第一部分：现代 JavaScript 简介

第一章 现代 JavaScript 编程

JavaScript 的演化是渐进而稳固的。历经过去十年的进程，JavaScript 在人们的认知里已经从一门简单的玩物式的语言逐渐发展成为一门倍受推崇的编程语言，被全世界的公司和开发者用来构造种种精彩的应用。现代 JavaScript 编程语言一如既往地可靠、健壮，强大得令人难以置信。在本书中我进行的许多论述，将揭示是什么使得现代 JavaScript 应用程序与从前有着那么明显的不同。本章中出现的许多概念远不能算新奇，然而成千上万聪明的程序员的认同促使它们的用途得以升华并最终形成今天的格局。既如此，我们这就来着眼现代 JavaScript 程序设计。

面向对象的 JavaScript

从语言的视角来看，面向对象的程序设计和面向对象的 JavaScript 语言绝对不是什么摩登的东西；JavaScript 最开始就是被设计成一种彻底的面向对象语言。然而，随着 JavaScript 在其使用和接受的过程中的"逐步发展"，其它语言(如 Ruby,Python,和 Perl 等)的程序员留意到了它并开始将他们的编程模式引入了 JavaScript。

面向对象的 JavaScript 代码的外观和内部运作都有别于其它具有对象能力的语言。在第二章我将深入论述使它如此独特的方方面面，而在这里，先来看一点基础的东西以体会编写现代 JavaScript 代码的初步感觉。程序 1-1 中的两个对象构造器的例子，演示了可用于学校课程的简单的对象搭配。

程序 1-1. 课程和课程表的面向对象 JavaScript 表述

CODE:

```
//类 Lecture 的构造器
//使用两个字符串参数，name 和 teacher
function Lecture( name, teacher ) {
    //把它们作为对象的本地属性保存
    this.name = name;
    this.teacher = teacher;
}
//类 Lecture 的方法，生成一个显示该课程信息的字符串
Lecture.prototype.display = function(){
    return this.teacher + " is teaching " + this.name;
};
//类 Schedule 的构造器
//使用一个 lectures 类型的数组作为参数
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
function Schedule( lectures ) {
    this.lectures = lectures;
}

//类 Schedule 的方法，用来构造一个描述该课程表的字符串
Schedule.prototype.display = function(){
    var str = "";
    //遍历每门课程，累加构成信息字符串
    for ( var i = 0; i < this.lectures.length; i++ )
        str += this.lectures[i].display() + " ";
    return str;
};
```

从程序 1-1 的代码中你或许已经看出，大部分的面向对象基本原则贯穿存在于其中，但它们是以不同于其它更常见的面向对象语言的方式组织起来的。你可以创建对象构造器和方法，并存取对象属性。程序 1-2 展示了在应用程序中使用上面两个类的一个示例。

程序 1-2. 给用户提供课程的列表

CODE:

```
//创建一个新的课表对象，存于变量 mySchedule 中
var mySchedule = new Schedule([
    //创建一个课程对象的数组，
    //作为传给课表(原文此处为 Lecture，疑为笔误)对象的唯一参数
    new Lecture( "Gym", "Mr. Smith" ),
    new Lecture( "Math", "Mrs. Jones" ),
    new Lecture( "English", "TBD" )
]);
// 弹出对话框显示课表的信息
alert( mySchedule.display() );
```

伴随对广大程序员对 JavaScript 的接受，设计良好的面向对象代码的使用也正日益普及。贯穿本书的始末，我将试图展示我认为能够最好地例示代码设计与实现的不同的面向对象的 JavaScript 代码片段。

测试你的代码

建立起良好的面向对象的基本代码之后，开发专业品质的 JavaScript 代码的第二个方面是确保拥有一个强劲的代码测试环境。当开发频繁使用的或将由其它开发者维护的代码时，严格调试的必要性会显得尤为突出。为其它开发者提供一个坚实的测试基础，是维持代码开发活动的关键。

在第四章，你将会看到一些可用来形成良好的测试/使用框架的不同工具，以及对复杂应用程序的简单调试。用于 Firefox 的插件 Firebug 就是其中一例。Firebug 提供了许多的有

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

用的工具，如错误控制台，HTTP 请求日志，调试，以及元素查看。图 1-1 展示了 Firebug 调试一段代码时的实况截屏。

图 1-1. Firefox Firebug 插件运行时的截屏

开发干净的、可测试的代码的重要性怎么强调都不会过分。一旦你开始开发一些干净的面向对象代码并将它们与合适的测试套件结合，相信你会倾向于同意这一点。

为分发而进行的封装

开发现代的专业 JavaScript 代码的最后一个方面是为了代码分发或在现实世界里中使用而进行的封装处理。随着开始开发者们在其页面中使用越来越多的 JavaScript 代码，冲突的可能性将会增加。如果两个 JavaScript 库里都有一个名为 data 的变量或者按各的意图添加事件，灾难性的冲突和莫名其妙的错误可能就会出现。

开发者简单地置入<script>指针无须任何变动就能正常工作的能力是开发一个成功的 JavaScript 库的精诣所在。开发者用以保持代码清洁和普遍兼容的的技术或解决方案有许多种。

使用命名空间是广泛使用的保证代码不与其它 JavaScript 代码互相影响和抵触的一种技术。这方面一个极端的（但未必是最好或最有用的）运作中的例子就是 Yahoo 开发的任何人都可使用的用户界面库。使用该库的一个示例见程序 1-3.

程序 1-3. 使用重度名称空间化的 YahooUI 库给一个元素添加事件

CODE:

```
//给 ID 为"body"的元素添加 mouseover 事件监听器
YAHOO.util.Event.addListener('body','mouseover',function(){
    //and change the background color of the element to red
    this.style.backgroundColor = 'red';
});
```

然而，这种命名空间方法存在一个问题，即库与库之间在构造和使用的方式上缺乏内在

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

的一致性。正是在这一点上，中心代码仓库如 JSAN(JavaScript Archive Network)变得非常有用。JSAN 提供一套代码库需遵从的一致规则，以及一种快捷导入代码所依赖的其它库的方式。图 1-2 展示了 JSAN 的主分发中心的一个截屏。

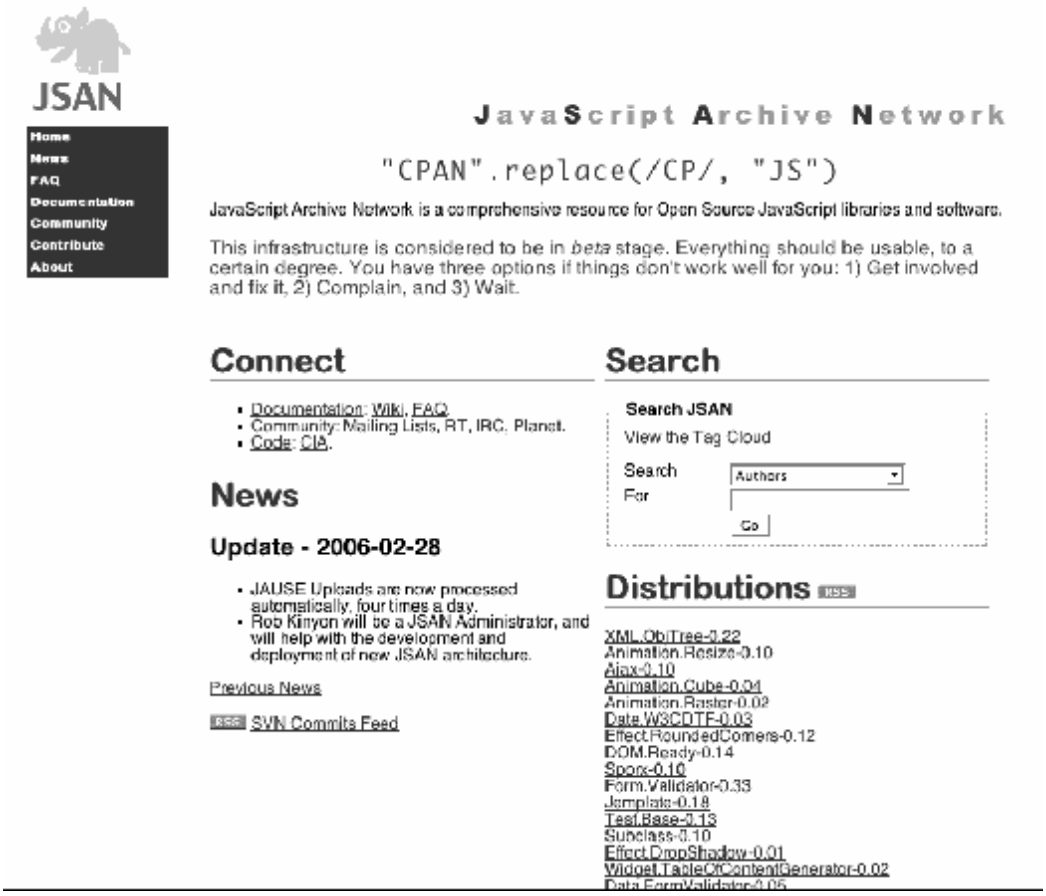


图 1-2. 公共代码仓库 JSAN 的截屏

我将在第三章阐述开发清洁的可封装代码的细节。此外，其它常见的事故多发点如事件处理冲突，将在第六章论述。

Unobtrusive DOM 脚本编程（非侵入的 DOM 脚本编程）

基于一个优良的可测试的核心创建你的代码和兼容的分发，是 Unobtrusive DOM 脚本编程的基本概念。编写 unobtrusive 代码意味着与你的 HTML 内容的彻底分离：数据来自服务器，而 JavaScript 代码用来使其动态化。达到这一彻底分离的最重要的副作用就是你的代码在不同的浏览器之间可以完美的升/降级。利用这一点，你可以提供高级的内容给支持它的浏览器，而在不支持的浏览器上从容隐藏之。

编写现代的、Unobtrusive 代码包括两个方面：文档对象模型(DOM)和 JavaScript 事件。本书中我对这两个方面都将作深入的解释。

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

文档对象模型

DOM 是表示 XML 文档的流行的方法。它未必是最快的、最轻便的、或者最易使用的，却是是最普及的，绝大多数 web 开发语言（如 Java，Perl，PHP，Ruby，Python，及 Javascript)都实现了对它的支持。DOM 旨在为开发者提供一种直观的方式来导航于 XML 的层次结构中。

因为有效的 HTML 只是 XML 的一个子集，保有一个方式来有效地解析和浏览 DOM 文档对于简化 JavaScript 开发来说是必不可少的。从根本上讲，出现在 JavaScript 中的大多数的交互是发生在 JavaScript 与页面所包含的不同 HTML 元素之间的；DOM 是使这此过程简单化的卓越工具。程序 1-4 展示了使用 DOM 在页内导航和查找不同的元素然后操作它们的一些例子。

程序 1-4. 使用文档对象模型定位并操纵不同的 DOM 元素

[Copy to clipboard] [-]

CODE:

```
<html>
<head>
<title>Introduction to the DOM</title>
<script>
//直到文档完全载入，我们才能操作 DOM
window.onload = function() {
  //找到文档中所有的<li>元素
  var li = document.getElementsByTagName("li");
  //然后给它们全部加上边框
  for ( var j = 0; j< li.length; j++ ) {
    li[j].style.border = "1px solid #000";
  }
  //定位 ID 为'everywhere'的元素
  var every = document.getElementById( "everywhere" );
  //并将它从文档中移除
  every.parentNode.removeChild( every );
};
</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,
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
      really quickly.</li>
  </ul>
</body>
</html>
```

DOM 是开发 Unobtrusive JavaScript 代码的第一步。借助简单快速导航 HTML 文档的能力，所有随之而来的 JavaScript/HTML 交互将变得如此简单。

事件

事件将一个应用程序之内所有的用户交互结合在一起。在一个设计良好的 JavaScript 应用程序里，你将拥有数据源和它的视觉的表示(在 HTML DOM 内部)。为了同步这两个方面，你必须监视用户的交互动作并试图相应地更新用户界面。使用 DOM 和 JavaScript 事件的结合是使得现代 web 应用程序赖以工作的基本组合。

所有的现代浏览器都提供一系列的只要特定交互动作发生即被触发的事件，如用户移动鼠标，敲击键盘，或离开页面等等。使用这些事件，你可以注册代码到特定事件，一旦该事件发生，你的代码就会被执行。程序 1-5 展示了这种交互的一个实例，该网页中的<li>元素在用户鼠标经过的时候会改变背景色。

程序 1-5. 使用 DOM 和事件来提供一些视觉效果

[Copy to clipboard] [-]

CODE:

```
<html>
<head>
<title>Introduction to the DOM</title>
<script>
//直到文档完全载入，我们才能操作 DOM
window.onload = function(){
    //查找所有的<li>元素，附以事件处理程序
    var li = document.getElementsByTagName("li");
    for ( var i = 0; i < li.length; i++ ) {
        //将鼠标移入事件处理程序附在<li>元素上，
        //该程序改变<li>背景颜色为蓝色
        li[i].onmouseover = function() {
            this.style.backgroundColor = 'blue';
        };
        //将鼠标移出事件处理程序附在<li>元素上，
        //该程序将<li>的背景颜色改回白色
        li[i].onmouseout = function() {
            this.style.backgroundColor = 'white';
        };
    }
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
};
</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 事件是复杂多样的。本书中的大多数代码或应用程序都以某种方式利用了事件。第六章和附属 B 完全专注于事件及其交互。

JavaScript 与 CSS

动态 HTML 建立在 DOM 和事件交互的基础上。在核心层面上，动态 HTML 表示发生在 JavaScript 和附着在 DOM 元素上的 CSS 信息的交互。

层叠式样式表(CSS)作为布局的标准服务于简单的不唐突的网页，在最小化了用户端兼容性问题的同时，提供给开发者以强大的可控制性。从根本上讲，动态 HTML 就是探索 JavaScript 和 CSS 彼此交互作用时能够达到什么以及怎样最好地利用该联合达成令人印象深刻的效果。

更高级的交互示例如拖放元素和动画效果见第七章。在那里我将围绕它们展开深入论述。

Ajax

Ajax，或曰异步 Javascript 与 XML，是由 Adaptive Path 公司的创办人之一兼董事长 Jesse James Garrett 在其论文 "Ajax:Web 应用程序的新途径 "(**http://www.adaptivepath.com/publ ... archives/000385.php**)中创造的一个术语。它描述了请求和提交额外的信息时发生于客户和服务器之间的高级交互。

术语 Ajax 包括了许多种数据据通讯的可能组合，但它们都围绕一个中心前提：附加的数据请求是在页面完全载入之后由客户端向服务器发起的。这允许应用程序开发者超越缓慢的、传统的应用程序流程，创建与用户相关的额外交互。图 1-3 是来自 Garrett 的 Ajax 论文的一个图示，说明了应用程序中由于额外的请求发生在后台(而且用户很可能并不知情)，交互的流程发生了怎样的改变。

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

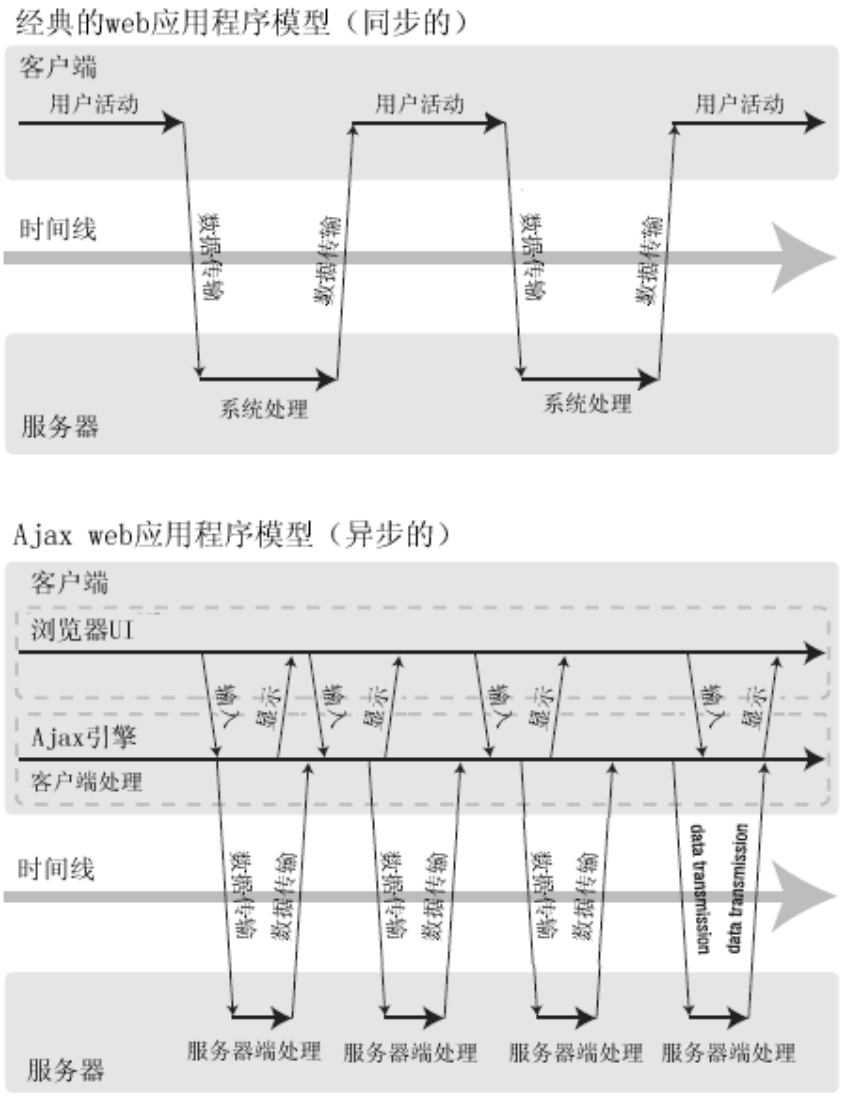


图 1-3：来自文章"Ajax:Web 应用程序的新途径"的图示，展示发生于客户和服务器之间的先进的异步的交互

Garrett 论文的最初发表，激起了用户、开发者、设计者以及管理者们的兴趣，使用高级交互的新型应用程序爆炸式地增长。讽刺的是，在这一兴趣复苏的同时，Ajax 背后的技术却是相当陈旧的(在 2000 年左右就有已有了商业上的应用)。然而最主要的不同在于，老的应用程序利用了浏览器特有的方式与服务器通讯（如仅 ie 具有的功能)。由于所有的现代浏览器都支持 XMLHttpRequest(发送或从服务器接收数据的主要手段)，the playing field has been leveled(见#12 cfs178 的回复)，每个人都可以享受其益处。

如果说有一个公司走在了利用 Ajax 技术创建优秀应用程序的最前列，那无疑是 Google。恰在最初的 Ajax 论文出现之前，它发布了一个高交互性的 domo:Google Suggest。该 demo 可以实时地自动完成你所键入的查询，这是旧式的页面重载不可能达到的功能。图 1-4 是 Google Suggest 运行时的一个截屏。

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

图 1-4. Google Suggest，早于 Garrett 的 Ajax 论文的利用了异步 XML 技术的应用程序

除此而外，Google 的另一个革命性的应用程序为 Google Map，用户在其地图上移动将会实时地看到相关的局部结果。这一程序通过使用 Ajax 技术而提供的速度和可用性的水平是其它任何可用的地图程序所无法比拟的，结果彻底地变革了在线地图市场。图 1-5 是 Google Map 的截屏.

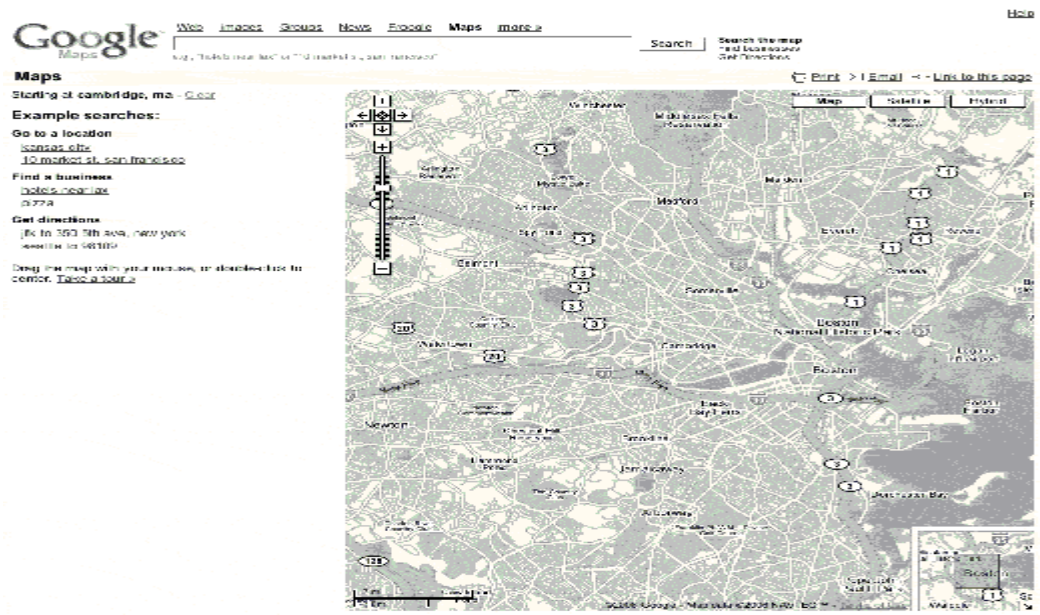


图 1-5. Google Maps，利用了一些 Ajax 技术来动态载入特定区域信息

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

浏览器支持

JavaScript 开发所面临的不幸事实是，因为与实现和支持它的浏览器关联过于紧密，它受到当前最流行浏览器的支配。由于用户未必会使用对 JavaScript 支持得最好的浏览器，我们被迫对哪些是最重要的功能作出抉择。

许多开发者已经开始做的是去掉对那些导致过多开发障碍的浏览器的支持。因为它们用户群的规模和还是因为它们拥有你喜欢的功能而考虑对浏览器的支持，需要仔细的权衡。

最近 Yahoo 发布了一个可用来扩展你的 web 应用程序的 JavaScript 库。连同那个库，它还发布了一些供 web 开发者遵从的指导方针。以我之见，从中产生的最重要的文档是 Yahoo 支持与不支持的各种浏览器的官方的列表。尽管任何人或任何公司都可以做类似的事情，一份由互联上访问最频繁的网站所提供的文档，其价值绝对无法估量。

Yahoo 开发了一个分级的浏览器支持策略，对浏览器指定特定的级别并依据其功能为它提供不同的内容。Yahoo 给浏览器三个级别：A,X 和 C:

A 级浏览器是得到完全支持和测试的，Yahoo 的所有程序都能有保障在其中运行。

X 级浏览器是 Yahoo 认可但是没能彻底测试的准 A 级浏览器，或者是崭新的从未遇到过的浏览器。X 级浏览器被期望能处理高级的内容，与 A 级浏览器等同对待。

C 级浏览器是已知的"劣质的"浏览器，不支持运行 Yahoo 应用程序所必须的功能。由于 Yahoo 应用程序完全 Unobtrusive(即使没有 JavaScript 它们也能继续工作)，这些浏览器只需处理不含 JavaScript 的功能性的内容。

顺便一提，Yahoo 的浏览器级别选择竟与我的不谋而合，这使得它尤其富有吸引力。在这本书里，我大量地使用了术语"现代浏览器"，当我用这一措词的时候，我指的 Yahoo 浏览器分级表评定为 A 级的任浏览器。给定一组赖以工作的一致的功能，减少因避免浏览器的不兼容而带来的痛苦，学习和开发的经历将会变得更加有趣。

我极力推荐你们去通读浏览器分级的支持文档(见 **http://developer.yahoo.com/yui/articles/gbs/gbs.html**，该文档包含了图 1-6 所示的浏览器支持表)，感受一下 Yahoo 力图实现的是什么。通过将些信息公之于 web 开发的大众，Yahoo 正在给出一条无价的"黄金标准"让其它所有人去靠拢，这是很了不起的。

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

	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 提供的浏览器分组支持表

更多的关于浏览器支持情况的信息请参看本书的附录 C，那里对每种浏览器的缺点和长处都有深入的论讨。或多或少地，你会发现所有的 A 级浏览器都走在发展的最前列，提供了远远超出你的开发所需要的功能。

当选择你希望支持哪些浏览器时，最终结果实际上可以归结为你的应用程序支持的一组功能。如果你想要支持 NS4 或 IE5（举个例子来说），则势必严格地限制你可以在程序中使用的功能的数量，因为它们缺乏对现代编程技术的支持。

尽管如此，了解哪些浏览器是现代的允许你利用其中可用的强大的功能并给你一个可供将来的开发所依据的稳固的基础。这一稳固的开发基础可由以下几种功能所定义：

核心 Javascript 1.5: 最近、最广泛接受的 JavaScript 版本。它全面支持面向对象的 JavaScript。IE5.0 不支持全部的 1.5，这是开发者们不愿意支持它的主要原因。

XML 文档对象模型（DOM）2:用来访问 HTML 和 XML 文档的的标准方案。这绝对是编写高效率的程序不可或缺的。

XMLHttpRequest: Ajax 技术的支柱——用来发起 HTTP 请求的一个简单层。所有的浏览器默认都支持这一对象，除了 IE5.0-6.0；而它们也都支持用 ActiveX 初始化功能相当的对象。

CSS: 网页设计的基本需求。这似乎像是一个额外的需求，但是拥有 CSS 对 web 应用程序开发者来说是必不可少的。由于每一种现代浏览器都支持 CSS，大多数问题的发生通常归结为呈现方面的差异。这正是 IE for Mac 较少被频繁支持的主要原因。

以上这些浏览器功能的结合构成了开发 JavaScript web 应用程序的支柱。所有的现代浏览器都以某种方式支持以上列举的功能。本书论述的所有内容都基于这一假设：你所使用的浏览器最起码能支持它们

本章摘要

本书试图完全包括所有现代、专业的 JavaScript 编程技术，以期它们被从独立开发者到大型公司的每一个人使用，使得其代码更加可用、可读、具有交互性。

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

在这一章里我们对这本书里将会出现的每一个知识点做了一个简短的总览。这包括专业 JavaScript 编程的基础：编写面象对代码，测试代码，为分发而进行封装。随后你看到了 Unobtrusive DOM 脚本编程的主要方面，包括一个关于文档对象模型，事件，JavaScript 与 CSS 交互的简短的总览。最后你看到了 Ajax 背后的前提和在现代浏览器中 JavaScript 的支持。这些话题加在一起，足够带你步入专业级 JavaScript 程序员的行列。

第二部分　专业的 JavaScript 开发

第二章 面对对象的 JavaScript

对象是 JavaScript 的基本单位。实际上 JavaScript 中一切都是对象并得益于这一事实。然而，为了增强这一纯粹的面向对象的语言，JavaScript 包括了一个庞大的功能集，使它无论是在潜在能力还是风格上，都成为一门极其独特的语言。

本章中我将开始覆盖 JavaScript 语言的最重要的一些方面，如引用，作用域，闭包，以及上下文，你会发现这正是其它 JavaScript 书籍中很少论及的。打下主要的基础以后，我们将开始探索面向对象 JavaScript 的几个重点，包括对象到底如何运作和怎样创建新的对象并在特定的许可条件下设置其方法。如果你认真去读的话，这很可能是本书中最重要的一章，它将彻底地改变你看待 JavaScript 作为一门编程语言的方式。

语言特性

引用

JavaScript 的一个重要的方面是引用的概念。引用就是指向对象实际位置的指针。这是一项极其强大的功能。前提是，实际的对象决不是一个引用：字符串总是一个字符串，数组总是一个数组。然而，多个变量可以引用相同的对象。JavaScript 就是以这种引用引用机制为基础。通过维护一系列的指向其它对象的引用，语言为你提供了更大的弹性。

另外，对象能包括一系列的属性，这些属性简单地引用其它对象（如字符串，数字，数组等等）。当几个变量指向相同对象时，修改底层对象类型将会在所有的指点向它的变量上有所反映。例 2-1 即此一例，两个变量指向同一个对象，但是对对象内容的修改的反映是全局的。

程序 2-1. 多变量引用单个对象的示例

[Copy to clipboard] [-]

CODE:

```
//设置 obj 为一个空对象
var obj = new Object();
//objRef 现在引用了别的对象
var objRef = obj;
//修改原始对象的属性
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
obj.oneProperty = true;
//我们可以发现该变化在两个变量中都可以看到
//(因为他们引用了同一个对象)
alert( obj.oneProperty === objRef.oneProperty );
```

我从前提到过自更改的对象在 JavaScript 里非常少见的。让我们看一个发生这一状况的实例。数组对象能够用 push 方法给它自己增加额外的项。因为在数组对象的核心，值是作为对象的属性存储的，结果类似程序 2-1 中的情形，一个对象成为全局被改动的(导致了多个变量的值被同时改变)。见程序 2-2.

程序 2-2. 自修改对象的例子

[Copy to clipboard] [-]

CODE:

```
//创建一组项目的数组
var items = new Array( "one", "two", "three" );
//创建一个对项目数组的引用
var itemsRef = items;
//给原始数组添加一项
items.push( "four" );
//两个数组的长度应该相同，
//因为它们都指向相同的数组对象
alert( items.length == itemsRef.length );
```

记住这一点是很重要的：引用总是只指向最终被引用的对象，而不会是引用本身。例如，在 Perl 语言里，很可能有一个引用指向另一个也是引用的变量。但在 JavaScript 里，它会沿着引用链向下追溯直到指向核心的对象。程序 2-3 演示了这种情形，物理的目标已经改变而引用仍然指向原来的对象。

程序 2-3. Changing the Reference of an Object While Maintaining Integrity(见#9 oerrite 的回复)

[Copy to clipboard] [-]

CODE:

```
// 设置 items 为一个字符串的数组(对象)
var items = new Array( "one", "two", "three" );
// 设置 itemsRef 为对 items 的引用
var itemsRef = items;
//让 items 指向一个新的对象
items = new Array( "new", "array" );
// items 和 itemsRef 现在指向不同的对象
// items 指向 new Array( "new", "array" )
// itemsRef 则指向 new Array( "one", "two", "three" )
alert( items !== itemsRef );
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

最后，让我们来看一个陌生的例子，表面似乎是一个自修改的对象，却作用于一个新的未被引用的对象。当执行字符串串联时，结果总是一个新的字符串对象，而非原字符串更改后的版本。这在程序 2-4 中可以看出。

程序 2-4. 对象修改作用于一个新的对象而非自修改对象的示例

[Copy to clipboard] [-]

CODE:

```
//让 item 等于一个新的字符串对象
var item = "test";
//itemRef 也引用相同的字符串对象
var itemRef = item;
//在字符串对象上串联一个新的对象
//注意：这创建了一个新的对象，并不修改初始对象
item += "ing";
//item 和 itemRef 的值并不相等，因为
//一个全新的对象被创建了
alert( item != itemRef );
```

如果你刚刚接触，引用可能是个令人头大的刁钻话题。然而，理解引用是如何工作的对于编写良好、干净的 JavaScript 代码是极其重要的。接下来的几节我们将探究几种未必新鲜和令人激动的，但是同样对编写良好、干净的代码很重要的特性。

函数重载和类型检查

其它面向对象的语言(比如 Java)的一种共有的特性是"重载"函数的能力：传给它们不同数目或类型的参数，函数将执行不同操作。虽然这种能力在 JavaScript 中不是直接可用的，一些工具的提供使得这种探求完全成为可能。

在 JavaScript 的每一个函数里存在一个上下文相关的名为 arguments 的变量，它的行为类似于一个伪数组，包含了传给函数的所有参数。参数不是一真正的数组(意味着你不能修改它，或者调用 push()方法增加新的项)，但是你可以以数组的形式访问它，而且它也的确有一个 length 属性。程序 2-5 中有两个示例。

程序 2-5. JavaScript 中函数重载的两个示例

[Copy to clipboard] [-]

CODE:

```
//一个简单的用来发送消息的函数
function sendMessage( msg, obj ) {
    //如果同时提供了一个消息和一个对象
    if ( arguments.length == 2 )
        //就将消息发给该对象
        obj.handleMsg( msg );
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
    //否则，刚假定只有消息被提供
    else
        //于是显示该消息
        alert( msg );
}
//调用函数，带一个参数 – 用警告框显示消息
sendMessage( "Hello, World!" );
//或者，我们也可以传入我们自己的对象用
//一种不同方式来显示信息
sendMessage( "How are you?", {
    handleMsg: function( msg ) {
        alert( "This is a custom message: " + msg );
    }
});
//一个使用任意数目参数创建一个数组的函数
function makeArray() {
    //临时数组
    var arr = [];
    //遍历提交的每一个参数
    for ( var i = 0; i < arguments.length; i++ ) {
        arr.push( arguments[i] );
    }
    //返回结果数组
    return arr;
}
```

另外，存在另一种断定传递给一个函数的参数数目的方法。这种特殊的方法多用了一点点技巧：我们利用了传递过来的任何参数值不可能为 undefined 这一事实。程序 2-6 展示一了个简单的函数用来显示一条错误消息，如果没有传给它，则提供一条缺省消息。

程序 2-6: 显示错误消息和缺省消息

[Copy to clipboard] [-]

CODE:

```
function displayError( msg ) {
    //检查确保 msg 不是 undefined
    if ( typeof msg == 'undefined' ) {
        //如果是，则设置缺省消息
        msg = "An error occurred.";
    }
    //显示消息
    alert( msg );
}
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

typeof 语句的使用引入了类型检查。因为 JavaScript（目前）是一种动态类型语言，使得这个话题格外有用而重要的话题。有许多种方法检查变量的类型；我们将探究两种特别有用的。

第一种检查对象类型的方式是使用显式的 typeof 操作符。这种有用的方法给我们一个字符串名称，代表变量内容的类型。这将是一种完美的方案，除非变量的类型或者数组或自定义的对象如 user（这时它总返回"ojbect"，导致各种对象难以区分）。

这种方法的示例见程序 2-7

程序 2-7. 使用 typeof 决定对象类型的示例

[Copy to clipboard] [-]

CODE:

```
//检查我们的数字是否其实是一个字符串
if ( typeof num == "string" )
    //如果是，则将它解析成数字
    num = parseInt( num );
//检查我们的数组是否其实是一个字符串
if ( typeof arr == "string" )
    //如果是，则用逗号分割该字符串，构造出一个数组
    arr = arr.split(",");
```

检查对象类型的第二种方式是参考所有 JavaScript 对象所共有的一个称为 constructor 的属性。该属性是对一个最初用来构造此对象的函数的引用。该方法的示例见程序 2-8。

程序 2-8. 使用 constructor 属性决定对象类型的示例

[Copy to clipboard] [-]

CODE:

```
//检查我们的数字是否其实是一个字符串
if ( num.constructor == String )
    //如果是，则将它解析成数字
    num = parseInt( num );
//检查我们的字符串是否其实是一个数组
if ( str.constructor == Array )
    //如果是，则用逗号连接该数组，得到一个字符串
    str = str.join(',');
```

表 2-1 显示了对不同类型对象分别使用我所介绍的两种方法进行类型检查的结果。表格的第一列显示了我们试图找到其类型的对象。每二列是运行 typeof Variable(Variable 为第一列所示的值)。此列中的所有结果都是字符串。最后，第三列显示了对第一列包含的对象运行 Variable.constructor 所得的结果。些列中的所有结果都是对象。

表 2-1. 变量类型检查

Variable	typeof Variable	Variable.constructor

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

{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

使用表 2-1 的信息你现在可以创建一个通用的函数用来在函数内进行类型检查。可能到现在已经明显，使用一个变量的 constructor 作为对象类型的引用可能是最简单的类型检查方式。当你想要确定精确吻合的参数数目的类型传进了你的函数时，严格的类型检查在这种可能会大有帮助。在程序 2-9 中我们可以看到实际中的一例。

程序 2-9. 一个可用来严格维护全部传入函数的参数的函数

[Copy to clipboard] [-]

CODE:

```
//依据参数列表来严格地检查一个变量列表的类型
function strict( types, args ) {
    //确保参数的数目和类型核匹配
    if ( types.length != args.length ) {
        //如果长度不匹配，则抛出异常
        throw "Invalid number of arguments. Expected " + types.length +
            ", received " + args.length + " instead.";
        }
    //遍历每一个参数，检查基类型
    for ( var i = 0; i < args.length; i++ ) {
        //如 JavaScript 某一项类型不匹配，则抛出异常
        if ( args[i].constructor != types[i] ) {
            throw "Invalid argument type. Expected " +
                types[i].name +", received " +
                args[i].constructor.name + " instead.";
        }
    }
}
//用来打印出用户列表的一个简单函数
function userList( prefix, num, users ) {
    //确保 prefix 是一个字符串，num 是一个数字，
    //且 user 是一个数组
    strict( [ String, Number, Array ], arguments );

