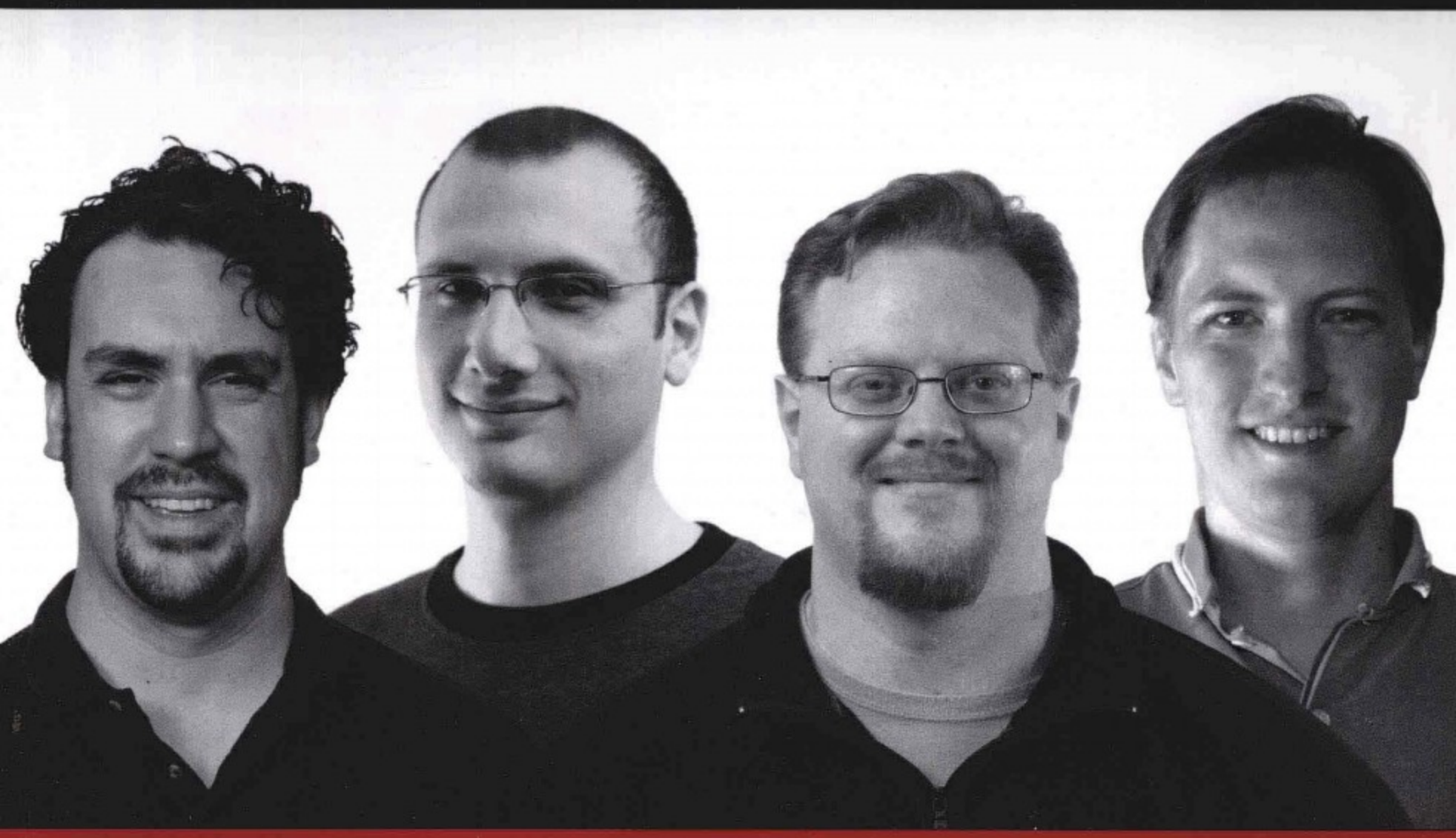
PROGRAMMER TO PROGRAMMER™



Professional JavaScript Frameworks: Prototype, YUI
Ext JS, Dojo and MooTools

JavaScript 框架高级编程

——应用Prototy Dojo、MooTools

(美) Leslie M. Orchard
Ara Pehlivanian 等著
杨明军 译

wrox™
WILEY
清华大学出版社

PROGRAMMER TO PROGRAMMER™

JavaScript框架高级编程

——应用Prototype、YUI、Ext JS、Dojo、MooTools

Professional JavaScript Frameworks: Prototype, YUI, Ext JS, Dojo and MooTools

近年来开发人员的需要和要求已经发生变化，JavaScript也是如此，它可以提供高性能的、令人印象深刻的Web用户体验。这个灵活的动态编程语言越来越多地用于正式的Web开发中，而且它的多种工具和项目正以代码库和框架的形式分享。《JavaScript框架高级编程——应用Prototype、YUI、Ext JS、Dojo、MooTools》涵盖了几个最流行的JavaScript框架，研究了这些框架如何采用独特的、各不相同的方式解决Web开发中的各种问题，每个框架都有各自的优缺点。

《JavaScript框架高级编程——应用Prototype、YUI、Ext JS、Dojo、MooTools》的作者团队汇集了目前最活跃、最流行的几个JavaScript框架，详细讲解了每个框架解决的常见Web开发问题，同时研究了每个框架如何解决特定的一组任务。此外，本书运用大量实用的示例和清晰的讲解来演示现代Web开发涉及的众多方面，以及JavaScript框架提供了什么选项来帮助我们快速构建并运行应用程序。

主要内容

- Prototype框架：处理跨浏览器事件、操作常见的数据函数、简化AJAX和动态数据处理以及其他方面
- Yahoo! User Interface(YUI)库：使用动画和拖放、利用窗口部件构建用户界面、使用YUI CSS工具以及其他方面
- Ext JS框架：与服务器交互、使用数据视图和网格、处理表单控件和数据验证以及其他方面
- Dojo框架：操作DOM、编排动画、部署和扩展Dojo以及其他方面
- MooTools框架：运用MooTools增强开发、构建用户界面、使用动画以及其他方面

读者对象

本书适合于渴望探究JavaScript框架带来的各种益处的Web开发人员。读者需要具备HTML、CSS和JavaScript的应用知识。

源代码下载及技术支持

http://www.wrox.com

http://www.tupwk.com.cn/downpage

Wrox Professional guides are planned and written by working programmers to meet the real-world needs of programmers, developers, and IT professionals. Focused and relevant, they address the issues technology professionals face every day. They provide examples, practical solutions, and expert education in new technologies, all designed to help programmers do a better job.

p2p.wrox.com
The programmer's resource center

www.wrox.com

wrox™
An Imprint of WILEY

图书上架分类建议	Web开发
	JavaScript、JavaScript框架

读者信箱：wkservice@vip.163.com
投稿信箱：bookservice@263.net

ISBN 978-7-302-24783-8

9 787302 247838 >

定价：98.00元

JavaScript 框架高级编程

——应用 Prototype、YUI、Ext JS、Dojo、MooTools

(美) Leslie M. Orchard
Ara Pehlivanian 等著

杨明军 译

清华大学出版社

北 京

Leslie M. Orchard, Ara Pehlivanian, et al.
Professional JavaScript Frameworks: Prototype, YUI, Ext JS, Dojo and MooTools
EISBN：978-0-470-38459-6

北京市版权局著作权合同登记号　图字：01-2009-7475

图书在版编目(CIP)数据

JavaScript 框架高级编程——应用 Prototype、YUI、Ext JS、Dojo、MooTools/(美)欧查德(Orchard, L.M.)，(美)佩里瓦尼安(Pehlivanian, A.) 著；杨明军 译. —北京：清华大学出版社，2011.2
书名原文：Professional JavaScript Frameworks: Prototype, YUI, Ext JS, Dojo and MooTools
ISBN 978-7-302-24783-8

Ⅰ. J…　Ⅱ. ①欧… ②佩… ③杨…　Ⅲ. Java 语言—程序设计　Ⅳ. TP312

中国版本图书馆 CIP 数据核字(2011)第 012845 号

责任编辑：王　军　张立浩
装帧设计：孔祥丰
责任校对：胡雁翎
责任印制：何　芊
出版发行：清华大学出版社　　**地　　址**：北京清华大学学研大厦 A 座
http://www.tup.com.cn　　**邮　　编**：100084
社　总　机：010-62770175　　**邮　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质　量　反　馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 刷 者：清华大学印刷厂
装 订 者：三河市新茂装订有限公司
经　　销：全国新华书店
开　　本：185×260　**印　张**：49　**字　数**：1317 千字
版　　次：2011 年 2 月第 1 版　　**印　　次**：2011 年 2 月第1 次印刷
印　　数：1～3000
定　　价：98.00 元

产品编号：031836-01

译 者 序

JavaScript 是一种由 Netscape 的 LiveScript 发展而来的、基于原型继承的、面向对象的、动态类型的客户端脚本语言，旨在为用户提供更流畅的 Web 体验。JavaScript 是一门具有非常丰富功能的语言，它有着与其他编程语言一样的复杂性，甚至更加复杂。实际上，要想采用 JavaScript 编写比较复杂的程序，必须对 JavaScript 语言有扎实的理解才行。JavaScript 发明人 Brendan Eich 誉为“JavaScript 的精神领袖”的 Douglas Crockford 曾经撰文宣称“JavaScript 是世界上最被误解的语言”(http://javascript.crockford.com/zh/javascript.html)。

但是，AJAX 给了 JavaScript 复兴的机会。随着 Web 2.0/AJAX 技术的成功推广，JavaScript 也得到了发展的绝佳机会。为了解决 JavaScript/DOM 跨浏览器问题、代码重用甚至在设计取向方面存在的问题，各种 JavaScript 框架相继涌现，力争解决这些问题。JavaScript 语言设计的灵活性为 JavaScript 框架的设计和实现带来了无限的可能性，而且这些框架身处 Web 开发热潮，因此快速进化中，这导致我们很难全面了解这些框架。本书精选了 5 种最流行的 JavaScript 框架，旨在让读者了解这些框架的设计思想并掌握它们的应用技巧。

本书通过大量的实用示例和详细的论述，演示了 5 种主流框架解决主要的 Web 开发问题的方式以及它们背后的设计哲学。相信通过这 5 种框架的学习和对比，读者能够对 JavaScript 编程有深刻的理解，并能够在自己的 Web 项目中利用这些框架来简化编程，提高工作效率。

本书由**杨明军**翻译。**Be Flying 工作室**负责人肖国尊负责本书译员的选定、翻译质量和进度的控制与管理。敬请广大读者提供反馈意见，读者可以将意见发到 wkservice@vip.163.com，我们会仔细查阅读者发来的每一封邮件，以求进一步提高今后译著的质量。

作 者 简 介

Leslie Michael Orchard 是一位来自底特律地区的作家和连续剧爱好者。他和心爱的妻子饲养了两只奥斯豹斑猫和一对袖珍兔子。忙里偷闲，他会在名为 0xDECAFBAD 的网站上(http://decafbad.com)分享一些代码、文档以及其他有趣的内容。

Ara Pehlivanian 自从 1997 年就一直从事 Web 相关工作。他曾经做过自由职业者、网站管理员，最近则成为 Nurun(一家全球交互式通信机构)的前端架构师和实践引领者。Ara 的经历来自于他曾经在自己的职业生涯中做过 Web 开发方方面面的工作，而现在则热衷于基于 Web 标准的前端开发。除了在工作上教授有关最佳实践的知识以及编写代码之外，他还维护自己的个人网站 http://arapehlivanian.com/。

Scott Koon 已经从事专业软件开发超过 13 年时间。他将大学期间的主要时间花在了研究一种遗留的、没有文档的符号语言(DNA)上，并最终取得了生物化学学位。他在 JavaScript 还被称为 LiveScript 时就已经在编写 JavaScript 代码，当时 Netscape 浏览器的右上角还有一个非常大的紫色字母“N”。他在 http://lazycoder.com 上面维护了一个博客，而且已经活跃在博客社区中将近 10 年时间。他经常光顾 Twitter(http://www.twitter.com/lazycoder)，并且为一个名为 Witty 的 WPF .NET Twitter 客户端做出贡献。他还与其他人共同创办了一个名为 Herding Code 的播客(http://www.herdingcode.com)，用来处理各种技术方面的问题。他和妻子以及两个女儿一起住在华盛顿州西雅图市。

Harley Jones 目前是 Perficient, Inc(NASDAQ: PRFT)的首席技术咨询师。他毕业于乔治亚州亚特兰大市的奥格尔绍普大学，获得英语文学学位。但是，他自从 10 岁起就一直在进行编程。他已经从事专业软件开发超过 10 年时间，酷爱研究编程语言。他积极地支持并教授几乎每个认真学习现代编程技术的人员。在不编写程序时，他会教育自己的孩子成为科学家。可以通过 harley.333@gmail.com 联系 Harley。

前　言

JavaScript 是一种用于 Web 应用程序的行业标准客户端脚本编程语言。本书研究了一些可用的顶级 JavaScript(JS)框架，提供了大量实用的示例和讲解来说明每个框架最擅长的领域。

在过去几年中，JavaScript 语言出现了复兴趋势。各种各样的项目如雨后春笋般出现，它们致力于构建可重用的 JS 库和框架，而在这方面，它们中有许多已经成熟并显示出持久力，值得在专业项目中认真考虑并依靠它们。

JavaScript 在 Web 的发展中得以流行，现在所有主要浏览器和新的 Web 技术都支持它。随着时间的推移，JavaScript 已经得到扩展，通过使用 Adobe Flash、AJAX 和 Microsoft Silverlight 等技术，可以交付高性能的、给人留下深刻印象的 Web 用户体验。

随着 JavaScript 越来越多地用于“正式的”Web 开发，开发人员将在这个过程中汲取的教训和发明的工具以库和框架的形式整理出来并进行分享。但由于 JavaScript 是一种如此灵活和动态的语言，每种框架都采用非常不同的方式来解决 Web 开发中的各种问题，每种方式都有各自的优缺点。

本书目标读者

本书面向热衷于学习 JavaScript 并研究各种 JavaScript 框架提供了哪些工具以及它们如何提高工作效率的 Web 开发人员。本书要求读者了解基本的 HTML、CSS 和 JavaScript 应用知识，而且具备一定的面向对象编程、服务器端 PHP 脚本编程经验，并了解诸如 AJAX 之类的现代 Web 开发技术。

本书涵盖内容

本书旨在成为一本在工作和查询其他联机资料来源时的简洁的、方便的手册，因此没有详细讲解有关 JavaScript 框架的更高级的、实验性的、尚在开发的方面。

使用本书的前提条件

一款浏览器、一个文本编辑器以及 Web 托管环境基本上就是使用本书示例所需的全部条件。强烈推荐使用安装了 Firebug 扩展的 Mozilla Firefox 作为浏览器，因为这个组合提供了非常强大的带有 JavaScript 日志和 DOM 探索工具的浏览器内开发环境。

可以在 http://getfirefox.com/中下载 Mozilla Firefox，一旦运行 Firefox，就可以在 http://getfirebug.com/

中获取 Firebug 扩展。

此外，本书中的一些示例还要求服务器端脚本能够完全运行，而且示例代码采用 PHP 编写。因此，能够访问 Web 服务器上的 PHP 将有所帮助，这些脚本在 PHP 4 及以上的任何版本中都应该能够运行。

源代码

在读者学习本书中的示例时，可以手工输入所有的代码，也可以使用本书附带的源代码文件。本书使用的所有源代码都可以从本书合作站点 http://www.wrox.com/或 www.tupwk.com.cn/downpage 上下载。登录到站点 http://www.wrox.com/，使用 Search 工具或使用书名列表就可以找到本书。接着单击本书细目页面上的 Download Code 链接，就可以获得所有源代码。

注释：

由于许多图书的标题都很类似，所以按 ISBN 搜索是最简单的，本书英文版的 ISBN 是 978-0-470-38459-6。

下载代码后，只需用自己喜欢的解压缩软件对它进行解压缩即可。另外，也可以进入 http://www.wrox.com/dynamic/books/download.aspx 上的 Wrox 代码下载主页，查看本书和其他 Wrox 图书的所有代码。

勘误表

尽管我们已经尽了各种努力来保证文章或代码中不出现错误，但是错误总是难免的，如果你在本书中找到了错误，例如拼写错误或代码错误，请告诉我们，我们将非常感激。通过勘误表，可以让其他读者更方便地学习，当然，这还有助于提供更高质量的信息。

请给 wkservice@vip.163.com 发电子邮件我们就会检查你的反馈信息，如果是正确的，我们将在本书的后续版本中采用。

要在网站上找到本书英文版的勘误表，可以登录 http://www.wrox.com，通过 Search 工具或书名列表查找本书，然后在本书的细目页面上，单击 Book Errata 链接。在这个页面上可以查看到 Wrox 编辑已提交和粘贴的所有勘误项。完整的图书列表还包括每本书的勘误表，网址是 www.wrox.com/misc-pages/booklist.shtml。

P2P.WROX.COM

要与作者和同行讨论，请加入 p2p.wrox.com 上的 P2P 论坛。这个论坛是一个基于 Web 的系统，便于你张贴与 Wrox 图书相关的消息和相关技术，与其他读者和技术用户交流心得。该论坛提供了订阅功能，当论坛上有新的消息时，它可以给你传达感兴趣的论题。Wrox 作者、编辑和其他业界专家和读者都会到这个论坛上来探讨问题。

在 http://p2p.wrox.com 上，有许多不同的论坛，它们不仅有助于阅读本书，还有助于开发自

己的应用程序。要加入论坛，可以遵循下面的步骤：

(1) 进入 p2p.wrox.com，单击 Register 链接。

(2) 阅读使用协议，并单击 Agree 按钮。

(3) 填写加入该论坛所需要的信息和自己希望提供的其他信息，单击 Submit 按钮。

(4) 你会收到一封电子邮件，其中的信息描述了如何验证帐户，完成加入过程。

提示：

不加入 P2P 也可以阅读论坛上的消息，但要张贴自己的消息，就必须加入该论坛。

加入论坛后，就可以张贴新消息，响应其他用户张贴的消息。可以随时在 Web 上阅读消息。如果要让该网站给自己发送特定论坛中的消息，可以单击论坛列表中该论坛名旁边的 Subscribe to this Forum 图标。

关于使用 Wrox P2P 的更多信息，可阅读 P2P FAQ，了解论坛软件的工作情况以及 P2P 和 Wrox 图书的许多常见问题。要阅读 FAQ，可以在任意 P2P 页面上单击 FAQ 链接。

目　录

第III部分 Ext JS

第Ⅳ部分 Dojo

第 I 部分

Prototype

Prototype 是 Web 2.0 复苏期间表现突出的第一批 JavaScript 库之一。当 2005 年第一次提出 AJAX 术语时，如何构造跨浏览器 XMLHttpRequests 还是浏览器相关代码的一个雷区。Prototype 为把事件绑定到相应的处理程序提供了通用的方法，并为创建能够在所有浏览器中运行的 AJAX 请求提供了通用的接口，以此来使事件处理变得顺畅，从而帮助您获得跨浏览器兼容性。它还能够处理所有浏览器中的特殊情况，让您能够将注意力集中在编写代码上，而会使用浏览器特有的“if-else”语句弄乱代码，从而能够以跨浏览器方式操作 DOM。

Prototype 不仅扩展了 JavaScript 语言，而且扩展了 DOM 元素。Prototype 扩展 JavaScript 原生 Object 以包含用来确定对象所表示数据类型的方法以及一些有用的序列化方法。Enumerable 类提供了诸如 each()和 map()这样的直接作用于数组的有用方法，可用来轻易地遍历和操作 JavaScript 对象数组和 DOM 元素数组。原生 Function 对象也被扩展，从而具有了一些有用的函数，包括 wrap()，该函数可用来为您自己的方法编写拦截器，以提供类似日志记录之类的有用功能。

Prototype 利用 Class 对象简化了继承机制。可以轻易地扩展对象并创建继承层次结构，而不需要担心静态类型语言中与普通继承机制相关的难题。所有这些功能使 Prototype 成为采用 JavaScript 编写逻辑的最佳选择，而且它为编写自定义 JavaScript 库提供了极佳的基础。既然 Prototype 已经自动完成所有繁重的任务，您就可以把精力集中在库开发中比较有趣的部分，即创建新的窗口部件和数据结构。

第 1 章

扩展和增强 DOM 元素

Prototype 是一款杰出的框架，既可以将其作为主 JavaScript 库，也可以将其作为另一个库的基础。Prototype 的部分魔力在于该框架扩展了 DOM 元素。通过向元素添加新的方法，Prototype 使得以更加生动的方式编写跨浏览器代码变得较为容易。还有一些方法可用来处理诸如元素定位这类比较麻烦的细节。通过利用诸如 getElementsByClassName 和 getElementsBySelectors 之类的辅助方法，我们可以很容易将样式或事件应用到一组具有某种共性的元素中，从而可以更容易编写简洁的 JavaScript 代码。

