TURING 图灵程序设计丛书 Web开发系列

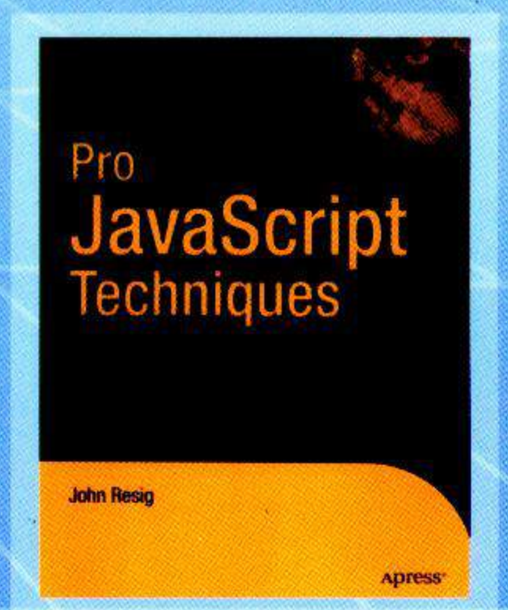
Apress

Pro JavaScript Techniques

精通JavaScript

[美] John Resig 著
陈贤安 江疆 译

- 让你大开眼界的JavaScript力作
- 跟随jQuery之父到达前所未有的深度
- Amazon五星盛誉图书



人民邮电出版社
POSTS & TELECOM PRESS

目　录

Part 1

第一部分

认识现代 JavaScript

本部分内容

第1章

现代JavaScript程序设计

JavaScript语言一直在稳步发展：变化不快但却未曾止步。经过过去这十年，JavaScript给人的印象已经从一门简单如玩具一般的语言演变到备受尊敬的编程语言，世界上众多公司和开发者都用它构造出种种不可思议的应用程序。现代的JavaScript编程语言一如既往地牢固、健壮，功能强大到令人难以置信的地步。本书讨论的诸多内容都会展示，为什么现代的JavaScript应用程序和从前有那么明显的不同。这一章提到的许多概念和方法也许算不上太新颖，但成千上万聪明程序员的认同促使它们的运用得以升华，并最终形成今天的格局。好吧，闲话少说，让我们看看究竟什么叫作现代的JavaScript程序设计。

1.1 面向对象的 JavaScript

从语言的角度来看，面向对象编程或面向对象的JavaScript一点都不算时髦，JavaScript一开始就被设计为一门彻底的面向对象语言。然而，随着JavaScript的广为运用和接受，其他语言（比如Ruby、Python和Perl）的程序员们开始注意到它，并将许多良好的编程习惯带到了JavaScript中来，从而促进了JavaScript的发展。

不管是在写法上还是在运行上，面向对象的JavaScript代码和其他具有对象特性的语言都不一样。在第2章将深入讨论是哪些方面让它如此独特，但现在让我们先感受一下现代的JavaScript该是怎么一个写法。代码清单1-1中是两个对象构造函数的例子，示范用一个对象组合表示学校中的课程。

代码清单1-1　用面向对象的JavaScript表示课程及其安排

```
// 'Lecture' 类的构造函数
// 用名称 (name) 和教师 (teacher) 作为参数
function Lecture(name, teacher) {
    // 将参数保存为对象的局部属性 (local property)
    this.name = name;
    this.teacher = teacher;
}
// Lecture 类的一个方法 (method)，用于生成
// 一条显示 Lecture 信息的字符串
Lecture.prototype.display = function() {
```

```
    return this.teacher + " is teaching " + this.name;
};

// Schedule 类的构造函数，以课程的数组作为参数
function Schedule(lectures) {
    this.lectures = lectures;
}

// 构造一条字符串，表示课程的安排表
Schedule.prototype.display = function() {
    var str = "";

    // 遍历每项课程，建立包含它们信息的字符串
    for (var i = 0; i < this.lectures.length; i++)
        str += this.lectures[i].display + " ";

    return str;
};
```

你很可能已经从代码清单1-1中看出，大部分面向对象基本要素已经存在，不过它们是以不同于其他更常见的面向对象语言的方式组织起来的。你可以创建对象的构造函数、可以添加方法，也可以访问对象的属性。一个使用了上面两个类的例子如代码清单1-2所示。

代码清单1-2 给用户提供一个课程列表

```
// 创建一个新的 Schedule 对象，保存在变量 'mySchedule' 中
var mySchedule = new Schedule([
    // 创建一个 Lecture 对象的数组，作为 Schedule 对象的唯一参数传入
    new Lecture("Gym", "Mr. Smith"),
    new Lecture("Math", "Mrs. Jones"),
    new Lecture("English", "TBD")
]);

// 以弹出窗口的形式显示这个课程信息
alert(mySchedule.display());
```

随着JavaScript被程序员们逐渐接受，设计良好的面向对象代码也日益普及。面向对象JavaScript的代码片段将贯穿全书，这些片段最能体现代码的设计与实现。

1.2 测试代码

建立好一个面向对象的代码基础后，开发专业质量的JavaScript代码的另一方面是，保证有一个健壮的代码测试环境。如果你开发的代码同时还会被其他开发者经常使用和维护，适当的测试就尤其有必要了。给其他开发者提供一个坚实的基础来进行测试，是开发实践中代码维护最重要的方面。

在第4章，你将会看到一系列工具，用于开发一套优秀的测试用例制度，同时给复杂应用程序提供简单的调试功能。其中一个是Firefox的Firebug插件。Firebug提供了一系列有用的工具，包

JavaScript代码则用来使之动态化。这种分离最重要的效果是，代码在不同的浏览器间使用时是完全可以降级（或者升级）运行的，这样你可以给支持某些高级特性的浏览器提供更丰富的交互，同时不支持这些特性的浏览器也能合理地降级运行它所支持的部分。

编写现代的、分离式的代码包括两个方面：文档对象模型（Document Object Model，DOM）
7 和JavaScript事件。在本书中稍后部分会深入讨论这两个方面。

1.4.1　DOM

DOM是表达XML文档的常见方式，它未必是最快的方式，也未必是最轻量级的或者最容易使用的，但的确是应用最广泛的，大部分Web开发的编程语言（比如Java、Perl、PHP、Ruby、Python和JavaScript）都提供了相应的实现。DOM给开发者提供了一种定位XML层级结构的直观方法。

考虑到正确的（valid）HTML不过是XML的一个子集，解析并浏览DOM文档的有效办法无疑能简化JavaScript的开发。归根结底，JavaScript中绝大部分的操作都是JavaScript和网页里不同的HTML元素之间的交互，而DOM则是简化这一过程的出色工具。代码清单1-4中是使用DOM来遍历和查找页面中不同元素并操作它们的例子。

代码清单1-4　使用DOM定位并操作不同的DOM元素

```
<html>
<head>
    <title>Introduction to the DOM</title>
    <script>
    // 我们必须在文档完成加载后再操作 DOM
    window.onload = function() {

        // 获取文档中的所有 <li> 元素
        var li = document.getElementsByTagName("li");

        // 并给它们全部添加一个黑色边框
        for (var j = 0; j < li.length; j++) {
            li[j].style.border = "1px solid #000";
        }

        // 获取 ID 为 'everywhere' 的元素
        var every = document.getElementById("everywhere");

        // 从文档中删除这个元素
        every.parentNode.removeChild(every);

    };
    </script>
</head>
<body>
    <h1>Introduction to the DOM</h1>
8   <p class="test">There are a number of reasons why the
        DOM is awesome, here are some:</p>
    <ul>
```

```
        <li id="everywhere">It can be found everywhere.</li>
        <li class="test">It's easy to use.</li>
        <li class="test">It can help you to find what you want, really quickly.</li>
    </ul>
</body>
</html>
```

DOM是开发分离式JavaScript代码的第一步，在HTML文档中进行简单快速的定位使JavaScript与HTML的互动容易了不少。

1.4.2 事件

事件（event）是黏合应用程序中所有用户交互的“胶水”。在设计良好的JavaScript应用程序中，既有数据来源，又有这些数据的视觉表现（在HTML DOM中的表现），要在这两个方面之间进行同步就必须通过与用户交互，并据此来更新用户界面。DOM和JavaScript事件的组合，是决定所有现代Web应用程序形态的根本所在。

所有现代浏览器都提供了在特定交互动作发生时引发的一系列事件，比如用户移动鼠标、敲击键盘或离开页面等。你可以给这些事件注册一些函数，一旦事件发生就会执行。代码清单1-5展示了这种交互动作的一个实例，当用户把鼠标移到<li>元素上时改变其背景颜色。

代码清单1-5 使用DOM和事件来提供视觉特效

```
<html>
<head>
    <title>Introduction to the DOM</title>
    <script>
    // 我们必须在文档完成加载后再操作 DOM
    window.onload = function() {

        // 获取所有的 <li> 元素，给它们添加上事件处理函数 (event handler)
        var li = document.getElementsByTagName("li");
        for (var j = 0; j < li.length; j++) {

            // 给这个 <li> 元素加上 mouseover 的事件处理函数，
            // 它将这个 <li> 的背景颜色改为蓝色。
            li[i].onmouseover = function() {
                this.style.backgroundColor = 'blue';
            };
            // 给这个 <li> 元素加上 mouseout 的事件处理函数，
            // 将它的背景颜色重置为默认的白色。
            li[i].onmouseout = function() {
                this.style.backgroundColor = 'white';
            };
        }

    };

    </script>
</head>
```

```
<body>
    <h1>Introduction to the DOM</h1>
    <p class="test">There are a number of reasons why the
        DOM is awesome, here are some:</p>
    <ul>
        <li id="everywhere">It can be found everywhere.</li>
        <li class="test">It's easy to use.</li>
        <li class="test">It can help you to find what you want, really quickly.</li>
    </ul>
</body>
</html>
```

JavaScript事件是复杂多变的，本书中大部分的代码和应用程序都用到了某种形式的事件。第6章和附录B就是完全针对事件和它们的交互的。

1.4.3　JavaScript与CSS

在DOM和事件交互的基础上产生了DHTML，它的实质其实就是JavaScript和DOM元素上的CSS属性之间的交互。

作为简便布局、分离式网页的标准，CSS（层叠样式表）在给你（开发者）提供大量功能的同时，只给你的用户带来了最少量的兼容性问题。总之，DHTML的意义在于JavaScript和CSS组合起来能做到什么，你又该如何使用这种组合来创造使人印象深刻的效果。

更高级的交互示例，比如拖放HTML元素和动画效果在第7章会有详细讨论。

1.5　Ajax

Ajax或异步JavaScript与XML（Asynchronous JavaScript and XML），是Jesse James Garrett在*Ajax: A New Approach to Web Applications*（http://www.adaptivepath.com/publications/essays/ archives/000385.php）一文中提出的名词，他是Adaptive Path这家信息体系结构（information architecture）公司的创始人和总裁。这个名词描述的是客户端和服务器之间，在请求和提交附加信息时进行的高级交互动作。

Ajax这个词涵盖了数据通信的数百种情况，不过全是围绕一个前提进行的：这些客户端到服务器的附加请求都是在页面完全载入之后才提交的。这使得应用程序的开发者让用户在传统应用程序的慢速交互方式之外有新式交互方式可选择。图1-3是Garrett的有关Ajax文章中的插图，展示了应用程序中的交互数据流是如何随着后台执行的附加请求而改变的（用户很有可能不需要知道这些请求）。

Garrett文章的发表激起了用户、开发者、设计师和管理者的兴趣，从而催化了这类利用高级交互手段的应用程序的大量涌现。具有讽刺意味的是，尽管兴趣现在才被激发起来，但Ajax背后的技术其实并不新颖（早在2000年就已投入商业使用）。最大的区别在于，原来的应用程序用来和服务器通信的技术都是针对个别浏览器的（比如只有IE才支持的特性）。自从所有的现代浏览器都支持XMLHTTPRequest（从服务器收发XML数据的主要手段）以后，它们的地位就平等了，允许所有人都能享受这一技术带来的便利。

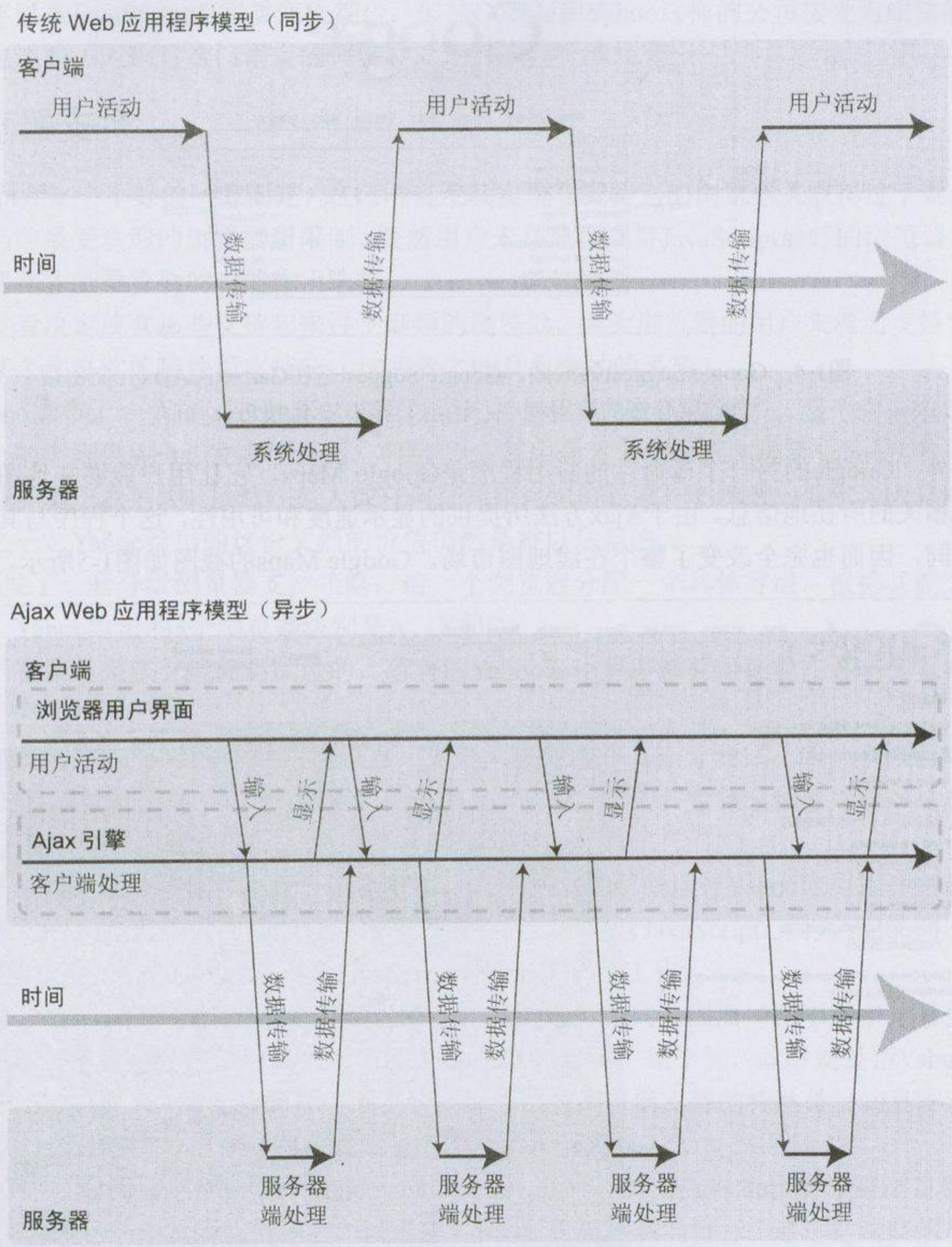


图1-3 来自*Ajax: A New Approach to Web Applications*一文，展示了先进的客户机与服务器间的异步交互

如果说有一个公司始终走在利用Ajax技术创造优秀应用程序的最前沿，这个公司无疑就是Google。在Garrett的Ajax文章快发表之前出现的一个高度可交互的例子是Google Suggest，这个示例允许你输入查询，即时得到自动补全，这样的特性用旧式页面刷新方式是永远不可能办到的。Google Suggest的截图可见图1-4。

图1-4　Google Suggest的截图。Google Suggest是在Garrett的Ajax一文出现之前就存在的应用程序，运用了异步XML技巧

此外，Google的另一个革命性的应用程序是Google Maps，它让用户能够在地图中移动，实时查看相关的所在地信息。由于Ajax方法所提供的显示速度和可用性，这个程序与其他地图程序截然不同，因而也完全改变了整个在线地图市场。Google Maps的截图如图1-5所示。

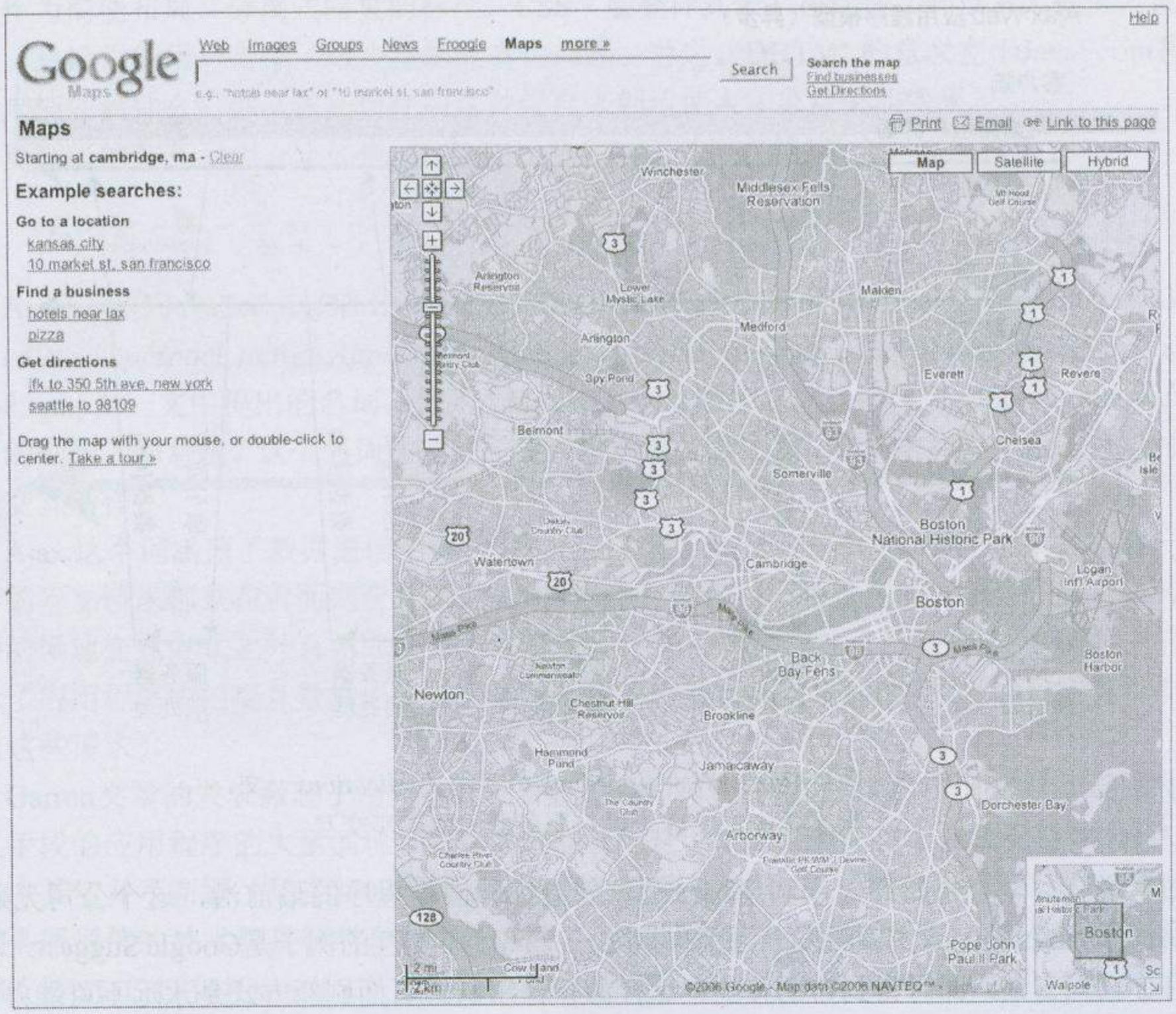


图1-5　Google Maps，运用了一系列Ajax技巧来动态载入位置信息

尽管多年以来JavaScript的实质变化甚少，但被Google和Yahoo这样的公司接受为成熟的编程环境，从而也说明了人们对这门语言的理解和欢迎有了巨大的转变。

12

1.6 浏览器支持

JavaScript开发中不好的一方面是，这门语言及其实现与支持它的浏览器关联得过于紧密，因而必然受到当前最受欢迎的浏览器所限制。既然用户未必选用支持JavaScript最好的浏览器，我们也就被迫对哪些才是最重要的功能作出选择。

许多开发者决定放弃那些支持起来过于麻烦的浏览器。因为浏览器的用户来源而支持它，或者因为它有某个你喜欢的特性而支持它，这两者之间具有微妙的平衡。

最近Yahoo发布了一个JavaScript库，它可以用来扩展你的Web应用程序。整个库中规定了一些设计模式上的守则供Web开发者们遵循，（我认为）其中最重要的一份文档是Yahoo官方关于它支持和不支持的浏览器列表。虽然任何人或任何公司都可以开这么一份列表，但作为因特网上最繁忙的网站之一，Yahoo给出的这份文档的价值仍然是难以估量的。

13

Yahoo开发了一套分级浏览器支持策略：给一个浏览器分配一个具体等级，根据其能力提供不同的内容。Yahoo给出的3个浏览器级别是A、X和C。

- A级浏览器是完整支持并测试过的，所有的Yahoo应用程序都要保证能在这些浏览器上工作。
- X级浏览器是Yahoo已知的A级浏览器，但尚未有能力对其充分测试，或一个以前未曾出现过的新浏览器。给X级浏览器提供的内容是和A级一样的，希望它们能处理好这些比较高级（复杂）的内容。
- C级浏览器通常被称作“不好”的浏览器，它们不支持那些运行Yahoo应用程序必需的特性。给这些浏览器提供的功能性应用程序内容应该不包含JavaScript，因为Yahoo的应用程序都是完全分离式的（即在不含JavaScript的情形下仍能工作）。

巧合的是，Yahoo的分级浏览器选择正好和我自己的选择相同，因而（在本书中）显得非常有说服力。在本书里我会经常使用现代浏览器（modern browser）这个词，此时就是指Yahoo浏览器支持图表判定的、具有A级支持的那些浏览器。因为这些浏览器支持的特性是已知且固定的，学习和开发经验会有趣快乐得多（因为避免了浏览器的不兼容性）。

强烈推荐你仔细阅读在http://developer.yahoo.com/yui/articles/gbs/gbs.html的分级浏览器支持文档，包括图1-6展示的浏览器支持图表，以了解Yahoo所希望达到的目标。通过把这些信息提供给Web开发大众，Yahoo设置了一套让所有人遵循的“黄金标准”，这是一件很棒的事情。

要了解更多关于浏览器支持的信息，可以看本书的附录C，在那里详细讨论了每个浏览器的优缺点。通常你会发现A级浏览器总处在开发的前列，能给你的开发提供许多额外的功能。

在选择支持的浏览器时，归根结底选择的是你的程序希望提供什么特性。如果你希望支持Netscape Navigator 4或者IE 5，这会严重限制程序里可以使用的功能，因为这些浏览器都缺乏对现代编程技术的支持。

14

	Win 98	Win 2000	Win XP	Mac 10.0	Mac 10.2	Mac 10.3	Mac 10.3.x	Mac 10.4
IE 7.0	n/a	n/a	A-grade	n/a	n/a	n/a	n/a	n/a
IE 6.0	A-grade	A-grade	A-grade	n/a	n/a	n/a	n/a	n/a
IE 5.5	A-grade	A-grade	n/a	n/a	n/a	n/a	n/a	n/a
IE 5.0	C-grade	C-grade	n/a	C-grade	C-grade	C-grade	C-grade	C-grade
Netscape 8.0	X-grade	X-grade	A-grade	n/a	n/a	n/a	n/a	n/a
Firefox 1.5	A-grade	A-grade	A-grade	A-grade	A-grade	A-grade	A-grade	A-grade
Firefox 1.0.7	A-grade	A-grade	A-grade	A-grade	A-grade	A-grade	A-grade	A-grade
Mozilla 1.7.12	X-grade	X-grade	A-grade	X-grade	X-grade	X-grade	X-grade	X-grade
Opera 8.5	X-grade	X-grade	A-grade	C-grade	C-grade	C-grade	X-grade	X-grade
Safari 1.0	n/a	n/a	n/a	X-grade	n/a	n/a	n/a	n/a
Safari 1.1	n/a	n/a	n/a	X-grade	X-grade	n/a	n/a	n/a
Safari 1.2	n/a	n/a	n/a	X-grade	X-grade	X-grade	n/a	n/a
Safari 1.3	n/a	n/a	n/a	n/a	n/a	X-grade	A-grade	n/a
Safari 2.0	n/a	n/a	n/a	n/a	n/a	n/a	n/a	A-grade

图1-6　Yahoo提供的分级浏览器支持表

尽管如此，了解哪些浏览器比较先进有助于运用它所支持的强大特性，作为进一步开发的基础。这个一致的基础是由以下这几套特性定义的：

- Core JavaScript 1.5：这是最新被接受的JavaScript版本，支持完整的面向对象JavaScript特性就靠它了。IE 5.0不完全支持1.5，所以开发者们多半不喜欢提供这个浏览器的支持。
- XML文档对象模型（DOM）2：遍历HTML和XML文档的标准。它对编写快速的应用程序非常重要。
- XMLHttpRequest：Ajax的核心——发起远程HTTP访问的简单一层，除IE 5.5～6.0以外，所有的现代浏览器都默认支持这种对象，不过这两个浏览器可以用ActiveX创建一个兼容的对象来支持它。
- CSS：设计网页的必备。这看起来似乎是个奇怪的要求，但对Web应用程序开发者来说，支持CSS是至关重要的。考虑到所有的现代浏览器都能支持CSS，问题就可以简化为它们在呈现时的区别上，大部分的问题都由此而生。这也是为什么IE的Mac版本很少得到支持的原因。

15

1.7　小结

本书尝试涵盖所有现代的、专业的JavaScript编程技巧，从独立开发人员到大型公司成员在内的所有人，都可以用这些技巧来让代码更可用、可读，也更易交互。

这一章里我们完成了本书所有需要讨论内容的一个简短概览，包括专业JavaScript编程的基础：编写面向对象的代码、测试代码和打包分发。然后讲解了分离式DOM脚本编写的基础知识，包括对文档对象模型的概览、事件，以及CSS与JavaScript之间的交互。最后你了解到了Ajax的基础和现代浏览器中JavaScript的实现。总而言之，这些内容足够将你带入专业JavaScript程序员的行列了。

Part 2

第二部分

专业 JavaScript 开发

本部分内容

- 第 2 章　面向对象的 JavaScript
- 第 3 章　创建可重用代码
- 第 4 章　调试与测试的工具

第2章

面向对象的JavaScript

对象（object）是组成 JavaScript 的基本单元，事实上，JavaScript 中的一切都是对象，而且充分发挥了这一点。不过 JavaScript 还包含了构成一门丰富坚实的面向对象语言的大量特性，这使它无论是在功能还是在风格上都显得极为独特。

本章将从介绍 JavaScript 语言中最重要的几个部分开始：引用（reference）、作用域（scope）、闭包（closure）以及上下文（context），而这些内容正是其他 JavaScript 图书很少提及的。有了这些重要的基础知识之后，我们就可以开始探索面向对象的 JavaScript 重要的特性了，包括对象的如何行为、如何创建新的对象和设置不同权限的方法（method）。严格地说，这恐怕是全书最重要的一章了，因为它会彻底改变你对 JavaScript 这门语言的看法。

2.1 语言特性

和许多其他语言不同，JavaScript 之所以成为 JavaScript，它的大量语言特性起到了至关重要的作用。我个人觉得这样的特性搭配得恰到好处，构成了这样一门比想象中还要强大的语言。

2.1.1 引用

引用（reference）的概念是 JavaScript 的基础之一，“引用”是一个指向对象实际位置的指针。这是一个极为强大的特性，但有一个前提：实际的对象肯定不会是引用。字符串永远是字符串，数组永远是数组。不过多个变量却能够指向同一对象。JavaScript 基于的就是这样一个引用系统。这门语言通过维护一系列对其他对象的引用，提供了极大的灵活性。

此外，对象可以包含一系列属性（property），这些属性也都不过是到其他对象（比如字符串、数字、数组等等）的引用。如果多个变量指向的是同一个对象，那该对象的类型一改变，所有这些变量也会跟着相应改变。在代码清单 2-1 中可以看到这样的一个例子：两个变量都指向同一个对象，而对该对象内容的修改对这两个变量都会产生影响。

代码清单2-1 多个变量引用同一个对象

```
// 将 obj 置为空对象
var obj = new Object();

// objRef 现在是另一个对象的引用
```

```
var objRef = obj;

// 修改原对象的一个属性
obj.oneProperty = true;

// 我们现在看到，这个改变在两个变量中都反映了出来
// （因为它们引用的是同一个对象）
alert( obj.oneProperty === objRef.oneProperty );
```

前文提到，自修改（self-modifying）对象在 JavaScript 中是很少见的。我们可以来看一个最常出现这种情况的实例：用 push()方法来给数组（array）对象添加新的元素。因为 Array 对象实质上是把这些值作为属性保存，所以结果和代码清单 2-1 的情形差不多，即多个变量的内容可以同时被修改（见代码清单 2-2）。

代码清单2-2 自修改对象的例子

```
// 创建一个数组
var items = new Array( "one", "two", "three" );

// 创建数组的一个引用
var itemsRef = items;

// 将一个元素添加到原数组中
items.push( "four" );

// 这两个数组的长度应该是一致的，
// 因为它们指向同一个数组对象
alert( items.length == itemsRef.length );
```

必须记住的是，引用指向的只能是具体的对象，而不是另一个引用。不像 Perl 语言中允许多层引用。JavaScript 里的结果是沿着引用链一直上溯到原来那个对象。代码清单 2-3 是这种情况的一个例子：实际对象已经改变了，但原来指向它的引用仍然保持指向旧的对象。

代码清单2-3 修改对象的引用，同时保持完整性

```
// 将 items 置为字符串的数组
var items = new Array( "one", "two", "three" );

// 将 itemsRef 置为 items 的引用
var itemsRef = items;

// 将 items 置为一个新对象
items = new Array( "new", "array" );

// items 和 itemsRef 现在指向不同的对象了。
// items 指向的是 new Array( "new", "array" )
// itemsRef 指向的是 new Array( "one", "two", "three" )
alert( items != itemsRef );
```

20

最后，让我们来看一个特殊的例子，看似是自修改对象，其结果却产生了一个新的非引用对象。在执行字符串连接操作时，结果总会是一个新的字符串对象，而非原字符串的修改版本（见代码清单 2-4）。

代码清单2-4　修改对象而生成新对象

```
// 将 item 置为一个新的字符串对象
var item = "test";

// itemRef 现在指向同一个字符串对象
var itemRef = item;

// 将一些新的文本接在这个字符串后面
// 注意：这会创建一个新对象，而非修改原对象
item += "ing";

// item 和 itemRef 的值不相等了，因为新的字符串对象已被创建
alert( item != itemRef );
```

如果你刚接触这些，可能会被难以捉摸的引用概念搞晕。不过理解引用究竟是如何运作的，对于写出优秀、整洁的 JavaScript 代码至关重要。下面几个小节里我们将了解的一些特性，虽然不那么新颖或者激动人心，但对于写出好的代码也是很重要的。

2.1.2　函数重载和类型检查

其他面向对象语言如 Java 的一个常见特性是，能够根据传入的不同数量或类型的参数，通过“重载（overload）”函数来发挥不同的功用。尽管这个特性在 JavaScript 中并没有被直接支持，也有很多办法能够实现。

函数重载（function overloading）必须依赖两件事情：判断传入参数数量的能力和判断传入
21 参数类型的能力。我们先来看看参数的数量。

JavaScript 的每个函数都带有一个仅在这个函数范围内作用的变量（contextual variable）称为参数（argument），它是一个包含所有传给函数的参数的伪数组（pseudo-array），所以它并非真正意义的数组（也就是说你不能修改它，也不能用 `push()` 来添加新元素），但可以访问其中的元素，它也具有 `.length` 属性。代码清单 2-5 是它的两个例子。

代码清单2-5　JavaScript 中函数重载的两个例子

```
// 发送一条消息的简单函数
function sendMessage( msg, obj ) {
    // 如果消息和对象（的参数）都被提供
    if ( arguments.length == 2 )
        // 给对象发送消息
        obj.handleMsg( msg );

    // 否则，假定只提供了一条消息
    else
        // 那么仅显示默认的错误信息
        alert( msg );
}

// 仅用一个参数调用这个函数 - 用 alert 来显示此消息
sendMessage( "Hello, World!" );
```

```
// 又或者我们可以将一个我们自己写好的对象传入
// 负责用另一套办法显示信息
sendMessage( "How are you?", {
    handleMsg: function( msg ) {
        alert( "This is a custom message: " + msg );
    }
});

// 一个接受任意数量参数并将其转换为数组的函数
function makeArray() {
    // 临时使用的数组
    var arr = [];

    // 遍历传入的每个参数
    for ( var i = 0; i < arguments.length; i++ ) {
        arr.push( arguments[i] );
    }

    // 返回结果数组
    return arr;
}
```

22

此外，虽然显得有点怪，但是还有一个用来判断传入参数类型的方法。如果没有提供参数，它的类型必为 undefined，我们要利用这一特性来作判断。代码清单 2-6 展示了这样一个简单函数，它用于显示错误信息，如果没有提供信息的内容，就显示默认的一条。

代码清单2-6 显示错误信息和默认信息

```
function displayerror( msg ) {
    // 检查并确认 msg 是否 undefined
    if ( typeof msg == 'undefined' ) {
        // 如果是，则置 msg 为默认信息
        msg = "An error occurred.";
    }

    // 显示该消息
    alert( msg );
}
```

typeof 语句的应用为我们引入了下一个话题：类型检查。既然 JavaScript（现在而言）是一个动态类型（dynamically typed）的语言，类型检查必然是个非常有用而且重要的话题。有许多方法可以检查变量的类型，但我们这里只讨论两种特别有用的方法。

第一种方法是使用显而易见的 typeof 操作符。这个工具提供了一个字符串名称（string name），用于表达变量内容的类型。当变量不是 object 或者 array 类型时，这应该算是最完美的解决方法了。但是对于自定义的对象，比如 user 就不能用这个方法进行类型检查，因为它只会返回 object，很难跟其他的 object 区分开来。这一方法的一个例子如代码清单 2-7 所示。

代码清单2-7 使用typeof 来判断对象类型的一个例子

```
// 检查我们的数字是否实际上是字符串
if ( typeof num == "string" )
```

```
    // 若是，则根据这个字符串解析出整数来
    num = parseInt( num );

// 检查我们的数组是否实际上是字符串
if ( typeof arr == "string" )
    // 若是，则根据逗号切分出数组来
    arr = arr.split(",")
```

第二种检查对象类型的方法，需要引用所有 JavaScript 对象都带有的一个的属性，称为构造函数（constructor）。这一属性引用的是原本用来构造该对象的那个函数。这种方法的一个例子如
23 代码清单 2-8 所示。

代码清单2-8　使用构造函数属性来判断对象的类型

```
// 检查我们的数字实际上是否为字符串
if ( num.constructor == String )
    // 如果是，则根据这个字符串解析出整数来
    num = parseInt( num );

// 检查我们的字符串实际上是否位数组
if ( str.constructor == Array )
    // 如果是，则根据数组用逗号归并出字符串来
    str = str.join(',');
```

表 2-1 展示了用上述的两种方法对不同类型对象进行类型检查的结果。表格的第一栏是尝试判断类型的对象。第二栏是执行 typeof 变量的结果（其中变量是第一栏的值）。这一栏的所有结果都是字符串。第三栏展示的是执行对第一栏的这些变量运行变量.构造函数的结果，这一栏的所有结果都是对象。

表 2-1　变量的类型检查

变　量	typeof 变量	变量.构造函数
{ an: "object" }	object	Object
["an", "array"]	object	Array
function() {}	function	Function
"a string"	string	String
55	number	Number
true	boolean	Boolean
new User()	object	User

现在你可以用表 2-1 提供的信息来构造一个通用的类型检查函数。很明显，把变量的构造函数作为对象类型引用恐怕是最不容易犯错的合法类型检查了。严格的类型检查可以帮助你判断，传入函数的参数是否有正确的数量和正确的类型，我们可以在代码清单 2-9 看到一个这样的例子。

代码清单2-9　一个函数，可以用来严格维护传入函数的所有参数

```
// 用一个变量类型列表严格检查一个参数列表
function strict( types, args ) {

    // 保证类型的数量和参数的数量相匹配
```

```
    if ( types.length != args.length ) {
        // 否则抛出一个有用的异常
        throw "Invalid number of arguments. Expected " + types.length +
            ", received " + args.length + " instead.";
    }

    // 遍历所有的参数，检查它们的类型
    for ( var i = 0; i < args.length; i++ ) {
        if ( args[i].constructor != types[i] ) {
            throw "Invalid argument type. Expected " + types[i].name +
                ", received " + args[i].constructor.name + " instead.";
        }
    }
}

// 一个简单的函数，打印用户列表
function userList( prefix, num, users ) {
    // 保证 prefix 是字符串，num 是数字，users 是数组
    strict( [ String, Number, Array ], arguments );

    // 遍历 'num' 个用户
    for ( var i = 0; i < num; i++ ) {
        // 显示每个用户的信息
        print( prefix + ": " + users[i] );
    }
}
```

事实上，变量类型的检查和检验参数数组的长度都是简单的概念，但可以用来提供一些可使用的复杂方法，能给开发者和使用者带来更好的体验。下面我们将看看 JavaScript 中的作用域（scope）的概念，以及如何更好去控制它。

2.1.3 作用域

作用域（scope）是 JavaScript 中一项让人感到棘手的特性。所有的面向对象编程语言都有某种形式的作用域，不过和把这个概念放在什么上下文中有关。在 JavaScript 里，作用域是由函数划分的，而不是由块（block）划分（比如 `while`，`if` 和 `for` 语句中间）的。这样导致的结果是某些代码不好理解（如果你曾经使用过用块划分域的语言）。代码清单 2-10 展示了一个例子，是根据函数划分作用域而来的代码。

代码清单2-10 展示JavaScript的变量作用域的例子

```
// 设置全局变量 foo，并置为 "test"
var foo = "test";

// 在 if 块中
if ( true ) {
    // 将 foo 置为 'new test'
    // 注意：现在还在全局作用域中！
    var foo = "new test";
}
```

```
// 如我们所见，现在 foo 等于 'new test' 了
alert( foo == "new test" );

// 创建一个会修改变量 foo 的新函数
function test() {
    var foo = "old test";
}

// 然而在调用时，foo 只在函数作用域内起作用
test();

// 这里确认了 foo 还是等于 'new test'
alert( foo == "new test" );
```

可以看到，在代码清单 2-10 中，变量都在全局作用域里。基于浏览器的 JavaScript 的一个有趣的特性是，所有属于全局作用域的变量其实都是 window 对象的属性（property）。尽管某些早期版本的 Opera 和 Safari 并非如此，但还是可以大致认为浏览器都遵循此规则。代码清单 2-11 展示了这种全局作用域的出现。

代码清单2-11　JavaScript 中的全局作用域和 window 对象

```
// 一个全局作用域下的变量，存储了字符串 'test'
var test = 'test';

// 你可以发现我们的全局变量和 window 对象的 test 属性是一致的
alert( window.test == test );
```

最后，让我们看看当变量缺乏声明时会是什么情况。在代码清单 2-12 中，对变量 foo 的赋值是在函数 test()的作用域中进行的，然而整个例子里并没有哪个作用域实际声明了这个变量。如果变量没有显式定义，它就是全局定义的，虽然它可能只在这个函数作用域的范围内使用。

代码清单2-12　隐式全局作用域的变量声明

```
// 一个设置了 foo 值的函数
function test() {
    foo = "test";
}

// 调用此函数以设置 foo 的值
test();

// 我们发现 foo 现在是在全局作用域下
alert( window.foo == "test" );
```

虽然 JavaScript 中的作用域规则不如块级作用域语言那么严格，但它还是非常强大和功能完备的。尤其是在和下一节讨论的闭包概念一起使用时，JavaScript 就能表现出脚本语言的强大本色。

2.1.4　闭包

闭包（closure）意味着内层的函数可以引用存在于包围它的函数内的变量，即使外层函数的执行已经终止。这个特性非常强大和复杂。强烈推荐你访问本节结尾所提及的站点，那里有关于

闭包的许多精彩资料。

让我们先来看看闭包的两个简单例子，如代码清单 2-13 所示。

代码清单2-13 闭包如何使代码更清晰的两个例子

```
// 找出 ID 为 'main' 的元素
var obj = document.getElementById("main");

// 修改它的 border 样式
obj.style.border = "1px solid red";

// 初始化一个在一秒后执行的回调函数 (callback)
setTimeout(function(){
    // 它将隐藏此对象
    obj.style.display = 'none';
}, 1000);

// 一个用于延时显示警告信息的通用函数
function delayedAlert( msg, time ) {
    // 初始化一个封装的回调函数
    setTimeout(function(){
        // 它将使用包含本函数的外围函数传入的 msg 变量
        alert( msg );
    }, time );
}

// 用两个参数调用 delayedAlert 函数
delayedAlert( "Welcome!", 2000 );
```

27

第一个对 setTimeout 的函数调用是新 JavaScript 开发者常犯错误之处，在新手的程序里容易看到这样的代码：

```
setTimeOut("otherFunction()", 1000);

// 或
setTimeout("otherFunction(" + num + "," + num2 + ")", 1000);
```

完全可以运用闭包的概念来避免这种糟糕的代码，第一个例子很简单，不过是个在注册后 1000 毫秒后发生的 setTimeout 回调函数，执行的时候还引用了 obj 变量（这肯定是全局变量，因为它连 ID 都有，ID 为 main）。第二个函数 delayedAlert 展示了解决 setTimeout 混乱的一个办法，并指明可以在函数作用域内使用闭包。

可以发现，在代码里使用这样的简单闭包功能后，原本乱成一锅粥的代码变得清晰多了。

让我们看看用闭包能实现什么有趣的额外作用。在一些函数式程序设计语言里，有一种称为 Curry 化[①]（currying）的技术。本质上，Curry 化是一种通过把多个参数填充到函数体中，实现将函数转换为一个新的经过简化的（使之接受的参数更少）函数的技术。代码清单 2-14 是 Curry 化的一个简单的例子，它通过向另外一个函数预填参数而创建了一个新函数。

① Curry化是以逻辑学家Haskell Curry命名的技术，但这种技术是Moses Schönfinkel和Gottlob Frege发明的。——译者注

代码清单2-14　用闭包实现的函数 Curry 化

```
// 数字求和函数的函数生成器
function addGenerator( num ) {
    // 返回一个简单的函数，求两个数字的和，其中第一个数字来自生成器
    return function( toAdd ) {
        return num + toAdd
    };
}

// addFive 现在包含一个接受单一参数的函数，这个函数能求得 5 加上该参数的和
var addFive = addGenerator( 5 );

// 这里我们可以看到，在传入参数为 4 时，addFive 函数的结果是 9
alert( addFive( 4 ) == 9 );
```

闭包还能解决另一个常见的 JavaScript 编写问题。JavaScript 开发新手经常会留下大量多余的
28 全局变量。而这通常被认为是一个坏习惯，因为这些多余的变量可能会悄悄地影响其他的库，导致怪异问题的出现。通过自执行的匿名函数你可以把所有原本属于全局的变量都隐藏起来，如代码清单 2-15 所示。

代码清单2-15　使用匿名函数来隐藏全局作用域变量的例子

```
// 创建一个新的匿名函数，作为包装
(function(){
    // 变量原本应该是全局的
    var msg = "Thanks for visiting!";

    // 将一个新函数绑定到全局对象
    window.onunload = function(){
        // 这个函数使用了“隐藏”的 msg 变量
        alert( msg );
    };

    // 关闭匿名函数并执行之
})();
```

最后，让我们看看使用闭包会遇到的一个问题。你应该记得，闭包允许你引用父函数中的变量，但提供的值并非该变量创建时的值，而是在父函数范围内的最终值。你会看到，这样带来的最常见的问题是在 for 循环中，有一个变量作为循环计数（比如 i），在这个循环里创建了新的函数，利用闭包来引用循环的计数器。问题是，在这个新的闭包函数被调用时，它引用的计数器值是其最后一次的赋值（比如数组的最后一个位置），而不是你期望的那个值。代码清单 2-16 展示了一个使用匿名函数来激发出作用域的例子，用实例证明了上面所期望的闭包效果是可以办到的。

代码清单2-16　使用匿名函数来激发出创建多个使用闭包的函数所需的作用域

```
// 一个 ID 为 main 的元素
var obj = document.getElementById("main");
```

```
// 用于绑定的一个数组
var items = [ "click", "keypress" ];

// 遍历数组的每个成员
for ( var i = 0; i < items.length; i++ ) {
    // 使用一个自执行的匿名函数来激发出作用域
    (function(){
        // 记住在这个作用域内的值
        var item = items[i];
        // 将一个函数绑定到该元素
        obj[ "on" + item ] = function() {
            // item 引用本 for 循环上下文所属作用域中的一个父变量
            alert( "Thanks for your " + item );
        };
    })();
}
```

29

闭包的概念不容易掌握，我花了大量的时间和精力才完全理解闭包的强大功能。幸运的是，现在已经有一篇精彩的文章解释 JavaScript 的闭包是如何工作的：Jim Jey 的 *JavaScript Closures* 一文，可以在 http://jibbering.com/faq/faq_notes/closures.html 中找到。

最后我们将看看上下文对象（context）的概念，这是构筑 JavaScript 面向对象功能的基础。

2.1.5 上下文对象

在 JavaScript 中，你的代码总是有一个上下文对象（代码处在该对象内）。这是面向对象语言的常见特点，但其他语言没有 JavaScript 发挥得那么极致。

上下文对象是通过 `this` 变量体现的，这个变量永远指向当前代码所处的对象中。回忆一下，全局对象其实是 `window` 对象的属性。这意味着即使是在全局上下文中，`this` 变量也能指向一个对象。上下文对象可以称为一个强大的工具，在面向对象代码中也是一个必备的工具。代码清单 2-17 展示了上下文对象的一些简单例子。

代码清单2-17 在上下文对象内使用函数并将其上下文对象切换为另一个变量

```
var obj = {
    yes: function(){
        // this == obj
        this.val = true;
    },
    no: function(){
        this.val = false;
    }
};

// 我们发现 'obj' 对象没有 val 属性
alert( obj.val == null );

// 执行了 yes 函数后，将 val 属性与 'obj' 对象关联起来
obj.yes();
alert( obj.val == true );
```

30

```
// 不过现在把 window.no 指向 obj.no 并执行之
window.no = obj.no;
window.no();

// 结果是 obj 对象的 val 不变（因为 no 的上下文已经改变为 window 对象了）
alert( obj.val == true );

// 而 window 的 val 属性被更新了
alert( window.val == false );
```

你可能注意到了，在代码清单 2-17 中，我们把 obj.no 变量的上下文对象切换为 window 变量时，代码变得不好理解了。幸运的是，JavaScript 提供了一套方法来让这一过程变得更好理解和实现。代码清单 2-18 展示了 call 和 apply 两个方法，可以用于实现这一功能。

代码清单2-18　修改函数上下文对象的例子

```
// 一个设置上下文对象颜色样式的简单函数
function changeColor( color ) {
    this.style.color = color;
}

// 在 window 对象中调用此函数会失败，因为 window 对象没有 style 属性
changeColor( "white" );

// 找出 ID 为 main 的文档
var main = document.getElementById("main");

// 使用 call 方法将它的颜色置为黑色。call 方法将上下文对象设置为第一个参数，
// 并将其他参数作为原函数的参数
changeColor.call( main, "black" );

// 设置 body 元素颜色的函数
function setBodyColor() {
    // apply 方法将上下文对象设置为第一个参数指定的 body 元素，第二个参数是
    // 传给函数的所有参数的数组
    changeColor.apply( document.body, arguments );
}

// 将 body 的背景色置为黑色
setBodyColor( "black" );
```

虽然现在上下文对象的用途还不那么明显，但在下一节的面向对象的 JavaScript 后就会显现出来。

2.2　面向对象基础

面向对象 JavaScript 这个词其实有些多余，因为 JavaScript 这门语言就是完全面向对象的，也不可能以非面向对象的方法来使用。不过大多数编程新手（包括使用 JavaScript 的）的常见弱点在于按照功能编写代码，而不考虑任何上下文或者组织。要完整理解如何编写最优化的 JavaScript 代码，就必须理解 JavaScript 对象是如何工作的，它们和其他语言的对象有何不同，以及怎样使用才对你有益。

本章的后半部分里，我们将介绍编写面向对象 JavaScript 代码的基础，然后在后续章节讨论以这种方式编写代码的可行性。

2.2.1 对象

对象是 JavaScript 的基础。事实上，这门语言里所有的东西都是对象。这门语言的大部分功用都是基于这一点的。从最基本的层次上说，对象是一系列属性的集合，和其他语言里的散列表结构类似。代码清单 2-19 是展示创建有多个属性的对象的两个例子。

代码清单2-19 两个创建简单对象并设置属性的例子

```
// 创建一个新的 Object 对象，存放在 'obj' 变量中
var obj = new Object();

// 给此对象设置一些属性
obj.val = 5;
obj.click = function(){
    alert( "hello" );
};

// 这是一段等价代码，用 {...} 简写方式，结合键值对（key/value pair）来定义属性
var obj = {

    // 用键值对方式来设置属性名和属性值
    val: 5,
    click: function() {
        alert( "hello" );
    }
};
```

2.2.2 对象的创建

和大部分的其他面向对象语言不同的是，JavaScript 并没有类（class）的概念。其他面向对象语言中你大多需要实例化某个具体类的实例，但 JavaScript 里不用。JavaScript 里对象本身可以用来创建新对象，而对象也可以继承自其他对象。这个概念称为原型化继承（prototypal inheritance），会在之后的“公共方法”一节作更详细地讨论。

32

不过不管 JavaScript 使用何种对象方案，首先还是应该有一种创建新对象的方法的。JavaScript 的做法是，任何函数都可以被实例化为一个对象。实际上，这个方法用起来并没有听起来这么令人迷惑。这很像把一块面团放进烤甜饼模具里，再切成一块一块的，其中面团是原对象，模具就是使用对象原型的构造函数。

让我们看看代码清单 2-20 里的例子，以了解它是如何运作的。

代码清单2-20 简单对象的创建和使用

```
// 一个简单的函数，接受名称并将其存入当前上下文中
function User( name ) {
    this.name = name;
}
```

```
// 指定名称来创建该函数的一个新对象
var me = new User( "My Name" );

// 我们可以看到，这个对象的名称被设为自身的 name 属性了
alert( me.name == "My Name" );

// 而且这是 User 对象的一个实例
alert( me.constructor == User );

// 现在，既然 User() 不过是个函数，
// 如果只把它作为函数来使用又如何呢？
User( "Test" );

// 因为它的 'this' 上下文对象未曾设定，所以默认为全局的 'window' 对象，
// 也就说 window.name 等于提供的这个名字
alert( window.name == "Test" );
```

代码清单 2-20 展示了 constructor 属性的使用，这一属性在每个对象中都存在，并一直指向创建它的函数。这样一来你就可以有效地复制对象了，用同一个基类创建对象并赋予不同的属性。这种方式的一个例子如代码清单 2-21 所示。

代码清单2-21　使用constructor属性的例子

```
// 创建一个新的简单的 User 对象
function User() {}

// 创建一个 User 对象
var me = new User();

// 还是创建一个新的 User 对象（用前一个对象的 constructor 引用来创建）
var you = new me.constructor();

// 你可以发现这两个对象的 constructor 实质上是一致的
alert( me.constructor == you.constructor );
```

33

现在我们知道如何创建简单对象了，是时候添加一些让对象更有用的东西了——上下文相关方法（contextual method）和属性。

1. 公共方法

公共方法（public method）在对象的上下文中是最终用户始终可以接触到的。要实现这种在对象的每个实例中都可以使用的公共方法，必须了解一个叫 prototype（原型）的属性，这个属性包含了一个对象，该对象可以作为所有新副本的基引用（base reference）。本质上说，所有对象原型的属性都能在该对象的每个实例中找到。这种创建/引用的过程带来了一种继承的简单版本，我们会在第 3 章讨论它。

因为对象的原型仍然是对象，和其他任何对象一样，你也可以给它们添加新的属性。给原型添加属性的结果是由该原型实例化的每个对象都会获得这些属性，也就使这些属性公有化了（能被所有对象访问）。代码清单 2-22 是一个这样的例子。

代码清单2-22 对象的方法通过 prototype 对象添加的例子

```
// 创建一个新的 User 构造函数
function User( name, age ){
    this.name = name;
    this.age = age;
}

// 将一个新的函数添加到此对象的 prototype 对象中
User.prototype.getName = function(){
    return this.name;
};

// 并再给此 prototype 对象添加一个函数,
//注意其上下文是实例化后的对象
User.prototype.getAge = function(){
    return this.age;
};

// 实例化一个新的 User 对象
var user = new User( "Bob", 44 );

// 可以看到我们添加的这两个属性都在刚才创建的对象中，并且有合适的上下文
alert( user.getName() == "Bob" );
alert( user.getAge() == 44 );
```

34

目前大多数 JavaScript 开发者开发新应用程序时已经掌握简单的构造函数和简单的原型对象处理了。在本节后半部分，将介绍一系列让你充分发挥面向对象代码功用的其他技巧。

2. 私有方法

私有方法（private method）和私有变量只允许其他的私有方法、私有变量和特权方法（下一小节讨论）访问。这种方法可以定义一些只让对象内部访问，而外部访问不到的代码，这一技巧来自 Douglas Crockford 的努力，他的网站提供了大量的文档，详细介绍面向对象的 JavaScript 如何运作及应该如何使用：

- JavaScript 文章一览：http://javascript.crockford.com/。
- *Private Members in JavaScript*一文：http://javascript.crockford.com/private.html。

现在让我们来看一个在应用程序中使用私有方法的例子，如代码清单 2-23 所示。

代码清单2-23 只能由构造函数访问的私有方法的例子

```
// 表示教室的一个对象构造函数
function Classroom( students, teacher ) {
    // 用于显示所有班上学生的私有方法
    function disp() {
        alert( this.names.join(", ") );
    }

    // 将班级数据存入公共对象属性中
    this.students = students;
    this.teacher = teacher;
```

```
    // 调用私有方法来显示错误
    disp();
}

// 创建一个新的 classroom 对象
var class = new Classroom( [ "John", "Bob" ], "Mr. Smith" );

// 调用disp方法会失败，因为它不是该对象的公共属性
class.disp();
```

35

虽然很简单，但私有方法和私有变量在保证代码没有冲突的同时，允许你对用户能使用和能看到的内容有更好的控制。下一步，我们将了解特权方法，它是私有方法和公共方法的混合体。

3. 特权方法

特权方法（privileged method）是 Douglas Crockford 采用的一个名词，用来指代那些在查看并处理（对象中）私有变量的同时允许用户以公共方法的方式访问的方法。代码清单 2-24 展示了使用特权方法的一个例子。

代码清单2-24　使用特权方法的例子

```
// 创建一个新的 User 对象构造函数
function User( name, age ) {
    // 尝试算出用户出生的年份
    var year = (new Date()).getFullYear() - age;

    // 创建一个新的特权方法，能够访问 year 变量，同时自身属于公共可访问的
    this.getYearBorn = function(){
        return year;
    };
}

// 创建 User 对象的一个新实例
var user = new User( "Bob", 44 );

// 验证返回的年份正确
alert( user.getYearBorn() == 1962 );

// 注意我们无法访问该对象私有的年份属性

alert( user.year == null );
```

本质上，特权方法是动态生成的，因为它们是在运行时才添加到对象中的，而不是在代码第一次编译时就已经生成的。虽然这个技巧要比往对象的 prototype 上绑定一个简单的方法开销更大，但功能也更强大、更灵活。代码清单 2-25 是动态生成方法能实现的一个例子。

36

代码清单2-25　动态生成方法的例子，这些方法在新对象实例化时创建[①]

```
// 创建一个新的用户对象，接受一个有许多属性的对象作为参数
function User( properties ) {
    // 遍历该对象的所有属性，并保证其作用域正确（如前面所述）
```

① 原代码没有正确设置上下文与闭包调用对象（caller object）的变量，所以不会正确执行，这里给出的是修改后的代码。——译者注

```
    for ( var i in properties ) { (function(which){
        var p = i;
        // 创建此属性的一个新的读取器 (getter)
        which[ "get" + p ] = function() {
            return properties[p];
        };

        // 创建此属性的一个新的设置器 (setter)
        which[ "set" + p ] = function(val) {
            properties[p] = val;
        };
        })(this);
    }
}

// 创建一个新的用户对象实例，并把具有两个属性的一个对象传入作为种子
var user = new User({
    name: "Bob",
    age: 44
});

// 注意 name 属性并不存在，
// 因为它是属性对象（properties object）的私有变量
alert( user.name == null );

// 不过我们可以使用新的 getname() 方法来获得这个值
// 因为此函数是动态生成的
alert( user.getname() == "Bob" );

// 最后，我们看到能够使用这个新生成的函数来设置或获得年龄
user.setage( 22 );
alert( user.getage() == 22 );
```

动态生成代码的能力不可小视，能够根据运行时变量来生成代码是非常有用的，这也是其他语言（如 Lisp）里的宏（macro）如此强大的原因，不过现在是以现代程序设计语言的形式来表达罢了。下一步我们将看到一种只在组织代码时有用的方法类型。

37

4. 静态方法

静态方法的实质与任何其他一般函数没有什么不同，最主要的区别在于，其他函数是以对象的静态属性形式存在的。作为一个属性，它们不能在该对象的实例的上下文中访问，而只属于主对象本身的那个上下文中。对习惯了传统类式继承的人来说，这就像类里定义的静态方法。

实际上，这样编写代码的唯一优点是保证对象的命名空间整洁，第 3 章会对这个概念作更详细讨论。代码清单 2-26 展示了一个把静态方法添加到对象中的例子。

代码清单2-26 静态方法的一个简单例子

```
// 添加到一个 User 对象的静态方法
User.cloneUser = function( user ) {
    // 创建并返回一个新的用户
    return new User(
        // 这是其他用户对象的复制
```

```
            user.getName(),
            user.getAge()
        );
    };
```

静态方法是我们遇到的第一种仅为组织代码而使用的方法。由此我们将转入下一章所要讨论的内容。开发出专业 JavaScript 代码的根本方法之一是，快速、静态地提供与其他代码的接口，同时保证自身的可理解性。这是一个值得努力的重要目标，也是我们要在下一章要达到的目标。

2.3　小结

理解本章所描述的这些概念非常重要。本章的前半部分大致讲解了 JavaScript 语言的概念以及如何使用，这是完全掌握专业 JavaScript 使用方法的起点。只要能理解对象的行为、引用的处理和作用域的判断，一定能改变你编写 JavaScript 代码的习惯。

在掌握了聪明的 JavaScript 编写方法后，编写干净的面向对象 JavaScript 代码的重要性就更为明显了。在本章的后半部分介绍了如何编写多种不同的面向对象代码，以适合所有原本使用各种其他编程语言的程序员。这是现代 JavaScript 语言的基础，能为你开发创新的应用程序带来巨大的帮助。

第3章

创建可重用代码

大多数公司或团队项目的标准流程是程序员之间协同开发代码，因此保持良好的编码习惯非常重要。随着 JavaScript 近年来的盛行，专业程序员们参与开发的 JavaScript 代码也急剧增加。他们对 JavaScript 接受与使用的改变让开发习惯也有了长足的进步。

在这一章里，我们将学到一系列容易使用的整理代码、组织代码和改进代码质量的方法。

3.1 标准化面向对象的代码

编写可重用代码的第一步，也是最重要的一步，是使用一套标准贯穿整个程序的所有代码，面向对象的代码尤为如此。在上一章中，当你看到了面向对象的 JavaScript 是什么样子的时候，大概也看到了 JavaScript 巨大的灵活性，可以让你模仿各式各样的编程风格。

首先，开发出一套符合你需求的系统，编写面向对象代码并实现对象继承（将一个对象的属性复制到另一个对象上）是很重要的。不过，看起来每个编写过一些面向对象 JavaScript 的人都有自己的一套实现方法，这就有点令人迷惑的了。在这一节里，我们会看到 JavaScript 中的继承是如何运作的，然后学习一系列可以交替使用的辅助方法，以及如何在你的应用程序里使用它们。

3.1.1 原型式继承

JavaScript 对象的创建与继承使用了一套特别的模式，称作原型式继承（prototypal inheritance）。（和大部分程序员熟悉的经典的类/对象方式相比）这种方法的原理是，对象的构造函数可以从其他对象中继承方法，它创建出一个原型对象后，所有其他的新对象都可以基于这个原型对象来构建。

整个过程都是通过原型属性来实现的（这是每个函数都有的一个属性，用作构造函数的函数自然也会有这个属性）。原型式继承的设计适用于单继承而非多继承。不过还是有方法解决多继承问题的，我会在下一节叙述。

这种继承方式之所以特别难以掌握，是因为原型本身并不会从其他原型或者构造函数中继承属性，而属性都是从实际对象那里继承过来的。代码清单 3-1 展示了几个使用原型属性来实现简单继承的例子。

代码清单3-1　原型式继承的例子

```
// 为 Person 对象创建一个构造函数
function Person( name ) {
    this.name = name;
}

// 给 Person 对象添加一个新方法
Person.prototype.getName = function() {
    return this.name;
};

// 创建一个新的 User 对象的构造函数
function User( name, password ) {
    // 注意，这里并没有支持方便的重载/继承，也就是说，不能调用父类的构造函数
    this.name = name;
    this.password = password;
};

// User 对象继承所有 Person 对象的方法
User.prototype = new Person();

// 我们添加一个新方法到 User 对象中
User.prototype.getPassword = function() {
    return this.password;
};
```

在上面这个例子中最重要的一行是 User.prototype=new Person();。让我们来仔细分析一下这究竟是什么意思：User 是对 User 对象构造函数的引用。new Person()使用 Person 构造函数创建了一个新的 Person 对象，然后把 User 构造函数的原型置为这个操作的结果。也就是说，每当你 new User()时，得到的新 User 对象都会带有 Person 对象所有的方法，如同通过操作 new Person()得到的一样。

了解这个技巧之后，我们可以看看开发者们为了简化 JavaScript 的继承而编写的一系列包装函数（wrapper）。

3.1.2　类式继承

类式继承对于大部分开发者来说都已经熟悉，只要有了带方法（method）的类（class）就可以把它们实例化（instantiate）为对象。JavaScript 面向对象编程新手的典型反应就是去模拟这种程序设计风格，不过却很少有人能找到正确的路子。

40

幸运的是，JavaScript 大师 Douglas Crockford 的目标之一就是，在 JavaScript 里开发一套简单的方法来模拟类式继承，他还在他的网页 http://javascript.crockford.com/inheritance.html 中作出了解释。

代码清单 3-2 展示了他构建的 3 个函数，提供了一套完整实现 JavaScript 类式继承的方法。每个函数分别实现了一种形式的继承：继承自单一函数、从单一父对象继承所有内容，以及从多个父对象继承独立的方法。

代码清单3-2 Douglas Crockford在JavaScript里模拟类式继承的3个函数

```
// 简单的辅助函数，让你可以将新函数绑定到对象的 prototype 上
Function.prototype.method = function(name, func) {
    this.prototype[name] = func;
    return this;
};

// 一个（相当复杂的）函数，允许你方便地从其他对象继承函数，
// 同时仍然可以调用属于父对象的那些函数
Function.method('inherits', function(parent) {
    // 记录我们目前所在父层次的级数
    var depth = 0;

    // 继承父对象的方法
    var proto = this.prototype = new parent();

    // 创建一个新的名为 'uber' 的“特权”函数，
    // 调用它时会执行所有在继承时被重写的函数
    this.method('uber', function uber(name) {

        var func;                       // 要执行的函数
        var ret;                        // 函数的返回值
        var v = parent.prototype;       // 父对象的 prototype

        // 如果我们已经在某个 'uber' 函数之内
        if (depth) {
            // 上溯必要的 depth，以找到原始的 prototype
            for (var i = d; i > 0; i += 1) {
                v = v.constructor.prototype;
            }

            // 从该 prototype 中获得函数
            func = v[name];

        // 否则这就是 'uber' 函数的第一次调用
        } else {
            // 从 prototype 获得要执行的函数
            func = proto[name];

            // 如果此函数属于当前的 prototype
            if ( func == this[name] ) {
                // 则改为调用父对象的 prototype
                func = v[name];
            }
        }

        // 记录我们在继承堆栈中所在位置的级数
        depth += 1;

        // 使用除第一个以外所有的 arguments 调用此函数。
        // （因为第一个参数是执行的函数名）
        ret = func.apply(this, Array.prototype.slice.apply(arguments, [1]));
```

```
        // 恢复继承堆栈
        depth -= 1;

        // 返回执行过的函数的返回值
        return ret;
    });

    return this;
});

// 只继承父对象特定函数的函数。而非使用 new parent() 继承所有的函数
Function.method('swiss', function(parent) {
    // 遍历所有要继承的方法
    for (var i = 1; i < arguments.length; i += 1) {
        // 需要导入的方法名
        var name = arguments[i];

        // 将此方法导入 this 对象的 prototype 中
        this.prototype[name] = parent.prototype[name];
    }

    return this;
});
```

让我们来看看这3个函数到底提供了什么，以及为什么应该就使用它们，而不是自己再去写一套原型式继承模型。这3个函数的用意很简单：

42

- Function.prototype.method：它提供了一个简单的方法，把函数与构造函数的原型关联起来。之所以有效，是因为所有的构造函数本身都是函数，所以能获得“method”这个新方法。
- Function.prototyope.inherits：这一函数可以用于提供简单的单对象继承，它的代码主要围绕在任意对象方法中调用this.uber('methodName')为中心，并在让这个uber方法去执行它要覆盖的父对象的方法。这是JavaScript继承模型中并未内建的部分。
- Function.prototype.swiss：这是.method()函数的增强版，可以用于从单一父对象获取多个函数。如果用在多个父对象上就能能获得可用的多对象继承。

现在你大致了解这些函数能给我们提供什么功能了，那么我们以新的类式继承风格来重写代码清单3-1中的Person/User示例，如代码清单3-3所示。此外，你还能看到这个库提供的一些其他功能，以及如何改进代码可读性的方法。

代码清单3-3　使用Douglas Crockford类式继承风格的JavaScript函数示例

```
// 创建一个新的 Person 对象构造函数
function Person( name ) {
    this.name = name;
}

// 给 Person 对象添加一个新的方法
Person.method( 'getName', function(){
    return name;
});
```

```
// 创建了一个新的 User 对象构造函数
function User( name, password ) {
    this.name = name;
    this.password = password;
},

// 从 Person 对象继承所有方法
User.inherits( Person );

// 给 User 对象添加一个新的方法
User.method( 'getPassword', function(){
    return this.password;
});
```

43

```
// 覆盖 Person 对象创建的 getName 方法，但通过 uber 函数来调用原有方法
User.method( 'getName', function(){
    return "My name is: " + this.uber('getName');
});
```

现在你对这套改进 JavaScript 继承的库有所了解，可以来看看一些其他受欢迎的常用方法了。

3.1.3 Base 库

最近一个改进 JavaScript 对象创建与继承功能的库是 Dean Edwards 开发的 Base 库，它提供了一系列不同的方法，以扩展对象的功能。它甚至还提供了一套比较直观的对象继承方法。Dean 的初始目的是为他的其他项目开发这个 Base 库作为基础，这些项目包括 IE7，这是一套完整的对 IE 功能更新的集合。Dean 的网站给出的例子非常全面，充分地展示了这个库的能力：http://dean.edwards.name/Weblog/2006/03/base/。你还能在 Base 库的源代码目录中找到更多例子：http://dean.edwards.name/base/。

Base 代码很多，也非常复杂，为更好阅读理解，它需要一些额外的注释（包括在 Apress 网站 http://www.apress.com 的 Soure Code/Download 部分）。在读完注释后，我还强烈建议你仔细查看 Dean 在他的网站上提供的例子，它们有助于解释常见的疑问。

不过我首先会带你体验 Base 一些重要的方面，因为它们对开发很有帮助。参见代码清单 3-4，它是创建对象、单父对象继承和覆盖父类方法的例子。

代码清单3-4 用Dean Edwards的Base库进行简单对象创建与继承的例子

```
// 创建一个新的 Person 类
var Person = Base.extend({
    // Person 类的构造函数
    constructor: function( name ) {
        this.name = name;
    },

    Person 类的一个简单方法
    getName: function() {
        return this.name;
    }
```

```
});

// 创建一个新的继承自 Person 类的 User 类
var User = Person.extend({
    // 创建 User 类的构造函数
    constructor: function( name, password ) {
        // 这个函数实际上调用了父类的构造函数
        this.base( name );
        this.password = password;
    },

    // 给 User 类提供了另一个简单的方法
    getPassword: function() {
        return this.password;
    }
});
```

让我们看看 Base 如何实现代码清单 3-4 概括的 3 个目标，以获得一种简单的对象创建与继承方法的。

- `Base.extend(...);`：这个函数用于创建新的基本构造函数对象。这个函数接受的唯一参数是包含属性和值的简单对象，对象中的所有这些属性和值都被添加到要生成的对象里，作为其原型方法。
- `Person.extend(...);`：这是`Base.extend()`语法风格的另一个版本，因为所有的构造函数都使用过`.extend()`方法来获得它们自身的`.extend()`方法，所以你也可以直接继承它们。在代码清单3-4中就是通过直接继承原始`Person`构造函数的方法来创建User构造函数的。
- `this.base();`：最后这个`this.base()`方法用来调用一个已被重载的父类方法。可以看到，这和Crockford的类式库提供的`this.uber()`函数很不一样，你不需要指定父类方法的名字（这有助于你保持代码的清晰与整洁）。在所有的面向对象JavaScript库中，Base的重载父类方法功能是最棒的。

我个人认为用 Dean 的 Base 库能够写出最可读、最有用也是最好理解的面向对象 JavaScript 代码。不过归根结底，应该由开发者来选择最适合自己的库。下面你会看到广受欢迎的 Prototype 库是如何实现面向对象代码的。

3.1.4　Prototype 库

Prototype 是一个为配合流行的 Ruby on Rails 的 Web 开发框架配合而设计的 JavaScript 库[①]，不过不应该把它的名字和构造函数的 `prototype` 属性相混淆。

撇去名字不谈，Prototype 使得 JavaScript 无论从表面上，还是从使用上都大大接近 Ruby 了。Prototype 的开发者是利用 JavaScript 的面向对象的实质来实现这一效果的，他们给核心 JavaScript 对象添加了一大堆的函数和属性。坏消息是开发者并没有提供库本身的文档；好消息是，库的代

① 其实Prototype称自己为Framework（框架），而不是库。——译者注

码非常清晰，所以有些用户能够自己编写它的文档。你可以在 Prototype 的网站 http://prototype.conio.net/上任意查阅它的代码。也能从文章 *Painless JavaScript Using Prototype*（http://www.sitepoint.com/article/painless-javascript-prototype/）获得它的文档[①]。

在这一小节中，我们只介绍 Prototype 中那些用以创建面向对象结构、提供基本继承的函数和对象。代码清单 3-5 提供了所有让 Prototype 实现这一目标的代码。

代码清单3-5 Prototype用来模拟面向对象JavaScript代码的两个函数

```
// 创建一个名叫 'Class' 的全局对象
var Class = {
    // 它只有一个函数，其作用是创建一个新的对象构造函数
    create: function() {
        // 创建一个匿名的对象构造函数
        return function() {
            // 调用它本身的初始化方法
            this.initialize.apply(this, arguments);
        }
    }
}

// 给 Object 对象添加一个新的静态方法，它的作用是把属性从一个对象复制到另一个中
Object.extend = function(destination, source) {
    // 遍历所有要扩展的属性
    for (property in source) {
        // 然后将他们添加到目标对象中
        destination[property] = source[property];
    }

    // 返回修改后的对象
    return destination;
}
```

Prototype 确实只使用这两个明显不同的函数来创建和管理整个面向对象结构，你可能注意到了，这比 Base 或者 Crockford 的类式方法要脆弱得多。这两个函数的出发点很简单：

- `Class.create()`：这个函数仅仅使用一个匿名函数封装起来作为构造函数。这种简单的构造函数只做一件事：调用并执行对象的`initialize`属性。也就是说，你的对象至少必须包含一个值为函数的`initialize`属性，否则代码会抛出异常。
- `Object.extend()`：这仅仅把所有属性从一个对象复制到另一个对象而已。如果你使用构造函数的`prototype`属性就可以设计出一种更简单的继承方式（比JavaScript默认的原型式继承更简单）。

现在你已经了解Prototype的内部代码是怎样的了，代码清单3-6展示了一些如何使用Prototype来给原生 JavaScript 对象扩展出更多层功能的例子。

46

① Prototype一直由于它文档的匮乏而备受诟病，不过官方最近成立的 http://prototypejs.org/ 已经提供了不少的文档，这是一个不错的进步。——译者注

代码清单3-6　Prototype如何使用面向对象函数来扩展JavaScript的默认字符串操作

```
// 给 String 的 prototype 添加额外的方法
Object.extend(String.prototype, {
    // 一个新的 StripTags 函数，去除字符串这给你所有的 HTML 标记
    stripTags: function() {
        return this.replace(/<\/?[^>]+>/gi, '');
    },

    // 将字符串转换为字符的数组
    toArray: function() {
        return this.split('');
    },

    // 将 "foo-bar" 形式的小写横线式文本转换为 "fooBar" 的大小写混合式文本
    camelize: function() {
        // 将在 '-' 处断开字符串
        var oStringList = this.split('-');

        // 如果没有 '-' 则立即返回
        if (oStringList.length == 1)
            returnoStringList[0];

        // 可选地将字符串转换为大小写混合形式
        var camelizedString = this.indexOf('-') == 0
            ? oStringList[0].charAt(0).toUpperCase() + oStringList[0].substring(1)
            : oStringList[0];

        // 大写每一部分的首字符
        for (var i = 1,len = oStringList.length; i < len; i++) {
            var s = oStringList[i];
            camelizedString += s.charAt(0).toUpperCase() + s.substring(1);
        }

        // 并返回修改过的字符串
        return camelizedString;
    }
});

// stripTags() 方法的例子
// 可以看到它去除了字符串中的所有 HTML标签，而只剩下文本
"<b><i>Hello</i>, world!".stripTags() == "Hello, world!"

// Array() 方法的一个例子，获取字符串的第 4 个字符
"abcdefg".toArray()[3] == "d"

// 使用 camelize() 方法的例子
// 将字符串的旧格式转换为新格式。
"background-color".camelize() == "backgroundColor"
```


下面让我们再看看本章开头的例子，分别有 Person 对象和 User 对象，User 对象继承自 Person 对象。使用 Prototype 的面向对象风格来重新实现的代码如代码清单 3-7 所示。

代码清单3-7 Prototype的辅助函数，用于创建类、实现简单继承

```
// 创建一个新的构造函数为空的 Person 对象
var Person = Class.create();

// 将下面的函数复制到 Person 的 prototype 中
Object.extend( Person.prototype, {
    // 这个函数由 Person 构造函数立即调用
    initialize: function( name ) {
        this.name = name;
    },

    // Person 对象的简单函数
    getName: function() {
        return this.name;
    }
});

// 创建一个新的构造函数为空的 User 对象
var User = Class.create();
// User 对象继承了所有父类的函数
User.prototype = Object.extend( new Person(), {

    // 将旧的初始化函数重载为新的
    initialize: function( name, password ) {
        this.name = name;
        this.password = password;
    },

    // 给这个对象添加一个新的函数
    getPassword: function() {
        return this.password;
    }
});
```

48

尽管 Prototype 提供的面向对象技术并非革命性的，但也足以帮助开发者创建更简单、更易编写的代码了。不过归根结底，如果需要编写大量的面向对象代码，你可能会选择类似 Base 这样的库以减少其中的开发难度。

下面我们将探讨如何部署写好的面向对象代码，让其他开发者或库能够使用，并能与之交互作用。

3.2 打包

在编写好优美的面向对象 JavaScript 代码之后（如果你足够机灵，那应该是在编写的过程中），是时候向与其他 JavaScript 库良好配合的方向发展了。需要注意的是，你的代码会被其他开发者和使用者使用，他们的需求跟你的可能会有所不同，尽可能地编写足够干净的代码对其会有所帮助，从他人已有的工作成果中学习也能实现这样的效果。

在这一节里，你会看到成千上万的开发者日常使用的几个大型的库。其中每个都提供了独特的方法来控制它的代码结构，使之更易学易用。你还会看到其他一些清理代码的方法，为（使用你代码的）他人提供最好的体验。

3.2.1　命名空间

清理简化代码的一个重要而简单的概念是命名空间（namespace）。目前的 JavaScript 并不支持命名空间（和 Java、Python 不同），所以我们不得不通过类似的方法取得合适的效果。

实际上，JavaScript 里并不存在“命名空间”的概念，但考虑到 JavaScript 的所有对象都有自己的属性，属性又可以包含对象，这样就能创造一些和其他语言里的命名空间神似的东西了。运
49 用这一技巧创建出来的独特结构如代码清单 3-8 所示。

代码清单3-8　JavaScript中的命名空间化及其实现

```
// 创建一个默认的、全局的命名空间
var YAHOO = {};

// 使用对象设置一些子命名空间
YAHOO.util = {};

// 创建最终命名空间，它包含一个值为函数的属性
YAHOO.util.Event = {
    addEventListener: function(){ ... }
};

// 调用某个具体命名控件中的函数
YAHOO.util.Event.addEventListener( ... )
```

让我们看看几个很受欢迎的库里使用命名空间的例子，来了解如何用命名空间搭建起一个牢固而又可扩展的插件架构。

1. Dojo

Dojo 是一个极受欢迎的开发框架，为开发完整的 Web 应用提供了所有必要的东西。这意味着它包含了许多可以单独引用、执行的子库，否则整个库就会由于过大而变得难以处理。关于 Dojo 的更多信息可以在该项目的网站找到：http://dojotoolkit.org/。

Dojo 运用 JavaScript 命名空间构建了整个程序包系统（package system）。你可以动态地单独导入新的程序包，自动执行并使用它。代码清单 3-9 展示了一个在 Dojo 上使用命名空间的例子。

代码清单3-9　Dojo的打包与命名空间化

```
<html>
<head>
    <title>Accordion Widget Demo</title>
    <!-- 包含 Dojo 框架 -->
    <script type="text/javascript" src="dojo.js"></script>
    <!-- 包含 Dojo 程序包 -->
    <script type="text/javascript">
        // 导入两个程序包，它们用来创建 Accordian Container 控件
        dojo.require("dojo.widget.AccordionContainer");
        dojo.require("dojo.widget.ContentPane");
    </script>
</head>
<body>
<div dojoType="AccordionContainer" labelNodeClass="label">
```

```
    <div dojoType="ContentPane" open="true" label="Pane 1">
        <h2>Pane1</h2>
        <p>Nunc consequat nisi vitae quam. Suspendisse sed nunc. Proin...</p>
    </div>
    <div dojoType="ContentPane" label="Pane 2">
        <h2>Pane2</h2>
        <p>Nunc consequat nisi vitae quam. Suspendisse sed nunc. Proin...</p>
    </div>
    <div dojoType="ContentPane" label="Pane 3">
        <h2>Pane3</h2>
        <p>Nunc consequat nisi vitae quam. Suspendisse sed nunc. Proin...</p>
    </div>
</div>
</body>
</html>
```

Dojo 的程序包架构非常强大，如果你对大量 JavaScript 代码的维护有兴趣，它值得一看。而且，由于它非常庞大，你肯定也能在其中找到一些对你有用的功能。

2. YUI

另一个具有大型的、大量使用命名空间的程序包架构的 JavaScript 库，是 Yahoo UI（http://developer.yahoo.com/yui/）。这个库设计的目的是为许多常见的 Web 应用场景（比如拖放功能）提供解决方案。所有的 UI（用户界面）元素都被分布在它的层级结构中。Yahoo UI 库的文档也非常不错，它非常的完整与详尽。

和 Dojo 很像，Yahoo UI 也使用了深层次的命名空间结构来组织其函数和特性。然而和 Dojo 不同的是，所有的外部代码的“导入”都由你自行完成，而不是通过 `import` 语句。代码清单 3-10 展示了 Yahoo UI 库里命名空间的表示形式和操作方式。

代码清单3-10　Yahoo UI库的打包与命名空间

```
<html>
<head>
    <title>Yahoo! UI Demo</title>
    <!-- 导入主 Yahoo UI 库 -->
    <script type="text/javascript" src="YAHOO.js"></script>

    <!-- 导入事件程序包 -->
    <script type="text/javascript" src="event.js"></script>
    <!-- 使用导入的 Yahoo UI 库 -->
    <script type="text/javascript">
        // 所有 Yahoo 事件与使用程序都包含在 YAHOO 这个全局命名空间下，
        // 并切割为多个子命名空间（如 'util'）
        YAHOO.util.Event.addListener('button', 'click', function() {
            alert("Thanksforclickingthebutton!");
        });
    </script>
</head>

<body>
    <input type="button" id="button" value="Click Me!"/>
</body>
</html>
```

把大量代码组织为一个单独的程序包以方便维护，Dojo 和 Yahoo UI 在这一点上都是非常成功的，了解它们是如何用 JavaScript 命名空间实现这一目标，将非常有助于你实现自己的程序包架构。

3.2.2　清理代码

在讨论调试和编写测试用例的话题之前（会在下一章讨论），懂得如何编写能够让其他人使用的代码是很重要的，如果你希望代码能在其他开发者的使用和修改后存活下来，你应该确保它的用意不被误解或错误使用。虽然你可以自己检查一遍代码，手工清理它们，但对于代码中复杂的部分，通常使用工具会更有效率，这就是 JSLint 出现的原因。JSLint 有一系列内建的规则，可以找出可能会引发问题的代码片段。JSLint 网站上提供了完整的分析工具：http://www.jslint.com/。另外 JSLint 的所有规则和设置都能在此找到：http://www.jslint.com/lint.html。

JSLint 是由 Douglas Crockford 开发的另一个工具，为配合他自己的编码风格而设计，如果你实在是不喜欢那些要求或者生成的修改，完全没有必要遵循。不过其中一些规则还是特别有用的，所以我会在这里作出详细的说明。

1. 变量声明

JSLint 提出的一个聪明的要求是程序里所有的变量都必须在使用前声明，虽然 JavaScript 并不明确要求你这么做，但不声明会导致变量的实际作用域不好理解。比如你希望在函数中设置一个未声明的变量，这个变量是属于函数内呢还是属于全局作用域呢？只观察代码是无法立即明白的，所以很有必要澄清。JSLint 声明变量习惯的例子如代码清单 3-11 所示。

52

代码清单3-11　JSLint要求的变量声明

```
// 不当的变量使用
foo = 'bar';

// 恰当的变量声明
var foo;
...
foo = 'bar';
```

!= 和 == 对 !== 和 ===

开发者容易犯的一个常见错误是缺乏对 JavaScript 中 false 值的理解，在 JavaScript 中，`null`、`0`、`''`、`false` 和 `undefined` 都两两相等（`==`），因为对它们求值的结果都为 `false`。亦即在 `test==false` 表达式中，即使 `test` 是 `undefined` 或 `null`，这个表达式的值也是 `true`，但你所期望的结果很有可能并非如此。

这正是`!==`和`===`的用处所在：这两个运算符都能判断变量的显式值（explict value），比如 `null`，而非其等价值（比如 `false`）。JSLint 就要求你每次对 `false` 进行比较时，都使用`!==`或`===`而不是`!=`或`==`。代码清单 3-12 的例子展示了这些运算符的区别。

代码清单3-12　!= 和== 如何与 !== 和 === 区分的例子

```
// 下面两个表达式皆为真
```

```
null == false
0 == undefined

// 下面应该使用 !== 或 ===
null !== false
false === false
```

2. 代码块与大括号

这是一个我不太能接受的规则，不过在协作开发代码的环境中倒是值得考虑。这个规则的前提是：不应该使用单行的代码块。比如在判断语句（比如 `if(dog==cat)`）后面即使只跟着一条语句（`dog=false;`），也应该保留大括号。对 `while()`和 `for()`代码块也是如此。虽然对单行语句不用大括号是 JavaScript 提供的巨大便利，但这可能导致其他人的误解，比如没能看出某条语句是否属于某块代码，等等。代码清单 3-13 很好的解释了这种情形。

代码清单3-13 缩进不当的单语句代码块

```
// 这段代码是合法的、正常的 JavaScript
if ( dog == cat )
    if ( cat == mouse )
        mouse = "cheese";

// 然而JSLint 要求它应该写成这样
if ( dog == cat ) {
    if (cat == mouse) {
        mouse="cheese";
    }
}
```

3. 分号

这最后一点的讨论对于在下一节所讨论的代码压缩很有帮助。JavaScript 里如果每个语句单独占一行，那语句后边的分号不是必需的。在未压缩的代码里不用分号当然没什么问题，但在去除换行符以压缩代码后就可能导致问题了。要避免这样的问题，你应该时时记住在所有语句后面加上分号，如代码清单 3-14 所示。

代码清单3-14 必须带有分号的语句

```
// 如果你希望压缩你的 JavaScript 代码，应确保所有语句之后都有分号
var foo = 'bar';
var bar = function(){
    alert('hello');
};
bar();
```

这一点把我们带到了最后关于 JavaScript 压缩概念的讨论。用 JSLint 来编写干净的代码，受益的是程序员自己，而压缩却能给用户带来很大方便，因为这样他们就可以更快地打开你的网站。

3.2.3 压缩

分发 JavaScript 库的关键之一是用代码压缩工具来节省带宽。因为压缩后的代码变得非常混

乱而不可读，所以这应该是在把代码交付使用前的最后一步。有3类JavaScript压缩工具：

- 只去除所有多余的空白和注释的压缩工具，它保留所有有意义的代码。
- 去除空白和注释，而且修改变量名以减小代码的压缩器。
- 完成上述两种工作，而且还最小化代码中所有单词长度，而不仅仅是变量名称。

我要讨论两个压缩工具是：JSMin和Packer。JSMin属于第一种压缩方式（尽可能去除非代码的其他内容），而Packer属于第三种（完全压缩所有的单词）。

54

1. JSMin

JSMin的原理很简单，它遍历一段JavaScript代码，去除所有非必要的字符，只留下实现功能所必须的代码。JSMin是通过去除所有多余的空白字符（包括制表符和换行符）和注释达到这一目的的。这个压缩工具的在线版本可见：

http://www.crockford.com/javascript/jsmin.html.