    //循环处理 num 个用户
    for ( var i = 0; i < num; i++ ) {
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
        //显示一个用户的信息
        print( prefix + ": " + users[i] );
```

变量类型检查和参数长度校验本身是很简单的概念，但是可用来实现复杂的方法，给开发者和你的代码的使用者提供更好的体验。接下来，我们将探讨 JavaScript 中的作用域以及怎么更好的控制它。

作用域

作用域是 JavaScript 中一个较难处理的特性。所有面向对象的编程语言都有某种形式的作用域；这要看是什么上下文约束着作用域。在 JavaScript 里，作用域由函数约束，而不由块约束(如 while,if,和 for 里的语句体)。最终可能使得一些代码的运行结果表面上显得怪异(如果你来自一种块作用域语言的话)。程序 2-10 的例子说明了"函数作用域代码"的含义。

代码 2-10. JavaScript 中变量作用域是怎样工作的例子

[Copy to clipboard] [-]

CODE:

```
//设置一个等于"test"的全局变量 foo
var foo = "test";
//在 if 块中
if ( true ) {
    //设置 foo 为"new test"
    //注意：这仍然是在全局作用域中
    var foo = "new test";
}
//正如我们在此处可见，foo 现在等于"new test"
alert( foo == "new test" );
//创建一个修改变量 foo 的函数
function test() {
    var foo = "old test";
}
//调用时，foo 却驻留在是在函数的作用域里面
test();
//确认一下，foo 的值仍然是"new test"
alert( foo == "new test" );
```

在程序 2-10 中你会发现，变量位于在全局作用域。基于浏览器的 JavaScript 有趣的一面是，所有的全局变量实际上都是 window 对象的属性。尽管一些老版本的 Opera 浏览器或 Safari 浏览器不是这样，假定浏览器这样工作通常是一个很好的经验规则。程序 2-11 展示了一个这种例子。

程序 2-11. JavaScript 的全局变量　　与 window 对象的例子

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

[Copy to clipboard] [-]

CODE:

```
//全局变量，包含字符串"test"
var test = "test";
//你会发现，我们的全局变量和 window 的 test 属性是相同的
alert( window.test == test );
```

最后，让我们来看看当一个变量漏定义时会怎样。程序 2-12 里，变量 foo 在 test()的作用域里被赋值。但是，程序 2-12 里实际并没有(用 var foo)定义变量的作用域。当变量 foo 没有明确定义时，它将成为全局变量，即使它只在函数的上下文使用。

程序 2-12. 隐式全局变量声明的示例

[Copy to clipboard] [-]

CODE:

```
//一个为变量 foo 赋值的函数
function test() {
    foo = "test";
}
//调用函数为 foo 赋值
test();
//我们发现 foo 现在是全局变量了
alert( window.foo == "test" );
```

到目前应该很明显，尽管 JavaScript 的作用域不如块作用域语言的严格，它还是相当强大和有特色的。尤其是与下节中叙述的闭包的概念结合起来时，JavaScript 语言的强大将展露无遗。

闭包

闭包意味着内层的函数可以引用存在于包绕它的函数的变量，即使外层的函数的执行已经终止。这一特殊的论题可能是非常强大又非常复杂的。我强烈推荐你们参考本节后面将提及的站点，因为它有一些关于闭包这一话题的精彩的信息。

我们先来看程序 2-13 所示的闭包的两个简单例子。

程序 2-13. 闭包改善的代码清晰性的两例

[Copy to clipboard] [-]

CODE:

```
//得到 id 为"main"的元素
var obj = document.getElementById("main");
//改变它的边框样式
obj.style.border = "1px solid red";
//初始化一个 1 秒钟以后被调用的回调函    数
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
setTimeout(function(){
    //此函数将隐藏该元素
    obj.style.display = 'none';
}, 1000);
//用来延迟显示消息的通用函数
function delayedAlert( msg, time ) {
    //初始化一个被封套的函数
    setTimeout(function(){
        //此函数使用了来自封套它的函数的变量 msg
        alert( msg );
    }, time );
}
//调用函数 delayedAlert，带两个参数
delayedAlert( "Welcome!", 2000 );
```

第一个对 setTimeout 的函数调用，展示了一个的 JavaScript 新手遇到问题的通俗的例子。在 JavaScript 新手的程序里像这样的代码时常可以看到：

[Copy to clipboard] [-]

CODE:

```
setTimeout("otherFunction()", 1000);
//或者甚至
setTimeout("otherFunction(" + num + "," + num2 + ")", 1000);
```

使用闭包的概念，完全可能的把这种混乱的代码清理掉。第一个例子很简单；有一个回调函数在调用 setTimeout 函数以后 1000 微秒以后被调用，而它仍引用了变量 obj（定义在全局范围，指向 id 为"main"的元素）。定义的第二个函数，delayedAlert，展示了一种解决出现的 setTimeout 混乱的方案，以及函数作用域内可以有闭包的能力。

你们应该可以发现，当在代码中使用这种简单的闭包时，你所写的东西的清晰性将会提高，免于陷入语法的迷雾之中。

我们来看一个闭包可能带来的有有趣的副作用。在某些函数化的编程语言里，有一个叫做 currying 的概念。本质上讲，currying 是就是为函数的一些参数预填入值，创建一个更简单的新函数的方法。代码 2-14 里有一个简单的 currying 的例子，创建了向另一个函数预填一个参数而得的新函数。

代码 2-14. 使用闭包的函数 currying

[Copy to clipboard] [-]

CODE:

```
//生成做加法的新函数的函数
function addGenerator( num ) {
    //返回一个简单函数用来计算两个数的加法，
    //其中第一个数字从生成器中借用
    return function( toAdd ) {
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
        return num + toAdd
    };
}
//addFive 现在是接受一个参数的函数，
//此函数将给参数加 5，返回结果数字
var addFive = addGenerator( 5 );
//这里我们可以看到，当传给它参数 4 的时候
//函数 addFive 的结果为 9
alert( addFive( 4 ) == 9 );
```

闭包还能解决另一个常见的 JavaScript 编码方面的问题。JavaScript 新手趋向于在全局作用域里放置许多变量。这一般被认为是不好的习惯，因为那些变量可能悄悄地影响其它的库，导致令人迷惑的问题的产生。使用一个自执行的、匿名的函数，你可以从根本上隐藏所有的通常的全局变量，使它们对其它代码不可见，如程序 2-15 所示。

代码 2-15. 使用匿名函数从全局作用域隐藏变量的例子

[Copy to clipboard] [-]

CODE:

```
//创建一个用作包装的匿名函数
(function(){
    //这个变量通常情况下应该是全局的
    var msg = "Thanks for visiting!";
    //为全局对象绑定新的函数
    window.onunload = function(){
        //使用了"隐藏"的变量
        alert( msg );
    };
//关闭匿名函数并执行之
})();
```

最后，让我们来看使用闭包时出现的一个问题。闭包允许你引用存在于父级函数中的变量。然而，它并不是提供该变量创建时的值；它提供的是父级函数中该变量最后的值。你会看到这个问题最通常是在一个 for 循环中。有一个变量被用作迭代器(比如 i)，在 for 内部新的函数被创建，并使用了闭包来引用该迭代器。问题是，当新的闭包函数被调用时，它们将会引用该 iterator 最后的值(比如，一个数组的最后位置)，而不是你所期望的那个。程序 2-16 的例子说明，使用匿名函数激发作用域，在其中创建一个合乎期望的闭包是可能的。

程序 2-16. 使用匿名函数激发一个创建多个闭包函数所需的作用域的例子

[Copy to clipboard] [-]

CODE:

```
//id 为"main"的一个元素
var obj = document.getElementById("main");
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
//用来绑定的 items 数组
var items = [ "click", "keypress" ];
//遍历 items 中的每一项
for ( var i = 0; i < items.length; i++ ) {
    //用自执行的匿名函数来激发作用域
    (function(){
        //在些作用域内存储值
        var item = items[i];
        //为 obj 元素绑定函数
        obj[ "on" + item ] = function() {
            //item 引用一个父级的变量,
            //该变量在此 for 循环的上文中已被成功地 scoped(?)
            alert( "Thanks for your " + item );
        };
    })();
}
```

闭包的概念并非轻易可以掌握的;我着实花了大量的时间和精力才彻底弄清闭包有多么强大。幸运的是，有一个精彩的资源解释了 JavaScript 中的闭包是怎么工作的：Jim Jey 的 "JavaScript 闭包"，网址是 http://jibbering.com/faq/faq_notes/closures.html。

最后,我们将研究上下文的概念,这是许多 JavaScript 的面向对象特性赖以建立的基石。

上下文

在 JavaScript 中，你的代码将总是有着某种形式的上下文(代码在其内部工作的对象)。这也是其它面向对象语言所共有的功能，但它们都不如 JavaScript 处理得这样极端。

上下文是通过变量 this 工作。变量 this 总是引用代码当前所在的那个对象。记住全局对象实际上是 window 对象的属性。这意味着即使是在全局上下文里，this 变量仍然引用一个对象。上下文可以成为一个强大的工具，是面向对象代码不可或缺的一环。程序 2-17 展示了一些关于上下文的简单例子。

程序 2-17. 在上下文中使用函数然后将其上下文切换到另一个变量的例子

[Copy to clipboard] [-]

CODE:

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


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
};
//我们看到，obj 对象没有"val"的属性
alert( obj.val == null );
//我们运行 yes 函数，它将改变附着在 obj 对象的 val 属性
obj.yes();
alert( obj.val == true );
//然而，我们现在让 window.no 指向 obj.no 方法，并运行之
window.no = obj.no;
window.no();
//这导致 obj 对象保持不变(上下文则切换到了 window 对象)，
alert( obj.val == true );
//而 window 的 val 属性被更新
alert( window.val == false );
```

你可能已经注意到，在程序 2-17 中，当我们切换 obj.no 方法的上下文到变量 window 时，笨重的代码需要切换函数的上下文。幸运的是，JavaScript 提供了两种方法使这一过程变得更加易于理解和实现。程序 2-18 展示了恰能些目的的两种不同方法，call 和 apply。

程序 2-18. 改变函数上下文的示例

[Copy to clipboard] [-]

CODE:

```
//一个简单的设置其上下文的颜色风格的函数
function changeColor( color ) {
    this.style.color = color;
}
//在 window 对象上调用这个函数将会出错，因为 window 没有 style 对象
changeColor( "white" );
//得到一个 id 为"main"的对象
var main = document.getElementById("main");
//用 call 方法改变它的颜色为黑
//call 方法将第一个参数设置为上下文，
//并其它所有参数传递给函数
changeColor.call( main, "black" );
//一个设置 body 元素的颜色的函数
function setBodyColor() {
    //apply 方法设置上下文为 body 元素
    //第一个参数为设置的上下文，
    //第二个参数是一个被作为参数传递给函数的数组
    // of arguments that gets passed to the function
    changeColor.apply( document.body, arguments );
}
//设置 body 元素的颜色为黑
setBodyColor( "black" );
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

上下文的有用性此处可能还没有立即显现。当我们进入下一节"面向对象的 JavaScript"时，它会变得更加明显。

面向对象基础

"面向对象的 JavaScript"这一说法多少有些冗余，因为 JavaScript 语言本就是完全面向对象的，不可能有另外的用法。但是，初学编程者(包括 JavaScript 编程者)共有的一个缺点就是，功能性地编写代码而不使用任何上下文或分组。要完全理解怎么编写优化的 JavaScript 代码，你必须理解 JavaScript 的对象是怎样工作的，它们与其它语言有怎样的不同，以及怎样让它们为你所用。

本章的剩余部分我们将讨论用 JavaScript 编写面向对象代码的基础，在后面的几章中，我们将看到以这种方式编写代码的实例。

对象

对象是 JavaScript 的基础。实际上 JavaScript 语言中的一切都是对象，JavaScript 的多数能力也正起源于此。在其最根本的层面上，对象作为属性的集合存在，差不多类似于你在其它语言中看到的哈希的概念。程序 2-19 展示了创建两个带有一组属性的对象的基本示例。

程序 2-19. 创建简单对象并设置其属性的两个例子

[Copy to clipboard] [-]

CODE:

```
//创建一个新对象并将其存放在 obj 里
var obj = new Object();
//将该对象的一些属性设置成不同的值
obj.val = 5;
obj.click = function(){
    alert( "hello" );
};
//下面是等效的代码，使用了{...}式缩写，
//和定义对象属性的"名称-值"对
var obj = {
    //用名称-值对设置对象属性
    val: 5,
    click: function(){
        alert( "hello" );
    }
};
```

实际上对象就这么回事了。然而，事情变得麻烦的地方，在于新对象(尤其是那些继承

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

其它对象属性的对象)的创建。

对象创建

不像大多数其它面向对象的语言，JavaScript 实际上并没有类的概念。在大多数其它的面向对象语言中，你可以初始化一个特定的类的实例，但是在 JavaScript 中的情况这是这样。在 JavaScript 中，对象能够创建新的对象，对象可以从继承自其它对象。整个概念被称为"prototypal inheritance"(原型标本继承)，将在"公有方法"一节中有更多论述。

然而，重要的是，不论 JavaScript 采用哪种对象方案，总归要有一个方式来创建新的对象。JavaScript 的做法是，任何一个函数也都能作为一个对象被实例化。实际上，事情听起来远比它本身更令人困惑。好比有一块生面团（相当于原始的对象），用小甜饼切割器（相当于对象构造器，使用对象的原型 prototype）为其成形。

让我们看看程序 2-20 中这一机制的工作的实例

程序 2-20. 创建并使用一个简单的对象

[Copy to clipboard] [-]

CODE:

```
//一个简单的函数，接受一个参数 name，
//并将其保存于当前上下文中
function User( name ) {
    this.name = name;
}
//用指定的 name 创建上述函数的新实例
var me = new User( "My Name" );
//我们可以看到 name 已经被成为对象本身的属性
alert( me.name == "My Name" );
//而且它确实是 User 对象的一个新实例
alert( me.constructor == User );
//那么，既然 User()只是一个函数，
//当我们这么处理它的时候，发生了什么？
User( "Test" );
//因为 this 上下文没有被设置，它缺省地指向全局的 window 对象，
//这意味着 window.name 将等于我们提供给它的那个 name
alert( window.name == "Test" );
```

程序 2-20 说明了 constructor 属性的使用。这个存在于每一个对象中的属性将总是指向创建该对象的那个函数。于是，你可以方便的复制该对象，创建一个新的有共同基类和不同属性的对象。示例见程序 2-21.

程序 2-21. 使用 constructor 属性一例

[Copy to clipboard] [-]

CODE:

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
//创建一个新的、简单的 User 对象(函数)
function User() {}
//创建一个新的 User 对象
var me = new User();
//也是创建一个新的 User 对象(使用上前一个对象的 constructor)
var you = new me.constructor();
//我们可以看到，实际上它们的 constructor 是同一个
alert( me.constructor == you.constructor );
```

公有方法

公有方法可以完全地被对象的上下文中的最终使用者访问。为了实现这些对于特定对象的所有实例都可用的公共方法，你需要学习一个名为"prototype"的属性。prototype 简单地包含一个对象，为一个父对象的所有新副本充当对基类的引用。本质上，prototype 的任何属性对该对象的所每一个实例都是可用的。创建/引用的过程给了我们一个廉价版的继承，这一点我将在第三章论及。

由于对象的 prototype 也是一个对象，就跟其它任何对象一样，你可以给它附加新的属性。附加给 prototype 的新的属性将成为从原来的 prototype 对象实例化的每个对象的一部分，有效地使得该属性成为公有的(且可为全部实例所访问)。程序 2-22 展示一个此类例子：

程序 2-22. 带有通过 prototype 附加的方法的对象的例子

[Copy to clipboard] [-]

CODE:

```
//创建一个新的 User 的构造器
function User( name, age ){
    this.name = name;
    this.age = age;
}
//为 prototype 对象添加一个新方法
User.prototype.getName = function(){
    return this.name;
};
//为 prototype 对象添加另一个方法
//注意此方法的上下文将是被实例化的对象
User.prototype.getAge = function(){
    return this.age;
};
//实例化一个新的 User 对象
var user = new User( "Bob", 44 );
//我们可以看到两个方法被附加到了对象上，有着正确的上下文
alert( user.getName() == "Bob" );
alert( user.getAge() == 44 );
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

私有方法

私有方法和变量只能被其它的私有方法、私有变量的特权方法(下一节将会论述)访问。这是一种定义只能在内象内部访问的代码的方式。这一技术得益于 Douglas Crockford 的工作。他的网站提供了大量的详述面向对象的 JavaScript 的工作机制和使用方法的文档：

JavaScript 文章列表：http://javascript.crockford.com/

文章"JavaScript 中的私有成员":http://javascript.crockford.com/private.html

我们来看一个私有方法可以怎样应用中的例子，如程序 2-23 所示.

程序 2-23. 私有方法只能被构造函数使用的示例：

[Copy to clipboard] [-]

CODE:

```
//一个表示教室的对象构造器
function Classroom( students, teacher ) {
    //用来显示教室中的所有学生的私有方法
    function disp() {
        alert( this.names.join(", ") );
    }

    //课程的数据存储在公有的对象属性里
    this.students = students;
    this.teacher = teacher;

    //调用私有方法显示错误
    disp();
}
//创建一新的教室对象
var class = new Classroom( [ "John", "Bob" ], "Mr. Smith" );
//失败,因为 disp 不是该对象的公有方法
class.disp();
```

尽管很简单，私有方法却是非常重要的，它可以在保持你的代码免于冲突同时允许对你的用户可见和可用的施以更强大的控制。接下来，我们来研究特权方法。它是你的对象中可以使用的私有方法和共有方法的联合。

特权方法

"特权方法"一语是 Douglas Crockford 创造的，用来称呼那种能够观察和维护私有变量而又可以作为一种公有方法被用户访问的方法。程序 2-24 展示了使用特权方法的一个例子。

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

程序 2-24　使用特权方法一例

[Copy to clipboard] [-]

CODE:

```
//创建一个新的 User 对象构造器
function User( name, age ) {
    //计算用户的出生年份
    var year = (new Date()).getFullYear() - age;
    //创建一个新特权方法，对变量 year 有访问权，
    //但又是公共可访问的
    this.getYearBorn = function(){
        return year;
    };
}
//创建一个 User 对象的新实例
var user = new User( "Bob", 44 );
//验证返回的出生年份是否正确
alert( user.getYearBorn() == 1962 );
//并注意我们不能访问对象的私有属性 year
alert( user.year == null );
```

本质上，特权方法是动态生成的方法，因为它们是在运行时而不是代码初次编译时添加给对象的。这种技术在计算量上要比绑定一个简单的方法到对象的 prototype 上来得昂贵，但同时也的强大和灵活得多。程序 2-25 展示了使用动态生成的方法可以实现什么。

程序 2-25. 新对象初始化时创建的动态方法的示例

CODE:

```
//创建一个新的接受 properties 对象的对象
function User( properties ) {
    //遍历对象属性，确保它作用域正确(如前所述)
    for ( var i in properties ) { (function(){
        //为属性创建获取器
        this[ "get" + i ] = function() {
            return properties[i];
        };
        //为属性创建设置器
        this[ "set" + i ] = function(val) {
            properties[i] = val;
        };
    })(); }
}
//创建一个新 user 对象实例，传入一个包含属性的对象作为种子
var user = new User({
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
    name: "Bob",
    age: 44
});
//请注意 name 属性并不存在，因为它在 properties 对象中，是私有的
alert( user.name == null );
//然而，我们能够使用用动态生成的方法 getname 来访问它
alert( user.getname() == "Bob" );
//最后，我们能看到，通过新生成的动态方法设置和获取 age 都是可以的
user.setage( 22 );
alert( user.getage() == 22 );
```

(译注：这段程序是错误的。那个匿名函数里的 this 错误地指向了匿名函数的上下文，而其中的变量 i 却又恰仍属 User 的上下文)

修正版：

CODE:

```
<script>

//创建一个新的接受 properties 对象的对象

function User( properties ) {

    //遍历对象属性，确保它作用域正确(如前所述)

    for ( var i in properties ) { (function(which){ var p=i

        //为属性创建获取器

        which[ "get" + i ] = function() {

            return properties[p];

        };

        //为属性创建设置器

        which[ "set" + i ] = function(val) {

            properties[p] = val;

        };

    })(this); }

}
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
//创建一个新 user 对象实例，传入一个包含属性的对象作为种子
var user = new User({
    name: "Bob",
    age: 44
});
//请注意 name 属性并不存在，因为它在 properties 对象中，是私有的
alert( user.name == null );
//然而，我们能够使用用动态生成的方法 getname 来访问它
alert( user.getname() == "Bob" );
//最后，我们能看到，通过新生成的动态方法设置和获取 age 都是可以的
user.setage( 22 );
alert( user.getage() == 22 );
</script>
```

原作者使用闭包本意大概是想要保持 for 循环中计数器 i 的状态（否则当方法被调用时，将指向 i 最终状态所对应的项目，也就是说 getname 将返回 age 的值），但是竟忘了为其设置变量（我这里的变量 p），而且还忘了传递正确的上下文。看来这位老兄写书时比较赶时间，没有测试代码。

动态生成的代码的力量不可低估。能够基于变量的值实时的生成代码是极其有用；这与在其它语言(如 Lisp)中宏那样强大的道理是一样的，不过是放在一种现代编程语言的背景里。接下来，我们将看到一类纯粹因其组织上的优势而有用的方法。

静态方法

静态方法背后的前提其实跟其它任何方法是一样的。然而，最主要的不同在于，这些方法作为对象的静态属性而存在。作为属性，它们在该对象的实例上下文中不可访问；它们只有在与主对象本身相同的上下文是可用的。这些与传统的类继承的相似点，使得他们有点像是静态的类方法。

实际上，以这种方式编写代码的唯一好处在于，这种方法保持对象名称空间的干净，——这一概念我就在第三章中更一步论述。程序 2-26 展示了附加在对象上的静态方法的一个例子。

程序 2-26. 静态方法的简单示例

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

[Copy to clipboard] [-]

CODE:

```
//附加在 User 对象上的一个静态方法
User.cloneUser = function( user ) {
    //创建并返回一个新的 User 对象
    return new User(
        //该对象是其它 user 对象的克隆
        user.getName(),
        user.getAge()
    );
};
```

静态方法是我们遇到的第一种纯粹以组织代码为目的的方法。这是向我们将要看到的下一章的重要过渡。开发专业品质 JavaScript 的一个基本侧观点，就是要有能力快速、平静地与其它代码段接口，同时保持可理解地可用性。这是一个重要的奋斗目标，也是我们下一章里所期望达到的。

本章摘要

理解本章概念的大纲的重要性是不容忽视的。本章的前半部分，让你对于 JavaScript 语言怎样运作和怎样最好地它用一个良好的理解，这是完全掌握专业地使用 JavaScript 的出发点。彻底地理解对象怎样运作、引用怎样处理、作用域怎样确定，将会毫无疑问地改变你编写 JavaScript 代码的方式。

有了广博的 JavaScript 编码技能，编写干净的面向对象 JavaScript 代码的重要性将会变得更加明显。本章的后半部分里我论述了怎样着手编写种种面向对象的代码以适应来自其它编程语言阵营的任何人。现代 JavaScript 正是基于这些技能，给予你开发新型的创新的应用程序时巨大的优势。

程序 2-25. 的错误分析

QUOTE:

(译注：这段程序是错误的。那个匿名函数里的 this 错误地指向了匿名函数的上下文，而其中的变量 i 却又恰仍属 User 的上下文)

Mozart0 在译注里说的"匿名函数的上下文"也就是全局上下文，此间 this 指向的是 window 对象。

通过下面的代码可以彰显程序 2-25 中的所有错误：

1、没有为匿名函数传递正确的上下文。

2、没有在闭包中设置变量保存 for 循环中计数器变量 i 的状态（似乎是忘了，或是想当然地以为它会自动保持？作者是大家的样子，应该不至于 ）。

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

第三章　创建可重用的代码

当与其它程序员共同开发代码时（这里对大多数合作或团队项目来说是很常见的），为了保持你们的清醒而维护良好的编程惯例将会变得极其重要。随着近年来 JavaScript 已经开始得到认可，专业程序员所编写的 JavaScript 代码量急剧增加。这种观念上的转变和 JavaScript 的使用导致围绕它的开发惯例得到了长足的发展。

在这一章里，我们将学到一些清理、更好地组织代码，并改进代码质量使其能够为他人所用的一些方法。

标准化面象对象代码

编写可重用代码的第一个也是最重要的步骤就是以一种贯穿整个应用程序的标准方式编写你的代码，尤其是面向对象的代码。通过上一章的面向对象的 JavaScript 的运作方式，你可以看到 JavaScript 语言相当灵活，允许你模拟许多种不同的编程风格。

作为开端，设计出一种最符合你需要的编写面向对象代码并实现对象继承(把对象的属性克隆到新的对象里)的体制是很重要的。然而表面看来，每一个写过一些面向对象的 JavaScript 代码的人都已经建立起了各自的实现方案，这可能相当令人困惑的。在这一节中，我们将弄清 JavaScript 的中继承是怎样工作的，随后了解几种不同的供选择的辅助方法的原理以及怎样将它们应用于你的程序当中。

原型继承

JavaScript 使用了一种独特的对象创建和继承的方式，称为原型继承（prototypal inheritance）。这一方法背后的前提（相对大多数程序员所熟悉的传统的类/对象方案而言）是,一个对象的构造器能够从另一个对象中继承方法，建立起一个原型对象，所有的新的对象都将从这个原型创建。

这整个过程由 prototype 属性（存在于每一个函数中，因为任何函数都可以是一个构造器）促成。原型继承是为单继承设计的；尽管如此，仍然存在可以实现多继承的手段，我将在下一节中讨论。

使得这种形式的继承特别难以掌握的是，原型并不从其它的原型或者其它的构造器继承属性，而是从实际的对象中继承。程序 3-1 展示了 prototype 属性怎样被用于简单继承的几个例子。

程序 3-1. 原型继承的例子

[Copy to clipboard] [-]

CODE:

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
//创建 Person 对象的构造器
function Person( name ) {
    this.name = name;
}
//为 Person 对象加入一个新方法
Person.prototype.getName = function() {
    return this.name;
};
//创建一个新的 User 对象构造器
function User( name, password ) {
    //注意这并不支持优雅的重载/继承，
    //如能够调用超类的构造器
    this.name = name;
    this.password = password;
};
//User 对象继承 Person 对象的全部方法
User.prototype = new Person();
//我们添加一个自己的方法给 User 对象
User.prototype.getPassword = function() {
    return this.password;
};
```

上例中最重要的一行是 User.prototype = new Person();。我们来深入地看看这到底意味着什么。User 是对 User 对象的函数构造器的引用。new Person()建创一个新的 Person 对象，使用 Person 构造器。将这一结果设为 User 构造器的 prototype 的值，这意味着不论任何时候你使用 new User()的时候，新建的 User 类对象也将拥有你使用 new Person()时创建的 Person 类对象的所有方法。

带着这一特殊的技巧，我们来看一些不同的开发者所编写的使得 JavaScript 中继承的过程简单化的封装。

类继承

类继承(classical inheritance)是多数开发者所熟悉的一种形式，拥有带方法的可被实例化为对象的类。对初学面向对象 JavaScript 的程序员来说这种情况是非常典型的：试图模拟这种程序构思，却很少真正悟出怎样正确地实现。

值得感激的是，JavaScript 大师之一，Douglas Crockford，把开发一套能用于 JavaScript 模拟类式继承的简单方法做为了他的目标，如他在网站上所解释的那样 (http://javascript.crockford.com/inheritance.html)。

程序 3-2 展示了他所编写的三个函数，用来建立起一种类风格的 JavaScript 继承的综合形式。每个函数实现了继承的一个方面：继承单个函数，继承单个父类的全部，和从多个父类中继承独立的方法。

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

程序 3-2. Douglas Crockford 的使用 JavaScript 模拟类形式继承的三个函数

[Copy to clipboard] [-]

CODE:

```
//一个简单的辅助函数，允许你为对象的原型绑定新的函数
Function.prototype.method = function(name, func) {
    this.prototype[name] = func;
    return this;
};
//一个(相当复杂的)函数，允许你优雅地从其它对象中继承函数，
//同时仍能调用"父"对象的函数
Function.method('inherits', function(parent) {
    //追踪所处的父级深度
    //Keep track of how many parent-levels deep we are
    var depth = 0;
    //继承 parent 的方法
    //Inhert the parent's methods
    var proto = this.prototype = new parent();

    //创建一个名为 uber 的新的特权方法，
    //调用它可以执行在继承中被覆盖的任何函数
    //Create a new 'priveledged' function called 'uber', that when called
    //executes any function that has been written over in the inheritance
    this.method('uber', function uber(name) {

        var func; //将被执行的函数(The function to be execute)
        var ret; // 该函数的返回值(the return value of then function)
        var v = parent.prototype; //父类的 prototype(The parent's prototype)
        //如果已经位于另一"uber"函数内
        //If we're already within another 'uber' function
        if (depth) {
            //越过必要的深度以找到最初的 prototype
            //Go the necessary depth to function the orignal prototype
            for ( var i = d; i > 0; i += 1 ) {
                v = v.constructor.prototype;
            }

            //并从该 prototype 取得函数
            //and get the functin from that prototype
            func = v[name];

        //否则，这是第一级的 uber 调用
        //Otherwise, this is the first 'uber' call
        } else {
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
            //从prototype中取得函数
            //Get the function to execute from the prototype
            func = proto[name];

            //如果该函数属于当前的prototype
            //If the function was a part of this prototype
            if ( func == this[name] ) {
                //则转入parent的prototype替代之
                //Go to the parent's prototype instead
                func = v[name];
            }
        }

        //记录我们位于继承栈中的'深度'
        //Keep track of how 'deep' we are in the inheritance stack
        depth += 1;

        //使用用第一个参数后面的所有参数调用该函数
        //(第一个参数保有我们正在执行的函数的名称)
        //Call the function to execute with all the arguments but the first
        //(which holds the name of the function that we're executing)
        ret = func.apply(this, Array.prototype.slice.apply(arguments, [1]));

        //重置栈深度
        //Reset the stack depth
        depth -= 1;

        //返回执行函数的返回值
        //Return the return value of the execute function
        return ret;
    });
    return this;
});
//一个用来仅继承父对象中的几个函数的函数,
//而不是使用new parent()继承每一个函数
Function.method('swiss', function(parent) {
    //遍历所有要继承的方法
    for (var i = 1; i < arguments.length; i += 1) {
        //要导入的方法名
        var name = arguments[i];

        //将方法导入这个对象的prototype
        this.prototype[name] = parent.prototype[name];
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
    }
    return this;
});
```

我们来看看这三个函数到底提供给我们些什么，以及为什么我们应该使用它们而不去试图写出我们自己的原型继承模型。这三个函数的前提是简单的：

*Function.prototype.method:*此函数是为构造器的 prototype 附加函数的简单方式。这一特殊的子句能够工作是因为所有的构造器都是函数，故能获得新的方法"method"。

*Function.prototype.inherits:*这一函数能用来提供简单的单父继承。函数代码的主体围绕着在你的对象的任何方法中调用 this.uber("方法名")使之执行它所重写了的父对象的方法的能力。这是 JavaScript 继承模型本身不具备的一个方面。

*Function.prototype.swiss:*这是.method()函数的一个高级版本，能用来从一个父对象中抓取多个方法。当将它分别用于多个父对象时，你将得到一种实用的多父继承的形式。

对上面三个函数提供给我们什么有了一个大致的了解之后，程序 3-3 重拾你在 3-1 中所见的 Person/User 的例子，不过这次使用了新的类风格的继承。另外，你可以看看在改善程序清晰性方面，这个库能够提供怎样的额外功能。

程序 3-3. Douglas Crockford 的类继承式 JavaScript 函数的例子。

[Copy to clipboard] [-]

CODE:

```
//创建一个新的 Person 对象构造器
function Person( name ) {
    this.name = name;
}
//给 Person 对象添加方法
Person.method( 'getName', function(){
    return name;
});
//创建新一个新的 User 对象构造器
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 对象添加一个新方法
User.method( 'getPassword', function(){
    return this.password;
});
//重写新 Person 对象创建的方法,
//但又使用 uber 函数再次调用它
User.method( 'getName', function(){
    return "My name is: " + this.uber('getName');
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
});
```

尝试过使用一个可靠的继承加强的 JavaScript 库所带来的可能性之后，我们再来关注其它的一些广通用的流行的方法。

Base 库

JavaScript 对象创建和继承领域近期的成果是 Dean Edwards 所开发的 Base 库。这一特别的库提供了一些不同的方式来扩展对象的功能。除此之外，它甚至提供了一种直觉式的对象继承方式。Dean 最初开发这个库是为了用于他的其它的项目，包括 IE7 项目（作为对 IE 一整套的升级)。Dean 的网站上列出的例子相当易于理解并确实很好的展示了这个库的能力:http://dean.edwards.name/weblog/2006/03/base 。除此而外，你可以在 Base 源代码目录里找到更多的例子:http://dean.edwards.name/base/。

Base 库是相当冗长而复杂的，它值得用额外的注释来说明（包含于 http://www.apress.com 的 Source Code/Download 所提供的代码中）。除了通读注释过的代码以外，强烈建议你去看 Dean 在他的网站上提供的例子，因为它们非常有助于澄清常见的疑惑。

但作为起点，我将带你一览 Base 库的几个可能对你的开发很有帮助的重要的方面。具体地，在程序 3-4 展示了类创建、单父继承和重写父类函数的例子。

程序 3-4. 利用 Dean Edwards 的 Base 库进行简单的类创建和继承的例子

[Copy to clipboard] [-]

CODE:

```
//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 类的简单方法
    getName: function() {
        return this.name;
    }
});
//创建一个新的继承了 Person 类的 User 类
var User = Person.extend({
    //创建 User 类的构造器,
    constructor: function( name, password ) {
        //该构造器顺次调用了父类的构造器方法
        this.base( name );
        this.password = password;
    },
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
    //为 User 类创建另一个简单的方法
    getPassword: function() {
        return this.password;
    }
});
```

我们来看看在程序 3-4 中 Base 库是如何达到先前所归纳的三个目标从而创造出一种对象创建和继承的简单形式的。

Base.extend(...);:这一表达式用来创建一个新的基本的构造器对象。此函数授受一个参数，即一个简单的包含属性和值的对象，其中的属性都会作为原型方法被被增添到(所创建的构造器)对象中。

Person.extend(...);:这是 Base.extend()语法的一个可替换版本。所有的创建的构造器都使用.extend()方法获取它们自己的.extend()方法，这意味着直接从它们继承是可能的。程序 3-4 中，正是通过直接从最初的 Person 构造器中直接继承的方式创建了 User 构造器。

this.base();:最后，this.base()方法用来调用父对象的被重写了的对象。你会发现这与 Corockford's 的类继承所使用的 this.uber()函数截然是截然不同的，你无需提供父类的方法名(这一点有助于真正地清理并明晰化你的代码)。在所有的面向对象的 JavaScript 库中，Base 库的重写父方法的功能是最好的。

个人而言，我觉得 Dean 的 Base 库能够出产最可读的、实用的和可理解的面向对象的 JavaScript 代码。当然，最终选择什么库要看开发者自己觉得什么最适合他。接下来你将看到面对对象的 JavaScript 代码如何在流行的 Prototype 库中实现。

Prototype 库

Prototype 是一个为了与流行的"Ruby on Rails"web 框架协同工作而发的 JavaScript 库。不要把库的名字与构造器的 prototype 属性混淆——那是只一种令人遗憾的命名情况。

撇开命名不谈，Prototype 库使得 JavaScript 外观和行为上者更接近于 Ruby。为达到这一点，Prototype 的开发者们利用了 JavaScript 的面向对象本质，并且附加了一些函数和属性给核心的 JavaScript 对象。不幸的是，该库根本不是由它的创造者们给出文档的；而幸运的是它写得非常清晰，而且它的一些用户介入编写了他们自己版本的文档。你们可以在 Prototype 的网站(http://prototype.conio.net/)上随意地浏览完整的代码，从文章"Painless JavaScript Using Prototype"里得到 Prototype 的文档。

在这一节里，我们将仅着眼于 Prototype 用于创建其面象对象结构并提供基本继承的特定的函数和对象。程序 3-5 展示了 Prototype 使用的达到此目标的全部代码。

程序 3-5. Prototype 所使用的模拟面向对象 JavaScript 代码的两个函数

[Copy to clipboard] [-]

CODE:

```
//创建一个名为"Class"的全局对象
var Class = {
    //它拥有一个用来创建新的对象构造器的函数
    create: function() {
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
        //创建一个匿名的对象构造器
        return function() {
            //调用它本身的初始化方法
            this.initialize.apply(this, arguments);
        }
    }
}
//为对象"Object"添加静态方法，用以从一个对象向另一个对象复制属性
Object.extend = function(destination, source) {
    //遍历欲扩展的所有属性
    for (property in source) {
        //并将它添加到目标对象
        destination[property] = source[property];
    }
    //返回修改过的对象
    return destination;
}
```

Prototype 确实只用了两个明显的函数来创建和维护其整个面向对象体系。你们可能已发现，仅通过看观察代码，也能断定它不如 Base 或者 Crockford 的类式方法那样强大。两个函数的前提很简单：