本章内容简介：

- 利用 Prototype 扩展 DOM 元素
- 修改以及操作内容和大小
- 使用 CSS 设置元素的样式

1.1 扩展 DOM 元素

在 Prototype 出现之前，跨浏览器代码经常看上去像是路线图：大量的分支以及反反复复进行的大量的相同检查。Prototype 扩展了代码操作的 DOM 元素，从而将日常 JavaScript 编程中为了跨浏览器而编写的繁杂代码集中起来。Prototype 为 Element.Methods 和 Element.Methods.Simulated 对象中的所有元素均提供了扩展方法。如果该对象是 input、select 或 textarea 标记，那么还包括 Form.Element.Methods 中的方法。表单元素自身也将扩展，包含 Form.Methods 对象中的方法。这些方法中的大部分返回原始元素，因此可以将多个方法链接起来，其格式类似于$(myElement).update("updated").show();。有一点需要注意，不仅是选择的元素会扩展，而且该元素的所有子元素也都会扩展。

在支持修改 HTMLElement.prototype 的浏览器中，Prototype 向 HTMLElement 中添加了一些方法。这意味着不需要在手动创建的每个元素上调用 Element.extends()方法。可以立即开始使用 Prototype 方法。

```
var newDiv = document.createElement("div");
newDiv.update("Insert some text");
newDiv.addClassName("highlight");
```

Internet Explorer 浏览器不支持修改 HTMLElement，因此必须调用 Element.extends()方法，或者使用$()或$$()方法获取该元素的引用。

1.1.1　美元符号函数：$()

扩展 DOM 元素最简单的方法是使用$()函数获取元素的引用，而不是使用 document.getElementById 或其他方法。当通过这种方式获得引用时，Prototype 会自动将 Element.Methods 中的所有方法添加到该元素中。如果在这个方法中传入一个字符串，那么该方法将自动获取带有指定 ID 的元素的引用。如果传入的是指向这个元素的引用，那么它将返回同一个引用，但是该元素将具有扩展方法。这是扩展元素的最常见方式。

```
<body>
<div id="myId"> Hello Prototype </div>
<script type="text/javascript">
        $("myId").hide();
</script>
</body>
```

1.1.2　$$()

这个方法的工作方式与$()函数类似。它带有一个表示 CSS 选择器的实参，返回由所有匹配该选择器的元素构成的数组。CSS 选择器是从 DOM 中取回特定元素的强大工具。该数组中的元素的顺序与它们在 DOM 中出现的顺序相同，而且每个元素都由 Prototype 扩展。

```
$$('input');
// select all of the input elements

$$('#myId');
//select the element with the id "myId"

$$('input.validate');
//select all of the input elements with the class "validate"
```

Prototype 并不使用浏览器的内置 CSS 选择器分析功能，因此它可以自由地实现较新版本的 CSS 中指定的选择器。其结果就是，Prototype 1.5.1 及更高版本支持 CSS3 的几乎所有功能。

```
$$('#myId> input');
//select all of the input elements that are children of the element with the id "myId"

$$('table <tr:nth-child(even)');
//selects all of the even numbered rows of all table elements.
```

1.1.3　Element.extend()

这个方法接受一个元素实参，并使用 Element.Methods 中的方法扩展该元素。这个方法与$()方法非常类似，但是它只接受 DOM 对象的引用，而如果传给它的是一个 id，那么它不会取回引用。

下面是一个使用 Element.extend()方法的简单示例。

```
Var newDiv = document.createElement("div");
Element.extend(newDiv);
newDiv.hide();
```

1.1.4　将 Element 对象用作构造函数

还可以将 Element 对象用作构造新的 DOM 元素的一种方式，而不是仅仅使用内置 DOM 方法。采用这种方式创建的新元素将自动由 Prototype 扩展，可以立即使用。

```
<head>
       <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
       <title> untitled </title>
       <style>
            .redText { color: red;}
       </style>
</head>
<body>
     <div id="myDiv" class="main"> Here is my div </div>

     <textarea id="results" cols="50" rows="10"> </textarea>
     <script type="text/javascript" src="prototype-1.6.0.2.js"> </script>
<script type="text/javascript">
        Event.observe(window,"load", function(e) {
           $("results").value = "";
           $("results").value += $("myDiv").id + "\n";
           $("results").value += $$(".main")[0].id + "\n";
           var newEl = new Element("h1",{"class": "redText"});
           $("myDiv").insert(newEl);
           newEl.update("I'm new here");

           var manuallyCreated = document.createElement("h2");
           $("myDiv").insert(manuallyCreated);
           Element.extend(manuallyCreated);
           manuallyCreated.update("I was extended");
     });
</script> </body>
```

注意：

因为$$()方法返回的是按照 DOM 中的出现顺序排列的元素数组，所以必须使用序数 0 引用该数组的第一个元素。

这里给出了扩展元素的几种方式。首先，可以使用$()方法通过 ID 来获取该元素并扩展它。接着，使用$$()方法并传入一个 CSS 选择器，通过类名来获取该元素。现在将 Element 对象用作构造函数来创建一个新的 H1 元素，将其类设为 redText，然后将其插入到 myDiv 元素中，并设置新创建的元素的文本。最后，采用传统的方式创建一个对象，并使用 Element.extend()方法扩展该元素，如图 1-1 所示。

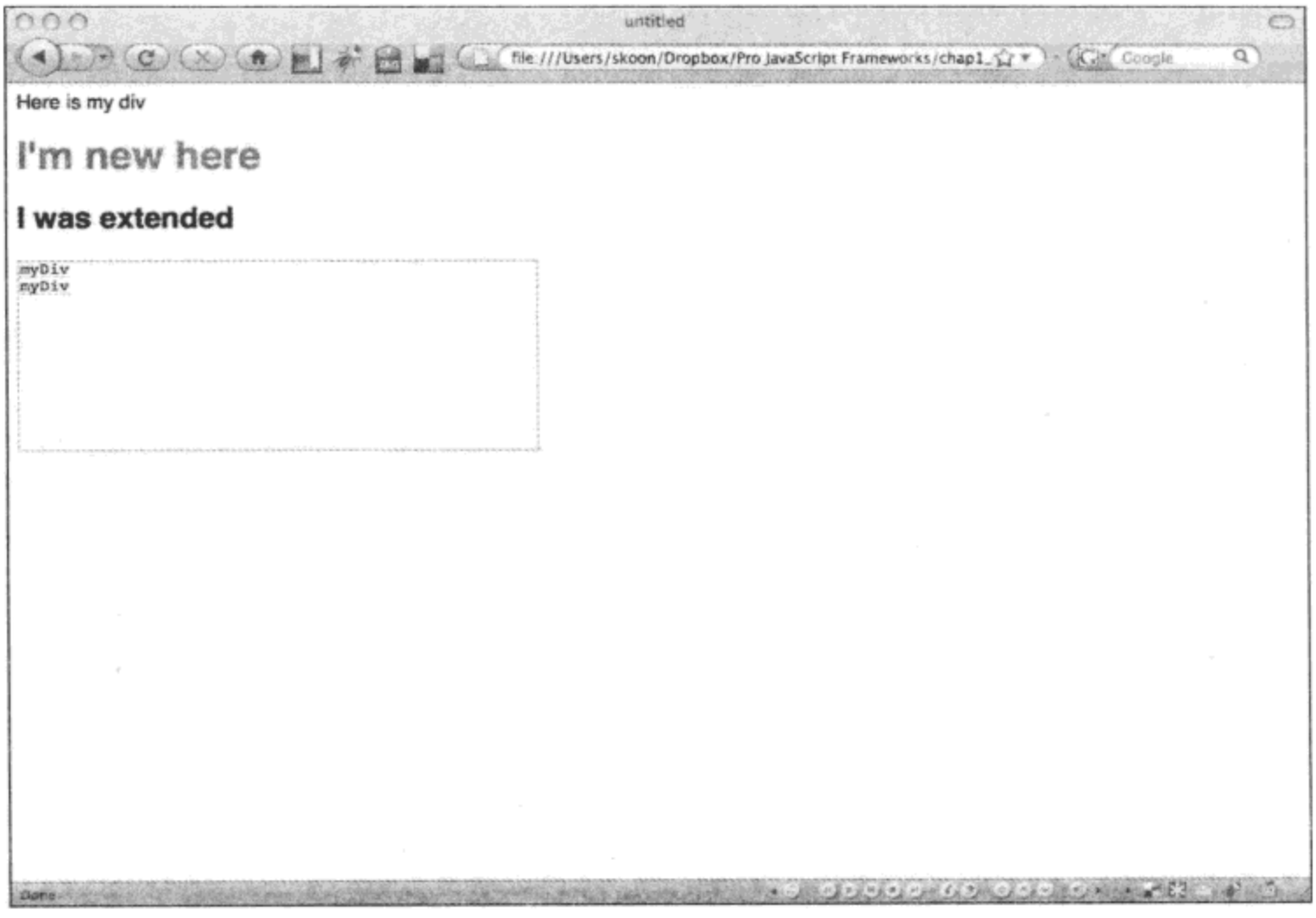


图　1-1

1.2　DOM 导航

尝试指出感兴趣的元素在 DOM 中的位置以及这个元素周围有哪些元素并不是一项简单的任务。Prototype 的 Element 对象提供了遍历 DOM 的多种方式。可以使用几个方法来指定 CSS 规则以缩小搜索范围。Prototype 的所有 DOM 导航方法都会忽略空白，而只返回元素节点。

1.2.1　adjacent 方法

这个方法查找某个元素匹配指定选择器的所有兄弟节点。对于处理列表或者表列来说，这个方法非常有用。

```
<body>
   <ul id="PeopleList">
         <li class="female" id="judy"> Judy </li>
         <li class="male" id="sam"> Sam </li>
         <li class="female" id="amelia"> Amelia </li>
         <li class="female" id="kim"> Kim </li>
         <li class="male" id="scott"> Scott </li>
         <li class="male" id="brian"> Brian </li>
         <li class="female" id="ava"> Ava </li>
   </ul>
```

```
    <textarea id="results" cols="50" rows="10"> </textarea>
    <script type="text/javascript" src="prototype-1.6.0.2.js"> </script>
    <script type="text/javascript">
          Event.observe(window,"load", function(e) {
               var els =$("kim").adjacent("li.female");
               $("results").value = "";
               for(var i = 0;i <els.length; i++) {
                    $("results").value += els[i].id + "\n";
               }
          });
     </script>
</body>
```

在这里，从 ID 为“kim”的列表元素开始，然后收集附近所有具有类名 female 的 li 元素，如图 1-2 所示。

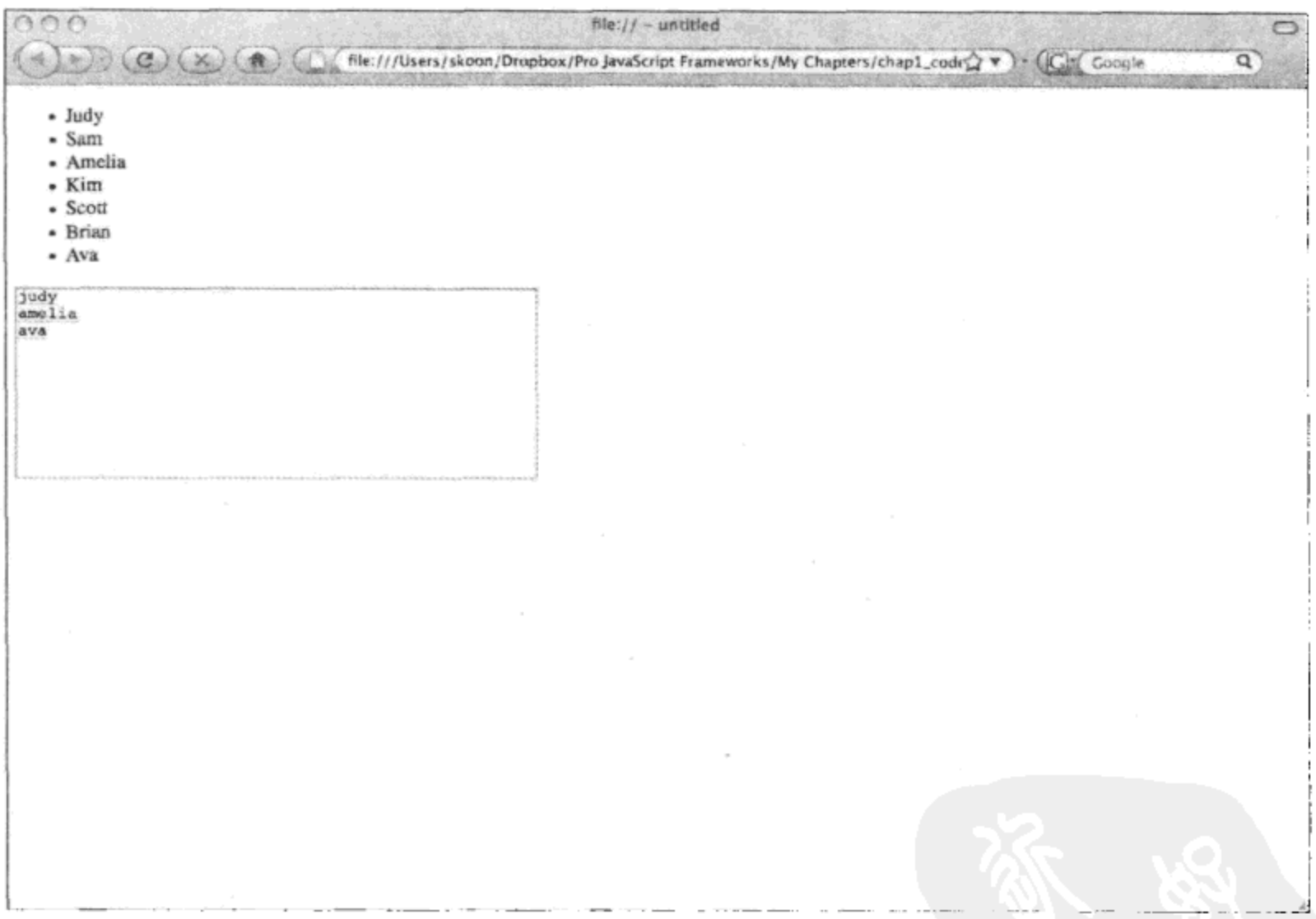


图　1-2

1.2.2　ancestors 方法

这个方法按照节点顺序收集该元素的所有祖先节点。任何给定元素的最后一个祖先节点总是 HTML 元素。在 HTML 元素上调用这个方法将返回一个空数组。给定下面的 HTML 代码片段：

```
<html>
      <body>
```

```
            <div id="myDiv">
                    <p id="myParagraph"> Hello pops </p>
            </div>
        </body>
</html>
```

这将返回一个带有元素的数组，其中元素的顺序如下：

```
DIV @--> BODY @--> HTML
```

可以使用下面的代码验证该行为：

```
<html>
        <body>
                <div id="myDiv">
        <p id="myParagraph"> Hello pops </p>
        </div>
        <textarea id="results" cols="50" rows="10"> </textarea>
        <script type="text/javascript" src="prototype-1.6.0.2.js"> </script>
        <script type="text/javascript">
            Event.observe(window,"load", function(e) {
                var a = $('myParagraph').ancestors();
                $('results').value = "";
                for(var i = 0;i <a.length;i++) {
                     $('results').value += a[i].tagName + "\n";
                }
            });
        </script>

    </body>
</html>
```

1.2.3 up/down/next/previous 方法

这 4 个方法构成了 Prototype 的核心 DOM 遍历功能。它们可用来定义一个起始元素，然后在需要的时候遍历 DOM。所有这些方法都是可链接的(chainable)，这样就可以在前一个函数返回的元素上接连地调用其中的任何一个方法。如果没有找到匹配您所定义的条件的元素，就会返回 undefined。每个方法均接受两个实参：一个 CSS 选择器或一个数字索引。如果没有传递实参，就会返回第一个匹配条件的元素。如果传入一个索引，就返回元素的对应数组中该索引指向位置的元素。例如，down()方法使用的最终数组将匹配该元素的子节点数组。如果传入的是 CSS 选择器，就会返回第一个匹配规则的元素。如果同时传入索引和 CSS 规则，那么首先处理 CSS 规则，然后使用索引选择由该 CSS 规则定义的数组中的元素。

1. up 方法

返回第一个匹配指定索引和/或 CSS 规则的祖先节点。如果没有祖先节点匹配该条件，就返回 undefined。如果没有指定实参，就返回该元素的第一个祖先节点。该方法等同于调用 element.parentNode，然后通过 Element.extend 传递父节点。

2. down 方法

返回第一个匹配指定索引和/或 CSS 规则的子节点。如果没有匹配该条件的子节点，就返回 undefined。如果没有指定实参，就返回该元素的第一个子节点。

3. next 方法

返回该元素的兄弟节点中排在该元素之后且匹配指定索引和/或 CSS 规则的节点。如果没有兄弟节点匹配该 CSS 规则，则考虑后面的所有兄弟节点。如果该元素后面没有兄弟节点，就返回 undefined。

4. previous 方法

返回该元素的兄弟节点中排在该元素之前且匹配指定索引和/或 CSS 规则的节点。如果没有兄弟节点匹配该 CSS 规则，则考虑前面的所有兄弟节点。如果该元素前面没有兄弟节点，就返回 undefined。

采用类似下面示例的 HTML 代码片段，其中定义了 4 个具有如下关系的元素：

```
<div id="up">
        <p id="prevSibling"> I'm a sibling </p> <div id="start"> <p id="down"> Start
Here </p> </div> <span id="nextSibling"> I'm next </span>
</div>
```

这段代码从 div 节点 start 开始，然后查看前面的、后面的、上面的和下面的元素。从具有 ID start 的元素开始。包含文本“Start Here”的段落元素是起始元素的第一个子节点，因此调用 down 方法就会返回该元素。up 方法返回的是 div 节点 topDiv。previous 方法返回 sibling 段落元素，而 next 方法返回 span 节点 nextSibling，如图 1-3 所示。

```
<body>
     <div id="up">
          <p id="prevSibling"> I'm a sibling </p> <div id="start"> <p id="down"> Start
Here </p> </div> <span id="nextSibling"> I'm next </span>
     </div>
     <textarea id="results" cols="50" rows="10"> </textarea>
     <script type="text/javascript" src="prototype-1.6.0.2.js"> </script>
     <script type="text/javascript">
          Event.observe(window,"load", function(e) {
               var startEl = $('start');
               var previousEl = startEl.previous();
               var upEl = startEl.up();
               var downEl = startEl.down();
               var nextEl = startEl.next();

               var resultTextArea = $("results");
               resultTextArea.value = "";
               resultTextArea.value += "start =" + startEl.id + "\n";
               resultTextArea.value += "previous =" + previousEl.id + "\n";
               resultTextArea.value += "next =" + nextEl.id + "\n";
               resultTextArea.value += "down =" + downEl.id + "\n";
```

```
                resultTextArea.value += "up =" + upEl.id + "\n";
            });
        </script>
    </body>
```