要感受一下代码经过JSMin处理后会变成什么样子，我们以一小段代码为例（如代码清单3-15所示），经过压缩器，输出如代码清单3-16所示。

代码清单3-15　判断用户浏览器的代码

```
// (c) 2001 Douglas Crockford
// 2001 年 6 月 3 日
// 这是一个用于判断浏览器的对象。每个浏览器版本都有自我判断的方法，
// 但的确没有可以适用于所有浏览器版本的通用标准方法，因为 Web
// 浏览器的作者都是骗子。比如，Microsoft 的 IE 浏览器自己生成的是
// Mozilla 4，而 Netscape 6 自己生成的版本则是 5。

var is = {
    ie:      navigator.appName == 'Microsoft Internet Explorer',
    java:    navigator.javaEnabled(),
    ns:      navigator.appName == 'Netscape',
    ua:      navigator.userAgent.toLowerCase(),
    version: parseFloat(navigator.appVersion.substr(21)) ||
             parseFloat(navigator.appVersion),
    win:     navigator.platform == 'Win32'
}
is.mac = is.ua.indexOf('mac') >= 0;
if (is.ua.indexOf('opera') >= 0) {
    is.ie = is.ns = false;
    is.opera = true;
}
if (is.ua.indexOf('gecko') >= 0) {
    is.ie = is.ns = false;
    is.gecko = true;
}
```

代码清单3-16　代码清单3-15所示代码经过压缩后的版本

```
// 压缩后代码
varis={ie:navigator.appName=='Microsoft Internet Explorer',java:
```

```
navigator.javaEnabled(),ns:navigator.appName=='Netscape',ua:
navigator.userAgent.toLowerCase(),version:parseFloat(
navigator.appVersion.substr(21))||parseFloat(navigator.appVersion),win:
navigator.platform=='Win32'} is.mac=is.ua.indexOf('mac')>=0;if(
is.ua.indexOf('opera')>=0){is.ie=is.ns=false;is.opera=true;}
if(is.ua.indexOf('gecko')>=0){is.ie=is.ns=false;is.gecko=true;}
```

55

注意所有的空白和注释都被去除了，这样可以极大地精简代码。

JSMin 可能是最简单的 JavaScript 压缩工具了，它是在实际代码中开始运用压缩方法的一个很好的起步。如果你还希望更进一步的压缩代码，就可能要改用 Packer，这个令人敬畏而又极其强大的 JavaScript 压缩工具。

2. Packer

Packer 是目前最强大的 JavaScript 压缩工具，由 Dean Edwards 设计。它提供了一种彻底精简代码，并在执行时即时重新展开的方式。通过这种技巧，Packer 能够得到可能是最小的代码。你可以认为它把 JavaScript 代码转换为了一个具备自解压功能的 ZIP 文件。这个脚本的在线版本可见：http://dean.edwards.name/packer/。

Packer 脚本非常庞大而复杂，所以建议你不要尝试自己去实现一个。此外，它生成的代码会多出几百字节的内容，用于进行自解压。所以它还不算最完美的代码压缩方法（JSMin 的效果可能会比它要好）。不过对大文件来说 Packer 肯定是最合适的。代码清单 3-17 展示了 Packer 生成的自解压代码的一部分。

代码清单3-17　使用Packer压缩的一部分代码

```
eval(function(p,a,c,k,e,d){e=function(c){return c.toString(36)};if(!''.replace(/^/,
String)){while(c--){d[c.toString(a)]=k[c]||c.toString(a)}k=[(function(e){return
d[e]})];e=(function(){return'\\w+'});c=1};while(c--){if(k[c]){p=p.replace(new
RegExp('\\b'+e(c)+'\\b','g'),k[c])}}return p}('u 1={5:2.f==\'t s
r\',h:2.j(),4:2.f==\'k\',3:2.1.m(),n:7(2.d.o(p))||7(2.d),q:2.g==\'i\'}1.
b=1.3.6(\'b\')>=0;a(1.3.6(\'c\')>=0){1.5=1.4=9;1.c=e}a(1.3.6(\'8\')>=0){1.5=
1.4=9;1.8=e}',31,31,'|is|navigator|ua|ns|ie....
```

压缩，尤其是用 Packer 来压缩代码的作用不可轻视。虽然和你的具体代码有关，但通常能够压缩到原来的 50%，因而能加快载入页面的速度，而这肯定是每个 JavaScript 应用程序的首要目标。

3.3 分发

根据你的实际情况，这个 JavaScript 编码的最后步骤是可选的。如果你只不过是给自己或者公司编写代码，那把代码复制给别的开发者或者上传到网站上就行了。

然而如果你开发了一套有趣的代码，希望让世界上其他的人都能按其意愿使用，就应该考虑 JSAN（JavaScript Archive Network）这样的服务了。JSAN 是一系列 Perl 开发者发起的，他们喜欢 CPAN（Comprehensive Perl Archive Network）的功能。更多关于 JSAN 的信息可以在它的网站找到：http://openjsan.org。

56

JSAN 要求所有提交的模块都用一种格式良好的、面向对象的风格来编写，以符合它特定的模块化架构。除了对代码进行集中贮藏外，JSAN 还支持在你的代码中导入外部 JSAN 模块的需求。这可以使编写独立应用程序的工作大为简化，因为你不再需要担心用户是否安装了某个特定的依赖模块。让我们通过一个简单的 JSAN 模块，DOM.Insert 来理解其典型的工作方式，它可以在这里找到：http: //openjsan.org/doc/r/rk/rkinyon/DOM/Insert/0.02/lib/DOM/Insert.html。

这一模块接受一个 HTML 字符串，将其插入网页的特定位置。除了很好的面向对象以外，这个模块还引用并加载了两个 JSAN 模块，如代码清单 3-18 所示。

代码清单3-18　一个JSAN模块（DOM.Insert）示例

```
// 我们要尝试包含一些其他的 JSAN 模块
try {
    // 载入两个必须的 JSAN 库
    JSAN.use( 'Class' )
    JSAN.use( 'DOM.Utils' )

// 如果 JSAN 并未载入，则抛出一个异常
} catch (e) {
    throw "DOM.Insert requires JSAN to be loaded";
}

// 保证 DOM 命名空间存在
if (typeof DOM == 'undefined' )
    DOM = {};

// 创建一个新的 DOM.Insert 构造函数，它继承自 'Object'
DOM.Insert = Class.create( 'DOM.Insert', Object, {

    // 接受两个参数的构造函数
    initialize: function(element, content) {
        // 将要把 HTML 插入其中的元素
        this.element = $(element);

        // 需要插入的 HTML 字符串
        this.content = content;

        // 尝试使用 Internet Explorer 方式插入 HTML
        if (this.adjacency && this.element.insertAdjacentHTML) {
            this.element.insertAdjacentHTML(this.adjacency, this.content);

        // 否则，尝试 W3C 标准方式
        } else {
            this.range = this.element.ownerDocument.createRange();
            if (this.initializeRange) this.initializeRange();
            this.fragment = this.range.createContextualFragment(this.content);
            this.insertContent();
        }
    }
});
```


写出清晰、容易处理的 JavaScript 代码肯定是你（或者其他 Web 开发者）开发过程中的一个

里程碑，以此为出发点，我们将探究 JavaScript 语言的其他部分。随着 JavaScript 继续得到更多人的认同，这种编程风格的重要性将会只增不减，将会变得更加的有用和盛行。

3.4 小结

在这章里，你看到了构建可重用代码结构的多种不同方法。运用上一章学到面向对象技巧，你可以创建出非常适合多开发者环境、清晰的数据结构来。此外，你还看到了创建可维护代码、精简 JavaScript 文件大小、打包以分发代码这些工作的最佳方式。了解如何编写格式整齐、可维护的代码能让你节省大量理解代码的时间。

第4章

调试与测试的工具

编程开发中最耗时间的过程恐怕就是测试和调试代码了。要开发出专业水准的代码，最重要的就是必须保证它是经过完整测试、可验证和没有 bug 的。而 JavaScript 与其他语言如此不同的一个原因在于，这门语言并不由个别公司或者组织掌握（和 C#、PHP、Perl、Python 或 Java 不同）。这一区别使得寻找一致的平台来测试和调试代码的工作颇具挑战性。

为了缓解压力、减少负荷，在捕获 JavaScript 代码的 bug 时，应该选用一套强大的开发工具。每个现代浏览器都带有它们自己的工具（虽然质量各异），使用这些工具大大增进了 JavaScript 开发的一致性，也令其前途更光明。

在这一章里，我将讨论用来调试 JavaScript 的多种不同工具，以及如何构建牢固、可重用的测试套件，用它来校验未来的开发。

4.1 调试

测试和调试是并行不悖的两个过程，虽然你应该给代码准备完善的测试用例，但在准备的过程中，极有可能会遇到各种值得注意的怪异错误，此时就需要进行调试了。通过运用手头最好的工具来寻找并解决 bug，你能更快地让代码重新回到正轨上。

4.1.1 错误控制台

在所有现代浏览器中都能找到的，也是最常见的一种工具是错误控制台（error console）。但控制台的质量、界面的亲和力和它提供的错误信息的质量都因浏览器的不同而稍有差异。总之，你一般会发现，选择一个拥有最适合开发者的错误控制台（或其他调试扩展）的浏览器来进行你的调试是最好的选择。

1. IE

最受欢迎的浏览器并不意味着就带有最好的调试工具。更无奈的是，IE 的错误控制台有诸多的不足。且不说其他的，首先这个控制台默认情况下还是禁用的，这让没有将 IE 作为默认浏览器的人在找寻错误的时候十分困难（任何有自尊的 JavaScript 开发者会用它作为默认浏览器吗？值得怀疑）。

除了前述的可用性问题以外，IE 的错误控制台最麻烦的问题还有以下几个：

- 每次只显示一个错误；你必须反复点击“上一条”或“下一条”才能找到其他错误信息。
- 错误信息非常含糊，难于理解，鲜有给出问题确切描述的。
- 报告的错误行号总是“差了一行”，意味着实际错误发生的行号总比它报告的行号少1。加上它含糊的错误信息，你必须费很大力气才能找到bug所在。

IE 的错误控制台显示错误的例子如图 4-1 所示。

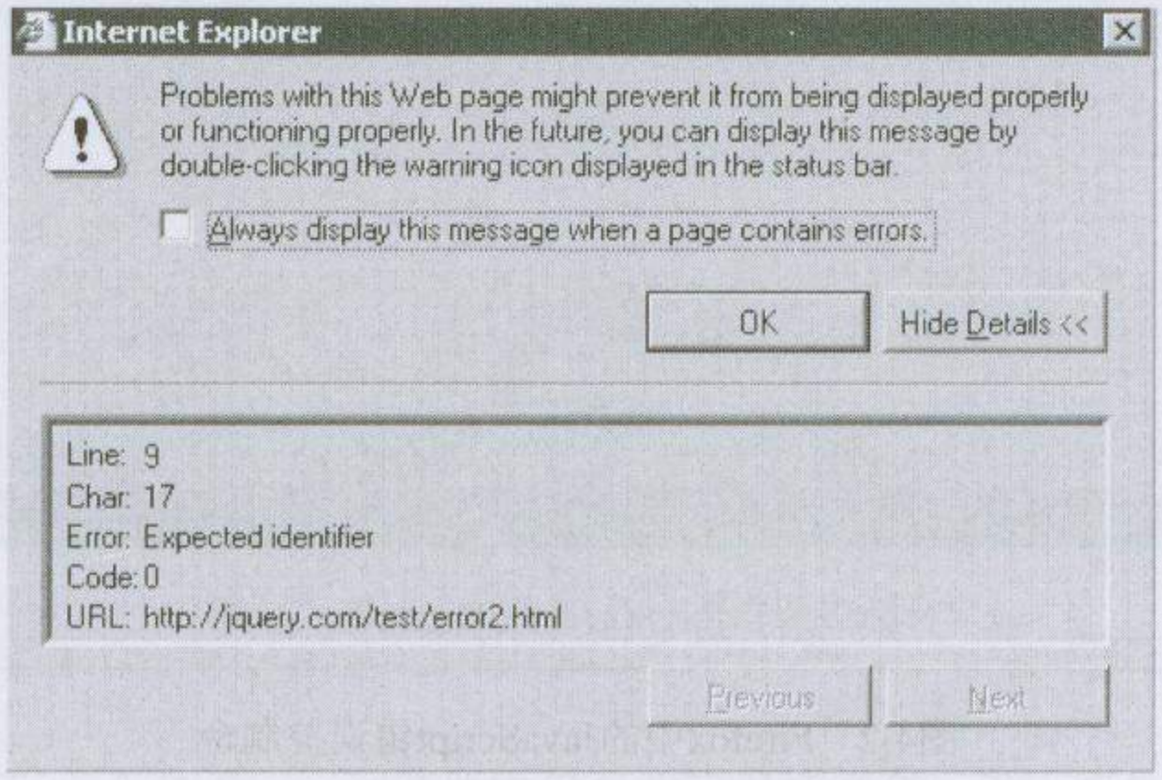


图4-1 IE中的JavaScript错误控制台

正如我在本节开头所说，最好从另一个（IE 以外的）浏览器里开始你的调试过程，等到你完全排除了所有在那个浏览器里找到的 bug，再回到 IE 中来处理那些错综复杂的细节就要容易得多了。

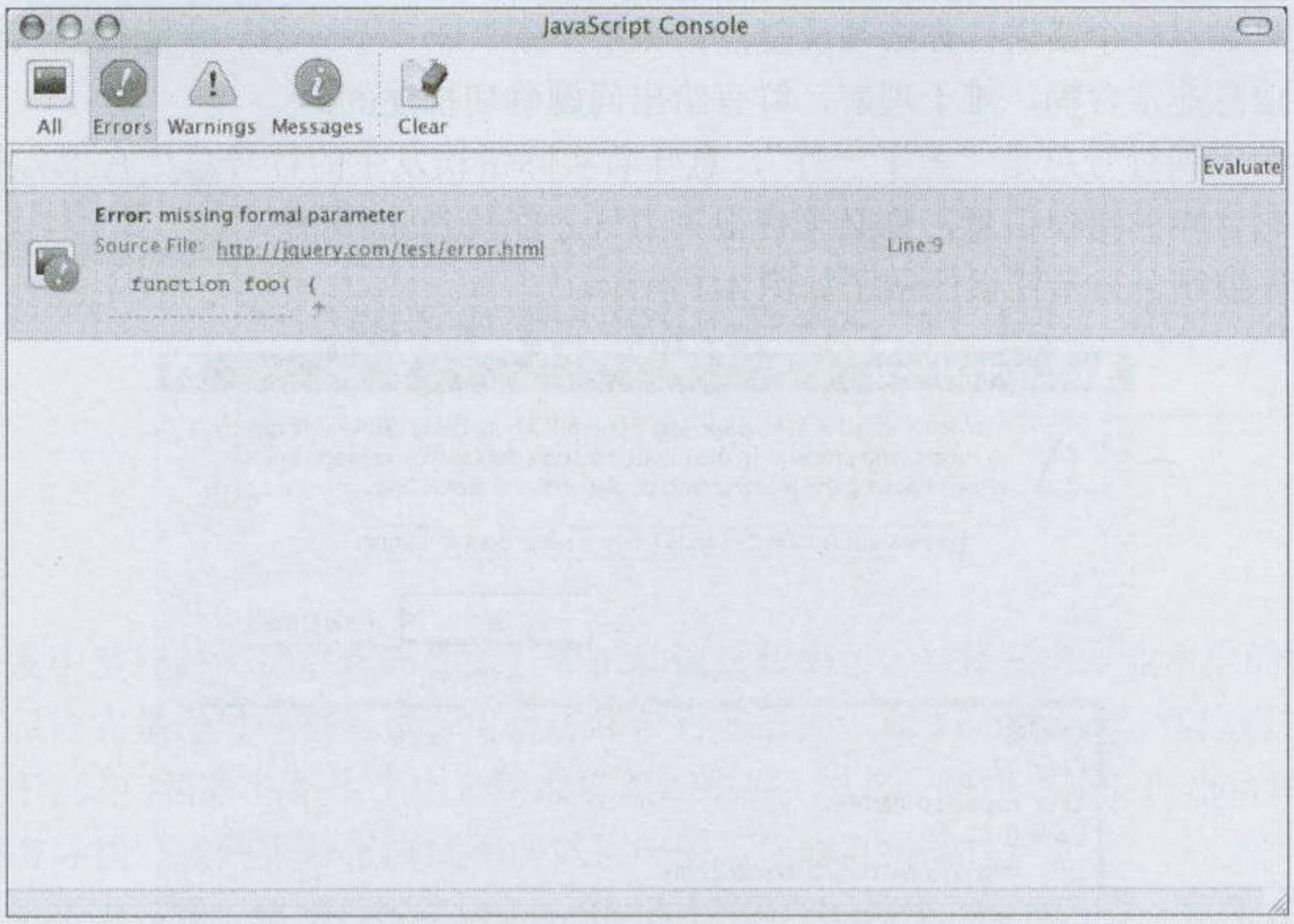
2. Firefox

Firefox 浏览器于过去几年在用户界面上作了许多非常好的改进，有助于 Web 开发者们更方便地开发网站。它的 JavaScript 错误控制台也经多次修改，达到了一个非常好用的层次，有以下这些特点：

- 允许你输入任意的JavaScript命令。这个功能特别适合在页面加载后查询变量的值。
- 能让你根据不同类型的消息进行分类，比如错误、警告和一般消息。
- 最新版本除JavaScript错误以外，还提供了样式表警告和错误信息。对于设计得不好的站点，这个功能会让控制台充塞大量的错误，不过用它来寻找你自己设计的页面出现的怪异布局问题还是很有用的。
- 有一个缺点，它不会根据你当前正在浏览的页面来过滤消息，也就是说你看到的错误来自于多个不同的页面。（我将在下一节讨论的Firebug扩展会解决这个问题。）

Firefox 错误控制台的截图如图 4-2 所示。你可以用不同的按钮来切换浏览不同类型的消息。

尽管 Firefox 错误控制台非常好用，但并非没有缺陷，所以开发者使用其他的 Firefox 扩展以便更好地调试程序。我会在这个关于调试的小节的后半部分介绍这些扩展。



61

图4-2　Firefox中的JavaScript错误控制台

3. Safari

Safari 是浏览器市场上的最新成员，也是发展最快的一个。由于发展过于迅速，JavaScript 的支持（包括对开发 JavaScript 和执行 JavaScript 的支持）曾经非常不稳定，所以要访问 JavaScript 控制台并不是很容易，甚至启用它也有些费力，因为它被放在一个普通用户不会接触到的隐藏的调试菜单中。

要启用调试菜单（由此才能得到 JavaScript 控制台），你必须在终端中执行代码清单 4-1 的命令（而且是在 Safari 不在运行的情况下）。

代码清单4-1　显示Safari调试菜单的命令

```
defaults write com.apple.Safari IncludeDebugMenu 1
```

再次打开 Safari，你就可以看到一个新的调试菜单，其中包含了 JavaScript 控制台。

可能你已经从它所处的偏僻角落猜到，这个控制台的功能还非常欠缺。关于它的缺点，主要有以下几个：

- 错误信息通常很含糊，和IE提供的差不多。
- 行号随错误一起提供，但常会被重置为零，导致你跳回起点。
- 错误信息不会按照页面过滤，不过所有的信息旁边都会标上抛出错误的脚本名称。

Safari 2.0 中错误控制台截图如图 4-3 所示。

Safari 作为 Web 开发平台来说还很落后，不过 WebKit 开发组（开发 Safari 渲染引擎的人员）已经在改进这个浏览器上有了很大的进展。值得期待的是，在未来几个月到几年间，这个浏览器会有大量的改进。

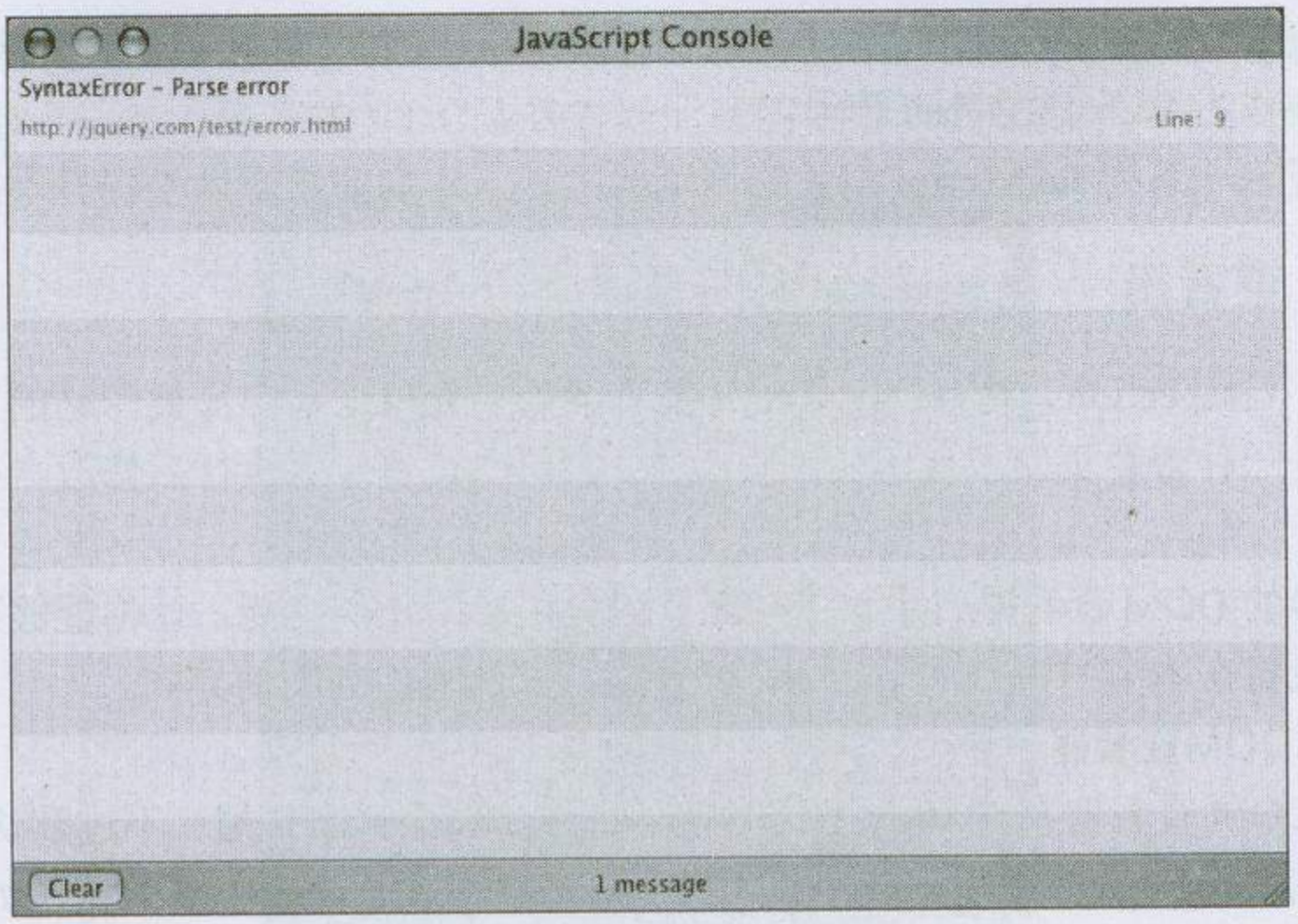


图4-3 Safari中的JavaScript错误控制台

62

4. Opera

我们要考察的最后一个是 Opera 浏览器中包含的错误控制台。幸运的是，Opera 花了大量的时间和精力，让错误控制台更有用更健全。除 Firefox 错误控制台提供的所有特性外，它还具有以下几个：

- 描述性的错误信息，让你很好理解问题所在。
- 行内代码片段，展示错误在代码中所处的位置。
- 错误信息可以根据类型来过滤（如JavaScript、CSS，等等）。

可惜这个控制台还缺乏执行 JavaScript 命令的能力，非常遗憾，因为这个特性很有用。不过把上述特性组合起来，仍然能为你提供一个非常好的错误控制台了。图 4-4 是 Opera 9.0 中错误控制台的一张截图。

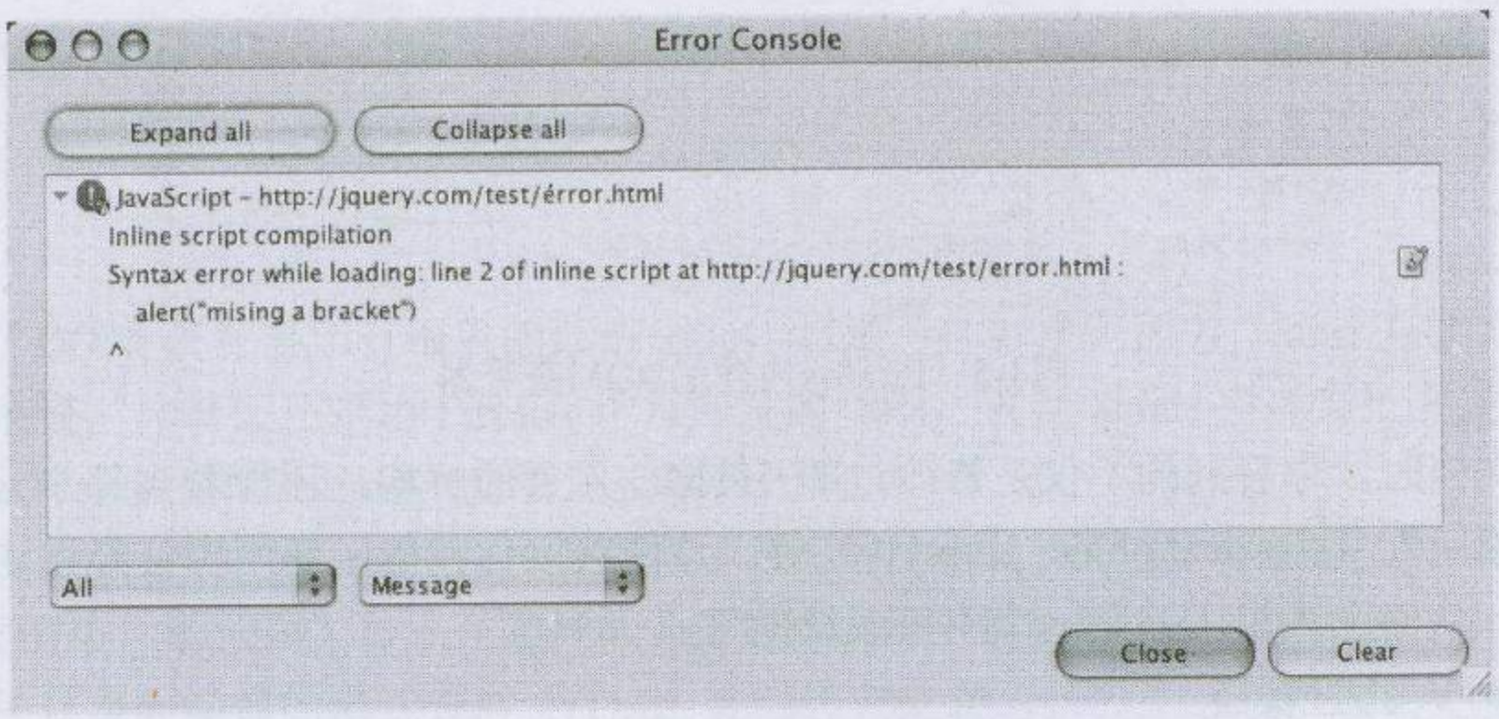


图4-4 Opera中的JavaScript错误控制台

Opera长期关注Web开发，在它的开发组中有大量积极的、干劲十足的开发者和规范编写者，所以它也有希望作为Web开发者的平台。

63

下面我将展示一系列JavaScript相关的浏览器扩展，它们非常有助于提升你的开发能力。

4.1.2　DOM查看器

DOM查看（inspection）是JavaScript开发者最重要但也是最容易被忽略的工具之一。这种查看可以视为是一种查看页面源代码的加强版本，它允许查看代码执行后修改自身内容页面的当前状态。

不同浏览器的DOM查看器效果也不同，有些提供了附加的功能，让你能够窥视到更多正在处理的内容。我将在本节中讨论3个DOM查看器，以及它们之间的异同。

1．Firefox DOM查看器

Firefox的DOM查看器在Firefox安装时就捆绑在一起的一个扩展（不过默认是禁用的）。这一扩展允许你在HTML文档构建完毕和载入后查看详细内容。这个扩展的截图如图4-5所示。

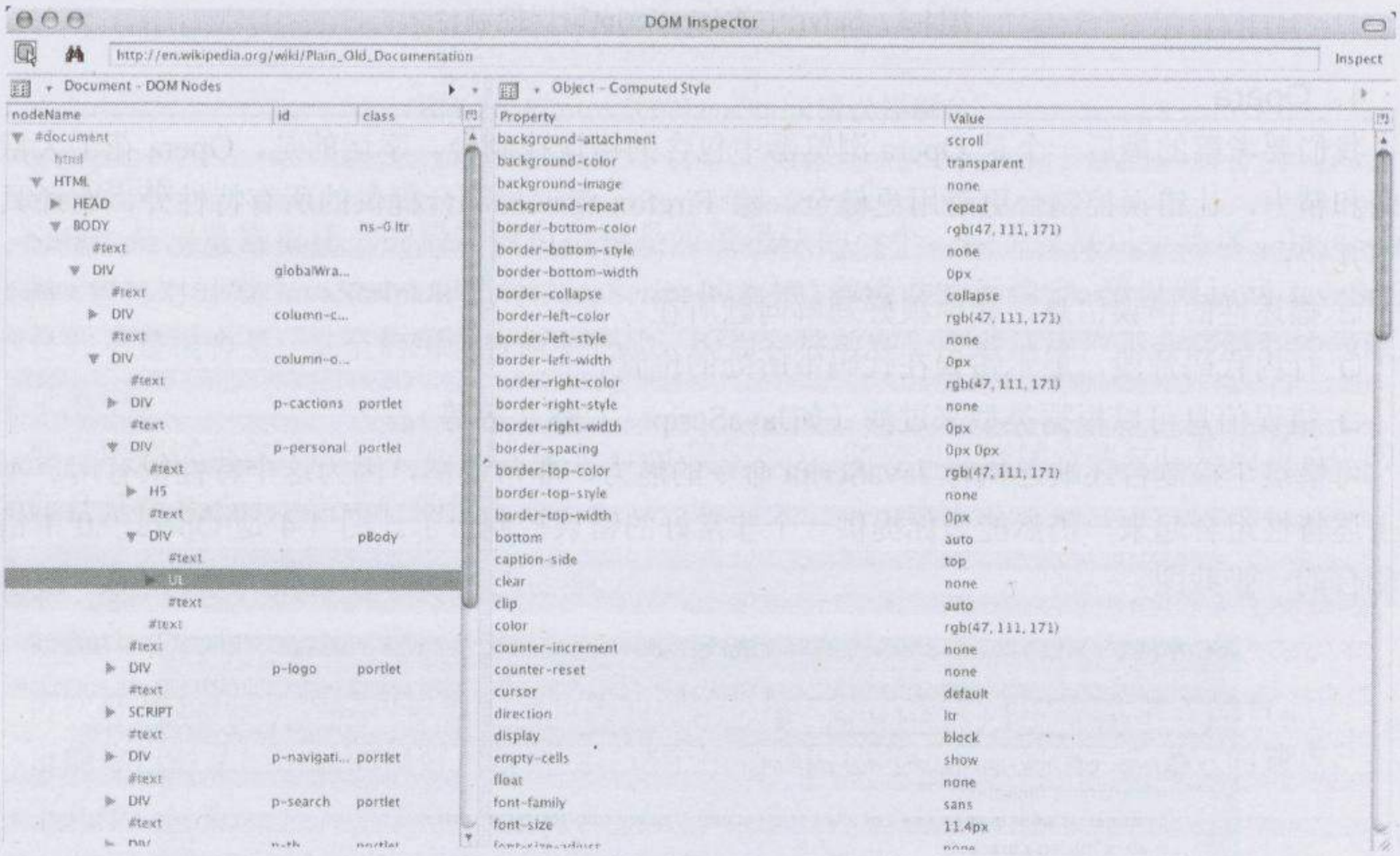


图4-5　Firefox内建的DOM查看器

在遍历文档时，你看到的不仅是修改过的HTML元素的解构，还能看到每个元素的样式属性、物理对象属性（physical object property）等。这能帮助你精确了解网页在经过了你修改之后的视觉效果。从而也决定了这是一个不可或缺的工具。

64

2．Safari Web查看器

Safari在最新版本的浏览器中包含了一个新的DOM查看器。某些情况下它要比Firefox的

DOM 查看器好用，因为你可以在右键点击任意页面的元素的时候，就能在查看器中立即定位到该元素。Safari 这个（设计精巧的）DOM 查看器的截图如图 4-6 所示。

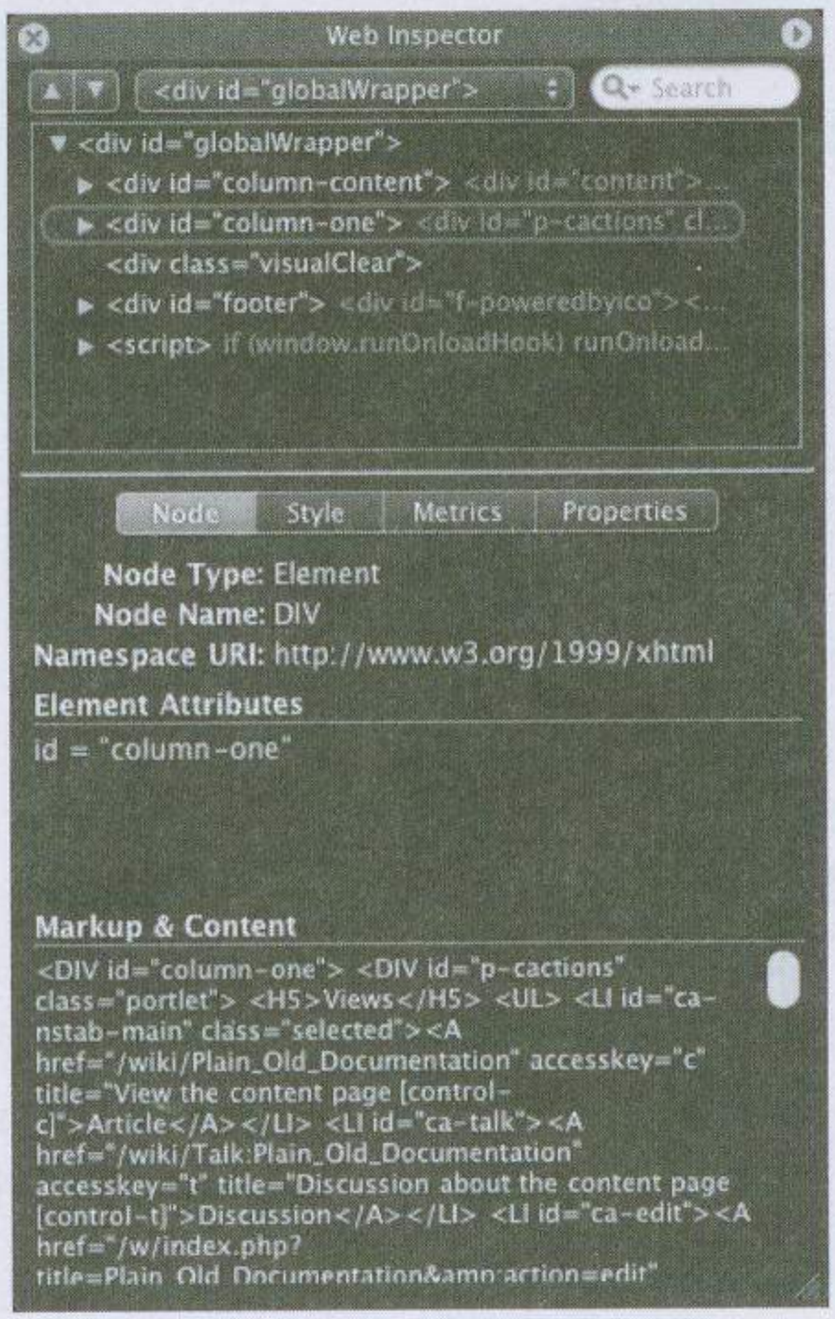


图4-6 Safari内建的DOM查看器

尽管这个扩展已经随最新版本的 Safari 发布，但相比前述的 JavaScript 控制台来说，启用它要更麻烦一点。Safari 团队花了如此多的精力编写并添加这些组件之后却把它们藏起来，为何不显示给希望使用它的开发者们呢，这真是一件很令人奇怪的事情。无论如何，要启用这个 DOM 查看器，你必须执行代码清单 4-2 中的这段命令：

代码清单4-2 启用Safari DOM查看器

```
defaults write com.apple.Safari WebKitDeveloperExtras -bool true
```

Safari 的 DOM 查看器还有很大的改进空间，这是一件好事，因为 Safari 开发组很有才华。不过，目前来说，你最好还是用 Firefox 作为开发的基础，直到 Safari 完全完工并发布出来。

65

3. View Rendered Source（查看渲染后的代码）

最后，我希望介绍一个 Web 开发者最容易获得的 DOM 查看器。Firefox 的 View Rendered Source（查看渲染后的代码）扩展在常规的“查看源代码”菜单项下面，提供了一个新的菜单项，允许你以一种直接而易懂的形式查看 HTML 文档的完整表达。关于这个扩展的更多信息可以在这个网站找到：http://jennifermadden.com/scripts/ViewRenderedSource.html。

除了提供非常自然的源代码视图外，它还提供了对文档每级元素进行层级式的高亮显示，让你更好分清目前在代码中的所处位置，如图4-7所示。

```
<body>
<div>
 <table>
  <tbody>
   <tr>
    <td>
     <blockquote>
      <span>
       <ul>
        <li>
         <p>
         </p>
        </li>
       </ul>
      </span>
     </blockquote>
    </td>
   </tr>
  </tbody>
 </table>
</div>
```

图4-7　Firefox的View Rendered Source扩展

View Rendered Source扩展应该是每个Web开发者工具箱里的必备，它的用途远远超过了基本的“查看源代码”功能，同时还是与更为复杂的DOM查看器扩展之间的一个很好的衔接。

66

4.1.3　Firebug

Firebug是近期出现的最重要的JavaScript开发扩展，这个扩展为JavaScript开发者提供了一个完整的套件，包括错误控制台、调试器和DOM查看器。更多关于此扩展的信息可以在这个网站找到：http://www.joehewitt.com/software/firebug/[①]。

将如此多的工具集成到一起的最大的好处是，你可以很好地定位。比如，当点击一条错误信息时，你就能看到出错的JavaScript文件和出错的那行代码。在这个位置你可以设置断点，这可以用来对代码作步进式执行，从而更好地找到出错的位置。这个扩展的一张截图如图4-8所示。

迄今为止所有的先进工具中，还没有哪个能超过Firebug的。我强烈建议你使用Firefox加上Firebug扩展作为你JavaScript编程的首选基础平台。

---

① Firebug现在有了新的网站：http://getfirebug.com。——译者注

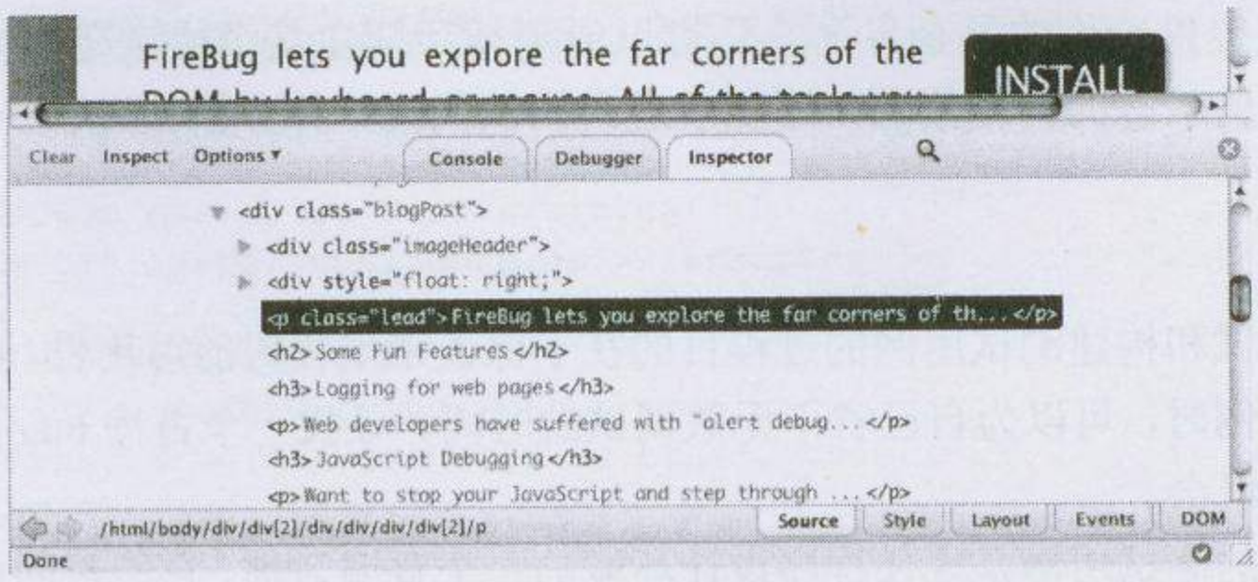


图4-8 Firebug调试扩展

4.1.4 Venkman

JavaScript开发的最后一个工具是Venkman扩展。它最开始作为Mozilla浏览器的一部分开发，是 Mozilla 发起的 JavaScript 调试器项目的代号。关于这个项目和更新后的 Firefox 扩展的更多信息可以在以下几个网站找到：

- Mozilla Venkman项目：http://www.mozilla.org/projects/venkman/。
- Firefox下的Venkman：https://addons.mozilla.org/firefox/216/。
- Venkman教程：http://www.mozilla.org/projects/venkman/venkman-walkthrough.html。

67

使用这个扩展而不是 Firebug 的一个好处在于，它与 JavaScript 引擎本身紧密集成，使你能够对代码的执行作出精确的控制。Firefox Venkman 扩展的截图如图 4-9 所示。

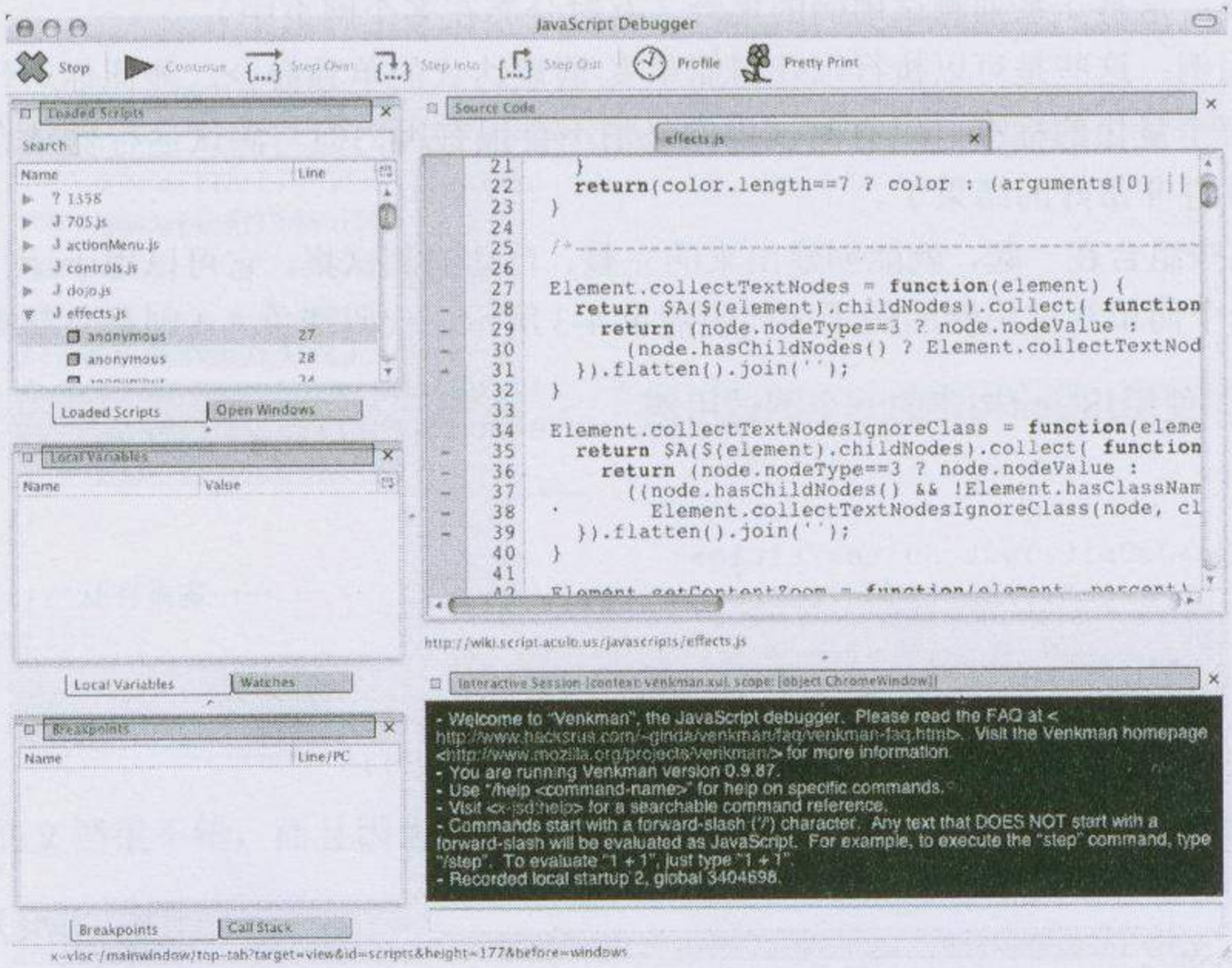


图4-9 很早出现的Venkman JavaScript调试器被移植到了Firefox上

通过这个扩展提供给你的复杂界面，你可以清晰地了解在某个特定范围内可以使用哪些变量、属性或者变量的状态的详细信息，还能步进式调试代码，并分析其过程。

4.2　测试

我个人认为测试和构建测试用例的过程目的在于保证未来代码的可用性。在为代码和程序库构建可靠的测试用例时，可以为自己省下无数调试的时间、寻找一个古怪 bug 上的时间，或者除去了引入新 bug 的机会。

68

构建一套牢固的测试用例是现代程序设计环境的一个普遍习惯，它能帮助你自己和使用你代码的其他人更好地添加新特性和寻找 bug。

在这一节里我将介绍 3 个可以用来构建 JavaScript 测试用例的程序库，它们都能在跨浏览器环境下自动运行。

4.2.1　JSUnit

JSUnit 长期是 JavaScript 单元测试的唯一标准，它的功能基于 Java 世界中很受欢迎的 JUnit 程序包，所以只要你对 JUnit 的运作方式熟悉，使用 JSUnit 库就不会有什么困难。在它的网站 http://www.jsunit 上有大量信息和文档（http://www.jsunit.net/documentation/）。

和大部分单元测试工具集（或者至少和我讨论的这几个）类似，JSUnit 有以下 3 个基本的组件：

- 测试运行器：集合的这个部分为测试整个操作中执行的过程提供了一个方便的图形化输出。它还提供了载入测试集并执行其内容的能力，而且能记录下测试集的输出。
- 测试集：这是一系列测试用例的集合（有时分布在多个网页中）。
- 测试用例：这些是可以执行以获得简单真/假表达式的单独命令，能让你对代码是否执行正确有个量化的标准。只有测试用例恐怕不会很有用，但与测试运行器结合起来就能得到交互性非常好的结果了。

将所有这些组合在一起，就能创建出来的完整、自动的测试集，它可以用于运行并添加进一步的测试。一个简单测试用例的例子如代码清单 4-3 所示，代码清单 4-4 则是一套测试用例。

代码清单4-3　使用JSUnit构建的一个测试用例

```
<html>
<head>
    <title>JsUnit Test Suite</title>
    <script src="../app/jsUnitCore.js"></script>
    <script>
        function suite() {
            var newsuite = new top.jsUnitTestSuite();
            newsuite.addTestPage("jsUnitTests.html");
            return newsuite;
        }
    </script>
</head>
<body></body>
</html>
```

69

代码清单4-4 一个典型测试页面中的多种JSUnit测试用例

```
<html>
<head>
    <title>JsUnit Assertion Tests</title>
    <script src="../app/jsUnitCore.js"></script>
    <script>
        // 测试一个表达式是否为真
        function testAssertTrue() {
            assertTrue("true should be true", true);
            assertTrue(true);
        }

        // 测试一个表达式是否为假
        function testAssertFalse() {
            assertFalse("false should be false", false);
            assertFalse(false);
        }

        // 测试两个参数是否相等
        function testAssertEquals() {
            assertEquals("1 should equal 1", 1, 1);
            assertEquals(1, 1);
        }

        // 测试两个参数是否不等
        function testAssertNotEquals() {
            assertNotEquals("1 should not equal 2", 1, 2);
            assertNotEquals(1, 2);
        }

        // 测试参数是否等于 null
        function testAssertNull() {
            assertNull("null should be null", null);
            assertNull(null);
        }

        // 测试参数是否不等于 null
        function testAssertNotNull() {
            assertNotNull("1 should not be null", 1);
            assertNotNull(1);
        }

        // 还有很多……
    </script>
</head>
<body></body>
</html>
```

70

JSUnit的文档很不错，而且因为它出现得比较早，所以也很容易找到很多不错的例子。

4.2.2 J3Unit

J3Unit是JavaScript单元测试世界中的新成员。这个库相比JSUnit多提供了对服务器端的测

试套件，比如 JUnit 和 Jetty 的直接集成。对 Java 开发者这会非常有用，因为这样他们就能一口气完成客户端和服务器端所有代码的测试了。不过考虑到不是所有人都喜欢 Java，和其他单元测试库一样，J3Unit 也提供了一个可以在浏览器里执行的静态模式。关于 J3Unit 的更多信息可以在它的网站 http://j3unit.sourceforge.net/找到。

因为把客户端测试用例和服务器端代码连接起来的工作很简单，我们只看看 J3Unit 的静态客户端单元测试是怎么进行的。幸运的是它们和其他测试套件几乎一模一样，所以改用 J3Unit 的话也会很简单，如代码清单 4-5 所示。

代码清单4-5　使用J3Unit的一个简单测试

```
<html>
<head>
    <title>Sample Test</title>
    <script src="js/unittest.js" type="text/javascript"></script>
    <script src="js/suiterunner.js" type="text/javascript"></script>
</head>
<body>
    <p id="title">Sample Test</p>
    <script type="text/javascript">
        new Test.Unit.Runner({
            // 测试隐藏或显示一个元素
            testToggle: function() { with(this) {
                var title = document.getElementById("title");
                title.style.display = 'none';
                assertNotVisible(title, "title should be invisible");
                element.style.display = 'block';
                assertVisible(title, "title should be visible");
            }},

            // 测试把一个元素添加到另一个后面
            testAppend: function() { with(this) {
                var title = document.getElementById("title");
                var p = document.createElement("p");
                title.appendChild(p);
                assertNotNull(title.lastChild);
                assertEqual(title.lastChild,p);
            }}
        });
    </script>
</body>
</html>
```


J3Unit 虽然还比较新，但展示了单元测试框架的部分前景。如果你喜欢它面向对象的风格，建议试试。

4.2.3　Test.Simple

JavaScript 单元测试的最后一个例子也是一个新成员。Test.Simple 曾经由 JSAN 引入，作为所有提交给 JSAN 的 JavaScript 模块的标准测试方案。因为它得到了广泛的使用，Test.Simple 也有

大量的文档和例子，这两个方面对使用一套测试框架都非常有益。关于 Test.Simple（和 Test.More，它的配套库）的更多信息可见：

- Test.Simple：http://openjsan.org/doc/t/th/theory/Test/Simple/。
- Test.Simple文档：http://openjsan.org/doc/t/th/theory/Test/Simple/0.21/lib/Test/Simple.html。
- Test.More文档：http://openjsan.org/doc/t/th/theory/Test/Simple/0.21/lib/Test/More.html。

Test.Simple 库提供了大量用于测试的方法和一个执行自动测试的运行器。范例 Test.Simple 测试集的一个例子如代码清单 4-6 所示。

代码清单4-6 使用Test.Simple和Test.More来执行测试

```
// 载入 Test.More 模块 (对自身进行测试!)
new JSAN('../lib').use('Test.More');
// 计划作 6 个测试 (以了解什么时候出现错误)
plan({tests: 6});
// 测试3个简单的用例
ok( 2 == 2,                'two is two is two is two' );
is( "foo", "foo",          'foo is foo' );
isnt( "foo", "bar",        'foo isnt bar');
// 运用正则表达式进行测试
like("fooble", /^foo/,     'foo is like fooble');
like("FooBle", /foo/i,     'foo is like FooBle');
like("/usr/local/", '^\/usr\/local', 'regexes with slashes in like' );
```

72

我个人喜欢 Test.Simple 和 Test.More 的简洁，因为应用它们不需要添加太多的额外代码，让你的代码更为简单。不过最终还是由你自己决定选择哪个测试套件，毕竟测试套件的重要性也是不容忽视的。

4.3 小结

尽管对有经验的程序员来说，本章内容没什么新颖的东西，但将这些概念和 JavaScript 的使用融合到一起，就能改进 JavaScript 的最终可用性，使之成为一个专业的程序设计语言。我强烈建议你尝试一下这里调试和测试的过程，它肯定能让你写出更好、更整洁的 JavaScript 代码来。

73

Part 3

第三部分

分离式 JavaScript

本部分内容

- 第 5 章　DOM
- 第 6 章　事件
- 第 7 章　JavaScript 与 CSS
- 第 8 章　改进表单
- 第 9 章　制作图库

第5章

DOM

在上一次 Web 开发领域的技术浪潮中，文档对象模型（Document Object Model，DOM）无疑是开发者能用来提升用户体验的最重要技术之一。

用 DOM 脚本编程为页面增加分离式（unobtrusive）JavaScript，既能为你的用户提供各种各样时髦的增效，同时又不会影响那些使用不支持或禁用了 JavaScript 的浏览器的人们。此外，DOM 脚本编程还能精细地分离和轻松地管理你的代码。

幸运的是，所有的现代浏览器都支持 DOM 和当前 HTML 文档的内置 HTML DOM 表达。有了这些后就可以通过 JavaScript 轻松访问它们，从而为现代 Web 开发者提供了巨大的便利。理解如何使用并尽情发挥这些技术，你的下一个 Web 应用程序将会领先一筹。

这一章探讨关于 DOM 的一系列话题。假定你是 DOM 新手，我们将从基础开始，并涉及所有重要的概念。而对已经熟悉了 DOM 的读者，我保证能提供一批你会喜欢的高级技术。

5.1 DOM 简介

DOM 是一个表达 XML 文档的标准（由 W3C 制定的）。它未必是最快的方式，也未必是最轻量级的或者最容易使用的，但确是应用最广泛的，大部分 Web 开发的编程语言（比如 Java、Perl、PHP、Ruby、Python 和 JavaScript）都提供了相应的 DOM 实现。DOM 给开发者提供了一种定位 XML 层级结构的直观方法。就算你尚未完全熟悉 XML 也没关系，HTML 文档（实际上从浏览器角度来说就是 XML 文档）已经有一个可用的 DOM 表达，你会为此而备受鼓舞。

5.2 遍历 DOM

可以把 XML 的 DOM 表达方式看作是一棵导航树。一切术语都跟家谱术语（父、子、兄弟）类似。而与家谱的不同之处在于，XML 文档从一个独立的根节点（称作文档元素，document element）开始，它包含指向子节点的指针。每一个子节点都包含指针指向它的父节点、相邻节点和子节点。

DOM 还使用了一些特殊的术语来描述 XML 树里的对象种类。DOM 树中的每个对象都是一个节点（node），每个节点可以有一个类型（type），比如元素、文本或者文档。要进一步学习下去，我们必须了解 DOM 文档是如何表现和如何定位的。请参考以下简单的 HTML 片段，让我们来研究研究它的 DOM 结构是如何工作的：

```
<p><strong>Hello</strong> how are you doing?</p>
```

这个片段的每一部分被分解为独立的 DOM 节点指针，指向其亲戚（父、子、兄弟）。使用图谱来表示的话，它应该如图 5-1 所示：片断的每部分（圆角框描述元素，直角框描述文本节点）和它的引用。

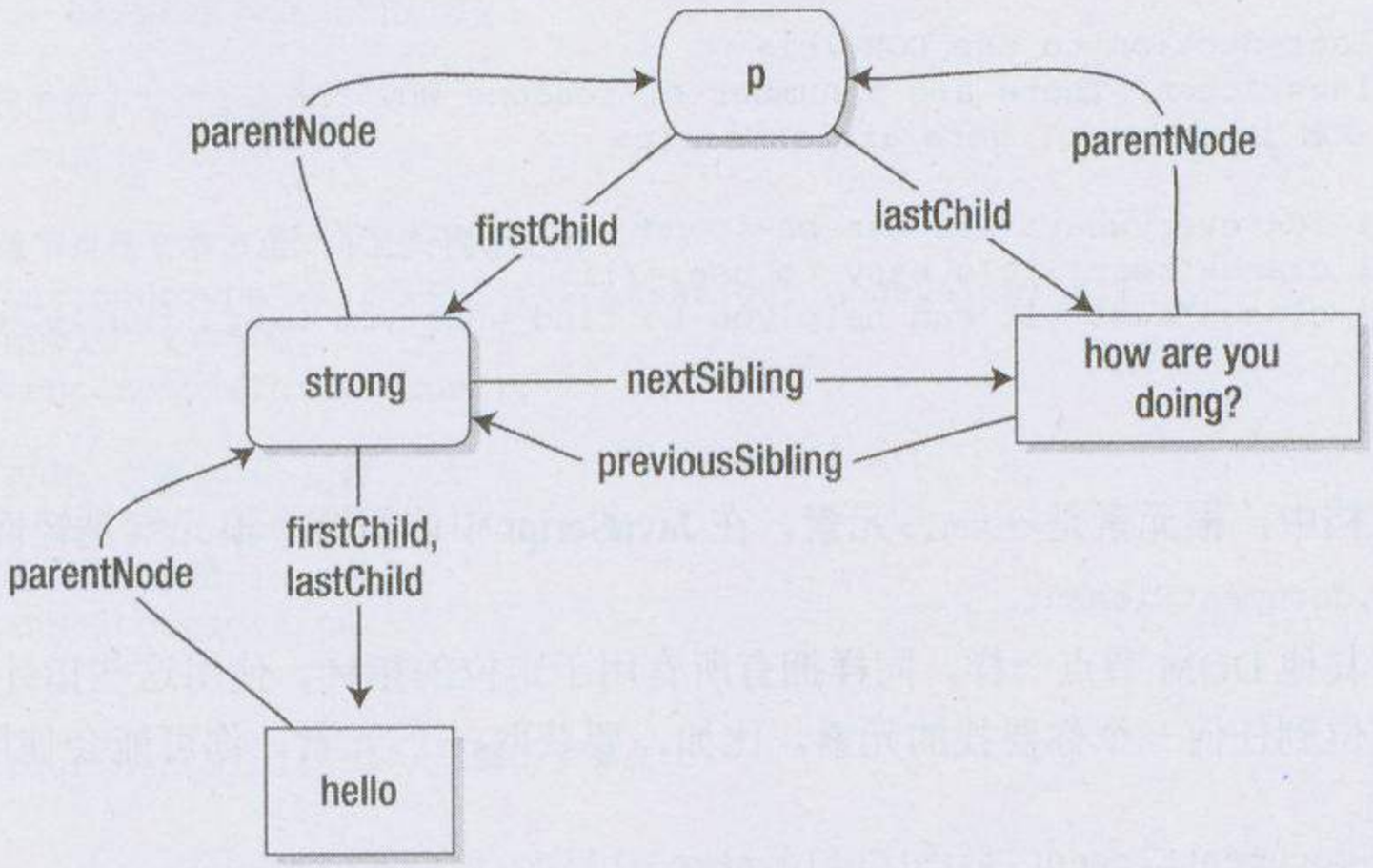


图5-1 节点间的关系

每个独立的 DOM 节点都包含指向它的相关节点的一系列指针。你需要使用这些指针来学习如何遍历 DOM。所有可用的指针如图 5-2 所示。DOM 节点指针的各个属性，分别是指向另一个 DOM 元素的指针（如果不存在则为 null）。

78

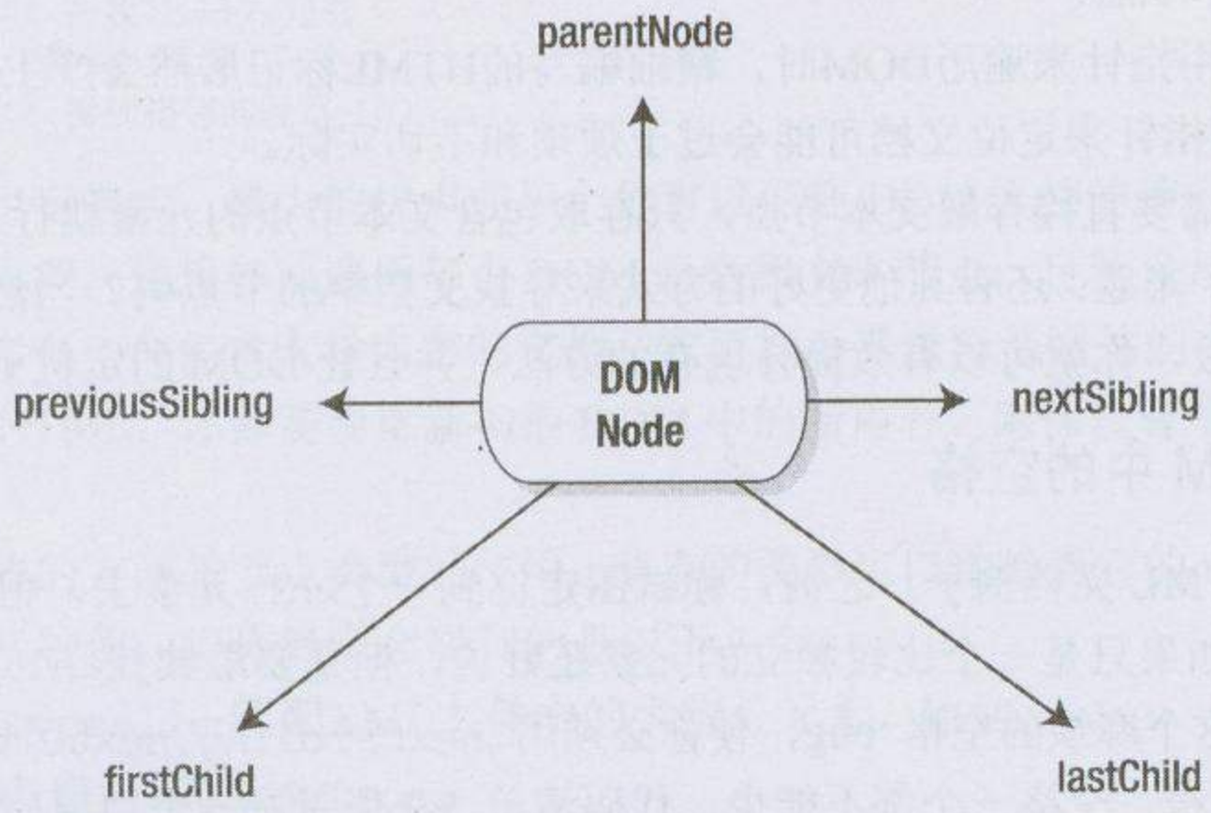


图5-2 使用指针遍历DOM树

只需使用不同的指针，就可以定位页面上的任何元素或者文本块。要理解它在实际环境中是如何工作的，最好的方式就是实践一下。代码清单 5-1 是一个简单的 HTML 页。

代码清单5-1　简单HTML页面，或者说是简单的XML文档

```
<html>
<head>
  <title>Introduction to the DOM</title>
</head>
<body>
  <h1>Introduction to the DOM</h1>
  <p class="test">There are a number of reasons why the
      DOM is awesome, here are some:</p>
  <ul>
    <li id="everywhere">It can be found everywhere.</li>
    <li class="test">It's easy to use.</li>
    <li class="test">It can help you to find what you want, really quickly.</li>
  </ul>
</body>
</html>
```

在这个文档中，根元素是<html>元素。在 JavaScript 中访问这个根元素是轻而易举的：

79

```
document.documentElement
```

根节点与其他 DOM 节点一样，同样拥有所有用于定位的指针。使用这些指针你就可以浏览整个文档，定位到任何一个你要找的元素。比如，要获取<h1>元素，你可能会使用如下方法：

```
//不起作用
document.documentElement.firstChild.nextSibling.firstChild
```

这是我们遇到的第一个难题：DOM 指针不仅可以指向元素，也可以指向文本节点。在这里，代码并不会真正指向<h1>元素，而是指向了<title>元素。为什么会发生这样的事？原因在于 XML 存在一个有争议的地方：空格。或许你已经注意到，在<html>和<head>元素之间有一个行结束符，它被认为是一个空格，这就意味着第一个节点是文本节点，而不是<head>元素。从中我们可以了解到以下 3 点：

- 当试图只使用指针来遍历DOM时，精细编写的HTML标记居然会产生混乱。
- 仅使用DOM指针来定位文档可能会过于烦琐和不切实际。
- 通常，你不需要直接存取文本节点，只存取包含文本节点的元素就行了。

这让我们遇到了难题：还有其他更好的方式来寻找文档中的节点吗？当然有！在你的工具箱中创建一些辅助函数，你就可以有效提升现有的方法，并且让 DOM 的定位更为方便。

5.2.1　处理 DOM 中的空格

回到我们的 HTML 文档例子。之前，你试图定位到一个<h1>元素去，但由于捣乱的文本节点使得困难重重。如果只是一个比较独立的元素还好说，但是要继续找<h1>元素的下一个元素呢？你仍然会遭遇这个麻烦的空格 bug，使你必须用 .nextSibling.nextSibling 来跳过<h1>和<p>之间的行结束符，空格一个都不能少。代码清单 5-2 所示是一个用以处理空格 bug 的补救办法。这个特别的技术删除所有空格——当然只是 DOM 文档中的文本节点，让 DOM 的遍历更容易。这样做不仅不会对你的 HTML 渲染产生副作用，还会让定位 DOM 变得更容易。但是需要注意的是，这个函数的结果并不是持久的，它需要在 HTML 文档的每一次加载时都重新执行一遍。

80

代码清单5-2 XML文档空格bug的补救方案

```
function cleanWhitespace( element ) {
  // 如果不提供参数，则处理整个HTML文档
  element = element || document;
  // 使用第一个子节点作为开始指针
  var cur = element.firstChild;

  // 一直到没有子节点为止
  while ( cur != null ) {

    // 如果节点是文本节点，并且只包含空格
    if ( cur.nodeType == 3 && ! /\S/.test(cur.nodeValue) ) {
      // 删除这个文本节点
      element.removeChild( cur );

      // 否则，它就是一个元素
    } else if ( cur.nodeType == 1 ) {
      // 递归整个文档
      cleanWhitespace( cur );
    }

    cur = cur.nextSibling; // 遍历子节点
  }
}
```

假设你要在前面例子中使用这个函数来查找位于<h1>元素后的元素。那么代码应该类似这样：

```
cleanWhitespace();

// 查找H1元素
document.documentElement
.firstChild // 查找Head元素
.nextSibling // 查找<body>元素
.firstChild // 得到H1元素
.nextSibling // 得到相邻的段落（p）
```

该技术有优点也有缺点。最大的优点在于，你可以保证 DOM 文档的遍历在一定程度上的稳定性。但明显性能太差，想想必须遍历每个 DOM 元素和文本节点，目的只是为了找出包含空格的文本节点。假设你有一个包含大量内容的文档，它可能会严重降低网站的加载速度。此外，每次为文档注入新的 HTML，你都需要重新扫描 DOM 中的新内容，确保没有增加新的有空格填充的文本节点。

81

此外此函数重要的方面是节点类型的使用。节点的类型可以由检查它的 nodeType 属性来确定。可能会出现好几种值，但你经常会碰到的是以下 3 个：

- 元素（nodeType=1）：匹配XML文件中的大部分元素。比如，<li>、<a>、<p>和<body>元素都有一个值为1的nodeType。
- 文本(nodeType=3)：匹配文档内的所有文本块。当使用previousSibling和nextSibling来遍历DOM结构时，你会经常碰到元素内和元素间的文本块。
- 文档（nodeType=9）：匹配文档的根元素。比如，在HTML文档内，它是<html>元素。

此外，你可以用常量来表明不同的DOM节点类型（但只是对非IE浏览器有用）。与其去记住1、3或9，还不如直接直观地使用document.ELEMENT_NODE、document.TEXT_NODE或者document.DOCUMENT_NODE。因为经常清空DOM的空格会让人感到厌烦，所以你应该探索其他遍历DOM结构更有效的方法。

5.2.2　简单的DOM遍历

可以使用纯粹的DOM遍历规则（每个遍历方向都有指针）来开发一些更适合你的HTMLDOM文档遍历函数。大部分Web开发者在大多数情况下仅仅需要遍历DOM元素而非相邻的文本节点，该规则就是基于这样的事实而制定的。以下一系列的辅助函数可以帮助你，它们能够取代标准的previousSibling、nextSibling、firstChild、lastChild和parentNode。

代码清单5-3展示的函数，返回的是当前元素的前一个元素，如果前一个元素不存在则是null，类似于元素的previousSibling属性。

代码清单5-3　查找相关元素的前一个兄弟元素的函数

```
function prev( elem ) {
  do {
    elem = elem.previousSibling;
  } while ( elem && elem.nodeType != 1 );
  return elem;
}
```

代码清单5-4展示的函数，返回的是当前元素的下一个元素，如果下一个元素不存在则是null，类似于元素的nextSibling属性。

82

代码清单5-4　查找相关元素的下一个兄弟元素的函数

```
function next( elem ) {
  do {
    elem = elem.nextSibling;
  } while ( elem && elem.nodeType != 1 );
  return elem;
}
```

代码清单5-5展示的函数，返回的是当前元素的第一个子元素，类似于firstChild元素属性。

代码清单5-5　查找元素第一个子元素的函数

```
function first( elem ) {
  elem = elem.firstChild;
  return elem && elem.nodeType != 1 ?
      next ( elem ) : elem;
}
```

代码清单5-6展示的函数，返回的是当前元素的最后一个子元素，类似lastChild元素属性。

代码清单5-6　查找元素最后一个子元素的函数

```
function last( elem ) {
  elem = elem.lastChild;
```

```
  return elem && elem.nodeType != 1 ?
  prev ( elem ) : elem;
}
```

代码清单 5-7 展示的函数，返回当前元素的父元素，类似 parentNode 元素属性。你可以一次用一个数字来操纵多个父元素，例如 parent(elem,2)就等同于 parent(parent(elem))。 83

代码清单5-7 查找元素父元素的函数

```
function parent( elem, num ) {
num = num || 1;
for ( var i = 0; i < num; i++ )
if ( elem != null ) elem = elem.parentNode;
return elem;
}
```

使用这些新函数你就可以迅速遍历 DOM 文档了，而且不必再为元素间的文本操心。比如，需要查找<h1>元素的下一个元素，现在可以这么做了：

```
// 查找<h1>元素的下一个元素
next( first( document.body ) )
```

在这段代码内有两件事值得注意。首先，此处有一个新的引用：document.body。所有的现代浏览器都在 HTML DOM 文档内通过 body 属性提供一个对<body>元素的引用。你可以用它来简化你的代码。另一件事情你可能已经注意到，那就是这些函数的书写方式十分违反直觉。一般来说，在遍历时会想：“从<body>元素开始，获取第一个元素，然后获取下一个元素”。但从我们实际编写方式来看，它是按相反的顺序。要解决这个问题，我们将讨论一些可以让自定义的遍历代码更清晰的方式。

5.2.3 绑定到每一个 HTML 元素

Firefox 和 Opera 中存在一个强大的对象原型（object prototype）叫 HTMLElement，它允许你为每一个 HTML DOM 元素绑定函数和数据。上一节所描述的函数显得特别愚钝，但它们应该可以更清晰的。一个完美的方案是，为 HTMLElement 的原型直接绑定函数，由此每个独立的 HTML DOM 元素也直接绑定了你的函数。为了能让上一节描述的函数继续有效，你必须做出 3 个改变：

(1) 你需要在函数的顶部增加一行代码，使用 this 来引用元素，而不是从参数的变量中获取它。

(2) 你需要删除不再需要的元素参数。

(3) 你需要把函数绑定到HTMLElement原型，由此DOM中的各个HTML元素可以使用该函数。

新的 next 函数如代码清单 5-8 所示。

代码清单5-8 为所有HTML DOM元素动态绑定新的DOM遍历函数

```
HTMLElement.prototype.next = function() {
  var elem = this;
  do {
    elem = elem.nextSibling;
  } while ( elem && elem.nodeType != 1 );
  return elem;
};
```

现在你可以这样来使用新的 next 函数（和经过同样调整后的其他函数）了：

84

```
// 一个简单的例子 —— 获取<p>元素
document.body.first().next()
```

这让你的代码更加清晰和易于理解。现在你可以按照自然想法的顺序来编写代码了，JavaScript 总体上也更清晰了。如果你对这种编程风格感兴趣，我特别推荐尝试一下 jQuery JavaScript 库（http://jquery.com），它大量使用到这种技术。

注意　因为只有3个现代浏览器（Firefox、Safari和Opera）支持HTMLElement，要让它也能在IE中工作，必须给予特别的处理。有一个可以直接使用的库，由Jason Karl Davis（http://browser-land.org）编写，它为两个不支持的浏览器提供了访问HTMLElement的方法。可以从这找到该库的具体信息：http://www.browserland.org/scripts/htmlelement/.