*Class.create():*这个函数简单地返回一个可用做构造器的匿名函数包装。这个简单的构造器做了一件事：调用和执行对象的 initialze 属性。这意味着，你的对象里至少有一个包含函数的 initialize 属性；否则，代码将会出错。

*Object.extend():*这个函数简单地从一个对象往另一个对象复制属性。当你使用构造器的 prototype 属性时你能设计出一种更简单的继承的形式（比 JavaScript 中可用的缺省的原型继承更简单）。

既然你已经了解了 Prototype 的底层代码是如何工作的，程序 3-6 展示了一些例子，说明它在 Prototype 库自身中是怎样用来通过添加功能层来扩展天然的 JavaScript 对象的。

程序 3-6. Prototype 怎样使用面对对象函数扩展 JavaScript 中字符串的缺省操作的例子。

[Copy to clipboard] [-]

CODE:

```
//为 String 对象的原型添加额外的方法
Object.extend(String.prototype, {
    //一个新的 stripTags 函数，删除字符串中的所有 HTML 标签
    stripTags: function() {
        return this.replace(/<\/?[^>]+>/gi, '');
    },
    //将一个字符串转换成一个字符的数组
    toArray: function() {
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
        return this.split('');
    },
    //将文本"foo-bar"转换成'骆驼'文本"fooBar"(译注：fooBar 中间的大写字符像是驼峰吧)
    //Converts "foo-bar" text to "fooBar" 'camel' text
    camelize: function() {
        //以'-'拆分字符串
        var oStringList = this.split('-');
        //若字符串中没有'-'则提前返回
        if (oStringList.length == 1)
            return oStringList[0];
        //随意地"骆驼化"字符串的开头
        //Optionally camelize the start of the string
        var camelizedString = this.indexOf('-') == 0
            ? oStringList[0].charAt(0).toUpperCase() +
oStringList[0].substring(1)
            /*
              译注：this.indexOf('-')==0，那 oStringList[0]显然就是空字符串了，
              有必要 toUpperCase 加 substring 吗？
            */
            : oStringList[0];

        //将后继部分的首字母大写
        for (var i = 1, len = oStringList.length; i < len; i++) {
            var s = oStringList[i];
            camelizedString += s.charAt(0).toUpperCase() + s.substring(1);
        }

        //返回修改的字符串
        return camelizedString;
    }
});
//stripTags()方法的一个例子
//可以看到它删除了字符串中的所有 HTML
//只保留纯文本
"<b><i>Hello</i>, world!".stripTags() == "Hello, world!"
//toArray()方法的一个例子
//我们将得到字符串中的第四个字符
"abcdefg".toArray()[3] == "d"
//camelize()方法的例子
//它将原字符串转换成新的格式
"background-color".camelize() == "backgroundColor"
```

接下来，让我们再一次回到这章我所用到的那个有着 User 和 Person 对象且 User 对象

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

从 Person 对象继承属性的例子。使用 Prototype 的面向对象风格的代码，见程序 3-7。

程序 3-7. Prototype 的用于创建类和实现简单继承的辅助函数

[Copy to clipboard] [-]

CODE:

```
//用名义上的构造器创建一个 Person 对象
var Person = Class.create();
//将下列的函数复制给 Person 的 prototype
Object.extend( Person.prototype, {
    //此函数立即被 Person 的构造器调用
    initialize: function( name ) {
        this.name = name;
    },
    //Person 对象的简单函数
    getName: function() {
        return this.name;
    }
});
//用名义上的构造器创建一个 User 对象
var User = Class.create();
//User 对象从其父类继承所有属性
User.prototype = Object.extend( new Person(), {
    //用新的初始化函数重写原来的
    initialize: function( name, password ) {
        this.name = name;
        this.password = password;
    },

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

尽管 Prototype 库所提出的面向对象技术不是革命性的，它们也强大到足以帮助开发者创建更简单、更易编写的代码了。然而，如果你正将编写数量巨大的面向对象代码，最终你可能更趋向于选择 Base 这样的库来辅助你的工作。

接下来我们将探讨怎样处理你的面向对象的代码，并使之准备好被其它的开发者或库所使用并与之相合。

封装

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

完成你的漂亮的面向对象 JavaScript 代码编写以后(或者之前，如果你够聪明的话)，是时候改善它使之能够与其它 JavaScript 库和谐共处了。意识到你的代码将被其他的可能需求跟你完全不同的开发者或者用户使用，这一点非常重要。尽可能编写最整洁的代码对此将有所帮助，但是从他人已经做到的去学习，也会达到同样的效果。

在这一节里，你将会看到成千上万的开发者日常使用的几个大型的库。每种库都提供了独特的方式来管理其结构，使之容易学习和使用。另外，你将会看到用来清理你的代码的几种方式，以便于尽可能提供最好的体验给其他人。

命名空间

一个重要和简单的可用来清理和简化你的代码的技术是命名空间(namespacing)的概念。JavaScript 目前并不缺省地支持命名空间(不像 Java 或 Python，比如说)，因此我们不得不设法以一种相似且能够满足需要的技术来实现。

实际上，JavaScript 里不存在固有的命名空间的这种东西。但是，利用 JavaScript 的前提，即所有的对象都能拥有属性，属性又能依次包含其它对象，你可以创造出起一种看起来和工作起来都跟你在其它语言中使用的命名空间极其相似的东西。使用这种技术，你可以建立起如程序 3-6 所示的独特的结构。

程序 3-8. JavaScript 中的命名空间及其实现

[Copy to clipboard] [-]

CODE:

```
//创建一个缺省的，全局的命名空间
var YAHOO = {};
//设置一些子命名空间，使用对象
YAHOO.util = {};
//创建最终的命名空间，包含作为属性的函数
YAHOO.util.Event = {
    addEventListener: function(){ … }
};
//在特定的命名空间里调用函数
YAHOO.util.Event.addEventListener( … )
```

我们来考察几种不同的浏行的库中使用命名空间的例子，以及它们对于一致的、可扩展的、插入式的体系结构有着怎样的益处。

Dojo

Dojo 是一个极其流行的框架，提供给开发者建造完整的 web 应用程序所需的一切。这意味着有大量的需被包含和独立评估的子库，否则整个库简直会庞大到不能处理。关于 Dojo 的更多信息可以在其项目站点上找到：http://dojotoolkit.org/。

Dojo 有一整套的围绕 JavaScript 命名空间的封装系统。基于这一系统你可以动态地导入新的程序包，它们会自动地执行以备使用。程序 3-9 展示了 Dojo 里使用的命名空间的一个例子。

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

程序 3-9. Dojo 中的封装和命名空间

[Copy to clipboard] [-]

CODE:

```
<html>
<head>
    <title>Accordion Widget Demo</title>
    <!-- 包含 Dojo 框架 -->
    <script type="text/javascript" src="dojo.js"></script>
    <!-- 包含不同的 Dojo 包 -->
    <script type="text/javascript">
        //导入两个不同的包,
        //用来创建 Accordian Container widget("可折叠容器"界面工具)
        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>Pane 1</h2>
        <p>Nunc consequat nisi vitae quam. Suspendisse sed nunc. Proin…</p>
    </div>
    <div dojoType="ContentPane" label="Pane 2">
        <h2>Pane 2</h2>
        <p>Nunc consequat nisi vitae quam. Suspendisse sed nunc. Proin…</p>
    </div>
    <div dojoType="ContentPane" label="Pane 3">
        <h2>Pane 3</h2>
        <p>Nunc consequat nisi vitae quam. Suspendisse sed nunc. Proin…</p>
    </div>
</div>
</body>
</html>
```

Dojo 的封装体系结构非常强大且值得一看，如果你有志于使用 JavaScript 维护大型的代码库。除此之外，就其库的庞大而论，你也一定能从中发现一些对你有用的功能。

YUI

Yahoo UI(http://developer.yahoo.com/yui/)是另一个维护了巨大的命名空间化封装体系结构的 JavaScript 库。这个库被设计用来为许多常见的 web 应用(如拖放)提供实现或解决方案。所有这些 UI 元素分布于层次结构中。Yahoo UI 的文档相当的出色，其完整和细致值得注意。

与 Dojo 相似，Yahoo UI 也使用了深度的命名空间层次来组织其函数和功能。但是，与 Dojo 不同的是，任何外部代码的"导入"由你来明确地完成，而不是通过导入语句。程序 3-10

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

是一个展示 Yahoo UI 库中命名空间是什么子和怎样工作的例子。

程序 3-10. Yahoo UI 中的封装和命名空间

[Copy to clipboard] [-]

CODE:

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

    <!-- 导入事件包 -->
    <script type="text/javascript" src="event.js"></script>

    <!-- 使用导入的 Yahoo UI 库 -->
    <script type="text/javascript">
        //Yahoo UI 的所有事件和实用功能都包含命名空间 YAHOO 里，
        //并被细分为小一些的命名空间，如'util'
        YAHOO.util.Event.addListener( 'button', 'click', function() {
            alert( "Thanks for clicking the button!" );
        });
    </script>
</head>
<body>
    <input type="button" id="button" value="Click Me!"/>
</body>
</html>
```

无论 Dojo 还是 Yahoo UI 都在于单个大型的封装里组织和维护大量的代码方面有非常杰出的表现。当需要实现你自己的封装架构时，理解它们是怎样使用 JavaScript 命名空间做到这一点的将带来极大的帮助。

清理你的代码

在我开始调试和编写测试案例的话题(这正是下一章要做的)之前，首先检查你是怎么编写代码、把它准备好为其他人所用，这是必不可少的。如果你想要你的代码在历经其他开发者的使用和修改仍能生存，你将需要保证你的代码中没有任何语句能被误会或错误地使用。尽管你也可以手工进行整理代码的工作，使用工具来帮助标记出难以处理的以后可能会出麻烦的代码片段通常更有效率。这正是 JSLint 的用武之地。JSLint 有一系列的内建的规则用来标记出以后可能会带来麻烦的代码片段。在 JSLint 的网站上有一个完整的分析器 http://www.jslint.com/。另外，JSLint 的所有规则和设置可以在这里找到：http://www.jslint.com/lint.html。

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

JSLint 是 Douglas Crockford 开发的另一个工具，体现了他自己的编码风格，因此如果你不喜欢或不太信奉他要求的一些更改，不依它们就是。然而，规则中的一部分的确非常有意义，我将在下面对它们进行特别的说明。

变量声明

JSLint 提出的一个明智的要求是，程序中出现的所有变量都必须在被使用之间声明。尽管 JavaScript 不明确要求你进行变量声明，不这么做可能会导致其实际作用域的混乱。比如说，如果你要为一个没有在一个函数之内声明的变量赋值，那个变量的作用域将是函数内还是全局的？这不是仅凭观察代码就能立即弄清的，因而需要明朗化。JSLint 的变量声明惯例的例子见程序 3-11。

程序 3-11. JSLint 要求的变量量声明

[Copy to clipboard] [-]

CODE:

```
//错误的变量使用
foo = 'bar';
//正确的变量声明
var foo;
…
foo = 'bar';
```

!=与== vs. !==与===

开发者易犯的一个常见的错误是对 JavaScript 中 false 值的缺乏理解。在 JavaScript 里，null，0，""，false，和 undefined 全部彼此相等(==)，因为它们的计算值都为 false。这意味着如果你使用代码 test==false，则它在 test 为 undefined 或 null 时，也会得到结果 true，这可能并非你所期望的。

这正是!==和===有用的地方。这两个操作符都将检查变量的精确值(比如 null)，而不是单看其计算值(如 false)。JSLint 要求你任何时候你使用==和!=进行真假判断时，都必须用!==或===替代。程序 3-12 展示了这些操作符有怎样的不同。

程序 3-12. !=与!==区别于!==和===的示例

[Copy to clipboard] [-]

CODE:

```
//这两个都为 true
null == false
0 == undefined
//你应该以!==和===取代之
null !== false
false === false
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

块与花括号

这是一条我比较难以接受的规则，但是在一个共享的代码环境里，遵照它仍然是有意义的。这条规则背后的前提是，不能使用单行的块。但你有一个子句(如 if(dog==cat))且只有一个语句在其中(dog=false;)，你可以省略通常需要的花括号；这对于 while()和 for()语句的块同样成立。尽管这是 JavaScript 提供的一个很好的便捷，省略代码中的括号对那些没有意识到哪些代码是属于块哪些代码不属于的人来说却可能会导致奇怪的结果。程序 3-13 很好地解释了这一状况。

程序 3-13. 缩进不当的单行代码块

[Copy to clipboard] [-]

CODE:

```
//这是合法的、正常的 JavaScript 代码
if ( dog == cat )
if ( cat == mouse )
mouse = "cheese";
//JSLint 要求它像这样书写
if ( dog == cat ) {
    if ( cat == mouse ) {
        mouse = "cheese";
    }
}
```

分号

最后这一点被证明在下一节中所谈到的代码压缩中是最重要的。在 JavaScript 里，如果你每行书写一条语句的话，语句末的分号是可选的。省略了分号的未压缩代码可能是好的，但是一旦你删除换行符以削减文件尺寸，问题就产生了。为了避免这一点，你应该总是记得在语句的结尾处包含分号，如程序 3-14 所示。

程序 3-14. 需要分号的语句

[Copy to clipboard] [-]

CODE:

```
//如果你打算压缩你的 JavaScript 代码，务必在所有语句的结尾加上分号
var foo = 'bar';
var bar = function(){
    alert('hello');
};
bar();
```

在这分号这一话题里我们最终引接触了 JavaScript 代码压缩的概念。使用 JSLint 编写整

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

洁的代码对其它开发者和你自己有益处，代码压缩却最终是对你的用户最有用的，因为那样他们就能更快地开始使用你的站点。

压缩

分发 JavaScript 库的一个不可缺少的方面是使用代码压缩来节省带宽。压缩应该作为把你的代码投入产品之前的最后一步来使用，因为它会使你代码混乱难以辩认。有三种类型的 JavaScript 压缩器：

简单地删除无关的空白字符和注释，仅保留必须的代码的压缩器；

删除空白和注释，也同将所有的变量名变得改短的压缩器；

删除空白和注释，同时还最小化代码中所有单词(而不仅仅是变量名)的压缩器。

我将要介绍两不同的库：JSMin 和 Paker。JSMin 属第一类压缩器(删除无关的非代码)而 Paker 属第三类(彻底压缩所有单词)。

JSMin

JSMin 的原理很简单。它检查一块 JavaScript 代码并删除所有非必须字符，仅保留纯粹的起作用的代码。JSMin 通过简地删除所有无关的空白字类字符(包括 tab 的换行符)和所有的注释。压缩器的在线版本可以在这里找到：http://www.crockford.com/javascript/jsmin.html。

为了对代码传给 JSMin 以后到底发生了什么有一个直观的感觉，我们举一个简单的代码块的例子（如程序 3-15 所示），传给压缩器，再看看得到的输出结果(程序 3-16 所示)。

程序 3-15. 用来判断用户浏览器的代码

[Copy to clipboard] [-]

CODE:

```
// (c) 2001 Douglas Crockford
// 2001 June 3
// The -is- object is used to identify the browser. Every browser edition
// identifies itself, but there is no standard way of doing it, and some of
// the identification is deceptive. This is because the authors of web
// browsers are liars. For example, Microsoft's IE browsers claim to be
// Mozilla 4. Netscape 6 claims to be version 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'
}
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
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;
}
```

程序 3-16. 3-15 所示程序的压缩版本

[Copy to clipboard] [-]

CODE:

```
var is={ie:navigator.appName=='Microsoft Internet
Explorer',java:navigator.javaEnabled(),ns:navigator.appName=='Netscape',ua:n
avigator.userAgent.toLowerCase(),version:parseFloat(navigator.appVersion.sub
str(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=fal
se;is.opera=true;}if(is.ua.indexOf('gecko')>=0){is.ie=is.ns=false;is.gecko=t
rue;}
```

注意到所有的空白和注释都被删掉了，大大减小了代码的总尺寸。

JSMin 可能是最简单的 JavaScript 代码压缩工具。这是一种很好的在你的产品代码中开始使用压缩的起步方式。然而，当你准备好节约更多的带宽时，你将会想要升级到使用 Paker 这一令人敬畏的、极其强大的 JavaScript 代码库。

Paker

Paker 是目前为止可得到的最强大的 JavaScript 压缩器。它由 Dean Edwards 开发，以一种彻底地削减代码尺寸并在运行时重新扩展和执行的方式工作。通过使用这种技术，Paker 创造出可能的理想地最小化的代码。你可以把它当成是为 JavaScript 代码设计的自解压的 ZIP 文件。该压缩器可用的在线版本见 http://dean.edwards.name/paker/。

Paker 的脚本十分庞大且非常复杂，建议你不要妄图自己去实现它。另外，它生成的代码有几百字节的代码头(使之能够释放其自己身)，因而对于极小的代码它并不理想(JSMin 对此最好一些)。然而，对于大型的文件，它绝对是完美的。程序 3-17 摘自 Paker 所生成的自解压代码。

程序 3-17. Packer 压缩生成代码的一部分

[Copy to clipboard] [-]

CODE:

```
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]
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
||c.toString(a)}k=[(function(e){returnd[e]})];e=(function(){return'\\w+'});c
=1};while(c--){if(k[c]){p=p.replace(newRegExp('\\b'+e(c)+'\\b','g'),k[c])}}r
eturn p}('u 1={5:2.f==\'t
sr\',h:2.j(),4:2.f==\'k\',3:2.l.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(尤其是使用 Paker)压缩你的代码的作用，是不可小视的。具体情况依赖于你的代码是怎么写的，通常你可以将其尺寸减少超过 50%，为你的用户改善页面载入时间，这应该是任何 JavaScript 应用程序的重要目标。

分发

JavaScript 编写过程的最后一步是可选的，这最依赖于你的个别的情况。如果你只是为你自己或者一个公司编写代码，最通常的情况是你简单地把你的代码分发给其它开发者或者上传到你的站点上供使用。

然而，如果你开发了一段有趣的代码并且希望全世界能够任意地使用它，这就是诸如 JavaScript Archive Network(JASN)之类的服务发挥作用的时候了。JSAN 是由几个喜爱 CPAN(Comprehensive Perl Archive Network)的功能的用途的 Perl 开发者发起的。更多关于 JSAN 的消息可以在其站点 http://openjsan.org/找到。

JSAN 要求提交的所有模块都以良好格式化的面向对象风格写成，遵从它特定的体系结构。除了中心代码仓库之外，JSAN 有一套方法，通过它可以导入你的代码所请求的外部 JSAN 模块依赖。这使得编写相互依赖的应用程序变得极其简单，无须担心用户已经安装了哪些模块。为了理解典型的 JSAN 模块是怎样工作的，我们来看一个简单的，DOM.Insert(可从些网址得到 http://openjsan.org/doc/r/rk/rki ... b/DOM/Insert.html)。

这一特定的模块授受一个 HTML 字符串并将它插入到网页中的指定位置。除了出色地面向对象以外，这个模块还引用并加载了两个其它的 JSAN 模块。如程序 3-18 所示。

程序 3-18. 一个简单的 JSAN 模块(DOM.Insert)

[Copy to clipboard] [-]

CODE:

```
//我们将试图使用 JSAN 包含一些其它的模块
try {
    //载入两所需的 JSAN 库
    JSAN.use( 'Class' )
    JSAN.use( 'DOM.Utils' )

    //如果 JSAN 没有被加载，将抛出异常
    } catch (e) {
        throw "DOM.Insert requires JSAN to be loaded";
}
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
//确保 DOM 命名空间存在
if ( typeof DOM == 'undefined' )
    DOM = {};
//创建一个新的 DOM.Insert 构造器，继承自"Object"
DOM.Insert = Class.create( 'DOM.Insert', Object, {
    //接受两个参数的构造器
    initialize: function(element, content) {
        //向其中插入 HTML 的元素
        this.element = $(element);

        //欲插入的 HTML 字符串
        this.content = content;

        //尝试使用 IE 的方式插入 HTML
        if (this.adjacency && this.element.insertAdjacentHTML) {
            this.element.insertAdjacentHTML(this.adjacency, this.content);
        //否则，使用 W3C 方式
        } else {
            this.range = this.element.ownerDocument.createRange();
            if (this.initializeRange) this.initializeRange();
            this.fragment = this.range.createContextualFragment(this.content);
            this.insertContent();
        }
    }
});
```

编写清晰的面向对象的、易交互的 JavaScript 代码的能力应该是你的开发活动的品质证明。正是基于这一方式，我们将建造和探究 JavaScript 语言的其它部分。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，变得更加的有用和盛行。

本章摘要

在本章中你看到了建造可重用代码结构的几种不向方法。运用你在前面章节所学的面向对象技术，你有能够利用它们创建最适合多开发者环境的干净的数据结构。另外，你看到了创建可维护的代码、削减 JavaScript 文件大小和为分发而封装代码的最好的几种方法。了解了怎样编写漂亮地格式的可维护的代码将会为你省去无数次的挫败。

程序 3-2.中的第二段代码显然是说不通的。（好像录入不完整的样子？）

在 Douglas Crockford 的网页（http://javascript.crockford.com/inheritance.html）上这三个函数函数是这样的：

[Copy to clipboard] [-]

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

CODE:

```
Function.prototype.method = function (name, func) {
    this.prototype[name] = func;
    return this;
};
Function.method('inherits', function (parent) {
    var d = {}, p = (this.prototype = new parent());
    this.method('uber', function uber(name) {
        if (!(name in d)) {
            d[name] = 0;
        }
        var f, r, t = d[name], v = parent.prototype;
        if (t) {
            while (t) {
                v = v.constructor.prototype;
                t -= 1;
            }
            f = v[name];
        } else {
            f = p[name];
            if (f == this[name]) {
                f = v[name];
            }
        }
        d[name] += 1;
        r = f.apply(this, Array.prototype.slice.apply(arguments, [1]));
        d[name] -= 1;
        return r;
    });
    return this;
});
Function.method('swiss', function (parent) {
    for (var i = 1; i < arguments.length; i += 1) {
        var name = arguments[i];
        this.prototype[name] = parent.prototype[name];
    }
    return this;
});
```

遗憾的是 Douglas Crockford 的代码其实并没有达到他自己所期望的效果，原因是靠 apply 传递 this 参数的方法不能确保父层的函数得到正确的上下文。

显然 Douglas Crockford 和本书作者 John Resig 都没有对这些代码进行过充分的测试。

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

下面的测试代码暴露了问题所在。

```
<script>
Function.prototype.method = function (name, func) {
    this.prototype[name] = func;
    return this;
};
Function.method('inherits', function (parent) {
    var d = {}, p = (this.prototype = new parent());
    this.method('uber', function uber(name) {
        if (!(name in d)) {
            d[name] = 0;
        }
        var f, r, t = d[name], v = parent.prototype;
        if (t) {
            while (t) {
                v = v.constructor.prototype;
                t -= 1;
            }
            f = v[name];
        } else {
            f = p[name];
            if (f == this[name]) {
                f = v[name];
            }
        }
        d[name] += 1;
        r = f.apply(this, Array.prototype.slice.apply(arguments, [1]));
        d[name] -= 1;
        return r;
    });
    return this;
});
function g0()
{}
function g1()
{}
function g2()
{}
function g3()
{}
function g4()
{}
g1.inherits(g0)
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
g2.inherits(g1)
g3.inherits(g2)
g4.inherits(g3)
g0.prototype.say=function(){return "我是亚当"}
g1.prototype.say=function(){return this.uber('say')+"的孩子 1"}
g2.prototype.say=function(){return this.uber('say')+"的孩子 2"}
g3.prototype.say=function(){return this.uber('say')+"的孩子 3"}
g4.prototype.say=function(){return this.uber('say')+"的孩子 4"}
si=new g4()
alert(si.say())
</script>
```

如果设法避免上下文的使用，则可以得到正确的结果。

```
<script>
Function.prototype.method = function (name, func) {
    this.prototype[name] = func;
    return this;
};
Function.method('inherits', function (parent) {
    var d = {}, p = (this.prototype = new parent());
    this.method('uber', function uber(name) {
        if (!(name in d)) {
            d[name] = 0;
        }
        var f, r, t = d[name], v = parent.prototype;
        if (t) {
            while (t) {
                v = v.constructor.prototype;
                t -= 1;
            }
            f = v[name];
        } else {
            f = p[name];
            if (f == this[name]) {
                f = v[name];
            }
        }
        d[name] += 1;
        r = f.apply(this, Array.prototype.slice.apply(arguments, [1]));
        d[name] -= 1;
        return r;
    });
    return this;
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
});
function g0()
{}
function g1()
{}
function g2()
{}
function g3()
{}
function g4()
{}
g1.inherits(g0)
g2.inherits(g1)
g3.inherits(g2)
g4.inherits(g3)
g0.prototype.say=function(){return "我是亚当"}
g1.prototype.say=function(){return g1.prototype.uber('say')+"的孩子 1"}
g2.prototype.say=function(){return g2.prototype.uber('say')+"的孩子 2"}
g3.prototype.say=function(){return g3.prototype.uber('say')+"的孩子 3"}
g4.prototype.say=function(){return this.uber('say')+"的孩子 4"}
si=new g4()
alert(si.say())
</script>
```

第四章：调试和测试工具

或许在用任何编程语言进行开发时最耗时的一个环节都是测试和调试代码。对于专业级的代码来说，确保你所编写的东西是经过完全测试的、可验证和零 bug 的，变得极度重要。使得 JavaScript 与其它编程语言如此不同的一个方面是，它并不被任何单独的公司或组织单独拥有和支持（不像 C#,PHP,Perl,Python,或者 Java)。这一差异使得拥有一个调试和测试代码的一致的基础变得极具挑战性。

为了削减你可能必须忍受的工作量和压力，捕获 JavaScript 错误时，有许多强大的开发工具可供使用。每一种现代浏览器都有各自的(质量参差的)工具。使用它们会使得 JavaScript 开发呈现加一致的局面和更加远大的前景。

本章中我讨论用于调试 JavaScript 代码的不同的工具，然后建立可靠的、可重用的用以检验未来开发的测试套件。

调试

测试和调试是不可分离的两个环节。当你在为你的代码建立广泛的测试用例的时候，你肯定

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

将会碰到一些奇怪的需要注意的错误。这正是调试环节发挥功用的地方。了解怎样使用可用的最好的工具来找到并修复你程序中的缺陷，可以使你的代码有保障且更快地运行。

错误控制台

大多数现代浏览器里都可用的最易使用的工具是某种形式的错误控制台。控制台的质量、界面的可访问性以及错误消息的质量，在不同的浏览器之间都有很大差异。最终，你可能会现最好使用某种单独的拥有最适合开发者的错误控制台(或其它用于调试的插件)的浏览器来开始你的调试过程。

Internet Explorer

拥有最受欢迎的浏览器并不意味着就拥有了最好的调试工具。不幸的是，IE 的错误控制台有着严重的不足。问题之一就是，控制台默认是关闭的，如果你不把 IE 用作缺省浏览器(十分怀疑任何有自尊心的 JavaScript 开发者会这么干)，这一点会使得错误的搜寻变得更加令人困惑。

除了上面所提到的可用性，IE 的错误控制台的最麻烦的问题是以下几点：

a. 每次只显示一个错误；你必须依赖菜单系统来找到其它错误。

b. 错误消息含义相当模糊，几乎没有逻辑意义。它们很少给出发生的问题的精确描述。

c. 报告的错误总是偏移一行，也就是说实际出错的行号总是比所报告的小一。这一点与含糊的错误结合起来，你可能会饱受 bug 之苦。

图 4-1 是发生错误时的 IE 错误控制台的例子。

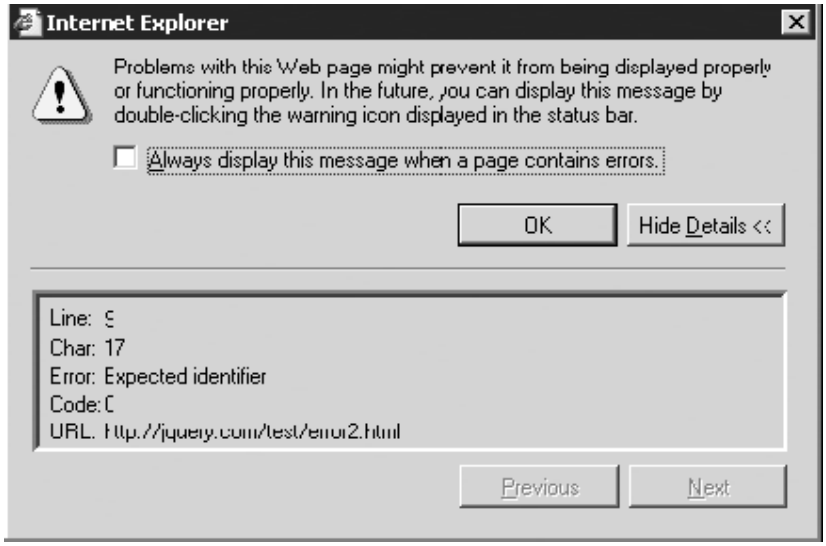


图 4-1.IE 中的 JavaScript 错误控制台

正如我在本节开头所提到的，在另一个(非 IE)浏览器中开始你的 JavaScript 调试过程或许是一个非常好的主意。一旦你在那种浏览器中彻底清除了所有 bug，你应该可以更容易地定位出现在 IE 里的错综复杂的问题。

Firefox

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

过去的几年里 Firefox 浏览器在 UI 方面取得了极大的发展，极大地帮助着 web 开发者更加轻松地开发更好的网站。JavaScript 错误控制台历经了多次更新，产生了一些很有用的东西。

a. 控制台允许你输入任意的 JavaScript 命令。这一点用于断定页面载入以后变量是什么值是极其有用的。

b. 控制台提供了依据其类型(如错误、警告或者消息)将消息分类的能力。

c. 最新版的控制台连同 JavaScript 错误一起提供了样式表警告。尽管在设计拙劣的网站上它可能会提供大量的不必要的错误消息，但是通常在找出你自己的布局缺陷时它还是很有帮助的。

d. 该错误控制台的一个缺点是它并不根据你正在浏览哪一个页面来过滤消息，这就是说你会得到不同页面的错误的混合。(下节我要讲到的 Firebug 插件，解决了这一问题。)

图 4-2 所示是 Firefox 错误控制台一个截屏。注意你可以使用不同的按钮在不同类型的错误消息之间切换。

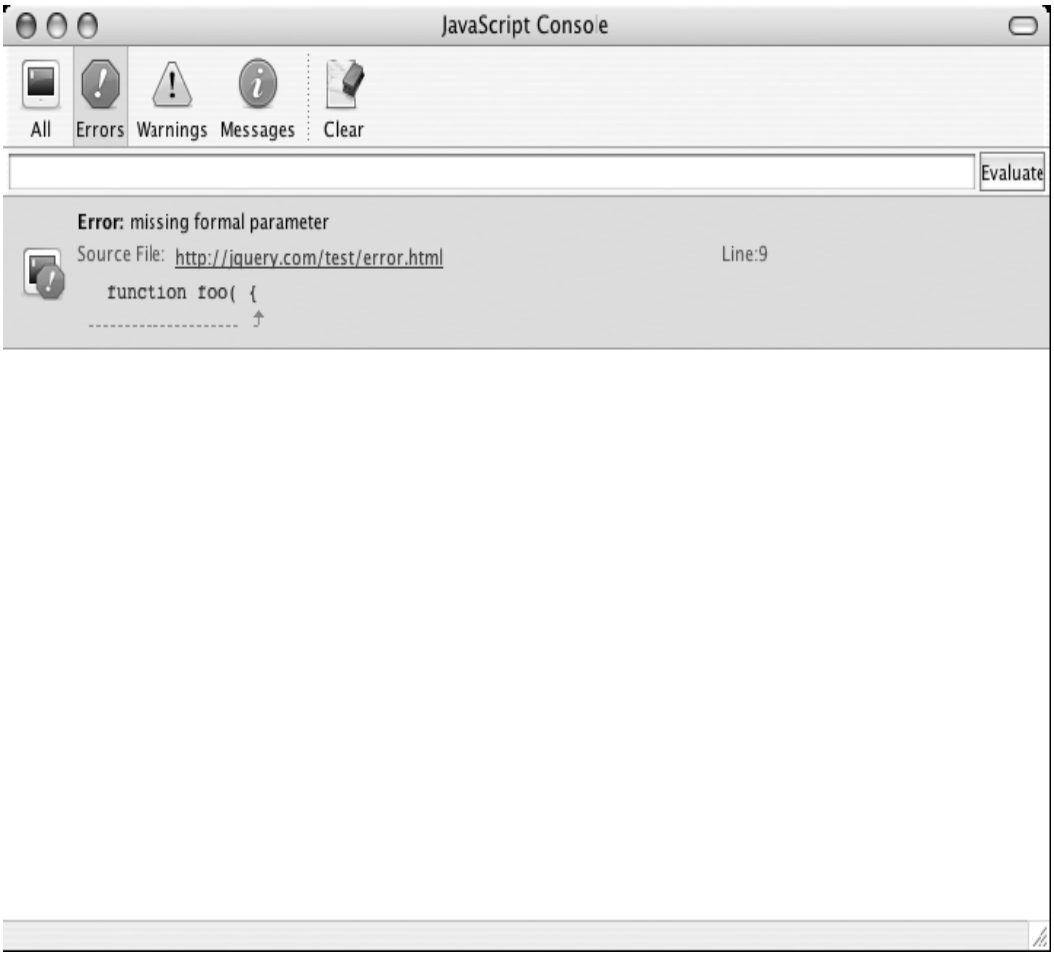


图 4-2. Firefox 的 JavaScript 错误控制台

虽说 Firefox 的错误控制台非常好，但它并不是完美的。正是因为如此，开发者们趋向

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

于求助于各种 Firefox 插件来更好地调试他们的应用程序。稍后我将讨论这些插件。

Safari

Safari 浏览器是市场上最新的浏览器之一，也是成长相当快的一个。这一成长导致其 JavaScript 支持(无论是开发还是执行时)都有时很不稳定。出于此，在该浏览器里 JavaScript 控制台不易进入。它甚至不是一个容易激活的选项，完全隐藏在一个一般用户不可访问的秘密的调试菜单里。

为了激活调试菜单(JavaScript 控制台)你需要(在 Safari 没有运行时)在一个终端上执行如 1-4 所示的命令。

程序 4-1. 让 Safari 显示调试菜单的的命令

[Copy to clipboard] [-]

CODE:

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

下一次打开 Safari 时，你就有了一个新的包含 JavaScript 控制台选项的调试菜单。

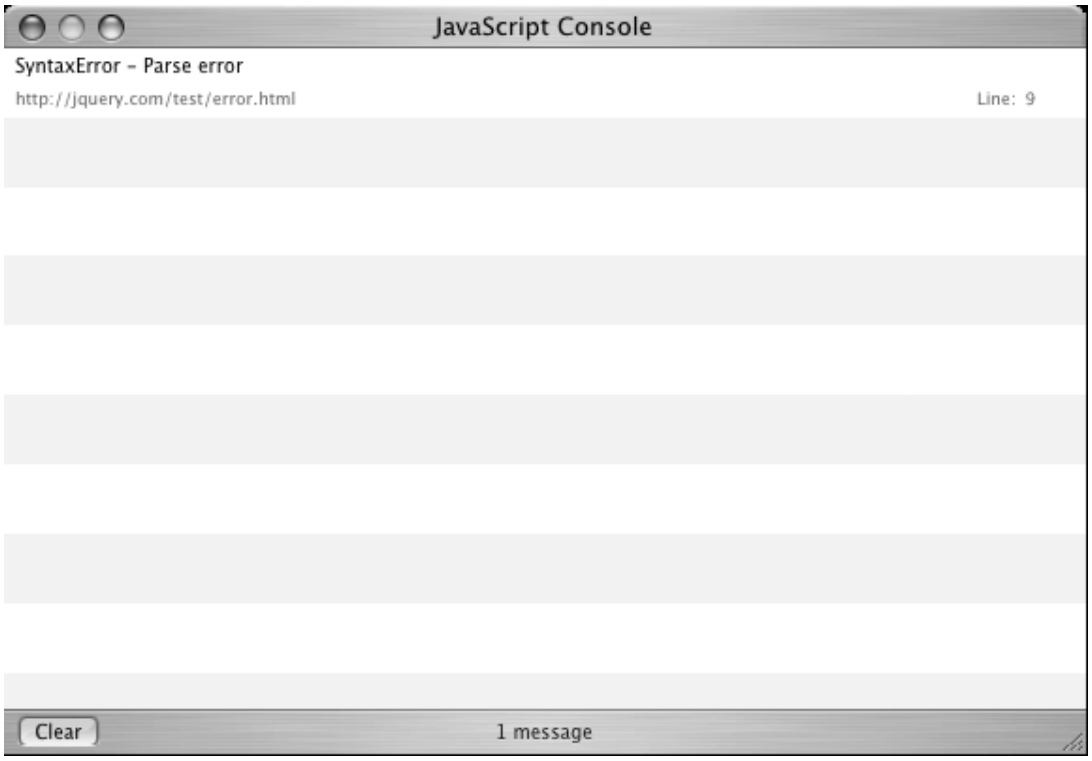
如同你可以从其隐密的位置推测的那样，这个控制台还处于一个很原始的状态。关于 Firefox 错误控制台需提及的有以下几点：

a. 错误消息通常十分含糊，大致跟 IE 的错误消息在同一个级别上。

b. 提供了错误所在的行号，但是经常被重置为零，把你扔回你开始的地方。

c. 没有根据页面将消息过滤，不过所有的消息都将抛出该错误的脚本列于其后。

图 4-3 展示了运行于 Safari2.0 上的错误控制台的一个截屏。



PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

图 4-3. Safari 的 JavaScript 错误控制台

作为一个 web 开发平台，Safari 仍然落后得很远。但是，WebKit 开发组(开发了 Safari 渲染引擎的团队)正致力于把该浏览器推向前沿并取得了良好的进展。期待未来的几个月和几年里该浏览器的会有很多新发展。