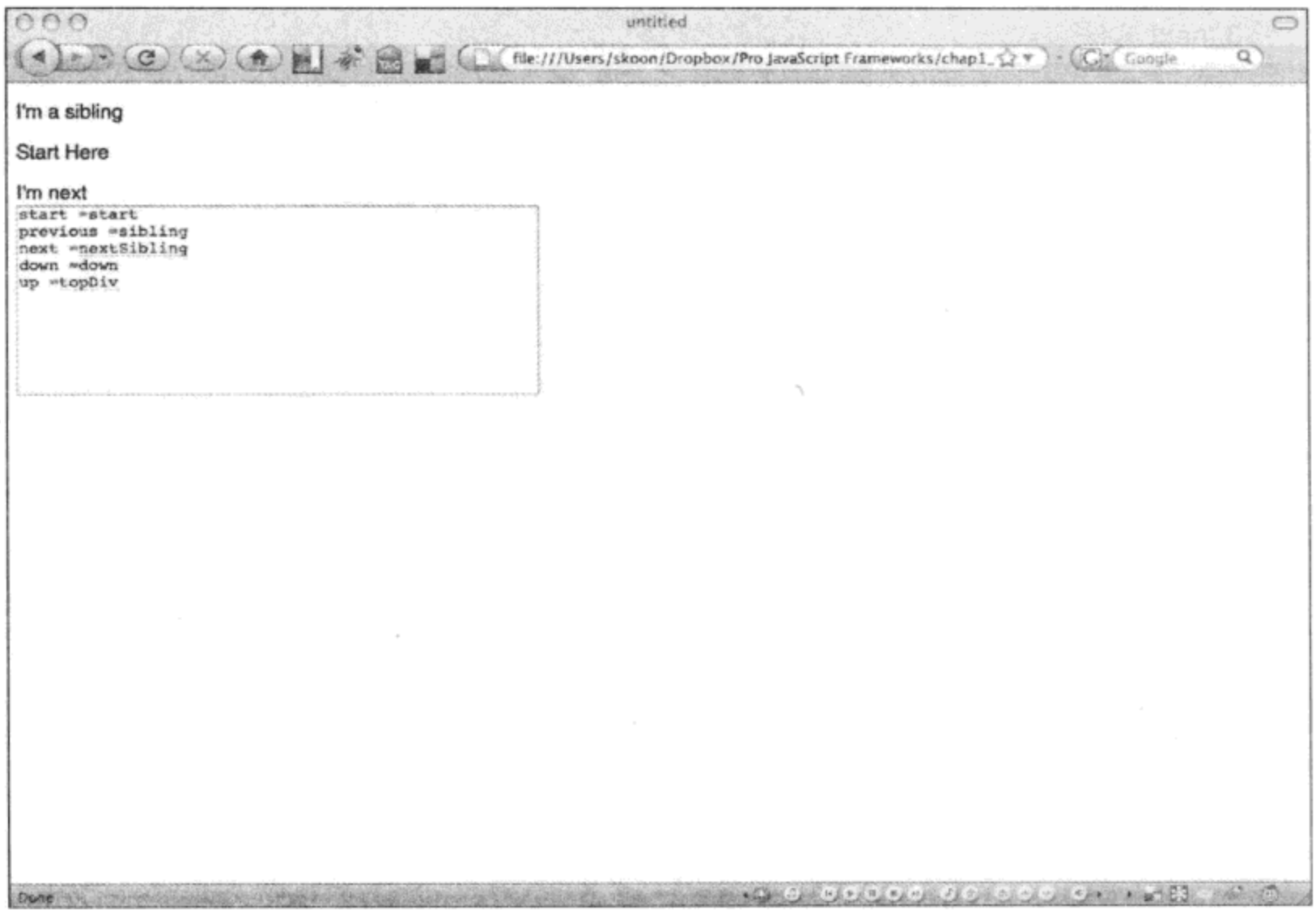


图　1-3

1.2.4　descendants/descendantOf/firstDescendant/immediateDescendants 方法

所有这些方法都用来处理给定元素的子节点。descendants 和 immediateDescendants 方法返回的是子元素数组。

- descendants　这个方法返回包含该元素的子节点的数组。如果该元素没有子节点，就会返回一个空数组。
- descendantOf　这个方法返回一个 Boolean 值，用于说明给定元素是否为给定祖先节点的子节点。
- firstDescendant　这个方法返回给定元素的第一个本身也是元素的子节点。
- immediateDescendants　(已不再使用)这个方法返回下一层元素(但不包括更低层次的元素)组成的数组。

1.2.5　getElementsBySelector 和 getElementsByClassName 方法

这些方法可用来根据元素的特性选取一组元素，或者定位并操作选中的元素。这两个方法都

已经不再使用，应该使用$$()方法代替它们。

1.2.6　childElements 函数

这个有用的函数用来收集指定元素的所有子节点并将其作为扩展元素的数组返回。返回元素的顺序与它们在 DOM 中的顺序相同。因此，索引 0 处的元素就是距离父元素最近的子节点，依此类推。

1.3　修改页面内容

Prototype 提供了 4 个修改页面内容的方法，分别是 insert、remove、replace 和 update。可以使用 Element 对象调用这些方法，并且可以将这些方法添加到任何经过扩展的元素中。这些方法都带有一个可选的实参，也就是待修改的元素。insert 和 replace 方法会对包含在传递给它们的内容中的任何脚本标记调用 eval()方法。在这些方法中，所有带有 content 实参的方法都接受纯文本、HTML 片段或者支持 toString()方法的 JavaScript 对象。

1.3.1　insert(element, content)和 insert(element, {position:content})

insert 方法将您提供的内容插入到元素中。如果没有指定位置(例如 top、bottom、before 或 after，分别表示顶部、底部、前面和后面)，那么该内容将会附加到元素的后面。对于动态地插入从 Web 服务获取的内容，或者出于性能方面的考虑而一次只往页面中加载一部分元素，这就是非常有用的方法。

```
<script type="text/javascript">

        function insertSample() {
             $("MainDiv").insert("New Content added at the end by default");
             $("MainDiv").insert({top:"Added at the top"});
             $("MainDiv").insert({before:"Added before the element"})
             $("MainDiv").insert({after:"Added after the element"});
             $("MainDiv").insert({bottom:"Added at the bottom"});
       };
       insertSample();
</script>
```

1.3.2　remove

在经过扩展的元素上调用 remove 方法会将其从 DOM 中完全移除。这个函数返回移除的元素。当用户选择删除 UI 中某个元素表示的项之后，最常见的操作就是使用这个方法移除该元素。

```
<body>
   <table id="myTable">
         <tr id="firstRow"> <td> First Row </td> </tr>
         <tr id="secondRow"> <td> Second Row </td> </tr>
   <tr id="thirdRow"> <td> Third Row </td> </tr>
```

```
        </table>
        <script type="text/javascript" src="prototype-1.6.0.2.js"> </script>
        <script type="text/javascript">

            function removeRow() {
                  $("secondRow").remove();
            };
            removeRow();
        </script>
</body>
```

1.3.3 replace

replace 方法将指定的元素移走并将其替换成 content 实参提供的元素。这个方法将指定的元素及其子节点从 DOM 中移除。

```
<body>
      <div id="MainDiv">
            <div id="tempDiv"> Place holder </div>
      </div>
      <script type="text/javascript" src="prototype-1.6.0.2.js"> </script>
      <script type="text/javascript">
            //simulate loading content from a web service
            setTimeout(function () {
                  $("MainDiv").replace(" <h1> Replaced Content </h2> ");
                }, 1000);
          </script>
</body>
```

1.3.4 update

update 方法将元素的内容替换成指定的内容。虽然该方法并不会把该元素从 DOM 中移除，但是它确实会将该元素的所有子节点全部移除。

```
<body>
      <div id="MainDiv"> Here is some content to be updated </div>
      <script type="text/javascript" src="prototype-1.6.0.2.js"> </script>
      <script type="text/javascript">

      //simulate loading content from a web service
      setTimeout(function() {
            $("MainDiv").update("updated the content");
      }, 1000);
      </script>
</body>
```

1.4　操作元素的大小、位置和可见性

对于在不同的浏览器中处理 DOM 而言，最困难的事情之一就是获取 DOM 中所包含元素的大小。对于如何指定 DOM 中元素的大小以及一个元素的大小如何影响它周围元素的流式布局，每种浏览器都有各自不同的行为。

1.4.1　放置元素

设置元素的位置是现代网页设计的基础操作之一。在设计动态网页时，需要能够四处移动元素，将这些元素放置在页面上需要用到它们的位置。要把一个元素放到页面的精确位置上，应该首先设置它的位置 CSS 样式。将位置样式规则设置为 absolute 意味着这个元素的 top 和 left 坐标是相对于文档的左上角计算而来的。将位置样式设置为 relative 则意味着，使用相对于容器块左上角计算而来的数值放置元素。Prototype 提供了一些方法可用来轻易地设置元素的位置样式。

1. makePositioned 和 undoPositioned

这两个方法可用来轻易地让 DOM 中的元素成为 CSS 定位块。在一个元素上调用 makePositioned 方法会将其位置样式设置为 relative(如果它当前的位置样式是 static 或 undefined)。undoPositioned 方法将该元素的位置样式设置回调用 makePositioned 方法之前的值。

```
$("myElement").makePositioned();
$("myElement").undoPositioned();
```

2. absolutize 和 relativize

这两个方法修改给定元素的定位设置(分别将位置样式设置为 absolute 或 relative)。

```
$("myElement").absolutize();
$("myElement").relativize();
```

3. clonePosition

这个方法创建一个新元素，它的位置和大小与当前元素完全相同。可以通过使用一个包含表 1-1 中选项的可选参数来指定将哪些设置应用到这个新元素上。

表　1-1

设　置	说　明
setLeft	应用源元素的 CSS left 属性。默认为 true
setTop	应用源元素的 CSS top 属性。默认为 true
setWidth	应用源元素的 CSS width 属性。默认为 true
setHeight	应用源元素的 CSS height 属性。默认为 true
offsetLeft	让新元素的 CSS left 属性偏移 n 个单位。默认为 0
offsetTop	让新元素的 CSS top 属性偏移 n 个单位。默认为 0

1.4.2　处理偏移

Prototype 的 Element 对象上有几个不同的方法，它们使得查找元素的偏移变得更加简单。

1. cumulativeOffset、positionedOffset 和 viewportOffset

这里的每个方法都返回两个数值，也就是以{ left: number, top: number}形式返回的给定元素的 top 和 left 值。cumulativeOffset 方法返回指定元素距离文档左上角的总偏移。positionedOffset 方法返回给定元素最近的已定位(也就是位置样式设置为 static)祖先的总偏移。viewportOffset 方法返回该元素相对于视口的偏移。

2. getOffsetParent

这个方法返回该元素的最近的已定位祖先，如果没有找到其他祖先，就会返回 body 元素。

下面的代码演示如何计算不同的偏移。在这段代码中有两个元素：父元素 div 和一个子元素。父元素自己的位置样式设置为 absolute，它的位置距离文档顶部 240 像素，距离文档左侧 50 像素。当使用 ID 为 start 的元素调用 getOffsetParent 方法时，返回已定位元素 positionedParent。textarea 元素 results 没有已定位的祖先。如果在该元素上调用 getOffsetParent()方法，就会返回 body 元素。因为 start 元素本身不是已定位元素，所以在它上面调用 positionedOffset()方法会返回 0,0，如图 1-4 所示。

```
<body>
    <div id="positionedParent" style="position:absolute;border:1px solid black;
top:240px;left:50px;">
        <div id="start"> Start Here </div>
    </div>

  <textarea id="results" cols="50" rows="10"> </textarea>
  <script type="text/javascript" src="prototype-1.6.0.2.js"> </script>
  <script type="text/javascript">
      Event.observe(window,"load", function(e) {
          $("results").value = "";
          $("results").value += "offsetParent = " + $("start").getOffsetParent()
.id + "\n";
          $("results").value += "cumulativeOffset = " + $("start")
.cumulativeOffset() + "\n";
          $("results").value += "positionedOffset = " + $("start")
.positionedOffset() + "\n";
          $("results").value += "parent positionedOffset = " + $("start")
.parentNode.positionedOffset() + "\n";
      });
    </script>
</body>
```

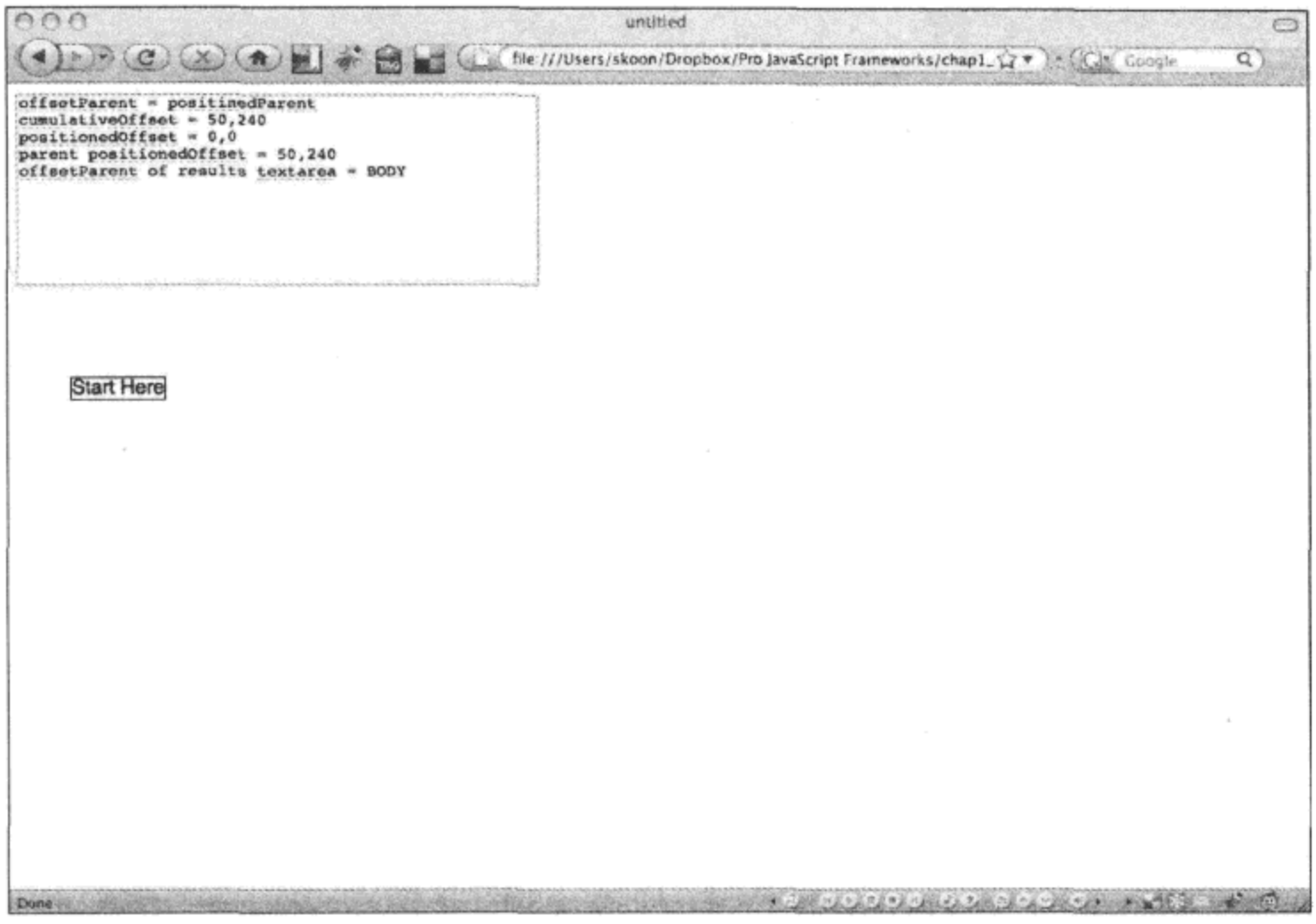


图　1-4

1.4.3　显示和隐藏元素

从编写第一个脚本标记开始，显示和隐藏元素就已经成为 Web 开发人员工具箱的一部分。

1. show 和 hide

这两个方法可用来快速修改元素的可见性，它们通过将元素的显示 CSS 样式设置为 none 来实现这一点。将显示样式设置为 none 会把该元素从文档流中移除，从而让浏览器显示页面中的其他元素，就好像该元素不存在一样。

```
$("myElement").show();
$("myElement").hide();
```

2. setOpacity

这个方法设置给定元素的不透明度，将您从处理各种浏览器之间不一致性的琐碎细节中解脱出来。该方法带有一个浮点数值，其中 0 表示完全透明，1 表示完全不透明。使用这个方法相当于通过 CSS 类或使用 setStyle 方法(传入一个表示不透明度的值)来设置不透明度。

```
$("myElement").setOpacity(0.5);
```

1.4.4　调整元素的大小

当在屏幕上表示元素以及计算元素的大小时，每种浏览器都有相关的某种程度的古怪行为。不同的浏览器采用不同的方式计算元素的计算样式。Prototype 处理各种差异，返回对应浏览器的正确的计算样式。

1. getDimensions、getHeight 和 getWidth

使用这些方法，可以在运行时获得指定元素的计算大小。getDimensions()方法返回一个包含该元素的计算高度和宽度的对象。在调用 getDimensions()方法时，最好将返回的值保存到一个局部变量中，然后引用这个变量，而不是多次调用该方法。如果只需要宽度或高度，那么最好只调用适当的对应方法。

```
Var dimensions = $('myDiv').getDimensions();
Var currentWidth = dimensions.width;
Var currentHeight = dimensions.height;
```

2. makeClipping 和 undoClipping

CSS 属性 clip 可用来定义如果元素内容的宽度或高度超出该元素允许的宽度或高度，那么是否应该显示该元素的内容。因为 clip 属性在各种浏览器之间的支持非常糟糕，所以 Prototype 提供了这个方法来自动将元素的 overflow 属性设置为 hidden。可以使用 undoClipping()方法让元素按照正常方式调整大小。

1.5　处理 CSS 和样式

为了响应一些事件而需要标记元素，这时候 CSS 类就非常有用。假设正在创建一个在线调查表单，希望将某些字段标记成必须填写，但又不想通过 ID 来获取每个必须填写的元素，然后逐个地检查这些元素以确保用户输入的是适当的值。可以创建一个名为 required 的 CSS 类，并将其应用到每个需要用户输入值的元素上。有时为了响应某个用户驱动的事件或者数据驱动的事件，需要在运行时修改某个元素的样式或类，例如将表中的某一行的状态由只读变为可编辑。在所有 Web 开发人员的工具箱中，CSS 类都是一款极具价值的工具。Prototype 使得我们能够更加容易地向 DOM 中的元素应用或移除 CSS 类。

1.5.1　addClassName、removeClassName 和 toggleClassNames

这 3 个方法都能够修改给定元素的 className 属性。所有方法都检查该元素是否确实具有给定的类名。当需要在某个元素上设置 CSS 类或者需要将某个 CSS 样式开启或关闭时，这些方法就非常有用。这些方法的名称已经说明了它们的作用。

```
<head>
      <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title> untitled </title>
  <style>
```

```
        .invalid { background:red;}
    </style>
</head>
<body>
    <form id="myForm" method="post">
        First Name: <input type="text" id="firstName" class="required"> <br/>
        Last Name: <input type="text" id="lastName" class="required"> <br/>
        Age: <input type="text" id="age"> <br/>
        <input type="button" id="submitButton" value="submit">
    </form>
    <script type="text/javascript" src="prototype-1.6.0.2.js"> </script>
    <script type="text/javascript">

        function init() {
            var requiredInputs = $$(".required");
            for(i = 0;i <requiredInputs.length;i++) {
                $(requiredInputs[i]).insert({"after":"*required"});
                Event.observe(requiredInputs[i], "change", function(e) {
                    if(this.hasClassName("invalid")) { this.removeClassName
("invalid")};
                });
            };
            Event.observe("submitButton", "click", validateUserInput);

        };

    function validateUserInput() {
            var requiredInputs = $$(".required");
            for(var i = 0; i <requiredInputs.length; i++) {
                if(requiredInputs[i].value == "") {
                    requiredInputs[i].addClassName("invalid");
                } else {
                    if (requiredInputs[i].hasClassName("invalid")) {
                        requiredInputs[i].removeClassName(invalid);
                    };
                };
            };
        };    Event.observe(window, 'load', init);
    </script>
</body>
```

一项常见的 JavaScript 任务就是表单验证。在这里，我们已经建立了一个简单的表单，并定义了一条简单的规则，用户必须在具有 required 类的元素中输入一些文本。利用$$()方法并传入一个 CSS 选择器，收集所有具有 required 类的元素，这样就可以实施这条规则。一旦有了包含这些元素的数组，就可以遍历该数组，检查每个元素的 value 属性是否不等于空字符串。如果确实等于空字符串，那么使用 addClass()方法将 invalid 类添加到该元素中。然后检查该元素是否已经有 invalid 类，而且用户已经输入了文本。如果某个元素包含文本，而且带有 invalid 类，那么将这个类从该元素中移除，这是因为该元素通过了验证规则，分别如图 1-5 和图 1-6 所示。