5.2.4　标准的 DOM 方法

所有的现代 DOM 实现都包含一系列让编程更轻松的方法。结合一些自定义函数，遍历 DOM 可以是一种平滑的体验。那么让我们从 JavaScript DOM 的两个强大方法开始：

- getElementById("everywhere")：该方法只能运行在document对象下，从所有中找出ID是everywhere的元素。这是一个非常强大的函数，并且是迅速的访问一个元素的最快方法。
- getElementsByTagName("li")：该方法能运行在任何元素下，找出该元素下的所有标签名为li的后代元素，并返回一个NodeList（节点列表）。

警告　getElementById本来会按照你的预期工作：查找所有元素，揪出一个包含id的属性，且值为指定值的元素。但是，如果你加载一个远程XML文档，并使用getElementById（或是使用除了JavaScript之外的实现DOM的语言），它默认并不会使用id属性。这是由于设计上的问题，一个XML文档如需明确指出id属性是什么，通常使用XML的定义或配置。

警告　getElementsByTagName返回一个节点NodeList.该结构跟普通的JavaScript数组非常相似，但一个重要的不同之处在于：它并不支持.push()、.pop()、.shift()等JavaScript数组的常用方法。使用getElementsByTagName时请务必记住这点，这能把你从很多混乱中解救出来。

85

这两个方法在所有现代的浏览器都可用，对定位到指定元素有非常大的帮助。回头看看我们前面尝试查找<h1>元素的例子，现在可以这么做了：

```
document.getElementsByTagName("h1")[0]
```

这行代码保证有效并返回文档中的第一个<h1>元素。再回头看看例子文档，假设你要获取所有<li>元素并为之加上边框。

```
var li = document.getElementsByTagName("li");
for ( var j = 0; j < li.length; j++ ) {
  li[j].style.border = "1px solid #000";
}
```

最后，需要使第一个<li>元素的文本加粗，假设它已经有了一个关联它的 id：

```
document.getElementById("everywhere").style.fontWeight = 'bold';
```

你可能已经注意到获取指定 ID 的单个元素需要冗长的代码，依靠标签名获取元素也如此。其实你可以封装一个函数来简化获取的过程：

```
function id(name) {
  return document.getElementById(name);
}
```

代码清单 5-9 展示了一个函数，它依靠 HTML DOM 内的文档标签来定位元素。该函数带一个或两个参数。如果只传入一个标签名作为参数，函数将会遍历整个文档。另外你可以传入一个 DOM 上下文元素作为第一个参数的辅助，则函数只遍历该参数元素下的标签。

代码清单5-9　依靠HTML DOM文档标签定位元素的函数

```
function tag(name, elem) {
  // 如果不提供上下文元素，则遍历整个文档
  return (elem || document).getElementsByTagName(name);
}
```

让我们重新回到寻找<h1>元素后的第一个元素的问题。十分幸运的是，现在的代码可以比之前任何一次的都要精炼：

```
// 查找<h1>元素后的第一个元素
next( tag("h1")[0] );
```

以上这些函数为你在 DOM 文档中获取元素提供了强大而迅速的解决方法。在进一步学习修改 DOM 的方法之前，首先需要大致了解脚本首次执行时 DOM 的加载问题。 86

5.3 等待 HTML DOM 的加载

处理 HTML DOM 文档存在的一个难题是，JavaScript 可以在 DOM 完全加载之前执行，这会给你的代码引发不少的潜在问题。浏览器的渲染和操作顺序大致如以下列表：

- HTML解析完毕。
- 外部脚本和样式表加载完毕。
- 脚本在文档内解析并执行。
- HTML DOM完全构造起来。
- 图片和外部内容加载。

- 网页完成加载。

在网页头部并且从外部文件加载的脚本会在 HTML 真正构造之前执行。如前所述，这是个至关重要的问题，因为这两处执行的脚本并不能访问还不存在的 DOM。幸好，我们还有若干的补救办法。

5.3.1　等待整个页面的加载

目前，最常用的技术是完全等待整个页面加载完毕才执行 DOM 操作。这种技术只需利用 window 对象的 load 事件来绑定一个函数，页面加载完毕即可触发。我们将会在第 6 章更详细地讨论事件。代码清单 5-10 展示了一个页面完成加载后执行 DOM 相关代码的例子。

代码清单5-10　使用addEvent函数为window.onload属性绑定回调函数

```
// 直到页面加载完毕
// （使用的addEvent，下一章会有阐述）
addEvent(window, "load", function() {
    // 执行HTML DOM操作
    next( id("everywhere") ).style.background = 'blue';
});
```

最简单的操作却是最慢的。在加载过程的顺序列表中，你会注意到页面的加载完毕与否完全被最后一步所掌控。这就是说，如果你页面有很多的图片、视频等，用户可能得等上一段时间 JavaScript 才执行。

5.3.2　等待大部分 DOM 的加载

第二种技术相当绕弯子，所以并不是十分推荐使用。如果你还记得，前面说过行内的脚本在 DOM 构造后就可立即执行。这是准真理。只有在 DOM 构造后，执行到该位置上脚本才真正执行。这意味着在你在页面中途嵌入的行内脚本只能访问该位置之前的 DOM。所以，在页面最后元素之前嵌入脚本，你就可以在 DOM 中访问这个元素之前全部的元素，成为一条模拟 DOM 加载的伪路子。该方法典型的实现如代码清单 5-11 所示。

87

代码清单5-11　依靠在HTML DOM最后插入一个<script>标签（包含一个函数调用）检查DOM是否已加载完毕

```
<html>
  <head>
    <title>Testing DOM Loading</title>
    <script type="text/javascript">
      function init() {
        alert( "The DOM is loaded!" );
        tag("h1")[0].style.border = "4px solid black";
      }
    </script>
  </head>
  <body>
    <h1>Testing DOM Loading</h1>
    <!-- 这里是大量的HTML -->
```

```
    <script type="text/javascript">init();</script>
  </body>
</html>
```

在这个例子中，你必须把行内脚本作为 DOM 的最后一个元素，以便它最后一个被解析和执行。它唯一执行的是初始化函数，该函数应该包含你需要操作的 DOM 的相关代码。这个解决方案的最大问题在于混乱：你为 HTML 增添无关标记的理由仅仅是检查 DOM 是否已经执行。该技术通常被认为是不合理的，因为你为页面增加额外的代码的目的只是检查加载状态而已。

5.3.3 判断 DOM 何时加载完毕

最后一种技术可用以监听 DOM 加载状态，可能是最复杂的（从实现角度来看），但也是最有效的。在该技术中，你既能像绑定 window 加载事件那般简单，又能获得行内脚本技术那样的速度。

这项技术在不堵塞浏览器加载的情况下尽可能快地检查 HTML DOM 文档是否已经加载了执行所必须的属性。以下是检查 HTML DOM 是否可用的几个要点：

(1) document：你需要知道 DOM 文档是否已经加载。若能足够快地检查，运气好的话你会看到 undefined。

(2) document.getElementsByTagName 和 documents.getElementById：频繁使用 document.getElementsByTagName 和 document.getElementById 函数检查文档，当存在这些函数则表明已完成加载。

(3) document.body：作为额外补充，检查<body>元素是否已经完全加载。理论上前一个检查应已能够作出判断，但我发现有些情况下还是不够。

使用这些检查就足够判断 DOM 是否可用了（“足够”在此表示可能会有一定毫秒级的时间差）。这个方法几乎没有瑕疵。单独使用前述检查，脚本应该可以在现代浏览器中运行得相对良好。但是，最近 Firefox 实现了缓存改进，使得 window 加载事件实际上可以在脚本能检查到 DOM 是否可用之前触发。为了能发挥这个优势，我同时为 window 加载事件附加检查，以期能获得更快的执行速度。

最后，domReady 函数集合了所有需要在 DOM 可用时就执行的函数的引用。一旦 DOM 被认为是可用的，就调用这些引用并按顺序一一执行。代码清单 5-12 展示了一个监听 DOM 何时加载完毕的函数。

代码清单5-12 监听DOM是否可用的函数

```
function domReady( f ) {
  // 假如 DOM 已经加载，马上执行函数
  if ( domReady.done ) return f();

  // 假如我们已经增加了一个函数
  if ( domReady.timer ) {
    // 把它加入待执行函数清单中
    domReady.ready.push( f );
  } else {
```

```
    // 为页面加载完毕绑定一个事件，
    // 以防它最先完成。使用addEvent(该函数见下一章)。
    addEvent( window, "load", isDOMReady );

    // 初始化待执行函数的数组
    domReady.ready = [ f ];

   // 尽可能快地检查DOM是否已可用
    domReady.timer = setInterval( isDOMReady, 13 );
  }
}
// 检查DOM是否已可操作
function isDOMReady() {
  // 如果我们能判断出DOM已可用，忽略
  if ( domReady.done ) return false;

  // 检查若干函数和元素是否可用
  if ( document && document.getElementsByTagName &&
       document.getElementById && document.body )
{

    // 如果可用，我们可以停止检查
    clearInterval( domReady.timer );
    domReady.timer = null;

    // 执行所有正等待的函数
    for ( var i = 0; i < domReady.ready.length; i++ )
      domReady.ready[i]();

    // 记录我们在此已经完成
    domReady.ready = null;
    domReady.done = true;
  }
}
```

现在该来看看这在 HTML 文档中是如何执行的。domReady 函数应该按 addEvent 函数（第6章将讨论）的方式使用，文档可操作时即绑定你需要执行的函数。代码清单 5-13 展示了如何使用 domReady 函数来监听 DOM 是否可用。在这个例子中，假设已经将 domReady 函数写到一个名为 domready.js 的外部文件中。

代码清单5-13　使用domReady函数来确定何时DOM可操作和修改

```
<html>
  <head>
    <title>Testing DOM Loading</title>
    <script type="text/javascript" src="domready.js"></script>
    <script type="text/javascript">
      function tag(name, elem) {
        // 如果不提供上下文元素，搜索整个文档
        return (elem || document).getElementsByTagName(name);
      }
domReady(function() {
    alert( "The DOM is loaded!" );
```

```
    tag("h1")[0].style.border = "4px solid black";
    });
  </script>
</head>
<body>
  <h1>Testing DOM Loading</h1>
  <!--这里是大量的HTML -->
</body>
</html>
```

现在你已经掌握若干定位一般 XML DOM 文档与如何补救 HTML DOM 文档加载难题的方法，另一个问题也该摆上台面了：还有更好的方法来查找 HTML 文档中的元素吗？当然，答案是十分肯定的。

5.4 在 HTML 文档中查找元素

在 HTML 文档中与在 XML 文档中查找元素通常情况下还是有很大不同的。现代 HTML 事实上是 XML 的一个子集，这么说来似乎有点自相矛盾。但 HTML 文档包含了许多本质的不同，而这正是你可以充分利用的地方。

对 JavaScript/HTML 开发者来说，最重要的两个优势是利用类和 CSS 选择器（selector）。将这熟记于心，你就可创建一系列强大的函数，使 DOM 的操作更简单、更易理解。

5.4.1 通过类的值查找元素

通过类名字定位元素是一种很普遍的技术，2003 年由 Simon Willison（http://simon.incutio.com）普及，而原创的则是 Andrew Hayward（http://www.mooncalf.me.uk）。该技术直截了当：遍历所有的元素（或者某元素的所有后代元素）直至找到指定类的元素。一种可能的实现方法如代码清单 5-14 所示。

代码清单5-14 找出全部有指定类值的元素的函数

```
function hasClass(name,type) {
  var r = [];
  // 定位到类值上（允许多类值）
  var re = new RegExp("(^|\\s)" + name + "(\\s|$)");
  // 限制类型的查找，或者遍历所有元素
  var e = document.getElementsByTagName(type || "*");
  for ( var j = 0; j < e.length; j++ )
    // 如果元素拥有指定类，把它添加到函数的返回值中
    if ( re.test(e[j]) ) r.push( e[j] );

  // 返回符合的元素列表
  return r;
}
```

91

现在你就可使用这个函数来在任意的或者指定类型（比如<li>或<p>）的元素中迅速查找有指定类值的元素。指定标签名会比遍历所有元素（*）的查找更快，因为需检索的目标更少。比

如，在我们的HTML文档中，如果需要找出类值有test的所有元素你可以这么做：

```
hasClass("test")
```

如果只需找类值有test的<li>元素，则可以这么做：

```
hasClass("test","li")
```

当然，如果只需找出类值有test的<li>元素的第一个，则可以这么做：

```
hasClass("test","li")[0]
```

这个函数本身就已非常强大，一旦结合getElementById和getElementsByTagName，你就可以创建更强大的工具集合解决大部分棘手的DOM问题了。

5.4.2　使用CSS选择器查找元素

作为一个Web开发者，你应该已经知道选择HTML元素的一种方案：CSS选择器。CSS选择器是用于赋予元素样式的表达式。随着CSS标准（1、2和3）的每次修订，选择器规范增加了越来越多的重要特点，由此开发者更容易精确定位到所需的元素。不幸的是，浏览器对CSS 2和CSS 3的实现慢得不可思议，所以你或许并不知道CSS中某些新奇酷的特点。如果你对这些感兴趣，建议你浏览W3C的这些主题页面：

- CSS 1选择器：http://www.w3.org/TR/REC-CSS1#basic-concepts。
- CSS 2选择器：http://www.w3.org/TR/REC-CSS2/selector.html。
- CSS 3选择器：http://www.w3.org/TR/2005/WD-css3-selectors-20051215/。

每种选择器规范可用的特点基本上都差不多，每一个后续的版本也都包含前一版的所有特点，同时，每次新版都会增加一系列的新特点。举个例子，CSS 2包含了特性（attribute）选择器和子选择器，而CSS 3则提供了额外的语言支持、性质类型选择以及逻辑非等。比如，以下所有这些都是正确的CSS选择器。

92

- #main p：该表达式查找一个id为main的元素下所有的 元素，然后是这些元素下所有的<p>元素。它们都是CSS 1下的选择器。
- div.items>p：该表达式查找所有的类值为items的 元素，然后定位到所有的子<p>元素。这是正确的CSS 2选择器。
- div:not(.items)：这则定位到所有没有值为items的类的所有 元素。这是正确的CSS 3选择器。

现在，你可能会觉得疑惑，实践中如果并不能使用它们来定位元素（而只能赋予样式）的话，为何会在此讨论CSS选择器。实际上很多前卫的开发者已经涉足开发能兼容CSS1甚至完全支持CSS 3的选择器的实现。使用这些库能让你顺利、方便地选择任意元素并对它们进行操作。

1．cssQuery

第一个完全支持CSS 1-3可用的公开库叫cssQuery，由Dean Edwards（http://dean.edwards.name）创立。其背后的动机很简单：你提供CSS选择器，cssQuery帮你找出所有匹配的元素。此外，cssQuery可以分解成多个子库，有一个是每个CSS选择器“管理员”，它甚至能执行CSS 3的语

法，如果有必要的话。这个独特的库非常复杂但能运行在所有现代浏览器上（Dean 是一位跨浏览器支持的忠实拥趸）。要应用整个库，你需要提供选择器，也可选择加入上下文元素以便加快搜索。以下是例子：

```
// 查找所有<div>元素的子<p>元素
cssQuery("div > p");

// 查找所有的<div>, <p>和<form>元素
cssQuery("div,p,form");

// 查找所有<p>元素和<div>元素，然后查找在这些元素内的a元素
var p = cssQuery("p,div");
cssQuery("a",p);
```

执行 cssQuery 函数会返回匹配的元素列表。由此你就可以像使用 `getElementsByTagName` 一样对元素进行操作。比如，为所有指向 Google 的链接增加边框，可按如下方法做：

```
// 为所有指向Google的链接增加边框
var g = cssQuery("a[href^='google.com']");
for ( var i = 0; i < g.length; i++ ) {
  g[i].style.border = "1px dashed red";
}
```

关于 cssQuery 的更多信息可以在 Dean Edwards 的网站上获取，那里同时也提供完整的源码下载：http://dean.edwards.name/my/cssQuery/.

93

提示 Dean Edwards是一位JavaScript魔术师，他的代码令人惊讶。强烈建议你稍加浏览他的cssQuery库，看看他的JavaScript扩展性写得多么好。

---

2. jQuery

尽管 jQuery 是 JavaScript 库世界的新成员，但它提供了一些新颖而引人注目的 JavaScript 编程方式。我最初只想把 jQuery 写成一个“简便”的 CSS 选择器库，类似于 cssQuery，直到 Dean Edwards 发布了卓越的 cssQuery 库，迫使我开辟了另一个不同的方向。这个库提供了完整的 CSS 1-3 的支持，同时也有基本的 XPath 支持。在此之上，它还提供了进一步定位和操作 DOM 的能力。跟 cssQuery 一样，jQuery 完全支持现代浏览器。以下是使用混合 CSS 和 XPath 的 jQuery 自定义方式来选择元素的例子：

```
// 查找所有类值为'links', 同时内有p元素的<div>元素
$("div.links[p]")

// 查找所有<p>, <div>元素的所有子孙元素
$("p,div").find("*")

// 查找指向Google的所有偶数链接
$("a[@href^='google.com']:even")
```

现在，如需进一步使用 jQuery 元素选择返回的结果，你有两种方式。第一，你可以运行

$("expression").get()来获取匹配元素的列表——这跟 cssQuery 完全一致。第二，你可以使用 jQuery 内置的特殊函数来操作 CSS 和 DOM。所以，回到使用 cssQuery 为指向 Google 的链接加边框的例子上，你现在就可以这么做：

```
// 为所有指向Google的链接增加边框
$("a[@href^=google.com]").css("border","1px dashed red");
```

你可以从 jQuery 项目网站上找到大量的例子、演示和文档等，此外，还有可定制的下载：http://jquery.com/。

94

注意　应该指出的是，cssQuery或jQuery实际上并不只是能定位HTML文档而已，它们可以在任何XML文档上使用。至于纯粹的XML形式定位，请继续阅读下面的XPath部分。

5.4.3　XPath

XPath 表达式是一种不可思议的定位 XML 文档的强大的方式。自问世几年来，几乎可以肯定 DOM 实现的背后必有 XPath。尽管相对冗长，但 XPath 表达式比起 CSS 选择器来，能做的事情更多也更强大。表 5-1 并行比较了一些 CSS 选择器和 XPath 表达式的某些不同。

表 5-1　CSS 3 选择器与 XPath 表达式的比较

目　标	CSS 3	XPath
所有元素	*	//*
所有<p>元素	p	//p
所有子元素	p> *	//p/*
由 ID 获取元素	#foo	//*[@id='foo']
由类获取元素	.foo	//*[contains(@class, 'foo')]
由特性（attribute）获取元素	*[title]	//*[@title]
<p>的第一个子元素	p >*:first-child	//p/*[0]
所有拥有子元素的<p>	不支持	//p[a]
下一个元素	p + *	//p/下一兄弟元素::*[0]

如果前面这些表达式激发了你的兴趣，推荐你浏览 XPath 的两个规范（尽管现代浏览器通常只完全支持 XPath 1.0）以初步了解它们是如何工作的：

- XPath 1.0：http://www.w3.org/TR/xpath/。
- XPath 2.0：http://www.w3.org/TR/xpath20/。

如果需要深入这个主题，我推荐你阅读 O'Reilly 出版的由 Elliotte Harold 和 Scott Means 所著的 *XML in a Nutshell*（2004），或者 Apress 出版的由 Jeni Tennison 所著的 *Beginning XSLT 2.0: From Novice to Professional*（2005）。此外，还有一些精彩的教程帮助你学习使用 XPath：

- W3Schools的XPath教程：http://w3schools.com/xpath/。
- ZVON XPath 教程：http://zvon.org/xxl/XPathTutorial/General/examples.html。

目前，浏览器对 XPath 的支持各不相同：IE 和 Mozilla 都比较完整（虽然不一样）地支持 XPath 实现，而 Safari 和 Opera 的版本还在开发中。作为补救，有几种完全使用 JavaScript 写的 XPath 实现方法。虽然它们通常会比较慢（跟基于浏览器的实现相比），但可以确保所有现代浏览器都能稳定运行：

- XML for Script：http://xmljs.sf.net/。
- Google AJAXSLT：http://goog-ajaxslt.sf.net/。

95

此外，一个名为 Sarissa 的项目（http://sarissa.sf.net/）打算为各种浏览建立一个通用的实现包装。它可以让你一次写成 XML 访问代码，同时仍能从支持的浏览器中获得更快的运行速度。这项技术的最大问题在于，Opera 和 Safari 浏览器对 XPath 支持仍不足，走不出 XPath 先前实现的困境。

使用浏览器内置的 XPath 与使用得到广泛支持的纯粹的 JavaScript 解决方案比起来，通常被认为是试验性质的技术。尽管如此，XPath 的使用和普及渐渐成长起来，可把它当作 CSS 选择器强有力的竞争对手。

现在你已经掌握了定位任意 DOM 元素或任意 DOM 元素集合的必要知识和工具，那么我们现在应该利用这些知识了：从特性的操作到增加和删除 DOM 元素。

5.5 获取元素的内容

所有的 DOM 元素无外乎包含以下三者之一：文本、元素、文本与元素的混合。一般来说，常见的是第一种和最后一种。在这一节中你将了解现有的获取元素内容的常用方法。

5.5.1 获取元素内的文本

获取元素内的文本可能会令 DOM 新手最容易迷惑，同时它也配合了 HTML DOM 文档和 XML DOM 文档。在图 5-3 所示的 DOM 结构例子中，有一个<p>主元素，它包含一个元素和一段文本块。元素本身同时包含文本块。

96

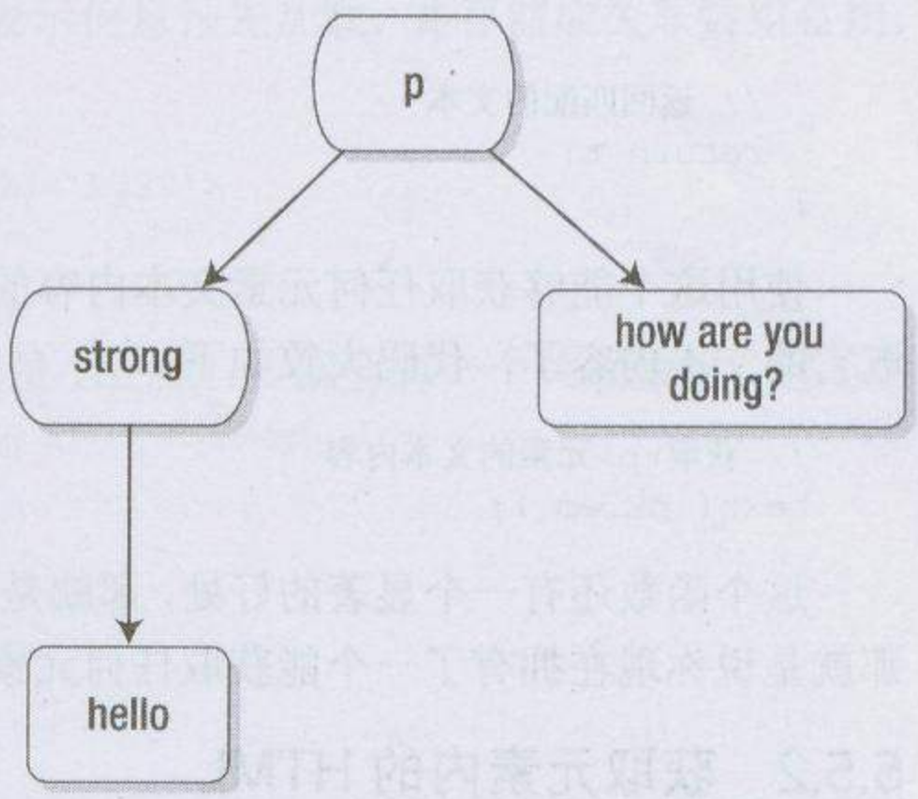


图5-3 包含元素与文本的DOM结构例子

让我们来看看如何获取这些元素中的文本。元素很简单，因为它除了文本节点没别的干扰信息。

应该注意的是，所有非基于 Mozilla 的浏览器中都有一个叫作 innerText 的属性，可用它来获取元素内的文本。这十分方便。不幸的是，因为它在一个比较流行的浏览器上并不可用，同时在 XML DOM 文档中也不行，你还是需要开发一个可行的替代方案。

获取元素内文本内容的一个窍门是，需要记住

元素并不是直接包含文本的，而包含在一个子文本节点中，这可能有点让人不习惯。假设变量strongElem包换一个元素的引用，让我们来看看代码清单5-15是如何使用DOM从元素内抽出文本的。

代码清单5-15　获取元素的文本内容

```
// 非Mozilla浏览器:
strongElem.innerTex

// 其他的浏览器:
strongElem.firstChild.nodeValue
```

了解如何从单个元素中获取文本内容后，你需要继续学习如何从<p>元素中获取经过组合的文本内容。为此，你或许需要开发一个通用的函数来获取任意元素的文本内容，而不用考虑元素究竟嵌套了几层，如代码清单5-16所示。调用text()将返回元素及其所有后代元素包含的文本内容。

代码清单5-16　一个获取元素文本内容的通用函数

```
function text(e) {
  var t = "";

  // 如果传入的是元素，则继续遍历其子元素，
  // 否则假定它是一个数组
  e = e.childNodes || e;

  // 遍历所有字节点
  for ( var j = 0; j < e.length; j++ ) {
    // 如果不是元素，追加其文本值
    // 否则，递归遍历所有元素的子节点
    t += e[j].nodeType != 1 ?
      e[j].nodeValue : text(e[j].childNodes);
  }

  // 返回匹配的文本
  return t;
}
```

97

使用这个能够获取任何元素文本内容的函数，在上面这个例子中，你就可以从<p>元素中获取它的文本内容了，代码大致如下：

```
// 获取<p>元素的文本内容
text( pElem );
```

这个函数还有一个显著的好处，那就是它可以确保在HTML和XML DOM文档中都能工作，那就是说你现在拥有了一个能获取任何元素文本内容的兼容方法。

5.5.2　获取元素内的HTML

与获取元素内的文本比起来，获取元素内的HTML是一项更容易上手的DOM任务。令人高兴的是，由于一个由IE小组开发的特色方法，所有的现代浏览器现在都实现了HTML DOM元

素的一个额外属性：innerHTML。使用这个属性你就可以从一个元素中提取所有 HTML 和文本了。此外，使用 innerHTML 十分快——通常会比递归查找元素内的所有文本内容快得多。但是并非事事称心如意。它取决于浏览器如何去实现 innerHTML 属性，而且因为并没有标准，每种浏览器都可以返回它自以为是的内容。例如，以下是一些使用 innerHTML 属性的怪异 bug：

- 基于Mozilla的浏览器在innerHTML声明中并不会返回<style>元素。
- IE返回的元素字符都是大写的，如果你想保持一致性可能会感到失望。
- innerHTML作为一个只能用在HTML DOM文档的元素中的属性，若在XML DOM文档中使用的话只会返回null值。

使用 innerHTML 属性是很直观的，存取该属性会返回一个包含该元素的 HTML 内容的字符串。如果元素仅包含文本而不包含任何子元素，则返回的字符串只包含文本。我们通过调试图 5-2 中两个元素来看它是如何工作的：

```
// 获取<strong>元素的innerHTML
// 应返回"Hello"
strongElem.innerHTML

// 获取<p>元素的innerHTML
// 应返回"<strong>Hello</strong> how are you doing?"
pElem.innerHTML
```

如果你确定元素只包含文本，这个方法也可用来取代前面那个获取元素文本的复杂方法，并且十分简便。另一方面，能获取元素内的 HTML 内容，你就可以利用就地编辑的特点来构建一些动态的时髦应用。在第 10 章我们会讨论这个主题。

98

5.6 操作元素特性

几乎跟获取元素内容一样，获取和设置元素特性（attribute）的值也是最频繁的操作之一。通常，元素自带的特性列表会随着元素自身的 XML 表示信息预先加载，并存储成关联数组备用，例如以下这个网页的 HTML 片段：

```
<form name="myForm" action="/test.cgi" method="POST">
...
</form>
```

一旦加载到 DOM 中，表示 HTML 表单元素的变量 formElem 就会有一个关联数组，它是名/值特性对。其结果大致如下：

```
formElem.attributes = {
  name: "myForm",
  action: "/test.cgi",
  method: "POST"
};
```

检查元素特性是否存在对于使用特性数组来说是最简单不过的，但还是存在一个问题：不知何故 Safari 不支持。最为严重的是，IE 也不支持可能是最有用的 hasAttribute 函数。那么如何

才能检查某个特性是否存在呢？一个可行的办法是使用 getAttribute 函数（我将会在下一节讲述）并且检查返回值是否为 null，如代码清单 5-17 所示。

代码清单5-17　检查元素是否有用一个指定的特性

```
function hasAttribute( elem, name ) {
  return elem.getAttribute(name) != null;
}
```

有了这个函数，并了解特性如何使用，那么你就可以获取和设置特性的值了。

获取和设置特性的值

根据你所使用的 DOM 文档的类型，从元素中获取特性的信息有两种方法。如果你希望保险并且总是兼容通用 XML DOM 的方式，那么可用 getAttribute 和 setAttribute。可以使用如下方法：

```
// 获取特性
id("everywhere").getAttribute("id")

// 设置特性的值
tag("input")[0].setAttribute("value","Your Name");
```

99

除了这对标准的 getAttribute/setAttribute 外，HTML DOM 文档还有作为快速特性获取器（getter）/设置器（setter）的额外属性集合。它们通常在现代浏览器的 DOM 实现中都可用（但只能给 HTML DOM 文档打包票），使用它们十分有助于编写精炼的代码。以下的代码向你展示了如何使用 DOM 属性（property）同时取得和设置 DOM 特性：

```
// 快捷获取特性
tag("input")[0].value

// 快捷设置特性
tag("div")[0].id = "main";
```

你应该注意到，特性存在某些怪异的情形。最常碰到问题的情形之一是存取类名称特性。为了能在所有的浏览器中都生效，你必须使用 elem.className 来存取 className 特性，而不是使用看起来更合适的 getAttribute("class")。这种情形也发生在 for 特性上，而它被另外起名为 htmlFor。此外，这也会在一些 CSS 特性上发生：cssFloat 和 cssText。导致这些特别的命名约定是因为 class、for、float 和 text 等都是 JavaScript 的保留字。

为了补救这些怪异的情形并且简化获取、设置正确特性的处理流程，你应该使用一个辅助函数来处理这些特殊情形。代码清单 5-18 展示了一个获取和设置元素特性值的函数。使用两个参数来调用这个函数时，比如 attr(element,id)，返回指定元素特性的值；而使用 3 个参数来调用的话，比如 attr(element,class,test)，则是设置特性的值并返回它的新值。

代码清单5-18　获取和设置元素特性的值

```
function attr(elem, name, value) {
```

```
// 确保提供的 name 是正确的
if ( !name || name.constructor != String ) return '';

// 检查name是否处在怪异命名的情形中
name = { 'for': 'htmlFor', 'class': 'className' }[name] || name;

// 如果用户传入了value参数的话，那么
if ( typeof value != 'undefined' ) {
  // 首先使用快捷方式
  elem[name] = value;

  // 可以的话，使用setAttribute
  if ( elem.setAttribute )
    elem.setAttribute(name,value);
}
// 返回特性的值
return elem[name] || elem.getAttribute(name) || '';
}
```

100

不用关心具体实现，使用标准的方法来同时存取和改变特性是一个强大的工具。代码清单 5-19 展示了一些例子，从中你可以学习到如何在多种场合中使用 attr 函数来简化处理特性的流程。

代码清单5-19 在DOM元素中使用attr函数设置和获取特性的值

```
// 设置<h1>元素的class
attr( tag("h1")[0], "class", "header" );
// 设置各个<input>元素的值
var input = tag("input");
for ( var i = 0; i < input.length; i++ ) {
  attr( input[i], "value", "" );
}
// 为name的值是'invalid'的<input>元素增加边框
var input = tag("input");
for ( var i = 0; i < input.length; i++ ) {
  if ( attr( input[i], "name" ) == 'invalid' ) {
    input[i].style.border = "2px solid red";
  }
}
```

到目前为止，我们已经讨论了 DOM 中常用的特性（比如 id、class、name 等）的获取和设置。但是，还有一个派得上用场的技术，那就是设置和获取非传统的特性。比如，你可以增添一个新特性（它只对存取 DOM 版本的元素可见）然后再次获取，完全不用修改文档的实际属性。举个例子，假设你有一个术语的自定义列表，并需要在点击术语的时候展开它的描述，HTML 的结构大致如代码清单 5-20 所示。

代码清单5-20 自定义列表的HTML，其中把描述事先隐藏起来

```
<html>
  <head>
    <title>Expandable Definition List</title>
    <style>dd { display: none; }</style>
  </head>
```

```
    <body>
      <h1>Expandable Definition List</h1>
      <dl>
        <dt>Cats</dt>
        <dd>A furry, friendly, creature.</dd>
        <dt>Dog</dt>
        <dd>Like to play and run around.</dd>
        <dt>Mice</dt>
        <dd>Cats like to eat them.</dd>
      </dl>
    </body>
</html>
```

101

我们将会在第6章详细讲解事件的细节，现在先尽量保持这些事件代码的简单。以下例子是一个能让你点击术语显示（或隐藏）描述的高效脚本。该脚本应在页面的头部或者从一个外部文件中引入。代码清单5-21展示了构建一个可扩展自定义列表的必要代码。

代码清单5-21 允许动态切换的自定义列表

```
// 直至DOM可用
domReady(function(){

    // 遍历所有的术语
    var dt = tag("dt");
    for ( var i = 0; i < dt.length; i++ ) {

    // 等待用户点击术语
    addEvent( dt[i], "click", function() {

      // 检查描述已经open与否
      var open = attr( this, "open" );

      // 切换描述的display
      next( this ).style.display = open ? 'none' : 'block';

      // 记录描述是否open
      attr( this, "open", open ? '' : 'yes' );
      });
    }
});
```

现在已经知道如何遍历DOM与如何检查及修改特性，接下来你需要学习如何创建新的DOM元素，插入到所需的地方，并删除不再需要的元素。

102

5.7 修改DOM

了解如何修改DOM后，你就可以干任何事了，从即时创建自定义的XML文档到建立接受用户键入的动态表单，几乎所有可能的操作。修改DOM有3个步骤：首先要知道怎么创建一个新元素，然后要知道如何把它插入DOM中，最后要学习如何删除它。

5.7.1 使用 DOM 创建节点

修改 DOM 的主要方法是使用 createElement 函数，它可以让你即刻创建新元素。但是，已经创建的新元素并不会马上插入到 DOM 中（刚开始使用 DOM 的人常会对此感到迷惑）。所以，我先把重点放在创建 DOM 元素上。

createElement 方法带有元素的标记名称，并返回该元素的实际 DOM 引用，这里没有特性和样式。如果你要开发使用 XSLT 驱动的 XHTML 页面（或者使用正确的 MIME 侍服的 XHTML 页面）的应用程序，必须记住，你使用的是 XML 文档，所以创建的元素必须使用正确的 XML 命名空间来关联它们。为无缝地解决问题，你可以使用一个简单的函数，用它来测试你正使用的 HTML DOM 文档是否支持使用命名空间（XHTML DOM 文档的一个特点）来创建新的元素。在这种情况下，你必须使用正确的 XHTML 命名空间来创建新的 DOM 元素，如代码清单 5-22 所示。

代码清单5-22 创建新DOM元素的通用函数

```
function create( elem ) {
  return document.createElementNS ?
    document.createElementNS( 'http://www.w3.org/1999/xhtml', elem ) :
    document.createElement( elem );
}
```

例如，使用这个函数你就可以创建一个简单的 元素，并附上一些额外的信息：

```
var div = create("div");
div.className = "items";
div.id = "all";
```

此外需要注意的是，一个创建新文本节点的 DOM 方法叫做 createTextNode。它需要传入一个参数，即你需要插入到节点中的文本，并返回已创建的文本节点。

使用新建立的 DOM 元素和文本节点，就可以把它们插入到 DOM 文档所需的位置中去了。 103

5.7.2 插入到 DOM 中

即使是经验丰富的 DOM 老手，有时也难免会感到插入元素到 DOM 中非常麻烦。在你的 JavaScript 工具库中准备以下两个函数可以让你把事情做好。

第一个函数是 insertBefore，可以在另一个子元素前插入一个元素。使用该函数的方法类似如下例子：

```
parentOfBeforeNode.insertBefore( nodeToInsert, beforeNode );
```

我使用了一个助记诀窍辅助记忆它的参数顺序：“insert 第一个元素，before 第二个。”没错吧，你只需片刻就记住了它。

有了一个能在其他节点前插入节点（可以是元素，也可以是文本节点），你是否该考虑考虑：“如何插入一个父节点中最后一个子节点？”一个名为 appendChild 的函数可以做到这个。appendChild 调用一个元素参数，追加指定的节点到子节点列表中的最后一个。这个函数的用法大致如下：

```
parentElem.appendChild( nodeToInsert );
```

为避免死记硬背 insertBefore 和 appendChild 的参数顺序，你可以使用我写的两个辅助函数解决问题，如代码清单 5-23 和代码清单 5-24 所示，参数以相关的元素/节点然后是需插入的元素/节点的顺序调用。此外，before 函数还有一个传入父节点的可选项，它可以为你节约一些代码。最后，两个函数都允许你传入需要插入/追加的字符串并自动地帮你转化为文本节点。建议你提供父元素作为引用（防止 elem 变成 null）。

代码清单5-23　在另一个元素之前插入元素的函数

```
function before( parent, before, elem ) {
  // 检查parent是否传入
  if ( elem == null ) {
    elem = before;
    before = parent;
    parent = before.parentNode;
  }
  parent.insertBefore( checkElem( elem ), before );
}
```

代码清单5-24　为另一个元素追加一个子元素的函数

```
function append( parent, elem ) {
  parent.appendChild( checkElem( elem ) );
}
```

104　代码清单 5-25 的辅助函数允许你插入元素和文本（自动转化为正确的文本节点）。

代码清单5-25　before和append的辅助函数

```
function checkElem( elem ) {
  // 如果只提供字符串,则把它转化成文本节点
  return elem && elem.constructor == String ? document.createTextNode( elem ) : elem;
}
```

现在，使用 before 和 append 函数，通过创建新的 DOM 元素，你就可以在 DOM 中增添更多的信息呈现给用户了，如代码清单 5-26 所示。

代码清单5-26　使用append和before函数

```
// 创建一个新的<li>元素
var li = create("li");
attr( li, "class", "new" );

// 创建文本内容并添加到<li>中
append( li, "Thanks for visiting!" );

// 把<li>插入到有序列表的顶端
before( first( tag("ol")[0] ), li );

// 运行这些语句会转化空<ol>
<ol></ol>
```

```
// 为以下：
<ol>
    <li class='new'>Thanks for visiting!</li>
</ol>
```

一将这些信息“插入”到 DOM（包括使用 insertBefore 和 appendChild），它马上就会渲染并呈现在用户前。因此，你可用之作实时的反馈。这对于需要用户输入的交互应用尤其有用。

现在你看到的只是基于 DOM 的方法来创建和插入节点，进一步了解其他向 DOM 注入内容的方法将更有用。

5.7.3 注入 HTML 到 DOM

比创建普通 DOM 元素并插入到 DOM 中更为流行的技术是，直接向文档注入 HTML。这个最为简便的实现方法使用了前面讨论的 innerHTML。除了从元素获取 HTML 之外，它同时也是设置元素内的 HTML 的一条途径。作为一个展示其简单性的例子，假设有一个空的<ol>元素并需要往里加入<li>，那么的代码大致如下：

105

```
// 给ol 元素加入部分li
tag("ol")[0].innerHTML = "<li>Cats.</li><li>Dogs.</li><li>Mice.</li>";
```

难道这不比繁复地创建大量的 DOM 元素和相应的文本节点要简单得多？更好的是，它还比使用 DOM 方法要快得多（根据 http://quirksmode.org 的说法）。但世事无完美，使用 innerHTML 注入方法也伴随着一些棘手的问题：

- 如前所述，XML DOM文档中并不存在innerHTML方法，就是说你得继续使用通用的DOM创建方法。
- 使用客户端XSLT创建的XHTML文档也不存在innerHTML方法，因为它们也是纯粹的XML文档。
- innerHTML完全删除了元素内容的任何节点，意味着没有像DOM方法一样追加或者插入的简便方法。

最后一条尤为棘手，因为在另一个元素之前插入元素或者追加到子元素列表末尾特别有用。但是，编织一些 DOM 魔法，去改造 append 和 before 方法，除了常规的 DOM 元素，还可以让它跟常规的 HTML 字符串和平共处。改造过程分两步走。第一步建立一个 checkElem 函数，它可以处理 HTML 字符串、DOM 元素以及 DOM 元素数组，如代码清单 5-27 所示。

代码清单5-27 转化一个DOM节点/HTML字符串混合型参数数组为纯粹的DOM节点数组

```
function checkElem(a) {
  var r = [];
  // 如果参数不是数组，则强行转换
  if ( a.constructor != Array ) a = [ a ];

  for ( var i = 0; i < a.length; i++ ) {
    // 如果是字符串
    if ( a[i].constructor == String ) {
      // 用一个临时元素来存放HTML
```

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

      // 注入HTML，转换成DOM结构
      div.innerHTML = a[i];
      // 提取DOM结构到临时 div 中
      for ( var j = 0; j < div.childNodes.length; j++ )
        r[r.length] = div.childNodes[j];
    } else if ( a[i].length ) { // If it's an array
      // 假定是DOM节点数组
      for ( var j = 0; j < a[i].length; j++ )
        r[r.length] = a[i][j];
    } else { // 否则，假定是DOM节点
      r[r.length] = a[i];
    }
  }
  return r;
}
```

106

第二步，你需要修改这两个插入函数以适应新的 checkElem，接受数组元素，如代码清单5-28所示。

代码清单5-28　插入和追加内容到DOM的改进函数

```
function before( parent, before, elem ) {
  // 检查是否提供 parent 节点参数
  if ( elem == null ) {
    elem = before;
    before = parent;
    parent = before.parentNode;
  }

  // 获取元素的新数组
  var elems = checkElem( elem );

  // 向后遍历数组，
  // 因为我们向前插入元素
  for ( var i = elems.length - 1; i >= 0; i-- ) {
    parent.insertBefore( elems[i], before );
  }
}

function append( parent, elem ) {
  // 获取元素数组
  var elems = checkElem( elem );

  // 把它们所有都追加到元素中
  for ( var i = 0; i <= elems.length; i++ ) {
    parent.appendChild( elems[i] );
  }
}
```

107

现在，使用这些新函数为有序列表追加<li>就非常简单了：

```
append( tag("ol")[0], "<li>Mouse trap.</li>" );
```

```
// 运行该简单的一行就能追加HTML到这个<ol>中
<ol>
<li>Cats.</li>
<li>Dogs.</li>
<li>Mice.</li>
</ol>

// 把它变成了这样：
<ol>
<li>Cats.</li>
<li>Dogs.</li>
<li>Mice.</li>
<li>Mouse trap.</li>
</ol>

// 而用before()运行一个类似的语句
before( last( tag("ol")[0] ), "<li>Zebra.</li>" );

// 则会变成这样：
<ol>
<li>Cats.</li>
<li>Dogs.</li>
<li>Zebra.</li>
<li>Mice.</li>
</ol>
```

这将能帮助你开发简洁和清晰的代码。但是，如果从 DOM 中移动元素和删除节点要怎么办呢？当然，肯定也会有其他方法处理这些问题。

5.7.4 删除 DOM 节点

删除 DOM 节点的操作几乎与创建和插入一样频繁。根据用户的需求允许创建无限量的项目，那么允许用户能够删除它们之中再也不需要处理的部分也就变得很重要了。删除节点的能力可封装成一个函数：removeChild。它跟 appendChild 用法一致效果相反。该函数的用法大致如下：

```
NodeParent.removeChild( NodeToRemove );
```

记住这点，你就可以创建两个独立的函数来删除节点了，如代码清单 5-29 所示。 108

代码清单5-29 删除DOM节点的函数

```
// 删除一个独立的DOM节点
function remove( elem ) {
  if ( elem ) elem.parentNode.removeChild( elem );
}
```

代码清单 5-30 展示了从一个元素中删除所有子节点的函数，仅需要 DOM 元素的引用作为参数。

代码清单5-30　从一个元素中删除所有子节点的函数

```
// 在DOM中删除一个元素的所有子节点
function empty( elem ) {
  while ( elem.firstChild )
    remove( elem.firstChild );
}
```

举个例子，要删除上一节例子中添加的<li>，同时假设你已经给用户充足的时间来浏览<li>了，在没有提示的情况下被删除。可以使用以下的 JavaScirpt 代码来完成这个操作：

```
// 删除<ol>的最后一个<li>
remove( last( tag("ol")[0] ) )

// 将会把
<ol>
<li>Learn Javascript.</li>
<li>???</li>
<li>Profit!</li>
</ol>

// 转换成：
<ol>
<li>Learn Javascript.</li>
<li>???</li>
</ol>

// 如果要求执行empty()函数而不是remove()
empty( last( tag("ol")[0] ) )

// 它将会简单地清空<ol>，只留下：
<ol></ol>
```

学习完删除 DOM 节点，你已经清楚了我们的文档对象模型是如何运作的，并学会如何充分运用它。

109

5.8　小结

在这一章里讨论了与文档对象模型相关的大量东西。不幸的是，其中有些主题比其他的要复杂得多（比如等待 DOM 加载），而且在一段时间里都一直会这么复杂。不过仅使用目前所学你也能搭建出任何动态的 Web 应用程序了。

如果想看看一些 DOM 脚本编程的实际运用，在附录 A 中包含了大量补充代码。此外，还可以从本书的网站 http://jspro.org 或 Apress 网站 http://www.apress.com 的源代码/下载部分找到更多 DOM 脚本编程例子。接下来，你要把注意力转向分离式 DOM 脚本编程的下一个组成部分——事件。

110

第6章

事　件

分离式 DOM 脚本编程最重要的一步是动态绑定事件。编写可用性 JavaScript 代码的终极目标是，不管用户使用的是哪种浏览器、哪种平台，都能够让网页正常运作。为此，你得制定一个特性指标，并排除任何不支持这些特性的浏览器。对于不支持的浏览器，你可以给它们一个交互更少的网页。而这种编写 JavaScript 和 HTML 交互的方式包含更整洁的代码、更具亲和力的网页和更佳的用户交互。使用 DOM 事件改进 Web 应用程序的交互，可以达成这些目标。

近年来，JavaScript 事件的概念得到发展，达到了现在的可靠的准稳定状态。幸运的是，利用事件已有的相似性，可以开发一些杰出的工具来辅助建立强大的、编码清晰的 Web 应用程序。

在这一章中，我们将从 JavaScript 事件如何运作、与其他语言事件模型的对比开始，然后再看看事件模型所提供的信息并如何更好地利用它们。学习完为 DOM 绑定事件和现有的不同类型的事件后，我将展示如何为页面集成一些有效的分离式脚本编程技术，并以此作为总结。

6.1　JavaScript 事件简介

如果已经看过某些应用程序的 JavaScript 代码的核心部分，你会发现事件是把所有东西粘在一起的胶水。在一个设计良好的 JavaScript 应用程序中，它包含有数据源和表现层（在 HTML DOM 内）。要使这两方面协调一致，你必须监听用户的交互并因此来更新页面。DOM 和 JavaScript 事件的结合是现代 Web 应用程序的根基。

6.1.1　异步事件与线程

JavaScript 事件模型是相当独特的。它不使用任何线程，完全是异步的。也就是说，程序中代码的运行由于其他动作，比如用户点击或者页面加载才会触发。

线程程序设计和异步程序设计的根本区别在于如何等待事件的发生。在线程程序中你可能需要不断检查条件是否满足。而在异步程序中，你只需简单地使用事件处理函数注册一个回调函数就行了，一旦该事件触发，该处理函数就会运行回调函数。现在让我们看看如果需要线程的话，JavaScript 程序应该如何编写，不需要线程的程序又是如何使用异步回调。

1. JavaScript线程

JavaScript 目前并不存在线程。最接近线程的也只是使用 `setTimeout()` 回调函数，但并不理

想。如果 JavaScript 是一门传统的线程编程语言，那么类似于代码清单 6-1 中的代码就能够被执行。它是一个直到页面完全载入才执行的部分代码的模拟。如果 JavaScript 是一门线程编程语言，你或许可以做一些类似于此模拟的事情。幸好，事情并不如此。

代码清单6-1　模拟线程的JavaScript代码

```
// 注意：该代码不会生效！
// 直到页面完全载入，一直检查
while ( ! window.loaded() ) { }

// 页面加载完毕，就可以执行了
document.getElementById("body").style.border = "1px solid #000";
```

或许你已注意到，在代码中有一个持续检查 window.loaded()是否返回 true 的循环。不管 window 对象并不存在 loaded()函数，就算使用一个类似的循环也不会在 JavaScript 中生效。这是因为 JavaScript 中的循环都被阻塞了(即如果运行没完成则不会触发任何东西)。如果 JavaScript 能处理线程，你可能会看到类似图 6-1 所示的情况，代码中的 while 循环不断检查 window 是否加载完毕。但是它并不会工作，因为所有的循环都被阻塞了（循环进行的同时阻塞了其他操作的运行)。

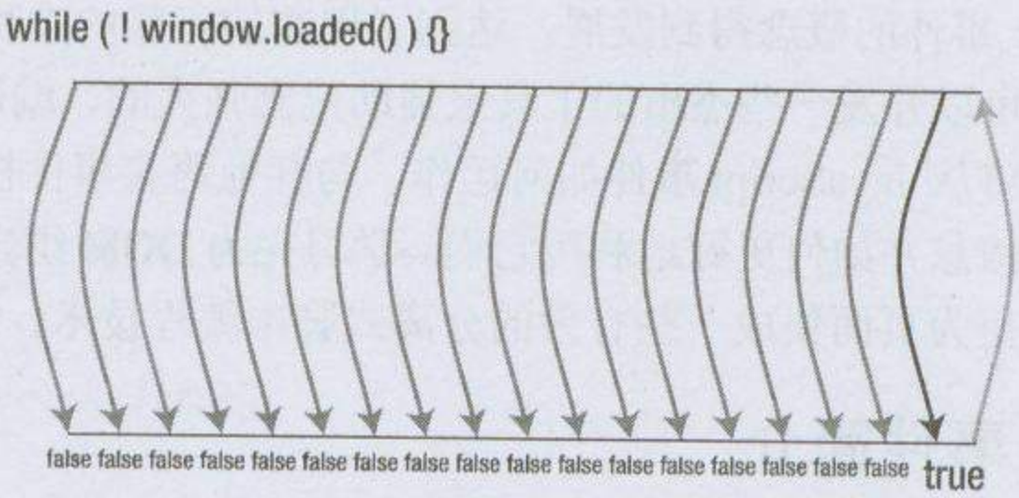


图6-1　假定JavaScript可以处理线程的情况示意图

实际上，因为 while 循环的持续运行并阻塞应用程序的其他普通流程，它永远无法达到 true 值。结果只能是用户的浏览器暂停、延迟甚至崩溃。从中你可获得的经验教训是，如果有人宣称使用 while 循环来等待动作的发生（在 JavaScript 中)，他不是在说谎就是误解甚深。

2. 异步回调

使用线程不断检查更新状态，实际可行的替代方案是使用异步回调，而它也正是 JavaScript 所使用的。使用直白的术语说，一个 DOM 元素触发某个特定事件的时候，你可以指派一个回调函数来处理它。也就是说，你可以为需要执行的代码提供一个引用，并且让浏览器处理一切细节。代码清单 6-2 展示了一段使用事件处理函数和回调函数的代码。可以看到 JavaScript 中的代码需要封装进函数并绑定到事件处理函数（window.onload）上。window.onload()在页面载入完毕后即可被调用。其他常用的事件比如点击事件、鼠标悬停事件和提交事件等也是如此。

代码清单6-2　JavaScript异步回调

```
// 注册一个函数，当页面载入完毕调用
```

```
window.onload = loaded;

// 页面载入完毕后调用的函数
function loaded() {
      // 页面载入完毕，开始干活了
      document.getElementById("body").style.border = "1px solid #000";
}
```

对比一下代码清单 6-2 和代码清单 6-1，可以发现两者明显的差别。正确运行的代码应该是绑定到事件监听函数（onload 属性）上的事件处理函数（loaded 函数）。浏览器在页面加载完毕后调用 window.onload 关联的函数并执行。JavaScript 代码的执行流程如图 6-2 所示，演示了 JavaScript 使用回调函数等待页面加载的表现。因为实际上并不可能实时等待，你只能注册一个回调函数（loaded）到处理函数（window.onload）上，它会在页面加载完毕后调用。

简单的事件监听函数和处理函数并不明显的一点是，取决于事件类型和元素在 DOM 中的位置，事件的执行顺序和处理会有很大的不同。我们将在下一节探索事件的两种不同阶段（phase）及其区别。

113

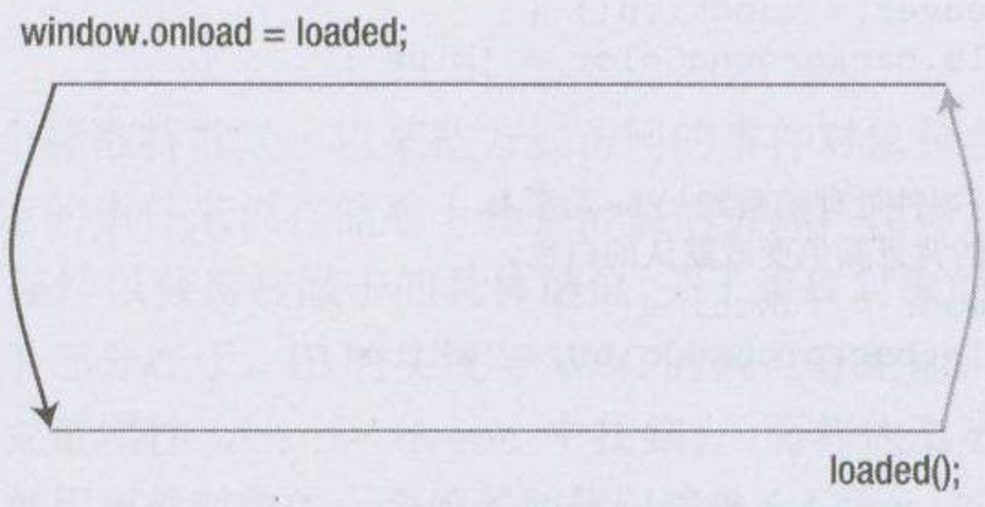


图6-2 使用回调等待页面加载的表现

6.1.2 事件阶段

JavaScript 事件在两个阶段中执行：捕获和冒泡。一旦元素触发了事件（比如，用户点击链接导致 click 事件的发生），那么允许处理事件元素和顺序都有很大的不同。你可以看到图 6-3 例子的执行顺序，演示了用户点击页面第一个<a>元素的时候，事件处理函数的触发顺序。

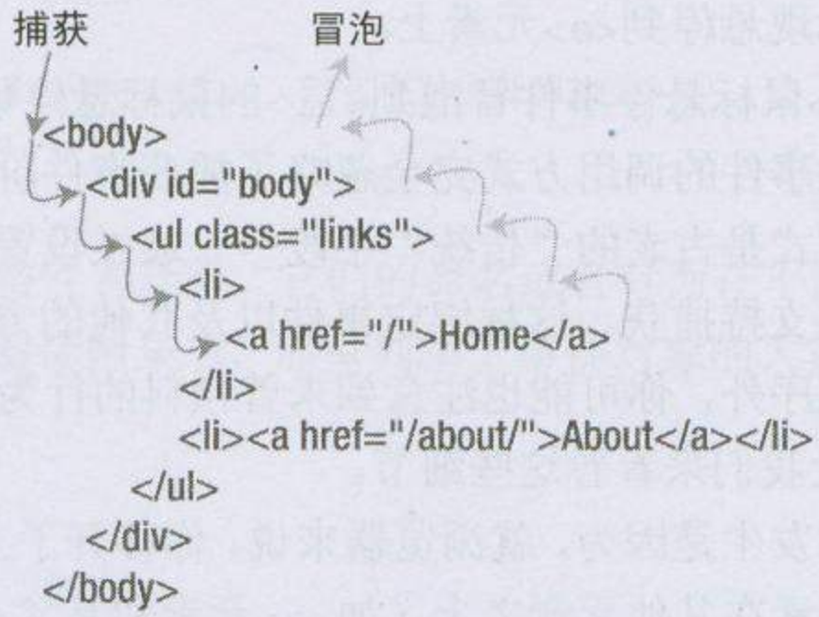


图6-3 事件处理的两个阶段

观察这个链接点击（见图 6-3）的简单例子，就能看到事件的执行顺序。假如用户点击<a>元素，document 的点击处理函数会首先发生，然后是<body>的处理函数，再次是 的处理
114 函数，等等，一直进行到<a>元素。这称为捕获阶段。一旦完成，它就重回树中，<li>、<ul>、 、<body>和 document 事件处理函数全部按顺序触发。

事件以这种方式处理有其特定的理由，而且工作良好。让我们来看一个简单例子。假设需要在用户鼠标悬停到每个<li>元素时改变背景颜色，并在鼠标移开时恢复默认，这也是大部分菜单都需实现的功能。代码清单 6-3 完全能满足这个需求。

代码清单6-3　页签导航的鼠标悬停效果场景

```
// 查找所有的<li>元素，并为它们绑定事件处理函数
var li = document.getElementsByTagName("li");
for ( var i = 0; i < li.length; i++ ) {

    // 绑定鼠标悬停事件处理函数到<li>元素上，
    // 它会把<li>的背景颜色变成蓝色。
    li[i].onmouseover = function() {
        this.style.backgroundColor = 'blue';
    };

    // 绑定鼠标离开事件处理函数到<li>元素上，
    // 它会把<li>的背景颜色变成默认的白色。
    li[i].onmouseout = function() {
        this.style.backgroundColor = 'white';
    };

}
```

这些代码会完全按照你的预期运作：你把鼠标悬停在<li>元素上其背景颜色改变，移开鼠标，则颜色恢复。但是，你未必会注意到的是，每次鼠标悬停在<li>上时你都在切换两个不同的元素。因为<li>元素也包含了<a>元素，鼠标悬停上的元素不仅仅是<li>而已。让我们来看看这个事件发生的真实流程：

(1) 鼠标悬停：移动鼠标到<li>元素上。

(2) 鼠标移开：从<li>移到被包含的<a>上。

(3) <a>鼠标悬停：鼠标现悬停到<a>元素上。

(4) 鼠标悬停：<a>鼠标悬停事件冒泡到<li>的鼠标悬停事件上。

或许你已经注意到这些事件的调用方式完全忽略了捕获事件阶段，别担心，我还记得呢。因为你绑定事件监听函数的方式是古老的“传统”手段——通过设置元素的 onevent 特性，而这
115 种方式只支持事件冒泡，不支持捕获。这种绑定事件以及其他的方式将在下一节讨论。

除了事件调用的奇怪顺序外，你可能也注意到未曾预料的行为：<li>元素的鼠标移开和<a>到<li>的鼠标悬停冒泡。让我们来看看这些细节。

第一个鼠标移开事件的发生是因为，就浏览器来说，你移开了上一级<li>元素的领域并进入了另一个元素。这是因为元素在其他元素之上（如<a>元素和其父<li>元素）时会得到鼠标的短暂聚焦。

<a>鼠标悬停会冒泡到其父<li>元素最终变成了这段代码的长处。因为实际上你并未为<a>元素绑定任何的监听函数，而事件直接继续上升到 DOM 树，查找另一个被监听的元素。在它冒泡过程中第一个碰到的是<li>元素，而<li>元素正监听进入的鼠标悬停事件（也正是你真正所期望的）。

应该思考的一点是，如果你绑定一个事件监听函数到<a>元素的鼠标悬停事件上又会有什么变化呢？有停止该事件冒泡的方法吗？这是一个重要且有用的主题，接下来将会涉及。

6.2 常见事件特性

JavaScript 事件的一个重要方面是它们拥有一些相对一致的特点，可以给你的开发提供更多的强大功能。其中最简单和最成熟的内容是，事件对象提供元数据（metadata）和上下文函数集合，它们可以让你处理诸如鼠标事件和键盘敲击方面的情况。此外，还有可以修改一般事件的捕获 / 冒泡流的函数。通过学习并精通这些特点能让你的编程变得更轻松简便。

6.2.1 事件对象

事件处理函数的一个标准特性是，以某些方式访问的事件对象包含有关于当前事件的上下文信息。这个对象对于特定的事件来说存储着十分有价值的资源。比如，当处理键盘敲击时你可以访问该对象的 keyCode 属性以获得被敲击的具体键位。关于事件对象的规范细节可以参看附录B。

然而事件对象的棘手部分在于，IE 的实现与 W3C 的规范有差别。IE 使用一个独立的全局事件对象（它可以在全局变量属性 window.event 中找到），而其他浏览器则使用独立的包含事件对象的参数传递。可靠地使用事件对象的一个例子如代码清单 6-4 所示，展示了修改普通<textarea>元素不同行为。通常，用户可以在 textarea 里键入回车，导致额外的换行符。如果并不想如此且只需要一个大文本框怎么办？使用这个函数正好。

116

代码清单6-4 使用DOM事件复写原有功能

```
// 查找页面的第一个<textarea>并绑定键盘敲击的监听函数
document.getElementsByTagName("textarea")[0].onkeypress = function(e){
    // 如果不存在事件对象，则获取全局的（仅IE）的对象
    e = e || window.event;

    // 如果敲击了回车键，返回false（使它不发生任何行为）
    return e.keyCode != 13;
};
```

事件对象内还有大量的特性和函数，它们的命名或者行为在不同的浏览器中是不同的。如果想进一步了解细节，推荐你阅读附录 B，那里列举了事件对象的大量特点、如何使用它们和实践运用的例子。

6.2.2 this 关键字

this 关键字（第 2 章已有讨论）在函数的范围内，作为一种访问当前对象的方式。现代浏

览器使用 this 关键字为所有的事件处理函数提供一些上下文。照常，只有部分浏览器（和部分的方法）能正常运行并将 this 等同于当前元素，稍后将会深入讨论。在代码清单 6-5 中，可以利用这个特点，虽然只创建了一个处理点击的通用函数，但使用 this 关键字能检查当前被操作的元素。这里虽然展示了只用一个函数处理点击事件，但因为使用 this 关键字来引用元素，它将会如期运行。

代码清单6-5　当被点击的时候，改变所有<li>元素的背景和前景颜色

```
// 查找所有的<li>元素并为每个绑定点击处理函数
var li = document.getElementsByTagName("li");
for ( var i = 0; i < li.length; i++ ) {
      li[i].onclick = handleClick;
}

// 点击处理函数——被调用时改变指定元素的背景和前景颜色
function handleClick() {
      this.style.backgroundColor = "blue";
      this.style.color = "white";
}
```

this 关键字用起来不仅是便利，你会发现，运用恰当的话，它能很大程度地降低 JavaScript
117 代码的复杂性。我们尝试把本书中的所有事件相关的代码都使用 this 关键字。

6.2.3　取消事件冒泡

现在已经了解事件捕获/冒泡是如何工作了，让我们来探索如何控制它。上一个例子里值得注意的重要一点是，你是否只希望事件发生在它的目标而非它的父元素上，但却没有方法去阻止。阻止事件冒泡的流动可能会导致类似图 6-4 的情况发生，图中演示了事件被第一个<a>元素捕获后，随之而来的冒泡被取消的结果。

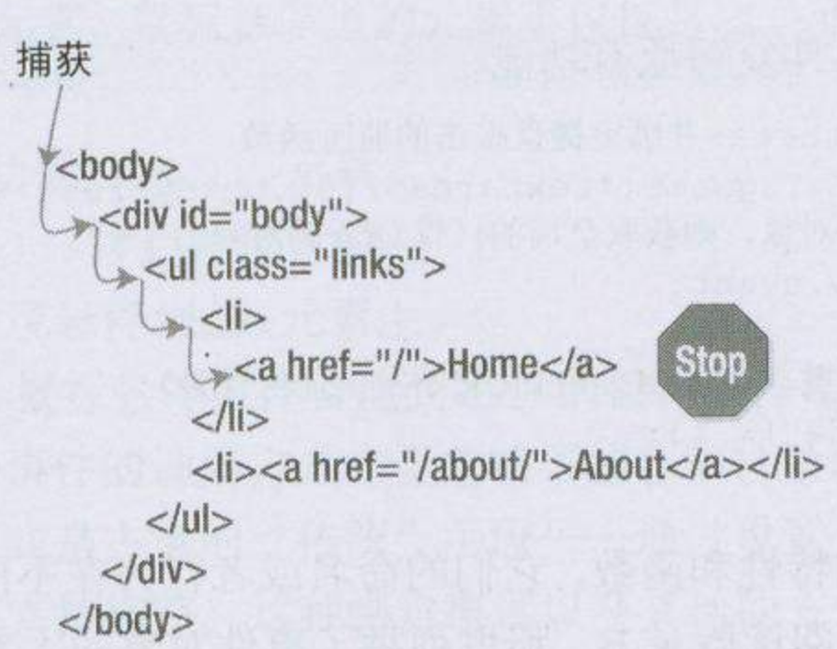


图6-4　事件被第一个<a>元素捕获后的结果

阻止事件的冒泡（或捕获）可以为复杂的应用程序提供非常大的益处。不幸的是，IE 提供了与其他浏览器不同的方式来停止事件的冒泡。取消事件冒泡的通用函数可以在代码清单 6-6 中找到。该函数带一个参数：传入事件处理函数的事件对象。该函数使用两种方式取消事件冒泡：标

准的 W3C 方式和非标准的 IE 方式。

代码清单6-6 阻止事件冒泡的通用函数

```
function stopBubble(e) {
      // 如果传入了事件对象，那么就是非IE浏览器
      if ( e && e.stopPropagation )
            // 因此它支持W3C的stopPropation()方法
            e.stopPropagation();
      else
            // 否则，我们得使用IE的方式来取消事件冒泡
            window.event.cancelBubble = true;
}
```

118

现在你可能想知道，什么时候需要阻止事件冒泡？事实上，现在绝大多数时候都可以不必在意它。但是当你开始开发动态应用程序（尤其是需要处理键盘和鼠标）的时候，就有必要了。

代码清单 6-7 展示了一个简短的片段，它会为鼠标悬停的当前元素加上红色的边框。可以通过为每一个 DOM 元素增加 mouseouver 和 mouseout 事件来实现。如果不阻止事件冒泡，每次把鼠标移到一个元素上时，该元素及其父元素都会有红色的边框，这并不如你所预期的。

代码清单6-7 使用stopBubble()来创建元素的交互集

```
// 定位，遍历所有的DOM元素
var all = document.getElementsByTagName("*");
for ( var i = 0; i < all.length; i++ ) {

      // 监听用户鼠标，当移动到元素上时
      // 为元素加上红色边框
      all[i].onmouseover = function(e) {
          this.style.border = "1px solid red";
          stopBubble( e );
      };

      // 检查用户鼠标，当移开元素时
      // 删除我们加上的边框
      all[i].onmouseout = function(e) {
          this.style.border = "0px";
          stopBubble( e );
      };

}
```

使用阻止事件冒泡的能力，你现在完全控制了元素对于事件的可见和处理。这是一个编写动态 Web 应用程序开发的必备基础工具。最后是取消浏览器的默认行为，可以让你完全重载浏览器的所作所为并实现新的功能。

6.2.4 重载浏览器的默认行为

对于大部分事件，浏览器都会发生默认行为。比如，点击<a>元素会把你带到相关的页面上，

这是浏览器本身的默认行为。无论在捕获阶段还是冒泡阶段这个行为都会发生，如图6-5所示。这个例子演示了用户点击页面的一个<a>元素后产生的结果。事件以捕获和冒泡阶段（像前面所讨论的那样）在DOM中移动为开始标志。但是，一旦事件完成移动，浏览器试图执行事件和元
119 素的默认行为。在这种情况下，它将访问其他页面。

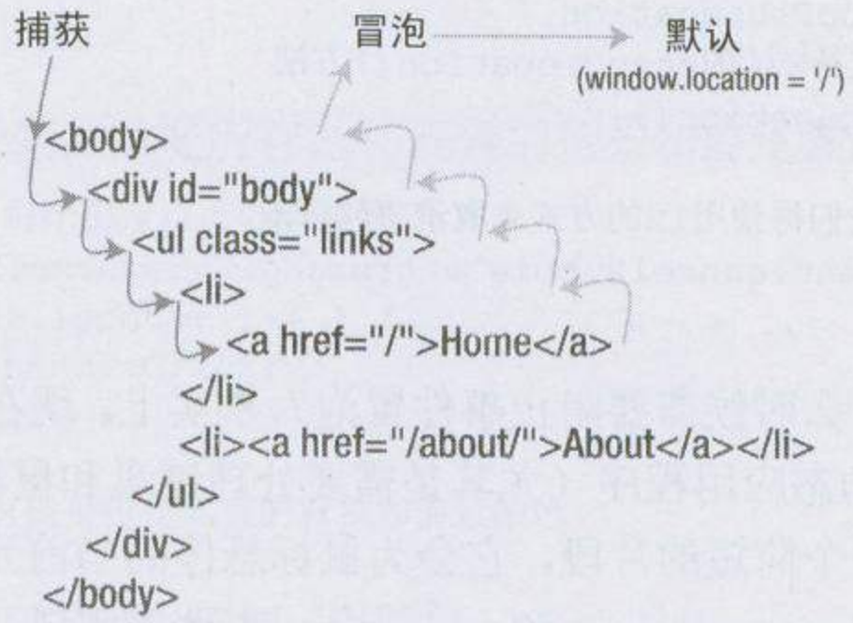


图6-5　事件的生命周期

默认行为可以归结为没有明确指令而浏览器自动执行的行为。以下是一些不同事件的不同类型默认行为的例子：

- 点击<a>元素将重定向到它的href特性上的URL。
- 使用键盘并按Ctrl+S，浏览器将保存网站的HTML文件。
- 提交HTML<form>将向指定的URL提交数据，并重定向浏览器到该地址上。
- 鼠标移动到一个有alt或title（取决于浏览器）<img>上将会出现工具条，提示<img>的描述。

就算阻止事件冒泡或者完全没有绑定事件，浏览器还是会执行所有这些行为。这会在你的脚本中引发重大问题。如果希望提交表单的表现标新立异，或者想要<a>元素表现与众不同，而非其原有的意图？因为取消事件冒泡并不能防止默认行为，你需要一些特定的代码直接处理。就如取消事件冒泡一样，有两种阻止发生默认行为的途径：IE的特定方式和W3C方式。两种方式均在代码清单6-8中展示。该函数带一个传入事件处理函数的事件对象参数，且应在时间处理函数的末尾使用，比如：return stopDefault(e);——因为处理函数同时需要返回 false（false
120 本身也从 stopDefault 返回）。

代码清单6-8　防止发生默认浏览器行为的通用函数

```
function stopDefault( e ) {
    // 防止默认浏览器行为（W3C）
    if ( e && e.preventDefault )
        e.preventDefault();

    // IE中阻止浏览器行为的捷径
    else
```

```
            window.event.returnValue = false;

        return false;
    }
```

使用 stopEvent 函数，就可以阻止浏览器的任何默认行为了。这可以让你为用户编写一些巧妙的交互，如代码清单 6-9 所示的例子。该代码使所有的链接在一个内置的<iframe>中加载，而不是重装入整个页面。这么做能让用户保持在一个页面上，因此也可能获得某种情况下更佳的交互体验。

注意 防止默认行为会在需要处理的95%的情况中有效。在不同浏览器之间转换会碰到相当棘手的问题，因为防止默认行为是由浏览器决定的（而浏览器不会总正确执行），尤其是在文本域中防止敲击和<iframe>内的行为。除此之外，其他的都应该无大碍。

代码清单6-9 使用stopDefault()重载浏览器功能