Opera

最后我们来看看 Opera 浏览器包含的错误控制台。值得感激的是，Opera 投入了大量的时间和精力，使得它功能丰富且非常有用。除了 Firefox 控制台所具有的所有特性之外，它还提供了以下几点：

a. 描述性的错误消息，给你对问题的良好理解

b. 内联的代码片段，用代码本身来说明错误出在哪里

c. 错误可以根据类型(如，Javascript,CSS，等等)过滤

遗憾的是，这个错误控制台没有执行 JavaScript 命令的能力（译注：这句及以上几句着实让人郁闷，明明说了除 firefox 所具有的还怎么地怎么地，列举的三条里倒有两条是 firefox 已经有的，到现在又搞出个 firefox 有而它没有的了），而那是如此有用的一个功能。尽管如此，这些加起来，已经构成一个很好的错误控制台了。图 4-4 展示了 Opera9.0 中错误控制台的一个截屏。

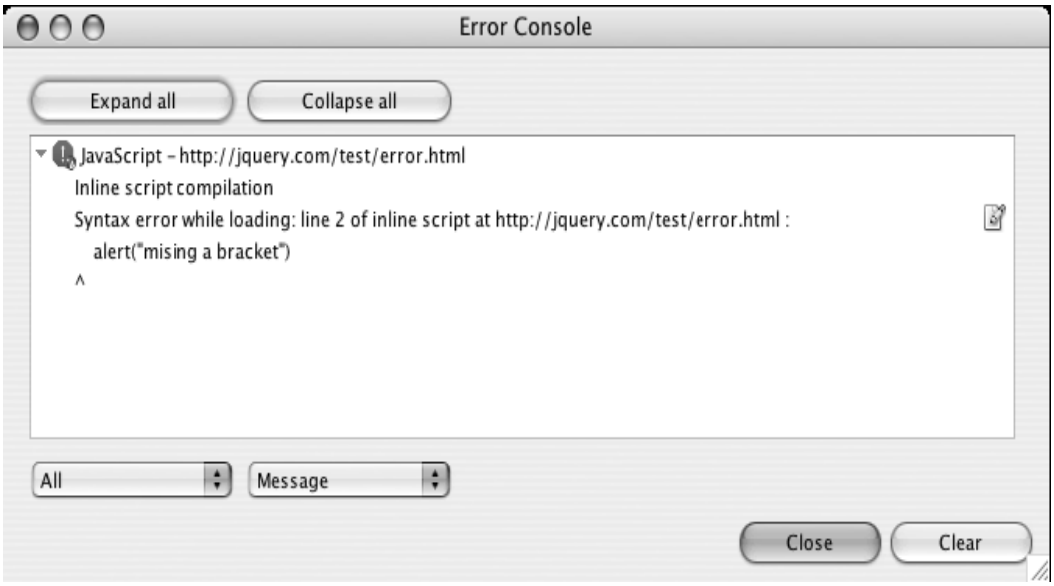


图 4-4.Opera 的 JavaScript 错误控制台

Opera 长期以来一直认真地对待 web 开发。开发队伍拥有众多主动而干劲十足的开发者和文档编写者，Opera 平台一直在为更好地服务于 web 开发者而奋斗。

DOM 查看器

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

DOM 查看是对 JavaScript 开发者最有用的而没有被充分利用的工具之一。DOM 查看可以看成是作页面源代码查看的一个高级版本，允许你看到经过你的代码修改其内容以后页面的当前状态。

每个浏览器里不同的 DOM 查看器行为各异，有些提供了额外的功能，允许你深入地观察你正操作的对象。这一节里我将介绍三个不同的浏览器并讨论是什么使它们彼此有那么大的差异。

Firefox 的 DOM 查看器

Firefox 的 DOM 查看器是一个预包含在所有 Firefox 安装包里的 Firefox 插件(但默认是不安装的)。这一插件允许你在 HTML 文档建立起来及被操作以后在其中导航。此插件的一个截屏见图 4-5。

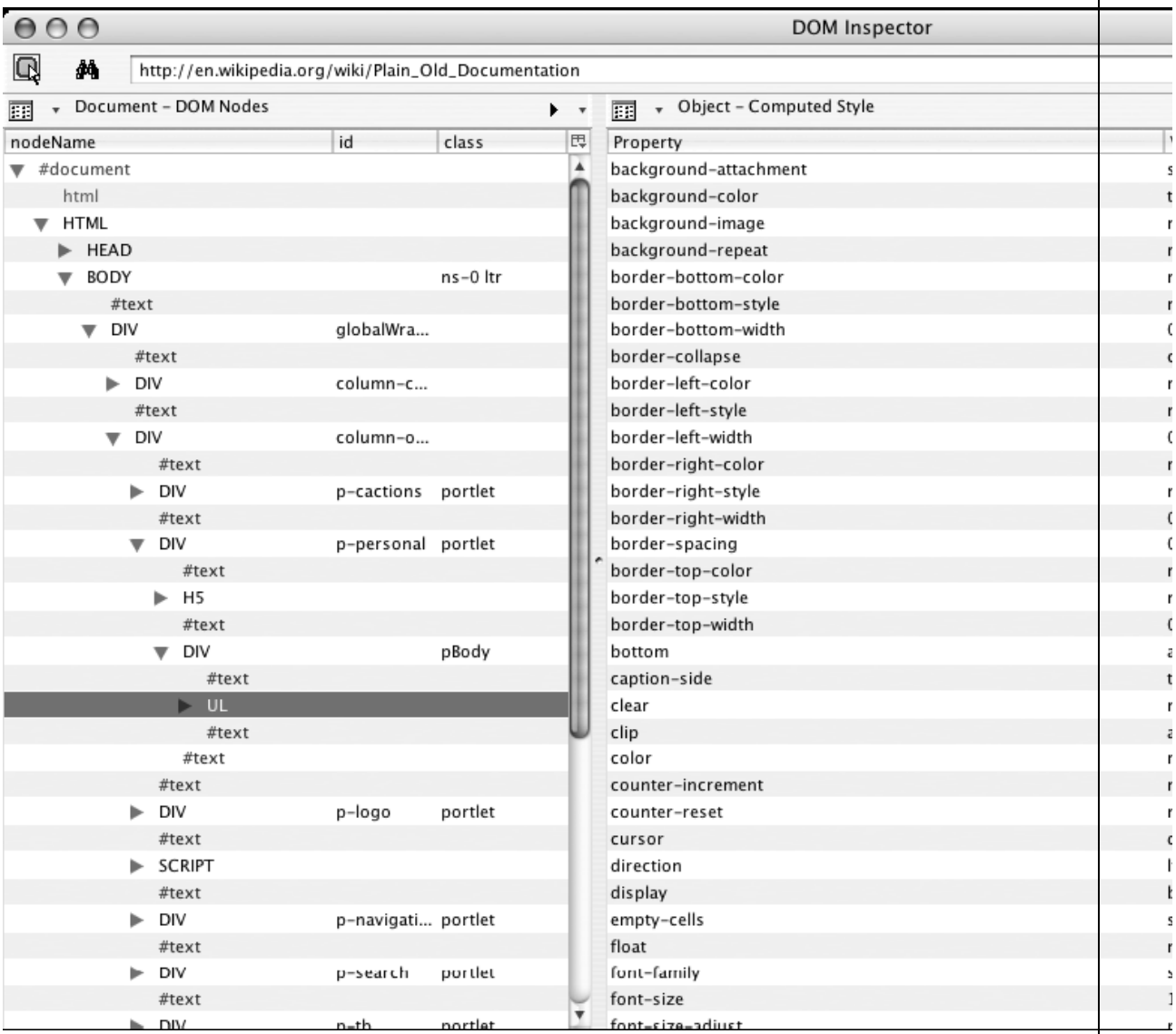


图 4-5. Firefox 的内建 DOM 查看器

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

在导航一个文档的时候，你不仅能够看到修改后的 HTML 元素的结构，还能看到每一个元素的 style 属性以及它们的实体对象属性。这能帮助你确切地知道修改过以后网页看起来是什么样子感觉怎么样。这使得查看器成为一个必不可少的工具。

Safari 的 web 查看器

Safari 有一个新的 DOM 查看器包含于其浏览器最近的版本里。在某些方面它比 Firefox 的 DOM 查看器还要好，你可以右击页面的任何元素使查看器立即导航到它。图 4-6 展示了(设计得非常雅致的)Safari DOM 查看器的一个截屏。

Safari 的内建 DOM 查看器

尽管这一插件包含于 Safari 的最近版本里，但激活它甚至比前面提及的 JavaScript 控制台更麻烦。这事实在让人摸不透：为什么 Safari 团队付出了那么多的努力去编写和加入这些部件，最后却又对希望使用它们的开发者隐藏起来。且不管这些，为了激活那个 DOM 查看器，你必须执行如程序 4-2 所示的语句。

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

程序 4-2. 激活 Safari 的 DOM 查看器

[Copy to clipboard] [-]

CODE:

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

Safari 的 DOM 查看器仍有许多地方需要发展和改进，好在 Safari 的开发团队是很有才能的。但是目前来说，你可能最好还是使用 Firefox 作为你的开发的基础，直到 Safari 彻底完成和发布。

View Rendered Source

最后，我要介绍 Web 开发者可用的最易使用的 DOM 查看器。Firefox 插件 View Rendered Source 在通常的查看看源代码选项下面加入一个供选择的菜单项，以一种直观的可理解的方式为你提供新的 HTML 文档的完整表述。关于该插件的更多信息可在其网站上找到：http://jennifermadden.com/。

除了提供一个感觉非常自然的源代码视图以外，它还为文档的每一层提供了分级的代码着色，让你更好的感觉到你到底位于代码的哪一处。如图 4-7 所示。

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

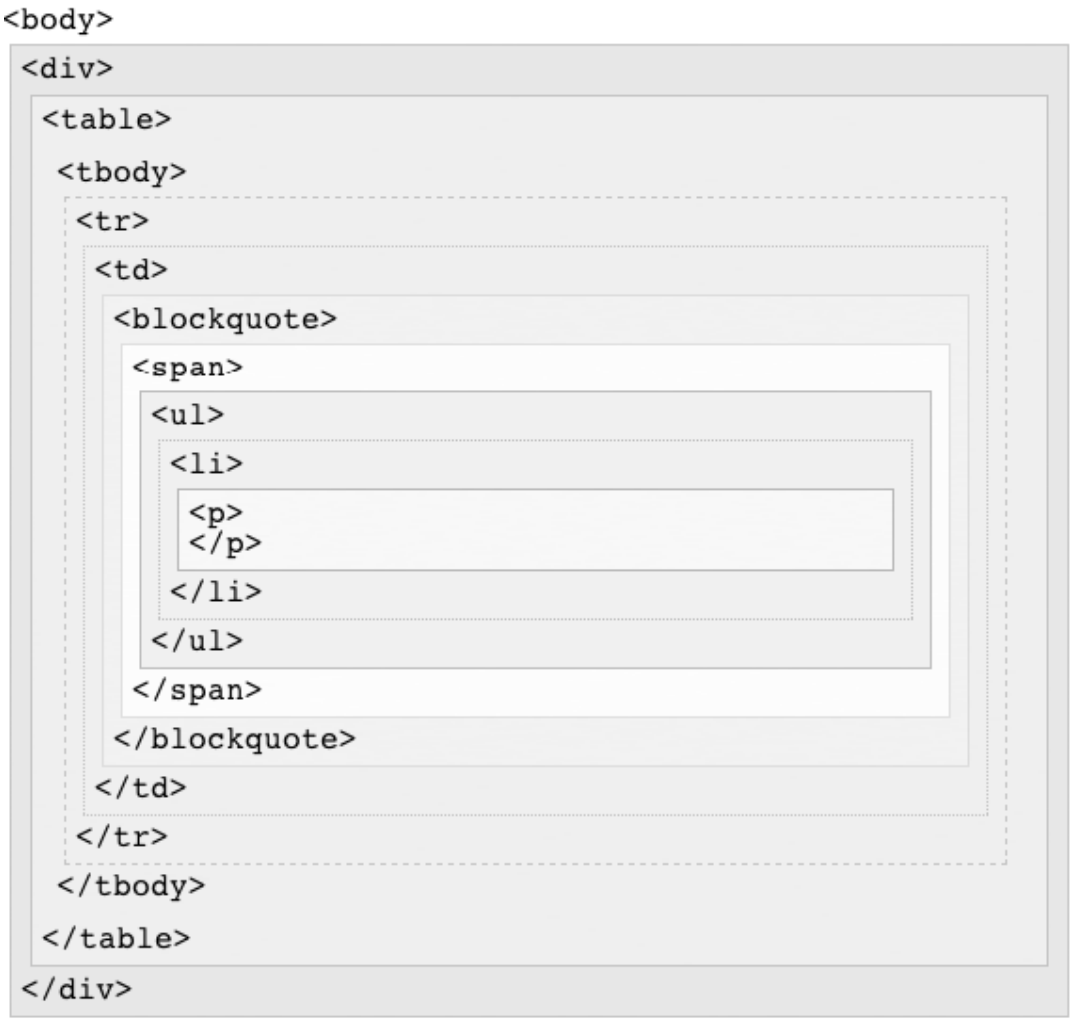


图 4-7. Firefox 的插件 View Rendered Source

View Rendered Source 插件应该是每一个 web 开发者工具箱的标准工具。它的基本用途远远超越了原始的源代码查看所给出的，同时仍允许向更复杂的 Firefox DOM 查看器插件的平滑升级。

Firebug

Joe Hewitt 创造的 Firebug 是近年来出现的最重要的 JavaScript 开发插件之一。作为一个完整的 JavaScript 开发包，它有一个错误控制台，一个调试器，和一个 DOM 查看器。关于此插件的更多信息可以从它的网站上找到：http://www.joehewitt.com/software/firebug/。

将这么多的工具结合起来的一个最主要的优点就是，你可以更好的推断出问题出在哪

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

里。比如说，当点击一条错误消息的时候，JavaScript 文件名及错误发生的行号将会呈现给你。于是你可以设置断点，介入代码的执行，更好地把握问题出现的来龙去脉。此插件的一个截屏见图 4-8。

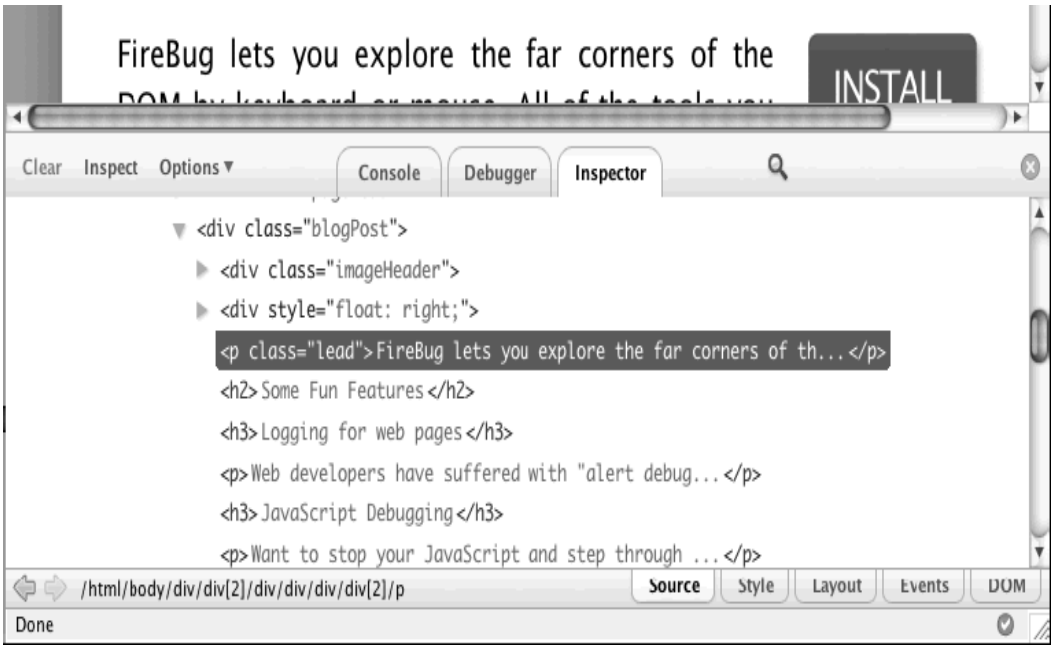


图 4-8. Firebug 调试插件

现代调试工具发展至今，还没有出现比 Firebug 更好的。我强烈推荐你选择 Firefox 作为你的基础 JavaScript 开发平台，并联合使用 Firebug 插件。

Venkman

The last piece of the JavaScript development puzzle is the Venkman extension(不知从何 puzzle 起)。最初作为 Mozilla 浏览器的一部分，Verkman 是由 Mozilla 发起的 JavaScript 调试器项目的代码名称。关于此项目的更多信息和更新了的 Firefox 插件可以在以下网站找到：

Mozilla 的 Venkman 项目：http://www.mozilla.org/projects/venkman/

Firefox 的 Venkman 插件：https://addons.mozilla.org/firefox/216/

Venkman 教程：http://www.mozilla.org/projects/venkman/venkman-walkthrough.html

使用这样一种插件的重要性在于，由于与 JavaScript 引擎本身的深入结合，它能够让你对代码到底在做些什么进行更高级的控制。图 4-9 是 Firefox 的 Vernkman 插件的一个截屏。

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

图 4-9. 与 Firefox 接口的历史悠久的 JavaScript 调试器 Venkman

利用这一插件引入的所有额外控制，除了能进入代码分析其执行过程以外，你还可以确切地知道在一个作用域里什么变量对你是可用的以及有关属性或变量状态的确切信息。、

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

测试

个人来说，我把测试的过程和测试用例的建立看作是"[color]future-proofing(没想出合适的词表达它)"你的代码。为你的基础代码或库创建可信赖的测试用例，可以为你省下无数用来调试代码和寻找你在调试时不经意引入的怪异的 bug 的时间。

作为这是多数现代编程环境共有的惯例，拥有一套可靠的测试用例，不仅能帮助你自己、同时也能帮助其它使用你的代码库的人添加新的功能并修复错误。

在这一节里我将介绍能用来建立 JavaScript 测试套件的三种不同的库，它们都能以跨浏览器的、自动化的方式执行。

JSUnit

JSUnit 长期以来几乎一直是 JavaScript 单元测试的黄金标准。它的大多数功能基于为 Java 设计的流行的 JUnit 包，这意味着如果你熟知 JUnit 是怎样通过 Java 工作的，你将能轻易使用此库。它的网站(http://www.jsunit.net/)上有大量的可用的信息和文档(http://www.jsunit.net/documentation/)。

跟大多数(至少是我这一节所讨论的全部)测试套件一样，它由三个基本部件组成：

测试运行器(Test runner)：套件的这一部分提供一个良好的图形化输出，显示当前已经运行到全部操作的哪一个阶段。它提供了载入测试组和并行其内容的能力，记录它们提供的所有输出。

测试组(Test suite)：这是所有测试用例（有时分布于多个网页）的集合。

测试用例(Test cases)：这是一些独立的可以求值到 true/false 表达式的命令，给你可计量的结果来判定你的代码是否正确地工作。单独的一个测试用例可能不是太有用，但是当配合测试运行器一起使用的时候，你将会得到有用的交互式的体验。

所有这些加起来，构成了全面的自动的测试套件，可用来运行和加入将来的测试。程序 4-3 展示了一个简单的测试组，程序 4-4 则是套测试用例。

代码 4-3. 使用 JSUnit 建立的测试组

[Copy to clipboard] [-]

CODE:

```
<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>
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

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

程序 4-4. JSUnit 里可用于典型的测试页的各种测试用例

[Copy to clipboard] [-]

CODE:

```
<html>
<head>
<title>JsUnit Assertion Tests</title>
<script src="../app/jsUnitCore.js"></script>
<script>
//测试一个表达式为真
function testAssertTrue() {
    assertTrue("true should be true", true);
    assertTrue(true);
}
//测试一个表达式为假
function testAssertFalse() {
    assertFalse("false should be false", false);
    assertFalse(false);
}
//检查两个参数是否相等的测试
function testAssertEquals() {
    assertEquals("1 should equal 1", 1, 1);
    assertEquals(1, 1);
}
//检查两个参数是否不相等的测试
function testAssertNotEquals() {
    assertNotEquals("1 should not equal 2", 1, 2);
    assertNotEquals(1, 2);
}
//检查参数是否为 null 的测试
function testAssertNull() {
    assertNull("null should be null", null);
    assertNull(null);
}
//检查参数不为 null 的测试
function testAssertNotNull() {
    assertNotNull("1 should not be null", 1);
    assertNotNull(1);
}
</script>
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

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

为 JSUnit 的编写的文档非常好，而且因为它已经出现很长一段时间了，你很有希望有找到很好的实际应用的例子。

J3Unit

J3Unit 是 JavaScript 单元测试领域的新兵。这一特别的库所提供的超越于 JSUnit 的功能在于，它能直接与服务器端的测试套件(如 JUnit 或 Jetty)溶合。对 JavaScript 开发者来说，这可能是极其有用的，因为他们能够同时为他们的客户端和服务器端代码快速地遍历所有的测试用例。然而，由于不是所有的人都使用 Java，J3Unit 也提供了一个静态模式，可以像其它的单元测试器一样执行于你的浏览器中。关于 J3Unit 的更多信息可以在它的网站上找到：http://j3unit.sourceforge.net]。

因为将服务器端代码与客户器端的测试用例挂勾是很罕见的特例，我们来看看 J3Unit 的静态客户端测试单元是如何工作的。可喜的是，它们实际是跟其它的测试套件没有什么不同，这使得移植非常简单。如程序 4-5 的代码所示。

程序 4-5. 使用 J3Unit 执行的简单测试

[Copy to clipboard] [-]

CODE:

```
<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({
    //测试显示和显示元素
    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");
    }},

    //测试将一个元素添加到另一个里
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
    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>
```

JSUnit,尽管还非常新，却已经展现了单元测试框架的光明前景。如果你对它的面对对象风格感兴趣，我推荐你试试。

Test.Simple

JavaScript 单元测试的最后一个例子是另一个新来者。Test.Simple 由 JSAN 的创建所引入，作为一种方式来标准化所有提交的 JavaScript 模块的测试。因为它的广泛应用，Test.Simple 拥用大量的文档和使用实例，这是使用一个测试框架时非常重要的两个方面。关于 Test.Simple（及其姊妹库 Test.More)的更多信息可以在这里找到：

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 库提供了大量的用来调试的方法，以及一个完整的测试运行器，提供自动化的测试执行。程序 4-6 是一个 Test.Simple 测试组的示例。

程序 4-6. 使用 Test.Simple 和 Test.More 执行测试

[Copy to clipboard] [-]

CODE:

```
//加载 Test.More 模块(用来测试它自身)
new JSAN('../lib').use('Test.More');
//计划发生 6 个测试(以便出问题的时候知道)
plan({tests: 6});
//测试 3 个简单的案例
ok( 2 == 2, 'two is two is two is two' );
is( "foo", "foo", 'foo is foo' );
isnt( "foo", "bar", 'foo isnt bar');
//使用正则表达的测试
like("fooble", /^foo/, 'foo is like fooble');
like("FooBle", /foo/i, 'foo is like FooBle');
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
like("/usr/local/", '^\/usr\/local', 'regexes with slashes in like' );
```

个人而言，我比较欣赏 Test.Simple 和 Test.More 的简单性，因为它们不会太多常规开销，有助于保持你的代码简洁。当然归根结底，决定使用哪种最适合你的测试套件还得看你自己，为你的代码选择测试套件是太过重要的而不可忽略的一个话题。

本章摘要

尽管这一章里出现的东西对于经验丰富的程序员来说没有什么新鲜的，但将这些概念与 JavaScript 的使用结合起来将大大提高 JavaScript 合为一种专业的编程语言的可用性和境界。我强烈推荐你尝试一下调试和测试环节。我确信它会用助于你编写出更好、更干净的 JavaScript 代码。

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

第五章：文档对象模型

在过去十年里 web 开发所取得的所有进步当中，DOM(文档对象模型)脚本是开发者可用来改进其用户体验质量的最重要的技术。

使用 DOM 脚本向页面加入非侵入的 JavaScript（意味着它不会与不支持的浏览器或禁用了 JavaScript 的用户发生冲突)，你将能提供各种你的用户可享受的现代浏览器的增强功能，同时又不会损害那些不能利用它们的用户。这么做的一个副作用是，你的所有代码最终都可以被很好的分离和更容易地管理。

可喜的是，所有的现代浏览器都支持 DOM 并额外地支持一个当前 HTML 文档的内建的 DOM 表述。所有这些都很容易通过 JavaScript 访问，这为现代 web 开发者带来巨大的利益。理解怎样使用这一技术和怎样最好地发挥它的功效，能够给这你开发下一个 web 应用程序的提供一个良好的开端。

本章中我将讨论与 DOM 相关的一些话题。考虑到你可能对 DOM 没有经验，我将从基础出发，涵盖所有的重要概念。对于已经熟悉了 DOM 的读者，我保证将会给出一些你肯定会喜欢并开始在自己的页面中使用的很酷的技术。

文档对象模型简介

DOM 是由 W3C 制定的表示 XML 文档的标准方式。它未必是最快的、最轻便的、或者最易使用的，却是是最普及的，绝大多数 web 开发语言（如 Java，Perl，PHP，Ruby，Python，及 Javascript)都实现了对它的支持。DOM 旨在为开发者提供一种直观的方式来导航于 XML 的层次结构中。即使你并不完全熟悉 XML，你也会非常高兴地看到所有的 HTML 文档(在浏览器的眼中也就是 XML 文档)都有一个可供使用的 DOM 表述。

导航 DOM

DOM 中描述 XML 结构的方式是作为一种可导航的树。使用的所有术语与一个家族树(parents,children,sibling,等等)是近似的。与典型的家族树不同的是，XML 文档以单个包含指向其子节点的指针的根节点（称为文档元素(*document element*)）开始。每一个子结节又包括指回其父结节、兄弟结点和子节点的指针。

DOM 使用特定的术语来代表 XML 树中的各种对象。DOM 树中的每一个对象都是一个*节点(node)[/i]。每个节点可以拥有不同的[i]类型(type)*，如元素(element)，文本(text)，或文档(document)。为了继续，我们需要等来了解 DOM 文档是什么样子的以及怎样在其中导航(一旦它已经构建完成)。通过一段简单的 HTML 片段，我们来考察这一 DOM 构建工作是怎样进

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

行的。

[Copy to clipboard] [-]

CODE:

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

这个片断的每一部分被分解成一个带有指向基直接亲属(父、子、兄弟)的指针的 DOM 节点。如果完全描绘出存在的关系，它将会是类似于图 5-1。片段的每一部分(圆角盒子代表元素，方盒子代表文本节节点)与它所有的引用一起显示。

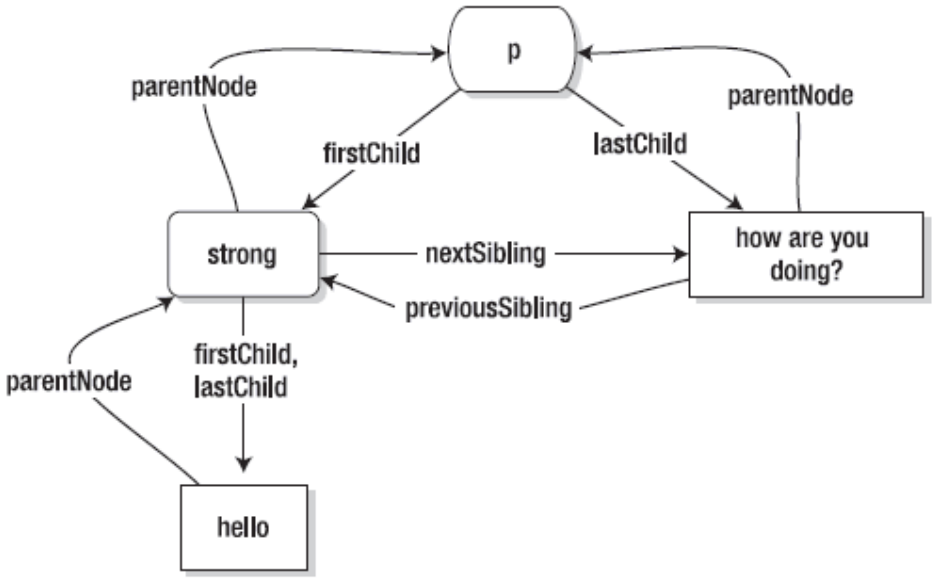


图 5-1. 节点间的关系

每个 DOM 节点都包含一个指针的集合，它使用这些指针引用其亲属。你将使用这些指针来学习怎样导航 DOM。图 5-2 显示了所有可用的指针。这些属性对每一个 DOM 节点都可用，是指向其它 DOM 元素的指针（或者是 null，如果不存在对应元素的话）。

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

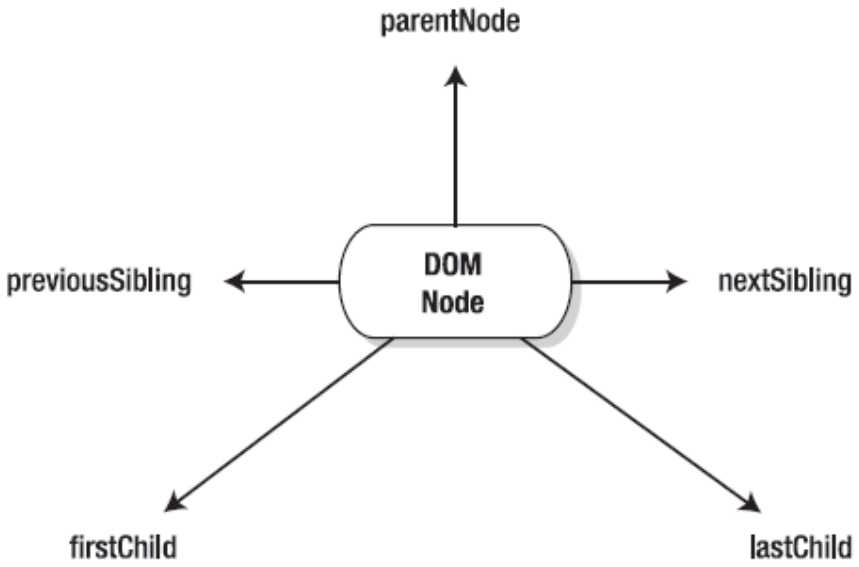


图 5-2. 使用指针导航 DOM

仅使用指针而导航到页面的任何元素元素和文本块是可能的。理解在现实环境里这一点怎样工作的最好的方式是来看一个普通的 HTML 页面，如程序 5-1 所示：

程序 5-1. 一个简单的 HTML 网页，兼一个简单的 XML 文档

[Copy to clipboard] [-]

CODE:

```
<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>
```

在这个示例文档中，根元素是<html>元素。在 JavaScript 访问这一个根元素是很轻松

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

的：

[Copy to clipboard] [-]

CODE:

```
document.documentElement
```

如同它的 DOM 节点一样，根结点拥有用来导航的所有指针。使用这些指针你就能开始浏览整个文档，导航到你想要的任何元素。例如，要得到<h1>元素，你可能使用以下语句：

[Copy to clipboard] [-]

CODE:

```
//Don't work!
document.documentElement.firstChild.nextSibling.firstChild
```

我们恰好撞上了我们的第一个暗礁：DOM 指针既能指向文本节点也能指向元素。于是，上面的语句实际并不是指向<h1>元素；它反倒指向<title>元素。为什么会出这种事呢？这要归咎于 XML 的最棘手和最受争议的一个方面：空白(white space)。你可能会注意到的，在<html>和<head>元素之间，实际上有一个换行符，它被认为是空白，这意味着那里实际上首先有一个文本节点，而不是<head>元素。我们从中可以学到三件事：

1. 当尝试只使用指针导航 DOM 的时候，编写漂亮、整洁的 HTML 标记可能反会使事情变得非常令人困惑。
2. 仅仅使用 DOM 指针导航文档可能是非常的冗长和不实际。
3. 通常，你并不需要直接访问文本节点，而是访问包绕它们的元素。

这把我们导向一个问题：有没有一种更好的方式用来在文档找到元素呢？有！通过使用工具箱里的几个有用的函数，你可以轻易改善现有的方法，把 DOM 导航变得简单得多。

处理 DOM 中的空白

让我们先回到那个示例 HTML 文档。先前，你试图定位那个单独的<h1>元素却因无关的文本节点而遇上了困难。这对于单个的元素可能还是好的，但是倘若你想要找到<h1>后面的那元素呢？你仍然会遭遇那个臭名昭著的空白 bug，不得不使用.nextSibling.nextSibling 来跳过<h1>和<p>之间的换行符。All is not lost though.(?)有一种技巧可以作为这一空白 bug 的补救办法，如程序 5-2 所示。这一特别的技巧去除了 DOM 文档所有的空白文本节点，便它变得更加易于穿行。这么做对你的 HTML 怎么渲染并没有明显的影响，却能使用你手工导航变得容易。应该注意的是，这个函数的结果并不是永久性的，每次 HTML 文档加载以后都需要重新运行。

程序 5-2. XML 文档中空白 bug 的补救办法

[Copy to clipboard] [-]

CODE:

```
function cleanWhitespace( element ) {
    //如果没有提供 element，则处理整个 HTML 文档
    element = element || document;
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
    //使用 firstChild 作为开始指针
    var cur = element.firstChild;

    //遍历所有子节点
    while ( cur != null ) {

        //如果该节点是文本节点，且只含有空白
        if ( cur.nodeType == 3 && ! /\S/.test(cur.nodeValue) ) {
            //删除些文本节点
            element.removeChild( cur );
        //否则，如果它是一个元素
        } else if ( cur.nodeType == 1 ) {
            //递归处理下一级节点
            cleanWhitespace( cur );
        }
        cur = cur.nextSibling; //移动到下一个子节点
    }
}
```

比方说你想要在上面的示例文档这个函数以找到<h1>后面的那个元素。完成这一工作的代码会是类似这样的：

[Copy to clipboard] [-]

CODE:

```
cleanWhitespace();
//获得文档元素
document.documentElement
    .firstChild //找到<head>元素
    .nextSibling //找到<body>元素
    .firstChild //找到<h1>元素
    .nextSibling //取得相邻的段落
```

这是一种既有好处又有缺点的技巧。最大的好处是，当你试图导航 DOM 文档的时候你可以保持某种程度的逻辑清晰。但是，考虑到你必须遍历所有的 DOM 元素和文本节点来寻找只包含空白的节点，这一技巧非常之慢。如果你的文档包含大量的内容，它会明显地拖慢你站点的加载。而且，每次你往文档中注入新的 HTML，你都需要重新扫描 DOM 的那一部分，确保没有附加的空白文本节点被加入。

上述函数里一个重要的方面就是节点类型的使用。一个节点的类型可以通过检查其 nodeType 属性为特定值来判定。有许多种可能的值，但最经常碰到的是以下三种

元素(nodeType=1): 匹配 XML 文档中的所有元素。例如，<li>，<a>，<p>，和<body>元素的 nodeType 全都是 1。

文本(nodeType=3): 匹配文档中所有的文本段。当在一个 DOM 结构中使用 previousSibling 的 nextSibling 导航时，你经常会在元素之间或元素内部遇到文本片段。

文档(nodeType=9): 匹配一个文档的根元素。比如，在一个 HTML 文档里，它就是<html>

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

元素。

另外，(在非 IE 浏览器中)你可以使用常数来代表不同的节点类型。比如，代替记住 1，3 或 9，你可以简单地使用 document.ELEMENT_NODE，document.TEXT_NODE，或 document.DOCUMENT_NODE。既然反复地清除 DOM 的空白文本节点大有累赘之嫌，我们自然应该寻求其它的方法来导航 DOM。

简单的 DOM 导航

使用纯 DOM 导航的原理(拥有每个方向的导航指针)你可以开发出可能更适合你的导航 HTML DOM 文档的函数。这一特殊的原则的依据是：多数 web 开发者只针对 DOM 元素而很少对其间的文本节点导航。下面提供几个函数，可以用来代替标准的 previousSibling，nextSIbling，firstChild，lastChild 以及 parentNode。程序 5-3 展示了一个返回元素的前一个元素的函数。类似于元素的 previousSibling 属性，如果没有前一个元素，该函数返回 null。

程序 5-3. 用来查找元素的前一个兄弟元素的函数

[Copy to clipboard] [-]

CODE:

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

程序 5-4 展示了一个返回元素的下一个兄弟元素的函数。与元素的 nextSibling 属性类似，当没有下一个元素时，函数返回 null。

程序 5-4. 用来查找元素的后一个兄弟元素的函数

[Copy to clipboard] [-]

CODE:

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

程序 5-5 展示了一个返回元素的第一个子元素的函数，与元素的 firstChild 属性类似。

程序 5-5.

