Б. Лоусон, Р. Шарп

ИЗУЧАЕМ HTML5

Библиотека специалиста

ПИТЕР®

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск
2011

ББК 32.988.02-018
УДК 004.438.5
Л81

Лоусон Б., Шарп Р.
Л81 Изучаем HTML5. Библиотека специалиста. — СПб.: Питер, 2011. — 272 с.: ил. — (Серия «Библиотека специалиста»).

ISBN 978-5-459-00269-0

Эта книга посвящена изучению нового стандарта веб-программирования HTML5 и использованию новых функциональных возможностей, предоставляемых веб-разработчикам. Написанная опытными программистами, давно использующими HTML5 в своей работе, 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 семантики и структуры HTML5, как использовать мультимедийные элементы HTML5 для размещения аудио- и видеоданных, как разрабатывать «умные» веб-формы и усовершенствовать приложения, использующие географическую привязку, с помощью API геолокации.

Книга адресована веб-разработчикам, знакомым с HTML или XHTML и желающим быстро освоить новые возможности языка HTML5.

ББК 32.988.02-018
УДК 004.438.5

ISBN 978-0321687296 (англ.)
ISBN 978-5-459-00269-0

Содержание

Введение

Добро пожаловать на шоу Реми и Брюса. 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 участвует здесь как автор, использующий язык.

Для кого эта книга?

Притом что никаких знаний HTML5 не требуется, мы все же предполагаем, что вы — достаточно опытный разработчик (X)HTML, знакомый с идеями семантической разметки. Не важно, 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, душещипательные, 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 900 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 6 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, потому что это поможет вам понять, почему некоторые свойства HTML5 выглядят именно так, и, я надеюсь, предвос-

хитить некоторые вопросы: «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, после чего была выпущена спецификация под названием XHTML, 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 нет, 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» — www.whatwg.org). Об этом сотрудничестве свидетельствуют сведения об авторских правах, указанные в версии спецификации WHATWG: «© Copyright 2004–2009 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 перейти на XML — включая кавычки вокруг атрибутов, закрытие пустых тегов и пространства имен — неожиданно провалилась», — так сказал Тим Бернерс-Ли (http://dig.csail.mit.edu/breadcrumbs/node/166).

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 также Сэм Руби из IBM и Крис Уилсон из Microsoft, а под конец к ним присоединились Руби, Пол Коттон из Microsoft и Мацей Сташовьяк 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 подписаться 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 CTO Брендан Айк, разработчик архитектуры Safari и WebKit Дэвид Хайятт и изобретатель CSS и Opera CTO Хокон Виум Ли).

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» (http://www.webstandards.org/2009/05/13/interview-with-ian-hickson-editor-of-the-html-5-specification/). Многим участникам это показалось

в высшей степени неприятным: они жаловались, что производители браузеров захватили «нашу Сеть», и в чем-то были правы.

ПРИМЕЧАНИЕ

Поскольку спецификацию HTML5 разрабатывают одновременно W3C и WHATWG, существует несколько версий.

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

Версию WHATWG можно найти на http://whatwg.org/html5, но будьте осторожны: она называется «HTML5 (including next generation additions still in development)» (комментарий в скобках означает «включая дополнения следующего поколения, находящиеся в стадии разработки») и в ней есть чрезвычайно экспериментальные идеи вроде элемента `<device>`. Не думайте, что если о нем говорится в этом документе, то он хоть где-то реализован или детально продуман. Но в этой спецификации есть полезные примечания по поводу реализации свойств в различных браузерах.

Существует одностраничная 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 очень объемная и содержит много скриптов.

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, откуда они приходили или даже где они ранее обсуждались. Хорошие идеи поступали из Твиттера, блогов, IRC.

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, и 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 `<b><i>Hello mum!</b></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 900 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.

ПРИМЕЧАНИЕ

Существует спецификация HTML5, охватывающая только те аспекты, которые имеют отношение к веб-разработчикам. Она создается автоматически из основного источника и доступна на http://dev.w3.org/html5/markup/.

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 4. 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 (свободного с точки зрения 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 расширенный вариант HTML 4, и в нем все еще определены правила работы браузеров с устаревшей разметкой (например, с `<font>`, `<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 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 в случае, если для всех документов будет использоваться парсер HTML5. (Сейчас парсер HTML5 изначально не встраивается ни в один браузер, хотя на момент написания этой книги в Firefox есть *экспериментальная* версия парсера HTML5, который можно включить в `about:config`, изменив значение параметра `html5.enable` на `true`.)

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 (как это ни удивительно) 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

Сейчас HTML5 развивается очень быстро, и даже хотя первая рабочая версия спецификации была готова только в октябре 2009 года, браузеры начали внедрять поддержку HTML5 (и особенно API) раньше этой даты. Поддержка HTML5 будет постепенно улучшаться по мере того, как браузеры будут приниматься за свойства.

Эта книга была написана в период с ноября 2009 по май 2010. И мы уже несколько раз исправляли ее главы, пытаясь отразить изменения в спецификации, которая, кстати, сейчас выглядит достаточно стабильной (эх, стоит ли нам так говорить?). (Мы еще пожалеем о своих словах, это уж точно!)

Безусловно, во многих случаях, где мы говорим что-то вроде «оно поддерживается только в браузере X», такие утверждения быстро станут неактуальными — и это хорошо.

Давайте перейдем к делу

Итак, вот и закончился ваш 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!

Глава 1

Основная структура

Брюс Лоусон

Хотя большинство возможностей HTML5 направлено на создание интерактивных приложений, многие из них будут интересны как тем, кто занимается версткой, так и тем, кто отвечает за написание скриптов. В HTML5 добавлены 28 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 и начнем с разметки самой обычной страницы с помощью HTML5. Мы попытаемся превратить текущую разметку, использующую элементы `<div>`, в семантически организованную систему с помощью новых структурных элементов HTML5: `<nav>`, `<header>`, `<footer>`, `<aside>`, `<article>` и др. Далее мы посмотрим, как эти элементы работают, и поговорим об однозначности структуры HTML5-документа и его «семантичности».

Но в первую очередь самое важное. Сначала откроем редактор кода и зададим тип документа и предварительные сведения.

<HEAD>

Итак, тип документа:

```
<!doctype html>
```

Готово. И никаких URL, которые даже самым опытным веб-разработчикам приходится вырезать и вставлять. Обратите внимание на то, что здесь не указан номер версии. 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.

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, вы заметите несколько странностей. 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, 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" />
<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.**

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 не важно, 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

существующее 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).

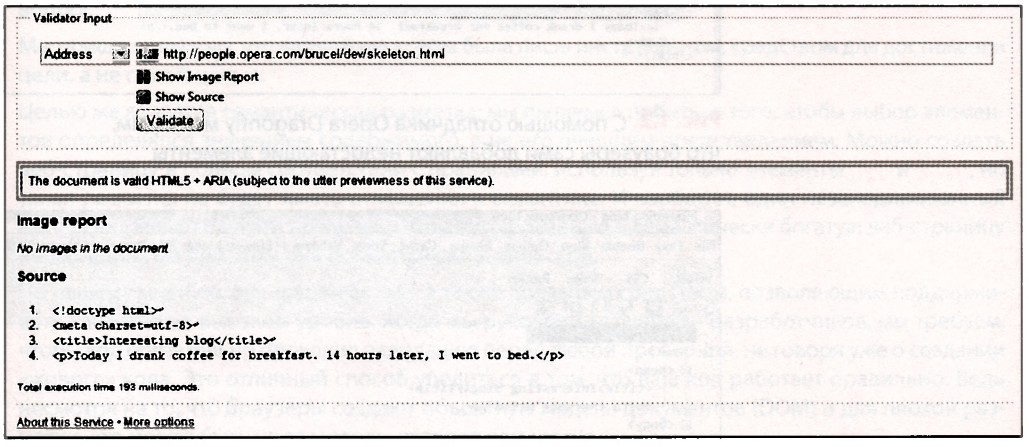


Рис. 1.1. Потрясающий факт: документ без тегов `<head>`, `<body>` и `<html>` успешно проходит валидацию.

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 IE8 Developer Tools (рис. 1.3).

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, пропуск тега `<html>` вызывает неудобства для пользователей программ экранного доступа, поскольку в нем задается основной язык документа:

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

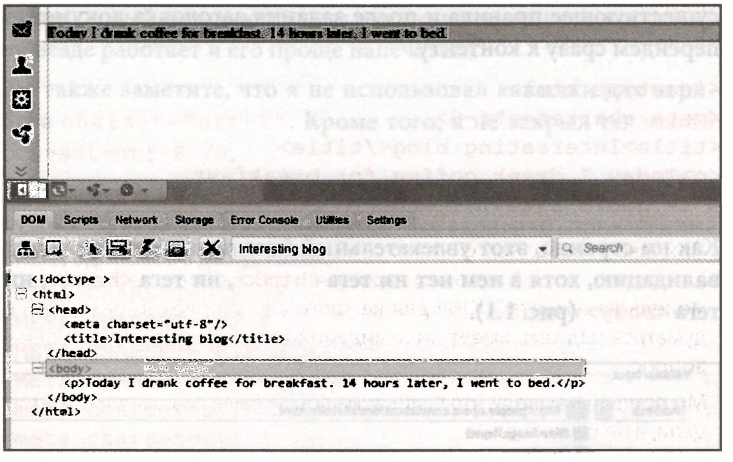


Рис. 1.2. С помощью отладчика Opera Dragonfly мы видим, что браузеры сами добавляют недостающие элементы

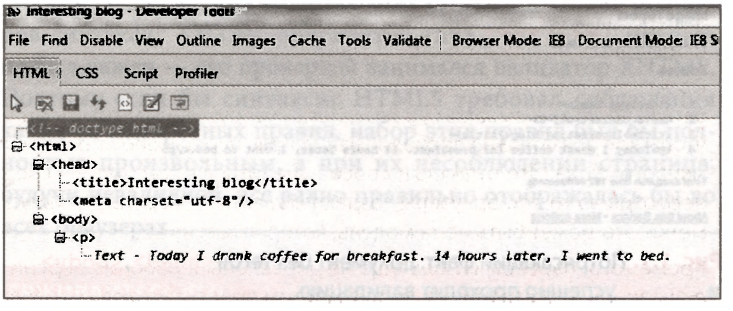


Рис. 1.3. Internet Explorer 8, как и все другие браузеры, добавляет недостающие элементы в DOM (однако IE, похоже, меняет местами элементы `<title>` и `<meta>`)

Это оказывается важным, так как, например, слово *six* произносится по-разному в английском и французском языках.

Кроме того, как мы позднее увидим, перед применением CSS для оформления новых элементов HTML5 браузеру IE требуется элемент `<body>`.

Поэтому 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>`), и, что гораздо важнее, поскольку HTML5 создает согласованную объектную модель документа (DOM) для каждого случая плохой разметки, вам будет простительно 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, 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 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.

По нашему мнению, валидация остается также полезным средством, позволяющим поддерживать качество на высоком уровне. 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 в рабочие версии браузеров, поэтому забота о правильности страниц является залогом предсказуемого поведения CSS и JavaScript.

Мы используем валидатор 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.

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 (см. http://devfiles.myopera.com/articles/572/idlist-url.htm) 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

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

Популярность	Значение	Частота
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

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

Точно так же, как в HTML 4 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 28 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 4 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 рисунке 1.4.

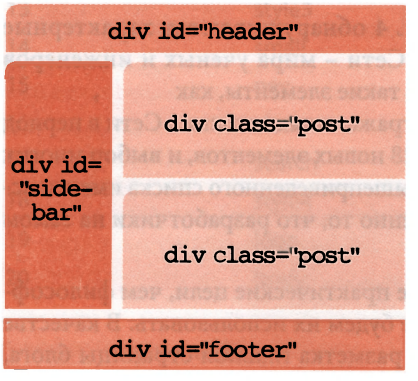


Рис. 1.4. Структура нашего блога в HTML 4

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 исполь-

зовать для выделения отдельных фрагментов страницы, — это семантически нейтральный `<div>` (определенный в HTML 4 как «универсальный механизм для оформления структуры документов»).

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 переходить? 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 MAMA наводят на мысль о том, что `menu`, `nav`, `sidebar` и `navigation` можно было бы считать синонимами.

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.

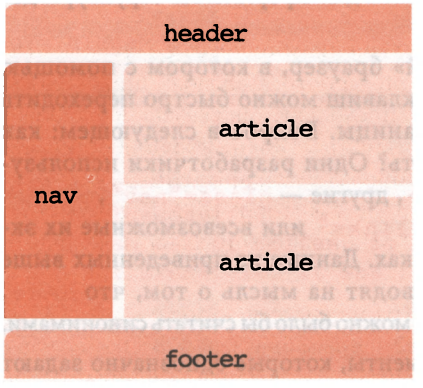


Рис. 1.5. HTML5-структура нашего блога

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 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. Но ведь в HTML5 нет элемента `<content>`, к которому должен осуществляться переход! Как в таком случае программа экранного доступа (или поисковая система) 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 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>`. Если вы хотите перестраховаться, используйте WAI-ARIA и добавьте `role=main` к тегу, обрамляющему основной контент (более подробно о WAI-ARIA рассказывается в главе 2).

Тот факт, что для второстепенных областей страницы используются отдельные элементы, а для основной — нет (поскольку она вычисляется путем исключения этих областей), кажется парадоксальным. Однако такой вариант все же является лучшим решением, так как добавление новых элементов в язык создает серьезные трудности для производителей браузеров, усложняя процесс тестирования и повышая риск регрессии.

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.

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

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>`.

Необычным может показаться то, что в CSS все элементы по умолчанию являются строковыми (`display:inline`), поэтому если просто задать ширину и высоту для структурного элемента (как мы это обычно делаем для элементов `<div>`), в современных браузерах он будет отображаться неправильно, и нам придется явно указать, что элемент является блочным (`display:block`). В нынешних браузерах до сих пор используется встроенная таблица стилей, где определены элементы, являющиеся, *по нашему мнению*, исконно блочными (пример такой таблицы стилей можно найти на http://www.w3.org/TR/CSS2/sample.html). Однако в них еще нет таких правил для элементов `<header>`, `<footer>`, `<nav>` и `<article>`, поэтому для них способ отображения должен быть задан вручную в таблице стилей. Когда в браузеры будут добавлены соответствующие правила, эти строки кода не только будут безвредными, но и 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 4, 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

В текущих версиях Internet Explorer наша страница окажется неоформленной. (На самом деле ситуация еще хуже: все новые элементы будут неоформленными, но к тем элементам HTML 4, которые IE поддерживает, таблицы стилей будут применены.)

Обмануть IE 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.

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 выполняться только в IE.

Когда вы используете новый HTML5-элемент, не забывайте добавлять в тег `<head>` правило `document.createElement` этого элемента. Альтернативный вариант — использование

небольшого скрипта Реми (http://remysharp.com/2009/01/07/html5-enabling-script/), который делает все то же самое в один прием.

ВНИМАНИЕ

Чтобы этот метод работал, у пользователей всех версий Internet Explorer (текущей версии IE8 и ее предшественников) должен быть включен JavaScript. IE9 работает так, как остальные браузеры, и не требует JavaScript.

Внимания заслуживает также тот факт, что наше JavaScript-шаманство не распространяется на модули печати IE, и поэтому то оформление, которое вы видите на экране, не передается на принтер. Решить эту проблему можно с помощью IE Print Protector (http://www.iecss.com/print-protector/) — специального JavaScript-кода, позволяющего печатать HTML5-страницы в IE. (Он включен в HTML5 Shiv Реми.) 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. 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).

ОФОРМЛЕНИЕ HTML5-ДОКУМЕНТОВ В СТАРЫХ ВЕРСИЯХ FIREFOX И CAMINO

Firefox и Camino используют движок Gecko, у которого есть неприятная особенность: HTML5-документ не отображается, если только это не правильно написанный XHTML-документ, который обрабатывается как XML. В Firefox 3+ и Camino 2 используется более современная версия Gecko, где этот дефект исправлен, и поскольку пользователи этих браузеров часто устанавливают обновления, это гораздо менее серьезная проблема, чем поведение IE.

ПРИМЕЧАНИЕ

Более подробная информация содержится в статье Реми на HTML5doctor «Как добиться того, чтобы HTML5 работал в IE и Firefox 2» (http://html5doctor.com/how-to-get-html5-working-in-ie-and-firefox-2/).

ДРУГИЕ ПРОБЛЕМЫ УСТАРЕВШИХ БРАУЗЕРОВ

При создании стилей для HTML5-документов возникают другие проблемы, связанные с устаревшими браузерами. В определенных случаях фон элемента приходится задавать явно в дочерних элементах, хотя по умолчанию это свойство должно наследоваться. В некоторых браузерах на движке Gecko могут возникнуть проблемы с оформлением блочных якорей (то есть элементов `<a>` вокруг блочных элементов, что стало возможным в HTML5 — см. главу 2 «Текст»), а во всех текущих версиях браузеров расположение (`display:block`) таких якорей нужно задавать явно — см. http://mattwilcox.net/sandbox/html5-block-anchor/test.html.

Мы не собираемся приводить здесь исчерпывающий список таких особенностей; это временные проблемы, которые, как мы надеемся, будут решены с появлением новых версий. Но если вы сами обнаружите такие ошибки, вам, возможно, захочется о них сообщить; Джон Резиг написал замечательную статью о том, как это сделать: «Ответственность веб-разработчика» (http://ejohn.org/blog/a-web-developers-responsibility/).

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 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.

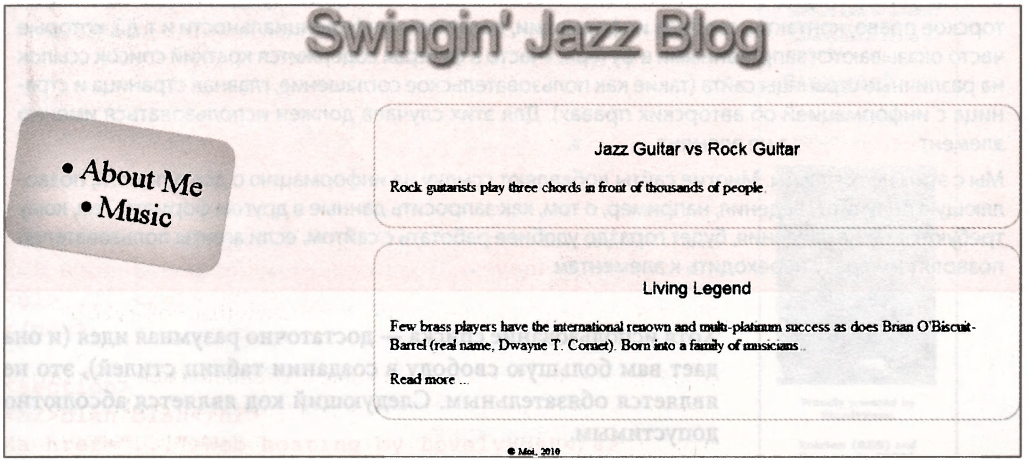


Рис. 1.7. Типичная страница с панелью навигации по сайту вне элемента `<header>`

<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 «Укрощение списков» (http://www.alistapart.com/articles/taminglists/)) или упорядоченный список (в случае «хлебных крошек»). Помните, что `<nav>` не заменяет элементы `<ol>` и `<ul>`, а обрамляет их. Поэтому устаревшие браузеры, не знакомые с этим элементом, отобразят панель навигации просто как обычный список.

<NAV>: ИСПОЛЬЗОВАТЬ ИЛИ НЕТ?

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 должен использоваться именно элемент `<footer>`, а не элемент `<nav>`».

Мы с этим не согласны. 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 является абсолютно допустимым.

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

В панели навигации можно использовать заголовки.

```
<nav>
<h2>Main navigation</h2>
<ul>
<li><a href="/about">About me</a></li>
<li><a href="/news">News</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, есть несколько областей навигации (они не изображены на рисунке): страницы, категории, архивы, новые комментарии. Для этого используется несколько элементов `<nav>` **(один такой элемент и список с подсписком — тоже возможный вариант, но я предпочитаю использовать отдельный элемент** `<nav>` **для каждого типа навигации).**

Объединение элементов `<nav>` **в новый элемент** `<aside>` **«может использоваться для создания типографских эффектов (на-**

пример, 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/sections.html#the-aside-element).

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

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

<section>
<h2>blah blah</h2>
<a href="...">Web hosting by LovelyHost</a>
<img src="...">
<p>Powered by <a href="...">WordPress</a></p>
<p><a href="...">Entries (RSS)</a>
¬ and <a href="...">Comments (RSS)</a></p>
</section>
</aside>
```

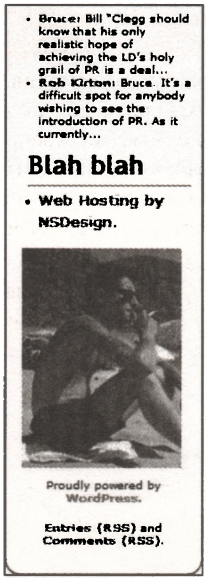


Рис. 1.8. Боковая панель моего блога, в которой помимо навигации содержатся сведения об издательствах и фотографии привлекательных мужчин

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). Другой способ разметки — использование одного общего элемента `<nav>` вместо нескольких. Как сказал бы вам любой таксидермист, есть не один способ снять шкуру кошки.

<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`; «структурированные корневые элементы» — это `blockquote`, `body`, `details`, `fieldset`, `td`.)

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 футере тоже есть сведения об авторских правах, которые мы заключили в тег `<small>`. В HTML5 этот элемент был переопределен; раньше он относился к оформлению, а в HTML5 он означает использование мелкого шрифта: «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 футере 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).

Вы можете даже создать очень популярный в настоящее время «большой футер», как на сайте http://www.thecssdiv.co.uk/ (рис. 1.9).

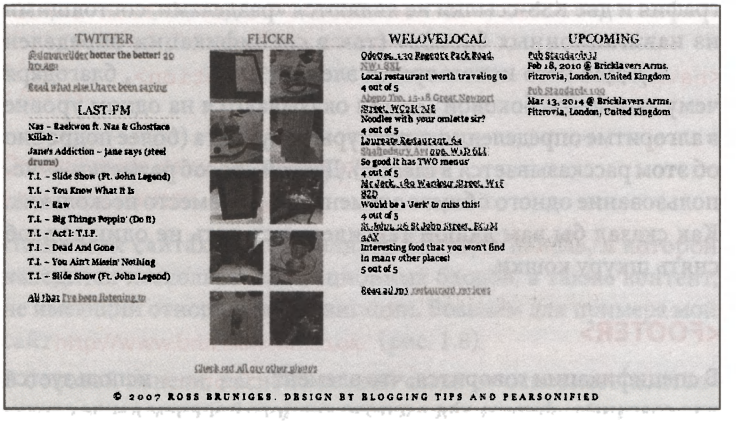


Рис. 1.9. «Большой футер» сайта http://www.thecssdiv.co.uk/

Логично помещать ссылки на другие страницы в элемент `<aside>` внутри элемента `<footer>` — но сначала спросите себя, является ли `<aside>` частью футера или же его лучше поместить на одном уровне с футером. Ведь ссылки на другие страницы скорее косвенно относятся ко всей странице, чем к содержимому футера.

Здесь я использовал бы такую разметку.

```
<body>
    <div id=mainContent>

    </div>
<aside>
    <nav>
        <h2>Twitter </h2>
        <ul>     </ul>
    </nav>
    <nav>
        <h2>Flickr </h2>
        <ul>     </ul>
    </nav>
    <nav>
        <h2>WeLoveLocal</h2>
        <ul>     </ul>
    </nav>
    <nav>
        <h2>Upcoming </h2>
        <ul>     </ul>
    </nav>
</aside>

<footer>
    <small>&copy;2007 Ross Bruniges. Design by
    ¬ Blogging Tips and Pearsonified</small>
</footer>

</body>
```

На рис. 1.10 показано, как можно обойтись без «большого футера».

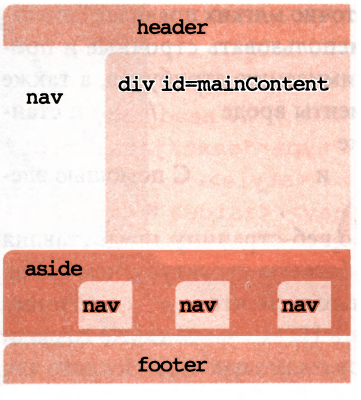


Рис. 1.10. Объединение внешних ссылок в элемент <aside>, относящийся ко всей странице, во избежание слишком большого футера

<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 или приложения, предназначенный для многократного использования и свободного распространения, например, при синдикации».

Запись блога, руководство, заметка новостей, комикс или видеозапись с расшифровкой полностью соответствуют этому определению. 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.

Заключение

В этой главе мы познакомились с HTML5 и его типом документа (DOCTYPE). 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>`. С помощью элементов `<header>`, `<footer>`, `<nav>`, `<aside>` и `<article>` мы создали структурированную веб-страницу, предоставляя браузерам семантически более богатую структуру, чем набор лишенных смысла универсальных элементов `<div>`, которые были единственным вариантом в HTML 4; мы также создали оформление для новых элементов с помощью CSS. 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 (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).

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.

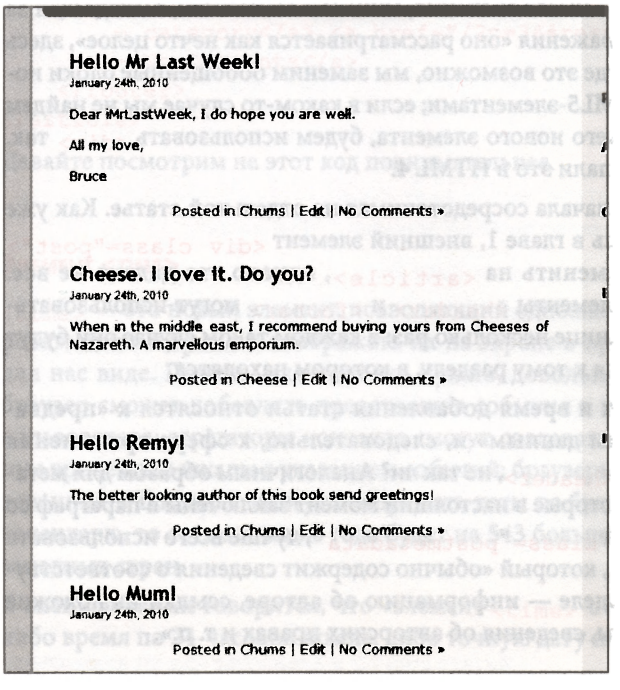


Рис. 2.1. Несколько статей в типичном блоге

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 4 элемента представления `<small>` для оформления даты). Она будет нормально работать в «HTML5-браузерах», но кроме заголовка сообщения блога в документе нет ни одной реальной структуры — только семантически нейтральные элементы `<div>` и параграфы. HTML 4 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 4.

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 «обычно содержит сведения о соответствующем разделе — информацию об авторе, ссылки на похожие документы, сведения об авторских правах и т. п.».

Схематически новая структура статьи блога показана на рисунке 2.2.

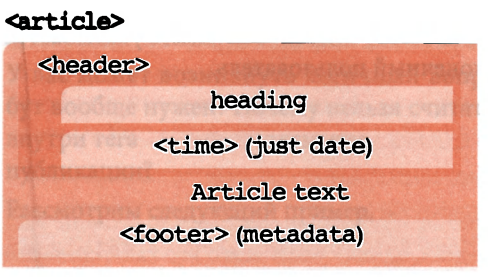


Рис. 2.2. Отдельная статья блога с использованием новых структур HTML5

```
<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" >Books
        ¬ category</a>.<a href="/?p=34#respond">
        ¬ No Comments</a>
    </footer>
</article>
```

Давайте посмотрим на этот код повнимательнее.

Элемент <time>

`<time>` — это новый элемент, позволяющий однозначно кодировать даты и время 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 пролеп-

тическому 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>13<sup>th</sup>
¬ 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.

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

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 HH:MM с добавлением разделяющего префикса «T». Если вы хотите добавить секунды, отделите 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).

Единственная проблема, связанная с элементом `<time>`, состоит в том, что дата по пролептическому 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.

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 переопределить их для заголовка и футера статьи.

```
header {display:block; color:red;
¬ text-align:right;}/*заголовок страницы */
article header {color:blue; text-align:center;}
¬ /*заголовок статьи */
```

Обратите внимание на то, что до сих пор вы еще не использовали приемы с атрибутами `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 (раньше атрибут `cite` элемента `<blockquote>` не поддерживался браузерами).

```
<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>
```

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.

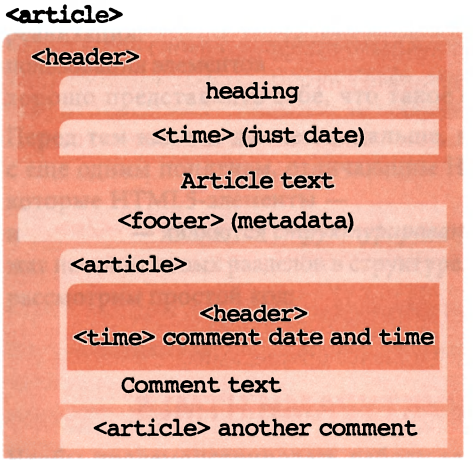


Рис. 2.3. Структура сообщения блога, где комментарии добавлены с помощью вложенных элементов `<article>`

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 washing my hair.</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.

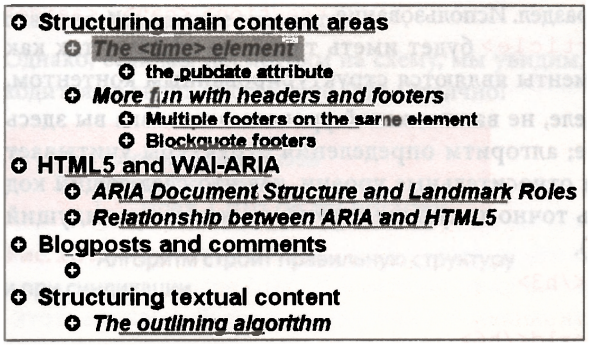


Рис. 2.4. Режим структуры Microsoft Word 2007

В этой части главы вы найдете достаточно много информации. И хотя новые возможности, описанные здесь, еще не реализованы ни в одном браузере (правда, существует JavaScript-реализация: http://code.google.com/p/h5o/), для правильного использования элементов `<section>`, `<aside>` и `<div>` вы должны хорошо представлять себе, что такое структура документа.

Перед тем как мы двинемся дальше, важно познакомиться с еще одним понятием, отличающим HTML5 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 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).

1. Hello
2. World

Рис. 2.5. Простая структура

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

1. Hello
   1. World

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

```
<h1>Hello</h1>
<article>
    <h1>World</h1>
</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, не важно, 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 — синдикация. 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>
```

Предположим, вы работаете в крупной интернет-газете и хотите добавить эту историю. Когда волшебная программа для синдикации 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 синдикации

<NAV>, <ASIDE> И РАЗДЕЛЫ БЕЗ ЗАГОЛОВКА

А сейчас я немного расскажу об инструменте gsnedders.html5.org/outliner/, 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.

Если это 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. Идея использовать для разделов `<nav>` без заголовка менее строгие сообщения пока что не приведена в исполнение.

<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 webblog». Как следует писать разметку для таких слоганов? В HTML 4 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 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, так чтобы он не отображался в структуре документа.

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

1. Metafilter

Рис. 2.8. Частью структуры является только заголовок «Metafilter»

Структура этого кода показана на рис. 2.8.

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 `<article>` 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. Так происходит потому, что 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. 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 76 % пользователей программ экранного доступа «часто или всегда» перемещаются по странице с помощью заголовков (с полными результатами исследования можно ознакомиться на www.webaim.org/projects/screenreadersurvey2/). 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 использует клавишу H 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 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 ни один браузер при построении внутренней модели страницы 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.

<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, jQuery 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 4 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.

В HTML5 одна ссылка может заключать в себе весь элемент `<article>`.

В XHTML 2 был похожий механизм, позволяющий добавлять `href` к любым элементам, однако он, конечно же, не обладает обратной совместимостью; современные браузеры просто посмеются над `<div href="full-story.html">` и проигнорируют ссылку. Интересно, что «блочные ссылки» в настоящее работают в браузерах после пары подсказок — вспомните http://mattwilcox.net/sandbox/html5-block-anchor/test.html.

Ну, почти. В Firefox есть ошибка парсинга, которую Реми называет «Vomit Bug» (remysharp.com/2009/08/10/defining-the-vomit-bug/). Из-за нее Firefox не отображает якори, содержащие некоторые новые HTML5-элементы вроде `<section>`. Правда, делать целые разделы ссылками — довольно странная идея. Если вы ограничиваетесь использованием HTML 4-элементов — параграфов, заголовков, изображений и т. д. — Firefox работает нормально. Это тоже достаточно известная ошибка, которая исчезает, если отключить парсер HTML5.

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.

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 с вложенными элементами `article`, внутренние элементы `article` представляют собой статьи, в общем-то, связанные с содержимым внешнего элемента `article`».

<SECTION>

Элемент `<section>`, напротив, не является «компонентом страницы, документа, приложения или сайта, предназначенным к многократному использованию и свободному распространению». Это либо способ разделения страницы на несколько те-

матических 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 4.

```
<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 4 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 рисунке 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 4 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 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 разделы.

Иными словами, можно создавать родительские элементы `<section>` с вложенными элементами `<article>`, в которых есть один ли несколько элементов `<section>`. Не все документы Pages 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 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 и 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.

Рис. 2.12. Главная страница сайта газеты Guardian

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

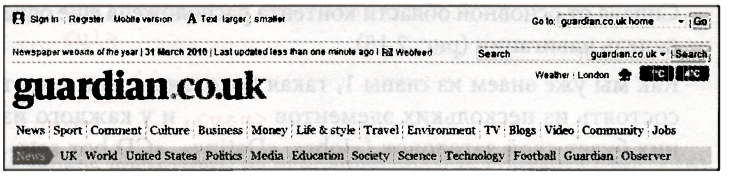


Рис. 2.13. Название бренда на главной странице сайта газеты The Guardian

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

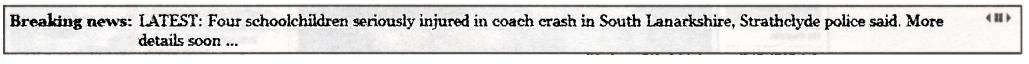


Рис. 2.14. Область «Breaking news» на сайте газеты The Guardian

Если не считать эффекта «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 — «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.

Рис. 2.15. Боковая панель навигации, расположенная в правой части сайта газеты The Guardian

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 нескольких элементов `<nav>`, и у каждого из них будет свой заголовок («Jobs», «Dating», «CD box sets», «Today's paper» и т. д.) на серо-голубом фоне с красной границей наверху.

```
<aside>
    <nav>
        <h2>guardianjobs</h2>
        <form role=search     >     </form>
        <ul>
            <li><a href=#>Upload your
            ¬ CV</a></li>

        </ul>
    </nav>
    <nav>
        <h2>Online Dating</h2>

    </nav>
    <section>
        <h2>Sponsored Features</h2>
    </section>
</aside>
```

Обратите внимание на то, что я добавил в `<nav>` форму поиска; я считаю, что формы поиска не менее важны, чем собственно навигация. ARIA-атрибуту `role` я присвоил значение, соответствующее функциям формы.

Раздел «Sponsored Features» я не поместил в элемент `<nav>`, поскольку это в целом не панель навигации; по-видимому, его основная цель — реклама. Хотя вы можете со мной не согласиться (обещаю, что не обижусь на это) и решите поместить все в один общий `<nav>`.

```
<nav>
    <h2>guardianjobs</h2>
    <form role=search     >     </form>
    <ul>
        <li><a href=#>Upload your CV</a></li>

    </ul>

    <h2>Online Dating</h2>
    <ul>

    </ul>
    ...
```

```
    <h2>Sponsored Features</h2>
    <ul>

    </ul>
</nav>
```

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

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

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

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).

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

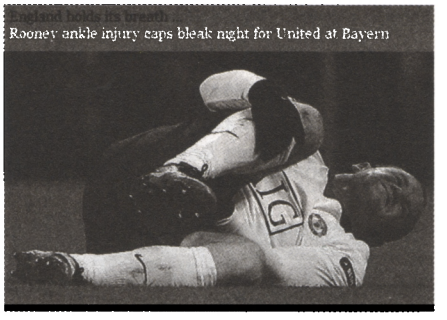


Рис. 2.17. Интересная фотография

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» футера, 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.

Что такое WAI-ARIA

WAI-ARIA (Web Accessibility Initiative's Accessible Rich Internet Applications) — отдельная спецификация, которая «заполняет дыры» в HTML 4 (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 4 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 4-страницы не будут проходить валидацию. Если в остальном ваша разметка построена правильно, не обращайте на это внимания — доступность важнее валидности.

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 `<span>`. 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 4).

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. Если семантика элемента выражает необходимый признак, разработчик может отказаться от использования 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 на html5.validator.nu/ рассчитан на HTML5 и ARIA (в то время как валидаторы HTML 4 в случае обнаружения ARIA-атрибутов выдают сообщение об ошибке, поскольку HTML 4 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.

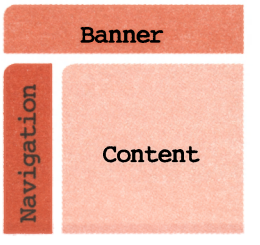


Рис. 2.18. Простая страница с заголовком, боковой панелью и основной областью контента и ARIA-ролями

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. 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.

```
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).

РЕСУРСЫ ARIA

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

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 (www.accessibleculture.org/research/html5-aria) показывает, что две самых популярных на рынке программы экранного доступа не могут правильно обрабатывать контент, для разметки которого используются одновременно HTML5 и ARIA, или элементы `<nav>` внутри `<header>`. Однако не все такие программы работают неправильно; VoiceOver компании Apple не пропускает контент, а свободная программа экранного доступа NVDA (www.nvda-project.org) читает весь контент *и* позволяет перемещаться по структурным элементам страницы.

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 dev.opera.com/articles/view/introduction-to-wai-aria/ и в блоге The Paciello Group (www.paciellogroup.com/blog/). Также я рекомендую две книги: «Universal Design for Web Applications» Уэнди

Чисхолма 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 www.w3.org/WAI/PF/aria/.

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 4. Мы рассмотрим глобальные атрибуты, которые могут быть применены к любому элементу, и скажем «привет» новым HTML5-свойствам, которые останутся за рамками этой книги.

<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"`.

<DETAILS>

Мне очень нравится элемент `<details>`, несмотря на то что он пока еще нигде не реализован. Реми написал гениальный скрипт, позволяющий создавать имитации таких элементов (http://gist.github.com/370590).

Используя его, мы получаем встроенную поддержку элемента с распространенным типом поведения и освобождаем себя от необходимости самим создавать JavaScript-код (или же нечто,

что я видел на многих сайтах, использующее всю библиотеку jQuery) — и это очень здорово. <details> представляет собой область, которая может разворачиваться и сворачиваться. Выглядит это так:

```
<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>.</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>
```

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.

<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 специальных 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
    alt=""> <!-- no alt, as it's covered by the
    ¬ figcaption -->
```

```
<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. 2.19).

Рис. 2.19. Элементы <figure> и <figcaption> плюс немного шикарных CSS3-эффектов. (Смотрите, это же заголовок к рисунку, иллюстрирующему заголовки и рисунки! Вот что я называю «метаинформацией»)

КАТЕГОРИИ HTML5-ЭЛЕМЕНТОВ И МОДЕЛИ КОНТЕНТА

В HTML 4 элементы подразделялись на «блочные» и «строковые». Эти термины не используются в HTML5, поскольку, по сути, относятся к представлению: они просто указывают на способ отображения элементов в браузере — на основе таблиц стилей, использующихся по умолчанию. В самих HTML-элементах нет ничего, что позволяло бы считать их блочными или строковыми.

По умолчанию в CSS каждый элемент обладает свойством `display:inline`, если только это правило не перекрывается встроенной таблицей стилей браузера или великолепным оформлением, которое необычайно талантливый дизайнер (то есть вы) применил к разметке. (Не смущайтесь, вы знаете, что это так; все об этом говорят.)

В HTML5 есть много новых моделей контента, в частности подразделение на *формулирующий* контент (примерно соответствует строковому) и *поточный* (примерно соответствует блочному). Некоторые элементы (<a>, <ins>, <del>) относятся к обоим типам. Некоторые — *заголовочные* (<h1>…<h6>), *структурированные* (<article>, <section>, <nav>, <aside>) — являются их подмножествами.

Также существует *встраиваемый* контент (контент, импортирующий в документ другие ресурсы, или контент, добавленный в документ с помощью элементов <audio>, <canvas>, <embed>, <iframe>, <img>, <math>, <object>, <svg>, <video>), *интерактивный* контент (<a>, <audio> (при наличии атрибута controls), <button>, <details>, <embed>, <iframe>, <img> (при наличии атрибута usemap), <input> (если атрибут type не находится в состоянии hidden), <keygen>, <label>, <menu> (если атрибут type не находится в состоянии toolbar), <object> (при наличии атрибута usemap), <select>, <textarea>, <video> (при наличии атрибута controls)), *метаданные* и др.

Не волнуйтесь! Здесь все очень логично: если отбросить тот факт, что элемент <a> теперь ведет себя как <ins> и <del> и может быть «строковым» или «блочным», вы не заметите никакой разницы, если будете использовать HTML5 Shiv (http://code.google.com/p/html5shiv/), чтобы определить новые HTML5-элементы, пока браузеры еще не сделали этого во встроенных таблицах стилей.

Обратите внимание на то, что `<figcaption>` может содержать только «формулирующий» контент — то есть, в терминах HTML 4, не должен включать блочные элементы.

<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 `<em>`. 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 (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 `<strong>` или `<em>`, так 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, 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 при-

меров 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 специальные скобки с помощью тега `<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. 2.20).

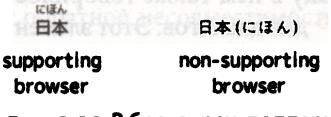


Рис. 2.20. В браузерах, поддерживающих `ruby`, текст-транскрипция отображается над основным текстом. В браузерах, не поддерживающих `ruby`, текст-транскрипция отображается в скобках и следует после основного текста

Во время написания этой книги элемент <ruby> активно обсуждается в W3C HTML5 Japanese Interest Group, так что в результате этих обсуждений спецификация вполне может измениться.

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.

<OL>

В HTML 4 использование атрибута start для элемента <ol> 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?

<DL>

В HTML 4 <dl> обозначал список определений, состоящий из слова и нескольких его определений. Такое описание было запутанным и непонятным, поскольку в нем также говорилось о возможности использования <dl> для диалогов. Этот элемент также 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>Directory of French plastic sandal
    ¬ manufacturers</dt>
    <dd>Phillipe Philloppe</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>
```

<CITE>

В HTML 4 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 4, 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») на 24ways.org Джереми Кейт написал: «Присоединяйтесь к кампании гражданского неповиновения ненужным, запрещающим и приводящим к обратной несовместимости изменениям в правилах использования элемента `cite`».

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

<ADDRESS>

Элемент <address> используется для указания контактной информации автора текущей статьи или документа, а *не* как универсальный элемент для почтовых адресов. В качестве контактной информации может быть указан адрес электронной почты, почтовый адрес и т. д. Для разметки могут быть использованы микроформаты, RDFa или microdata.

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.

<EM>, <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.

Элемент <i> «представляет собой небольшой отрезок текста, который произносится другим голосом или с другим настроением или же 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>
```

<STRONG>, <B>

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 <b> «представляет собой отрезок контента, который необходимо стилистически отделить от остального текста, не придавая ему дополнительного значения; это могут быть ключевые слова в аннотации, названия продукта в обзоре или другой текст, который в типографике выделяется полужирным шрифтом».

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>
```

<HR>

Элемент <hr> теперь не зависит от среды и обозначает «тематический разрыв на уровне параграфа». На HTML5doctor есть очень тонкий 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 cHTML-браузерах для мобильных телефонов, поддерживающих только базовые CSS и не отображающих визуальное оформление (и вместе с ним визуальное обособление разделов)».

Наш совет: используйте вместо этого структурированный контент и заголовки вместе с изящными разделителями CSS.

<SMALL>

Элемент <small> был полностью переопределен. 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».

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 `<strong>` и `<em>`.

УДАЛЕННЫЕ ЭЛЕМЕНТЫ

Некоторые элементы, знакомые вам по HTML 4, были признаны полностью устаревшими в HTML5: `<applet>` (вместо него используйте `<embed>`), `<blink>`, `<center>`, `<font>` 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.

CONTENTEDITABLE

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

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, внесенные в документ изменения затрагивают выделенный контент, который специально подобран и является редактируемым, то есть можно сделать текст 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 требует написания `contentEditable` (обратите внимание на заглавную `E`). В DOM также есть свойство `isContentEditable`, которое позволяет определить, является ли элемент редактируемым, поскольку флажок `contenteditable` мог быть унаследован от родительского элемента.

Чтобы сделать весь документ редактируемым, можно использовать `document.designMode = 'on'` (не `'true'`). Однако для этого необходим JavaScript.

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, также затрагивают DOM.

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`).

CONTEXTMENU

Атрибут `contextmenu` относится к элементам `<menu>` и `<command>`, которые не обсуждаются в этой книге (см. раздел «Свойства, не рассматриваемые в этой книге»).

DATA-*

Чтобы передавать информацию скриптам из разметки, можно использовать атрибут `data-*`. Передавать можно любое XML-совместимое имя. Более подробно об этом рассказывается в главе 4.

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 ком случае не должен отображаться. Это позволяет не просто скрывать контент, а 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 больше 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`.

ITEM, ITEMPROP, SUBJECT

Эти атрибуты относятся к спецификации microdata, которая не обсуждается в этой книге (см. раздел «Свойства, не рассматриваемые в этой книге»).

АТРИБУТЫ ROLE, ARIA-*

Как вы уже поняли, HTML5 рассматривает WAI-ARIA как разрешенные дополнения к языку.

SPELLCHECK

Этот булев атрибут заставляет браузер проверять правописание и грамматику элемента. Если его нет, «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 элементов. Но сейчас это не всегда 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. www.juicystudio.com/article/ie-keyboard-navigation.php). В HTML 4 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!

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 откладывать его валидацию. HTML5 решил облагородить эту «коровью тропу» и наконец включил `<embed>` в официальную спецификацию языка.

<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.

<PROGRESS>, <METER>

Элемент `<progress>` используется для отображения «индикатора выполнения», который показывает степень завершенности процесса — например, загрузки.

Элемент `<meter>` «отображает скалярное значение в известном диапазоне или дробное значение; например, использование диска, релевантность результатов запроса или долю избирателей, выбравших данного кандидата, от общего числа голосовавших».

<MENU>, <COMMAND>

Это потрясающие элементы: они позволяют создавать для контекстных меню вашего приложения панели инструментов 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.

MICRODATA

microdata — это способ разметки элементов с помощью дополнительных считываемых данных для того, чтобы поисковые роботы, поисковые движки и браузеры могли извлекать информацию из содержимого страницы. Он похож на RDFa (стандарт W3C), и уже индексируется поисковым движком Google. Однако ни один браузер не поддерживает соответствующий API.

<STYLE SCOPED>

Атрибут `scoped`, примененный к элементу `<style>`, заставляет браузер применить стили к элементу, внутри которого расположен `<style scoped>`, и его детям. Это позволяет локализовать стилевое оформление внутри HTML-документа. Так, например, элемент `<article>`, содержащий блок `<style scoped>`, сохранит свое оформление при синдикации.

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 `<span>`. Последние остаются важной частью языка, но, как и в HTML 4, 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 4 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 ❤

HLML5 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`.

Но есть кое-что более интересное: ряд элементов, которые раньше обязаны были располагаться внутри формы (`<button>`, `<fieldset>`, `<input>`, `<label>`, `<select>`, `<textarea>` и `<object>`), а также новые элементы (`<keygen>`, `<meter>`, `<output>` и `<progress>`) теперь можно помещать в любое место страницы, сохраняя связь с формой при помощи атрибута `form`, указывающего на ее `id`.

Рассмотрим следующий пример.

```
<form id=foo>
<input type="text">

</form>
<textarea form=foo></textarea>
```

Элемент `<input>` принадлежит форме `foo`, поскольку он находится внутри нее и не содержит атрибута, переопределяющего его принадлежность. Элемент `<textarea>` расположен снаружи `foo`, но все равно относится к ней, потому что его атрибут `form` указывает на `id` этой формы.

Благодаря этому дизайнер будет чувствовать себя гораздо свободнее, если захочет, чтобы визуально (и структурно) эти элементы располагались вне родительской формы.

Но самое главное — это новые типы форм и их встроенная валидация. 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).

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 ввода. Поскольку все браузеры по умолчанию используют `<input type=text>`, устаревшие версии, не распознающие новых расширений, обратятся к значению по умолчанию и предложат пользователю обычное текстовое поле ввода.

ПРИМЕЧАНИЕ

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 Safari/iPhone (рис. 3.1).

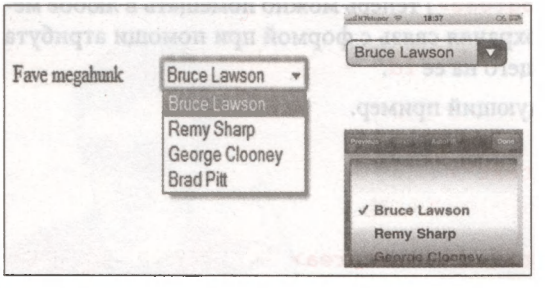


Рис. 3.1. Одно и то же поле выбора в Safari/Windows (*слева*) и Safari/iPhone. Большинство скриншотов сделано в Opera, так как на момент написания книги (май 2010 года) этот браузер обладал самой полной реализацией; но если какое-то свойство реализовано в другом браузере, для иллюстрации мы используем его

Тип ввода данных email

Строка `<input type=email>` сообщает браузеру, что форма не должна быть отправлена, если 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).

По поводу того, как браузер должен сообщать об ошибках, в спецификации ничего не сказано. Пример реализации (экспериментальной) в Opera 10.5 показан на рис. 3.2.

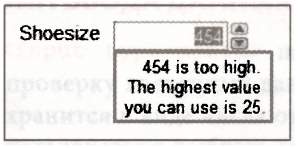


Рис. 3.2. Автоматически сгенерированное сообщение об ошибке в Opera 10.5

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 флажками.

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/).

ТИП ВВОДА ДАННЫХ URL

Строка `<input type=url>` заставляет браузер проверять, что пользователь ввел правильный URL. Браузер может помочь пользователю это сделать — Opera, например, показывает список недавно посещенных URL (из журнала браузера) и добавляет http:// в начало URL, начинающихся с www. (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 различаются внешним видом, удобством использования и доступностью.

`<input type=date>` решает эту проблему. Opera, к примеру, открывает виджет с календарем (рис. 3.3).

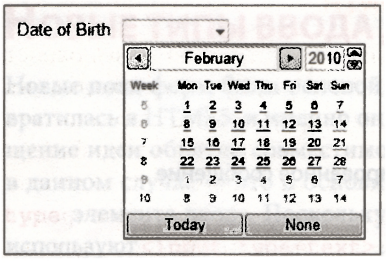


Рис. 3.3. Opera 10.5 открывает виджет с календарем

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

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

Безусловно, это только начало. Браузер может 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 не более чем элементами `<div>`, `<span>` 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

`date` и `time` проверяют введенные точную дату и время. 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 `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 — вас*. 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 10.5

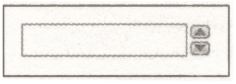


Рис. 3.6. `<input type=number>` в Opera 10.5

ТИП ВВОДА ДАННЫХ NUMBER

Как это ни удивительно, но тип ввода данных `number` выдает ошибку, если пользователь не ввел численные символы. Он прекрасно работает с атрибутами `min`, `max` и `step`. В Opera 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).

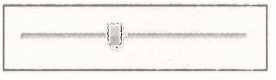


Рис. 3.7. `<input type=range>` в Google Chrome

ТИП ВВОДА ДАННЫХ RANGE

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

ТИП ВВОДА ДАННЫХ SEARCH

Этому типу ввода данных требуется поисковый термин. В Safari есть 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. Дополнительная клавишная панель в iPhone для заполнения `<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).

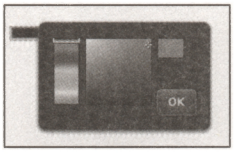


Рис. 3.9. `<input type=color>` в BlackBerry

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

Элемент `<datalist>` отличается от поля выбора тем, что пользователь может вводить свой собственный текст, если его не устраивают предложенные 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=Ms value=Ms>
        <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, о которых говорилось выше: например, `url`, `email` и т. д. Кроме того, по мере того как пользователь вводит текст, параметры можно динамически изменять, имитируя возможности Google Suggest. Более подробно об этом см. http://dev.opera.com/articles/view/an-html5-style-google-suggest/.

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> k<datalist> превращается в<input type=text>, благодаря чему пользователь может хотя бы что-то ввести, а вы уже можете дополнить реализацию с помощью JavaScript.

АТРИБУТ AUTOFOCUS

Булев атрибутautofocus — способ задания установки фокуса на определенном элементе управления формы во время загрузки страницы. Раньше разработчику требовалось писать для этого JavaScript-код, используяcontrol.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 атрибутаplaceholder. В спецификации говорится: «Для добавления более подробной подсказки или совета лучше использовать атрибутtitle».

КОГДА БРАУЗЕРАМ ТРЕБУЕТСЯ ПОМОЩЬ

В старые — и иногда даже новые — браузеры некоторые из этих возможностей не встроены. Но это не такая уж большая проблема, ведь *большинство* новых возможностей HTML5 можно реализовать с помощью JavaScript (или других технологий, таких как Flash и Silverlight). Это не относится ни к изящному ухудшению качества, ни к прогрессивному улучшению; это что-то другое. Это то, что я мои коллеги называем «заполнением», при котором во время создания кода мы опираемся на встроенные возможности, а затем заполняем «дыры» в поддержке, используя внешнюю библиотеку. Библиотека может быть универсальной или специализированной. Например, если вы хотите воспользоваться атрибутами autofocus или placeholder, достаточно добавить маленькую JavaScript-библиотеку, и все современные браузеры от IE6 до Firefox 3.6 будут поддерживать эти атрибуты (http://introducinghtml5.com/examples/ch03/pollyfilling.html).

Сейчас этот атрибут поддерживается в браузерах на движке WebKit (который изначально был собственностью Apple) и бета-версии Firefox 4.

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. 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>
```

Этот атрибут можно добавлять и к другим типам ввода данных: например, `<input type=email multiple>` позволяет вводить несколько адресов электронной почты. На сегодняшний день это реализовано только в браузерах WebKit.

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.">
```

ПРИМЕЧАНИЕ

Если регулярные выражения пугают вас, но вы хотите побольше о них узнать или просто подкрепить свои знания, можете взглянуть на блог Стивена Левитана: http://blog.stevenlevithan.com/.

Можно, кстати, добавить `placeholder="9AAA"` 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 использование `^(:?` 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). Тогда нам потребуется усложнить регулярное выражение.

```
<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 шаблоне 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, опреде-

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

```
/^(:?[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.

АТРИБУТ AUTOCOMPLETE

В большинстве браузеров есть свои функции автозаполнения. Новый атрибут `autocomplete` позволяет вам контролировать его работу.

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 мрачного 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 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 mix=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); но обычно такие элементы управления требуют ввода времени с точностью до одной минуты.

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

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 можно воспользоваться автоматической валидацией: для этого требуется всего лишь несколько новых

типов форм, каждая их которых должна содержать атрибут `name` (согласно спецификации). 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! И никакого JavaScript!

ПОЛЗУНОК СО СКРИПТОМ ДЛЯ ВЫВОДА

Мы уже знакомы с `<input type=range>`, так что 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. 3.10) приведем результат работы следующего кода.

```
input[type=range]{width:500px; color:red;
¬ font-family: cursive; font-size:2em;}
111
```

Рис. 3.10. <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 относиться к какой-нибудь форме, и тогда он должен либо располагаться внутри нее, либо содержать ее атрибут. Для удобства взаимодействия с пользователем к форме могут быть также привязаны новые элементы `<progress>` и `<meter>`.

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

```
<output onforminput="value=tap.value">5</output>
```

Фактическое содержимое элемента `<output>` (в нашем случае «5») будет отображаться только до тех пор, пока ползунок не будет перемещен. Необходимо внимательно следить за тем, чтобы это значение совпадало со значением атрибута `value` соответствующего поля ввода. В нашем примере часть `value=...` задает в качестве отображаемого значения то, которое вы выберете с помощью ползунка. Это похоже на `this.innerHTML`, а `value` выступает в роли краткой формы записи.

Для элемента `<output>` можно создать таблицы стилей (правда, сейчас это не поддерживается в Opera).

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=tap
    name=tap
    type=range
    min=1
    max=10
    value=0
    role=slider
    aria-valuemin=1
    aria-valuemax=11
    aria-valuenow=0>
```

`role=slider` сообщает вспомогательным технологиям, как сопоставить этот элемент управления и элементы управления операционной системы. При изменении положения ползунка значение `aria-valuenow` нужно обновить с помощью JavaScript. В таком случае вы захотите обратиться 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=10
    value=0
    role=slider
    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, проследив за элементом 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; оставьте ее в качестве запасного варианта и выполните проверку возможностей. 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», то 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), и тогда браузерам, поддерживающим HTML5, не нужно будет загружать и обрабатывать JavaScript, который им не нужен.

ПРИМЕЧАНИЕ

В качестве альтернативного варианта можно использовать библиотеку Modernizr, и она сделает всю работу за вас. Это небольшая библиотека (в настоящий момент 7 Кбайт), которая выполняет проверку возможностей и возвращает булевы значения JavaScript (например, `Modernizr.inputtypes[email]`). В ней есть сбои (осторожнее с WebKit), но она пока еще находится в процессе разработки. Загрузить ее можно с www.modernizr.com/.

```
var i = document.createElement("input");
i.setAttribute("type", "email");
return  .type !== "text";
```

Атрибуты тоже можно проверять.

```
return 'autofocus' in document.
createElement("input");
```

ОСТОРОЖНЕЕ С WEBKIT

Стоит отметить, что современные браузеры WebKit для ПК, а именно Safari и Chrome, судя по результатам проверки возможностей, поддерживают типы `email`, `url`, `tel` и `number`. На самом деле, в «живых» версиях этой поддержки еще нет, а причиной путаницы является тот факт, что Mobile Safari (тоже на движке WebKit) поддерживает эти типы ввода данных, чтобы настраивать клавиатуру в зависимости от типа (такая клавиатура показана на рис. 3.8). То есть сам браузер добавляет специфическую для устройства поддержку, обычно в виде клавиатуры (особенно на iPhone), зависящей от типа ввода данных.

Итак, что вам это дает? Во-первых, вы создаете код, ориентированный на будущее — время, когда все браузеры будут поддерживать эти дополнения к формам. Во-вторых, вы выигрываете в удобстве использования и доступности.

Оформление новых полей формы и сообщений об ошибках

Каждый раз, когда мы рассказываем о наших новых логичных полях форм на конференциях, кто-то спрашивает, как создавать стили для таких полей. Практически ко всем новым элементам управления можно добавлять базовое оформление: шрифты, цвета и т. п. К этим элементам также применимы некоторые возможности CSS Basic User Interface Module (http://www.w3.org/TR/css3-ui/) — в частности, псевдоклассы `:invalid` и `:required`. Но если вы хотите, к примеру, сделать выходные дни в поле выбора даты зелеными или сделать сообщения об ошибках фиолетовыми, у вас это не получится. Селекторы и дополнительные CSS-средства для этих частей элементов управления формами еще не определены.