```
// 假设页面中已经存在一个iframe，它的id是'iframe'
var iframe = document.getElementById("iframe");

// 定位页面上所有的<a>元素
var a = document.getElementsByTagName("a");
for ( var i = 0; i < a.length; i++ ) {

        // 为<a>绑定点击处理函数
        a[i].onclick = function(e) {
            // 设置iframe的地址
            iframe.src = this.href;

        // 防止浏览器访问<a>所指向的网站（这是一个默认行为）
             return stopDefault( e );
        };

    }
```


重载默认事件无疑是 DOM 和事件的重要部分，它们一起构成了分离式 DOM 脚本编程。我们会在 6.5 节中更具体地从功能的角度讨论它是如何工作的。但是，它并不是十全十美的。在你需要把事件处理函数绑定到 DOM 元素上时，得预防一个由此引发的问题。绑定事件有 3 种方式，下一节将会讨论。

6.3 绑定事件监听函数

在 JavaScript 中如何为元素绑定事件处理函数已经成为一个不断发展演进的课题。最初，浏览器强迫用户在 HTML 文档内编写行内的事件处理函数代码。幸运的是，这种技术越来越不受欢迎（这是好事情，它其实会破坏分离式 DOM 脚本编程的数据抽象原则）。

Netscape 和 IE 的竞争白热化时，它们虽有很大不同，但开发的事件注册模型却非常相似。最终，Netscape 的模型经过修改成了 W3C 的标准，但 IE 的依旧不变。

现在，有 3 种可靠的事件注册方式。传统方式是绑定事件处理函数的古老的行内方式的一个分支，它十分可靠稳定。另外两种就是 IE 和 W3C 方式。最后，我将展示一系列可靠的方法，开发者可用它们来注册和删除事件，而不必担心浏览器兼容性。

6.3.1　传统绑定

在本章中我一直在使用的是绑定事件的传统方式。目前为止，它是绑定事件处理函数最简单且兼容性最好的方式。要使用这种最流行的方法，你为需要监听的 DOM 元素绑定一个函数作它的一个属性。使用传统方式绑定事件的一些例子如代码清单 6-10 所示。

代码清单6-10　使用事件绑定的传统方法绑定事件

```
// 查找第一个<form>元素并为它绑定'submit'事件处理函数
document.getElementsByTagName("form")[0].onsubmit = function(e){
        // 停止表单提交的默认行为
        return stopDefault( e );
};
// 为文档的<body>元素绑定敲击事件处理函数
document.body.onkeypress = myKeyPressHandler;

// 为页面绑定加载事件处理函数
window.onload = function(){ ... };
```

122

这个技术有很多优点和缺点，使用它们的时候请注意。

1. 传统绑定的优点

以下是使用传统方法的优点：

- 使用传统方法最大的优点在于其非常简单和稳定，可以确保它在你使用的不同浏览器中运作一致。
- 处理事件时，this关键字引用的是当前元素，这很有帮助（如代码清单6-5所示）。

2. 传统绑定的缺点

以下是使用传统方法的缺点：

- 传统方法只会在事件冒泡中运行，而非捕获和冒泡。
- 一个元素一次只能绑定一个事件处理函数。在使用常用的window.onload属性时极有可能会引发令人迷惑的结果（实际上覆盖了使用相同绑定事件方法的其他代码）。如代码清单6-11所示，你可以看到一个事件处理函数覆盖了另一个旧的事件处理函数。
- 事件对象参数仅非IE浏览器可用。

代码清单6-11　事件处理函数相互覆盖

```
// 绑定载入处理的初始化函数
window.onload = myFirstHandler;

// 在某处，你引入了另外的库，
```

```
// 初始的处理函数被覆盖
// 当页面完成载入，仅'mySecondHandler'会被调用
window.onload = mySecondHandler;
```

了解到可能会被其他事件覆盖，你应该选择只在简单的，与其他外来代码一起运作的可信任场合下使用事件绑定的传统手段。尽管如此，解决这个棘手问题还是有办法的，那就是浏览器提供的现代事件绑定方法。

123

6.3.2 DOM 绑定：W3C

为 DOM 元素绑定事件处理函数，W3C 的方法是唯一真正的标准化手段。需要注意的是，现代浏览器都支持这种方式的绑定事件，但 IE 除外。

绑定新处理函数的代码十分简单。它作为每个 DOM 元素的函数出现（名为 addEventListener）并带 3 个参数：事件的名称（比如 click）、处理事件的函数和一个启用或禁用事件捕获的布尔标记。代码清单 6-12 是使用 addEventListener 的一个例子。

代码清单6-12 使用W3C方式绑定事件处理函数的例子

```
// 查找第一个<form>元素并为其绑定一个'submit'事件处理函数
document.getElementsByTagName("form")[0].addEventListener('submit',function(e){
      // 停止表单提交的默认行为
      return stopDefault( e );
}, false);

// 为文档的<body>元素绑定敲击事件处理函数
document.body.addEventListener('keypress', myKeyPressHandler, false);

// 为页面绑定一个载入事件处理函数
window.addEventListener('load', function(){ ... }, false);
```

1. W3C绑定的优点

以下是 W3C 事件绑定方法的优点：

- 该方法同时支持事件处理的捕获和冒泡阶段。事件阶段取决于addEventListener最后的参数的设置：false（冒泡）或true（捕获）。
- 在事件处理函数内部，this关键字引用当前元素。
- 事件对象总是可以通过处理函数的第一个参数获取。
- 可以为同一个元素绑定你所希望的多个事件，同时并不会覆盖先前绑定的事件。

2. W3C绑定的缺点

以下是 W3C 事件绑定方法的缺点：

- IE不支持，你必须使用IE的attachEvent函数替代。

如果 IE 能利用 W3C 绑定事件处理函数的方法，本章篇幅将会比现在看到的要短，因为并没有讨论绑定事件的其他方法的必要了。直到现在，W3C 事件绑定方法的使用依然是最直观和最简单的。

124

6.3.3　DOM 绑定：IE

在诸多方式中，IE 的绑定事件方式跟 W3C 的最为接近。但是，当你认真处理细节时，会发现在某些明显的地方有所差异。在代码清单 6-13 中可以找到在 IE 中绑定事件处理函数的一些例子。

代码清单6-13　使用IE方式为元素绑定事件处理函数的例子

```
// 查找第一个<form>元素并为其绑定一个'submit'事件处理函数
document.getElementsByTagName("form")[0].attachEvent('onsubmit',function(){
        // 停止表单提交的默认行为
        return stopDefault();
},);

// 为文档的<body>元素绑定敲击事件处理函数
document.body.attachEvent('onkeypress', myKeyPressHandler);

// 为页面绑定一个载入事件处理函数
window.attachEvent('onload', function(){ ... });
```

1. IE绑定的优点

以下是 IE 事件绑定方法的优点：

- 可以为同一个元素绑定你所希望的多个事件，同时并不会覆盖先前绑定的事件。

2. IE绑定的缺点

以下是 IE 事件绑定方法的缺点：

- IE仅支持事件捕获的冒泡阶段。
- 事件监听函数函数内的this关键字指向了window对象，而不是当前元素（IE的一个巨大缺点）。
- 事件对象仅存在于window.event参数中。
- 事件必须以ontype的形式命名，比如，onclick而非click。
- 仅IE可用。你必须在非IE浏览器中使用W3C的addEventListener。

就半标准事件特性的现状而言，IE 的事件绑定的实现非常缺乏。因为它的许多缺点，为了让它表现合理还是得继续补救。但是，事无尽失，存在一个为 DOM 增加事件的标准方法，并在很大程度上减轻我们的痛苦。

125

6.3.4　addEvent 和 removeEvent

在 2005 年末由 Peter-Paul Koch（http://quirksmode.org）举办的一个比赛上，他要求使用 JavaScript 的程序员来开发一对新函数——addEvent 和 removeEvent，它们可以为用户提供一套增加和删除 DOM 元素事件的可靠方法。最终我依靠一段工作得够好的精炼代码赢得了比赛。但是后来裁判之一（Dean Edwards）拿出了这对函数的另一个版本，远远优于我写的。他的实现使用绑定事件处理函数的传统手段，完全无视现代的方法。由于这个原因，他的实现可以在一个更大范围内的浏览器中使用，同时还提供必要的事件细节（比如 this 关键字和标准事件对象）。代码清单 6-14 展示了一段代码，它使用了事件处理的不同方面，尽可能利用新的 addEvent 函数，

包括防止浏览器默认事件、正确事件对象的引入和正确 this 关键字的引入。

代码清单6-14 使用addEvent函数的代码片段例子

```
// 等待页面完成载入
addEvent( window, "load", function(){

        // 监听用户的任意敲击
        addEvent( document.body, "keypress", function(e){
            // 如果用户敲击 空格 + Ctrl键
            if ( e.keyCode == 32 && e.ctrlKey ) {

                // 展示我们的特定表单
                this.getElementsByTagName("form")[0].style.display = 'block';

                // 确保没有任何奇怪的事情发生
                e.preventDefault();

            }
        });

});
```

对于操作 DOM 事件，addEvent 函数提供了一个简单但很强大的方式。只要看看优缺点，就可知道它相当清晰，可作为处理事件的一个稳定可靠的方式。它的完整源代码可在代码清单6-15 中找到，它可以在所有浏览器中工作、不会内存泄露、正确处理 this 关键字与事件对象并且合乎普通事件对象函数的标准。 126

代码清单6-15 Dean Edwards所写的addEvent/removeEvent库

```
// 由 Dean Edwards所编写的addEvent/removeEvent, 2005
// 由Tino Zijdel整理
// http://dean.edwards.name/Weblog/2005/10/add-event/

function addEvent(element, type, handler) {
    // 为每一个事件处理函数赋予一个独立的ID
    if (!handler.$$guid) handler.$$guid = addEvent.guid++;

    // 为元素建立一个事件类型的散列表
    if (!element.events) element.events = {};

    // 为每对元素/事件建立一个事件处理函数的散列表
    var handlers = element.events[type];
    if (!handlers) {
        handlers = element.events[type] = {};

        // 存储已有的事件处理函数（如果已存在一个）
        if (element["on" + type]) {
            handlers[0] = element["on" + type];
        }

    }
```

```
        // 在散列表中存储该事件处理函数
        handlers[handler.$$guid] = handler;

        // 赋予一个全局事件处理函数来处理所有工作
        element["on" + type] = handleEvent;
};

// 创建独立ID的计数器
addEvent.guid = 1;

function removeEvent(element, type, handler) {
        // 从散列表中删除事件处理函数
        if (element.events && element.events[type]) {
            delete element.events[type][handler.$$guid];
        }
};

function handleEvent(event) {
        var returnValue = true;

        // 获取事件对象（IE使用全局的事件对象）
        event = event || fixEvent(window.event);

        // 获取事件处理函数散列表的引用
        var handlers = this.events[event.type];

        // 依次执行每个事件处理函数
        for (var i in handlers) {
            this.$$handleEvent = handlers[i];
            if (this.$$handleEvent(event) === false) {
                returnValue = false;
            }
        }

        return returnValue;
};

// 增加一些IE事件对象的缺乏的方法
function fixEvent(event) {
    // 增加W3C标准事件方法
    event.preventDefault = fixEvent.preventDefault;
    event.stopPropagation = fixEvent.stopPropagation;
    return event;
};

fixEvent.preventDefault = function() {
    this.returnValue = false;
};

fixEvent.stopPropagation = function() {
    this.cancelBubble = true;
};
```

127

1. addEvent的优点

以下是 Dean Edward 的 addEvent 事件绑定方法的优点：

- 可以在所有浏览器中工作，就算是更古老无任何支持的浏览器。
- this关键字可在所有的绑定函数中使用，指向的是当前元素。
- 中和了所有防止浏览器默认行为和阻止事件冒泡的各种浏览器特定函数。
- 不管浏览器类型，事件对象总是作为第一个对象传入。

2. addEvent的缺点

以下是 Dean Edward 的 addEvent 事件绑定方法的优点：

- 仅工作在冒泡阶段（因为它深入使用事件绑定的传统方法）。

128

addEvent/removeEvent 函数如此强大，绝对没有任何理由不在你的代码中使用。从 Dean 的默认代码所展示的过人之处可看到，诸如增加事件对象更好的标准化、事件触发和批量事件删除是十分繁琐的，使用普通的事件结构让所有的事情都难上加难。

6.4 事件类型

常用的 JavaScript 事件可以归结为几类。最常用的类别可能是鼠标交互，紧接着是键盘和表单事件。接下来的列表是可用在 Web 应用程序中的不同类别的事件总览。请参考附录 B 查看大量实践中的事件。

- 鼠标事件：这也可以分成两类：追踪鼠标当前定位（mouseover、mouseout）的事件，和追踪鼠标点击（mouseup、mousedown、click）的事件。
- 键盘事件：这负责追踪键盘敲击和其上下文，比如，追踪表单元素内的敲击而不是发生在整个页面的敲击。与鼠标一样，追踪键盘有3种类型：keyup、keydown和keypress。
- UI事件：这些用来追踪用户使用页面的某一方面是否覆盖了另一方面。比如，使用它们你就可以可靠地掌握用户何时开始在表单元素中输入，而追踪输入的是通过focus和blur（当对象失去焦点的时候使用）两个事件。
- 表单事件：这些相关的交互只会直接发生在表单和表单输入元素上。submit事件用来追踪表单的提交与否，change事件监听用户在一个元素的输入，而select事件在<select>元素更新后触发。
- 加载和错误事件：事件类型的最后一类关于页面本身，监听它自身的加载状态。当用户首次载入页面（load事件）和用户离开页面（unload和beforeunload事件）即被绑定。此外，JavaScript的错误使用错误事件追踪，可以让你独立地处理错误。

记住这些通用的事件类型，建议你翻阅附录 B 的材料，我在那剖析了所有流行的事件如何工作、在不同的浏览器中如何表现，并描述了所有必要的难题以便我们随心使用。

6.5 分离式脚本编程

学习了以上这么多的内容，其实是为了实现一个相当重要的目标：分离地、自然地编写用户交互的 JavaScript。这种编程风格背后的驱动力是，你可以集中精力开发优秀的代码，它可以在现代浏览器中工作，而对于更古老（无支持的）的浏览器可以做到预留退路。

129

为了达到这个目标，你可以结合已经学过的3种技术来完成一个分离式编程的应用程序：

(1) 应用程序中的所有功能都需检验。比如，如果希望访问 HTML DOM 则需要检验它是否存在，是否拥有你需要使用的函数（例如，`if(document && document.getElementById)`）。我们在第2章讨论了这个技术。

(2) 使用 DOM 来快速和统一访问文档中的元素。因为你已经掌握了浏览器所支持的 DOM 函数，你可以自由编写简练的代码，没有 hacks，没有杂乱。

(3) 最后，使用 DOM 和 `addEvent` 函数为文档动态绑定所有的事件。不再允许写诸如此类的代码：`<a href="#" onclick="doStuff();">...</a>`。从分离式编程的角度来说这是非常不好的，因为当 JavaScript 被禁用或者用户使用无 JavaScript 支持的老旧浏览器时，这些代码什么也不干。又因为你给用户指向一个没有意义的 URL，对于已不能使用你的脚本功能的用户来说当然没有任何的交互。

如果你觉得这些论述还不够清晰，可以通过某些方法来重现禁止了任何 JavaScript 或者使用支持较次的浏览器的用户感受。打开浏览器，访问网页，并且关闭 JavaScript。它还能运作吗？关闭所有 CSS，你还能导航到所需要的地方吗？最后，不用鼠标能够继续使用吗？所有这些都是网站应该追求的目标。幸运的是，因为你已经对如何编写真正高效的 JavaScript 代码有了深入的理解，向分离式脚本编程的迁移的代价变得很小，可以用最少的精力就能完成。

6.5.1　JavaScript 禁用的未雨绸缪

第一个需要达到的目标是删除 HTML 文档内的所有行内绑定事件。以下是一些可在文档中找到的常见问题：

- 如果禁止了JavaScript并点击页面的任何或者所有的链接，它们会把你引导到一个页面吗？经常会有开发者把URL写成`href=""`或者`href="#"`，这意味着它们只是为启用了JavaScript的用户带来额外的效果。
- 如果禁止了JavaScript，表单还会工作并能正确提交吗？一个普遍的问题发生在使用`<select>`动态菜单中（这只能在启用了JavaScript才会工作）。

充分利用这些知识，你现在可以开发对用户完全有用的页面了，无论他们禁止了 JavaScript 还是继续使用无支持的浏览器。

6.5.2　确保链接不依赖于 JavaScript

现在用户在页面里执行所有的动作，需要确保在用户执行动作前提供适当的提示。Google 发布了一款 Google 加速器（Google Accelerator），当它遍历所有的页面链接并缓存起来时，用户
130 发现他们的邮件、帖子和消息魔术般地消失了。这是因为开发者在他们的页面里把链接作为删除链接的操作（举例），然后弹出一个确认框（使用 JavaScript）确认删除与否。但是 Google 加速器完全无视弹出框（也是预期中的）并穿越确认最终执行了该链接。

该场景是 Web 中传输文档和文件的 HTTP 规范的一个具体方式。简单地说，当点击链接时发生 GET 请求，而提交表单时则发生 POST 请求。规范指出 GET 请求不应有破坏性的副作用（比

如删除一条消息），这就是为什么 Google 加速器是在尽本分的原因。并非 Google 方面的编程考虑不周，而是 Web 应用程序开发者首先创建了这些链接。

总而言之，所有网站上的链接都不应有破坏性。如果通过链接可以删除、编辑或者修改任何用户的数据，你应该使用表单来进行。

6.5.3 监听 CSS 何时禁用

还有一个更棘手的情形是新旧浏览器之间的交集：那些老到不支持 JavaScript 技术但还足以支持 CSS 样式的浏览器。一条流行的 DHTML 技术是事先隐藏元素（设置 `display` 为 `none` 或者 `visibility` 为 `hidden`）然后在用户首次访问页面时渐显（使用 JavaScript）。但如果用户并未启用 JavaScript 就永远没有机会看到这个元素了。这个问题的一个解决方案如代码清单 6-16 所示。

代码清单6-16 使加载淡入的技术在JavaScript禁用时不失去作用

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta http-equiv="Content-Type" content="text/html;charset=utf-8" />

    <!--一旦JavaScript运行，为<html>元素增加一个新的class，
        我们就可判断JavaScript启用与否。-->
    <script>document.documentElement.className = "js";</script>

    <!--如果JavaScript启动，先隐藏文本块，它随后会淡入。-->
    <style>.js #fadein { display: none }</style>
</head>
<body>
    <div id="fadein">Block of stuff to fade in...</div>
</body>
</html>
```

131

该技术的可用范围当然不止于简单 DHTML 淡入。判断 JavaScript 的启用与否和赋予样式的能力对于小心谨慎的 Web 开发者来说是一个摆脱困境的有力工具。

6.5.4 事件的亲和力

开发纯粹的分离式 Web 应用程序需要考虑的最后一方面是，确保事件在不使用鼠标的情况下依然具备亲和力（accessibility，又称可访问性）。这么做就可以同时服务两类人：需要无障碍辅助工具（例如视力受损者）和不喜用鼠标的人们。（如果有闲心，拔掉鼠标，学习只用键盘来导航网站。这绝对是一件让你大开眼界的体验。）

要使 JavaScript 事件更具亲和力，那么在需要使用点击、鼠标悬停和鼠标离开事件的任何场合下，都要考虑提供非鼠标绑定的可选事件。幸运的是，解决方法也很简单：

- 点击事件：从浏览器角度来说，开发者可用的一条捷径是让点击事件在敲击回车键的时

候能触发。这就完全没必要为点击事件提供另一可选的版本了。但需要注意的一点是，有些开发者喜欢在表单的提交按钮绑定点击事件处理函数来监听用户是否提交。其实，开发者应该为表单对象绑定的是提交事件，这是更加可靠的选择。

- 鼠标悬停事件：当使用键盘来导航网页，实际上改变的是不同元素的焦点。通过同时为鼠标悬停和聚焦事件绑定事件，你就可以确保为键盘和鼠标用户提供可相提并论的解决方案了。
- 鼠标离开事件：就如为鼠标悬停事件准备的聚焦事件，当用户的焦点从元素移开就会发生模糊事件。因此你就可以使用键盘的模糊事件来模拟鼠标离开事件了。

现在已经掌握表现相同的相应事件的配对，你可以修改代码清单 6-3，让它在没有鼠标的情况下也可以建立类似悬停的可运行效果，如代码清单 6-17 所示。

代码清单6-17　为元素绑定事件配对以增加网页的可访问性

```
// 查找所有的<a>元素，并为其绑定事件处理函数

var li = document.getElementsByTagName("a");
for ( var i = 0; i < a.length; i++ ) {

  // 绑定鼠标悬停和聚焦事件处理函数到<li>元素上，
  // 当用户把鼠标悬停到链接上，或者（使用键盘）聚焦到链接上时，它会把<li>的背景颜色变成蓝色，
  a[i].onmouseover = a[i].onfocus = function() {
    this.style.backgroundColor = 'blue';
  };
  // 绑定鼠标离开和模糊事件处理函数到<li>元素上，
  // 当用户从链接移开时，它会把<li>的背景颜色变成默认的白色。
  a[i].onmouseout = a[i].onblur = function() {
    this.style.backgroundColor = 'white';
  };

}
```

132

在实际应用程序中，增加键盘事件的处理，作为典型鼠标事件的额外补充，尽管显得微不足道，但可以作为一个辅助的方法让键盘用户更好地使用你的网站，这样对每个人都有好处。

6.6　小结

到现在为止，你已经掌握了如何遍历 DOM 和为 DOM 元素绑定事件处理函数，并且了解到编写分离式 JavaScript 代码的好处，你已经可以实现一些大型应用程序和眩目的效果了。

在本章中，开始介绍了 JavaScript 中的事件如何工作以及与其他语言的事件模型的对比，继而讲解了事件模型提供的信息和如何最大程度地控制它。然后探索为 DOM 元素绑定事件，以及可用事件的不同类型。最后以展示一些如何为页面集成分离式脚本技术作为总结。

第 7 章你将继续学习如何编写一些动态效果和交互，它们会运用到你所学到的这一章的大量技术。

133

第7章

JavaScript与CSS

JavaScript 与 CSS 之间的交互是现代 JavaScript 编程的支柱之一，运用某种形式的动态交互几乎是所有现代 Web 应用程序的必备条件。由此带来的好处是，页面加载时间更短，用户能够快速地使用。加上第 6 章关于事件的动态技术，它们构成了创建无缝且强大用户体验的基础。

CSS 是产生可用的、引人注目的 Web 页面的事实标准，它不仅给开发者提供最强大的工具，同时给用户提供最小的使用难度。更有趣的是，结合 JavaScript 的能力，你就可以构建强有力的界面，包括动画、小物件（widget）和动态显示等诸如此类的效果。

7.1 访问样式信息

JavaScript 和 CSS 的结合无非是交互产生的结果。理解哪些是可用的信息，更有助于精确地实现你所需要达到的交互目标。

访问和设置 CSS 属性的主要工具是元素自身的样式属性。假设需要获取元素的高度，则可以编写类似这样的代码：`elem.style.height`。设置元素的高度则可以执行这样的代码：`elem.style.height = '100px'`。

刚开始在 DOM 元素中使用 CSS 属性，你可能会遇到两个问题，因为它们可能并不会如你所期望的去工作。首先，JavaScript 要求你在设置任何几何属性时必须明确尺寸单位（如上一个设置高度的例子中，必须使用 px 单位）。同时，任何几何属性都会返回表示元素样式的字符串而非数值（比如，是 100px 而非 100）。

其次，如果一个元素是 100 像素高，想获取它的当前高度，你可能希望只从样式属性中就能得到 100px 这样的精确结果。但是并不能时时如愿。这是因为你使用样式表或者行内 CSS 所预设的样式信息并不能精确可靠地反映到当前的样式属性中。

这迫使我们只能去编写一个用于处理 JavaScript 中的 CSS 的重要函数：一种获取元素的真实、最终样式的方法，它可以给你精确、预期的值。解决最终样式问题相对可靠的方法有好几种，使用这些方法（W3C 或者 IE 特有的种种方法）就可得到元素的真实、最终的样式值。在开发一些需要追求绝对精确的元素视觉效果时，使用这些方法可立竿见影。

同时需要注意的是，获取元素的最终样式，不同浏览器之间存在大量的不同。比如，获取元素的当前样式，IE 有它自己的独特方式，而其他浏览器使用 W3C 定义的方式。

代码清单 7-1 展示了获取元素的最终样式值的函数，代码清单 7-2 则是使用这个新函数的具体例子。

代码清单7-1　获取元素的真实、最终的CSS样式属性值的函数

```
// 获取指定元素（elem）的样式属性（name）
function getStyle( elem, name ) {
        // 如果属性存在于style[]中，那么它已被设置了（并且是当前的）
        if (elem.style[name])
            return elem.style[name];

        // 否则，尝试使用IE的方法
        else if (elem.currentStyle)
            return elem.currentStyle[name];

        // 或者W3C的方法，如果存在的话
        else if (document.defaultView && document.defaultView.getComputedStyle) {
            // 它使用的是通用的 'text-align'的样式规则而非 'textAlign'
            name = name.replace(/([A-Z])/g,"-$1");
            name = name.toLowerCase();

            // 获取样式对象并获取属性（存在的话）值
            var s = document.defaultView.getComputedStyle(elem,"");
            return s && s.getPropertyValue(name);

        // 否则，用户使用的是其他浏览器
        } else
            return null;
    }
```

136

代码清单7-2　一个元素的CSS最终值并不一定等于样式对象值的情形

```
<html>
<head>
        <style>p { height: 100px; }</style>
        <script>
        window.onload = function(){
            // 定位到段落并检查它的高度
            var p = document.getElementsByTagName("p")[0];

            // 使用传统的方式来检查
            alert( p.style.height + " should be null" );

            // 检查最终高度的值
            alert( getStyle( p, "height" ) + " should be 100px" );
        };
        </script>
</head>
<body>
        <p>I should be 100 pixels tall.</p>
</body>
</html>
```

代码清单 7-2 展示了如何获取一个 DOM 元素的 CSS 属性的最终真实值。在这个例子中，你得到了元素的真实像素高度，尽管这个高度值是通过文件的头部 CSS 设置的。需要注意的是，这个函数忽略其他的计量单位（比如使用百分比）。所以这个方法虽不十分完美，但已经是一个很好的开端。

有了这个工具在手，你就可以进一步了解如何获取和设置一些基本的 DHTML 交互的属性了。

7.2 动态元素

动态元素的前提条件是使用 JavaScript 和 CSS 创建非静态效果的元素。一个简单的例子是，你对新闻列表有兴趣而选中了复选框，它会弹出一个输入电子邮件的区域。

从根本上说，创建动态效果有 3 个至关重要的属性：位置、尺寸和可见性。使用这 3 个属性你就可以在现代浏览器上模拟绝大部分常见的用户交互效果了。

7.2.1 元素的位置

利用元素的位置是在页面内构造交互元素的重要基石。访问和修改 CSS 位置属性让你能高效地模拟部分流行的动画和交互（如拖放）。

137

利用元素位置重要的一步是了解 CSS 的位置系统是如何工作的，而这也将会在后面大量用到。在 CSS 中，元素使用偏移来定位，它的测量是通过元素到它父亲的左上角的偏移量来衡量的。CSS 中的坐标体系可以参考图 7-1。

图7-1 一个使用CSS的页面的坐标体系例子

页面上所有的元素都有某种程度上的 top（垂直坐标）和 left（水平坐标）的偏移。一般来说，绝大部分元素都是简单地静态定位于包含它的相关元素上。元素可以有一系列不同的定位方法，如 CSS 标准的提议一样。为更好地理解它，先看代码清单 7-3 中一个简单的 Web 页面。

代码清单7-3 一个可用来显示不同定位的HTML Web页面

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

```
p{
        border: 3px solid red;
        padding: 10px;
        width: 400px;
        background: #FFF;
}
p.odd {
    /* 定位信息在此 */
    position: static;
    top: 0px;
    left: 0px;
}
</style>
</head>
<body>
    <p>Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam ...</p>
    <p class='odd'>Phasellus dictum dignissim justo. Duis nec risus id nunc...</p>
    <p>Sed vel leo. Nulla iaculis, tortor non laoreet dictum, turpis diam ...</p>
</body>
</html>
```

有了这个架好的简单 HTML 页面，让我们来看看如何来定位第二个段落，它可以导致页面不同的布局。

- 静态定位：这是元素定位的默认方式，它简单地遵循文档的普通流动（flow）。当元素是静态定位时，top和left属性无效。图7-2展示了CSS定位为position:static;top: 0px;left: 0px的段落的结果。

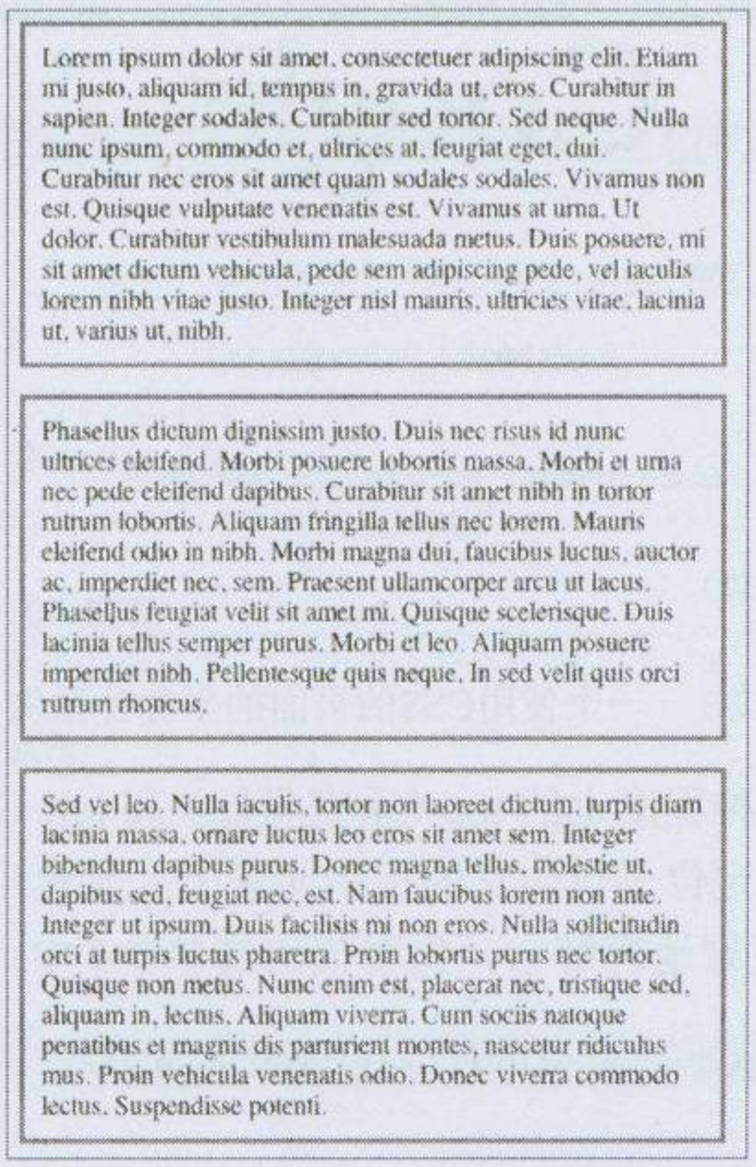


图7-2　页面内的普通（静态）流动段落

- 相对定位：这种定位形式与静态定位非常相似，因为元素会继续遵循文档的普通流动，除非受到其他指令的影响。但是，设置top或者left属性会引起元素相对于它的原始（静态）位置进行偏移。图7-3是一个相对定位的例子，它的CSS定位是position: relative; top: -50px; left: 50px;。

140

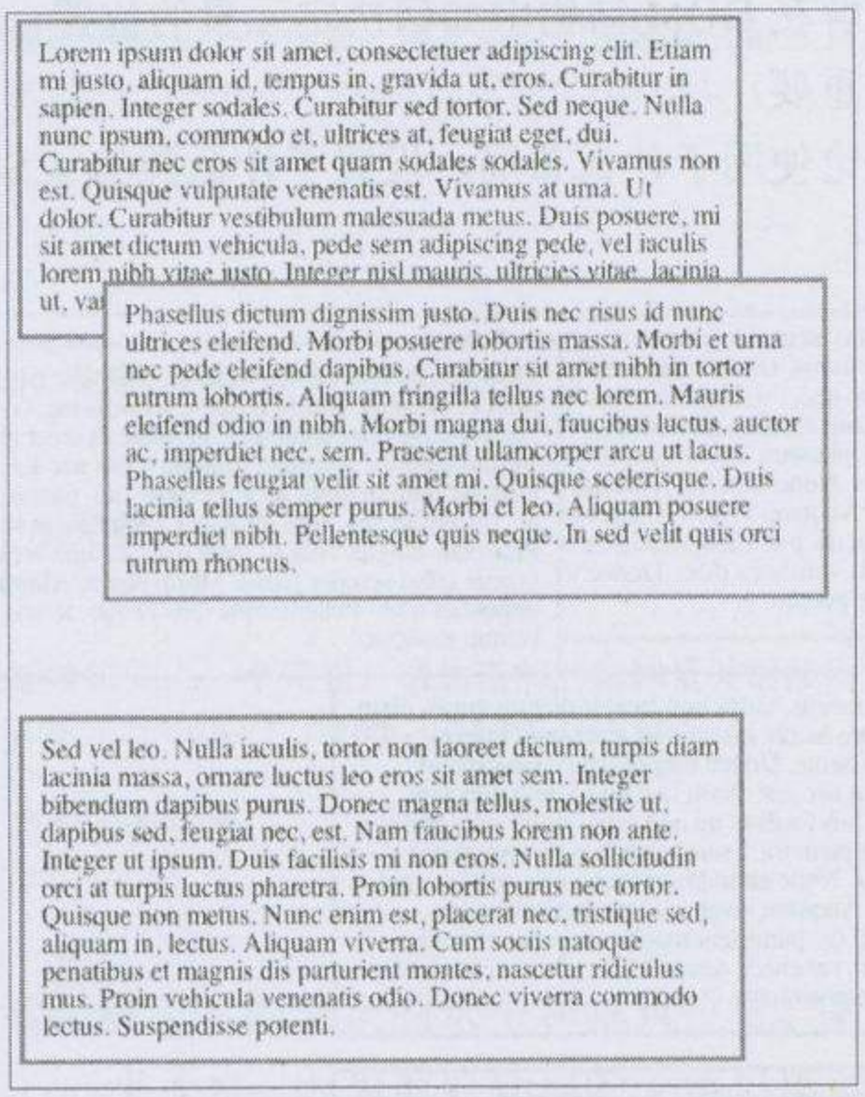


图7-3 相对定位的元素偏移原有位置，而不遵循文档的普通流动

- 绝对定位：绝对定位的元素完全跳出页面布局的普通流动，它会相对于它的第一个非静态定位的祖先元素而展示。如果没有这样的祖先元素，则相对于整个文档。图7-4是绝对定位的例子，它的CSS定位是absolute; top: 20px;left: 0px;。

141

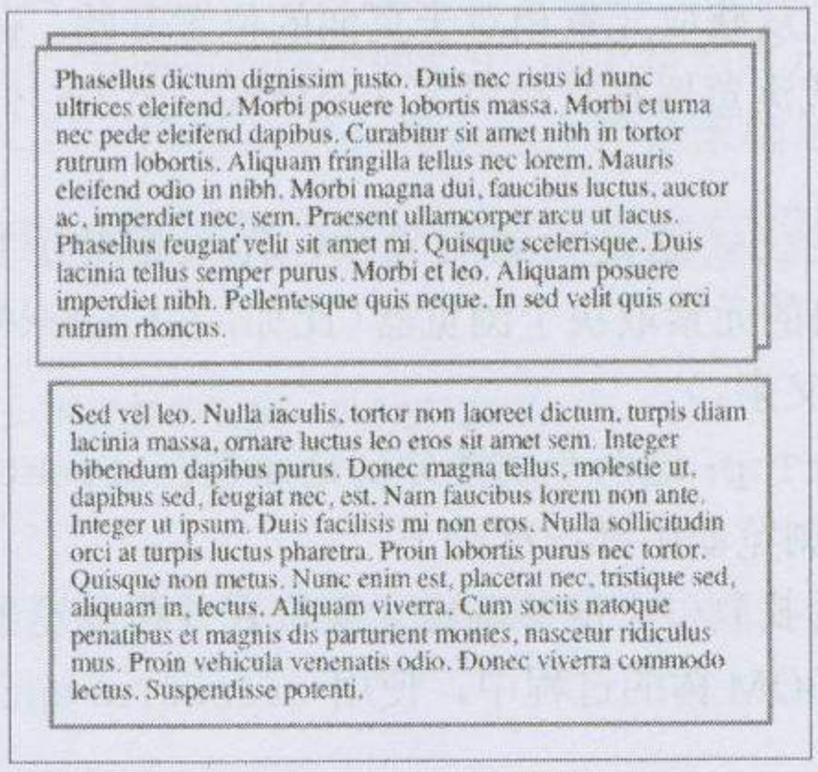


图7-4 靠近页面左上角的绝对定位的元素，在这个元素之上的元素也在此显示

- 固定定位：固定定位把元素相对于浏览器窗口而定位。设置元素的top和left为0会使它显示在浏览器左上角，它完全忽略浏览器滚动条的拖动，一直会出现在用户的视野。图7-5是一个固定定位的例子，它的CSS定位是position: fixed; top: 20px; right: 0px;。

理解元素的定位对于理解在DOM结构中如何决定元素的位置，或者说，该使用哪种定位手段才能达到最佳效果都非常重要。

接下来，我们将探索无论使用了什么布局，设置了什么样的CSS属性，应该如何提取和操作元素的精确位置。

142

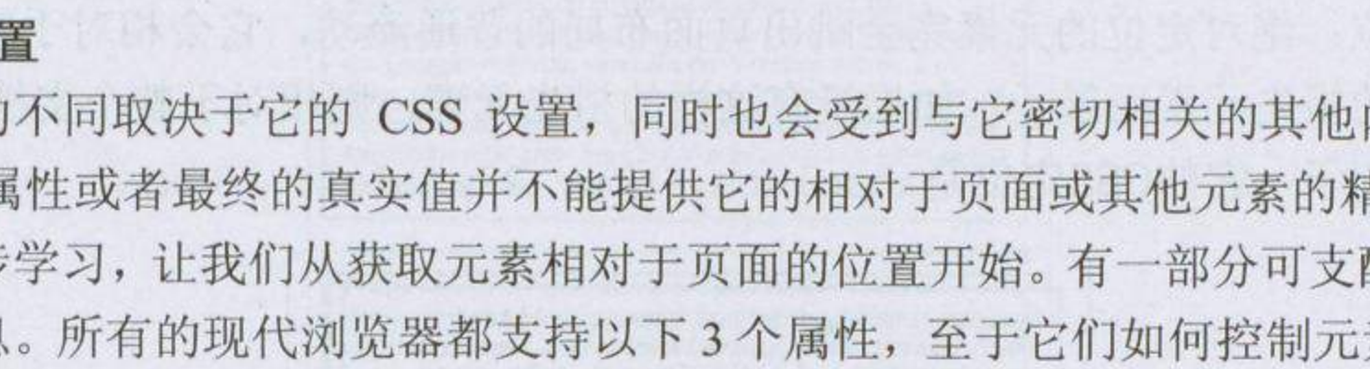


图7-5　固定定位，就算浏览器窗口已经滚动页面，元素也固定于页面的右上角

1. 获取位置

元素位置的不同取决于它的CSS设置，同时也会受到与它密切相关的其他内容的影响。访问元素的CSS属性或者最终的真实值并不能提供它的相对于页面或其他元素的精确位置。

为了进一步学习，让我们从获取元素相对于页面的位置开始。有一部分可支配的元素属性以供查找这些信息。所有的现代浏览器都支持以下3个属性，至于它们如何控制元素的位置，则是另外的话题了。

- offsetParent：理论上，这是元素的父亲，元素相对于它定位。但是，在实际中，offsetParent所指向的元素取决于浏览器（比如，在Firefox中，它指向根节点，而在Opera中，就是元素的直接父亲。）
- offsetLeft和offsetTop：这两个属性分别是元素在offsetParent上下文中的水平和垂直偏移量。它在现代浏览器中都很准确。

现在棘手的问题是，能否找到一个能够确定元素位置并跨浏览器的牢固方法。这个方法如代码清单7-4所示，它在遍历DOM树的过程中，使用offsetParent属性并累加它们的偏移量来计算元素的位置。

143

代码清单7-4 两个确定元素相对于整个文档的x和y位置的辅助函数

```
// 获取元素的x（水平，左端）位置
function pageX(elem) {
        // 查看我们是否位于根元素
        return elem.offsetParent ?

            // 如果我们能继续得到上一个元素，增加当前的偏移量并继续向上递归
            elem.offsetLeft + pageX( elem.offsetParent ) :

            // 否则，获取当前的偏移量
            elem.offsetLeft;
}

// 获取元素的Y（垂直，顶端）位置
function pageY(elem) {

        // 查看我们是否位于根元素
        return elem.offsetParent ?

            // 如果我们能继续得到上一个元素，增加当前的偏移量并继续向上递归
            elem.offsetTop + pageY( elem.offsetParent ) :

            // 否则，获取当前的偏移量
            elem.offsetTop;
}
```

接下来的难题是找出元素相对于它的父亲的水平和垂直位置。需要注意的是，简单使用元素的 style.left 或者 style.top 并不够，因为需要处理的元素可能尚未经过 JavaScript 或者 CSS 的样式化。

使用元素相对于其父亲的位置，就可以为 DOM 增加额外的元素，并相对定位于它的父亲。例如，它对制作上下文工具条提示非常有用。

要找到元素相对于它的父亲元素的位置，你必须再次使用 offsetParent 属性。因为该属性并不保证能够返回指定元素的真实父亲，你必须使用 pageX 和 pageY 函数来找出父亲元素和子元素之间的差距。在代码清单 7-5 中展示的两个函数中，如果是当前元素的真实父亲，使用 offsetParent；否则，继续遍历 DOM，使用 pageX 和 pageY 方法来确定它的真实位置。 144

代码清单7-5 确定元素相对于父亲的位置的两个函数

```
// 获取元素相对于父亲的水平位置
function parentX(elem) {
        // 如果offsetParent是元素的父亲，那么提前退出
        return elem.parentNode == elem.offsetParent ?
            elem.offsetLeft :

            // 否则，我们需要找到元素和元素的父亲相对于整个页面位置，并计算他们之间的差
            pageX( elem ) - pageX( elem.parentNode );
}

// 获取元素相对于父亲的垂直位置
```

```
function parentY(elem) {
      // 如果offsetParent是元素的父亲，那么提前退出
      return elem.parentNode == elem.offsetParent ?
          elem.offsetTop :

          // 否则，我们需要找到元素和元素的父亲相对于整个页面位置，并计算他们之间的差
          pageY( elem ) - pageY( elem.parentNode );
}
```

关于元素位置的最后一个难题是，获取元素相对于它的 CSS 容器的位置。如前所述，即使元素包含在一个元素内，但可以相对于其他的父亲元素而定位（使用相对和绝对定位）。记住这一点，你就可以求助 getStyle 函数得到 CSS 偏移的最终值，这等于元素的位置值。

要解决这个问题，你可使用代码清单 7-6 所示的两个简单的封装函数。它们都只调用 getStyle 函数，同时去除了干扰性的单位信息（如 100px 会变成 100，单位只有在不基于像素的布局才显得重要）。

代码清单7-6　获取元素的CSS位置的辅助函数

```
// 查找元素的左端位置
function posX(elem) {
      // 获取最终样式并得到数值
      return parseInt( getStyle( elem, "left" ) );
}
// 查找元素的顶端位置
function posY(elem) {
    // 获取最终样式并得到数值
    return parseInt( getStyle( elem, "top" ) );
}
```

145

2. 设置位置

与获取元素的位置相比，设置位置更缺乏弹性。但是结合不同的布局手段（绝对、相对、固定等）你就可以得到有可比性也可用的结果。

目前，调整元素位置的唯一方法是修改它的 CSS 属性。尽管还有其他属性存在（如 bottom 和 right），但仅需要修改的是 left 和 top 属性。从创建如代码清单 7-7 中所示的两个函数开始，可以用它们来设置元素的位置，不管这个的当前位置在哪里。

代码清单7-7　设置元素x和y位置（与当前位置无关）的一对函数

```
// 设置元素水平位置的函数
function setX(elem, pos) {
      // 使用像素单位设置CSS的'left'属性
      elem.style.left = pos + "px";
}

// 设置元素垂直位置的函数
function setY(elem, pos) {
      // 使用像素单位设置CSS的'top'属性
      elem.style.top = pos + "px";
}
```

最后，还需要开发另外一对函数，如代码清单 7-8 所示，你可用它们来设置元素相对于它之前最后一次变更位置之间的差距，比如，调整元素的位置向左偏移 5 像素。各种不同的动画效果（DHTML 开发重要部分），都跟这些方法密切相关。

代码清单7-8 调整元素相对于当前位置的距离的一对函数

```
// 在元素的水平位置上增加像素距离的函数
function addX(elem,pos) {
      // 获取当前水平位置，然后增加偏移量
      setX( posX(elem) + pos );
}
// 在元素的垂直位置上增加像素距离的函数
function addY(elem,pos) {
    // 获取当前垂直位置，然后增加偏移量
    setY( posY(elem) + pos );
}
```

146

到目前为止，我们已经走完关于元素位置的旅程。理解元素的定位是如何工作的、如何设置和获取精确的位置，是动态元素起作用的基础。下一节探索的是元素的精确尺寸。

7.2.2 元素的尺寸

找出元素的高度和宽度可以很容易，也可以很困难，取决于它所处的不同场合。在大多数情况下，仅需要使用 getStyle 函数（如代码清单 7-9 所示）的修改版本就可得到元素的当前高度和宽度。

代码清单7-9 获取元素当前的高度和宽度

```
// 获取元素的真实高度（使用CSS最终样式值）
function getHeight( elem ) {
      // 获取CSS的最终值并解析出可用的数值
      return parseInt( getStyle( elem, 'height' ) );
}

// 获取元素的真实宽度（使用CSS最终样式值）
function getWidth( elem ) {
      // 获取CSS的最终值并解析出可用的数值
      return parseInt( getStyle( elem, 'width' ) );
}
```

当你试图做如下两件事情的时候，会出现问题。其一，需要获取有预定义高度元素的完整高度。比如，以 0 像素开始的动画，但你需要事先知道元素究竟能有多高或多宽。其二，当元素的 display 为 none 的时候，你会得不到这个数值。在需要执行动画的时候这两个问题就会发生。一个对象的动画从 0 像素开始（也可能是它的 display 是 none），你需要把高度增大到它的潜在尺寸。

代码清单7-10的两个函数展示了如何找到元素的潜在的完整高度和宽度。这需要通过访问clientWidth和clientHeight属性来实现，它们提供了元素可能达到的总尺寸。

147

代码清单7-10　即使元素隐藏，亦能分别获取它潜在的完整高度和宽度的两个函数

```
// 查找元素完整的、可能的高度
function fullHeight( elem ) {
    // 如果元素是显示的，那么使用offsetHeight就能得到高度，如果没有offsetHeight，则使用
       getHeight()
    if ( getStyle( elem, 'display' ) != 'none' )
        return elem.offsetHeight || getHeight( elem );

    // 否则，我们必须处理display为none的元素，所以重置它的CSS属性以获取更精确的读数
    var old = resetCSS( elem, {
        display: '',
        visibility: 'hidden',
        position: 'absolute'
    });

    // 使用clientHeight找出元素的完整高度，如果还不生效，则使用getHeight函数
    var h = elem.clientHeight || getHeight( elem );

    // 最后，不要忘记恢复CSS的原有属性
    restoreCSS( elem, old );

    // 并返回元素的完整高度
    return h;
}

// 查找元素完整的、可能的宽度
function fullWidth( elem ) {
    // 如果元素是显示的，那么使用offsetHeight就能得到高度，如果没有offsetWidth，则使用
       getWidth()
    if ( getStyle( elem, 'display' ) != 'none' )
        return elem.offsetWidth || getWidth( elem );

    // 否则，我们必须处理display为none的元素，所以重置它的CSS属性以获取更精确的读数
    var old = resetCSS( elem, {
        display: '',
        visibility: 'hidden',
        position: 'absolute'
    });
    // 使用clientWidth找出元素的完整宽度，如果还不生效，则使用getWidth函数
    var w = elem.clientWidth || getWidth( elem );

    // 最后，不要忘记恢复CSS的原有属性
    restoreCSS( elem, old );

    // 并返回元素的完整宽度
    return w;
}

// 设置CSS一组属性的函数，它可以恢复到原有设置
function resetCSS( elem, prop ) {
    var old = {};

    // 遍历每一个属性
```

148

```
    for ( var i in prop ) {
        // 记录旧的的属性值
        old[ i ] = elem.style[ i ];

        // 并设置新的值
        elem.style[ i ] = prop[i];
    }

    // 返回已经变化的值的集合，预留给restoreCSS函数使用
    return old;
}

// 恢复CSS原有属性值，防止resetCSS函数副作用的函数
function restoreCSS( elem, prop ) {
    // 重置所有属性，恢复它们的原有值
    for ( var i in prop )
        elem.style[ i ] = prop[ i ];
}
```

有了获取元素当前和潜在高度及宽度的能力，就可以使用它们来实现一些动画了。在具体讨论动画之前，先来看看如何修改元素的可见性。

7.2.3 元素的可见性

元素的可见性是一个强大的工具，用在 JavaScript 中可以创建包括从动画效果到快速模板化的各个方面。更重要的是，它可以迅速从视觉中隐藏元素，提供一些基本的用户交互能力。 149

CSS 有两种不同的方式可以有效地隐藏元素，它们均有自己的优缺点，但会导致不同的结果，取决于你如何使用它们。

- visibility属性在切换元素可见性的同时会保持元素普通流的属性的相关影响。它有两个值：visible（默认的）和hidden（不可见的）。假设一小段文本包含在<b>标签内，同时<b>的visibility设置为hidden，那么结果就是文本内有一块空白，它的尺寸刚好等于被包裹文本的原有尺寸。比较以下两行文本：

```
//普通文本：
Hello John, how are you today?

//'John'设置了visibility: hidden
Hello, how are you today?
```

- display属性为开发者控制元素的布局提供了更丰富的选项。它可以是inline（比如<b>和的标签是inline的，它们都遵循文本的普通流动）、block（比如<p>和 的标签是block的，它们都打破文本的普通流动）或none（它完全从文档中隐藏了元素）。设置元素display属性的结果跟从文档中删除了该元素的情形看起来一样，虽然如此，情况还是有所不同的，因为该元素还可以迅速的切换回文档的视觉中来。以下的几行代码展示了display属性的行为：

```
//普通文本:
    Hello John, how are you today?

    //'John'设置了display: none
    Hello, how are you today?
```

但 visibility 属性有它特定的用法，所以并不能过于夸大 display 属性的重要性。事实是，在大部分应用程序中，设置 visibility 为 hidden 的元素依然存在于文档普通流动中是可行的，虽然并不是很流行。代码清单 7-11 中是两个使用 display 属性来切换元素的可见性的方法。

代码清单7-11　使用CSS的display属性来切换元素可见性的一组函数

```
// 使用display隐藏元素的函数
function hide( elem ) {
        // 找出元素display的当前状态
        var curDisplay = getStyle( elem, 'display' );

        // 记录它的display状态
        if ( curDisplay != 'none' )
            elem.$oldDisplay = curDisplay;
        // 设置display为none（即隐藏了元素）
        elem.style.display = 'none';
}

// 使用display显示元素的函数
function show( elem ) {
    // 设置display属性为它的原始值，如没有记录有原始值，则使用'block'
    elem.style.display = elem.$oldDisplay || '';
}
```

150

关于元素可见性的第二个方面是透明度（opacity）。调整元素的透明度产生的结果与调整元素的 visibility 类似，但可以提供表现可见性效果的更多控制。这表示你可以使一个元素 50% 透明，让你可以透过它看到位于它之下的元素。所有的现代浏览器都在一定程度上支持透明度，包括 IE（从 5.5 起）和兼容 W3C 的浏览器，但它们的实现有所不同。为解决这个不一致的问题，你可以创建一个标准的函数，用以操作元素的透明度，如代码清单 7-12 所示。0 表示完全透明，而 100 则相反。

代码清单7-12　调节元素透明度的函数

```
// 设置元素的透明度（级别从0-100）
function setOpacity( elem, level ) {
        // 如果存在filters这个属性，则它是IE，所以设置元素的Alpha滤镜
        if ( elem.filters )
            elem.style.filters = 'alpha(opacity=' + level + ')';

        // 否则，使用W3C的opacity属性
        else
            elem.style.opacity = level / 100;
}
```

有了这些调整元素位置、尺寸和可见性的方法，下面就可以探索一些有趣的东西了，来发挥它们结合到一起的威力。

7.3 动画

现在你已经有了构建基本 DHTML 操作的能力，让我们来看看动态 Web 应用程序中最流行的、最显著的效果：动画。巧妙地使用动画可以为用户提供十分有用的反馈，可以把用户的注意力吸引到屏幕上新建的一个元素上来。

首先看看两种受欢迎的动画，然后在探讨流行的 DHTML 库时再来重温这些主题。 151

7.3.1 滑动

第一个动画是把一个隐藏元素（使用 `display:none` 实现）显示出来。与使用效果生硬的 `show()` 函数相比，你更希望能将这个元素在短时间内随着高度的增加而逐步出现。那么你可以使用代码清单 7-13 所示的函数，它以向下拉伸的形式取代 `show()` 函数的直接方式，提供了更为平滑的用户视觉体验。

代码清单7-13 通过在短时间内增加高度逐步显示隐藏元素的函数

```
function slideDown( elem ) {
    // 从0高度开始滑动
    elem.style.height = '0px';

    // 先显示元素（但是看不到它，因为它的高度是0）
    show( elem );

    // 找到元素的完整的潜在高度
    var h = fullHeight( elem );

    // 我们在1秒钟内执行一个20帧的动画
    for ( var i = 0; i <= 100; i += 5 ) {
        // 保证我们能够保持正确的'i'的闭包函数
        (function(){
            var pos = i;

            // 设置timeout以让它能在指定的时间点运行
            setTimeout(function(){

                // 设置元素的新高度
                elem,.style.height = ( pos / 100 ) * h ) + "px";

            }, ( pos + 1 ) * 10 );
        })();
    }
}
```

7.3.2 渐显

接下来的动画与第一个非常相似，但利用了 `setOpacity()` 函数改变透明度而不是改变高度。这个具体的函数（如代码清单 7-14 所示）显示一个隐藏元素并使透明度从 0（完全透明）到 100%（完全不透明）逐渐显现。与代码清单 7-13 的函数非常相似，它也提供了更为平滑的用户视觉体验。 152

代码清单7-14　通过在短时间内增加透明度逐步显示隐藏元素的函数

```
function fadeIn( elem ) {
        // 从0透明度开始
        setOpacity( elem, 0 );

        // 先显示元素（但是看不到它，因为它的透明度是0）
        show( elem );

        // 我们在1秒钟内执行一个20帧的动画
        for ( var i = 0; i <= 100; i += 5 ) {
            // 保证我们能够保持正确的'i'的闭包函数
            (function(){
                var pos = i;

                // 设置timeout以让它能在指定的事件内运行
                setTimeout(function(){

                    // Set the new opacity of the element
                    setOpacity( elem, pos );

                }, ( pos + 1 ) * 10 );
            })();
        }
}
```

这些动画以及其他一些效果的例子，在第 9 章中将会讲述。

7.4　浏览器

掌握对特定的 DOM 对象进行操作后，了解如何修改或追踪浏览器及其组件的相关信息，让你可以为站点用户增加更多的交互。最常用的是检查鼠标光标的位置和用户滚动页面的距离两个方面。

7.4.1　鼠标位置

获取鼠标的位置是编写拖放操作和上下文菜单的基础，这两种效果都只能通过 JavaScript 和 CSS 的相互作用产生。

首先需要确定的两个变量是，光标相对于整个页面的 x 和 y 位置（如代码清单 7-15 所示）。因为只能从鼠标事件中才能得到鼠标坐标的信息，所以需要通过一般的鼠标事件来捕获，比如 mousemove 或者 mousedown（可以在 7.5 节中找到更多的例子）。

153

代码清单7-15　两个通用函数，用以获取鼠标光标相对于整个页面的当前位置

```
// 获取光标的水平位置
function getX(e) {
        // 标准化事件对象
        e = e || window.event;

        // 先检查非IE浏览器的位置，再检查IE的位置
```

```
        return e.pageX || e.clientX + document.body.scrollLeft;
}

// 获取光标的垂直位置
function getY(e) {
        // 标准化事件对象
        e = e || window.event;

        // 先检查非IE浏览器的位置，再检查IE的位置
        return e.pageY || e.clientY + document.body.scrollTop;
}
```

最后，与鼠标相关的变量还有，光标相对于它当前正在交互的元素的 x 和 y 位置。代码清单 7-16 展示的两个函数可以用来获取这些数值。

代码清单7-16 两个获取鼠标相对于当前元素位置的函数

```
// 获取鼠标相对于当前元素（事件对象'e'的属性target）的X位置
function getElementX( e ) {
        // 获取正确的元素偏移量
        return ( e && e.layerX ) || window.event.offsetX;
}

// 获取鼠标相对于当前元素（事件对象'e'的属性target）的Y位置
function getElementY( e ) {
        // 获取正确的元素偏移量
        return ( e && e.layerY ) || window.event.offsetY;
}
```

我们还会在 7.5 节中的实现浏览器拖放的方法中重温鼠标交互。此外，更多鼠标事件的例子，请参考第 6 章和附录 B。

154

7.4.2 视口

浏览器的视口（viewport）可以看作是浏览器滚动条内的一切东西。视口还包含的部分组件是视口窗口、页面和滚动条等。获取这些组件正确的位置和尺寸，对于开发显示很长的内容（比如自动滚动和聊天室等）的交互来说是必需的。

1. 页面尺寸

首先需要了解的属性是当前页面的高度和宽度。大部分的页面都被视口所切分（可以通过检查视口尺寸和滚动条位置确定）。代码清单7-17所示的两个函数使用前面提到的scrollWidth和scrollHeight属性，它们详细描述了元素的潜在宽度和高度，而不只是当前所看到的尺寸。

代码清单7-17 确定当前页面高度和宽度的两个函数

```
// 返回页面的高度（增加内容的时候可能会改变）
function pageHeight() {
        return document.body.scrollHeight;
}

// 返回页面的宽度
```

```
function pageWidth() {
      return document.body.scrollWidth;
}
```

2. 滚动条位置

接下来要了解的是如何确定浏览器滚动条的位置（或者，换言之，页面相对于视口的顶端距离）。获取这些数值（可以通过如代码清单 7-18 所示的函数得到）对于开发超越浏览器默认滚动的动态应用程序是必要的。

代码清单7-18　确定文档视口定位的两个函数

```
// 确定浏览器水平滚动位置的函数
function scrollX() {
      // 一个快捷方式，用在IE6/7的严格模式中
      var de = document.documentElement;

      // 如果浏览器存在pageXoffset属性，则使用它
      return self.pageXOffset ||

      // 否则，尝试获取根节点的左端滚动的偏移量
      ( de && de.scrollLeft ) ||

      // 最后，尝试获取body元素的左端滚动的偏移量
      document.body.scrollLeft;
}

// 确定浏览器垂直滚动位置的函数
function scrollY() {
    // 一个快捷方式，用在IE6/7的严格模式中
    var de = document.documentElement;

    // 如果浏览器存在pageYoffset属性，则使用它
    return self.pageYOffset ||

        // 否则，尝试获取根节点的顶端滚动的偏移量
        ( de && de.scrollTop ) ||

        // 最后，尝试获取body元素的顶端滚动的偏移量
        document.body.scrollTop;
}
```

3. 移动滚动条

现在你已经知道了页面内滚动条的当前偏移量和页面本身的尺寸，那么可以看看 scrollTo 方法了，它由浏览器自身提供，可以用来调整页面视口的当前位置。

scrollTo 方法作为 window 对象（和其他包含滚动内容的元素或者<iframe>）的一个属性而存在，它带两个参数，即 x 和 y 偏移量，可以滚动到视口（元素或<iframe>）指定位置。代码清单 7-19 展示了两个使用 scrollTop 方法的例子。

代码清单7-19　使用scrollTo方法调整浏览器窗口位置的例子

```
// 如果需要滚动到浏览器的顶端，你可以这么做：
```

```
window.scrollTo(0,0);

// 如果需要滚动到指定的元素，则可以这么做：
window.scrollTo( 0, pageY( document.getElementById("body") ) );
```

4. 视口尺寸

视口的最后一方面可能是最显而易见的——本身的尺寸。获取视口的尺寸就可深入了解用户当前可以看到的内容有多少，而不必考虑用户的屏幕分辨率或者浏览器窗口的大小。你可以使用如代码清单 7-20 所示的两个函数来确定这些数值。

156

代码清单7-20　确定浏览器视口的高度和宽度的两个函数

```
// 获取视口的高度
function windowHeight() {
        //  一个快捷方式，用在IE6/7的严格模式中
        var de = document.documentElement;

        // 如果浏览器存在innerHeight属性，则使用它
        return self.innerHeight ||

            // 否则，尝试获取根节点高度偏移量
            ( de && de.clientHeight ) ||

            // 最后，尝试获取body元素的高度偏移量
            document.body.clientHeight;
}

// 获取视口的宽度
function windowWidth() {
        //  一个快捷方式，用在IE6/7的严格模式中
        var de = document.documentElement;

        // 如果浏览器存在innerWindth属性，则使用它
        return self.innerWidth ||

            // 否则，尝试获取根节点宽度偏移量
            ( de && de.clientWidth ) ||

            // 最后，尝试获取body元素的宽度偏移量
            document.body.clientWidth;
}
```

视口的作用不能不重视。只要看看一些现代的 Web 应用程序，比如 Gmail 或 Campfire，它们都是由视口的操作产生引人注目的结果（Gmail 提供上下文的覆盖图，而 Campfire 提供聊天内容自动滚动的效果）。在第 11 章中我们会讨论视口的不同方式，它们在高度交互的 Web 应用中，可以提供更好的体验。

7.5　拖放

浏览器中最为流行的用户交互之一是在页面内拖动元素。使用已经学习到的知识（确定元素

157 的位置、如何调整它的位置和各种不同类型的定位等），你就完全能够理解如何拖放元素。

为探索这项技术，我选择由 Aaron Boodman 创建的 DOM-Drag 库（http://boring.youngpup.net/2001/domdrag）来解释。这个库提供了很多易用的特性，包括：

- 拖放处理函数：你可以在移动一个父元素的同时拖放另一个次级元素。这对创建类似窗口的元素界面十分有用。
- 回调函数：你可以监听特定的事件，比如用户开始拖放元素、正在拖放元素和停止拖放元素等事件，以及元素的当前位置信息。
- 拖放区域的范围：你可以限制元素在特定的区域内拖放（比如不能超出屏幕的范围），这对创建滚动条十分有效。
- 自定义坐标系统：如果你不太喜欢CSS坐标系统，可以选择任意组合x/y坐标体系的映射。
- 自定义x和y坐标系统的转换：你可以让拖放元素使用非传统方式（比如在周围摆动或波动）进行移动。

使用 DOM-Drag 系统相对直白明了。一开始，你可以为元素绑定一个拖放处理函数（当然还可以指定其他额外的选项），并加上其他监听函数。代码清单 7-21 是使用 DOM-Drag 的一些例子。

代码清单7-21　在浏览器内使用DOM-Drag来模拟可拖放窗口

```
<html>
<head>
        <title>DOM-Drag - Draggable Window Demo</title>
        <script src="domdrag.js" type="text/javascript"></script>
        <script type="text/javascript">
        window.onload = function(){
            // 初始化DOM-Drag函数，使ID为'window'的元素可拖放
            Drag.init( document.getElementById("window") );
        };
        </script>
        <style>
        #window {
            border: 1px solid #DDD;
            border-top: 15px solid #DDD;
            width: 250px;
            height: 250px;
        }
        </style>
</head>
<body>
        <h1>Draggable Window Demo</h1>
        <div id="window">I am a draggable window, feel free to move me around!</div>
</body>
</html>
```

158

代码清单 7-22 是 DOM-Drag 库的带完整文档的一份副本。代码以一个独立的全局对象 Drag 存在，它的方法可以在初始化拖放的过程中调用。

代码清单7-22 完整存档的DOM-Drag库

```
var Drag = {

    // 被拖放的当前元素
    obj: null,

    // 拖放对象的初始化函数
    // o = 作为拖放处理函数的元素
    // oRoot = 被拖放的元素，如果不指定，则把处理函数作为拖放元素
    // minX, maxX, minY, maxY = 元素允许的坐标大小峰值
    // bSwapHorzRef = 切换水平坐标系统
    // bSwapVertRef = 切换垂直坐标系统
    // fxMapper, fyMapper =  映射x和y坐标的函数
    init: function(o, oRoot, minX, maxX, minY,
            maxY, bSwapHorzRef, bSwapVertRef, fXMapper, fYMapper) {

        // 监听拖放事件的开始
        o.onmousedown = Drag.start;

        // 获取使用中的坐标系统
        o.hmode = bSwapHorzRef ? false : true ;
        o.vmode = bSwapVertRef ? false : true ;

        // 获取作为拖放处理函数的元素
        o.root = oRoot && oRoot != null ? oRoot : o ;

        // 初始化指定的坐标系统
        if (o.hmode && isNaN(parseInt(o.root.style.left )))
            o.root.style.left   = "0px";
        if (o.vmode && isNaN(parseInt(o.root.style.top )))
            o.root.style.top    = "0px";
        if (!o.hmode && isNaN(parseInt(o.root.style.right )))
            o.root.style.right  = "0px";
        if (!o.vmode && isNaN(parseInt(o.root.style.bottom)))
            o.root.style.bottom = "0px";
        // 检查用户是否提供了 x/y 坐标的大小峰值
        o.minX = typeof minX != 'undefined' ? minX : null;
        o.minY = typeof minY != 'undefined' ? minY : null;
        o.maxX = typeof maxX != 'undefined' ? maxX : null;
        o.maxY = typeof maxY != 'undefined' ? maxY : null;

        // 检查任何指定的 x 和 y 坐标映射器
        o.xMapper = fXMapper ? fXMapper : null;
        o.yMapper = fYMapper ? fYMapper : null;

        // 为所有用户定义的函数加壳
        o.root.onDragStart = new Function();
        o.root.onDragEnd  = new Function();
        o.root.onDrag = new Function();
    },

    start: function(e) {
        // 获取拖放中的对象
```

159

```
        var o = Drag.obj = this;

        // 标准化事件对象
        e = Drag.fixE(e);

        // 获取当前的 x 和 y 坐标
        var y = parseInt(o.vmode ? o.root.style.top  : o.root.style.bottom);
        var x = parseInt(o.hmode ? o.root.style.left : o.root.style.right );

        // 在当前的 x 和 y 坐标上调用用户的函数
        o.root.onDragStart(x, y);

        // 记录鼠标的开始位置
        o.lastMouseX = e.clientX;
        o.lastMouseY = e.clientY;

        // 如果使用的是 CSS 坐标系统
        if (o.hmode) {
            // 设置坐标适当的大小峰值
            if (o.minX != null) o.minMouseX    = e.clientX - x + o.minX;
            if (o.maxX != null) o.maxMouseX    = o.minMouseX + o.maxX - o.minX;

        // 否则，我们使用的是传统的数学坐标系统
        } else {
            if (o.minX != null) o.maxMouseX = -o.minX + e.clientX + x;
            if (o.maxX != null) o.minMouseX = -o.maxX + e.clientX + x;
        }
        // 如果正在使用的是 CSS 坐标系统
        if (o.vmode) {
            // 设置坐标适当的大小峰值
            if (o.minY != null) o.minMouseY    = e.clientY - y + o.minY;
            if (o.maxY != null) o.maxMouseY    = o.minMouseY + o.maxY - o.minY;

        // 否则，我们使用的是传统的数学坐标系统
        } else {
            if (o.minY != null) o.maxMouseY = -o.minY + e.clientY + y;
            if (o.maxY != null) o.minMouseY = -o.maxY + e.clientY + y;
        }

        // 检查“拖放中”和“拖放结束”事件
        document.onmousemove = Drag.drag;
        document.onmouseup = Drag.end;

        return false;
    },

    // 在拖放事件过程中监听所有鼠标移动的函数
    drag: function(e) {
        // 标准化事件对象
        e = Drag.fixE(e);

        // 获取被拖放元素的引用
        var o = Drag.obj;
```

160

```
        // 获取窗口内鼠标的位置
        var ey = e.clientY;
        var ex = e.clientX;

        // 获取当前的x和y坐标
        var y = parseInt(o.vmode ? o.root.style.top  : o.root.style.bottom);
        var x = parseInt(o.hmode ? o.root.style.left : o.root.style.right );
        var nx, ny;

        // 如果设置了X的最小值，则确保不要把它传递进去
        if (o.minX != null) ex = o.hmode ?
            Math.max(ex, o.minMouseX) : Math.min(ex, o.maxMouseX);

        // 如果设置了X的最大值，则确保不要把它传递进去
        if (o.maxX != null) ex = o.hmode ?
            Math.min(ex, o.maxMouseX) : Math.max(ex, o.minMouseX);

        // 如果设置了Y的最小值，则确保不要把它传递进去
        if (o.minY != null) ey = o.vmode ?
            Math.max(ey, o.minMouseY) : Math.min(ey, o.maxMouseY);

        // 如果设置了Y的最大值，则确保不要把它传递进去
        if (o.maxY != null) ey = o.vmode ?
            Math.min(ey, o.maxMouseY) : Math.max(ey, o.minMouseY);

        // 得到经过转换的新的x和y坐标
        nx = x + ((ex - o.lastMouseX) * (o.hmode ? 1 : -1));
        ny = y + ((ey - o.lastMouseY) * (o.vmode ? 1 : -1));

        // 并使用x和y映射处理函数（假如有提供的话）来转换它们
        if (o.xMapper) nx = o.xMapper(y)
        else if (o.yMapper) ny = o.yMapper(x)

        // 为元素设置信的x和y坐标
        Drag.obj.root.style[o.hmode ? "left" : "right"] = nx + "px";
        Drag.obj.root.style[o.vmode ? "top" : "bottom"] = ny + "px";

        // 并记录鼠标的最后位置
        Drag.obj.lastMouseX = ex;
        Drag.obj.lastMouseY = ey;

        // 在当前x和y坐标上调用用户的onDrag函数
        Drag.obj.root.onDrag(nx, ny);

        return false;
    },

    // 处理拖放事件结束的函数
    end: function() {
        // 不再监听鼠标事件（由于拖放业已完成）
        document.onmousemove = null;
        document.onmouseup = null;

        // 拖放事件结束时，在元素的x和y坐标上调用我们特殊的onDragEnd函数
```

161

```
        Drag.obj.root.onDragEnd(
            parseInt(Drag.obj.root.style[Drag.obj.hmode ? "left" : "right"]),
            parseInt(Drag.obj.root.style[Drag.obj.vmode ? "top" : "bottom"]));
        // 不再监听拖放对象
        Drag.obj = null;
    },

    // 标准化事件对象的函数
    fixE: function(e) {
        // 如果不存在事件对象，则它是IE，所以使用IE的事件对象
        if (typeof e == 'undefined') e = window.event;
        // 如果layer属性未曾设置，则获取等效的offset属性的值
        if (typeof e.layerX == 'undefined') e.layerX = e.offsetX;
        if (typeof e.layerY == 'undefined') e.layerY = e.offsetY;

        return e;
    }
};
```

坦白地说，DOM-Drag 是众多 JavaScript 拖放库中较为简单的一个。但是，我特别喜欢它，因为它清晰的面向对象语法及其相对的简洁性。在下一节中我会讨论 Scriptaculous 库，它拥有精彩和强大的拖放实现，强烈推荐。

7.6　库

对于大部分乏味的 JavaScript 任务来说，你需要开发的一些效果或交互，说不定已经有现成的直接可用的了。下面会简要介绍 3 不同的库，它们提供不同的 DHTML 交互效果，作为开发者的你可以探索哪些对你是有帮助的。

7.6.1　moo.fx 和 jQuery

有两个轻量级的库非常擅长于处理简单的效果：moo.fx 和 jQuery。这两个库都提供基本的效果组合，结合使用可以创建高效而简洁的动画。关于它们的详情可以在各自的网站上找到。代码清单 7-23 是一些使用这两个库的基本例子。

代码清单7-23　使用moo.fx和jQuery的动画例子

```
// 隐藏元素逐渐展开，完成后又收回的简单动画
// moo.fx的实现方式

new fx.Height( "side", {
      duration: 1000,
      onComplete: function() {
          new fx.Height( "side", { duration: 1000 } ).hide();
      }
}).show();

// jQuery的实现方式
$("#side").slideDown( 1000, function(){
      $(this).slideUp( 1000 );
```

```
});
// 元素高度、宽度和透明度一起收缩的另一个简单动画，隐藏效果很酷

// moo.fx的实现方式
new fx.Combo( "body", {
    height: true,
    width: true,
    opacity: true
}).hide();

// jQuery的实现方式
$("#body").hide( "fast" );
```

163

或许你已经从例子中了解到，moo.fx 和 jQuery 让一些灵巧的动画更容易实现。两个项目都在各自网站上提供了大量丰富的例子，学习这些例子是了解 JavaScript 中简单动画的不错方式。

- moo.fx主页：http://moofx.mad4milk.net/。
- mootoolkit文档及例子：http://moofx.mad4milk.net/documentation/。
- jQuery主页：http://jquery.com/。
- jQuery效果文档及例子：http://jquery.com/docs/fx/。

7.6.2 Scriptaculous

如果 DHTML 库有王者的话，那么当属 Scriptaculous。基于流行的 Prototype 库，Scriptaculous 提供数千种不同的交互，从动画效果到交互行为（比如拖放）。大量的信息及例子可以在 Scriptaculous 网站上找到。

- 主页：http://script.aculo.us/。
- 文档：http://wiki.script.aculo.us/scriptaculous/。
- 演示：http://wiki.script.aculo.us/scriptaculous/show/Demos/。

Scriptaculous 提供的众多效果中，强大的同时能够最大程度保持简洁的，就是它的拖放实现。接下来看一些简单的例子。

1. 拖放重排序

Scriptaclous 难以置信的简单效果之一是列表的重排序。考虑到代码是如此的简洁（与 Ajax 功能的结合也是如此的简单，在其网站上有演示），对大部分 Web 开发者来说绝对是一个值得推荐的解决方案。代码清单 7-24 展示的例子是一个使用 Scriptaculous 库建立的简单重排序列表。

164

代码清单7-24 如何使用Scriptaculous中的拖放技术创建可重排序的列表

```
<html>
<head>
    <title>script.aculo.us - Drag and Drop Re-Ordering Demo</title>
    <script src="prototype.js" type="text/javascript"></script>
    <script src="scriptaculous.js" type="text/javascript"></script>
    <script src="effects.js" type="text/javascript"></script>
    <script src="dragdrop.js" type="text/javascript"></script>
    <script type="text/javascript">
    window.onload = function(){
```

```
        // 使id为'list'的元素变成一个可重排序的列表
        Sortable.create('list');
    };
    </script>
</head>
<body>
    <h1>Drag and Drop Re-Ordering</h1>
    <p>Drag and drop an item to re-order it.</p>
    <ul id="list">
        <li>Item number 1</li>
        <li>Item number 2</li>
        <li>Item number 3</li>
        <li>Item number 4</li>
        <li>Item number 5</li>
        <li>Item number 6</li>
    </ul>
</body>
</html>
```

希望这能够证明这个库的强大。如还不够，你可以继续看看下一个例子，它创建一个可输入的滑块。

2. 滑块输入

Scriptaculous提供了很多可以解决常见界面开发问题的控制器。大部分拖放库相对容易实现的一个控制器是滑块输入（移动滑动条来获取数值），Scriptaculous也不例外，如代码清单7-25所示。

165

代码清单7-25　使用Scriptaculous的滑块输入创建一个另类的表单输入框

```
<html>
<head>
    <title>script.aculo.us - Slider Input Demo</title>
    <script src="prototype.js" type="text/javascript"></script>
    <script src="scriptaculous.js" type="text/javascript"></script>
    <script src="effects.js" type="text/javascript"></script>
    <script src="dragdrop.js" type="text/javascript"></script>
    <script src="controls.js" type="text/javascript"></script>
    <script type="text/javascript">
    window.onload = function(){
        // 把ID为ageHandle的元素变成可拖动的滑块处理函数，而ID为ageBar的变成滑块
        new Control.Slider( 'ageHandle', 'ageBar', {
            // 一旦滑块移动，或者完成移动，调用updateAge函数
            onSlide: updateAge
        });

        // 处理滑块上发生的任何移动
        function updateAge(v) {
            // 当滑块更新，更新表示用户年龄的元素的值
            $('age').value = Math.floor( v * 100 );
        }
    };
    </script>
</head>
```

```
<body>
      <h1>Slider Input Demo</h1>

      <form action="" method="POST">
          <p>How old are you? <input type="text" name="age" id="age" /></p>

          <div id="ageBar" style="width:200px; background: #000; height:5px;">
              <div id="ageHandle" style="width:5px; height:10px;
                  background: #000; cursor:move;"></div>
          </div>

          <input type="submit" value="Submit Age"/>
      </form>
</body>
</html>
```