[Copy to clipboard] [-]

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

CODE:

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

程序 5-5 展示了一个返回元素的最后一个子元素的函数，与元素的 lastChild 属性类似。

程序 5-6.

[Copy to clipboard] [-]

CODE:

```
Listing 5-6. A Function for Finding the Last Child Element of an Element
function last( elem ) {
    elem = elem.lastChild;
    return elem && elem.nodeType != 1 ?
        prev ( elem ) : elem;
}
```

程序 5-7 展示了一个返回元素父元素的函数，与元素的 parentNode 属性类似。你可以提供一个可选的参数 number，以一次向上移动几层——比如说，parent(elem,2)与 parent(parent(elem))等价。

程序 5-7. 用来查找元素父元素的函数

[Copy to clipboard] [-]

CODE:

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

使用这些新的函数，你可以快速地浏览一个 DOM 文档，而无需担心元素之间的文本。例如，为了找到<h1>元素的下一个元素，像从前一样，你现在可以像下面这么做：

[Copy to clipboard] [-]

CODE:

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

注意到这行代码的两个特点。第一，有一个新的引用：document.body。所有现代浏览

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

器都在 HTML DOM 文档的 body 参数里提供一个对<body>元素的引用。你可以利用这一点使你的代码更加简短和更加可理解。第二，函数的书写方式是非常地违背直觉的。通常，当你想到导航时你可能会说：从<body>元素开始，得到第一个元素，再得到第二个元素。但是在它实际的书写方式里，好像是倒着来的。为了替代这一方式，我将会讨论一些使得你定制的导航代码更加清晰的办法。

绑定到每一个 HTML 元素

在 Firefox 和 Opera 里，有一种可用的非常强大的对象原型，称为 HTMLElement，它允许你将函数和数据附加到每个单独的 HTML DOM 元素上。前面一节所介绍的函数是很呆板的，可以进行某种清理。一种完美的方式是把你的函数直接绑定到 HTMLElement 原型上，以此来把它们直接绑定到每一个单独的 HTML 元素。为了进行这一工作，对前面所建立的函数需要作三个更改：

1. 在函数里最上面添加一行将使 elem 指向 this，而不再是从参数列表取得。
2. 删除那个你不再需要的元素参数。
3. 将函数绑定到 HTMLElement 原型，这样你才能在第一个 DOM 中的 HTML 元素上使用它。

举例来说，新的 next 函数将会是如程序 5-8 所示的样子。

程序 5-8. 向所有 HTML　DOM 元素动态地绑定一个新的导航函数

[Copy to clipboard] [-]

CODE:

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

现在你可以像这样使用 next 函数(或者是经过造的前述的第一个函数)：

[Copy to clipboard] [-]

CODE:

```
//一个简单的例子:得到第一个的<p>元素
document.body.first().next()
```

这使得你的代码更加清晰而易于理解，因为你能以自然思考的顺序书写代码。如果你对这种书写风格有兴趣，我极力推荐你去看看 JQuery 库，它极好地利用了这一技术。

注意：因为 HTMLElement 只存在于三种现代浏览器中(Firefox,Safari,和 Opera)，你需要采取特殊的预防措施使它能够在 IE 中工作。Jason Karl(http://browserland.org)编写了一个特别便利的库，在两种不支持的浏览器中提供了对 HTMLElement(及其它相关功能)的访问。关于此库的更多信息可以在这里　　找到：

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

http://www.browserland.org/scripts/htmlelement/。

标准的 DOM 方法

所有的现代 DOM 实现都包含几种使工作更加有条理的方法。将它们和一些自定义函数结合使用，DOM 导航将会变成一种流畅得多的体验。首先，我们来看 JavaScript DOM 中包含的两个功能强大的函数：

*getElementById("everywhere"):*此方法只能应用于 document 对象，它在所有元素中查找 ID 等于 everywhere 的元素。这一强大的函数是立即访问一个元素的最快的方式。

*getElementsByTagName("li"):*此方法可以在任意元素上使用，它在所有后代元素中查找标签名为 li 的，并将它们作为一个(几乎与数组相同的)节点列表返回。

警告：对 HTML 文档来说，getElementById 会如你想象的那样工作：它检查所有的元素直到找到 id 属性与给定的值相同的那一个。然而，如果你载入一个远程的 XML 文档并使用 getElementById(或使用 JavaScript 以外的另一种语言里的 DOM 实现)，它默认并不根据 id 属性查找。它是由设计决定的；一个 XML 文档必须明确地(一般用 XML 定义或 XML 模式)指定 id 属性是什么。

警告：getElementsByName 返回一个节点列表。该结构外观的行为都跟通常的 JavaScript 数组十分相似，但是有一个重要的例外：它不具有通常的.push(),.pop(),.shift()等等这些 JavaScript 数组所具有的方法。使用 getElementsByName 时牢记这一点，会省去你许多的疑惑。

这两个方法在所有的现代浏览器中都是可用的，且对于定位特定元素极有帮助。回到前面我们试图找到<h1>元素的例子，现在我们可以像下面这么做：

[Copy to clipboard] [-]

CODE:

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

这段代码会有保障地工作并总是返回文档中的第一个<h1>元素。回到前面的示例文档，假设你想到得到所有的<li>元素并给它们加上边框：

[Copy to clipboard] [-]

CODE:

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

再一次地，我们回头看查找第一个<h1>元素后面的元素的问题，完成这一工作的代码到期可以减短得更多：

[Copy to clipboard] [-]

CODE:

```
//找到第一个<h1>元素的后一个元素
next(tag("h1")[0]);
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

这些函数提供了快速得到你想操作的 DOM 元素的能力。在学习使用这一能力来修改 DOM 之前，你需要先快速地看看你的脚本第一次执行以后 DOM 加载的问题。

等待 HTML DOM 加载

操作 HTML DOM 文档的一个难题是，你的 JavaScript 代码可能在 DOM 完全载入之前运行，这会导致你的代码产生一些问题。页面加载时浏览器内部操作的顺序大致是这样的：

1. HTML 被解析。
2. 外部脚本/样式表被加载。
3. 文档解析过程中内联的脚本被执行。
4. HTML DOM 构造完成。
5. 图像和外部内容被加载。
6. 页面加载完成。

头部包含的和从外部文件中载入的脚本实际上在 HTML DOM 构造好之前就执行了。正如前面提到的，这一个问题是很重要的，因为在那两种地方的执行的所有脚本将不能访问 DOM。可喜的是，存在许多绕开这一问题的办法。

等待页面加载

到目前为止，最常用的技术是在任何 DOM 操作之前简单地等待整个页面加载。使用这一技术，可以通过简单地给 window 对象的 load 事件附加一个在页面载入后触发的函数。在第六章中我将讨论关于事件的更多细节。程序 5-10 展示了一个在页面加载完成后执行 DOM 相关代码的例子。

程序 5-10. 为 window.onload 属性附加回调函数的 addEvent 函数

[Copy to clipboard] [-]

CODE:

```
//等待页面加载完成
//(使用了下一章描述的 addEvent 函数)
addEvent(window, "load", function() {
    //执行 HTML DOM 操作
    next( id("everywhere") ).style.background = 'blue';
});
```

尽管这一操作可能是最简单的，它也将总是最慢的。从加载操作的顺序中，你可能已发现页面加载完成绝对是最后一步。这意味着如果在你的页面上有大量的图像、视频等等，你的用户在 JavaScript 最终执行前得等待很大一阵子。

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

等待大部分 DOM 加载

第二种技术很迂回，不太推荐使用。如果你还记得，我在上一节里说了，内联的脚本是在 DOM 构造以后执行的。这是一个半真半假的说法。那些脚本实际上是在 DOM 构造时遇上了就执行的。这就是说如果你有一段内联的脚本嵌在页面的中间部分，则该脚本只能立即拥有前半部分 DOM 的访问权。然而，把脚本作为非常靠后的元素嵌入页面中，就意味着你能够有效地对先于它出现的所有的 DOM 元素进行访问，获得一种假冒的模拟 DOM 加载的方式。这种方法的典型实现通常如程序 5-11 所示。

程序 5-11. 通过向 HTML DOM 的结尾置入(包含函数调用的)<script>标签来判定 DOM 是否已经加载

[Copy to clipboard] [-]

CODE:

```
<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>
```

在这个例子里，一个内联脚本作为 DOM 的最后一个元素；它将是最后一个被解析和执行的。它所做的唯一的事情是调用 init 函数(函数内部应包含你想要处理的任何 DOM 相关的代码)。这一解决方案的存在的最大的问题在于，它是混乱的：给你的 HTML 里加入了额外的标记，只为了判定 DOM 是否已经加载。

断定 DOM 何时加载完成

最后一种可用来监视 DOM 加载的技术，可能是最复杂(从实现的角度来看)但也是最有效的。它结合了绑定到 window 的 load 事件的简易性和内联脚本技术的速度。

这一技术的原理是在不阻塞浏览器的前提下尽可能快地反复检查 HTML DOM 是否已经具有了你所需的特性。有几种东西可以被检查以判断 HTML 文档是否已经可以操作了：

1. *document:* 你需要检查 DOM document 是否已经存在。如果你检查得够快的话，它一

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

开始可能仅仅是 undefined。

2. *document.getElementsByTagName 和 document.getElementByID:* 检查 document 是否已经具备了经常使用的 getElementsByTagName 和 getElementById 函数；这些函数将在它们准备好被使用以后存在。

3. *document.body:* 作为额外的保障，检查<body>元素是否已完成被载入。理论上讲，前面的检查应该已经足够了，但是我发现过它们还不够好的例子。

使用这些检查，你将对 DOM 何时准备好被使用有一个足够好的把握（好到可能只错过了几毫秒）。这一方法近乎没有瑕疵。仅使用前面的检查，脚本可以在所有的现代浏览器里运行得相对很好了。然而，Firefox 某些新的缓存机制的实现，导致了 window 的 load 事件实际上能够在你的脚本判断 DOM 是否就绪之前就触发。为了利用这一优势，我也加入了对 window 的 load 事件的检查，希望获得一些额外的速度。

最终，domReady 函数在 DOM 就绪之前一直在收集所有的待运行函数的引用。一旦 DOM 确实准备好了，就遍历这些引用并一个一个地执行它们。程序 5-12 展示了一个可用来监视 DOM 何时完全载入的函数。

程序 5-12. 监视 DOM 直到它准备好的一个函数

[Copy to clipboard] [-]

CODE:

```
function domReady( f ) {
    //如果 DOM 已经载入，立即执行函数
    if ( domReady.done ) return f();
    //如果我们已经添加过函数
    if ( domReady.timer ) {
        //则将函数添加到待执行的函数列表
        domReady.ready.push( f );
    } else {
        //为页面完成加载时附加一个事件，以防它率先发生
        //使用了 addEvent 函数
        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 是否已就绪
        domReady.timer = setInterval( isDOMReady, 13 );
    }
}
//检查 DOM 是否已经准备好导航
function isDOMReady() {
    //如果我们断定页面已经加载完成了，则返回
    if ( domReady.done ) return false;
    //检查一些函数和元素是否已可访问
    if ( document && document.getElementsByTagName &&
        document.getElementById && document.body ) {
        //如果它们已就绪，则停止检查
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
        clearInterval( domReady.timer );
        domReady.timer = null;

        //执行所有正在等待的函数
        for ( var i = 0; i < domReady.ready.length; i++ )
            domReady.ready[i]();
        //记住现在我们已经完成
        domReady.ready = null;
        domReady.done = true;
    }
}
```

现在我们应该看看这在一个 HTML 文档里会是什么样。使用 domReady 函数就像使用 addEvent 函数(见第 6 章)一样，绑定你的特定函数到文档准备好导航和操作的时候被触发。在下面的例子里我把 domReady 函数放入了一个名为 domready.js 的外部 JavaScript 文件里。程序 5-3 展示了怎样使用新的 domReady 函数来监视 DOM 何时已载入。

程序 5-13. 使用 domReady 函数在判定 DOM 何时准备好导航和修改

[Copy to clipboard] [-]

CODE:

```
<html>
<head>
    <title>Testing DOM Loading</title>
    <script type="text/javascript" src="domready.js"></script>
    <script type="text/javascript">
        function tag(name, elem) {
            //如果上下文元素未提供，则搜索整个文档
            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>
```

既然你了解了用来导航一般的 XML DOM 文档的和克服 HTML DOM 文档加载难题的几种方法，这个问题应该被摆在眼前了：有没有更好的在 HTML 文档中查找元素的方法呢？可

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

喜的是，答案是响亮的"有"。

在 HTML 文档中查找元素

在一个 HTML 文档中查找元素的方式常常与在一个 XML 文档中有很大的不同。考虑到现代 HTML 实际上是 XML 的一个子集，这看起来可能有些矛盾；但是 HTML 文档包含一些你可以利用的基础的不同点。

对 JavaScript/HTML 开发者来说，最重要的两个优势是 CSS 类的使用的 CSS 选择符的知识。记住这些，你就可以创建一些强大的函数用来使得 DOM 导航更加简单和可理解。

通过类名查找元素

用类名定位元素是一种广泛流传的技术，由 Simon Willison(http://simon.incutio.com)于 2003 年推广，最初由 Andrew Hayward(http://www.mooncalf.me.uk)编写。这一技术是非常易行的：遍历所有元素(或所有元素的一个子集)，选出其中具有特定类名的。程序 5-14 展示了一种可能的实现。

程序 5-14. 从所有元素中找出具有特定类名的元素的一个函数

[Copy to clipboard] [-]

CODE:

```
function hasClass(name,type) {
    var r = [];
    //限定类名(允许多个类名)
    var re = new RegExp("(^|\\s)" + name + "(\\s|$)");
    //用类型限制搜索范围，或搜索所有的元素
    var e = document.getElementsByTagName(type || "*");
    for ( var j = 0; j < e.length; j++ )
        //如果元素类名匹配，则加入到返回值列表中
        if ( re.test(e[j]) ) r.push( e[j] );

    //返回匹配的元素
    return r;
}
```

现在你可以通过一个指定的类名使用些函数来快速地查找任何元素，或特定类别的任何元素(比如，<li>或<p>)。指定要查找的标签名总会比查找全部(*)要快，因为查找元素的范围被缩小了。比如，在我们的 HTML 文档里，如果想要查找所有类名包含"test"的元素，你可以这么做：

[Copy to clipboard] [-]

CODE:

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


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

如果你只想查找类名包含"test"的所有<li>元素，则这样：

[Copy to clipboard] [-]

CODE:

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

最后，如果你想找到第一个类名包含"test"的<li>元素，则这么做：

[Copy to clipboard] [-]

CODE:

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

这个函数单独使用已经很强大了，而当与 getElementById 和 getElementByTagname 联合使用时，你就拥有了非常强大的可完成最复杂的 DOM 工作的一套工具。

通过 CSS 选择符查找元素

作为一个 web 开发者，你已经知道一种选择 HTML 元素的方式：CSS 选择符。CSS 选择符是用来将 CSS 样式应用于一组元素的表达式。随着 CSS 标准的每一次修订(1,2 和 3)，更多的功能被加入了选择符规范中，允许开发者更容易地精确确定他们想要的元素。不幸的是，浏览器一直是极其缓慢地提供 CSS2 和 CSS3 选择符的完全实现，这意味着你可能还不知道它们所提供的一些很酷的新功能。如果你对 CSS 的所有的新而酷的功能感兴趣，我建议探究一下 W3C 的关于该项目的网页：

CSS1 selectors: http://www.w3.org/TR/REC-CSS1#basic-concepts/

CSS2 selectors: http://www.w3.org/TR/REC-CSS2/selector.html

CSS3 selectors: http://www.w3.org/TR/2005/WD-css3-selectors-20051215/

每种 CSS 选择符规范中可用的的功能大体上是相似的，因为后继的版本总是包含前面版本的所有功能。然而，每一个版本都加入了一些新的功能。举例来说，CSS2 包含属性和子代选择符，而 CSS3 提供了额外的语言支持，通过属性类型和否定来选择。比如，下面都是有效的 CSS 选择符：

#main div p(译注：原文写作"#main p"，疑有误): 此表达式查找一个 ID 为"main"的元素的所有的后代 元素的所有的后代<p>元素。这是一个正确的 CSS1 选择符。

div.items > p: 此表达式查找所有的类名包含"items"的 元素，然后找出所有的子代<p>元素。这是一个有效的 CSS2 选择符。

div:not(.items): 此表达式查找所有类名不包含"items"的 元素。这是一个有效的 CSS3 选择符。

现在，你可能会奇怪为什么我会讨论 CSS 选择符，如果不能实际地使用它们来定位元素(只应用于 CSS 样式)的话。一些富有进取精神的开发者在着手于此，创建了能够处理从 CSS1 到全部的 CSS3 的 CSS 选择符实现。使用这些库，你将能够快速而容易地选择任何元素并对它们进行操作。

cssQuery

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

第一个公开可用的支持全部 CSS1-CSS3 的库被称为 cssQuery，由 Dean Edwards(http://dean.edwards.name)创建。它背后的前提是简单的：给出一个选择符，cssQuery 将找到所有匹配的元素。另外，cssQuery 被分割成许多子库，每一个对应于 CSS 选择符的一个时期，这意味着如果不需要 CSS3 支持的话，你可以把它排除在外。这一特别的库彻底而广泛，可工作于所有的现代浏览器中(Dean 是一个坚定的跨浏览器支持者)。为了使用这个库，你需要提供选择器，以及可选的在其中搜索的上下文元素。下面是几个示例：

[Copy to clipboard] [-]

CODE:

```
//查找<div>元素的所有子代<p>元素
cssQuery("div > p");
//查找所有的<div>，<p>，<form>
cssQuery("div,p,form");
//查找所有的<p>和<div>，然后查找他们内部的所有<a>元素
var p = cssQuery("p,div");
cssQuery("a",p);
```

执行 cssQuery 函数会返回一个匹配元素的数组。你可以对它实施操作，就好像你刚刚执行了一次 getElementsByTagName。比如说，为了给所有链接到 Google 的链接加上边框，你可以执行以下操作：

[Copy to clipboard] [-]

CODE:

```
//为所有指向 Google 的链接加上边框
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 的更多信息及完整的源代码下载可以在 Dean Edwards 的网站上找到：http://dean.edwards.name/my/cssQuery/。

提示：Dean Edwards 是一位 JavaScript 奇才；他的代码绝对是令人吃惊的。我极力推荐你到他的 cssQuery 库里去逛逛，至少看看优秀的可扩展的 JavaScript 代码是怎么写的。

jQuery

这是 JavaScript 库的世界里新近的加入者，但是提供了一些值得注意的编写 JavaScript 代码的方式。我最初只是想把他写成"简单的"CSS 选择符库（跟 cssQuery 相似），直到 Dean Edwards 发布他的杰出的 cssQuery 库迫使这些代码向另一个不同的方向发展。这个库提供完全的 CSS1-CSS3 选择符的支持以及一些基本的 XPath 功能。在此之上，它还提供了进行更深入的 DOM 导航和操作的能力。跟 cssQuery 一样，jQuery 也完全支持现代浏览器。这里有几个使用 jQuery 自定义的 CSS 的 XPath 的混合物选择元素的例子：

[Copy to clipboard] [-]

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

CODE:

```
//查找所有的类名包括"links"且其内部有<p>元素的<div>元素
$("div.links[p]")
//查找所有<p>元素和<div>元素的后代
$("p,div").find("*")
//查找不指向 Google 的所有<a>超链接
$("a[@href^='google.com']:even")
```

为了使用 jQuery 得到的结果，你有两种选择。首先，你可以执行$("expression").get()来得到匹配元素的一个数组——与 cssQuery 完全相同的结果。你可以做的第二件事是使用 jQuery 的独有的内建函数操作 CSS 和 DOM。于是，回到用 cssQuery 为所指向 Google 的链接加边框的例子，你可以这么做：

[Copy to clipboard] [-]

CODE:

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

在 jQuery 的项上网站上可以找到大量的示例、演示和文档，以及可定制的下载：http://jquery.com。

注意：应该指出的是，无论是 cssQuery 还是 jQuery，实际上都不要一定求使用 HTML 文档来导航；它们适用于任何 XML 文档。下节的 XPath 将为你讲述纯 XML 形式的导航。

XPath

XPath 表达式是一种导航 XML 文档的极其强大的方法。XPath 已经存在了好几年；几乎可以说只要有 DOM 的实现，就会有 XPath 紧随其后。XPath 表达式要比用 CSS 选择符可以写出的任何东西都强大得多，即便他们更加冗长。表 5-1 列举了一些不同的 CSS 选择符与 XPath 表达式的并行比较。

表 5-1. CSS3 选择符与 XPath 表达式的比较

目标	CSS3	XPath
所有元素	*	//*
所有<p>元素	p	//p
所有子元素	p>*	//p/*
特定 ID 的元素	#foo	//*[@id='foo']
特定 class 的元素	.foo	//*[contains(@class,'foo')]
带属性的元素	*[title]	//*[@title]
<p>的第一个子元素	p>&.first-child	//p/*[0]
拥有一个子元素的所有<p>	不能实现	//p[a]
下一个元素	p+*	//p/following-sibling::*[0]

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

如果前面的表达式激起了你的兴趣，我推荐你去浏览两个 XPath 规范(不过，XPath 1.0 通常是唯一被现代浏览器所完全支持的)，感觉一下那些表达式是如何工作的。

XPath 1.0: http://www.w3.org/TR/xpath/

XPath 1.0: http://www.w3.org/TR/xpath20/

如果你想对该主题进行深入的研究，我推荐你阅读 Elliotte Harold 和 Scott Means 所著的 XML in a Nutshell（O'Reilly，2004），或者 Jeni Tennison 所著的 XSLT 2.0:From Novice To Professional(Apress，2005)。另外，有一些很好的教程可以帮助你开始使用 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 的支持是零星的；IE 和 Mozilla 都支持全部(尽管各不相同)的 XPath 实现，而 Safari 和 Opera 都只有正在开发中的版本。为解决这一问题，有几种完全用 JavaScript 编写的 XPath 实现。它们一般都很慢(与基于浏览器的 XPath 实现相比)，但是可以在所有浏览器里稳定地工作。

XML for Script: http://xmljs.sf.net/

Google AJAXSLT: http://goog-ajaxslt.sf.net

另外，一个名为 Sarissa(http://sarissa.sf.net)的项目立志于针对每种浏览器实现创建一个通用的包装。这能给你只须一次编写 XML 访问代码的能力，而仍能获得浏览器所支持的 XML 解析的所有速度优势。这一技术最大的问题是在 Opera 和 Safari 浏览器里它仍缺乏对 XPath 的支持。

与广泛支持的纯 JavaScript 方案相比，使用浏览器内建的 XPath 通常被认为是实验性的技术。但是，XPath 的使用和流行只会增长，它肯定应该被看作 CSS 选择器王位的强劲的竞争者。

既然你已经拥有了定位任何一个甚至是一组 DOM 元素必须的知识和工具，我们现在应该讨论你可以使用该能力做些什么。从属性的操作到 DOM 元素的添加或删除，一切都是可能的。

获取元素的内容

所有的 DOM 元素可以包含一种或三种东西：文本，元素，或文本与元素的混合。大致说来，最常见的是第一种和第三种情况。在这一节里你将学到检索元素内容的几种常见的方式。

获取元素内的文本

对于新接触 DOM 的人来说，获取元素内部的文本可能是最令人困惑的任务。然而，它也是一种在 HTML DOM 和 XML DOM 里都能需要的，因而了解怎样实现将很适合你。在图 5-3 中的示例 DOM 结构里，一个根元素<p>包含了一个元素和一个文本块。元素本身又包含了一个文本块。

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

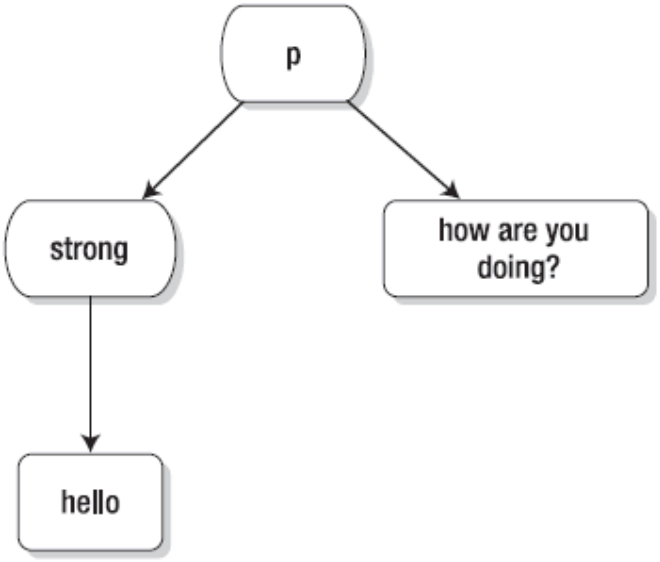


图 5-3. 同时包含元素和文本的示例 DOM 结构

我们来看看怎样取得这些元素中每一个元素的文本。元素是最容易拿来作为开始的，它仅包含一个文本节点。

应该注意的是在所有的不基于 Mozilla 的浏览器里，存在一个属性 inenrText 用来获取元素内部文本。它在这方面极其便利。不幸的是，因为支持它的浏器在浏览器市场里并不占有显著份额(译注：估计作者没怎么用过 IE)，而且它不能在 XML DOM 文档里运行，你仍然需要探索一种可行的替代方案。

获取元素的文本内容的诀窍在于，你需要记住文本并不是直接包含在元素里的，它被包含在子文本节点里(这可能看起来有点奇怪)。程序 5-15 展示了怎样用 DOM 取出一个元素内部的文本。设其中的变量 strongElem 包含了对元素的引用。

程序 5-15. 获取元素的文本内容

[Copy to clipboard] [-]

CODE:

```
//非 Mozilla 浏览器:
strongElem.innerText
//所有平台:
strongElem.firstChild.nodeValue
```

知道了怎样得到单个元素的文本内容，我们再来看看怎样取得<p>元素内文本内容的联合。在此过程中，也可以开发出一个可以获取任何元素的文本内容的通用函数，不论该元素实际上包含什么，如程序 5-16 所示。调用 text(Element)将会返回组合了该元素及其所有后代元素包含的文本的一个字符串。

程序 5-16. 检索元素文本的一个通用函数

[Copy to clipboard] [-]

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

CODE:

```
function text(e) {
    var t = "";
    //如果传递的是元素，则取其子元素，
    //否则假定它是一个数组
    e = e.childNodes || e;
    //检查所有子节点
    for ( var j = 0; j < e.length; j++ ) {
        //如果它不是一个元素，追加其文本到返回值
        //否则，对其所有子元素递归处理
        t += e[j].nodeType != 1 ?
            e[j].nodeValue : text(e[j].childNodes);
    }

    //返回所得的文本
    return t;
}
```

有了一个可用来获取任何元素文本内容的函数，你现在可以用如下代码检索前面例子中的<p>元素的文本内容：

[Copy to clipboard] [-]

CODE:

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

这个函数特别美妙的一点是不论在 XML 还是 HTML 的 DOM 中它都可以有保障地运行，也就是说你现在拥有了检索任何元素文本内容的一致的方法。

获取元素内的 HTML

与获取元素内部文本相反，获取元素内部的 HTML 是可以执行的最容易的 DOM 任务之一。因为 IE 开发组所开发的一个功能，所有的现代浏览器里的任何 HTML DOM 元素都包含一个额外的属性:innerHTML。通过这一属性你就可以获得一个元素内部的所有 HTML 和文本。另外，使用 innerHTML 也非常快，速度通常是递归获取元素文本内容的很多倍。然而，事情也不是尽善尽美。怎样实现 innerHTML 属性依赖于浏览器，因为在方面并不存在真正的标准，浏览器可以返回它认为有价值的任何东西。比如说，这里有一些你在使用 innerHTML 属性时会遇上的一些怪异的 bug：

1. 基于 Mozilla 的浏览器在 innerHTML 语句里不会返回<style>元素。
2. IE 返回的所有元素标签都是大写的，这在寻求一致性时可能会十分令人沮丧。
3. innerHTML 属性只对 HTML DOM 文档元素里可用；尝试将它用于 XML DOM 文档将会得到 null 值。

innerHTML 的使用是很简单的；访问该属性就可以得到一个包含元素 HTML 内容的字符

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

串。如果元素不包含子元素而只含有文本，返回 HTML 内容将只包括文本。为了弄清它是怎样工作的，我们来看看图 5-2 中所示的两个元素：

[Copy to clipboard] [-]

CODE:

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

如果你确定你的元素里只包含文本，这个方法可以作为复杂的元素文本内容获取的超级简单的替换方式。另一方面，能够检索元素的 HTML 内容意味着你现在可以建立一些很酷的使用了即时编辑功能的动态应用程序——关于此话题的更多内容可在第十章找到。

操作元素属性

紧跟着检索元素内容，获取和设置元素的属性值也是最常进行的操作。典型地，元素具有的属性列表是与收集自元素本身 XML 表示的信息一起预载入的，并存储在一个关联数组中供稍后访问。比如如下网页中的 HTML 片段：

[Copy to clipboard] [-]

CODE:

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

一旦它被载入 DOM，HTML 表单元素(变量 formElem)将拥有一个关联数组，你可以从中收集"名称/值"对。这一结果类似于以下形式：

[Copy to clipboard] [-]

CODE:

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

使用属性数组判定一个元素是具有某种属性绝对应该是很普通的，但是有一个问题：出于某种原因，Safari 不支持这一点。更过分的是，有很大潜在价值的 hasAttribute 函数在 IE 里不被支持。那么应该怎么确定一个属性是不是存在呢？一个可能的方式是使用 getAttribute 函数(此函数将在下节讨论)并测试返回值是否为 null，如程序 5-17 所示。

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

程序 5-17. 判定元素是否具有指定属性

[Copy to clipboard] [-]

CODE:

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

有了这个函数，以及知道属性是怎么使用的，现在你已经准备好检索和设置属性值了。

获取和设置属性值

存在两种不同的方式从一个元素获取属性值，依赖于你使用的 DOM 文档类型，。如果你希望安全并总是使用兼容 XML DOM 的方法，可以使用 getAttribute 和 setAttribute 方法。它们可以以这种方式被使用：

[Copy to clipboard] [-]

CODE:

```
//得到一个属性
id("everywhere").getAttribute("id")
//设置一个属性值
tag("input")[0].setAttribute("value","Your Name");
```

除了这一对标准的 getAttribute/setAttribute 方法以外，HTML DOM 还有一个额外的属性集，行为类似于属性的获取器/设置器。这在现代的 DOM 实现(不过只保证 HTML DOM 文档)中是普遍可用的，因此编写更短的代码时使用它们可以给你很大的优势。下面的代码说明怎样使用这种方法获取和设置 DOM 元素属性：

[Copy to clipboard] [-]

CODE:

```
//快速获取一个属性
tag("input")[0].value
//快速设置一个属性
tag("div")[0].id = "main";
```

在属性中有一些例外的情况是你应该意识到的。最常碰到的是访问类名属性的问题。在所有的浏览器中，为了一致地操作类名，你必须使用 elem.className 属性访问 className 属性，以代替本应更合适的 getAttribute("class")。这一问题同样也出现在属性"for"上，它被重命名为"htmlFor"。另外，这种情况还见于两个 CSS 属性：cssFloat 和 cssText。这种特殊的命名方式的出现是因为 class，for，float，和 text 这些单词是 JavaScript 中的保留字。

为了解决这些特例带来的问题并简化整个读取和设置正确的属性的过程，你应该使用一个为你照顾这些特殊情况的函数。程序 5-18 展示了一个设置和获取元素属性的函数。使用两个参数调用该函数，如 attr(element,id)，将返回属性的值。使用三个参数调用该函数，如 attr(element,class,test)，将设置该属性的值并返回新的值。

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

程序 5-18. 获取和设置元素的属性值

[Copy to clipboard] [-]

CODE:

```
function attr(elem, name, value) {
    //确保传递进来的是一个有效的属性名称
    if ( !name || name.constructor != String ) return '';

    //判定属性名称是不是异常的命名情况之一
    name = { 'for': 'htmlFor', 'class': 'className' }[name] || name;
    //如果用户正在设置值
    if ( typeof value != 'undefined' ) {
        //首先用快速的方法设置
        elem[name] = value;

        //如果可以，使用 setAttribute
        if ( elem.setAttribute )
            elem.setAttribute(name,value);
    }
    //返回属性的值
    return elem[name] || elem.getAttribute(name) || '';
}
```

拥有一个访问和改变属性的标准方式而无须顾及它们的实现，这是一个很强大的工具。程序 5-19 的例子展示了在一些通常的情况下怎样使用 attr 函数来简化属性的处理过程。

程序 5-19. 使用 attr 函数来设置和获取 DOM 元素的属性值

[Copy to clipboard] [-]

CODE:

```
//为第一个<h1>元素设置类
attr( tag("h1")[0], "class", "header" );
//为每一个<inpu>元素设置值
var input = tag("input");
for ( var i = 0; i < input.length; i++ ) {
    attr( input[i], "value", "" );
}
//为一个名为"invalid"的<input>元素添加边框
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";
    }
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
}
```

到目前为止，我只讨论了在 DOM 里固有的属性(如 ID,class,name，等等)值的获取/设置。但是，有一个非常便利的技术是，设置/获取非传统的属性。比如说，你可以设置一个新的属性(只能通过访问元素的 DOM 版本看到)然后在稍后检索它，而无须修改文档的物理属性。举例来说，假设你有一个名词及其定义的列表，想要实现当一个名词被点击时，将它的定义展开。这一结构的 HTML 将大致如程序 5-20 所示。

程序 5-20. 带有隐藏了定义的定义列表的 HTML 文档

[Copy to clipboard] [-]

CODE:

```
<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>
```

我将在第六章谈到与事件有关的更多细节，而现在我将试图保持我们的代码足够的简单。这便有了一个快速的脚本，允许你通过点击被定义的名词而显示（或隐藏）定义本身。这个脚本应该包含在页面的头部或从一个外部文件被包含。程序 5-21 展示了建立一个可展开的定义列表所需的代码。

程序 5-21. [color]Allowing for Dynamic Toggling to the Definitions(?)

