ГЛАВА 22

ПРИМЕНЕНИЕ JAVASCRIPT

Мэтт Маркус

Теперь, когда вы познакомились с языком JavaScript, давайте рассмотрим некоторые способы его применения в современном веб-дизайне. Во-первых, мы изучим написание сценариев DOM, что позволит управлять элементами, атрибутами и текстом на странице. Я познакомлю вас с некоторыми ресурсами готовых сценариев JavaScript и DOM, так что вам не придется делать это в одиночку. Вы узнаете о «полизаполнениях», предоставляющих более старым браузерам современные возможности и нормализующих функциональность. Я также познакомлю вас с библиотеками JavaScript, которые упрощают жизнь разработчика благодаря подборкам полизаполнений и сценариев для часто выполняемых задач.

В этой главе

- Использование объектной модели документа (DOM) для получения доступа к элементам, атрибутам, их содержимому и для их изменения
- Использование полизаполнения для обеспечения согласованности работы версий браузеров
- Использование библиотек JavaScript
- Краткое введение в Ajax

Объектная модель документа

В этой книге вам несколько раз встречались упоминания об *объектной модели документа* (Document Object Model (сокращенно DOM)), а теперь самое время уделить ей внимание, которого она заслуживает. DOM предоставляет нам способ получить доступ к содержимому документа и управлять им. Обычно она используется для HTML, но DOM можно применять с любым языком XML. И хотя мы сосредоточены на связи объектной модели документа с JavaScript, стоит отметить, что модель DOM может быть доступна для иных языков, таких как PHP, Ruby, Python, C ++, Java, Perl, и многих других. Несмотря на то что версия DOM Level 1 была выпущена консорциумом Всемирной паутины в 1998 году, ее сценарии начали приобретать популярность только спустя почти пять лет.

DOM представляет собой программный интерфейс (API-интерфейс) для HTML и XML-страниц. Модель обеспечивает структурированную схему документа, а также набор методов для слаженной работы с элементами, содержащимися в нем. По сути, объектная модель документа переводит разметку в формат, понятный JavaScript (и другим языкам). Звучит довольно сухо, но основная суть в том, что DOM служит схемой для всех элементов на странице. Мы можем использовать ее, чтобы найти элементы по их именам и атрибутам, а затем добавлять, изменять или удалять элементы и их содержимое. Без DOM JavaScript не поймет

Объектная модель документа предоставляет способ получить доступ к содержимому документа и управлять им.

содержимое документа, я имею в виду, совершенно ничего из контента документа. Ко всему, начиная от типа документа страницы до каждой отдельной буквы в тексте, можно получить доступ через DOM и управлять с помощью JavaScript.

Дерево узлов

Простой способ представить DOM — в виде дерева документа (рис. 22.1). Вы уже видели документы, представленные подобным схематичным образом, когда изучали селекторы CSS.

```
<html>
<head>
<title>Заголовок документа</title>
<meta charset="utf-8">
</head>
<body>
<div>
<h2>Подзаголовок</h2>
<p>Абзац текста с <a href="foo.html">ссылкой</a> здесь</p>
</div>
<div>
>Еще текст</p>
</div>
</body>
</html>
```

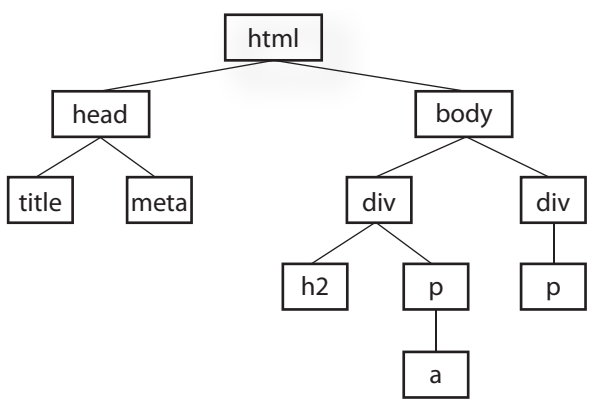

Рис. 22.1. *Простое дерево документа*

Каждый элемент на странице считается *узлом*. Если представить DOM в виде дерева, то каждый узел — это отдельная ветка, на которой могут расти другие. Однако DOM позволяет получить более полный доступ к содержимому, чем CSS, поскольку объектная модель воспринимает также и само содержимое как узел. На рис. 22.2 показана структура

первого элемента **p**. Элемент, его атрибуты и содержимое — все это узлы в дереве DOM.

Также эта модель предоставляет стандартный набор методов и функций, посредством которых JavaScript может взаимодействовать с элементами на странице. Большинство сценариев DOM включают в себя чтение документа и запись в него. Использовать DOM для поиска нужного содержимого документа можно несколькими способами. Рассмотрим некоторые конкретные методы, которые можно применить для доступа к объектам, определенным объектной моделью документа (специалисты по JS называют этот метод «обойти DOM»), а также некоторые методы для управления этими элементами.