166

强烈建议你在编写下一个交互应用程序之前深入了解一些 DHTML 库，因为库作者开发某一方面的交互所花费的大量时间和努力，可能比你开发整个应用程序的时间还要多。手中掌握了某些库，可以显著地提高开发效率。

7.7 小结

利用动态交互的能力，是在 Web 应用程序中为用户提供更高速度和更佳的可用性的强大方法。此外，如果使用流行的库可以极大地减少开发时间。第 8 章中你将运用所有这些所学到的交互技术，创建一个高度可用的、交互的应用程序。

在本章中，你已经了解 JavaScript 和 CSS 相互结合的各种不同的技术。掌握了这些技术，就可以创建令人印象深刻的动画和动态用户交互。

你应该记住，为页面增加任何形式的动态交互都有疏远某部分用户的潜在可能。应该时刻在意的是，当 JavaScript 或者 CSS 禁用的时候，Web 应用程序应该具备最基本可用性。开发预留退路（degrading gracefully）的 Web 应用程序应该是每一位 JavaScript 程序员的理想。

167

第8章

改进表单

表单作为接受用户输入数据的一种手段，对 Web 开发者非常有用。但是，用户能做什么、能输入什么数据和表单的可用性如何等情况，却有不少的限制。

构建一个基于语义设计的表单后，就到增加一些能为用户提供反馈的 JavaScript 的开发阶段了。理解表单某些效果的前因后果，用户就可以快捷地使用表单，也因此拥有更好的用户体验。

在本章中，我们将探索客户端表单的相关基本操作——验证并在一定程度上反馈验证的结果给用户，基于分离式脚本编程技术。接着探索全面改进表单的一些可行方法。这两种技术结合起来可以让表单的可用性显著提升，从而用户也更乐意于填写表单。

8.1 表单验证

增加客户端的表单验证可以为用户提供更快的体验，但决不能忽视的是，客户端表单验证永远不应该取代服务器端的验证，而只能是辅助和增强。因此，为页面增加客户端表单验证是你刚学过的分离式脚本编程技术的不错案例。

在开始任何形式的表单脚本编程之前，你应该已经制作好表单并确保它完全能按要求工作（比如，验证用户的输入、反馈恰当的错误信息等）。在本章中，你使用的都是语义化的 XHTML 表单。在表单内，所有<input>元素都精细地分好类（比如，type 为 text 的元素其 class 也为 text），并与正确的 label 一起被包含在合适的 fieldset 内。所有这些都可参考代码清单 8-1。

代码清单8-1　一个简单的XHTML表单，你将要使用JavaScript改进它

```
<html>
<head>
      <title>Simple Form</title>
</head>
<body>
<form action="" method="POST">
    <fieldset class="login">
       <legend>Login Information</legend>
       <label for="username" class="hover">Username</label>
       <input type="text" id="username" class="required text"/>

       <label for="password" class="hover">Password</label>
       <input type="password" id="password" class="required text"/>
```

```
</fieldset>
<fieldset>
    <legend>Personal Information</legend>

    <label for="name">Name</label>
    <input type="text" id="name" class="required text"/><br/>

    <label for="email">Email</label>
    <input type="text" id="email" class="required email text"/><br/>

    <label for="date">Date</label>
    <input type="text" id="date" class="required date text"/><br/>

    <label for="url">Website</label>
    <input type="text" id="url" class="url text" value="http://"/><br/>

    <label for="phone">Phone</label>
    <input type="text" id="phone" class="phone text"/><br/>

    <label for="age">Over 13?</label>
    <input type="checkbox" id="age" name="age" value="yes"/><br/>

    <input type="submit" value="Submit Form" class="submit"/>
  </fieldset>
</form>
</body>
</html>
```

下一步是赋予这个表单一些基本的 CSS 样式，让它看起来更美观。这可帮助你显示更加友好的错误信息和反馈，在后面的章节会讨论到。这个表单用到的 CSS 见代码清单 8-2。 170

代码清单8-2 用以提高表单视觉质量的CSS样式

```
form {
    font-family: Arial;
    font-size: 14px;
    width: 300px;
}

fieldset {
    border: 1px solid #CCC;
    margin-bottom: 10px;
}

fieldset.login input {
    width: 125px;
}

legend {
    font-weight: bold;
    font-size: 1.1em;
}

label {
```

```
        display: block;
        width: 60px;
        text-align: right;
        float: left;
        padding-right: 10px;
        margin: 5px 0;
}

input {
        margin: 5px 0;
}

input.text {
        padding: 0 0 0 3px;
        width: 172px;
}

input.submit {
        margin: 15px 0 0 70px;
}
```

171

图 8-1 是该表单的屏幕截图，可以让你对这个表单（已经为 JavaScript 行为层做好准备）有个初步的印象。

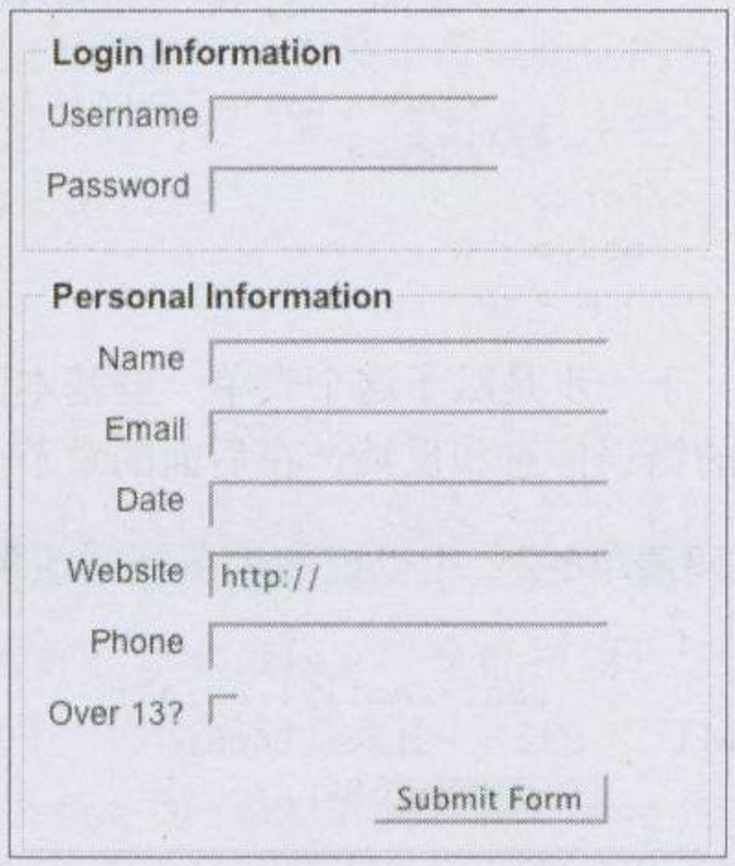


图8-1　你将为之增加JavaScript行为的表单，经过样式化后的屏幕截图

有了这么一个漂亮的表单，现在就可以在更深程度上探索客户端的验证了。用在表单上验证技术有好几种，它们都需要保证用户输入的数据是服务器端软件所预期的。

提供客户端验证的主要优点在于，用户可以有一个校验他们输入的实时反馈，从而全面提高表单的输入体验。但这并不等于，实现了客户端的表单验证就可以忽视或者删除服务器端的验证。应该继续测试禁止 JavaScript 情况下的表单，确保不使用 JavaScript 的用户可继续拥有其他的可用性体验。

在这一部分中，你将会看到校验各种不同输入元素的特定代码，以确保它们包含符合表单要求的特定数据。这些验证程序看起来相对独立，但结合起来就可以提供完整的验证和测试系列，我们会稍后讨论。

8.1.1　必填字段

字段验证中最重要的可能就是必填字段（表示该条目用是户必须填写的）了。通常，这个必要条件可以简化为检查字段的值是否为空。但有时候，字段可能有一个默认的值，这就要求在注意这种可能性的同时要检查用户是否至少改变了字段默认值。这两种检查包含了表单字段的主要形式，包括<input type="text">，<select>和<textarea>。

172

尽管如此，当你试图检查用户是否修改了必填的复选框或者单选框时，还是可能产生问题。要巧妙地解决这个问题，你需要找出所有拥有相同 name（这是字段元素关联的纽带）的字段，然后检查用户是否选择了其中的一个。

代码清单 8-3 是一个检查必填字段的例子。

代码清单8-3 检查一个必填字段是否被修改（包括复选框和单选框）

```
// 检查输入元素是否键入了信息的通用函数
function checkRequired( elem ) {
      if ( elem.type == "checkbox" || elem.type == "radio" )
           return getInputsByName( elem.name ).numChecked;
      else
           return elem.value.length > 0 && elem.value != elem.defaultValue;
}

// 找出指定name的所有input元素（对查找以及处理复选框或单选框十分有用）
function getInputsByName( name ) {
      // 匹配的input元素的数组
      var results = [];
      // 追踪被选中元素的数量
      results.numChecked = 0;

      // 找出文档中的所有input元素
      var input = document.getElementsByTagName("input");
      for ( var i = 0; i < input.length; i++ ) {
           // 找出所有指定name的字段
           if ( input[i].name == name ) {
                // 保存结果，稍后会返回
                results.push( input[i] );
                // 记录被选中字段的数量
                if ( input[i].checked )
                     results.numChecked++;
           }
      }

      // 返回匹配的字段集合
      return results;
}
// 等待文档完成加载
window.onload = function()
    // 获得表单并监听提交事件
   document.getElementsByTagName("form")[0].onsubmit = function(){

        // 获取需检查的input元素
        var elem = document.getElementById("age");

        // 确保年龄的必填字段已经被选中
        if ( ! checkRequired( elem ) ) {
             // 否则显示错误并阻止表单提交
             alert( "Required field is empty - " +
                  "you must be over 13 to use this site." );
             return false;
```

173

```
        }

        // 获取需检查的input元素
        var elem = document.getElementById("name");

        // 确保名字字段有文本输入
        if ( ! checkRequired( elem ) ) {
          // 否则显示错误并阻止表单提交
          alert( "Required field is empty - please provide your name." );
          return false;
        }

    };
};
```

检查必填字段后，还需要确保字段的内容符合要求。接下来，你会看到如何验证字段的内容。

8.1.2　模式匹配

验证输入元素（尤其是文本字段）的第二大组件是模式匹配，它检验字段的内容是否符合要求。

使用以下技术的要点在于明白无误地定义字段的必须条件，否则，你可能会让用户产生困惑。比如日期格式会在特定文化的差异，甚至是不同规范的情况下而差别巨大。

在这一部分中，你会看到几种不同的验证字段内容的技术，包括电子邮件地址、URL、电话
174 号码和日期。

1. 电子邮件

要求填写电子邮件地址无疑是Web表单中再普通不过的需求了，因为它是鉴定和交流的普遍手段。但要完全检查电子邮件地址的正确性（取决于它的规则）却非常复杂。你可以只概括性地检查所有可能的输入情况。代码清单8-4展示了一个检查输入字段是否包含电子邮件地址的例子。

代码清单8-4　检查指定的input元素是否包含电子邮件地址

```
// 检查input元素内容是否符合emial地址要求的通用函数
function checkEmail( elem ) {
      // 确保输入的内容是正确的email地址
      return elem.value == '' ||
          /^[a-z0-9_+.-]+\@([a-z0-9-]+\.)+[a-z0-9]{2,4}$/i.test( elem.value );
}

// 获取需要检查的input元素
var elem = document.getElementById("email");

// 检查这个字段是否正确
if ( ! checkEmail( elem ) ) {
      alert( "Field is not an email address." );
}
```

2. URL

表单（和其他的网络相关区域）的一个常见需求是要求用户输入网站的 URL。与电子邮件

地址一样，URL 又是一个难以实现完全符合规范定义的例子，但只需实现规范中的一小部分即可达到目的。实际上，你只需检查基于 http 或 https 的 Web 地址（当然，即使有所不同也可方便修改）。此外，URL 字段通常会有 http://这样的字符串打头，在检查表单时必须考虑这种情形。代码清单 8-5 是检查表单中 URL 正确性的一个例子。

代码清单8-5 检查input元素是否包含URL

```
// 检查input元素是否包含URL的通用函数
function checkURL( elem ) {
    // 确保有文本的键入，而且不是默认的http://文本
    return elem.value == '' || !elem.value == 'http://' ||
        // 确保它是一个正确的URL
        /^https?:\/\/([a-z0-9-]+\.)+[a-z0-9]{2,4}.*$/.test( elem.value );
}

// 获取需要检查的inpu元素
var elem = document.getElementById("url");

// 检查它是否是一个正确的URL
if ( ! checkURL( elem ) ) {
    alert( "Field does not contain a URL." );
}
```

3. 电话号码

接下来你会看到两个的字段是：电话号码和日期。取决于你的所在地，它们有着不同的情形。为简化起见，我将使用美国式的电话号码（和日期）。当然，改变它们的规则以适应另一个国家或地区并不是十分困难。

记住了这些，你还要试着处理电话号码字段中的一些特别之处。电话号码可以以不同的形式书写，所以你需要允许这些形式的输入（如 123-456-7890 或(123)456-7890）。

这样一来你不仅需要验证数字本身，还要验证这种特殊的格式。你可以简单地逆向检查电话号码字段的值，看它是否被分为包含 3 个数字的两部分以及一个包含 4 个数字的部分，而忽略包住电话号码的任何多余的信息。

执行这个验证并强行检查字段值的代码如代码清单 8-6 所示。

代码清单8-6 检查字段是否包含电话号码

```
// 检查input元素是否包含电话号码的通用函数
function checkPhone( elem ) {
    // 检查是否符合电话号码的要求
    var m = /(\d{3}).*(\d{3}).*(\d{4})/.exec( elem.value );

    // 如果是，可能也只是表面正确而已——强行检查它的格式是否符合我们的要求：(123) 456-7890
    if ( m !== null )
        elem.value = "(" + m[1] + ") " + m[2] + "-" + m[3];

    return elem.value == '' || m !== null;
}
// 获取需要检查的inpu元素
```

```
var elem = document.getElementById("phone");

// 检查这个字段是否包含正确的电话号码
if ( ! checkPhone( elem ) ) {
    alert( "Field does not contain a phone number." );
}
```

4. 日期

最后要探索的是日期的验证。再次，你会看到美国式的日期书写格式（MM/DD/YYYY）。就像电话号码或者其他由国别决定的不同字段，如果有必要，你可以方便修改验证的正则表达式以满足本土化需求。使用如代码清单 8-7 所示的具体验证函数，你就可以验证日期字段的内容了。

代码清单8-7　检查字段是否包含日期

```
//检查input元素是否包含日期的通用函数
function checkDate( elem ) {
      // 确保输入了内容并检查是否符合MM/DD/YYYY的时间格式
      return !elem.value || /^\d{2}\/\d{2}\/\d{2,4}$/.test(elem.value);
}

// 获取需要检查的inpu元素
var elem = document.getElementById("date");

// 检查这个字段是否包含正确的日期
if ( ! checkDate( elem ) ) {
      alert( "Field is not a date." );
}
```

8.1.3　规则集合

运用上一部分所述的各种不同验证函数，现在你可以构建一个用以处理不同验证的通用技术架构了。需要注意的是，所有的测试都分别需要通用的名称和语义化的错误信息。完整的规则集合数据结构可以参考代码清单 8-8。

177

代码清单8-8　构建验证引擎的规则和错误描述的标准集合

```
var errMsg = {
      // 检查特定字段是否为必填
      required: {
          msg: "This field is required.",
          test: function(obj,load) {
                // 确保字段尚未有内容输入，并在页面加载时不作检查
                //（在加载时显示“必填字段”可能会让用户感到厌烦）
                return obj.value.length > 0 || load || obj.value == obj.defaultValue;
          }
      },

      // 确保字段内容是正确的email地址
      email: {
          msg: "Not a valid email address.",
          test: function(obj) {
```

```
            // 确保有内容的输入并符合email地址的格式
            return !obj.value ||
                /^[a-z0-9_+.-]+\@([a-z0-9-]+\.)+[a-z0-9]{2,4}$/i.test( obj.value );
        }
    },

    // 确保字段内容是电话号码并将其自动格式化
    phone: {
        msg: "Not a valid phone number.",
        test: function(obj) {
            // 检查它是否符合电话号码的要求
            var m = /(\d{3}).*(\d{3}).*(\d{4})/.exec( obj.value );

            // 如果是，可能也只是表面正确而已——
            // 强行检查它的格式是否符合我们的要求：(123) 456-7890
            if ( m ) obj.value = "(" + m[1] + ") " + m[2] + "-" + m[3];

            return !obj.value || m;
        }
    },

    // 确保字段内容符合MM/DD/YYYY的时间格式
    date: {
        msg: "Not a valid date.",
        test: function(obj) {
            // 确保输入了内容并检查是否符合MM/DD/YYYY的时间格式
            return !obj.value || /^\d{2}\/\d{2}\/\d{2,4}$/.test(obj.value);
        }
    },

    // 确保字段内容是一个正确的URL
    url: {
        msg: "Not a valid URL.",
        test: function(obj) {
            / 确保有文本的键入，而且不是默认的http://文本
            return !obj.value || obj.value == 'http://' ||
                // 确保它是一个正确的URL
                /^https?:\/\/([a-z0-9-]+\.)+[a-z0-9]{2,4}.*$/.test( obj.value );
        }
    }
};
```

使用这个新的规则集合的数据结构，现在你就可以编写通用的、牢固的表单验证工具了，并可以显示错误信息，这正是接下来我们要讨论的。

8.2 显示错误信息

如果说表单验证程序的实现并不是很容易的话，那么显示错误的上下文信息，以帮助用户更好地填写表单，更是一个不小的挑战。你将用到前面所学习的技术，来构建一个完整的验证和信息显示系统。而在这一部分中你会看到表单的验证如何与信息显示的综合使用，并在适当的时候触发，让用户更容易理解。

8.2.1　验证

使用新的数据结构就可以构建一对牢固的、可扩展的函数，它们可用来校验整个表单或者单个字段并依此显示上下文错误信息。

实现表单动态验证的技术有好几种。第一种是浏览器所提供，并且是HTML DOM规范的组成部分。所有的<form>元素（在DOM中）都有一个被称为elements的属性，这个属性是一个
179 包含表单所有字段的数组，使用这个数组就可以轻松地遍历所有可能的字段，并检查错误。

第二种技术也比较重要，它为所有字段附加class以触发不同的验证规则。比如，有class为required（必填）的字段要求必须填写。所有这些class都应该匹配由代码清单8-8所提供的规则集合。

使用这两种技术就可以构建这两个通用的函数了，它们能够验证整个表单和单个字段（两者在完整的功能性验证场合中都需用到）。这两个函数如代码清单8-9所示。

代码清单8-9　执行表单验证和触发错误信息显示的函数

```
// 验证表单所有字段的函数
// form参数应是一个表单元素的引用
// load参数应该是一个布尔值，用以判别验证函数在页面加载时执行还是动态执行
function validateForm( form, load ) {
    var valid = true;

    // 遍历表单的所有字段元素
    // form.elements是表单所有字段的一个数组
    for ( var i = 0; i < form.elements.length; i++ ) {

        // 先隐藏任何错误信息，以防不意的显示
        hideErrors( form.elements[i] );

        // 检查字段是否包含正确的内容
        if ( ! validateField( form.elements[i], load ) )
            valid = false;

    }

    // 如果字段是不正确的内容返回false，反之返回true
    return valid;
}

// 验证单个字段的内容
function validateField( elem, load ) {
    var errors = [];

    // 遍历所有可能的验证技术
    for ( var name in errMsg ) {
        // 查看字段是否有错误类型指定的class
180      var re = new RegExp("(^|\\s)" + name + "(\\s|$)");
        // 检查元素是否带有该class并把它传递给验证函数
        if ( re.test( elem.className ) && !errMsg[name].test( elem, load ) )
            // 如果没有通过验证，把错误信息增加到列表中
```

```
            errors.push( errMsg[name].msg );
        }

        // 如果存在错误信息，则显示出来
        if ( errors.length )
            showErrors( elem, errors );

        // 如果字段始终没有得到验证，返回false
        return errors.length > 0;
}
```

或许你已经注意到这段代码尚缺乏两个函数，它们跟隐藏和显示验证的错误信息息息相关。基于你要显示错误信息的具体需求，你可能需要对这两个函数稍作调整。在这个特定的表单上，我决定在表单的每一个字段后显示错误信息。这两个函数如代码清单 8-10 所示。

代码清单8-10 显示和隐藏相应字段的错误信息

```
// 隐藏当前正显示的任何错误信息
function hideErrors( elem ) {
        // 获取当前字段的下一个元素
        var next = elem.nextSibling;

        // 如果下一个元素是ul并有class为errors
        if ( next && next.nodeName == "UL" && next.className == "errors" )
            // 删掉它（这是我们“隐藏”的含义）
            elem.parenttNode.removeChild( next );
}

// 显示表单内特定字段的错误信息
function showErrors( elem, errors ) {
        // 获取当前字段的下一个元素
        var next = elem.nextSibling;

        // 如果该字段不是我们指定的包含错误的容器
        if ( next && ( next.nodeName != "UL" || next.className != "errors" ) ) {
            // 我们得生成一个
            next = document.createElement( "ul" );
            next.className = "errors";
            // 并在DOM中把它插入到恰当的地方
            elem.paretNode.insertBefore( next, elem.nextSibling );
        }

        // 现在有了一个包含错误的容器引用，我们可以遍历所有的错误信息了
        for ( var i = 0; i < errors.length; i++ ) {
            // 为每一条错误信息创建新的li包裹器
            var li = document.createElement( "li" );
            li.innerHTML = errors[i];

            // 并插入到DOM中
            next.appendChild( li );
        }
}
```

既然已经有了所有的 JavaScript 代码，剩下的步骤就是增加一些样式使其美观了。CSS 代码如代码清单 8-11 所示。

代码清单8-11　使错误信息可读性更佳的CSS

```
ul.errors {
        list-style: none;
        background: #FFCECE;
        padding: 3px;
        margin: 3px 0 3px 70px;
        font-size: 0.9em;
        width: 165px;
}
```

最后，各方面终于凑齐了，你可以看到如图 8-2 所示的最终效果，它是 JavaScript 和样式化的综合结果（如果你对事件的监听感到为难的话，稍后会有解释）。

现在你已经详细地了解如何验证表单（以及它里面的字段）并显示基于任意字段验证的错误信息，但还需要决定在哪个合适的时候运行验证程序。所有的字段同时检验并不总是最好的，通常增量式检验才是最恰当的。接下来，我们会解释在
182 各个合适的不同运行时间执行验证的优点。

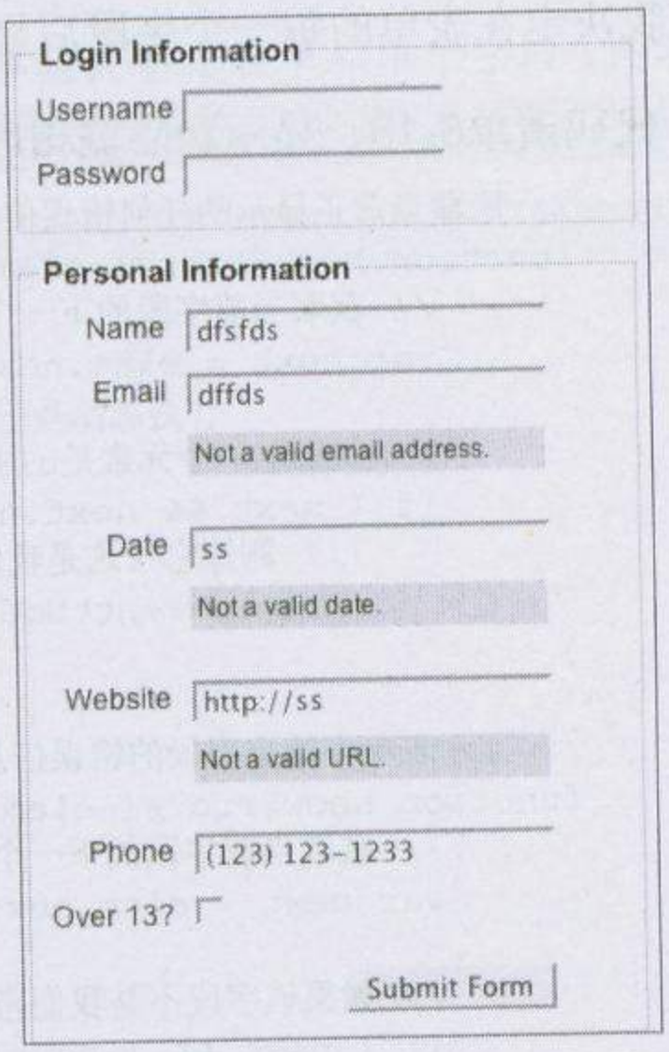


图8-2　样式化和脚本化表单后正确与不正确输入的例子

8.2.2　何时验证

表单验证比较麻烦的一点是决定何时才是显示错误信息的合适时机。表单（或字段）有 3 个不同的验证时间点：表单提交时、字段改变时和页面加载时。它们都有各自的优缺点，稍后会分别讨论。使用上一部分提到的函数，这些过程都变得更简单和更容易理解。

1. 表单提交时验证

在表单提交时验证是最常用的技术，因为它跟普通的表单验证技术最为接近。为监听表单的提交，你必须绑定一个事件处理函数，来等待用户完成表单并点击提交按钮（或敲击回车键）。但是用户是否在所有的字段都输入了内容并不是先决条件，一旦表单提交它就由规则集合所指定的各种规则去检验。如果任何一个字段不通过，表单会停止提交（这通过阻止提交事件处理函数的默认行为做到），而用户也会收到错误信息的提示，也有机会去更正错误。实现这个技术的代
183 码如代码清单 8-12 所示。

代码清单8-12　在表单提交的时候运行表单验证函数

```
function watchForm( form ) {
        // 监听表单的提交事件
        addEvent( form, 'submit', function(){
```

```
            // 确保表单内容通过验证
            return validateForm( form );

        });
}

// 获取页面的第一个表单
var form = document.getElementsByTagName( "form" )[0];

// 并在它提交的时候作出验证
watchForm( form );
```

2. 在字段改变时验证

可用在表单验证的另一种技术是监听表单内单个字段的变化。虽然可以通过 keypress 事件达到，但往往会导致意想不到的结果。在一个字段里，每次敲击键盘都触发错误检查的话会让用户感到困惑和厌烦。在这种情况下，他们一开始输入电子邮件地址就看到错误信息提示地址不正确。但不应该是这样的，因为他们需要继续在这个字段输入内容。一般而言，这种方法并不值得推荐，它确实不是一种好的用户体验。

检查字段变化的第二种方法是直到用户离开字段才验证（这样用户可能已经完成了信息的填写）。这种情况下的验证提供的用户体验更为平滑，因为用户有充分的时间来完整填写所需信息，同时仍能较快地收到验证的错误信息。

实现这个技术的例子如代码清单 8-13 所示。

代码清单8-13　在运行任何字段验证函数之前监听字段的变化

```
function watchFields( form ) {
        // 遍历表单内的所有字段
        for ( var i = 0; i < form.elements.length; i++ ) {

            // 并绑定'change'事件处理函数（它监听input元素的失焦）
            addEvent( form.elements[i], 'change', function(){
                // 一旦失去焦点，重验证该字段
                return validateField( this );
            });
        }
}

// 定位到页面的第一个表单
var form = document.getElementsByTagName( "form" )[0];

// 监听表单所有字段的变化
watchFields( form );
```

3. 在页面加载时验证

在页面加载时验证并不像前两者那么有必要，但在某些边缘场合也是非常重要的。假设用户在表单中输入信息后重新载入（或是由浏览器或应用程序本身预先填充了信息），在这些预加载的信息中有可能会触发错误。这个特定的技术在页面加载时运行表单验证程序，以验证已经存在的数据。这给予用户迅速处理错误的机会，而不用等到表单提交时才作出验证。

在页面加载时执行验证所需要的代码例子如代码清单 8-14 所示。

代码清单8-14　在页面加载时执行表单验证

```
addEvent( window, "load", function() {
        // 获取页面的所有表单
        var forms = document.getElementsByTagName("form");

        // 遍历所有的表单
        for ( var i = 0; i < forms.length; i++ ) {

            // 逐一验证，记得设置参数'load'为ture，它可以防止显示某些不必要的错误
            validateForm( forms[i], true );

        }
});
```

学习了所有不同的表单验证、显示错误信息的方式甚至是执行验证的恰当时机，你已经达到一个很高的目标——完成了客户端的表单验证。跨越这些障碍后，你可以进一步探索额外的一些
185 技术了，使用这些技术可以提升整个表单和特定类型字段的可用性。

8.3　可用性的提升

表单作为页面中最为常用的元素之一，提升它们可用性的结果只能是有益于用户。在这一部分中，我继续带领你探索两种常用的技术，它们常用来提高表单总体的可用性。

此外，你也有机会尝试使用 JavaScript 库来简化乏味无趣的 DOM 遍历和修改，提高可用性的操作也变得更容易。在这两种技术中我都选择使用 jQuery JavaScript 库（http://jquery.com），它特别擅长于 DOM 的遍历和修改。

8.3.1　悬停的说明

我们要讨论的第一条提高可用性的技术是把说明定位（悬停）在相关的字段内，并在字段得到焦点后隐藏。这个技术的目的有两种。第一，它清晰地给用户传达这个特定的字段需要填写什么内容；第二，它有助于节省字段及其说明所占据的物理空间。

在原有的表单上，你要为 username 和 password 两个字段增加新的悬停说明，产生如图 8-3 所示的结果。

但实现这个具体效果的 JavaScript 代码相当复杂。为了达到无缝结合需要了解很多细节。让我们来看看实现最终效果所需的部分细节。

第一，为了把说明定位于输入元素的顶端，必须先把 label 和 input 元素都用一个 div 包裹起来。可以用这个 div

图8-3　在username和password字段内使用悬停说明

把说明绝对定位在字段的顶端上。

第二，必须让说明在字段得到或失去焦点时恰如其分地隐藏（或显示）。此外，当用户离开字段后，需要检查字段是否输入了值。如果是，就不必再把说明显示出来了。 186

第三，如果字段本身带有默认值的，要保证说明不要显示出来。否则，用户会看到混乱的文本。

记住了这些，让我们来看看实现表单内悬停说明效果的所需代码，如代码清单 8-15 所示。

代码清单8-15 使用jQuery JavaScript库实现字段中的悬停说明

```
// 获取所有在class为hover的label（说明）后的input元素
$("label.hover+input")

    // 为input元素包裹一个div（class为hover-wrap），
    // HTML大致如此：
    // <div class='hover-wrap'><input type="text" .../></div>
    .wrap("<div class='hover-wrap'></div>")

    // 当input元素得到焦点（无论是通过鼠标的点击还是通过键盘的敲击），隐藏label
    .focus(function(){
        $(this).prev().hide();
    })

    // 当用户离开input元素（并且没有输入文本）时，再次显示label
    .blur(function(){
        if ( !this.value ) $(this).prev().show()
    })

    // 遍历所有的input元素
    .each(function(){
        // 把label移动到<div class='hover-wrap'></div>内
        $(this).before( $(this).parent().prev() );

        // 如果表单带有默认值，确保label会自动隐藏
        if ( this.value ) $(this).prev().hide();
    });
```

但是这段独立的 JavaScript 还不足够实现具体的需求，你仍要增加 CSS 的样式化，以保证能把说明和字段放到恰当的位置上，如代码清单 8-16 所示。

代码清单8-16 CSS的样式化，以保证说明能够定位于相关的字段之上

```
div.hover-wrap {
      position: relative;
      display: inline;
}

div.hover-wrap input.invalid {
    border: 2px solid red;
}

div.hover-wrap ul.errors {
```

187

```
    display: none;
}

div.hover-wrap label.hover {
    position: absolute;
    top: -0.7em;
    left: 5px;
    color: #666;
}
```

不用太困难，你已经有效地改进了表单的可用性。使用这个技术，在节省屏幕空间的同时，你仍然能够给用户恰当的引导。这不失为一个双赢的方案。

8.3.2　标记必填字段

第二条技术是使用视觉化的提示标记必填字段。比较普遍的做法是使用红色星号标记必填字段，大部分的 Web 开发者都在他们的网站上采用。但是，增加额外的内容以引进星号是相当不语义的，所以并不鼓励这样做[①]。相反，这是一个使用 JavaScript 增加可视化提示的好机会。这个技术的一个例子如图 8-4 所示。

188

为必填字段加上提示还需要注意的一个方面是，需要增加特定的可以引导用户的帮助文本。你可以使用 `title` 特性为用户提供一条提示，正确解释红色星号的含义，因为某些用户并不熟悉*号的意义。好了，实现这个改进相当简单，如代码清单 8-17 所示。

图8-4　为表单的必填字段增加上下文星号的结果

代码清单8-17　使用jQuery JavaScript库为必填字段的label增加上下文*号和帮助信息

```
// 获取所有有必填记号的字段
$("input.required")
        // 然后定位到前一个label
        .prev("label")

        // 改变label的光标让它更有用
        .css("cursor", "help")

        // 当用户悬停到*号上，显示解释
        .title( errMsg.required )

        // 最后，在label后增加*，表示这是必填的
        .append(" <span class='required'>*</span>");
```

为显示恰当的样式，你需要标红这个新提示，如代码清单 8-18 所示。

① 如果没有JavaScript，则无法提示必填字段，因此这种说法有待商榷。——译者注

代码清单8-18 为*号增加样式

```
label span.required {
      color: red;
}
```

增加可视化提示和使用悬停说明的结合有效改进了可用性，而且是通过分离式的 JavaScript 实现的。在一些特定应用程序内，会找到很多表单和字段的改进例子，而它们只需要简单的 JavaScript 就能实现。

8.4 小结

Web 应用程序中有这么多能让表单陷入困境的各种情况，对此有了更好理解，再为网站增加一些简单的 JavaScript，将会全面提升表单的可用性。在本章中，我们的学习成果如图 8-5 所示。 189

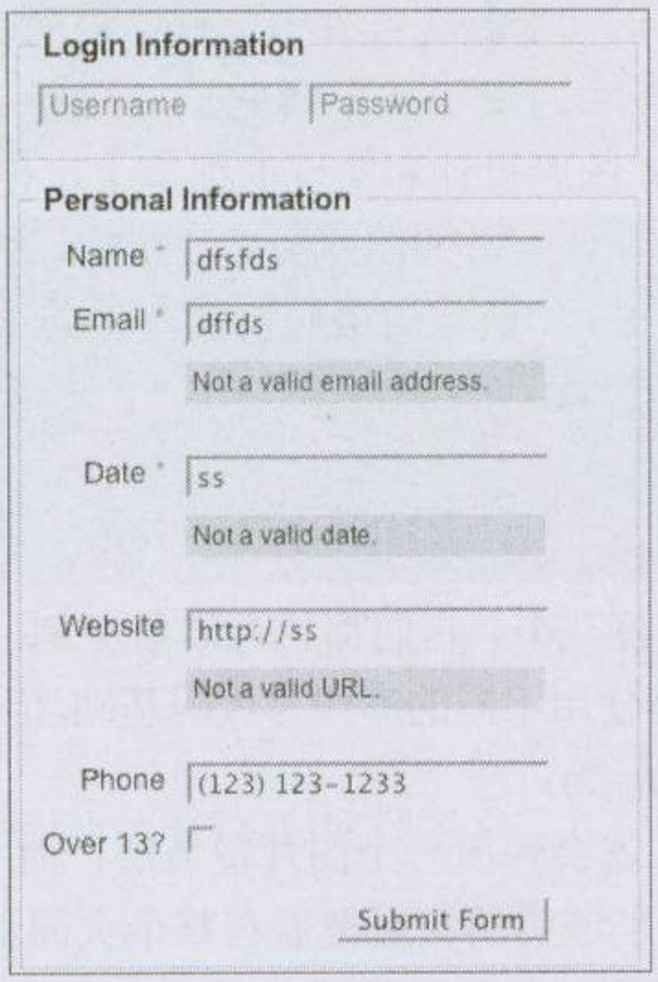


图8-5 JavaScript改进表单的最终结果

在本章的开始，讲述如何在很大程度上准确地实现客户端验证，从而为用户提供最佳的体验。讲解了如何创建一系列验证规则的集合、在恰当的时机验证表单字段以及为用户显示有用的错误信息。同时，通过悬停说明和标记必填字段的例子，你也看到了一些提升表单可用性的技术。

希望你能将所有这些介绍过的技术结合起来，可以在即将开发的表单中大显身手。 190

第9章

制作图库

DOM操作、遍历以及动态CSS操作能为网站的最终用户创建更敏捷的体验，而在比较充分利用这些体验的应用程序中最典型的应该是图库了，可以用来查看和浏览图片。浏览器越来越强大，动态脚本和功能也日臻完善，而这种提升使得一些高质量的图库相继面世。

在本章中，你会看到部分图库的演示并了解它们的独特之处，然后使用动态的、分离式JavaScript制作你自己的图库。在图库设计和实现的细节上，你会碰到一些问题。我们期望实现的最终成果是一个可以简单融入任何网站的图库脚本。此外，这是利用在第5章和第7章所讲解的有关DOM、JavaScript以及CSS的函数的最好机会，将其结合起来就能开发无缝并易于理解的代码。

9.1 图库示例

精彩现代的图库脚本已经有好几个，它们都让人印象深刻，它们易于使用并且完全是分离式的。本节将要讨论的两个脚本虽然使用不同的库作为代码基础，但它们的具体视觉效果都非常相似。

下面总结了这两个图库示例行为：

- 当点击图库中的图片时，就会弹出一个图片覆盖层，而不是直接把用户带到实际图片上。
- 图片覆盖层显示时，一个半透明的灰层覆盖在整个页面上（模糊化所有在它之下的东西）。
- 图片覆盖层会有当前显示图片的某种形式的说明。
- 在图库内会有一些从图片到图片的导航方法。

接下来看看这两个相当流行的图库，分别是Lightbox和ThickBox。

9.1.1 Lightbox

Lightbox是“新式”DOM图库的第一位吃螃蟹者，它的发布刺激了其他数种类似图库的发展。

这个图库是从零开始发展起来的（没有使用特定的JavaScript库），也因此能适应一些不同的库（这样可以降低它的代码总量）。关于这个脚本的更多信息可以从http://www.huddletogether.com/projects/lightbox/中找到，而http: //particletree.com/features/lightbox-gone-wild/提供了基于Prototype JavaScript库的Lightbox的相关信息。

图9-1展示了运行中的Lightbox图库的屏幕截图，从中可以看到它独特的半透明覆盖层和居中的图片。

图9-1 在图库内显示一张图片的Lightbox

Lightbox 以完全分离式的方法来运行。要使用它，你只需在 HTML 文件的头部包含脚本，并修改希望使用 Lightbox 来显示的图片的 HTML 就行了。

```
<a href="images/image-1.jpg" rel="lightbox" title="my caption">image #1</a>
```

192

不过，这里的代码分离式的作用并没有达到理想状态，它需要等待网页的所有图片加载完，而不是在 DOM 可用时就执行。即使如此，这种 DOM 脚本（如代码清单 9-1 所示）也有它存在的道理和适应性。

代码清单9-1 定位到所有Lightbox锚点元素并使它们能正确显示

```
// 定位页面的所有锚点标签
var anchors = document.getElementsByTagName("a");

// 遍历所有的锚点标签
for ( var i=0; i < anchors.length; i++ ) {
      var anchor = anchors[i];

      // 确保该链接是"lightbox"的链接
      if ( anchor.href && anchor.rel == "lightbox" ) {
```

```
            // 显示Lightbox的点击
            anchor.onclick = function () {
                showLightbox(this);
                return false;
            };
        }
    }
```

随着新用户的不断反馈，Lightbox 逐渐进化，它也增加了一些新特性，比如键盘导航和动画。虽然日益复杂，但 Lightbox 因为极为简洁的核心，为你制作属于自己的类似图库提供了大量的灵感。

9.1.2　ThickBox

第二个图库是 ThickBox，由 Cody Lindley 使用 jQuery JavaScript 库创建。在具体实现上，它跟 Lightbox 非常相似，但更为小巧，并支持使用 Ajax 加载外部的 HTML 文件。在它网站上（http://codylindley.com/Javascript/257/thickbox-one-box-to-rule-them-all）可以找到关于这个库的更多信息，同时有一个演示（http://jquery.com/demo/thickbox/）。

从图 9-2 的屏幕截图可以看到，ThickBox 显示图片的效果跟 Lightbox 的非常相似。

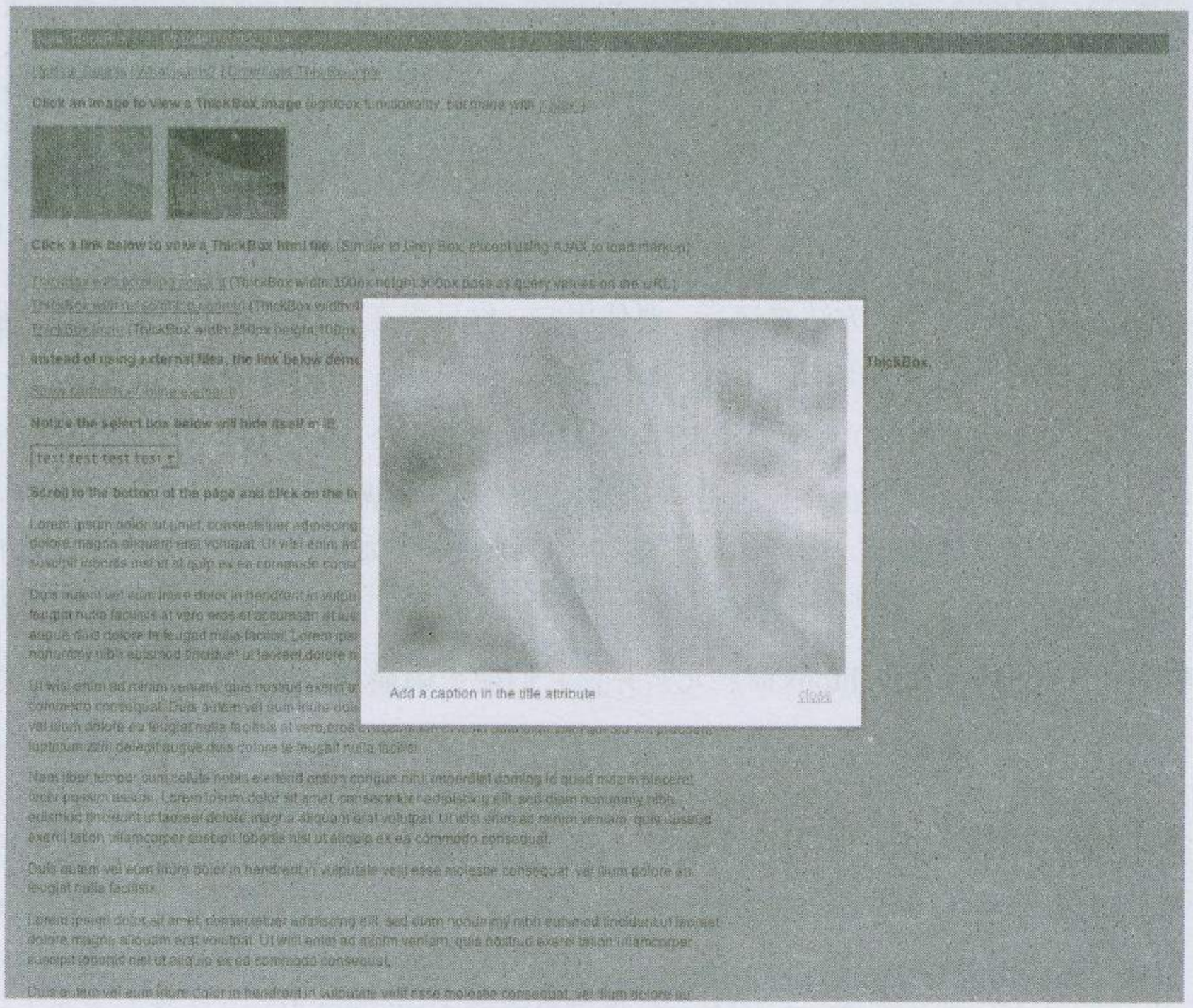


图9-2　在页面最佳位置显示一张图片的Thickbox

与 Lightbox 一样，Thickbox 在加载和自运行上使用分离式的手段。简单地在网页的头部引入脚本，就可以自动遍历 DOM 并找到所有 class 带"thickbox"的链接，如以下代码所示：

```
<a href="ajaxLogin.htm?height=100&width=250" class="thickbox">ThickBox login</a>
```

193

代码清单 9-2 展示了 Thickbox 动态并分离地增加它的功能，不管 DOM 可用与否（它在所有图片加载完毕之前执行，从而提供更好的用户体验）。

代码清单9-2 为class有"thickbox"的所有锚点元素赋予功能

```
// DOM一可用就开始查找thickbox
$(document).ready(function(){

    // 为有class为.thickbox的元素加上thickbox功能
    $("a.thickbox").click(function(){
        // 得到该thickbox的说明
        var t = this.title || this.name || this.href || null;

        // 显示该thickbox
         TB_show(t,this.href);
        // 删除链接的foucs
        this.blur();

        // 确保链接不会像平常般跳转
        return false;
    });

});
```

194

由于 Thickbox 包含大量其他的特性并具有更少的代码，它当然成为可以替代 Lightbox 的另一个选择。

接下来你会看到如何克隆一个属于你自己的图库，而正确地制作一个图库，需要考虑采用一些必要的精妙复杂手段或技术。

9.2 制作图库

制作图库的第一步是准备足够让你遍历的图库图片集合。假设一个页面可以包含多个图库，每个图库也可以包含多个图片。此外非常重要的是，就算 JavaScript 不执行，你也要让图片以语义和可理解的形式显示。这对禁用 JavaScript（或者缺乏 CSS 支持）的人来说，仍能确保拥有可接受的用户体验。

用以制作图库的基本 HTML 如代码清单 9-3 所示。

代码清单9-3 支撑图库的图片的基本HTML页面

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
            "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
<head>
      <title>Random Cat Pictures</title>
```

```
</head>
<body>
        <h1>Random Cat Pictures</h1>

        <p>Lorem ipsum dolor . . . </p>

        <!--
             我们的图库，必须是一个class为"gallery"的<ul>————————也应该有相关的title.
         --->
195     <ul class="gallery" title="Random Cat Pictures">
          <!--
               每一个图片都应该在<li>内，并带一个指向真是图片的链接。
               如果图片过高，确保加上"tall"的class.
          -->

             <li><a href="image1.jpg"><img src="image1.jpg"
                  alt="Fat Cat Sleeping"/></a></li>
             <li><a href="image2.jpg"><img src="image2.jpg"
                  alt="Sharing the Bed"/></a></li>
             <li class="tall"><a href="image3.jpg"><img src="image3.jpg"
                  alt="Out the Window"/></a></li>
             <li class="tall"><a href="image4.jpg"><img src="image4.jpg"
                  alt="In the Sun"/></a></li>
             <li><a href="image5.jpg"><img src="image5.jpg"
                  alt="Very Comfortable"/></a></li>

        </ul>

        <p>Lorem ipsum dolor . . . </p>
</body>
</html>
```

然后，需要增加一些样式，以让它更美观和可导航，如代码清单 9-4 所示。

代码清单9-4　适当样式化页面的CSS

```
body {
        font-family: Arial;
        font-siz: 14px;
}

/* 为图库增加一个漂亮的盒子样式。 */
ul.gallery {
        list-style: none;
        padding: 5px;
        background: #EEE;
        overflow: auto;
        border: 1px solid #AAA;
196      margin-top: 0px;
}

/* 为每一个图片片建立一个标准宽度和高度盒子。 */
ul.gallery li {
     float: left;
     margin: 6px;
```

```
    width: 110px;
    height: 110px;
    background: #FFF;
    border: 2px solid #AAA;
}

/* 水平图片100px宽 */
ul.gallery img {
    width: 100px;
    margin: 5px;
    border: 0px;
    margin-top: 17px;
}

/* 垂直图片100px高 */
ul.gallery li.tall img {
    height: 100px;
    width: auto;
    margin-top: 5px;
    margin-left: 17px;
}
```

最后，基本 HTML 和 CSS 的结果如图 9-3 所示。

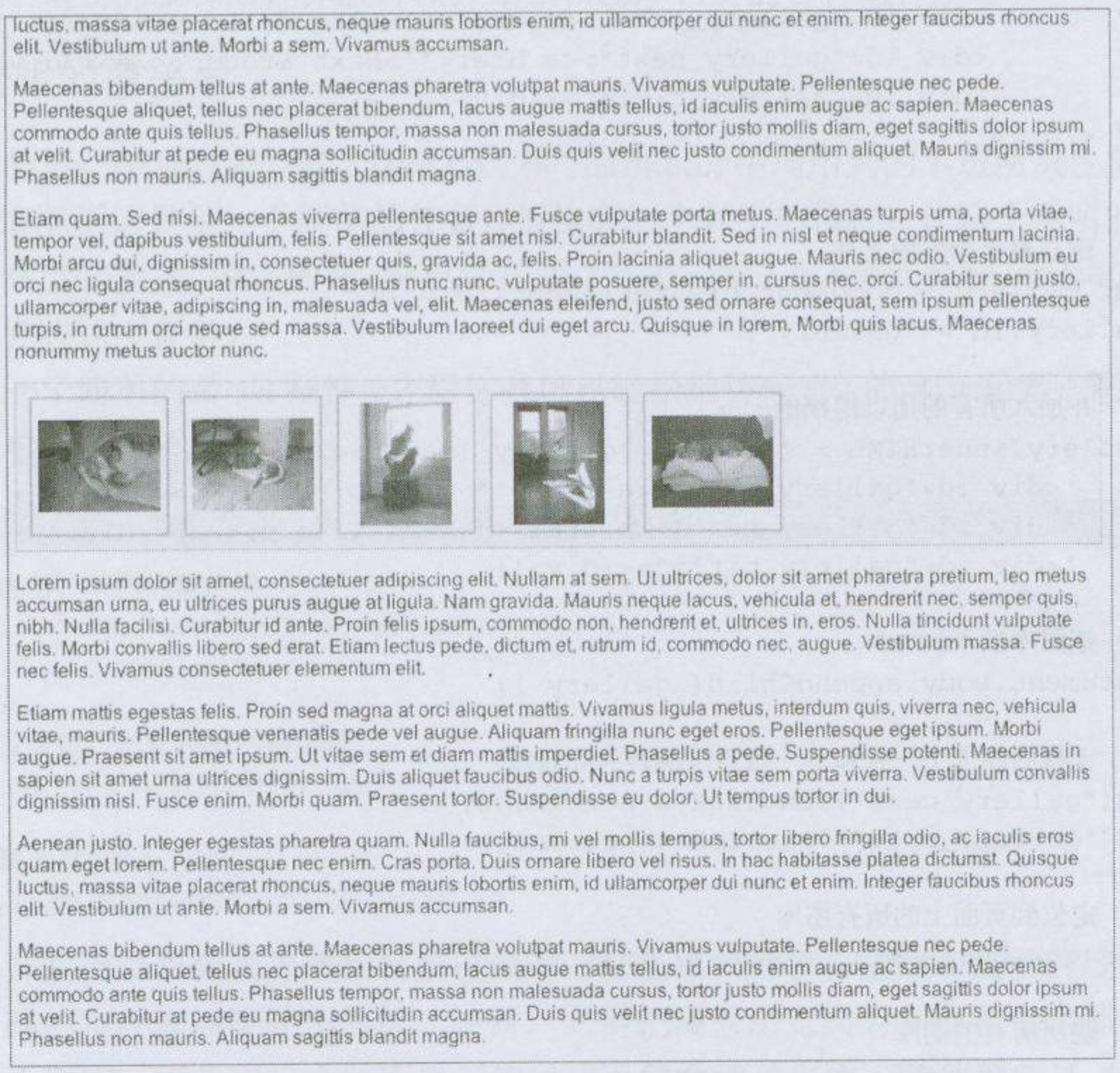


图9-3 一个简单样式化的图片页面

建立这个基本 HTML 页面后，你可以准备综合组件，用它们来制作漂亮的、JavaScript 驱动
197 的图库了。

9.2.1　分离加载

再次强调，这个图库必须用分离式脚本编程来制作。于是，对于支持脚本的用户来说，你再也不用只因为需要加入一些诱人的视觉效果而在页面上增加多余（且不语义）的 HTML。你要做
198 的就是在页面完成加载时注入部分的 HTML 元素，如代码清单 9-5 所示。

代码清单9-5　在DOM中注入初始化的HTML并为每个元素绑定必要的事件处理器

```
// 跟踪记录当前所看的图片
var curImage = null;

// 在修改或遍历DOM之前等待文档的完成加载
window.onload = function() {
      /*
       * 建立如下的DOM结构
       * <div id="overlay"></div>
       * <div id="gallery">
       *       <div id="gallery_image"></div>
       *       <div id="gallery_prev"><a href="">&laquo; Prev</a></div>
       *       <div id="gallery_next"><a href="">Next &raquo;</a></div>
       *       <div id="gallery_title"></div>
       * </div>
       */

      // 建立整个图库的支架
      var gallery = document.createElement("div");
      gallery.id = "gallery";

      // 并加入所有的用以组织的div
      gallery.innerHTML = '<div id="gallery_image"></div>' +
          '<div id="gallery_prev"><a href="">&laquo; Prev</a></div>' +
          '<div id="gallery_next"><a href="">Next &raquo;</a></div>' +
          '<div id="gallery_title"></div>';

      // 把图库插入DOM中
      document.body.appendChild( gallery );

      // 支持图库内上一张和下一张点击跳转的处理器
      id("gallery_next").onclick = nextImage;
      id("gallery_prev").onclick = prevImage;

      // 定位到页面上的所有图库
      var g = byClass( "gallery", "ul" );

      // 遍历所有的图库
      for ( var i = 0; i < g.length; i++ ) {
            // 并定位到幻灯图片的所有链接
```

199

```
    var link = tag( "a", g[i] );
  // 遍历所有的图片链接
  for ( var j = 0; j < link.length; j++ ) {
      // 确保做到：当点击的时候，显示图库而不是跳转图片
      link[j].onclick = function(){
          // 显示灰色背景的覆盖层
          showOverlay();

          // 在图库内显示图片
          showImage( this.parentNode );

          // 确保浏览器不会像普通情况下跳转图片
          // image, like it normally would
          return false;
      };
}
        }

        // 在图库内加入幻灯导航
        addSlideShow( g[i] );
      }
};
```

处理完这个重要的步骤后，你就可以开始制作图库自身的各种组件了。

9.2.2 半透明的覆盖层

你要制作的第一个项目是，在 Lightbox 和 ThickBox 中均用到的半透明灰色覆盖层。你会看到，大部分情况下这只是一个简单的任务，但也有一个棘手的情况：让这个透明的覆盖能够符合当前页面的高度和宽度。幸运的是，你已经在第 7 章中开发了 pageWidth()和 pageHeight()，它们可以帮你完成这个任务。

只需建立一个简单的带 id 特性（之后你就可以轻松地访问它）的 div 元素，并插入到 DOM 中，如代码清单 9-6 所示。

代码清单9-6 创建一个简单的div元素并插入到DOM中

```
// 创建半透明、灰色的覆盖层
var overlay = document.createElement("div");
overlay.id = "overlay";

// 当点击覆盖层，把它和图库都隐藏
overlay.onclick = hideOverlay;

// 把这个覆盖层插入DOM中
document.body.appendChild( overlay );
```

200

接下来，开发两个函数来隐藏和触发覆盖层。这也比较棘手。隐藏和显示的过程相当容易，但是要找出正确的宽度和高度并不简单。通常，只需把覆盖层定义为高度和宽度都是 100%就可以了。但这并不现实，因为用户在图库展示的时候可能会滚动页面，致使覆盖层“不告而别”。

解决方案是，让覆盖层与当前页面保持一致的宽度和高度。你可以使用第7章中的pageWidth()和pageHeight()函数来得到这些数值。

隐藏和显示覆盖层的完整代码如代码清单9-7所示。

代码清单9-7　隐藏和显示图库半透明覆盖层的两个必要函数

```
// 隐藏当前图库的灰色覆盖层
function hideOverlay() {
        // 确保重置当前图片
        curImage = null;

        // 并隐藏覆盖层和图库
        hide( id("overlay") );
        hide( id("gallery") );
}

// 显示灰色覆盖层
function showOverlay() {
        // 获取覆盖层
        var over = id("overlay");

        // 让它与整个页面保持一致的高度和宽度（避免页面滚动后产生问题）
        over.style.height = pageHeight() + "px";
        over.style.width = pageWidth() + "px";

        // 并渐隐
        fadeIn( over, 50, 10 );
}
```

最后，加上必要的CSS，以正确显示半透明覆盖层，如代码清单9-8所示。

代码清单9-8　正确显示半透明覆盖层的必要CSS

```
#overlay {
        background: #000;
        opacity: 0.5;
        display: none;
        position: absolute;
        top: 0px;
        left: 0px;
        width: 100%;
        height: 100%;
        z-index: 100;
        cursor: pointer;
        cursor: hand;
}
```

这些加入的HTML和CSS结合在一起的效果如图9-4所示。

建立覆盖层并加入到页面后，你可以着手准备在它之上显示图片了。

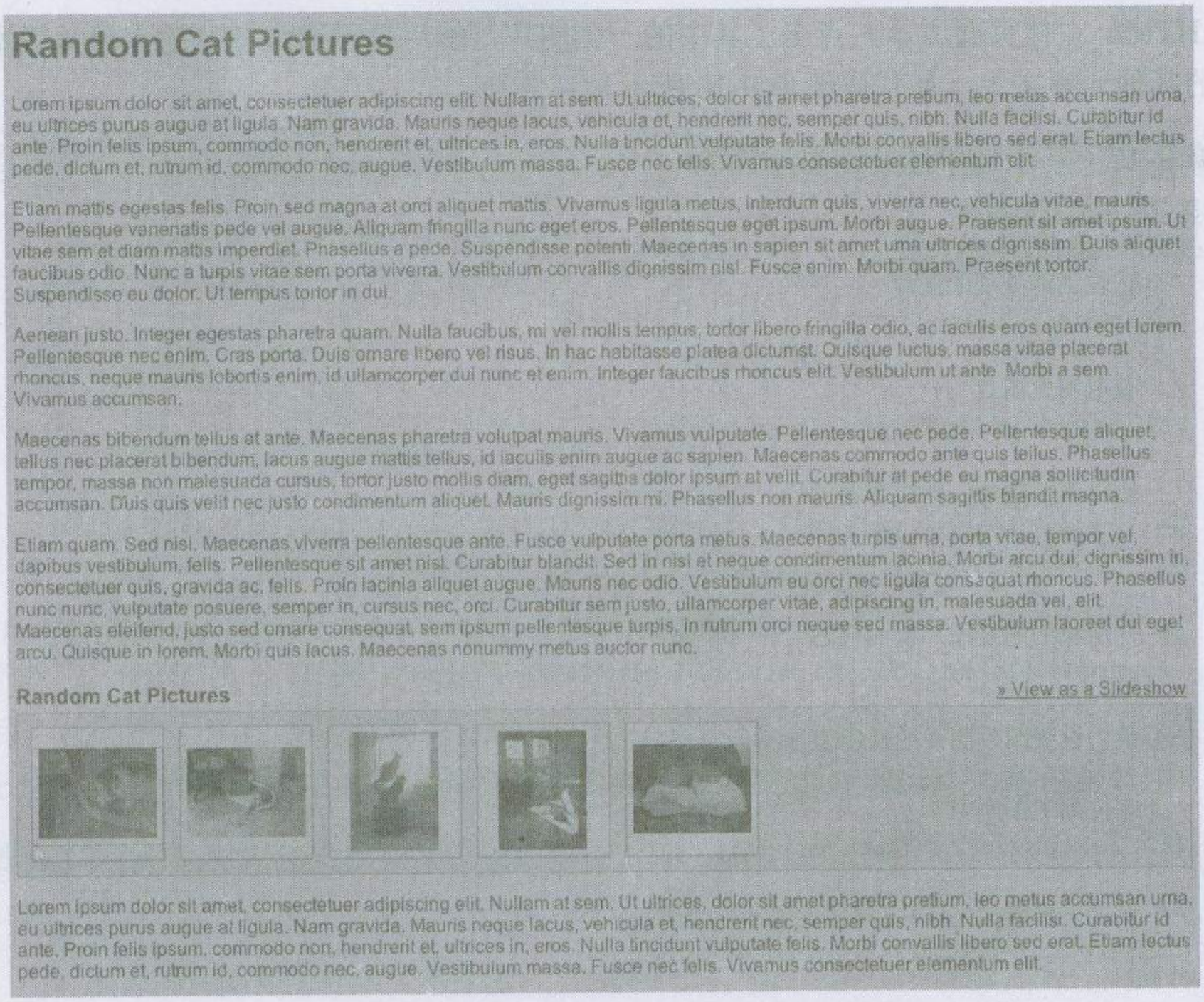


图9-4 在页面显示一个半透明覆盖层的效果

202

9.2.3 定位盒子

图库还需要制作的组件是一个包含当前图片的容器，它会浮动在半透明覆盖层之上。如果正确支持 CSS 的话，只需使用固定定位（fixed position），这样就可以定位到任何东西之上，脱离页面滚动位置的影响。不幸的是，由于某些现代浏览器（如 IE6）对 CSS2 支持的严重不足，这个步骤显得相当困难。

在开始之前先得假设你已经在页面中有了如代码清单 9-9 所示的 DOM 结构（可以把它增加到代码清单 9-5 的页面中去）。

代码清单9-9 定位在半透明覆盖层之上的图库的HTML

```
<div id="gallery">
        <div id="gallery_image"></div>
        <div id="gallery_prev"><a href="">&laquo; Prev</a></div>
        <div id="gallery_next"><a href="">Next &raquo;</a></div>
        <div id="gallery_title"></div>
 </div>
```

有了这个基本的 HTML 结构，还需要开发一个合适的函数来显示图库的 `div` 并把图片放到这个 `div` 中去。触发这个特定函数的方法有好几种，但最明显的方法莫过于判断用户是否在图库

内（如 HTML 主体页面所示）点击了其中的一张图片，然后显示这张图片的大尺寸版本，覆盖在页面的其他元素之上（如代码清单 9-10 所示）。

代码清单9-10　基于选中图片而显示图库

```
// 显示图库的当前图片
function showImage(cur) {
    // 记住当前处理的图片
    curImage = cur;

    // 获取图库图片
    var img = id("gallery_image");

    // 删除当前图片，如果存在的话
    if ( img.firstChild )
        img.removeChild( img.firstChild );

    // 并用我们的新图片取而代之
    img.appendChild( cur.firstChild.cloneNode( true ) );

    // 我们设置图库图片的说明为该图片的'alt'里的内容
    id("gallery_title").innerHTML = cur.firstChild.firstChild.alt;
    // 定位到主图库中
    var gallery = id("gallery");

    // 设置正确的class(这样才能显示恰当的尺寸)
    gallery.className = cur.className;

    // 然后平滑地渐隐
    fadeIn( gallery, 100, 10 );

    // 确保图片在屏幕中的位置正确
    adjust();
}
```

203

在 showImage 函数中，最后调用 adjust 函数。adjust 函数负责把图片定位到用户窗口的绝对中间（就算用户滚动鼠标或者重置窗口的大小）。这是重要的一步，如代码清单 9-11 所示，它让图库行为和表现都更为自然。

代码清单9-11　基于图片宽高和用户滚动的具体情况，重定位图库

```
// 重定位图库到页面的中心，就算页面经过了滚动
function adjust(){
    // 定位到图库
    var obj = id("gallery");

    // 确保图库是存在的
    if ( !obj ) return;

    // 得到它当前的高度和宽度
    var w = getWidth( obj );
    var h = getHeight( obj );
```

```
        // 定位这个盒子，相对于窗口垂直居中
        var t = scrollY() + ( windowHeight() / 2 ) - ( h / 2 );

        // 但不能超过页面的顶端
        if ( t < 0 ) t = 0;

        //定位这个盒子，相对于窗口水平居中
        var l = scrollX() + ( windowWidth() / 2 ) - ( w / 2 );

        // 但不能超过页面的左端
        if ( l < 0 ) l = 0;

        // 设置元素经过调整后的位置
        setY( obj, t );
        setX( obj, l );
};

// 用户滚动页面或者重置浏览器大小，每次都重新调整图库的位置
window.onresize = document.onscroll = adjust;
```

204

最后，代码清单 9-12 是保证图库正确定位的必要 CSS。你会注意到，这只不过是一个绝对定位的 div，且它的样式属性 z-index 比较大，所以能够在页面的所有元素之上。

代码清单9-12　定位图库到正确位置上的CSS

```
#gallery {
        position: absolute;
        width: 650px;
        height: 510px;
        background: #FFF;
        z-index: 110;
        display: none;
}

#gallery_title {
        position: absolute;
        bottom: 5px;
        left: 5px;
        width: 100%;
        font-size: 16px;
        text-align: center;
}

#gallery img {
        position: absolute;
        top: 5px;
        left: 5px;
        width: 640px;
        height: 480px;
        border: 0px;
        z-index: 115;
}
```

205

```
#gallery.tall {
    width: 430px;
    height: 590px;
}

#gallery.tall img {
    width: 420px;
    height: 560px;
}
```

CSS、HTML 和 JavaScript 结合起来的效果如图 9-5 所示，现在拥有了一个定位过的图库，看起来相当耀眼呢。

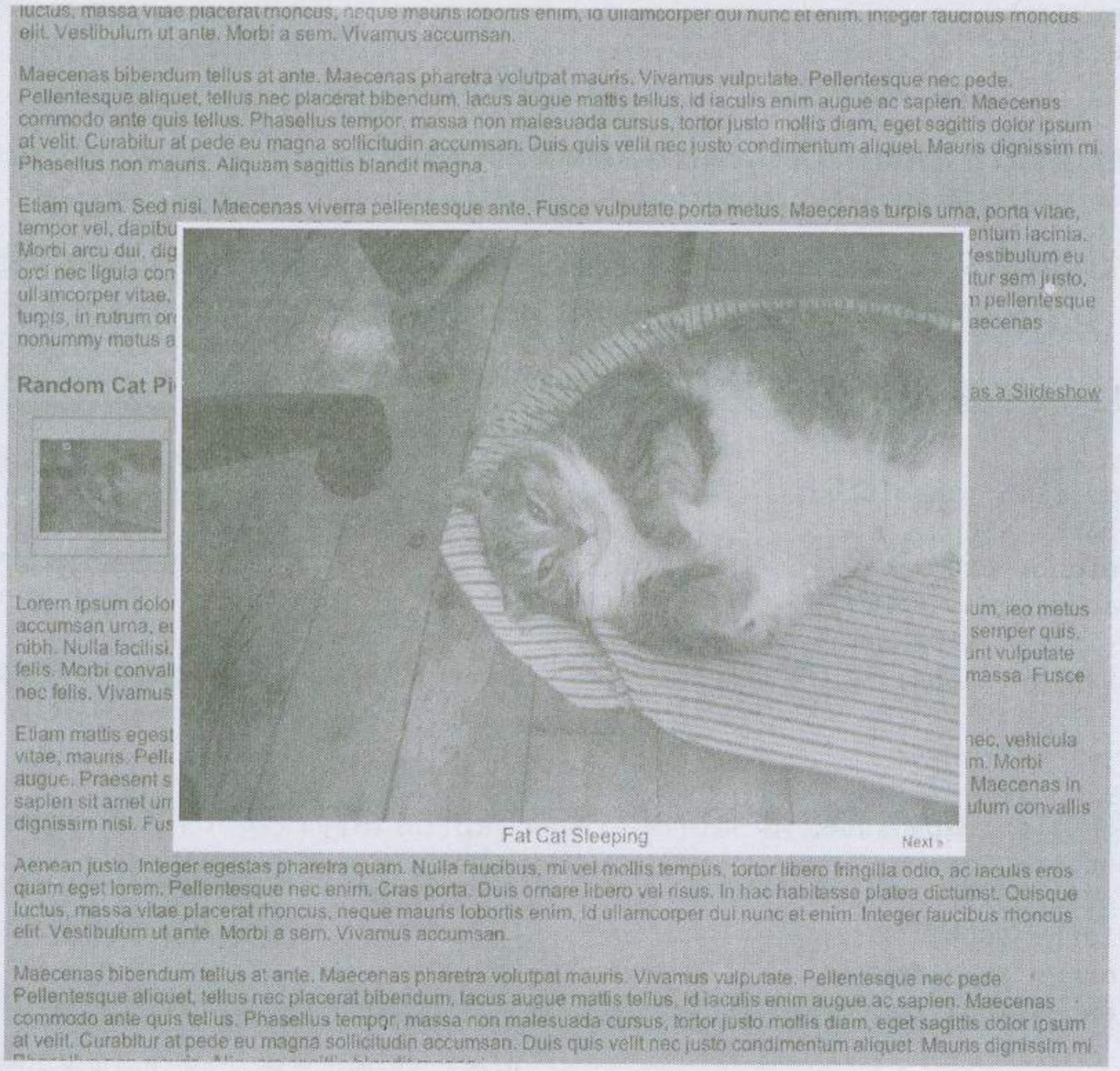


图9-5　定位过的图库显示在半透明覆盖层之上

现在已经跨过了创建漂亮观感这一重要步骤的难关，你接下来要关注如何让用户方便地导航图库里的每张图片了。

9.2.4　导航

图片的显示占了页面的剩余部分（和页面之间还有一个半透明覆盖层），需要使用一些更好的方式，以便导航图库中的不同图片。之前，在为图库的覆盖层指定 HTML 时，你已经加入了

用以导航的链接了。使用这些链接就完全可以让用户在显示的覆盖层上导航图库了。

可以跟踪当前正查看的图片（在这种情况下，图片的引用记录在变量 curImage 中）来为图库增加导航功能。由此，你可以轻松地确定用户在图库中的位置，并引导他们到期望的位置，如代码清单 9-13 所示。

代码清单9-13 在图库中引导用户到期望位置的两个必要函数

```
// 获取上一个图片并显示它
function prevImage() {
        // 定位到上一个图片并显示它
        showImage( prev( curImage ) );

        // 防止链接跳转
        return false;
}

// 获取下一个图片并显示它
function nextImage() {
        // 定位到下一个图片并显示它
        showImage( next( curImage ) );

        // 防止链接跳转
         return false;
}
```

导航链接的难题在于需要确定显示它们的恰当时机，要保证当前图片（图库中的）有上一张或者下一张的时候才显示它们。根据具体情形隐藏或者显示图库的链接会用到 showImage()函数，以在恰当的时候显示导航。处理导航状态的代码如代码清单 9-14 所示。 207

代码清单9-14 检查何时隐藏或显示下一张和上一张的导航链接

```
// 如果到达了幻灯的最后一张，那么隐藏下一张的链接
if ( !next(cur) )
        hide( id("gallery_next") );

// 反之，确保它是可见的
else
        show( id("gallery_next") );
// 如果在幻灯的第一张，那么隐藏上一张的链接 slideshow

if ( !prev(cur) )
        hide( id("gallery_prev") );

// 反之，确保它是可见的
else
        show( id("gallery_prev") );
```

最后，代码清单 9-15 是导航链接正确定位所需的 CSS 样式。

代码清单9-15 定位导航链接的CSS

```
#gallery_prev, #gallery_next {
```

```
        position: absolute;
        bottom: 0px;
        right: 0px;
        z-index: 120;
        width: 60px;
        text-align: center;
        font-size: 12px;
        padding: 4px;
}

#gallery_prev {
        left: 0px;
}

#gallery_prev a, #gallery_next a {
        color: #000;
        text-decoration: none;
}
```

导航的实际例子可以参看图 9-5。注意，在图库的下方有一个引导用户访问图库下一张图片
208 的链接。该链接适时隐藏或显示，取决于用户在图库中所处的位置。

9.2.5　幻灯片

图库的最后一个内容是大多数用户都喜闻乐见的：图库中所有图片的动态优美的幻灯片。与前述的导航链接相比，要增加这个具体的补充效果相当简单。创建幻灯的步骤可分解为两步：

(1) 在文档内建立一个额外的链接，用户点击它后开始展示幻灯。

(2) 构建幻灯播放的过程，控制显示哪张图片和何时切换图片。

第一步如代码清单 9-16 所示。

代码清单9-16　在DOM中增加用来初始化幻灯的导航

```
function addSlideshow( elem ) {
        // 我们会在幻灯的周围创建一些额外的上下文信息

        // 创建幻灯的头部和包裹器
        var div = document.createElement("div");
        div.className = "slideshow";

        // 显示幻灯的名字，这里使用的是图库的title
        var span = document.createElement("span");
        span.innerHTML = g[i].title;
        div.appendChild( span );

        // 创建一个链接，由此我们可以把图库所有的图片都当作幻灯中的一幕
        var a = document.createElement("a");
        a.href = "";
        a.innerHTML = "&raquo; View as a Slideshow";

        // 点击后开始幻灯
        a.onclick = function(){
```

```
            startShow( this.parentNode.nextSibling );
            return false;
        };

        // 为页面插入新的导航和头部
        div.appendChild( a );
        elem.parentNode.insertBefore( div, elem );
}
```

209

现在需要制作用来管理整个幻灯系列动画的控制器，整个控制器有一系列的超时管理，它们都同时初始化（尽管被设置为交替进行）。最终的显示结果平滑而优美，而表现也会非常无缝。触发幻灯的代码如代码清单 9-17 所示。

代码清单9-17 在具体图库上初始化幻灯的代码

```
// 在一个具体的图库上开始所有图片的幻灯
function startShow(obj) {
        // 定位到图库的每一张图片
        var elem = tag( "li", obj );

        // 定位到显示的整个图库
        var gallery = id("gallery");

        // 遍历每一个匹配的图库图片
        for ( var i = 0; i < elem.length; i++ ) new function() {
            // 记录被引用的当前图片
            var cur = elem[i];

            // 我们每5秒显示一张新图片
            setTimeout(function(){
                // 显示指定的图片
                showImage( cur );

                // 并在3.5秒后渐隐（因为需要有1秒的渐隐时间）
                setTimeout(function(){
                    fadeOut( gallery, 0, 10 );
                }, 3500 );
            }, i * 5000 );

        };

        // 然后在结束时隐藏全部
        setTimeout( hideOverlay, 5000 * elem.length );

        // 但还是显示覆盖层，因为幻灯刚开始
        showOverlay();
}
```

最后，还要有一些必要的 CSS 来样式化幻灯的初始化链接，如代码清单 9-18 所示。

210

代码清单9-18 显示幻灯链接导航的CSS