[Copy to clipboard] [-]

CODE:

```
// Wait until the DOM is Ready
domReady(function(){
    //找到所有的定义名词
    var dt = tag("dt");
    for ( var i = 0; i < dt.length; i++ ) {
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
        //监视用户对该名词的点击
        addEvent( dt[i], "click", function() {
            //检查定义是否已经展开
            var open = attr( this, "open" );
            //切换定义的显示状态
            next( this ).style.display = open ? 'none' : 'block';
            //记住定义是否已经展开
            attr( this, "open", open ? '' : 'yes' );
        });
    }
});
```

了解怎样穿行于 DOM 中和怎样检查并修改属性之后，我们再来学习怎样创建新的 DOM 元素，将它们插入你希望的地方，并在你不再需要它们的时候将其删除。

修改 DOM

通过了解如何修改 DOM，你将能够在 DOM 中为所欲为：从实时创建自定义的 XML 文档到建立适合用户输入的动态表单；可能性是无限的。修改 DOM 分成三个步骤：首先创建一个新的元素，然后将它插入到 DOM，最后把它再删除(如果需要的话)。

使用 DOM 创建节点

修改 DOM 背后的主要方法是 createElement 函数，它给你实时创建新元素的能力。然而，新的元素在你创建它的时候并没有立即插入到 DOM 中(这是 DOM 初学者普遍迷惑的一点)。首先，我来集中说说 DOM 元素的创建。

createElement 方法授受一个参数，元素的标签名，并返回元素的虚拟的 DOM 表示——不包含任何的属性和样式。如果你正在用 XSLT 生成的 XHTML 页(或者是指明了精确的内容类型的 XHTML 页)开发应用程序，你必须记住你实际上正在使用 XML 文档，并且你所有的元素都需要拥有正确的 XML 名称空间与之相关联。为了无缝地解决这一问题，你可以使用一个简单的函数来无声地测试你正使用的 HTML DOM 文档是否具有创建带名称空间的新元素的能力(XHTML DOM 文档的特性)。如果是在这一情况下，你必须用正确的 XHTML 名称空间来创建新的元素。如程序 5-22 所示。

程序 5-22. 新建 DOM 元素的一个通用的函数

[Copy to clipboard] [-]

CODE:

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


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

例如，使用这个函数你可以创建简单的 元素并给它附加一些额外的信息。

[Copy to clipboard] [-]

CODE:

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

另外，应该注意的是有一个用来创建新的文本节点的叫做 createTextNode 的方法。它接受单个参数(你相要放在该节点里的文本)，返回所创建的文本节点。

使用新建的 DOM 元素的文本节点，你现在可以在你需要地方将它们正确地插入 DOM 文档。

插入到 DOM

即使对于有经验的人来说,插入到 DOM 也是非常令人困惑的而且有时会感觉很不灵活。在你的工具库里有两个函数可用来完成这一工作。

第一个函数，insertBefore，允许你将元素插入到另一个子元素的前面。当你使用这一函数的时候，代码会像是这个样子的：

[Copy to clipboard] [-]

CODE:

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

我使用的帮助记住其参数顺序的方法是短语"将第一个元素插入(insert)到第二个之前(before)"。我将会给你一个在一分钟之内记住这个的更容易的方式。

既然你拥有了一个在其它节点前面插入节点的函数，你应该问你自己了："那我怎样把节点作为最后一个子节点插入呢？"有一个叫做 appendChild 的另一个函数正好让你那么做。在一个元素上调用 appendChild，将把指定的节点追加到元素的子节点列表的末尾。使用些函数的大致代码如下：

[Copy to clipboard] [-]

CODE:

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

为了避免记忆 insertBefore 和 appendChild 的特殊的参数顺序，你可以使用我创建的用来解决这一问题的两个辅助函数：使用 5-23 和 5-24 中的新的函数，参数的顺序总是先是与你要插入的元素/节点相关的元素然后是你要插入的元素/节点。另外，before 函数允许你有选择性地提供父元素，可能为你省去一些代码。最后，这两个函数都允许你传入一个将被插入/追加的字符串并为你将它自动地转换成一个文本节点。推荐传递一个父元素作为引用(以防 elem 参数碰巧是 null)。

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

插入到 DOM

即使对于有经验的人来说，插入到 DOM 也是非常令人困惑的而且有时会感觉很不灵活。在你的工具库里有两个函数可用来完成这一工作。

第一个函数，insertBefore，允许你将元素插入到另一个子元素的前面。当你使用这一函数的时候，代码会像是这个样子的：

[Copy to clipboard] [-]

CODE:

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

我使用的帮助记住其参数顺序的方法是短语"将第一个元素插入(insert)到第二个之前(before)"。我将会给你一个在一分钟之内记住这个的更容易的方式。

既然你拥有了一个在其它节点前面插入节点的函数，你应该问你自己了："那我怎样把节点作为最后一个子节点插入呢？"有一个叫做 appendChild 的另一个函数正好让你那么做。在一个元素上调用 appendChild，将把指定的节点追加到元素的子节点列表的末尾。使用些函数的大致代码如下：

[Copy to clipboard] [-]

CODE:

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

为了避免记忆 insertBefore 和 appendChild 的特殊的参数顺序，你可以使用我创建的用来解决这一问题的两个辅助函数：使用 5-23 和 5-24 中的新的函数，参数的顺序总是先是与你要插入的元素/节点相关的元素然后是你要插入的元素/节点。另外，before 函数允许你有选择性地提供父元素，可能为你省去一些代码。最后，这两个函数都允许你传入一个将被插入/追加的字符串并为你将它自动地转换成一个文本节点。推荐传递一个父元素作为引用(以防 elem 参数碰巧是 null)。

程序 5-23. 在另一个元素前面插入元素的函数

[Copy to clipboard] [-]

CODE:

```
function before( parent, before, elem ) {
    //检查是否提供了父节点
    if ( elem == null ) {
        elem = before;
        before = parent;
        parent = before.parentNode;
    }
    parent.insertBefore( checkElem( elem ), before );
}
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

程序 5-24. 将元素作为另一元素的子元素追加的一个函数

[Copy to clipboard] [-]

CODE:

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

程序 5-25 所示的辅助函数允许你容易地插入元素和文本(文本将自动地转换成合适的文本节点)。

程序 5-25. 用于 before()和 append()函数的一个辅助函数

[Copy to clipboard] [-]

CODE:

```
function checkElem( elem ) {
    //如果提供了一个字符串，则将其转换为文本节点
    return elem && elem.constructor == String ?
        document.createTextNode( elem ) : elem;
}
```

现在，使用 before 和 append 函数和创建新的 DOM 元素，你可以向 DOM 加入更多的信息供用户浏览，如程序 5-26 所示。

程序 5-26. 使用 append 和 before 函数

[Copy to clipboard] [-]

CODE:

```
//新建一个<li>元素
var li = create("li");
attr( li, "class", "new" );
//创建新文本内容并将它加下到<li>元素
append( li, "Thanks for visiting!" );
//将<li>元素加入到第一个 Ordered List 的顶部
before( first( tag("ol")[0] ), li );
//运行以上语句将会把空的<ol>
<ol></ol>
//转换成如下：
<ol>
    <li class='new'>Thanks for visiting!</li>
</ol>
```

在你将些信息插入到 DOM 的瞬间(不论使用 insertBefore 还是 appendChild)，它将立即被渲染并被用户看到。因此，你可以使用它提供即时的反馈。这在需要用户输入的交互式的

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

应用程序中尤其有用。

看过了怎样仅使用基于 DOM 的方法创建和插入节点，再来学习向 DOM 注入内容的替代方法将会是特别有益的。

注入 HTML 到 DOM

甚至比创建一般的 DOM 节点并插入到 DOM 的更加流行的技术是直接将 HTML 注入到文档里。最简单的达到这一点的方法是使用前面讨论过的 innerHTML 属性。它除了作为一种检索元素内部的 HTML 的方法之外，也能用来设置元素内部的 HTML。作为其简单性的一个例子，假设你有一个空的<ol>元素并且你想要给它加入一些<li>元素；完成这一任务的代码将是像这样的：

[Copy to clipboard] [-]

CODE:

```
//往一个<ol>元素中加入一些<li>元素
tag("ol")[0].innerHTML = "<li>Cats.</li><li>Dogs.</li><li>Mice.</li>";
```

这岂不是要比别着了魔似地创建一大堆地 DOM 元素和相关的文本节点要简单得多？你将会很高兴地知道它也比使用 DOM 方法要快得多(参见 http://www.quirksmode.org)。不过，它也并不是完美的——使用 innerHTML 注入的方法同样存在一些棘手的问题：

1. 正如前面所提到的，innerHTML 方法不存在于 XML DOM 文档中，这意味着你将不得不继续使用传统的 DOM 方法。

2. 用客户端 XSLT 生成的 XHTML 文档不具有 innerHTML 方法，因为它们同样是纯 XML 文档。

3. 设置 innerHTML 会完全删除原来已经存在于元素中的节点，这就是说没有办法像纯 DOM 方法里那样方便地追加或在前面插入。

最后一点尤其麻烦，因为在其它元素之前插入或在子节点列表的后面追加是特别有用的一个功能。但是，借助一些 DOM 的魔力，你可以让你的 append 和 before 方法在常规的 DOM 元素之外也适用于常规的 HTML 字符串。这一转换分为两步。首先，你创建一个能够处理 HTML 字符串、DOM 元素、以及 DOM 元素数组的新的 checkElem 函数，如程序 5-27 所示。

程序 5-27. 将混合了 DOM 节点和 HTML 字符串的数组参数转换成一个纯 DOM 节点数组

[Copy to clipboard] [-]

CODE:

```
function checkElem(a) {
    var r = [];
    //如果参数不是一个数组，强迫它是
    if ( a.constructor != Array ) a = [ a ];
    for ( var i = 0; i < a.length; i++ ) {
        //如果是一个字符串
        if ( a[i].constructor ==  String ) {
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
            //创建一个临时的元素来储藏 HTML
            var div = document.createElement("div");

            //注入 HTML，将它转换成一个 DOM 结构
            div.innerHTML = a[i];

            //从临时<div>元素中取出 DOM 结构
            for ( var j = 0; j < div.childNodes.length; j++ )
                r[r.length] = div.childNodes[j];
        } else if ( a[i].length ) { //如果它是一个数组
            //假定它是 DOM 节点的数组
            for ( var j = 0; j < a[i].length; j++ )
                r[r.length] = a[i][j];
        } else { //否则，假定它是一个 DOM 节点
            r[r.length] = a[i];
        }
    }
    return r;
}
```

第二，你需要使那两个插入函数适于与修改后的 checkElem 协同工作，接受元素的数组，如程序 5-28 所示。

程序 5-28. 加强了的用于向 DOM 插入和追加内容的函数

[Copy to clipboard] [-]

CODE:

```
function before( parent, before, elem ) {
    //检查是否提供了父节点
    if ( elem == null ) {
        elem = before;
        before = parent;
        parent = before.parentNode;
    }
    //取得新的节点数组
    var elems = checkElem( elem );
    //倒序遍历数组,
    //因为我们正在往前插入元素
    for ( var i = elems.length - 1; i >= 0; i-- ) {
        parent.insertBefore( elems[i], before );
    }
}
function append( parent, elem ) {
    //得到元素数组
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
    var elems = checkElem( elem );
    //将它们全部追加给元素
    for ( var i = 0; i <= elems.length; i++ ) {
        parent.appendChild( elems[i] );
    }
}
```

现在，使用这些新函数，追加一个<li>元素给一个有序列表将变成一件极其简单的任务：

[Copy to clipboard] [-]

CODE:

```
append( tag("ol")[0], "<li>Mouse trap.</li>" );
//运行上面简单的一行代码将会给此<ol>元素追加 HTML：
<ol>
    <li>Cats.</li>
    <li>Dogs.</li>
    <li>Mice.</li>
</ol>
//将它变成下面这样：
<ol>
    <li>Cats.</li>
    <li>Dogs.</li>
    <li>Mice.</li>
    <li>Mouse trap.</li>
</ol>
//而为 before()函数运行一个相似的语句
before( last( tag("ol")[0] ), "<li>Zebra.</li>" );
//则会将原来的<ol>变成：
<ol>
    <li>Cats.</li>
    <li>Dogs.</li>
    <li>Zebra.</li>
    <li>Mice.</li>
</ol>
```

这的确有助于使你的代码更加简明清晰，利于开发。然而，要是你想走到上另一条道路，从 DOM 里删除节点呢？跟往常一样，也有处理这一问题的方法。

从 DOM 中删除节点

从 DOM 中删除结点几乎跟对应的创建和插入一样频繁。例如当为询问不限数目的项而动态地创建表单时，允许用户从页面中删除它们不想再处理的部分变得非常重要。删除节点的能力被压缩在一个函数中：removeChild。它像 appendChild 一样使用，但是具有相反的

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

效果。该函数看起来是这个样子的：

[Copy to clipboard] [-]

CODE:

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

你可以创建两个独立的函数来快速地删除节点，如程序 5-29 所示。

程序 5-29. 从 DOM 中删除一个节点的函数

[Copy to clipboard] [-]

CODE:

```
//从 DOM 中删除单个节点
function remove( elem ) {
    if ( elem ) elem.parentNode.removeChild( elem );
}
```

程序 5-30 展示了一个函数用来删除一个元素的所有子节点，只使用对 DOM 元素的一个引用。

程序 5-30. 删除一个元素的所有子结点的函数

[Copy to clipboard] [-]

CODE:

```
//从 DOM 中删除一个元素的所有子结点
function empty( elem ) {
    while ( elem.firstChild )
        remove( elem.firstChild );
}提示
```

作为一个例子，我们假设你要删除你在上一节里添加的<li>元素；此前你已经给了用户足够的时间的来浏览<li>元素现在可以不加提示地删除。下面的代码展示了你可以用来执行这种行动的 JavaScript 代码，得到你想到的结果：

[Copy to clipboard] [-]

CODE:

```
//从<ol>元素中删除最后一个<li>
remove( last( tag("ol")[0] ) )
//以上语句将会把
<ol>
<li>Learn Javascript.</li>
<li>???</li>
<li>Profit!</li>
</ol>
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
//转换成:
<ol>
<li>Learn Javascript.</li>
<li>???</li>
</ol>
//如果我们用 empty()代替 remove()
empty( last( tag("ol")[0] ) )
//它将简单地清空<ol>，剩下:
<ol></ol>
```

掌握了从 DOM 中删除节点的能力以后，你已经完成了文档对象模型怎样工作和怎样最大地利用它这一课。

本章摘要

在这一章里我论述了与文档对象模型相关的许多东西。不幸的是，主题中的一些(比如等待 DOM 加载)比其余的要复杂得多，而且在可预见的将来还将继续。尽管如此，使用你所学到的东西，你将几乎能够建立任何动态的 web 应用程序。

如果你想要看一些实际应用中的 DOM 脚本，请阅读附录 A，它包括了大量的附加的示例代码。另外，可以在线找到更多的 DOM 脚本示例，在本书的网站：

http://jspro.org，或者 Apress 网站的 Source Code/Download 区：

http://www.apress.com。接下来，我将把注意力转向非侵入式 DOM 脚本的另一个要素：事件。

nodeType

值	说明
1	元素节点
2	属性节点
3	文本节点
4	CDATA 选择节点
5	实体引用节点
6	实体节点
7	处理指令节点
8	注释节点
9	文档节点
10	文档类型节点
11	文档片断节点
12	符号节点

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

第六章：事件

非侵入的 DOM 脚本最重要的一个方面就是使用动态绑定的事件。编写可用性良好的 JavaScript 代码的最终目的是，不论用户使用何种浏览器和系统平台，页面总能够正常工作。为了达到这一点，你需要设定想要使用的一系列的功能，并排除掉任何不支持它们的浏览器。对于不支持的浏览器，给它们一个较少交互性但仍然功能完备的网站版本。以这种方式编写 JavaScript 与 HTML 交互的好处包括更干净的代码、更具可访问性的页面和更好的用户交互。这一切都依靠使用 DOM 事件来改善 web 应用程序中的交互来完成。

JavaScript 里事件的概念经过多年来的发展，达到了现在所处的可信赖的、半可用的水平。可喜的是，由于事件的通用的相似性的存在，你可以开发一些优秀的工具来帮且你构建功能强大的、编码清晰的 web 应用程序。

在这一章里，我将首先介绍 JavaScript 里事件是怎样工作的，并将它与其它语言里的事件进行一些比较。然后你将看到事件模型给你提供了些什么以及怎样最好地控制它。论述过向 DOM 元素绑定事件和可用的不同的事件类型以后，作为结尾，我将展示怎样将一些有效的非侵入的脚本技术整合到任何网页中。

JavaScript 事件简介

审视任何 JavaScript 代码的核心，你会发现正是事件是把所有东西融合在一些。在一个设计良好的 JavaScript 应用程序里，你将拥有数据源和它的视觉的表示(在 HTML DOM 内部)。为了同步这两个方面，你必须监视用户的交互动作并试图相应地更新用户界面。DOM 和 JavaScript 事件的结合是任何现代 web 应用程序赖以工作的至关重要的组合。

异步事件 vs.线程

JavaScript 里的事件系统是相当独特的。它完全地异步工作而根本不使用线程。这意味着你程序中的所有代码将依赖于其它的动作——比如用户的点击或者页面的加载——来触发。

线程化的程序设计与异步的程序设计根本的不同点在于你怎样等待事情发生。在线程化的程序里，你需要不停地反复检查条件是否满足了。而在异步程序里你只须简单地通过事件句柄注册一个回调函数，一旦事件发生，句柄就会通过执行回调函数来让你知道。我们来探索一下假如使用线程 JavaScript 程序将会怎么编写和实际使用异步回调函数 JavaScript 又是怎么编写的。

JavaScript 线程

按目前的情况来看，JavaScript 线程并不存在。你最多是使用 setTimeout 回调函数来模拟，但即使是那样，也并不理想。程序 6-1 中所示是一段假想的线程化的 JavaScript 代码，在其中你正在等待，直到页面完成加载。如果 JavaScript 真是一个线程化语言的语言，你将不得不做那样的事。

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

程序 6-3. 模拟线程的 JavaScript 伪码

[Copy to clipboard] [-]

CODE:

```
// 注意：这段代码不能生效!
// 在页面加载之前，持续地检查
while ( ! window.loaded() ) { }
//页面现在已经加载了，于是开始做些事情
document.getElementById("body").style.border = "1px solid #000";
```

如你所见，这段代码里有一个循环，一直在检查 window.loaded()是否返回 true。且不说 window 对象根本没有 loaded()这个函数，那样的循环也决不会在 JavaScript 中起作用。这是因为 JavaScript 中的循环是阻塞式的(也就是说它们运行完成之前别的什么事都不会发生)。假如 JavaScript 能够处理线程，你看到的情形将如图 6-1 所示。在图中，代码中的 while 循环持续地检查 window 是否已经加载。

图 6-1. 如果 JavaScript 能处理线程你将会看到什么

在实际的情况里，因为 while 循环持续地运行并阻断了应用程序的正常流程，true 值永远不可到达。结果是用户的浏览器将会停止响应并可能崩溃。由此可知，如果有任何人声称在 JavaScript 里用 while 循环等待动作能够成功，他要么是说着玩，要么是迷糊得厉害。

异步回调函数

使用线程不断检查更新的替代方案是使用异步的回调，这正是 JavaScript 所使用的。直白地说，你告诉一个 DOM 元素，当指定的事件发生，你想要一个函数被调用以处理它。这意味着你只提供一个对希望执行的代码的引用，而浏览器处理所有的细节。程序 6-2 展示了使用事件句柄和回调函数的一段简单的代码。你会看到在 JavaScript 里把一个函数绑定到事件句柄(window.onload)上所需要的实际的代码。一但页面加载完成，window.onload 就会被调用。其它通常的事件如 click,mousemove 和 submit 的情形也是这样。

程序 6-2. JavaScript 里的异步回调

[Copy to clipboard] [-]

CODE:

```
//注册一个页面加载完成时被调用的函数
window.onload = loaded;
//页面加载完成时被调用的函数
function loaded() {
    //页面现已加载完成了，于是干点事情
    document.getElementById("body").style.border = "1px solid #000";
}
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

将程序 6-2 的代码与 6-1 中的进行比较，你会看到显著的不同。唯一被立即执行的代码是将事件句柄(loaded 函数)向事件监听器(onload 属性)的绑定。一旦页面完全加载，浏览器将调用与 window.onload 相关联的函数并执行它。JavaScript 代码的流程如图 6-2 所示。图中展示了在 JavaScript 中使用回调函数来等待页面加载的一个图示。因为实际上不可能等待事情的发生，你将一个回调函数(loaded)注册到页面加载完成时会被调用的句柄(window.onload)上。

图 6-2. 使用回调函数等待页面加载的示意图

我们的简单的事件监听器和处理程序还没有立即显现的一个问题是，取决于事件类型和元素在 DOM 中位置的不同，事件会变得多样化并能以不同的方式来处理。下一节我们将看到事件的两个阶段及其不同点。

事件的阶段

JavaScript 事件分为两个阶段执行：捕获(capturing)和冒泡(bubbling)。这意味着当事件从一个元素触发时(比如，用户点击一个链接导致 click 事件被触发)，哪些元素允许处理它、以什么顺序处理它，变得多样化了。我们来看图 6-3 中的一个执行顺序的例子。图中说明了当用户点击页面中的第一个<a>元素时，哪些事件句柄以什么顺序被触发。

图 6-3. 事件处理的两个阶段

从这个简单的点击链接的例子里，你们可以看到事件的执行顺序。假设用户点击了一个<a>元素，文档的 click 句柄首先被触发，然后是<body>的句柄，然后是 的，等等，一直下行到<a>元素；这称为捕获阶段。此阶段完成以后，它又再次沿着树往上爬，<li>,<ul>, ,<body>,以及文档的事件句柄依次全部被触发。

为什么事件处理会以这种方式建立有着很特别的原因，它工作得也非常好。假设你想每一个<li>元素在用户把鼠标移到上面时会改变背景颜色，当鼠标移开时又变回来(这是许多菜单的一般需要)，程序 6-3 里的代码可以确切地做到这一点。

程序 6-3. 带鼠标悬停效果的标签导航方案

[Copy to clipboard] [-]

CODE:

```
//查找所有的<li>元素，并附以事件处理函数
var li = document.getElementsByTagName("li");
for ( var i = 0; i < li.length; i++ ) {
    //为<li>元素附加 mouseover 事件处理函数，
    //用来将元素的背景色改为蓝色
    li[i].onmouseover = function() {
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
        this.style.backgroundColor = 'blue';
    };
    //为<li>元素附加 mouseout 事件处理函数，
    //用来将元素的背景色改回缺省的白色
    li[i].onmouseout = function() {
        this.style.backgroundColor = 'white';
    };
}
```

这些代码会确实如你所设想的那样运作：鼠标移到<li>元素上，它的背景色会改变，把鼠标移开，颜色又将还愿。但是，你可能没有意识到的是，当你每次把鼠标移到<li>的时候你实际上切换了两个元素。因为<li>元素包含<a>元素，你的鼠标同样滑过了它，而不仅仅是<li>元素。我们来看看事件调用的精确的流程：

1. mouseover：鼠标到了<li>元素上
2. mouseout：鼠标从<li>移到了它所包含的<a>元素
3. <a> mouseover：鼠标现在到了<a>元素上
4. mouseover：<a>的 mousever 事件向上冒泡成为<li>的 mouseover

从事件调用的方式上你可能已经觉察到，你完全忽略了事件的捕获阶段；不用担心，我可没忘记它。你绑定事件监听器的方式是古老的"传统"方式：设置元素的 onevent 属性；它只支持事件冒泡，不支持捕获。事件的这一绑定方式及其它方式，将在下一主题论述。

除了事件调用的奇怪的顺序以外，你可能还注意到了两个意外的动作：鼠标移出<li>元素和<a>向<li>的 mouseover 冒泡。我们来仔细地看看。

第一个 mouseover 事件发生，因为如浏览器认为，你离开了父级<li>元素的范围，进入了另一个元素。这是因为当前位于最上层的元素(正如<a>相对于其父元素<li>)将会接收到鼠标即时的焦点。

<a>的 mouseover 向<li>元素的冒泡最终成就了我们那一段代码的优美。因为你实际上没有绑定任何种类的监听器给<a>元素，事件于是简单地沿着 DOM 上行，寻找另一个正在监听的元素。冒泡过程中它所遇到的第一个元素是<li>元素，恰巧监听着鼠标移入事件(这也正好是你想要的)。

你需要考虑的是，要是你确实给<a>元素的 mouseover 绑定了事件处理程序呢？有什么方法可以停止事件的冒泡吗？这是我将要论述的另一个重要的主题。

事件的一般特性

JavaScript 事件很好一面是，它们有着一些相对一致的特性，给予你开发时的更多的能力和控制。最简单和最古老的概念是事件对象，它给你一系列的元数据和上下文相关的函数，允许你处理诸如鼠标事件和键盘按键事件等。另外，有一些函数用来修改事件的通常的捕获/冒泡流程。深入学习这些特性可以让你事半功倍。

事件对象

事件处理函数的一个标准功能是以某种方式访问包含当前事件的上下文信息的事件对象。这

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

一对象在特定的事件中充当着非常有用的资源。比如，当处理键盘按下的事件时，你可以访问事件对象的 keyCode 属性，以得知被按下的是哪的键。在附录 B 中可以找到关于事件对象的更详细的说明。

然而，事件对象棘手之处在于，IE 的实现与 W3C 的规范并不相同。IE 有一个单独的全局事件对象(可以可靠地通过全局属性 window.event 访问)，而其它的每一种浏览器都把事件对象作为单个的参数传递给事件处理函数。可靠地使用事件对象的一个例子见程序 6-4，代码修改一个普通的<textarea>元素，使其行为发生了改变。典型地，用户可以在<textarea>里按下回车键，产生一个换行符。但是假如你不希望那样做呢？函数正是提供了这一功能。

程序 6-4. 使用 DOM 事件重写功能

[Copy to clipboard] [-]

CODE:

```
//找到页面里的第一个<textarea>并为它绑定 kerpress 监听器
document.getElementsByTagName("textarea")[0].onkeypress = function(e){
    //如果不存在事件对象，就抓取那个全局的(ie only)
    e = e || window.event;
    //如果按下了回车键，返回 false(导致它什么也不干)
    return e.keyCode != 13;
};
```

事件对象包含了大量的属性和函数，且它们的命名与行为在浏览器之间各不相同。我不想现在就进入那些细节，但是我强烈建议你阅读附录 B，那是所有的事件对象属性的一个的列表，包括使用方法以及实际使用中的例子。

this 关键字

this 关键字(见第二章)提供了一种在函数作用域中的访问当前对象的方式。现代浏览器使用 this 关键字给所有的事件处理函数提供上下文信息。它们中只有一部分(而且只有部分方法)良好地运行，将它设为当前对象；这将很快地被深入讨论到。例如，在程序 6-5 中，我可以利用这一事实，只建立一个通用的函数来处理所有点击而通过 this 关键来确定作用于哪一个元素，它将如预期地工作。

程序 6-5. 点击时改变<li>元素的背景色

[Copy to clipboard] [-]

CODE:

```
//查看所有的<li>元素并给每一个绑定 click 处理函数
var li = document.getElementsByTagName("li");
for ( var i = 0; i < li.length; i++ ) {
    li[i].onclick = handleClick;
}
//click 处理函数，调用时改变特定元素的     前景色和背景色
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
function handleClick() {
    this.style.backgroundColor = "blue";
    this.style.color = "white";
}
```

this 关键字的确只是为了方便而设的，但我想你会发现，当使用它的属性，将会极大地降低你的 JavaScript 代码的复杂性。在本书中，我将试图使用 this 关键字编写所有的事件相关的代码。

取消事件冒泡

知道了事件的捕获/冒泡怎样工作以后，我们再来探讨怎样控制它。前面的例子里引入的一个很重要的问题是，如果你想要一个事件只在其目标上而不在基父级元素上出现，你处配办法停止它。阻止事件冒泡的过程将导致出现如图 6-4 所示的情形，在其中被第一个<a>元素捕获的事件的后续的冒泡被取消。

图 6-4. 第一个<a>元素所捕获的事件被取消的结果

停止事件的冒泡(或捕获)被证明在复杂的应用程序中是极其有用的。不幸的是，IE 提供了一种与所有其它浏览器不同的方式来阻止事件冒泡。程序 6-6 是一个通用的取消事件冒泡的函数。该函数接受单个参数：传递到事件处理程序的事件对象。该函数处理取消事件冒泡的两种方式：标准的 W3C 方式和非标准的 IE 方式。

程序 6-6. 停止事件冒泡的通用函数

[Copy to clipboard] [-]

CODE:

```
function stopBubble(e) {
    //如果提供了事件对象，则这是一个非 IE 浏览器
    if ( e && e.stopPropagation )
        //因此它支持 W3C 的 stopPropagation()方法
        e.stopPropagation();
    else
        //否则，我们需要使用 IE 的方式来取消事件冒泡
        window.event.cancelBubble = true;
}
```

现在你可能想知道的是，什么时候我想要阻止事件冒泡？老实说，多数时间里你可能从来不需担心这个。当你开始开发动态的应用程序(尤其是需要处理键盘和鼠标事件)时，这一需求才会变得突出。

程序 6-7 展示了一个简明的代码片段：为你鼠标悬停的当前元素加上红色边框。如果不阻止事件冒泡，每一次你把鼠标移动到一个元素时，该元素及其所有的父级元素都将有一个

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

并非我们想要的红色的边框。

程序 6-7. 使用 stopBubble()创建一系列交互式的元素

[Copy to clipboard] [-]

CODE:

```
//查找并遍历 DOM 中的所有元素
var all = document.getElementsByTagName("*");
for ( var i = 0; i < all.length; i++ ) {
    //监视用户何时把鼠标移到元素上，
    //为该元素添加红色边框
    all[i].onmouseover = function(e) {
        this.style.border = "1px solid red";
        stopBubble( e );
    };
    //监视用户何时把鼠标移出元素，
    //删除我们所添加的红色边框
    all[i].onmouseout = function(e) {
        this.style.border = "0px";
        stopBubble( e );
    };
}
```

拥有阻止事件冒泡的能力，你就能对事件到达哪个元素并进行处理有了完全的控制。这是开发动态的 web 应用程序所需的一个非常工具。最后一点，取消浏览器的默认动作，这允许你完全改写浏览器的行为并实现新的功能以替代之。

改写浏览器的默认动作

对于发生的大多数事件，浏览器有一些总会发生的默认动作。比如说，点击一个<a>元素将会把你带到它所关联的网页；这是浏览器的一个默认动作。这一动作总是在事件的捕获和冒泡阶段都完成以后发生，如图 6-5 所示。该示例说明了用户在页面点击<a>元素的结果。事件起初经历在 DOM 中的捕获和冒泡阶段(如前所述)。然而，一旦事件完成了其旅程，浏览器将试图执行该事件及元素的默认动作。在这里也就是访问链接的网页。

图 6-5. 事件的完整生命周期

默认动作可以概括为浏览器所执行的你没有明确指定的操作。下面是特定事件发生时几种不同类型的默认动作：

a. 点击一个<a>元素将会跳转到元素的 href 属性指定的 URL。
b. 按下 Ctrl+S 键，浏览器将试图保存当前网页。
c. 提交一个 HTML<form>元素将从指定 URL 查询数据并将浏览器重定向到该地址。

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

d. 移动鼠标到带有 alt 或 title 属性(取决于不同的浏览器)的<img>元素将导致出现一个工具提示，提供该<img>元素的描述。

即使你阻止了事件的冒泡或者根本没有设置事件处理函数，上述事件也会被浏览器执行。这在你的脚本中会导致显著的问题。如果你想让你的表单有不同的行为呢？或者如果你想要<a>元素以不同于它们本来目的的方式运作呢？因为取消事件冒泡不足以阻止默认行为，你需要一些特别的代码在直接处理它们。跟取消事件冒泡一样，有两种方式来阻止默认动作的发生：IE 特有的方式和 W3C 方式。两种方式见于程序 6-8。其中展示的函数接受单个参数：传递到事件处理函数的事件对象，使用方法如 return stopDefault(e)；——当你的处理函数也需要返回 false(这是 stopDefault 为你所返回的)时。

程序 6-8. 阻止浏览器默认动作发生的通用函数

[Copy to clipboard] [-]

CODE:

```
function stopDefault( e ) {
    //阻止默认浏览器动作(W3C)
    if ( e && e.preventDefault )
        e.preventDefault();
    //IE 中阻止函数器默认动作的方式
    else
        window.event.returnValue = false;
    return false;
}
```

使用 stopDefault 函数，你现在可以阻止浏览器给出的任何默认动作。这允许你用脚本为用户编写出灵巧的交互，如程序 6-9 所示。此代码使一个页面内所有的链接在一个自包含的<iframe>中加载，而不是打开整个新的页面。这么做可以使你把用户保持在页面上，并可能给出更具交互性的体验。

注意：在 95%的情况下，阻止默认动作会生效。然而事情在你跨越浏览器时会变得棘手起来，因为阻止默认事件取决于浏览器(它们并不总能做对)，尤其是当阻止文本输入框里的按键事件的动作和阻止 iframe 里的动作时；除了这些以外，应该还是足够健全的。

程序 6-9. 使用 stopDefault()来改写浏览器功能

［code]

```
//假设页面中已经有一个 ID 为 iframe 的<iframe>元素
var iframe = document.getElementById("iframe");

//查找页面中的所有<a>元素
var a = document.getElementsByTagName("a");
for ( var i = 0; i < a.length; i++ ) {

    //为<a>元素绑定事件处理函数
    a[i].onclick = function(e) {
        //设置 iframe 的 location
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
            iframe.src = this.href;

            //阻止浏览器访问<a>元素所指定的网页(默认动作)
            return stopDefault( e );
        };

}
```
[/code]

改写默认事件绝对是共同组成了非侵入式脚本的 DOM 和事件的关键所在。在本章后面的"非侵入的 DOM 脚本"中我将立足于功能，更多地谈到这一点；当你实际地将事件处理函数绑定到 DOM 元素时，争论的要点出现了。事实上有三种绑定事件的方式，其中一些比另一些要好。下一节将会讨论它们。

绑定事件监听器

怎样将事件处理程序绑定到元素是 JavaScript 里一直以来不断推进的追求。起初，浏览器强制用户将处理代码内联地写在 HTML 文档中。好在那一些技术已经变得远远过时了(说这是一件好事，是考虑到它与非侵入的 DOM 脚本里数据抽象的精神相悖)。

当 IE 与 NetScape 激烈竞争的时候，它们各自开发出两个独立但又非常相似的注册事件的模型。最终 NetScape 的模型被修改成为 W3C 标准，而 IE 的则保持不变。

于是乎，目前存在三种可用的注册事件的方式。传统方式是老式的内联附加事件处理函数方式的一个分支，但是它很可靠而并能一致地工作。另外两种是 IE 和 W3C 的注册事件的方式。最后，我将给出一套可靠的方法，开发者可以用它们来注册和注销事件而不需再担心底层是什么浏览器。

传统绑定

传统的绑定事件的方式是我在本章中到目前为止所一直使用的。它是到目前为止最简单最兼容的绑定事件处理程序的方式。使用这种方式时，你只需将函数作为一个属性附加到你想要监视的 DOM 元素上。6-10 展示了使用传统方式绑定事件的一些例子。

程序 6-10. 使用传统的事件绑定方式附加事件

[Copy to clipboard] [-]

CODE:

```
//找到第一个<form>元素并为它附加"提交"事件处理函数
document.getElementsByTagName("form")[0].onsubmit = function(e){
    //阻止表单提交
    eturn stopDefault( e );
};
//为文档的 body 元素附加一个按键事件处理函数
document.body.onkeypress = myKeyPressHandler;
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
//为页面的加载事件附加一个处理函数
window.onload = function(){ … };
```

这一技术有一系列的优势和缺点，使用时必须注意。

传统绑定的优势：

a. 使用传统绑定的最大的好处在于它无比地简单和一致，也就是说在很大程度上它能保障无论使用什么浏览器都能生效。

b. 当处理事件时，this 关键字指向当前的元素，这一点是非常有用的(如程序 6-5 示范的那样)。

传统绑定的缺点：

a. 传统绑定只作用于事件冒泡，而非捕获和冒泡。

b. 只能每次为一个元素绑定一个事件处理函数。当使用流行的 window.onload 属性时，这将会潜在地导致令人困惑的结果(因为它会覆盖其它的使用相同方法绑定的代码片段)。程序 6-11 展示了这一问题的一个实例，一个新的事件处理函数覆盖了原来的事件处理函数。

c. event 对象参数只在非 IE 浏览器上有效

程序 6-11. 事件处理函数互相覆盖

[Copy to clipboard] [-]

CODE:

```
//绑定初始的 load 处理函数
window.onload = myFirstHandler;
//在某个地方，你所引用的其它库里，你的第一个处理函数被覆盖，
//页面加载完成时只有 mySecongHandler 函数被调用
window.onload = mySecondHandler;
```

懂得了盲目覆盖其它事件的可能性，你可能会选择只在可以信任所有其它的代码简单的情况下使用传统绑定。解决这一混乱情况的一种方式是使用浏览器提供的现代绑定方法。

DOM 绑定：W3C

W3C 的为 DOM 元素绑定事件处理函数的方法是这方面唯一真正的标准方式。除了 IE，所有其它的现代浏览器都支持这一事件绑定的方式。

附加新的处理函数的代码很简单。它作为每一个 DOM 元素的名为 addEventListener 的方法存在，接收 3 个参数：事件的名称(如 click)，处理事件的函数，以及一个来用使用或禁用事件捕获的布尔标志。程序 6-12 展示一个实际使用 addEventListener 的例子。

程序 6-12. 使用 W3C 方式绑定事件处理函数的示例代码片段

[Copy to clipboard] [-]

CODE:

```
//找到第一个<form>元素并为它附加"提交"事件处理函数
document.getElementsByTagName("form")[0].addEventListener('submit',function(
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
e){
    //阻止表单提交
    return stopDefault( e );
}, false);
//为文档的 body 元素附加一个按键事件处理函数
document.body.addEventListener('keypress', myKeyPressHandler, false);
//为页面的加载事件附加一个处理函数
window.addEventListener('load', function(){ … }, false);
```

W3C 绑定的优势：

1. 这一方法同时支持事件处理的冒泡和捕获阶段。事件的阶段通过设置 addEventListener 的最后一个参数为 false(指示冒泡)或 true(指示捕获)来切换。
2. 在事件处理函数内部，this 关键字引用当前元素。
3. 事件对象总是作为事件处理函数的第一个参数被提供。
4. 你可以绑定任意多个函数到一个元素上，而不会覆盖先前所绑定的。

W3C 绑定的缺点

1. 它在 IE 里面无效。你必须使用 IE 的 attachEvent 函数来代替。

如果 IE 采用了 W3C 的方法来绑定事件处理函数，这一章将会比现在短得多，因为那将会不再需要讨论绑定事件的替代方法。然而，到目前为止，W3C 的事件绑定方法仍然是最可理解和最易使用的。