НА ЗАМЕТКУ

Модель DOM — это набор узлов:

- Узлы элементов
- Узлы атрибутов
- Текстовые узлы

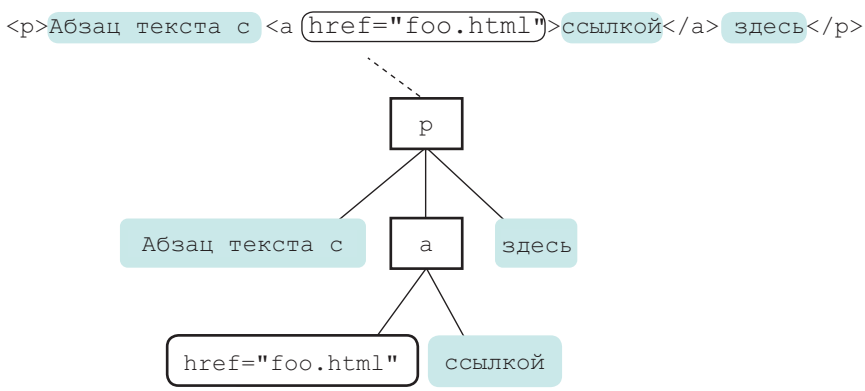

Рис. 22.2. Узлы первого элемента p в документе-образце

Получение доступа к узлам DOM

В DOM объект **document** определяет саму страницу и чаще всего служит в качестве отправной точки для «обхода» DOM. Объект **document** обладает набором стандартных свойств и методов для осуществления доступа к коллекциям элементов. Он похож на свойство **length**, с которым мы познакомились в главе 21. Подобно тому, как **length** является стандартным свойством всех массивов, объект **document** имеет ряд встроенных свойств, содержащих информацию о документе. Мы прокладываем себе путь к нужному элементу, соединяя в цепочку эти свойства и методы и разделяя их точками, чтобы составить своего рода маршрут по документу.

Чтобы дать вам общее представление о том, что я имею в виду, инструкция в данном примере указывает посмотреть на страницу (**document**), найти элемент, имеющий значение **id** «beginner», найти HTML-контент внутри этого элемента (**innerHTML**), и сохранить этот контент в переменной (**foo**).

```
var foo = document.getElementById( "beginner" ).innerHTML;
```

Поскольку подобные цепочки, как правило, получаются длинными, часто можно увидеть, что каждое свойство или метод указывается на

отдельной строке, чтобы код было легче прочитать с первого взгляда. Помните, что пробелы в JavaScript игнорируются, и они не влияют на анализ инструкции.

```
var foo = document
.getElementById( "beginner" )
.innerHTML;
```

Существует несколько способов получения доступа к узлам в документе.

По имени элемента

`GetElementsByTagName()`

Мы можем получить доступ к отдельным элементам с помощью самих тегов, используя метод **`document.getElementsByTagName()`**. Он извлекает любой элемент или элементы, указанные в качестве аргумента.

ПРИМЕЧАНИЕ

Списки узлов — это живые коллекции. Если вы управляете документом с помощью цикла списка узлов, например проходите по всем абзацам, добавляя новые, в итоге у вас может получиться бесконечный цикл.

Например **`document.getElementsByTagName ("p")`** извлекает все абзацы страницы, заключив их в оболочку, под названием *коллекция* или *список узлов* (*nodeList*), в том порядке, в каком они следуют в документе сверху вниз. Списки узлов во многом ведут себя так же, как массивы. Чтобы получить доступ к конкретным пунктам в списке узлов мы ссылаемся на них по индексу, так же, как в массиве.

```
var paragraphs = document.getElementsByTagName("p");
```

На основании этой инструкции переменной **`paragraphs[0]`** — ссылка на первый абзац в документе, **`paragraph[1]`** — на второй, и так далее.

Если бы нам потребовалось получить доступ к каждому элементу в списке узлов отдельно, по одному за раз... ну, хорошо, что мы уже знаем, как перебирать массив с помощью циклов. В списках узлов циклы работают точно так же.

```
var paragraphs = document.getElementsByTagName("p");
for( var i = 0; i < paragraphs.length; i++ ) {
// выполнить некое действие
}
```

Теперь мы можем получить доступ к каждому абзацу на странице по-отдельности, указав **`paragraphs[i]`** внутри цикла, так же, как это делалось с массивом, но задав элементы страницы вместо значений.

По значению атрибута `id`

`GetElementById()`

Извлекает один элемент по его идентификатору (значению его атрибута **`id`**), которое мы предоставляем для этого метода в качестве аргумента. Например, чтобы получить доступ к конкретному изображению:

```
<img src="photo.jpg" alt="" id="lead-photo">'
```

мы включаем значение **id** в качестве аргумента метода **GetElementById()**:

```
var photo = document.getElementById("lead-photo");
```

По значению атрибута class

getElementsByClassName()

Как указано в названии, этот метод позволяет получить доступ к узлам в документе по значению атрибута **class**. Данная инструкция присваивает переменной **firstColumn** любой элемент со значением «column-a» атрибута **class**, чтобы к нему можно было с легкостью получить доступ из сценария.

```
var firstColumn = document.getElementsByClassName("column-a");
```

Как и в случае с методом **getElementsByTagName**, он извлекает список узлов, в котором мы можем указать индекс элемента или перебирать их по одному с помощью цикла.

По селектору

querySelectorAll()

Метод **querySelectorAll** позволяет получить доступ к узлам DOM по селектору CSS-стилей. Синтаксис аргументов в следующих примерах должен быть вам знаком. Это может быть просто, как получение доступа к потомкам конкретного элемента:

```
var sidebarPara = document.querySelectorAll(".sidebar p");
```

или сложно, как выбор элементов по атрибутам:

```
var textInput = document.querySelectorAll("input[type='text']");
```

Подобно методам **getElementsByTagName** и **getElementsByClassName**, метод **querySelectorAll** выводит список узлов (даже если селектору соответствует только один элемент).