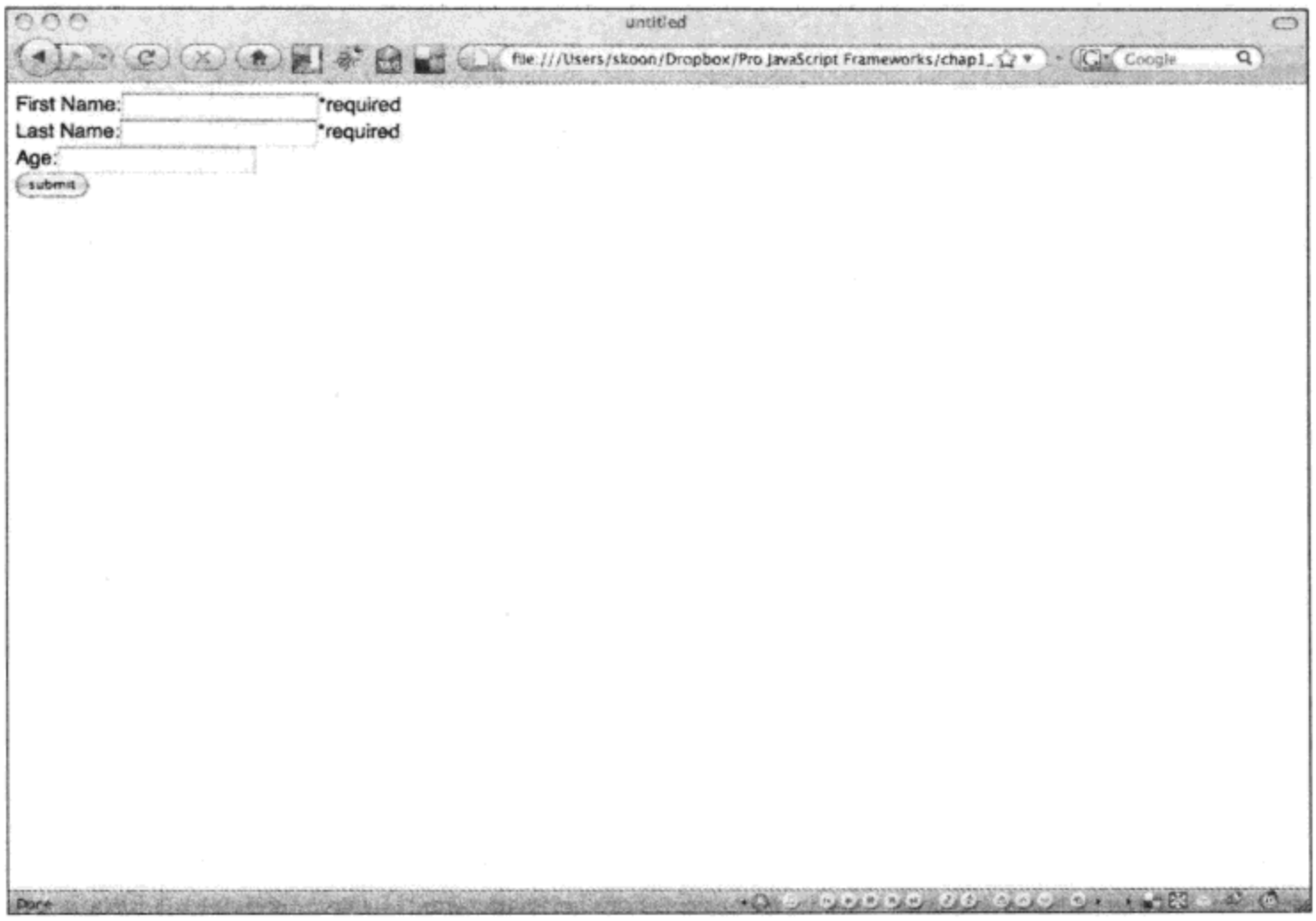

untitled
file:///Users/skoon/Dropbox/Pro JavaScript Frameworks/chap1_
Google
First Name: *required
Last Name: *required
Age:
submit
Done

图　1-6

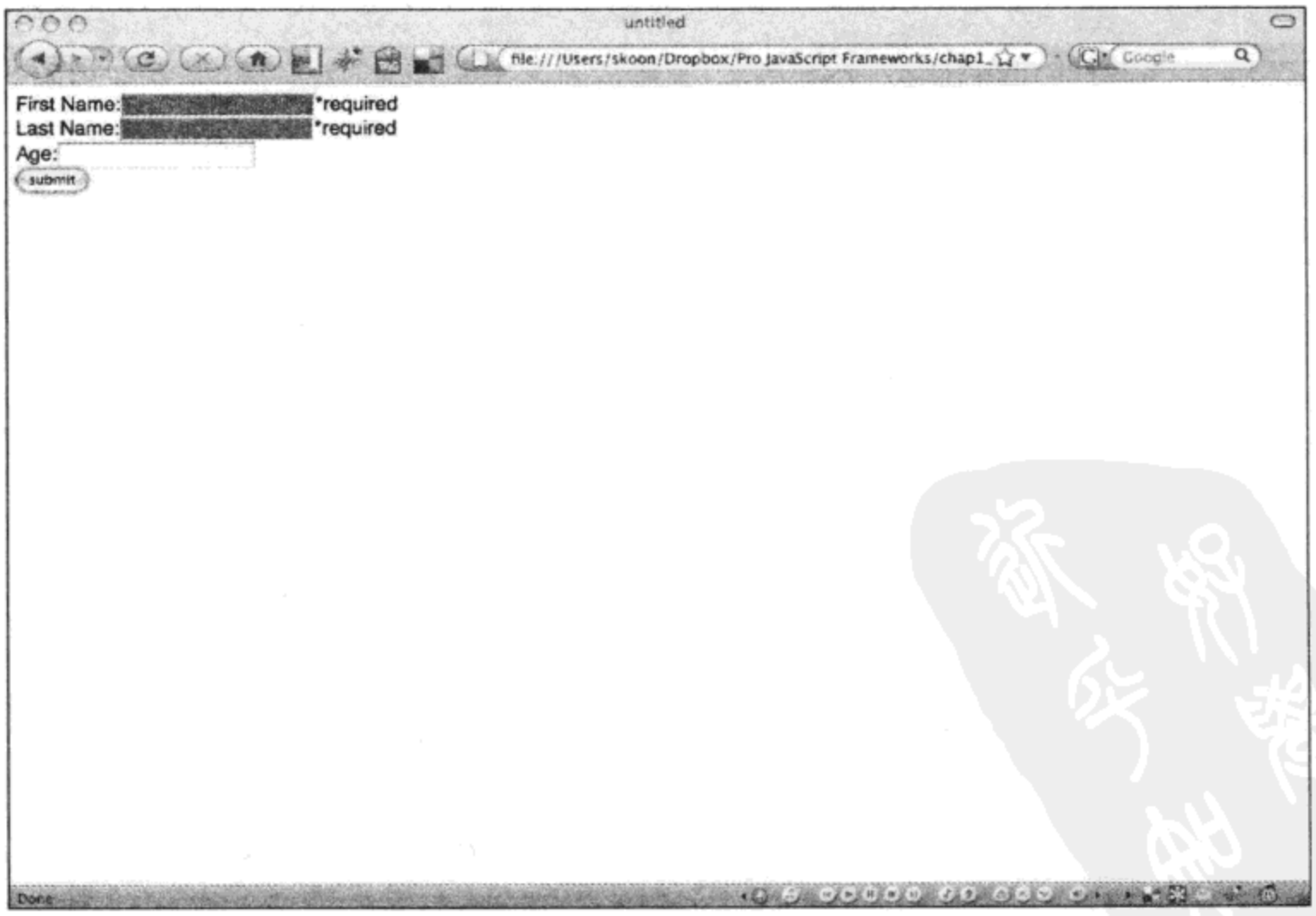

untitled
file:///Users/skoon/Dropbox/Pro JavaScript Frameworks/chap1_
Google
First Name: *required
Last Name: *required
Age:
submit
Done

图　1-6

1.5.2　hasClassName 和 classNames

这两个方法用来判断有哪些 CSS 类已经应用到指定元素中。hasClassName()方法可用来确定给定元素的 className 属性中是否有该类名。classNames()方法已经不再使用，它返回一个包含所有已经应用到该元素中的类的数组。

1.5.3　setStyle 和 getStyle

这两个方法可用来快速地在元素上设置样式或者获取特定样式的值。可以只查询由文档对象模型(Document Object Model，DOM)Level 2 样式规范(Level 2 Style Specification)定义的样式。要在指定元素上设置某个样式，需要传入一个表示希望设置样式的键-值对的对象散列。

```
El.setStyle( { "font-family": "Arial", "color" : "#F3C" });
```

要获取特定样式的值，可以传入该样式的名称作为实参。

```
El.getStyle("font-size");
```

注意：

Internet Explorer 浏览器返回字面值，而所有其他浏览器则返回样式计算后的值。例如，如果将 font-size 指定为 1em，那么 IE 浏览器将返回 1em，而其他浏览器则可能返回表示该 font-size 的像素值。

1.6　使用自己编写的方法扩展 Element 对象

Prototype 允许使用 addMethods 方法非常容易地向 Element 对象中添加自己的方法。addMethods 方法的实参是希望添加的方法所构成的散列。假设希望向所有元素添加一个方法，用来将空白从元素的文本中移除。下面就是这个函数的基本代码：

```
function removeWhiteSpace(element) {
       if(element.innerText) {
            return element.innerText.replace(" ", "", "gi");
       } else if(element.textContent){
            return element.textContent.replace(" ", "", "gi");
       }
};
```

首先，需要稍微修改这个方法以符合 Prototype 的规定。然后就可以调用 Element.addMethods 方法。

```
<body>
     <div id="myDiv"> Remove the whitespace </div>
     <script type="text/javascript" src="prototype-1.6.0.2.js"> </script>
     <script type="text/javascript">
          var myFunc = {
               removeWhitespace : function (element) {
                    if(element.innerText) {
```

```
                    return element.innerText.replace(" ", "", "gi");
                } else if(element.textContent){
                    return element.textContent.replace(" ", "", "gi");
                }
            }
        };

    Element.addMethods(myFunc);

    alert($("myDiv").removeWhitespace());
    </script>
</body>
```

这里所做的工作就是将这个函数使用一个内部对象包装起来，从而让 addMethods()方法得以运行。可以更进一步，返回该元素自身，这样就可以执行链式调用。

```
var myFunc = {
     removeWhitespace : function (element) {
        if(element.innerText) {
            element.innertText = element.innerText.replace(" ", "", "gi");
        } else if(element.textContent){
            element.textContent = element.textContent.replace(" ", "", "gi");
        }
        return element;
     }
};
```

现在，这个方法已经可以交给 Prototype 调用。

1.7　本章小结

在本章中，我们讲解了利用 Prototype 可以非常容易地根据 ID、CSS 类以及相对于其他元素的位置来获取 DOM 元素的引用。Prototype 在任何可能的情况下都会自动地向元素中添加辅助方法，并在您使用 Element.extends()、$()或$$()方法获取某个元素的引用时添加这些方法。Prototype 还简化了与放置元素和查找给定元素大小相关的操作。

处理跨浏览器事件

事件处理是编写现代 Web 应用程序时比较棘手的一部分。Internet Explorer 浏览器和 W3C 具有不同的事件处理模型。Internet Explorer 浏览器支持 attachEvent 方法，该方法用来向元素中添加事件处理程序。W3C 标准定义了 addEventListener 方法来完成同样的事情。Prototype 提供了连接事件处理程序的跨浏览器方法，并扩展了事件对象，使其具有多个有用的方法。

本章内容简介：

- 使用 Event.observe()方法连接事件处理程序
- 响应事件，包括键盘和鼠标事件
- 定期触发事件

2.1 注册事件处理程序

在各种 JavaScript 框架出现之前，大多数 Web 开发人员必须按如下方式连接事件处理程序：

```
var myInput = document.getElementById("myInput");
     if(myInput.addEventListener ) {
          myInput.addEventListener('keydown',this.keyHandler,false); //W3C
method
     } else if(myInput.attachEvent ) {
          myInput.attachEvent('onkeydown',this.keyHandler); //IE method
     };
```

那么，为什么使用这些方法之一来连接事件处理程序，而不是使用所有浏览器都支持的事件特性(如 onClick)以及 DOM Level 0 属性(如 onclick 和 onload)呢？所有这些属性在同一时刻只能指向一个事件处理程序。如果希望在窗口的 onload 事件触发期间调用多个函数，那么必须定义一个函数来调用所有其他的函数，并将该函数赋给 window.onload 事件。如果有任何其他的代码(例如某个第三方窗口部件库)将一个事件处理程序赋给 window.onload 事件，就不会调用您定义的函数。

Prototype 的 Event 对象为把多个事件处理程序连接到一个元素上提供了一种简单的方式。它还提供了访问有关该事件、触发该事件的元素以及在触发事件附近带有给定标记名称的第一个元素的信息的跨浏览器方式，以及停止事件的默认操作的方式。

Event.observe()

Event.observe()方法的通用形式如下：

```
Event.observe(element, eventName, handler [,useCapture = false]);
```

该方法的实参如下：

- element　将事件处理程序绑定到这个元素上。可以传入这个元素的引用或该元素的字符串 ID。
- eventName　这是该事件的 W3C DOM Level 2 标准名称。
- handler　应该处理该事件的函数。这可以是一个预先声明的函数或匿名函数。
- useCapture　这个函数确定是否使用事件捕获或事件冒泡(event bubbling)。大多数时候不需要使用事件捕获，因此采用默认值 false 即可。

注意：
为了将一个事件处理程序绑定到元素的某个事件上，在尝试绑定时该元素必须存在于 DOM 中。

```
<body>
     <ul id="PeopleList">
          <li class="female" id="judy"> Judy </li>
          <li class="male" id="sam"> Sam </li>
          <li class="female" id="amelia"> Amelia </li>
          <li class="female" id="kim"> Kim </li>
          <li class="male" id="scott"> Scott </li>
          <li class="male" id="brian"> Brian </li>
          <li class="female" id="ava"> Ava </li>
     </ul>

     <textarea id="results" cols="50" rows="10"> </textarea>
     <script type="text/javascript" src="prototype-1.6.0.2.js"> </script>
<script type="text/javascript">
          $("results").value = "";
          Event.observe("PeopleList", "click", function(e) {
             $("results").value += "clicked on " + e.target.id + "\n";
          });
     </script>
</body>
```

图 2-1 给出了一个表示人员列表的无序列表。您希望了解人员的姓名，也就是 li 元素(用户单击的元素)的 id。

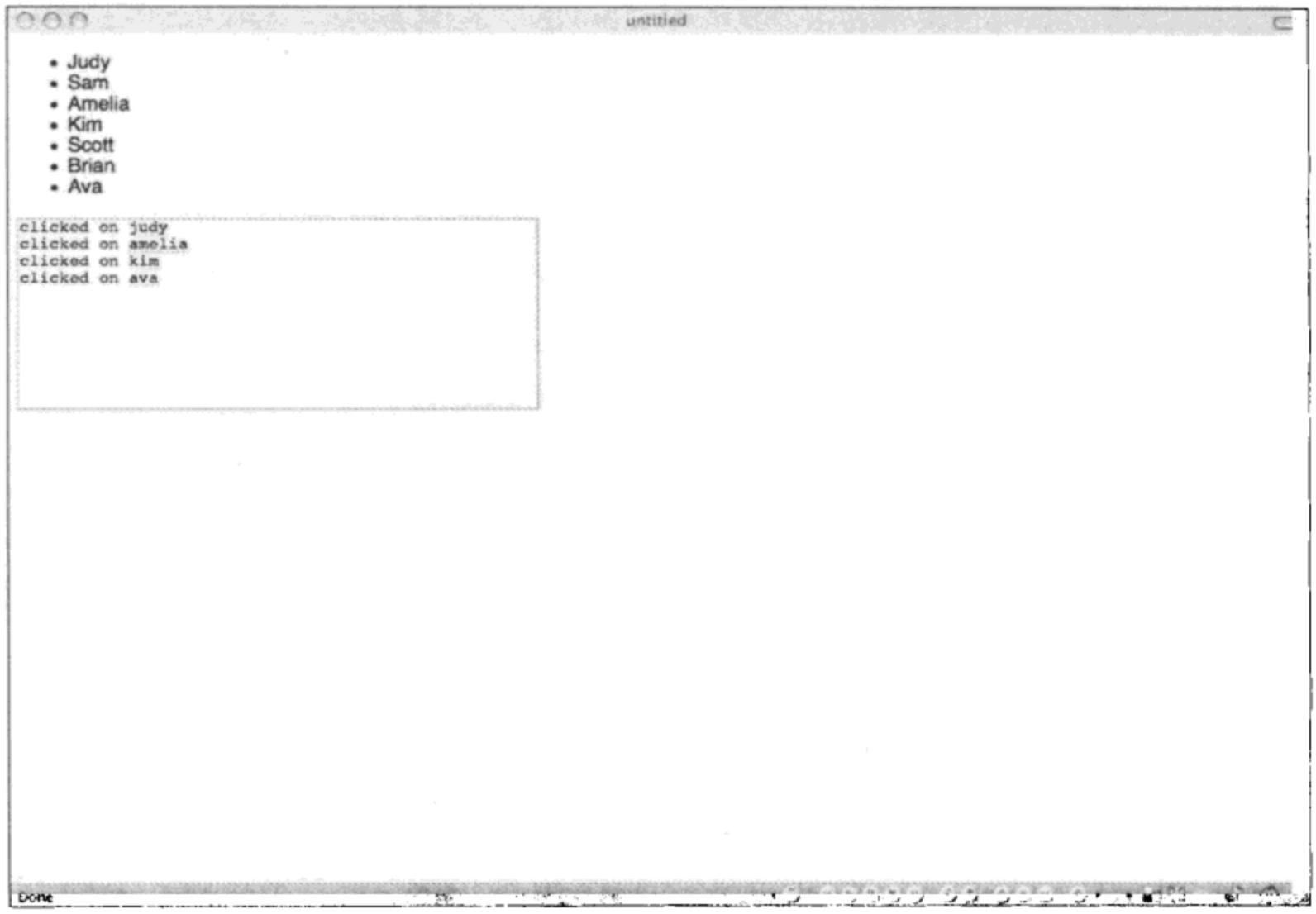


图　2-1

2.2　响应事件

当事件触发时捕获该事件只是完成了一半的工作，我们还必须响应该事件。所有浏览器都提供了 event 对象(包含有关该事件的信息)以及一些方法(用于停止事件)。Internet Explorer 浏览器将 event 对象放在全局的 window 对象中，而 W3C 标准规定将 event 对象传递给事件处理程序。最终需要编写类似于如下的代码：

```
function clickHandler(evt) {
        evt = evt || window.event;
target = evt.target || evt.srcElement;
        ///remainder of the event handler goes here
};
```

Prototype 提供了 Event 对象，可用来访问一些常见的方法和属性，从而简化了事件处理。可以在事件处理程序内部访问 Event 对象，将 event 对象传递给它作为上下文，然后使用它，而不是编写上面给出的分支代码。

2.2.1　event.target 属性、this 属性和 Event.element 方法

查看本章前面给出的示例，但是不再引用 event.target，而是修改代码来引用 this 以及 Event.element()

方法返回的元素。结果如图 2-2 所示。

```
<script type="text/javascript">
        $("results").value = "";
        Event.observe("PeopleList", "click", function(e) {
                e = e || window.event;
                e.target = e.target || e.srcElement;
        $("results").value += "this= " + this.id + "\n";
        $("results").value += "Event.element(event)= " + Event.element(e).id + "\n";
        $("results").value += "event.target= " + e.target.id + "\n";
    });
</script>
```

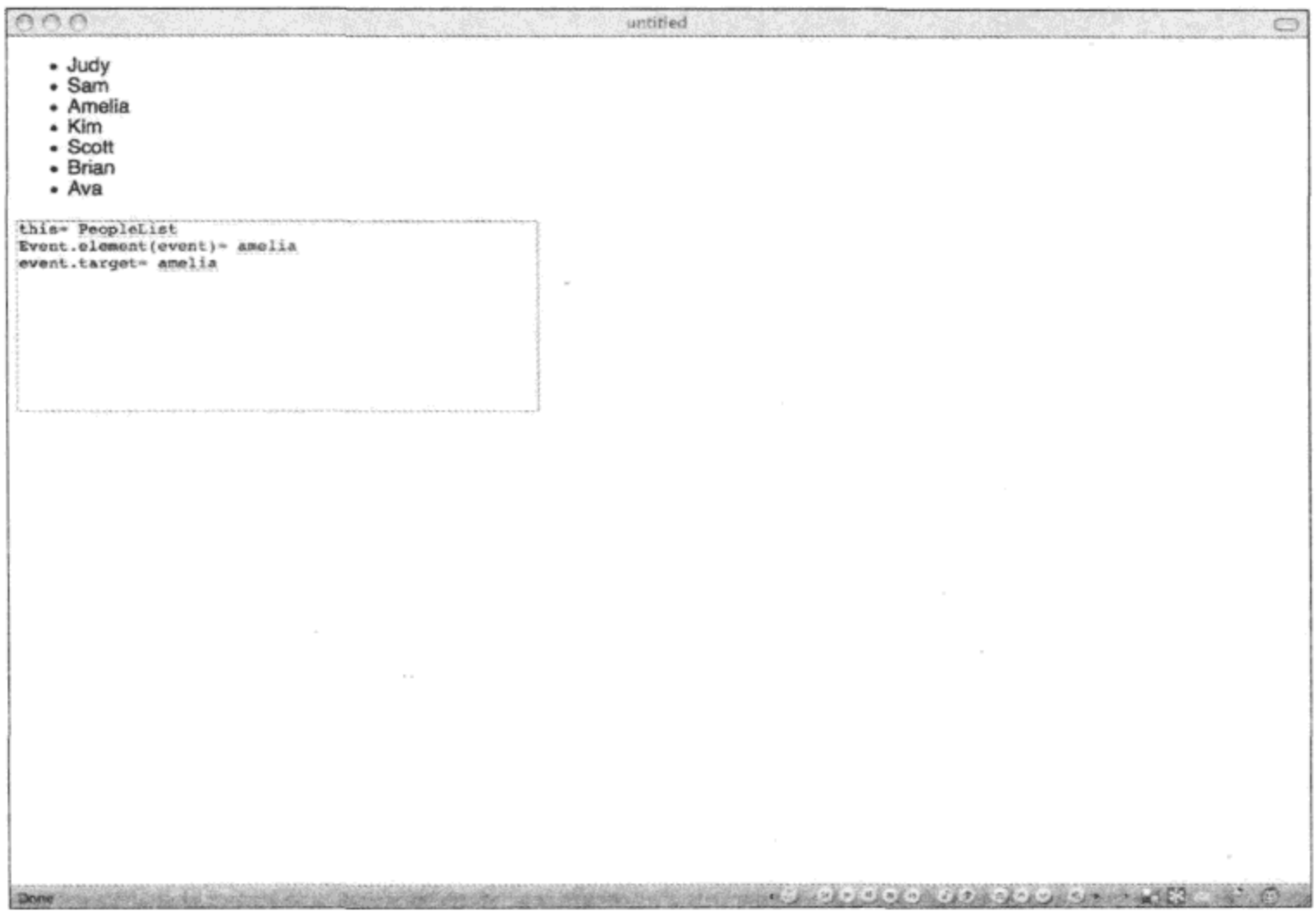


图　2-2

可以看到 Event.element()方法所返回的元素与 W3C 标准的 event.target 属性和 Internet Explorer 浏览器的 event.srcElement 属性所返回的元素是一样的。但是使用 Event.element()方法，只需要编写一行代码就可以获取引发事件的元素，而不是必须在 event 或 window.event 对象上进行对象检测。