DOM 绑定：IE

在许多方面，IE 的绑定事件的方式看起来跟 W3C 的非常相似。但是，当你触及细节的时候，它又在某些方面有着非常显著的不同。程序 6-13 是 IE 中绑定事件处理函数的一些例子。

程序 6-13. 使用 IE 的方式绑定事件处理函数的示例

[Copy to clipboard] [-]

CODE:

```
//找到第一个<form>元素并为它附加"提交"事件处理函数
document.getElementsByTagName("form")[0].attachEvent('onsubmit',function(){
    //阻止表单提交
    return stopDefault();
});
//为文档的 body 元素附加一个按键事件处理函数
document.body.attachEvent('onkeypress', myKeyPressHandler);
//为页面的加载事件附加一个处理函数
window.attachEvent('onload', function(){ … });
```

IE 绑定的优势

1. 你可以绑定任意多个函数到一个元素上，而不会覆盖先前绑定的。

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

IE 绑定的缺点

1. IE 只支持事件的冒泡阶段。

2. 事件监听函数内部的 this 关键字指向 window 对象，而非当前函数(这是 IE 的巨大败笔)。

3. 事件对象只能从 widnow.event 得到。

4. 事件名称必须形如 onxxxx——比如，要使用"onclick"而不能只是"click"。

5. 它只对 IE 有效。对于非 IE 平台，你必须使用 W3C 的 addEventListener。

相对其半标准的事件特性，IE 事件绑定的实现是严重短缺的。鉴于它的许多不足之处，弥补方案必须继续存在以强制它合理的运转。然而，幸运的是，向 DOM 添加事件的通用的函数确实存在，它能极大的减轻我们的痛苦。

addEvent 和 removeEvent

addEvent 还有一个重大的缺点就是：使用 addEvent 为这个元素添加某事件处理方法后，再直接在该元素上使用传统绑定某事件，就会导致前面注册的事件全部被覆盖！

2005 年末，Peter-Paul Koch(http://quirksmode.org)发起了一个竞赛，向 JavaScript 代码编写者公开征求一对新的函数，addEvent 和 removeEvent，用来提供一个可靠的方式为 DOM 元素添加和删除事件。最终我以一段非常简练而又能足够好地运行的一段代码在其中获胜。但是，后来，其中一位裁判(Dean Edwards)给出了函数的另一个版本，远远地超越了我所编写的。它的实现使用传统的方式来附加事件处理函数，完成忽略现代方法。因此，它可以在大量的浏览器上运行，而仍然提供了必要的事件的优美性(比如 this 关键字和标准的事件对象)。程序 6-14 展示的一段示例代码很好的利用新的 addEvent 函数，使用了事件处理的所有的不同侧面，包括浏览器默认动作的阻止，正确的事件对象的引入，和正确的 this 关键字的引入。

程序 6-14. 使用 addEvent 函数的示例代码片段

[Copy to clipboard] [-]

CODE:

```
//等待页面加载完成
addEvent( window, "load", function(){
    //监视用户的任何按键
    addEvent( document.body, "keypress", function(e){
        //如果用户按下了 Ctrl+Spance
        if ( e.keyCode == 32 && e.ctrlKey ) {

            //显示我们的特别的表单
            this.getElementsByTagName("form")[0].style.display = 'block';
            //确保没有怪异的事情发生
            e.preventDefault();
        }
    });
});
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

addEvent 函数提供了一个绝妙的简单而强大的方式来处理 DOM 事件。只要看看优势和不足，就可以明显地看出这一函数可以作为一致而可靠的方式来处理事件。程序 6-15 是完整的源代码，它能在所有的浏览器中运行，不泄露任何内存，处理了 this 关键字和事件对象，并标准化了事件对象。

程序 6-15. Dean Edwards 编写的 addEvent/removeEvent 库

[Copy to clipboard] [-]

CODE:

```
// addEvent/removeEvent written by Dean Edwards, 2005
// with input from Tino Zijdel
// http://dean.edwards.name/weblog/2005/10/add-event/
function addEvent(element, type, handler) {
    //为每一个事件处理函数分派一个唯一的 ID
    if (!handler.$$guid) handler.$$guid = addEvent.guid++;
    //为元素的事件类型创建一个哈希表
    if (!element.events) element.events = {};
    //为每一个"元素/事件"对创建一个事件处理程序的哈希表
    var handlers = element.events[type];
    if (!handlers) {
        handlers = element.events[type] = {};
        //存储存在的事件处理函数(如果有)
        if (element["on" + type]) {
            handlers[0] = element["on" + type];
        }
    }
    //将事件处理函数存入哈希表
    handlers[handler.$$guid] = handler;
    //指派一个全局的事件处理函数来做所有的工作
    element["on" + type] = handleEvent;
};
//用来创建唯一的 ID 的计数器
addEvent.guid = 1;
function removeEvent(element, type, handler) {
    //从哈希表中删除事件处理函数
    if (element.events && element.events[type]) {
        delete element.events[type][handler.$$guid];
    }
};
function handleEvent(event) {
    var returnValue = true;
    //抓获事件对象(IE 使用全局事件对象)
    event = event || fixEvent(window.event);
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
    //取得事件处理函数的哈希表的引用
    var handlers = this.events[event.type];
    //执行每一个处理函数
    for (var i in handlers) {
        this.$$handleEvent = handlers[i];
        if (this.$$handleEvent(event) === false) {
            returnValue = false;
        }
    }
    return returnValue;
};
//为 IE 的事件对象添加一些"缺失的"函数
function fixEvent(event) {
    //添加标准的 W3C 方法
    event.preventDefault = fixEvent.preventDefault;
    event.stopPropagation = fixEvent.stopPropagation;
    return event;
};
fixEvent.preventDefault = function() {
    this.returnValue = false;
};
fixEvent.stopPropagation = function() {
    this.cancelBubble = true;
};
```

addEvent 函数的优势

1. 它可以在所有浏览器上工作，甚至是很老的不被支持的浏览器
2. this 关键字对所有的绑定的函数可用，指向当前元素
3. 浏览器特有的阻止浏览器默认动作和停止事件冒泡的函数都被统一了
4. 不管是哪种浏览器，事件对象总是作为第一个参数传给处理函数

addEvent 函数的缺点

1. 它只能工作于冒泡阶段(因为它在底层使用了传统事件绑定方法)

考虑到 addEvent/removeEvent 函数是如此的强大，绝对没有理由不在你的代码中使用它们。在 Dean 的代码的基础上，添加一些诸如更好的事件对象标准化、事件触发、大量事件删除这类在通常的事件结构中原本极难的事情委实是轻而易举。

事件的类型

JavaScript 事件可以被归入几种不同的类别。最常用的类别可能是鼠标交互事件，然后是键盘和表单事件。下面的列表提供了 web 应用程序中存在并可被处理的不同各类的事件的粗略预览。参考附录 A 和附录 B，可以得到大量的事件的实例。

鼠标事件: 分为两种，追踪鼠标当前位置的事件(mouseover,mouseout)，和追踪鼠标在哪儿被点击的事件(mouseup,mousedown,click)。

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

键盘事件: 负责追踪键盘的按键何时以及在何种上下文中(比如说，追踪一个 form 元素内的按键相对于出现在整个页面的按键)被按下。与鼠标相似，三个事件用来追踪键盘：keyup,keydown,keypress。

UI 事件: 用来追踪用户何时从页面的一部分转到另一部分。例如，使用它你能够可靠地知道用户何时开始在一个表单中输入。用来追踪这一点的两个事件是 focus 和 blur(用于对象失去焦点时)。

表单事件: 直接与只发生于表单和表单输入元素上的交互相关。submit 事件用来追踪表单何时提交；change 事件监视用户向元素的输入；select 事件当<select>元素被更新时触发。

加载和错误事件: 事件的最后一类是与页面本身有关的，关注页面的加载状态。它们被关联到何时用户第一次加载页面(load 事件)和最终离开页面(unload 和 beforeunload 事件)。另外，JavaScript 错误使用 error 事件追踪，这给了你以独立处理错误的能力。

记住这些大致的事件分类，我推荐你积极地查看附录 A 和附录 B 的材料，其中剖析了所有的常用的事件：它们怎样工作，在不同的浏览器中有着怎样的差别，并描述了使它们如你所希望的那样工作所需的所有复杂细节。

非侵入的 DOM 脚本

到目前为止你所学到的每样东西都指向这一重要的目标：编写 JavaScript 代码使它与你的用户自然而非侵入地交互。这一脚本风格背后的驱动力量是，你可以将你的精力集中于编写良好的代码，它们运行于现代浏览器而在老的(不支持的)浏览器里则优雅地隐退。

为实现这一点，你可以结合已经学到的三种技术来使一个应用程序以非侵入的形式被构造：

1.应用程序中的所有功能都应经过验证。比如说，如果你希望访问 HTML DOM，你需要验证它存在且具有你需要使用的所有的功能(如 if(document&&document.getElementById))。这一技术在第二章讨论过。

2.使用 DOM 来快速而一致地访问你文档中的元素。因为你已经知道浏览器支持 DOM 函数，你可以放心地简单的编写代码而无需 hack 或者拼凑。

3.最后，使用 DOM 和你的 addEvent 函数动态地将所有事件绑定到文档中。不要让代码中出现这样的东西：...。用非侵入编码的角度来看，这是非常不好的，如果 JavaScript 被关闭或者用户使用了不支持的老版本的浏览器，这些代码就是一堆垃圾。因为你只是将用户指向了没有意义的 URL，它将不能对不支持你脚本功能的用户提供任何有效的交互。

如果这还不够明显，你需要假设用户根本没有安装 JavaScript，或者他的浏览器在某方面很低能。打开你的浏览器，访问你最喜欢的站点，划关闭 JavaScript：它还能工作吗？禁用所有的 css 又如何呢？你还能导航到你想到的那个位置吗？最后，不使用鼠标能否访问你的站点？所有这些都应该是你的网站的最终目标的一部分。可喜的是，因为你已经建立起一个对于怎样编写真正有效的 JavaScript 代码的良好的认识，这一过渡的代价是可以忽略，通过最少的努力就能实现。

假设 JavaScript 被禁用

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

你应该达到的第一个目标是彻底删除 HTML 文档中所有的内联事件绑定。可以检查文档里几个经常出问题的方面：

a. 如果你禁用了页面上的 JavaScript，并点击任何/所有链接，它们能把你带到一个网页吗？开发者可能会频繁地使用形如 href=""或 href="#"的 URL，这意味着它们运行一些额外的 JavaScript 伎俩来为用户的提供结果。

b. 如果你禁用了 JavaScript，你所有的表单都还能正常工作和提交吗？当使用<select>作为动态菜单时(那只能在 JavaScript 使能的情况下工作)，通常就会出问题。

遵从使用这些重要的训诫，你将得到对禁用了 JavaScript 和仍然使用不被支持的老浏览器的用户来说完全可以使用的网页。

确保链接不依赖 JavaScript

因为用户可以在页面上执行所有的动作，你需要确保用户在任何动作执行之前都得到了恰当的通知。当 Google 发布它的 Google Accelerator(遍历页面上的所有链接并为你缓存它们)时，用户发现它们的 e-mail，帖子，和消息被不加明显提示地魔法般地删除了。这是因为开发者在它们的页面中放置了用来删除消息的链接，并通过弹出对话框(使用 JavaScript)来确认删除。但是 Google Accelerator 完成忽略了那些弹出对话框，如它应该做的那样，终究访问了那些链接。

这一情境是向你指引用于在网上传输所有的文档和文件的 Http 规范的一个巧妙的方式。最简单的情形里，GET 请求产生于点击链接时，POST 请求发生于提交表单时。这个规范的开头就说到，GET 请求不应该产生破坏性的副作用(比如删除消息)，这就是 Google Accelerator 自行其是的原因。可见问题并不是出在 Google 方面的不良的程序设计，而在于创建了那些链接的 web 应用程序的开发者方面。

简单地说，你站点上的所有链接都应该是非破坏性的。如果通过点击链接你可以删除、编辑、或者修改任何用户拥有的数据，你或许应该使用表单来实现那一目的。

监视 CSS 何时被禁用

新老浏览器的交叉点是一个特别棘手的情况：浏览器太老而不能支持现代 JavaScript 技术但是又足够新可以支持 CSS 样式。一种流行的 DHTML 技术是起初将一个元素隐藏(通过设置 display 为 none 或者 visibility 为 hidden)，然后在用户进入页面时(使用 JavaScript)使它逐渐显示。但是，如果用户禁用了 JavaScript，他将永远看不到那个元素。程序 6-16 给了一个针对这一问题的解决方案。

程序 6-16. 提供一个即使 JavaScript 被禁用也不会失败的加载渐显技术

[Copy to clipboard] [-]

CODE:

```
<!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" />
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
    <!--一旦脚本运行，一个新的类将被附加到<html>元素上，可以藉此判断 JavaScript 是否可用
-->
    <script>document.documentElement.className = "js";</script>
    <!--如果 JavaScript 可用，则隐藏稍后我们将渐显的文本块-->
    <style>.js #fadein { display: none }</style>
</head>
<body>
    <div id="fadein">Block of stuff to fade in…</div>
</body>
</html>
```

这一技术超越了简单的渐显 DHTML。对 web 开发者来说，知道 JavaScript 是否可用并据此来应用样式是很有益的。

事件的可访问性

开发纯粹的非侵入的 web 应用程序时应该注意的最后一点是确保你的事件在不使用鼠标的情况下也能工作。做到这一点，你帮助了两类人：需要访问辅助的(有视觉缺陷的)用户，和不喜欢使用鼠标的人。某一天功夫从电脑上拔掉鼠标，坐下来学习怎样只使用键盘访问网站，这绝对是很有启发性的体验。

为了确保你的 JavaScript 事件更易访问，任何时候你使用 click、mouseover 和 mousemove 时，你需要周密地考虑提供非鼠标绑定的替代方案。好在存在着快速补救这一情形的容易的方式：

click 事件: 浏览器开发者方面的一个巧妙的举措是使得当 Enter 键被按下，click 事件就会触发，完全省去了为这一事件提供替换方案的必要性。但是，应该注意的一点是，一些开发者喜欢为提交按钮绑定处理函数来监视用户何时提交一个网页。替代使用该事件，开发者应该绑定到 form 对象的 submit 事件上，这是一个明智而可靠的替代方案。

mouseover 事件: 当使用键盘导航网页时，你实际是在把焦点移到不同的元素。通过同时附加事件处理函数到 mouseover 和 focus 事件上，你可以确保对键盘和鼠标用户都有一个相当的解决方案。

mouseout 事件: 与用于 mouseover 事件的 focus 事件对应，blur 事件在用户焦点移出一个元素时发生。你可以使用 blur 事件作为用键盘模拟 mouseout 事件的一种方式。

学会了哪些事件对能够如你所想地运行，现在你可以重拾程序 6-3，建立一个即使没有鼠标也能生效的类似鼠标悬停效果，如程序 6-17 所示。

程序 6-17. 为元素的成对的事件附加处理函数以改善网页的可访问性

[Copy to clipboard] [-]

CODE:

```
// 找到所有的<a>元素，以向其附加事件处理函数
var li = document.getElementsByTagName("a");
for ( var i = 0; i < a.length; i++ ) {
    //为<a>元素附加 mouseover 和 focus 事件处理函数，
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
    //当用户鼠标移到链接上或者使用键盘将移动焦点到链接上时，改变<a>的背景色为蓝色
    a[i].onmouseover = a[i].onfocus = function() {
        this.style.backgroundColor = 'blue';
    };
    //为<a>元素附加 mouseout 和 blur 事件处理函数，
    //当用户鼠标移出该链接或者使用键盘使链接失去焦点时，将<a>的背景色为改回白色
    a[i].onmouseout = a[i].onblur = function() {
        this.style.backgroundColor = 'white';
    };
}
```

实际上，在典型的鼠标事件以外再添加处理键盘事件的能力也费不了多少事。即使没有别的用处，也有助于作为依赖键盘的用户更好的访问你的站点的一种方式，这对每个人来说都是件好事。

本章摘要

学会了怎样穿行于 DOM，为 DOM 元素绑定事件处理函数，并且了解了非侵入地编写 JavaScript 代码的益处之后，现在你已经可以开始着手对付一些更大的应用程序和更酷的效果了。

在这一章，我以对 JavaScript 中事件怎样工作的介绍开头，并将它们与其它语言里的事件模型进行了比较。随后你看到了事件模型提供了什么信息以及怎样最好的控制它。然后我们探索了绑定事件到 DOM 元素的方法，以及可用的不同的事件类型。作为结尾，我介绍了怎样把有效的非侵入的脚本技术整合到任何网页中。

接下来你将会看到怎样利用你刚刚学到的这些技术实现一些动态的交互。

第七章：JavaScript 与 CSS

JavaScript 和 CSS 的交互是现代 JavaScript 程序设计的支柱。事实上对于所有的现代 web 应用程序来说，至少使用某些形式的动态交互是必须的。那么做过之后，用户可以更快地操作而更少地把时间浪费在等待页面加载上。将动态技术与第六章提出的事件方面的观念相结合，对于实现无缝而强大的用户体验是非常重要的。

层叠式样式表是用来对易用的、有吸引力的网页进行修饰和布局的事实标准，它在给用户提供最少的困难的同时为开发者提供最多的能力。当你将那一能力与 JavaScript 相结合时，你将能够构造强健的用户界面，包括动画、窗口部件(widgets)，或动态显示等。

访问样式信息

JavaScript 与 CSS 的结合全部是以表现作为结果的交互。理解什么对你是可用的，对于精确地达到你想要的交互非常重要。

用来设置和获取元素的 CSS 属性的主要工具是其 style 属性。比如说，如果想要取得一

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

个元素的高度，你可以编写如下代码：elem.style.height。如果你想要设置元素的高度为某个特定值，你可以执行如下代码：ele.style.height="100px"。

当处理 DOM 元素的 CSS 属性时，有两个会碰到的问题，它们并不像一般人所期望的那样运作。首先，JavaScript 要求你在设置任何空间尺度时指明单位(就像前面设置高度时所做的那样)。同时，任何空间属性也将返回一个代表元素属性的字符串而非数字(如"100px"而非 100)。第二，如果一个元素高为 100 像素，而你试图获取它的当前高度，你期望从 style 属性里取得那个"100px"，情况却未必会如你所愿。这是因为任何样式表文件或内联 CSS 预设的样式信息并不能可靠地反映到 style 属性上。

这一状况将我们引向 JavaScript 中处理 CSS 的一个重要函数：获取一个元素真正的当前样式属性的方法，给你一个预期的精确值。存在一组(来源于 W3C 和 IE 特有的变种)相当可靠的方法可以用来得到 DOM 元素的真正的计算后的样式属性。它们能顾及所有相关的样式表、元素特定属性以及 JavaScript 所作的修改。当需要得到你正操作的元素的精确视图信息时这些方法将会是极其有用的。

获取元素的计算后样式值时应该考虑到存在于不同的浏览器间的大量的差异。跟在大多数的情形一样，IE 有它自己的方法，而其它所有的浏览器都使用 W3C 定义的方式来实现。

程序 7-1 给出了一个用来找出元素的计算后样式属性值的一个函数，7-2 则给出了使用此函数的一个示例。

程序 7-1. 用来得到元素的计算后的实际 CSS 样式值的一个函数

[Copy to clipboard] [-]

CODE:

```
//获取一个特定元素(elem)的样式属性(name)
function getStyle( elem, name ) {
    //如果该属性存在于 style[]中，则它最近被设置过(且就是当前的)
    if (elem.style[name])
        return elem.style[name];
    //否则，尝试 IE 的方式
    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) {
        //它使用传统的"text-Align"风格的规则书写方式，而不是"textAlign"
        name = name.replace(/([A-Z])/g,"-$1");
        name = name.toLowerCase();
        //获取 style 对象并取得属性的值(如果存在的话)
        var s = document.defaultView.getComputedStyle(elem,"");
        return s && s.getPropertyValue(name);
    //否则，就是在使用其它的浏览器
    } else
    return null;
}
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

程序 7-2.　元素的计算后的 CSS 样式值未必是 style 对象里可用值的一种情况

[Copy to clipboard] [-]

CODE:

```
<html>
<head>
    <style>p { height: 100px; }</style>
    <script>
    window.onload = function(){
        //找到欲检查高度的段落对象
        var p = document.getElementsByTagName("p")[0];
        //使用传统方式检查其高度
        alert( p.style.height + " should be null" );

        //检查计算后的高度值
        alert( getStyle( p, "height" ) + " should be 100px" );
    };
    </script>
</head>
<body>
    <p>I should be 100 pixels tall.</p>
</body>
</html>
```

程序 7-2 说明了怎样得到一个 DOM 元素的实际的 CSS 属性值。在这种情形里你得到的是元素的实际的像素高度，即使其高度是通过头部的 CSS 来设定的。应该注意的是，你的函数将会忽略度量的单位(比如使用的是百分比)。尽管这一解决方案并不是绝对安全的，它的确是一个良好的出发点。

动态的元素

动态元素的隐含的意思也就是使用 JavaScript 和 CSS 维护或创建的非静态的元素。简单的例子是指示你对时事通讯感兴趣的复选框和弹出式的 e-mail 输入域。

在最基本的层面上，有三个关键的属性可用来构造动态效果：位置、尺寸、可见性。使用这三个属性你可以在现代浏览器上模拟大多数常见的用户交互效果。

元素的位置

操作元素的位置是在页面中开发动态元素的一个重要构成部分。访问和修改 CSS 位置属性让你有效地模拟许多流行的动画和交互效果(比如拖放)。

知道 CSS 的定位系统是怎样工作是操作元素位置的一个重要步骤。在 CSS 中，元素使用偏移来定位，使用的度量是相对父元素的左上角的偏移量。图 7-1 是 CSS 中使用的坐标系统的一个例子。

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

图 7-1. 使用 CSS 的网页里坐标系统示例

页面上的所有元素都有着某种形式的 top(垂直坐标)和 left(水平坐标)偏移。大体来说，多数元素简单地根据其周围的元素静态定位。依照 CSS 标准的提议，一个元素可以有几种不同的定位方案。为了正好地理解这一点，我们来看看程序 7-3 所示的一个简单的网页。

程序 7-3. 演示使用不同的定位方案的一个网页

[Copy to clipboard] [-]

CODE:

```
<html>
<head>
<style>
p {
    border: 3px solid red;
    padding: 10px;
    width: 400px;
    background: #FFF;
}
p.odd {
    /* Positioning information goes in here */
    position: static;
    top: 0px;
    left: 0px;
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
}
</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>
```

我们来看看在上面这个简单的 HTML 页面的情境里，改变第二个段落的定位方式将会产后怎样不同的布局：

静态定位: 这是元素定位的默认方式；它简单地遵从文档的自然流向。当一个元素静态定位时，top 和 left 属性将不起作用。静态定位的一个例子见图 7-2，其中用于定位的 css 为：position:static;top:0px;left:0px。

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

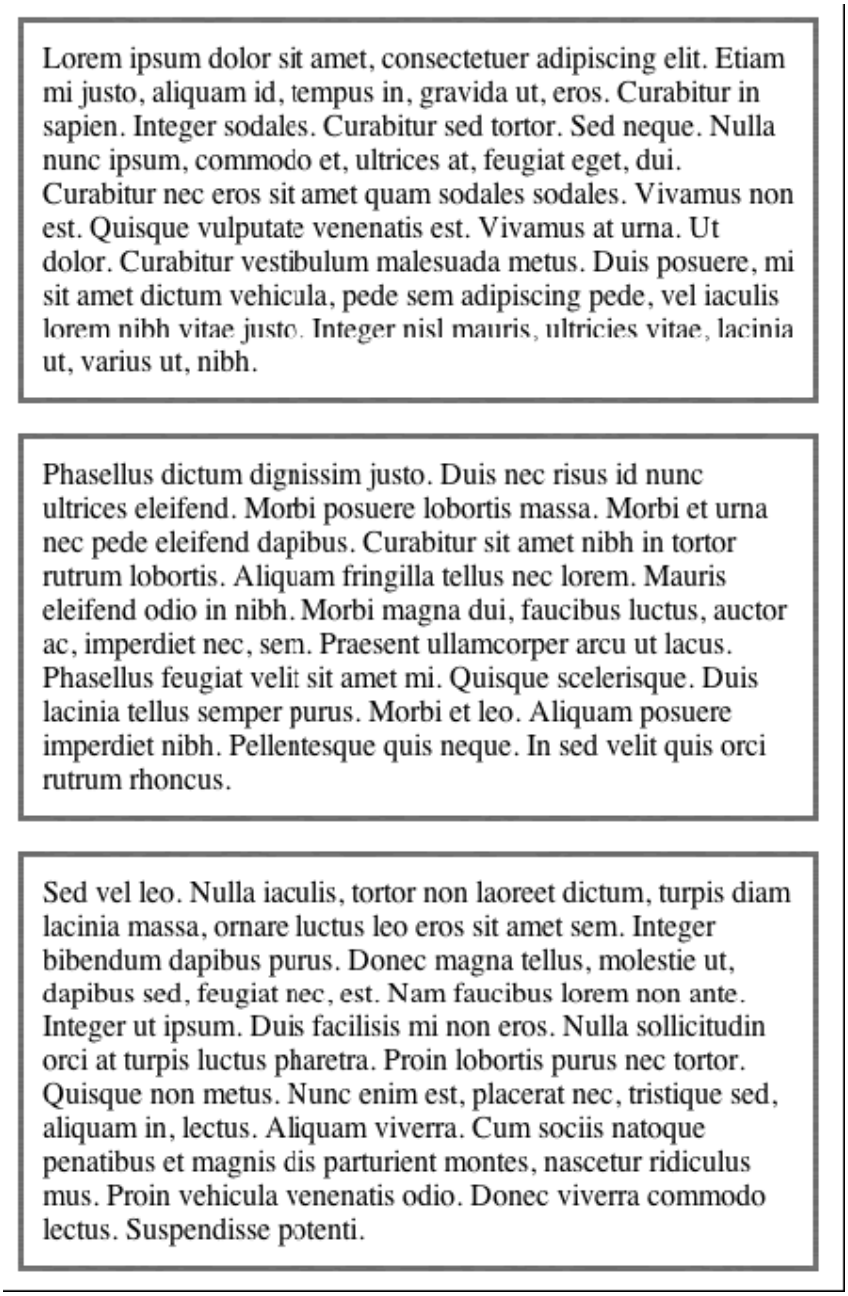


图 7-2. 页面正常(static)布局流里的段落

相对定位: 这一定位方式与静态定位非常相似，因为元素仍然会遵循正常的文档流直到得到其它指示。但是，设置 top 或 left 属性将会导致元素相对它的原来的(静态的)位置发生偏移。相对定位的一个例子如图 7-3 所示，其中的 CSS 定位为 position:relative;top:-50px;left:50px。

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

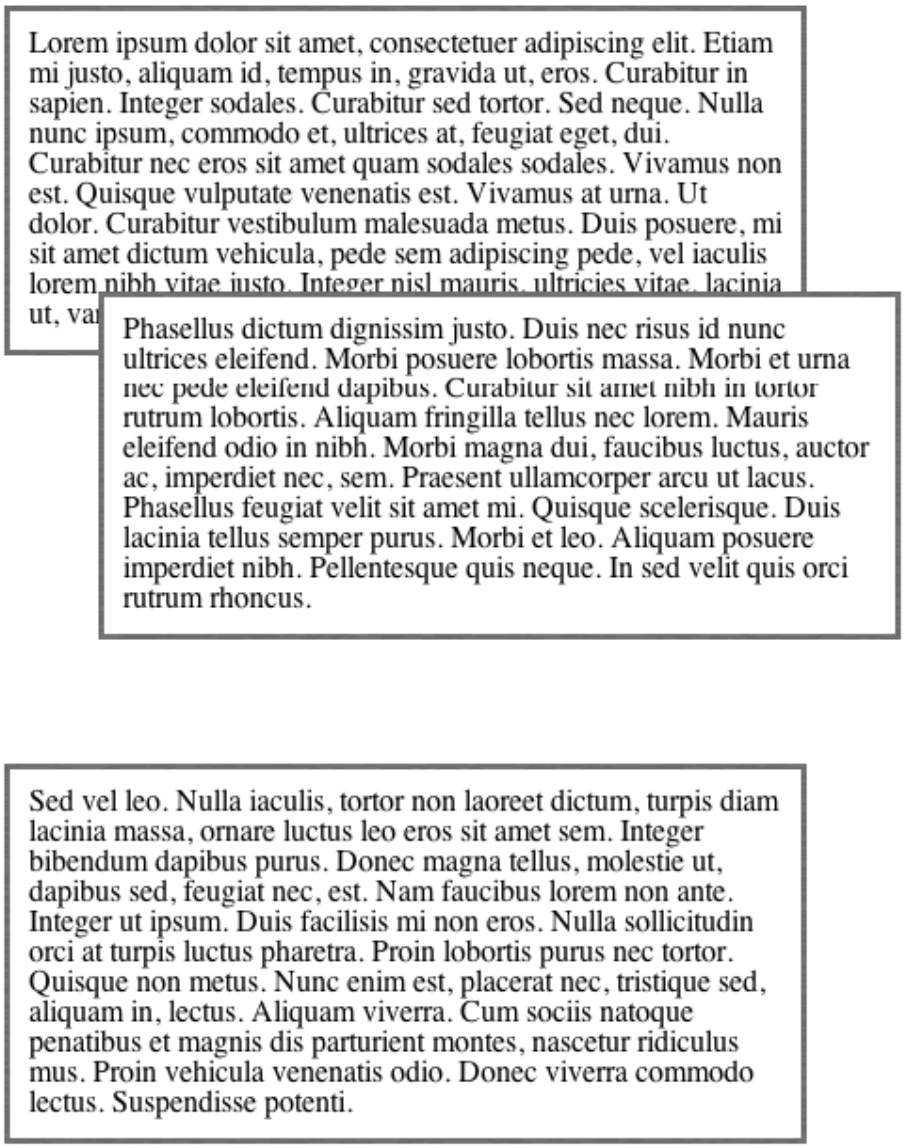


图 7-3. 相对定位，元素被移位到了前一个元素上面，而不再遵循正常的文档流

绝对定位: 将一个元素完全从正常的页面布局流中抓出来。被绝对定位的元素将相对于其第一个非静态定位的父级元素来显示。如果不存在父元素，它将相对于整个文档被定位。绝对定位的一个例子见图 7-4，其中用于定位的 css 为：position:absolute;top:20px;left:0px。

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

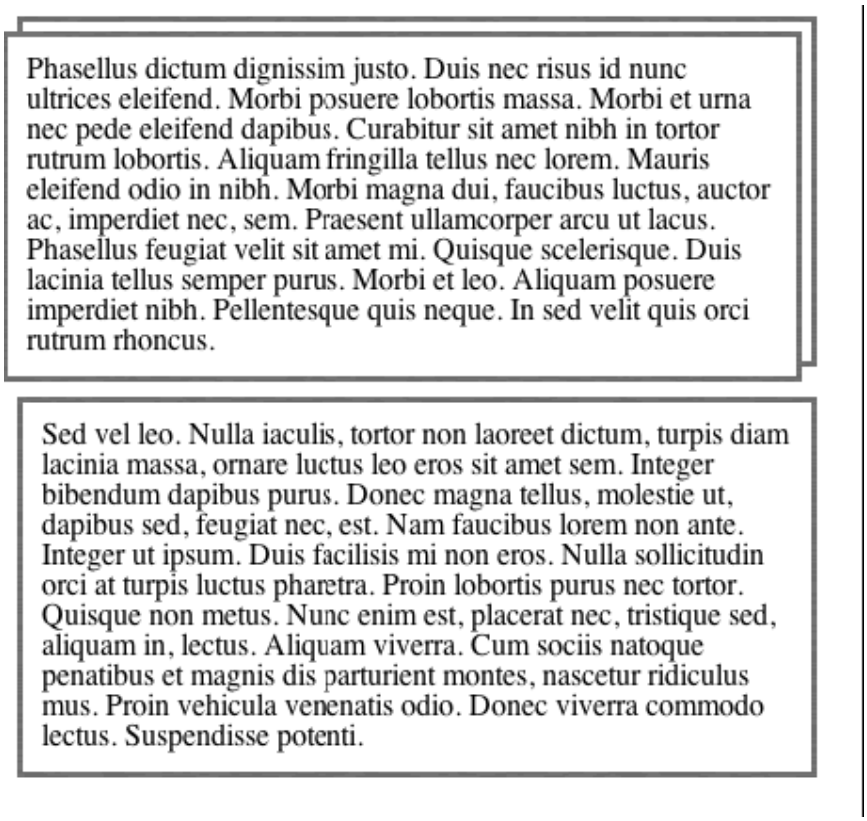


图 7-4. 绝对定位，元素的位置与页面的左上角相关，显示在已经存在的元素之上

固定定位: 固定定位将一个元素相对于浏览器的窗口定位。设置一个元素的 top 和 left 为 0 相素，将会使得该元素显示在浏览器的左上角(只要用户还在那个页面上)，完全忽略浏览器滚动条的任何动作。固定定位的一个例子见图 7-5,其中用于定位的 css 为：position:fixed;top:20px;right:0px。

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

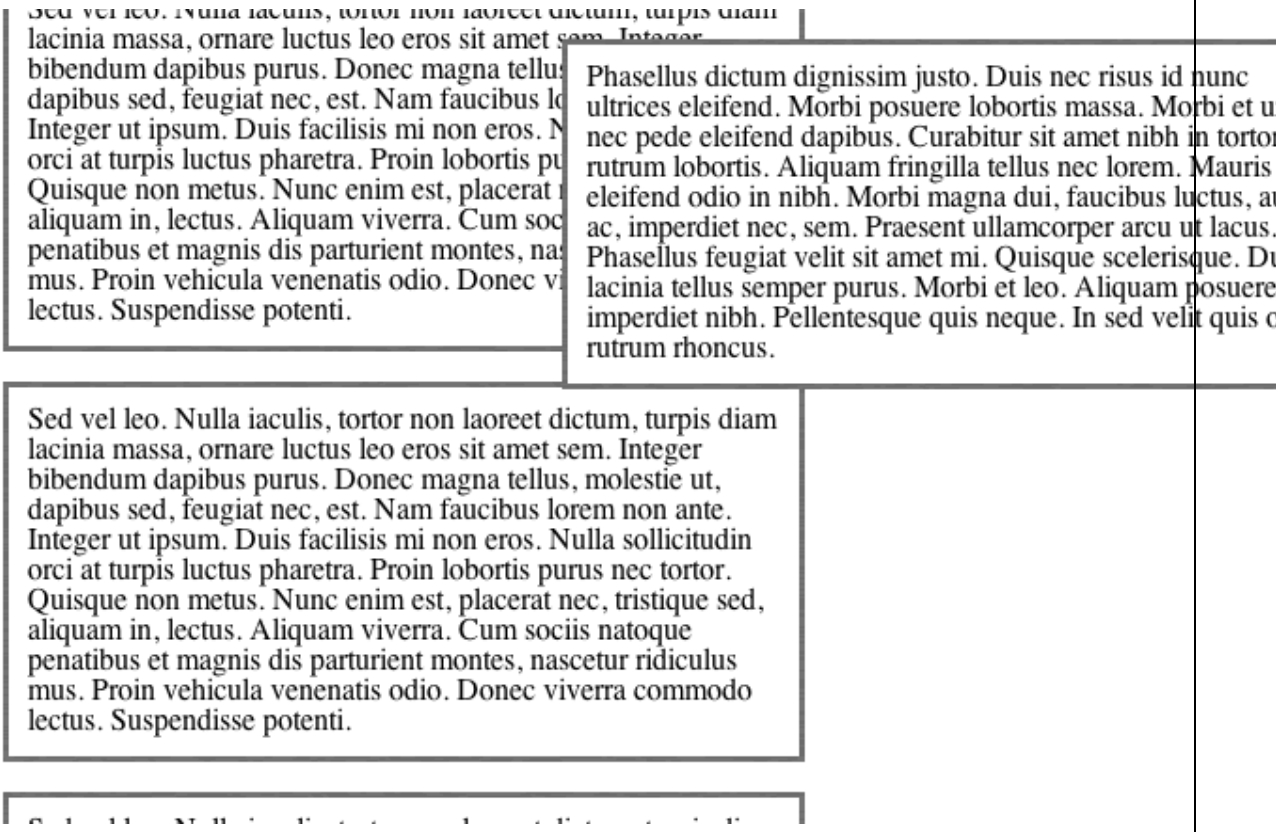


图 7-5. 固定定位，元素被定位到页面的右上角，尽管浏览器窗口被向下滚动了。

取得位置

元素被定位在何处依赖于它的 css 参数以及与其邻接的内容而不同。访问 CSS 属性或计算后的实际值都没有提供的一个能力是，获取元素在页面中或者仅在其它元素中的确切位置。