Доступ к значению атрибута

GetAttribute()

Как я упоминал ранее, с помощью DOM можно получить доступ не только к элементам. Чтобы извлечь значение атрибута, прикрепленного к узлу элемента, мы используем метод **getAttribute()** с одним аргументом: именем атрибута. Давайте предположим, что у нас есть изображение, stratocaster.jpg, указанное в разметке таким образом:

```
<img src="stratocaster.jpg" alt="electric guitar" id="lead-image">
```

В следующем примере мы получаем доступ к определенному изображению (**getElementById**) и сохраняем ссылку на него в переменной (**bigImage**). На этом этапе мы можем получить доступ к любому из

атрибутов элемента (**alt**, **src** или **id**), указав его в качестве аргумента для метода **getAttribute**. В этом примере мы получим значение атрибута **src** и используем его как содержание уведомления.

```
var bigImage = document.getElementById("lead-image");

alert( bigImage.getAttribute("src") ); // Уведомление
"stratocaster.jpg".
```

Управление узлами

После того как мы получим доступ к узлу с помощью одного из описанных ранее методов, DOM предлагает нам несколько встроенных методов для управления этими элементами, их атрибутами и контентом.

setAttribute()

В предыдущем примере показано, как получить значение атрибута, но что если необходимо *задать* значение атрибута **src**, указав совершенно иной путь? Примените метод **setAttribute()**! Для его использования потребуются два аргумента: атрибут, который необходимо изменить, и новое значение этого атрибута. В данном примере мы добавим немного кода JavaScript, чтобы поменять изображения, изменив значение атрибута **src**.

```
var bigImage = document.getElementById("lead-image");

bigImage.setAttribute("src", "lespaul.jpg");
```

Только подумайте, что можно сделать с документом, изменив значения атрибутов. Здесь мы заменили изображение, но тот же самый метод можно использовать, чтобы произвести целый ряд изменений по всему документу:

- Обновлять атрибуты **checked** флажков и переключателей на основе действий пользователя в другом месте страницы.
- Найти элемент **link** файла *.css* и указать в значении **href** другую таблицу стилей, изменив все стили на странице.
- Обновлять атрибут **title**, добавляя информацию о состоянии элемента (например, «этот элемент выбран в данный момент»)

InnerHtml

InnerHtml предоставляет нам простой способ доступа к тексту и разметке внутри элемента и изменения их. Он ведет себя иначе, чем рассмотренные выше методы. Допустим, нам нужен быстрый способ добавить абзац текста к первому элементу на странице с классом **intro**:

```
var introDiv = document.getElementsByClassName("intro");

introDiv[0].innerHTML = "<p>Это наш вводный текст</p>";
```

Вторая инструкция здесь добавляет содержимое строки в **introDiv** (элемент, у которого значение атрибута **class** — **intro**) как *живой элемент*, потому что **innerHtml** указывает JavaScript обрабатывать строки ** ** и ** ** как разметку.

style

DOM также позволяет добавлять, изменять или удалять из элемента стиль CSS с помощью свойства **style**. Его действие похоже на применение стиля с использованием встроенного атрибута **style**. Отдельные свойства CSS доступны в качестве свойств свойства **style**.

```
document.getElementById("intro").style.color = "#fff";

document.getElementById("intro").style.backgroundColor =
"#f58220";

//оранжевый
```

Имена свойств, которые указывались в CSS через дефис (например, **background-color** и **border-top-width**), в JavaScript и DOM пишутся слитно, но каждое новое слово начинается с заглавной буквы (**backgroundColor** и **borderTopWidth**, соответственно), чтобы дефис (**-**) случайно не приняли за математическую операцию.

В только что рассмотренных примерах свойство **style** используется для задавания стилей узла. Его также можно использовать, чтобы получить значение стиля для применения его в другом месте в сценарии. Данная инструкция извлекает цвет фона элемента **#intro** и присваивает его переменной **brandColor**:

```
var brandColor = document.getElementById("intro").style.
backgroundColor;
```

Добавление и удаление элементов

До сих пор мы рассматривали примеры извлечения и задавания узлов в существующем документе. DOM также позволяет разработчикам изменять саму структуру документа, добавляя и удаляя узлы на ходу. Мы начнем с создания новых узлов, что довольно просто сделать, а затем посмотрим, как добавить их на страницу. Методы, показанные здесь, более точные, чем добавление контента с помощью метода **innerHtml**. В процессе мы также будем удалять узлы.

createElement()

Чтобы создать новый элемент, примените метод **createElement()**. Эта функция принимает один аргумент: элемент, который должен быть создан. Применение этого метода сначала кажется немного нелогичным, потому что новый элемент не появится на странице сразу же. Как только мы создаем элемент таким образом, он остается «плавать в эфире JavaScript», пока мы не добавим его в документ. Представьте, что это создание *ссылки* на новый элемент, который хранится только в памяти и которым мы можем управлять в JavaScript по своему усмотрению, а затем добавить на страницу, когда понадобится.

```
var newDiv = document.createElement("div");
```

createTextNode()

Если мы хотим ввести текст в созданный нами или в уже существующий элемент на странице, можно вызвать метод **createTextNode()**.

Чтобы его использовать, предоставьте в качестве аргумента текстовую строку, и метод создаст удобную для использования в DOM версию этого текста, готовую к добавлению на страницу. Почти как и метод **createElement**, данный метод создает ссылку на новый текстовый узел, который можно сохранить в переменной и добавить на страницу, когда придет время.

```
var ourText = document.createTextNode («Это наш текст.");
```

appendChild()

Итак, мы создали новый элемент и новую строку текста, но как сделать их частью документа? Введите метод **appendChild**. Он принимает один аргумент: узел, который вы хотите добавить в DOM. Вы вызываете его для существующего элемента, который будет его *родительским* элементом в структуре документа. Пора привести пример.

Ниже указан простой элемент **div** на странице с идентификатором **our-div**:

```
<div id="our-div"> </ div>
```

Допустим, мы хотим добавить к элементу **#our-div** абзац, содержащий текст «Здравствуй, мир!». Начнем с создания элемента **p (document.createElement())**, а также текстового узла для контента, который будет в нем содержаться **(createTextNode())**.

```
var ourDiv = document.getElementById("our-div");
var newParagraph = document.createElement("p");
var copy = document.createTextNode("Здравствуй, мир!");
```

Теперь у нас есть элемент, и текст, и мы можем соединить их вместе с помощью метода **appendChild()**.

```
newParagraph.appendChild( copy );
ourDiv.appendChild( newParagraph );
```

Первая инструкция добавляет к новому созданному нами абзацу **(newParagraph)** копию **copy** (это наш текстовый узел «Здравствуй, мир!») так, что теперь у элемента есть контент. Вторая строка добавляет **newParagraph** к исходному элементу **div (ourDiv)**. Теперь наш элемент **ourDiv** не хранится пустым в DOM, и он будет отображаться на странице с контентом «Здравствуй, мир!».

Вы должны получить представление о том, как это работает. Как насчет еще пары примеров?

insertBefore()

Метод **insertBefore()**, как можно догадаться, вставляет один элемент перед другим. Он принимает два аргумента: первый — вставляемый узел, а второй — элемент, перед которым его вставляют. Кроме того, необходимо знать, к какому родительскому элементу добавляется элемент. Так, например, чтобы вставить новый заголовок перед абзацем в следующую разметку:

```
<div id="our-div">
<p id="our-paragraph">Текст нашего абзаца</p>
</div>
```

Мы начнем с присвоения имен переменных содержащимся в ней элементам **div** и **p**, затем создадим элемент **h1** и его текстовый узел, а потом объединим их, как было показано в последнем примере.

```
var ourDiv = document.getElementById("our-div");
var para = document.getElementById("our-paragraph");
var newHeading = document.createElement("h1");
var headingText = document.createTextNode("A new heading");
newHeading.appendChild( headingText );
// Добавляем наш новый текстовый узел к новому заголовку
```

Наконец, в последней инструкции, показанной ниже, с помощью метода **insertBefore()** перед элементом **para** внутри **ourDiv** вставляется элемент **newHeading h1**.

```
ourDiv.insertBefore( newHeading, para );
```

replaceChild()

Метод **replaceChild()** заменяет один узел другим и принимает два аргумента. Первый— новый дочерний элемент (то есть узел, который вам в итоге нужен). Второй — это узел, который будет заменен первым. Как и с методом **insertBefore()**, вам также необходимо определить родительский элемент, в котором происходит замена. Для простоты предположим, что мы начнем со следующей разметки:

```
<div id="our-div">
<div id="swap-me"></div>
</div>
```

и нам нужно заменить элемент **div** с идентификатором **swap-me** на изображение. Начнем с создания нового элемента **img** и задания атрибуту **src** пути к файлу изображения. В итоговом выражении, мы используем метод **replaceChild()**, чтобы заменить элемент **swapMe** на **newImg**.

```
var ourDiv = document.getElementById("our-div");
var swapMe = document.getElementById("swap-me");
var newImg = document.createElement("img");
// Создать новый элемент изображения
newImg.setAttribute( "src", "path/to/image.jpg" );
// Задать новому изображению атрибут "src"
ourDiv.replaceChild( newImg, swapMe );
```

removeChild()

Мы эти элементы породили, и мы можем снова их уничтожить. С помощью метода **removeChild()** легко удалить узел или целую ветвь дерева документа. Метод принимает один аргумент, являющийся узлом, который требуется удалить. Помните, что DOM мыслит узлами, а не просто элементами, поэтому дочерним элементом может оказаться текст (узел), содержащийся в элементе, а не только другие элементы.

Как и метод **appendChild**, **removeChild** метод всегда вызывается для родительского элемента того элемента, который необходимо удалить

(отсюда «удалить *дочерний* элемент»). Это означает, что нам нужны ссылки, как на родительский узел, так и на узел, который мы хотим удалить. Предположим, что схема разметки следующая:

```
<div id="parent">
<div id="remove-me">
<p>Фу, все равно мне тут никогда не нравилось.</p>
</div>
</div>
```

Наш сценарий будет выглядеть следующим образом:

```
var parentDiv = document.getElementById("parent");
var removeMe = document.getElementById("remove-me");
parentDiv.removeChild( removeMe );
// Удаляет со страницы элемент div с id "remove-me".
```

Для дополнительного чтения

Вы должны получить четкое представление о том, что такое написание сценариев DOM. Конечно, я едва коснулся темы о том, что можно сделать с помощью DOM, но если вы хотите узнать больше, обязательно прочитайте книгу «DOM Scripting: Web Design with JavaScript and the Document Object Model» Джереми Кейта и Джеффри.

Полизаполнения

В этой книге вы уже познакомились с огромным количеством новых технологий: новыми элементами HTML5, новыми способами работы с CSS3, использования JavaScript для управления DOM, и многими другими. В идеальном мире все браузеры совершенствовались бы одинаково, идя в ногу с передовыми технологиями и попутно осваивая уже упрочившиеся (см. врезку «Войны браузеров»). В этом идеальном мире отставшие браузеры просто исчезали бы полностью. К сожалению, мы живем в ином мире, и недостатки браузеров остаются занозой в боку всех разработчиков.

Я буду первым, кто признает, что ему нравится заново изобретать колесо. Во-первых, это хороший способ научиться. Кроме того, именно поэтому наши автомобили ездят не на округлых камнях и фрагментах стволов деревьев. Но когда дело доходит до всех странных причуд браузеров, нам не обязательно начинать с нуля. Множество людей умнее меня сталкивались с этими проблемами раньше и уже нашли умные способы обойти их или исправили фрагменты JavaScript и DOM, с которыми браузеры не смогли справиться.

Полизаполнения — это термин, придуманный Реми Шарпом для описания «прокладки» JavaScript, которая нормализует различное поведение сценариев в разных браузерах.

Войны браузеров

Язык JavaScript появился в смутное время беззакония до зарождения движения в поддержку веб-стандартов, когда все основные игроки в мире браузеров (как бы это лучше сказать) импровизировали. Скорее всего, никого не удивит, что компании Netscape и Microsoft применяли радикально отличающиеся друг от друга версии DOM, а преобладающим настроением было «пусть победит сильнейший».

Я избавлю вас от подробностей битвы за высоту JavaScript, но две конкурирующих реализации были настолько различны, что обе оказались по большому счету бесполезны, если вам не хотелось поддерживать две разные базы кода или добавлять на свои сайты предупреждение: «Рекомендуется просматривать в Internet Explorer/Netscape».

В то беспощадное время консорциум Всемирной паутины собирал основы для современной стандартизированной модели DOM, которую позднее мы узнали и полюбили. К счастью для нас, компании Netscape и Microsoft присоединились к движению в поддержку стандартов. Стандартизированная модель DOM поддерживается вплоть до версий Internet Explorer 5 и Netscape Navigator 6. К сожалению, прогресс в Internet Explorer в этой области остановился на достаточно долгое время после версии Internet Explorer 6. В результате в более старых версиях этого браузера есть несколько существенных отличий от современной модели DOM. К счастью, они возвращают свои позиции с версией Internet Explorer 9 и более поздними.

Проблема в том, что ваш проект, вероятно, еще должен будет обеспечивать поддержку пользователям старых версий браузера Internet Explorer. Это нелегко, но мы готовы. В нашем распоряжении восхитительный набор инструментов, таких как полизаполнения и библиотеки JavaScript со множеством вспомогательных функций, нормализующих мелкие причуды, которые могут встретиться нам в разных браузерах.

«Прокладка, которая имитирует будущий API-интерфейс, предоставляя старым браузерам альтернативные функции».

— Пол Айриш

В этой фразе множество скачков во времени, но в основном в ней говорится, что мы заставляем новинки работать в браузерах, которые изначально их не поддерживали — будь то совершенно новая технология, такая как обнаружение физического местоположения пользователя или исправление вещей, которые просто неправильно работают в одном из браузеров.

Существует множество полизаполнений, ориентированных на выполнение конкретных задач, таких как распознавание старыми браузерами новых элементов HTML5 или селекторов CSS3, и с возникновением новых проблем все время появляются новые задачи. Я расскажу вам о наиболее часто используемых полизаполнениях, имевшихся в арсенале современного разработчика на момент публикации книги. К тому времени, как вы окажетесь на передовой веб-дизайна, возможно, вы обнаружите, что нужны новые полизаполнения.

HTML5 Shiv (или Shim)

Возможно, вы помните, что встречали ее в главе 5, но давайте уделим этой прокладке больше внимания теперь, когда вы уже приобрели опыт работы с JavaScript.

Полизаполнение HTML5 shiv/shim применяется, чтобы дать возможность браузеру Internet Explorer 8 и более ранним версиям распознавать и добавлять стили к новым элементам HTML5, таким как **`article`**, **`section`** и **`nav`**.

Как это работает?

Существует несколько вариаций полизаполнения HTML5 shiv/shim, но все они работают очень похоже: обходят DOM в поисках элементов, которые не распознает браузер Internet Explorer, а затем сразу же заменяют их таким же элементом так, что Internet Explorer видит их в DOM. Теперь любые стили, написанные для этих элементов, будут работать так, как нужно.

Кто создал?

Данную технику первым открыл Сьерд Вишер, и сейчас существует множество вариаций этих сценариев. Версия Реми Шарпа, похоже, сегодня является наиболее широко используемой.

Как использовать?

У всех вариаций этого сценария одно и то же требование: ссылка на него должна находиться в разделе **`head`** документа, чтобы сообщить Internet Explorer о новых элементах, до того, как браузер завершит визуализацию страницы.

```
<!--[if lt IE 9]>
<script src="html5shim.js"></script>
<![endif]-->
```

Потенциальные недостатки

Основной нюанс в том, что в старых версиях Internet Explorer, где JavaScript отключен или недоступен, элементы будут визуализироваться без стилей.

Где взять и как узнать больше?

- Статья в Википедии (**en.wikipedia.org/wiki/HTML5_Shiv**)
- Публикация Реми Шарпа (**remysharp.com/2009/01/07/html5-enabling-script**)
- Статья «Что такое HTML5 SHIV — уроки HTML5» (**tinyurl.com/html5shiv**)

Modernizr

Сам по себе Modernizr — это не полизаполнение, а, скорее, набор тестов, который можно использовать для обнаружения наличия функций бра-

узера и загрузки полизаполнений по мере необходимости. Разработчики Modernizr также курируют массивное хранилище полизаполнений для огромного количества функций (см. примечание).

Как это работает?

Modernizr выявляет наличие методов и функций, используемых API-интерфейсами JavaScript для более новых функций HTML5 и CSS3, и определяет, поддерживает ли браузер функцию изначально. или ему необходимо полизаполнение. Например, если браузер содержит встроенные методы для взаимодействия с HTML5-элементом **`canvas`**, мы можем предположить, что он поддерживает элемент **`canvas`**. Это известно как «функция обнаружения» и находится в резком контрасте с устаревшей практикой обнаружения при помощи пользовательского агента (User Agent, UA) или самого браузера. Modernizr также содержит совершенно новую прокладку HTML5 shim, похожую на ту, которая подробно описана выше.

Кто создал?

Полизаполнение Modernizr разработал Фарук Атес, а Пол Айриш, Алекс Сэкстон, Райан Седдон и Александр Фаркас активно его развивают.

Как использовать?

На сайте Modernizr.com есть инструмент разработчика, позволяющий добавлять только те тесты, которые имеют отношение к вашему проекту, а также форма «разработки», содержащая целую библиотеку тестов. После того как вы загрузите пользовательскую форму, просто добавьте Modernizr, как и любой другой внешний сценарий.

Где взять и как узнать больше?

- На сайте Modernizr (**modernizr.com**)

Selectivizr

Полизаполнение Selectivizr позволяет более старым версиям браузера Internet Explorer понимать сложные селекторы CSS3, такие как **`:nth-child`** и **`::first-letter`**.

Как это работает?

Полизаполнение Selectivizr использует код JavaScript для извлечения и анализа содержимого вашей таблицы стилей и «залатывания дыр» там, где не справляется встроенный синтаксический анализатор CSS браузера.

Кто создал?

Полизаполнение Selectivizr создал и поддерживает Кейт Кларк.

Как использовать?

Полизаполнение Selectivizr необходимо использовать вместе с библиотекой JavaScript (я расскажу о ней в следующем разделе). Ссылка

ПРИМЕЧАНИЕ

Архив полизаполнений, поддерживаемый разработчиками Modernizr, доступен по адресу **github.com/Modernizr/Modernizr/wiki/HTML5-Cross-Browser-Polyfills**.

на сценарий указывается в условном комментарии браузера Internet Explorer после ссылки на файл библиотеки .js, вот так:

```
<script type="text/javascript" src="[JS library]"></script>
<!--[if (gte IE 6)&(lte IE 8)]>
<script type="text/javascript" src="selectivizr.js"></
script>
<noscript><link rel="stylesheet" href="[fallback css]" /></
noscript>
<![endif]-->
```

Потенциальные недостатки

Поскольку здесь мы отказываемся от использования встроенного синтаксического анализатора CSS, может наблюдаться падение производительности браузеров, в которых применяется полизаполнение.

Где взять и как узнать больше?

- На сайте Selectivizr (**selectivizr.com**)

Respond.js

Respond.js — быстрое и простое полизаполнение, позволяющее старым версиям браузеров (чаще Internet Explorer 8 и более ранним) понимать медиазапросы **`min-width`** и **`max-width`**, широко используемые в адаптивном дизайне.

Как это работает?

Как и Selectivizr, полизаполнение Respond.js просматривает таблицы стилей независимо от встроенного синтаксического анализатора браузера и, найдя медиазапрос **`min-width`** или **`max-width`**, применяет эти стили к элементам на странице с помощью JavaScript в зависимости от ширины окна браузера.

Кто создал?

Полизаполнение Respond.js было создано компанией Filament Group и членом команды разработчиков jQuery Mobile Скоттом Джелом. Первоначально оно разрабатывалась для использования на адаптивном веб-сайте BostonGlobe.com, а позднее было выпущено как проект с открытым исходным кодом.

Как использовать?

Следует загрузить только сценарий Respond.js и сослаться на полизаполнение в теге **`script`** в разделе **`head`** документа (после таблицы стилей).

Потенциальные недостатки

Опять же, как в случае с Selectivizr, при использовании этого сценария может наблюдаться небольшое снижение производительности, но только в тех браузерах, где он в итоге применяется.

Где взять и как узнать больше?

Страница ответов Скотта Джела на сервисе GitHub (**github.com/scottjehl/Respond**).

Библиотеки JavaScript

В продолжение разговора о том, что вам не обязательно писать весь код с нуля самостоятельно, пришло время взяться за библиотеки JavaScript. *Библиотека JavaScript* — это коллекция готовых функций и методов, которые вы можете использовать в своих сценариях для выполнения распространенных задач или упрощения сложных. Существует очень много библиотек JS. Некоторые из них — крупные базы, содержащие все наиболее распространенные полизаполнения, сокращения и виджеты, которые вам когда-либо понадобятся для разработки полноценных веб-приложений по технологии Ajax (см. врезку «Что такое Ajax?»). Другие ориентированы на конкретные задачи, такие как обработка

Что такое Ajax?

Ajax (иногда пишется AJAX) означает *Асинхронный JavaScript и XML*. Часть «XML» не так важна, вам не нужно использовать его, чтобы применять Ajax (подробнее об этом чуть позже). «Асинхронный» — вот что имеет значение.

Традиционно, когда пользователь взаимодействовал с веб-страницей, чтобы получить данные с сервера, вся работа должна была прекращаться и ожидать их поступления, а когда данные оказывались доступны, необходимо было перезагружать страницу целиком. Это не способствовало созданию у пользователей впечатления плавной работы.

Однако, поскольку с помощью Ajax страница может получить данные с сервера в фоновом режиме, обновление страницы можно производить плавно и в реальном времени. Благодаря этому веб-приложения становятся более похожи на настольные приложения.

Вы видите это на многих современных веб-сайтах, хотя иногда работа проводится тонко. В социальной сети Twitter, например, при прокручивании страницы вниз загружаются новые твиты. Они не указаны в разметке страницы, а загружаются динамически, по мере необходимости. Подобный подход используется при поиске изображений в Google. Когда вы достигаете конца текущей страницы, появляется кнопка, которая позволяет загружать больше изображений, но вы никогда не уходите с текущей страницы.

Термин «Ajax» был впервые введен Джесси Джеймсом Гарреттом в статье под названием «Ajax: A New Approach to Web Applications». Ajax — это не одна технология, а, скорее, сочетание технологий HTML,CSS, DOM и JavaScript, включая объект **`XMLHttpRequest`**, который позволяет передавать данные в асинхронном режиме. Ajax может использовать XML для передачи данных, однако все чаще используется *JSON* (*JavaScript Object Notation*), удобочитаемый формат для обмена данными на основе JavaScript.

Вам не придется заниматься созданием веб-приложений с помощью Ajax сразу же, но во многих библиотеках JavaScript, обсуждаемых в этой главе, есть встроенные помощники и методы Ajax, которые позволят вам начать работу, приложив значительно меньше усилий.

форм, анимации, диаграмм или математических функций. Для опытных профессионалов в области разработки на языке JavaScript начинать работу с библиотеки — замечательный способ сэкономить время. А для новичков библиотека способна выполнять задачи, которые могут быть им пока еще не по силам.

Недостаток библиотек заключается в том, что, поскольку они обычно хранят все функции в одном большом файле .js, в конечном итоге, возможно, посетителям ваших сайтов придется загружать большие объемы кода, который никогда не используется. Но разработчики библиотек, зная об этом, сделали многие из них модульными и продолжают прилагать усилия, чтобы оптимизировать свой код. В некоторых случаях также возможно превратить сценарий в пользовательский и задействовать только те его фрагменты, которые нужны.

Несколько библиотек, которые следует знать

К некоторым из наиболее популярных на момент написания книги библиотекам JS относятся:

- ***jQuery*** (**jQuery.com**). Библиотека JQuery, написанная в 2005 году Джоном Резигом, на сегодняшний день является самой популярной библиотекой JavaScript, применяется более чем на половине из 10000 наиболее посещаемых веб-сайтов. Она бесплатна, с открытым исходным кодом и использует синтаксис, который облегчает работу, если вы уже свободно себя чувствуете с CSS, JavaScript и DOM. Дополнить JQuery можно библиотекой JQuery UI, которая добавляет такие элементы интерфейса, как виджеты календаря, функции перетаскивания, списки-аккордеоны и простые эффекты анимации. Я уже упоминал, что работаю на jQueryMobile. Это еще одна библиотека на основе JQuery, которая предоставляет элементы пользовательского интерфейса и полизаполнения, предназначенные для различных мобильных браузеров и их печально известных причуд.
- ***Dojo*** (**dojotoolkit.org**). Dojo — модульный набор инструментов с открытым кодом, который особенно полезен для разработки веб-приложений с использованием Ajax.
- ***Prototype*** (**prototypejs.org**). Библиотека Prototype JavaScript Framework, написанная Сэмом Стивенсоном, была разработана, чтобы добавить поддержку Ajax для фрейморка Ruby on Rails.
- ***MooTools*** (**mootools.net**). MooTools (расшифровывается как «Мои объектно-ориентированные инструменты») — еще одна модульная библиотека с открытым исходным кодом, написанная Валерио Проетти.
- ***YUI*** (**yuilibrary.com**). Библиотека пользовательских интерфейсов Yahoo! — еще одна бесплатная библиотека JS с открытым исходным кодом для создания многофункциональных веб-приложений. Она является частью проекта «The YUI Library» компании Yahoo!, основанного Томасом Ша.

Что касается небольших библиотек JS, которые выполняют специализированные функции, потому что они постоянно создаются и устаревают, я рекомендую выполнить поиск во Всемирной паутине по запросу «библиотеки JavaScript для _____» и изучить результаты.

Некоторые категории библиотек включают в себя:

- Формы
- Анимацию
- Игры
- Графическую информацию
- Изображения и трехмерные эффекты элемента **`canvas`**.
- Строки и математические функции
- Работу с базами данных.

ПРИМЕЧАНИЕ

Сравнение более 20 библиотек JavaScript, а также их размеров и особенностей приводится в Википедии по адресу: **en.wikipedia.org/wiki/Comparison_of_JavaScript_frameworks**.

На сайте Google Developers также поддерживается список наиболее популярных библиотек JavaScript с открытым исходным кодом. Найти его можно по адресу: **developers.google.com/speed/libraries/**.

Как пользоваться библиотекой JS (на примере jQuery)

Перечисленные библиотеки легко применить. Все, что нужно сделать — загрузить файл сценария JavaScript (*.js*), разместить на своем сервере, указать путь к нему в теге **`script`**, и можно начинать. Именно файл *.js* выполняет всю работу, предоставляя готовые функции и сокращения синтаксиса. Добавив его, можно писать собственные сценарии, расширяющие функции, встроенные в структуру. Конечно, самое интересное — что вы с ним сделаете (и, к сожалению, это выходит далеко за рамки данной книги).

В следующих примерах мы будем работать с библиотекой jQuery.

Загрузка файла *.js* библиотеки jQuery

Чтобы начать работу с jQuery, перейдите на сайт **jQuery.com**, нажмите кнопку **Download jQuery** и получите копию файла *jQuery.js*. Вы можете выбрать версию *production*, в которой удалены все лишние пробельные символы для обеспечения меньшего размера файла, или версию development, код которой проще читать, но размер этого файла почти в восемь раз больше. Версия production вполне подойдет, если вы не собираетесь редактировать код самостоятельно.

Скопируйте код, вставьте его в новый текстовый документ и сохраните под тем же именем файла, которое указано в адресной строке браузера. На момент написания этой главы последней версией jQuery была 1.10.2 (а также версия 2.0.3), а имя файла *production* версии production было *jQuery-1.10.2.min.js* (min означает «минимизированный»). Поместите этот файл в папку с другими файлами вашего сайта. Некоторые разработчики для порядка хранят сценарии в папке *js*. Куда бы вы ни решили его поместить, обязательно запомните путь к файлу, потому что он понадобится вам в разметке.

Добавление сценария в документ веб-страницы

Добавьте сценарий jQuery в документ таким же образом, как любой другой сценарий: с помощью элемента **script**.

```
<script src="pathtoyourjs/jQuery-1.10.2.min.js"></script>
```

Вот и все. Однако существует альтернатива, о которой стоит упомянуть. Если вы не хотите размещать файл самостоятельно, можете указать на одну из публично размещенных версий и использовать его таким образом. Несколько вариантов перечислено на странице загрузки jQuery, а также предоставлена ссылка на код на сервере корпорации Google. Скопируйте этот код в точности так, как вы видите здесь, вставьте его в раздел **head** документа или перед тегом **</body>**, и у вас будет подключена библиотека jQuery!

```
<script src="https://ajax.googleapis.com/ajax/libs/
jQuery/1.10.2/jQuery.
min.js"></script>
```

Режим готовности

Не стоит запускать сценарии раньше, чем документ и DOM будут к ним готовы, не так ли? В jQuery есть событие, известное как *ready event*, которое проверяет документ, и ждет, пока он не будет готов к тому, чтобы им управляли. Это требуется не для всех сценариев (например, не для запуска уведомления браузера), но если вы выполняете какие-либо действия с DOM, рекомендуется сначала создать условия для ваших сценариев, включив следующую функцию в ваш пользовательский файл *.js*:

```
<script src="pathtoyourjs/jQuery-1.10.2.min.js"></script>
<script>
$(document).ready(function(){
// Здесь будет ваш код
});
</script>
```

Написание сценариев с помощью jQuery

Как только вы будете готовы, можете начать писать собственные сценарии с использованием jQuery. Сокращения, предлагаемые jQuery, делятся на две основные категории:

- Огромный набор встроенных возможностей для обнаружения функций и полизаполнений;
- Более короткий и интуитивно понятный синтаксис для нацеливания на элементы (движок селекторов jQuery). Изучив этот последний раздел, вы уже должны неплохо разбираться в работе полизаполнений, поэтому давайте посмотрим, чем вам может помочь движок селекторов.

jQuery упрощает перемещение по DOM, потому что вы можете использовать синтаксис селекторов, который узнали для CSS. Ниже приведен пример получения элемента по значению его идентификатора без библиотеки:

```
var paragraph = document.getElementById( "status" );
```

Выражение находит элемент с идентификатором **status** и сохраняет ссылку на него в переменную (**paragraph**). Многовато символов для простой задачи. Вы, вероятно, может себе представить, каким объемным станет код, когда вы будете обращаться ко множеству элементов на странице. Однако теперь, когда в игру вступила библиотека jQuery, можно использовать следующее сокращение.

```
var paragraph = $("#status");
```

Все верно — это селектор идентификатора, который вы знаете и любите со времен написания CSS. И это еще не все. *Любой* селектор, используемый в CSS, будет работать в специальной вспомогательной функции.

Хотите найти все вхождения класса **header**? Используйте сокращение **$(".header");**.По имени элемента? Разумеется, **$("div");**. Каждый подзаголовок во врезке? Легче легкого: **$("#sidebar .sub");**. Вы можете даже нацелиться на элементы на основе значения атрибута: **$("[href='http://google.com"]");**. Однако селекторами дело не ограничивается. Мы можем использовать огромное количество вспомогательных функций, встроенных в jQuery и подобные библиотеки, чтобы обойти DOM. jQuery также позволяет нам соединять объекты в цепочку таким образом, чтобы нацелиться на элементы, которые даже CSS не под силу (родительский элемент элемента, например). Скажем, у нас есть абзац, и мы хотим добавить элемент **class** к родительскому элементу этого абзаца. Допустим, мы не знаем, что это будет за родительский элемент, поэтому не можем нацелиться на него напрямую. В jQuery можно добраться до него с помощью объекта **parent()**

```
$("p.error").parent().addClass("error-dialog");
```

Еще одним важным преимуществом является то, что код очень удобочитаем с первого взгляда: «найти любые абзац(ы) с классом **error** и добавить класс **error-dialog** к их родительскому(им) элементу(ам)».

А что если я не знаю, как писать сценарии?

Для изучения JavaScript потребуется время, и возможно, вы не скоро сможете писать собственные сценарии. Но не волнуйтесь. Если поискать во Всемирной паутине то, что вам нужно (например, «Карусель изображений jQuery» или «jQuery аккордеон»), велика вероятность, что вы найдете множество сценариев, которыми люди, создавшие их, делятся в комплекте с документацией о том, как их использовать. Поскольку jQuery использует синтаксис селекторов, очень похожий на CSS, это облегчает процесс настройки сценариев jQuery для использования с вашей собственной разметкой.

Резюме

Всего за две главы мы перешли от знакомства с самыми основами переменных к управлению DOM и использованию библиотеки JavaScript. Даже с учетом всего рассмотренного здесь материала, мы едва начали охватывать все возможности JavaScript. Когда в следующий раз вы увидите веб-сайт, на котором происходит что-то классное, просмотрите исходный код в браузере и поищите сценарии JavaScript. Вы может многому научиться, читая и даже разбирая чужой код.

И помните, в JavaScript ничего нельзя испортить настолько, чтобы это невозможно было исправить несколькими нажатиями клавиши **Delete**. Более того, у JavaScript есть целое сообщество увлеченных разработчиков, которые жаждут учиться и не менее сильно жаждут учить. Ищите разработчиков-единомышленников и делитесь тем, что узнали в процессе. Если вы застряли на сложной проблеме, не стесняйтесь искать помощи и задавать вопросы. Редко случается так, что вы сталкиваетесь с проблемой, которая ни у кого больше не возникала, а в сообществе разработчиков открытого кода всегда рады поделиться тем, что они узнали.