```
div.slideshow {
        text-align: right;
```

```
        padding: 4px;
        margin-top: 10px;
        position: relative;
}

div.slideshow span {
        position: absolute;
        bottom: 3px;
        left: 0px;
        font-size: 18px;
        font-weight: bold;
}

div.slideshow a {
        color: #000;
}
```

幻灯实际运行情况的截屏非常困难，但至少可以看看为页面添加导航链接后的效果，如图9-6所示。

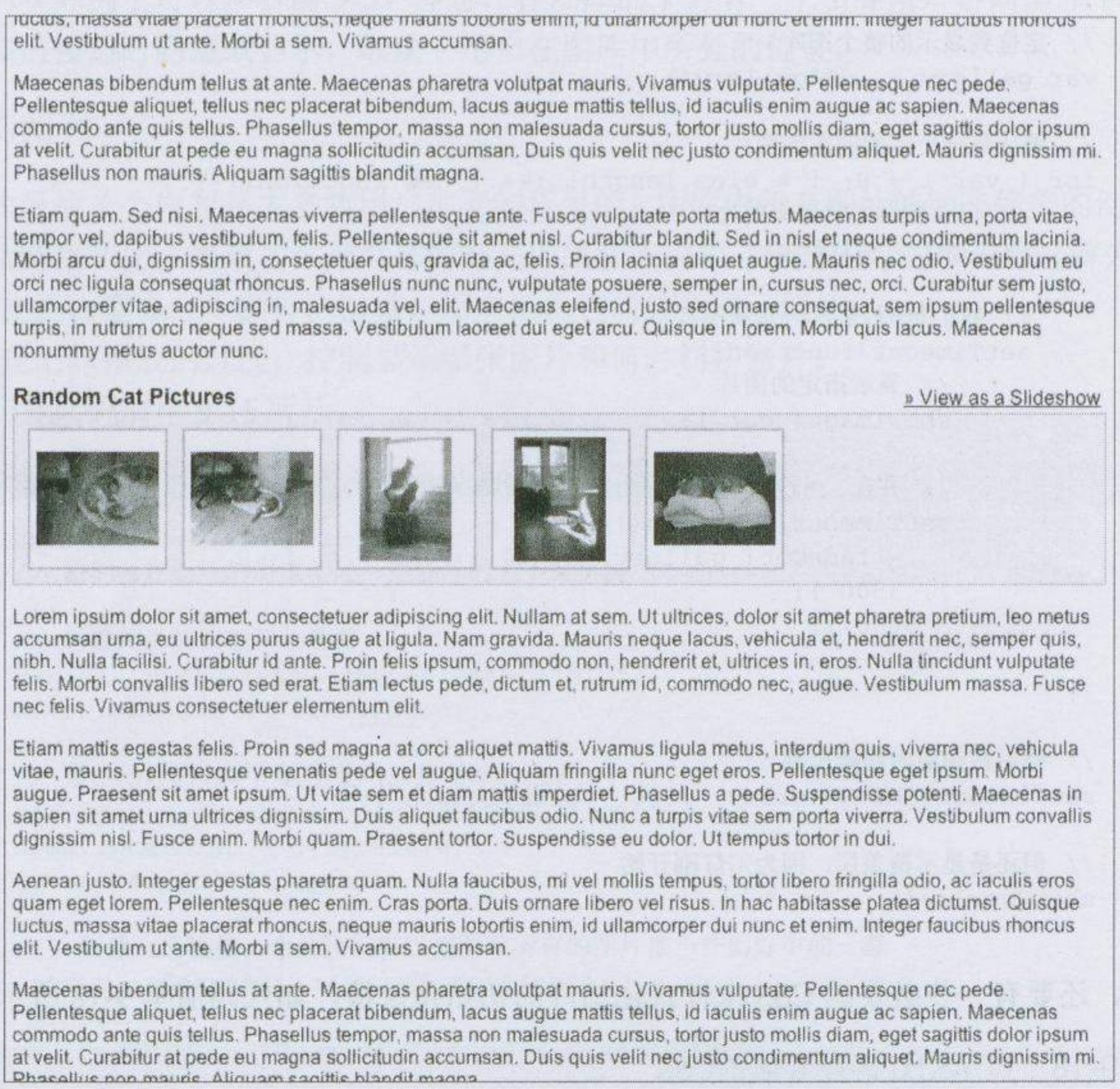


图9-6　加入到页面中的幻灯导航

综合前述的幻灯和导航技术，构建你自己的动态 Web 应用程序（比如某个幻灯软件）就有了开始的可能。为了加深了解幻灯是如何运作的，我建议你运行这一章的所有代码，耳濡目染它简洁但令人信服的效果。

211

9.3 小结

本章中所演示的图库、导航和幻灯，确实能够作为页面内创建补充功能的 DOM 脚本编程的最佳实践。它们基于你所学过的知识而制作，很明显，使用动态、分离式的 DOM 脚本编程，你将会轻松很多。

在本章中你还看到了一些其他的图库，它们能为你制作自己的图库提供灵感。然后定义了一段标准的 HTML 结构和显示图库，并制作显示它的基础单元（包括覆盖层、定位盒子和导航）。最后，加入能让用户开始的幻灯动画。使用最少的代码和付出，你已经建立了一个强大的动态 DOM 脚本编程应用程序。

212

Part 4 第四部分

Ajax

本部分内容

第 10 章

Ajax导引

10

Ajax 是 Adaptive Path 的 Jesse James Garrett 提出的一个名词，他在解释 XMLHttpRequest 对象（大部分现代浏览器都提供了这个对象）所进行的异步的客户端到服务器端通信时，使用了这一名词。作为异步 JavaScript 与 XML（Asynchronous JavaScript and XML）的缩写，它其实只是对创建动态 Web 应用程序所必备技术的一个统称。而且 Ajax 技术的个别组成部分并非不可替换，比如使用 HTML 来替代 XML，就根本不会影响它的正确性。

在本章里，你将了解到组成整个 Ajax 过程的细节（围绕从浏览器发送请求到服务器这个中心）。我们讨论的内容从具体的请求到 JavaScript 的交互和完成工作的必要的数据操作。它包括：

- 分析不同类型的HTTP请求，决定以何种方式将数据对象发送到服务器最合适。
- 观察整个HTTP响应过程，尝试处理所有可能发生的错误，包括服务器的超时。
- 读取、遍历并操作来自服务器响应的结果数据。

这样你就能够完整地理解整个 Ajax 过程是如何运作的，应该如何实现它，从而能掌握它的各种应用场景——从常见的工作到完整的应用程序。在第 11 章～第 13 章里，你还能看到一系列运用 Ajax 技巧的实例分析。

10.1 使用 Ajax

创建一个简单的 Ajax 不需要很多代码，虽然这样的实现已能给你提供非常多的特性了。比如，在提交了表单之后，不需要用户请求一个完整的新页面，而是异步地进行提交，在提交完成后加载表单结果所期望的那一小部分内容即可。再比如，搜索可购买域名的过程通常麻烦又费时，每次需要查找一个新域名的时候都要往表单里输入请求和提交它，等待页面加载完成。而在 Ajax 的帮助下，你可以获得即时的输出，类似在线应用站点 Instant Domain Search（http://instantdomainsearch.com/），如图 10-1 所示。

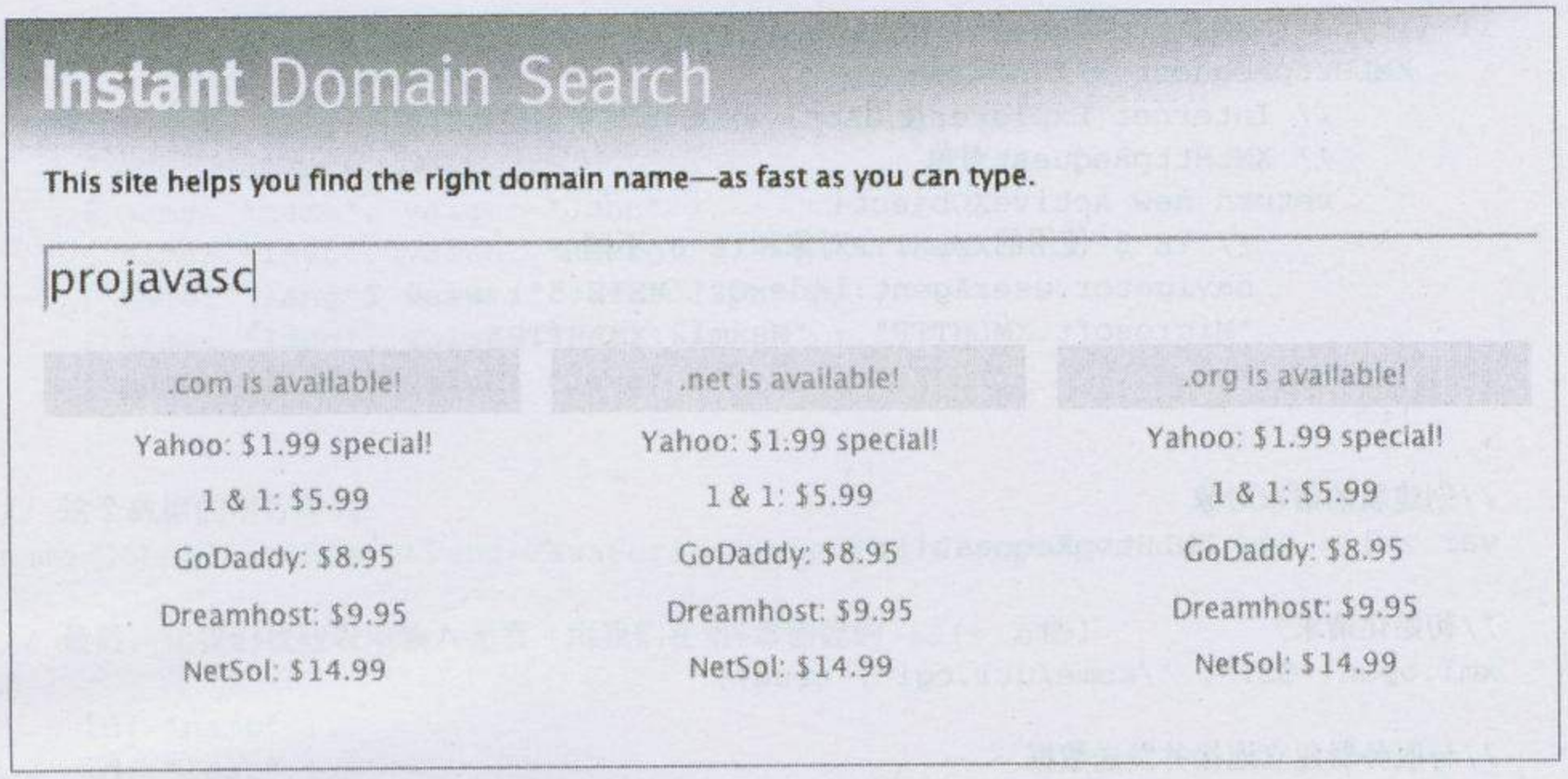


图10-1 Instant Domain Search随输入域名进行查询的一个例子

10.1.1 HTTP 请求

Ajax 中最重要也是最固定的部分是 HTTP 请求。HTTP（Hypertext Transfer Protocol，超文本传输协议）就是为了传输 HTML 文档和相关文件设计的一种协议。令人欣慰的是，所有的浏览器都支持一种使用 JavaScript 来动态建立 HTTP 连接的方式。这在开发更具丰富响应的 Web 应用程序中显得尤为有用。

归根结底，Ajax 的目的是异步地向服务器发送数据并接收数据。至于数据是什么格式，取决于你的具体文档，将在 10.2 节详细讨论。

在下面几个小节里，你将看到如何格式化数据，以便使用不同的 HTTP 请求传输到服务器上。你还能看到如何与服务器建立基本的连接，以及在跨浏览器环境中处理这些的必要细节。

1. 建立连接

Ajax 的关键在于创建到服务器的链接，实现这一目的有许多方法。不过这里只考虑一种发送和接受数据都很方便的方法，这种方法通常叫做“使用 XMLHttpRequest 对象”。

根据用户浏览器的不同，XMLHttpRequest 对象的数据通信通常可用两种方法实现：

(1) IE 是这种基于浏览器通信技术的创始者，使用 ActiveXObject 来创建连接（ActiveX 对象的版本根据 IE 的版本而异）。幸运的是，IE 7 对 XMLHttpRequest 对象已经有了直接的支持。

(2) 其他所有的现代浏览器支持直接使用 XMLHttpRequest 对象。包括 Firefox、Opera 和 Safari。 216

令人欣慰的是，尽管 IE 创建 XMLHttpRequest 对象的方法和其他现代浏览器不一样，但创建出来的对象还是支持同一套功能的。XMLHttpRequest 对象有一系列用于建立连接、读取数据的方法。代码清单 10-1 就展示了如何向服务器发送一个基本的 GET 请求。

代码清单10-1 一种向服务器发送HTTP GET请求的跨浏览器方法

```
// 如果使用了IE，需要在XMLHttpRequest对象外包装一层
```

```
if ( typeof XMLHttpRequest == "undefined" )
    XMLHttpRequest = function(){
        // Internet Explorer使用ActiveXObject来创建新的
        // XMLHttpRequest对象
        return new ActiveXObject(
            // IE 5 使用的XMLHTTP对象和IE 6 不同
            navigator.userAgent.indexOf("MSIE 5") >= 0 ?
            "Microsoft.XMLHTTP" : "Msxml2.XMLHTTP"
        );
    };

//创建新的请求对象
var xml = new XMLHttpRequest();

//初始化请求①
xml.open("GET", "/some/url.cgi", true);

//与服务器建立连接并发送数据
xml.send();
```

如你所见，与服务器建立连接所需的代码非常简单。困难之处在于你需要高级特性（比如检查超时或修改过的数据）的时候，将在 10.1.2 节中介绍这些细节。

Ajax 方法中最重要的特性是允许数据在客户端（即 Web 浏览器）和服务器端间传输。鉴于这样的情况，让我们看看如何包装数据来发给服务器。

2. 数据串行化

要发送一系列数据到服务器上，第一步就是整理它的格式，使服务器易于读取，这一过程称为“串行化（serialization）”。串行化有两种不同的情况，但都能满足多种不同的传输需求。

(1) 传输一个常规 JavaScript 对象，其中可能包含键/值的对（值可能是字符串也可能是数字）。

(2) 从一系列表单的输入栏中提交。这种情况和第一种不同之处在于提交的元素要按顺序排列，而第一种情况可以任意排序。

217

让我们看看一些例子，分别发送不同类型的数据，以及由它们转换得到的服务器友好的、串行的输出（如代码清单 10-2 所示）。

代码清单10-2　将原生JavaScript对象转换为串行形式的例子

```
// 一个包含“键/值”对的例子
{
    name: "John",
    last: "Resig",
    city: "Cambridge",
    zip: 02140
}

// 串行形式
```

① 原文为open the socket（打开套接字），这是错误的。考虑XHR对象完全不需要考虑到socket编程的细节，Microsoft、Apple和Mozilla的文档也无一提到过open和socket有任何关系，都只是说open用于初始化一个准备发起仍在“pending”状态中的请求。——译者注

```
name=John&last=Resig&city=Cambridge&zip=02140

// 另一组数据，包含许多值
[
    { name: "name", value: "John" },
    { name: "last", value: "Resig" },
    { name: "lang", value: "JavaScript" },
    { name: "lang", value: "Perl" },
    { name: "lang", value: "Java" }
]

// 这个数据的串行形式
name=John&last=Resig&lang=JavaScript&lang=Perl&lang=Java

// 最后，让我们找些表单输入元素（用我们在第5章创建的 id() 方法）
[
    id( "name" ),
    id( "last" ),
    id( "username" ),
    id( "password" )
]

// 将其串行化为字符串
name=John&last=Resig&username=jeresig&password=test
```

这种用于串行化上述数据的格式是发送 HTTP 请求的标准格式，你很可能曾见过这样的 HTTP GET 标准请求：

```
http://someurl.com/?name=John&last=Resig
```

218

这样的数据也可以通过 POST 请求（而且比 GET 允许传输的数据量要多得多）来传输，你将在接下来的一节看到它们的区别。

现在，让我们建立一个将代码清单10-2中的数据结构串行化的标准方法。在代码清单10-3可以看到这么一个函数，它能对大多数的表单元素进行串行化，不过多选按钮除外。

代码清单10-3 一个标准函数，将数据结构串行化为兼容HTTP的参数模式

```
// 串行化一系列数据。支持两种不同的对象：
//  - 表单输入元素的数组
//  - 键/值对的散列表
// 本函数返回串行化后的字符串
function serialize(a) {
    // 串行化结果的集合
    var s = [];

    // 若传入的参数是数组，假定它们是表单元素的数组
    if ( a.constructor == Array ) {

        // 串行化表单元素
        for ( var i = 0; i < a.length; i++ )
            s.push( a[i].name + "=" + encodeURIComponent( a[i].value ) );
```

```
    // 否则，假定这是一个键值对对象
    } else {

        // 串行化键值对
        for ( var j in a )
            s.push( j + "=" + encodeURIComponent( a[j] ) );

    }

    // 返回串行化结果
    return s.join("&");
}
```

现在数据都转成了串行形式的字符串，接下来看看如何用 GET 或者 POST 请求发送数据到服务器上。

3. 发送GET请求

让我们重新看看用 XMLHttpRequest 发送 GET 请求到服务器的方法，不过这次要连带发送串行数据了。代码清单 10-4 是一个简单的例子。

219

代码清单10-4　一种跨浏览器的向服务器发送HTTP GET请求的方法（并且不读取任何结果数据）

```
// 创建请求对象
var xml = new XMLHttpRequest();

// 初始化异步 GET 请求
xml.open("GET", "/some/url.cgi?" + serialize( data ), true);

// 与服务器建立连接
xml.send();
```

要注意的是串行数据是附在服务器 URL 后面的，用?字符分隔。所有的 Web 服务器和应用程序框架都知道如何将?后面这部分数据解析为键值对。我们会在 10.2 节讨论如何处理响应，这些响应是服务器根据你提交的数据而返回的。

4. 发送POST请求

使用 XMLHttpRequest 发送 HTTP 请求的另一种方式是 POST，这是一种与 GET 截然不同的方式。POST 支持发送任意格式、任意长度的数据，而不仅限于串行化字符串。

用于发送串行化格式数据的 MIME 类型（content type）通常是 application/x-www-form-urlencoded。这意味着你还能以 text/xml 或 application/xml 的形式给服务器直接发送 XML，甚至以 application/json 的形式发送 JavaScript 对象。

发送这种请求并传送附加串行数据的一个简单例子如代码清单 10-5 所示。

代码清单10-5　一种跨浏览器的向服务器发送HTTP POST请求的方法（并且不读取任何结果数据）

```
// 创建请求对象
var xml = new XMLHttpRequest();
```

```
// 初始化异步 POST 请求
xml.open("POST", "/some/url.cgi", true);

// 设置 content-type 首部，告知服务器如何解析我们发送的数据
xml.setRequestHeader(
    "Content-Type", "application/x-www-form-urlencoded");

// 保证浏览器发送的串行化数据长度正确 -
// 基于 Mozilla 的浏览器有时处理这个会碰到问题
if ( xml.overrideMimeType )
    xml.setRequestHeader("Connection", "close");

// 与服务器建立连接，并发送串行化数据
xml.send( serialize( data ) );
```

220

为了展开讨论先前的观点，我们来看一种不以“串行化”格式发送数据的情形（如代码清单10-6所示）。

代码清单10-6 将XML数据发送到服务器的例子

```
// 创建请求对象
var xml = new XMLHttpRequest();

// 初始化异步 POST 请求
xml.open("POST", "/some/url.cgi", true);

// 设置 content-type 首部，告知服务器如何解析我们发送的数据
xml.setRequestHeader( "Content-Type", "text/xml");

// 保证浏览器发送的串行化数据长度正确 -
// 基于 Mozilla 的浏览器有时处理这个会碰到问题
if ( xml.overrideMimeType )
    xml.setRequestHeader("Connection", "close");

// 与服务器建立连接，并发送串行化数据
xml.send( "<items><item id='one'/><item id='two'/></items>" );
```

这种发送大量数据的能力是非常重要的，它和 GET 请求依浏览器不同最多只能发几 KB 数据不同，而不用限制数据的长度。使用它你也可以实现许多不同通信协议，比如 XML-RPC 或 SOAP。

尽管如此，为简单起见，考虑 HTTP 响应时，不妨只限定在几种最常见，也最有用的数据格式的范围内。

10.1.2 HTTP 响应

创建、使用 `XMLHttpRequest` 要比其他简单的单向通信优越之处在于，它能够从服务器读取不同形式的文本数据。这包括 Ajax 的基石之一：XML。不过并没有规定必须用到 XML 才算一个 Ajax 应用程序，10.2 节介绍了其他的替代数据格式。

221

让我们先来看一个非常浅显的处理服务器响应数据的例子，如代码清单 10-7 所示。

代码清单10-7　与服务器建立连接并读取结果数据

```
// 创建请求对象
var xml = new XMLHttpRequest();

// 初始化异步 GET 请求
xml.open("GET", "/some/url.cgi", true);

// 在文档的状态更新时调用
xml.onreadystatechange = function(){
    // 等到数据完整加载
    if ( xml.readyState == 4 ) {

        // xml.responseXML 包含 XML 文档 (如果返回的是 XML)
        // xml.responseText 包含返回的文本
        // (如果返回的不是 XML)

        // 为避免内存泄漏，清理文档
        xml = null;
    }
};

// 建立到服务器的连接
xml.send();
```

在这个例子里，你能看到如何从 HTTP 响应中获取不同部分的数据。responseXML 和 responseText 这两个属性将分别包含对应格式的数据。比如，如果服务器返回的是 XML 文档，那 responseXML 里存储的就是 DOM 文档，而其他所有的响应和结果都存放在 responseText 中。

在对响应数据进行实际的处理、遍历与操作之前，我们先创建一个更健壮的 onreadystatechange 函数（代码清单 10-7 里出现过的），用来处理服务器错误和连接超时。

1. 处理错误

如果 XMLHttpRequest 对象有内建服务器错误处理机制的话，会大大节省我们的时间，不幸的是它没有。不过只要花点工夫，就可以创建自己的错误处理机制。你需要检查的是以下这些请求状况，以判断服务器在处理请求时是否遇到了问题：

- *成功响应代码*：可以通过HTTP规范里定义的响应状态码来检查错误，通过读取状态码，客户端能够知道服务器的情形。状态码在200到300之间的属于成功的请求。
- *未修改响应*：服务器返回的文档可能会加上“Not Modified（未修改）”的标记，也就是状态码304。说明服务器返回的数据和浏览器的缓存内容一致，并未修改过。这其实不算是个错误，因为客户端仍能读出正确的数据。

- *本地存储的文件*：如果你在本机上直接执行Ajax应用程序，而不通过Web服务器，就算请求成功了，也不会得到任何返回的状态码。这意味着，在执行本地文件且得不到状态码时，你应该把这种情况算做成功的响应。
- *Safari与未修改状态*：如果文档自上次请求（或者通过浏览器明确地发送一个IF-MODIFIED-SINCE首部，指定上次修改过的时间给服务器）未曾修改过，Safari返回的状态码会是"undefined"。这是一个比较怪异的情形，也让人难于调试。

考虑了上述这些情形，我们可以看看代码清单 10-8，实现了刚才概述的响应检查。

代码清单10-8 用于检查服务器HTTP响应的成功状态（Success State）的一个函数

```
// 检查 XMLHttpRequest 对象是否有 'Success' 状态。
// 此函数需要一个 XMLHttpRequest 对象作为参数。
function httpSuccess(r) {
    try {
        // 如果得不到服务器状态，且我们正在请求本地文件，认为成功
        return !r.status && location.protocol == "file:" ||

            // 所有 200 到 300 间的状态码表示成功
            ( r.status >= 200 && r.status < 300 ) ||

            // 文档未修改也算成功
            r.status == 304 ||

            // Safari 在文档未修改时返回空状态
            navigator.userAgent.indexOf("Safari") >= 0 &&
                typeof r.status == "undefined";
    } catch(e){}

    // 若检查状态失败，就假定请求是失败的
    return false;
}
```

检查 HTTP 响应的成功状态是非常重要的一步，不作检查可能会导致许多难以预料的结果，比如服务器返回的其实是 HTML 错误页面，而非 XML 文档。

我们将把这个函数集成在 10.3 节给出的完整 Ajax 方案中。

2. 检查超时

在 XMLHttpRequest 的默认实现中缺乏的另一个有用功能是如何判断请求超时。 223

这一功能的实现并不那么直接，但花点工夫（像上一节那样）判断请求的成功状态还是可行的。代码清单 10-9 展示了如何检查自己程序中的请求超时。

代码清单10-9 检查请求超时的一个例子

```
// 创建请求对象
var xml = new XMLHttpRequest();

// 初始化异步 GET 请求
xml.open("GET", "/some/url.cgi", true);

// 我们在请求后等 5 秒，然后放弃
var timeoutLength = 5000;

// 记录请求是否成功完成
var requestDone = false;

// 初始化一个 5 秒后执行的回调函数，用于取消请求（如果尚未完成的话）。
setTimeout(function(){
    requestDone = true;
```

```
}, timeoutLength);

// 监听文档状态的更新
xml.onreadystatechange = function()
{
    // 保持等待，直到数据完全加载，并保证请求并未超时
    if ( xml.readyState == 4 && !requestDone ) {

        // xml.responseXML 包含 XML 文档（如果返回的是 XML）
        // xml.responseText 包含返回的文本
        // （如果返回的不是 XML）

        // 为免内存泄漏，清理文档
        xml = null;
    }
};

// 与服务器建立连接
xml.send();
```

在考虑了服务器通信的细节，并处理了许多可能的出错后，现在可以来看看如何处理服务器
224 的响应数据了。

10.2　处理响应数据

迄今为止的所有例子中，你都只用一个符号来代替服务器的响应数据，理由很简单——服务器返回的数据格式有无限种可能性。不过现实中 XMLHttpRequest 只处理基于文本的数据格式。而且它在处理某些格式（XML）上要胜过另一些（JSON）。在本章里，你将看到服务器可能会返回的 3 种数据格式，并用客户端来读取并操作它们：

- XML：幸运的是，所有的现代浏览器都提供了原生的XML文档处理支持，自动将它们转换为可用的DOM文档
- HTML：和XML文档的区别在于，它通常以纯文本字符串的形式存在，存放一个HTML片段。
- JavaScript/JSON：这包括两种格式——原始的可执行JavaScript代码和JSON（JavaScript Object Notation，JavaScript对象表示）格式。

这 3 种格式都各有其适合的用途。比如有时返回 HTML 要比返回 XML 更有意义。

获取 HTTP 响应数据的重点是 XMLHttpRequest 对象的两个属性：

- responseXML：如果服务器返回的是XML文档，这个属性包含到预处理后DOM文档的引用，它是XML文档的表达。只在服务器明确指定其内容首部（content header）是"Content-type:text/xml"或类似的XML数据类型时，这一点才起作用。
- responseText：这一属性包含到服务器返回的原始文本数据的引用。HTML和JavaScript类型的数据都依赖这一方法来获得。

通过这两个属性，开发从 HTTP 响应中确定性地获取数据的通用函数就很简单了（甚或判断正在处理的数据是 XML 还是纯文本）。代码清单 10-10 展示的就是这样一个函数。

代码清单10-10 从HTTP服务器响应中解析正确数据的一个函数

```
// 从 HTTP 响应中解析数据的函数
// 有两个参数：一个 XMLHttpRequest 对象和一个可选参数——期望从服务器得到的数据类型
// 正确的值包括：xml, script, text, 或 html - 默认是 ""，根据 content-type 的首部得到
function httpData(r, type) {
    // 获取 content-type 首部
    var ct = r.getResponseHeader("content-type");

    // 若没有提供默认的类型，判断服务器返回的是否是 XML 形式
    var data = !type && ct && ct.indexOf("xml") >= 0;

    // 若是，获得 XML 文档对象，否则返回文本内容
    data = type == "xml" || data ? r.responseXML : r.responseText;

    // 若指定类型是 "script"，则以 JavaScript 形式执行返回文本
    if ( type == "script" )
        eval.call( window, data );

    // 返回响应数据（或为 XML 文档或为文本字符串）
    return data;
}
```

225

随着这个数据解析函数的完成，你现在拥有构建一套完整 Ajax 函数的所有组件了，可以实现常用的 Ajax 调用。我们在下一节能看到这样一个函数的完整实现。

10.3 完整的 Ajax 程序包

运用迄今为止学到的所有概念，你就可以构建出一个处理所有 Ajax 请求和相关响应的通用函数了。这个函数可以作为后续章节里你所有 Ajax 开发的基础，让你能迅速地从服务器查获信息。

这个完整的 Ajax 函数如代码清单 10-11 所示。

代码清单10-11 能够执行必要的Ajax相关任务的一个完整函数

```
// 执行 Ajax 请求的通用函数
// 带一个参数，是包含一系列选项的对象，这些选项在下面的注释中简述
function ajax( options ) {

    // 如果用户没有提供某个选项的值，就用默认值替代
    options = {
        // HTTP 请求的类型
        type: options.type || "POST",
        // 请求的 URL
        url: options.url || "",

        // 请求超时的时间
        timeout: options.timeout || 5000,
        // 请求失败、成功或完成（不管成功还是失败都会调用的）时执行的函数
        onComplete: options.onComplete || function(){},
        onError: options.onError || function(){},
        onSuccess: options.onSuccess || function(){},
```

226

```
        // 服务器将会返回的数据类型，这一默认值用于判断服务器返回的数据
        // 并作相应动作
        data: options.data || ""
};

// 创建请求对象
var xml = new XMLHttpRequest();

// 初始化异步请求
xml.open(options.type, options.url, true);

// 我们在请求后等待 5 秒，超时则放弃
var timeoutLength = options.timeout;

// 记录请求是否成功完成
var requestDone = false;

// 初始化一个 5 秒后执行的回调函数，用于取消请求（如果尚未完成的话）。
setTimeout(function(){
    requestDone = true;
}, timeoutLength);

// 监听文档状态的更新
xml.onreadystatechange = function(){
    // 保持等待，直到数据完全加载，并保证请求并未超时
    if ( xml.readyState == 4 && !requestDone ) {

        // 检查是否请求成功
        if ( httpSuccess( xml ) ) {

            // 以服务器返回的数据作为参数调用成功回调函数
            options.onSuccess( httpData( xml, options.type ) );

        // 否则就发生了错误，执行错误回调函数
        } else {
            options.onError();
        }

        // 调用完成回调函数
        options.onComplete();

        // 为避免内存泄漏，清理文档
        xml = null;
    }
};

// 建立与服务器的连接
xml.send();

// 判断 HTTP 响应是否成功
function httpSuccess(r) {
try {
    // 如果得不到服务器状态，且我们正在请求本地文件，认为成功
    return !r.status && location.protocol == "file:" ||
```

```
            // 所有 200 到 300 间的状态码表示成功
            ( r.status >= 200 && r.status < 300 ) ||

            // 文档未修改也算成功
            r.status == 304 ||

            // Safari 在文档未修改时返回空状态
            navigator.userAgent.indexOf("Safari") >= 0
                && typeof r.status == "undefined";
    } catch(e){}

        // 若检查状态失败，就假定请求是失败的
        return false;
    }

    // 从 HTTP 响应中解析正确数据
    function httpData(r,type) {
        // 获取 content-type 的首部
        var ct = r.getResponseHeader("content-type");

        // 若没有提供默认的类型，判断服务器返回的是否是 XML 形式
        var data = !type && ct && ct.indexOf("xml") >= 0;

        // 若是，获得 XML 文档对象，否则返回文本内容
        data = type == "xml" || data ? r.responseXML : r.responseText;

        // 若指定类型是 "script"，则以 JavaScript 形式执行返回文本
        if ( type == "script" )
            eval.call( window, data );

        // 返回响应数据 (或为 XML 文档或为文本字符串)
        return data;
    }
}
```

228

需要注意的是，请求与本页面在不同域下的数据是不可能的，因为所有的现代浏览器都有安全限制（避免别人试图获得你的个人信息）。现在你已经拥有了这个强大的函数，是时候通过一些例子试试你新开发的 Ajax 函数的能力了。

10.4 数据的不同用途

从根本上说，每次进行的简单 Ajax 请求之间并没有什么差别，不一样的是服务器返回的数据。根据希望实现的目的选择不同的数据格式完成任务会很有帮助。这也是接下来我将展示给你如何根据几种不同数据格式执行一些常见任务的原因。

10.4.1 基于 XML 的 RSS Feed

目前服务器返回的数据格式中最受欢迎的是 XML，这不是毫无理由的。所有现代的浏览器都对 XML 文档有直接支持，能即时将它们转换为 DOM 的表达形式。因为服务器把解析的重任

完成了，你所需要做的只是像遍历其他 DOM 文档一样的遍历它。不过需要注意的是，通常无法直接用 getElementById 函数来遍历远程返回的 XML 文档，这是因为浏览器对非 HTML 的 XML 文档没有预备好的唯一 ID 属性选择器，所以也就无法仅通过 ID 来选择纯 XML 元素。尽管如此，还是有办法在 XML 文档中做有效的遍历。

代码清单 10-12 展示了一个用返回的 XML 来给你的网站创建动态 RSS Feed 控件的例子。

代码清单10-12　从基于XML的远程RSS Feed加载项目的标题

229

```
<html>
<head>
    <title>Dynamic RSS Feed Widget</title>
    <!- 载入我们的通用 Ajax 函数 -->
    <script src="ajax.js"></script>
    <script>
        // 等待文档完整加载
        window.onload = function(){
        // 然后使用 Ajax 载入 RSS feed
        ajax({
            // RSS feed 的 URL
            url: "rss.xml",

            // 这是一个 XML 文档
            type: "xml",

            // 此函数会在请求结束后执行
            onSuccess: function( rss ) {
                // 我们将把所有 RSS 项目的标题都插入到 id 为 "feed" 的 <ol> 中
                var feed = document.getElementById("feed");

                // 获取 RSS XML 文档中所有的标题
                var titles = rss.getElementsByTagName("title");

                // 遍历每个匹配的标题
                for ( var i = 0; i < titles.length; i++ ) {
                    // 创建一个 <li> 元素来存放标题
                    var li = document.createElement("li");

                    // 将其内容设为项目标题
                    li.innerHTML = titles[i].firstChild.nodeValue;

                    // 将其添加到 DOM 的 <ol> 元素中
                    feed.appendChild( li );
                }
            }
        });
    };
    </script>
</head>
<body>
    <h1>Dynamic RSS Feed Widget</h1>
    <p>Check out my RSS feed:</p>
    <!--这里将插入 RSS feed -->
```

```
    <ol id="feed"></ol>
</body>
</html>
```

你可以看到，一旦把 Ajax 请求/响应过程的复杂性解决了，剩下的其实并不复杂。此外，考虑到浏览器使 XML 文档的遍历非常方便，XML 确实是一种从服务器端快速传输数据到客户端的好方法。

230

10.4.2 HTML 注入器

可以用 Ajax 实现的另一个有用的技巧是，动态地将 HTML 片段载入到文档中。这个技巧和前面讨论的 XML 文档的方法区别在于，你不必解析或遍历从服务器收到的数据，就可以把它马上插入到文档中。这种快刀斩乱麻的方法能让你的页面更新得简便而快捷。它的一个例子如代码清单 10-13 所示。

代码清单10-13 从远程文件中载入一段HTML代码，注入到当前网页

```
<html>
  <head>
      <title>HTML Sports Scores Loaded via Ajax</title>
      <!-- 载入我们的通用 Ajax 函数 -->
      <script src="ajax.js"></script>
      <script>
          // 等待文档完全加载
          window.onload = function(){

              // 然后使用  Ajax 载入 RSS feed
              ajax({
                  // HTML sports score 的 URL
                  url: "scores.html",

                  // 是 HTML 文档
                  type: "html",

                  // 此函数会在请求结束后执行
                  onSuccess: function( html ) {
                      // 我们将插入到 id 为 'scores' 的 div 中
                      var scores = document.getElementById("scores");

                      // 将新的 HTML 注入文档
                      scores.innerHTML = html;
                  }
              });
          };
      </script>
</head>
<body>
    <h1> HTML Sports Scores Loaded via Ajax </h1>
     <!--这里插入 Sports Score -->
    <div id="scores"></div>
</body>
```

231

```
</html>
```

对这种动态 HTML 技巧来说，最重要的一点在于，你仍然可以在服务器端代码里使用所有应用程序级别的模板工具，这使得模板代码集中也易于维护。

尽管这很简单，但也不要小看这种载入 HTML 文件的功能，它是创建更具用户响应 Web 应用程序的最便捷的方法。

10.4.3　JSON 与 JavaScript：远程执行

最后（在第 13 章 wiki 的例子中将会遇到）我们要讨论的数据格式是从 JSON 数据串到纯 JavaScript 代码的转换。对 JSON 化数据的转换可以作为从服务器到客户端传输 XML 文档的一种轻量级替代。此外，从服务器提供纯 JavaScript 代码则是一个构建动态多用户 Web 应用程序的好方法。为了简化，让我们看看如何将一个远程 JavaScript 文件载入你的应用程序中，如代码清单 10-14 所示。

代码清单10-14　动态载入并执行一个远程JavaScript文件

```
<html>
<head>
    <!-- 载入我们的通用 Ajax 函数 -->
    <script src="ajax.js"></script>
    <script>
        // 载入一个远程 Javascript 文档
        ajax({
            // JavaScript 文件的 URL
            url: "myscript.js",

            // 强制以 JavaScript 的形式执行
            type: "script"
        });
    </script>
</head>
<body></body>
</html>
```

10.5　小结

尽管看似简单，Ajax Web 应用程序的概念却是非常强大的。通过动态地把其他信息载入到基于 JavaScript 的运行中应用程序里，你可以创建响应更为丰富的程序界面。

在本章里，你了解到 Ajax 的基本概念，包括 HTTP 请求和响应的规范、错误处理、数据格式化与解析。我们同时得到了一个可供重用的通用函数，用它可以很简单地给任何 Web 应用程序赋予动态特性。你会在接下来的 3 章里使用这个函数来构建一系列的动态 Ajax 交互。

232

第 11 章

用Ajax改进blog

Ajax 技术使我们获得的能力之一是，在静态的网页上提供额外的用户交互。这意味着你可以在提供无中断的用户体验的同时修改静态页面的工作方式。

可以接受这类改进的领域之一是网络日志（简称 blog，又称博客）。若仅从纯数据的角度看，网络日志不过是日志项的一个列表，其中每项都包含一个正文段、标题和到全文的链接。不过浏览旧日志或检查新日志的功能就非常有限了。

在本章里，你能看到两种用基于 Ajax 技术的 JavaScript 代码来改进传统网络日志的方法。其一用于快速滚动一长串日志项列表，而无需离开当前页面；其二在不需要反复刷新页面的情况下就能看到新日志项的出现。

11.1 永不终止的 blog

要改进 blog 的第一部分是，能在甚至不需要点击导航链接的情况下回滚到所有存档。blog 或其他以时间为单位记录的内容站点的常见特性之一是导航查阅旧的日志。通常在页面底部会有一个“下一页”或“上一页”链接，给用户提供存档的导航。

你将在这里看到如何用 Ajax 把整个过程环绕一种方法来实现。在实现之前，需要作如下假定：

- 你有一个网页，其中有许多按时间排序的条目。
- 当用户滚动到页面底部时，意味着他们希望读到更多先前的条目。
- 你有一个可以读取所有条目的数据源。在这里我们要用的WordPress blog软件(http://wordpress.org/)就对此支持得很好。

这个脚本的效果是，只要用户滚动到了页面底部附近，就会自动载入其他文章，让用户能继续滚动并浏览存档，使他们产生一种页面永不终止的错觉。它将使用 WordPress 网志软件提供的功能来构建。如果你用的就是基于 WordPress 的 blog，可以简单地把这个脚本放入其中，给它添加这个新功能。

11.1.1 blog 的模板

你将使用默认 WordPress 安装时提供的一个称作 Kubrik 的基本模板来起步，这个模板很受欢迎。图 11-1 展示了一个基本 Kubrik 主题页面。

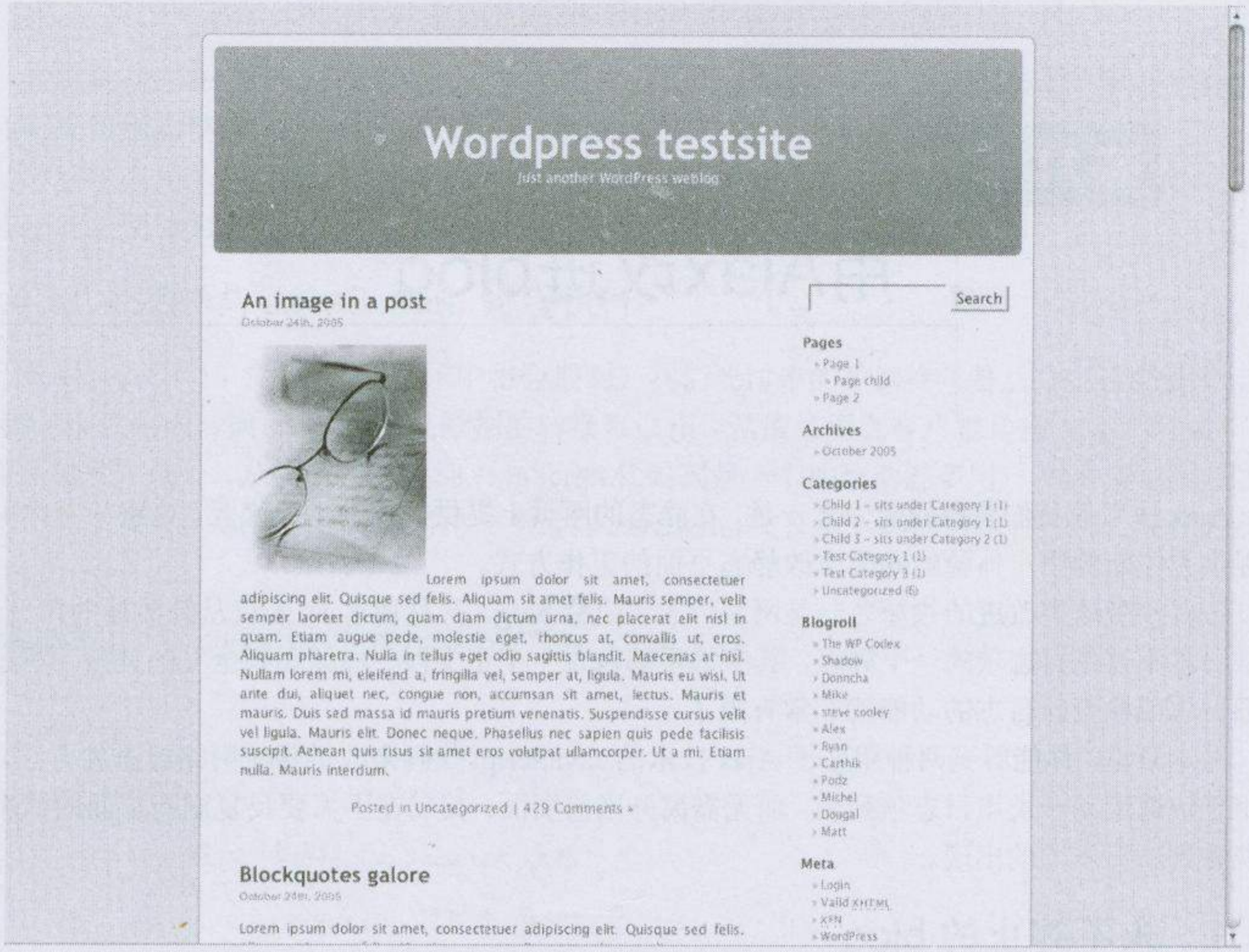


图11-1　WordPress的默认Kubrik主题

你可以发现，这个页面有一个主栏位、一个标题栏和一个侧栏。主栏位[①]是最重要的区域，你可以在此看到网志文章，并添加新的信息。让我们看看构成这个 blog 结构化、简化后的 HTML，如代码清单 11-1 所示。

代码清单11-1　Kubrik主题和WordPress生成的HTML的简化版本

```
<html>
<head>
    <title>Never-ending Wordpress</title>
    <script>
    <!-- 我们的脚本放在这里 -->
    </script>
</head>

<body>
    <div id="page">
        <div id="header">
            <!-- 标题栏内容 -->
```

234

① 原文为标题栏，应该是主栏位。——译者注

```
</div>
<div id="content">

    <!-- 第一篇文章 -->
    <div class="post">
        <!-- 文章的标题 -->
        <h2><a href="/test/?p=1">Test Post</a></h2>
        <small>October 24th, 2006</small>

        <div class="entry">
        <!-- 文章的内容 -->
        </div>

        <p class="postmetadata">
            <a href="/test/?p=1#comments">Comments</a>
        </p>
    </div>

    <!-- 更多文章... -->

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

注意，所有的网志项目都包含在 ID 为"content"的 中。此外，所有文章的结构都用一种专门的格式规划。然后，你需要构建一套简单的 DOM 函数来提取这个网志页面的数据。代码清单 11-2 展示了完成这个页面里需要的 DOM 操作。

代码清单11-2 添加HTML以完成页面的DOM操作

```
// 我们要把新文章载入到 id 为 "content" 的 <div> 中
var content = document.getElementById("content");

// 我们将遍历 RSS feed 中所有的文章
var items = rss.getElementsByTagName("item");
for (var i = 0; i < items.length; i++) {

    // 让我们从每篇 feed 文章中解析出链接、标题和描述数据
    var data = getData( items[i] );

    // 创建一个新的 <div> 用来包裹这篇文章
    var div = document.createElement("div");
    div.className = "post";

    // 创建文章标题
    var h2 = document.createElement("h2");

    // 这将包含 feed 的标题和到文章的链接
    h2.innerHTML = "<a href='" + data.link + "'>" + data.title + "</a>";

    // 将它添加到封装它的 <div> 中
    div.appendChild( h2 );
```

235

```
    // 现在创建一个 <div> 来存放比较长的部分，即文章内容
    var entry = document.createElement("div");
    entry.className = "entry";

    // 将内容添加到 <div> 内部
    entry.innerHTML = data.desc;
    div.appendChild( entry );

    // 最后，让我们添加一个有返回链接的底部
    var meta = document.createElement("p");
    meta.className = "postmetadata";

    var a = document.createElement("a");
    a.href = data.link + "#comments";
    a.innerHTML = "Comment";
    meta.appendChild( a );

    div.appendChild( meta );

    // 将这个新项目插入文档
    content.appendChild( div );

}
```

然而，如果你不了解处理的是什么数据，所有这些 DOM 操作都没什么意义。下一节里你将看到从服务器传给你的数据，以及如何使用这些 DOM 操作将它插入文档中。

11.1.2　数据源

WordPress 提供了访问网志文章的一套简单方法，即可以通过一个默认 RSS feed 阅读 10 篇最新的文章。不过仅通过 RSS feed 是不够的，因为你可能还需要访问所有的文章，甚至返回到网站本身。这时你可以用一个隐藏特性来实现这个功能。

236

在 WordPress 中，RSS feed 的 URL 通常类似这样：/blog/?feed=rss，不过只要再加上一个参数，/blog/?feed=rss&paged=N，你就可以访问到 blog 任意时间以前的历史记录。当 N 等于 1 时得到最新的 10 篇文章，等于 2 时得到它们之前的 10 篇。代码清单 11-3 展示了包含文章数据的 RSS feed 是什么样子的。

代码清单11-3　WordPress返回的XML RSS feed，包含10篇格式化好的文章

```
<?xml version="1.0" encoding="UTF-8"?>
<rss version="2.0">

<channel>
    <title>Test Wordpress Web log</title>
    <link>http://someurl.com/test/</link>
    <description>Test Web log.</description>
    <pubDate>Fri, 08 Oct 2006 02:50:23 +0000</pubDate>
    <generator>http://wordpress.org/?v=2.0</generator>
```

```
    <language>en</language>

    <item>
        <title>Test Post</title>
        <link>http://someurl.com/?p=9</link>
        <pubDate>Thu, 07 Sep 2006 09:58:07 +0000</pubDate>
        <dc:creator>John Resig</dc:creator>
        <category>Uncategorized</category>
        <description><![CDATA[ 这里存放文章内容……]]></description>
    </item>

    <!-- 大量其他项目 -->
</channel>
</rss>
```

通过分析 RSS feed，你获得了一个格式良好的 XML 文件可供使用（而且 JavaScript 非常适于遍历）。代码清单 11-4 展示了用于遍历这份 RSS XML 文档并从中解析相关数据的必要代码。

代码清单11-4　从XML RSS Feed中解析文章信息

```
// 我们将遍历 RSS feed 中的每篇文章
var items = rss.getElementsByTagName("item");

for (var i = 0; i < items.length; i++) {
    // 从 RSS feed 的 item 元素中解析出标题、描述和链接
    var title = elem.getElementsByTagName("title")[0].firstChild.nodeValue;
    var desc = elem.getElementsByTagName("description")[0].firstChild.nodeValue;
    var link = elem.getElementsByTagName("link")[0].firstChild.nodeValue;
}
```

237

在保证数据来源完整，而插入 HTML 文档的结构也稳定的情况下，现在可以用 Ajax 请求和一些基本的事件检测方法把代码黏合在一起了。

11.1.3　事件检测

要激活你的脚本，用户需要完成的主要交互是滚动到页面的底部，所以不管浏览器的视口（viewport）移动到哪里，你必须能判断出它是否在页面的底部。

处理这个的脚本相对简单，你只需要给窗口的滚动事件绑定一个简单的事件处理器。每当用户移动页面的视口（有可能是移动到了页面底部附近），这个事件处理器都能让你知道。所以你只需要使用第 7 章定义的一套简单方法来判断用户的视口到底在哪里。pageHeight（判断整个页面有多高），scrollY（获知当前视口的顶部滚动到了哪里）和 windowHeight（获知视口有多高）。如代码清单 11-5 所示。

代码清单11-5　判断用户视口的位置

```
// 我们要根据当前用户在页面中所处的位置判断是否应该载入更多内容
window.onscroll = function(){
    // 检查视口在页面中的位置
    if ( curPage >= 1 && !loading &&
        pageHeight() - scrollY() - windowHeight() < windowHeight() ) {
```

```
            // 用 Ajax 请求来获取 RSS XML feed
        }
    };
```

现在你拥有所需的所有组件了，最后一步是使用 Ajax 请求来获取数据、无缝衔接所有部分。

11.1.4　请求

整个程序的核心要和一段 Ajax 请求结合在一起，并动态载入一部分文章或条目，以便插入页面中。要发送的请求很简单：建立到某个 URL（指向下一部分文章的 URL）的 HTTP GET 连接，并获取这个 URL 指定的 XML 文档。代码清单11-6使用第10章的完整 Ajax 函数就实现了这一点。

238

代码清单11-6　载入一部分新文章的Ajax请求

```
// 用我们方便的 ajax() 函数来载入文章
ajax({

    // 我们只不过是请求一个简单网页，所以就用 GET
    type: "GET",

    // 要获得的 RSS feed 就是一个 XML 文件
    data: "xml",

    // 获得第 N 个页面的 RSS feed。在我们受此载入这个页面时处在第
    // 1 页，所以从 2 开始往回追溯
    url: "./?feed=rss&paged=" + ( ++curPage ),

    // 等待成功获得 RSS feed
    onSuccess: function( rss ){
        // 通过 DOM 来遍历 RSS XML 文档
    }

});
```

构建好发出页面请求的方法后，你就可以把所有功能结合在一起成为一个紧凑的程序包，简单地放在 WordPress blog 里。

11.1.5　结果

将 DOM 构建代码和 RSS XML 遍历代码合在一起，就得到了这个程序最简单的形式，而一旦加上滚动事件的检测和 Ajax 请求，你就得到了对 blog 的一项很吸引人的改进——不离开当前页面的情况下持续滚动所有 blog 文章的能力。代码清单 11-7 展示了用这个功能改进 WordPress blog 所需的完整代码。

代码清单11-7　WordPress blog里添加永不终止的页面功能所需的JavaScript代码

```
// 记录我们目前所在的页面编号
```

```
var curPage = 1;

// 确保我们不会同一时间重复载入同一个页面
var loading = false;

// 我们要根据当前用户在页面中所处的位置判断是否应该载入更多内容
window.onscroll = function(){

    // 我们要在尝试载入新内容前验证几件事情:
    // 1) 必须确保不在内容的最后一页。
    // 2) 必须确保现在不是正在载入新文章。
    // 3) 只在滚动到靠近页面底部才尝试载入新文章。
    if ( curPage >= 1 && !loading &&
        pageHeight() - scrollY() - windowHeight() < windowHeight() ) {

        // 记住我们现在开始载入新文章了。
        loading = true;

        // 用我们方便的 ajax() 函数来载入文章
        ajax({

            // 我们只不过是请求一个简单网页，所以就用 GET
            type: "GET",

            // 要获得的 RSS feed 就是一个 XML 文件
            data: "xml",

            // 获得第 N 个页面的 RSS feed。在我们受此载入这个页面时处在第
            // 1 页，所以从 2 开始往回追溯
            url: "./?feed=rss&paged=" + ( ++curPage ),

            // 等待成功获得 RSS feed
            onSuccess: function( rss ){

                // 我们要把新文章载入到 ID 为 "content" 的 <div> 中
                var content = document.getElementById("content");

                // 我们将遍历 RSS feed 中所有的文章
                var items = rss.getElementsByTagName("item");

                for (var i = 0; i < items.length; i++) {
                    // 将这个新项目插入文档
                    content.appendChild(makePost( items[i] ) );
                }

                // 如果 XML 文档中已无任何项目，我们必须后退到最早的位置
                if (items.length == 0) {
                    curPage = 0;
                }
            },

            // 只要请求完成，我们就可以允许再次尝试载入新的内容了
            onComplete:function(){
                loading = false;
```

239

240

```
            }
        });
    }
};

// 为一篇文章创建复杂 DOM 结构的函数
function makePost( elem ) {
    // 让我们从每篇 feed 文章中解析出链接、标题和描述数据
    var data = getData( elem );

    // 创建一个新的 <div> 用来包裹这篇文章
    var div = document.createElement("div");
    div.className = "post";

    // 创建文章标题
    var h2 = document.createElement("h2");

    // 这将包含 feed 的标题和到文章的链接
    h2.innerHTML = "<a href='" + data.link + "'>" + data.title + "</a>";

    // 将它添加到包裹它的 <div> 中
    div.appendChild( h2 );

    // 现在创建一个 <div> 来存放比较长的部分，文章内容
    var entry = document.createElement("div");
    entry.className = "entry";

    // 将内容添加到 <div> 内部
    entry.innerHTML = data.desc;
    div.appendChild( entry );

    // 最后，让我们添加一个有返回链接的底部
    var meta = document.createElement("p");
    meta.className = "postmetadata";
    meta.innerHTML = "<a href='" + data.link + "#comments'>Comment</a>";
    div.appendChild( meta );

241     return div;
}

// 从 DOM 元素中解析数据的简单函数
function getData( elem ) {
    // 返回数据是格式化良好的对象
    return {
        // 从 RSS feed 的 <item> 元素中解析出标题、描述和链接
        title:elem.getElementsByTagName("title")[0].firstChild.nodeValue,
        desc:elem.getElementsByTagName("description")[0].firstChild.nodeValue,
        link:elem.getElementsByTagName("link")[0].firstChild.nodeValue
    };
}
```

将这些代码添加到你的 WordPress 的 header 模板中应该足以达到类似图 11-2 的效果了。注意滚动条会变得很小，这说明已经动态载入了额外的文章。

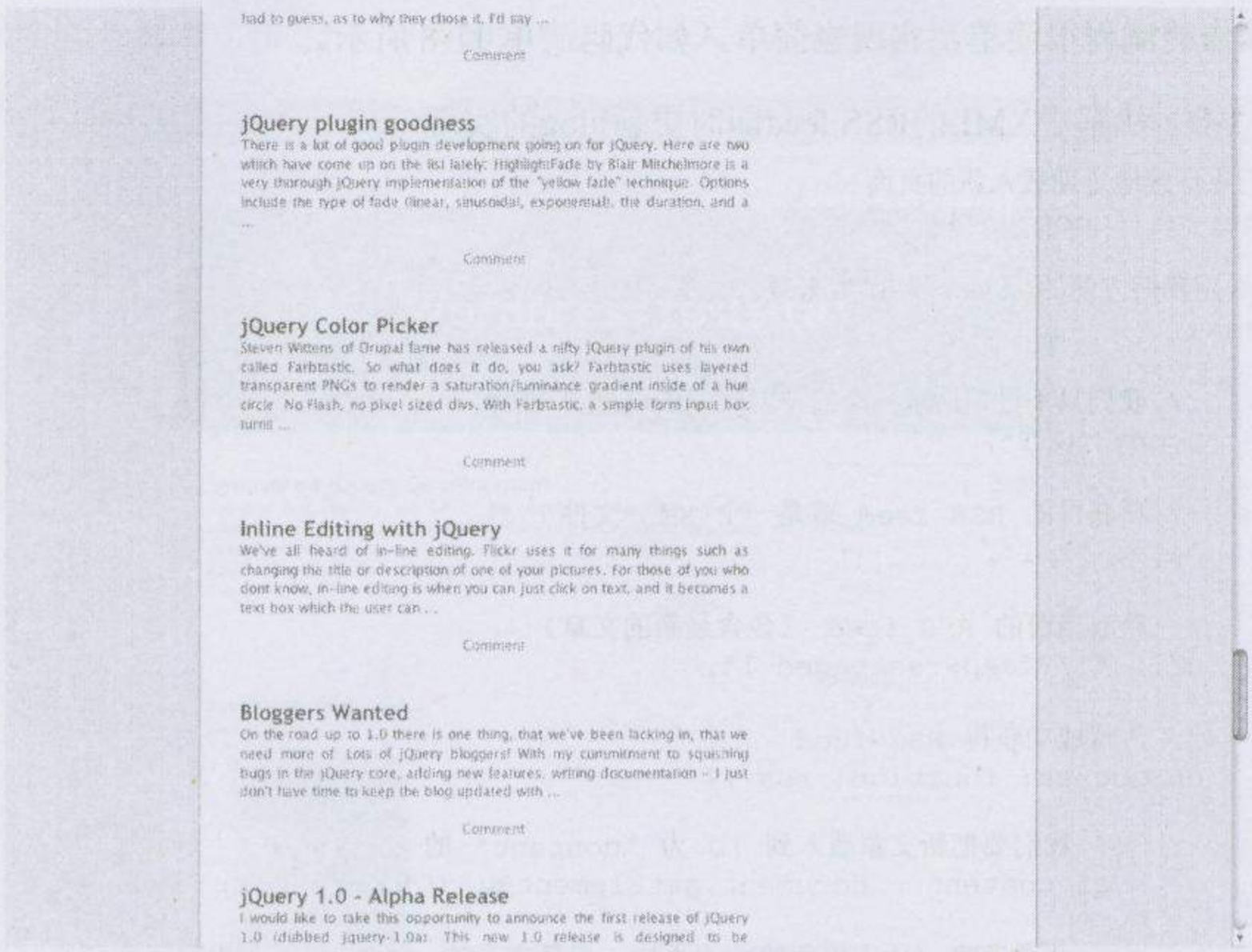


图11-2 额外的内容随向下滚动页面被载入到视口

242

动态内容载入是 Ajax 的常见用途之一。可以肯定的是，这种技巧能简化用户浏览页面的过程，还能减轻服务器的负荷。请求额外信息的应用程序当然并不限于网络日志。基于 Web 的应用程序都能够从这种技术中大大获益，你可以在第 12 章看到更多关于这方面的内容。

在下一节里，你会见到使用同一种 Ajax 技巧来动态载入内容的另一个例子，在这个例子里，能给浏览者提供实时的网志体验。

11.2 实时网志

已经完成了所有比较困难的工作：动态构建文章、获取文章数据和解析数据，现在可以直接在 WordPress 网志里体验到这些工作的另一应用了。

新闻和更新的即时性是传统网络日志比较缺乏的。用户看到的是一个静态页面，列出了最新 blog 中的文章，如果他们需要看到更新，就必须在浏览器中刷新页面。不过许多时候，网志的主人（blogger）都希望能即时告知用户一段信息，就算用户当时还在阅读当前页面。理论上说，应该允许用户载入页面，保持开启，偶尔（从别的窗口或者 Tab）切换回来，就能见到新的文章。

这又是一个 Ajax 技术的应用，你可以使用在上一节提及的技巧来从 RSS XML feed 里载入新内容。下面列出了给普通 blog 创建实时网志体验的必要操作：

- 以固定间隔时间（比如每分钟一次），获取网络日志中最近发布的文章列表。
- 找出未曾显示的文章。
- 将这些文章添加到页面的顶部。

这个操作的流程很简单，实现也简单，如代码清单 11-8 所示。

代码清单11-8　从基于XML的RSS feed即时更新blog的实现

```
// 我们将持续地定期载入新的页面
setInterval(function(){

    // 用我们方便的 ajax() 函数来载入文章
    ajax({

        // 我们只不过是请求一个简单网页，所以就用 GET
        type: "GET",

        // 要获得的 RSS feed 就是一个 XML 文件
        data: "xml",

        // 获取当前的 RSS feed （包含最新的文章）
        url: "./?feed=rss&paged=1",

        // 等待成功获得 RSS feed
        onSuccess: function( rss ){

            // 我们要把新文章载入到 ID 为 "content" 的 <div> 中
            var content = document.getElementById("content");

            // 获得当前 （还未更新的） 页面中，最新文章的 URL （以避免出现冗余）
            var recentURL = content.getElementsByTagName("h2")[0].firstChild.href;

            // 我们要遍历 RSS feed 中的所有文章
            var items = rss.getElementsByTagName("item");

            // 我们将把所有新的项目放入一个单独的数组中
            var newItems = [];

            // 遍历每个项目
            for ( var i = 0; i < items.length; i++ ) {

                // 如果找到了“旧”文章，强制停止循环
                if ( getData(items[i]).link == recentURL )
                    break;

                // 将这个新项目添加到临时数组中
                newItems.push(items[i]);

            }

            // 反向遍历所有新的项目，以保证他们插入页面的顺序正确
            for (var i = newItems.length-1; i >= 0; i--) {
                // 将新项目插入文档
                content.insertBefore(makePost( newItems[i] ), content.firstChild );
            }

        }
    });

// 每分钟载入一次新页面
}, 60000 );
```

243

244

如果你把这个脚本（和 11.1 节的脚本一起）放入 WordPress 模板中，就能得到类似图 11-3 的结果。

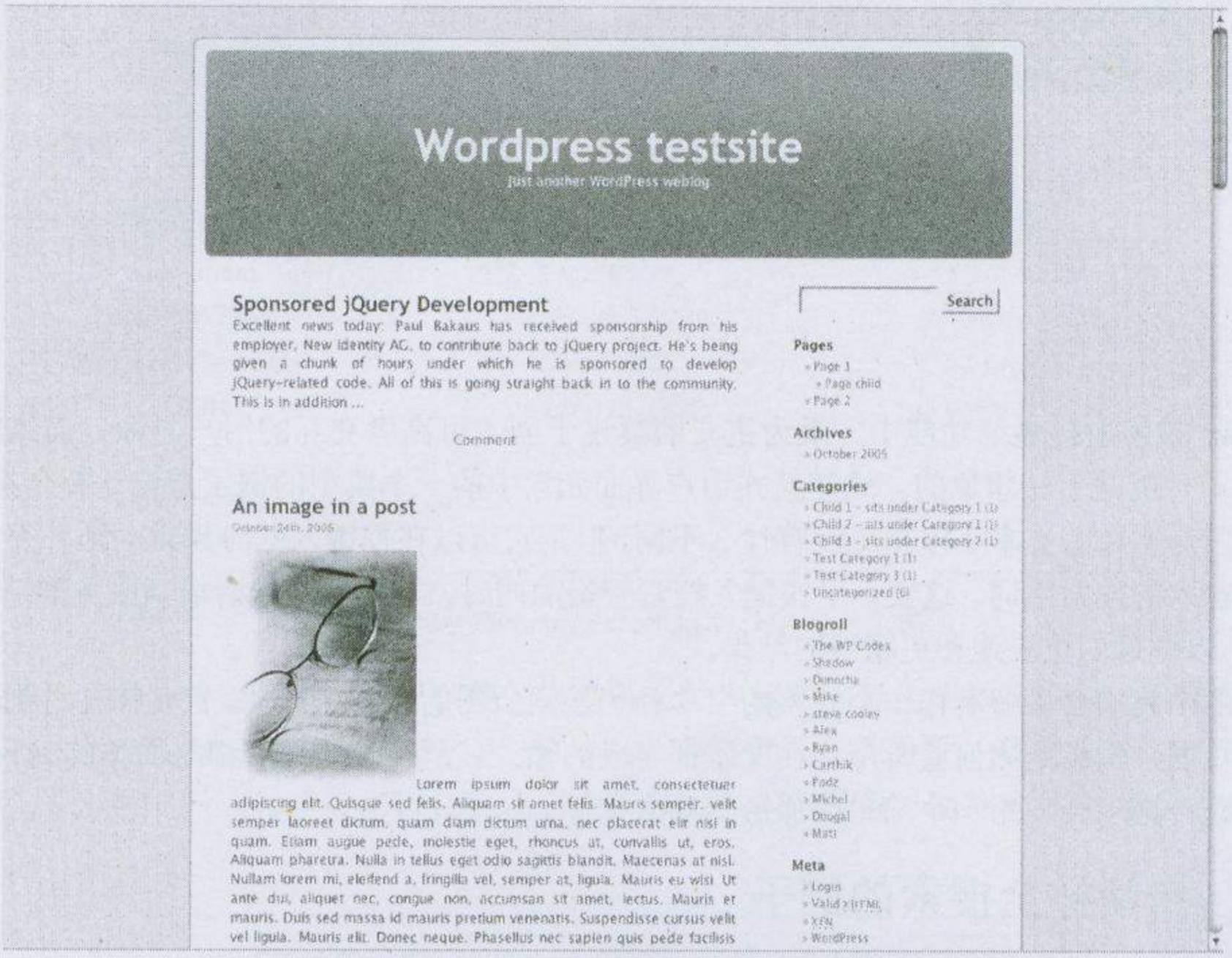


图11-3 在用户没有刷新页面的情况下WordPress自动把新文章插入到旧文章的前面

通过这个改进，可以有效地把一个简单的 blog 转换为一个实时网志平台。如果在参加某个会议时需要即时更新 blog，你就可以把这段脚本加入站点，读者就能迅速获得更新，而不需要手动刷新页面了。

11.3 小结

本章讨论的最重要概念是，Ajax 相关的技术让你能为传统的静态应用程序构想新的运作方式。因为处理 XML 文档并将其转换为可用的 HTML 是如此简单，在你自己的应用程序里实现起来也是完全可能的。

245

本章给传统的 WordPress blog 平台构建了两个新改进。第一，你把传统的链接加分页的导航方法替换为了在用户浏览时动态载入文章的方法。第二，如果在用户阅读页面时发表了新文章，会被即时添加出来，而且并不影响用户的阅读。这两个改进都能使浏览体验变得更具动态性、更平滑，而不受多页间断的影响。

下一章我们将构建一个先进的 Ajax 功能：自动补全的搜索。

246

第 12 章

自动补全的搜索

12

Ajax 技术所提供的功能中，最为重要的莫过于创建可高度交互的用户界面，而这种界面在 Ajax 之前可能是无法想象的。这些新式用户界面元素中的一个典型的例子是自动补全搜索字段。这种字段与普通的文本字段相比没有什么不同，但是它可以在你键入的时候自动地补充完成你可能想要输入的搜索字词。这表示了在键入搜索查询的同时，客户端发送请求到服务器端（在后台运行），以便返回更快速和更精确的结果。

本章中我们会讲解制作一个完整的自动补全搜索所需的每一个组件。首先你会看到如何构建并生成页面，然后是如何监听用户在文本框字段的输入。所有这些内容都与简单的 Ajax 请求挂钩，而这个请求会使用一个简单的服务器端数据库。

12.1 自动补全搜索的例子

自动补全的搜索的字段能以部分不同的方式出现。比如，Google 的搜索框有一个自动补全的版本叫做 Google Suggest（http://www.google.com/Webhp?complete=1）。当你在这个字段开始输入搜索查询，它会给你显示其他用户常用的搜索，而这些搜索会以你所输入的相同字词开头（如图 12-1 所示）。

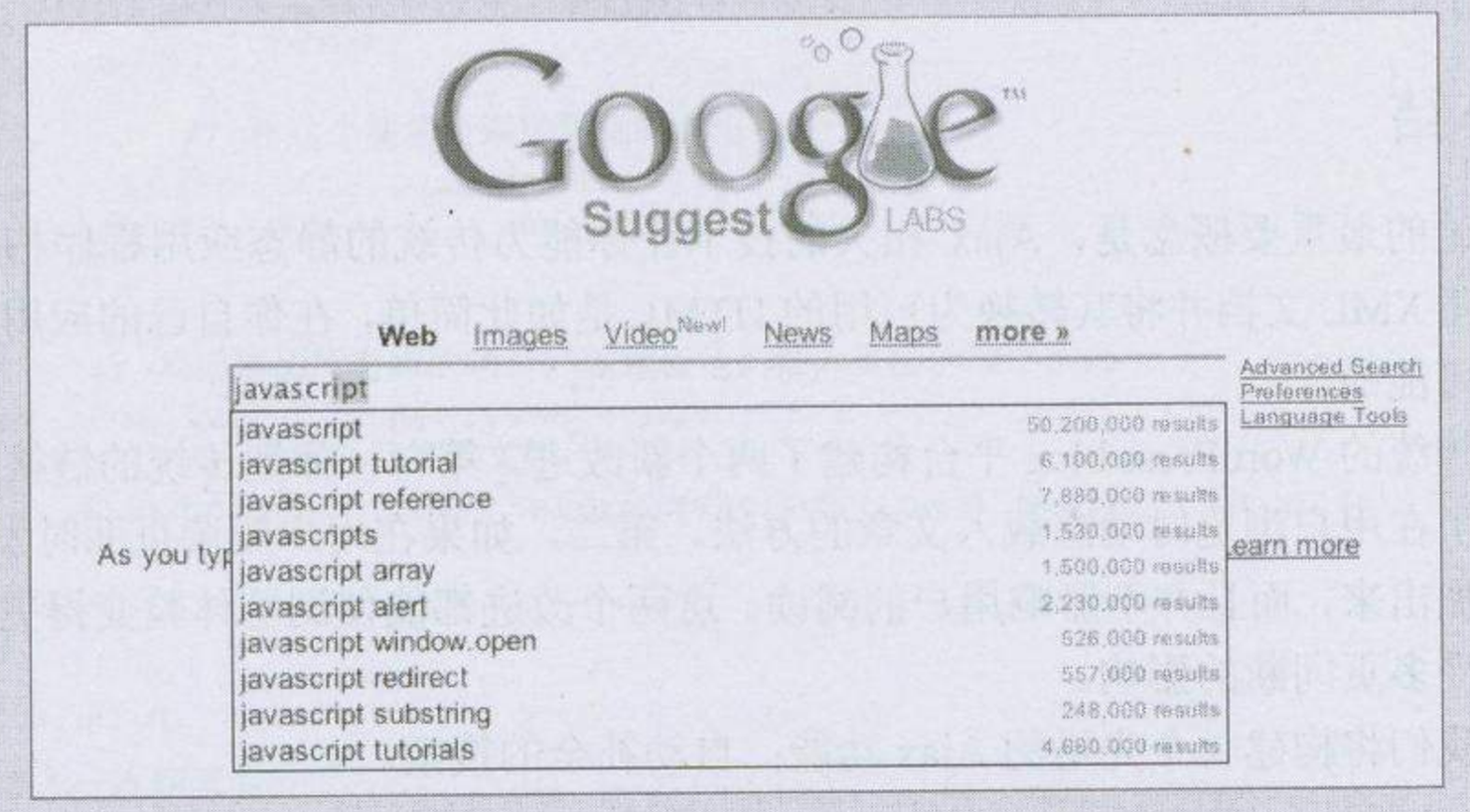


图12-1 经典的自动补全搜索的视觉效果例子

另一个受欢迎的例子是 Instant Domain Search（http://instantdomainsearch.com/）。这个有特色的应用程序让你可以在输入域名的时候就知道该域名是否可交易。这跟 Google 实现的自动补全搜索截然不同，它本身自动补全了请求。这表示，在你输入域名进行查询时，服务器在后台自动处理了你的请求，并给予相应的回复。例子如图 12-2 所示。

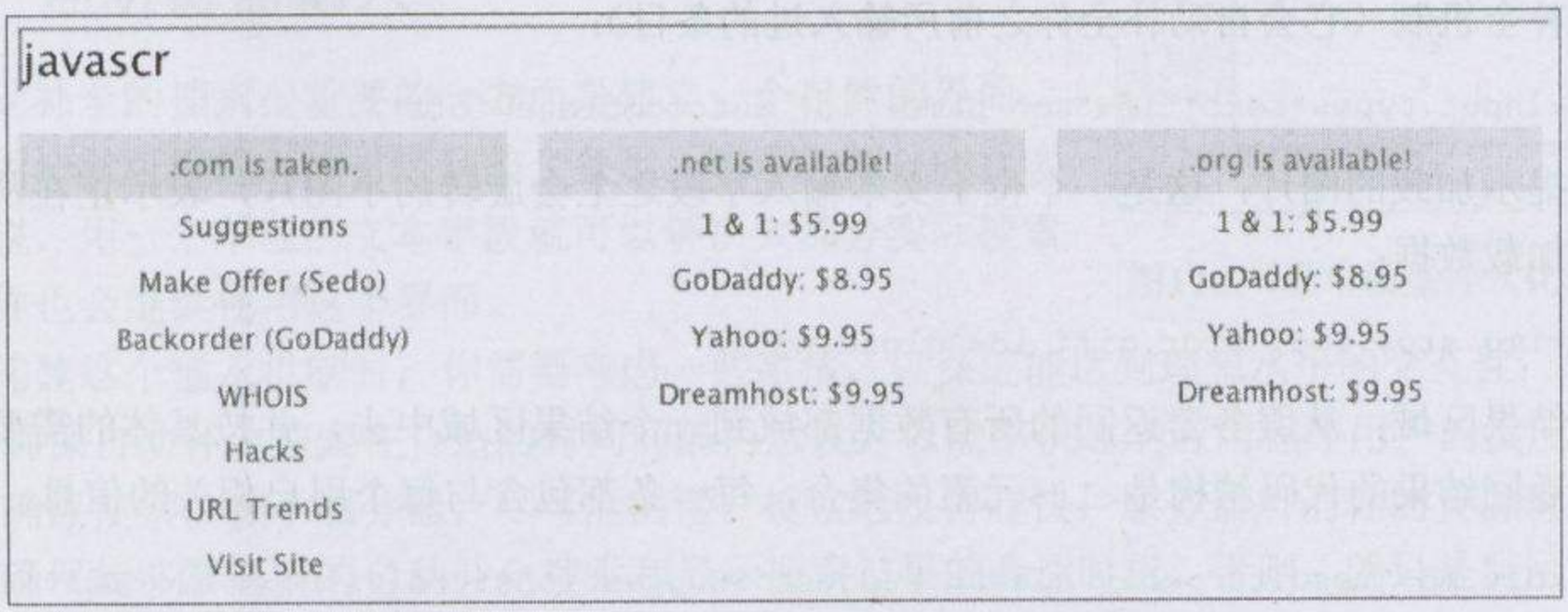


图12-2 Instant Domain Search自动补全的一个例子

最后一个例子，你接下来要自己创建的自动补全的案例跟它最为相似，是由在线书签服务商 del.icio.us(http://del.icio.us/)提供的自动补全的结构。它提供了一种手段，通过它你就可以使用特定的词语标记链接，而这种手段可以让你在一个单独的文本字段内自动补全多个词语（如图 12-3 所示）。

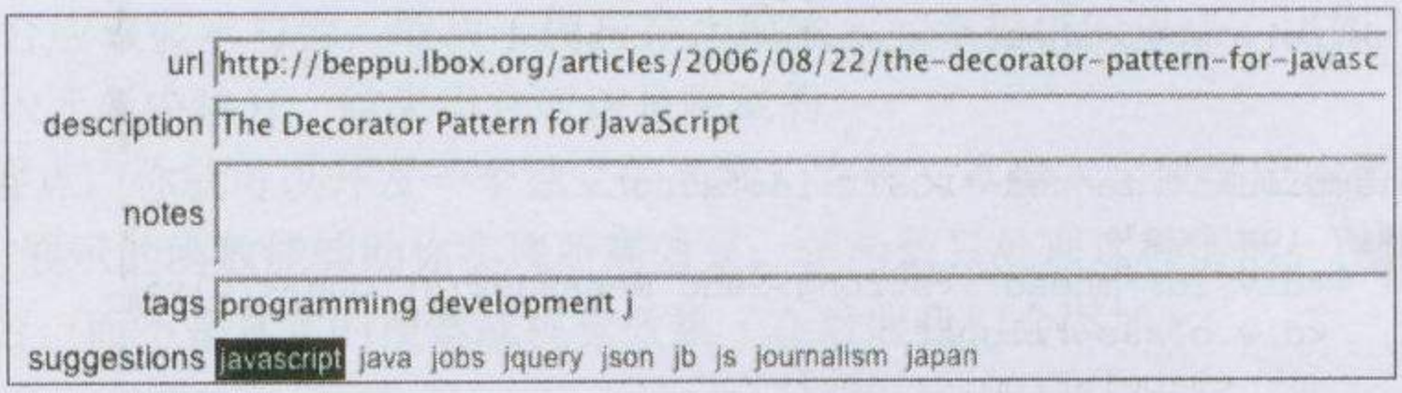


图12-3 运行中的del.icio.us自动补全的例子，它补充完一个新标签

接下来制作你自己的自动补全搜索。它是一个简单的表单，用在网站中发送一条信息给一组朋友。这个自动补全字段行为与 del.icio.us 非常相似，在表单上会有一个输入用户名的字段，这个字段使用 Ajax 在后台自动补全。这样你就能够自动补全每一位朋友的用户名，以存储在中心数据库所发送的以逗号分隔的用户名信息为准。实现这个效果的例子如图 12-4 所示。

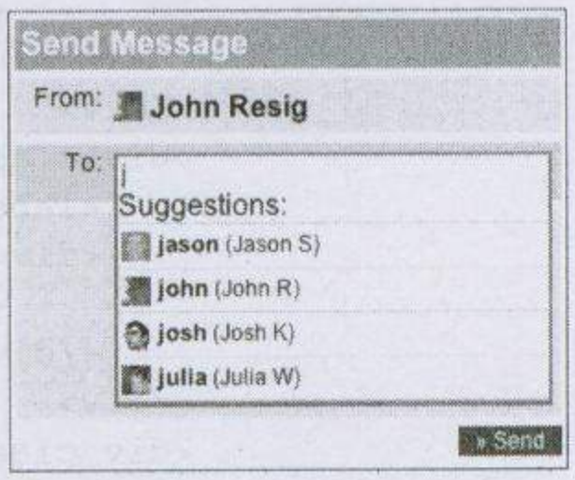


图12-4 用户名自动补全的例子，基于所输入的单个字母

248

12.2 制作页面

制作你自己的自动补全搜索字段的第一步是，构建一个简单

的表单作为整个构架的基础。这个页面的结构就像一个在网站内发送消息到一组用户的简单表单。跟普通的发送消息表单比起来，你还需要在页面中包含3个重要的方面（由一个To字段和正在发送消息的区域）：

- 文本字段：需要确定它的autocomplete特性设置为off。这可以禁止浏览器默认的自动补全机制（它会自动补全你之前所输入过的条目）：

```
<input type="text" id="to" name="to" autocomplete="off"/>
```

- 提示加载的图片：这是一个位于文本输入字段之上会旋转的小图片，表示正在从服务器加载数据：

```
<img src="indicator.gif" id="qloading"/>
```

- 结果区域：从服务器返回的所有数据都放到一个结果区域中去，并按具体的需要显示。返回结果的代码结构是<li>元素的集合，每一条都包含与每个用户相关的信息。

```
<div id="results"><div class="suggest">Suggestions:</div><ul></ul></div>
```

加载提示器和结果区域都要通过JavaScript引入。页面的完整HTML如代码清单12-1所示。

代码清单12-1　发送消息给用户的自动补全表单的完整HTML

```
<html>
<head>
      <script src="dom.js"></script>
      <script src="delay.js"></script>
      <script src="script.js"></script>
      <link rel="stylesheet" href="style.css"/>
</head>
<body>
    <form action="" method="POST" id="auto">
        <div id="top">
            <div id="mhead"><strong>Send Message</strong></div>
            <div class="light">
                <label>From:</label>
                <div class="rest from">
                    <img src="icons/john_icon.jpg"/>
                    <strong>John Resig</strong>
                </div>
            </div>
            <div class="query dark">
                <label>To:</label>
                <div class="rest">
                    <input type="text" id="to" name="to" autocomplete="off"/>
                </div>
            </div>
            <div class="light"><textarea></textarea></div>
            <div class="submit"><input type="submit" value="&raquo; Send"/></div>
        </div>
    </form>
</body>
</html>
```

249

图 12-5 展示了该页面充分样式化后的视觉截屏。

现在已经架好表单并准备好让用户输入，接下来就是在用户名文本字段上监听用户的输入信息并正确地反馈结果。

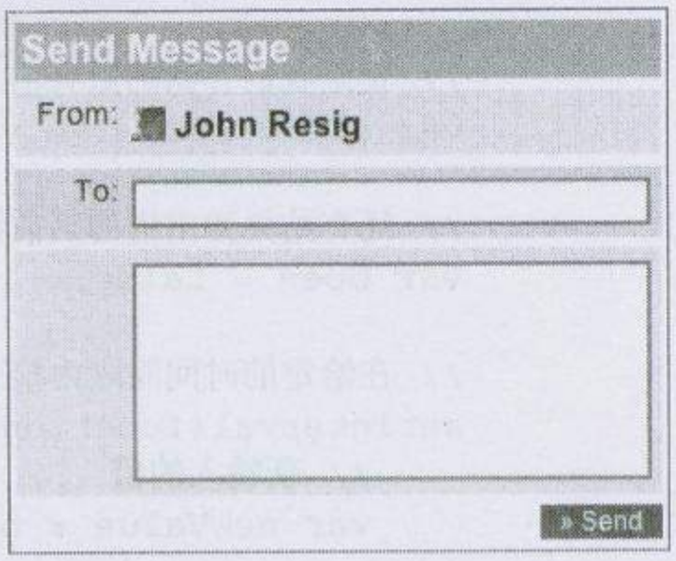


图12-5 表单模型样式化后的效果

250

12.3 监听键盘输入

自动补全的搜索很重要的一方面是建立一个自然的界面，通过它用户就可以输入并扩展到整个条目。对于大部分的搜索输入来说，用一个单独的文本字段就可以解决大部分实际搜索界面。你也会继续使用这个界面。

在构建这个输入机制时，你需要考虑一些事情，以保证能达到理想水平的交互性：

- 确保自动补全搜索在合适的时间间隔内触发，以能够快速响应用程序用户的反应。
- 确保搜索依赖于服务器，尽可能的慢。搜索触发得越快，服务器所消耗的资源也就越多。
- 确保能够掌握新的自动补全搜索和显示搜索结果的合适时机。否则，就只是显示旧结果，并隐藏当前结果。

安排好这些之后，现在可以定义需要实现的那些更为精确的用户交互了：

- 自动补全结果的显示应该由用户在文本字段所输入的内容为准。此外，应该提供少量的字符以触发搜索（避免过度的模糊搜索）。
- 结果搜索应该在常规的时间间隔内触发（避免服务器由于用户的快速输入而超载），但也仅在文本输入框内容已经变化了的情况了。
- 结果集合应该显示与否，取决于用户在当前输入元素提供的焦点。比如，如果用户把焦点从输入元素中移开，结果集合应该是隐藏的。

记住这些要点，你就可以开发一个独立的函数，它为文本输入字段绑定你需要的交互行为。代码清单 12-2 展示的函数能够助你实现所需效果。该函数只处理这种情形：在何地触发搜索或者显示结果与否，而不是真实的搜索或视觉效果（在后面我们会添加）。

代码清单12-2 为文本输入字段绑定一个能自动补全搜索的函数

```
function delayedInput(opt) {
    // 用户输入新内容之前等待的时间间隔
    opt.time = opt.time || 400;

    // 触发请求的最小字符数
    opt.chars = opt.chars != null ? opt.chars : 3;

    // 结果应该弹出时或者可能产生一个新的请求时触发的回调
    opt.open = opt.open || function(){};

    // 结果应该关闭时要触发的回调
    opt.close = opt.close || function(){};

    // 需要考虑字段焦点问题，以开启或关闭结果的弹出
    opt.focus = opt.focus !== null ? opt.focus : false;
```

251

```
// 记录我们开始的原有值
var old = opt.elem.value;

// 还有结果弹出框的当前状态（打开或者关闭）
var open = false;

// 在给定的时间间隔内检查输入框是否有变化
setInterval(function(){
    // 新输入的值
    var newValue = opt.elem.value;

    // 已输入字符的长度
    var v = newValue.length;

    // 快速检查自最后一次检查输入后，值是否被改变
    if ( old != newValue ) {

        //如果输入的字符数不足，并且“弹出框”当前是开启状态
        if ( v < opt.chars && open ) {

            // 关闭显示
            opt.close();

            // 记录关闭的状态
            open = false;

        // 否则，输入的字符达到了要求的最小数量，同时也多于一个字符
        } else if ( v >= opt.chars && v > 0 ) {

            // 打开当前值的结果弹出框
            opt.open( newValue, state );

            // 记录弹出框的当前打开状态
            open = true;
        }
        // 为后面保存当前值
        old = newValue;
    }
}, opt.time );

// 监听键盘的敲击
opt.elem.onkeyup = function(){
    // 如果键盘敲击后左边再也没有字符，关闭结果的弹出框
    if ( this.value.length == 0 ) {
        // 关闭弹出框
        opt.close();

        // 记录它的关闭状态
        open = false;
    }
};

// 同时检查用户的焦点（处理打开 / 关闭结果弹出框）
```

```
if ( opt.focus ) {
    // 监听用户何时从移开输入框
    opt.elem.onblur = function(){
        // 如果当前是打开的
        if (open ) {
            // 关闭弹出框
            opt.close();

            // 并储存它的关闭状态
            open = false;
        }
    }

    // 监听用户何时把焦点移回弹出框
    opt.elem.focus = function(){
        // 如果有值，并且当前是关闭状态
        if ( this.value.length != 0 && !open ) {
            // 重新打开弹出框，但值设置为空
            // （这会通知'open' 函数不要重新从服务器抓取结果了，只需重新打开弹出框）
            opt.open( '', open );

            // 并记录弹出框的打开状态
            open = true;
        }
    };
}
}
```

253

代码清单12-3展示了在自动补全的实现中如何使用这个delayedInput函数监听用户的输入。

代码清单12-3 在你的自动补全实现中使用通用的delayedInput()函数

```
// 在我们的输入框中初始化延迟输入检查
delayedInput({
        // 我们绑定到文本输入字段上
        elem: id("to"),

        // 我们在输入了1个字符后就开始搜索
        chars: 1,

        // 一旦文本字段失去焦点，关闭结果弹出框
        focus: true,

        // 处理结果弹出框应该打开的情况
        open: function(q,open){
            // 获取逗号分隔词列表中的最后一个词
            var w = trim( q.substr( q.lastIndexOf(',')+1, q.length ) );

            // 保证我们至少在处理一个词
            if ( w ) {
                // 显示加载提示的动画
                show( id("qloading") );

                // 加载并处理从服务器传来的结果
```

```
        }
    },

    // 当需要关闭弹出框时
    close: function(){
        // 隐藏结果集合
        hide( id("results") );
    }
});
```

有了这个监听用户输入的通用函数，现在要把这个函数跟服务器端脚本结合起来了，它伺服着可载入到网站中的用户数据。

12.4　抓取结果

制作一个自动补全的搜索，接下来需要掌握的基础方面是载入显示给用户的数据。通过 Ajax 载入数据并不是必须的（在这个特定实现中也是如此），反之，它可以作为数据结构并在运行时
254 （run time）中载入到页面即可。

自动补全的实现还需要完成一件事情：用户。这些用户名将在更多的上下文信息（包括用户全名和用户图标）中显示。有了这些，这就变得十分容易了，只需从服务器返回一大块 HTML（以一些<li>元素的形式）即可，而这些 HTML 中包含了你所需的所有匹配用户的信息。

代码清单 12-4 展示了一个简单 Ajax 调用，它是把从服务器传来的 HTML 片段，加载到结果弹出框所必需的。

代码清单12-4　加载HTML片段（包含用户信息）到自动补全结果集合中的Ajax请求

```
// 为新数据发起一个请求
ajax({
    // 发起一个CGI脚本简单的GET请求，它会返回包含一块<li>元素的
    type: "GET",
    url: "auto.cgi?to=" + w,

    // 监听何时返回HTML
    onSuccess: function(html){
        // 把它插入结果的<ul>元素中去
        results.innerHTML = html;

        // 并隐藏加载动画
        hide( id("qloading") );

        // 处理结果……
    }
});
```

在这段代码中需要注意的是，你是从一个名为 auto.cgi 的服务器端应用程序中取得 HTML 结果的，它带一个参数，这个参数正是你查询的当前文本（极有可能是用户名的部分）。auto.cgi 脚本使用 Perl 编写，如代码清单 12-5 所示。它会检索一个小型数据集合以找到相匹配的，将一段长的 HTML 片段返回给所有匹配的用户。

代码清单12-5 检索匹配用户的一个简单的Perl脚本

```
#!/usr/bin/perl

use CGI;

# 从进入的查询字符串中获取'q'参量
my $cgi = new CGI();
my $q = $cgi->param('to');
# 我们有限的“数据库”包含了5个用户，分别是用户名和全名
my @data = (
    {
        user => "bradley",
        name => "Bradley S"
    },
    {
        user => "jason",
        name => "Jason S"
    },
    {
        user => "john",
        name => "John R"
    },
    {
        user => "josh",
        name => "Josh K"
    },
    {
        user => "julia",
        name => "Julia W"
    }
);

# 确保我们输入正确的HTML头
print "Content-type: text/html\n\n";

# 现在我们来“查询”整个数据
foreach my $row (@data) {

    # 查找匹配我们自动补全搜索的用户
    if ( $row->{user} =~ /$q/i || $row->{name} =~ /$q/i ) {

        # 如果用户匹配，输出必要的HTML
        print qq~<li id="$row->{user}">
            <img src="icons/$row->{user}_icon.jpg"/>
            <div>
                <strong>$row->{user}</strong> ($row->{name})
            </div>
        </li>~;

    }

}
```

255

256

这个返回的结果只是包含匹配用户相关的<li>元素的 HTML 片段而已。代码清单12-6展示了搜索字母 j 的结果。

代码清单12-6　一段从服务器返回的HTML，它是部分不同的用户的描述

```
<li id="jason">
      <img src="icons/jason_icon.jpg"/>
      <div>
          <strong>jason</strong> (Jason S)
      </div>
</li><li id="john">
      <img src="icons/john_icon.jpg"/>
      <div>
          <strong>john</strong> (John R)
      </div>
</li><li id="josh">
      <img src="icons/josh_icon.jpg"/>
      <div>
          <strong>josh</strong> (Josh K)
      </div>
</li><li id="julia">
      <img src="icons/julia_icon.jpg"/>
      <div>
          <strong>julia</strong> (Julia W)
      </div>
</li>
```

有了搜索小型数据集合的服务端，返回 HTML 片段，并注入到站点中，下一个步骤就是给用户添加一些方法来导航结果了。

12.5　导航结果列表

最后，用户已经在文本字段键入了一些文本，并且从服务器加载了一些结果，是时候给用户添加方法以导航这些返回的结果集合了。在这个自动补全的搜索实现中，将用两种不同的方式来导航结果：键盘导航和鼠标导航。

12.5.1　键盘导航

通过键盘来导航结果似乎是最应该实现的功能，因为现在需要用户输出用户名。在整个过程
257 中要让用户将手指一直保持在键盘之上，这是十分有必要的。

需要支持使用 Tab 键来确定当前选中的用户，并在返回的结果集中使用上下键选择不同的用户。代码清单 12-7 展示了如何实现这个功能。

代码清单12-7　可导航按键的事件处理器

```
// 检查文本字段中的输入
id("to").onkeypress = function(e){
      // 获取结果集合中的所有用户
      var li = id("results").getElementsByTagName("li");
```

```
        // 如果 敲击了[Tab]或者[Enter]If the [TAB]
        if ( e.keyCode == 9 || e.keyCode == 13 ) {
            // 把用户添加到文本输入字段中去

        // 如果敲击了上箭头键
        } else if ( e.keyCode == 38 )
            // 选择上一个用户，或者最后一个用户（假如我们处于开始的话）
            return updatePos( curPos.previousSibling || li[ li.length - 1 ] );

        // 如果敲击了下箭头键
        else if ( e.keyCode == 40 )
            // 选择下一个用户，或者第一个用户（加入我们处于结尾的话）
            return updatePos( curPos.nextSibling || li[0] );
};
```

12.5.2 鼠标导航

与键盘导航不同，所有的鼠标导航都必须在服务器每一次返回的结果集中动态绑定。鼠标导航的前提是每一次移动鼠标到其中一个用户<li>元素上时，这个元素就变成了当前“选中”元素。如果你点击它，与<li>相关的用户名会追加到文本字段中去。实现这个功能的必要代码例子如代码清单12-8所示。

代码清单12-8 绑定鼠标导航事件到用户<li>元素上

```
// 当用户鼠标在li上，则高亮当前这个用户
li[i].onmouseover = function(){
        updatePos( this );
};

// 当用户点击时
li[i].onclick = function(){
        // 把用户添加到输入框中
        addUser( this );
        // 并把焦点再次返回输入框返回HTML
        id("to").focus();
};
```

有了在合适位置的导航后，现在你已经完成自动补全搜索的主要组件了。这些成果会在下一节中得以应用和展示。

12.6 最终成果

自动补全搜索的所有必要组件都准备好了：监听用户输入、与服务器通信和结果的导航。是时候把它们结合起来放到页面中去了。代码清单12-9展示了自动补全搜索完整的、最终的JavaScript代码。

代码清单12-9 自动补全搜索的JavaScirpt完整代码

```
domReady(function(){
```

```
// 确保在开始的时候结果弹出框是关闭的
hide( id("results") );

// 跟踪记录哪些用户已经键入了用户名
var doneUsers = {};

// 跟踪记录当前选中的用户名
var curPos;

// 检查文本字段的输入
id("to").onkeypress = function(e){
    // 获取结果集合中的所有用户
    var li = id("results").getElementsByTagName("li");

    // 如果 敲击了[Tab]或者[Enter]
    if ( e.keyCode == 9 || e.keyCode == 13 ) {
        // 重置当前的用户列表
        loadDone();

        // 如果当前选中的用户并不在选中的用户列表当中，把它添加到输入框中
        if ( !doneUsers[ curPos.id ] )
            addUser( curPos );

        // 阻止键盘触发它的默认行为
        e.preventDefault();
        return false;

    // 如果敲击了上箭头键
    } else if ( e.keyCode == 38 )
        / 选择上一个用户，或者最后一个用户（假如我们处于开始的话）
        return updatePos( curPos.previousSibling || li[ li.length - 1 ] );

    // 如果敲击了下箭头键
    else if ( e.keyCode == 40 )
        // 选择下一个用户，或者第一个用户（加入我们处于结尾的话）
        return updatePos( curPos.nextSibling || li[0] );
};
// 初始化输入框的延迟输入检查
delayedInput({
    // 我们绑定到文本输入字段上
    elem: id("to"),

    // 我们在输入了1个字符后就开始搜索最终成果
    chars: 1,
    // 当文本字段失去焦点，关闭结果弹出框
    focus: true,

    // 处理结果弹出框应该打开的情况
    open: function(q,open){
        // 获取逗号分隔词列表中的最后一个词
        var w = trim( q.substr( q.lastIndexOf(',')+1, q.length ) );
        // 保证我们至少在处理一个词
        if ( w ) {
            // 显示加载提示的动画
```

259

260

```
show( id("qloading") );
// 确保当前没有用户被选中
curPos = null;

// 获取支撑所有结果的<ul>元素
var results = id("results").lastChild;

// 并清空它
results.innerHTML = "";
// 为新数据发起一个请求
ajax({
    // 发起一个CGI脚本简单的GET请求，它会返回包含一块<li>元素的
    type: "GET",
    url: "auto.cgi?q=" + w,

    // 监听何时返回HTML
    onSuccess: function(html){
        // 把它插入结果的<ul>元素中去
        results.innerHTML = html;

        // 并隐藏加载动画
        hide( id("qloading") );

        // 重新初始化我们业已灌入的用户列表
        loadDone();

        // 遍历所有返回的用户
        var li = results.getElementsByTagName( "li" );
        for ( var i = 0; i < li.length; i++ ) {

            // 如果我们已经添加了用户，删除它的<li>元素
            if ( doneUsers [ li[i].id ] )
                results.removeChild( li[i--] );

            // 否则，为其绑定一些事件
            else {

                // 当用户鼠标在li上，则高亮当前这个用户
                li[i].onmouseover = function(){
                    updatePos( this );
                };

                // 当用户点击时
                li[i].onclick = function(){
                    // 把用户添加到输入框中
                    addUser( this );

                    // 并把焦点再次返回输入框
                    id("q").focus();
                };
            }
        }
}

// 遍历用户li列表
```

261

```
            li = results.getElementsByTagName( "li" );

            // 如果没有剩余用户（我们添加完了）
            if ( li.length == 0 )
                // 隐藏结果
                hide( id("results") );

            else {

                // 为剩余用户添加'odd'的class以便区隔
                for ( var i = 1; i < li.length; i += 2 )
                    addClass( li[i], "odd" );

                // 把当前选中的用户设置为第一个
                updatePos( li[0] );

                // 然后显示结果
                show( id("results") );
            }
        }
    });
  }
},

// 当需要关闭弹出框时
close: function(){
    // 隐藏结果集合
    hide( id("results") );
}
});

function trim(s) {
    return s.replace(/^\s+/,"").replace(/\s+$/, "");
}
// 改变当前被选中用户的高亮
function updatePos( elem ) {
    // 更新当前选中元素的位置
    curPos = elem;

    // 获取所有的用户<li>元素
    var li = id("results").getElementsByTagName("li");

    // 从当前选中者中删除'cur'的class
    for ( var i = 0; i < li.length; i++ )
        removeClass( li[i], "cur" );

    // 并高亮当前用户项
    addClass( curPos, "cur" );

    return false;
}

// 重新初始化业已输入的用户列表
function loadDone() {
```

```
        doneUsers = {};

        // 遍历用户列表（以逗号分隔）
        var users = id("q").value.split(',');
        for ( var i = 0; i < users.length; i++ ) {

            // 在一个对象散列表中保存用户名（作为键）
            doneUsers[ trim( users[i].toLowerCase() ) ] = true;
        }
    }

    // 把用户添加到文本输入字段中
    function addUser( elem ) {
        // 文本字段的值
        var v = id("q").value;

        // 在输入框的最后添加用户名
        // 确保它地被正确的分号分隔
        id("to").value =
            ( v.indexOf(',') >= 0 ? v.substr(0, v.lastIndexOf(',') + 2 ) : '' )
            + elem.id + ", ";

        // 把用户名添加到主列表（避免完全重载列表）
        doneUsers[ elem.id ] = true;

        // 删除用户<li>元素
        elem.parentNode.removeChild( elem );

        // 并隐藏结果列表
        hide( id("results") );
    }
});
```

263

图 12-6 演示了最终结果的样子。

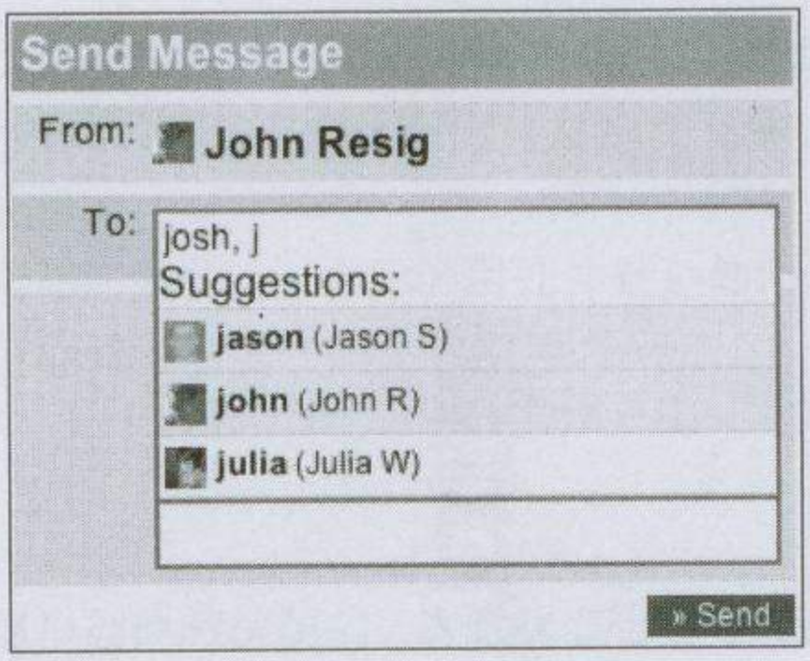


图12-6 实际运行的自动补全搜索，正在完成第二个用户名的屏幕截图

最终成果令人印象深刻，而且十分有用。自动补全搜索的基本概念并不复杂也不难以实现，但结合起来就可以创造一些十分漂亮的交互效果。

12.7　小结

对于任何一个应用程序来说，自动补全的搜索都是十分不错的补充。几乎在所有的文本条目交互中，它都能提供辅助用户输入的机会。

本章涵盖了为一个简单表单的用户名输入构建自动补全搜索的所有方面。重点讲解了如何从用户捕抓文本条目、发送请求到服务器端脚本和返回相关数据，并允许用户导航返回的结果。利用这些概念，会给你的用户带来更多的益处。

本章例子的一个实际演示可在本书的网站上 http://www.jspro.org/找到，并附有设置服务器端
264 的详细材料。源代码可以在 Apress 网站 http://www.apress.com/的 Source Code/Download 部分找到。

第 13 章

Ajax wiki

13

随着各种不同的服务器端 MVC 框架被摆上 Web 应用程序开发的台面（比如 Ruby on Rails 和 Django），我认为是时候学习一些不同的 Web 应用程序开发的编程语言了。本章探索了浏览器内一个简单的 wiki。

13.1 wiki 是什么

根据 wikipedia.org（目前为止最受欢迎的基于 wiki 的网站）的描述，wiki 允许来访者非常快速并容易地添加、删除或者编辑所有的内容，有时甚至不需要注册。这种方便快捷的交互和操作使得 wiki 成为一种协作书写的高效工具。

此外，它还提供了一组格式化工具，让你可以定制 wiki 的条目。图 13-1 演示的是一个 wiki 的主页面，它已经由不同的用户修订过。

这个案例的独特之处在于，wiki 引擎的逻辑纯粹是用 JavaScript 编写的，使用 JavaScript 代码向服务器发送数据请求。这个案例还介绍了开发一个现代 Web 应用程序所必需的基本概念，尽管我们并没有着眼于传统应用程序上服务器的任何逻辑。

这个应用程序可分解为 3 个部分：客户端、服务器端和数据库（大部分 Ajax 应用程序都是这样），每一部分我们都会发散式讲解。客户端负责与用户交互并处理界面，而服务器端则负责维护客户端和数据源之间的通信。

为更进一步理解这个应用程序是什么、有什么用处并且是如何运作的，本章将带你学习它的每一个功能特性，并解释如何能够让它的代码为你所用。

13.2 对话数据库

wiki 的每一页都可能是用户创建和编辑的。这意味着你必须把用户贡献的内容保存起来，以备不时之需。最好的解决方案是建立一个简单的数据库，可以保存所有的数据。为了让 JavaScript 代码化的客户端能够与数据库对话，你需要开发一些代码位于代码服务器端，起连接客户端与 wiki 数据库的作用。每一次数据库的查询流程如图 13-2 所示。这个过程与各种形式的语句执行（如 SELECT、INSERT 等）是一致的。

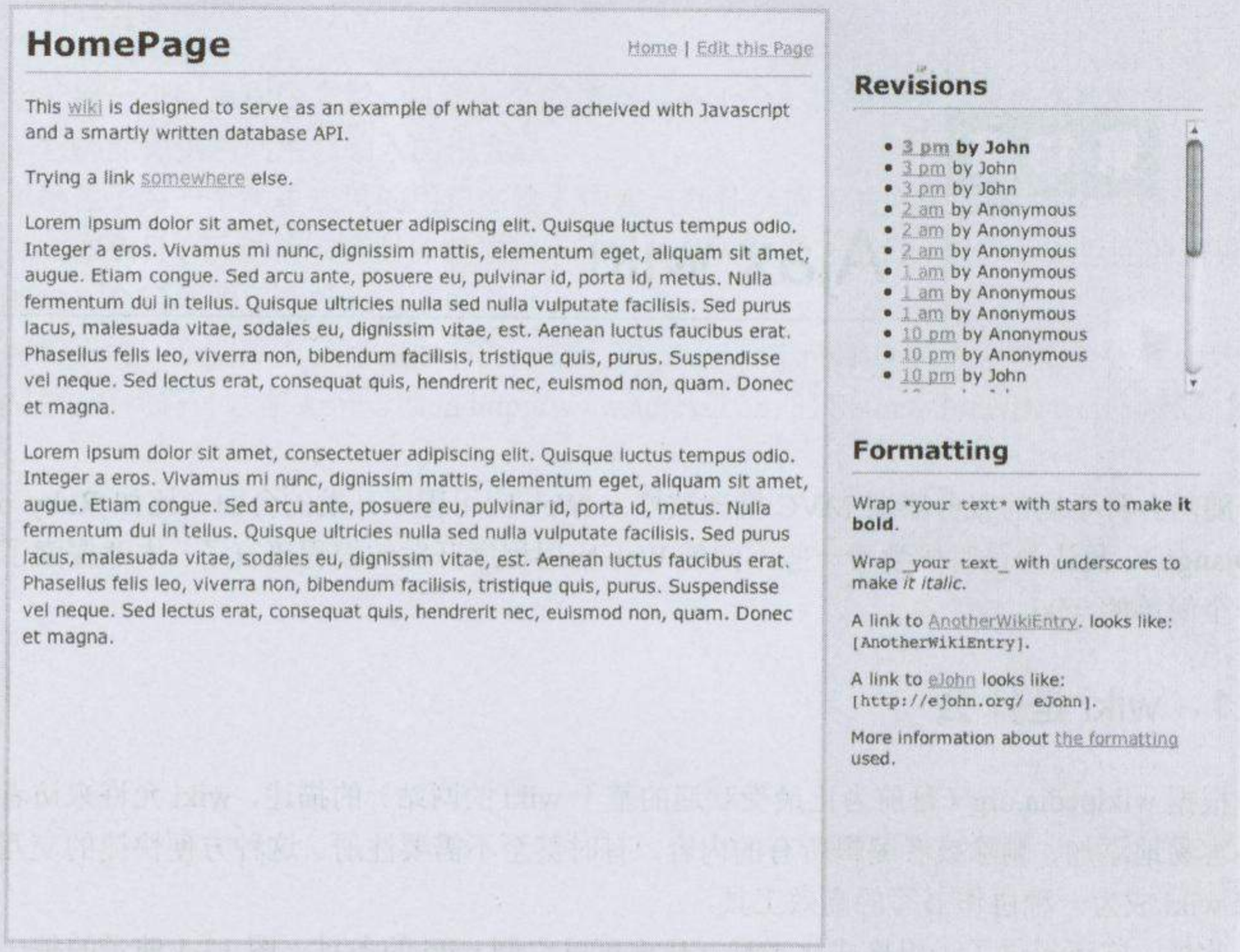


图13-1　运行中的 wiki 截图

客户端与数据库需要进行两次通信。第一次是查询数据库，以获取某个具体页面的所有修订。第二次是把新的修订插入到数据库中。这两者的流程是一样的，这样就可以容易地构建一个通用的、简单的通信层，这个层可以用在这里或者其他地方的 JavaScript 应用程序中。拥有这个通用通信层的另一个有益之处，在于你可以容易地在整个通信流程（见图 13-2）步步为营，并了解 JavaScript 是怎么与数据库通信的。

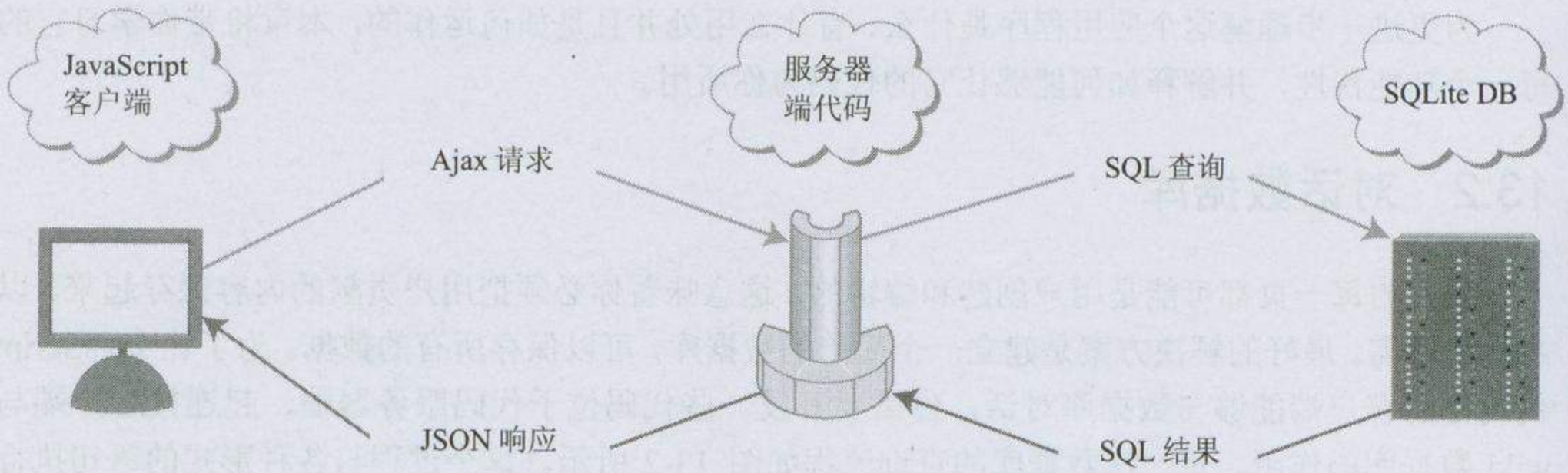


图13-2　应用程序执行客户端请求的流程

> **注意** 重要的是，你不应该从客户端直接发送原始的SQL请求。因为这样做的话会让服务器端和数据库无法抵抗恶意的攻击。因此应用程序中的所有SQL请求都应映射为关键字，然后转换为服务器端的真实请求。这会在保持客户端灵活的同时，仍能保护数据库免遭攻击。

为了进一步理解这个流程是如何运行的，你将逐步理解从 JavaScript 客户端发送请求到服务器端和数据库的过程。

13.3 Ajax 请求

Ajax 请求发生在需要与数据库（因此也跟服务器）通信的时候。比如，在需要保存 wiki 一次修订的时候触发 Ajax 请求。修订包含 4 个方面的信息：

- wiki页面的标题：命名wiki页面的常规约定是使用一种大小写混合的方案，它由首字母大写的单词构成。比如，wiki的主页可以命名为HomePage.
- 修订的作者名称：修订内容的用户可以选择留下他们的名字。但是，如果不愿意具名的话也允许他们匿名（这是大部分wiki一个十分流行的特性）。
- 修订的真正内容：用户输入的条目文本有可能很长。内容由Textile——一套流行的文本格式器，来负责格式化。
- 修订的精确时间：这由客户端生成并（但愿）用作唯一的标识符。因为时间精确到毫秒级，这就足以满足这个应用程序的需求了。

应用这部分知识，现在你可以构造一个发送到数据库的请求了。一个保存修订所需要代码的简单版本如代码清单 13-1 所示。

267

代码清单13-1 保存客户端数据到服务器端数据库的代码

```
// 把修订插入到数据库中
sqlExec(
        // 一段查询语句，但它的真实代码保存在服务器上
        "insert",
        [
            document.title, // wiki条目的标题
              $("#author").val(), // 用户提供的作者值
              $("#text").val(), // 修订的文本
              (new Date()).getTime() // 修订的精确时间
        ],
        // 重新载入修订的列表，一旦查询完毕
        reload
);
```

在 sqlExec 函数的背后，你需要准备好 SQL 查询（之后就会转换为 CGI 查询字符串）、构建请求的 URL 并发起 Ajax 请求。你可以使用 jQuery 的 Ajax 功能将它们结合起来，它能用于处理 HTTP POST 请求（发送长的修订文本需要使用 POST 请求）。你会在 13.5 节中看到最终的 Ajax 请求。

已经向服务器发送了请求，接下来你需要了解服务器端代码是如何处理数据库查询的。

13.4　服务器端代码

这个应用程序服务器端部分的流程非常简单，并且是可复用的。它总体目标就是获取来自客户端的 SQL 查询，在 SQL 数据库内运行，并以 JSON 字符串的格式返回结果。为把这个应用程序投入实践，需要精简这个应用程序的流程，并在几种流行的脚本语言如 Perl、PHP、Python 和 Ruby 等上复制不同的版本。

13.4.1　处理请求

客户端初始化这个应用程序的服务器端连接，请求执行数据库一段指定的 SQL 查询。你可以访问这个查询和数据库的名称，通过访问传入这个应用程序的 CGI 参数来执行它。在上一节中，你已经看到传入服务器端脚本的参数是数据库名称和 SQL 查询文本。为了解参数是如何获取的，代码清单 13-2 展示了 Ruby 版本的服务器端代码。

268

代码清单13-2　获取传入应用程序的服务器端部分的CGI参数，用Ruby编写

```
# 引入CGI库
require 'cgi'

# 创建一个新的CGI对象，它会解析传入的CGI参数
cgi = CGI.new

# 捕获查询参数
sql = cgi['sql']

# 从用户那里得到数据库名称，确保没有包含攻击性的字符
d = cgi['db'].gsub(/[^a-zA-Z0-9_-]/, "")
```

得到数据库名称和查询后，接着连接数据库。这引发了一个问题，该用什么类型的数据库呢？我们决定使用基于 SQL 的数据库，因为它在 Web 应用程序开发中被非常普遍的使用。而对于这个应用程序来说，使用的数据库是 SQLite。

SQLite 是一个很有潜力的 SQL 数据库实现，非常轻量和快速。为了简洁和速度，它牺牲了一些特性如用户、角色和权限等。这个应用程序非常符合需求。SQLite 数据库对应一个单独的文件，所以有多少个这样的文件就可以有多少个数据库。除了非常快之外，SQLite 对出于简单的应用程序或者测试的目的而建立数据库的方式也是非常迅速和简便。与其费力安装一个大型数据库（如 MySQL、PostgreSQL 或者 Oracle），不如使用恰到好处的 SQLite。

我们探讨的这几种语言（Perl、PHP、Python 和 Ruby）都有 SQLite 的一种或多种的支持方式：

- Perl有DBD::SQLite模块。这个模块特别有名，因为开发者决定在模块自身内完全实现SQLite的规范，这就是说，再也不需要额外的下载就能让数据库跑起来。
- PHP5有一个内置的SQLite支持。不幸的是，它只支持SQLite2（对一些应用程序来说不还算很坏），但若要完全兼容不同的代码，你需要安装PHP SQLite 3库。

- Python和Ruby都有SQLite库，官方SQLite的安装就直接跟它们挂钩。Python2.5直接内置SQLite支持（但我们并没有使用它，因为相对较新）。

强烈推荐你在一些小项目中使用 SQLite。它的确是建置和执行的好方法，而不需要浪费时间和成本在大型数据库的安装上。

269

每一种语言中如何连接 SQLite 数据库都不尽相同。但它们都需要两个步骤：第一步是引入 SQLite 库（以提供通用的连接函数），第二步是连接到数据库并记录连接状态以备后面之用。代码清单 13-3 展示了 Ruby 的实现方法。

代码清单13-3 引入外部的SQLite库并连接到数据库的服务器端代码，Ruby版本

```
# 引入外部SQLite库
require 'rubygems'
require_gem 'sqlite3-ruby'

# 稍候程序……

# 'd' 需要清晰，确保不会提供带攻击性字符的数据库文件名
d = cgi['db'].gsub(/[^a-zA-Z0-9_-]/, "")

# 连接到SQLite数据库，它只是一个文件而已
# 'd' 包含了我们数据库的名称，称为'wiki'
db = SQLite3::Database.new('../../data/' + d + '.db')
```

开启了 SQLite 数据库的连接后，现在你可以执行客户端的查询并获取相应结果了。

13.4.2 执行和格式化 SQL

开启数据库的一个连接后，现在你应该可以执行 SQL 查询了。我们的最终目标是可以把查询获得的结果转换成某种形式，这种形式可以容易地转换成 JSON 字符串并返回到客户端。SQL 结果中最方便的可摘要形式是散列的数组（array of hash），如代码清单 13-4 所示。散列数组以数据库匹配的每一行来描述，每一行数据的键/值对分别描述了列名称和列的值。

代码清单13-4 服务器返回JSON结构的例子

```
[
    {
        title: "HomePage",
        author: "John",
        content: "Welcome to my wonderful wiki!",
        date: "20060324122514"
    },
    {
        title: "Test",
        author: "Anonymous",
        content: "Lorem ipsum dolem...",
        date: "20060321101345"
    },
  ...
]
```

270

你所选择语言的不同，导致把 SQL 结果转换成所需结果的难度也不同。但最常见的情形就是 SQL 库返回的两个东西：列的名称数组，以及包含所有行数据数组的数组（如代码清单 13-5 所示）。

代码清单13-5　在Ruby中，查找wiki修订信息而执行SQL查询后返回的数据结构

```
rows.columns = ["title","author","content","date"]

rows = [
      ["HomePage","John","Welcome to my wonderful wiki!","20060324122514"],
      ["Test","Anonymous","Lorem ipsum dolem...","20060321101345"],
      ...
]
```

将代码清单 13-5 所示的 SQL 结果转换为如代码清单 13-4 所示的数据结构的过程，是相当棘手的。显然，你需要迭代匹配每一行，创建一个临时的散列（hash），把所有的列数据加进去，然后把这个新散列添加到全局数组中去。Ruby 的实现如代码清单 13-6 所示。

代码清单13-6　在Ruby中如何执行SQL语句并把结果添加到最终的数据结构中（称为r）

```
# 如果sql有返回的行 (例如一条SELECT语句)
db.query( sql ) do |rows|
      # 遍历返回的每一行
      rows.each do |row|
          # 建立一个临时的散列
          tmp = {}

          # 强制数组栏目转换为散列的键/值对
          for i in 0 .. rows.columns.length-1
              tmp[rows.columns[i]] = row[i]
          end

          # 把行散列添加到已找到的行数组中去
          r.push tmp
      end
end
```


已经有了最终的合适数据结构，你可以把它转换成 JSON 字符串了。本质上，JSON 是一种使用兼容 JavaScript 的对象标识来描述值（字符串和数字）、值的数组和散列（键/值对）的方式。因为你已经小心应对并确保这个数据结构包含的就是数组、散列和字符串，你可以轻松地把它转换成 JSON 格式的字符串了。

在这个应用程序上使用的所有语言都有 JSON 串行化（serialization，把原始的数据结构转换成 JSON 字符串）的实现，而这正是你所需要的。此外，因为大部分 JSON 格式数据的输出都是十分轻量的，所以输出到浏览器也非常方便。而且，每一门语言的实现到最后都很相似，从而能够简化语言的迁徙过程。

- 每种实现都以库或者模块的形式存在。
- 每种实现都可以转换语言的原始对象（比如字符串、数组和散列）。
- 每种实现都很容易从JSON格式对象的字符串中获取数据。

尽管如此，实现 JSON 串行化尤为优雅的语言莫过于 Ruby 了。下面是一个把对象转换成 JSON 字符串（在载入 JSON 库之后）并输出到客户端上的例子：

```
# 把对象(r)转换成JSON字符串，并输出
printr.to_json
```

强烈推荐你探索一下服务器端实现的代码，并了解它如何处理 SQL 查询和 JSON 串行化的。我想你会因为他们的简单而感到非常愉悦。

13.5 处理 JSON 响应

你已经收到服务器返回的包含格式化 JSON 字符串的响应。回到代码清单 13-2，现在你可以编写必要的代码来处理和对这段 JSON 代码求值了。JSON 的长处在于求值和导航的简洁性。你所需要的不是一个繁冗的解析器（相对于 XML），而是一个 `eval()`函数（它会执行 JavaScript 代码，而 JSON 字符串正是 JavaScript 代码）。使用 jQuery 库的话，它会得到细致的处理，如代码清单 13-7 所示。而你所需要做的只是指定数据类型（称为 dataType）为“json”，即可从服务器接收到 JSON 数据。

代码清单13-7 从服务器获取结果并发送JavaScript数据结构到回调函数中

```
// 向服务器提交查询
$.ajax({
        // 发送到API的URL
        type: "POST",
        url: apiURL,
        // 串行化数据数组
        data: $.param(p),

        // 我们希望返回的是JSON数据
        dataType: "json",

        // 等待请求的成功完成
        // 如果回调由用户指定，把数据送到回调中
        success: callback
});
```

272

如果你稍加留意，会发现最后执行的动作是运行名为 `callback` 的函数，它指向由 `sqlExec` 函数提供的回调函数。传入这个函数的单一参数是在服务器端细心构造并显示给客户端的完整数据结构。要了解这个 sqlExec-Ajax 响应的流程是如何运作的，看看 wiki 实际运行的一个简化版本，如代码清单 13-8 所示。它的规定是，如果页面有过修订，就必须显示最近的一次修订。否则，应该显示一个表单以便用户能够创建新的修订。

代码清单13-8 用在客户端的代码的简化版本，从服务器获取修订并在页面上显示

```
// 请求当前wiki页面的所有修订
// 一旦载入，由 'loaded' 函数去调用返回的数据
sqlExec("select", [$s], loaded);

// 处理服务器返回的SQL结果
```

```
function loaded(sql) {
        // 如果这条wiki存在修订
        if ( sql.length > 0 ) {
            // 显示wiki页面
            showContent();

            // 重新生成修订，使用textile格式化
            $("#content").html(textile(sql[0].content));

            // 让修订内容可编辑
            $("textarea").val( sql[0].content );

        // 否则，不存在修订的话，显示“创建”的表单
        } else {
            // 显示默认的编辑表单
            showForm();
        }
273     }
```

这段代码中重要的部分是 `textile(sql[0].content)`这个小片段，它从数据结构中抓取 wiki 修订文本后运行 Textile 格式器。与 HTML 一样，Textile 也是一种标记内容的方式。Textile 非常简单，只提供基本的、容易理解的格式化标记，任何人都可以轻松上手。一个 Textile 格式化的例子如图 13-3 所示。

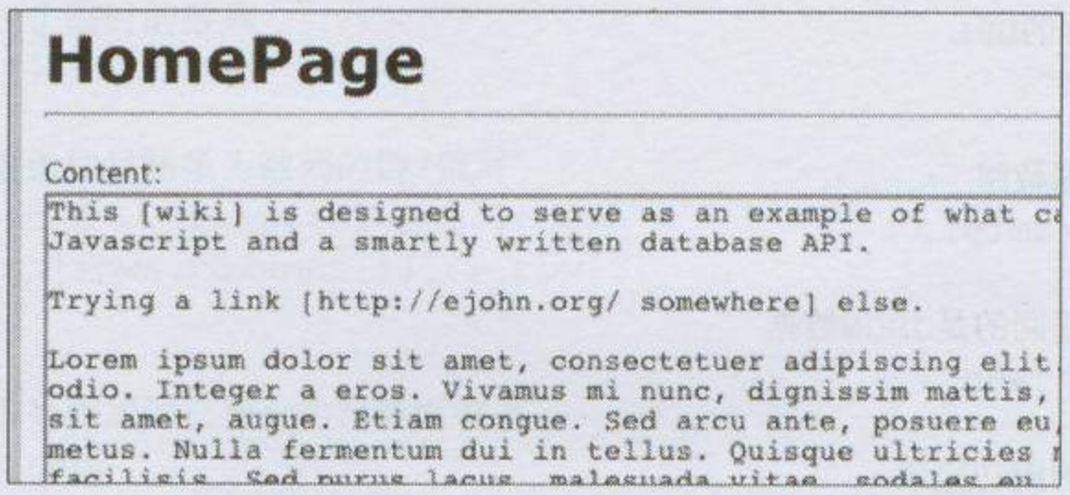


图13-3　使用Textile格式化后的wiki修订

Textile 优秀的一面还在于，有人经过努力的工作创建了一个 Textile JavaScript 库，它能够将 Textile 格式的文本转换成最终可发布和展现的 HTML。关于 Textile 更多的信息可以在以下网站找到：

- Textile概览：http://www.textism.com/tools/textile/。
- JavaScript Textile实现（在这个项目中用到）：http://jrm.cc/extras/live-textile-preview.php。

格式化从服务器返回的数据并插入到文档中后，你已经完成了整个 wiki。建议你架设自己的 wiki 代码并运行。代码已经有足够多的注释，因此这个操练的应该很轻松。

13.6　附加的案例研究：JavaScript blog

如果你还希望这个客户端–服务器端–SQLite 的配置能做些事情的话，那么可以看看接下来的

另外一个演示。这个应用程序是一个简洁私人的浏览器内的 blog。当然它也非常容易使用，当你访问由这个私人 blog 时，可以查看也可以发言。此外，它还有一个帖子（post）的实时预览和一个小型 SQL 终端，也相当有趣。这个 blog 应用程序的代码和演示可以在本书的网站(http://jspro.org)上找到。一个运行中的屏幕截图如图 13-4 所示。

274

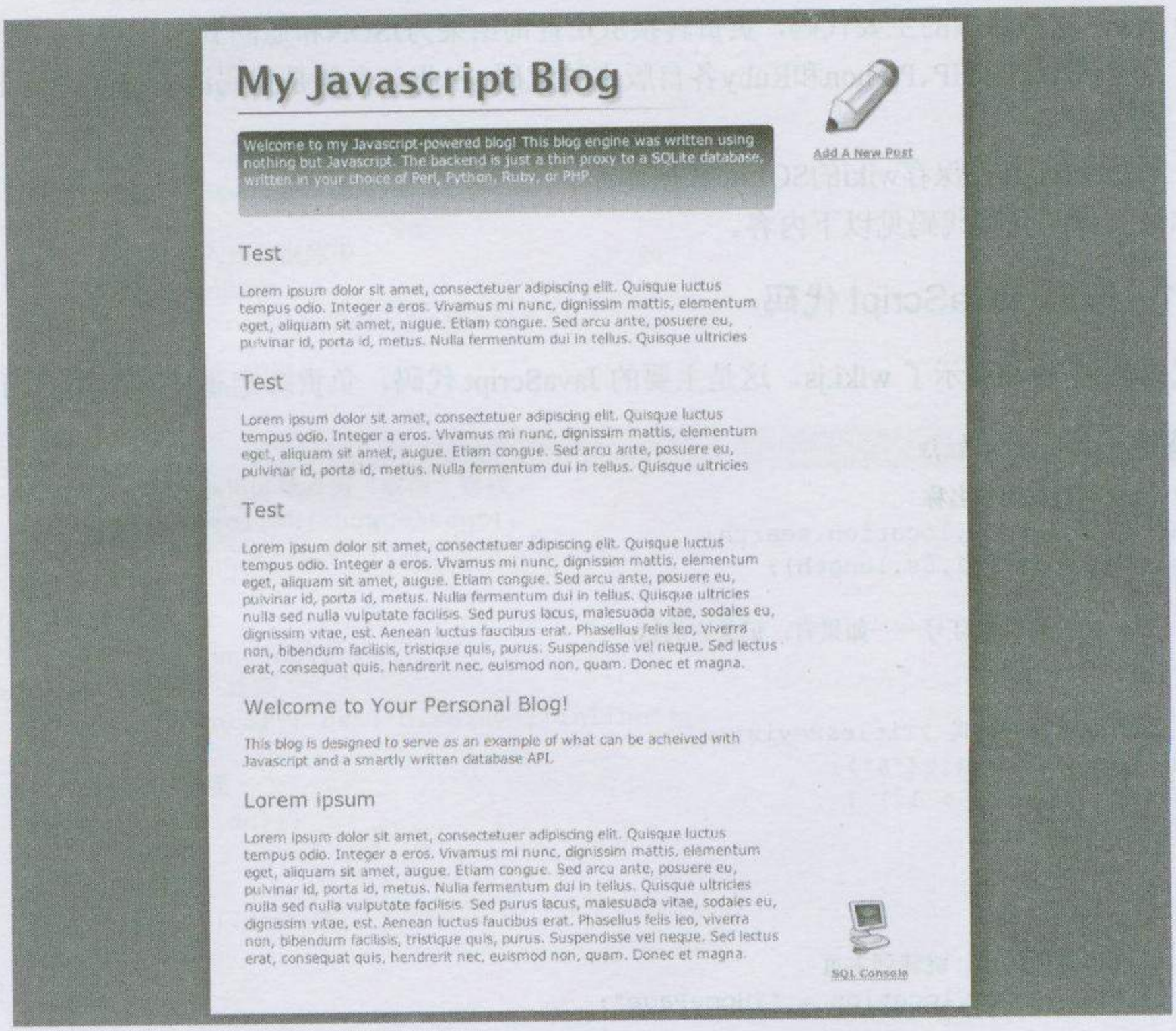


图13-4 浏览器内的blog和SQL终端的屏幕截图

13.7 应用程序的代码

本章演示的案例比先前的要复杂得多。这部分详述了使这个应用程序的主要代码（与本章中之前讨论的主题相关）。

以下是这个 wiki 应用程序运行必需的所有文件的列表包括了本章所讨论过的客户端和服务器端代码，以及所有没有直接提及的不同库和样式文件的混合。

- index.html：应用程序的主页，把所有的客户端代码结合到一起。
- install.html：安装的主文件，在第一次使用此应用程序时运行。
- css/style.css：样式化应用程序的客户端的CSS。
- js/wiki.js（见代码清单13-9）：JavaScript的主要代码，负责绑定事件和运行SQL查询。

275

- js/sql.js（见代码清单13-10）：负责通信服务器，从SQL查询中获取数据。
- js/textile.js：一个JavaScript Textile库的副本（为把文本转换成HTML）：http://jrm.cc/extras/live-textile-preview.php。
- js/jquery.js：一个jQuery当前发行版本的副本：http://jquery.com/。
- api/：服务器端的主要代码，负责转换SQL查询结果为JSON和返回到客户端中。这个目录下包含Perl、PHP、Python和Ruby各自版本的代码。在此包含的是代码清单13-11所示的Ruby版本。
- data/wiki.db：保存wiki的SQLite数据库。

本章文件的完整代码见以下内容。

13.7.1　核心 JavaScript 代码

代码清单 13-9 展示了 wiki.js，这是主要的 JavaScript 代码，负责绑定事件和与用户的交互。

代码清单13-9　*js/wiki.js*

```
// 获取当前页面的名称
var $s = window.location.search;
$s = $s.substr(1,$s.length);

// 确定是否提供修订号——如果有，记录它的ID
var $r = false;

// 修订是这样的格式 ?Title&RevisionID
var tmp = $s.split("&");
if ( tmp.length > 1 ) {
  $s = tmp[0];
  $r = tmp[1];
}

// 不提供页面的话，跳转到主页
if (!$s) window.location = "?HomePage";

// 设置数据库名字
var db = "wiki";

// 我们需要等待DOM的加载完毕
$(document).ready(function(){

  // 设置页面的标题
  document.title = $s;
  $("h1").html($s);

  // 载入所有的wiki修订版本
  reload();

  // 如果点击了“编辑页面”的连接
  $("#edit").click(showForm);
  // 当用户提交新的修订
```

```
$("#post form").submit(function(){
  // 获取作者名字
  var author = $("#author").val();

  // 获取内容文本
  var text = $("#text").val();

  // 重新生成内容
  $("#content").html(textile(text));

  // 生成当前修订的时间（有助于高亮）
  $r = (new Date()).getTime();

  // 把修订插入到数据库中
  sqlExec("insert,
    [$s,author,text,$r], reload);

  return false;
 });

 // 如果用户在编辑区域点击“取消”连接
 $("#cancel").click(showContent);
});

// 显示当前修订
function showContent() {
  // 显示“编辑”连接
  $("#edit,#cancel").css("display","inline");

  // 隐藏编辑区域
  $("#post").hide();

  // 显示内容
  $("#content").show();
  return false;
}

// 显示编辑当前修订的标单
function showForm() {
  // 隐藏“编辑”连接
  $("#edit").hide();

  // 显示编辑区域
  $("#post").show();

  // 隐藏内容
  $("#content").hide();

  return false;
}

// 从数据库加载所有的修订版本
function reload(t) {
  // 请求所有的修订版本
```

```
sqlExec("select, [$s],
  function(sql) {
    // 如果wiki存在修订
    if ( sql.length > 0 ) {
      if ( !$r ) $r = sql[0].date;

      // 显示wiki页面
      showContent();

      // 显示所有的修订
      $("#side ul").html('');

      // 遍历所有的修订
      for ( var i = 0; i < sql.length; i++ ) {

        // 如果修订正是当前显示的
        if ( sql[i].date == $r ) {

          // 生成修订
          $("#content").html(textile(sql[i].content));

          // 让修订的内容可编辑
          $("textarea").val( sql[i].content );

        }

        // 获取可工作的日期对象
        var d = new Date( parsent sql[i].date);
        // 判断修订是否在一天之内
        if ( d.getTime() > (new Date()).getTime() - (3600 * 24000) )

          // 如果是，生成漂亮的 am/pm 时间格式
          d = d.getHours() >= 12 ?
            (d.getHours() != 12 ? d.getHours() - 12 : 12 ) + " pm" :
            d.getHours() + " am";

        // 否则，直接显示修订的月日
        else {
          var a = d.toUTCString().split(" ");
          d = a[2] + " " + d.getDate();
        }

        // 把修订添加到修订列表中
        $("#side ul").append("<li class='" + ( $r == sql[i].date ? "cur" : "" )
            + "'><a href='?" + $s + ( i > 0 ? "&" + sql[i].date : "" )
            + "'>" + d + "</a> by " + sql[i].author + "</li>");
      }

    // 否则，这个页面未经修订过
    } else {
      // 把这个事实显示到修订面板上
      $("#rev").html("<li>No Revisions.</li>");
      // 隐藏编辑控制
      $("#edit,#cancel").hide();
```

```
            // 显示默认的编辑表单
            showForm();
        }
    });
}
```

13.7.2 JavaScript SQL 库

代码清单 13-10 展示了 sql.js，这些代码负责通信服务器并从 SQL 查询中获取 JSON 数据。

代码清单13-10 *js/sql.js*

```
// 根据你的服务器端脚本的所在更新这个变量
var apiURL = "api/ruby/";

// 一些默认的全局变量
var sqlLoaded = function(){}
// 处理长的SQL提交
// 这个函数可用以发送大量的数据（比如，大的INSERT），但只能发送到服务器上与客户端相同的位置
function sqlExec(q, p, callback) {

  // 加载所有参数到结构化数组中
  for ( var i = 0; i < p.length; i++ ) {
    p[i] = { name: "arg", value: p[i] };
  }

  // 包含数据库名
  p.push({ name: "db", value: db });

  // 执行SQL查询名
  p.push({ name: "sql", value: q });

  // 提交查询到服务器
  $.ajax({
    // POST to the API URL
    type: "POST",
    url: apiURL,

    // 数组数据序列化
    data: $.param(p),

    // 返回JSON数据
    dataType: "json",

    // 等待成功完成响应
    // 如果用户指定了反馈，返回数据
    success: callback
  });
}
```

13.7.3 Ruby 服务器端代码

以下代码清单 13-11 中的代码，是 Ajax wiki 应用程序的服务器端部分，所有这些代码是以

Ruby 编程语言编写的。而以 PHP、Perl 或者 Python 编写的相同代码，可以访问本书的网站
280 http://jspro.org/得到。

代码清单13-11　Wiki应用程序的服务器端部分代码，以Ruby编写

```
#!/usr/bin/env ruby

# 引入所有的外部库
require 'cgi'
require 'rubygems'
require_gem 'sqlite3-ruby'
require 'json/objects'

# 输出JavaScript头
print "Content-type: text/javascript\n\n"

# 初始化应用程序变量
err = ""
r = []
cgi = CGI.new

# 捕获由用户传入的值
call = cgi['callback']
sql = cgi['sql']

# 从用户处获取数据库并确保没有使用攻击性的字符
d = cgi['db'].gsub(/[^a-zA-Z0-9_-]/, "")

# 如果不提供数据库，则使用默认的'test'
if d == '' then
  d = "test"
end

# 获取放入SQL查询中的参数列表
args = cgi.params['arg']

# 只接受两种不同SQL查询

# 在数据库中插入wiki新修订版
if sql == "insert" then
sql = "INSERT INTO wiki VALUES(?,?,?,?);"

# 获取所有的wiki修订版
elsif sql == "select" then
281 sql = "SELECT * FROM wiki WHERE title=? ORDER BY date DESC;"

# 否则，查询失败
else
sql = ""
end

# 如果提供了SQL查询
```

```
if sql != '' then

  # 遍历每个提供的参数
  for i in 0 .. args.length-1
    # 代替所有''的引用（相当于在SQLite中避开它们），避开所有?
    args[i] = args[i].gsub(/'/, "''").gsub(/\?/, "\\?")

    # 遍历SQL查询替代第一个符合条件的数据
    sql = sql.sub(/([^\\])\?/, "\\1'" + args[i] + "'")
  end

  # 完成后，un-escape标志所避开的东西
  sql = sql.gsub(/\\\?/, "?")

  # 确保我们能捕获所有抛出的错误
  begin
    # 连接到SQLite数据库，它只是一个文件而已
    db = SQLite3::Database.new('../../data/' + d + '.db')

    # 如果sql返回部分行（比如，它是一条SELECT）
    db.query( sql ) do |rows|
      # 遍历所有返回的行
      rows.each do |row|
        # 创建一个临时散列
        tmp = {}

        # 强制数组栏目转换为散列的 键/值对
        for i in 0 .. rows.columns.length-1
          tmp[rows.columns[i]] = row[i]
        end

        # 把行三列添加到已找到的行数组中去
        r.push tmp
      end
    end

    rescue Exception => e
      # 如果出错，记录信息备用
      err = e
    end
  else
    # 如果没有提供SQL查询，显示一条错误
    err = "No query provided."
  end

  # 如果出错，则返回一个散列，它包含错误信息的键和值对
  if err != '' then
    r = { "error" => err }
  end

  # 把返回的对象转换成JSON字符串
```

282

```
jout = r.to_json
  # 如果提供了回调函数
  if call != '' then
    # 在回调字符串中包裹返回的对象
    print call + "(" + jout + ")"
  else
    # 否则直接输出JSON字符串
    print jout
  end
```

13.8　小结

在讲述这个应用程序的过程中，希望你能够从中学到一些有用的观念。首先，JSON 非常强大灵活，在 Web 应用程序中可以作为 XML 的可替代选择。其次，保持服务器端代码尽可能简洁，就可以让前端对用户数据更多的控制，但它应该有所节制，以隐藏最重要的商业逻辑）。最后，所有现代服务器端脚本语言行为都非常相似（因为它们都有很多相同的特性）且非常容易相互转换，这也是为何用这 4 种最流行的脚本语言编写相同版本的服务器端代码的原因。

主要的代码都可以在本章的应用程序代码一节中找到。应用程序的完整的可安装版可以在本书的网站 http://jspro.org/或者 http://www.apress.com/上找到，并有安装指南。一个在线的演示可以在 http://jspro.org/demo/wiki/找到。

283　网站上有如何使用这些代码的完整指南。此外，如果碰到一些问题，还有一个论坛可供讨论[①]。

① 目前，http://jspro.org/上并没有提供论坛。——译者注

Part 5

第五部分

JavaScript 的未来

本部分内容

第 14 章

JavaScript 路在何方

最近几年，JavaScript 语言的开发得到了大量的投入，这些投入来自许多不同方面。Mozilla 基金会一直在改进 JavaScript 语言的质量、保持对 ECMAScript 4 标准（JavaScript 所基于的语言）的支持；另一方面，WHAT-WG 组织——一个由 Web 浏览器开发商组成的致力于开发新技术，让应用程序能在 Web 上编写配置的团体，也创建了下一代基于浏览器的应用程序规范。最后，程序库的开发者和公司都在致力于把基于浏览器的应用程序串流化技术凝聚为一套称作 Comet 的技术。所有这些都代表了 JavaScript 语言和基于浏览器开发的未来。

在本章里，我们将看到 JavaScript 1.6 和 1.7 版本已经作出的改进，并为完整的 JavaScript 2.0 版本发布做好准备。还能看到 Web Applications 1.0 规范带来的改进：用 JavaScript 绘图。最后，我们将简单地了解 Comet 和串流式 Web 应用程序的概念。

14.1 JavaScript 1.6 与 1.7

过去的 10 年，JavaScript 语言一直在缓步前进，对许多功能性做了改进。尽管许多现代浏览器都支持了 JavaScript 1.5（或其等价版本），但它们一直对推进这门语言不甚积极。

Brendan Eich 和其他 Mozilla 基金会的成员则一直在努力推动 JavaScript 语言并 ECMAScript 4 标准的进步。可以在这里找到更多 Mozilla 的工作：

- Mozilla的JavaScript相关工作：http://www.mozilla.org/js/language/。
- Mozilla的JavaScript 2.0提案：http://www.mozilla.org/js/language/js20/。
- Mozilla的ECMAScript 4提案：http://www.mozilla.org/js/language/es4/。

虽然 JavaScript 2.0 还未曾确定，Mozilla 已在开辟道路，它们发布了 JavaScript 的 1.6 和 1.7 版本，包含了一系列要加入到最终改良过的语言中去的特性。这一系列特性也非常重要，我们会在这一节里简单叙述它们。

14.1.1 JavaScript 1.6

更新 JavaScript 语言的第一个版本是 1.6，它与 Mozilla 基金会的 Firefox 浏览器的 1.5 版本一同出现。这部分修改的简短说明可见 Mozilla 网站：http://developer.mozilla.org/en/docs/New_in_JavaScript_1.6。

这个版本里两个重要的特性是 E4X（ECMAScript for XML）和一系列给数组提供的新函数。目前所有其他浏览器都没有提供这几个特性，但很有可能以后 Opera 和 Safari 会提供。我们会在

此展示这些特性的优点。

1. E4X

E4X 给 JavaScript 语言添加了一套新的语法，允许你把 XML 直接书写在 JavaScript 代码中间，其影响有趣而复杂。关于 E4X 的更多信息可在 Mozilla 网站上它的规范中找到：

- ECMAScript for XML规范：http://www.ecma-international.org/publications/standards/Ecma-357.htm。
- E4X快速概览：http://developer.mozilla.org/presentations/xtech2005/e4x/。

总而言之，这个规范允许你用 JavaScript 般的语法来获得 XML DOM 形式的结果。比如说写 `var img = <img/> + <hr/>`将给图像元素添加一条水平分隔线，并将 DOM 结果元素存入变量，以供后续的使用。更复杂的一个例子可见代码清单 14-1。得到的 XML 文档可见代码清单 14-2。

代码清单14-1 用E4X构建HTML文档，来自Brendan Eich的演示

```
<script type="text/javascript;e4x=1">
    // 创建一个 HTML 元素并存入变量
    var html = <html/>;

    // 将 title 元素的内容设为一段字符串，E4X 能自动创建所有缺少的元素，
    //并正确处理文本结点
    html.head.title = "My Page Title";

    // 设置 body 元素背景色属性，这一 body 元素是自动创建的
    html.body.@bgcolor = "#e4e4e4";

    // 给 body 中的 form 元素添加一些属性
    html.body.form.@name = "myform";
    html.body.form.@action = "someurl.cgi";
    html.body.form.@method = "post";
    html.body.form.@onclick = "return somejs();";

    // 指定名称地创建一个空 input 元素
    html.body.form.input[0] = "";
    html.body.form.input[0].@name = "test";
</script>
```

288

代码清单14-2 调用代码清单14-1中的E4X代码生成的HTML文档

```
<html>
    <head>
        <title>My Page Title</title>
    </head>
    <body bgcolor="#e4e4e4">
        <form name="myform" action="someurl.jss"
                method="post" onclick="return somejs();">
            <input name="test"></input>
        </form>
    </body>
</html>
```

虽然 E4X 的语法与传统 JavaScript 风格有所背离——这可能会吓走一部分新用户——却非常有用，能让你减少许多重复性的 DOM 操作。

2. 数组的扩展

JavaScript 1.6 新增的主要特性是关于数组的。在 1.6 版本里，数组提供了一系列新方法，可以用作通用的操作：

- `indexOf()`和`lastIndexOf()`与`string`对象的同名方法类似，这两个方法能让你查出数组中某个对象的位置，返回其下标（index），如果查找的对象不存在则返回`-1`。
- 3个新方法`forEach()`、`some()`和`many()`能帮助简化常见的迭代操作，让你对数组的每个元素执行某个函数。
- 新的`filter()`和`map()`函数让你执行内联（inline）数组变换，类似其他语言（如Perl）中的map和grep操作。

289　代码清单 14-3 列出了所有这些新的 JavaScript 1.6 数组函数的例子。

代码清单14-3　JavaScript 1.6中新数组函数的例子

```
// 一个简单的整数数组
var tmp = [ 1, 2, 3, 4, 5, 3 ];

// indexOf( Object )
// 找出数组中某一对象的下标
tmp.indexOf( 3 ) == 2
tmp.indexOf( 8 ) == -1

// lastIndexOf( Object )
// 找出数组中某一对象最后一次出现的下标
tmp.lastIndexOf( 3 ) == 5

// forEach( Function )
// 对数组的每个元素调用一个函数，此函数传入三个参数：对象，下标和数组的引用
tmp.forEach( alert );

// every( Function )
// 对数组的每个元素调用一个函数，只当此函数对所有对象都返回 true 时，它才
// 返回 true
tmp.every(function(num){
    return num < 6;
}) // true

// some( Function )
// 对数组的每个元素调用一个函数，只要此函数对其中一个对象返回 true，它就
// 返回 true
tmp.some(function(num){
    return num > 6;
}) // false

// filter( Function )
// 通过只保留满足某一具体条件的对象来过滤数组。如果此函数返回 true，则保
// 留下来
tmp.filter(function(num){
    return num > 3;
}) // [ 4, 5 ]
```

```
// map( Function )
// 将数组中所有对象转换为另一套对象集合。对原数组的每个对象执行传入的函数，
// 用函数的返回值替换数组中的该对象
tmp.map(function(num) {
    return num + 2;
}) // [ 3, 4, 5, 6, 7, 5 ]
```

290

除了这些简单的例子以外，这些新方法还为 JavaScript 和数组提供了大量迫切的速度和功能上的改进。我们非常期望有一天这些方法能跨浏览器得到支持。

14.1.2 JavaScript 1.7

JavaScript 语言的这一新版本添加了大量功能，其中许多特性是为了拉近 JavaScript 与其他功能完整语言的距离。此外，这个新发布的 JavaScript1.7 要比先前版本更新更为强大，因为它所增加的特性能改变这门语言所应用的方式。这个网站上详细介绍了 JavaScript 1.7 提供的一部分新特性：http://developer.mozilla.org/en/docs/New_in_JavaScript_1.7。

对 JavaScript 语言的此次更新是与 Mozilla 的 2.0 版本 Firefox 浏览器同时发布的。这个浏览器包含了下面概述功能的全部实现，它也是目前唯一一个，在 JavaScript 语言上有如此重要而先进更新的浏览器。

1. 数组包含

一个不错的新改进是，你现在可以在一行里写出生成数组的代码了。以前要给数组添加一组数组时，必须遍历集合，将其元素逐个插入数组，现在用数组包含（array comprehension）的语法可以简单一步完成。代码清单 14-4 展示的例子很好地解释了它。

代码清单14-4 JavaScript 1.7中的数组包含

```
<script type="application/javascript;version=1.7">
    // 将一系列数字放入数组
    var array = [];
    for ( var i = 0; i < 10; i++ ) {
        array.push( i );
    }

    // 新方法
    var array = [ i for ( i = 0; i < 10; i++ ) ];

    // 将对象索引放入数组的旧方法
    var array = [];
    for ( var key in obj ) {
        array.push( key );
    }

    // 新方法
    var array = [ key for ( key in obj ) ];
</script>
```

这一特性已经在其他语言（如 Python）中存在了一段时间，现在能在 JavaScript 中出现真是一件好事情。

291

2. let作用域控制

let 作用域是一个奇妙的新特性，而且很可能会成为最广泛接受和使用的一个特性。到目前为止，JavaScript 都不具有任何块级作用域控制功能（如第 2 章所述）。随着这套新的 let 语句、表达式和定义的出现，现在可以在许多不同等级内定义变量的作用域了。代码清单 14-5 展示了 let 作用域控制能实现的功能。

代码清单14-5　JavaScript 1.7的let作用域控制示例

```
<script type="application/javascript;version=1.7">
    // let 语句
    var test = 10;
    let( test = 20 ) {
        alert( test ); // 弹出窗口显示 20
    }
    alert( test ); // 弹出窗口显示 10

    // let 表达式
    var test = 10;
    alert( let( test = 20 ) test ); // 弹出窗口显示 20
    alert( test ); // 弹出窗口显示 10

    // let 定义
    var test = 10;
    if ( test == 10 ) {
        let newTest = 20;
        test += newTest;
    }
    alert( test ); // 弹出 30
    alert( newTest ); // 失败，在 if 语句外 newText 未定义

    // 在一块 for 循环中使用 let
    for ( let i = 0; i < 10; i++ ) {
        alert( i );
    }
    alert( i ); // 失败，在 for 语句外 i 未定义
</script>
```

随着这个简单的改进，你可以使代码更整洁、工作更有效，并避免一系列常见的命名空间冲突（这套技术将被发展为 JavaScript 2.0 的类与命名空间）。

3. 解构式赋值

JavaScript 1.7 中引入的最后一个比较重要的概念是解构式赋值（destructuring）[①]。这是来自函
292 数式程序设计语言（如 Lisp）的概念，能让你把复杂的数据结构作为操作符的左值（left-handvalue）。ECMAScript 4 中的解构式赋值在 Mozilla 的网站上有介绍：http://developer.mozilla.org/es4/proposals/destructuring_assignment.html。

① 传统的变量赋值是把操作符的左值作为一个整体看待，所以不能有效处理复杂数据结构，如对象、数组的赋值，如果希望在初始化时就能对对象或数组中的元素进行一一指定具体的值，就应该把这些复杂的数据结构“解构”开来，分解成简单数据结构的组合，从而能达到一一赋值的目的。虽然destructuring的功能并不仅限于赋值的应用，但这里为了简明，姑且如此翻译。——译者注

解构式赋值概念并不简单，花点时间来学是值得的。代码清单 14-6 展示了一系列解释 JavaScript 1.7 中的解构式赋值如何工作的例子。

代码清单14-6 JavaScript 1.7中解构式赋值的例子

```
<script type="application/javascript;version=1.7">
    // 运用解构式赋值来交换两个变量值的例子
    [ b, a ] = [ a, b ]

    // 一个返回字符串数组的简单函数
    function test() {
        return [ "John", "October" ];
    }

    // 我们可以把这个函数返回的数据解构为两个变量 - name 和 month
    var [ name, month ] = test();

    // 对对象进行解构式赋值的例子
    var { name: myName } = { name: "John" };
    // 现在 myName == "John"

    // 一个简单数据结构
    var users = [
        { name: "John", month: "October" },
        { name: "Bob", month: "December" },
        { name: "Jane", month: "May" }
    ];

    // 在循环中进行解构
    for ( let { name: name, month: month } in users ) {
        // 依次弹出包含 John, Bob 和 Jane 的提示框
        alert( name + " was born in " + month );
    }
</script>
```

总而言之，JavaScript 语言正在向一个非常可靠的方向发展，全面吸收许多其他语言（如 Python 和 Lisp）的有用特性。这些特性的可用性，很大程度上决定于不同浏览器厂商对这门语言的支持。虽然可能要过一段时间你才可能在跨浏览器 Web 应用中用到 JavaScript 1.6 或 1.7 的版本，不过现在可以用它们来开发针对 Mozilla 系列浏览器的扩展。

293

14.2 Web Applications 1.0

推动 JavaScript 开发的第二个规范是 WHAT-WG（Web Hypertext Application Technology Working Group，Web 超文本应用技术工作组）这个新定制的 Web Applications 1.0 规范。这一规范将对 HTML、DOM 和 JavaScript 的一些改进进行整合。许多人认为这份规范将促成 HTML 5[①]。幸运的是，和 JavaScript 的新版本不同，这一规范的部分功能已经得到许多浏览器的支持了。

尽管整个规范很长，但我还是强烈建议你仔细阅读，以了解这个很快将会出现的新技术。在这一节里，我们只会讨论这个新规范中的一个特性：<canvas>元素。这个新元素允许你用

① 现在W3C已经正式接纳它，作为HTML 5的一个基础。——译者注

JavaScript 编写绘制 2D 图形的程序。这一技术被广泛接受，并成为了一个易于学习和尝试的主题。关于 Web Applications 1.0 和<canvas>元素的更多信息可见：

- 完整的Web Applications 1.0规范：http://whatwg.org/specs/Web-apps/current-work/。
- 规范中专门讨论新的<canvas>元素的小节：http://whatwg.org/specs/Web-apps/current-work/#the-2d。
- 运用新的<canvas>元素的一系列例子：http://developer.mozilla.org/en/docs/Canvas_tutorial:Basic_animations。

<canvas>元素填补了许多 Web 应用程序开发者的需要，可以实现旋转图像、绘制线段和创建图形等功能。这个改进能把 Web 应用程序的互动性带入崭新的一个层面。

绘制任意形状的功能是如此有用，令浏览器开发者们都受到鼓舞并给它们最新的浏览器版本提供了 Canvas API 的支持，目前除 IE 以外所有现代浏览器都支持了 Canvas API，而 Google 也出来给 IE 实现了一套完整的 Canvas API，这是通过 IE 本身提供的 VML 支持实现的。关于 Google 的 Internet Explorer Canvas 实现 ExplorerCanvas 的信息可在 http://code.google.com/p/explorercanvas/ 找到。

下面我们将仔细看看 Mozilla 的基本<canvas>教程中展示的两个例子。

14.2.1　制作时钟

第一个例子是制作一个简单的时钟，时钟的每一部分都将使用 Canvas 2D API 绘制，整个过程都不需要用到图片或其他的 HTML 元素。图 14-1 展示了绘制出来的时钟。

294

图14-1　用Canvas API绘制的活动时钟

<canvas>元素的工作方式和真正的画布差不多，每绘制这个动态时钟里的一帧，必须先完全清除画布上原有的图案进行重绘。如果你使用过 OpenGL API，就会很习惯 Canvas API 的。

使用 Canvas API 时，你经常需要旋转图像画布。这可能有些令人迷惑，因为当时你可能不知道正处的角度，或“笔刷”的位置，所以需要 Canvas API 以堆栈的方式工作。你可以把原来的位置状态保存，旋转画布，对当前的图像执行一些改变，之后再返回原来的画布状态，继续绘制。

代码清单 14-7 包含了使用 Canvas 来绘制时钟的代码，注意，这部分代码大量使用了 Canvas 提供的堆栈系统以简化绘制过程。

代码清单14-7　使用Canvas API来绘制一个活动时钟

```
<html>
<head>
    <title>Canvas Clock Demo</title>
    <script>
        // 等待浏览器载入
        window.onload = function() {
            // 绘制时钟
            clock();
```

```
    // 此后每秒重绘一次时钟
    setInterval(clock, 1000);
};

function clock() {
    // 获得当前日期和时间
    var now = new Date();
    var sec = now.getSeconds();
    var min = now.getMinutes();
    var hr  = now.getHours();
    hr = hr >= 12 ? hr - 12 : hr;

    // 获得 <canvas> 元素的绘图区域
    var ctx = document.getElementById('canvas').getContext('2d');

ctx.save();
  // 初始化绘图 Canvas
  ctx.clearRect(0,0,150,150);

  // 将原点移动到 75,75
  ctx.translate(75,75);

  // 使所有操作的效果缩小为 0.4 倍
  ctx.scale(0.4,0.4);

  // 让指针开始指向 12:00
  ctx.rotate(-Math.PI/2);

  // 初始化绘图属性
  ctx.strokeStyle = "black";
  ctx.fillStyle = "black";
  ctx.lineWidth = 8;
  ctx.lineCap = "round";

  // 小时刻度
  ctx.save();
ctx.save();
  ctx.beginPath();
    // 每小时一个
    for ( var i = 0; i < 12; i++ ) {
        // 将 Canvas 旋转一周的 1/12
        // (注意: 一周 = 2 * PI)
        ctx.rotate(Math.PI/6);

        // 将游标往 Canvas 外侧移动一些
        ctx.moveTo(100,0);

        // 绘制长 (20px) 线
        ctx.lineTo(120,0);
    }
  ctx.stroke();
ctx.restore();

// 分钟刻度
```

295

```
ctx.save();
  // 这些刻度要比小时刻度细一些
  ctx.lineWidth = 5;

  ctx.beginPath();
    // For each minute
    for ( var i = 0; i < 60; i++ ) {
        // 除去那些整点上的分钟
        if ( i % 5 != 0 ) {
            // 将游标移远一些
            ctx.moveTo(117,0);

            // 绘制短 (3px) 线
            ctx.lineTo(120,0);
        }
        //  将 Canvas 再旋转 1/60 圆周
        ctx.rotate(Math.PI/30);
    }
  ctx.stroke();
ctx.restore();

// 绘制时针
ctx.save();
  // 将 Canvas 旋转到正确方向
  ctx.rotate( (Math.PI/6) * hr +
              (Math.PI/360) * min +
              (Math.PI/21600) * sec );

  // 这根线段必须足够粗
  ctx.lineWidth = 14;

  ctx.beginPath();
    // 将时针开头处移动到稍稍偏离中心的位置 (使之更真实)
    ctx.moveTo(-20,0);

    // 在小时刻度附近绘制
    ctx.lineTo(80,0);
  ctx.stroke();
ctx.restore();

// 绘制分针
ctx.save();
  // 将 Canvas 绘制到当前的分钟位置
  ctx.rotate( (Math.PI/30) * min + (Math.PI/1800) * sec );
  // 这根线段要比时钟更细
  ctx.lineWidth = 10;

  ctx.beginPath();
    // 但分针要更长一点，所以要更偏离中心一点
    ctx.moveTo(-28,0);

    // 将其绘制得长一点
    ctx.lineTo(112,0);
  ctx.stroke();
```

296

```
        ctx.restore();

        // 绘制秒针
        ctx.save();
          // 将 Canvas 旋转到当前的秒针位置
          ctx.rotate(sec * Math.PI/30);

          // 这根线段是接近红色的
          ctx.strokeStyle = "#D40000";
          ctx.fillStyle = "#D40000";

          // 这根线段要比其他指针都细
          ctx.lineWidth = 6;

          ctx.beginPath();
            // 而且起始点要偏离中心更远
            ctx.moveTo(-30,0);

            // 但更短
            ctx.lineTo(83,0);
          ctx.stroke();
        ctx.restore();

        // 绘制外侧蓝色圆圈
        ctx.save();
          // 边框必须足够粗
          ctx.lineWidth = 14;

          // 蓝色
          ctx.strokeStyle = '#325FA2';

          ctx.beginPath();
            // 绘制一个完整的圆圈，半径为 142 像素
            ctx.arc(0,0,142,0,Math.PI*2,true);
          ctx.stroke();
        ctx.restore();

        ctx.restore();
      }
    </script>
</head>
<body>
    <canvas id="canvas" height="150" width="150"></canvas>
</body>
</html>
```

297

一旦你理解了所有的细节和计算（这一点因绘制的复杂程度而异），Canvas 2D API 就会显得极为有用。

14.2.2 简单行星模拟

在第二个例子中，你将看到图像的旋转，并与直接绘制形状作对比。你可以从 3 个基本的图

像开始：太阳、地球和月球，然后创建一个简单的模拟（模拟起
298 来会很好看，但远不够精确）来展示地球围绕太阳的旋转和月球围绕地球的旋转。此外，还能粗略地展示地球哪一面因背向太阳而处在黑暗中。图 14-2 展示了实际运转起来的最终结果。

图14-2　地球围绕太阳转，月球围绕地球转，一个简单的Canvas行星模拟

你很可能注意到，这份代码中很多概念都和上一个例子类似（如存储和恢复 Canvas 位置），但需要注意的是如何处理单独图像的绘制和转换。代码清单 14-8 展示了 Canvas 行星模拟的完整代码。

代码清单14-8　使用Canvas 2D API来模拟地球围绕太阳旋转

```
<html>
<head>
    <title>Canvas Sun Demo</title>
    <script>
        // 初始化要使用的一系列图像
        var imgs = { sun: null, moon: null, earth: null };

        // 等待窗口完全载入
        window.onload = function() {
            // 从文档中载入所有的图像
            for ( var i in imgs )
                imgs[i] = document.getElementById(i);

            // 每秒绘制 10 次
            setInterval( draw, 100 );
        };

299     function draw() {
            // 获得我们所需的时间间隔
            var time = new Date();
            var s = ( (2 * Math.PI) / 6) * time.getSeconds();
            var m = ( (2 * Math.PI) / 6000 ) * time.getMilliseconds();

            // 获得 Canvas 的绘图上下文
            var ctx = document.getElementById('canvas').getContext('2d');

            // 清空 Canvas
            ctx.clearRect(0,0,300,300);

            // 新内容总在旧内容之下绘制 (以实现阴影)
            // 更多信息可见:
               http://developer.mozilla.org/en/docs/Canvas_tutorial:Compositing
            ctx.globalCompositeOperation = 'destination-over';

            ctx.save();
              // 现在在 0,0 点绘制等于是在 150,150 点绘制
              ctx.translate(150,150);

              // 将 Canvas 像地球的位置旋转
```

```
ctx.rotate( (s + m) / 10 );

// 向外移动 105 像素
ctx.translate(105,0);

// 填充阴影 (半透明状态)
ctx.fillStyle = 'rgba(0,0,0,0.4)';
ctx.strokeStyle = 'rgba(0,153,255,0.4)';

// 绘制阴影区域 (不甚完美，不过也算接近)
ctx.fillRect(0,-12,50,24);

// 绘制地球
ctx.drawImage(imgs.earth,-12,-12);

ctx.save();
  // 相对地球的旋转来转动 Canvas
  ctx.rotate( s + m );

  // 将月球放入“轨道”
  ctx.translate(0,28.5);

  // 绘制月球图案
  ctx.drawImage(imgs.moon,-3.5,-3.5);
ctx.restore();
ctx.restore();

// 绘制地球轨道
ctx.beginPath();
ctx.arc(150,150,105,0,Math.PI*2,false);
ctx.stroke();

// 绘制静态的太阳
ctx.drawImage(imgs.sun,0,0);
}
</script>
</head>
<body style="background:#000;">
<canvas id="canvas" height="300" width="300"></canvas>

<!-- 预先载入所有的源图像 -->
<div style="display:none;">
<img src="sun.png" id="sun"/>
<img src="moon.png" id="moon"/>
<img src="earth.png" id="earth"/>
</div>
</body>
</html>
```

300

<canvas>元素和其相关 API 最近吸引了很多注意，它们在 Apple 的 Dashboard 和 Opera 的 Widget 中得到了应用。这里揭示了“未来的”JavaScript 的一部分，而且现在已经得到了应用。

14.3　Comet

最后这个新概念是最近才被提出来并将成为事实标准。尽管 Ajax 的概念非常直接（单一的异步连接），但它并没有涵盖任何形式的串流内容。这种将一串更新（new update）流加入 Web 应用程序的概念，现在常被称做 *Comet*（由 Dojo 的 Alex Russell 所提出）。在 Web 应用中加入串流式更新能让开发的自由度更大，从而构建出能迅速响应的应用程序（如聊天程序）。

与 Ajax 并不包含新的技术很类似，Comet 也一样。不过许多开发者正在努力改进目前 Web 应用程序中用到的一般串流的概念，使之最终成为 *Cometd* 的标准。关于 Cometd 标准和 Comet 概念的更多信息，可以在这里找到：

- Alex Russell的原文，详细介绍Comet的概念：http://alex.dojotoolkit.org/?p=545。
- 定义Comet的规范：http://cometd.com/。

301

Comet 通过保持一个长期即时更新的连接，持续从某个中心服务器获取新的信息，由此来改进现有的 Ajax 应用程序。这个中心服务器必须保证通信均衡、准确送达正确的信道。图 14-3 展示了 Comet 和传统 Ajax 应用程序的区别。

大部分标准化 Comet 的工作，并且其迅速普及的工作都是由 Dojo 基金会负责的。代码清单 14-9 展示了一个运用 Dojo 程序库来初始化与服务器端资源的 Comet 连接的例子。

代码清单14-9　运用Dojo和它的Cometd库来连接一个串流服务器并监听不同的广播信道

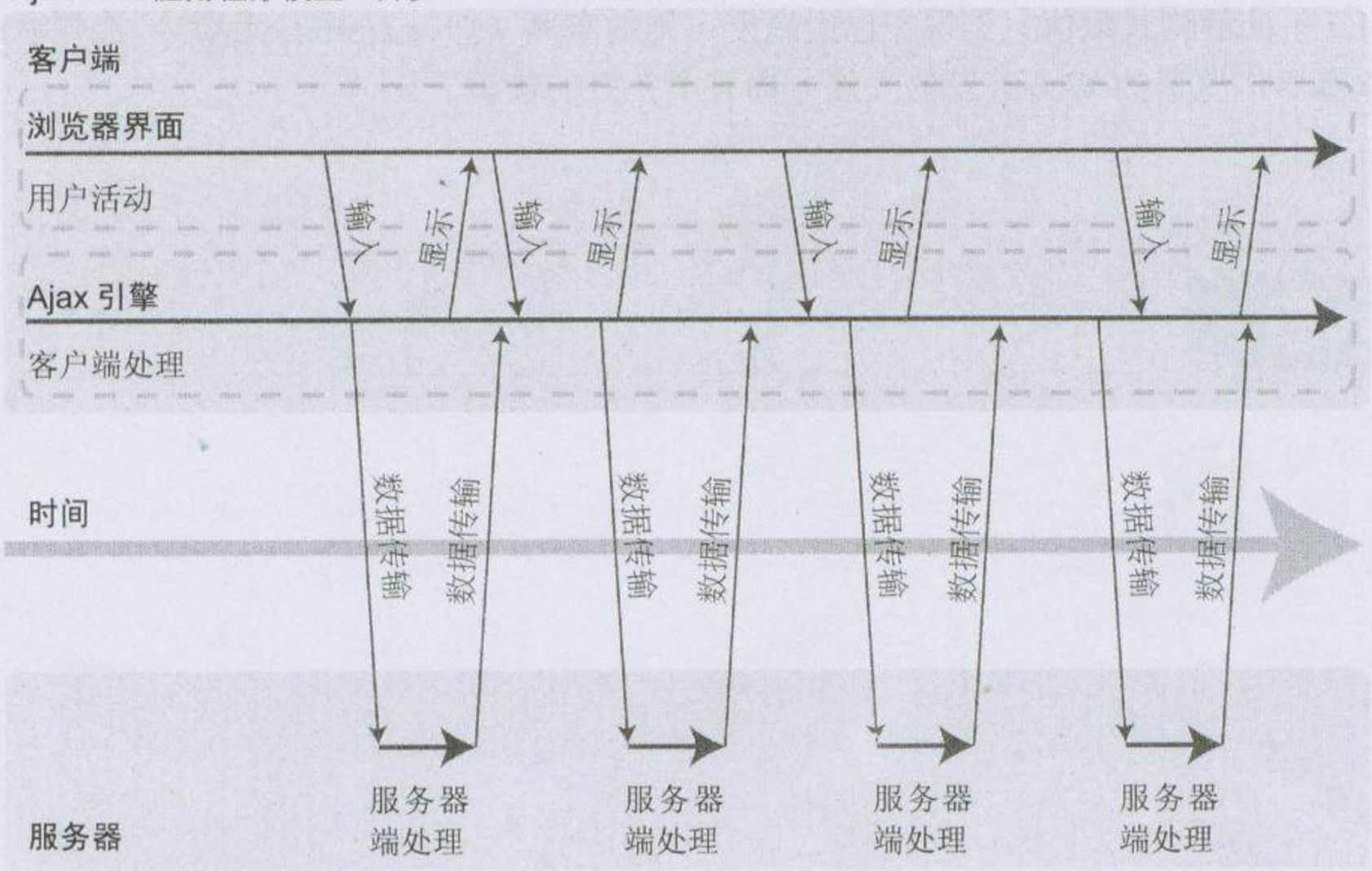
```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
    "http://www.w3.org/TR/html4/strict.dtd">
<html>
    <head>
    <title>Cometd Client/Server Test Page</title>
        <script type="text/javascript">
        // 将所有交互记录在调试面板
        djConfig = { isDebug: true };
        </script>
        <script type="text/javascript" src="../dojo/dojo.js"></script>
        <script type="text/javascript">
            dojo.require("dojo.io.cometd");
            dojo.addOnLoad(function(){
                // 设置所有 cometd 交互的基础 URL
                cometd.init({}, "/cometd");

                // 订阅一个广播服务，这样将会监听所有来自这一服务的串流输出，
                // 将其记录在调试面板中
                cometd.subscribe("/foo/bar/baz", false, dojo, "debugShallow");

                // 将两则消息广播到两个不同的服务中
                cometd.publish("/foo/bar/baz", { thud: "thonk!"});
                cometd.publish("/foo/bar/baz/xyzzy", { foo: "A simple message" });
            });
        </script>
    </head>
    <body></body>
</html>
```

尽管目前还只有一小部分 Comet（或类似 Comet 技术）的应用程序，但随着更多的人意识到这一技术对构建高性能的 Web 应用程序是多么有益，其应用程序的数量一定会急剧增加。 302

Ajax Web 应用程序模型（异步）



Comet Web 应用程序模型

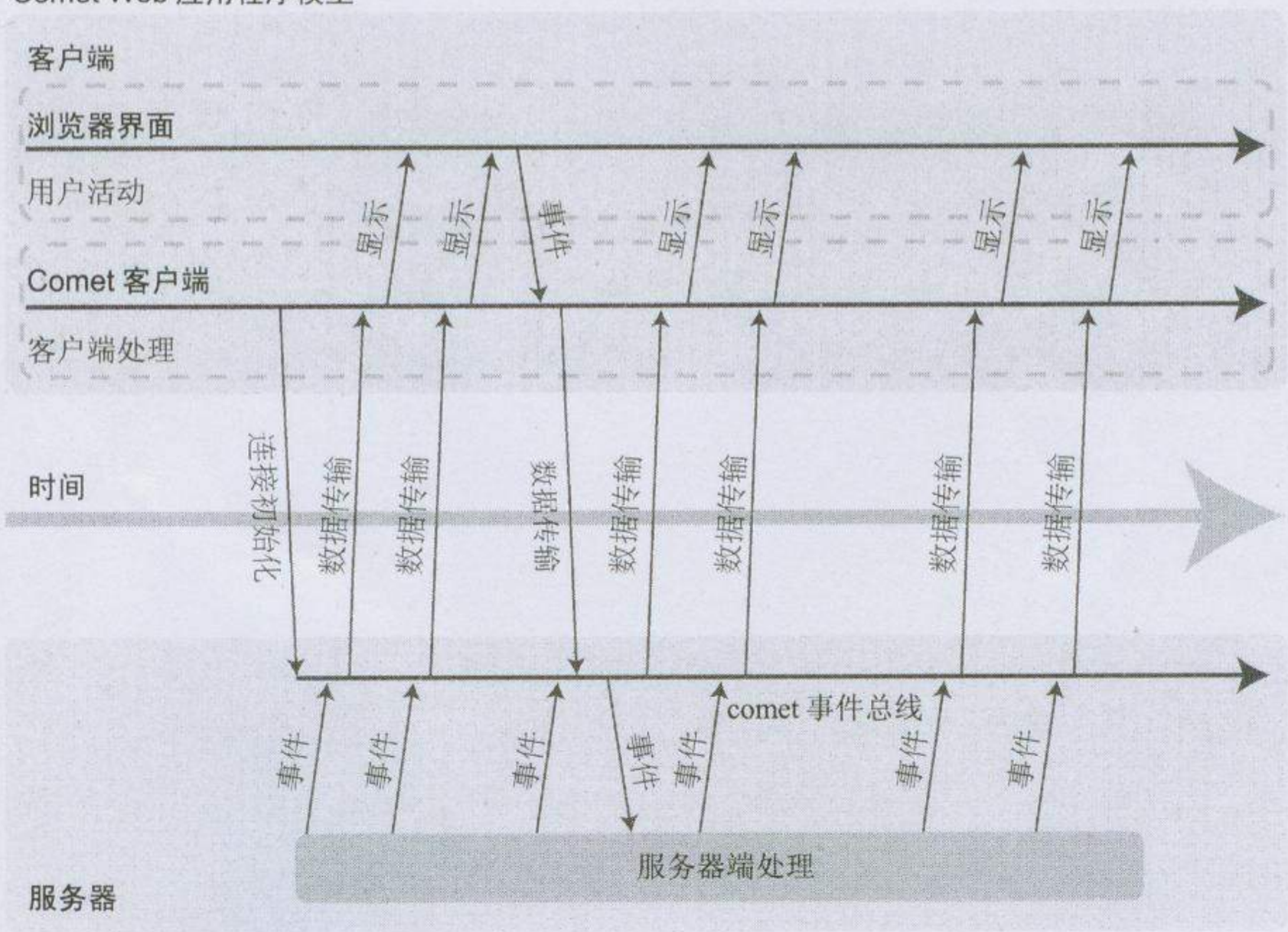


图14-3 对传统Ajax模型和新式Comet风格的Web应用程序模型之比较 303

14.4　小结

本章展示的技术涉及的范围很广，包括了所有从复杂而且距离现在很遥远的技术（如 JavaScript 1.7 里的解构式赋值）到现在已经高度可用的东西（如<canvas>元素）。希望能让你对
304 不远的未来基于浏览器的 Web 开发的发展方向有个良好的认识。

Part 6

第六部分

附　录

本部分内容

- 附录 A　DOM 参考手册
- 附录 B　事件参考手册
- 附录 C　浏览器

附录 A

DOM参考手册

这个附录作为第 5 章所讨论的 DOM 的功能参考。

A.1 资源

DOM 的功能描述来自多种渠道，从未规范的 0 级 DOM 到良好定义的 1 级和 2 级 DOM。因为所有的现代浏览器都几乎完全支持 W3C 的 1 级和 2 级 DOM，所以 W3C 网站是了解 DOM 如何工作的最佳参考：

- 1级DOM：http://www.w3.org/TR/REC-DOM-Level-1/level-one-core.html。
- 1级HTML DOM：http://www.w3.org/TR/REC-DOM-Level-1/level-one-html.html。
- 2级DOM：http://www.w3.org/TR/DOM-Level-2-Core/。
- 2级HTML DOM：http://www.w3.org/TR/DOM-Level-2-HTML/。

此外，虽然存在各种各样的参考，但是没有一个能比得上来自 Peter-Paul Koch 创建的 Quirksmode.org 上的资源。他彻底测试了现有可用的 DOM 方法并比较它们在所有现代浏览器（当然还有一些其他浏览器）中的运行结果。它是解决浏览器中的 DOM 究竟是什么、不是什么，或可能是什么的问题的无价资源：

- 1级和2级W3C DOM核心参考：http://www.quirksmode.org/dom/w3c_core.html。
- 1级和2级W3C DOM HTML参考：http://www.quirksmode.org/dom/w3c_html.html。

A.2 术语

关于第 5 章和此附录中的DOM，我使用通用的XML和DOM术语来描述XML文档的DOM实现的不同方面。下文中的名词和短语是关于DOM和XML文档常用术语。代码清单A-1 展示了一个HTML文档，下文中所有的术语例子都取自于这个代码清单。

代码清单A-1 讨论DOM和XML术语的参考文档

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
      <title>Introduction to the DOM</title>
```

```
</head>
<body>
      <h1>Introduction to the DOM</h1>
      <p class="test">There are a number of reasons why the DOM is awesome,
           here are some:</p>
      <ul>
           <li id="everywhere">It can be found everywhere.</li>
           <li class="test">It's easy to use.</li>
           <li class="test">It can help you to find what you want, really quickly.</li>
      </ul>
</body>
</html>
```

A.2.1 祖先（Ancestor）

与族谱术语非常相近，祖先引用当前元素的父元素、父元素的父元素、父元素的父元素的父元素，等等。在代码清单 A-1 中，<ul>元素的祖先元素是<body>元素和<html>元素。

A.2.2 特性（Attribute）

特性是指元素的属性，它包含元素相关的额外信息。在代码清单 A-1 中，<p>元素的特性 class 包含一个 test 的值。

A.2.3 儿子（Child）

任何元素都可以包含若干数量的节点（每一个均可当作父元素的子）。在代码清单 A-1 中，<ul>包含 7 个子节点，其中 3 个子节点是<li>元素，其他 4 个是出现在每个元素间的行结束符（包含文本节点）。

A.2.4 文档（Document）

XML 文档由一个元素（称为根节点或者文档元素）组成，它包含文档的所有其他信息。在代码清单 A-1 中，<html>是文档元素并包含文档的剩余部分。

A.2.5 后代（Descendant）

元素的后代包含它的子节点、子节点的子节点、子节点的子节点的子节点，等等。在代码清单 A-1 中，<body>元素的后代包括<h1>、<p>、<ul>，以及所有<li>元素和它们内部所包含的所有文本节点。

308

A.2.6 元素（Element）

元素是一个保存特性和其他节点的容器。最需要注意的是，任何 HTML 文档的组件都是它的元素。在代码清单 A-1 中存在诸如此类的元素：<html>、<head>、<title>、<body>、<h1>、<p>、<ul>和<li>等，这些标签都是元素的描述。

A.2.7　节点（Nodes）

节点是 DOM 实现的常用单位。元素、特性、注释、文档和文本都是节点，因此它们都会拥有典型的节点属性（比如，每一个节点都有的 nodeType、nodeName 和 nodeValue 等属性）。

A.2.8　父亲（Parent）

父亲是用以引用包裹当前节点的元素的术语。所有的节点都会有父亲，除了根节点。在代码清单 A-1 中，<p>元素的父亲是<body>元素。

A.2.9　兄弟（Sibling）

兄弟节点是拥有相同父节点的一组子节点集合。通常这个术语用在所有 DOM 节点都有的两个特性 previousSibling 和 nextSibling 的上下文中。在代码清单 A-1 中，<p>元素的兄弟是<h1>和<ul>元素（同时还有一些空白的文本节点）。

A.2.10　文本节点（Text Node）

文本节点是仅包含文本的特殊节点，它同时包含可见的文本和所有形式的空格。所以当你到看元素内的文本（比如，hello world!），实际上元素内还会有一个独立的包含"hello world!"的文本节点。在代码清单 A-1 中，第二个<li>元素的"It's easy to use"文本就是被包含在一个文本节点中的。

A.3　全局变量

尽管全局变量存在于代码的全局作用域中，但它们可以帮助你完成常见的 DOM 操作。

A.3.1　document

这个变量包含浏览器的 HTML DOM 文档的引用。但是，它并不会因为存在和拥有一个值就可以完全载入内容并进行解析。请翻阅第 5 章查看更多关于等待 DOM 加载的信息。document 变量拥有访问文档元素的 HTML DOM 实现，代码清单 A-2 展示了一些使用它的例子。

309

代码清单A-2　使用document变量访问文档元素

```
// 定位到ID为'body'的元素
document.getElementById("body")

// 定位到标签名为div的所有元素
document.getElementsByTagName("div")
```

A.3.2　HTMLElement

这个变量是所有 HTML DOM 元素的超类对象。扩展这个变量的原型（prototype）就可扩展所有的 HTML DOM 元素。这个超类是基于 Mozilla 的浏览器和 Opera 的默认变量，但也可以通

过第5章描述的方法添加到 IE 和 Safari 中去。代码清单 A-3展示了为全局 HTML 元素超类绑定一个新函数的例子。加上一个 hasClass 函数可以查看元素是否拥有一个特定的 class。

代码清单A-3 为全局HTML元素超类绑定一个新函数

```
// 为所有的HTML DOM元素增加一个新方法
// 可以用来查看一个元素是否拥有特定的class
HTMLElement.prototype.hasClass = function( class ) {
      return new RegExp("(^|\\s)" + class + "(\\s|$)").test( this.className );
};
```

A.4 DOM 的操作

以下属性作为所有 DOM 元素的组成部分，可以用来操作 DOM 文档。

A.4.1 body

这个全局 HTML DOM 文档（document 变量）的属性直接指向 HTML 的<body>元素（有且只存在一个）。这个特别的属性是从 DOM 0 时代遗留下来的。代码清单 A-4 展示了一些访问 HTML DOM 文档中的<body>元素的例子。

代码清单A-4 访问HTML DOM文档中的<body>元素

```
// 改变<body>的margin
document.body.style.margin = "0px";

// document.body等于:
document.getElementsByTagName("body")[0]
```

310

A.4.2 childNodes

这是所有 DOM 元素都有的属性，是一个包含所有子节点（包括元素、文本节点和注释等）的数组。该属性是只读的。代码清单 A-5 展示了如何使用 childNodes 属性来为一个父元素的所有子元素增加样式。

代码清单A-5 使用childNodes属性为<body>的子元素增加红色边框

```
// 为<body>的所有子元素增加边框
var c = document.body.childNodes;
for ( var i = 0; i < c.length; i++ ) {
      // 确保该节点是一个元素
      if ( c[i].nodeType == 1 )
          c[i].style.border = "1px solid red";
}
```

A.4.3 documentElement

这是一个在文档中作为所有 DOM 节点的根的节点（在 HTML 文档中，这总是指向<html>元素）。代码清单 A-6 展示了一个使用 documentElement 来查找 DOM 元素的例子。

代码清单A-6　从任何DOM节点定位到文档根元素的例子

```
// 获得documentElement，由ID查找一个元素
someRandomNode.documentElement.getElementById("body")
```

A.4.4　firstChild

这是所有 DOM 元素共有的一个属性，指向该元素的第一个子元素。如果元素没有子节点，则 firstChild 等于 null。代码清单 A-7 展示了一个使用 firstChild 属性来删除一个元素所有子节点的例子。

代码清单A-7　从一个元素中删除所有子节点

```
// 从一个元素中删除所有子节点
var e = document.getElementById("body");
while ( e.firstChild )
      e.removeChild( e.firstChild );
```

A.4.5　getElementByById(elemID)

这是一个定位到文档中特定 ID 的元素强大的函数。该函数只存在于 document 元素中。此外，该函数在非 HTML DOM 文档中可能不起作用，在 XML DOM 文档中一般需要在 DTD（文
311 档类型定义，Document Type Definition）或者配置（schema）中精确指定 ID 属性。

该函数只带一个参数：你需要查找的 ID 的名字，如代码清单 A-8 所示。

代码清单A-8　由其ID特性定位HTML元素的两个例子

```
// 使用名为body的ID查找元素
document.getElementById("body")

// 隐藏ID为notice的元素
document.getElementById("notice").style.display = 'none';
```

A.4.6　getElementsByTagName(tagName)

这个属性在所有的后代元素中——从当前元素开始——查找指定标签名的元素。该函数在 XML DOM 和 HTML DOM 文档中均起作用。

在所有的现代浏览器中，你可以指定 * 作为标签名而查找所有的后代元素，它比使用纯 JavaScript 的递归函数要快得多。

该函数也只需一个参数：需要查找元素的标签名。代码清单 A-9 展示了一个 getElementsByTagName 的例子。第二块查找 ID 为 body 的元素下所有元素，并隐藏所有 class 为 hilite 的元素。

代码清单A-9　展示getElementsByTagName如何使用的两个代码块

```
// 查找当前HTML文档所有<div>元素
// 并设置它们的class为'highlight'
var d = document.getElementsByTagName("div");
for ( var i = 0; i < d.length; i++ ) {
```

```
        d[i].className = 'hilite';
}

// 遍历ID为body的元素下所有后代元素。
// 然后查找所有的class为'hilite'的元素。
// 然后隐藏所有匹配的元素。
var all = document.getElementById("body").getElementsByTagName("*");
for ( var i = 0; i < all.length; i++ ) {
        if ( all[i].className == 'hilite' )
            all[i].style.display = 'none';
}
```

A.4.7 lastChild

这是一个所有 DOM 元素都可用的引用，指向元素的最后一个子节点。如果不存在子节点，则 lastChild 等于 null。代码清单 A-10 展示了一个使用 lastChild 属性来为文档插入元素的例子。

312

代码清单A-10 创建一个新的 元素并插入到<body>的最后一个元素之前

```
// 在<body>的最后一个元素之前插入一个新元素
var n = document.createElement("div");
n.innerHTML = "Thanks for visiting!";

document.body.insertBefore( n, document.body.lastChild );
```

A.4.8 nextSibling

这是一个所有 DOM 元素都可用的引用，指向下一个兄弟节点。如果节点是最后一个，则 nextSibling 是 null。需要记住的是 nextSibling 指向的可能是 DOM 元素，可能是注释甚至可能是文本节点，所以它并不是一个定位 DOM 元素的好方式。代码清单 A-11 是一个使用 nextSibling 属性来建立一个交互定义列表的例子。

代码清单A-11 一旦点击，使所有<dt>元素展开起相邻<dd>元素

```
// 查找所有<dt>(Definition Term, 定义的术语)元素
var dt = document.getElementsByTagName("dt");
for ( var i = 0; i < dt.length; i++ ) {
        // 监听术语元素是否被点击
        dt[i].onclick = function() {
            // 因为每一术语都会有一个相邻的<dd>(定义)元素t
            // 点击以后我们可以把它显示出来

            // 注意：仅在<dd>元素之间没有空格的时候起作用
            this.nextSibling.style.display = 'block';
        };
}
```

A.4.9 parentNode

这是一个所有 DOM 节点都有的属性。每一 DOM 节点的 parentNode 指向包含该节点的元

素，除了 document 元素指向 null 之外（因为没有任何东西能够包含根元素）。代码清单 A-12 是一个使用 parentNode 属性来创建自定义交互的例子。点击取消按钮会隐藏父元素。

代码清单A-12　使用parentNode属性来创建自定义交互

```
// 监听链接是否被点击（比如，一个取消链接），并隐藏父元素
document.getElementById("cancel").onclick = function(){
      this.parentNode.style.display = 'none';
};
```

313

A.4.10　previousSibling

这是一个所有 DOM 元素都可用的引用，指向前一个兄弟节点。如果该节点是第一个，则 previousSibling 是 null。需要记住的是 nextSibling 指向的可能是 DOM 元素，可能是注释甚至可能是文本节点，所以它并不是一个定位 DOM 元素的好方式。代码清单 A-13 展示了一个使用 previousSibling 属性来隐藏元素的例子。

代码清单A-13　隐藏当前元素的所有元素

```
// 查找该元素之前的所有元素并隐藏它们
var cur = this.previousSibling;
while ( cur != null ) {
      cur.style.display = 'none';
      cur = this.previousSibling;
}
```

A.5　节点信息

这些属性存在于大部分的 DOM 元素中，它们能帮助你更容易访问元素的一般信息。

A.5.1　innerText

这是一个所有 DOM 元素都可用的属性（由于它不是 W3C 标准的一部分只存在于非基于 Mozilla 的浏览器中）。该属性返回一个包含当前元素所有文本的字符串。由于基于 Mozilla 的浏览器不支持它，你可以使用类似第 5 章中的补救办法（使用函数来收集后代文本节点值）。代码清单 A-14 展示了一个使用 innerText 属性和第 5 章中的 text()函数的例子。

代码清单A-14　使用innerText属性来提取元素的文本信息

```
// 假设有这样一行的<li>元素，而且已经存储在变量'li'中：
// <li>Please visit <a href="http://mysite.com/">my web site</a>.</li>

// 使用innerText属性
li.innerText

// 或者第 5 章中描述的text()函数
text( li )

// 属性或者函数的结果都是：
"Please visit my web site."
```

314

A.5.2 nodeName

这是一个所有 DOM 元素都可用的属性，它是元素名字的大写形式。比如，你有一个<li>元素并且处理它的 nodeNode 属性，它返回的是 LI。代码清单 A-15 展示了一个使用 nodeName 属性来修改父元素的 class 的例子。

代码清单A-15 定位到所有<li>父元素并设置其class为'current'

```
// 查找该元素的所有父元素中的<li>元素
var cur = this.parentNode;
while ( cur != null ) {
      // 一旦找到元素，并确认了名字，添加一个class
      if ( cur.nodeName == 'LI' )
          cur.className += " current";
      cur = this.parentNode;
}
```

A.5.3 nodeType

这是一个所有 DOM 元素都可用的引用，指向一个跟文档类型相关的数字。在 HTML 文档中最常用的节点类型是以下 3 个：

- 元素节点（值为1或者document.ELEMENT_NODE）。
- 文本节点（值为3或者document.TEXT_NODE）。
- 文档节点（值为9或者document.DOCUMENT_NODE）。

使用 nodeType 属性是确保你所访问的节点具有你所预期的所有属性的可靠方式（比如，nodeName 属性仅在 DOM 元素中可用，所以你可能需要在访问它之前使用 nodeType 来确认它的值是否为 1）。代码清单 A-16 展示了一个使用 nodeType 属性为部分元素增加一个 class 的例子。

代码清单A-16 定位到HTML <body>的第一个元素并赋予一个class'header'

```
// 查找<body>的第一个元素
var cur = document.body.firstChild;
while ( cur != null ) {
      // 如果元素找到了，给它增加一个class 'header'
      if ( cur.nodeType == 1 ) {
          cur.className += " header";
          cur = null;

      // 否则，继续遍历子节点
    } else {
          cur = cur.nextSibling;
      }
}
```

315

A.5.4 nodeValue

这是一个文本节点十分有用的属性，它可用用来访问和操作其所包含的文本。最好的例子是

第 5 章中的 text 函数，它可以用来取得一个元素内的所有文本。代码清单 A-17 展示了一个使用 nodeValue 属性来构建一个简单的文本值函数的例子。

代码清单A-17　接受一个元素参数并返回它及其后代元素所包含的所有文本的函数

```
function text(e) {
    var t = "";

    // 元素传入后就获取其后代元素，否则假定它是一个数组
    e = e.childNodes || e;

    // 遍历所有子节点
    for ( var j = 0; j < e.length; j++ ) {
        // 如果不是一个元素，则追加起文本值
        // 否则，递归遍历元素的所有子节点
        t += e[j].nodeType != 1 ?
            e[j].nodeValue : text(e[j].childNodes);
    }

    // 返回匹配的文本
    return t;
}
```

A.6　特性

大部分的特性（attribute）都作为包含它们的元素的属性存在。例如，特性 ID 可以简单地使用 element.id 来访问。这是 DOM 0 时代的残留物，但由于过于简单和流行，它并不会扩展了。

A.6.1　className

这个属性允许你增加或者删除 DOM 元素的 class，它存在于有 DOM 元素中。需要特别指
316 出的是它的名字——className，跟我们所预期的 class 并不同。这个奇怪的命名是由于 class 是绝大部分面向对象编程语言的保留字，所以使用 className 是为了避免 Web 浏览器编程中的麻烦。代码清单 A-18 展示了一个使用 className 属性来隐藏部分元素的例子。

代码清单A-18　查找所有带特定class的 元素并隐藏它们

```
// 查找文档内的所有<div>元素
var div = document.getElementsByTagName("div");
for ( var i = 0; i < div.length; i++ ) {
    // 查找所有带'special' class的<div>元素
    if ( div[i].className == "special" ) {
        // 并隐藏它们
        div[i].style.display = 'none';
    }
}
```

A.6.2　getAttribute(attrName)

这个函数是用来访问DOM元素特性值的特有方法。HTML 文档会以用户所提供的特性初始化。

该函数带一个参数：你需要获得的特性的名字。代码清单A-19展示了一个使用getAttribute()函数来查找指定类型的input元素。

代码清单A-19 查找特性name为text的<input>元素并复制其值到一个ID为preview的元素中

```
// 查找所有表单的<input>元素
var input = document.getElementsByTagName("input");
for ( var i = 0; i < input.length; i++ ) {

    // 查找特性name为text的元素
    if ( input[i].getAttribute("name") == "text" ) {

        // 复制值到另一个元素
        document.getElementById("preview").innerHTML =
            input[i].getAttribute("value");

    }

}
```

A.6.3 removeAttribute(attrName)

这是一个用以完全删除元素某个特性的函数。一般来说，使用该函数的结果跟使用setAttribute设置一个""（空字符串）或者null值的结果一样，但在实践中，你可能得删除额外的特性，以防止未预期的后果。 317

该函数带一个参数：你需要删除的特性的名字。代码清单A-20展示了一个在表单中取消部分复选框（check box）的例子。

代码清单A-20 查找文档中所有复选框元素并取消选择

```
// 查找所有表单的input元素
var input = document.getElementsByTagName("input");
for ( var i = 0; i < input.length; i++ ) {

    // 查找所有的复选框元素
    if ( input[i].getAttribute("type") == "checkbox" ) {

        // 取消选择
        input[i].removeAttribute("checked");

    }

}
```

A.6.4 setAttribute(attrName, attrValue)

这个函数用以设置包含在DOM元素内的特性的值。此外，它还能增加自定义的特性，如果保持这个自定义的特性，那么它的值之后也可继续访问。setAttribute在IE中的表现相当怪异，不能设置某些特殊的特性（比如class或者maxlength）。这在第5章中有详细的描述。

这个函数带两个参数，一个是特性的名字，另一个是需要设置特性的值。代码清单A-21展

示了一个为 DOM 元素设置特性及其值的例子。

代码清单A-21　使用setAttribute函数来创建一个指向Google的<a>链接

```
// 创建一个新的<a>元素
var a = document.createElement("a");

// 设置URL为Google的链接
a.setAttribute("href","http://google.com/");

// 设置内部文本，给用户有可点的东西
a.appendChild( document.createTextNode( "Visit Google!" ) );

// 在文档末追加链接
document.body.appendChild( a );
```

318

A.7　DOM 修改

以下是修改 DOM 的所有可用的属性和函数。

A.7.1　appendChild(nodeToAppend)

这是一个可用来为元素追加子节点的函数。如果需要追加节点已存在于文档中，那么它会从原有位置移动到当前元素上。appendChild 函数必须在你所期望追加的元素上调用。

该函数带一个参数：DOM 节点的引用，也可以是新创建的或者文档中已有节点的引用）。代码清单 A-22 展示了一个例子，它创建一个新的<ul>元素并把原有 DOM 的<li>元素移入，然后把该<ul>追加到文档主体中。

代码清单A-22　追加一系列<li>元素到一个<ul>中

```
// 创建一个新的<ul>元素
var ul = document.createElement("ul");

// 查找所有的<li>元素
var li = document.getElementsByTagName("li");
for ( var i = 0; i < li.length; i++ ) {

        // 把每个匹配的<li>元素追加到新<ul>元素中
        ul.appendChild( li[i] );

}

// 把我们新的<ul>元素追加到body的最末
document.body.appendChild( ul );
```

A.7.2　cloneNode(true|false)

该函数是一种通过克隆已有的节点为开发者简化代码的方法，而克隆出来的节点也能够插入到 DOM 中去。因为一个普通的 insertBefore 或者 appendChild 会把文档中的 DOM 节点移开，但 cloneNode 函数能够克隆新的节点而保持原有节点的位置。

该函数有一个值是ture或者false的参数。如果参数是true，节点及其内部的所有东西都会克隆，如果是false，则只有节点本身被克隆。代码清单A-23展示了使用该函数克隆一个元素并追加到该元素后面的例子。

319

代码清单A-23 查找文档的第一个<ul>元素，完全复制它，并追加到<ul>元素之后

```
// 查找第一个<ul>元素
var ul = document.getElementsByTagName("ul")[0];

// 克隆节点并追加到旧元素之后
ul.parentNode.appendChild( ul.cloneNode( true ) );
```

A.7.3 createElement(tagName)

这个函数用来在DOM结构中创建新元素。该函数作为你需要创建元素的document的属性而存在。

> **注意** 如果你使用的是application/xhtml+xml的内容类型伺服的XHTML而不是常规的内容类型为text/html伺服的HTML，你应该使用createElementNS函数代替createElement函数。

该函数有一个参数：需要创建元素的标签名。代码清单A-24展示了一个使用该函数来创建元素并用它包裹其他元素内容的例子。

代码清单A-24 包裹含有<p>的内容的元素

```
// 创建一个新的<strong>元素
var s = document.createElement("strong");

// 查找第一个段落
var p = document.getElementsByTagName("p")[0];

// 在<strong>元素内包裹<p>的内容
while ( p.firstChild ) {
      s.appendChild( p.firstChild );
}

// 把<strong>元素（包括<p>之前的内容）放回<p>中
p.appendChild( s );
```

A.7.4 createElementNS(namespace, tagName)

这个函数跟createElement非常相似，都能创建一个新元素，但同时它还能够为元素指定一个命名空间（比如，为XML或者XHTML文档增加项目）。

320

该函数有两个参数：待增加元素的命名空间和元素的标签名。代码清单A-25展示了一个使用该函数在正确的XHTML文档来创建DOM元素的例子。

代码清单A-25　创建一个新的XHTML <p>元素，填充一些文本，并追加到文档主体中

```
// 创建一个兼容XHTML的<p>元素
var p = document.createElementNS("http://www.w3.org/1999/xhtml", "p");

// 为<p>元素增加一些文本
p.appendChild( document.createTextNode( "Welcome to my site." ) );

// 把<p>元素追加到文档中
document.body.insertBefore( p, document.body.firstChild );
```

A.7.5　createTextNode(textString)

这是创建新的待追加到 DOM 中的字符串的合适方法。因为文本节点仅是文本的 DOM 包裹器，需要记住它们不能样式化或者在它之中追加内容。该函数仅可作为 DOM 文档的一个属性。

该函数带一个参数：文本节点内容的字符串。代码清单 A-26 展示了使用该函数来创建一个新的文本节点并追加到 HTML 页面主体中的例子。

代码清单A-26　创建一个新的<h1>元素并追加一个新的文本节点

```
// 创建一个新的<h1>元素
var h = document.createElement("h1");

// 创建标题文本并追加到<h1>元素中
h.appendChild( document.createTextNode("Main Page") );

// 把标题插入到<body>的开始
document.body.insertBefore( h, document.body.firstChild );
```

A.7.6　innerHTML

这是 HTML DOM 特有的属性，用以访问和操作 DOM 元素的 HTML 内容的字符串形式。如果只需要使用 HTML 文档（而非 XML），该方法十分有用，因为使之生成一个新的 DOM 元素可以在极大程度上精简代码（更不用提它的速度比传统的 DOM 方法要快得多）。虽然此属性并非 W3C 某个标准的组成部分，但仍能在所有的现代浏览器上使用。代码清单 A-27 展示了一个使用 innerHTML 属性的例子，它在<textarea>变化的时候改变元素的内容。

321

代码清单A-27　监听<textare>的变化并使用它的值更新一个实时的预览

```
// 获取需要监听的textarea
var t = document.getElementsByTagName("textarea")[0];

// 捕获<textarea>的每次改变时的当前值并更新实时预览
t.onkeypress = function() {
      document.getElementById("preview").innerHTML = this.value;
};
```

A.7.7　insertBefore(nodeToInsert, nodeToInsertBefore)

这是一个用来在文档的任意地方插入 DOM 节点的函数。该函数必须在你希望插入到前面的

元素的父元素上调用。所以在完成后，可以明确指定 nodeToInsertBefore 为 null，而且节点的作为最后一个子节点插入。

该函数带两个参数。一个是你希望插入到 DOM 中的节点，另一个是需要插入前面位置的 DOM 节点。它们都应该是正确的节点引用。代码清单 A-28 展示了一个使用该函数为一组页面 URL 插入站点 favicon（在浏览器地址栏 URL 旁的 icon）的例子。

代码清单A-28 遍历所有<a>元素并增加站点favicon的icon

```
// 查找文档内的所有<a>链接
var a = document.getElementsByTagName("a");
for ( var i = 0; i < a.length; i++ ) {

    // 创建一个链接到站点favicon的图片
    var img = document.createElement("img");
    img.src = a[i].href.split('/').splice(0,3).join('/') + '/favicon.ico';

    // 把图片插入到链接之前
    a[i].parentNode.insertBefore( img, a[i] );
}
```

A.7.8 removeChild(nodeToRemove)

这个函数用来删除文档的节点。该函数必须在你希望删除节点的父元素上调用。

该函数有一个参数：需要从文档中删除的 DOM 节点引用。代码清单 A-29 展示了一个遍历文档所有 元素，并将所有 class 为 warning 的 删除的例子。

322

代码清单A-29 删除所有带特定class名字的元素

```
// 查找所有<div>元素
var div = document.getElementsByTagName("div");
for ( var i = 0; i < div.length; i++ ) {
    // 如果该div的class是'warning'
    if ( div[i].className == "warning" ) {

        // 那么从文档中删除它们
        div[i].parentNode.removeChild( div[i] );

    }
}
```

A.7.9 replaceChild(nodeToInsert, nodeToReplace)

这个函数是删除一个节点并在原地插入另外一个节点的替代方法。该函数必须在你需要替换的节点的父元素上调用。

该函数带两个参数：你希望插入到 DOM 的节点和希望替换的节点。代码清单 A-30 展示了一个例子，它把所有<a>元素替换为包含原有 URL 的元素。

代码清单A-30 转换一组链接为无格式URL

```
// 转换所有的链接为可视的URL（对打印有用处）
```

```
// 查找文档中的所有<a>链接
var a = document.getElementsByTagName("a");
while ( a.length ) {

        // 创建一个<strong>元素
        var s = document.createElement("strong");

        // 使内容等于<a>链接的URL
        s.appendChild( document.createTextNode( a[i].href ) );

        // 使用<strong>元素替换原有<a>元素
        a[i].replaceChild( s, a[i] );
}
```

323

附录 B

事件参考手册 B

这个附录作为第 6 章事件的补充参考，全面覆盖所有可能的 DOM 事件，是通用理论的进一步阐述。在此你可找到更多的资源、相关事件的常用术语以及所有通用事件对象和交互的详细解释。

B.1 资源

如果关于事件的资源只让我们介绍一个，那非 Quirksmode.org 莫属。该站点为所有现代浏览器提供事件的并行对比。特别推荐你访问该站点并探索各个浏览器对事件的支持(http://www.quirksmode.org/js/events_compinfo.html)。

此外，现在常用的两个热门资源是 W3C 的 DOM 事件和 IE 的 HTML 事件。它们分别提供了全面的事件代码清单和表现的种种方面。

- W3C DOM第2级事件：http://www.w3.org/TR/DOM-Level-2-Events/events.html。
- IE HTML事件：http://msdn.microsoft.com/workshop/author/dhtml/reference/ events.asp。

B.2 术语

这部分定义一系列关于处理 JavaScript 事件的新术语，没有接触过 JavaScript 事件或者异步事件处理的人对此可能会感到有些陌生。

1. 异步

异步事件使用通用的回调函数结构，跟线程程序结构形成鲜明对比。这意味着一块独立的代码（回调函数）注册到一个事件处理函数之上。一旦事件发生，回调函数就会执行。

2. 附加/绑定/注册回调函数

附加（有时称为绑定）事件到事件处理函数上是指代码如何注册到一个异步事件模型上。一旦事件发生就调用事件处理函数，它是一个包含已注册回调函数的引用。回调函数是一段函数引用形式的代码，它在事件完成后即可调用。

3. 冒泡

事件冒泡阶段在事件捕获阶段之后发生。冒泡阶段开始于事件的源头（比如用户点击的链接）并一直上升到文档 DOM 树的根上。

4. 捕获

事件捕获阶段（只会在 W3C 事件模型中发生）是首先发生的事件阶段，它包含 DOM 树中一个移动到事件初始化元素的位置上的事件。

5. 默认行为

默认行为是基于浏览器的行为，无论用户是否绑定事件处理函数都会发生。一个浏览器默认行为的典型例子是用户点击链接，会把你引向另一个页面。

6. 事件

事件是页面内发生的行为。一般来说，事件由用户发起（比如移动鼠标、敲击键盘等），但也可以由非交互的行为触发（比如页面加载、或错误的发生等）。

7. 事件处理函数

事件处理函数（比如函数引用）是事件发生时的代码调用。如果没有回调函数注册到事件处理函数上，不会发生任何事情（除了默认行为）。

8. 线程

在线程程序中，通常会存在一些不同的进程流，用以执行一个持续的任务（如检查资源是否可用）。JavaScript 不存在任何方面线程的支持，它只是一门异步语言。

B.3　事件对象

事件对象是事件处理函数提供的，或者存在于它自身内的对象。IE 与其他浏览器的处理方式有所不同。

W3C 兼容的浏览器为事件处理函数提供独立的参数，它包含一个事件对象的引用。IE 的事
326 件对象总是存在于 `window.event` 中，它只能在事件处理函数内访问。

B.3.1　通用属性

在被捕获时事件对象会存在一系列的属性。所有这些事件对象属性直接关联事件本身而非事件类型特有。以下是所有事件对象类型的解释和代码样例。

1. 类型（type）

这个属性包含当前触发的事件名称（比如 `click`、`mouseover` 等）。这用以提供一个事件处理函数，以确定执行相关函数（比如第 6 章中讨论的 `addEvent/removeEvent`）。代码清单 B-1 展示了一个使用该属性来创建一个具有不同效果的处理函数，不同的效果取决于事件的类型。

代码清单B-1　使用type属性为元素提供类似于hover的效果

```
// 定位我们需要hover的<div>
var div = document.getElementsByTagName('div')[0];

// 同时为mouseover和mouseout绑定一个函数
div.onmouseover = div.onmouseout = function(e){
        // 标准化事件对象
```

```
        e = e || window.event;

        // 切换<div>的背景颜色，取决于所发生的鼠标事件类型
        this.style.background = (e.type == 'mouseover') ? '#EEE' : '#FFF';
};
```

2. 目标（target/srcElement）

这个属性包含触发事件的元素的引用。比如，绑定点击处理函数到<a>元素会使其拥有一个等同于本身的 target 属性。srcElement 是等同于 target 的 IE 版本。代码清单 B-2 展示了一个使用该属性来操作全局事件处理函数的例子。

代码清单B-2 双击HTML DOM中的任何节点并删除它

```
// 为document绑定一个双击监听函数
document.ondblclick = function(e) {
        // 标准化事件对象
        e = e || window.event;

        // 查找正确的目标节点
        var t = e.target || e.srcElement;

        // 从DOM中删除节点
        t.parentNode.removeChild( t );
};
```

3. 停止冒泡（stopPropagation()/cancelBubble）

stopPropagation()方法能阻止事件冒泡（或者捕获）的进行，使得当前元素成为接收特定事件的最后一个。事件阶段在第 6 章中有具体描述。cancelBubble 属性可用在 IE 中，设置其值为 true 时等同于兼容 W3C 的浏览器使用 stopPropagation()方法。代码清单 B-3 展示了一个使用该技术来停止事件冒泡的例子。

代码清单B-3 动态高亮文档内的所有<li>元素

```
// 查找文档内的所有<li>元素
var li = document.getElementsByTagName('li');
for ( var i = 0; i < li.length; i++ ) {

        // 监听用户是否移动鼠标到<li>上
        li[i].onmouseover = function(e){

            // 如果是兼容W3C的浏览器
            if ( e )
                // 使用stopPropogation来停止事件冒泡
                e.stopPropagation();

            // 否则，处理Internet Explorer
            else
                // 因此设置 cancelBubble 为 true  来停止事件冒泡
                e.cancelBubble = true;

            // 最后，高亮<li>的背景
```

```
        this.style.background = '#EEE';
    };

    // 当鼠标移出<li>
    li[i].onmouseout = function(){
        // 重置背景为白色
        this.style.background = '#FFF';
    };

}
```

4. 阻止默认行为（preventDefault()/returnValue=false）

调用 preventDefault()方法可以阻止兼容 W3C 的现代浏览器的默认行为。IE 则需要设置
328 事件对象的 returnValue 属性的值来阻止浏览器的默认行为。

关于浏览器默认行为的讨论可以参考第 6 章。代码清单 B-4 中的代码效果是，在页面上点击链接时，不是访问页面（普通情况下会），而是设置文档的标题为链接的 URL。

代码清单B-4　停止默认的浏览器行为

```
// 定位到页面的所有<a>元素
var a = document.getElementsByTagName('a');
for ( var i = 0; i < a.length; i++ ) {

    // 为<a>绑定一个点击事件处理函数
    a[i].onclick = function(e) {
        // 不访问该链接，而是设置页面的标题为该链接
        document.title = this.href;

        // 阻止浏览器访问指向<a>链接的页面（它是默认的行为）
        if ( e ) {
            e.preventDefault();

        // 在IE中阻止默认行为
        } else {
            window.event.returnValue = false;
        }
    };

}
```

B.3.2　鼠标属性

鼠标属性仅在鼠标相关的事件（比如 click、mousedown、mouseup、mouseover、mousemove 和 mouseout）发生时才存在于事件对象中。而其余场合，你可以假定这些返回值并不存在或不确定。这一部分列举了所有鼠标事件对象的属性。

1. clientX/clinetY

这两个属性包含相对于窗口的鼠标光标的 x 和 y 坐标。这两个属性的例子如代码清单 B-5 的

代码所示。

代码清单B-5 查找网页内鼠标光标的当前位置

```
// 查找光标的水平坐标
function getX(e) {
      // 先是非IE的位置，然后是IE的，最后一个返回0
      return e.pageX || (e.clientX +
          (document.documentElement.scrollLeft || document.body.scrollLeft));
}

// 查找光标的垂直坐标
function getY(e) {
    // 先是非IE的位置，然后是IE的，最后一个返回0
    return e.pageY || (e.clientY +
        (document.documentElement.scrollTop || document.body.scrollTop));
```

329

2. pageX/pageY

这两个属性包含相对于呈现文档的鼠标光标的 x 和 y 坐标（例如，如果你向下滚动文档，得到的数字将不会等于 clinetX/clinetY 属性所包含的值）。它们在 IE 中无效。要在 IE 中获取光标的位置，你必须使用 clientX/clientY 属性并加上当前的滚动偏移值。

3. layerX/layerY和offsetX/offsetY

这些属性包含相对于事件目标元素的鼠标光标的 x 和 y 坐标。layerX/layerY 属性可用在基于 Mozilla 的浏览器和 Safari，而 offsetX/offsetY 则可用在 Opear 和 IE（你可以看看代码清单 B-17 中的例子。）

4. button

这个属性是表示当前点击（仅可用于 click、mousedown 和 mouseup 事件）的鼠标键的数字。不幸的是，使用什么数字表示敲击哪个鼠标键也比较混乱。幸好，所有的浏览器都使用 2 来表示右键的点击，所以你可以在最低限度下放心地使用它。表 B-1 展示了 IE 和兼容 W3C 浏览器 button 属性所有可能的值。

表 B-1 事件对象的 button 属性可能的值

点　击	IE	W3C
左键	1	0
右键	2	2
中键	4	1

代码清单 B-6 展示了一段代码，它防止网页的右键点击事件（会出现菜单）。

代码清单B-6 使用事件对象的button属性

```
// 绑定点击处理函数到整个文档
document.onclick = function(e) {
      // 标准化事件对象
      e = e || window.event;
```

330

```
        // 如果执行的是右键点击
        if ( e.button == 2 ) {
            // 阻止发生默认的行为
            e.preventDefault();
            return false;
        }
};
```

5. relatedTarget

这个事件对象包含一个鼠标刚离开的元素的引用。除此之外，这可用在这样的场合：需要使用 mouseover/mouseout，而且必须知道鼠标所处，或者即将进入的元素。代码清单 B-7 展示了一个树形菜单（<ol>元素包含其他的<ol>元素），它在用户首次移动鼠标到次级<li>元素上时显示次级分支。

代码清单B-7　使用relatedTarget属性来创建导航树

```
// 查找文档的所有<li>元素
var li = document.getElementsByTagName('li');
for ( var i = 0; i < li.length; i++ ) {

    // 并为它们绑定mouseover事件处理函数
    li[i].onmouseover = function(e){

        // 如果鼠标首次进入（来自父元素）
        if ( e.relatedTarget == this.parentNode ) {
            // 显示最后一个子元素 （发生在另一个<ol>上）
            this.lastChild.style.display = 'block';
        }

    };

}

// HTML样本:
<ol>
    <li>Hello <ol>
        <li>Another</li>
        <li>Item</li>
    </ol></li>
    <li>Test <ol>
        <li>More</li>
        <li>Items</li>
    </ol></li>
</ol>
```


B.3.3　键盘属性

键盘属性一般只在键盘相关的事件（比如 keydown、keyup 和 keypress）发生时才存在于事件对象中，除了 ctrlKey 和 shiftKey 属性，它们可以在鼠标事件中存在（这就允许 Ctri+Click 一个元素）。而其余场合，你可以假定这些返回值并不存在或不确定。

1. ctrlKey

该属性返回一个布尔值，表示键盘的 Ctrl 键是否被按住。该属性可存在于键盘和鼠标事件中。代码清单 B-8 中的代码监听用户是鼠标否点击并按住了 Ctrl 键，一旦如此，被点击的元素会从文档中删除。

代码清单B-8 使用ctrlKey属性来创建一种鼠标点击的交互

```
// 绑定一个点击处理函数到整个文档
document.onclick = function(e){
        // 标准化事件对象
        e = e || window.event;
        var t = e.target || e.srcElement;

        // 点击的同时如果Ctrl键被按住
        if ( e.ctrlKey )

            // 删除被点击的元素
            t.parentNode.removeChild( t );
};
```

2. keyCode

这个属性包含一个键盘相应键位的数字。某些键位（比如 Page Up 和 Home 键）的可用性并不能确定，但一般来说，其他的键位均工作稳定。表 B-2 是一个常用键位和其相应代码的参考。

表 B-2 常用键位代码

键　位	代　码
退格键（Backspace）	8
制表键（Tab）	9
回车键（Enter）	13
空格键（Space）	32
左箭头键（Left arrow）	37
上箭头键（Up arrow）	38
右箭头键（Right arrow）	39
下箭头键（Down arrow）	40
0~9	48~57
A~Z	65~90

332

代码清单 B-9 展示了运行简单幻灯的必要代码。该代码假定部分的<li>元素在一个<ol>或者<ul>元素内。每个<li>可以包含任何东西（比如图片）。当敲击左箭头和右箭头键时，上一个或下一个<li>会显示在用户面前。

代码清单B-9 使用keyCode属性来创建一个简单的幻灯片

```
// 定位到页面的第一个<li>元素
var cur = document.getElementsByTagName('li')[0];

// 并确保它是可见的
cur.style.display = 'block';
```

```
// 监听页面的任何keypress事件
document.onkeyup = function(e){
        // 标准化事件对象
        e = e || window.event;

        // 如果敲击了左箭头键或者有箭头键
        if ( e.keyCode == 37 || e.keyCode == 39 ) {

            // 隐藏当前显示的<li>元素
            cur.style.display = 'none';

            // 如果左箭头键被敲击，查找上一个<li>元素
            // 或者位于一个元素，则跳到最后一个
            if ( e.keyCode == 37 )
                cur = cur.previousSibling || cur.parentNode.lastChild;

            // 如果右箭头键被敲击，查找下一个<li>元素
            // 或者在到了最后一个元素，则跳到第一个
            else if ( e.keyCode == 39 )
                cur = cur.nextSibling || cur.parentNode.firstChild;

            // 在序列中显示下一个<li>
            cur.style.display = 'block';
        }

333     };
```

3. shiftKey

这个属性返回一个布尔值，表示 Shift 键是否被按住。该属性可存在于键盘和鼠标事件中。代码清单 B-10 展示的代码监听用户是否鼠标点击并按住了 Shift，一旦如此，将显示一个上下文菜单。

代码清单B-10　使用shiftKey属性来显示特殊菜单

```
// 绑定一个点击处理函数到整个文档
document.onclick = function(e){
        // 标准化事件对象
        e = e || window.event;

        // 如果点击的同时shift键被按住
        if ( e.shiftKey )
            // 显示菜单
            document.getElementById('menu').style.display = 'block';
};
```

B.4　页面事件

所有的页面事件都明确地处理整个页面的函数和状态。主要是处理加载和卸载页面（用户访问页面和离开页面）的事件类型。

B.4.1　load

load 事件在页面完全加载完毕的时候触发。该事件包含所有的图片、外部 JavaScript 文件和

外部 CSS 文件。这可以当作是运行 DOM 相关代码的一种方法，但是，如果需要一个更快的响应时间，你应该查看第 6 章中的 domReady() 函数。

代码清单 B-11 展示的代码等待页面加载，在它之内又绑定一个点击处理函数到一个 ID 为 cancel 的元素上。一旦触发了点击处理函数，会隐藏 ID 为 main 的元素。

代码清单B-11 使用load事件来等待整个页面的加载

```
// 等待页面完成加载
window.onload = function(){

    // 定位到ID为cancel的元素上并绑定点击处理函数
    document.getElementById('cancel').onclick = function(){

        // 一旦点击，隐藏'main'元素
        document.getElementById('main').style.display = 'none';

    };

};
```

334

B.4.2 beforeunload

这个事件有些古怪，它完全不是标准的但却得到广泛的支持。它的表现与 unload 事件非常接近，不过有一个非常明显的区别。在 beforeunload 事件的事件处理函数内，如果返回的是字符串，那么字符串就会显示在一个确认窗口中，询问用户是否希望离开当前页面。动态的 Web 应用，比如 Gmail，使用该事件来防止用户丢失潜在的未保存数据。

代码清单 B-12 附加一个简单的事件处理函数（仅返回一个字符串）阐述了用户为何不应该离开当前所在页面的理由。浏览器会显示一个有解释理由的确认框，当然也包含你的自定义信息。

代码清单B-12 使用beforeunload事件来防止用户离开页面

```
// 附加到beforeunload处理函数上
window.onbeforeunload = function(){

    // 返回用户不该离开页面的理由
    return 'Your data will not be saved.';

};
```

B.4.3 error

error 事件在 JavaScript 代码发生错误时触发。这可以作为捕捉并优雅地展现和处理错误信息的方法。这个事件处理函数的表现跟其他事件的有所不同，它无需传递事件对象参数，而包含一条已发生错误的解释信息。

代码清单 B-13 展示了一个处理并在列表内显示错误的自定义方式，而不是传统的错误终端方式。

代码清单B-13　使用error事件生成一个可视化的错误记录

```
// 附加一个错误事件处理函数
window.onerror = function( message ){

        // 创建一个<li>元素来保存我们的错误信息
        var li = document.createElement('li');
        li.innerHTML = message;

        // 找出显示错误的列表（它有一个ID是'errors'）
        var errors = document.getElementById('errors');

        // 并把错误信息添加到列表的顶端去
        errors.insertBefore( li, errors.firstChild );

};
```

335

B.4.4　resize

resize 事件在用户重置浏览器窗口时触发。当用户调整浏览器窗口的尺寸时，resize 事件仅在完成重置的事件触发一次，而不是每一步都会触发。

代码清单 B-14 展示了一段代码，它监听用户重置浏览器窗口，一旦尺寸过小，它会为 document 元素赋予一个其他的 class（为了给小窗口提供更好的文档样式化）。

代码清单B-14　使用resize事件来动态重置一个元素的尺寸

```
// 监听用户是否重置浏览器窗口
window.onresize = function() {
        // 定位到document元素（可用于找出浏览器宽度）
        var de = document.documentElement;

        // 找出浏览器宽度（不幸的是，每种浏览器的处理都不尽相同）
        var w = window.innerWidth || (de && de.clientWidth)
            || document.body.clientWidth;

        // 如果window过小，为document元素增加一个class
        de.className = w < 990 ? 'small' : '';
};
```

B.4.5　scroll

scroll 事件于用户在浏览器窗口内移动文档的位置时触发。这会在键盘敲击（比如箭头，翻页或者空格键）或使用滚动条时触发。

B.4.6　unload

这个事件在用户离开当前页（可以是点击链接，后退键甚至是关闭浏览器窗口）时触发。阻止默认行为并不会在此事件中生效（这可是 beforeunload 的又一优点）。

代码清单 B-15 展示的代码，它为 unload 事件绑定一个事件处理函数，在用户离开当前页时显示一条信息。

336

代码清单B-15 unload事件

```
// 监听用户是否要离开当前页
window.onunload = function(){

        // 给用户显示一条信息，感谢他们的访问
        alert( 'Thanks for visiting!' );

};
```

B.5 UI 事件

UI 事件专门处理用户与浏览器或页面元素本身的交互。UI 事件可以帮助你找到用户正在交互的页面元素，并为他们提供更多的上下文（比如高亮或者帮助菜单）。

B.5.1 focus

focus 事件是确定页面内鼠标的当前定位的方式。它默认在整个文档内，但是，点击或者使用键盘 Tab 键切换到任何一个链接或表单输入元素，focus 就会进入该元素（例子见代码清单 B-18）。

B.5.2 blur

blur 事件在用户从一个元素到另一元素改变焦点（在上下文链接、输入元素或者页面自身内）时触发（例子见代码清单 B-18）。

B.6 鼠标事件

鼠标事件在用户移动鼠标光标或者使用任意鼠标键点击时触发。

B.6.1 click

click 事件于用户在元素敲击鼠标左键（见 mousedown 事件）并在相同的元素上松开左键（见 mouseup 事件）时触发。代码清单 B-16 展示了一个使用 click 事件来防止用户访问指向当前页面的链接的例子。

337

代码清单B-16 禁止点击所有意图指向当前页面的链接

```
// 查找文档内的所有<a>元素
var a = document.getElementsByTagName('a');
for ( var i = 0; i < a.length; i++ ) {

        // 如果链接指向我们所处的当前页面
        if ( a[i].href == window.location.href ) {

            // 那么让它的点击不再生效
            a[i].onclick = function(e){
                return false;
            };
        }

}
```

B.6.2 dblclcik

dblclick 事件在用户完成迅速连续的两次点击后触发。双击的速度取决于操作系统的设置。

B.6.3 mousedown

mousedown 事件在用户敲击鼠标键时触发。跟 keydown 事件不一样，该事件仅在按下鼠标的时候触发。这个事件的例子，请参考代码清单 B-17 所展示的代码。

B.6.4 mouseup

mouseup 事件在用户松开鼠标时触发。如果在与按下鼠标的相同元素上松开，那么 click 事件也会触发。这个事件的例子，请参考代码清单 B-17 所展示的代码。

B.6.5 mousemove

mousemove 事件于用户在页面内鼠标光标移动至少一像素时触发。mousemove 事件触发的多寡（鼠标的完全移动而言）取决于用户的移动速度和浏览器跟踪更新的速度。代码清单 B-17 展示了一个简单的拖放实现。

代码清单B-17　用户可拖放拥有class为draggable的元素

338

```
// 初始化我们需要使用的所有的变量
var curDrag, origX, origY;

// 监听用户对元素的每次点击
document.onmousedown = function(e){
    // 标准化事件对象
    e = fixEvent( e );

    // 仅拖放有class为'draggable'的元素
    if ( e.target.className == 'draggable' ) {
        // 我们当前拖放的元素
        curDrag = e.target;

        // 记录光标的开始位置和元素的位置
        origX = getX( e ) + (parseInt( curDrag.style.left ) || 0);
        origY = getY( e ) + (parseInt( curDrag.style.top ) || 0);

        // 监听鼠标的移动或停止
        document.onmousemove = dragMove;
        document.onmouseup = dragStop;
    }

};
// 监听鼠标的移动
function dragMove(e) {
    // 标准化事件对象
    e = fixEvent( e );
```

```
    // 确保我们监听的是正确的元素
    if ( !curDrag || e.target == curDrag ) return;

    // 设置光标的新位置
    curDrag.style.left = (getX(e)) + 'px';
    curDrag.style.top = (getY(e)) + 'px';
}

// 监听拖放的结束
function dragStop(e) {
    // 标准化事件对象
    e = fixEvent( e );

    // 重置我们所有的监听函数
    curDrag = document.mousemove = document.mouseup = null;
}

// 调整事件对象，使其更健壮
function fixEvent(e) {
    // 让IE特有的参数W3C化
    if (!e) {
        e = window.event;
        e.target = e.srcElement;
        e.layerX = e.offsetX;
        e.layerY = e.offsetY;
    }
    return e;
}
```

339

B.6.6 mouseover

mouseover 事件于用户把鼠标从一个元素移动到另一个元素上时触发。如果需要知道来自哪个元素，可以使用 relatedTarget 属性。关于这个事件的例子，请参考代码清单 B-18 所展示的代码。

B.6.7 mouseout

mouseout 事件在用户把鼠标移出一个元素时触发。这包括从父元素移动到子元素上（看起来似乎违反直觉）。如果需要知道移动到哪个元素，可以使用 relatedTarget 属性。

代码清单 B-18 展示了的例子，它绑定的一对事件允许用户可以使用键盘操作（和鼠标操作）。当用户移动鼠标到元素上，或者使用键盘跳到元素上，链接会出现额外的颜色高亮。

代码清单B-18 使用mouseover和mouseout事件创建交替效果

```
// 查找所有的<a>元素，并赋予事件处理函数
var a = document.getElementsByTagName('a');
for ( var i = 0; i < a.length; i++ ) {

    // 绑定mouseover和focus事件出来起到<a>元素上，
    // 当用户移动鼠标进入或者聚焦（使用键盘）到<a>元素上时，
    // 改变它的背景为蓝色
```

```
        a[i].onmouseover = a[i].onfocus = function() {
            this.style.backgroundColor = 'blue';
        };

        // 绑定mouseout和blur事件出来起到<a>元素上，
        // 当用户移动鼠标离开或者失焦（使用键盘）于<a>元素上时，
        // 改变它的颜色回默认的白色
        a[i].onmouseout = a[i].onblur = function() {
          this.style.backgroundColor = 'white';
      };

  }
```

340

B.7　键盘事件

键盘事件处理所有用户在键盘敲击的情况，不管在文本输入区域内部还是外部。

B.7.1　keydown/keypress

keydown 事件是键盘敲击时触发的第一个键盘事件。如果用户继续按住键位，keydown 事件会持续进行。keypress 是 keydown 事件的同义事件，它们的表现完全一致，只有一个例外。如果需要阻止按键的默认行为，你必须使用 keypress 事件。代码清单 B-19 展示了一个使用 keypress 处理函数来阻止特定的按键在<input>元素内输入的例子。

代码清单B-19　防止在<input>元素内漫不经心的回车导致的表单提交

```
// 查找文档内的所有<input>元素
var input = document.getElementsByTagName('input');
for ( var i = 0; i < input.length; i++ ) {

    // 绑定keypress事件处理函数到<input>元素
    input[i].onkeypress = function(e) {

        // 如果敲击了回车，阻止它的默认行为
        return e.keyCode != 13;
    };

}
```

B.7.2　keyup

keyup 是最后一个发生的键盘事件（在 keydown 事件之后）。不像 keydown 事件，该事件在松开键盘时仅触发一次（因为松开键盘并不是一个持续的状态）。

341

B.8　表单事件

表单事件主要处理<form>、<input>、<select>、<button>和<textarea>元素，它们都是 HTML 表单的主要组成成分。

B.8.1 select

select 事件在用户使用鼠标于输入框内选择不同区块的文本时触发。在该事件内，你可以重定义用户与表单的交互。代码清单 B-20 展示了一个使用 select 事件来防止选择表单域内的文本的例子。

代码清单B-20 防止用户在<textarea>内选择文本

```
// 定位到文档的第一个<textarea>
var textarea = document.getElementsByTagName('textarea')[0];

// 绑定select事件监听函数
textarea.onselect = function(){
    // 当产生了新的选择，防止该行为
    return false;
};
```

B.8.2 change

change 事件在用户改变了输入元素（包括<select>和<textarea>元素）的值时触发。该事件仅在用户已经离开了元素，使其失去焦点时触发。

代码清单 B-21 的代码能够监听一个 ID 为 entryArea（是一个<textarea>）的元素的改变并实时更新相关的内容预览。

代码清单B-21 监听change事件并更新相应的元素

```
// 监听'entryArea'（它是一个<textarea>）的任何改变
document.getElementById('entryArea').onchange = function(){

    // 当该区域改变了，更新实时预览
    document.getElementById('preview').innerHTML = this.value;

};
```

B.8.3 submit

submit 事件仅在表单内并且仅当用户点击了提交按钮（在表单内）或者在其中一个输入元素内敲击了回车键时触发。通过为表单的提交按钮绑定 submit 处理函数而非点击处理函数，你才可以确保捕获用户提交的表单的意图。

342

代码清单 B-22 展示的代码监听页面的第一个表单，中断它的提交动作，并为用户显示一条信息，而不是把结果发送到服务器。

代码清单B-22 使用事件执行一个另类行为

```
// 绑定submit处理函数到文档的第一个表单
document.getElementsByTagName('form')[0].onsubmit = function(e) {

    // 获取用户键入的name
    var name = document.getElementById('name').value;
```

```
        // 设置包含Hello Name!(Name就是用户所键入的name)的<h1>元素
        document.getElementsByTagName('h1')[0].innerHTML =
          'Hello ' + name + '!';

        // 确保表单不提交到服务器
        return false;

};
```

B.8.4　reset

reset 事件仅在用户点击表单内的重置按钮（跟提交按钮相反，但它不能由回车键触发）。代码清单 B-23 展示了的例子，它以监听表单的 reset 事件来提供一个另类的行为。

代码清单B-23　处理重置表单的一种快速但粗糙的方式

```
// 查找页面的第一个表单
var form = document.getElementsByTagName('form')[0];

// 监听重置按钮是否被点击
form.onreset = function(){

    // 查找表单内的所有<input>元素
    var input = form.getElementsByTagName('input');

    // 并重置它们的值为空字符串
    for ( var i = 0; i < input.length; i++ )
        input[i].value = '';

};
```

343

附录 C

浏 览 器

JavaScript 编程总是受 Web 浏览器的发展而驱动。因为非浏览器端的 JavaScript 环境（比如服务器端的 JavaScript）还是实验性质的，未来的 JavaScript 仍然是以浏览器为中心的。所以，JavaScript 的特点跟浏览器的发展密切相关，它的哪些特点最重要也需要从浏览器的角度分析得出。

互联网所有可用的浏览器中，只有少部分能够追随最新技术的发展，这还不够，有些浏览器甚至筑起障碍。最后，你决定要支持哪些浏览器，可能还取决于你的受众，取决于创建人人都能正确使用与否的应用程序所需的预算时间。

开发成功的 JavaScript 应用程序，不管怎么样，现在的情况比过去要好得多。随着多种实践的标准化（DOM、`XMLHttpRequest` 等），支持所有的现代浏览器将不需要各种各样不同的特别处理。但是，这一天还很遥远，今天依旧是今天。所以，以下是流行的现代浏览器和它们的支持范围的简要说明。

C.1 IE

由微软开发并绑定到它的操作系统上（从 Windows 95 开始），IE 到目前为止是最为流行的浏览器。该浏览器的开发已经停滞多年，因为微软在垄断的浏览器市场里自得其乐。最近，浏览器小组重启动，因为 Windows Vista——微软的新操作系统（包括 IE 7）的发布。

1. 版本5.5和6.0

版本 5.5 是 Windows 98 中 IE 的更新版本，而版本 6 是 Windows XP 的默认浏览器。它们的 CSS 和 DOM 实现都有一些 bug 和脆弱性，但它们的功能还算强大，因此任何 Web 应用程序均应支持它们。

2. 版本7

最新版本的 IE，它可用在 Windows XP 和 Windows Vista 上。它的使用率依然有限，除非 Vista 的真正发布和使用率的上升。

在 IE 7 中，JavaScript 引擎虽只稍作改动，但大量 CSS 相关的 bug 都得以修正。需要注意的是，它摒弃了需要 `ActiveXObject` 的 `XMLHttpRequest`，你可直接使用 `XMLHttpRequest`（默认的）。

C.2　Mozilla

从 NetScape 的灰烬中涅槃，Mozilla 代表了开源的努力，开发了这个受欢迎的浏览器，随着 Firefox 的发布和使用率的上升，Mozilla 已经坐在了现代浏览器开发的头排座椅。

1．Firefox 1.0、NetScape 8和Mozilla 1.7

这 3 个浏览器均基于 1.7 版本的 Gecko 渲染引擎。该引擎能够兼容所有现代的脚本技术。它们是可靠的、稳定的、合用的浏览器。

2．Firefox 1.5和2.0

Gecko 渲染引擎的最新版本（1.8），在最新版的 Firefox 浏览器中得到使用，它支持大量的先进技术，包含 SVG 1.1 的部分支持、<cancas>元素的支持和 JavaScript 1.6（在第 14 章中有讨论它的不同特点）的支持。

C.3　Safari

Safari 诞生于 Apple 公司为 Mac OS X 制作一个更好浏览器（相比差劲的 IE5）的首次尝试。开始时（版本 1.0），它的渲染引擎过于粗糙，开发者并不专注于完全支持它。但是，随着新版本的发布，它的引擎也稳定提升。

Safari 最常用的版本（OS X 10.3 的 1.3，OS X 10.4 的 2.0）包含了部分重大 bug 修正和新特点。在 Safari 中载入 JavaScript 应用程序并运行良好的情况非常普遍。

此外，Safari 引入新元素<canvas>的支持，它可以让开发者在 Web 页面中进行实际的绘图——动态应用程序中一个非常重要的特色。

C.4　Opera

作为微软和网景浏览器大战中的老兵，随着这个挪威浏览器的完全免费（之前是附加广告或者需要授权），Opera 的流行日益高涨。与 Mozilla 并肩作战，Opera 已经活跃在开发和规划新的
346 HTML 5 规范并实现了部分的规范。

1．版本8.5

Opera 8.5 是第一个完全免费的版本，它比其他版本用得更广泛。它有着稳固的现代特性，尽管开发者偶尔会发现其 CSS 实现跟其他浏览器有所不同，且其 JavaScript 实现也很丰富。

2．版本9.0

347 这是 Opera 最近一次更新，修复了 JavaScript 引擎的一些 bug，同时支持新的<canvas>元素。