首先，我们来看如何获取元素在页面中的位置。你拥有几个可支配的元素属性可用来找到这一信息。所有的现代浏览器都支持以下三个属性；当然它们各自是怎么处理的，又是另外一回事了：

offsetParent: 理论上，这是元素在其中定位的父级元素。但是在实际情况下，offsetParent 引用的元素取决于浏览器(比如说，在 FireFox 中，它引用根节点，而在 Opera 中，则是直接父元素。

offsetLeft 和 offsetTop: 这些参数是元素在其 offsetParent 上下文中的水平和垂直偏移。在现代浏览器上，它总是精确的。

现在，问题在于寻求一种可以跨浏览器工作的用来判定方式元素位置的一致的办法。实现这一点的最一致的办法如程序 7-4 所示：使用 offsetParent 属性沿着 DOM 树上行，一路累加偏移值。

程序 7-4. 计算元素相对于文档的 x 和 y 坐标的辅助函数

[Copy to clipboard] [-]

CODE:

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
//计算元素的X(水平，左)位置
function pageX(elem) {
    //检查我们是否已经到了根元素
    return elem.offsetParent ?
        //如果我们还能往上，则将当前偏移与向上递归的值相加
        elem.offsetLeft + pageX( elem.offsetParent ) :
        //否则，取当前偏移
        elem.offsetLeft;
}
//计算元素的Y(垂直，顶)位置
function pageY(elem) {
    //检查我们是否已经到了根元素
    return elem.offsetParent ?
        //如果我们还能往上，则将当前偏移与向上递归的值相加
        elem.offsetTop + pageY( elem.offsetParent ) :
        //否则，取当前偏移
        elem.offsetTop;
}
```

定位问题的另一部分是计算元素在其父元素中的偏移。需要注意的重要的一点是，简单地使用元素的 style.left 或 style.top 属性是不够的，因为你可能想要找出没有用 Javascript 或 CSS 定义样式的元素的位置。

使用元素相对于其父元素的位置，你可以向 DOM 添加额外的相对该父元素定位的元素。比如，这个值用来建造上下文相关的工具提示是非常理想的。

为了找到元素相对于其父元素的位置，你必须再一次求助于 offsetParent 属性。因为该属性并不能保证返回特定元素的实际的父元素，你不得不使用你的 pageX 和 pageY 函数来找到父元素与子元素之间的位置差异。在程序 7-5 所示的两个函数中，我试图首先使用 offsetParent，如果它是当前元素的实际的父元素；否则，我将继续使用 pageX 和 pageY 方法沿 DOM 上行，以确定它的实际位置。

程序 7-5. 用来确定元素相对于其父元素位置两个函数

[Copy to clipboard] [-]

CODE:

```
//查找元素在其父元素中的垂直位置
function parentX(elem) {
    //如果 offsetParent 就是元素的 parent，则提前返回
    return elem.parentNode == elem.offsetParent ?
        elem.offsetLeft :
        //否则，我们需要找出两个元素相对整个页面的位置，计算差值
        pageX( elem ) - pageX( elem.parentNode );
}
//查找元素在其父元素中的垂直位置
function parentY(elem) {
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
    //如果 offsetParent 就是元素的 parent，则提早返回
    return elem.parentNode == elem.offsetParent ?
        elem.offsetTop :
        //否则，我们需要找出两个元素相对整个页面的位置，计算差值
        pageY( elem ) - pageY( elem.parentNode );
}
```

定位问题的最后一方面是找出元素相对于其 CSS 容器的位置。如前面所讨论的，元素可能实际被包含在一个元素中而相对于另一个元素被定位(通过使用相对和绝对定位)。记住这一点，你可以回头利用 getStyle 函数来得出计算后的 CSS 偏移值，因为那正是等效的定位。

有两个可用的简单的包装函数可以处理这一点，如程序 7-6 所示。它们都只是简单调用 getStyle 函数，但同时也删除任何多余的(除非你不是使用基于像素的布局，它才是有用的)单位信息(比如说，100px 将变成 100)。

程序 7-6. 找出元素的 CSS 定位的辅助函数

[Copy to clipboard] [-]

CODE:

```
//得到元素的 left 位置
function posX(elem) {
    //取得计算后样式并从中提取数字
    return parseInt( getStyle( elem, "left" ) );
}
//得到元素的 top 位置
function posY(elem) {
    //取得计算后的样式并从中提取数字
    return parseInt( getStyle( elem, "top" ) );
}
```

设置元素位置

不同于取得元素的的位置，设置位置要少许多变数。但是联合使用各种方式的布局(absolut,relative,fixed)时，你将能得到相当的、可用的结果。

目前，调整元素位置的唯一的办法是通过修改它的 CSS 属性。为了保持方法上的一致性，你应该仅修改 left 和 top 属性，尽管存在着其它的属性(如 bottom 和 top)。作为开端，你可以轻松地创建一对函数，如程序 7-7 所示，用来设置一个元素的位置，而不考虑其当前位置。

程序 7-7. 不考虑其当前位置，设置元素的 x 和 y 位置的一对函数

[Copy to clipboard] [-]

CODE:

```
//设置元素垂直位置的一个函数
function setX(elem, pos) {
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
    //使用像素为单位，设置 CSS 属性'left'
    elem.style.left = pos + "px";
}
//设置元素水平位置的一个函数
function setY(elem, pos) {
    //使用像素为单位，设置 CSS 属性'top'
    elem.style.top = pos + "px";
}
```

最终，你需要开发第二套函数，如程序 7-8 所示，你可以用它们来设置一个元素相对于其原来的位置的位置——比如，调整一个元素使其水平位置比当前值小 5 个像素。这一方法的使用直接与许多作为 DHTML 开发的支柱的动画效果相关。

程序 7-8. 用来调整元素相对于基原来的位置的一对函数

[Copy to clipboard] [-]

CODE:

```
//用来把元素的水平位置增加几个像素的一个函数
function addX(elem,pos) {
    //取得当前的水平位置并向其加入偏移
    setX( posX(elem) + pos );
}
//用来把元素的垂直位置增加几个像素的一个函数
function addY(elem,pos) {
    //取得当前的垂直位置并向其加入偏移
    setY( posY(elem) + pos );
}
```

现在我已经将处理元素位置的问题完全贯串了一遍。理解元素定位怎样工作和怎么设置及获取元素的精确位置是处理动态元素的一个基本的方面。你将看到的下一个侧面是元素的确切尺寸。

元素的尺寸

计算元素的高度和宽度可能是既无比简单又痛苦的一件事，这取决于具体的情况和你需要它做什么。许多情况下，你只需要使用 getStyle 函数的一个修改版本来得到元素的当前宽度和高度，如程序 7-9 所示。

程序 7-9. 检索 DOM 元素的当前高度和宽度的两个函数

[Copy to clipboard] [-]

CODE:

```
//获取元素的实际高度(使用计算后的 CSS)
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
function getHeight( elem ) {
    //取得计算后的 CSS 值并解析出一个可用的数字
    return parseInt( getStyle( elem, 'height' ) );
}
//获取元素的实际宽度(使用计算后的 CSS)
function getWidth( elem ) {
    //取得计算后的 CSS 值并解析出一个可用的数字
    return parseInt( getStyle( elem, 'width' ) );
}
```

当你试图做这两件事的时候麻烦就来了：第一，当你想要得到元素预定义的完整高度(比如说，你将一个动画从 0px 开始，但你需要知道该元素应该到达多高多宽)，第二，当一个元素的 display 设为"none"时，你将取不到值。

程序 7-10 里给出的两个函数说明了怎样找出元素潜在的完整高度和宽度，不论它当前的高度是多少。这是通过访问 clientWidth 和 clientHeight 属性实现的，它们提供元素能够展开到的可能的总的区域。

程序 7-10. 用来找出元素的潜在高度和宽度的两个函数，即使元素是隐藏的

[Copy to clipboard] [-]

CODE:

```
//找出元素完整的、可能的高度(不是实际的、当前的高度)
function fullHeight( elem ) {
    //如果元素当前是显示的，那么 offsetHeight 应该可以成功，即使失败，getHeight 也可以生效
    if ( getStyle( elem, 'display' ) != 'none' )
        return elem.offsetHeight || getHeight( elem );
    //否则，我们必须处理 display 为'none'的元素，
    //这时我们需要重置它的 CSS 属性以得到更精确的读数
    var old = resetCSS( elem, {
        display: '',
        visibility: 'hidden',
        position: 'absolute'
        });
    //计算元素的完整高度，如果 clientHeight 无效，则用 getHeight()
    var h = elem.clientHeight || getHeight( elem );
    //最后，我来来恢复元素本来的 CSS 属性
    restoreCSS( elem, old );
    //并返回元素的完整高度
    return h;
}
//找出元素完整的、可能的宽度(不是实际的、当前的宽度)
function fullWidth( elem ) {
    //如果元素当前是显示的，那么 offsetWidth 应该可以成功，即使失败，getWidth 也可以生效
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
    if ( getStyle( elem, 'display' ) != 'none' )
        return elem.offsetWidth || getWidth( elem );
    //否则，我们必须处理 display 为'none'的元素，
    //这时我们需要重置它的 CSS 属性以得到更精确的读数
    var old = resetCSS( elem, {
        display: '',
        visibility: 'hidden',
        position: 'absolute'
    });
    //计算元素的完整宽度，如果 clientWidth 无效，则用 getWidth()
    var w = elem.clientWidth || getWidth( elem );
    //最后，我来来恢复元素本来的 CSS 属性
    restoreCSS( elem, old );
    //并返回元素的完整宽度
    return w;
}
//用来设置一系列的 CSS 属性的一个函数，这些属性稍后可以恢复
function resetCSS( elem, prop ) {
    var old = {};
    //遍历每一个属性
    for ( var i in prop ) {
        //记录原来的属性
        old[ i ] = elem.style[ i ];
        //并设置新的值
        elem.style[ i ] = prop[i];
    }
    //返回改变的值的集合，以备 restoreCSS 使用
    return old;
}
//恢复 resetCSS 函数引用的副作用的函数
function restoreCSS( elem, prop ) {
    //将所有的属性重新设置为它们原来的值
    for ( var i in prop )
        elem.style[ i ] = prop[ i ];
}
```

同时拥有了取得元素当前的和潜在的宽度与高度的能力，你可以使用这些值来开发出一些你能够达到的动画。但是，在我进入动画的细节之前，你还需要看看怎样修改元素的可见性

元素的可见性

元素的可见性是可在 JavaScript 中用来创建从动画到快速模板效果的每一样东西的强大工

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

具。然而，更重要的是，它也能用来从视图中快速地隐藏元素，提供一些基本的用户界面功能。

在 CSS 里有两种不同的方式来有效地从视图中隐藏元素；它们各自有其益处但也能产生无意的后果，这取决于你怎样使用它们：

◇ visibility 属性决定一个元素是否可见，同时仍保持它在布局流中的正常占位。visibility 属性有两个值：visible(缺省值)和 hidden(使一个元素完全不可见)。比如说，如果你有一些<b>标签中换行的 visibility 属性设为 hidden 的文本，结果将简单地显示为文本中的一个空白块，尺寸与原始文本完全相同。例如，比较以下两行文本：

//正常文本:

Hello John, how are you today?

//对"John"应用了"visibility:hidden"的文本

Hello , how are you today?

◇ display 属性为开发者提供了更多的选项来控制元素的而局。这些选项是 inline(类似于<b>和的标签是行内的，也就是说他们遵循正常的文本流布局)，block(类似于<p>和 的标签是块级的，他们打破正常的文本流)，和 none(从文档中彻底隐藏元素)。将 display 属性设为 none 的结果表面上跟你把它从文档中删除完全一致；然而，实情并非那样，因为它可以在迟些时候快速地被切换回视图里。下面的两说明了 display 属性是怎样工作的：

//正常文本:

Hello John, how are you today?

//对"John"应用了"display:none"的文本

Hello , how are you today?

尽管 visibility 属性有它的特定的用途，display 属性的重要性不容忽视。元素的 visibility 属性被设为 hidden 时它仍存在于正常的文本流中，这使用得在许多应用中 visibility 的可行性打了折扣。程序 7-11 中展示了使用 display 属性来切换元素可见性的两种方法。

程序 7-11. 使用 CSS Display 属性来切换元素的可见性的一组函数

[Copy to clipboard] [-]

CODE:

```
//用来(使用 display)隐藏元素的一个函数
function hide( elem ) {
    //找到元素的当前显示状态是什么
    var curDisplay = getStyle( elem, 'display' );
    //记录它的显示状态
    if ( curDisplay != 'none' )
        elem.$oldDisplay = curDisplay;
    //将 display 设为 none(隐藏元素)
    elem.style.display = 'none';
}
//用来(使用 display)显示元素的函数
function show( elem ) {
    //将 display 属性设为它曾经的值，或者使用''，如果没有保存过先前的 display 的话
    elem.style.display = elem.$oldDisplay || '';
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
}
```

可见性的另一面是元素的透明度。调整元素透明度产生的结果与设置元素的 visibility 非常相似，但是了对元素有多可见的更好的控制。这意味着你可以使一个元素 50%可见，使得你可以看到在它下面的元素。所有的现代浏览器都在一定程度上支持透明度，IE 和 W3C 兼容的浏览器再一次在实现方式上有着不同。为了解决这一问题，你可以建创一个标准的函数用来维护元素的透明度，如 7-12 所示。level 为 0 表示着元素是完全透明的，而 level 为 100 表示完全不透明。

程序 7-12. 调整元素的透明度级别的一个函数

[Copy to clipboard] [-]

CODE:

```
//设置元素的透明度级别(level 是 0-100 的数字)
function setOpacity( elem, level ) {
    //如果滤镜存在，这是 IE，于是设置 Alpha 滤镜
    if ( elem.filters )
        elem.style.filter = 'alpha(opacity=' + level + ')';
        //译注：此处原文为 ele.style.filters='alpha(opacity=' + level + ')'，有误
        //多谢 02062007 同学的热心验证
    //否则，使用 W3C 的 opacity 属性
    else
        elem.style.opacity = level / 100;
}
```

掌握了调整元素的位置、大小、及可见性的方法，是时候开始探索你联合使用这些能力做一些有意思的事情了

动画

现在你已经掌握了执行基本的 DHTML 操作的基本技能，我们再来看 web 应用程序中流行的视觉效果之一：动画。如果使用得宜，动画将能够为用户提供有用的反馈，比如将注意力引向屏幕上的新创建的元素。

我们将先看两个流行的动画效果，再在研究广泛使用的 DHTML 库时重访它们。

滑入

在第一个动画里将处理一个(display 属性为"none"的)隐藏元素，你将通过在一秒之内逐渐增加其高度来渐渐地显示它，以取代粗糙的 show()函数。程序 7-13 所示的函数可以用作 show()函数的合适的替代，为用户提供更加平滑的视觉体验。

程序 7-13. 通过一在秒之内递增其高度来慢慢显示隐藏元素的函数

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

[Copy to clipboard] [-]

CODE:

```
function slideDown( elem ) {
    //从 0 开始扩张
    elem.style.height = '0px';
    //显示元素(但你看不到它，因为高度为零)
    show( elem );
    //得到元素的完整的潜在高度
    var h = fullHeight( elem );
    //我们将做一个在一秒钟内播放的 20"帧"的动画
    for ( var i = 0; i <= 100; i += 5 ) {

        //保证我们有一个正确的 i 的闭包
        (function(){
            var pos = i;
            //设置未来特定时间的定时器
            setTimeout(function(){
                //设置元素的新高度
                elem.style.height = ( pos / 100 ) * h  + "px";
                }, ( pos + 1 ) * 10 );
        })();
    }
}
```

淡入

你们将看到的下一个动画与第一个非常相似，不过用先前所创建的 setOpacity()函数代替了修改高度。这一函数(如程序 7-14)显示一个隐藏的元素，并将其透明度从 0(完全透明)逐渐变化到 100%(完全不透明)。与 7-13 的函数一样，它为用户提供更加平滑的视觉体验。

程序 7-13. 通过一在秒之内递增其透明度来慢慢显示隐藏的元素的函数

[Copy to clipboard] [-]

CODE:

```
function fadeIn( elem ) {
    //透明度从零开始
    setOpacity(elem,0);
    //显示元素(但是你看不到它，因为透明度为零)
    show( elem );
    //我们将做一个在一秒钟内播放的 20"帧"的动画
    for ( var i = 0; i <= 100; i += 5 ) {

        //保证我们有一个正确的 i 的闭包
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
        (function(){
            var pos = i;
            //设置未来特定时间的定时器
            setTimeout(function(){
                //设置元素的新的透明度
                setOpacity(elem,pos);
                }, ( pos + 1 ) * 10 );
        })();
    }
}
```

浏览器

除了操作特定的 DOM 元素以外，懂得怎样修改或追踪浏览器及其部件，将大大提升站点与用户间的交互。与浏览器协同工作最重要的两个方面是判定鼠标的位置和及用户将页面滚动了多少。

鼠标位置

获取鼠标位置是为用户提供拖放操作和上下文菜单的基本要求，这两种效果都只能通过 JavaScript 与 CSS 的交互来实现。

你首先需要检测的两个变量是光标相对于整个页面的 x 和 y 坐标(如程序 7-15 所示)。因为当前的鼠标坐标只可能从鼠标事件中取得，你最终需要使用一个普通的鼠标事件来捕获它们，如 MouseOver 或 MouseDown(这方面的更多例子见"拖放"小节)。

程序 7-15. 用来在获取鼠标在整个页面中位置的两个通用函数

[Copy to clipboard] [-]

CODE:

```
//取得鼠标的水平位置
function getX(e) {
    //获取事件对象
    e = e || window.event;
    //先取非 IE 的位置，不成功则用 IE 的位置
    return e.pageX || e.clientX + document.body.scrollLeft;
}
//取得鼠标的垂直位置
function getY(e) {
    //获取事件对象
    e = e || window.event;
    //先取非 IE 的位置，不成功则用 IE 的位置
    return e.pageY || e.clientY + document.body.scrollTop;
}
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

需要知道的第二组与鼠标相关的变量是鼠标光标相对于当前交互的元素的位置。可用于取得这些值的两个函数如程序 7-16 所示。

程序 7-16. 取得鼠标相对于当前元素位置的两个函数

[Copy to clipboard] [-]

CODE:

```
//取得鼠标相对于事件对象里 e 的目标元素(target)的 x 坐标
function getElementX( e ) {
    //取合适的元素偏移
    return ( e && e.layerX ) || window.event.offsetX;
}
//取得鼠标相对于事件对象里 e 的目标元素(target)的 x 坐标
function getElementY( e ) {
    //取合适的元素偏移
    return ( e && e.layerY ) || window.event.offsetY;
}
```

在学习本章的"拖放"这一节里浏览器中元素拖放的实现时，我们将重新回到鼠标交互问题。如需更多的鼠标事件的例子，可参见第六章和附录 B。

视口

浏览器的视口可认为就是浏览器里被滚动条围住的区域。视口包含几个部件：视口窗口，页面，滚动条。正确计算它们的位置和尺寸，是在有大段内容的情况下(比如自动滚屏的聊天室)开发漂亮的交互效果的需要。

页面尺寸

你需要关注的第一组属性是当前页面的宽度和高度。一般来说，大多数的实际的页面都被视口所裁切(通过检查视口尺寸和滚动条位置来判定)。程序 7-17 所示的两个函数使用了前面提到过的 scrollWidth 和 scrollHeight 属性(它们表征了一个部件可能的总的宽度和高度，而不仅仅是当前的可见的那一部分)。

程序 7-17. 判定当前页面的宽和高的两个函数

[Copy to clipboard] [-]

CODE:

```
//返回网页的高度(如果新的内容被加入，这一值可能会改变)
function pageHeight() {
    return document.body.scrollHeight;
}
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
//返回网页的宽度
function pageWidth() {
    return document.body.scrollWidth;
}
```

滚动条位置

接下来，你们将看到怎样断定浏览器滚动条的位置(或者，在另一个意义上，视口在当前页面上的定位情况)。掌握这些数字(使用 7-18 所示的函数了得)，对于超越浏览器提供的贫乏的滚动、建立自己更好的动态滚动，是必不可少的。

程序序 7-18. 用来判定视口在文档上定位于何处和两个函数

[Copy to clipboard] [-]

CODE:

```
//用来检测浏览器在水平方向滚动了多少的函数
function scrollX() {
    //ie6 strict 模式里的快捷方式
    var de = document.documentElement;
    //如果浏览器的 pageXOffset 可用，则使用之
    return self.pageXOffset ||

        //否则，尝试取得根节点的水平滚动量
        ( de && de.scrollLeft ) ||

        //最后，尝试取得 body 元素的水平滚动量
        document.body.scrollLeft;
}
//用来检测浏览器在垂直方向滚动了多少的函数
function scrollX() {
    //ie6 strict 模式里的快捷方式
    var de = document.documentElement;
    //如果浏览器的 pageYOffset 可用，则使用之
    return self.pageYOffset ||

        //否则，尝试取得根节点的垂直滚动量
        ( de && de.scrollTop ) ||

        //最后，尝试取得 body 元素的垂直滚动量
        document.body.scrollTop;
}
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

移动滚动条

拥有了页面中滚条的偏移量和页面本身的长度信息之后，就可以利用浏览器提供的 scrollTo 方法，调整页面上的视口的当前位置。

scrollTop 方法作为 window 对象(以及任何其它包含可滚动内容元素的元素或者 <iframe>)的一个属性存在，它接收两个参数，要将视口(或者元素或者<iframe>滚动到的 x 偏移和 y 偏移。程序 7-19 展示了两个使用 scrollTo 方法的例子。

程序 7-19. 使用 scrollTo 方法调整浏览器视口位置的例子

[Copy to clipboard] [-]

CODE:

```
//如果你想将视口滚动到浏览器的顶部，可以这么做：
window.scrollTo(0,0);
//如果你想将滚动到特定元素所在位置，可以这么做：
window.scrollTo( 0, pageY( document.getElementById("body") ) );
```

视口尺寸

有关视口的最后一个可能也是最明显的方面：视口本身的尺寸。知道视口的尺寸将能够洞悉用户当前可以看到多少内容，不管其屏幕分辨率和浏览器窗口是多大。可以使用程序 7-20 所示的两个函数来得到那些值。

程序 7-20. 判定浏览器视口高度和宽度的两个函数

[Copy to clipboard] [-]

CODE:

```
//取得视口高度
function windowHeight() {
    //ie6 strict 模式里的快捷方式
    var de = document.documentElement;
    //如果浏览器的 innerHeight 可用，则使用它
    return self.innerHeight ||
        //否则，尝试获得根节点的高度
        ( de && de.clientHeight ) ||
        //最后，尝试获得 body 元素的高度
        document.body.clientHeight;
}
//取得视口高度
function windowWidth() {
    //ie6 strict 模式里的快捷方式
    var de = document.documentElement;
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
    //如果浏览器的 innerWidth 可用，则使用它
    return self.innerWidth ||
        //否则，尝试获得根节点的宽度
        ( de && de.clientWidth ) ||
        //最后，尝试获得 body 元素的宽度
        document.body.clientWidth;
}
```

操作视口的重要性不容忽视。随便查看一个现代 web 应用程序，如 Gmail 或 Campfire，都能找到操作视口以提供引人注目的结果的实例(Gmail 提供了上下文相关的覆盖图，而 Campfire 提供了自动滚动的聊天室)。在第十一章中我将讨论在高交互性的 web 应用程序中视口可用来提供更好的体验的不同的方式。

拖放

浏览器上可实现的最流行的交互之一，是将一个元素在页面里拖动。使用你所学的技能(判定元素位置的能力，怎样调整位置，各种定位方式的不同)，你现在已可以完全理解拖放系统是怎样工作的。

为了探索这一技术，我选择参照 Aaron Boodman 创建的 DOM-Drag 库(http://boring.youngpup.net/2001/domdrag)。他的库里提供了许多便捷的功能，包含以下几种：

拖动手柄: 你可以拥有一个真正被移动的父元素和另一个被拖动的子元素。这对于创建具有窗口外观的界面是很好的。

回调函数: 你可以监视指定事件，如用户何时开始拖动元素、正在拖动元素、停止拖动元素，并得到元素的当前位置信息。

最小/最大拖动区域: 你可以限定一个元素不能拖动到特定的区域之外(如屏幕之外)。这对于建造滚动条是很完美的。

自定义坐标系统: 你可以选择操作一套 x/y 坐标系统映射，如果对使用 css 坐标系统感觉不爽的话。

自定义 x 和 y 坐标系统转换: 你可以使你拖动的元素以非传统的方式移动(如上下摇动或波浪形运动)。

DOM-Drag 系统的使用是相当简单的。程序 7-12 展示了一些使用 DOM-Drag 的例子。

程序 7-21. 使用 DOM-Drag 模拟浏览器内的可拖动窗口

[Copy to clipboard] [-]

CODE:

```
<html>
<head>
    <title>DOM-Drag – Draggable Window Demo</title>
    <script src="domdrag.js" type="text/javascript"></script>
    <script type="text/javascript">
    window.onload = function(){
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
        //初始化 DOM-Drag 函数，使 id 为"window"的元素可拖动
        Drag.init( document.getElementById("window") );
    };
    </script>
    <style>
    #window {
        border: 1px solid #DDD;
        border-top: 15px solid #DDD;
        width: 250px;
        height: 250px;
        position:relative;//译者加的，原文漏了
    }
    </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>
```

7-22 是 DOM-Drag 的一个带有完整文档的版本。代码作为单个的全局对象存在，该对象的方法可在页面元素上调用，以初始化拖放过程。

程序 7-22. 带有完整文档的 DOM-Drag 库

[Copy to clipboard] [-]

CODE:

```
var Drag = {
    // 被拖动的当前元素
    obj: null,
    //拖动元素的初始化函数
    // o = 做为拖动手柄的元素
    // oRoot = 被拖动的元素，如未指明，拖动手柄本身将为被拖动元素
    // minX, maxX, minY, maxY = 允许的元素最小和最大坐标
    // bSwapHorzRef = 切换到水平坐标系统
    // bSwapVertRef = 切换到垂直坐标系统
    // fxMapper, fyMapper = 另行映射 x 和 y 坐标的函数
    init: function(o, oRoot, minX, maxX, minY,
    maxY, bSwapHorzRef, bSwapVertRef, fXMapper, fYMapper) {
        //监视拖动事件的开始
        o.onmousedown = Drag.start;

        //确定使用哪个坐标系统
        o.hmode = bSwapHorzRef ? false : true ;
        o.vmode = bSwapVertRef ? false : true ;
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
        //确定哪个元素做做为句柄
        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";
        //检查用户是否提供的最小/最大坐标限定
        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;
        //检查指定的任何坐标映射函数
        o.xMapper = fXMapper ? fXMapper : null;
        o.yMapper = fYMapper ? fYMapper : null;
        //为所有的用户定义的函数添加外壳(shells)
        o.root.onDragStart = new Function();
        o.root.onDragEnd = new Function();
        o.root.onDrag = new Function();
    },
    start: function(e) {
        //找到正被拖动的元素
        var o = Drag.obj = this;
        //规范化事件对象
        e = Drag.fixE(e);
        //取得当前 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 );
        //以 x 和 y 坐标调用用户的函数
        o.root.onDragStart(x, y);
        //记录起始鼠标位置
        o.lastMouseX = e.clientX;
        o.lastMouseY = e.clientY;
        //如果我们正使用 CSS 坐标系统
        if (o.hmode) {
            //设定可用的最小和最大坐标
            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，我们正使用传统的数学坐标系统
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
        } else {
            if (o.minX != null) o.maxMouseX = -o.minX + e.clientX + x;
            if (o.maxX != null) o.minMouseX = -o.maxX + e.clientX + x;
        }
        //监听拖动和拖动结束事件
        document.onmousemove = Drag.drag;
        document.onmouseup = Drag.end;
        return false;
    },
    //在拖动事件中监视所有的鼠标运行的函数
    drag: function(e) {
        //规范化事件对象
        e = Drag.fixE(e);
        //得到正被拖动的对象的引用
        var o = Drag.obj;
        //取得鼠标在窗口中的位置
        var ey = e.clientY;
        var ex = e.clientX;
        //得到当前坐标
        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;
        //如果设定了最小 x 坐标，确保不会超过它
        if (o.minX != null) ex = o.hmode ?
            Math.max(ex, o.minMouseX) : Math.min(ex, o.maxMouseX);
        //如果设定了最大 x 坐标，确保不会超过它
        if (o.maxX != null) ex = o.hmode ?
            Math.min(ex, o.maxMouseX) : Math.max(ex, o.minMouseX);
        //如果设定了最小 y 坐标，确保不会超过它
        if (o.minY != null) ey = o.vmode ?
            Math.max(ey, o.minMouseY) : Math.min(ey, o.maxMouseY);
        //如果设定了最小 y 坐标，确保不会超过它
        if (o.maxY != null) ey = o.vmode ?
            Math.min(ey, o.maxMouseY) : Math.max(ey, o.minMouseY);
        //计算转换后的新的 x 和 y 坐标
        nx = x + ((ex - o.lastMouseX) * (o.hmode ? 1 : -1));
        ny = y + ((ey - o.lastMouseY) * (o.vmode ? 1 : -1));
        //并再次使用 x 和 y 映射函数（如果提供了）转换它们
        if (o.xMapper) nx = o.xMapper(y)
        else if (o.yMapper) ny = o.yMapper(x)
        //为元素设置新的 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";
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
        //并记录鼠标的最后位置
        Drag.obj.lastMouseX = ex;
        Drag.obj.lastMouseY = ey;
        //使用当前的 x 和 y 坐标调用用户的 onDrag 函数
        Drag.obj.root.onDrag(nx, ny);
        return false;
    },
    //处理拖放结束的函数
    end: function() {
        //不在监视鼠标事件(因为拖放已经结束)
        document.onmousemove = null;
        document.onmouseup = null;
        //在拖放事件的最后，用元素的 x 和 y 坐标调用我们的特别的 onDragEnd 函数
        Drag.obj.root.onDragEnd(
            parseInt(Drag.obj.root.style[Drag.obj.hmode ? "left" : "right"]),
            parseInt(Drag.obj.root.style[Drag.obj.vmode ? "top" : "bottom"]));
        //不再监视拖动的对象
        Drag.obj = null;
    },
    //规范化事件对象的函数
    fixE: function(e) {
        //如果 e 不存在，则是 IE 浏览器，于是使用 IE 的 event 对象
        if (typeof e == 'undefined') e = window.event;
        //如果 layer 属性没有设置，则从等效的属性中取得
        if (typeof e.layerX == 'undefined') e.layerX = e.offsetX;
        if (typeof e.layerY == 'undefined') e.layerY = e.offsetY;
        return e;
    }
};
```

平心而论，DOM-Drag 算是数以百计的 JavaScript 拖放库中比较简单的一个。但是我个人特别喜欢它，因其整洁的面向对象语法和相对的简单性。下一节中我将论述 Scriptaculous 库，它拥有一个出色而强大的拖放的实现，我强烈推荐你们去研究一下。

库

正如 JavaScript 里多数枯燥的任务一样，如果你想要开发出一个效果或者交互，极有可能它已经被创造出来了。下面将快速地浏览三种提供了不同的 DHTML 交互的库，你将知道作为开发者，有些什么拿来就可以用。

moo.fx 和 jQuery

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

有两种擅长于简单效果的轻量级的库：moo.fx 和 jQuery。这两个库都提供了一套基本的效果，可以用来组合以创建有效而简单的动画。关于它们的更多信息可以在其各自的相关网站上找到。程序 7-23 展示了这两个库的一些基本的例子。

程序 7-23. 使用 moo.fx 和 jQuery 的动画的基本例子

[Copy to clipboard] [-]

CODE:

```
//一个简单的动画：元素的隐藏部分被展开，完成以后，再次收缩
//这一动画的 moo.fx 实现
new fx.Height( "side", {
    duration: 1000,
    onComplete: function() {
        new fx.Height( "side", { duration: 1000 } ).hide();
    }
}).show();
// jQuery 的实现
$("#side").slideDown( 1000, function(){
    $(this).slideUp( 1000 );
});
//另一个简单的动画：元素的高度、宽度和透明度全部同步收缩或降低，产生一个很酷的隐藏效果
//此动画的 moo.fx 实现
new fx.Combo( "body", {
    height: true,
    width: true,
    opacity: true
}).hide();
//此动画的 jQuery 实现
$("#body").hide( "fast" );
```

正如你从例子中可能已看到的，moo.fx 和 jQuery 使得执行一些小巧的动画变成非常容易。两个项目都在其网站上提供了大量的应用其代码的例子，这是了解简单的 JavaScript 动画怎样工作的很好的方式。

moo.fx 的主页：http://moofx.mad4milk.net/

mootoolkit 文档和示例：http://moofx.mad4milk.net/

jQuery 的主页：http://jquery.com/

jQuery 效果和文档和示例：http://jquery.com/docs/fx/

Scriptaculous

如果说有一个库在所有的 DHTML 库中可以王者称之的话，那非 Scriptaculous 莫属。基于流行的 Prototype 库，Scriptaculous 提供了海量的各式效果，从动画效果到动作(诸如拖放)应有尽有。在 Scriptaculous 的网站上可以找到大量的信息和示例：

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

主页: http://script.aculo.us/
文档: http://wiki.script.aculo.us/scriptaculous/
演示: http://wiki.script.aculo.us/scriptaculous/show/Demos/

在拖放的实现方式上，Scriptaculous 引入强大的能力，同时又保持了最大限度的简单性。我给出两个简单的例子。

拖放重排序

Scriptaculous 使得列表的重排度变得极其简单。考虑到它代码是多么地简单(以及它与 Ajax 功能挂勾有多简单，如其网站上演示的)，它绝对是大多数 web 开发者的推荐方案。7-24 所示是使用 Scriptaculous 创建的一个简单的可重排序列表。

程序 7-24. 怎样使用 Scriptaculous 中提供的技术创建一个可重排序的列表

[Copy to clipboard] [-]

CODE:

```
<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(){
        //将 id 为 list 的元素转换成一个可拖放重排序的列表
        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>
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

我希望上面这个例子可以使你相信这个库所包含的能力，但如果还没有的话，你可以看看下一个例子，它创建了一个滑块输入控件。

滑块输入

Scriptaculous 提供了一些控件，可以用来解决常见的界面开发问题。使用多数拖放库都很容易实现的一个控件是滑块输入(滑动条块，得到数字输入)，Scriptaculous 也不例外，如程序 7-25 所示：

程序 7-25. 使用来自 Scriptaculous 的滑块输入控件创建向表单中年龄输入框的一个替代

[Copy to clipboard] [-]

CODE:

```
<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(){
        //将 id 为 ageHandle 的元素变成可拖动的手柄，
        //且将 id 为 ageBar 的元素变成滑动条
        new Control.Slider( 'ageHandle', 'ageBar', {
            //当滑块移动或完成移动时，调用 updateAge 函数
            onSlide: updateAge
        });

        //处理滑动块上发生的的动作
        function updateAge(v) {
            //滑动条更新时，更新 age 元素的值，代表用户的当前年龄
            $('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>
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
        <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>
```

我强烈推荐你在决定编写下一段交互的代码时查看一些 DHTML 库，基于这样一个简单的事实：库的作者通常在开发那一段交互代码时投入的时间和气力甚至比你开发整个应用程序所付出的还要多。掌握那些库，将显著缩短你的开发周期。

本章小结

在 web 应用程序中使用动态的交互是为用户提供更高水平的速度和可用性的一种非常有效果的方式。另外，如果使用流行的库，你实现这一切的时候将能够轻易保持较少的开发时间。下一章中，你将会综合这一章所学到的所有的交互技术，创建一个完全可用的、交互式的应用程序。

在这一章中，你已经学习过了实现 JavaScript 和 CSS 协作的所有不同的技术。你将拥有创建醒目的动画和动态的用户交互的能力。

应该记住的是，向一个网页添加任何形式的动态交互都有可能疏远你潜在的受众。应该确保你的应用程序总是可用的，即便是 JavaScript 和 CSS 被禁用。创建可平滑地降级的应用程序应该是任何 JavaScript 开发的理想状况。

继续追看

很期待第三节呢

看看 jq 的动画类是什么样子的 我也写过一个动画类 比较比较^^

```
<script language="javascript">
function Animation(target,targetProperty,closure,Duration,onfinish)
{
    if(!onfinish)onfinish=function(){};
    this.Begin=function anonymous()
    {
        target[targetProperty]=closure(0);
        var starttime=new Date();
        setTimeout(StoryBoard(),1);
        function StoryBoard()
        {
            return function()
            {
                var now=new Date();
                var d=now.getTime()-starttime.getTime();
                target[targetProperty]=closure(d);
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
                    if(d<Duration)setTimeout(StoryBoard(),10);
                    else {
                        target[targetProperty]=closure(Duration);
                        onfinish();
                    }
                }
            }
        }
        this.setTo=function(val){to=val;}
        this.setFrom=function(val){from=val;}
    }
    </script>
    <div id="come" style="position:absolute;">come</div>
    <script>
    var slide=new Animation(come.style,"left",function(time){return
1000-time/5000*1000;},5000,function(){slide.Begin();})
    slide.Begin();
    var hide=new Animation(come.style,"filter",function(time){return
"alpha(opacity="+(time/5000*100)+")";},5000,function(){hide.Begin();})
    hide.Begin();
    </script>
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn