину холста (в противном случае автоматически устанавливается размер 300×150 пикселов).

Функция `draw` выполняется каждую сотую долю секунды (каждые 10 миллисекунд), постоянно увеличивая позиции `x` и `y` и перерисовывая холст с мячиком на главном холсте. Разумеется, перед тем как перерисовывать прыгающий мячик, я очищаю главный холст методом `ctx.clearRect(0, 0, width, height)`, то есть убираю с экрана все содержимое. Вот и все. Вся анимация. Примерно то же самое, что рисовать мультику на листах блокнота.

СОХРАНЕНИЕ И ВОССТАНОВЛЕНИЕ УСЛОВИЙ РИСОВАНИЯ

В API 2D можно найти еще кое-что замечательное: условия рисования. У объекта контекста есть два метода, `save` и `restore`, которые позволяют управлять текущим стеком условий рисования. Метод `save` вталкивает текущие условия в стек, а метод `restore` выталкивает набор условий, находящийся наверху стека.

Условия рисования описывают не все, что происходит на холсте, а лишь следующее:

- трансформации;
- области отсечения (в этой книге не рассматриваются);
- текущие значения атрибутов `fillStyle`, `font`, `globalAlpha`, `globalCompositeOperation`, `lineCap`, `lineJoin`, `lineWidth`, `miterLimit`, `shadowBlur`, `shadowColor`, `shadowOffsetX`, `shadowOffsetY`, `strokeStyle`, `textAlign` и `textBaseline`.

Например, следующий фрагмент кода из учебника по работе с холстами в Mozilla демонстрирует, как нарисовать на холсте 50 звездочек в случайных точках. Позиции устанавливаются методом `translate`. Однако в конце каждой итерации цикла восстанавливается исходное состояние холста, то есть верхний левый угол холста перемещается в реальный верхний левый угол окна, а не остается в позиции, установленной последним `translate`:

```

for (var j=1;j<50;j++){
    ctx.save();
    ctx.fillStyle = '#fff';
    ctx.translate(75-Math.floor(Math.random()*150),
                 75-Math.floor(Math.random()*150));
    drawStar(ctx,Math.floor(Math.random()*4)+2);
    ctx.restore();
}

```

ПРИМЕЧАНИЕ

Обратите внимание, что методы `save` и `restore` не влияют на пути и растровые рисунки, уже находящиеся на холсте (невозможно восстановить предыдущее состояние рисунка на холсте).

ВИЗУАЛИЗАЦИЯ ТЕКСТА

На холсте можно вывести какой-нибудь текст, причем указав для него шрифт, размер, выравнивание и положение нижней линии. Также можно залить буквы (то есть сделать их похожими на обычные буквы в книге) или только обвести их очертания. Некогда существовавший проект Bespin представлял собой отличный пример применения пользовательской визуализации текста для создания полнофункционального редактора кода исключительно с использованием возможностей API холста (на смену ему пришел Ace от ребят из Ajax.org, но их версия не использует холст).

Для того чтобы нарисовать текст, вам нужна сама строка и координаты. Например, я использовал рисованный текст для того, чтобы продемонстрировать, как работает метод `translate` (рис. 5.19, а также рис. 5.13 ранее в этой главе). С помощью `fillText` я подписывал место, куда переносится новая центральная точка холста, а также координаты точек (размер холста в данном примере равен 300×300 пикселов):

```
function dot(string) {
    ctx.beginPath();
    ctx.arc(0,0,5,0,Math.PI*2,true); // рисую
    // круг
    ctx.fill();
    ctx.fillText(string, 5, 10); // вывожу текст
}
```

Теперь можно транслировать холст и вызывать функцию `dot`, передавая ей строку с текстом подписей для точек:

```
dot('1. no translate'); // рисую точку
ctx.translate(150, 150);
dot('2. (150, 150)'); // рисую точку
ctx.translate(-100, 20);
dot('3. (-100, 20)'); // рисую точку
```

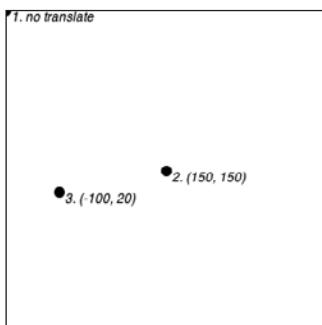


Рис. 5.19. Подписываем рисунки на холсте с помощью `fillText`

По умолчанию в методе `fillText` используется шрифт без засечек высотой 10 пикселов. Настройки шрифта можно изменить, установив свойство `font` контекста. Синтаксис такой же, как для настройки шрифтов в CSS (например, `ctx.font = 'italic 400 12px/2 helvetica neue, sans-serif'`). Можно даже использовать веб-шрифты CSS, при условии, что они полностью загружаются до того, как вы пытаетесь обратиться к ним в коде. При вызове `fillText` для визуализации текста используются установленные ранее свойства `fillStyle` (либо настройки холста по умолчанию). Точно так же для `strokeText` используются свойства `strokeStyle`.

ДОСТИЖИМОСТЬ В ПРЕДЕЛАХ ЭЛЕМЕНТА ХОЛСТА

Одна из причин, почему холст так быстро работает на современных оптимизированных интерпретаторах JIT языка JavaScript, — это отсутствие DOM. Речь идет только о наборе пикселов, никакая информация о геометрических фигурах, тексте и изображениях не сохраняется (сравните это с подходом SVG, где все эти разнообразные объекты входят в структуру DOM). Таким образом, если вам необходимо, например, отслеживать коллизии, то приходится все это реализовывать вручную. Нет никакого образа, с которым JavaScript мог бы взаимодействовать.

ПРИМЕЧАНИЕ

JIT означает компиляцию Just in Time, то есть «компиляцию вовремя». Эта технология значительно повышает производительность программ.

Обратная сторона хорошей производительности — трудности с достижимостью. Если ваши игры управляются *и* мышью *и* клавиатурой, это означает, что вы удовлетворяете требованиям большинства пользователей. Но что касается пользователей с нарушениями зрения, то все далеко не так радужно. Вспомогательным технологиям просто не за что зацепиться. Когда текст попадает на холст, он перестает быть настоящим текстом, а превращается в обычный пиксельный рисунок. Это еще хуже, чем вывод изображений в теге ``, потому что там хотя бы есть атрибут `alt`. Хотя с помощью сценариев содержимое элемента (текст между тегами холста) можно менять в зависимости от того, какой текст вставляется в коде JavaScript, не думаю, что разработчики будут этим серьезно заниматься.

ПРИМЕЧАНИЕ

Встраиваемый модуль jQuery Visualize от Filament Group умеет с помощью jQuery вставлять на страницу элемент `canvas` с графической информацией, отражающей данные из таблицы в разметке страницы. Вспомогательные технологии обращаются к необработанным данным в таблице, а пользователи с хорошим зрением получают дополнительное преимущество в виде графиков и диаграмм на холсте.

Команда по реализации вспомогательных технологий в составе рабочей группы HTML ищет способы, как сделать доступ к элементам холста более удобным. Нельзя сказать, что это невозможно; в конце концов, дополнительные возможности были добавлены в Flash 5.

Однако пока что я не рекомендую применять холст в пользовательских интерфейсах, а также как единственный способ передачи информации пользователю. Хороший пример использования холста в качестве дополнения к обычному тексту — это встраиваемый модуль jQuery Visualize от Filament Group (см. примечание).

ЗАКЛЮЧЕНИЕ

Благодаря API холста разработчики наконец-то получили способ динамически генерировать и манипулировать графическими объектами на клиентской стороне, прямо в браузерах, не пользуясь приложениями на основе Flash. Холст особенно удобен для обработки отдельных пикселов, и я уверен, что у пиксельных приложений большое будущее. Совсем скоро мы увидим то, чего еще не было в истории Всемирной паутины.

Однако нужно с большим вниманием подходить к выбору правильной технологии. Прежде чем бросаться с головой в разработку новой версии приложения Awesome 3.0, подумайте, не стоит ли взглянуть в сторону SVG. И не забывайте о вероятных трудностях для пользователей с ограниченными возможностями.

Глава 6

ХРАНЕНИЕ ДАННЫХ

Реми Шарп



Одна из важнейших задач, которую приходится решать разработчикам приложений — идет ли речь о веб-приложениях или приложениях для настольной системы, — это хранение данных. Запись уникальных ключей, определяющих способ отображения страниц, сохранение имен пользователей, индивидуальных настроек, — список можно продолжать бесконечно.

Раньше, для того чтобы запомнить данные веб-приложения, разработчикам приходилось либо дробить их, сохраняя часть на сервере и создавая некое связующее звено между клиентом и сервером, либо использовать файлы cookie.

Файлы cookie — это кошмар и ужас. Нет, само слово переводится прекрасно, печенье любят все. И все же, оно должно быть на столе, а не в браузере. С файлами cookie связано множество проблем, превращающих работу с ними в сплошную головную боль. Приступая к новому проекту, в котором будут использоваться файлы cookie, первым делом я отправляюсь на поиски библиотеки JavaScript, предназначеннной специально для таких приложений. Если мне не удается найти ее, то я обращаюсь за помощью к Google и копирую нужные фрагменты из кода, написанного Питером-Полом Кохом.

ПРИМЕЧАНИЕ

Код Питера-Поля Коха для приложений, использующих файлы cookie, можно загрузить с сайта <http://www.quirksmode.org/js/cookies.html>.

Программная реализация файлов cookie чрезмерно усложнена. Только взгляните: создание файла cookie на JavaScript выглядит примерно так:

```
document.cookie = "foo=bar; path=/";
```

Это сеансовый файл cookie. Если же вы не хотите, чтобы данные стерлись в момент окончания сеанса, при создании файла нужно указать для него срок жизни (чтобы подарить файлу cookie бессмертие, придется периодически прибавлять ему n дней жизни):

```
document.cookie = "foo=bar; path=/; expires=Tues,
- 13 Sept 2010 12:00:00";
```

Формат времени исключительно важен — вы уже поняли, что это еще один источник головной боли? А теперь покроем наши черствые печеньушки прокисшей глазурью: удалить файл cookie можно, только присвоив ему пустое значение:

```
document.cookie = "foo=; path=/";
```

Но не все так просто: в действительности, файл при этом *не удаляется*. Меняется его значение и срок жизни — до конца сеанса, то есть до закрытия браузера. Вы согласны, что в реальном мире удаление должно означать удаление?

Файлы cookie не работают, потому что это не технология, а букет проблем. Новые спецификации хранения позволяют полностью отказаться от устаревшего подхода к созданию, извлечению и удалению данных. Теперь в распоряжении разработчиков есть новый понятный интерфейс прикладного программирования.

Однако будучи британцем, я обязан в некоторой мере опровергнуть свое предыдущее заявление о том, что «файлы cookies ужасны». Не бойтесь, я не собираюсь противоречить себе во всем. Если перед вами стоит задача доставить данные с клиентской стороны на

серверную, то файлы cookie — это самое подходящее решение, так как они автоматически присоединяются к каждому запросу. Если же этого делать не требуется, вам понадобится механизм хранения данных на серверной стороне. По моему мнению, технология веб-хранилищ сумела достигнуть той степени развития, до которой не добрались файлы cookie, и они намного функциональнее. Поэтому не обращайте на это опровержение слишком много внимания — давайте лучше посмотрим, что там спрятано в хранилищах современных браузеров!

ВАРИАНТЫ ХРАНЕНИЯ

Что касается хранения данных на клиентской стороне, существует три варианта:

- веб-хранилище (Web Storage), которое поддерживается во всех современных браузерах (<http://www.w3.org/TR/webstorage/>);
- базы данных SQL для веб (Web SQL Database), которые поддерживаются браузерами Opera, Chrome и Safari (<http://www.w3.org/TR/webdatabase/>);
- индексированные базы данных, IndexedDB, которые на экспериментальном уровне на момент написания этой главы поддерживаются в Chrome 12, Firefox 5 и Internet Explorer 10 (<http://www.w3.org/TR/Indexeddb/>).

Из названия «база данных SQL для веб» сразу становится понятно, как это работает: браузер обращается к локальной базе данных при помощи SQL-запросов. Возможно, вы сразу же решили, что это ваш вариант, потому что вы уже знакомы с SQL. Небольшая потенциальная проблема заключается в том, что, как только вы открываете страницу спецификации, вам в глаза сразу же бросается предупреждение: разработка данной спецификации продолжаться не будет. Однако она уже хорошо поддерживается, особенно в мобильном пространстве, поэтому я хочу продемонстрировать вам, как использовать этот API. Кроме того, Google реализовала базу данных SQL для веб в мобильной версии Gmail, поэтому я не сомневаюсь, что эта технология просуществует в браузерах еще достаточно долго.

Название «IndexedDB» не так описательно, хотя вы наверняка догадались, в чем суть. IndexedDB — это хранилище документов сродни популярным сегодня базам данных «NoSQL», таким как MongoDB. Фактически у вас есть ключ, и вы можете сохранить с этим ключом данные любого типа, а не оперировать заранее установленным количеством столбцов определенного типа, как в традиционных базах данных SQL. Интересно, что в центре взаимодействия с API находятся события. Подробнее об этом мы поговорим ближе к концу главы.

Веб-хранилище — это куда более простая система, в которой различные значения просто связываются с уникальными ключами. Здесь нет таких объемов кода, как при работе с Web SQL и IndexedDB. Вообще, поддержка API веб-хранилища реализована намного лучше, чем существующих альтернатив, но в будущем ситуация должна поменяться. У нас будет простой метод хранения (веб-хранилище) и крупное хранилище данных (IndexedDB, когда его поддержка будет реализована в остальных браузерах). Я рассмотрю все три интерфейса прикладного программирования. Вы узнаете, как они работают и как отлаживать добавление и хранение данных в обеих системах.

Что касается доступных объемов, то стандартный лимит веб-хранилища — 5 Мбайт (хотя при превышении этого лимита браузеры обычно выводят предупреждение, и пользователь может предоставить конкретному веб-сайту больше места на диске).

На противоположном конце спектра находятся базы данных SQL для веб, в спецификации которых никаких лимитов не предусмотрено. Создавая базу данных, разработчик должен сам определить для нее подходящий размер.

В спецификации IndexedDB ограничения не упоминаются (это может быть противоположный конец спектра по отношению к первым двум?), однако создается впечатление, что предел, установленный в Chrome, равен 5 Мбайт. Какие ограничения накладывают Firefox и IE10, неясно. Уверен, как только технология получит распространение в браузерах, все эти тонкости будут хорошо задокументированы.

Все данные привязаны к *источнику* документа, то есть к протоколу, хосту и номеру порта (по умолчанию 80). Это означает, что данные на сайте <http://remysharp.com> не могут обращаться к данным на безопасной версии <https://remysharp.com>.

В любом случае, если API не удается записать данные, браузер возвращает ошибку. Я буду рассматривать небольшие приложения, в которых объем данных в хранилище не превышает 100 Кбайт.

ВЕБ-ХРАНИЛИЩЕ

В двух словах, API веб-хранилища — это файлы cookie на стероидах (но если это слишком огорчает вас, вспомните мое опровержение чуть раньше). Ключевое преимущество данного API — логичное и понятное разделение сеансовых и долговременных данных. Если создать сеансовый файл cookie (то есть не указывать для него срок жизни), то до тех пор, пока открыт браузер, этот элемент данных будет доступен всем окнам, способным обращаться к соответствующему домену. С другой стороны, веб-хранилища позволяют определять объект `sessionStorage`, относящийся только к определенному окну или вкладке, на которой работает пользователь. Как только окно или вкладка закрывается, данные исчезают, а не болтаются где-то в памяти, дожидаясь, когда браузер будет закрыт целиком.

ПРИМЕЧАНИЕ

Когда я говорю, что у «окон есть доступ», я имею в виду объект окна. Он обычно привязан к определенному источнику (протокол + хост + порт) и описывает окно или вкладку браузера. Просто напоминаю на случай, если вы запутались!

Данный API хранения предлагает два типа хранилищ: `sessionStorage` и `localStorage`.

ПРИМЕЧАНИЕ

Файлы cookie на стероидах в сравнении с обычными: IE6 поддерживает только 20 файлов cookie для каждого домена, а максимальный размер одного файла — 4 Кбайт. Веб-хранилище не ограничивает количество элементов для каждого домена, только их совокупный объем — до 5 Мбайт.

Данные, созданные в сеансовом хранилище, то есть хранилище типа `sessionStorage`, доступны только одному окну и только до его закрытия (то есть до завершения сеанса). Если открыть другое окно в том же домене, оно не сможет обращаться к данным первого сеанса. Благодаря этому предотвращается «протечка» данных сеанса из одного окна в другое.

Хранилище типа `localStorage` привязано к конкретному источнику и охватывает все окна, открытые в этом домене. Данные, записанные в локальное хранилище одним окном, сразу же становятся доступны во всех остальных окнах. Они сохраняются в общем доступе до тех пор, пока их не удаляет пользователь или ваше веб-приложение. Можно закрыть браузер, выключить компьютер, снова включить его через несколько дней, и если данные не были явно удалены, то они все так же будут доступны. Вы получаете возможность постоянного хранения информации без сложностей с файлами cookie, которым приходится снова и снова переопределять срок жизни.

ОБРАТИТЕ ВНИМАНИЕ: БЕЗОПАСНОСТЬ ФАЙЛОВ COOKIE В FIREFOX

В браузере Firefox безопасность доступа к сеансовому и локальному хранилищу реализована несколько иначе: если использование файлов cookie отключено, то при попытке доступа к хранилищам `sessionStorage` и `localStorage` возвращается ошибка. Следовательно, прежде чем обращаться к любому из этих двух API хранения, нужно проверить, можно ли создавать файлы cookie:

```
var cookiesEnabled = (function () {
    // id – это наше тестовое значение
    var id = new Date().getTime();

    // создаем файл cookie для проверки доступа к таким файлам
    document.cookie = '__cookieprobe=' + id + ';path=/';

    // если файл cookie создан, значит, все в порядке
    return (document.cookie.indexOf(id) !== -1);
})();
```

Этот код создает файл cookie и сразу же пытается снова его считать. Если чтение возвращает ошибку, значит, система безопасности блокирует попытки создания файлов cookie, и, следовательно, обратиться к `sessionStorage` или `localStorage` тоже не получится. Когда использование файлов cookie отключено, считывание данных из хранилищ `sessionStorage` или `localStorage` приводит к ошибке, а исполнение кода JavaScript прерывается.

В качестве альтернативы поддержку Web Storage можно протестировать с помощью оператора `try/catch`, а также добавить поддержку метода заполнения на базе JavaScript (как рассказывается в главе 12 и немного в этой).

Но веб-хранилище так сильно отличается в лучшую сторону от файлов cookie не только за счет API, но также благодаря связанной системе событий. Мы поговорим о событиях ближе к концу главы.

15-СЕКУНДНЫЙ УРОК

У меня не вызывает никаких сомнений то, что вы сразу поймете, как использовать `localStorage`. Поэтому еще до всяких объяснений я решил продемонстрировать этот код, и я уверен, что основы веб-хранилищ сразу же закрепятся у вас в голове!

```

localStorage.superHero = "Remy";
localStorage.superVillain = "Bruce";
// здесь происходит схватка супергероев
delete localStorage.superVillain;
// страница перезагружается, браузер закрывается и открывается –
// нам все равно, мы супергерои!
alert("The world's baddest badass is: " +
  localStorage.superHero);

```

Да, все именно настолько просто. Если вы закроете браузер, перезагрузите компьютер и позже вернетесь к домену, где были установлены эти данные, то найдете их на месте в целости и сохранности. Вы сможете проверить значение `localStorage.superHero` и получите, разумеется, строку *Remy!*

Обзор API

Поскольку и `sessionStorage`, и `localStorage` происходят от API веб-хранилища, то их API совершенно одинаковые. Из спецификации:

```

readonly attribute unsigned long length;
getter DOMString key(in unsigned long index);
getter DOMString getItem(in DOMString key);
setter creator void.setItem(in DOMString key, in any data);
deleter void.removeItem(in DOMString key);
void clear();

```

В API веб-хранилища создавать и извлекать данные очень просто. Метод `setItem` принимает ключ и значение. Метод `getItem` принимает ключ данных и возвращает соответствующее значение:

```

sessionStorage.setItem('twitter', '@rem');
alert( sessionStorage.getItem('twitter') ); // выводится "@rem"

```

Стоит заметить, что метод `getItem` поддерживает только строковые данные. Это важно, потому что при попытке сохранить объект в действительности сохраняется [Object object]. Что еще важнее, в строки преобразуются даже числовые данные, и это нужно учитывать, чтобы избежать ошибок во время разработки.

Для того чтобы подчеркнуть важность этой особенности, приведу пример: предположим, Брюс на своем веб-сайте продает видеоролики с познавательными лекциями на разнообразные научные темы. Заинтересовавшись вопросом синергизма, вы добавили в корзину несколько видео. Общая стоимость элементов в корзине — 12 долларов, и эта стоимость записывается в объект `sessionStorage`. Когда вы заходите на страницу оформления заказа, Брюс добавляет еще 5 долларов в качестве оплаты за доставку. Как я уже говорил, сумма в 12 долларов уже сохранена в объекте `sessionStorage`. Вот как мог бы выглядеть код преобразования:

```

sessionStorage.setItem('cost', 12);

// после добавления стоимости доставки сайт Брюса
// возвращает полную цену:
function costWithShipping(shipping) {
  alert(sessionStorage.getItem('cost') + shipping);
}

```

```
// потом отображается стоимость содержимого
// корзины с доставкой:
costWithShipping(5);
```

Если бы в `sessionStorage` было записано значение числового типа, то в окне результата вы увидели бы значение 17. Но сумма в 12 долларов была записана в виде строки. Так как в JavaScript для сложения и для соединения строк используется один и тот же метод (знак плюс), то JavaScript воспринимает эту операцию как *присоединение* числа к строке, поэтому в окне результата выводится значение 125 — намного больше реальной стоимости видеороликов Брюса! Происходящее за сценой называется *приведением типов*: любые данные, сохраняемые в API хранилища, приводятся к строковому типу.

Наконец, важно отметить, что если вызывать `getItem` с несуществующим ключом, то API вернет значение `null`. Если вы собираетесь использовать API хранилища для инициализации значений, что вполне возможно, то перед продолжением всегда выполняйте проверку на `null`. Попытавшись обработать `null` как объект любого другого типа, вы сами себе вставите огромные палки в колеса.

Способы доступа к хранилищу

Вы, наверное, думаете: «Подождите-ка минуточку, Реми за пятнадцать секунд обучил меня тонкостям использования веб-хранилищ, а теперь вещает о каких-то `getItem`, `setItem` и прочей чепухе!» Я рад, что вы внимательно читаете книгу. Если снова взглянуть на описание API, то вы поймете, что `getItem`, `setItem` и `removeItem` — это `getter`, `setter` и `deleter` соответственно. Это означает, что при вызове `delete localStorage.superVillain` в действительности JavaScript вызывает для нас `removeItem`. Если вы сами догадались, молодцы. Возьмите еще пирожок.

Синтаксис `expando` — это быстрый и эффективный способ записи и извлечения данных из объекта хранилища, и поскольку `sessionStorage` и `localStorage` оба происходят от API веб-хранилища, эти типы хранилищ поддерживают установку значений напрямую через объект хранилища.

Вспоминая наш пример с сохранением имени пользователя из системы микроблогов Twitter, попробуем реализовать то же самое с применением синтаксиса `expando`:

```
sessionStorage.twitter = '@rem';
alert( sessionStorage.twitter ); // отображается: "@rem"
```

К сожалению, как и в предыдущем примере с видеороликами Брюса, методы `expando` также страдают от «строкофикации» значений, так как действие происходит через метод `setItem`.

Использование метода `key`

В этом API также есть метод `key`, который принимает индексный параметр и возвращает соответствующий ключ. Он удобен для перечисления данных в объекте хранилища. Например, вы хотите вывести на экран все ключи и связанные с ними данные, но значения ключей вам неизвестны. Чтобы узнать их, просто пройдитесь по всему объекту хранилища, используя метод `key`:

```
for (var i = 0; i < sessionStorage.length; i++) {
  alert( sessionStorage.key(i) + '=' +
    sessionStorage.getItem( sessionStorage.key(i) ) );
}
```

Предупреждение: вполне возможно, что какое-то значение будет записано с ключом «key», например, вы создадите его с помощью такого кода:

```
sessionStorage.setItem('key',
  '27152949302e3bd0d681a6f0548912b9');
```

У вас есть значение, которому соответствует ключ «key», и метод `key` для объекта хранилища. Чуете, куда ветер дует?

Некоторые браузеры, в частности WebKit, переопределяют метод `key`, используя указанное вами новое значение. Однако в инструментах разработки WebKit метод `key` применяется для перечисления и отображения всех данных, связанных с объектом хранилища. Получается, что представление данных для указанного типа хранилища (в нашем случае — `sessionStorage`) будет «сломано» до тех пор, пока двойственное значение не будет удалено.

ПРИМЕЧАНИЕ

Надеюсь, по мере развития браузеры избавятся от подобных ошибок, но пока что могу только посоветовать не использовать имена, уже имеющиеся в API хранилища.

В других браузерах, таких как Firefox, метод `key` и значение «key» разделяются. Получить метод можно, применяя синтаксис `expand`, а для получения значения можно использовать `getItem('key')`.

Удаление данных

Существует три способа программного удаления данных из объекта хранилища: напрямую с помощью `deleter`, `removeItem` и `clear`. Метод `removeItem` принимает ключ (тот же, который используется с `setItem` и `getItem`) и удаляет соответствующую запись.

Для того чтобы очистить объект хранилища полностью, удалив все записи, используйте `clear`, например:

```
sessionStorage.setItem('remy', "Master of the Universe");
sessionStorage.setItem('bruce', "Master of the Puniverse");
alert( sessionStorage.length ); // отображается "2"
sessionStorage.removeItem('bruce');
alert( sessionStorage.length ); // отображается "1"
sessionStorage.clear();
alert( sessionStorage.length ); // отображается "0"
```

Как сохранять не только строки

Обойти проблему «строкофикации» объектов можно с помощью JSON. Так как в стандарте JSON для представления объектов JavaScript используется обычный текст, его можно применять для записи объектов в хранилище и для преобразования сохраненных данных

обратно в объекты. Для этого понадобится добавить оболочки вокруг методов `set` и `get`, но, в зависимости от вашего приложения, это может и не создать вовсе никаких проблем.

ПРИМЕЧАНИЕ

JSON (JavaScript Object Notation) — это открытый стандарт представления данных в текстовом виде. Спецификация, которую можно найти на сайте <http://json.org>, настолько проста, что полностью помещается на обороте визитки!

Все новейшие браузеры (ночные сборки и финальные релизы) поддерживают кодирование JSON с помощью методов `JSON.parse` и `JSON.stringify`. Для браузеров без поддержки JSON можно использовать библиотеку JSON Дугласа Крокфорда (загрузите ее с веб-сайта <https://github.com/douglascrockford/JSON-js>).

Для создания оболочки методов `setItem` и `getItem` можно использовать такой код:

```
var videoDetails = {
  author: 'bruce',
  description: 'how to leverage synergies',
  rating: '-2'
};

sessionStorage.setItem('videoDetails', JSON.
stringify(videoDetails) );

// позже, например, при повторной загрузке
- страницы можно извлечь
- сохраненные данные
- stored data
var videoDetails = JSON.parse(sessionStorage.getItem
('videoDetails'));
```

Как говорилось в разделе, посвященном обзору API, если в объекте хранения запрашиваемый ключ отсутствует, то метод возвращает значение `null`. Это не проблема для стандартных парсеров JSON, так как `JSON.parse(null)` вполне предсказуемо возвращает значение `null`. Однако если передать `null` коду на JavaScript от Дугласа Крокфорда, это приведет к ошибке. Следовательно, если вам известно, что будет использоваться библиотека Дугласа для JSON на JavaScript, то заранее предусмотрите обработку ошибки, добавив такой код:

```
var videoDetails = JSON.parse(sessionStorage.getItem
('videoDetails') || 'null');
```

Так вы гарантируете, что если метод `getItem` вернет значение `null`, то дальше вы передадите версию этого значения, подготовленную для JSON, и парсер JSON на основе JavaScript отработает без ошибок.

Использование инструментов отладки

Хотя API веб-хранилища реализован и поддерживается очень качественно, отладчики все еще находятся на стадии развития. За исключением инструментов анализа содержащегося `localStorage` и `sessionStorage`, больше выбирать, в общем-то, не из чего. Правда,

зачастую инструменты отладки позволяют модифицировать ключи и значения, а также удалять записи.

Инструменты разработчика в WebKit

Несмотря на то что в названии раздела стоит WebKit, здесь я имею в виду Safari, ночную сборку Safari (WebKit) и Google Chrome. Инструменты разработчика WebKit позволяют просматривать значения `localStorage` и `sessionStorage` в окне, показанном на рис. 6.1.

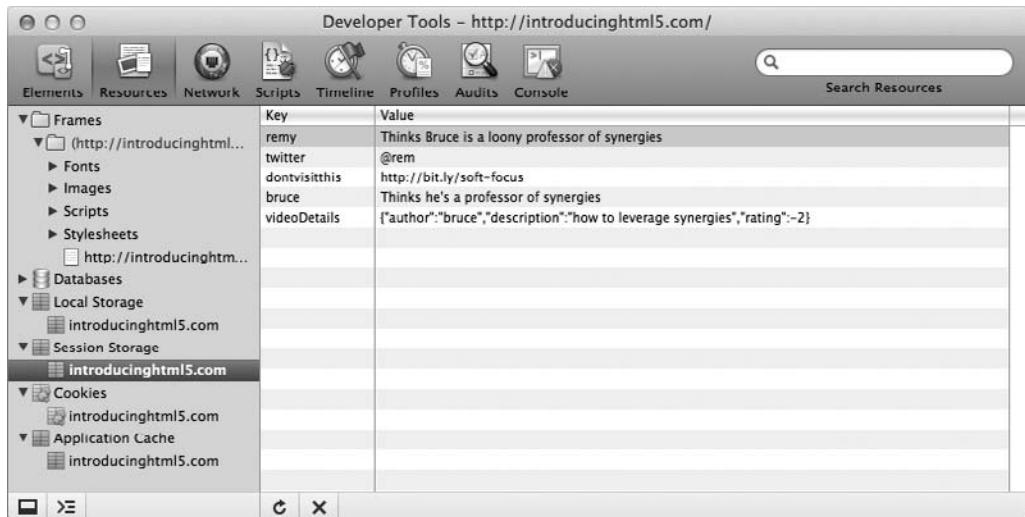


Рис. 6.1. Отладчик хранилища Chrome (в Safari интерфейс почти не отличается)

ПРИМЕЧАНИЕ

Для того чтобы включить в Safari отображение меню Developer (Разработчик), откройте меню Preferences (Настройки) и установите флагок Show Developer Menu (Показывать меню разработчика) в поле Menu Bar (Полоса меню) на вкладке Advanced (Дополнительно). В Chrome инструмент отладки можно открыть, нажав на значок гаечного ключа, затем меню Tools (Инструменты), Developer Tools (Инструменты разработчика).

Отладчик в Firefox

С помощью встроенной JavaScript-консоли Firefox можно с легкостью заглянуть в объекты хранилища. Укажите в команде консоли `sessionStorage` или `localStorage`, выполните код и сможете щелкнуть объект хранилища и просмотреть его содержимое (рис. 6.2).

DragonFly в Opera

Инструмент DragonFly поставляется в составе Opera. На вкладке Storage (Хранилище) можно просмотреть все данные, записанные для текущей страницы. В частности, предусмотрены отдельные вкладки для локального и сеансового хранилищ, упрощающие изучение данных, связанных с этими типами хранилищ (рис. 6.3).

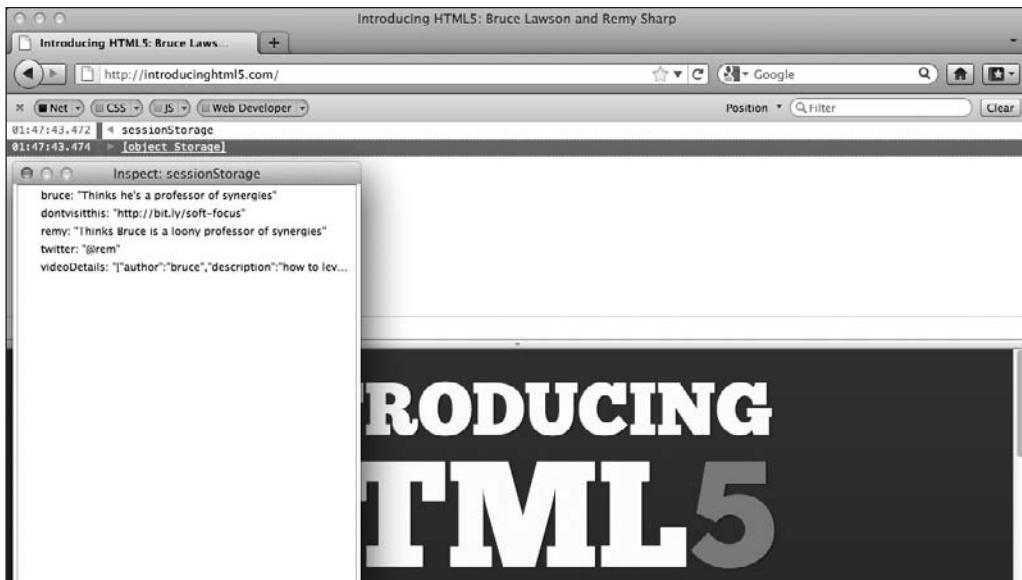


Рис. 6.2. Встроенный отладчик Firefox

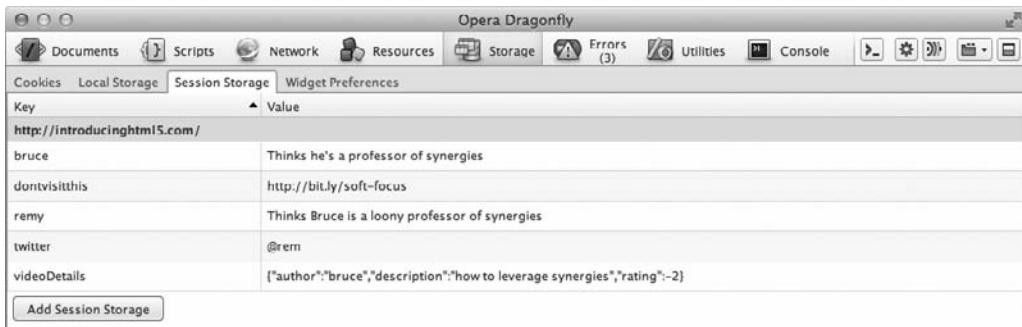


Рис. 6.3. Отладчик DragonFly в браузере Opera позволяет просматривать содержимое хранилища

СОБЫТИЯ ХРАНИЛИЩА

Что определяет истинную ценность веб-хранилища — так это встроенные события, уведомляющие вас об обновлениях в хранилище данных.

В первую очередь необходимо запомнить, что событие хранилища срабатывает не на окне, которое сохраняет фактические данные. Оно сработает только на *другом* окне, чье хранилище в данный момент модифицируется.

Это означает, что события хранилища для `sessionStorage` срабатывают только на объектах `iframe` с тем же источником и всплывающих окнах, открытых с помощью `window.open()`, так как при этом новая сессия не создается. События хранилища для `localStorage` сраба-

тывают на всех окнах, открывающихся для того же источника. Через секунду мы взглянем на пример ситуации, когда это может быть полезно.

Сработавшее событие способно предоставить всю информацию об изменениях данных, что демонстрирует объект события хранилища:

```
StorageEvent {
    readonly DOMString key;
    readonly any oldValue;
    readonly any newValue;
    readonly DOMString url;
    readonly Storage storageArea;
};
```

Помните, что хотя в спецификации говорится, что он поддерживает «любое» значение, это не так. В настоящее время все браузеры приводят эти значения к строковой форме, поэтому не сомневайтесь, что `oldValue` и `newValue` обязательно будут строками!

`storageArea` указывает либо на `localStorage`, либо на `sessionStorage` — очевидно, в зависимости от того, где сохранены данные.

Синхронизация окон с помощью событий хранилища

Предположим, что мы создаем веб-сайт радиостанции, включающий всплывающее окно проигрывателя, однако в этом окошке отображается только название текущей композиции и несколько элементов управления. Я могу выбирать музыку в главном окне, от которого отделилось всплывающее окно проигрывателя (или даже в любом другом окне, принадлежащим тому же веб-сайту радио, как мы вскоре увидим). Уверен, вы уже слушали таким способом музыку через Сеть, и это приемлемый способ сохранения окна проигрывателя открытым все время.

ПРИМЕЧАНИЕ

Если вы молоды и проживаете не в Великобритании, то, скорее всего, никогда не слышали о Smashie и Nicey — загляните на YouTube и поищите!

Мы запустили проигрыватель с сайта и продолжаем просматривать его страницы. В какой-то момент мы решаем послушать Кэти Пэрри, резко оборвав вещание Smashie и Nicey. Но возникает проблема: мы нажали кнопку Play в главном окне, но во всплывающем окошке не появилась информация о том, что теперь мы слушаем «I Kissed a Girl». События хранилища сохраняют определенный веб-сайт.

Можно применять возможности API хранилища для проверки, открыто ли всплывающее окно, чтобы случайно не включился второй звуковой поток (проверьте, вы не захотите одновременно слушать Кэти Пэрри и Smashie с Nicey).

На главном веб-сайте с помощью следующего кода можно отслеживать состояние всплывающего окна. В коде всплывающего окна (показан чуть далее) мы будем прослушивать событие, сообщающее, что была запущена новая песня, и своевременно обновлять информацию.

```

function popupPlayer() {
    if (!localStorage.playerOpen) {
        // сведено к "true", но когда окно закроется,
        // мы удалим значение.
        localStorage.playerOpen = true;

        // открыть всплывающее окно
    }
}

function play(song) {
    localStorage.currentlyPlaying = song.title;
    // запускается и воспроизводит песню в каком-то квази-DRM
    // потоковом режиме
}

function stop() {
    // когда песня заканчивается, или когда пользователь
    // останавливает музыку вручную,
    // содержимое всплывающего проигрывателя также должно обновляться
    localStorage.removeItem('currentPlaying');
}

В то же время в коде всплывающего окна, помимо воспроизведения аудиопотока нужно
также прослушивать событие хранилища, сообщающее, что значение currentPlaying по-
менялось. Тогда можно запускать новую веселую песню:
function handleStorage(event) {
    event = event || window.event; // для поддержки IE8
    if (event.newValue === null) { // событие удалено
        stopPlaying();
    } else {
        // начать воспроизведение и обновить информацию
        startPlaying(event.newValue);
    }
}

window.addEventListener('storage', handleStorage, false);
window.attachEvent('storage', handleStorage);

```

Очевидно, что существуют и другие ситуации применения данной функциональности, помимо сайта радио. Запомните только, что события хранилища позволяют отдельным окнам в одном домене обмениваться данными — в старые времена для реализации такой возможности вам пришлось бы немало потрудиться.

Обходные варианты

Поскольку API хранилища устроен относительно просто, его функциональность можно воспроизвести с помощью JavaScript — на случай, если API хранилища окажется недоступным.

ПРИМЕЧАНИЕ

Если вам хочется побольше узнать о заполнении, отправляйтесь в главу 12. Мы подождем вас здесь.

Для `localStorage` можно использовать файлы cookie, или, для лучшей поддержки, встроенные методы `userData` браузера, или даже объект Flash. Для `sessionStorage` используйте `заполнитель`, работающий со свойством `name` объекта `window`. В следующем фрагменте кода показано, как воспроизвести функциональность `sessionStorage` (и гарантировать, что данные будут доступны только текущему окну, а не «протекут» в другие, как бывает с файлами cookie), вручную реализовав каждый из методов API хранилища. Обратите внимание, что в коде подразумевается, что в вашем браузере есть поддержка JSON — либо с помощью стандартных методов, либо через библиотеку Дугласа Крокфорда.

```
if (typeof sessionStorage === 'undefined') {
    sessionStorage = (function () {
        var data = window.top.name ? JSON.parse(window.top..
        ~ name) : {};
        return {
            clear: function () {
                data = {};
                window.top.name = '';
            } ,
            getItem: function (key) {
                return data[key] || null;
            } ,
            key: function (i) {
                // не идеально, но работает
                var ctr = 0;
                for (var k in data) {
                    if (ctr == i) return k
                    else ctr++;
                }
            } ,
            removeItem: function (key) {
                delete data[key];
                window.top.name = JSON.stringify(data);
            } ,
            setItem: function (key, value) {
                data[key] = value+''; // превращаем значение
                ~ в строку
                window.top.name = JSON.stringify(data);
            }
        } ;
    })();
}
```

Проблема с реализацией `sessionStorage` вручную (как показано в предыдущем фрагменте) заключается в том, что у нас нет методов `setter` и `getter` (или, скорее, в том, что IE7 и более ранние версии не поддерживают их, а это, в действительности, единственный популярный устаревший браузер без встроенной поддержки `sessionStorage`). Это означает, что невозможно использовать, например, такой код: `sessionStorage.twitter = '@rem'`. Хотя технически он сработает, он не будет правильно зарегистрирован в объекте хранилища, и `sessionStorage.getItem('twitter')` не вернет результата. Обойти конкретную проблему можно, но самое главное, что обновление окна браузера приведет к утере данных. Помня это, и в зависимости от того, для какого браузера предназначен код (имеется в виду, нужно ли вручную реализовывать свой вариант хранилища), внутри команды разработчиков обязательно стоит договориться, можно ли ограничиться только методами `getItem` и `setItem`.

БАЗЫ ДАННЫХ SQL ДЛЯ ВЕБ

Еще один вариант хранилища, позволяющего записывать и обращаться к данным, — это базы данных SQL для веб. Как подразумевает название, речь идет об обычной базе данных, в которую можно отправлять запросы и объединять результаты. Если вы знакомы с SQL, то в API баз данных будет чувствовать себя как рыба в воде. С другой стороны, если вы ничего не знаете о языке запросов SQL и, в частности, об SQLite, учить вас я не собираюсь: для этого предназначены другие толстые и страшные книги. Веб-сайт SQLite со всеми необходимыми ресурсами находится по адресу <http://sqlite.org>.

Размер баз данных в спецификации указан несколько расплывчато. При создании новой базы данных предполагаемый максимальный размер должны указать вы сами как ее автор. Следовательно, это ваше дело — *оценить* ее в 2 Мбайт или в 20 Мбайт. Если предполагаемый размер базы данных оказывается больше размера хранилища по умолчанию, в Safari пользователю предлагается разрешить или запретить базе данных превысить размер по умолчанию. Opera и Google Chrome разрешают создавать базы данных любого размера, не выводя никаких предупреждений. Я настоятельно рекомендую вам относиться к размерам баз данных с большой осторожностью. В целом, по умолчанию браузеры ограничивают размер базы данных для каждого домена пятью мегабайтами. Теперь вы вполне обоснованно волнуетесь о размере базы данных SQL, но у меня для вас хорошие новости. В API баз данных SQL для веб все методы позволяют передавать обратные вызовы, которые исполняются после того, как совершается некое внутреннее волшебство SQL. Обратные вызовы — это характерная черта многих библиотек JavaScript, таких как jQuery. Если вы незнакомы с синтаксисом, обратный вызов выглядит примерно так (однако не беспокойтесь раньше времени, я помогу вам с последующими примерами):

```
transaction.executeSql(sql, [], function () {
    // здесь находится исполняемый код
});
```

Из-за самой природы обратных вызовов это также означает, что API баз данных — *асинхронный*. Будьте очень осторожны при реализации общения JavaScript и базы данных, так как ошибка может привести к тому, что события будут происходить в неверной последовательности. В то же время операторы SQL помещаются в очередь и исполняются по

порядку, и это дает хоть и небольшое, но все же преимущество в деле контроля над порядком исполнения: создавая таблицы, вы можете твердо знать, что на момент исполнения запросов они уже существуют.

ВНИМАТЕЛЬНЕЕ С КОНТРОЛЕМ ВЕРСИЙ!

В настоящее время реализации баз данных SQL для веб поддерживают более старую версию API и, в частности, старую модель контроля версий.

Хотя спецификация описывает управление и миграцию с разных версий базы данных, реализовано это пока что не очень хорошо. Согласно требованиям модели, вы должны знать версию базы данных на машине пользователя, чтобы суметь открыть ее. Проблема в том, что если вы сменили несколько версий собственной базы данных, становится невозможно определить, какая версия установлена на компьютере пользователя, а попытка открыть базу данных с неправильным номером версии возвращает ошибку INVALID_STATE_ERROR. Можно обернуть все попытки открытия базы данных в соответствующие ловушки try/catch, но это потребует отдельной обертки для каждой версии — представьте, во что превратится код через несколько лет обновлений.

Проще говоря, если вы хотите, чтобы ваш код был выполнен после завершения обмена информацией с базой данных, то используйте обратный вызов. Если ждать вам не нужно и исполнение кода не должно задерживаться, то продолжайте после вызова API базы данных.

Работа с API баз данных SQL для веб

Типичная схема работы с API базы данных включает открытие базы данных и исполнение какого-то кода SQL. Обратите внимание, что если бы я работал с базой данных на серверной стороне, мне нужно было бы закрыть соединение с базой данных. В API баз данных это не требуется, и, если уж на то пошло, соответствующего метода вовсе не предусмотрено. Это означает, что базу данных можно открыть несколько раз безо всяких неприятных последствий.

Открытие и создание баз данных

При первом открытии базы данных происходит ее создание. В любой момент времени может существовать только одна версия именованной базы данных для каждого домена: если вы создадите версию 1.0, то не сможете открыть версию 1.1, если только версия базы данных не будет явно изменена приложением. Я не собираюсь больше возвращаться к проблеме версий вплоть до конца главы, и для простоты изложения буду придерживаться варианта с единственной версией, однако не забывайте о предыдущем предупреждении.

```
var db = openDatabase('mydb', '1.0', 'Моя первая
- база данных', 2 * 1024 * 1024);
```

В новейшей спецификации баз данных SQL у openDatabase пять аргументов, но пока что ни один из браузеров такое количество аргументов не поддерживает. В спецификации предлагается использовать обратный вызов при первом создании базы данных. Итак, вы создали новую базу данных с именем mydb, версии 1.0, с текстовым описанием "Моя первая база данных", а ее размер составляет 2 Мбайт (размер нужно указывать в байтах, поэтому я умножил три значения, $2 \times 1024 \times 1024$). Для того чтобы наше приложение работало

и понимало, поддерживается ли API баз данных SQL для веб, в браузере следует также выполнить соответствующий тест. Для этого `openDatabase` нужно обернуть оператором `if`:

```
var db;
if (window.openDatabase) {
    db = openDatabase('mydb', '1.0', 'Моя первая
        база данных', 2 * 1024 * 1024);
}
```

Все очень просто. Дальше мы создадим в базе данных новую таблицу, для чего воспользуемся тем же способом, что и для выбора и обновления таблиц: через `executeSql`.

Создание таблиц

При создании таблиц (это относится и к любым другим операциям в базе данных) нужно сначала запустить «транзакцию» с базой данных, а потом в обратном вызове выполнить соответствующий SQL-код. Обратный вызов транзакции получает аргумент, содержащий объект транзакции, и именно он позволяет исполнять операторы SQL и использовать метод `executeSql` (в следующем примере — `tx`). Объект базы данных `db` — это объект, полученный от `openDatabase`, а метод транзакции вызывается так:

```
var db;
if (window.openDatabase) {
    db = openDatabase('tweetdb', '1.0', 'All my tweets',
        2 * 1024 * 1024);
    db.transaction(function (tx) {
        tx.executeSql('CREATE TABLE tweets (id, date, tweet)');
    });
}
```

Метод `executeSql` принимает четыре аргумента, из которых здесь используется только первый:

1. Запрос SQL.
2. Аргументы запроса SQL (например, значения полей).
3. Обратный вызов при успешном завершении.
4. Обратный вызов при ошибке.

В предыдущем примере используется только первый параметр, запрос SQL. Конечно, если при создании таблицы выясняется, что такая таблица уже существует, то возвращается ошибка, но так как вы не отлавливаете ее и на исполнение программы она не влияет, в данном случае такое развитие событий вас не волнует.

Однако на следующем шаге приложения нам нужно будет загрузить в базу данных сообщения из Twitter, а для этого таблица должна существовать (вспомните об асинхронной природе API баз данных SQL для веб). Следовательно, эти сообщения должны загружаться в обратном вызове. Здесь и кроется проблема: если таблица уже существует, предыдущая транзакция завершится ошибкой, и обратный вызов из третьего аргумента выполнен не будет. Код сможет успешно выполниться только один раз; во второй раз ничего не получится. Чтобы обойти эту сложность, нужно запрограммировать создание таблицы только в том случае, если она еще не существует. Таким образом, обратный вызов, соответствую-

щий успешному завершению, будет выполнен и для только что созданной таблицы, и для существующей, а обратный вызов для ошибки будет выполнен только в случае каких-то других неполадок.

```
var db;
if (window.openDatabase) {
    db = openDatabase('tweetdb', '1.0', 'All my tweets',
        - 2 * 1024 * 1024);
    db.transaction(function (tx) {
        tx.executeSql('CREATE TABLE IF NOT EXISTS tweets
            - (id, date, tweet)', [], function () {
            // теперь можно загружать в таблицу
            - сообщения
        });
    });
}
});
```

Вставка и опрос

Предположим, что вы нашли в Twitter все упоминания HTML5 и сохранили эти сообщения в своей базе данных, а пользователю предоставляется возможность выбрать временной диапазон создания интересующих его сообщений — последние 5 минут, 30 минут, 2 часа или без ограничения времени создания. Время создания будет выбираться переключателями с обработчиками нажатий, и ваш запрос будет возвращать только сообщения, соответствующие выбранному временному промежутку.

Самая большая сложность этого приложения заключается в разделении двух задач: сохранения сообщений в базе данных и отображения сообщений в зависимости от времени создания.

Перед выполнением любого кода нужно создать базу данных и таблицу сообщений, причем в таблице должен присутствовать столбец дат целочисленного типа — это важно, чтобы в последующих разделах приложения можно было реализовать опрос базы данных:

```
function setupDatabase() {
    db = openDatabase('tweets', '1.0', 'db of tweets',
        - 2 * 1024 * 1024);
    db.transaction(function (tx) {
        tx.executeSql('CREATE TABLE tweets (id unique,
            - screen_name, date integer, text)');
    });
    getTweets();
}
```

Несколько замечаний относительно кода:

1. Я использую глобальную переменную `db`. (Я использую ее только в выдуманном примере. Вообще-то, в JavaScript не рекомендуется использовать глобальные переменные.)
2. Я сообщаю базе данных `tweets`, что столбец `id` должен содержать уникальные значения. Это означает, что оператор `INSERT` с дублирующимися данными вернет ошибку.
3. `CREATE TABLE` может вернуть ошибку, только если такая таблица уже существует, ведь в этой транзакции вы больше ничего не делаете.

4. Закончив с подготовкой, я вызываю `getTweets`, которая заставит API опросить Twitter, а API, в свою очередь, вызовет функцию сохранения данных.

ПРИМЕЧАНИЕ

Вы создаете новую транзакцию для каждого сообщения, которое записывается в базу данных. Подробнее о транзакциях я поговорю в следующем разделе, но, обворачивая каждый оператор `INSERT`, вы гарантируете, что будут сохранены все новые сообщения, независимо от того, есть они уже в базе данных или нет.

5. А что произойдет, если `getTweets` выполнится до того, как таблица будет создана? Это не имеет значения. Когда мы получаем записи, создается новая транзакция, которая вставляет новый SQL-код. Транзакции выполняются в том порядке, в котором они отправляются в базу данных. Следовательно, даже если оператор `create table` не успел отработать до того, как мы собираемся с помощью новой транзакции вставить новые строки, мы можем быть уверены, что в очереди он все равно окажется впереди этой транзакции, и все произойдет в нужном порядке.
6. Метод `forEach` в следующем фрагменте кода — это новый метод JavaScript, доступный в последних версиях браузеров. Он позволяет пройти по содержимому массива от начала до конца. На сайте Mozilla можно найти простой код для реализации той же функциональности в браузерах, где этого метода нет: https://developer.mozilla.org/en/Core_JavaScript_1.5_Reference/Global_Objects/Array/foreach. После того как отрабатывает вызов API Twitter, он вызывает `saveTweets`; эта функция сохраняет все найденные сообщения:

```
function saveTweets(tweets) {
  tweets.results.forEach(function (tweet) {
    db.transaction(function (tx) {
      var time = (new Date(Date.parse(tweet.created_at))).getTime();
      tx.executeSql('INSERT INTO tweets (id, screen_name, date, text) VALUES (?, ?, ?, ?)', [tweet.id, tweet.from_user, time / 1000, tweet.text]);
    });
  });
}
```

Самая важная часть здесь — это оператор `INSERT`. Теперь вы видите, как работают аргументы поля:

```
tx.executeSql('INSERT INTO tweets (id, screen_name, date, text) VALUES (?, ?, ?, ?)', [tweet.id, tweet.from_user, time / 1000, tweet.text]);
```

Каждый вопросительный знак в операторе `INSERT` соответствует элементу массива, который передается в `executeSql` в качестве второго параметра. Следовательно, первый «?» соответствует `tweet.id`, второй — `tweet.from_user` и т. д.

Также вы видите, что я разделил значение времени на 1000. Время в JavaScript определяется с точностью до миллисекунд, тогда как SQLite понимает только целые секунды. Нам это понадобится дальше, для отображения сообщений, созданных не более пяти минут назад. Даты мы сохраняем как целые числа. Для метода `getTime` в JavaScript одна секунда соответствует значению 1000, а для SQLite одна секунда — это 1. Следовательно, мы делим значение на 1000, чтобы записать секунды, а не миллисекунды.

Наконец, когда пользователь щелкает переключатель, мы вызываем функцию `show`, передавая ей время в качестве аргумента:

```
var tweetEl = document.getElementById('tweets');
function show(amount) {
    db.transaction(function (tx) {
        tx.executeSql('SELECT * FROM tweets' + (amount != '- all' ? ' WHERE date > strftime("%s", "now", "-' +
            amount + ' minutes")' : ''), [], function
        (tx, results) {
            var html = [],
                len = results.rows.length;

            for (var i = 0; i < len; i++) {
                html.push('<li>' + results.rows.item(i).text +
                    '</li>');
            }
            tweetEl.innerHTML = html.join('');
        } );
    });
}
```

На первый взгляд код может выглядеть слишком сложно, но в действительности там всего пары вещей, на которые стоит обратить внимание:

- 1) запуск новой транзакции;
- 2) исполнение единственного оператора SQL, структура которого определяется тем, хотите вы получить все или нет;
- 3) просмотр результатов и составление кода HTML, который потом помещается в `tweetEl` (элемент ``) `innerHTML`.

У нас может быть два варианта запроса SQL:

```
SELECT * FROM tweets
```

или

```
SELECT * FROM tweets WHERE date > strftime("%s", "now",
    "-5 minutes")
```

Я указал -5 минут, но в действительности это может быть -30 минут или любое другое значение, переданное функции `show`. Функция SQLite `strftime` генерирует число секунд, прошедшее с 1 января 1970 года до текущего момента минус N минут. Так как поле даты у нас целочисленное, это позволяет извлечь все сообщения, написанные в течение последних N минут.

Итак, мы использовали третий аргумент метода `executeSql`, обратный вызов для случая успешного завершения. Этот обратный вызов получает объект транзакции (так же, как и обратный вызов транзакции, что позволяет при необходимости выполнить еще один метод `executeSql`) и, что еще важнее, результирующий набор. Результирующий набор включает три атрибута:

- 1) `insertId` (устанавливается, только если вы вставили одну или несколько строк) — в данном примере я его не использовал;
- 2) `rowsAffected` — так как это оператор `SELECT`, данное значение равно 0;
- 3) `rows` — это *не* массив, это коллекция, в которой есть указание на ее длину и содержитится метод извлечения элемента. Мы работаем с объектом строк в цикле `for` от 0 до значения, равного длине коллекции, а получить отдельные строки нам позволяет `results.rows.item(i)`. Отдельная строка — это объект, представляющий столбы с различными названиями; таким образом, `results.rows.item(0).screen_name` дает нам содержимое поля `screen_name` из первой строки.

Наконец, после того как мы прошли по всем подходящим строкам, можно сформировать элемент `` для HTML-кода. В нашем примере элемент `` содержится в переменной с именем `tweetEl`.

Далее приведен полный листинг кода, включающий распознавание поддержки баз данных и код обработки щелчков для переключателей:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset=utf-8 />
<title>HTML5 tweet time range</title>
<style>
    body { font-family: helvetica, arial; }
</style>
</head>
<body>
    <form>
        <fieldset>
            <legend>Select a time range of recent HTML5 tweets</
            ~ legend>
            <input type="radio" value="5" id="t5m" name="timerange"
            ~ /><label for="t5m">5 minutes</label>
            <input type="radio" value="30" id="t30m" name=
            ~ "timerange" /><label for="t30m">30 minutes</label>
            <input type="radio" value="120" id="t2h" name=
            ~ "timerange" /><label for="t2h">2 hours</label>
            <input type="radio" value="all" id="tall" name=
            ~ "timerange" checked="checked" /><label for="tall">
            ~ all time</label>
        </fieldset>
    </form>
    <ul id="tweets"></ul>
<script>
```

```

var tweetEl = document.getElementById('tweets');
var db;

function setupDatabase() {
    if (!window.openDatabase) {
        tweetEl.innerHTML = '<li>Web SQL Database API is not
            ~ available in this browser, please try nightly Opera,
            ~ Webkit or Chrome.</li>';
        return;
    }
    db = openDatabase('tweets', '1.0', 'db of tweets',
        ~ 2 * 1024 * 1024);
    db.transaction(function (tx) {
        tx.executeSql('CREATE TABLE tweets (id unique,
            ~ screen_name, date integer, text)');
    });
    getTweets();
}

function getTweets() {
    var script = document.createElement('script');
    script.src = 'http://search.twitter.com/search.
        ~ json?q=html5 -RT&rpp=100&callback=saveTweets';
    document.body.appendChild(script);
}

// функция обратного вызова API Twitter
function saveTweets(tweets) {
    tweets.results.forEach(function (tweet) {
        db.transaction(function (tx) {
            var time = (new Date(Date.parse(tweet.created_at))).
                ~ getTime();
            tx.executeSql('INSERT INTO tweets (id, screen_name,
                ~ date, text) VALUES (?, ?, ?, ?)', [tweet.id,
                ~ tweet.from_user, time / 1000, tweet.text]);
            ~ // divide by 1000 to get to seconds
        } );
    });
}

function show(amount) {
    db.transaction(function (tx) {
        tx.executeSql('SELECT * FROM tweets' + (amount !=
            ~ 'all' ? ' WHERE date > strftime("%s", "now", "-" +
            ~ amount + ' minutes")' : ''), [], function
            ~ (tx, results) {
                var html = [],
                    len = results.rows.length;

                for (var i = 0; i < len; i++) {

```

продолжение ➔

```

        html.push('<li>' + results.rows.item(i).text +
        - '</li>');
    }
    tweetEl.innerHTML = html.join('');
} );
});
}

// привязываем обработчики нажатий для переключателей
[].forEach.call(document.querySelectorAll('input
- [type=radio]'), function (el) {
    el.onclick = function () {
        show(this.value);
    }
});
};

// вперед!
setupDatabase();

</script>
</body>
</html>

```

Создание транзакций — и для чего они нужны

Пока что я касался транзакций лишь поверхностно. В действительности, мы видим только вершину айсберга; транзакции намного сложнее и глубже, чем может показаться на первый взгляд. Это не просто способ выполнения запросов; они служат определенной и очень важной цели. Транзакция — это словно закрытая среда, в которой вы исполняете свои запросы. Внутри транзакции можно выполнить один запрос или целую группу. Если честно, запрос вообще *невозможно* выполнить, не находясь *внутри* транзакции, так как метод `executeSql` доступен *только* из объекта `SQLTransaction`.

Возможно, наиболее важная особенность транзакций заключается в том, что если что-то внутри транзакции завершается ошибкой (простой код или операторы SQL), то откатывается вся транзакция, целиком (включая вставки, модификации и удаление). Как будто никакой код, входящий в транзакцию, никогда не выполнялся.

Метод `transaction` принимает два аргумента: первый — это содержимое транзакции; второй, необязательный, — это обработчик ошибок для случаев, когда он необходим. Следующий код ошибочный, он служит только для примера, как откатываются транзакции:

```

var db = openDatabase('foo', '1.0', 'foo', 1024);
db.transaction(function (tx) {
    tx.executeSql('CREATE TABLE foo (id unique, text)');
    tx.executeSql('INSERT INTO foo (id, text) VALUES
    - (1, "foobar")');
});

db.transaction(function (tx) {
    tx.executeSql('DROP TABLE foo');
}

```

```
// здесь ошибка, поэтому для оператора DROP будет выполнен откат
tx.executeSql('INSERT INTO foo (id, text) VALUES
  ↵ (1, "foobar")');
}, function (error) {
  // error.message выглядит как "no such table: foo"
  alert('Rollback triggered, the table "foo" was never
    ↵ dropped due to: ' + error.message);
});

db.transaction(function (tx) {
  tx.executeSql('SELECT * FROM foo', [], function (tx,
    ↵ results) {
    alert('found ' + results.rows.length + ' row');
  }, function (tx, error) {
    // это исполняться не будет никогда
    alert('something went wrong: ' + error.message);
  });
});
```

Итак, что мы делаем в предыдущем коде:

- 1) начинаем транзакцию, которая создает таблицу `foo` и вставляет в нее одну строку;
- 2) начинаем транзакцию, которая удаляет таблицу `foo`, а затем ошибочно пытается вставить в таблицу `foo` новую строку;
- 3) эта транзакция завершается ошибкой, происходит откат операторов (то есть шаг 2 словно бы никогда и не происходил);
- 4) начинаем транзакцию, выбирающую все строки из таблицы `foo` и сообщающую число строк;
- 5) запрос SQL завершается успешно, и мы видим сообщение «найдено 1 строк».

Транзакции нужны для разбиения запросов на блоки. Такой блок выполняется и, если любая часть внутри него дает сбой, откатывается целиком.

INDEXEDDB

Когда первое издание этой книги готовилось к публикации, об IndexedDB еще только говорили, и примеров реализации в то время еще не было. На сегодняшний день существует единственная реализация, использующая вендорный браузерный префикс. Тем не менее я подозреваю, что совсем скоро большинство, если не все браузеры, начнут успешно поддерживать IndexedDB.

IndexedDB — это хранилище объектов документов. Оно в чем-то похоже на базу данных, но в данном случае вы не получаете в нагрузку SQL и прочие сложности реляционной базы данных.

Используя IndexedDB, вы создаете новую базу данных и присваиваете ей имя и версию, для того чтобы ее можно было повторно открыть позже. Затем вы создаете хранилище объектов — это то же самое, что добавить в книжный шкаф закладки, для того чтобы быстро просмотреть и найти нужную книгу. Когда хранилище готово, с индексом, который вы используете для заполнения, можно сохранить объект любого типа. Неважно, что он

содержит, и он не обязательно должен обладать такими же свойствами, как другие объекты. Кстати, именно на этом моменте в SQL зачастую возникают проблемы. Если таблица очень велика, а вам нужно добавить один крохотный незаметный столбец, это означает огромную головную боль. С IndexedDB у вас таких горестей не будет.

ИСПОЛЬЗОВАНИЕ БЕТА-ВЕРСИИ INDEXEDDB

К моменту, когда вы будете читать эту главу, разработка IndexedDB все еще может оставаться на стадии, когда необходимо использование вендорных префиксов.

В таком случае вы все равно можете работать с IndexedDB, но в коде придется создавать разветвления, обрабатывающие разные принципы именования:

```
function importIndexedDB(prefix) {
    var indexedDB = window[prefix + 'IndexedDB'],
        IDBTransaction = window[prefix + 'IDBTransaction'];

    if (indexedDB !== undefined) {
        window.indexedDB = indexedDB;
    }
    if (IDBTransaction !== undefined) {
        window.IDBTransaction = IDBTransaction;
    }
}

// попробуем все вендорные префиксы
['moz webkit o ms'].split(' ').forEach(function (vendor) {
    importIndexedDB(vendor);
});
```

Создание новых баз данных IndexedDB

Первым делом нужно открыть новую индексированную базу данных. Возвращенное значение вы будете использовать для создания хранилищ объектов и обработки ошибок — аналогично тому, как при работе с базой данных SQL для веб вы получаете объект базы данных, а затем открываете базу данных.

Однако в общении с IndexedDB каждый процесс представляет собой *запрос*. Как и в случае с базами данных SQL для веб, обмен данными всегда происходит в асинхронном режиме. Очень часто возвращаемому значению присваивают имя *request*, и мы поступим таким же образом:

```
var request = indexedDB.open('videos');

request.onerror = function () {
    console.log('failed to open indexedDB');
};

request.onsuccess = function (event) {
    // обработка контроля версий
    // затем создание нового хранилища объектов
};
```

Итак, база данных открыта, и, при условии, что ошибки не произошло, сработает событие `onsuccess`. Однако прежде, чем создавать новое хранилище объектов, нужно сделать две вещи:

- сохранить фактический обработчик базы данных, чтобы выполнять транзакции по получению и записи данных;
- установить версию базы данных. Если версия не указана, это означает, что базу данных только что впервые создали (установка версии производится способом, рассмотренным в следующих примерах).

Обработчик события успеха передает объект события, похожий на объект события, который вы бы получили, если бы прослушивали событие нажатия. Внутри события можно найти свойство под названием `target`, внутри которого, в свою очередь, хранится `result`. Свойство `result` содержит, как вы уже наверняка догадались, результат данного конкретного вызова функции. В нашем случае, `event.target.result` содержит обработчик открытой базы данных, то есть нашей базы данных `videos`.

Вот как обновляется обработчик `onsuccess`:

```
var db = null;

request.onsuccess = function (event) {
    // кэшируем копию обработчика базы данных для
    // использования в будущем
    db = event.target.result;
    // обработка контроля версий
    // затем создание нового хранилища объектов
};

request.onerror = function (event) {
    alert('Something failed: ' + event.target.message);
};
```

Обратите внимание на обработчик ошибок. В IndexedDB ошибки всплывают из запроса в транзакцию, а затем в саму базу данных. Это означает, что если в любой момент во время выполнения запроса произойдет ошибка, вы увидите окно сообщения. Однако помните: окна сообщений уродливы и не дружат с браузерами, особенно в промышленных условиях — они злят сотрудников отделов продаж. Превратите свое окно сообщения в элегантное окошко со скругленными углами, которое изящно обработает ошибку. А теперь перейдем к инициализации базы данных.

Контроль версий

После открытия соединения с базой данных в первую очередь необходимо заняться контролем версий. В качестве номера версии можно использовать любую строку, но если выбранная вами строка позволяет понятным и логичным образом увеличивать значение версии, в будущем это сэкономит вам много времени. Давайте назовем версию нашего приложения «`0.1`». Во время обновления мы сможем сравнить значение «`0.1`» (строку), представляющее версию базы данных, со значением «`0.2`», представляющим код обновления. Так как мы выполняем код впервые и даже пытаемся открыть новую базу данных, давайте проверим версию, и если значение не совпадает, то зададим номер версии и создадим хранилища

объектов. Должен предупредить, что в данном случае контроль версий не похож на SVN или Git — в случае необходимости изменить число хранилищ объектов в базе данных вам также потребуется запросить изменение версии.

```
var db = null,
    version = '0.1';

request.onsuccess = function (event) {
    // кэшируем копию обработчика базы данных для
    // использования в будущем
    db = event.target.result;

    // handle version control
    if (version != db.version) {
        // установка номера версии: 0.1
        var verRequest = db.setVersion(version);
        verRequest.onsuccess = function (event) {
            // все готово к созданию хранилища объектов!
        } ;
        verRequest.onerror = function () {
            console.log('unable to set the version :(');
        } ;
    }
};
```

После того как срабатывает событие успеха, можно создавать новые хранилища объектов с помощью метода `setVersion`.

Создание хранилищ объектов

Во время контроля версии внутри обработчика события успеха создайте новые хранилища объектов, добавив такой код:

```
var verRequest = db.setVersion(version);
verRequest.onsuccess = function (event) {
    var store = db.createObjectStore('blockbusters', {
        keyPath: 'title',
        autoIncrement: false
    });
    // в этот момент мы уведомим наш код о том, что
    // хранилище объектов готово
};
```

Для данного приложения мы создали одно хранилище объектов, но в следующей версии, возможно, мы добавим в нашу базу данных видеофильмов хранилище объектов для имен режиссеров. Относительно метода `createObjectStore` важно обратить внимание на то, какие аргументы настроек мы передали. Мы сообщаем хранилищу объектов, что нам требуется индекс (с помощью которого мы сможем извлечь наш блокбастер) и что идентификаторы не должны автоматически увеличиваться. На самом деле, флаг `autoIncrement` принимает значение `false` по умолчанию; я явно добавил его просто для того, чтобы показать, как он используется.

Теперь, когда дело дойдет до сохранения новых объектов, я должен буду удостовериться, что свойство `title` у фильма уникальное, так как оно будет индексироваться в IndexedDB для последующего извлечения.

Я решил, что мы будем хранить имена режиссеров вместе с данными о фильмах-блокбастерах, а не в отдельном хранилище объектов. Кроме того, мне нужна возможность поиска по режиссеру, поэтому в хранилище данных следует добавить еще один индекс:

```
store.createIndex('director', 'director', { unique: false });
```

В этой строке мы присвоили индексу имя (первый аргумент), а затем указали имя свойства (в данном случае, `'director'`), по которому необходимо проводить индексацию при сохранении новых объектов. Помимо этого, мы допускаем, что у одного режиссера будет несколько фильмов, поэтому не требуем, чтобы значения были уникальными.

Все это означает, что теперь можно с легкостью сохранить и извлечь запись, описывающую фильм-блокбастер, например, такую:

```
{
  title: "Belly Dance Bruce - Final Strike",
  date: (new Date).getTime(), // на экранах СЕГОДНЯ!
  director: "Bruce Awesome",
  length: 169, // в минутах
  rating: 10,
  cover: "/images/wobble.jpg"
}
```

Итак, давайте добавим описания фильмов из нашей коллекции.

Добавление и помещение объектов в хранилища

Ага, «добавление и помещение». Не сомневаюсь, вы думаете, что это какая-то нелепая двусмысленность! Существует два метода вставки данных: `add` и `put`. Первый добавляет *новые* данные и требует, чтобы до этого в базе таких данных не было. Второй метод вставляет в хранилище *обновленный* объект, но если такого объекта в базе нет, то он добавляется как новый.

Что касается нашей базы данных с информацией по фильмам, мы не будем обращать внимания на предупреждение и просто воспользуемся методом `put`. Если вы не настолько бессердечны, к тому же знаете, что существует риск возникновения дублирующихся объектов (например, названия двух фильмов совпадают, но различаются другие поля — в таком случае будут созданы два объекта, хотя мы ожидаем, что для каждого названия будет один уникальный объект), добавьте в свой код оба метода, `add` и `put` с соответствующей валидацией и проверками.

```
var video = {
  title: "Belly Dance Bruce - Final Strike",
  date: (new Date).getTime(),
  director: "Bruce Awesome",
  length: 169,
  rating: 10,
  cover: "/images/wobble.jpg" },
READ_WRITE = IDBTransaction.READ_WRITE
```

```
var transaction = db.transaction(['blockbusters'], READ_WRITE),
    store = transaction.objectStore('blockbusters'),
    request = store.put(video);
```

В последних трех строках кода происходит довольно много интересного, и если ход наших мыслей совпадет, то вам наверняка кажется, что он полон бессмысленных повторений.

В действительности, мы выполняем три разные задачи, описанные ниже.

1. Создание транзакции

```
transaction = db.transaction(['blockbusters'], READ_WRITE)
```

Первая задача — создать новую транзакцию с разрешениями на чтение и запись в именованные хранилища объектов. Транзакция может быть привязана к нескольким хранилищам объектов — вот почему здесь мы передаем массив. На практике я обнаружил, что можно передать значение `null` или даже пустую строку в качестве аргумента, описывающего имя хранилища объектов. Не уверен, что по замыслу создателей это допускается и должно работать, но, как я уже говорил раньше, базы IndexedDB появились совсем недавно и пока что используются только с вендорными префиксами. В любом случае, лучше придерживаться правил, определенных в спецификации.

Если бы мы планировали только считывать данные в этой транзакции, то могли бы использовать `IDBTransaction.READ_ONLY`.

ОДИН ДЕНЬ В КОРОТКОЙ ЖИЗНИ ТРАНЗАКЦИИ

У объектов транзакции очень короткий срок жизни, связанный с циклом событий в браузере. Если создать транзакцию и не использовать ее, после возвращения к циклу событий транзакция тут же умрет и будет бесполезна.

Если же выполнить запрос сразу же и если он завершится успешно, то в ту же транзакцию можно поместить и следующий запрос, не сомневаясь в ее пригодности и работоспособности. При условии, что цикл событий не разорвется, можно продолжать так делать до бесконечности.

Если вы когда-нибудь использовали в своем коде `setTimeout(fn, 0)`, это значит, что вы «отпускали» цикл событий — например, чтобы браузер смог перекрасить страницу или сделать что-то еще.

Это уникальная концепция, с которой я раньше не встречался. Определенно, с жизненным сроком транзакций стоит поэкспериментировать. Разобравшись, в каких условиях транзакция сохраняется живой, а в каких умирает, вы сможете сделать так, чтобы ваши транзакции могли потягаться в жизнеспособности даже с зомби.

2. Получение ссылки на хранилище

```
store = transaction.objectStore('blockbusters')
```

Здесь мы получаем жесткую привязку к хранилищу объектов, с которым собираемся работать, и здесь также необходимо задать имя хранилища. Это может быть любое из хранилищ, перечисленных при создании транзакции. Получив ссылку на хранилище, можно выполнять различные действия, такие как добавление, помещение и даже извлечение данных — об этом мы поговорим через мгновение.

3. Сохранение данных

```
request = store.put(video)
```

Переменная `request` получит связанное с объектом событие успеха или ошибки. Возможно, меня волнует, правильно ли был сохранен объект. А может, я хочу, чтобы код продолжал выполняться, а прыщеватый пятнадцатилетний клерк продолжать добавлять в базу данных пачку описаний фильмов Брюса Великолепного. Это приятная особенность *асинхронности* IndexedDB — веб-сайт может продолжать работу, и его не будут прерывать различные события, происходящие во время сохранения данных.

Итак, мы сохранили данные, как же теперь их извлечь?

Выдергиваем их обратно

Если вы внимательно изучили процесс сохранения данных, то процесс их извлечения покажется вам очень простым и знакомым.

Все так же в начале необходимо создавать транзакцию. Поскольку мы всего лишь извлекаем данные, транзакции можно было бы дать права только на чтение, но в действительности это неважно, так что мы используем права на чтение и запись. Мы так же получаем хранилище объектов, но вместо добавления или помещения данных мы *извлекаем* их:

```
var transaction = db.transaction(['blockbusters'], READ_WRITE),
    store = transaction.objectStore('blockbusters'),
    request = store.get(key);
```

Важно отметить, что переменная `key`, которую мы передаем методу `get`, сравнивается с `keyPath`, определенным при создании хранилища объектов. В нашем примере с базой данных по фильмам мы решили, что ключ записей о блокбастерах будет зависеть от названий фильмов.

А как извлечь данные обо *всех* фильмах из хранилища? Одного метода `get` недостаточно. Нужно пройти по всему хранилищу данных:

```
var transaction = db.transaction(['blockbusters'], READ_WRITE),
    store = transaction.objectStore('blockbusters'),
    data = [];

var request = store.openCursor();

request.onsuccess = function (event) {
    var cursor = event.target.result;
    if (cursor) {
        // значением является сохраненный объект
        data.push(cursor.value);
        // получить следующий объект
        cursor.continue();
    } else {
        // теперь у нас есть все данные, можно совершить
        // обратный вызов, сообщающий об успехе,
        // и передать объект данных.
    }
};
```

В этом фрагменте кода мы открываем хранилище объектов обычным способом, но вместо того чтобы выполнять метод `get`, мы открываем курсор (`cursor`). Он позволяет в цикле пройти по всем сохраненным объектам. Аналогичным образом можно было бы с легкостью найти все фильмы с рейтингом выше пяти звезд, нужно только добавить вложенную проверку значения `cursor.value.rating`, прежде чем помещать текущий объект из хранилища в массив результатов `data`.

Например:

```
function find(filter, callback) {
    // разрешение READ_WRITE было объявлено ранее в коде
    var transaction = db.transaction(['blockbusters'],
        - READ_WRITE),
        store = transaction.objectStore('blockbusters'),
        data = [];

    var request = store.openCursor();

    request.onsuccess = function (event) {
        var cursor = event.target.result;
        if (cursor) {
            if (filter(cursor.value) === true) {
                // значением является сохраненный объект
                data.push(cursor.value);
            }
            // получить следующий объект
            cursor.continue();
        } else {
            callback(data);
        }
    };
}

// пример использования
find(function (data) {
    return data.rating > 5;
}, function (found) {
    alert('Found ' + found.length + ' videos with a high
        - rating');
});
```

Удаление данных

Что, если вы сделали ошибку или хотите удалить данные? Возможно, на вас посыпались жалобы из-за последнего фильма Брюса, «Битье бананов, розовая ярость», и эту запись необходимо выкинуть из хранилища.

Процесс такой же, как для метода `get`, за исключением того, что при передаче ключа вызывается метод `delete` (обратите внимание, что для этого требуется разрешение на запись):

```
var transaction = db.transaction(['blockbusters'], READ_WRITE),
    store = transaction.objectStore('blockbusters'),
    request = store.delete(key);
```

Также предусмотрен метод для полной очистки хранилища объектов, `clear`. Он не требует никаких аргументов, а процесс его применения аналогичен.

Существует еще два способа удаления данных: `deleteObjectStore` и `deleteDatabase`. Из названий методов понятно, что они делают — удаляют хранилище объектов и базу данных соответственно, но применять их не так просто.

`deleteObjectStore` может выполняться только внутри транзакции. Этот метод привязывается к объекту базы данных (который возвращается методом `indexedDB.open`). Однако невозможно удалить хранилище объектов с помощью обычной транзакции, в этом вам может помочь *только* транзакция установки версии. Чуть выше в этой главе мы поднимали этот вопрос, когда я показывал, как создавать хранилища объектов. Только имея обработчик успеха `setVersion`, можно добавлять или удалять хранилища.

Наконец, `deleteDatabase`. Этот метод был бы крайне полезен для отладки, но, к сожалению, он еще не реализован ни в одном из браузеров. Надеюсь, по мере расширения спецификации положение дел исправится, но пока что он здорово усложняет отладку — с помощью имеющихся средств к исходной точке вернуться крайне сложно.

УДАЛЕНИЕ БАЗЫ ДАННЫХ INDEXEDDB В CHROME

Наверное, существуют тайные способы, как сделать это в любом браузере, но я обнаружил, как удалить базу данных, роясь в глубинах файлов Chrome.

Перейдите к папке `~/Library/Application Support/Google/Chrome/Default/IndexedDB` на Mac или `C:\Users\<you>\AppData\Local\Google\Chrome\User Data\Default\IndexedDB\` в Windows 7 (о других операционных системах рассказывается на <http://goo.gl/v702q>).

Оказавшись в каталоге IndexedDB, вы увидите базы данных, перечисленные по доменам. Для того чтобы очистить базу данных, удостоверьтесь, что Chrome закрыт, и просто удалите файл.

Но делайте это на свой страх и риск! И не прибегайте ко мне жаловаться, если ваш браузер взорвется и забрызгает весь монитор! При этом должен отметить, что у меня данный способ до сих пор работал просто прекрасно.

Отладка

Отладка — тяжелая штука. На момент написания этой главы в IndexedDB отсутствуют два ключевых компонента:

- **нет инструментов отладки.** Для баз данных SQL для веб и веб-хранилищ прямо в веб-инспекторах браузеров (таких как ChromeDevTools и Opera Dragonfly) существуют визуальные инструменты. Для IndexedDB таких полезностей не предусмотрено, из-за чего отладка превращается в довольно-таки тяжелую работу. Это не невозможно — но очень сложно!
- **нет возможности удалить базу данных.** Пока что хранилища IndexedDB реализованы только в Firefox и Chrome, и в обоих до сих пор используются вендорные префиксы, так что подозреваю, что способ удаления базы данных — лишь один из пунктов длинного списка отсутствующей функциональности. Поскольку восстановление текущего состояния приложения — это обязательный шаг разработки, невозможность удалить базу данных здорово связывает руки. Можно обойти эту проблему, вручную удалив все хранилища объектов, а затем сбросив версию, но это не одно и то же!

Тем не менее мы говорим о недавно появившейся технологии, а простота использования IndexedDB много значит для разработчика, особенно если вспомнить базы данных SQL для веб. По крайней мере, если этот разработчик — я.

ЗАКЛЮЧЕНИЕ

В этой главе вы узнали о трех разных API для локального хранения данных браузера, которые с легкостью затыкают за пояс файлы cookie.

Новые API хранилищ позволяют записывать намного больше данных, чем традиционные файлы cookie, и значительно упрощают программирование. Главная новость — API веб-хранилища уже поддерживается всеми новейшими браузерами (а в старых браузерах его поддержку можно реализовать с помощью JavaScript).

По скромному мнению автора, несмотря на то что разработка баз данных SQL для веб остановлена, они просуществуют еще достаточно долго — в основном потому, что эту технологию используют такие гиганты, как Google, которая реализует на ее базе свою почтовую систему Gmail и, к тому же, производит браузеры.

Тем временем IndexedDB уже достаточно близко подошли к стабильной реализации и в итоге могут получить куда более широкую поддержку, чем базы данных SQL для веб. Однако не стоит забывать, что если на хранилища IndexedDB будет накладываться ограничение в 5 Мбайт, которое никак нельзя обойти, то разработчики наверняка посмотрят в сторону веб-хранилищ, которые тоже предлагают 5 Мбайт, но при этом работать с ними проще простого.

Что это значит? А то, что какой бы API вы ни выбрали, зачерствевшее печенье можно выбросить на свалку истории уже сегодня!

Глава 7

АВТОНОМНАЯ РАБОТА

Реми Шарп



Знаете, сколько раз я работал, находясь в пути, и отчаянно пытался подключиться к 3G-сети через дешевый USB-модем, а в самый ответственный момент подключение к нужному веб-приложению пропадало, потому что поезд въезжал в туннель? Очень, очень много раз. Если вы работаете на компьютере, не подключенном к Интернету, либо на компьютере с крайне нестабильным подключением, то, вероятнее всего, вы в дороге. Но это также может означать, что вам нужно поработать автономно. Чем мобильнее становимся мы и наши компьютеры, тем важнее становится возможность использовать веб-сайты даже там, где подключиться к Интернету попросту невозможно.

Мы привыкли создавать веб-приложения, полностью зависящие от сети. Наши веб-сайты работают в браузерах; а браузер — это окно в сеть! Та часть спецификации HTML5, где говорится об автономных веб-приложениях, убирает частицу «веб» из слова «веб-приложение». Браузер должен поддерживать локальный кэш, чтобы приложение могло работать и при отсутствии подключения к Интернету.

ВЫДЕРНите ВИЛКУ ИЗ РОЗЕТКИ: РАБОТАЕМ АВТОНОМНО

ПРИМЕЧАНИЕ

В контексте спецификации автономных приложений манифест — это список файлов, необходимых для вашего автономного приложения.

Чтобы работать автономно, приложению требуется только *манифест*, сообщающий браузеру, какую информацию требуется сохранить в локальном кэше. Манифест — это всего лишь список файлов, в соответствии с которым браузер сохраняет в кэше ресурсы, код CSS и JavaScript, изображения и прочие данные. Когда пользователь обновляет страницу, браузер восстанавливает эти данные из локального хранилища.

Помимо информации о том, какие данные нужно кэшировать, браузеру можно также сообщить, что он сохранять *не должен*, чтобы запросы тех или иных URL-адресов всегда направлялись в сеть, а не обрабатывались локально. Кроме того, HTML5 позволяет обрабатывать исключения. Например, ваш компьютер не подключен к Интернету, а вы пытаетесь обратиться к объекту, отсутствующему в локальном (автономном) кэше. В таком случае, согласно резервному плану, вам предоставляется *другой* ресурс. Скажем, вы открываете в своем приложении чат, но поскольку он недоступен, ваш запрос перенаправляется на страницу, сообщающую, что с данной функцией можно работать только при наличии подключения.

Первая часть автономного приложения — это манифест, сообщающий браузеру, что сохранять в кэше (и, в некоторых случаях — что сохранять не нужно). Вторая часть — это *applicationCache*. Данный объект содержит методы, позволяющие запускать обновления и загружать новейшую версию данных кэша. Также у него есть события, с помощью которых автор приложения может уведомлять пользователей о наличии обновлений.

ЧТО ТАМ С СОБЫТИЯМИ OFFLINE И ONLINE?

Действительно, как насчет автономных событий? В спецификации HTML5 автономные события определены. В действительности, в некоторых браузерах они появились еще несколько лет назад. Спецификация определяет два события, которые должны срабатывать для документов: `online` и `offline`. В настоящее время уровень их поддержки в разных браузерах сильно отличается, но проблема не в этом. Пока что эти события работают ожидаемым образом только на мобильных устройствах (и, учитывая огромное разнообразие мобильных устройств, можно с уверенностью заявить, что мне не удалось протестировать все существующие). А на обычных машинах ситуация значительно отличается.

Эти события срабатывают только в том случае, если пользователь — да, вы не ослышались, именно пользователь — явно переключает браузер в режим автономной работы. Они не срабатывают автоматически в ситуациях, когда по какой-то причине подключение к сети пропадает или через некоторое время восстанавливается.

Чтобы еще больше усложнить проблему, в некоторых браузерах (в частности, Chrome и Safari) отсутствует элемент меню «Работать автономно», поэтому события запустить вообще невозможно. Честно говоря, в моей книге это звучит совершенно глупо, поэтому в реальных приложениях я не полагаюсь на такие события. Единственная возможная альтернатива — непрерывно опрашивать службу, проверяя, что она работает, и я покажу, как это сделать с помощью манифеста, в разделе чуть ниже, озаглавленном «Определение наличия подключения с помощью манифеста».

МАНИФЕСТ КЭША

Манифест — это документ, сообщающий браузеру, когда и что извлекать из автономного хранилища, когда пользоваться сетевыми ресурсами, а когда прибегать к резервному плану. Как только манифест загружается или обновляется, он запускает обновление объекта `applicationCache`. Для того чтобы заставить браузер искать манифест, просто добавьте соответствующий атрибут к элементу `<html>` и укажите файл, в котором содержится манифест вашего приложения:

```
<!DOCTYPE html>
<html lang="en" manifest="/time.appcache">
<!-- здесь находится мое шикарное приложение под названием time --&gt;
&lt;/html&gt;</pre>

```

Приложение из моего примера будет показывать пользователю время на его компьютере и на моем сервере. Чуть проще, чем Google Docs, но достаточно, чтобы продемонстрировать, что приложение способно работать, даже когда подключение отсутствует. Просто вместо серверного времени, к которому подобраться невозможно, оно будет сообщать, что данная функция временно недоступна. На рис. 7.1 показано окно приложения при первой загрузке на подключенном к сети компьютере.

Моему полному приложению требуются следующие ресурсы:

- страница приложения — в данном случае, `index.html`;
- `time.js` — код для обработки данных времени;
- `time.css` — простые стили для моего приложения;
- `server-time.js` — в данном примере, предположим, что он генерируется моим сервером каждую минуту.



Рис. 7.1. Посторонитесь, приложения Google! Наше приложение умеет показывать время!

ПРИМЕЧАНИЕ

При использовании специального расширения файла .appcache вам придется применять специальные методы, проверяющие, что манифест правильно загружается в браузер. Подробнее об этом чуть далее в разделе «Как правильно подавать манифест».

Все, за исключением `server-time.js`, будет записано в манифест. Наконец, мне потребуется файл, который будет использоваться вместо `server-time.js`, когда компьютер находится в автономном режиме. Это будет

- `fallback-server-time.js` — файл с сообщением об автономном режиме.

Так выглядит содержимое файла `time.manifest`:

```
CACHE MANIFEST
index.html
time.js
time.css
```

```
FALLBACK:
server-time.js fallback-server-time.js
```

```
NETWORK:
*
```

```
# version 8
```

Очень важно сохранять правильный формат файла. Первая строка содержит два слова: CACHE MANIFEST. Благодаря ей браузер понимает, что далее перечисляются файлы манифеста. Эти файлы разбиты на категории, также известные как пространства имен. Категория по умолчанию — это CACHE, и если она не указана, то все встреченные имена файлов помещаются в эту категорию, пока браузер не находит новую категорию. Таким образом, я мог бы написать свой манифест так — и он бы ничем не отличался от предыдущего:

```
CACHE MANIFEST
```

```
CACHE:
index.html
time.js
time.css
```

```
FALLBACK:
server-time.js fallback-server-time.js
```

NETWORK:

*

version 9

Категории можно использовать несколько раз. Чтобы добавить в кэш новые файлы, допишите их в конце манифеста. Если категории будут следовать в таком порядке: `cache`, `fallback`, `cache`, то это нормально, манифест будет работать без ошибок.

ПРИМЕЧАНИЕ

Веб-страница, на которой в теге `<html>` приводится имя манифеста, также неявно входит в манифест кэша. По этой причине я рекомендую обязательно указывать в манифесте имя этого файла — в моем случае `index.html`. Это нужно просто для того, чтобы вы не запутались.

В категории `FALLBACK` вы говорите браузеру, что если он встречает URL-адрес, указанный слева (в моем случае — `server-time.js`), и если соответствующий ресурс отсутствует в манифесте, а обратиться к нему с текущим подключением невозможно, следует использовать файл, указанный справа (в моем случае — `fallback-server-time.js`). Резервный файл `fallback-server-time.js` находится среди файлов, кэшированных браузером, точно так же, как файлы из категории `CACHE`. Подробнее о категории `FALLBACK` мы поговорим в следующем разделе.

В категории `FALLBACK` можно указывать URL-адреса, например:

FALLBACK:

```
server-time.js fallback-server-time.js
/ offline.html
```

Здесь вы сообщаете браузеру, что? если `server-time.js` недоступен, следует использовать файл `fallback-server-time.js`. Если же пользователь запрашивает любой другой путь, например `/foo.html`, и соответствующий ресурс недоступен (либо потому, что он отсутствует в кэше, либо потому, что компьютер не подключен к сети), следует использовать `offline.html`. Благодаря такой технике приложение может работать полностью автономно, не допуская пользователя в те области, которые требуют подключения к Интернету (и, разумеется, уведомляя его об этом с помощью файла `offline.html`). Обратите внимание, что это правило реагирует на *все* ошибочные запросы, то есть даже если запрос, отправленный на URL-адрес изображения, возвращает ошибку 404, все равно пользователь видит файл `offline.html`. Я рекомендую добавить в категорию `FALLBACK` несколько продуманных правил.

На рис. 7.2 показано окно приложения `time`, когда компьютер не подключен к Интернету. Запрос к файлу `server-time.js` был перенаправлен на файл `fallback-server-time.js` — вот почему вы видите совершенно другое сообщение.

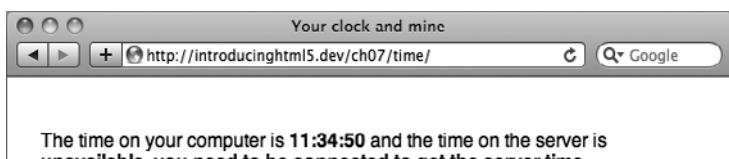


Рис. 7.2. Мое приложение `time` работает и в автономном режиме, но в этом случае для обработки серверного времени обращается к другому ресурсу

Наконец, я включил в файл комментарий, начинающийся с символа # (обратите внимание, что комментарий должен находиться на отдельной строке). В комментарии содержится номер версии. Это важно для *подавления кэша*, соответствующего данному манифесту.

ПРИМЕЧАНИЕ

Подавление кэша (cachebusting) — это техника, позволяющая запретить кэширование ресурса. В случае манифеста, для того чтобы ресурсы загрузились заново, можно поменять его содержимое.

Комментарий добавлять необязательно, но *что-то* в манифесте должно меняться. Лично я обычно использую комментарий или версию ревизии MD5-кэша всех файлов приложения. Такое изменение в файле манифеста приказывает браузеру заново загрузить содержимое, описанное в манифесте автономных файлов. Обратите внимание, что для запуска принудительной загрузки недостаточно поменять время последней модификации манифеста или один из файлов приложения. В то же время любое изменение в *тексте манифеста* приведет к повторной загрузке ресурсов.

ПРИМЕЧАНИЕ

Браузеры не любят вот так запросто расставаться с содержимым кэша. Обязательно добавьте в манифест комментарий, включающий номер версии, ревизию или отметку времени, которую можно изменить и заставить браузер обновить кэш.

Помимо категорий CACHE и FALLBACK также существует категория NETWORK, и в нашем примере она уже заполнена. Категория NETWORK представляет собой «белый список» и сообщает браузеру, что все запросы к любым ресурсам, отсутствующим в автономном кэше, должны обслуживаться через сеть. Если мы с вами мыслим одинаково, вы наверняка думаете: «Так это же поведение по умолчанию, разве нет?» Боюсь, что нет. Если в браузере отображается страница из кэша приложения (AppCache), то есть страница была обслужена с помощью манифеста, то все ресурсы на этой странице должны отвечать тому или иному правилу в манифесте; в противном случае загрузка страницы завершится неудачей. Это кажется нелепым, но добавленное нами правило гарантирует, что нас это нелепое поведение никак не затронет. Можно было бы добавить больше конкретных URL-адресов, но добавляя правил со звездочкой (*), мы заявляем, что *все* остальное, не перечисленное здесь, обслуживается из сети. Следовательно, на нашей странице никогда не появятся эти уродливые рамки вместо потерявшихся изображений.

ПОДРОБНЕЕ О РАЗДЕЛАХ NETWORK И FALLBACK

ДОБАВЛЕНИЕ СОДЕРЖИМОГО В РАЗДЕЛ FALLBACK

Как в пространстве имен FALLBACK, так и в пространстве имен NETWORK поддержка шаблонов в определениях ссылок на ресурсы отсутствует, хотя при взгляде на пример манифеста со строкой, состоящей из звездочки (*), вы могли подумать иначе. В действительности, это всего лишь специальный символ, о котором мы поговорим в следующем разделе, посвященном «белому списку».

В разделе FALLBACK необходимо указывать местоположение ресурса или префикс ресурса, или то, с чего *начинается* URL-адрес. Обратите внимание, что о регулярных выражениях речь не идет, нам нужно именно то, с чего адрес начинается.

Когда ваш веб-сайт работает в автономном режиме и вы используете кэш приложения, можно сделать так, чтобы все запросы к странице payment.html перенаправлялись на ее автономную версию, задав для этого такое правило:

```
FALLBACK
payment.html payment-offline.html
```

Однако если сайт запросит страницу /money/payment.html, то это правило не сработает, для этого нужно определить следующий URL-адрес:

```
FALLBACK
/money/ payment-offline.html
```

Теперь любой запрос, URL-адрес в котором начинается с /money/, в автономном режиме и в случае, когда данный URI-адрес недоступен в кэше, будет перенаправляться на страницу payment-offline.html.

ИСПОЛЬЗОВАНИЕ «БЕЛОГО СПИСКА»

Пространство имен NETWORK работает приблизительно так же, как FALLBACK, то есть вы можете указать полный URL-адрес ресурса или только префикс.

Также существует особое правило: если в разделе NETWORK содержится только символ *, то все запрошенные URL-адреса, отсутствующие в кэше приложения, обслуживаются через сеть. Создается впечатление, что это должно быть поведением по умолчанию, но без данного флага браузер *не в состоянии* сделать веб-запрос. Причем в обоих случаях: когда подключение к сети есть, и когда его нет. Бред, правда?

Получается, что как только вы добавляете файл манифеста, браузер начинает выполнять все запросы через этот файл, и если какому-то запросу никакое правило не соответствует, возвращается ошибка. Это относится как к локальным ресурсам в домене, так и к удаленным ресурсам. Наверное, именно с этим у меня возникало больше всего сложностей в работе, поэтому я настоятельно рекомендую вам запомнить данную тонкость!

Символ * представляет собой *открытый белый список*, и это важно, так как обеспечивает возможность запросить любой ресурс, не учтенный в манифесте. Если данное правило не добавлено, а вы используете что-то вроде Google Analytics, то запросы к серверам Google, отслеживающим использование вашей страницы, завершаются ошибкой. Все это потому, что вы используете манифест, но кэшированной копии запроса нет, и вы не предусмотрели правило, разрешающее такой запрос, ни в разделе NETWORK, ни в разделе FALLBACK.

В целом, я бы порекомендовал использовать следующее итоговое правило задания белого списка. Оно поможет вам во время разработки и избавит от головной боли при отладке приложений:

```
NETWORK:
/remote/
*
```

Если вы также поддерживаете Firefox 3.6, вам нужно добавить следующие две строки, аналогичные правилу * (знаю, они выглядят как регулярные выражения, которых, как я утверждал ранее, здесь нет, но, похоже, создатели Firefox допустили тогда небольшую ошибку; в новых версиях браузера она исправлена, и следующие две строки игнорируются):

```
http://*
https://*
```

Эти правила NETWORK говорят браузеру, что все запросы, начинающиеся с /remote/, будут обрабатываться по сети, а также по сети будут обрабатываться любые другие запросы — так что теперь Google Analytics будет правильно отслеживать статистику для пользователей, подключенных к сети.

КАК ПРАВИЛЬНО ПОДАВАТЬ МАНИФЕСТ

Вам осталось решить единственную проблему, и вы сможете полностью перевести приложение в автономный режим. Итак, файл манифеста нужно правильно подать: у него должно быть расширение .appcache и правильный тип mime.

ПРИМЕЧАНИЕ

Изменение типов mime на веб-сервере не входит в список тем данной книги. Обратитесь к компании, на сервере которой хранится ваш сайт, и она подскажет вам, в каком направлении смотреть.

Если у вас один из распространенных веб-серверов, например Apache, то добавьте в файл mime.types следующую строку:

```
text/cache-manifest appcache
```

Существует несколько способов предоставить содержимое правильного типа; все зависит от выбранного сервера. Просто удостоверьтесь, что веб-сервер отправляет заголовок файла text/cache-manifest при запросе любого файла с расширением .appcache. Для проверки просмотрите заголовок запрошенного файла с помощью инструмента, подобного curl:

```
curl -I http://mysite.com/time.appcache
```

Результат должен быть приблизительно таким:

```
HTTP/1.1 200 OK
Date: Mon, 13 Sep 2010 12:59:30 GMT
Server: Apache/2.2.13 (Unix) mod_ssl/2.2.13 OpenSSL/0.9.81
- DAV/2 PHP/5.3.0
Last-Modified: Tue, 31 Aug 2010 03:11:00 GMT
Accept-Ranges: bytes
Content-Length: 113
Content-Type: text/cache-manifest
```

Теперь ваш сервер отправляет правильные заголовки, а файл манифеста готов к использованию. Поздравляю! Давайте проверим его в действии.

ПРОЦЕССЫ МЕЖДУ БРАУЗЕРОМ И СЕРВЕРОМ

При работе с автономными приложениями полезно знать, что за процессы обмена данными происходят между браузером и сервером. Если возможно, я рекомендую выполнять на серверах следующую команду, чтобы *дописывать* журналы доступа при обновлении страниц, использующих манифест. Так вы будете точно знать, какие ресурсы извлекаются — запрашиваются ли файлы из манифesta и обслуживает ли сервер эти запросы:

```
tail -f logs/access_log
```

При посещении веб-страницы с манифестом кэша, такой как моя программа time, происходит следующее:

- 1) браузер: запрашивает <http://introducinghtml5.com/examples/ch07/time/>;
 - 2) сервер: возвращает index.html;
 - 3) браузер: разбирает index.html и запрашивает все ресурсы, указанные на странице, — изображения, CSS, JS и файл манифesta;
 - 4) сервер: возвращает запрошенные ресурсы;
 - 5) браузер: обрабатывает манифест и запрашивает все перечисленные в нем элементы, независимо от того, получил он их только что или нет. Вполне вероятно, что это будет дублирующий запрос, если ваше приложение кэширует все ресурсы;
 - 6) сервер: возвращает запрошенные ресурсы манифesta;
- браузер: кэш приложения обновлен, можно запустить событие, сообщающее об этом.
- Итак, браузер полностью загрузил кэш файлами, перечисленными в манифесте. Если манифест не менялся, то при перезагрузке страницы происходит следующее:
- 1) браузер: заново запрашивает <http://introducinghtml5.com/examples/ch07/time/>;
 - 2) браузер: определяет, что для данной страницы существует локальный кэш, и обслуживает страницу автономно;
 - 3) браузер: разбирает файл index.html, и все ресурсы из локального кэша обслуживаются автономно;
 - 4) браузер: запрашивает файл манифesta с сервера;
 - 5) сервер: возвращает код 304, сообщая браузеру, что манифест не менялся.

Когда браузер определяет, что в кэше присутствуют все необходимые ресурсы, он обслуживает их локально и только после этого запрашивает манифест. Как показано на рис. 7.3, Safari подгружает все ресурсы для моего приложения time, но в то же время, проверяя журнал доступа этого приложения, я вижу, что по сетевому подключению передаются только time.manifest и server-time.js.

Предположим, что в следующий раз вы обновляете приложение после изменения содержимого манифesta. Если манифест изменился, шаги с 1 по 4 остаются такими же, но затем браузеру приходится перезагружать кэш:

- 1) браузер: заново запрашивает <http://introducinghtml5.com/examples/ch07/time/>;
- 2) браузер: определяет, что для данной страницы существует локальный кэш, и обслуживает страницу автономно;

- 3) браузер: разбирает файл index.html, и все ресурсы из локального кэша обслуживаются автономно;
- 4) браузер: запрашивает файл манифеста с сервера;
- 5) сервер: возвращает обновленный файл манифеста;
- 6) браузер: обрабатывает манифест и запрашивает все перечисленные в нем элементы;
- 7) сервер: возвращает запрошенные ресурсы манифеста;
- 8) браузер: кэш приложения обновлен, можно запустить событие, сообщающее об этом.

БЕРЕГИСЬ ИЗВОРОТЛИВЫХ ЛИС!

Firefox хвалится поддержкой автономных приложений, но в действительности она работает не так хорошо, как в других браузерах, и о паре важных ошибок нельзя забывать. Если вы тестируете свои приложения в Firefox, постоянно проверяйте в журналах, что на самом деле запрашивается у вашего сервера. Браузер должен всегда запрашивать манифест — при каждом визите (и каждом обновлении страницы) в домене. Старые версии Firefox (3.6 и 4) не запрашивают манифест повторно, то есть у вас на руках навсегда остаются манифест и ресурсы, загруженные браузером при первом запросе, независимо от того, были ли изменения! Но надежда есть! Можно приказать браузеру никогда не кэшировать файл манифеста. Для этого добавьте следующий код в конфигурацию сервера или файл .htaccess (или подобный):

```
<filesMatch ".appcache$">
  Header set Cache-Control "max-age=0, private, no-store, no-cache,
  - must-revalidate"
</filesMatch>
```

Если для Apache вы используете mod_expires, то в файл httpd.conf нужно добавить следующее:

```
<IfModule mod_expires.c>
  ExpiresActive on
  ExpiresByType text/cache-manifest "access plus 0 seconds"
</IfModule>
```

Также в ответ на запрос файла .appcache можно отправлять пользовательские заголовки и заголовок no-cache:

```
Header set Pragma "no-cache"
```

Я также хочу опровергнуть все вышесказанное одним простым фактом: разработчики Firefox пекут обновления как горячие пирожки. Возможно, к тому времени, как вы прочитаете это, у нас уже будет Firefox 13, и о такой проблеме никто вспоминать не будет. Тем не менее это правило no-cache не повредит, если вы возьмете за привычку добавлять его в свой код.

Однако важно знать, что даже если ресурсы изменились, ранее загруженные ресурсы не будут просто так удалены (например, изображения не меняются моментально, и старые функции JavaScript не заменяются новыми). В действительности, в данный момент жизни приложения новое содержимое кэша попросту недоступно. Новые кэшированные ресурсы станут доступны только после перезагрузки страницы.

Чтобы понять, как получить в свое распоряжение новые ресурсы, нужно познакомиться с объектом applicationCache.

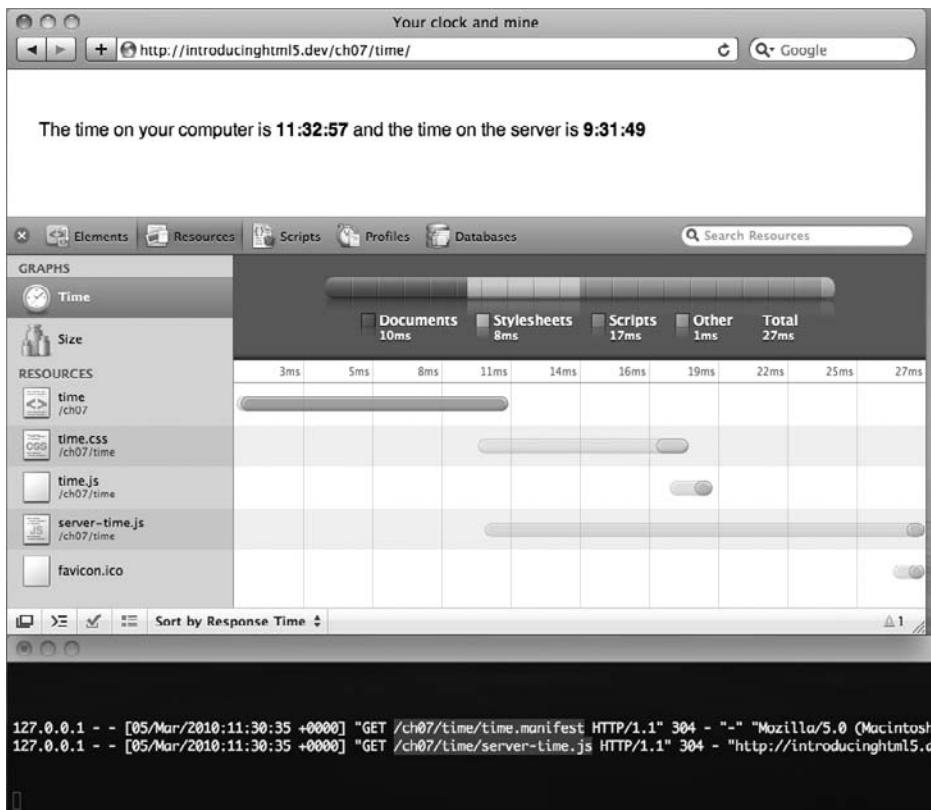


Рис. 7.3. Safari загружает ресурсы приложения из локального кэша, запрашивая у сервера только манифест и server-time

APPLICATIONCACHE

`applicationCache` — это объект, уведомляющий об изменении локального кэша, который также позволяет вручную запустить его обновление. Объект `applicationCache` получает событие, объявляющее об обновлении кэша, только при изменении содержимого манифеста.

В последовательности шагов из предыдущего раздела, после того как браузер загружает в кэш файлы из манифеста, в объекте `applicationCache` срабатывает событие. Благодаря этому событию вы можете сообщить пользователям, что приложение было обновлено на серверной стороне и нужно перезагрузить окно браузера, чтобы насладиться новейшей и наилучшей версией программы. Для этого вам потребуется реализовать простейшее прослушивание событий и уведомление:

```
applicationCache.onUpdateReady = function () {
    // манифест кэша изменился, давайте попросим
    // пользователя
    // перезагрузить приложение, чтобы получить
```

```

→ версию 2.0
if (confirm("Do you want to update to the latest version
→ of this app?")) {
    // если пользователь согласен, принудительно выполнить
    → обновление
    window.location.reload();
}
};

```

Но как сообщить пользователю, что именно в приложении изменилось? Как попробовать динамически перезагрузить только часть функциональности? Строго говоря, это возможно, но, в зависимости от приложения, может быть довольно сложно. В любом случае для загрузки в память новейшей версии кэшированных ресурсов используйте метод `applicationCache.swapCache()`:

```

applicationCache.onUpdateReady = function () {
    applicationCache.swapCache();

    // манифест кэша изменился, давайте попросим
    → пользователя
    // перезагрузить приложение, чтобы получить
    → версию 2.0
    notifyUserOfUpgrade();
};

```

Когда вы подгружаете кэш, теоретически старое содержимое удаляется и заменяется новым, но в действительности изображения не меняются, и код не обновляется. Это происходит только в том случае, когда ресурсы загружаются вручную либо при обновлении всего окна приложения. Тем не менее можно вручную загрузить один из ресурсов, динамически создав новый узел DOM для нужного сценария или изображения.

Предположим, что в вашем манифесте присутствует файл `version.js`, включающий описание новейшей версии приложения. Если у браузера уже готов обновленный кэш (вы определили это благодаря соответствующему событию), загрузите файл `version.js` внутри функции `notifyUserOfUpgrade`. При этом код JavaScript в файле `version.js` будет выполнен заново, и вы сможете рассказать пользователям, какие изменения появятся после обновления приложения.

Мне кажется, что практические возможности применения `swapCache` довольно ограничены. Однако событие готовности обновления весьма полезно. Оно сообщает пользователю, что тот может перезагрузить окно браузера и получить новейший код приложения — так же, как настольное приложение сообщает о готовом к загрузке обновлении. За исключением подобных случаев, обновления загружаются незаметно для пользователя.

СОВЕТЫ ПО ОТЛАДКЕ

Уверен, вы думаете, что все это очень сложно. Это так и не так одновременно. Набив руку на написании автономных приложений, вы будете делать это с легкостью. Но как выполнить отладку, если при первом же запуске выясняется, что приложение не работает?

В браузерах встречаются самые разные инструменты отладки. И как бы нам ни хотелось, чтобы все эти инструменты работали одинаково, иногда все же проще добавлять отладочную информацию в процессе разработки.

На сегодняшний день лучшим инструментом мне кажется веб-инспектор в браузере Google Chrome, в частности, потому что он сообщает о прогрессе кэширования приложения, но также здесь можно сразу понять, используется локальная версия приложения или нет.

На рис. 7.4 представлено приложение jsconsole.com, которое для автономной работы применяет локальный кэш. Вы видите содержимое веб-консоли Chrome, каким оно бывает при первом посещении сайта.

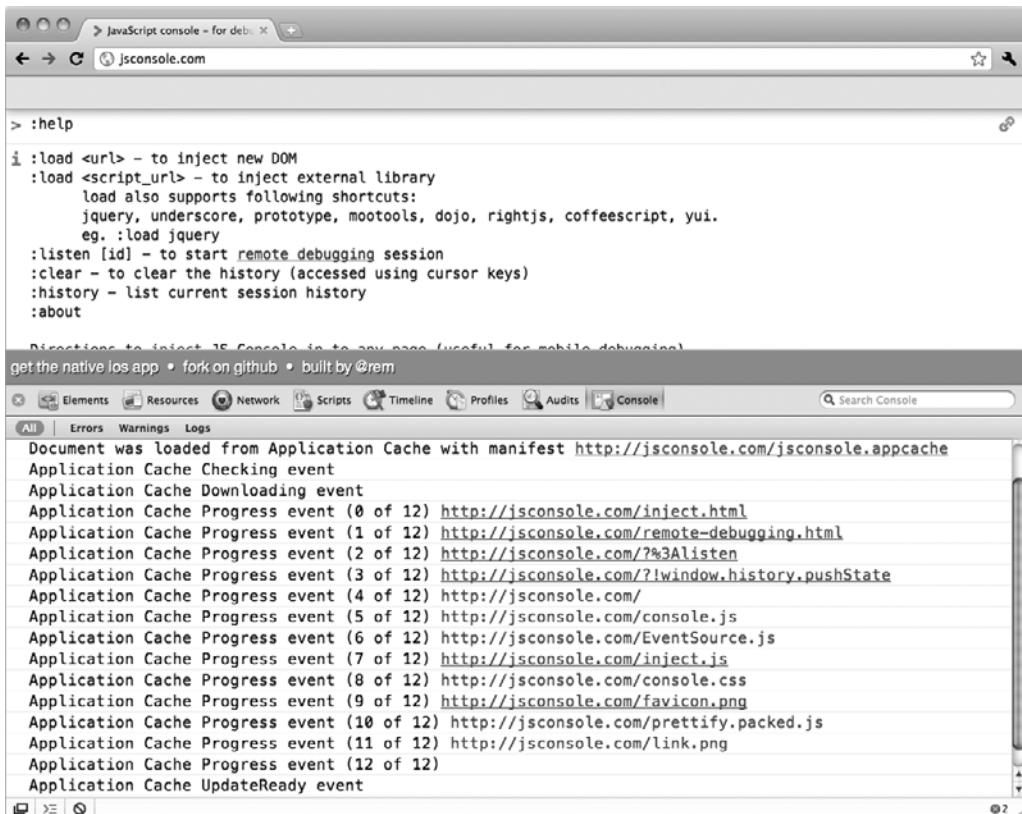


Рис. 7.4. jsconsole впервые загружается в Google Chrome (<http://jsconsole.com>)

Посмотрите на рис. 7.5: все ресурсы зарегистрированы и помещены в кэш. Их можно просмотреть по отдельности на панели ресурсов.

Я обновляю jsconsole.com после завершения загрузки манифеста, и запрашивается только сам файл манифеста — обновление приложения не требуется, что демонстрирует журнал на рис. 7.6.

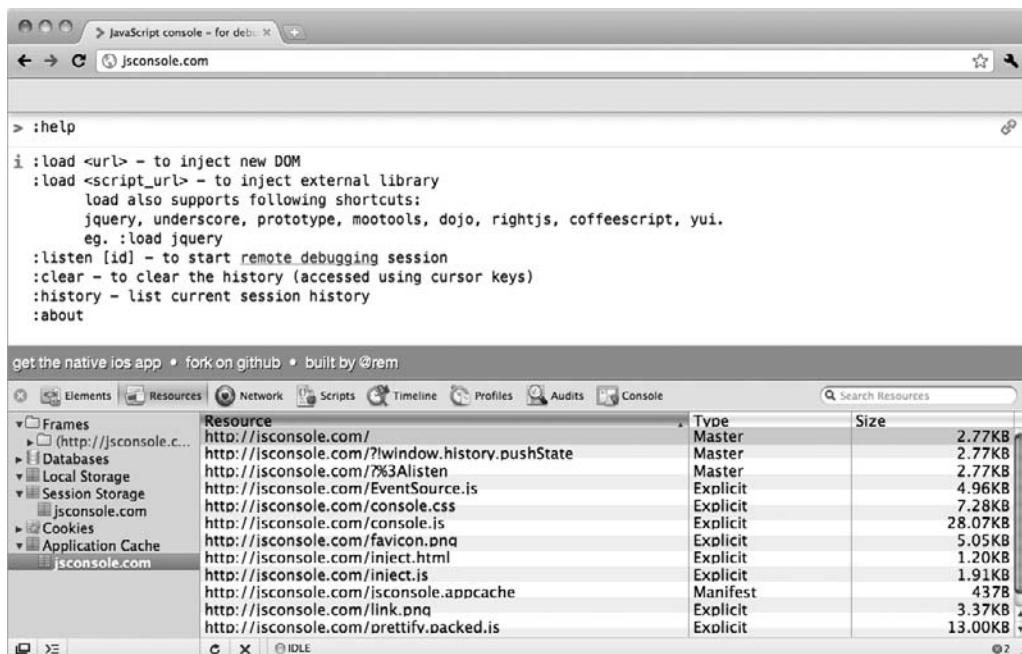


Рис. 7.5. Список ресурсов для текущего манифеста приложения в Google Chrome

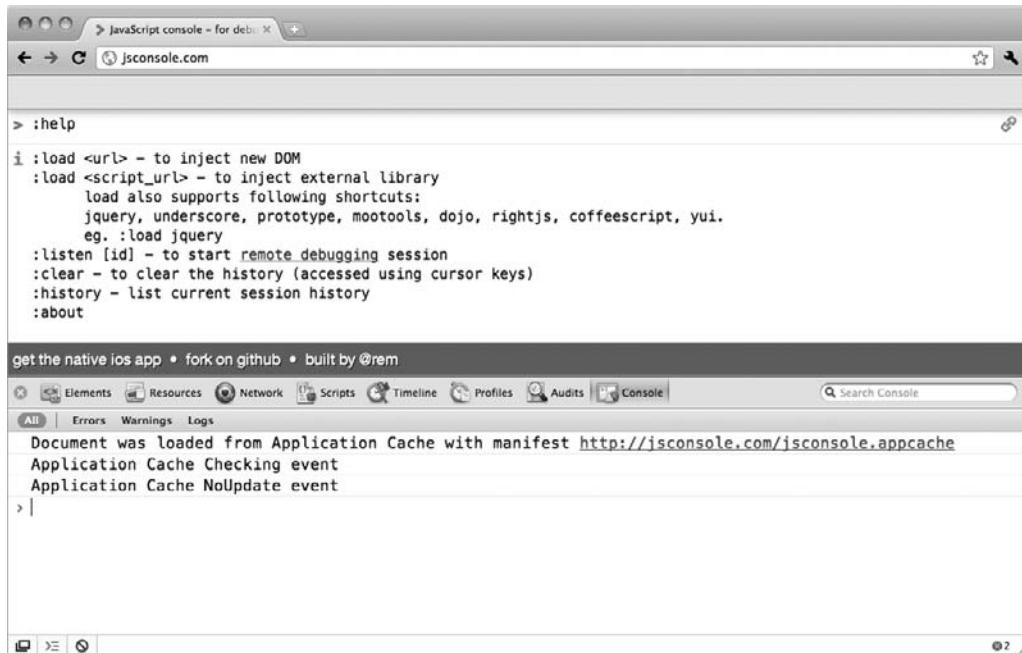


Рис. 7.6. Список ресурсов для текущего манифеста приложения в Google Chrome после обновления страницы

Уверен, что и инструменты в других браузерах научатся предоставлять более полную информацию (если еще не научились к моменту, когда эта книга вышла из печати). Тем не менее, для того чтобы понять, что ваш сайт делает во время загрузки, можно привязываться к отдельным событиям `applicationCache`, таким как `updateReady`.

ОПРЕДЕЛЕНИЕ НАЛИЧИЯ ПОДКЛЮЧЕНИЯ С ПОМОЩЬЮ МАНИФЕСТА

В HTML5 у объекта `navigator` есть свойство, *вроде бы* сообщающее, находится ли браузер в сети. Речь идет о свойстве `navigator.onLine`. Однако, как мы уже видели при обсуждении событий `online` и `offline`, изменять его значение можно только через меню приложения, устанавливая или сбрасывая флагок «Работать автономно» (за исключением некоторых браузеров для мобильных устройств). Как разработчик, вы нуждаетесь в надежном способе определения, может ли браузер соединиться с сервером приложения. Для этого можно использовать категорию `FALLBACK` манифеста кэша. Включив в состав манифеста правило `FALLBACK`, вы можете загружать небольшой код на JavaScript и определять, подключен браузер к сети или нет.

Содержимое манифеста:

```
CACHE MANIFEST
```

`FALLBACK:`

```
online.js offline.js
online.js contains:
setOnline(true);
offline.js contains:
setOnline(false);
```

В приложение мы добавляем функцию под названием `testOnline`. Она динамически создает элемент сценария, который *пытается загрузить* файл JavaScript `online.js`. Если попытка завершается успешно, исполняется код `setOnline(true)`. Если же браузер работает в автономном режиме, незаметно от пользователя вызывается файл JavaScript `offline.js`, который исполняет `setOnline(false)`. Выполнив проверку и убедившись, что все в порядке, можно вызывать `applicationCache.update`:

```
function testOnline(fn) {
    var script = document.createElement('script')
    script.src = 'online.js';

    // назначить функции setOnline новую функцию
    window.setOnline = function (online) {
        document.body.removeChild(script);
        fn(online);
    };

    // добавить в код страницы тег script для
    // запуска программы
    document.body.appendChild(script);
}
```

```

testOnline(function (online) {
  if (online) {
    applicationCache.update();
  } else {
    // демонстрируем пользователям
    // ненавязчивое сообщение об отсутствии
    // сетевого подключения
  }
});
```

ОЧИСТКА КЭША

Как я уже говорил ранее в этой главе, браузеры прилипают к кэшу всем сердцем. Можно запросто попасть в ситуацию, когда очистить локальный кэш, чтобы протестировать внесенные изменения, будет просто невозможно. Пока что в спецификации не предусмотрено методов для программной очистки кэша (например, это нельзя сделать из объекта `applicationCache`).

Помня об этом, я настоятельно рекомендую вам во время разработки избегать манифеста кэша. Удостоверьтесь, что разработка приложения полностью завершена, и только после этого добавляйте атрибут манифеста. И все же, однажды кэш будет заполнен — так как его очистить?

ПРИМЕЧАНИЕ

Очищая кэш, убедитесь, что не осталось открытых окон с вашим приложением, использующим манифест.

Вручную. Для того чтобы обновить кэш, нужно всего лишь изменить содержимое файла манифеста. Как я уже сказал, используйте комментарий с указанием версии или подобной информацией.

Но что делать, если вам нужно начать сначала или если вы попросту хотите удалить атрибут манифеста? Это возможно только во время разработки, так как требует очистки кэша браузера (и, в зависимости от браузера, кэш может храниться в разных местах). Невозможно сделать это программно: только пользователь может вручную запустить очистку автономного кэша.

В Safari кэш нужно очищать полностью. Под словом «кэш» я имею в виду всю информацию, которую браузер сохранил с целью ускорить работу в Интернете.

Safari очищает кэш для любых доменов, за исключением того, с которым связано открытое окно, поэтому даже если вы обновите содержимое окна, манифест кэша никуда не денется. То же самое относится к iPhone.

В Chrome можно зайти на служебную страницу `chrome://appcache-internals/`, где предусмотрен способ удаления кэша для определенного URL-адреса.

В Firefox нужно открыть меню настроек, перейти на вкладку Advanced (Дополнительно), а затем выбрать вкладку Сеть (Network). Здесь можно очистить кэш для отдельных доменов.

В Opera откройте меню Preferences (Настройки), перейдите к разделу Advanced (Дополнительно) и выберите Storage (Локальное хранилище). Здесь можно удалить хранилище для отдельного URL-адреса. Несмотря на то что данный раздел называется *долгосрочным хранилищем* (в том смысле, что он связан с объектами `localStorage` и `sessionStorage`), вы встретите дополнительные URL-записи с данными, зачастую имеющими формат домена с хеш-кодом в конце, например `html5demos.com/$7b8e3c7f`. Неясно, для чего они нужны (при удалении такой записи не очищается ни кэш приложения, ни `localStorage`), но для надежности я рекомендую вручную удалять все.

ЗАКЛЮЧЕНИЕ

В прошлом веб-сайты полностью зависели от подключения к Интернету. Сегодня браузеры начинают постепенно поддерживать возможность автономной работы и даже позволяют распознавать наличие подключения (с некоторыми хитростями). В этой главе вы узнали, как заставить веб-приложения работать без подключения к сети.

Глава 8

ПЕРЕТАСКИВАНИЕ

Реми Шарп



Итак, мы подобрались к черной овце нашей книги — перетаскиванию по технологии drag and drop. Это не крутая черная овца типа навороченного имперского истребителя Дарта Вейдера — нет, к сожалению, это та черная овца, которую хочется забыть в поле в полном одиночестве, и пусть она делает все, что ей заблагорассудится. У многих разработчиков нашлись бы куда более цветистые выражения для описания этой спецификации.

Так почему же она здесь? Почему она входит в спецификацию HTML5 — и да, не сомневайтесь, drag and drop действительно представляет собой полноценную часть спецификации HTML5. Что ж, все дело в том, что Microsoft добавила перетаскивание в Internet Explorer версии 5 еще в 1999 году — очень, очень давно. С тех пор API из Internet Explorer был реализован в Safari, поэтому Ян Хиксон, разработчик спецификации HTML5, всерьез решил разобраться, что там происходит (к слову, документацию MSDN он назвал чем-то вроде «туманных шаманских напевов»). В общем, он выполнил обратное проектирование и за-документировал API.

ПРИМЕЧАНИЕ

С исследованием Яна можно ознакомиться на веб-сайте <http://ln.hixie.ch/?start=1115899732&count=1>.

Таким образом, сейчас мы находимся в положении, когда этот API поддерживается в Firefox, Safari, Chrome и IE. Это плохой API — наверное, худший среди существующих, — но его все же реализовали, поэтому стоит взглянуть, на что он способен.

В этой главе вы имеете полное право воскликать «Что за чертовщина?!», пробираясь сквозь лабиринты кроличьей норы, которую по ошибке назвали API drag and drop. И все же, у данного API есть несколько интересных возможностей. Как подразумевает его название, API drag and drop позволяет перетаскивать элементы внутри окна браузера. И даже за его пределами. Элементы из браузера можно перетаскивать во внешние приложения — окна других браузеров, Photoshop, текстовые редакторы, — а ваше приложение должно подготовливать эти данные, чтобы они были совместимы с той программой, в которую в итоге попадают. Таким образом, API перетаскивания подтверждает, что HTML5 — действительно спецификация веб-приложений, дающая разработчикам намного больше возможностей, чем обычные настольные приложения.

Начинаем перетаскивать

Мы начнем с абсолютного минимума, необходимого для погружения в волшебный мир перетаскивания. По умолчанию перетаскивать можно все ссылки, текстовые узлы (выделенный текст) и элементы изображения. Это означает, что для того, чтобы пользователь мог перемещать такое содержимое по странице, вам не нужно сообщать браузеру никакие дополнительные сведения.

В нашей простой демонстрационной программе будет зона приема и пара изображений, которые вы сможете перетаскивать в эту зону. Поместив изображение в зону приема и отпустив клавишу мыши, вы увидите источник (адрес) изображения (рис. 8.1).



Рис. 8.1. По умолчанию перетаскивать можно все ссылки и изображения. Но добавив еще немного кода, вы позволите пользователям не только перетаскивать элементы по экрану, но и помещать их в специально предназначенную для этого область

С самими изображениями, которые мы будем перетаскивать, ничего делать не нужно, поэтому займемся зоной приема. Для нее нам потребуются следующие обработчики событий:

- 1) `drag over`: сообщает браузеру, что именно этот элемент принимает перетасченные данные;
- 2) `on drop`: когда на этот элемент попадают какие-то данные, с ними нужно что-то делать.

ПРИМЕЧАНИЕ

В этом разделе под «зоной приема» я подразумеваю всего лишь место, на которое пользователь перетаскивает некий элемент. Я не ссылаюсь на недавно добавленный атрибут W3C `dropzone`, о котором подробнее рассказывается во врезке «Встроенные зоны приема» далее в этой главе.

Речь сейчас пойдет об абсолютном минимуме, который требуется для реализации перетаскивания, причем этот метод работает только в Safari. Далее я расскажу вам о нескольких трюках, которые позволят реализовать его в Firefox, Chrome и IE.

Также стоит упомянуть, что спецификация на веб-сайте <http://dev.w3.org/html5/spec/editing.html#dnd> утверждает, что для обработки перетаскивания вам потребуются *три* события. Это не так, по крайней мере, если дело касается практического программирования. Три события нужны, чтобы заставить приложение работать во всех браузерах, но в Firefox и Safari можно обойтись меньшим.

Давайте на минуту отложим в сторонку все эти трудности и перейдем к демонстрационной программе. Следующий фрагмент кода — это суперминималистичное приложение, позволяющее взглянуть на API перетаскивания в действии:

```
<!DOCTYPE html>
<title>Simple drag demo</title>
<style>#drop { height: 100px; border: 5px solid #ccc; }</style>
```

```

~ </style>


<div id="drop"></div>
<script>
    var drop = document.getElementById('drop');
    drop.ondrop = function (event) {
        this.innerHTML += '<p>' + event.dataTransfer.
            ~ getData('Text') + '</p>';
    };
    drop.ondragover = function () { return false; };
</script>

```

Я использую минимальный объем кода HTML — просто чтобы пример не получился слишком громоздким. Как вы видите, я захватываю ссылку на элемент `div#drop`, а затем устанавливаю два встраиваемых обработчика событий: `ondrop` и `ondragover`.

Когда *что-то* попадает на элемент зоны приема, срабатывает событие `drop`, и вы можете считать объект `event.dataTransfer`. Тип данных по умолчанию — `Text`, поэтому в методе `getData` запрашивается текстовый тип. В случае изображения текст содержит ссылку на источник этого изображения (обычно при запросе текстового типа данных IE возвращает `null`, но чуть позже мы это исправим). Для ссылок текст — это данные из тега `href`, а для простого текста, очевидно, это выделенные и переташенные символы.

Именно здесь и начинают твориться чудеса. Для того чтобы сказать браузеру, что зона приема может принимать элементы, которые на нее перетаскивают, событие `dragover` необходимо *отменить*. Так как я использую встраиваемый обработчик событий (а именно, `ondragover`), то я возвращаю `false`, запрещая, таким образом, действие браузера по умолчанию. А что, собственно, браузер делает по умолчанию? Из спецификации это неясно, но будет справедливо сказать, что по умолчанию объект отдается под контроль браузера. Если бы я использовал `addEventListener`, то добавил бы `event.preventDefault()`.

Чтобы окончательно развеять ваши сомнения, потому что, честно говоря, все это далеко не очевидно, — вот цитата из спецификации:

If the drop is to be accepted, then this event (dragover) has to be canceled.

(Чтобы принять переташенный элемент, это событие (dragover) необходимо отменить.)

Что ж, ваше первое приложение с перетаскиванием теперь работает, поэтому можно перейти к трюкам, о которых я упомянул выше. Давайте сначала исправим ошибки в Firefox — это просто. Когда вы отпускаете изображение на зоне приема в Firefox, то если значение `getData('Text')` выглядит как ссылка, зона приема передает его браузеру, то есть вы получаете источник изображения или значение `href` для ссылки. Таким образом, нужно всего лишь запретить действие браузера по умолчанию в событии `drop`. При использовании встраиваемых обработчиков для этого необходимо вернуть `false` либо применить `event.preventDefault()`. В результате получаем такой обработчик события `drop`:

```

drop.ondrop = function (event) {
    this.innerHTML += '<p>' + event.dataTransfer.getData
        ~ ('Text') + '</p>';
    return false;
};

```

Теперь Internet Explorer. Это будет не так сложно, как может показаться с первого взгляда; возможно, потому, что именно Microsoft придумала API перетаскивания. IE не прослушивает событие `dropover`, этот браузер слушает событие `dropenter` — и именно его необходимо отменить, чтобы наше приложение заработало в IE. Итак, давайте добавим еще один обработчик событий, возвращающий `false`. Так как мы везде делаем одно и то же, я создал функцию, предназначенную исключительно для того, чтобы возвращать `false`:

```
function cancelEvent() { return false; }
drop.ondragenter = cancelEvent;
drop.ondragover = cancelEvent;
```

И снова, поскольку мы пытаемся заставить приложение работать в IE, IE не передает объект `event` обработчику событий. Следовательно, если вы не получили этот объект, нужно изменить обработчик события `drop`, чтобы захватить глобальный объект `event`:

```
drop.ondrop = function (event) {
    event = event || window.event;
    this.innerHTML += '<p>' + event.dataTransfer.getData
        - ('Text') + '</p>';
    event.cancelBubble = true;
    return false;
};
```

Помимо этого, необходимо запретить событию вызывать DOM, чтобы не вспыпало новое окно. Обычно это обрабатывается в `return false` (во всяком случае, в других браузерах), но Internet Explorer необходима небольшая помощь (см. строку `event.cancelBubble=true` в коде выше).

И еще одно небольшое исправление: когда вы отпускаете изображение над зоной приема в IE или Chrome, то вместо текста получаете `null`. Для того чтобы исправить это, нужно определять какие-то данные типа `Text` в момент, когда начинается перетаскивание элемента. Для этого используйте событие `dragstart`:

```
var imgs = document.getElementsByTagName('img'),
    i = imgs.length;
while (i--) {
    imgs[i].ondragstart = function (event) {
        event = event || window.event;
        event.dataTransfer.setData('Text', this.getAttribute
            - ('alt'));
    };
}
```

Вы видите, что я устанавливаю данные типа `Text`, используя атрибут `alt` изображения. Теперь, когда вы отпускаете изображение над зоной приема, эти текстовые данные считаются, и вместо источника изображения вы видите имя пользователя из Twitter. Это демонстрационное приложение работает в IE5 и последующих версиях, в Firefox, Chrome и Safari. Что еще важнее, именно метод `setData`, демонстрирующий возможности модели перетаскивания, обнажает некоторые потенциальные проблемы в спецификации.

СОВМЕСТИМОСТЬ ПЕРЕТАСКИВАЕМЫХ ДАННЫХ

Используя методы `setData` и `getData` с объектом `dataTransfer`, вы можете передавать данные от внутренних элементов приложения другим страницам этого приложения или даже в другие окна браузера. Например, приложение 280 Slides поддерживает перетаскивание слайдов из одного окна в другое — в совершенно отдельный документ (рис. 8.2). Кроме того, данные можно принимать и отправлять в обычные настольные приложения.



Рис. 8.2. Один из первых примеров возможностей перетаскивания реализован в приложении 280 Slides

ПЕРЕТАСКИВАНИЕ ДАННЫХ В ДРУГИЕ ПРИЛОЖЕНИЯ

Если известно, что какое-то приложение способно принимать данные определенного типа, можно устанавливать соответствующий тип содержимого в своем приложении и перетаскивать данные из своего приложения в другое. Например, на Macintosh на рабочий стол можно перетаскивать текстовые фрагменты. Я создам некое содержимое, установлю для него тип `text/plain` и буду перетаскивать его на рабочий стол, создавая текстовую заметку (рис. 8.3):

```
img.ondragstart = function (event) {
    event = event || window.event;
    // здесь будет одна длинная строка
    event.dataTransfer.setData('text/plain',
        'This is the screen name for ' + this.getAttribute
        ('data-screen_name') +
        ', whose image can be found here: ' + this.src);
};
```

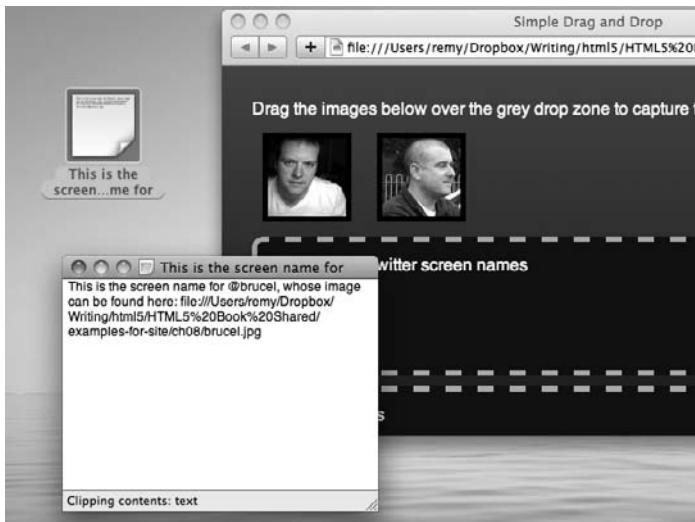


Рис. 8.3. При перетаскивании содержимого с веб-страницы на рабочем столе создается текстовый фрагмент

Еще одно замечание относительно `setData`: этот метод принимает только строки. Это означает, что невозможно выбрать в качестве типа содержимого «сложный» объект JavaScript. Однако решить проблему относительно просто: используйте `JSON.stringify`.

Все новейшие браузеры поддерживают кодирование (`stringify`) и декодирование (`parse`) JSON, поэтому вы с легкостью можете превращать сложные объекты в строки и передавать данные соответствующего типа. Если вы работаете со старыми браузерами, то для поддержки `stringify` и `parse` добавьте библиотеку JSON, которую можно загрузить на веб-сайте <https://github.com/douglascrockford/JSON-js>.

Проблемы при установке перетаскиваемых данных

В настольных приложениях функция перетаскивания используется уже довольно давно, но для того чтобы API начал правильно работать, потребовались годы отладки. Большое преимущество родных приложений заключается в том, что установка данных происходит не тогда, когда пользователь начинает перетаскивать объект. Это производится, когда пользователь отпускает его.

За этим кроется одна важная причина: именно когда пользователь отпускает элемент, становится понятно, какой единственный тип содержимого вам требуется.

ПРИМЕЧАНИЕ

Обратите внимание, что всего существует семь событий перетаскивания. Вы уже познакомились с `dragenter`, `dragover`, `drop` и `dragstart`. Помимо этого, есть `dragend` (дополнение к `dragstart`), `dropenter` и `dragleave`. События `enter` и `leave` срабатывают на зоне приема, когда перетаскиваемый объект оказывается на соответствующем элементе.

Необходимость конструировать различные типы содержимого при срабатывании события `dragstart` зачастую приводит к исполнению ненужного кода. Например, если я разрешаю пользователю перетаскивать элемент холста в Photoshop, мне приходится кодировать данные элемента в виде файла, совместимого с Photoshop, и сохранять их, используя правильный тип содержимого. Но что, если моя программа должна также поддерживать и *другие* форматы, не только формат файлов Photoshop? Мне придется выполнять все это разнообразное кодирование в момент, когда срабатывает событие `dragstart`, а пользователь в итоге перетащит данные максимум в одно приложение. К тому же, он может просто перетаскивать элементы по экрану, чтобы позабавиться, не передавая их другим приложениям. И все равно мне придется исполнять ненужный код, зазря расходуя вычислительные ресурсы. Если речь идет о простом приложении, вряд ли это как-то повлияет на производительность, однако разработчикам комплексных приложений стоит трезво оценивать нагрузку на системные ресурсы и тщательно выбирать варианты. Возможно, ваше приложение не поддерживает все эти форматы. Возможно, оно поддерживает только один совместимый формат. Возможно, оно вовсе не поддерживает перетаскивание.

Уже поступали предложения исправить это (не говоря уж о предложениях полностью переписать всю модель перетаскивания с нуля), но пока что необходимо учитывать существующую проблему и искать обходные пути.

КАК ПЕРЕТАСКИВАТЬ ЛЮБОЙ ЭЛЕМЕНТ

Итак, мы дошли до места, где спецификация HTML5 добавляет нечто новое в API перетаскивания. Разрешить перетаскивать любой элемент невероятно просто. Возьмите свой `div` и добавьте к нему новый атрибут: `draggable`. Например:

```
<div draggable="true">Этот элемент можно перетаскивать</div>
```

Да, я сказал «невероятно». Что ж, это работает в Firefox; любой элемент с атрибутом `draggable` теперь можно перетаскивать в окне браузера. И разумеется, поскольку это новинка в HTML5, в стандарт IE она не входит, так что сразу забудьте про Internet Explorer. Возможно, это будет работать в IE9 и более поздних версиях. Даже с Safari 4 все намного проще.

Хотя даже школьник может разрешить перетаскивание любого элемента с помощью атрибута `draggable`, по причинам, которые ни автор, ни многие другие блоггеры до сих пор уразуметь не в состоянии, для того чтобы включить перетаскивание элемента в Safari 4, вам понадобится предоставить ему отдельный стиль CSS. Все верно, вы включаете определенное *поведение*, определяя атрибут *представления*. В Safari 5 недочет исправили, и CSS больше не требуется, но для того чтобы приложение работало в старых версиях Safari, добавить код CSS все же необходимо. Используйте следующий код CSS для поиска элементов с атрибутом `draggable`:

```
[draggable] { -webkit-user-drag: element; }
```

Здесь селектор атрибутов CSS (в квадратных скобках) ищет все элементы, для которых данное свойство включено. Потом к ним применяется нужное поведение, чтобы пользователь мог перетаскивать элементы в любой версии браузера.

ВСТРОЕННЫЕ ЗОНЫ ПРИЕМА

Нет, мы говорим не о ящике для пожертвований, встроенным в каждый браузер. Теперь в спецификации техники перетаскивания появился новый атрибут под названием `dropzone`. Он устанавливается для областей, на которые вы хотите перетаскивать данные. Атрибут принимает следующие значения, определяющие реакцию зоны приема на перетаскиваемый элемент: `copy` (по умолчанию), `move` и `link`. Кроме того, можно указывать допустимые типы данных, которые данная зона способна принимать. Например:

```
<div dropzone="copy s:text/plain f:image/png"> Drop 'em on my head </div>
```

Этот код разрешает моему сценарию копировать на зону приема любое перетаскиваемое содержимое типа «простой текст» или «изображение PNG». В настоящее время спецификация ничего другого не поддерживает.

Подозреваю, что идея при создании всего этого заключалась в том, чтобы избавиться от шаманства с `dragover` и `dragenter`. Однако поскольку никакие современные браузеры данную возможность не поддерживают, проверить это невозможно. Определенно, это шаг в нужном направлении — к улучшению и упрощению API drag and drop. Тем не менее, пока мы не увидим реализацию в браузере, боюсь, это останется всего лишь мечтой о будущем техники перетаскивания, а не полезной функциональностью для реальной жизни.

Помимо трюка с CSS, необходимого для вразумления Safari 4, включить перетаскивание любого элемента не так сложно. Это означает, что теперь вы можете создавать в DOM сложные объекты, которые пользователь будет перетаскивать по экрану и переносить в другие окна и приложения.

ДОБАВЛЕНИЕ ПОЛЬЗОВАТЕЛЬСКОГО ЗНАЧКА ПЕРЕТАСКИВАНИЯ

Во время перетаскивания элемента отображается специальный значок — и вы можете добавить вместо него собственное уникальное изображение. Когда сработает событие `dragstart`, используйте метод `setDragImage` и свяжите выбранное изображение с указателем мыши на определенном удалении от него.

Разумеется, без недоразумений не обойдется и здесь — этот трюк не работает в Internet Explorer, и в Safari вам не удастся изменить вид указателя при перетаскивании текста, изображений и ссылок. Но не будем терять оптимизма и давайте создадим собственный значок перетаскивания:

```
var dragIcon = document.createElement('img');
// устанавливаем в качестве значка перетаскивания
- мини-логотип twitter
dragIcon.src = 'http://img.twimg.es/i/twitter_m';
// позднее в коде...
element.ondragstart = function (event) {
    event.dataTransfer.setDragImage(dragIcon,
        -10, -10);
    // и всякие другие операции с dataTransfer
};
```

В результате мы видим уникальный маленький значок, гораздо лучше представляющий данные, которые мы перетаскиваем по экрану (рис. 8.4). Видимо, для того чтобы не снижать градус общего безумства спецификации drag and drop, при перетаскивании изображения используется система координат, противоположная той, к которой мы все привыкли. Так как во всех (по крайней мере, я не могу вспомнить ни одного исключения) сетевых технологиях применяется система координат, в которой центральная точка находится в левом верхнем углу, вы, вероятно, ожидаете, что отрицательное смещение переносит нашу иконку на 10 пикселов влево и вверх. Но нет, очевидно, вы ошибаетесь... в действительности изображение сдвигается на 10 пикселов вниз и вправо. В общем, запомните золотое правило при работе с перетаскиванием: все ваши предположения неверны.

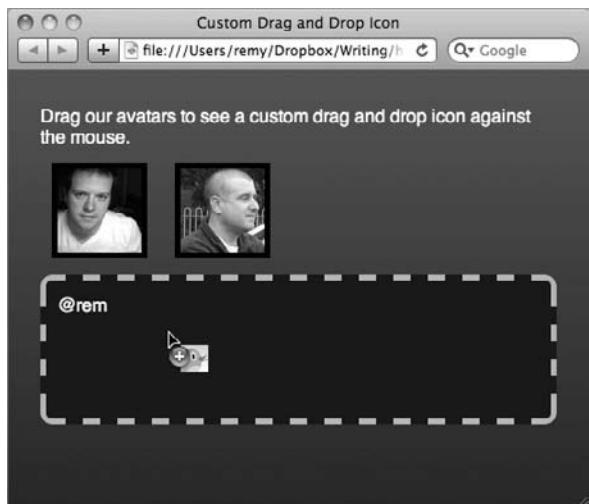


Рис. 8.4. Мы создали собственный указатель с логотипом Twitter для перетаскивания данных, связанных с Twitter

Доступность

Если вы успешно пробрались сквозь все сложности, разветвления и тупики этой спецификации, то, возможно, перед вашим приложением с возможностью перетаскивания встала проблема доступности для пользователей с нарушениями зрения. Поддерживает ли API drag and drop соответствующие возможности? Можно ли сделать этот API доступным? Что ж, как и ожидалось, спецификация создавалась с наилучшими намерениями. Так что да, при проектировке API доступность учитывалась. Нельзя сказать, что реализация соответствующих возможностей понятна с первого взгляда, но, вроде бы, пользователи смогут управлять перетаскиванием по принципу копирования с клавиатуры.

Процесс должен быть примерно таким: перейдите к элементу, который нужно перетащить, скопируйте его в буфер обмена, нажав соответствующее клавишное сочетание, затем перейдите к зоне приема и вставьте элемент другим клавишным сочетанием.

Как вы уже наверняка догадались, ни в одном из браузеров это (пока что) не реализовано.

Однако уже сейчас можно включить поддержку ARIA в программы, демонстрирующие перетаскивание. Мы должны будем устанавливать атрибуты ARIA в событии `dragstart`, сообщая, что началось перетаскивание элемента. Чтобы удалить атрибут ARIA, понадобится событие `dragend`. Помимо этого, нужно добавить визуальные индикаторы, подсказывающие, какие элементы можно перетаскивать и куда. Я не буду расписывать все в деталях, но вы можете прочитать подробную статью Геза Лемона, посвященную ARIA и доступности возможностей перетаскивания пользователям с нарушениями зрения. Его советы по реализации перетаскивания в собственных приложениях применимы и к родным приложениям, поддерживающим технологию drag and drop: <http://dev.opera.com/articles/view/accessible-drag-and-drop/>.

```

var drop = document.getElementById('drop'),
    boxes = document.getElementsByTagName('div'),
    i = boxes.length;

while (i--) {
    if (boxes[i].getAttribute('draggable') != undefined) {
        boxes[i].ondragstart = function (event) {
            event = event || window.event;
            this.setAttribute('aria-grabbed', 'true');
            // поддержка ARIA в зонах приема
            drop.tabIndex = 0; // для поддержки
            // клавиатуры
            drop.setAttribute('aria-dropeffect',
                'copy');

            // делаем что-нибудь
            // с dataTransfer.setData
        };

        boxes[i].ondragend = function () {
            this.setAttribute('aria-grabbed',
                'false');

            // сброс зон приема
            drop.tabIndex = -1; // для поддержки
            // клавиатуры
            drop.removeAttribute('aria-dropeffect');
        };

        boxes[i].tabIndex = 0; // для поддержки
        // клавиатуры
        boxes[i].setAttribute('aria-grabbed',
            'false');
    }
}

```

В этом коде производится поиск элементов `div`, для которых установлен атрибут `draggable`. Затем, начиная с события `dragstart`, добавляется поддержка ARIA. Как только пользователь начинает перетаскивать элемент, атрибуту `aria-grabbed` присваивается значение `true`,

чтобы вспомогательное устройство могло сообщать приложению свои сведения. Помимо этого, зона приема превращается в элемент, на который может переводиться фокус клавиатуры. Для этого используется `tabIndex = 0`. Наконец, мы говорим, что когда пользователь «отпускает» элемент, должно производиться копирование. Воспроизвести `allowedEffect` и `dropEffect` можно и в родных приложениях с поддержкой drag and drop, но пока что мы сосредоточимся на поддержке ARIA.

Затем мы добавляем новый обработчик события `dragend`, а когда перетаскивание элемента прекращается, удаляем атрибут `aria-grabbed` и сбрасываем атрибуты зоны приема, то есть `tabIndex` и `dropEffect`. В конце мы инициализируем перетаскиваемый элемент, устанавливая атрибут `tabIndex` и флаг `aria-grabbed`.

Благодаря этому коду пользователи могут перемещаться по приложению и его компонентам, поддерживающим перетаскивание. Устройства считывания экрана (если они поддерживают ARIA) будут сообщать приложению текущее состояние операции.

Однако — и это очень значимое *однако* — так как ни в одном браузере клавиатурная поддержка перетаскивания пока не реализована, вероятно, вам придется создавать с помощью JavaScript собственную технологию drag and drop, которая позволит перетаскивать любые элементы. Довольно печально, учитывая, что мы говорим о весьма распространенной в сети операции.

ЗАКЛЮЧЕНИЕ

API drag and drop находится не в лучшем состоянии и его довольно сложно реализовать во всех браузерах, которые может поддерживать ваше приложение. Вполне вероятно, что вам придется искать помощь в решениях на основе JavaScript, чтобы реализовать перетаскивание там, где поддержка drag and drop отсутствует.

Тем не менее родные возможности drag and drop в сочетании с новейшими API, такими как API File (не рассматривается в этой книге, но это API, позволяющий браузеру считывать файлы внутри JavaScript-кода, без участия серверной стороны), дают пользователям возможность перетаскивать файлы прямо в браузер. Подобные функции можно встретить в различных новых приложениях, например Gmail. Пользователи, браузеры которых поддерживают самые свежие возможности API drag and drop, уже сегодня могут наслаждаться современными технологиями. Помимо поддержки в браузерах, вторая большая проблема на сегодняшний день — это доступность с клавиатуры.

Вам же, как разработчику, необходимо тщательно взвесить все за и против и решить, подходит ли для вашего приложения родная технология drag and drop.

Глава 9

ГЕОЛОКАЦИЯ

Реми Шарп



Когда красные телефонные будки еще встречались в Великобритании на каждом углу, внутри них обязательно было наклеено объявление с адресом будки. Таким образом, если вам нужно было вызвать полицию или скорую помощь, вы могли сразу сказать, где находитесь. Разумеется, это было на руку и любителям погулять в пятницу вечером, ведь для того чтобы позвать друга на помощь, надо точно знать, куда его зовешь. Суть геолокации в том же самом — только пива не наливают.

API Geolocation — это один из тех API, которые никоим образом не связаны со спецификацией HTML5. Он создан Консорциумом Всемирной паутины (W3C), а не рабочей группой WHATWG и никогда даже не был частью исходной спецификации веб-приложений (хотя и ссылается теперь на спецификацию HTML5). И все же очень здорово, что мы можем включить его в нашу книгу. В действительности, что касается создания приложений с опасными — да, опасными! — возможностями социального взаимодействия, это ключевой API.

С этим API чрезвычайно просто работать, и вы с легкостью усовершенствуете свои веб-приложения, использующие географические данные. Подключив API геолокации, вы избавите посетителей от необходимости пролистывать длинные списки или просматривать карту, чтобы найти свое местоположение.

На данный момент Internet Explorer 9 и новейшие версии всех распространенных браузеров (а также их предыдущее поколение) поддерживают API геолокации — неплохо для технологии, которая только-только появилась на свет! Кроме того, геолокацию поддерживают и многие смартфоны и мобильные браузеры — в частности, iOS, Android WebKit, Firefox Mobile и Opera Mobile. Если вы используете для построения приложений технологии Open Web, то сможете воспользоваться API геолокации благодаря PhoneGap — инфраструктуре для развертывания мобильных приложений Open Web.

ПРИШПИЛЬТЕ ПОСЕТИТЕЛЯ К КАРТЕ

API геолокации позволяет распознать точное местоположение посетителя вашего сайта. Эта возможность реализована во множестве современных приложений, начиная от картографических, таких как Google Maps (рис. 9.1), и заканчивая социальными сетями с привязкой к местоположению, например Gowalla и Google Buzz.

API геолокации предлагает два метода получения географической информации от пользователя:

- 1) `getCurrentPosition` — это одноразовый метод захвата текущего местоположения пользователя;
- 2) `watchPosition` — следит за изменением местоположения, периодически отправляя новые запросы. Функциональность `watchPosition` совпадает с возможностями `getCurrentPosition`, за одним исключением — если местоположение пользователя меняется, данный метод сообщает об этом вашему коду. Обратите внимание, что, несмотря на постоянный опрос позиции, `watchPosition` отправляет приложению информацию только при изменении местоположения. Для того чтобы прекратить отслеживание (опрос), необходимо передать значение, возвращаемое `watchPosition`, методу `clearWatch` (об этом чуть позже).

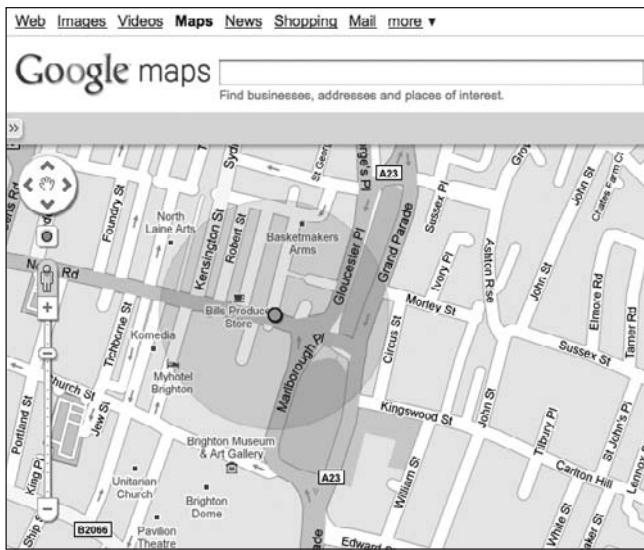


Рис. 9.1. Google Maps распознает поддержку геолокации и добавляет функции обнаружения местоположения

Оба метода, `getCurrentPosition` и `watchPosition`, при определении местоположения пользователя работают асинхронно. Из этого правила есть одно исключение, а именно: момент, когда пользователь заходит на сайт впервые, и браузер раскрывает диалоговое окно с вопросом, можно ли передать приложению текущее географическое положение. Если пользователь не согласен сообщать свое местоположение, то API геолокации вызовет обработчик ошибок (если вы его предусмотрели).

В спецификации говорится:

User agents must not send location information to web-sites without the express permission of the user. (Пользовательские агенты не должны отправлять сведения о местоположении на веб-сайты без явного разрешения пользователя.)

Таким образом, браузер решает, сообщать ли пользователю о том, что мы пытаемся определить его текущее местоположение. В разных браузерах это обрабатывается по-разному. Например, в Firefox открывается не модальное и не блокирующее окно, в котором пользователь может выбрать, сообщать о своем местоположении или нет (рис. 9.2). Ваше приложение в это время продолжает выполняться. В действительности, все современные настольные браузеры запрашивают разрешение приблизительно одинаково, то есть не выводят блокирующее сообщение, которое пользователь игнорировать не сможет.

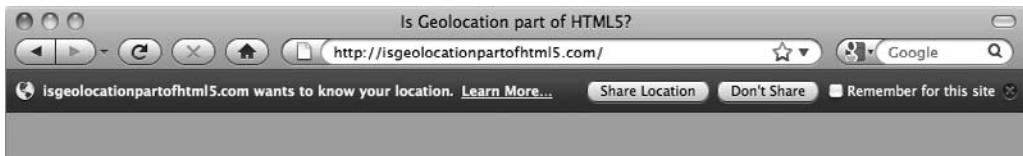


Рис. 9.2. Приложение просит Firefox сообщить ему о местоположении пользователя

В мобильных браузерах, таких как Mobile Safari и Opera Mobile, используется модальное диалоговое окно, которое открывается при каждом обращении к API. Это означает, что до того, как пользователь даст ответ, исполнение кода приостанавливается (рис. 9.3). Другие мобильные браузеры, например Firefox Mobile и WebKit для Android, не блокируют исполнение кода страницы, но все же недвусмысленно подталкивают пользователя к тому, чтобы он поскорее дал ответ.

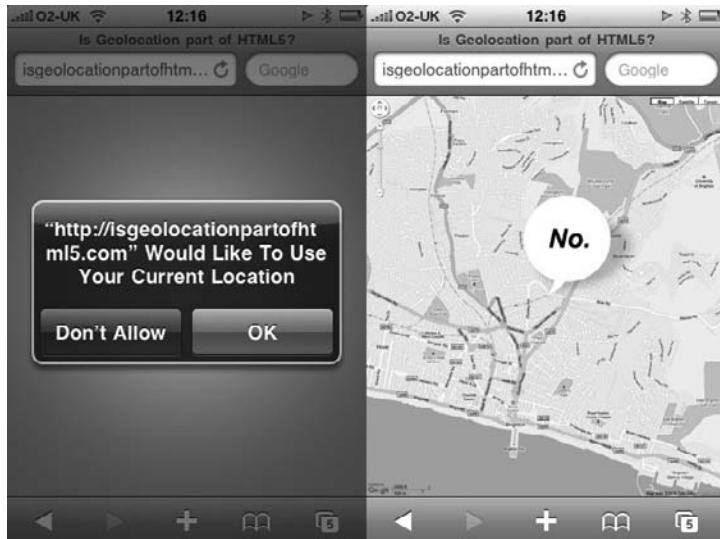


Рис. 9.3. Окно браузера Mobile Safari и модальное диалоговое окно, блокирующее исполнение приложения

МЕТОДЫ API

API геолокации находится внутри объекта `navigator` и содержит всего три метода:

- `getCurrentPosition`
- `watchPosition`
- `clearWatch`

Методы `watchPosition` и `clearWatch` — парные. Они работают по тому же принципу, что и `setInterval`, `setTimeout`. `watchPosition` возвращает уникальный идентификатор, который передается методу `clearWatch`, когда необходимо сбросить определенное отслеживание.

Как уже говорилось выше, методы `getCurrentPosition` и `watchPosition` очень похожи; они принимают одинаковые аргументы:

- обработчик успешного исполнения;
- обработчик ошибки;
- параметры геолокации.

Самый простой вариант использования API геолокации — передать методу `getCurrentPosition` обработчик успешного исполнения:

```
navigator.geolocation.getCurrentPosition(function (position) {
    alert('Мы тебя нашли!');
    // что-нибудь делаем с данными
    // о местоположении
});
```

ТЫ ПОПАЛСЯ: ОБРАБОТЧИК УСПЕШНОГО ИСПОЛНЕНИЯ

Если пользователь разрешает браузеру передать дальше информацию о его местоположении и никакой другой ошибки не происходит, то вызывается обработчик успешного исполнения, который передается методам `getCurrentPosition` и `watchPosition` в качестве первого аргумента.

Обработчик получает объект `Position`, обладающий двумя свойствами. Первое — это объект `coords`, содержащий информацию о координатах, а второе — отметка времени `timestamp`. Все самое интересное хранится в объекте координат. Данные объекта `Position` можно разбить на два класса. Данные первого класса отображаются во всех браузерах с поддержкой геолокации:

- `readonly attribute double latitude`
- `readonly attribute double longitude`
- `readonly attribute double accuracy`

Обратите внимание, что `accuracy` — это точность определения широты и долготы в метрах. С помощью этого значения вы можете сообщать пользователю, с каким диапазоном точности распознается его местоположение.

Хотя это трудно проверить вручную, вероятнее всего, эти данные предоставляются собственной службой производителя браузера. Например, у Google есть большая база геолокационных данных, с помощью которой, а также основываясь на информации запроса, сведениях об оборудовании, IP-адресе и черной магии, браузер обнаруживает ваше местоположение. Информация о местоположении — это всего лишь мгновенное состояние неких географических данных, и она никак не может помочь в определении скорости или направлении движения пользователя. Подробнее о вуду-магии, применяемой для проверки местоположения пользователя, мы поговорим в конце главы.

Используя данные координат, можно с легкостью привязать текущее местоположение пользователя к какому-нибудь объекту, например карте Google:

```
if (navigator.geolocation) {
    navigator.geolocation.getCurrentPosition
    // (function (position) {
        var coords = position.coords;
        showMap(coords.latitude,
            // coords.longitude, coords.accuracy);
    });
}
```

Во многих приложениях пользователю предоставляется возможность вручную указать собственное местоположение. Если доступен метод геолокации, то веб-сайт может пред-

лагать посетителям другие интересные возможности, постепенно обновляя содержимое страницы в зависимости от предоставленной информации.

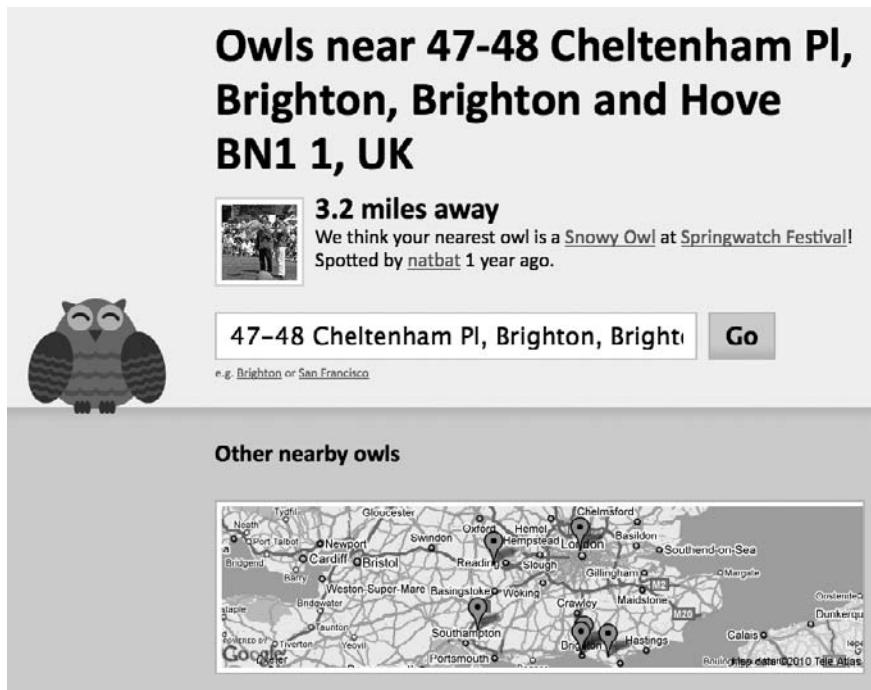


Рис. 9.4. Это приложение выводит информацию с учетом географических данных пользователей

Пример подобного приложения — веб-сайт «Совы рядом с вами», <http://owlsnearyou.com>. Если возможности геолокации доступны, то после входа на веб-сайт страница перезагружается с учетом вашего географического положения (рис. 9.4). Она сообщает, куда вы можете отправиться, чтобы поглядеть на сов после долгого трудового дня, посвященного разработке HTML5. Если функции геолокации недоступны, вы можете сами указать свое местоположение.

Данные второго класса объекта координат поддерживаются, но по умолчанию большинство (а в настоящее время — все) настольных браузеров возвращают значение `null`. Однако если пользовательское устройство поддерживает что-нибудь вроде технологии GPS, эти значения получить можно. Допустимые значения — `null`, 0 и числа с плавающей точкой:

- `readonly attribute double altitude`
- `readonly attribute double altitudeAccuracy`
- `readonly attribute double heading`
- `readonly attribute double speed`

Давайте рассмотрим данные второго класса.

ВЫЧИСЛЯЕМ СКОРОСТЬ

Все чаще и чаще разработчики смартфонов, планшетных компьютеров и ноутбуков оборудуют свои устройства GPS-модулями. На этих устройствах — в зависимости от установленного веб-браузера и степени интеграции с операционной системой — API геолокации может предоставить намного больше информации, не только статический набор координат (широта, долгота, точность). Через данные второго класса вы получаете доступ к информации о скорости, высоте и направлении движения.

Чаще всего при выполнении простого геолокационного запроса вам требуются только широта, долгота и точность — этих сведений для большинства приложений достаточно. Однако через API геолокации можно также извлечь информацию о скорости, высоте и направлении движения.

Обычно для того чтобы включить GPS-устройство, нужно приказать API использовать `highaccuracy`. Помните, однако, что включенное устройство GPS быстро расходует заряд аккумулятора, так что удостоверьтесь, что применяете данную технологию, потому что вам это нужно, а не просто потому что можете.

Для вычисления текущей скорости устройству необходимо вычислить среднее между несколькими сериями данных о местоположении (или что-то вроде этого, так?). Поэтому вам понадобится метод `geolocation.watchPosition`, и вы будете обновлять текущее значение скорости с получением каждого нового набора данных:

```
var speedEl = document.getElementById('speed');
navigator.geolocation.watchPosition(function (geodata) {
    var speed = geodata.coords.speed;
    if (speed === null || speed === 0) {
        speedEl.innerHTML = "You're standing still!";
    } else {
        // скорость дана в метрах в секунду
        // умножаем на 2,23 для получения миль в час
        speedEl.innerHTML = (speed * 2.23693629) + "Mph";
    }
}, function () {
    speedEl.innerHTML = "Unable to determine speed :-(";
}, { enableHighAccuracy: true });
```

Помимо скорости устройства, можно также извлечь высоту, точность высоты и направление движения. Если эти значения недоступны — из-за особенностей устройства, или потому что API геолокации не может считать именно эти данные, — то возвращается значение `null`.

АДРЕС 404: ОБРАБОТЧИК ОШИБОК

Второй аргумент методов `getCurrentPosition` и `watchPosition` — это обработчик ошибок. Его особенно важно использовать, если ваше приложение должно предоставлять альтернативный способ выбора местоположения (например, вручную) или сообщать пользователю об ошибках при определении координат. Обработчик ошибок срабатывает, когда пользователь запрещает распознавание местоположения. Однако также возможно,

что пользователь открыл веб-сайт с мобильного телефона, дал разрешение на определение координат, но потом вышел из зоны приема — при этом также происходит срабатывание обработчика ошибок.

Обработчик ошибок принимает единственный аргумент, содержащий объект ошибки позиционирования. У этого объекта два свойства:

- **readonly attribute unsigned short code**
- **readonly attribute DOMString message**

Свойство `code` может принимать только одно из следующих значений:

- `PERMISSION_DENIED` (числовое значение 1);
- `POSITION_UNAVAILABLE` (числовое значение 2);
- `TIMEOUT` (числовое значение 3).

Свойство `message` удобно применять во время разработки и отладки, однако его нельзя использовать для вывода сообщений пользователю. Но не потому, что оно содержит какую-то загадочную шифровку, получаемую из микросхем GPS-модуля, а потому, что оно определяется разработчиками браузеров (и не регулируется спецификацией). Кроме того, это будет, скорее, техническая информация, неинтересная вашему посетителю. К тому же оно не всегда доступно (например, в настоящее время оно недоступно в Firefox 3.6+).

АЛЬТЕРНАТИВНАЯ ОШИБКА: УСПЕХ

Однажды я зашел на веб-сайт, определяющий местоположение пользователя, из дома в Брайтоне на южном побережье Англии. Однако судя по карте на этой странице, приложение посчитало, что я находился прямо в центре Лондона. С помощью веб-консоли я заглянул во внутренности приложения и обнаружил, что диапазон точности составлял 140 000 метров. Определение местоположения с точностью до 140 километров — это крайне неточно! Вполне понятно, почему приложение не могло определиться, что же считать моим домом. Именно поэтому в приложениях с возможностями геолокации я настоятельно рекомендую проверять диапазон точности, даже если вызов завершается успешно. Если значение атрибута `accuracy` оказалось чрезмерно большим, лучше попросту проигнорировать географические данные, обработав данную ситуацию как ошибочную, и применить один из обходных путей, например попросить пользователя указать свое местоположение вручную. Однако степень точности зависит от контекста. Если ваше приложение должно помогать мне найти ближайшую больницу, то погрешность не может быть больше размера города. Если же приложение содержит данные о погоде по отдельным графствам, то может делать это с точностью до 90 миль — все равно мимо Англии не промахнется.

Вот простой пример, чтобы вы поняли, как применять обработчик ошибок:

```
if (navigator.geolocation) {
    navigator.geolocation.getCurrentPosition(function
        (position) {
            var coords = position.coords;
            showMap(coords.latitude, coords.longitude,
                coords.accuracy);
        }, function (error) {
            var errorTypes = {
```

```

        1: 'Permission denied',
        2: 'Position is not available',
        3: 'Request timeout'
    } ;

    alert(errorTypes[error.code] + ": means we can't
        - determine your position");
});
}

```

Если в приложении используется следующий код, а пользователь запрещает определять его координаты, то на странице выводится сообщение: «Permission denied: means we can't determine your position» («Доступ запрещен: невозможно определить ваше местоположение»).

НАСТРОЙКА ГЕОЛОКАЦИИ

Наконец, третий аргумент обоих методов, `getCurrentPosition` и `watchPosition` — это параметры геолокации. Как вы уже знаете, они необязательные, но могут быть следующими:

- `enableHighAccuracy` (булево значение, по умолчанию `false`);
- `timeout` (в миллисекундах, по умолчанию бесконечность (значение 0));
- `maximumAge` (в миллисекундах, по умолчанию 0).

Например, чтобы запросить высокую точность и тайм-аут длиной две секунды и запретить использовать старые географические данные, вызовите `getCurrentPosition` со следующими параметрами (`success` и `error` — это предопределенные функции):

```

navigator.geolocation.getCurrentPosition(success,
    - error, {
        enableHighAccuracy: true,
        timeout: 2000,
        maximumAge: 0
});

```

Мы уже встречались с параметром `enableHighAccuracy` — он заставляет устройство пытаться получить более точные значения широты и долготы. Параметр `timeout` сообщает приложению геолокации, как долго следует ждать, прежде чем можно будет сдаться и вызвать обработчик ошибки (обратите внимание, что отсчет не ведется, если приложение ожидает разрешения пользователя на определение местоположения). Когда допустимое время проходит, устанавливается код ошибки 3 (`TIMOUT`). Если значение параметра `timeout` равно нулю (это значение по умолчанию), то время ожидания не ограничивается.

Наконец, с помощью `maximumAge` можно разрешить или запретить браузеру использовать недавно кэшированные географические данные. Если в течение времени, указанного в параметре `maximumAge` (в миллисекундах), поступает новый запрос на определение местоположения, то возвращаются кэшированные данные, а не запрашиваются новые. Значение `maximumAge` также может быть равным `Infinity`, при этом браузер *всегда* использует кэшированное местоположение. Если присвоить параметру нулевое значение (это значение по умолчанию), то приложение при каждом запросе заново определяет географическое положение пользователя.

КАК ЭТО РАБОТАЕТ: СПЛОШНОЕ ВОЛШЕБСТВО

Для определения местоположения пользователя API геолокации использует несколько различных техник. Для обычных людей, включая меня, большая часть этого выглядит как черная магия, однако стоит все же иметь общее представление о том, что происходит в глубинах API и как это влияет на точность данных.

Самый очевидный метод получения данных геопозиционирования — GPS. В наши дни все больше устройств — от мобильных телефонов до ноутбуков — оснащаются модулями GPS. Если линия между вашим устройством и наземной станцией GPS (которая собирает данные со спутников для вычисления вашего местоположения методом триангуляции — да, еще больше черной магии) достаточно чистая, можно получить очень точные географические данные. С помощью GPS также можно вычислять высоту, скорость и направление, которые, как мы видели, входят во второй класс свойств объекта координат и используются, когда включен параметр высокой точности.

Еще один способ — использовать информацию сети. Так обычно и делается, если речь идет о настольном браузере, таком как Firefox. Триангуляция по данным сети wi-fi с учетом IP-адресов позволяет довольно точно определять местоположение пользователей. Разработчик в своем приложении обращается к API геолокации браузера, а браузер, в свою очередь, отправляет запрос службе стороннего производителя, например Skyhook или Google. Хотя в данном случае результат может быть не таким точным, как полученный с помощью GPS, это все же очень хороший альтернативный вариант для помещений или тесно застроенных городов, где GPS работает не очень хорошо.

В целом, не так уж важно, как делаются все эти фокусы с геолокацией. Нужно лишь помнить, что GPS дает более точные данные, но при этом сильнее разряжает аккумулятор устройства. Короче, это очень классная черная магия.

ЗАКЛЮЧЕНИЕ

Если ваше приложение каким-то образом использует геолокацию, то с вашей стороны будет неимоверной глупостью обойти внимание этот удивительно простой API. Затраты на реализацию почти нулевые. Фактически вся работа заключается в создании пользовательского интерфейса. Что еще важнее для меня как для разработчика — независимо от того, *как* браузер определяет местоположение моего пользователя, все будет работать одинаково на настольном компьютере, ноутбуке и мобильном телефоне. Код везде один и тот же, и это очень простой код.

Так как функции геолокации встроены в IE9 и остальные браузеры, поддержка этой возможности находится на хорошем уровне. Кроме того, мы можем применять техники заполнения и откатываться к методам геолокации, основанным на JavaScript и традиционных сервисах вычисления IP-адресов (об этом подробнее в главе, посвященной заполнителям, в конце книги).

Помните только, что геолокация должна прогрессивно улучшать страницу, а не заставлять пользователя полагаться на данные, приходящие от API геолокации, так как в некоторых случаях они оказываются не совсем точными.

Глава 10

Сообщения и фоновые вычисления

Реми Шарп



Web Messaging и Web Workers — это разные API, объединенные общим коммуникационным API, и поэтому мы будем рассматривать их вместе, в одной главе. В спецификацию HTML5 входит только API сообщений, однако оба станут ценным дополнением к любому веб-приложению.

API сообщений (API Messaging) позволяет отправлять сообщения между доменами. Эту возможность очень долго блокировали политики безопасности Ajax, и не без причины, однако сегодня она становится доступной — при условии, конечно, обеспечениякой безопасности.

ПРИМЕЧАНИЕ

Новый объект XMLHttpRequest уровня 2 (<http://www.w3.org/TR/XMLHttpRequest2/>), который мы в данной книге не рассматриваем, хотя он уже присутствует в WebKit и Firefox, поддерживает междоменные запросы (с достаточно высоким уровнем серверной безопасности). У него также есть события хода исполнения, позволяющие контролировать загрузки.

Фактически браузеры — это *однопоточные* приложения. Содержимое страницы не отображается, пока исполняется или, возможно, разбирается код JavaScript. Аналогично, когда JavaScript выполняет длинную и сложную функцию, браузер застывает. Благодаря фоновым вычислениям (API Web Workers) в браузерах можно реализовать упрощенный вариант *потоков*. Используя фоновые вычисления, разработчик выделяет определенный блок кода, который исполняется, никак не влияя на функциональность браузера — словно это делается в отдельном потоке операций, пока основная работа браузера продолжается без изменений.

КРОШЕЧНЫЙ ЧАТИК С ПОМОЩЬЮ API MESSAGING

В первую очередь я хочу познакомить вас с API сообщений, так как фоновые вычисления, веб-сокеты и API событий, отправляемых сервером (последние два обсуждаются в следующей главе), базируются на этом методе обмена информацией. Считайте это небольшим введением в обмен данными.

API сообщений хорошо поддерживается во всех браузерах (да, включая IE) и предлагает очень простой способ публикации обычных текстовых сообщений из одного источника (или, для нас с вами — из одного домена) в другом. Например, если нужно отправить информацию в окно, которое находится в плавающем фрейме (iframe), это можно сделать с помощью API Messaging. И даже если окно находится в совершенно другом домене, а не на том же сайте, которому принадлежит плавающий фрейм, все равно можно использовать тот же API.

ОТПРАВКА СООБЩЕНИЙ МЕЖДУ ДОМЕНАМИ

Предположим, что у Брюса есть документ, который должен обмениваться данными с моим документом — например, в плавающем фрейме или всплывающем окне. Для этого документу Брюса необходима ссылка на объект `window` (моего документа). Имея ее, он может

вызвать метод `postMessage` и передать в нем какой-то текст. Код JavaScript в документе Брюса будет выглядеть примерно так:

```
var t = document.getElementsByTagName('iframe')[0];
t.contentWindow.postMessage('favourite instrument?',
  'http://brucelawson.co.uk');
```

Аргумент целевого источника, который передается `postMessage` вторым, обязательный. Он должен совпадать с источником вашего объекта `contentWindow` (целевым окном, то есть в данном примере — с моим документом). Если источники не совпадают, то возвращается ошибка безопасности и выполнение сценария останавливается. Если источник не передается, JavaScript возвращает синтаксическую ошибку — не слишком полезно, но все же помогает не забывать об обязательном аргументе. И еще один совет: дождитесь момента, когда загрузка целевого объекта завершится. В конце концов, это документ, который необходимо разобрать и загрузить. Если попытаться отправить сообщение документу, загруженному не полностью, JavaScript вернет похожую синтаксическую ошибку и перестанет работать.

ПРИМЕЧАНИЕ

В этом коде используется метод `addEventListener`, а не `onmessage`, так как старые версии Firefox не реагировали на `onmessage` для объекта `window`. В любом случае, лучше всегда делать именно так, как здесь. Помните только, что при этом также нужно будет подцеплять IE с помощью `attachEvent`, что не показано в моем примере.

Плавающий фрейм на странице Брюса ссылается на мой документ, содержащий следующий код JavaScript:

```
window.addEventListener('message', function (event) {
  if (event.data == 'favourite instrument?') {
    if (event.origin == 'http://remysharp.com') {
      event.source.postMessage('brand new clarinet',
        event.origin);
    } else if (event.origin == 'http://brucelawson.co.uk') {
      event.source.postMessage('rusty old trombone',
        event.origin);
    }
  }
}, false);
```

В этом сценарии я устанавливаю приемник событий для прослушивания сообщения, передаваемых объекту `window`. У объекта `event` есть свойство `data` — оно содержит полученное сообщение. Помимо свойства `data`, у объекта `event` есть и другие полезные свойства: `origin` и `source`.

Свойство `event.origin` позволяет узнать домен, из которого пришло сообщение. Как показано в предыдущем коде, основываясь на содержимом данного свойства, я решую, обрабатывать ли сообщение. Это простейший уровень управления политиками.

Свойство `event.source` указывает на объект окна, создавший исходный вызов к моему документу, то есть на документ Брюса. Это полезно, если мы собираемся обмениваться сообщениями в обе стороны. Разумеется, обработчик событий `onmessage` может делать намного больше, например конструировать запросы Ajax к серверу в том же домене.

А МОЖНО ЛИ ОТПРАВЛЯТЬ ЧТО-НИБУДЬ ЕЩЕ КРОМЕ СТРОК?

В предыдущих примерах я передавал только строки. Но что делать, если вам необходимо отправить какие-то другие данные? Как поступить с объектом, отягощенным свойствами и значениями?

Что ж, хорошие новости — в спецификации рассказывается, что должно происходить, если браузеру нужно безопасно переслать данные из одного источника в другой. Там описано, как клонировать данные и как их обрабатывать.

Однако... в большинстве браузеров этот процесс не поддерживается. В действительности, браузеры всего лишь превращают объекты в строки. Это отвратительно, потому что вместо продуманно сконструированного объекта вы получаете в свойстве `event.data [object Object]`. Мы это уже видели раньше — в главе 6 «Хранение данных», когда пытались сохранить объекты в хранилище `localStorage`. Таким образом, как и в том случае, можно использовать `JSON.stringify` для преобразования объекта JavaScript в строку, передавать его `postMessage`, а потом, на стороне получателя, снова превращать в исходный объект JavaScript с помощью `JSON.parse`.

`JSON.stringify` и `JSON.parse` — это удобные методы для пересылки сложных объектов от окна к цели, и мы еще встретимся с ними в следующем разделе, посвященном фоновым вычислениям (а также в следующей главе, которая рассказывает о веб-сокетах и событиях, отправляемых сервером).

РАЗБИЕНИЕ НА ПОТОКИ С ПОМОЩЬЮ ФОНОВЫХ ВЫЧИСЛЕНИЙ

Фоновые вычисления — это часть отдельной спецификации. Они не входят в спецификацию HTML5, но незаменимы при построении веб-приложений.

Фоновым вычислением (Web Worker) называется блок кода JavaScript, выполняющийся в фоновом процессе по отношению к основному процессу браузера. Фактически это поток. Как бы это ни называлось, нам важно, что фоновое вычисление не мешает работе главного потока вычислений браузера.

Браузер отвечает за то, чтобы запрашивать и разбирать файлы, визуализировать представления, исполнять код JavaScript, поддерживать пользовательский интерфейс и выполнять массу других задач. Если время и ресурсы браузера будут расходоваться на что-то еще, всем этим задачам придется просто ждать. И здесь на помощь приходят фоновые вычисления.

ЗАЧЕМ ИСПОЛЬЗОВАТЬ ФОНОВОЕ ВЫЧИСЛЕНИЕ?

Если из-под ваших рук когда-нибудь выходил код JavaScript, буквально сшитый на живую нитку, запуск которого заставлял браузер возмущаться и кипеть, то вам прекрасно известны все недостатки исполнения программ в единственном потоке. В конце концов, смышленые браузеры все же сдаются и предлагают прервать выполнение нестабильного кода, как показано на рис. 10.1.

Намного больше беспокойства вызывают неочевидные ситуации. Предположим, вы разрабатываете новейшее и крутецкое веб-приложение, предлагающее пользователям исклю-

чительные возможности обработки фотографий. Вы используете все навороченные трюки, о которых узнали из этой книги — холст, хранилища, функции автономного режима, — но когда дело доходит до наложения фильтра на фотографию, выясняется, что эта процедура занимает 30 секунд. И вас беспокоит не то, что 30 секунд — это слишком много. Просто в течение этих тридцати секунд браузер больше вообще *ничего* не способен делать. А что, если пользователь изменит решение и захочет отменить наложение фильтра? Очевидно, именно таких ситуаций больше всего и хочется избежать.



Рис. 10.1. Некоторые браузеры прерывают исполнение кода JavaScript, давая вам возможность утихомирить сорвавшуюся с катушек программу

Для решения подобной проблемы фоновое вычисление подойдет просто идеально. Если убрать обработку фильтра в фон, то тогда главное окно браузера — и остальные функции вашего приложения — сможет реагировать на запросы пользователя как обычно.

Создание и реализация фоновых вычислений

Для того чтобы узнать, поддерживаются ли фоновые вычисления, проверьте, определен ли соответствующий объект:

```
if (typeof Worker != "undefined") {
    // всякие крутые штуки
}
```

Теперь, зная, что нужная функциональность поддерживается (Safari, Safari Mobile, Chrome, Opera, Firefox и IE10 все поддерживают фоновые вычисления), давайте создадим новое фоновое вычисление:

```
var worker = new Worker('my_worker.js');
```

Новый объект `worker` со сценарием в файле JavaScript `my_worker.js` запущен и готов к работе. И вас даже можно простить, если вы считаете, что можете из своего документа вызывать методы объекта фонового вычисления и получать данные обратно в документ. Ха-ха! Нет,

общаться с объектом фонового вычисления можно исключительно сообщениями. Словно в сцене из «Ромео и Джульетты», браузер и объект `worker` будут обмениваться любовными письмами.

Единственный способ передать информацию объекту фонового вычисления — отправить ему сообщение через `postMessage`:

```
worker.postMessage('hello worker!');
```

Обратите внимание, что, в отличие от `postMessage` в API сообщений, метод `postMessage` для фоновых вычислений требует наличия только одного аргумента. Единственный способ получить информацию от объекта фонового вычисления — применить обработчик событий `onmessage`:

```
worker.onmessage = function (event) {
    alert('The worker just sent me this: ' + event.data);
};
```

Вы наверняка узнали комбинацию `postMessage/onmessage` из раздела, посвященного API `Messaging`. Помните, что в этом API можно отправлять и получать только строки? Значит, вас не удивит, что API фоновых вычислений накладывает то же ограничение.

Аналогично, код *внутри* объекта фонового вычисления также должен полагаться на сочетание `postMessage/onmessage`. Однако этот объект ограничен намного сильнее, чем обычный документ: он изолирован почти от всего, что его окружает, и может обращаться только к нескольким избранным API и функциям, как вы увидите в следующем разделе.

Единственный метод, доступный через объект фонового вычисления, — это `terminate`, который делает в точности то, о чём говорит его название: останавливает фоновое вычисление. Вычисление прекращается, и соответствующий объект становится бесполезным. Кстати, возобновить вычисление невозможно, можно только создать новое.

Что можно делать внутри фонового вычисления

Внутри фонового вычисления у вас нет доступа к таким излишествам, как DOM. В действительности, для того чтобы выполнить какую-то задачу в DOM, нужно сначала *подготовить* данные в фоновом вычислении, а затем передать родительскому документу, чтобы он выполнил всю работу с DOM.

ПРИМЕЧАНИЕ

В настоящее время реализаций фоновых вычислений, которые бы поддерживали доступ к базам данных `IndexedDB`, нет, хотя работа в этом направлении идет, и определенные исправления разрабатываются. В первом издании этой книги мы говорили то же самое о базах данных SQL для веб — но с тех пор ситуация изменилась. Спецификация `IndexedDB` еще очень нова, и я надеюсь, что поддержка этой технологии появится сразу же, как только стабилизируется реализация `IndexedDB`.

Однако внутри фонового вычисления (согласно спецификации) можно выполнять определенные манипуляции:

- передавать сообщения с помощью `postMessage` и слушать входящие сообщения, используя `onmessage`;
- закрывать текущее фоновое вычисление с помощью `close`;

- устанавливать приемники событий;
- устанавливать объект XMLHttpRequest для запросов Ajax;
- использовать таймеры, такие как setTimeout, setInterval, и соответствующие методы сброса таймеров;
- применять все ключевые функции JavaScript: eval, isNaN, escape и т. д.;
- работать с объектами местоположения (`href` сценария фонового вычисления);
- использовать веб-сокеты (о которых мы поговорим в следующей главе);
- использовать EventSource (также в следующей главе);
- работать с базами данных SQL для веб (реализовано только в Safari и Chrome);
- работать с IndexedDB;
- использовать фоновые вычисления;
- загружать сценарии и библиотеки с помощью importScripts.

Следующий короткий код позволяет моему фоновому вычислению my_worker.js общаться с документом из предыдущего фрагмента кода:

```
this.onmessage = function (event) {
    if (event.data == "hello worker!") {
        postMessage("hello there, right back at you");
    } else {
        postMessage("Can't you see I'm busy, leave me alone");
    }
};
```

Полезно знать, что в нормальном документе ключевое слово `this` ссылалось бы на глобальный контекст, то есть объект `window`. Глобальный контекст для фонового вычисления — это сам экземпляр фонового вычисления. Это означает, что ключевое слово `this` в `setTimeout` и `setInterval` также относится к экземпляру фонового вычисления (в обычном документе это был бы объект `window`).

В предыдущих примерах наше фоновое вычисление ничего особенно полезного не делало. А как насчет вычисления, которое будет искать простые числа? Для этого потребуется очень загруженный цикл на JavaScript, который будет повторяться много раз в поисках простых значений. И все это в то время, пока посетитель рисует на холсте, не отрывая приложение от поиска. Может, это и не самый реальный пример, но для демонстрации возможностей фоновых вычислений подходит.

В главном документе мы будем обрабатывать запуск фонового вычисления и рисование на холсте. В фоновое вычисление мы сбросим только код для поиска простых чисел.

```
var worker = new Worker('prime.js'),
    prime = document.getElementById('prime');
worker.onmessage = function(event) {
    prime.innerHTML = event.data;
};
```

Страница продолжит обрабатывать события `mousedown`, `mousemove` и `mouseup`, с помощью которых пользователь рисует на холсте. Сценарий `prime.js` выглядит так:

```

onmessage = function (event) {
    // неважно, какое сообщение, главное -
    // запустить задание
    run();
};

function run() {
    var n = 1;
    search: while (true) {
        n += 1;
        for (var i = 2; i <= Math.sqrt(n); i += 1)
            if (n % i == 0)
                continue search;
        // нашли простое число!
        postMessage(n);
    }
}

```

Как только фоновое вычисление prime.js получает любое сообщение, оно запускает поиск простых чисел. Когда ресурсоемкий процесс поиска выводится за пределы приложения, все остальное выполняется без задержек, и пользователь может без лишних волнений посвятить себя созданию произведения искусства, как показано на рис. 10.2.



Рис. 10.2. Все, о чём вы когда-либо мечтали: одновременный поиск простых чисел и рисование на холсте. Спасибо фоновым вычислениям!

Матрешки: фоновые вычисления внутри фоновых вычислений

Если вы достаточно наблюдательны, то заметили, что существует возможность создавать фоновые вычисления внутри других фоновых вычислений. В настоящее время это поддерживается только в Firefox и Opera, но данная возможность входит в спецификацию, так что можно ожидать, что в скором времени она появится и в других браузерах.

Что же это все означает? А то, что можно создать одно фоновое вычисление, которое затем разделит свою работу на множество маленьких заданий и передаст их фоновым вычислениям второго уровня. Давайте снова вернемся к примеру с наложением сложного фильтра на фотографию в вашем супернавороченном веб-приложении для обработки изображений. Для того чтобы ускорить обработку фотографии — если, конечно, это применимо с конкретным типом фильтра, — ее можно разбить на зоны и передать каждый фрагмент данных изображения отдельному фоновому вычислению.

По мере того как каждое из фоновых вычислений второго уровня возвращает данные, счетчик заданий уменьшается, и, когда он обнуляется, главное фоновое вычисление возвращает готовые обработанные данные изображения родительскому документу. Приложение будет выглядеть приблизительно так (я опустил реализацию некоторых функций, просто чтобы продемонстрировать идею):

```
var pendingWorkers = 0,
    results = {},
    workingWidth = 100;

onmessage = function (event) {
    var imageData = JSON.parse(event.data),
        worker = null;

    pendingWorkers =
        ~ getNumberOfWorkers(imageData.width/
        ~ workingWidth);
    // сбрасываем старые результаты
    results = {};

    for (var i = 0; i < pendingWorkers; i++) {
        worker = new Worker('photofilter.js');
        worker.postMessage(JSON.stringify({
            imageData: imageData,
            x: i * workingWidth,
            width: workingWidth
        }));
        worker.onmessage = storeResult;
    }
};

function storeResult(event) {
    var result = JSON.parse(event.data);

    buildUpImageData(result);
}
```

```

pendingWorkers--;
if (pendingWorkers <= 0) {
    postMessage(JSON.stringify(results));
}
}

```

Получая сообщение от фонового вычисления второго уровня, главное фоновое вычисление уменьшает количество незавершенных фоновых вычислений второго уровня. Как только получены все фрагменты данных изображения, итоговый результат возвращается родительскому документу.

Фоновое вычисление второго уровня, `photofilter.js`, содержит такой код для обработки небольшой порции данных изображения:

```

onmessage = function (event) {
    var data = JSON.parse(event.data);

    // удивительные фокусы с обработкой приложения
    var imageData = amazingImageProcess(data.imageData,
        - data.x, data.width);
    postMesage(JSON.stringify({
        imageData: imageData,
        x: data.x
    }));

    // самозакрытие
    close();
};


```

Также обратите внимание, что, выполнив задание, `photofilter.js` вызывает метод `close()`. Так как повторно оно не понадобится, это фоновое вычисление второго уровня просто завершается.

Импорт сценариев и библиотек в фоновое вычисление

Фоновые вычисления предназначены именно для того, чтобы вы могли выделить блок кода или некоторую функциональность и выполнять их в автономной среде (то есть путем фонового вычисления). Помимо этого, фоновые вычисления позволяют загружать внешние файлы JavaScript и библиотеки с помощью метода `importScripts`.

Это один из немногих методов, применение которого ограничено фоновыми вычислениями. Он принимает в качестве аргументов список адресов URL и синхронно загружает их в фоновое вычисление. Внутри одного вычисления можно одновременно загрузить один сценарий или сразу несколько:

```

importScripts('xhr.js');
importScripts('prime.js', 'number_crunch.js',
    - 'captain_crunch.js');

```

Сценарии обрабатываются по одному. Кроме того, сценарии должны находиться там же, где файл фонового вычисления — тот же домен, спаме и т. д. Объект фонового вычисления синхронно загружает в себя код JavaScript и возвращается к исполнению дальнейшего кода только после того, как завершается работа этого сценария.

Разделение нагрузки с помощью общих вычислений

Еще один тип фоновых вычислений называется общими вычислениями, `SharedWorker`. В настоящее время они поддерживаются только в Chrome, Safari и Opera. Общее вычисление — это почти то же самое, что обычное фоновое вычисление. Единственное отличие — к одному экземпляру общего вычисления могут обращаться несколько документов. Таким образом, если в вашем приложении несколько всплывающих окон или плавающих фреймов, все эти документы могут использовать единственный объект общего вычисления. Это полезно в таких приложениях, как, например, Gmail или Facebook, где одни и те же клиентские данные (скажем, сообщения или чат) необходимо обновлять сразу в нескольких окнах. Объект вычисления способен открывать и менять информацию на клиентской стороне в базе данных SQL для веб и в базе данных IndexedDB (подробнее о хранилищах рассказывается в главе 6), а также поддерживать подключение к серверу, в том числе через веб-сокеты, управляя обменом данных в режиме реального времени. Все изменения объекта общего вычисления помещает в базу данных сообщений на стороне клиента, а затем отправляет обновления в каждое из всплывающих окон, фреймов и т. д. с помощью метода `postMessage`.

Это означает, что условий состязания не возникает и синхронизация данных не нарушается, что могло бы происходить, если бы все эти окна и фреймы индивидуально обращались к серверу и пытались менять данные на клиентской стороне. Объект общего вычисления — это общая точка контакта, через которую проходят все данные и осуществляются все операции.

Что касается обмена данными, `SharedWorker` работает немного непривычно. Во-первых, он не использует концепцию портов — этот объект подобен массиву, он содержит ссылки на каждый из открытых каналов обмена данными. Кроме того, если добавлена привязка к событию `message` при помощи `addEventListener`, то общее вычисление нужно запускать вручную — я продемонстрирую это в следующем примере кода.

При создании объекта `SharedWorker` внутри него срабатывает событие `connect`, которое можно использовать для подсчета количества соединений между общим вычислением и другими документами.

В документе, создающем объект `SharedWorker`, должен содержаться следующий код:

```
var worker = new SharedWorker('messages.js');
worker.port.addEventListener('message', function(event) {
    var messages = JSON.parse(event.data);
    showNewMessages(messages);
}, false);
worker.port.start();
```

Вы видите, что обращение к общему вычислению осуществляется через свойство `port`. Благодаря ему не только происходит общение с общим вычислением; это свойство отличает общее вычисление от обычного фонового вычисления. Поскольку в коде используется привязка к событию `message` с помощью `addEventListener`, общее вычисление необходимо подключать вручную — через метод `.start()`. Если бы использовалось событие `onmessage`, то этот метод в коде не потребовался бы. Теперь рассмотрим структуру общего вычисления `messages.js`:

```

importScripts('xhr.js');
importScripts('database.js');

var connections = [];

onconnect = function(event) {
    connections.push(event.ports[0]);
}

var xhr = new XHR('/get-new-messages');
xhr.oncomplete = function (messages) {
    database.updateMessages(messages);

    for (var i = 0; i < connections.length; i++) {
        connections[i].postMessage(JSON.stringify(messages));
    }

    xhr.send(); // создает бесконечный цикл
};

xhr.send();

```

Когда клиентский документ подключается к общему вычислению, срабатывает событие `connect`, что позволяет мне захватить порт соединения. Я получаю значение с помощью ссылки `event.ports[0]` — несмотря на то, что более одного элемента внутри свойства `ports` никогда не бывает. Тем не менее ссылка на общее вычисление находится там, и можем использовать ее для отправки и получения сообщений.

Как вы видите из предыдущего примера, когда исполняется функция Ajax `oncomplete`, я в цикле прохожу по всем открытым портам и на каждый из них отправляю сообщение о пришедшем в почту новом письме. Таким образом, все подключенные клиенты работают как обычные терминалы и не знают ничего о реальной работе, которая происходит при сохранении сообщений в клиент-серверной базе данных.

Отладка фонового вычисления

В сфере разработки веб-приложений мы дошли до этапа, когда каждому стали доступны инструменты отладки, не идущие ни в какое сравнение с тем, чем приходилось довольствоваться десять лет назад. Все новейшие браузеры поставляются с собственным отладчиком JavaScript (хотя в Firefox до сих пор требуется устанавливать встраиваемый модуль Firebug). Это просто рай для отладчиков, которым в старые и недобрые времена некуда было деваться от ужасных всплывающих сообщений.

Однако, работая с фоновыми вычислениями, вы погружаетесь в изолированную среду без доступа к консольным отладчикам. Для фонового вычисления не существует стандартного способа выполнить что-то вроде `console.log("кто виноват?")`. Если вы еще недостаточно разочарованы, то скажу также, что и всплывающие сообщения здесь тоже недоступны.

Тем не менее! Надежда есть! Ребята из WebKit задумались над этой проблемой, и в инструментах разработчика на панели `scripts` (сценарий) в строке `Web Workers` (фоновые вычисления) появился флагок `Debug` (Отладка) (в Chrome и Safari). Он меняет способ выполнения

фоновых вычислений (за кулисами они выполняются в `setTimeout`), но все же вы теперь можете добавлять в свой код `console.log` для упрощения отладки (см. рис. 10.3).

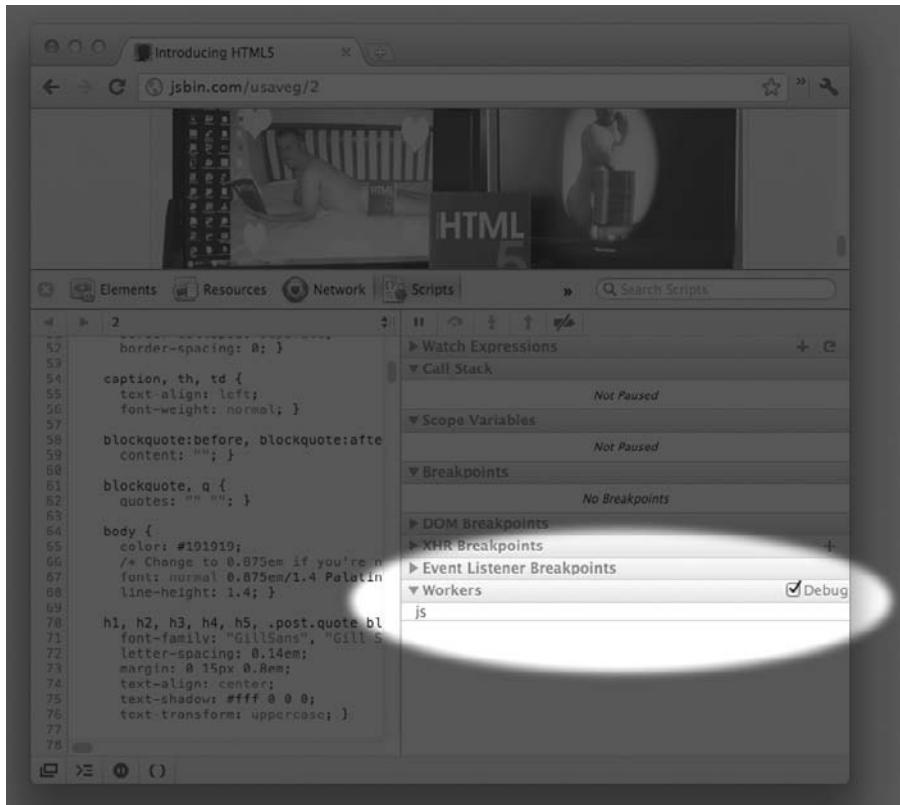


Рис. 10.3. Установите флажок Debug (Отладка), чтобы исполнять внутри фонового вычисления `console.log`

То, что в Chrome и Safari появились инструменты отладки, прекрасно. Но как отлаживать приложения, скажем, в Firefox? Вам потребуется создать собственную систему публикации отладочных сообщений. Однако, как и для любых видов общения с фоновыми вычислениями — будь то отладочное сообщение или результаты делегированных задач, — вам необходим некий согласованный язык, который смогут использовать фоновые вычисления и главный документ. Данный язык должен разграничивать все эти типы событий, и он будет ограничен только вашим приложением. Например, перед отладочными сообщениями можно вставлять ключевое слово `log`:

```
importScripts('xhr.js');

var xhr = new XHR('/someurl');
xhr.oncomplete = function (data) {
  log('data contains ' + data.length + ' items');
};
```

```
xhr.send();

function log(msg) {
    postMessage('log ' + msg);
}
```

Обратите внимание, что xhr.js — это мой сценарий для XMLHttpRequest, который возвращает данные JSON. Вам придется написать свой!

В главном документе, в событии onmessage я буду проверять префиксы сообщений и действовать соответствующим образом:

```
var worker = new Worker('xhr_thang.js');
worker.onmessage = function (event) {
    var data = event.data.split(' '),
        action = data.shift(), // захватить первое слово
        msg = data.join(' '); // поместить сообщение обратно
        // together

    if (action == 'log') {
        console.log(msg);
    } else {
        // другое действие
    }
};
```

ПРИМЕЧАНИЕ

Вполне возможно, что выполнение фонового вычисления прервется методом, о котором вашему коду неизвестно. Если браузер по какой-то причине прерывает фоновое вычисление, то срабатывает событие `worker.onerror`. Для того чтобы останавливать его вручную, прямо в код фонового вычисления добавляется метод `.close()`. Так вы получаете возможность уведомлять об остановке все подключенные документы.

В данном примере в начале всех сообщений я добавляю префикс с указанием действия: `log`, `set`, `run` и т. п. Самое важное здесь то, что теперь в других браузерах, отличных от Safari и Chrome, у меня появилась возможность проверять данные из фонового вычисления, отправляя данные в функцию журнала.

Также полезно иметь возможность порыться внутри фонового вычисления — по крайней мере, мне это очень пригодилось во время экспериментов с JavaScript. В другой среде, за пределами фонового вычисления, можно открыть любую консоль (Firebug, Dragonfly и т. д.) и там регистрировать и проверять любые свойства объекта `window`, `document`, свойств их свойств... просто чтобы понять, что именно поддерживается и какие инструменты вам предоставлены. Так как фоновое вычисление — это закрытая среда, изучать ее приходится собственными методами. В один из примеров для этой книги я включил консоль, при помощи которой вы сможете проинспектировать фоновое вычисление, протестировать его код и посмотреть, что оно дает на выходе. Эта консоль находится по адресу <http://introducinghtml5.com/examples/ch10/echo.html> (рис. 10.4).

```

Worker sandbox
http://introducinghtml5.com/examples/ch10/echo
Google

console to inspect the contents of a worker

console.log(this);

> location
http://introducinghtml5.com/examples/ch10/echo.js{
  href: "http://introducinghtml5.com/examples/ch10/echo.js",
  protocol: "http:",
  host: "introducinghtml5.com",
  hostname: "introducinghtml5.com",
  port: "",
  pathname: "/examples/ch10/echo.js",
  search: "?",
  hash: ""
}

> navigator
[object Navigator]{
  appName: "Netscape",
  appVersion: "5.0 (Macintosh; en-GB)",
  platform: "MacIntel",
  userAgent: "Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.6; en-GB;
rv:1.9.2.3) Gecko/20100401 Firefox/3.6.3"
}

> console.log(this);
[object DedicatedWorkerGlobalScope]{
  onmessage: "[xpconnect wrapped nsIDOMEventListener]"
}

```

Рис. 10.4. Демонстрационная консоль для изучения содержимого фонового вычисления

РЕЗЮМЕ

Существует несколько API, использующих схожие техники обмена сообщениями, — мы поговорили о части из них и подробнее поговорим о еще парочке в следующей главе. Разобравшись во множестве различных способов обмена данными внутри веб-приложения, вы получите целый арсенал инструментов для создания многопоточных, многооконных, кросс-доменных монструозных приложений.

Фоновые вычисления дают вам ту функциональность, появления которой с нетерпением ожидали все разработчики: возможность исполнять задания JavaScript в фоновом режиме — так, чтобы пользователь в это время продолжал спокойно работать в браузере. Теперь, когда вы знаете, что реализовать это проще простого, я уверен, что на ваших веб-сайтах никогда не будет появляться отвратительный индикатор, сообщающий пользователю: «ждите ответа, браузер работает!»

Глава 11

РАБОТА В РЕЖИМЕ РЕАЛЬНОГО ВРЕМЕНИ

Реми Шарп



Функции, работающие в реальном времени, — это сетевой аналог сундуков с золотыми монетами. Обнаружить их — большая радость, но проделать всю необходимую подготовительную работу бывает крайне непросто. В этой главе я продемонстрирую вам избранные технологии и докажу, что код на клиентской стороне может быть очень простым.

Для внедрения возможностей реального времени в веб-приложения можно использовать любой из двух вариантов: веб-сокеты (WebSockets) и события, отправляемые сервером (Server-Sent Events). Веб-сокеты позволяют определять направленный поток, соединяющий приложение с сервером (для специалистов по серверным технологиям: TCP-подключение). Это означает двусторонний обмен данными между клиентом и сервером, который осуществляется в режиме реального времени. Типичным приложением уровня «Hello, world» будет клиент чата, но в действительности потенциал огромен. Сокеты уверенно приближаются к тому, чтобы полностью заменить собой код, основанный на Comet. Comet использует множество разнообразных (зачастую хакерских) техник для реализации потоковой передачи данных с сервера в режиме реального времени. Веб-сокеты упрощают процесс на клиентской стороне, о чем мы поговорим в этой главе.

События, отправляемые сервером, — это технология, также известная под названием EventSource. Она в реальном времени проталкивает в клиентский браузер сообщения, отправляемые сервером. Это очень удобно для приложений, которым требуется получать от сервера информацию, даже когда пользователь не производит никаких действий, инициирующих обмен данными; к такой информации относится поток ценовых данных, «живые» обновления диаграмм, любые данные реального времени, которые вам может понадобиться отслеживать.

ВЕБ-СОКЕТЫ: РАБОТА С ПОТОКОВЫМИ ДАННЫМИ

Web Sockets — это один из новейших API. Он лежит за пределами спецификации HTML5, но его возможности весьма удобны и очень важны для многих веб-приложений реального времени.

Веб-сокеты позволяют создавать двухстороннее соединение между сервером и клиентом, то есть браузером. Это соединение работает в режиме реального времени и открыто постоянно — до того момента, пока вы явно не закроете его. Таким образом, если сервер хочет отправить какую-то информацию клиенту, это сообщение вталкивается в соединяющий его с браузером канал немедленно, без задержек.

Именно эта цель ставилась разработчиками Comet, и они добились желаемого. Comet — это неплохой способ для поддержания соединения с сервером и передачи данных в режиме реального времени, правда, требующий применения множества различных хитростей. В конечном итоге, если ни одна из хитростей не срабатывает, вся схема превращается в обычное опрашивание Ajax, не поддерживающее масштабирование и постоянно бомбардирующее сервер запросами.

ПРИМЕЧАНИЕ

Если браузер не поддерживает веб-сокеты, всегда можно использовать Flash. Хироши Ичкава написал заполнитель для веб-сокетов на Flash. Адрес его сайта — <https://github.com/gimite/web-socket-js>.

Если сокет открыт, сервер может отправлять данные во все подключенные сокеты, и ему не приходится непрерывно отвечать на входящие запросы Ajax. Это большой шаг вперед, переход от опроса к проталкиванию данных — от модели с обратной связью к упреждающей модели. Именно этого в Comet добивались с помощью трюков, а веб-сокеты предлагаю безо всяких дополнительных надстроек прямо в браузере.

СОКЕТЫ РЕШАЮТ ПРОБЛЕМУ ЗАДЕРЖЕК В ПРИЛОЖЕНИЯХ РЕАЛЬНОГО ВРЕМЕНИ

Огромное преимущество веб-сокетов — малое время ожидания. Так как сокет всегда открыт и слушает соединение, как только сервер проталкивает данные, они сразу же оказываются в вашем браузере — за вычетом времени на пересылку. Благодаря этому время ожидания становится исключительно низким по сравнению, например, с запросами Ajax на базе XMLHttpRequest.

Рассмотрим, например, Google Wave — уже почившее веб-приложение для обмена электронной почтой и совместной работы в режиме реального времени. К одному и тому же документу подключено много пользователей и все они что-то вводят — каждый хочет, чтобы остальные видели напечатанное им моментально, безо всяких задержек. Однако если пересыпать данные при помощи унылого Ajax, то новый объект XHR придется создавать после каждого нажатия клавиши, причем каждый запрос будет содержать все заголовки обычного XHR-запроса, такие как строка пользовательского агента, URL-адрес ссылающегося, поддерживающий тип содержимого и т. д. Куча данных для единичного нажатия клавиши. Если же используются сокеты, то, так как соединение открыто постоянно, серверу отправляется исключительно код нажатой клавиши, а сервер затем пересыпает всем подключенными клиентам тот же код клавиши и ничего кроме него.

По сравнению с Ajax, с которым в нашем примере на один запрос ушло бы приблизительно 200–300 Кбайт, с использованием сокетов объем данных уменьшается до 10–20 байт. Естественно, что при таком раскладе данные рассыпаются и ответы от подключенных клиентов приходят намного быстрее.

API ВЕБ-СОКЕТОВ ПРОСТ

С API веб-сокетов чрезвычайно просто работать. В настоящее время браузеры поддерживают только отправку строк (за исключением Firefox и фоновых вычислений). Это мы уже наблюдали в главе 10, изучая API сообщений и фоновых вычислений, метод postMessage и обработчик onmessage. Сокеты работают точно так же.

ПРИМЕЧАНИЕ

Относительно серверной части протокола ws:// — настройка серверной стороны не входит в список тем этой книги, но в Интернете уже можно найти несколько библиотек, позволяющих добавить протокол WebSocket. Используя серверы, подобные Node.js, я настраивал сервер с веб-сокетами за двадцать минут. Прочитайте мою документацию на веб-сайте <http://remysharp.com/slicehost-nodejs-websockets/>.

Это означает, что (пока что) двоичные данные отправлять невозможно. Однако в сетевом мире мы все уже привыкли работать с JSON, так что кодировка объектов, проходящих

через сокеты, не представляет проблемы, тем более что это все равно приходилось делать для запросов JSON Ajax.

Функции API ограничиваются созданием соединения, отправкой данных в сокет, получением данных и закрытием сокета. Также есть обработчик ошибок и флаг состояния с четырьмя вариантами: соединение, открыт, закрытие и закрыт. После закрытия сокета повторно открыть его невозможно. Если вам снова нужен сокет, придется создать еще один и использовать его.

Создать новый веб-сокет очень легко — это почти то же самое, что создание фонового вычисления. В URL-адресе необходимо указать протокол `ws://`, но все остальное остается как обычно:

```
var socket = new WebSocket('ws://myserver.com/tweets:  
- 8080/');
```

В этом примере я настрою прослушивание только сообщений с адреса `tweets`. Каждое из них будет всего лишь новой записью из Twitter, которые прослушиваются на моем сервере (рис. 11.1).



Рис. 11.1. Потоковое подключение для отображения записей Twitter, которые прослушиваются на моем сервере

Сообщения с сервера доставляются в форме сообщений JSON из потокового API Twitter. Следовательно, при получении мне нужно преобразовать сообщение JSON в данные и вывести запись на экран в привычном виде:

```
socket.onmessage = function(event) {
    var tweetNode = renderTweet(JSON.parse(event.data));
    document.getElementById('tweets').appendChild(tweetNode);
};
```

Итак, всего в четырех строках JavaScript (не считая функцию `renderTweet`, которая всего лишь преобразует входящие разобранные данные JSON в понятный фрагмент HTML, который можно присоединить к странице) я настроил потоковую пересылку и отображение записей из Twitter на своей странице.

СОВЕТ

URL-адрес, который вы указываете для веб-сокета, не обязательно должен совпадать с источником вашего документа. Это означает, что подключаться можно и к серверам сторонних поставщиков услуг, что значительно расширяет круг возможностей.

БОЛЬШЕ, ЧЕМ ПРОСЛУШИВАНИЕ СОКЕТА

Как я уже упоминал ранее, у сокета есть еще методы, помимо прослушивания. Так как чат — это «hello, world!» технологии Comet, мне показалось, что будет справедливо продемонстрировать пример реализации чата с помощью веб-сокетов:

```
var socket = new WebSocket("ws://my-chat-server.com:8080/"),
    me = getUsername();

socket.onmessage = function(event) {
    var data = JSON.parse(event.data);
    if (data.action == 'joined') {
        initialiseChat();
    } else {
        showNewMessage(data.who, data.text);
    }
};

socket.onclose = function () {
    socket.send(JSON.stringify({
        action: 'logoff',
        username: me
    }));
    showDisconnectMsg();
};

socket.onopen = function() {
    socket.send(JSON.stringify({
        action: 'join',
        username: me
    }));
};
```

Этот простой псевдокод показывает, как те же техники, которые мы применяли с API сообщений, помогают обойти ограничения, накладываемые пересылкой только простого текста. Серьезно, ничего сложнее в API веб-сокетов нет. Вы этого не видите, но браузер сам пересыпает данные и помещает их в буфер (хотя у вас есть возможность проверять текущее заполнение данными с помощью `bufferedAmount`). Весь процесс обмена информацией проще, чем единственная настройка объекта XHR!

События, отправляемые сервером

В некоторых ситуациях требуется, чтобы сообщения, поступающие от сервера, просто вталкивались в браузер. События, отправляемые сервером (Server-Sent Events), хорошо подходят для мониторинга цен в режиме реального времени, обновления списка статей, добавляемых на сайт, или любых других приложений, в которых информация поступает в одном направлении — от сервера к клиенту, — и делается это в реальном времени. Если же вам нужно приложение реального времени для двустороннего обмена данных, для его реализации следует воспользоваться технологией веб-сокетов, с которыми мы познакомились чуть раньше в этой главе.

Для получения событий, отправляемых сервером, используется объект `EventSource`. Принцип аналогичен тому, как вы добавляете в свои приложения веб-сокеты. Нужно создать новый объект `EventSource` и передать ему URL-адрес, с которым требуется установить соединение. Браузер немедленно займется этим.

У события `EventSource` есть несколько простых событий:

- `Open`: когда устанавливается соединение;
- `Message`: когда приходит новое сообщение — необработанные данные сообщения содержатся в свойстве `data` объекта события;
- `Error`: когда что-то идет не так.

Уникальным объект `EventSource` делает способ обработки разрыва соединения и отслеживания сообщения.

Если соединение `EventSource` по каким-то причинам обрывается, API автоматически пытается восстановить его. Если вы работаете с идентификаторами сообщений, после того как `EventSource` восстановит соединение, он сообщит серверу, какой идентификатор был получен последним. Таким образом, сервер с легкостью сможет отправить клиенту только те сообщения, которые тот недополучил из-за разрыва (если, разумеется, клиентскому приложению это требуется).

Например, вы создали приложение, которое отслеживает в Twitter все упоминания Брюсом его любимого розового плюшевого мишки и, основываясь на этой информации, в режиме реального времени строит диаграммы. Приложение сообщает о текущем настроении Брюса, так что вы в любой момент можете проверить, устраивает ли моего друга цвет, текстура и общее ощущение от держания в руках этой игрушки.

Поскольку браузер всего лишь пассивно принимает данные от сервера, для реализации приложения лучше всего подходят события, отправляемые сервером.

А теперь предположим, что, пока мы старательно отслеживали в своем приложении степень умиления Брюса, соединение прервалось. Когда соединение восстановится,

`EventSource` сообщит серверу, что последним мы видели сообщение с идентификатором 69; тем временем на сервере активно уже сообщение с идентификатором 78. Приложение на серверной стороне понимает, что мы пропустили несколько сообщений, и отправляет нам их, начиная с семидесятого. В клиентском коде это никак не отражено, и он не меняется, поскольку любое из пропущенных восьми сообщений всего лишь будет запускать событие `message`, обработка которого у нас уже предусмотрена.

Вот пример такого приложения:

```
var es = new EventSource('/bruces-pink-toy');

es.onopen = function () {
    initialiseChart();
};

es.onmessage = function (event) {
    var data = JSON.parse(event.data);
    chart.plot(data.time, data.sentiment);
};
```

СОБЫТИЯ, ОТПРАВЛЯЕМЫЕ СЕРВЕРОМ — ТЕХНОЛОГИЯ НА СЕРВЕРНОЙ СТОРОНЕ

На серверной стороне можно использовать схему на базе PHP (например, LAMP), но так как Apache (которому соответствует буква A в LAMP) плохо поддерживает постоянные соединения, очень часто будут происходить разрывы, в ответ на которые `EventSource` будет автоматически восстанавливать соединение. Фактически в результате получится нечто, более напоминающее приложение с опрашиванием Ajax.

Это не лучший сценарий, но у PHP есть большое преимущество — а именно, его очень легко выучить. Таким образом, приятно знать, что и эта технология может где-то пригодиться. Для того чтобы применить `EventSource` в реальном приложении, вам потребуется постоянное подключение к серверу, которое обычная схема LAMP обеспечить не в состоянии.

Можно и, вероятно, нужно выбрать сервер, оперирующий событиями. Я не буду углубляться в подробности, всего лишь порекомендую вам взглянуть в сторону Node.js (серверная платформа на базе JavaScript) или чего-нибудь вроде Twisted для Python.

На сервере подключение также должно быть постоянно открытым для клиента, и он должен отправлять клиенту заголовок с типом `mime text/event-stream`.

Сервер будет отправлять сообщения в таком формате:

```
id: 1\n
data: { "semiment": "like", "time": "2011-06-23 16:43:23"
- }\n\n
```

Два символа новой строки указывают на конец сообщения. Если бы мы отправляли простые предложения (а не данные JSON, как в случае с Брюсом), то для отправки нескольких строк использовали бы такой формат:

```
data: Это первая очень, очень, очень длинная строка,
- но -\n
data: это еще не все, у меня есть, что сказать
- вдогонку.\n\n
```

`data: Поскольку я следую за двумя пустыми строками,
- я новое сообщение\n\n`

В примере выше отправлено было бы только *два* сообщения. Также обратите внимание, что я не использую никакие идентификаторы. Это необязательно, однако если вы хотите, чтобы ваше приложение поддерживало возможность возобновлять вещание с того сообщения, на котором подключение прервалось, вам следует добавлять идентификационные номера.

ПРОСТОЙ СЕРВЕР EVENTSOURCE

Далее показан очень простой код Node.js, принимающий соединение с сервером на базе `EventSource` и отправляющий сообщения. И снова я не буду углубляться в детали серверной логики — рассматривайте этот код как некую отправную точку. Я также упростил решение: сервер всего лишь уведомляет подключенных пользователей о строках пользовательских агентов *других* посетителей, соединенных в настоящий момент с той же службой. Продлим эксперимент с любимым мишкой Брюса еще на денек!

```
/** Когда создается новый Event Source */
response.writeHead(200, {'Content-Type':
- 'text/event-stream', 'Cache-Control': 'no-cache'});
// получить id последнего события и присоединить номер
var lastId = req.headers['last-event-id']*1;
if (lastId) {
    for (var i = lastId; i < eventId; i++) {
        response.write('data: ' + JSON.stringify
- (history[eventId]) + '\nid: ' + eventId + '\n\n');
    }
}

// наконец, кэшировать соединение для ответов
connections.push(response);
```

```
/** Когда приходит обычный веб-запрос */
connections.forEach(function (response) {
    history[++eventId] = { agent:
- request.headers['user-agent'], time: +new Date };
    response.write('data: ' + JSON.stringify
- (history[eventId]) + '\nid: ' + eventId + '\n\n');
});
```

Код на клиентской стороне выглядит так:

```
var es = new EventSource('/eventsourcer');
es.onmessage = function (event) {
    var data = JSON.parse(event.data);
    log.innerHTML += '<li><strong>' + data.agent +
- '</strong><br> connected at <em>' +
- (new Date(data.time)) + '</em></li>';
};
```

Самое приложение очень простое. Вся тяжелая работа происходит за кулисами благодаря событиям, отправляемым сервером в режиме реального времени.

ПОДДЕРЖКА РЕАЛИЗАЦИИ

EventSource поддерживается очень даже неплохо. Chrome, Safari, Firefox и Opera отлич-но справляются с этой технологией, в то время как вопрос, будет ли IE10 поддерживать EventSource, к большому нашему сожалению остается без ответа (по крайней мере, пока что нам это неизвестно, но я продолжаю надеяться). В любом случае, поскольку EventSource в некоторых ситуациях все равно возвращается к обычному опросу, данный API очень просто имитировать с помощью JavaScript или Ajax (несколько примеров вы найдете здесь в разделе EventSource: <https://github.com/Modernizr/Modernizr/wiki/HTML5-Cross-browser-Polyfills>).

Кстати, я заметил одну вещь: если создать новый EventSource во время или сразу после загрузки страницы, в некоторых браузерах индикатор загрузки страницы не останавливается, создавая впечатление, что должно загрузиться что-то еще, хотя это не так. Не уверен, это ошибка реализации или хитрость, нацеленная на то, чтобы мы, разработчики, не теряли бдительность. Как бы то ни было, если просто дождаться завершения загрузки документа, а затем обернуть код инициализации EventSource в `setTimeout(init, 10)`, вы избежите этой неприглядной и раздражающей ошибки.

ЗАКЛЮЧЕНИЕ

Эта глава вооружила вас последними новинками из арсенала средств HTML5 для построения веб-приложений реального времени. Добавление возможностей реального времени на веб-сайты дает вам отличное конкурентное преимущество. А теперь, когда вы увидели, какой простой код JavaScript для этого требуется, уверен, вы тут же броситесь внедрять полученные знания. Очевидно, что с серверной стороной придется повозиться, но как только все окажется на своих местах, все эти пользователи с синдромом рассеянного внимания будут ваши. Так идите и напишите что-нибудь потрясающее!

Глава 12

ЗАПОЛНЕНИЕ: ЛАТАЕМ СТАРЫЕ БРАУЗЕРЫ И УЧИМ ИХ ПОНИМАТЬ HTML 5

Брюс Лоусон и Реми Шарп



Большое спасибо всем, кто дочитал до этого места. Новые структурные элементы наверняка разожгли ваш аппетит, а новые API подогрели интерес, и вы нетерпеливо ищете возможность применить полученные знания на деле. Однако на вашем пути стоят Internet Explorer 6, 7 и 8. Скоро, благосклонный читатель, мы преодолеем эти препятствия.

Знакомство с заполнителями

Занимаясь ремонтом жилища, вы обнаруживаете в стенах и деревянных поверхностях трещины и дыры, которые необходимо чем-то заполнять, чтобы поверхность сделалась ровной. Вы используете белую шпатлевку, обеспечивающую гладкую, ровную поверхность, которую затем можно окрасить, наклеить на нее обои или декорировать иным способом.

В этой главе вы познакомитесь с приемами *заполнения*, то есть научитесь «затыкать дыры» в браузерах с помощью JavaScript (или любой другой подходящей технологии, такой как Flash, если она в конкретной ситуации может дать лучший результат).

Мы узнали, что HTML5 shiv помогает стилизовать новые элементы HTML5 в старых браузерах. Но как насчет всех этих замечательных API, описанных ранее в нашей книге?

ПРИМЕЧАНИЕ

Shiv или Shim? Брюс и Реми сказали «shiv»? Да, ребята. Сценарий HTML5 shiv был создан Джоном Резигом, который позднее признался (или понял), что в действительности имел в виду «shim». Однако к тому времени название shiv ужеочно закрепилось, так что теперь мы все говорим «HTML5 shiv». Это суровый подход, но он работает.

Метод включает следующие шаги:

- создайте код страницы по правилам стандартных API и протестируйте ее в браузерах, поддерживающих все необходимые вам возможности;
- с помощью JavaScript выполните обнаружение возможностей, которые должны работать на вашем веб-сайте;
- если определенная возможность в браузере недоступна, спокойно загрузите сценарий-заполнитель, имитирующий поддержку данной возможности;
- похлопайте себя по плечу, налейте чашку чая и возьмите с полки пирожок.

Однако существует опасность, что консервативным браузерам придется загружать очень много скриптов, вследствие чего пострадает производительность. Это означает, что вам придется большое внимание уделить тестированию.

Возможно, вас удивляет такой «хакерский» подход, ведь смысл внедрения HTML5 в том и заключается, чтобы современные веб-приложения могли обходиться без трюков и обходных путей. Но в действительности так и есть — важнейшее отличие метода заполнения заключается в том, что в новых браузерах используются только современные стандарты, никаких трюков. Трюки предназначены исключительно для помощи старым браузерам; этот подход еще называют «регрессивным усовершенствованием», так как трюки предназначены для все уменьшающегося числа устаревших браузеров, а не для новых, в которых они сохранятся еще надолго.

ПРИМЕЧАНИЕ

Помните только, что заполнение не относится к прогрессивному усовершенствованию — точно так же нельзя сказать, что оно поддерживает изящное ухудшение. Заполнение в первую очередь требует наличия JavaScript.

Метод заполнения также означает чистоту кода; наша разметка и сценарии отвечают всем требованиям спецификации и лишь иногда по необходимости загружают дополнительные решения, используя сторонние сценарии заполнения. Люди, занимающиеся маркетингом, называют это ситуацией обоюдного выигрыша — даже эти товарищи иногда придумывают удачные определения.

Так чем же заполнение отличается от техник, имеющихся в нашем арсенале уже сегодня, таких как shim? Вот чем: если убрать сценарий заполнения, код все так же успешно будет работать, и никакие изменения не потребуются.

Распознавание возможностей

Распознавание поддержки определенной технологии — это первый шаг к принятию решения, требуется ли в вашем приложении заполнение. Обратите внимание, что я говорю о *распознавании поддержки*, а не об обычном «вынюхивании» браузера. Второй вариант построен на простом принципе: в данном конкретном браузере определенная возможность отсутствует, следовательно, для устранения этого недостатка нужно применить наш трюк.

Распознавание возможностей обеспечивает будущее вашего метода заполнения. Если пользователь обновит свой браузер, метод заполнения увидит, что в новой версии поддержка нужной технологии встроена по умолчанию и, следовательно, выполняться не будет. Кроме того, пользователь может работать в старом браузере, таком как IE7, но при этом установить какое-то супер-пупер расширение, благодаря которому на IE слизойдет вся благодать HTML5. Любой агент «вынюхивания» провалит этот тест, а распознавание возможностей сработает на ура, и вы будете довольны собой больше, чем Брюс, когда он приходит в футбольке с надписью «Я лучше всех».

Однако стоит помнить, что невозможно распознать *все*, и об этом мы тоже поговорим.

Существуют разные способы распознавания поддержки различных возможностей в браузерах, зачастую уникальные для каждой отдельной возможности. Важно помнить, что если вы пишете метод заполнения, то обязаны убедиться, что распознавание возможностей правильно работает во всех браузерах, которые вы планируете поддерживать.

В большинстве случаев, однако, распознавание возможностей заключается всего лишь в проверке, существует ли функция или свойство в текущей версии браузера.

Распознавание свойств

К примеру, помимо ссылочного свойства `document.body` в HTML5 также появилось свойство `document.head` — может быть, не такое впечатляющее, но все же полезное, например,

для вставки какого-то элемента сценария. Поскольку это всего лишь свойство, очень просто проверить его существование и задать значение, если оно доступно:

```
if (document.getElementsByTagName('head')[0] !==  
    document.head) {  
    document.head = document.getElementsByTagName('head')[0];  
}
```

В упрощенном варианте это выглядит так:

```
document.head || (document.head =  
    document.getElementsByTagName('head')[0]);
```

В нашем коде мы проверяем свойство `document.head`: если у него *фальшивое* значение (фактически, неопределенное), то мы явно связываем `document.head` с элементом `head`. Однако в бескрайних пустошах кроссбраузерной поддержки крайне редко встречаются действительно простые вещи. С этим кодом также связана потенциальная проблема: если в JavaScript динамически генерировать кадр `iframe`, у его элемента `document` не будет свойства `head` до тех пор, пока для него не будет выполнен данный код. Нельзя сказать, что это такая уж большая проблема — мы всего лишь выполним код повторно, но о ней не стоит забывать (спасибо Маттиасу Байненсу за код и Леа Веру за то, что обратила внимание на `iframe`).

ПРИМЕЧАНИЕ

В JavaScript фальшивым (*falsy*) называется значение, возвращающее при проверке значение `false`, хотя само оно не обязательно должно быть равно `false`. Например, при проверке (в операторе `if`) значения пустой строки число `0` и выражение `1 - 1` возвратят фальшивые значения.

Еще один распространенный метод распознавания — проверка существования определенного свойства элемента HTML. Например, для того чтобы узнать, есть ли в браузере встроенная поддержка элемента `details`, мы создаем такой элемент, а затем проверяем, существует ли свойство `open` (которое, как нам известно, входит в стандартную реализацию `details`). Таким образом, мы на лету создаем элемент и тестируем указанное свойство:

```
if (!'open' in document.createElement('details')) {  
    // встроенная поддержка отсутствует, нужен заполнитель...  
}
```

В этом коде мы спрашиваем, существует ли свойство `open` у элемента `details`. Он не обязательно должен входить в DOM, для того чтобы дать точное значение `true` или `false`.

РАСПОЗНАВАНИЕ НОВЫХ ФУНКЦИЙ

Если вы еще не слышали, сообщаю: все, что существует в JavaScript, считается объектом (и этому аспекту JavaScript посвящено множество прекрасных книг). Так как мы работаем в браузере, глобальной корневой переменной считается объект `window`, а функции и методы представляют собой свойства этого объекта `window`. Таким образом, мы можем проверять наличие новых функций, методов и конструкторов точно так же, как делали это со значениями свойств.

Для того чтобы проверить, доступен ли в браузере объект `sessionStorage`, можно выполнить такую строку кода:

```
typeof window.sessionStorage !== 'undefined'
```

Однако старые версии Firefox (в частности, 3.x) на такой строке выбрасывают исключение безопасности, если в браузере отключено использование файлов cookie (мы коснемся этой темы в главе 6). Так как я хочу, чтобы мой метод заполнения поддерживал старые версии Firefox, а не выбрасывал исключения направо и налево, я оберну проверку оператором `try/catch`:

```
var sessionStorageAvailable = (function() {
    try {
        return typeof window.sessionStorage !== 'undefined';
    } catch (e) {
        return false;
    }
})();
```

ПРИМЕЧАНИЕ

Стоит также отметить, что Марк Пилгрим, автор «очередной книги об HTML5» под названием «HTML5: Up and Running», собрал совершенно удивительный и, возможно даже, всеобъемлющий список методов для распознавания возможностей в браузерах. Заходите проверить: <http://diveintohtml5.org/everything.html>.

Как вы уже убедились, с каждым свойством и методом, которое вы будете проверять для реализации заполнения, связаны собственные хитрости. Но это часть веб-разработки, к которой мы все давно привыкли.

РАСПОЗНАВАНИЕ ЛЮБЫХ ОБЪЕКТОВ БЕЗ ПОМОЩИ JAVASCRIPT

Даже если JavaScript не входит в ваш арсенал, не теряйте надежды. Проект Modernizr (<http://modernizr.com>), поддержкой которого занимаются Фарук Атез, Пол Айриш и Алекс Секстон, — это небольшая библиотека JavaScript, дающая вам полное программное представление о том, что конкретный браузер поддерживает и что он не поддерживает.

Однако не пугайтесь названия: эта библиотека не станет *модернизировать* ваш браузер. Зато она даст вам базу для распознавания поддержки более 40 различных новых аспектов HTML5 и CSS3. Если вам интересно побольше узнать о JavaScript, «загляните под капот» Modernizr — это отличный способ понять, каким образом распознаются некоторые возможности. Это стоит сделать просто из любопытства. Вы обнаружите, что очень часто браузеры лукавят относительно поддержки той или иной возможности или же недостаточно точно следуют спецификациям.

ПРОИЗВОДИТЕЛЬНОСТЬ РАСПОЗНАВАНИЯ ВОЗМОЖНОСТЕЙ И ЗАПОЛНЕНИЯ

Когда для того чтобы справиться с отсутствием чего-то в браузере предлагается JavaScript, первым делом разработчики интересуются производительностью такого решения.

Разумеется, любой дополнительный JavaScript-сценарий, выполняющийся в браузере (даже если он включает одну строчку, проверяющую наличие какой-то возможности), определенным образом влияет на производительность. Однако в реальном мире очень большая доля проверок на поддержку возможностей принимает форму микрофрагментов кода, которые никак не могут повлиять на работу приложения. Время выполнения измеряется миллисекундами — в большинстве ситуаций этим можно пренебречь.

Кроме того, стоит помнить, что механизмы JavaScript отличаются высокой скоростью работы. А поскольку новые быстрые браузеры очень редко нуждаются в заполнителях, сложных заплатках или вспомогательных сценариях — вместо этого вам всего лишь нужно выполнить несколько тестов, — у пользователей современных браузеров почти не возникает дополнительных затрат.

Распознавание возможностей также не влечет особых издержек для пользователей старых браузеров с более медленными механизмами JavaScript, но им иногда приходится немного подождать, пока загружается метод заполнения. К сожалению, этого не избежать: мы должны как-то перетащить эти старые браузеры в прекрасный современный мир, несмотря на все их капризы и нежелание меняться.

Также важно помнить о том, что не следует заполнять все подряд, включая кухонную раковину (простите, не смог избежать этой не самой остроумной шутки). Добавление JavaScript-сценария, делающего то, на что не способен определенный браузер, всегда «отъедает» немного производительности. Если вы добавите слишком много лишних функций, которые даже не будут использоваться, очевидно, что вы попусту растратите ресурсы. Например, вы создаете очередную версию приключенческой игры «Супер Брюс и Супер Реми» на базе элемента `canvas` и обнаруживаете, что заполнение замедляет IE6 почти до полной остановки, при этом пользователи IE6 составляют вашу основную аудиторию. Решение очень простое: не используйте заполнение! Будучи ответственными разработчиками, вы должны принимать обдуманные решения относительно того, подходит та или иная технология для решения конкретной задачи или нет.

ЧТО РАСПОЗНАТЬ НЕВОЗМОЖНО

Как мы уже упоминали раньше, существуют определенные нюансы написания методов заполнения для различных технологий. Но также нельзя забывать о черных дырах — технологиях, не поддающихся распознаванию.

Простой пример «нераспознаваемой» технологии — объект, возвращающий при проверке фальшивое положительное значение.

Рассмотрим, например, атрибут `contenteditable` из главы 2. При тестировании данного атрибута с помощью методов, которые мы обсуждали выше, создается впечатление, что `contenteditable` поддерживается всегда. Однако в действительности на устройствах с браузером Safari Mobile (до версии iOS 5) фокус клавиатуры не переводится на редактируемую область, то есть данная возможность не поддерживается, что бы ни говорил нам тест на распознавание.

Трудно сказать, какие именно возможности поддаются и не поддаются распознаванию. Тестируя свой метод заполнения во всех браузерах, которые должен поддерживать ваш

сайт, вы будете узнавать, действительно ли та или иная функциональность может быть надежно и достоверно распознана.

Для того чтобы получить первое представление о нераспознаваемых возможностях и условиях, в которых проверка возвращает ложные результаты, загляните на страницу проекта Modernizr, посвященную данному вопросу (<http://github.com/Modernizr/Modernizr/wiki/Undetectables>).

Но что делать, если возможность не поддается распознаванию? Досадно сознавать это, но одним из решений может быть «вынюхивание» браузера. Это ненадежная техника, так как браузеры не всегда сообщают правду о своей версии и даже имени — Microsoft Internet Explorer все еще притворяется браузером Mozilla! Прибегая к помощи техник «вынюхивания», помните, что в будущем появятся новые версии браузеров, которые ваш код может распознавать неправильно. Это очень зыбкая почва, вступать на которую следует с осторожностью.

ГДЕ НАЙТИ ЗАПОЛНИТЕЛИ

Как и большинство других библиотек JavaScript-сценариев, заполнители разбросаны по сети тут и там. Но Пол Айриш, наш супергерой, примчавшийся из самых дальних, темных уголков сети, создал wiki-страницу, где он теперь собирает все, что ему удается найти (<https://github.com/Modernizr/Modernizr/wiki/HTML5-Cross-Browser-Polyfills>). Так как это страница wiki-энциклопедии, со временем она значительно разрослась: свой вклад внесли множество участников сообщества веб-разработчиков, включая ваших покорных слуг. Теперь здесь можно найти всевозможные решения заполнения: для canvas, SVG, веб-форм, веб-сокетов, EventSource, элемента details, атрибутов data-* и много-много другого.

Кроме того, этот ресурс предлагает не только заполнители, но и сценарии shim и другие полезные библиотеки.

ЗАПОЛНИТЕЛИ CSS

Эта концепция распространяется также и на CSS3: CSS3Pie (<http://css3pie.com>) и IE-CSS3 (<http://fetchak.com/ie-css3/>) реализуют отображение нескольких наиболее полезных декоративных возможностей CSS3 в IE версий с 6 по 8. selectivizr (<http://selectivizr.com>) — это утилита JavaScript, эмулирующая псевдоклассы и селекторы атрибутов CSS3 в IE версий с 6 по 8. Еще один заполнитель — eCSStender (<http://ecsstender.org>). Разумеется, техники CSS не настолько важны для функционирования сайта, как API, — если ваш сайт не может работать без CSS, значит, вы что-то делаете неправильно. И пожалуйста, запомните: CSS3 никак не связан с HTML5. Считайте все приведенное выше дополнительной справочной информацией: мы просто очень вас любим.

РАБОЧИЙ ПРИМЕР С ИСПОЛЬЗОВАНИЕМ MODERNIZR

Итак, вы до зубов вооружены техниками распознавания возможностей и написали замечательный сценарий заполнения, который сгладит дыры и шероховатости в браузерах. Но как заставить все это работать?

Мы покажем вам, как с помощью Modernizr и супер-пупер полезной утилиты под названием *уэрпоре* (авторства Алекса Секстона и Ральфа Холzmanна, <http://уэрпореjs.com>) сначала понять, поддерживает ли браузер нужную вам технологию, а затем (в случае отрицательного ответа) подгрузить сценарий заполнения и продолжить загрузку оставшейся части приложения.

Поскольку милые ребята из Modernizr очень заботятся о вас, *уэрпоре* интегрируется в Modernizr с помощью метода `Modernizr.load`, поэтому вам не приходится беспокоиться о связке этих двух инструментов.

Предположим, что ваше приложение будет использовать `sessionStorage`. Вы нашли написанный Реми заполнитель для `sessionStorage`, но он требует поддержки JSON, поэтому, прежде чем запускать приложение, вам потребуется протестировать в браузере поддержку *и* `sessionStorage`, *и* JSON.

```
Modernizr.load({
    // сначала проверки
    test: function () {
        return !window.JSON;
    },
    nope: 'json2.js'
}, {
    // еще проверки
    test: Modernizr.sessionStorage,
    nope: 'sessionStorage.js'
},
// теперь, когда все готово, можно добавить app.js
'app.js'
});
```

Обязательно используйте версию Modernizr, которую вы собрали самостоятельно. Это очень просто: зайдите на веб-сайт Modernizr, щелкните на розовой кнопке *Production* и выберите нужные компоненты. Установите флагшки напротив тех возможностей, поддержку которых необходимо тестировать в браузерах. Преимущество такого подхода в том, что посетителям вашей страницы приходится загружать меньше кода.

Вы добавляете JavaScript-файл Modernizr, а затем код, приведенный выше, который всего лишь приказывает Modernizr выполнить два теста, прежде чем добавлять файл `app.js` (содержащий код вашего приложения). Хотя мы вызываем Modernizr, все проверки и условные загрузки осуществляются в волшебной библиотеке *уэрпоре*. Вам не приходится беспокоиться об этом; сфокусируйтесь на том, чтобы перечислить все необходимые тесты поддержки возможностей — они должны быть выполнены, прежде чем будет запущен ваш код.

Первый тест проверяет, доступна ли встроенная поддержка JSON. Это не часть спецификации HTML5, но почему бы и не упомянуть о нем — мы же посвятили целую главу геолокации, которая тоже не входит в HTML5! Смысл в том, что заполнение используется для любой технологии, если перед вами стоит задача компенсировать ее отсутствие. Если тест завершится ошибкой, например в IE7, где нет встроенной поддержки JSON, то наш код подгрузит файл `json2.js`.

Затем выполняется проверка поддержки `sessionStorage`. Так как эта версия Modernizr создавалась специально для теста `sessionStorage`, в зависимости от наличия поддержки

вернется значение `true` или `false`. В случае `false` загружается заполнитель `sessionStorage.js` (обратите внимание, что в приведенном выше примере файл `sessionStorage.js` должен находиться в локальном домене, чтобы его можно было найти по указанному относительному пути).

Наконец, после того как все проверки завершены и нужные заполнители загружены, загружается файл `app.js`, и ваше приложение запускается правильно, зная, что даже в дряхлом и ограниченном IE6 оно все так же может использовать новейшие и крутейшие технологии, такие как `sessionStorage` и `JSON`.

Резюме

Мы узнали, что заставить приложения HTML5 работать в старых браузерах не только можно — сделать это достаточно легко. Для этого нужно выполнить распознавание возможностей, то есть понять, какую необходимую вам функциональность браузер поддерживает, а какую нет. Благодаря особенностям конструкции языка большинство (хотя, к сожалению, все же не все) возможностей HTML5 поддаются программному распознаванию. После этого необходимо подгрузить сценарий, решающий проблему отсутствия нужных возможностей. Однако важно помнить, что заполнители вряд ли сравнятся по производительности со встроенными аналогами и что устаревшие браузеры, такие как IE6, работают медленнее современных, в том числе из-за непроизводительных механизмов JavaScript. В подобных случаях, возможно, стоит вообще отказаться от заполнения и предоставить совершенно иное альтернативное решение.

И напоследок

Надеемся, вас вдохновило это краткое знакомство с новыми структурами и API HTML5, и вы готовы начать их использовать.

В спецификации осталось еще много великолепных вещей, о которых мы не поговорили, потому что они пока что не реализованы. Например, браузер можно зарегистрировать как обработчик содержимого (`registerContentHandler`), и тогда при щелчке на документе, или фотографии, или видео с рабочего стола будет сразу же открываться окно браузера с веб-приложением, предназначенным для обработки документа именно такого типа. И само приложение, и все его панели инструментов (использующие `<menu>` и `<command>`) будут написаны на HTML5. Но такие возможности пока что только ожидают реализации.

Забудьте маркетинговую чушь о «Web 2.0». Мы стоим на пороге *веб-разработки 2.0*: мощные языки, такие как HTML5, SVG и CSS3, поставят мир разработки для веб с ног на голову. Уже сегодня браузеры поддерживают немало аспектов этих языков (и можете не сомневаться — с каждым днем добавляются все новые и новые функции).

Однако чрезвычайно важно помнить, что мы имеем дело с *веб-разработкой*. Основы Сети — это URL-адреса и гиперссылки, обеспечивающие метод доставки содержимого. Если ваше поразительное демо-приложение представляет собой, фактически, бесчисленные `div`-блоки, перемещаемые по экрану JavaScript-сценарием; если его содержимое — это

растровое изображение текста, которое выводится на экран с помощью `<canvas>`; если для него требуется наличие мыши или сенсорного экрана; если в приложении нет ссылок, нет контента и нет URL-адресов, с которыми можно связать закладки или ссылки; или если вы генерируете код разметки из нечитаемого и скрытого ото всех JavaScript-кода, спросите себя: это разработка для Сети или переосмысление основ DHTML и Flash, которые по какой-то случайности выполняются в браузере вместо плагина?

Поиграйте с новыми возможностями. Поэкспериментируйте с новыми структурами разметки, с редактированием видео налету, с построением смешных и интересных игр и приложениями на базе холста. Прочитав эту книгу, вы доказали, что готовы учиться, принимать и воплощать все самое новое. Однако, пожалуйста, подайте своим коллегам хороший пример: уважайте тех пользователей ваших суперсовременных приложений, которые работают со старыми браузерами или вынуждены прибегать к вспомогательным технологиям.

Все примеры из книги (и многое другое) можно найти на веб-сайте <http://www.introducinghtml5.com>. Спасибо за то, что купили эту книгу. Еще увидимся!

Приятного путешествия! Наслаждайтесь разработкой потрясающих программ. Удачи!

Брюс Лоусон и Реми Шарп

Бирмингем и Брайтон

Брюс Лоусон, Реми Шарп

Изучаем HTML5. Библиотека специалиста

2-е издание

Перевела с английского Е. Шикарева

Заведующий редакцией

А. Кривцов

Руководитель проекта

А. Юрченко

Ведущий редактор

Ю. Сергиенко

Литературный редактор

О. Некруткина

Художественный редактор

Л. Адуевская

Корректор

В. Листова

Верстка

Е. Егорова

ООО «Мир книг», 198206, Санкт-Петербург, Петергофское шоссе, 73, лит. А29.
Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 95 3005 — литература учебная.
Подписано в печать 31.01.12. Формат 70x100/16. Усл. п. л. 24,510. Тираж 2000. Заказ
Отпечатано в соответствии с предоставленными материалами в ЗАО «ИПК Парето-Принт».
Тверь, www.pareto-print.ru.

ВАМ НРАВЯТСЯ НАШИ КНИГИ? ЗАРАБАТЫВАЙТЕ ВМЕСТЕ С НАМИ!

У Вас есть свой сайт?

Вы ведете блог?

Регулярно общаетесь на форумах? Интересуетесь литературой, любите рекомендовать хорошие книги и хотели бы стать нашим партнером?

ЭТО ВПОЛНЕ РЕАЛЬНО!

СТАНЬТЕ УЧАСТНИКОМ ПАРТНЕРСКОЙ ПРОГРАММЫ ИЗДАТЕЛЬСТВА «ПИТЕР»!



Зарегистрируйтесь на нашем сайте в качестве партнера по адресу www.piter.com/ePartners



Получите свой персональный уникальный номер партнера



Выбирайте книги на сайте www.piter.com, размещайте информацию о них на своем сайте, в блоге или на форуме и добавляйте в текст ссылки на эти книги (на сайт www.piter.com)

ВНИМАНИЕ! В каждую ссылку необходимо добавить свой персональный уникальный номер партнера.

С этого момента получайте 10% от стоимости каждой покупки, которую совершил клиент, придя в интернет-магазин «Питер» по ссылке с Вашим партнерским номером. А если покупатель приобрел не только эту книгу, но и другие издания, Вы получаете дополнительно по 5% от стоимости каждой книги.

Деньги с виртуального счета Вы можете потратить на покупку книг в интернет-магазине издательства «Питер», а также, если сумма будет больше 500 рублей, перевести их на кошелек в системе Яндекс.Деньги или Web.Money.

Пример партнерской ссылки:

<http://www.piter.com/book.phtml?978538800282> – обычная ссылка

<http://www.piter.com/book.phtml?978538800282&refer=0000> – партнерская ссылка, где 0000 – это ваш уникальный партнерский номер

**Подробно о Партнерской программе
ИД «Питер» читайте на сайте
WWW.PITER.COM**

издательский дом
ПИТЕР[®]
WWW.PITER.COM



**ПРЕДСТАВИТЕЛЬСТВА ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР»
предлагают профессиональную и популярную литературу по различным
направлениям: история и публицистика, экономика и финансы, менеджмент
и маркетинг, компьютерные технологии, медицина и психология.**

РОССИЯ

Санкт-Петербург: м. «Выборгская», Б. Сампсониевский пр., д. 29а
тел./факс: (812) 703-73-73, 703-73-72; e-mail: sales@piter.com

Москва: м. «Электроразводная», Семеновская наб., д. 2/1, стр. 1
тел./факс: (495) 234-38-15; e-mail: sales@msk.piter.com

Воронеж: тел.: 8 951 861-72-70; e-mail: hitsenko@piter.com

Екатеринбург: ул. Бебеля, д. 11а
тел./факс: (343) 378-98-41, 378-98-42; e-mail: office@ekat.piter.com

Нижний Новгород: тел.: 8 960 187-85-50; e-mail: yashny@yandex.ru

Новосибирск: Комбинатский пер., д. 3
тел./факс: (383) 279-73-92; e-mail: sib@nsk.piter.com

Ростов-на-Дону: ул. Ульяновская, д. 26
тел./факс: (863) 269-91-22, 269-91-30; e-mail: piter-ug@rostov.piter.com

Самара: ул. Молодогвардейская, д. 33а, офис 223
тел./факс: (846) 277-89-79, 229-68-09; e-mail: pitvolga@smrlc.ru, pitvolga@mail.ru

УКРАИНА

Киев: Московский пр., д. 6, корп. 1, офис 33
тел./факс: (044) 490-35-69, 490-35-68; e-mail: office@kiev.piter.com

Харьков: ул. Сузdalские ряды, д. 12, офис 10
тел./факс: (057) 7584145, +38 067 545-55-64; e-mail: piter@kharkov.piter.com

БЕЛАРУСЬ

Минск: ул. Розы Люксембург, д. 163
тел./факс: (517) 208-80-01, 208-81-25; e-mail: gv@minsk.piter.com

 Издательский дом «Питер» приглашает к сотрудничеству зарубежных торговых партнеров или посредников, имеющих выход на зарубежный рынок
тел./факс: (812) 703-73-73; e-mail: rodionova.tatyana@piter.com

 Издательский дом «Питер» приглашает к сотрудничеству авторов
тел./факс издательства: (812) 703-73-72, (495) 974-34-50

 Заказ книг для вузов и библиотек
тел./факс: (812) 703-73-73, доб. 6250; e-mail: fokina@piter.com

 Заказ книг по почте: на сайте www.piter.com; по тел.: (812) 703-73-74, доб. 6225