Но это не плохо. Специалисты по брендингу, конечно, расстроятся, если текст-подсказка в поле не будет оформлена в корпоративном стиле. Но зато вы выигрываете в удобстве использования и доступности. Хотя иногда очень хочется самому создать стили для содержимого полей формы (что может иногда доходить до безумия — см. http://meyerweb.com/eric/thoughts/2007/05/15/formal-weirdness/), 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

Когда мы добавляем новые HTML5-типы элементов управления формами или атрибут `required`, браузер использует готовую валидацию со встроенными сообщениями об ошибках. Это замечательно, но что если вы хотите изменить эти сообщения в соответствии с требованиями заказчика? Что если они написаны так, как будто 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. С помощью атрибутов контента (которые трудно добавлять в разметку) вы не сможете создать ваши собственные сообщения валидации, однако для этого можно использовать атрибуты DOM (которые доступны благодаря JavaScript). Так что у вас есть два варианта: добавить их до запуска валидации или же сразу после отправки формы добавить специальное сообщение валидации и в этот момент динамически создать это сообщение, добавив в него значение, вызвавшее ошибку.

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.

humptydumptybe not a legal e-mail address

Метод `setCustomValidity` позволяет задать ваше собственное сообщение валидации.

```
var email = document.getElementById('email');
email.form.onsubmit = function () {
    email.setCustomValidity(email.value +   be
    ¬ not a legal e-mail address");
};
```

На рис. 3.11 и 3.12 показано, как Opera отображает специальное сообщение валидации.

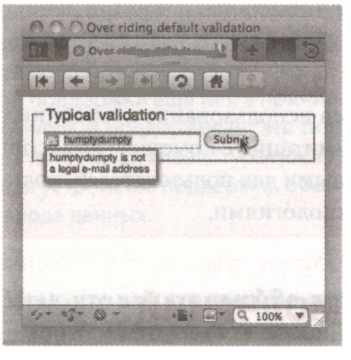


Рис. 3.11. Сообщение валидации для адреса электронной почты в Opera

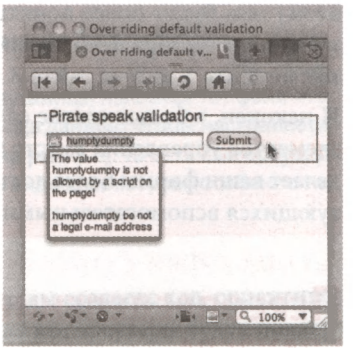


Рис. 3.12. Ваше собственное сообщение валидации, имитирующее речь пиратов, в Opera

Сейчас Opera является единственным браузером с поддержкой встроенных сообщений валидации, так что здесь нечего

сравнивать. Я говорю это потому, что вы наверняка заметили, что перед специальным сообщением валидации добавлено еще одно: «The value humptydumpty is not allowed by a script on the page!». Его добавил сам браузер, и мы никак не можем этого изменить (сейчас). Это стоит учитывать, если вы собираетесь использовать свои собственные сообщения валидации.

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 чем мы и поговорим в следующем разделе.

Использование JavaScript для DIY-валидации

Наряду с атрибутами контента, которые позволяют осуществлять валидацию со стороны клиента и создавать качественные виджеты для веб-форм (как в большинстве HTML5-спецификаций), существует дополнительный интерфейс JavaScript API, который дает нам возможность полностью управлять валидацией и отправкой формы.

С помощью JavaScript API 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

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. Если вы хотите самостоятельно управлять отображением результатов валидации, то 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 = docuement.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 `valid` 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`.

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 `nonvalidate`, который может быть задан только для элемента формы, отменяет валидацию данного конкретного поля. На практике это пока не используется, и все примеры являются чисто теоретическими. Представьте, что вы используете `type="email"`, причем в браузерах, поддерживающих этот `type="email"`, который позволяет пользователю выполнять поиск в записной книге. Но вас не интересует правильность текста, введенного вручную — возможно, потому, что этот адрес может относиться к внутренней сети, и, следовательно, не похож на обычные электронные адреса. И в таком случае вам понадобится атрибут `nonvalidate`.

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>
```

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 `oninputchange`. 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.

Например, если бы мне нужно было создать поле выбора цвета одновременно в модели RGBA и HSLA, мне бы, скорее всего, пришлось добавить приемника 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.onforminput = function () {
    var i = this.length, values = [], value = 0;
    while (i--, value = this[i].value) {
        if (this[i].type == 'range') {
            switch (this[i].name) {
                // альфа-канал имеет значение
                ¬ от 0 до 1
                case 'alpha_channel'
                ¬ values.push(value / 100);break;
                // тон имеет простое значение от
                ¬ 0 до 360
                case 'hue'  values.push(value);
                ¬ break;
                // по умолчанию насыщенность
                ¬ и яркость указываются
                ¬ в процентах
                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 `onforminput` и новые элементы вывода для отображения значения (хотя это можно было сделать с помощью `.innerHTML`). Результат показан на рис. 3.13.

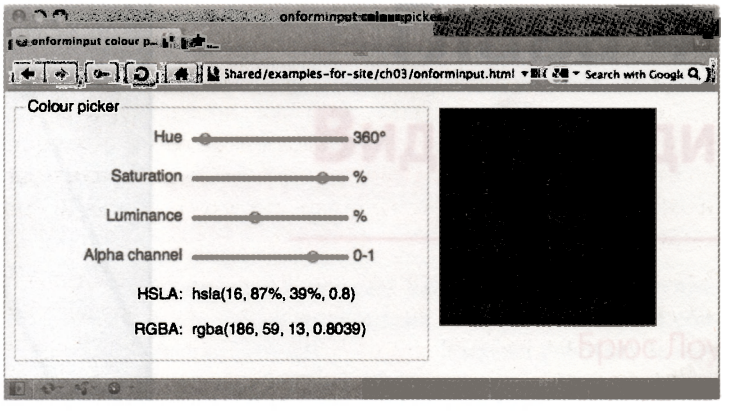


Рис. 3.13. Поле выбора цвета в HSLA с использованием события onforminput

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 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 совершенствования компьютерных технологий музыка в формате MP3 вытеснила MIDI, и стало появляться привычное нам видео. Разнообразные плееры боролись за место под солнцем — Real Player, Windows Media и так далее, — 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 видео, и для аудио.

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>`), кроме того, у него есть особая семантика `<video>`, похожая на семантику `<img>`».

В то время как API стал более сложным, первоначальная мысль ван Кестерена и сейчас используется во всех основных браузерах, и уже по ходу написания этой книги Microsoft объявила о том, что в Explorer 9 в этом отношении тоже ничего не изменится.

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 другом, а также отметим их сходства и отличия.

<VIDEO>: зачем нужен элемент <VIDEO>?

Раньше, если разработчики хотели разместить видео на вебстранице, им приходилось использовать элемент `<object>`, универсальный контейнер для «внешних объектов». Из-за раз-

личий 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 пользователь добавляет на страницу плагин, под него резервируется область на странице, которая потом передается плагину браузером. Пока браузер не установил соединение, область, предназначенная для плагина, остается черным ящиком, и браузер совершенно не интересует, что там происходит.

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 размеры страницы динамической меняются. 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 стандартизированный 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>` могут быть стилизованы при помощи CSS; изменение размеров можно реализовать при помощи CSS. Видео можно настроить и отобразить в теге `<canvas>` с использованием JavaScript. А самое большое достоинство заключается в том, что пользователь может менять настройки так, как считает нужным, а не ограничиваться лишь тем, какие возможности были заложены создателем ПО. 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.

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 так же просто, как сейчас изображения. Некоторые браузеры предоставляют простой способ доступа к видео (или даже его сохранения) через контекстное меню. Мы не то чтобы говорили о том, что цифровое управление правами — штука бессмысленная. Но оно будет только раздражать честных пользователей и создавать лишь небольшие неудобства пиратам.

Существует не до конца разработанный элемент `<device>`, предназначенный для следующих версий HTML, но пока

браузеры его не поддерживают. Так что плагины остаются лучшим вариантом передачи видео и аудио через браузер с машины пользователя на веб-страницу типа Daily Mugshot или Chat Roulette. Конечно, без Chat Roulette мы бы остались совершенно одиноки в этом мире, но не забывайте и об огромном объеме данных на этой станице, с которым в далеком будущем придется справляться плагинам.

Анатомия элемента видео

В самом простом случае для размещения видео на странице в HTML5 потребуется такой код:

```
<video src=turkish.ogv></video>
```

Расширение файла `.ogv` используется здесь, чтобы указать на формат видео Ogg Theora. Так же как и при использовании `<object>`, между тегами можно вставить сообщение, которое всплывет, если элемент не сработает (для более ранних веб-браузеров, которые не поддерживают 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.ogv>
    Download the
<a href=leverage-a-synergy.ogv>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.ogv autoplay>
</video>
```

ЭЛЕМЕНТЫ УПРАВЛЕНИЯ (CONTROLS)

Если вы предоставляете пользователю возможность управлять проигрыванием видео при помощи элементов управления, это будет в сотню раз лучше, чем воспроизводить видео автоматически (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.ogv controls>
</video>
```

Между элементами управления, так же как и между браузерами, могут быть некоторые различия. В этом нет ничего необычного. Вы в любом случае сможете поставить видео на воспроизведение или паузу, воспользоваться панелью поиска и прибавить/убавить звук.

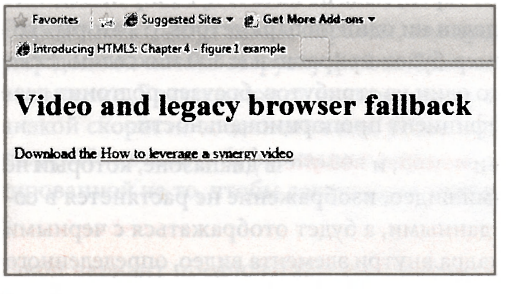


Рис. 4.2. Стандартные элементы управления в Firefox 3.6 (во всех современных браузерах примерно те же)

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 на странице вообще ничего не будет отображаться, но, конечно, все сохранится в модели DOM, и содержанием можно будет управлять при помощи JavaScript и новых API.

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.

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 и `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, сохраняя привязку к коэффициенту пропорциональности.

ПРИМЕЧАНИЕ

У каждого из браузеров есть свои особенности, обуславливающие разный уровень доступности видео с клавиатуры. Если в Firefox отключен JavaScript, элементы управления не будут отображаться (контекстное меню позволит пользователю воспроизвести и остановить фильм, но без JavaScript это будет весьма нетривиальной задачей). Встроенные в Opera элементы управления всегда отображаются, когда отключен JavaScript, вне зависимости от того, определен ли атрибут `controls`. У Chrome и Safari также есть проблемы с доступностью с клавиатуры. Мы, в свою очередь, постарались упредить возникновение каких-либо сложностей в этом вопросе.

ПОВТОР (LOOP)

Атрибут `loop` также является логическим выражением. Как вы, наверное, догадались, он определяет бесконечный повтор воспроизведения.

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 `автоматическое воспроизведение`. 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>
```

У атрибута `preload` есть свои особенности. Если вы просто добавите его в свой код и не определите никак иначе, вы дадите пользователю возможность самостоятельно решить, что ему делать. Ведь, например, браузер мобильного телефона может по умолчанию не начинать буферизацию, пока этого не затребовал пользователь.

1. `preload=auto` **(или просто** `preload`**)**

Определяя атрибут таким образом, мы предполагаем, что браузер должен начать загрузку файла. Заметьте, это только предположение. Браузер может игнорировать его, например, из-за низкой скорости соединения или установки в мобильном браузере «Не начинать буферизацию медиа по умолчанию», ориентированной на то, чтобы сэкономить трафик.

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

Технические характеристики атрибута `preload` изменились в марте 2010 и с апреля 2010 больше не применяются.

ИСТОЧНИК (SRC)

Этот атрибут, так же как и `<img>`, указывает на то, что файл должен быть отображен. Однако так как некоторые браузеры не могут воспроизводить определенные форматы, следует указать несколько исходных файлов. Мы поговорим об этом в следующем разделе. При работе с одним исходным файлом атрибут `src` обычно используется только для быстрого макетирования или для сайтов интранет, когда вы знаете, какой браузер у пользователя и какие он поддерживает кодеки.

Кодеки: ужас-ужас

Ранние версии технических характеристик 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. Например, в играх Microsoft Halo он часто используется с Theora для видео и объединяется в контейнер Ogg.

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. Opera и Firefox поддерживают Theora и Vorbis. Safari, напротив, предоставляет поддержку видеокодека H.264 и аудиокодека MP3. Microsoft объявила, что IE9 также будет поддерживать H.264 (та же ситуация с iPhone и Android). Google Chrome поддерживает видео Theora и H.264, а также аудио Vorbis и MP3. Запутались?

Когда мы заканчивали работу над этой книгой, корпорация Google объявила, что она открывает исходные коды видеокодека под названием VP8. Этот кодек обеспечивает высокое качество

данных, а в сочетании с Vorbis в контейнере, основанном на формате Matroska, все это известно как «webM».

Opera, Firefox и Chrome объявили о том, что они реализуют поддержку кодека VP8. IE9 также обеспечит его поддержку при условии того, что кодек будет инсталлироваться отдельно. VP8 будет включен в Adobe Flash Player и в видео YouTube (в формате webM). Так же как и Theora, этот кодек бесплатный. В этой главе вы можете заменить примеры .ogv на .webm для видео в хорошем качестве (если, конечно, браузер поддерживает этот кодек).

Золотое правило: чтобы все пользователи могли просматривать контент, необходимо на сайте обеспечить поддержку бесплатного формата webM или Theora 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.

Множественные элементы <source>

Чтобы это сделать, нужно закодировать мультимедиа дважды: один раз как Theora 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. Вместо того чтобы использовать отдельный атрибут `src`, вы представляете отдельные элементы `<source>` в форме вложений для каждой части кода с соответствующими атрибутами `type` внутри элемента `<audio>` или `<video>` и позволяете браузеру подгрузить тот формат, в котором он может отображать данные.

Заметьте, в этом случае мы не используем атрибут `src` в самом элементе медиа:

```
1 <video controls>
2     <source src=leverage-a-synergy.ogv
      ¬ type='video/ogg; codecs="theora,
      ¬ vorbis"'>
3     <source src=leverage-a-synergy.mp4
      ¬ type='video/mp4; codecs="avc1.42E01E,
      ¬ mp4a.40.2"'>
4 <p>Your browser doesn't support video.
5 Please download the video in <a href=leverage-a-
  ¬ synergy.ogv>Ogg</a> or <a href=leverage-a-
  ¬ synergy.mp4>mp4</a> format.</p>
6 </video>
```

ПРИМЕЧАНИЕ

Мы обнаружили баг, который влияет на работу некоторых iPad. Они могут считать только первый элемент `<source>`. Поэтому версия mp4 в списке источников должна идти перед бесплатной версией.

Первая строка указывает браузеру на то, что в отображаемый контент нужно включить видео и предоставить стандартные элементы управления. Вторая предлагает видео в формате Ogg Theora и использует атрибут `type`, чтобы браузер мог определить, какой тип контейнера используется (при помощи указания типа MIME) и какой кодек использовался для кодировки видео- и аудиопотока. Здесь мы также можем предложить видео webM как опцию для изображения в высоком качестве. Обратите внимание на то, что мы использовали кавычки для указания параметров. 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.

В третьей строке предлагается видео H.264. Строка с кодеками для H.264 и AAC сложнее, чем для Ogg, так как для H.264 и AAC есть несколько профилей. Более высокие профили сильнее нагружают ЦПУ при декодировании, но коэффициент сжатия у них выше, так что на передачу видео уходит меньше трафика.

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). 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 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 уступающем 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)? Благодаря тому, что файл MP4 может проигрываться при помощи плагина Flash player, вы можете использовать фильм в данном формате в качестве дополнения для более ранних версий браузеров, таких как Internet Explorer 8. Код, который нам придется для этого написать, конечно, ужасен. Но он выглядит ничуть не хуже, чем код для любых задач подобного рода. Кроме того, он будет работать в любой версии браузера, если установлен Flash Player. Вы можете ознакомиться с этой остроумной техникой в статье под названием «Видео для всех!», написанной Кроком Карменом (http://camendesign.com/code/video_for_everybody).

Вы также можете добавить на сайт с видео сообщение, которое всплывет при ошибке загрузки, и разместить ссылку между тегами элемента `video`:

```
<video controls>
    <source src=leverage-a-synergy.ogv
    ¬ type='video/ogg; codecs="theora, vorbis"'>
    <source src=leverage-a-synergy.mp4
    ¬ type='video/mp4; codecs="avc1.42E01E,
    ¬ mp4a.40.2"'>
<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. Несколько раз это создавало мне проблемы.

Можно использовать библиотеку html5media http://static.etianen.com/html5media/, чтобы исключить элемент `<video>` и автоматически добавить необходимый вариант исправления ошибки, написав одну строку на JavaScript в верхней части страницы.

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. 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 www.webmproject.org/tools/, чтобы ознакомиться с постоянно пополняемым списком инструментов для этой задачи.

Бесплатное приложение evom (http://thelittleappfactory.com/evom/) может генерировать Ogg Theora на Mac при помощи приятного графического интерфейса. Пользователи Windows и Mac могут загрузить Miro Video Converter (www.mirovideoconverter.com/), который позволяет перетащить файл в окно программы и таким образом преобразовать его в Theora или H.264. Эта программа оптимизирована для различных устройств, таких как iPhone, Android Nexus One, PS2 и т. д. Бесплатная программа VLC (www.videolan.org/vlc/) может конвертировать файлы в формат Ogg для Windows или Linux. OggConvert (http://oggconvert.tristanb.net/) — утилита для пользователей Linux.

Кроме того, программа Firefogg, доступная через Firefox и расположенная на сайте http://firefogg.org, предоставляет простой способ конвертации онлайн. TinyOgg (http://tinyogg.com/) конвертирует Flash-видео в Ogg для загрузки. В нее можно даже ввести URL YouTube.

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. Также на сервер можно инсталлировать открытую библиотеку ffmpeg (http://ffmpeg.org/) 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 видио) в соответствии с одной из различных CC-лицензий, ознакомьтесь со средством Internet Archive (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.

Видеофайлы 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 отсылать видео в формате HD, предназначенном для больших экранов, на экран смартфона или КПК. Сжатие видео до размера, подходящего для маленького экрана, может сэкономить трафик. А это, безусловно, поднимет настроение обладателю карманного устройства, использующего ваш сервер.

HTML5 позволяет использовать атрибут `media` по отношению к исходному элементу. При этом браузер должен определить размер экрана (количество цветов, коэффициент пропорци-

ональности и т. д.) и активировать тот файл, который лучше подходит под характеристики экрана.

Эта функциональность и синтаксис заимствованы из технических характеристик CSS Media Queries (dev.w3.org/csswg/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 в нем ширина экрана устройства в 800px, то есть допустим ли в нем широкоформатный режим. Если ответ на запрос положительный, он получает `hi-res.ogv`; если нет — `lo-res.ogv`:

```
<video controls>
        <source src=hi-res.ogv
        ¬ media="(min-device-width: 800px)">
        <source src=lo-res.ogv>
</video>
```

Обратите внимание на то, что вы все еще должны использовать атрибут `type` с параметрами `codecs` и контентом, демонстрирующемся при ошибке воспроизведения, о котором мы говорили ранее. В данном случае мы его выпустили, чтобы не перегружать код.

Пользовательские элементы управления

Одно из крайне удачных решений в плане элемента медиа, а следовательно, и элементов аудио и видео, — это то, что API JavaScript крайне прост. API и для аудио, и для видео наследуется от одного и того же API медиа, так что они почти одинаковы. Единственное отличие этих элементов заключается в том, что у элемента видео есть такие атрибуты, как `height` и `width`, а также атрибут `poster`. События, методы и все остальные атрибуты одинаковы. На этом основании мы перейдем к обсуждению самого интересного элемента медиа, `<video>`, и работы с ним при помощи JavaScript.

Как вы уже поняли из начала этой главы, Анне ван Кестерен при обсуждении нового API говорит о том, что у нас появились простые методы, такие 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. 4.1 вы найдете список атрибутов и событий медиа.

ПРИМЕЧАНИЕ

Мы используем `min-device-width`, а не `min-width`, для работы с устройствами, у которых есть окно просмотра содержимого, то есть любыми полнофункциональными браузерами смартфонов, потому что таким образом мы можем получить информацию о ширине отображения окна просмотра.

Таблица 4.1. Медиаатрибуты и события

Состояние ошибки	
`error`	`canPlayType(type)`
Состояние сети	
`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`	
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`	

Используя JavaScript и API, 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 видео; 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 флеш-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.

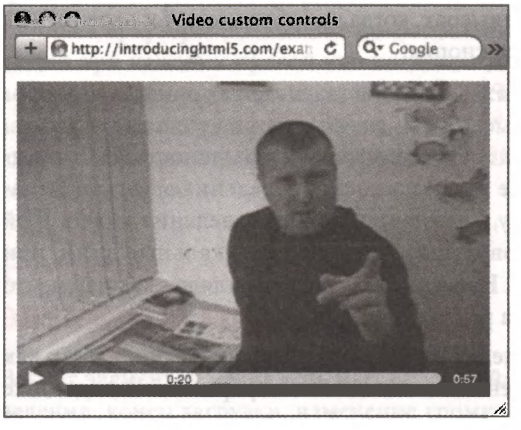


Рис. 4.3. Наши простые, но уже видоизмененные элементы управления плеером

Начнем мы с видео со стандартными элементами управления. Затем мы используем JavaScript, чтобы переписать их в соответствии с нашими желаниями. Таким образом, если JavaScript не сработает, у пользователя все-таки будет возможность контролировать видео:

```
<video controls>
    <source src="leverage-a-synergy.ogv"
    ¬ type="video/ogg" />
    <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.ogv">Ogg</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, реагирующую на нажатие. Таким образом, с ее помощью мы сможем реализовать функциональность «воспроизведение/пауза». В наших примерах, когда мы обращаемся к переменной `play`, активируется кнопка:

```
<button class="play"
¬ title="play">&#x25BA;</button/>
```

Мы используем `&#25BA;`, геометрическую сущность XML, представленную в виде кнопки «воспроизведение». Когда мы нажимаем на кнопку, начинается воспроизведение видео, и ей приписывается новое значение, две вертикальные лини, при помощи `&#x2590;`. Наша кнопка теперь выглядит как пауза, что мы и наблюдаем на рис. 4.4.

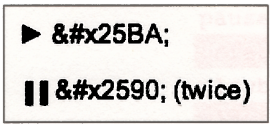


Рис. 4.4. Использование XML для кнопок паузы и воспроизведения

Для простоты я сделал кнопку при помощи метки, но так как мы продолжаем менять наши элементы управления видео, все дополнительные элементы (для воспроизведения, паузы и т. д.) должны описываться при помощи JavaScript.

К переключателю «воспроизведение/пауза» нам нужно добавить еще несколько деталей:

1. Если видео стоит на паузе, на начале воспроизведения или воспроизведение завершено, нужно сбросить текущее значение времени на 0, то есть перевести указатель воспроизведения на начало видео.
2. Изменить значение кнопки-переключателя, чтобы, когда пользователь нажмет на нее, она изменилась с паузы на воспроизведение или с воспроизведения на паузу.

3. **Наконец, мы воспроизводим видео (или ставим его на паузу):**

```
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();
}
```

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 иконку 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:

```
if (video.ended) {
    video.currentTime = 0;
}
if (video.paused) {
    video.play();
} else {
    video.pause();
}
```

```
// which could be written as: video[video.paused
¬   'play'  'pause'](};
```

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 панели 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);
```

Теперь каждый раз, когда видео проигрывается, ставится на паузу или заканчивается, запускается функция, которая ассоциируется с релевантным событием, и проверяет, правильный ли вариант кнопки отображается на панели.

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. Сейчас на подходе уже IE9, который будет поддерживать видео, кроме того, он будет поддерживать еще и стандартный `addEventListener`, так что и в этом браузере наш код будет работать.

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 понимать, какая часть видео уже была проиграна, так что бегунок на панели воспроизведения должен перемещаться в соответствующую область, как показано на рис. 4.5. И, в конце концов, нам необходимо описать событие, которое говорит о том, что можно начать просмотр, то есть у нас достаточное количество буферизованного видео для начала воспроизведения.

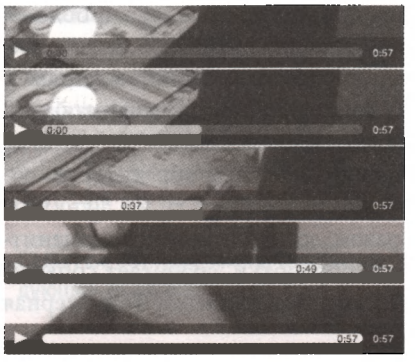


Рис. 4.5. Наш специальный прогресс-бар для видео, включая поиск по контенту и текущую позицию воспроизведения

ОТСЛЕЖИВАЕМ ХОД ЗАГРУЗКИ

У элемента медиа 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 доступный буфер — 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 видео. Именно оно является контекстом, в рамках которого будет реализовываться функция `updateSeekable`. А атрибут `duration` — это длина медиафайла в секундах.

Однако в видеоэлементе Firefox иногда возникает проблема, которая выливается в то, что значение `video.seekable.end()` оказывается не равно длительности. Или же когда медиафайл полностью загружен и обработан, полученная длительность не совпадет со значением `video.seekable.end()`. Чтобы разрешить эту проблему, мы можем обратиться к событию `durationchange`, использующему ту же функцию `updateSeekable`. Таким образом, если длительность изменится после завершения процесса обработки, сработает событие `durationchange` и у буферизованного элемента будет верная длина:

```
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()   0)) + '%'
}
```

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 будет выглядеть 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 теперь нам придется заняться наблюдением за событием `loadeddata`.

ПРИМЕЧАНИЕ

События, которые имеют отношение к загрузке, запускаются в следующем порядке: `loadstart`, `durationchange`, `loadeddata`, `progress`, `canplay`, `canplaythrough`.

Оно говорит нам о том, что есть загруженные данные, 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. Таким образом, `loadeddata` — это как раз та точка цикла событий, в которой должен быть активирован пользовательский интерфейс:

```
video.addEventListener('loadeddata'
¬ initialiseControls, false);
```

ПРЕДВАРИТЕЛЬНАЯ ЗАГРУЗКА МЕТАДАННЫХ

Недавно к элементу медиа был добавлен атрибут `preload` (он такой новый, что его пока даже браузеры не поддерживают). Этот атрибут позволяет разработчикам заставить браузеры загружать только заглавную информацию элемента медиа, которая включает метаданные. Если поддержка этого атрибута будет реализована в браузерах, нам, прежде всего, придется дожидаться срабатывания события `loadedmetadata`, а не `loadeddata`, разумеется, при условии, что мы хотим инициализировать такие элементы управления медиа, как длительность и бегунок.

Перемотка вперед, замедленное и обратное воспроизведение

В технических характеристиках присутствует 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), 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 () {
            video.currentTime += direction;
            return arguments.callee; // allows us
            ¬ to run once and setInterval
        })(), 500));
    }
}
function playnormal(video) {
    if (video.playbackRate != undefined) {
        video.playbackRate = 1;
    } else { // do it manually
        clearInterval(video.getAttribute
        ¬ ('data-playbackRate'));
    }
}
```

Как вы видите 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 %).

Возможности управления мультимедиа

Мы говорили о возможности управления элементом видео с клавиатуры, но что насчет заголовков и субтитров мультимедиа? В конце концов, для аудио или видео нет атрибута `alt`,

как для <img>. 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 по каким-либо причинам (например, в шумной обстановке или из-за того, что было необходимо сэкономить трафик), если браузеры корректно отображают медиа.

Теория доступности мультимедиа HTML5 совершенна. Автор должен предоставить подходящий файл и поместить его в контейнер файла Ogg или MP4 вместе с файлами мультимедиа, и браузер предоставит пользовательский интерфейс, при помощи которого пользователь сможет управлять заголовками или субтитрами. Даже если видео размещено на тысяче различных сайтов (при помощи использования внешнего URL как источника элемента видео/аудио), все они получат информацию о субтитрах, так что мы получаем доступность уровня «напиши один раз, используй везде».

Это теория. На практике никто не знает, как это сделать; в технических характеристиках нет об этом ни слова, браузеры ничего не делают. Но уже идут изменения. В то время как пишется эта книга (май 2010), WHATWG добавил новый элемент <track> к техническим характеристикам, что позволяет добавлять различные виды информации, такие как субтитры, заголовки, описание, названия частей и метаданные. Сообщество WHATWG разрабатывает под эту информацию новый синхронизированный текстовый формат под названием WebSRT (www.whatwg.org/specs/web-apps/current-work/multipage/video.html#websrt). Это одна из причин, по которым этого неопределенного 29-го элемента нет в версии технических характеристик W3C. Формат элемента <track> такой:

```
<track kind=captions src=captions.srt>
```

Но что можно сделать сейчас? Не существует готового решения этой проблемы, но мы предоставим один из возможных способов (правда, несколько кривой).

Брюс разработал концепт, в соответствии с которым отображаются отдельные строки расшифровки, привязанные ко времени при помощи новых атрибутов HTML5 data-*:

```
<article class=transcript lang=en>
<p><span data-begin=3 data-end=5>Hello, good
¬ evening and welcome.</span>
<span data-begin=7.35 data-end=9.25>Let's welcome
¬ Mr Last Week, singing his poptabulous hit
```

```
¬ &ldquot;If I could turnback
¬ time!&rdquot;</span>
</p>

</article>
```

АТРИБУТЫ DATA-* (НАСТРОЕННЫЕ ВРУЧНУЮ)

HTML5 позволяет применять специальные атрибуты к любому элементу. Это может быть использовано для передачи информации локальным скриптам. 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, чтобы извлечь значение, приходилось бы тратить время на сбор имен классов, таких как `shields-5` и обработку их отображения (в данном случае, выделение цветом). В своей книге PPK на JavaScript (New Riders, ISBN 0321423305) Петер-Пауль Кох объясняет, как это сделать и почему на некоторых страницах HTML 4 он решил использовать специальные атрибуты. Таким образом он делал JavaScript более стройным и простым для написания, но также провоцировал техническую невалидность страниц. Так как для передачи пар «имя/значение» скриптам гораздо проще использовать `data-shields=5`, HTML5 вводит и стандартизует эту полезную практику.

Мы используем `data-begin` и `data-end`; они также могут называться `data-start` и `data-finish`, или (в другом типе видео) `data-ooh-matron` и `data-slapandtickle`. Как и при выборе имен класса или id, имя должно быть осмысленным.

Специальные атрибуты данных предназначены только для передачи информации собственным скриптам сайта, для которых не существует более подходящих атрибутов или элементов.

В технических характеристиках сказано: «Эти атрибуты не предназначены для использования в ПО, которое независимо от сайта, на котором они определены», и, таким образом, не должны передавать информацию поисковым агентам или парсерам третьей стороны. Этой работой займутся микроформаты, микроданные или RDF.

Когда атрибуты `data-*` будут полностью поддерживаться браузером, JavaScript сможет получить доступ к свойствам при помощи `element.dataset.foo` (где атрибут `data-foo` содержит значение). Поддержка может быть эмулирована при помощи расширения объекта `HTMLElement` с помощью JavaScript, что невозможно в альфа-релизе IE9 и предыдущих версиях. Это вы можете увидеть здесь http://gist.github.com/362081. В ином случае скрипты могут получить доступ к значениям при помощи методов `get/setAttribute`. Преимущество свойства набора данных по отношению к `setAttribute` состоит в том, что ему можно приписать числовые значения, а, кроме того, когда все это будет полностью реализовано в браузерах, автоматическое определение атрибута `dataset` установит его значение по отношению к отдельному элементу. Это предоставит вам простой синтаксис для установки специальных данных.

Чтобы узнать больше, посетите сайт http://dev.w3.org/html5/spec/Overview.html#custom-data-attribute.

JavaScript используется, чтобы спрятать расшифровку `<article>`, связанную с событием `timeupdate` API видео, и перекрывает диапазоны как простой текст (стилизованный для этого при помощи CSS) над (или за) элементом видео, в зависимости от текущего времени воспроизведения видео и отметок

времени на отдельных диапазонах. Посмотрите на него в действии на http://dev.opera.com/articles/view/accessible-html5-video-withjavascripted-captions/. См. рис. 4.6.

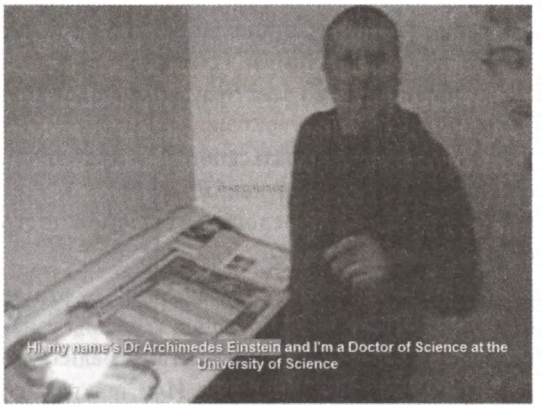


Рис. 4.6. Скрипт выводит текст поверх видео

BBC поставила похожий эксперимент на http://open.bbc.co.uk/rad/demos/html5/rdtv/episode2/. Здесь субтитры извлекаются из файла JavaScript http://open.bbc.co.uk/rad/demos/html5/rdtv/episode2/rdtv-episode2.js, что довольно близко к варианту HTML5, но здесь не возникает побочного эффекта предоставления возможности поисковым машинам проиндексировать содержание расшифровки. У Silvia Pfeiffer, контрактора Mozilla, есть несколько хороших демо использования видео HTML5 и некоторых дополнительных расширений (это не часть технических характеристик) на www.annodex.net/~silvia/itext/.

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. Видео и аудио становятся стандартными элементами, которые поддерживаются браузерами (а не загадочными плагинами), для них предлагается эффективный API, 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 подстра-

ховаться 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 это код самим. Плеер Kaltura (http://www.html5video.org/), работающий во всех браузерах, находится в свободном доступе. jPlayer (http://www.happyworm.com/jquery/jplayer/) — это аудиоплеер jQuery audio со свободной лицензией, который действует как флеш в существующих браузерах, может быть изменен при помощи CSS и предоставит возможность работы с плейлистами.

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. Или, по крайней мере, вы узнаете, как мы во всем этом разбираемся. Что может быть более удивительно, чем это?

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, формы, текст и т. п. 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). 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.

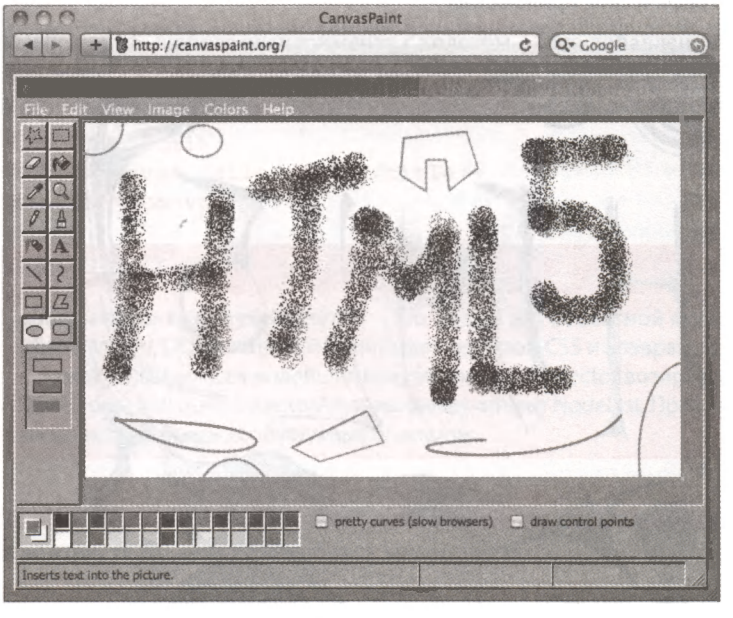


Рис. 5.1. Приложение MS Paint, воспроизведенное при помощи элемента `<canvas>`

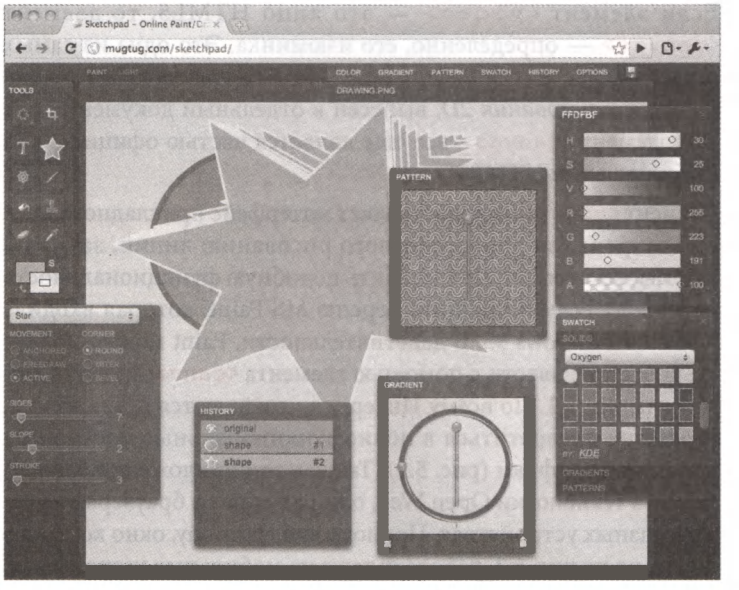


Рис. 5.2. На основе элемента `<canvas>` создаются и более сложные графические приложения

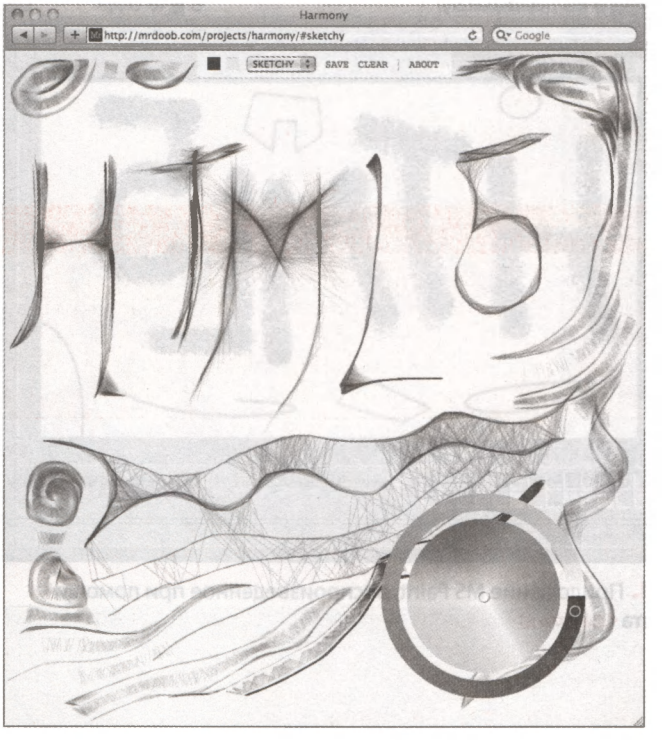


Рис. 5.3. Демонстрационная версия приложения Harmony в неизменном состоянии работает даже в мобильных браузерах

Этот API применяется для создания массы разнообразных приложений, 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!

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 экспортировать 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>
```

ПРИМЕЧАНИЕ

querySelector и querySelectorAll — это новый API объектной модели документов (Document Object Model, DOM), который принимает селектор CSS и возвращает соответствующие элементы. Доступный во всех новейших браузерах, querySelector возвращает первый найденный узел DOM, тогда как querySelectorAll возвращает объект NodeList. Пройдя по содержимому объекта, вы можете выбрать необходимые элементы.

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:

```
ctx.fillRect(10, 20, 50, 50);
```

ЧТО 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 поддерживают (возможно, это случится в версии IE9), его можно имитировать с помощью пары хитростей. Например, применить Silverlight и библиотеку 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. Теоретически, можно попытаться смешать эти трюки: переключаться на excanvas, если Silverlight недоступен. Пока что на практике я этого не встречал, но почему бы и нет? Главное — суметь распознать поддержку Silverlight.

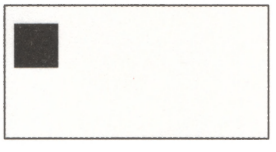


Рис. 5.4. Залитый прямоугольник, созданный с параметрами холста по умолчанию

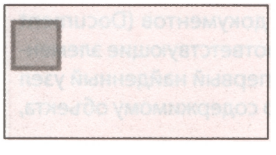


Рис. 5.5. Используем стиль заполнения и контур прямоугольника

`fillRect` принимает четыре аргумента: координаты `x` и `y`, высоту и ширину. Координаты `x` и `y` указывают на верхний левый угол 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); // контур
```

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.

РИСОВАНИЕ С ГРАДИЕНТАМИ И УЗОРАМИ

Используя объект контекста, можно создать стиль — 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 (доступные в настоящее время в WebKit

и Firefox 3.6) — 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.width);
```

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 (`x1` и `y1`) и конечная точка градиента (`x2` и `y2`). 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.width);
```

Рис. 5.6. Вертикальный градиент на элементе `<canvas>`

Вид градиента можно варьировать. 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

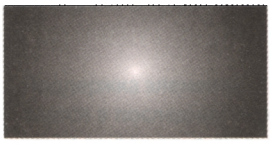


Рис. 5.7. Радиальный градиент симметричен относительно центральной точки, однако конечный радиус у него намного больше начального. В результате мы получаем круг с гладкой градиентной заливкой

находится там же, где центр начальной окружности: `canvas.width/2`, `canvas.height/2`. Вот почему мой градиент выглядит так аккуратно (рис. 5.7).

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'
```

ОТ ГРАДУСОВ К РАДИАНАМ

Все методы radius и arc 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. Строка `'repeat'` соответствует синтаксису `background-repeat` в CSS; `repeat-x`, `repeat-y` и `no-repeat` тоже будут работать в этом методе.

Рис. 5.8. Укладываем изображение на холсте с помощью метода createPattern

Помните, что при растягивании холста (как в примере) его содержимое тоже растягивается. Аналогично выглядит Flash-содержимое, если изменить размер поля. На рис. 5.9 показан тот же самый холст, что и на рис. 5.8. Я просто растянул окно браузера после того, как изображение загрузилось.

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.

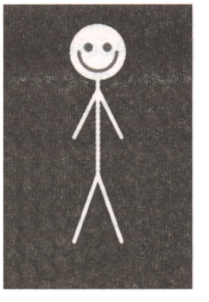


Рис. 5.10. Придуманный мной человечек нарисован с помощью API путей

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.

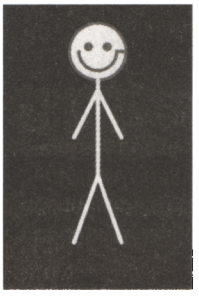


Рис. 5.11. Пример того, как неразорванный путь приводит к ошибке в рисунке

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.

```
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'
// начинаю левый глаз
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 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.

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 строит дерево, представляющее текущее состояние всех объектов на экране. 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. Кроме того, этот API хорошо поддерживается настольными инструментами, такими как Adobe Illustrator и Inkscape. Они позволяют импортировать и экспортировать рисунки SVG, и вам не приходится для создания собственных изображений сражаться с кодом XML. SVG — это API векторной графики, он намного лучше поддерживает масштабирование. В то же время изображения на холсте — растровые, они не масштабируются, их можно только сжать или растянуть.

Если вы еще не до конца уверены, хороший ли это инструмент, то 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 хорошо подходит для создания анимированных изображений и приложений, сконцентрированных вокруг JavaScript. По сравнению с SVG, это более низкоуровневый API. Так как дерева состояния холста здесь нет, данный интерфейс прикладного программирования лучше подходит для ситуаций, когда движения мыши отслеживать *не* требуется. Однако его удобно использовать в приложениях, которые управляются с клавиатуры — таких, как демонстрационные версии 8-битных игр, которых за последний год появилось великое множество. Поскольку холст — очень JavaScript-центричный интерфейс, события клавиатуры можно обрабатывать на уровне документа. Наконец, при работе с холстом вы оперируете отдельными пикселами, что прекрасно иллюстрирует пример на рис. 5.10, 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!

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.

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, преображение 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 рисованию 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.

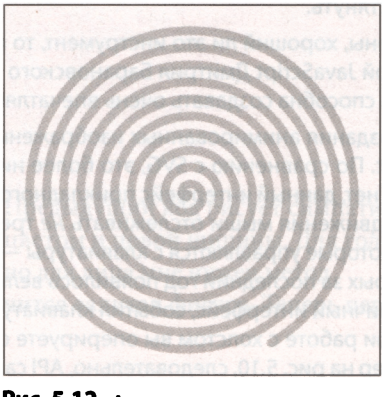


Рис. 5.12. Анимированная спираль вращается, вращается и никак не остановится

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 помещу 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 три точки, которые я нарисовал, меняя начало координат с помощью метода `translate`. Каждый раз я передавал методу указанные на рисунке координаты, и `translate` переносил начало координат в соответствии с передаваемыми ему параметрами.

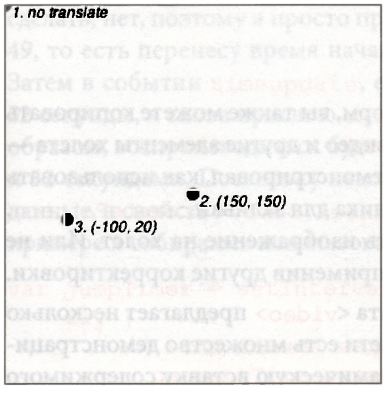


Рис. 5.13. Пример перемещения начала координат методом translate

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 `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);
```

СОВЕТ

У объекта context есть атрибут под названием canvas, включающий обратную ссылку на элемент холста, частью которого он является. С помощью него можно вернуться к холсту и узнать его высоту и ширину — удобно, если в вашем распоряжении имеется только контекст.

Единственная сложность, с которой мне приходится столкнуться, — 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.

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`, который работает с различными источниками элементов HTML, перечисленными выше. Он принимает три следующих набора аргументов:

- `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 картинку, на которой 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. 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 поддерживают, 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, то JavaScript возвращает ошибку. Если возвращается ошибка, то код не доходит до `clearInterval`, однако, если он исполняется успешно, то `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 выключу звук видеофрагмента (для того чтобы эффект не был слишком навязчивым!) и определю, 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; // thumbnail size
    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.

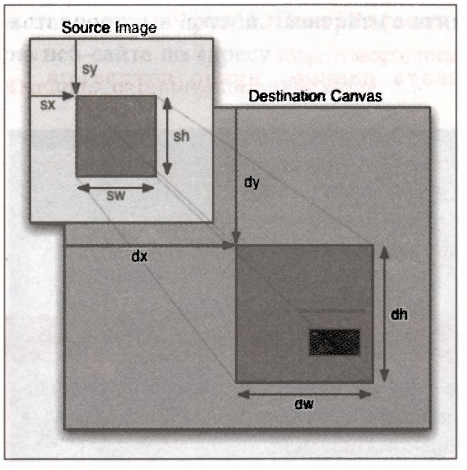


Рис. 5.14. Визуальное представление аргументов drawImage

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 пикселы. Альтернативная

технология рисования, SVG, такого не умеет. Работая с холстом, вы можете извлечь из объекта контекста 2D любой пиксел, 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 пиксел, `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 пиксела используется 4 канала.

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 со-

держимое — 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 визуального 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.

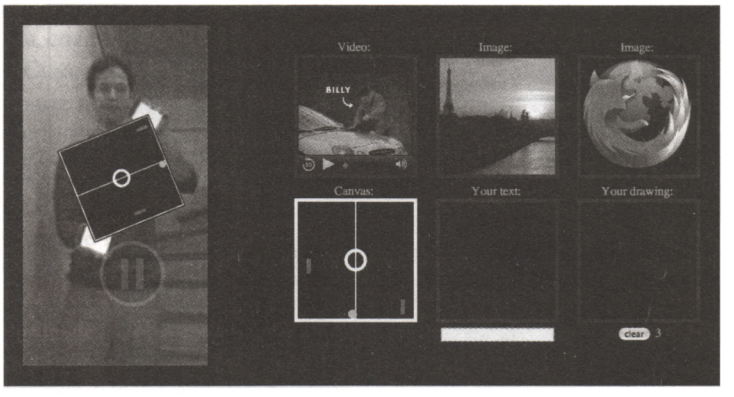


Рис. 5.16. Сканирование видео в поисках яркого пятна, поверх которого вставляется динамическое содержимое

В следующем примере кода я загружаю на холст изображение и инвертирую все пикселы, 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   = 0, n = pixels.data.length;
    ¬   < 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 считываю данные пикселов, чтобы применить к ним нужную корректировку.

В цикле `for` **я использую выражение** `i += 4`**, чтобы пройти по всем пикселам, но не по каждому каналу каждого пиксела. 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 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 экспортировать 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 экспортирование текущего состояния холста в URL данных.

ЧТО ТАКОЕ URL ДАННЫХ?

Большинство браузеров поддерживают возможность считывания ресурсов, таких как изображения, в кодировке base64. Схема URL выглядит примерно так:

```
...и т.д.
```

Адрес начинается со слова data, затем указан тип mime, кодировка — base64, а после этого идут необработанные данные. Необработанные (или сырые) данные — это то, что экспортируется 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!

Экспортировать 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. Firefox (на Mac) поддерживает только PNG, а для всех остальных типов данных возвращает 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 рисунок и сразу же сохраняем его в формате PNG:

```
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 экспортирование типа `image/jpg` с возможностью выбора уровня качества изображения. Допустимые значения аргумента — от 0.0 до 1, где единица обозначает высочайшее качество. Уровень качества влияет 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.

Анимировать изображение очень просто и делается это вручную. Если в какой-то области окна изображение должно двигаться, то вы рисуете первый кадр, очищаете область, снова рисуете, снова стираете и т. д.

БИБЛИОТЕКА JAVASCRIPT PROCESSING

Как вы вскоре увидите, перемещаться по холсту, 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.

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.

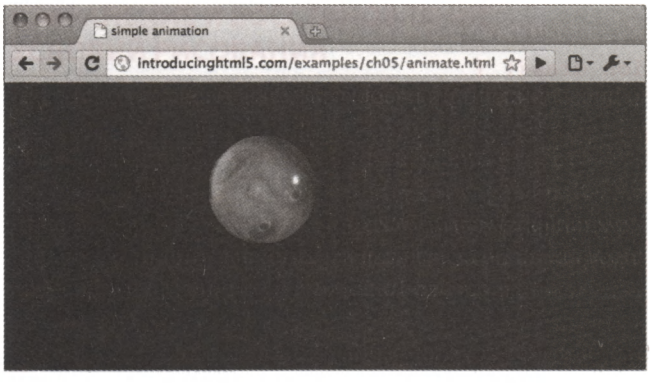


Рис. 5.18. Скрасить скучные часы ожидания вам поможет голова Брюса, скачущая по холсту и вращающаяся вокруг своей оси

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, скрываю его (потому что он не должен быть виден на странице — это всего

лишь для удобства создания эффекта) и транслирую холст, чтобы вращение происходило вокруг центра. Изучив мой код, вы видите, что я создал новый элемент `<canvas>`, но не поместил его в DOM. Я все так же могу использовать контекст 2D, но только установив высоту и ширину холста. Если не сделать этого, то по умолчанию они будут считаться равными нулю, то есть, фактически, области для рисования или для извлечения пикселов у меня не будет. Как только вы помещаете холст в DOM, ему автоматически присваивается ширина 300 пикселов и высота 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 мультик 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();
}
```

Обратите внимание, что методы save и restore не влияют на пути и 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 или только обвести их очертания. Проект Bespin (https://bespin.mozillalabs.com) — отличный пример применения пользовательской визуализации текста для создания полнофункционального редактора кода исключительно с использованием возможностей API холста.

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 демонстрировал, куда переносится новая центральная точка холста, и подписывал 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
```

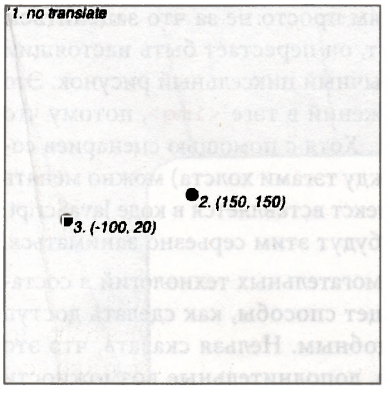


Рис. 5.19. Подписываем рисунки на холсте с помощью fillText

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'`). 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. 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.

Оборотная 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 тэге `<img>`, потому что там хотя бы есть атрибут `alt`. Хотя с помощью сценариев содержимое элемента (текст между тэгами холста) можно менять в зависимости от того, какой текст вставляется в коде JavaScript, не думаю, что разработчики будут этим серьезно заниматься.

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. Этот модуль с помощью jQuery вставляет элемент холста на страницу с информацией из размеченной таблицы данных. Вспомогательные технологии могут обращаться к необработанным данным в таблице, а пользователи с хорошим зрением помимо текстовых данных видят также удобные для восприятия графики.

Заключение

Благодаря API холста разработчики наконец-то получили способ рисовать 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 подходит 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.

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.

Прежде, 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, предназначенной 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-Пола 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, то 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 печенюшки 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.

Варианты хранения

Что касается хранения данных на клиентской стороне, существует два варианта:

- веб-хранилище (Web Storage), которое поддерживается во всех современных браузерах;
- базы данных SQL для веб (Web SQL Database), которые поддерживаются браузерами Opera, Chrome и Safari.

Из названия последних сразу становится понятно, как это работает: браузер обращается к локальной базе данных при помощи SQL-запросов.

Веб-хранилище — это куда более простая система, в которой различные значения просто связываются с уникальными ключами. Знание языка запросов SQL не требуется. Вообще, поддержка API веб-хранилища реализована намного лучше, чем API баз данных SQL для веб. Я рассмотрю оба этих 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, например, в Safari при превышении этого лимита выводится 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 любом случае, если API не удается записать данные, браузер возвращает ошибку. Я буду рассматривать небольшие при-

ложения, в которых объем данных в хранилище не превышает 100 Кбайт.

ВЕБ-ХРАНИЛИЩЕ

В двух словах, API веб-хранилища — это файлы cookie на стероидах. Ключевое преимущество данного API — логичное и понятное разделение сеансовых и долговременных данных. Это правильно, потому что сеанс должен длиться до закрытия окна, а не браузера целиком. В противоположность этому, 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. 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, то оно не сможет обращаться к данным первого сеанса. Благодаря этому вы можете, например, ограничить сеанс покупки в интернет-магазине одним окном. В случае с файлами cookie сеанс может «просочиться» из одного окна в другое, добавив случайную ненужную транзакцию.

Хранилище типа `localStorage` привязано к конкретному домену 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.

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 any 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 в новейших браузерах метод `getItem` не возвращает данные типа `any`. Независимо от содержимого, браузеры преобразуют тип данных в строковый. Это важно, потому что при попытке сохранить объект в действительности сохраняется [Object object]. Что еще важнее, в строки преобра-

зуются 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);
```

ПРИМЕЧАНИЕ

Хотя в спецификации говорится, что можно сохранять «любые данные», в действительности это означает, что можно сохранять любые данные типа DOMString.

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.

С учетом этого, на самом деле спецификации API должна выглядеть так:

```
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 DOMString
¬ data);
deleter void removeItem(in DOMString key);
void clear();
```

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

Как мы уже видели, для записи и извлечения данных можно использовать `setItem` и `getItem`. Однако существует еще несколько методов доступа и манипулирования данными в объектах хранилища.

Использование синтаксиса expando

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.

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, так:

```
sessionStorage.setItem('key',
¬ '27152949302e3bd0d681a6f0548912b9');
```

Теперь 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 `expando`, а для получения значения можно использовать `getItem('key')`.

Удаление данных

Существует два способа программного удаления данных из объекта хранилища: с помощью `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('bruce', "считает себя
¬ профессором синергизма");
sessionStorage.setItem('remy'  "считает Брюса
¬ спятившим профессором синергизма");
alert( sessionStorage.length ); // результат — 2
sessionStorage.removeItem('bruce');
```

```
alert( sessionStorage.length ); // результат — 1
sessionStorage.clear();
alert( sessionStorage.length ); // результат — 0
```

Как сохранять не только строки

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!

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, но в действительности это всего лишь оболочка, позволяющая обойти неудобство. Все зависит от того, начнут ли браузеры когда-либо поддерживать хранилища типа `any`. Это легко проверяется, но приходится добавлять оболочки для методов `set` и `get`, что, в зависимости от вашего приложения, может быть нежелательно.

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 http://www.json.org/json2.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
```

```
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, то 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 содержимого `sessionStorage` и `localStorage`, больше выбирать, в общем-то, не из чего.

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. Здесь можно изменять ключи и значения, а также удалять записи целиком.

ПРИМЕЧАНИЕ

Для того чтобы включить в Safari отображение меню Developer (Разработка), откройте меню Preferences (Настройки) и установите флажок Show Developer Menu (Показывать меню разработчика) в поле Menu Bar (Полоса меню) на вкладке Advanced (Дополнительно).

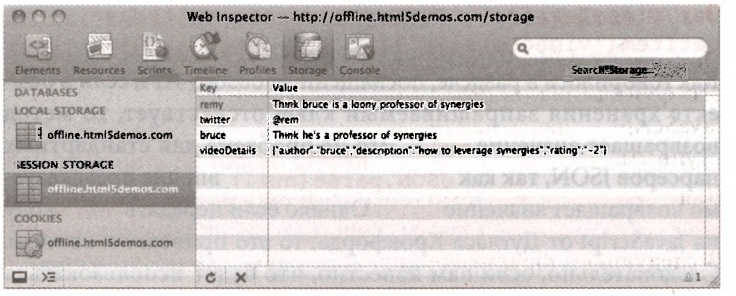


Рис. 6.1. Отладчик хранилища WebKit

Firebug в Firefox

С помощью плагина Firebug можно с легкостью заглянуть в объекты хранилища браузера Firefox. Укажите в команде консоли `sessionStorage` или `localStorage`, выполните код и сможете щелкнуть объект хранилища и просмотреть его содержимое (рис. 6.2).

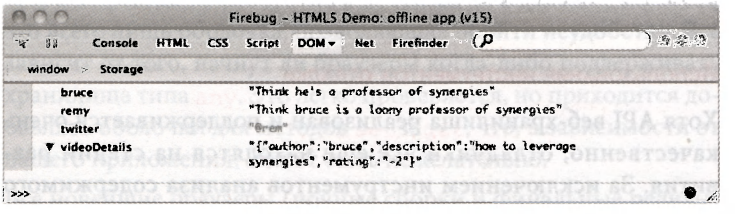


Рис. 6.2. Инструмент анализа содержимого хранилища Firebug

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).

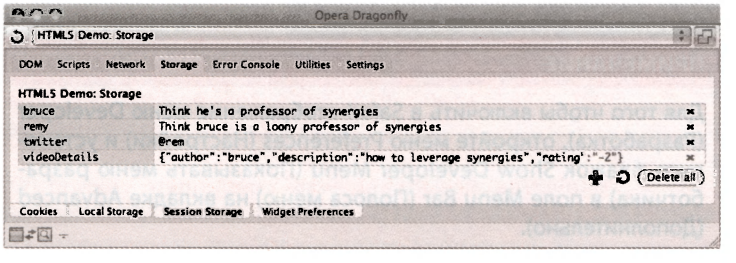


Рис. 6.3. Отладчик DragonFly в браузере Opera позволяет просматривать содержимое хранилища

Обходные варианты

Поскольку API хранилища устроен относительно просто, его функциональность можно воспроизвести с помощью JavaScript — на случай, если API хранилища окажется недоступным.

Для `localStorage` можно использовать файлы cookie, а для `sessionStorage` — трюк со свойством имени 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, однако 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+''  // заносит
                ¬ значение в строку
```

```
                window.top.name =
                ¬ JSON.stringify(data);
            }
        };
    })();
}
```

Проблема с реализацией `sessionStorage` **вручную (как показано в предыдущем фрагменте) заключается в том, что невозможно будет использовать, например, такой код:** `sessionStorage.twitter = '@rem'`. **Хотя технически он сработает, он не сможет правильно зарегистрировать объект в хранилище, и** `sessionStorage.getItem('twitter')` **не вернет результата.**

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, будете ли вы использовать `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 будете чувствовать себя как рыба в воде. С другой стороны, если вы ничего не знаете о языке запросов SQL и, в частности, об SQLite, то 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.

> **ПРИМЕЧАНИЕ**
>
> Mozilla и Microsoft сомневаются насчет поддержки баз данных SQL. Mozilla смотрит в сторону спецификации под названием Indexed Database API — API индексированных баз данных, — которая нигде пока что не реализована, поэтому в этой книге не рассматривается. Тем не менее стоит взять ее на заметку, на случай, если она получит популярность в будущем.

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, то 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, то 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, то 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, 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, то 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.

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 × 1024 × 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 создать таблицу. 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, то 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, соответствующий 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; в целом, 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, оборачивая каждый оператор INSERT, вы гарантируете, что будут сохранены все новые сообщения, независимо от того, есть они уже в базе данных или нет.

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]);
            // деление на 1000 — чтобы добраться
            ¬ до секунд
        });
    });
}
```

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 пара 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 `<ul>`) `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 содержится 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 `<ul>` для HTML-кода. В нашем примере элемент `<ul>` 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> Выберите диапазон создания
        ¬ последних сообщений об HTML5 в
        ¬ Twitter</legend>
        <input type="radio" value="5" id="t5m"
        ¬ name="timerange"/><label for="t5m">
        ¬ 5 минут</label>
        <input type="radio" value="30" id="t30m"
        ¬ name="timerange" /><label for="t30m">
        ¬ 30 минут</label>
        <input type="radio" value="120" id="t2h"
        ¬ name="timerange" /><label for="t2h">
        ¬ 2 часа</label>
        <input type="radio" value="all" id="tall"
        ¬ name="timerange" checked="checked"
        ¬ /><label for="tall">за все
        ¬ время</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 недоступен в этом браузере.
```

```
        ¬ Попробуйте использовать ночную сборку
        ¬ Opera, Webkit или 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);
}

// наша 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]);// делим на 1000,
            ¬ чтобы добраться до секунд
        });
    });
}

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('');
            });
        });
}

// привязываем обработчики щелчков
¬ к переключателям
[].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. 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');

    // это ошибочный код; должен произойти откат
    ¬ оператора DROP
    tx.executeSql('INSERT INTO foo (id, text)
    ¬ VALUES (1, "foobar")');
}, function (error) {
    // error.message — "no such table: foo"
    alert('Выполнен откат, таблица "foo"
    ¬ не удалена. Причина: ' + error.message);
});

db.transaction(function (tx) {
    tx.executeSql('SELECT * FROM foo'  [],
    ¬ function (tx, results) {
        alert('найдено ' + results.rows.length +
        ¬ ' строк');
    }, function (tx, error) {
        // это исполняться не будет никогда
        alert('что-то пошло не так:  +
        ¬ 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.

Заключение

В этой главе вы узнали о двух API для локального хранения данных браузера, оба из которых с легкостью затыкают за пояс файлы cookie (а учитывая API индексированных баз данных, в процессе разработки находится еще больше спецификаций хранения).

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). Что это значит? А то, что зачерствевшее печенье можно выбросить на свалку истории уже сегодня!

Глава 7

Автономная работа

Реми Шарп

Знаете, сколько раз я работал, находясь в пути, отчаянно пытался подключиться к 3G-сети через дешевый USB-модем, и 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.

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.manifest">
<!-- здесь находится мое шикарное приложение под
названием time -->
</html>
```

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.

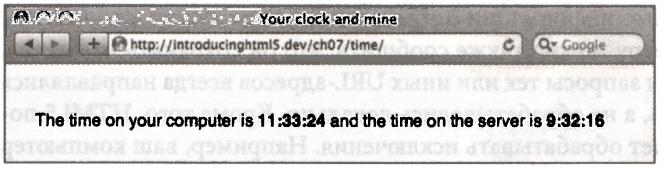


Рис. 7.1. Посторонитесь, приложения Google! Наше приложение умеет показывать время!

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.

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

# 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

# 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 тэге <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, то 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`.

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, то 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), то 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). 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.

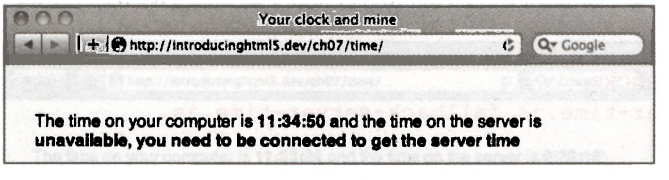


Рис. 7.2. Мое приложение time работает и в автономном режиме, но в этом случае для обработки серверного времени обращается к другому ресурсу

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 использование кэшированого ресурса. В случае манифеста, для того чтобы ресурсы загрузились заново, можно поменять его содержимое.

Это не обязательно должен быть номер или другое специальное содержимое. Однако для того, чтобы приказать браузеру заново загрузить содержимое манифеста, необходимо изменить текст манифеста, а не только время его последней модификации. Я использую номер версии, но это может быть идентификатор ревизии из вашей системы контроля ревизий, может быть отметка времени или другие уникальные данные, которые меняются, когда возникает необходимость заново загрузить ресурсы.

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`, но в моем примере она не требуется. Категория `NETWORK` представляет собой «белый список» и сообщает браузеру, что все запросы к файлам или путям в данной категории должны обслуживаться через сеть, а не из кэша. Например,

```
NETWORK:
live-time.php
/realtime-stuff/
```

говорит браузеру, что подключение к сети необходимо не только для доступа к live-time.php, но также для любых файлов, соответствующих абсолютному пути http://mysite.com/real-timestuff/. Аналогично, с помощью категории `NETWORK` можно указать, что любые ресурсы, не отнесенные явно к кэшу, должны обслуживаться через сетевое подключение. Используйте для этого груп-

повой символ * (обратите внимание, что это и есть поведение по умолчанию):

```
CACHE MANIFEST
index.html
time.js
time.css

FALLBACK:
server-time.js fallback-server-time.js

NETWORK:
*

# version 10
```

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 .manifest 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 manifest
```

Это нужно для того, чтобы веб-сервер Apache отправлял заголовок файла `text/cache-manifest` при запросе любого файла с расширением .manifest. Для проверки просмотрите заголовок запрошенного файла с помощью инструмента, подобного `curl`, и параметра `-I`:

```
curl -I http://mysite.com/time.manifest
```

Результат должен быть приблизительно таким:

```
HTTP/1.1 200 OK
Date: Thu, 04 Mar 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: Mon, 01 Mar 2010 16:20:24 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 манифеста 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 манифеста;
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 манифеста;
7) 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 манифеста с сервера;
5) сервер: возвращает код 304, сообщая браузеру, что манифест не менялся.

БЕРЕГИСЬ ИЗВОРОТЛИВЫХ ЛИС!

Firefox хвалится поддержкой автономных приложений, но в действительности они работают неправильно, а этот браузер всегда был полон ошибок. Если вы тестируете свои приложения в Firefox, постоянно проверяйте в журналах, что на самом деле запрашивается у вашего сервера. В частности, когда браузер встречает упоминание о манифесте кэша, он должен запрашивать его. В настоящее время Firefox 3.6 не запрашивает манифест, то есть ваша версия кода не обновляется, а это плохо! Но надежда есть! Можно приказать браузеру никогда не кэшировать файл манифеста. Для этого добавьте следующий код в конфигурацию сервера или файл .htaccess (или подобный):

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 .manifest можно отправлять пользовательские заголовки и заголовок no-cache:

```
Header set Pragma "no-cache"
```

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 манифеста. Если манифест изменился, то 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.

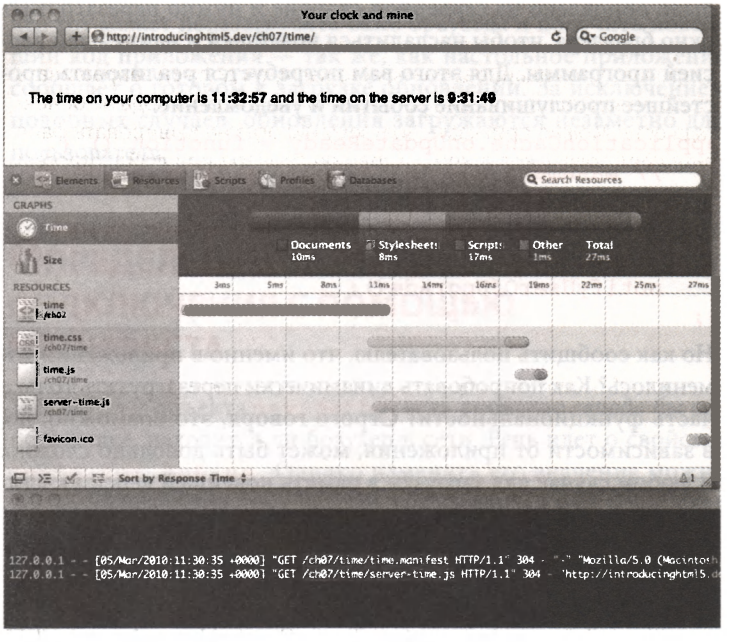


Рис. 7.3. Safari загружает ресурсы приложения из локального кэша, запрашивая у сервера только манифест и server-time

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`.

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
    notifyUserOfUpgrade();
};
```

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), то 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.

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 изменять его значение можно только через меню приложения, устанавливая или сбрасывая флажок «Работать автономно» (Work Offline в Opera, IE и Firefox). Согласен, это не самый удачный способ переключать режимы. Для определения, есть ли у браузера подключение к сети, 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 содержит:

```
setOnline(true);
```

offline.js содержит:

```
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, то исполняется код `setOnline(true)`. Если же браузер работает в автономном режиме, то 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 прикипают 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?

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).

ПРИМЕЧАНИЕ

Очищая кэш, убедитесь, что не осталось открытых окон с вашим приложением, использующим манифест.

В браузерах на основе WebKit кэш нужно очищать полностью. Под словом «кэш» я имею в виду всю информацию, которую браузер сохранил с целью ускорить работу в Интернете. Под «WebKit» я подразумеваю настольную и мобильную версии Safari и Chrome.

WebKit 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 Firefox нужно открыть меню настроек, перейти на вкладку Advanced (Дополнительно), а затем выбрать вкладку Сеть (Network). Здесь можно очистить кэш для отдельных доменов.

Заключение

В прошлом веб-сайты полностью зависели от подключения к Интернету. Сегодня браузеры начинают постепенно поддерживать возможность автономной работы, и 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 задокументировал 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 восклицать «Что за чертовщина?!», пробираясь сквозь лабиринты кроличьей норы, которую по ошибки назвали API drag and drop. Так почему мы вообще решили включить эту главу в книгу? Потому что 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 подготавливать 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).

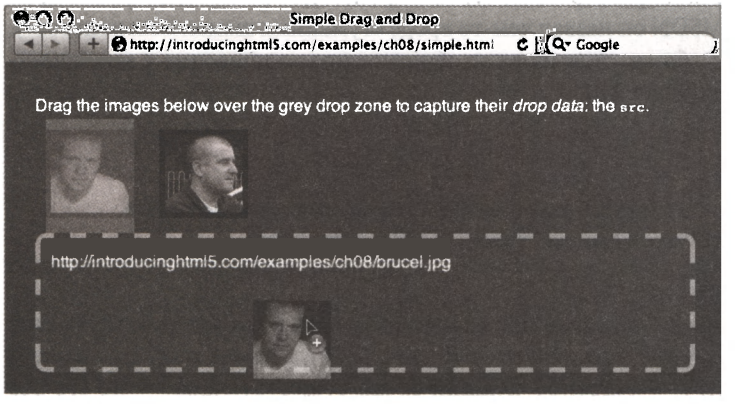


Рис. 8.1. По умолчанию перетаскивать можно все ссылки и изображения. Но добавив еще немного кода, вы позволите пользователям не только перетаскивать элементы по экрану, но и помещать их в специально предназначенную для этого область

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 перетащенные данные;
2) on drop: когда на этот элемент попадают какие-то данные, с ними нужно что-то делать.

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>Простая демонстрация
¬ перетаскивания</title>
<style>#drop { height: 100px; border: 5px solid
¬ #ccc; }
</style>
<img src="http://img.tweetimag.es/i/rem"
¬ alt="@rem" />
<img src="http://img.tweetimag.es/i/brucel"
¬ alt="@brucel" />
<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 соответствующее событие, 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. Для ссылок текст — это данные из тега `href`, а для простого текста, очевидно, это выделенные и перетащенные символы. (По умолчанию при запросе текстового типа данных IE возвращает `null`, но чуть позже мы это исправим.)

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, `oneventname`), 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 перетащенный 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, то 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, то 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 всплывало новое окно. Обычно это обрабатывается в `return false` (во всяком случае, в других браузерах), но Internet Explorer необходима небольшая помощь.

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 считываются, 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.

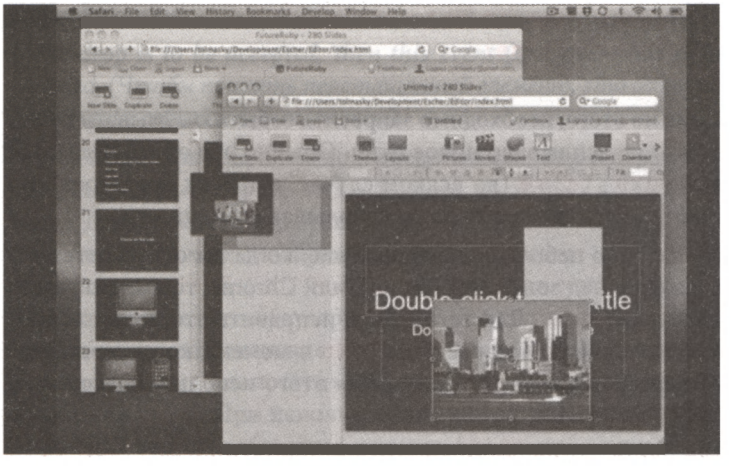


Рис. 8.2. Один из первых примеров возможностей перетаскивания реализован в приложении 280 Slides

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 из Chrome или Safari (рис. 8.3):

```
img.ondragstart = function (event) {
    event = event || window.event;
```

```
    // сейчас 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);
};
```

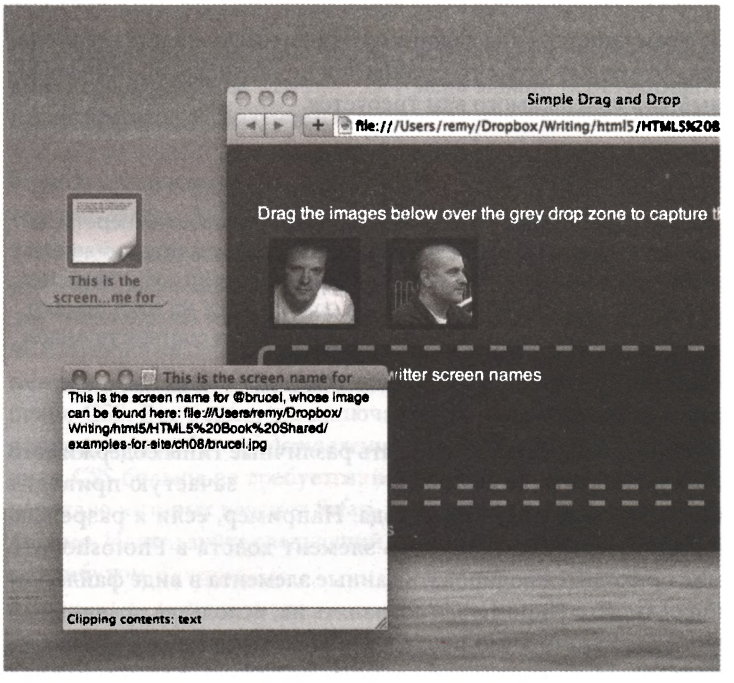


Рис. 8.3. При перетаскивании содержимого с веб-страницы на рабочем столе создается текстовый фрагмент

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** http://json.org/json2.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, то 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. Если речь идет о простом приложении, то 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.

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.tweetimag.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).

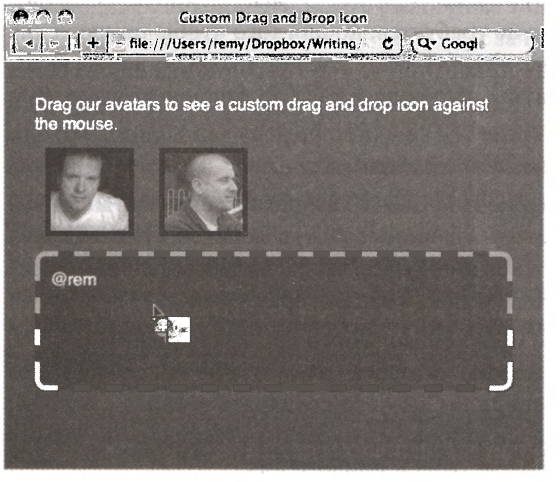


Рис. 8.4. Мы создали собственный указатель с логотипом Twitter для перетаскивания данных, связанных с Twitter

Доступность

Если вы успешно пробрались сквозь все сложности, разветвления и тупики этой спецификации, то, возможно, перед вашим приложением с возможностью перетаскивания встала про-

блема 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 dataTranfer.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 захвата.

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. К сожалению, от родной функциональности перетаскивания придется отказаться. 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.

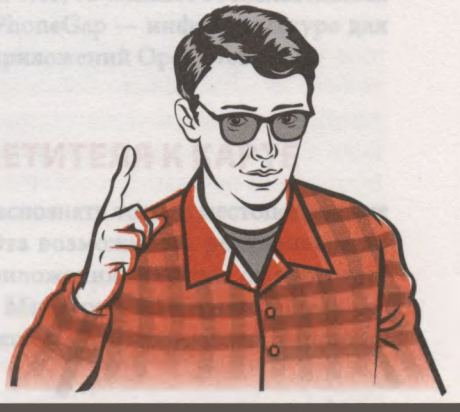
Тем не менее родные возможности drag and drop в сочетании с новейшими API, такими как API File (не рассматривается в этой книге, но это API, позволяющий браузеру считывать файлы без участия серверной стороны), дают пользователям возможность перетаскивать файлы прямо в браузер. Подобные функции можно встретить в различных новых приложениях, например Gmail. Пользователи, браузеры которых поддерживают самые свежие возможности API drag and drop, уже сегодня могут наслаждаться современными технологиями.

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



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, Firefox 3.5+ — это единственный настольный браузер, поддерживающий API геолокации, но команда Chrome уже работает над тем, чтобы включить его в свой браузер, так что можно надеяться на скорый результат. Несмотря на то что среди настольных браузеров поддержка этого API не так распространена, многие новые мобильные телефоны и браузеры поддерживают геолокацию — в частности, через мобильную версию WebKit. Интересно, что настольная версия WebKit этот API не поддерживает; хотя я полагаю, что это связано исключительно с внедрением в мобильные телефоны функций GPS. Fennec, мобильный браузер Mozilla, также поддерживает геолокацию. Помимо этого, 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. Функциональ-

ность `watchPosition` совпадает с возможностями `getCurrentPosition`, за одним исключением — если местоположение пользователя меняется, данный метод сообщает об этом вашему коду.

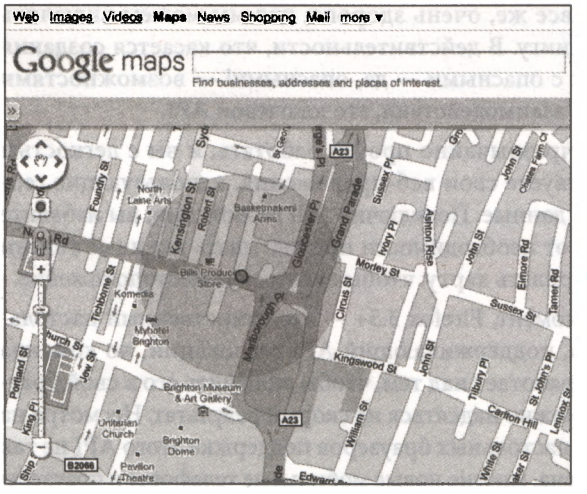


Рис. 9.1. Google Maps распознает поддержку геолокации и добавляет функции обнаружения местоположения

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. При использовании любого из этих методов большинство устройств спрашивают у пользователя, согласен ли тот сообщить свое местоположение. Если пользователь не согласен и вы запрограммировали в API геолокации пересылку сообщений об ошибке определенной функции, то эта функция вызывается, и ей передаются сведения об ошибке.

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 блокиру-

ющее окно, в котором пользователь может выбрать, сообщать о своем местоположении или нет (рис. 9.2). Ваше приложение в это время продолжает выполняться.

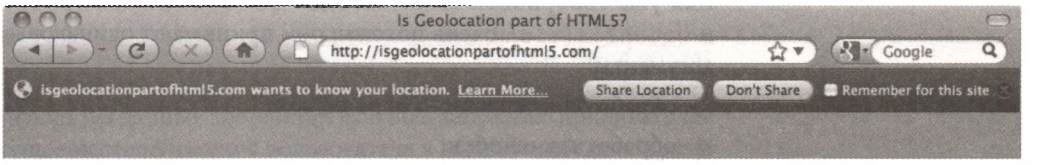


Рис. 9.2. Приложение просит Firefox сообщить ему о местоположении пользователя

В других браузерах, таких как Mobile Safari, используется модальное диалоговое окно. Это означает, что до того, как пользователь даст ответ, исполнение кода приостанавливается (рис. 9.3).

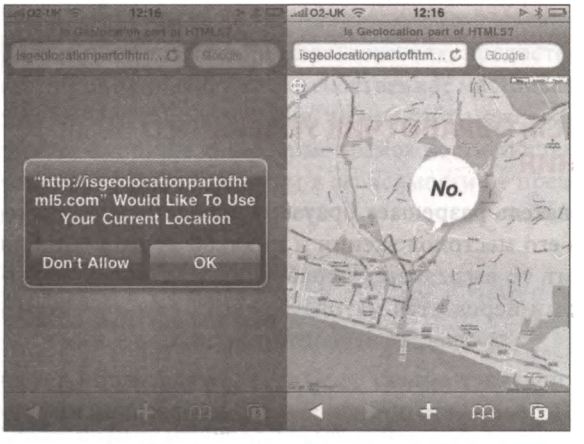


Рис. 9.3. Окно браузера Mobile Safari и модальное диалоговое окно, блокирующее исполнение приложения

Методы API

API геолокации находится внутри объекта навигатора и содержит всего три метода:

- `getCurrentPosition`
- `watchPosition`
- `clearWatch`

Методы `watchPosition` и `clearWatch` — парные. Методы `watchPosition` и `clearWatch` работают по тому же принципу, что и `setInterval` и `setTimeout`. `watchPosition` возвра-

щает 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.

Данные второго класса объекта координат поддерживаются, но с ними пока что не связываются никакие значения. Эти атри-

буты равны нулю, значению `null` или `NaN` во всех браузерах с поддержкой геолокации:

- `readonly attribute double altitude`
- `readonly attribute double altitudeAccuracy`
- `readonly attribute double heading`
- `readonly attribute double speed`

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 предлагать посетителям другие интересные возможности, постепенно обновляя содержимое страницы в зависимости от предоставленной информации.

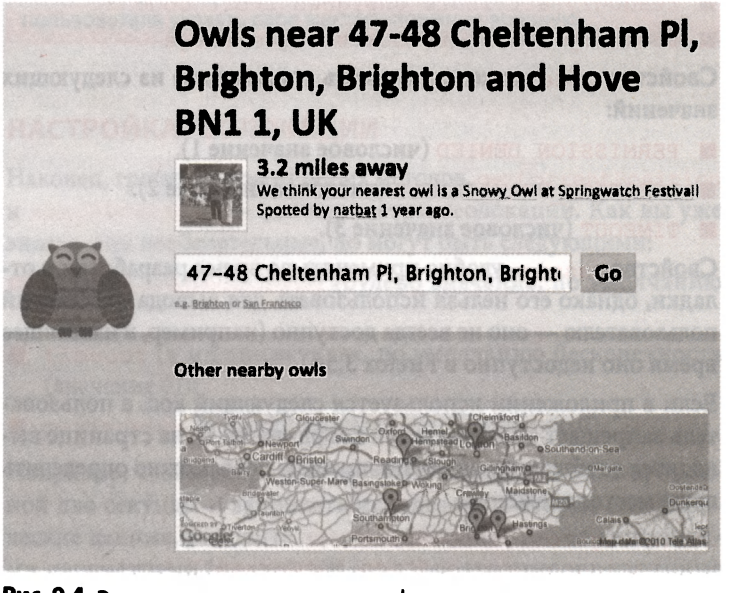


Рис. 9.4. Это приложение выводит информацию с учетом географических данных пользователей

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, то вы можете сами указать свое местоположение.

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.

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 — оно не всегда доступно (например, в настоящее время оно недоступно в Firefox 3.5+).

Если в приложении используется следующий код, а пользователь запрещает определять его координаты, то на странице выводится сообщение: «Доступ запрещен: невозможно определить ваше местоположение».

```
if (navigator.geolocation) {
    navigator.geolocation.getCurrentPosition
    ¬ (function (position) {
```

```
        var coords = position.coords;
        showMap(coords.latitude,
        ¬ coords.longitude, coords.accuracy);
    }, function (error) {
        var errorTypes = {
            1: 'Доступ запрещен'
            2: 'Координаты недоступны',
            3: 'Время ожидания истекло'
        };

        alert(errorTypes[error.code] +
        ¬ невозможно определить ваше
          местоположение");
    })
}
```

АЛЬТЕРНАТИВНАЯ ОШИБКА: УСПЕХ

У обработчика ошибок есть альтернатива: проверка атрибута accuracy. Однажды я зашел на вебсайт, 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 журнала консоли Firebug 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, полученные автоматически, и попросить пользователя указать свое местоположение вручную.

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
});
```

Параметр `enableHighAccuracy` заставляет устройство пытаться получить более точные значения широты и долготы. На мобильном устройстве это может быть необходимо для работы GPS, однако влечет за собой больший расход питания. Чаще всего в целях экономии заряда батареи используется значение параметра `enableHighAccuracy`, по умолчанию — `false`.

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 местоположение). Когда допустимое время проходит, устанавливается код ошибки 3 (`TIMEOUT`). 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, которая, как мы видели, входит во второй класс свойств объекта координат. Пока что эти данные в браузерах не поддерживаются, но мне кажется, что с учетом быстрого развития рынка мобильных устройств, скоро мы сможем работать и с подобными координатами.

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

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.

Современные браузеры обладают той же функциональностью, и это один из простейших API на сегодняшний день. Как вы уви-

дели, если вашему веб-сайту нужно получать от пользователя географическую информацию, реализовать это легче легкого. Более того, вы можете динамически обновлять содержимое страницы в зависимости от получаемых данных. Появление веб-сайтов, которые в зависимости от текущего времени и вашего местоположения будут сразу сообщать адрес ближайшей забегаловки или рюмочной — это лишь вопрос времени.

Глава 10

Сообщения, фоновые вычисления и сокеты

Реми Шарп

Web Messaging, Web Workers и Web Sockets — 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 все три 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, обеспечения должной 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.

Наконец, сокеты позволяют создавать прямое соединение с сервером (соединение по протоколу TCP), что означает двусторонний обмен данными между сервером и клиентом в режиме реального времени. Типичное базовое приложение на основе сокетов — это чат, однако возможности сокетов безграничны. Используя сокеты, можно почти полностью отказаться от кода на базе Comet. В Comet применяется множество трюков для того, чтобы передача данных с сервера осуществлялась в режиме реального времени. Веб-сокеты значительно упрощают этот процесс на клиентской стороне, как мы увидим далее в этой главе.

Крошечный чатик с помощью API Messaging

В первую очередь я хочу познакомить вас с API сообщений, так как оба следующих API, фоновые вычисления и веб-сокеты, 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 виджет, который находится в плавающем фрейме (iframe), это можно сделать с помощью API Messaging. И даже если виджет 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('любимый инструмент?'
¬ 'http://brucelawson.co.uk');
```

Аргумент целевого источника, который передается `postMessage`, 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, то 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 предыдущем коде используется метод addEventListener, а не onmessage, так как Firefox не реагирует 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 == 'любимый инструмент?') {
        if (event.origin ==
        ¬ 'http://remysharp.com') {
            event.source.postMessage('новый
            ¬ кларнет', event.origin);
        } else if (event.origin ==
        ¬ 'http://brucelaweson.co.uk') {
            event.source.postMessage('ржавый
            ¬ тромбон'  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 но содержимом данного свойства, я решаю, 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 следующих разделах, посвященных фоновым вычислениям и веб-сокетам.

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. 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.

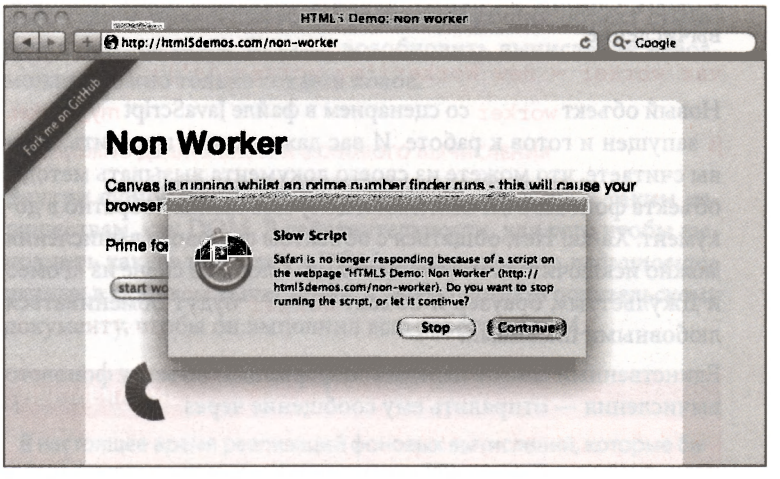


Рис. 10.1. Некоторые браузеры прерывают исполнение кода JavaScript, давая вам возможность утихомирить сорвавшуюся с катушек программу

Намного больше беспокойства вызывают неочевидные ситуации. Предположим, вы разрабатываете новейшее и крутейшее веб-приложение, предлагающее пользователям исключительные 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.

Для решения подобной проблемы фоновое вычисление подойдет просто идеально. Если убрать обработку фильтра в фон, то тогда главное окно браузера 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 (в настоящее время фоновые вычисления можно использовать в Safari, Chrome и Firefox), 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('Привет, worker!');
```

Единственный способ получить информацию от объекта фонового вычисления — применить обработчик событий `onmessage`:

```
worker.onmessage = function (event) {
    alert('Объект worker только что прислал мне
    ¬ это: ' + 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.

ПРИМЕЧАНИЕ

Небольшая хитрость: предполагается, что API фоновых вычислений должен уметь отправлять не только строки, но и объекты — так же, как API сообщений, — но большинство браузеров, тем не менее, превращают объекты в строки. Однако в Firefox реализовано внутреннее преобразование JSON.stringify/JSON.parse для отправки и получения сообщений от объектов фоновых вычислений, чтобы вам не приходилось заниматься этим самостоятельно. Можно надеяться, что хорошему примеру последуют и разработчики других браузеров, и однажды мы сможем отправлять объекты через систему пересылки сообщений.

Аналогично, код *внутри* объекта фонового вычисления также должен полагаться на сочетание `postMessage`/`onmessage`. Однако доступ к объекту ограничен намного сильнее, чем к обычному документу: только несколько избранных API и функций могут обращаться к содержимому фонового вычисления.

Единственный метод, доступный через объект фонового вычисления, — это `terminate`, который делает в точности то, о чем 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 SQL для веб, не существует, однако работа в этом направлении идет.

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 последнем разделе этой главы);
- работать с базами данных SQL для веб;

- 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:

```
onmessage = function (event) {
    if (event.data == "Привет, worker!") {
        postMessage("Привет, чем могу помочь?");
    } else {
        postMessage("Не видишь, что я занят?
        ¬ Оставь меня в покое!");
    }
};
```

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 будет 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 нажатия на клавишу мыши, перемещения мыши и отпускания клавиши мыши, 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.

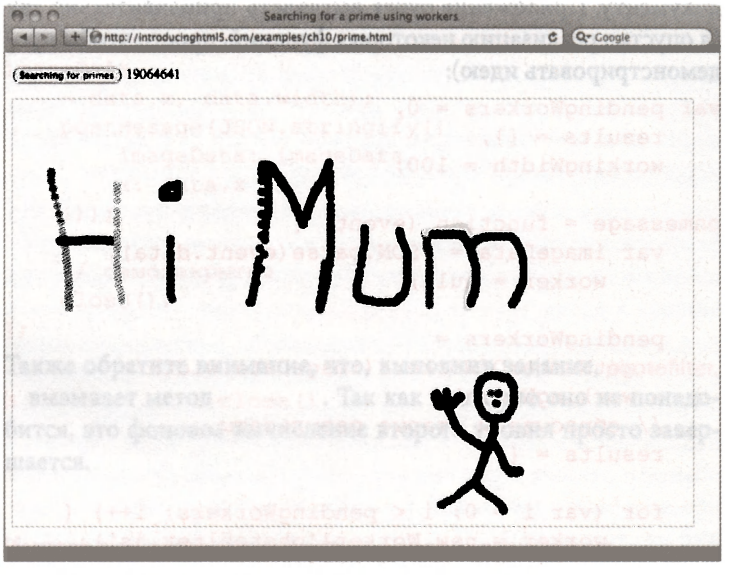


Рис. 10.2. Все, о чем вы когда-либо мечтали: одновременный поиск простых чисел и рисование на холсте. Спасибо фоновым вычислениям!

Матрешки: фоновые вычисления внутри фоновых вычислений

Если вы достаточно наблюдательны, то заметили, что существует возможность создавать фоновые вычисления внутри существующих фоновых вычислений. В настоящее время ее

поддерживает только Firefox, но она 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   = 0;   < 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,
           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, cname 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 и WebKit (но не в Safari). 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 в базе данных SQL для веб на клиентской стороне (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 объект 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 происходит 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 полная функция Ajax, 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.

Для того чтобы отлаживать фоновое вычисления, вам потребуется создать собственный метод.

Так как вы все равно будете обмениваться сообщениями с родительским документом, можно создать какую-нибудь систему для публикации специальных сообщений, предназначенных для журнала консоли. Хотя возникает сложность — помимо

системы, умеющей обрабатывать строки, вы должны будете придумать некий согласованный язык, который смогут использовать фоновые вычисления и главный документ. Данный язык 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(), // захватываем
        ¬ первое слово
        msg = data.join(' '); // снова собираем
        ¬ сообщение

    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 у меня появилась возможность проверять данные из фонового вычисления, захватывая его сообщения с помощью функции 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`, документа, 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. 10.3).

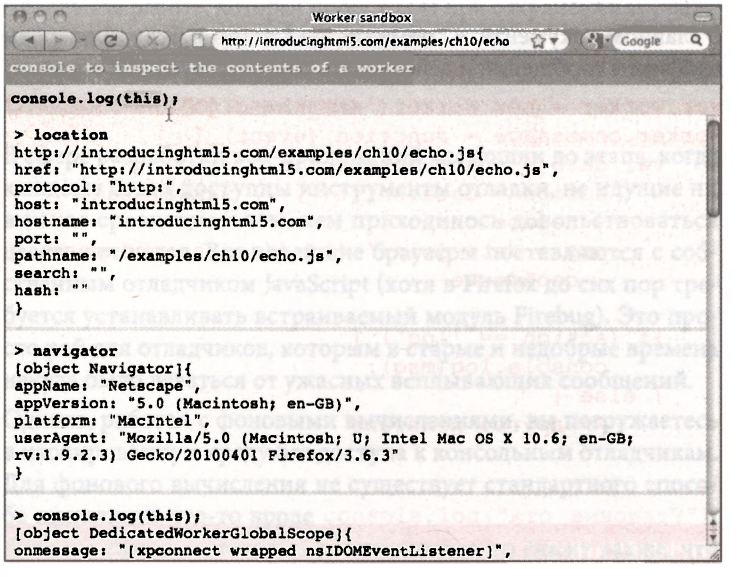


Рис. 10.3. Демонстрационная консоль для изучения содержимого фонового вычисления

Веб-сокеты: работа с потоковыми данными

Web Sockets — это один из новейших API. Он лежит за пределами спецификации HTML5, но его возможности весьма удобны 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 корректировочный блок для веб-сокетов на Flash. Адрес его сайта — https://github.com/gimite/web-socket-js.

Если сокет открыт, то 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 предлагают 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`.

В теории, если в Google Wave к одному и тому же документу подключено много пользователей и все они что-то вводят, то каждый хочет, чтобы остальные видели напечатанное им моментально, безо всяких задержек. Однако если пересылать 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, поддерживаемый 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 пересылает всем подключенным 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 рассылаются 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, 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 Web Socket. Используя серверы, подобные Node.js, я настраивал веб-сокеты 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 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. 10.4).

Рис. 10.4. 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);
};
```

> **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.

Итак, менее чем в четырех строках JavaScript (не считая функцию `renderTweet`) я настроил потоковую пересылку и отображение записей из Twitter на своей странице.

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') {
        initiliseChat();
    } 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 пересылает 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!

Заключение

Эта глава вооружила вас последними новинками из арсенала средств HTML5 для построения веб-приложений. Теперь вы умеете создавать многопоточные, многооконные, междоменные, не тормозящие и не отстающие от реального времени приложения, применяя простейшие методы пересылки строковых данных. Так идите и напишите что-нибудь потрясающее!

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, 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 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).

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 вспомогательными технологиям.

Спасибо за то, что купили эту книгу. Еще увидимся! Все примеры из книги (и многое другое) можно найти на веб-сайте http://www.introducinghtml5.com. Приятного путешествия! Наслаждайтесь разработкой потрясающих программ. Удачи!

Брюс Лоусон и Реми Шарп

Бали, Бирмингем и Брайтон, ноябрь 2009–май 2010

Брюс Лоусон, Рэми Шарп

Изучаем HTML5. Библиотека специалиста

Перевели с английского Т. Качковская, Е. Шикарева

Заведующий редакцией	*А. Кривцов*
Руководитель проекта	*А. Юрченко*
Ведущий редактор	*Ю. Сергиенко*
Научный редактор	*А. Гребеньков*
Литературный редактор	*В. Шрага*
Корректор	*В. Листова*
Верстка	*Е. Егорова*

Подписано в печать 17.12.10. Формат 70x100/16. Усл. п. л. 21,93. Тираж 2000. Заказ 24848.
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.
Отпечатано по технологии CtP в ОАО «Печатный двор» им. А. М. Горького.
197110, Санкт-Петербург, Чкаловский пр., 15.