1. Event.findElement(element, tagname)

有时我们希望查找位于触发事件的元素附近的某个元素。Event.findElement 方法从触发事件的元素向上搜索，并返回匹配标记名称(作为参数传入)的第一个元素。

返回到前面的列表示例。可以将列表包装在一个 div 标记中，然后通过调用 Event.findElement()方法(传入标记名称“div”)来获取指向这个 div 标记的引用。

```
<body>
     <div id="container">
          <ul id="PeopleList">
               <li class="female" id="judy"> Judy </li>
               <li class="male" id="sam"> Sam </li>
               <li class="female" id="amelia"> Amelia </li>
               <li class="female" id="kim"> Kim </li>
               <li class="male" id="scott"> Scott </li>
               <li class="male" id="brian"> Brian </li>
               <li class="female" id="ava"> Ava </li>
          </ul>
       </div>

       <textarea id="results" cols="50" rows="10"> </textarea>
       <script type="text/javascript" src="prototype-1.6.0.2.js"> </script>
       <script type="text/javascript">
            $("results").value = "";
            Event.observe("PeopleList", "click", function(e) {
                     e = e || window.event;
               var container = Event.findElement(e,"div");
               $("results").value = container.id;
            });
     </script>
</body>
```

注意：

Event.findElement 方法提供了一种非常简单的搜索功能，它只搜索标记名称。如果需要更加复杂的搜索，就应该通过调用 Event.element()方法并使用 up()方法来获取在其上发生该事件的元素。在 up()方法中可以使用 CSS 选择器语法。

2. stopObserving()和 unloadCache()

stopObserving 方法将事件处理程序从元素中移除(针对您指定的事件)，从而阻止该事件处理程序启动以响应事件。可以使用与第一次连接该事件时用过的相同实参来调用该方法。unloadCache()方法将使用 Event.observe 方法注册的所有事件处理程序注销。它不会移除采用其他方式添加的事件处理程序；您必须记住自己将这些事件处理程序移除。Internet Explorer 浏览器存在一个烦人的内存泄漏问题：如果在把对象从 DOM 中移除之前没有将事件处理程序从对象中移除，就会造成内存泄漏。从 1.6 版本开始，当引发 document.unload 事件时，Prototype 就会自动调用 unloadCache()方法，并在清理 DOM 对象之前将所有事件处理程序移除。

2.2.2　Event.extend(event)

这个方法扩展了 event 对象，使其具有 Event.methods 中定义的方法(前提是该对象尚未经过扩展)。所有传给事件处理程序(使用 Event.observe 方法注册它们)的事件都已经被 Prototype 扩展。只有在使用元素特性或者使用原生 DOM 方法 attachEvent 或 addEventListener 连接的事件处理程序内部才需要调用这个方法。

2.2.3　Event.stop(event)

假设希望阻止事件冒泡传给元素的父节点。Mozilla 浏览器在 Event 对象上提供了 stopPropogation 方法，而 Internet Explorer 浏览器则有 cancelBubble()方法。停止事件的传播并不会阻止引发该事件的默认操作。必须从事件处理程序中返回 false 才能阻止引发默认操作。Prototype 提供了一个方法，该方法停止事件传播并阻止引发该事件的默认操作。

下面有一个简单的输入验证脚本。如果用户没有在顶部输入框中输入单词 pass，而试图按下 Tab 键进入顶部输入框，那么该操作将被阻塞，同时弹出一个警告框，指示用户输入内容。

```
<body>
     <input type="text" id="input1"> <br/>
     <input type="text" id="input2">
     <script type="text/javascript" src="prototype-1.6.0.2.js"> </script>
     <script type="text/javascript">
          Event.observe("input1", "keydown", function(e) {
            e = e || window.event;
              if(e.keyCode == 9) {
                 if(Event.element(e).value != "pass") {
                    Event.stop(e);
                    alert("You must enter 'pass!'");
                 }
              }
          });
    </script>
</body>
```

以跨浏览器方式接受键盘和鼠标输入

Prototype 在 Event 对象上提供了几个辅助方法，还定义了一些代表键盘编码的常量，从而使键盘和鼠标事件的处理变得更加简单。

isLeftClick 方法

这个方法可用来判断用户是否单击鼠标按键，从而抵消用户配置系统的不同之处(例如“左撇子”用户会把鼠标的左右按键互换)。

pointerX 或 pointerY 方法

这个方法返回在事件发生时鼠标指针的水平位置或垂直位置的绝对值。重点是要记住，这个位置是相对于页面而不是视口。如果将该页面向下滚动一点，那么在视口中的同一位置发生该事件时会返回不同的坐标。

键盘常量

下面这些常量均在 Event 名称空间中定义。

- KEY_BACKSPACE
- KEY_TAB
- KEY_RETURN
- KEY_ESC
- KEY_LEFT

- KEY_DOWN
- KEY_RIGHT
- KEY_UP
- KEY_DELETE
- KEY_HOME
- KEY_PAGEUP
- KEY_PAGEDOWN

2.3　触发调度事件

现代浏览器提供了两种方法来按照一定时间间隔执行某个函数：

- window.setInterval
- window.clearInterval

Prototype 提供了名为 PeriodicalExecuter 的 setInterval 方法增强版本，它可以确保在同一时刻只运行一个回调函数。TimedObserver 抽象类为 Form.Observer 和 Form.Element.Observer 类提供了基类，这两个类分别用来在表单中包含的元素发生变化时发出通知，从而可以针对这些变化做出响应。

PeriodicalExecuter

这个方法与 window.setInterval 方法非常类似，区别在于该方法会进行检查以防止出现对该回调函数的多次并发调用。PeriodicalExecuter 的用法如下：创建该类的一个新实例，传入回调函数和时间间隔(单位为秒)。

```
var pe = new PeriodicalExecuter(callback function, interval);
```

让 PeriodicalExecuter 停止运行的唯一方式就是重新加载页面或者在 PeriodicalExecuter 对象上调用 stop()方法。

```
<script type="text/javascript">
          Event.observe(window, "load", function(e) {
              var pe = new PeriodicalExecuter(function(e) {
                   if(confirm("Have you filled out your TPS reports?")) {
                        pe.stop();
                   }
              }, 5);
     });
</script>
```

1. TimedObserver

TimedObserver 是一个抽象类，它是下面两个 Form 辅助类的基类：Form.Observer 和 Form.Element.Observer。

Form.Observer

Form.Observer 类检查指定表单上的所有元素并调用回调参数指定的函数。当希望弄清楚自从上次

执行该回调函数以来表单上的一组元素是否发生改变时，就可以使用这个类。该类的用法如下：创建 Form.Observer 类的一个新的实例，并传入表单元素或 ID、时间间隔(单位为秒，将以该时间间隔检查表单)以及回调函数。

```
new Form.Observer(element, interval, callback);
```

下面建立了一个表单，它只有一个文本输入控件，您希望当用户在这个文本框中输入不同的内容时能够得到通知。其运行结果如图 2-3 所示。

```
<body>
    <form id="myForm" action="#">
        <p id="msg" class="message"> Current message: </p>
        <div>
            <label for="message"> message </label>
            <input id="message" type="text" name="message" value="Hello world!" />
        </div>
    </form>
    <script type="text/javascript">
        new Form.Observer('myForm', 0.3, function(form, value){
            $('msg').update('form changed to ' + value).style.color = 'blue'
            form.down().setStyle({ background:'lime', borderColor:'red' })
        })
    </script>
</body>
```

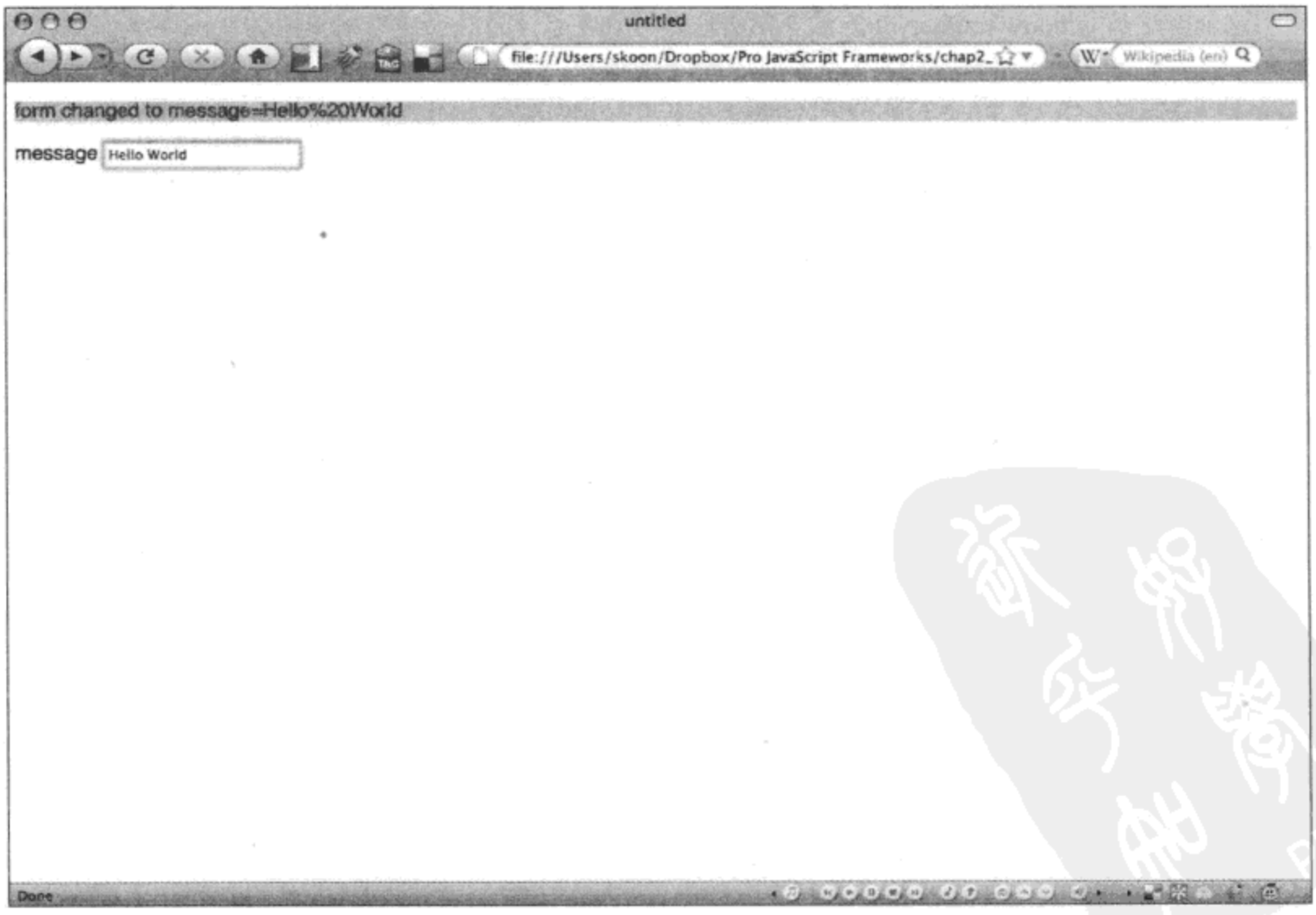


图　2-3

Form.Element.Observer

有时我们希望在表单中的某一个元素发生变化时得到通知。Form.Element.Observer 类就是一个极佳的选择。下面代码的运行结果如图 2-4 所示。

```
<body>
    <form id="myForm" action="#">
        <p id="msg" class="message"> Current message: </p>
        <div>
            <label for="message"> message </label>
            <input id="message" type="text" name="message" value="Hello world!" />
            <label for="selection"> Make a selection </label>
            <select id="selection">
                <option> first </option>
                <option> second </option>
            </select>
        </div>
    </form>
    <script type="text/javascript">
    new Form.Element.Observer('selection', 0.3, function(form, value){
        $('msg').update('form changed to ' + value).style.color = 'blue'
        form.down().setStyle({ background:'lime', borderColor:'red' })
    })
  </script>
</body>
```

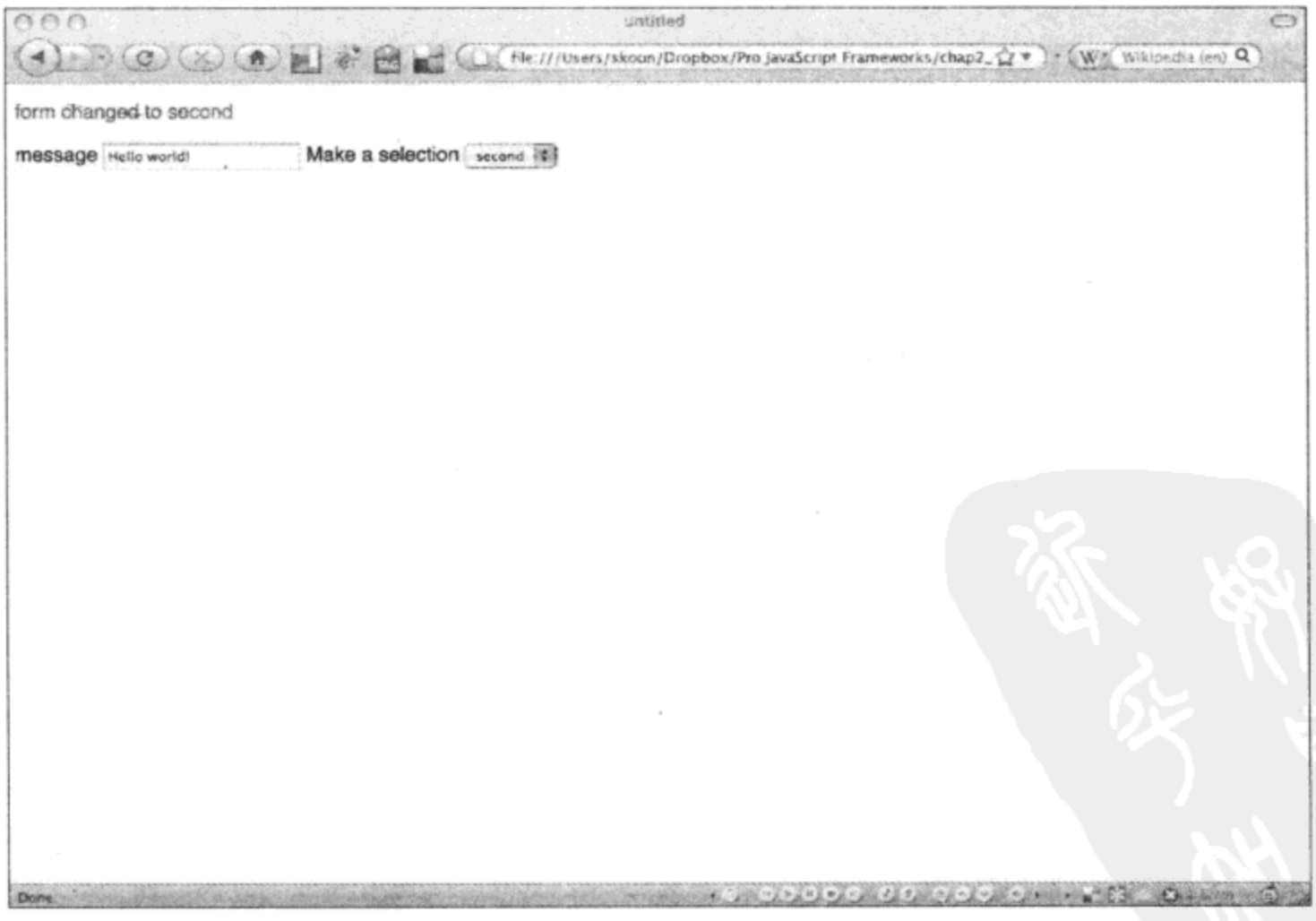


图　2-4

2.4　本章小结

在不同的浏览器中连接和处理事件是一项棘手的任务，必须将代码进行分支处理以设置事件处理程序，检查对于相同属性具有不同名称的不同对象。Prototype 简化了该操作，方式是提供了一个中心对象，不仅能够以跨浏览器的方式连接事件，而且可以调用它来获取有关触发该事件处理程序的元素的信息。

第 3 章

简化 AJAX 和动态数据

1999 年，Microsoft 发布了 Internet Explorer 5 浏览器，其中包含了名为 XMLHTTP 的新类型的浏览器对象。这个新的对象可以让 Web 开发人员从 Web 服务器中请求数据，而无须重新加载整个页面。Microsoft 将 XMLHTTP 请求的使用整合到它们的 Outlook Web Access 产品中。3 年后，Mozilla 项目发布了支持 XMLHttpRequest 的 Mozilla 1.0。开发人员开始使用新的 XMLHttpRequest 对象和旧的 XMLHTTP 对象，让自己的数据在与用户的交互中响应更加灵敏。2005 年，Jesse James Garrett 提出新的单词 AJAX(它是 Asynchronous JavaScript and XML 的简写，表示“异步 JavaScript 和 XML”)，用来描述 Web 开发人员正在使用的一组技术和交互。AJAX 工具是用来构建界面更加丰富、响应更加灵敏的 Web 应用程序的强大工具。为了迎合 AJAX 技术的使用和流行，越来越多的 JavaScript 库和框架开始创建包装器来包装这两种不同的 XMLHttpRequest 对象，从而让开发人员能够无缝地使用它们。Prototype 不仅能够以跨浏览器方式从服务器处请求数据，而且提供了几种不同风格的 AJAX 请求以满足当前 Web 开发人员的需求。

本章内容简介：

- 建立到服务器的请求
- 建立响应器来更新网页
- 建立周期性的更新器来轮询服务器以获取新数据

3.1 建立到服务器的请求

在使用 Prototype 进行 AJAX 调用时将要用到的所有对象都包含在 Ajax 对象中。Prototype 的 AJAX 的主要工作由 Ajax.Request 方法完成。创建一个新的 Ajax.Request 对象，并且传入一个 URL 和一个选项实参。下面是一个基本示例：

```
//Ajax.Request(url [,options]);

var url = "http://myserver/api/get";

var ajaxCall = new Ajax.Request(url, {
        method: "get",
        onSuccess: function() { alert("success") },
        onFailure: function() { alert("Failed") },
```

```
            onComplete: function() { alert("Complete") }
});
```

这里有几点值得注意。首先，可以使用 HTTP 动词指定请求方法。HTTP POST 和 HTTP GET 是最为常用的两个 HTTP 动词。其次，这里创建了两个函数，分别处理该请求的成功和失败。XMLHttpRequest 和 XMLHTTP 均提供了当请求完成时设置回调的方法，但是必须手动检查响应代码以确定该请求是否成功。Prototype 查看响应代码并根据一些简单的条件来调用在 onSuccess 或 onFailure 中指定的回调。如果请求状态是未定义或者属于 200 系列，那么调用 onSuccess 回调；否则调用 onFailure 回调。

在上面的示例中，事件发生的顺序如下：

(1) 创建请求。

(2) 初始化请求。

(3) 发送请求。

(4) 接收响应。该操作可能多次发生。

(5) 如果状态代码为 2xx，就调用 onSuccess，否则调用 onFailure。

(6) 完成请求。

3.1.1 Ajax.Request

Prototype 的 Ajax 对象支持多个用于构造请求的选项(参见表 3-1)。在创建新的 Ajax.Request 对象时将传入这些选项。大多数时候，默认值就是正确的值，但是如果需要设置这些选项，那么最好了解它们的作用。

表 3-1

选　项	说　明
asynchronous	告诉 Prototype 是否以异步方式建立请求。同步请求常常具有一些通常不便于处理的不必要的副作用。默认为"true"
contentType	指定该请求的内容类型。默认为"application/x-www-form-urlencoded"
encoding	指定该请求的编码。默认为 UTF-8
method	指定该请求要使用的 HTTP 方法。默认为"post"
parameters	指定包含在该请求中的参数。如果将请求的 HTTP 方法设置为"get"，那么这些参数将经过编码并附加到 url 中
postBody	如果该请求的 HTTP 方法设置为"post"，那么这个参数指定该请求的正文内容。如果没有设置 postBody 选项，就会使用 parameters 选项
requestHeaders	代表要随该请求发送的报头，可以是一个对象("{header: value}")或一个数组(偶数索引保存报头名称，而奇数索引保存值)。Prototype 提供了 4 种默认报头，可以重写它们来满足自己的需求： • X-Requested-With　设置为"XMLHttpRequest"。 • X-Prototype-Version　设置为正在使用的 Prototype 库的版本号。 • Accept　设置为"text/javascript, text/html, application/xml, text/xml, */*"。 • Content-type　设置为由请求选项所定义的当前内容类型(content-type)和编码(encoding)

(续表)

选　项	说　明
evalJS	如果服务器响应 contentType 设置为以下的值之一，就计算 responseText： • application/ecmascript • application/javascript • application/x-ecmascript • application/x-javascript • text/ecmascript • text/javascript • text/x-ecmascript • text/x-javascript 默认为"true"。可以通过传递"force"来强制执行计算，或者通过传递"false"来取消计算
evalJSON	如果服务器响应的 contentType 设置为"application/json"，就会计算 responseText，而 Ajax.Request 对象的 responseJSON 属性将设置成该计算的结果。可以通过传递"force"来强制执行计算，或者通过传递"false"来取消计算
sanitizeJSON	在进行计算之前清理 responseText。对于本地请求，默认为"false"，否则为"true"

为了自动将 responseText 中包含的所有 JavaScript 代码发送到 eval()方法，该 AJAX 请求必须把内容类型设置为 JavaScript 相关的 content-type 报头之一并遵循同源策略(Same Origin Policy)。同源策略规定，请求必须来自于与响应该请求的服务器相同的域名、协议和端口，这样才能够使用数据。

3.1.2　回调

Prototype 为请求循环中的不同阶段定义了许多回调函数(参见表 3-2)。所有这些回调都有两个参数。第一个参数是用于该请求的 XMLHttpRequest 对象。第二个参数是计算服务器响应(如果 content-type 设置为 JSON 类型)得出的结果。通过在传递给请求对象的选项中定义或指派一个函数，就可以利用这些回调。

表　3-2

回 调 名 称	说　明
onCreate	在请求对象初始化完毕但尚未调用任何 XHR 方法之前调用该回调
onComplete	当请求完成时调用该回调
onException	只要 XHR 对象抛出异常，就会调用该回调。将请求对象作为第一个实参，同时将该 XHR 对象抛出的异常对象作为第二个实参，传递给该回调
onFailure	如果响应的状态代码不是 2xx，就会调用该回调
onInteractive	当接收到部分响应(该响应分为多部分发送)时调用该回调。不保证一定会调用
onLoaded	当建立好请求且连接已经打开，但是没有向服务器发出调用时，调用该回调。不保证一定会调用

(续表)

回调名称	说　明
onLoading	当建立好请求且连接已经打开但是尚未准备好向服务器发送请求时调用该回调。不保证一定会调用
onSuccess	如果响应的状态代码属于 2xx 系列，就会调用该回调
onUninitialized	在创建 XHR 对象之后立即调用该回调
on{status code}	这些代表特定 HTTP 状态代码的回调，例如 on200。如果为某个状态代码指定了一个回调，就不会调用 onSuccess 和 onFailure。在调用 onComplete 回调之前调用这些回调

3.1.3　Ajax.Response

Ajax.Response 是传递给所有 Ajax.Request 回调的第一个参数，它基本上是浏览器特有的 XMLHttpRequest 对象的包装器。该对象的属性如表 3-3 所示。

表　3-3

属　性	说　明
status	服务器返回的 HTTP 状态代码
statusText	服务器返回的 HTTP 状态文本
readyState	该请求的当前状态： • 0—"Uninitialized"(未初始化) • 1—"Loading"(加载中) • 2—"Loaded"(已加载) • 3—"Interactive"(交互中) • 4—"Complete"(完成)
responseText、responseXML、responseJSON	带有指定格式的 HTTP 响应的正文
headerJSON	X-JSON 报头(如果有的话)的内容
request	该请求所使用的 Ajax.Request 或 Ajax.Updater
transport	该请求所使用的原生 XMLHttpRequest 或 XMLHTTP 对象

1. Ajax.Response 对象的方法

Prototype 为下面这两种原生 XMLHttpRequest 方法提供了新的实现：getResponseHeader 和 getAllResponseHeaders。主要的差别在于，如果指定的报头不存在，那么 Prototype 的方法将不会抛出错误，相反它们会返回 null。您应该使用 Prototype 的方法，而不是使用原生 XMLHttpRequest 方法的包装器：

- getHeader(name)　返回请求的报头。如果 name 指定的报头不存在，那么这个方法返回 null。
- getAllHeaders()　返回一个包含所有报头(由换行符隔开)的字符串。

- **getResponseHeader(name)**　这是原生 XMLHttp getResponseHeader 方法的包装器，它将返回 name 指定的响应报头(如果有的话)。
- **getAllResponseHeaders()**　返回一个包含所有报头(由换行符隔开)的字符串。这是另一个包装的方法。

3.2　以全局方式响应数据变化

现在，我们可以利用 Prototype 执行 AJAX 调用并对服务器返回的响应做出反应。不管具体执行的是什么 AJAX 请求，我们都希望发生某些事情，例如在覆盖图中显示一个旋转的图形来告诉用户后台正在执行某项任务。Prototype 已经定义了一个储存库来存放通用的 Ajax.Request 事件响应器。

Ajax.Responders

Ajax.Responders 是您所定义的全局回调的集合。为了建立一个全局回调，需要定义一个对象，并使用与希望注册的回调相同的名称来定义函数。

```
Ajax.Responders.register({
onCreate: function() {
//do some stuff
}
});
```

注意：

如果定义了一个全局回调，那么对于所有已经定义了该回调的 AJAX 请求，都不会调用这个回调。

必须使用曾经用来注册响应器的相同对象注销该响应器。因此，如果认为自己希望在稍后注销某个响应器，那么一定要在某个地方存放一个指向该响应器的引用，并将其传递给注销方法。

```
var globalResponders = {
        onComplete: function() { //hide the "Loading … div },
        onCreate : function() { //show the "Loading … " div. }
};
Ajax.Responders.register(globalResponders);
// code happens here
Ajax.Responders.unregister(globalResponders);
```

3.3　动态更新页面

本节描述如何动态更新页面。

3.3.1 Ajax.Updater

Ajax.Updater 对象是 Ajax.Request 对象的一种特殊化形式。将容器元素作为第一个参数传入，Ajax.Updater 对象将把请求的 responseText 的内容放在指定的容器中。其余的实参与 Ajax.Request 对象相同，但是它还包括两个传递给构造函数的新选项：evalScripts 和 insertion。

```
<div id="myDiv"> Remove the whitespace </div>
     <script type="text/javascript" src="prototype-1.6.0.2.js"> </script>
     <script type="text/javascript">
        var updater = new Ajax.Updater('myDiv', '/getItems'{
                method: "get",
                onSuccess: function() { alert("success") },
                onFailure: function() { alert("Failed") },
                onComplete: function() { alert("Complete") },
                insertion: 'top'
     });
</script>
```

1. evalScripts

evalScripts 选项可用来指定是否将 responseText 中的<script>元素传给 JavaScript 函数 eval()。这并不意味着把这些<script>元素添加到页面的 DOM 中，而只意味着<script>标记中包含的 JavaScript 代码将得以执行。这会带来几个副作用：

- 待运行代码的执行作用域是 Prototype 的内部处理函数。
- 如果想要在页面的其他部分访问某些函数，就必须使用下面的语法定义这些函数。如果以普通方式定义这些函数，那么一旦 Prototype 停止处理响应，它们就会丢失。

```
Function Myfunction() {}; // This won't work.

Myfunction = function() {};//This will allow other code on your page to call
Myfunction after the response has ended.
```

2. insertion

通常，使用 Element.update 函数把新内容放入 Ajax.Updater 初始化程序中指定的容器内。如果将 insertion 作为选项传入，那么可以使用标准字符串('top'、'bottom'、'before'和'after')告诉 Prototype 将内容插入到哪里。

如果请求返回的是 404 错误，那么会发生什么情况呢？我们并不希望将 404 HTML 插入到容器中，这会使其他人认为我们不够专业。Ajax.Updater 对象支持另一种可以附带一个对象的构造函数，这个对象定义了在调用成功或调用失败的情况下分别要更新的元素。

```
<body>
     <div id="myDiv"> Remove the whitespace </div>
     <div id="errorDiv"> </div>
     <script type="text/javascript" src="prototype-1.6.0.2.js"> </script>
     <script type="text/javascript">
          var updater = new Ajax.Updater(
               {
```

```
                    success: "myDiv",
                    failure: "errorDiv"
                },
            '/getItems'{
                method: "get",
                onSuccess: function() { alert("success") },
                onFailure: function() { alert("Failed") },
                onComplete: function() { alert("Complete") },
                insertion: 'top'
        });
    </script>
</body>
```

3.3.2 Ajax.PeriodicalUpdater

Ajax.PeriodicalUpdater 对象按照在请求的选项中指定的时间间隔定期地执行请求，它并不是 Ajax.Request 或 Ajax.Updater 的特殊化形式：

- frequency　默认为“2”。这是每次请求之间的时间间隔，单位为秒。
- decay　默认为“1”。只要请求的 responseText 与前一次请求相比没有发生变化，就把当前时间间隔乘以这个衰减系数。可以使用这个选项根据数据变化的频率来改变请求的次数。如果认为数据不会非常频繁地发生改变，那么可以将其设置成一个较高的数值。一旦响应文本发生变化，衰减系数就会重置为 1。
- Ajax.PeriodicalUpdater　如果希望轮询服务器来获取新数据，但是希望限制应用程序向服务器发出的请求次数(如果数据变化不是特别频繁)，那么这个对象就非常有用。

```
<body>
    <div id="myDiv"> Remove the whitespace </div>
    <script type="text/javascript" src="prototype-1.6.0.2.js"> </script>
    <script type="text/javascript">
        var periodicalUpdater = new Ajax.PeriodicalUpdater('myDiv', '/getItems'
        {
            method: 'get'
            frequency: 3,
            decay: 2
        });
    </script>
</body>
```

在某些情况下，我们可能希望在达到特定频率之后或者请求失败之后完全停止请求。Ajax.PeriodicalUpdater 对象提供了名为 start()和 stop()的两个方法来完成这些工作。

```
<body>
    <div id="myDiv"> Remove the whitespace </div>
    <script type="text/javascript" src="prototype-1.6.0.2.js"> </script>
    <script type="text/javascript">
        var periodicalUpdater = new Ajax.PeriodicalUpdater('myDiv', '/getItems'
        {
            method: 'get'
```

```
            frequency: 3,
            decay: 2,
            onComplete: function() {
                  if(periodicalUpdater.frequency> 48) { periodicalUpdater.stop(); }
            },
            onFailure: function() { periodicalUpdater.stop();}
      });
    </script>
</body>
```

3.4　本章小结

Prototype 是第一个用于进行跨浏览器 AJAX 调用且用起来比较顺手的 JavaScript 库。它能够自动解释服务器返回的 JavaScript 和 JSON 响应内容。它使得我们能够更加容易地定期从服务器处请求数据以更新用户界面，并可以控制向服务器发送请求的频率。

第 4 章

处 理 表 单

使用表单从用户处收集数据，这是每个 Web 开发人员工作的一部分。处理表单是一件有些费时且麻烦的工作。Prototype 提供了一些方法来快速获取表单元素的值，从而使得表单和表单元素的处理变得更加简单。您还可以检查表单元素是否有值，以及把输入焦点设置为任何所需的表单元素。此外，利用 Prototype 可以方便地使用 AJAX 调用序列化表单数据并将其提交到服务器。

本章内容简介：

- 操作表单元素和数据
- 验证表单数据
- 使用 AJAX 提交表单

4.1 操作表单元素和数据

在诸如 Prototype 之类的现代 JavaScript 库得到广泛应用之前，我们必须花费大量时间编写代码以查看 document.formName 对象并弄清楚表单中有什么元素，然后才可以遍历表单元素并对它们进行验证。有了 Prototype 之后，就可以按照与处理其他 HTML 元素相同的方式来扩展表单元素，但是可以使用一些特有的方法。使用$F()方法，可以快速获取页面上任何表单元素的值。Form.Elements 对象包含一些方法，Prototype 使用这些方法扩展表单元素。可以很容易地把所有表单元素序列化成一个适合于传入 AJAX 调用的对象字面量或字符串。

4.1.1 Form 对象

Form 对象包含 Prototype 提供用来处理 HTML 表单元素的所有方法。与 Element.Methods 对象非常类似，Prototype 使用 Form.Elements 对象中的方法扩展页面中的表单元素。

1. disable 和 enable 方法

这两个方法遍历表单中的元素，然后将其启用或禁用。需要重点注意的是，序列化方法不会序列化禁用的表单元素。

```
$('myForm').enable();
$('myform').disable();
```

2. findFirstElement 方法

这个方法查找第一个非隐藏的、非禁用的表单元素，该元素属于 input、select 或 textarea 元素之一。当该方法返回元素时会扩展该元素。元素出现在文档中的顺序(不是 Tab 键顺序)决定了该元素是否被视为第一个元素。

```
var firstElement = $('myForm').findFirstElement();
```

3. focusFirstElement 方法

这是一个聚合方法，可用来快捷地将输入焦点置于表单中的第一个元素。它调用 findFirstElement()方法，然后在返回的元素上调用 activate()方法。这样，用户就可以立刻开始输入数据，而不必先单击某个元素，这是一种改进 Web 表单易用性的优秀方法。

```
$('myForm').focusFirstElement();
```

4. reset 方法

该方法将表单中的所有元素恢复到各自的默认值。

```
$('myForm').reset();
```

5. getInputs 方法

这个方法返回一个由表单中包含的所有输入控件组成的数组。可以传入一个类型和名称来限制返回的输入控件元素。

```
var inputs = $('myForm').getInputs('text','firstName');
//returns any text inputs named "firstName"
```

6. getElements 方法

这个方法返回表单中的所有元素。它不会返回 OPTION 元素，而是只返回它们的父元素 SELECT。

```
var inputsArray = $('myForm').getElements();
```

7. serialize 方法

serialize 方法是 Form 对象的主要方法。它获取表单中所有已启用元素的值，然后返回一个适合于附加到 GET 或 POST 请求的 URL 中的键-值对字符串。它接受一个可选的 Boolean 型参数

getHash，此时返回一个用来表示表单元素和值的对象。以下面的表单为例：

```
<form id="myForm" name="myForm" action="self" method="get">
        <div> Name: <input type="text" id="name" name="name" /> </div>
        <div>
            <select id="gender" name="gender">
               <option value=""> -=Select a gender </option>
               <option value="M"> Male </option>
               <option value="F"> Female </option>
            </select> <br/>
         <input type="submit" />
</form>
```

在这个表单对象上调用$('myForm').serialize()和$('myForm').serialize(true)方法，将返回以下值：

```
name=Scott & gender=M //non hashed value
{name:Scott, gender:M} //hash value
```

8. serializeElements 方法

这个方法可用来通过名称指定在序列化表单时希望包括哪些元素。如果希望获取特定表单元素的值并单独提交这些值，例如有两个 select 元素，其中一个元素中的选项依赖于另一个元素中选择的选项，这个方法就比较有用。

```
var formHash = Form.serializeElements([$('name')],true);
```

4.1.2 结合使用 Form 对象的方法

现在，我们将构建一个简单的表单，让用户输入他的姓名和性别。我们希望当页面加载完毕时把输入焦点设置到姓名字段，而一旦用户单击 Next 按钮就把输入结果序列化以提交给服务器。下面就是这个 HTML 表单的外观(参见图 4-1 和图 4-2)。

```
<form name="emptyform"> <!--this is just a placeholder form so the element will
appear in FF-->
        <div> <input type="checkbox" id="chkUseForm" /> I want to enter my
        information </div>
    </form>
    <form id="myForm" name="myForm" action="" method="get">
        <div> Name: <input type="text" id="name" name="name" /> </div>
        <div>
            <select id="gender" name="gender">
                  <option value=""> -=Select a gender </option>
                  <option value="M"> Male </option>
                  <option value="F"> Female </option>
            </select> <br/>
            <input type="button" id="submitButton" value="submit" />
            <input type="button" id="resetButton" value="reset" />
</form>
```

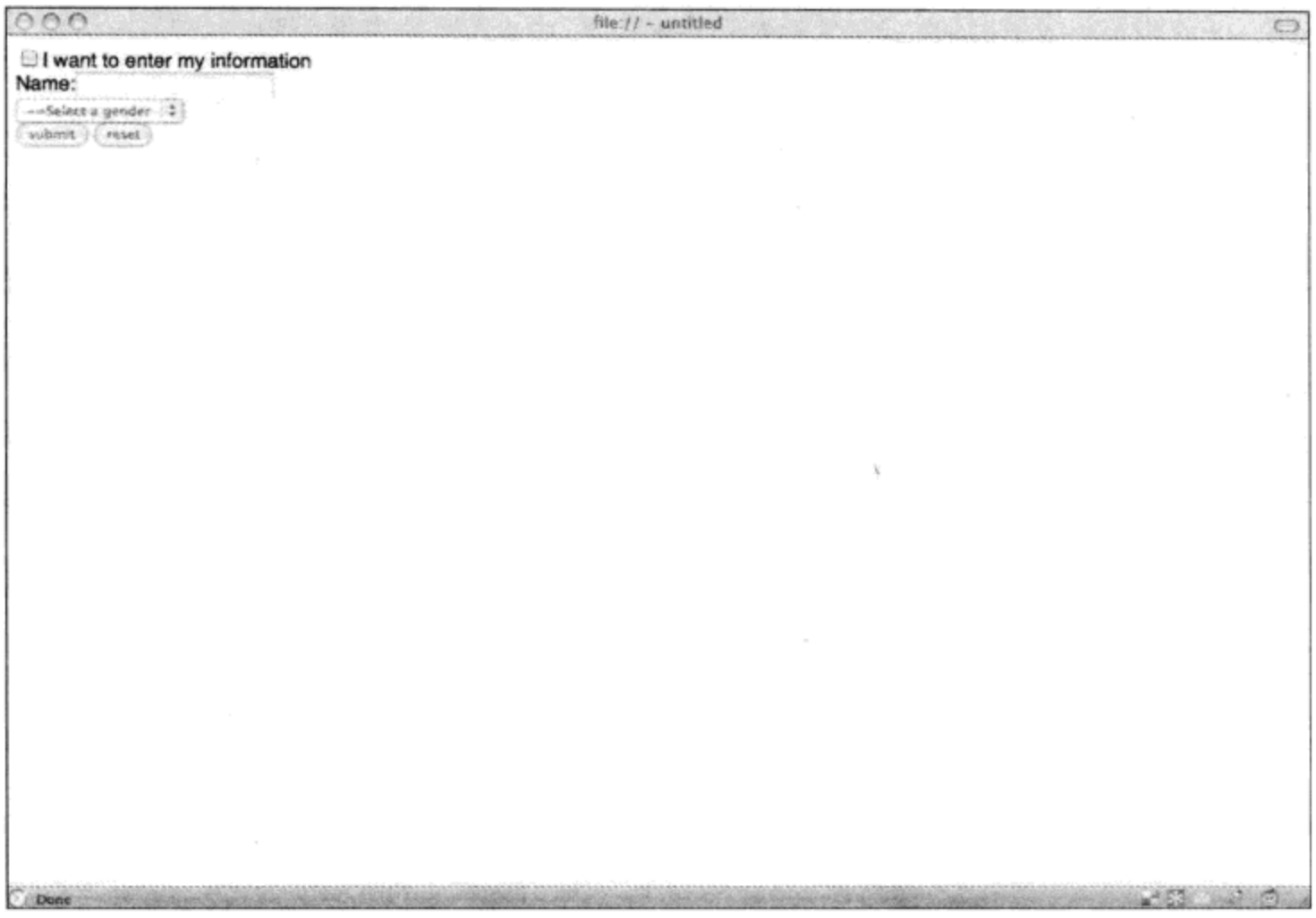

file:// - untitled
I want to enter my information
Name:
--Select a gender
submit
reset
Done

图　4-1


file:// - untitled
I want to enter my information
Name:
--Select a gender
submit
reset
Done

图　4-2

因为在这里由用户控制如何将表单提交给服务器，所以我们使用了一个常规的 input type ="button"按钮而不是提交按钮。这里还给出了另一个按钮，用来将表单元素重置为它们的起始值。此外，这里还向页面中添加了一个复选框来启用或禁用表单，从而模拟该表单是可选的。这个表单初始时是禁用状态。如果用户选中了该复选框，这个表单就会变成启用状态。当启用该表单之后，就把输入焦点设置到姓名字段。下面是实现该行为的代码。

```
<script type='text/javascript'>
Event.observe(window, "load", function(e) {
     $('myForm').disable();

});

Event.observe("submitButton", "click", function(e) {
       alert($('myForm').serialize());

});
Event.observe("chkUseForm", "click", function(e) {
     e.target.checked? function() {$('myForm').enable();$('myForm')
.focusFirstElement();}() :$('myForm').disable();
});

Event.observe("resetButton", "click", function(e) {
    $('myForm').reset();
});
</script>
```

虽然这里的代码大部分非常易于理解，但是应该查看两个事件处理程序，它们执行了一些比较有趣的操作，如下面的示例所示：

```
Event.observe("chkUseForm", "click", function(e) {
       e.target.checked? function() {$('myForm').enable();$('myForm')
.focusFirstElement();}() :$('myForm').disable();
});
```

这里的代码的作用就是处理复选框的单击事件。必须将这个复选框包装在 form 标记中，这是因为如果把某些表单元素(如 input 和 select)放在 form 标记之外，那么有些浏览器就不会识别这些表单元素。因此，处理该事件，并检查这个复选框的 checked 属性。如果没有选中该复选框，就禁用该表单。如果已经选中该复选框，就要做两件事情：

- 启用该表单。
- 将输入焦点设置到该表单的第一个元素，在这里就是姓名字段。

由于这个三元运算符的每一条语句只能执行一项任务，因此我们把希望调用的两个方法包装到一个匿名的自执行函数中。也可以将这段代码重写成如下所示：

```
Event.observe("chkUseForm", "click", function(e) {
    function enableForm()
    {
         $('myForm').enable();
```

```
        $('myForm').focusFirstElement();
    };
    e.target.checked? enableForm() :$('myForm').disable();
});
```

还应该查看的另一个函数是 Submit 按钮的事件处理程序：

```
Event.observe("submitButton", "click", function(e) {
     alert($('myForm').serialize());

});
```

这个方法将完成表单代码的主要工作。现在它只是将表单元素序列化，并在一个 JavaScript 警告框中将这些元素的值显示出来，这样就能够查看它们。

4.2 验证表单数据

我们已经完成数据收集表单。用户可以使用复选框启用和禁用该表单，而当用户单击 Submit 按钮时，可以将该表单序列化。现在，我们希望确保用户在姓名字段中输入文本，但是他们不必选择性别。Prototype 为单个表单元素的处理提供了几个方法，并使用$()或$F()快捷方式扩展指定的表单元素。

Form.Elements 对象

Form.Elements 对象包含 Prototype 用来扩展 input、select 和 textarea 元素的所有方法。这些方法也有 Field 对象的别名。因此，在任何使用 Form.Elements 对象的地方都可以改用 Field 快捷方式。

1. activate 方法

这个方法将输入焦点设置到指定的元素，并选中该元素中的文本(如果它是文本输入控件的话)。

```
$('name').activate()
```

2. clear 方法

这个方法将指定元素的内容清空。

```
$('name').clear()
```

3. disable 和 enable 方法

这两个方法的工作方式与 Form.disable 和 Form.enable 方法类似，但是它们作用于单个表单元素。注意，在某个已经启用或禁用的元素上调用 enable 或 disable 方法并不会抛出异常。

```
$('name').disable()
$('name').enable()
```

4. focus 方法

顾名思义，这个方法的功能就是把输入焦点设置到指定元素。

```
$('name').focus()
```

5. getValue 方法

这个方法返回指定元素的值。它也有一个别名，即全局快捷方式$F()。在大多数情况中，这个方法会返回一个文本值。如果指定元素是一个多项选择框，那么这个方法将返回一个值数组。还可以使用快捷方式$F()方法快速地获取任何表单元素的值。

```
$('name').getValue()
$F('name');
```

6. present 方法

如果指定元素中包含任何内容，那么这个方法返回 true，否则返回 false。

```
$('name').present()
```

7. select 方法

这个方法选中指定元素中的文本(如果有的话)。

```
$('name').select()
```

8. serialize 方法

这个方法与 Form.serialize 方法的工作方式类似，但是它只序列化单个元素。

```
$('name').serialize()
```

9. 验证表单

下面再次给出这个表单的代码：

```
<form name="emptyform"> <!--this is just a placeholder form so the element will
appear in FF-->
          <div> <input type="checkbox" id="chkUseForm" /> I want to enter my
          information </div>
    </form>
    <form id="myForm" name="myForm" action="" method="get">
       <div> Name: <input type="text" id="name" name="name" /> </div>
       <div>
          <select id="gender" name="gender">
               <option value=""> -=Select a gender </option>
               <option value="M"> Male </option>
               <option value="F"> Female </option>
```

```
            </select> <br/>
            <input type="button" id="submitButton" value="submit" />
            <input type="button" id="resetButton" value="reset" />
</form>
```

如果启用了表单，那么用户必须输入他的姓名。如果用户没有输入自己的姓名就单击 Submit 按钮，就会弹出一个 JavaScript 警告框，向用户显示一条消息以提示其输入自己的姓名，并将姓名字段的背景色改为红色。

```
<script type='text/javascript'>
Event.observe(window, "load", function(e) {
       $('myForm').disable();

});
```

```
Event.observe("submitButton", "click", function(e) {
     if(!$('name').present()) {
        alert("You must enter your name");
        $('name').setStyle( {backgroundColor : '#F34851'} );
        return;
     };
     alert($('myForm').serialize());

});
```

```
Event.observe("chkUseForm", "click", function(e) {
     function enableForm()
     {
            $('myForm').enable();
            $('myForm').focusFirstElement();
     };
     e.target.checked? enableForm() :$('myForm').disable();
});

Event.observe("resetButton", "click", function(e) {
     $('myForm').reset();
});
</script>
```

在图 4-3 中可以看到，我们已经将 Submit 按钮的单击事件处理程序稍微完善了一些。我们添加了对 present()方法的调用，这将检查 Name 文本框，并确保用户已经输入了一些文本。如果用户没有输入任何文本，就会显示一个 JavaScript 警告框，然后使用 setStyle 方法将 Name 文本框的背景色改为淡红色。

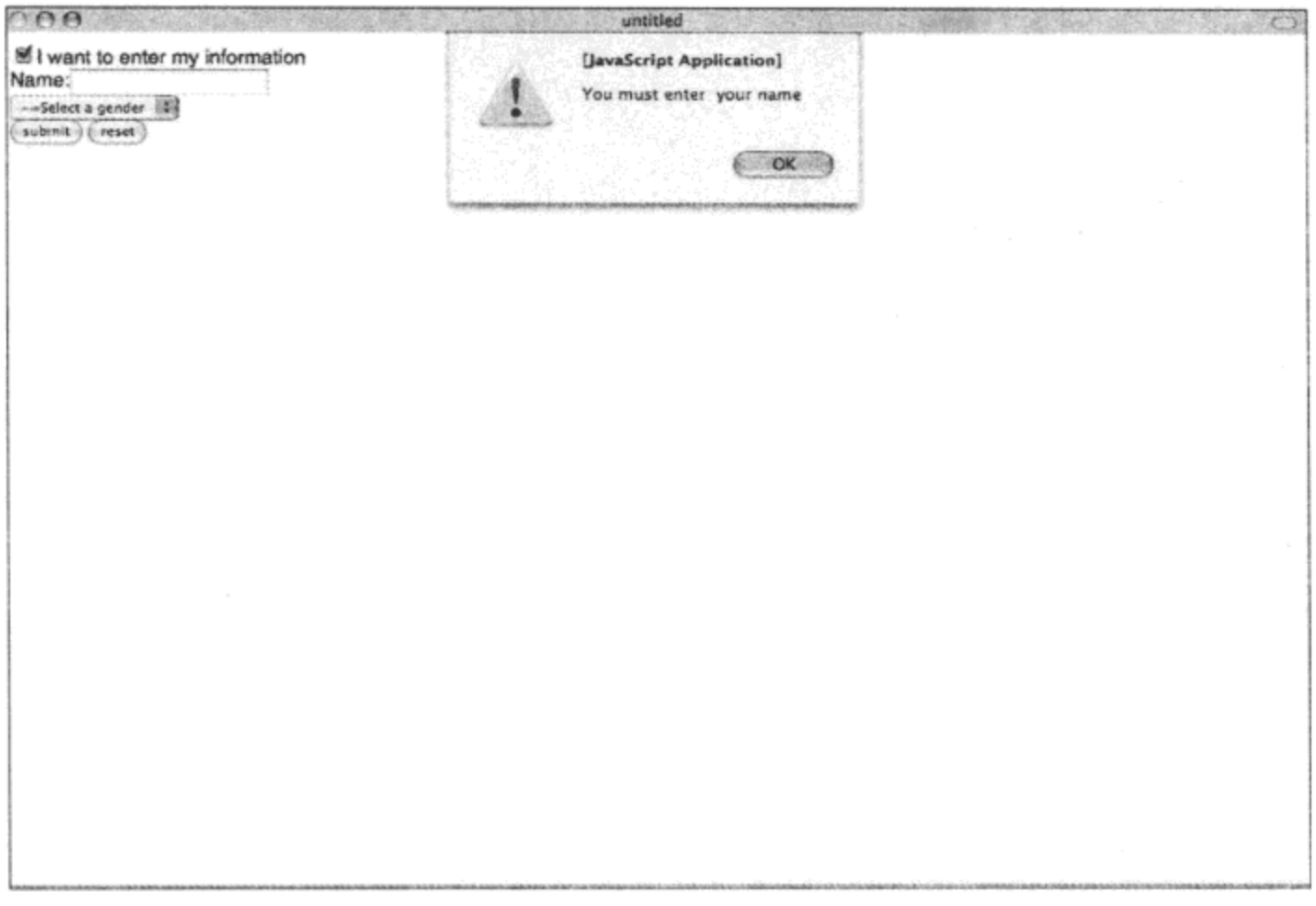


图 4-3

4.3 使用 AJAX 提交表单

到目前为止，我们已经验证了用户输入的数据，并且已经准备好将结果发送给服务器。可以使用 Prototype 的 Ajax.Request 对象所带的序列化方法，但是 Form 对象提供了一个名为 request 的方法，该方法将完成我们所需的所有工作。我们将修改 Submit 按钮的单击事件处理程序，以使用 Form.request 方法将数据发送回服务器。

```
Event.observe("submitButton", "click", function(e) {
     if(!$('name').present()) {
       alert("You must enter your name");
       $('name').setStyle( {backgroundColor : '#F34851'} );
       return;
     };
    $('myForm').request(
        {
            onSuccess: function(){alert("form successfully posted")},
            onFailure: function(){alert("form failed to post")}
        });
});
```

Form.request 方法所带的选项实参与 Ajax.Request 对象完全一样。Form.request 方法根据表单元素声明中提供的方法来推断方法类型。如果在选项实参中传入一个方法属性，那么 Form.request 方法将使用这个指定的方法而不是表单元素方法。如果没有指定方法，而且不能从表单元素中推断出方法，那么这个方法将默认为 POST。如果传入的参数使用与表单元素相同的名称，那么该

参数将取代序列化后该表单元素的值。

4.4　本章小结

在这一章中，我们查看了 Prototype 提供用来处理表单和表单元素的一些方法。通过使用 Form 对象的方法，可以减少使用 AJAX 把表单提交给服务器时必须编写的代码量。此外，我们还了解了验证和操作表单元素的方式。

操作通用数据结构和函数

Prototype 最大的强项之一在于它扩展原生 JavaScript 对象以使其具有多种方法，从而让 JavaScript 更加易用。原生的 Object、Date 以及 Array 对象都得到扩展，从而具有许多有用的方法。Function 对象扩展可用来执行许多高级任务，例如利用自己编写的函数进行柯里化(currying)和绑定(binding)。

本章内容简介：

- 增强原生对象并使用 Class 对象向代码中引入对象层次结构
- 分析和操作字符串
- 使用模板动态构建用户界面
- 绑定和操作函数
- 引入 Enumerable 类
- 增强 JavaScript 数组并引入 Hash 类
- 处理数值和日期

5.1 增强原生对象并引入类

Prototype 最大的强项之一在于它扩展原生 JavaScript 对象以使其具有多个方法，从而使编写 JavaScript 代码变得更加容易。在使用 Prototype 时，程序员不会感觉自己一直在使用框架，而是感觉在编写 JavaScript 代码。原生 Object 对象获得克隆和扩展其他对象的能力。Prototype 包括几个全局函数，可用来确定对象的类型。Prototype 还包括一个 Class 对象，可用来方便地在自己的 JavaScript 代码中创建并维护一个对象层次结构。

5.1.1 对象扩展

Prototype 向 Object 对象中添加了一些方法，使我们在 JavaScript 代码中操作或复制对象变得更加简单。可以使用键集合将 Object 对象转换成 3 种传输格式(HTML、JSON、QueryString)中的任何一种，从而找出一个 Object 对象具有哪些属性。

1. clone 方法

这个方法用来创建指定对象的一个浅复制(shallow copy)副本。

```
Object.clone(o)
```

2. extend 方法

这个方法与 Element.extend 方法非常类似。它将对象的属性复制到一个新的对象中。

```
Object.extend(dest, src)
```

3. inspect 方法

这个方法查找指定对象中的检查方法。如果找到一个检查方法，就会运行这个方法；否则，就会在该对象上调用 toString 方法。可以定义一个检查方法，然后使用该方法提供关于这个对象的更详细或结构更好(相对于 toString 方法来说)的信息。

```
Object.inspect(o)
```

4. isArray、isElement、isFunction、isHash、isNumber、isString 和 isUndefined 方法

所有这些方法均用来确定指定的对象是否属于方法名所指定的类型。如果属于该类型，那么该方法返回 true；否则，它返回 false。

```
Object.isArray(o)
Object.isElement(o)
Object.isFunction(o)
Object.isHash(o)
Object.isNumber(o)
Object.isString(o)
Object.isUndefined(o)
```

5. keys 和 values 方法

keys 方法将对象看成一个 Prototype Hash 对象，它返回由该对象的属性名称构成的数组。values 方法返回该对象中每个属性的值。

```
Object.keys(o)
Object.values(o)
```

6. toHTML、toJSON 和 toQueryString 方法

这 3 个方法均将指定的对象转换成该方法指定的格式。当需要快速地把一个对象的值转换成某种特定格式时，使用这些方法就会非常方便。

```
var o = {firstName:"Douglas",
lastName:"Crockford",
toHTML: function(){ return " <p> #{firstName} #{lastName} </p> ".interpolate(this);}
};
Object.toHTML(o);
// <p> Douglas Crockford </p>
```

```
Object.toJSON(o);
// { "firstName" : "Douglas", "lastName":"Crockford" };
Object.toQueryString(o);
//firstName=Douglas & lastName=Crockford
```

5.1.2　Class 对象

JavaScript 中的继承通常涉及跟踪对象的原型链(prototype chain)。必须记住哪些对象从基对象继承而来。Prototype 中的 Class 对象使我们在 JavaScript 代码中创建和扩展对象变得更加简单。

1. create 方法

当我们希望创建一个新的类时，Class.create 方法可以自动完成所有的繁重工作。可以将一个对象作为参数传入，该对象用作新创建类的超类。下面是一个用来演示如何使用 Class.create 方法的简单示例。

```
var myParentClass = Class.create({ parentFunction: function() { return "parent";}});
        var myClass = Class.create(myParentClass,{ classFunction: function() {return
"class";}});
        var c = new myClass();
        alert(c.classFunction());
        alert(c.parentFunction());
```

在这里创建了两个类，一个类名为 myParentClass；另一个类名为 myClass。myClass 类从它的超类 myParentClass 那里继承了 parentFunction 成员。

2. 特殊属性

创建的每个类对象都含有两个特殊的属性，用以说明该类位于对象层次结构中的什么位置。superclass 属性说明这个类是从什么类继承而来的，而 subclasses 数组则包含了所有继承自这个类的对象。

addMethods 方法

addMethods 方法用于在创建类之后向其中添加新的方法。需要重点注意的是，可以在任何时候为类添加新方法，所有使用该类实例化的对象都可以使用它们。假设在前一个示例中向对象添加了两个新方法。在实例化该对象之后，向父类中添加了一个方法，然后又向该对象所属的类添加了一个方法。

```
    var myParentClass = Class.create({ parentFunction: function() { return "parent";}});
    var myClass = Class.create(myParentClass,{ classFunction: function() {return
"class";}});
    var c = new myClass();

    myParentClass.addMethods({ newParentMethod: function(){ return "new Method added to
    the parent after creating the class";}})
    myClass.addMethods({ newMethod: function(){ return "new Method added after creating
    the class";}})
```

```
alert(c.classFunction());
alert(c.parentFunction());
alert(c.newParentMethod());
alert(c.newMethod());
```

可以看到，即使这些方法是在创建 myClass 类的新实例之后才添加进去的，我们仍然能够访问它们。

5.2　修改和分析字符串

作为 Web 开发人员，我们将大量时间花费在字符串的处理上。有时我们希望对用户输入进行清理以防止跨站点脚本攻击，有时则需要一种更好的匹配字符串模式并返回结果的方式。Prototype 在 String 对象上构建并提供了几种用于修改和分析字符串的有用方法。

1. blank 和 empty 方法

如果字符串匹配方法名所指定的条件，那么这两个方法均返回一个 Boolean 值。需要重点注意这两个方法之间的差别。如果在空字符串("")上调用 blank 方法，那么该方法返回 true。但是如果字符串中含有一个或多个空格(即使该字符串可能看上去是空的)，那么 empty 方法并不会返回 true。

```
''.blank() // true
' '.blank() //true
' foo'.blank() //false
''.empty() //true
' '.empty() //false
```

2. camelize、capitalize、dasherize 和 underscore 方法

所有这些方法执行的都是类似的功能，但是每个方法都多少有些不同。所有方法均以现有字符串为参数并修改它们的格式。表 5-1 描述了每个方法。

表　5-1

方　法	说　明	代　码
camelize	以一个由连字符隔开的字符串为参数，将其转换成 camelCase(驼峰式大小写)等价形式	'first-name'.camelize() //firstName
capitalize	确保字符串的首字母大写，并将该字符串的剩余部分变成小写	'firstName'.capitalize() //Firstname
dasherize	将每个下划线字符"_"替换成连字符"-"	'first_name'.dasherize() //first-name
underscore	以一个 camelCase(驼峰式大小写)字符串为参数，使用下划线字符将每个单词隔开	'firstName'.underscore() //first_name

3. startsWith、endsWith 和 include 方法

这些方法完成预期的操作。如果字符串含有指定的子字符串，它们就返回 true 或 false。

```
'fool'.startsWith('f') // true
'fool'.endsWith('l') //true
'fool'.include('foo') //true
'fool'.include('food') //false
```

4. evalJSON 和 toJSON 方法

Prototype 在 String 对象中提供了两个用于处理 JSON 的方法：evalJSON 和 toJSON。evalJSON 方法试着分析指定字符串并返回一个 JSON 对象。如果该字符串并不是一个格式规范的 JSON 数据，就会抛出一个语法错误。它还带有一个名为 sanitize 的可选参数。如果传入 sanitize 参数，那么 evalJSON 方法将查找该字符串的已知漏洞。如果发现相应的漏洞，就不会调用 eval()方法。如果字符串中包含从远程源加载的数据，那么总是应该传入 sanitize 参数。toJSON 方法将返回一个经过 JSON 编码的字符串。

```
var person = '{"firstName" : "Douglas", "lastName":"Crockford"}'.evalJSON()
// person .firstName = "Douglas"
var JSONname = 'Johnny "The Killer" Bambam'.toJSON()
//JSONname = 'Johnny \"The Killer\" Bambam'
```

5. evalScripts、extractScripts 和 stripScripts 方法

这 3 个方法可用来对可能嵌入在待处理字符串中的脚本进行一定程度的控制。stripScripts 方法将字符串中找到的所有脚本块删除。可以使用这个方法清理用户输入的数据并阻止跨站点脚本攻击。可以使用 extractScripts 方法移除嵌入在字符串中的脚本，并将它们放到一个数组中以供后面使用。evalScripts 方法使用 extractScripts 方法将脚本从字符串中取出，然后在它找到的每个脚本上调用 eval 方法。在每个脚本上调用 eval 方法的结果就是该脚本作为一个数组元素返回。

```
"Given this string <script> alert('hiya'); </script> ".stripScripts()
// will return "Given this string "
"Given this string <script> alert('hiya'); </script> ".extractScripts()
//retuns ["alert('hiya');"]
"Given this string <script> alert('hiya'); </script> ".evalScripts()
// will display "hiya" in an alert box.
```

6. escapeHTML 和 unescapeHTML 方法

escapeHTML 方法将所有特殊字符转换成各自的 HTML 转义序列。unescapeHTML 方法剥离给定字符串中的所有 HTML 标记，并将转义的 HTML 字符转换成它们的正常形式。这两个方法都可用来在把用户输入的数据存储到数据库或 XML 文件中之前进行清理。

```
'1 <2'.escapeHTML()
//'1 &lt; 2'
' <p> 1 &lt; 2 </p> '.unescapeHTML()
//' 1 <2 '
```

7. gsub 和 sub 方法

sub 和 gsub 方法均在给定字符串中搜索子字符串并将其替换成指定的字符串。gsub 方法搜索整个字符串并替换找到的所有指定子字符串，而 sub 方法可以附带一个可选的 count 参数。如果没有传入 count 参数，那么 sub 方法只替换它找到的第一个指定子字符串。否则，它将替换找到的一定数量的子字符串，具体数量由 count 参数指定。

8. scan 方法

这个方法可用来遍历字符串中的每一个匹配项，并将每个匹配项传给一个函数。用来查找匹配项的模式可以是一个字符串，也可以是一个正则表达式。

```
var iCount = 0;
'Mississippi'.scan('i', function(match) { if(match == 'i') { iCount++; }; });
// iCount = 3;
```

5.3　生成模板化内容

模板是重用 HTML 并把数据与表示分离的一种优秀方式。Prototype 提供了一种易于使用的创建模板和绑定数据的机制。可以使用字符串定义模板。Prototype 使用一种简单的模式来确定将数据插入到模板中的何处。只需要使用文本“#{}”将属性名称包装起来即可。例如，如果希望将属性 firstName 插入到模板中，那么可以将文本“#{firstName}”放入到模板字符串中。

下面是一个简单的模板，它把数据使用 h1 标记包装起来。当希望 Prototype 插入数据时，调用 evaluate 方法并传入希望绑定的对象即可。

```
<body>
      <script type="text/javascript">
         var data = {value:'foo'};
         var template = new Template(" <h1> #{value} </h1> ");
         Event.observe(window, "load", function(e) {
              $("contentHolder").insert(template.evaluate(data));
         });

     </script>
     <div id="contentHolder"/>
</body>
```

Template.evaluate()方法

Template.evaluate 方法将指定的对象数据绑定到模板中并返回一个字符串。设计模板的目的就是在代码中进行重用。修改前一个示例，将模板绑定到一个数据对象数组中。可以在数据数组上使用 each 方法遍历这个数据对象数组，并且每次使用一个新的数据对象在模板上调用 evaluate 方法。如果需要构建较长的列表(参见图 5-1)，那么这就是一种可用的优秀技术。

```
<body
     <script type="text/javascript">
```

```
            var data = [{value:'foo'}, {value:'bar'}, {value:'baz'}];
            var template = new Template(" <h1> #{value} </h1> ");
            Event.observe(window, "load", function(e) {
                data.each(function(value) {
                    $("contentHolder").insert(template.evaluate(value));
                });
            });

        </script>
        <div id="contentHolder"/>
</body>
```

图　5-1

Template.evaluate 方法还接受第二个可选的参数，这个参数中含有一个正则表达式，用于定义想要使用的所有自定义数据模式语法。

5.4　绑定和操作函数

在 JavaScript 中，函数属于第一类对象(first-class object)。也就是说，可以像操作其他对象一样操作函数。在 JavaScript 中，函数附带的实参数量也是可变的，这意味着我们可以在任何时候重载任何函数。Prototype 提供了几个方法来扩展原生的 Function 对象，并为 JavaScript 编程提供了更大的灵活性。

5.4.1　绑定函数

在使用 JavaScript 编程时，想要弄清楚 this 关键字在不同时刻所表示的含义是一个稍微有点棘手的问题。根据调用函数的方式以及在什么上下文中调用该函数，this 关键字可以引用包装函数、函数自身或全局的 window 对象。Prototype 提供了一种绑定函数并定义 this 关键字应该引用什么对象的方式。

下面查看根据调用函数时所处上下文的不同，this 关键字引用的元素是如何发生变化的。

```
window.name = "Global window object";

function showName() {
       return this.name
};

alert(showName()); //shows "Global window object"

var namespace = {
     name : "namespace object",
     showName: function() { return this.name;}
};

alert(namespace.showName());//shows "namespace object"

window.namespaceShowName = namespace.showName;
alert(window.namespaceShowName());//shows "Global window object"
```

这里声明了一个名为 showName 的函数，它返回被调用时 this 关键字所引用的对象的 name 属性。使用与 showName 函数相同的代码在 namespace 对象中声明了一个函数。当调用 showName 函数时，它是在全局 window 对象的上下文中声明的，并显示在第一行代码中为 window 对象指派的名称。当调用 namespace.showName 函数时，它查看自己的当前执行作用域并了解到自己包含在 namespace 对象中，因此返回它在其中找到的 name 属性。最后几行代码是比较棘手的地方。我们将 namespace.showName 函数指派给全局 window 对象的新成员，然后使用新别名调用 namespace.showName 函数。该函数查看自己的执行作用域，并了解到自己是在全局 window 对象内部调用的，因此它返回的名称与全局 showName 函数一样。

1. bind 方法

bind 方法将指定的函数包装到另一个函数中，并将作用域锁定到第二个参数中指定的作用域。它返回绑定的函数(bound function)。该方法还接受一个可选参数，该参数中包含由要传入绑定函数的实参构成的数组。

```
function foo(name) {
            this.name = name;
        };

        var bar = {
            name : "bar"
```

```
};

var fooBound = foo.bind(bar);
alert(bar.name);
fooBound("foo");
alert(bar.name);
```

curry 方法与 bind 方法非常类似。它们的工作方式完全一样，只是在 curry 方法中不必传入作用域参数。

2. bindAsEventListener 方法

这是 bind 方法的特殊形式，它确保把事件对象传入绑定的函数。

3. Function.argumentNames 方法

这个方法将函数参数作为字符串数组返回。如果函数定义没有指定任何参数，那么该方法返回一个空数组。

```
function twoParams(foo, bar) {
        //do stuff here
};
var paramNames = twoParams.argumentNames();
//returns ['foo','bar']
```

5.4.2　操作函数的其他方法

有时我们需要改变一个函数，例如执行柯里化，调度以使其在以后运行，或者使用另一个函数将其包装起来。Prototype 提供了几个用于操作函数的方便方法。

1. defer 方法

defer 方法等待解释器闲置后才调用函数。如果希望等待不相关的 AJAX 调用完成之后再更新某个 UI 元素，那么这个方法就非常有用。

2. wrap 方法

这个方法接受一个函数作为参数，它将返回把第一个函数包装起来的新函数。这个方法可用来拦截函数调用，并执行我们想要的任何操作。下面是一个简单的 wrap 方法，它将一个函数包装起来，并把一些文本记录到 Firebug 或 Safari 控制台(如果这些控制台存在的话)。

```
var doStuff = function () {
            alert("Do stuff");
        }
doStuff = doStuff.wrap(
            function (func) {
               if(console) { console.log("calling doStuff"); }
               func();
            });
```

```
doStuff();
```

3. delay 方法

这个方法可用来把一个函数的调用调度到固定秒数之后执行。该方法的行为类似于全局方法 setTimeout。它甚至返回一个 ID，我们可以将该 ID 传给 clearTimeout 方法以停止这个延迟执行的函数。还可以传递一个在调用函数时要传入的实参数组。需要重点注意的是，在调用 setTimeout 方法时，指定的延迟单位是秒而不是毫秒。

4. methodize 方法

如果有一个接受对象作为参数的函数，而我们希望将该函数作为它所接受的对象的成员，那么可以使用 methodize 函数。这时会将这个函数包装到另一个函数中，而包装函数会把 this 关键字作为第一个参数传入。

5. Try.these 方法

这个有用的方法并没有包含在 Function 名称空间中。它接受 n 个参数，并返回第一个没有抛出异常的函数的结果。如果传给这个方法的函数没有一个能够成功执行，那么这个方法将返回 undefined。有时我们可能希望了解 Try.these 调用是否能够执行一组函数中的某一个函数，而不会抛出异常。在这些情况中，可以使用短路“或”运算符(||)返回 false。

```
function tryToExecuteSomeFunctions() {
return Try.these( function() { throw "exception";}, function() { throw "exception";
}) || false;
};
```

5.5　改进数组、散列和迭代器

JavaScript 中的所有对象的核心均是关联数组(associative array)。可以使用 for 循环遍历对象的属性和方法。Prototype 增加了一个新类 Enumerable，并使用该类的方法扩展了 JavaScript 的内置 Array 和 Object 对象。使用这些新方法，可以方便地采用任何合适的方式来操作数组和集合。each 方法可用来可靠地检查对象或数组中的元素。

5.5.1　使用 for...in 循环会导致一些问题的原因

如果曾经试着使用 for...in 循环遍历数组中的元素，那么可能已经注意到其结果并非总是符合预期情况。ECMA 262 规范(定义 ECMAScript 第三版的规范)规定只有标记为不可枚举的属性才应该被 for...in 循环忽略。但问题是，Prototype(以及其他 JavaScript 库)没有任何方式可以将其添加到 Array 和 Object 对象中的方法标记为不可枚举。因此，当试着使用 for...in 循环遍历一个经过 Prototype 扩展的数组时，除了数据元素之外，还会返回所有的扩展方法。通过使用 Prototype 提供的用于遍历的扩展方法，我们可以避免这个问题。

5.5.2　Enumerable 类

对于 Prototype 来说，Enumerable 类非常重要，它是所有基于集合的类或充当集合的现有对象扩展的基础。它为处理集合中的元素提供了大量方法。这个类是一个模块，因此不是直接使用它。相反，借助 Object.extend 方法将其添加到其他对象中。

1. 迭代器

迭代器就是一些能够传给 Enumerable 模块中定义的某些方法的函数，这些函数可以使用某种方式处理数据。Enumerable 类中的下列方法支持迭代器：all、any、collect、detect、each、eachSlice、find、findAll、grep、inject、map、max、min、partition、reject、select、sortBy 和 zip。

2. 设置迭代器的上下文

Enumerable 类的所有方法都带有第二个上下文参数。这个上下文参数决定了在该迭代器内部 this 关键字所引用的对象。如果没有指定上下文，就使用该迭代器的默认上下文。

all 方法

这个方法遍历 Enumerable 对象中的元素并确定该对象是否为 true 或 false。一旦找到某个计算结果为 false 的元素，它就会停止执行并返回 false。否则，它返回 true。可以提供一个迭代器来确定是否每一项应该返回 true 或 false。

any 方法

any 方法与 all 方法的工作方式类似，但是它在遇到第一个计算结果为 true 的元素时就会停止执行。

collect 和 map 方法

collect 方法是一个多功能的方法。它遍历 Enumerable 集合并返回结果，如图 5-2 所示。map 方法是 collect 方法的别名。

```
<body>
    <script type="text/javascript">
        var data = ['Deoxyribo', 'nucleic', 'acid'];

        function acronymize(dataArray) {
             return dataArray.collect(function(item) {
                  return item.charAt(0).toUpperCase();
             }).join('');
        };

        Event.observe(window, "load", function(e) {
            $("contentHolder").insert(acronymize(data));
        });


      </script>
      <div id="contentHolder"/>
</body>
```

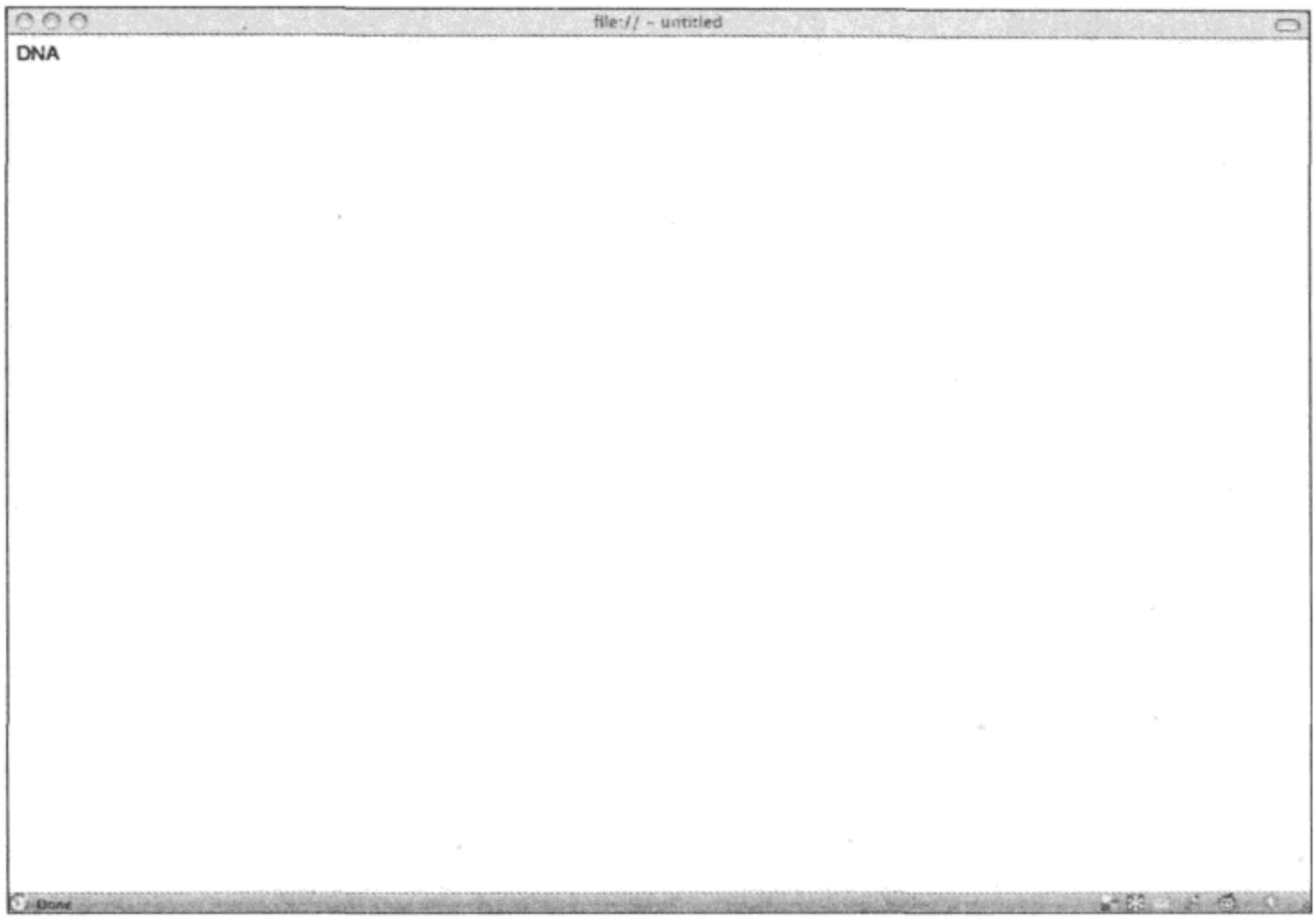


图　5-2

detect 和 reject 方法

detect 方法查找集合中第一个从迭代器中返回 true 的元素。reject 方法的作用与 detect 方法相反，它返回所有导致给定函数返回 false 的元素。

each 方法

each 方法是 Enumerable 类的主要方法。这个方法遍历 Enumerable 类，在每个元素上调用传给它的函数，然后返回 Enumerable 类，以便可以进行链式调用。

eachSlice 方法

如果希望按照大小划分 Enumerable 集合，那么 eachSlice 方法非常有用。假如需要使用 AJAX 调用某个 Web 方法，但是正在调用一个将其参数硬编码为 4 个参数(例如“name1, name2, name3, name4”)的遗留系统。此时可以使用 eachSlice 方法遍历数组，然后每次取出 4 个元素。

entries 方法

这是 toArray 方法的别名。

find、findAll 和 select 方法

find 和 findAll 方法均遍历 Enumerable 集合，然后返回导致迭代器函数返回 true 的集合元素。findAll 方法将返回每一个使迭代器函数返回 true 的元素，而 find 方法(它是 detect 方法的别名)只返回第一个满足条件的元素。select 方法是 findAll 方法的别名。

grep 方法

grep 方法可用于使用正则表达式搜索集合并返回结果。但是，不一定要使用正则表达式，可以使用字符串模式或者任何具有 match 方法的对象。

inGroupsOf 方法

这是 eachSlice 方法的一个变体，它不带迭代器函数。另一个主要的不同之处在于，它将使用 null 值填充最终返回的数组(如果需要满足大小条件的话)。

include 和 member 方法

include 方法确定 Enumerable 集合是否含有指定的值。需要重点注意的是，这个方法使用等于运算符(==)而不是恒等运算符(===)，因此只检查值而不是类型和值。member 方法是 include 方法的别名。

inject 方法

inject 方法利用在指定集合上的每次迭代来构建一个返回值。如果基于 Enumerable 集合来构建数组或者对一定范围的数值执行连续的计算，那么这是一个极好的方法。

invoke 方法

invoke 方法针对 Enumerable 集合中的每个元素执行给定的函数。因为没有在匿名函数上建立词法闭包(如果将一个匿名函数传给 each 方法，就会创建词法闭包)，所以这个方法可以较好地执行。

max 和 min 方法

max 方法返回集合中最大的元素。根据比较值目录或使用迭代器来确定最大的元素。如果 Enumerable 集合为空，就会返回 undefined。如果集合中出现相等的值，就会返回最后一个该值。min 方法的作用与 max 方法相反，它返回 Enumerable 集合中最小的值。

partition 方法

partition 方法将集合划分成两组。第一组将被视为 true，而第二组将被视为 false。记住，在 JavaScript 中，null 和 undefined 都被视为 false。可以提供一个迭代器函数来确定元素是 true 或 false。

pluck 方法

这是 collect 方法的优化版本。当希望根据所有元素的某个属性从集合中选取元素时，就可以使用这个方法。

size 方法

size 方法返回 Enumerable 集合中元素的数量。它与 Arrays 对象的 length 属性类似。

sortBy 方法

这个方法为排序 Enumerable 集合中的元素提供了一种方式。可以提供一个函数来返回希望进行比较的属性或计算值。只有在无法使用 sort 方法时才应该使用这个方法。

toArray 方法

这个方法将返回 Enumerable 集合的数组表示。

5.5.3　改进 Array 对象

除了添加 Enumerable 模块中的所有方法之外，Prototype 还增强了原生的 Array 对象，使其具有一些有用的方法，如表 5-2 所示。

表　5-2

方　法	说　明
clear	这个方法将指定数组的所有元素移除
clone	这个方法返回原始数组的副本。它不会改变原始数组
compact	这个方法将任何含有 null 值或 undefined 值的元素移除
each	这个方法从最低索引处开始遍历数组
first	这个方法返回数组的第一个元素。如果数组为空，那么它返回 undefined
flatten	这个方法返回一个一维数组。所有嵌套数组都将提取出来，它们的元素将插入到嵌套数组所在的位置
from	这个方法克隆现有的数组，或者根据一个类似于数组的集合来创建一个新数组。还可以使用辅助方法 $A()来表示该方法
indexOf	这个方法与 String.indexOf 方法的工作方式类似。它返回传入的值在指定数组中第一次出现时的索引。如果没有在数组中找到该值，那么这个方法返回-1
inspect	这个方法返回该数组的用于调试的字符串版本
last	这个方法的作用与 first 方法相反，它返回数组的最后一个元素
reduce	这个方法返回单元素数组的唯一值。如果是多元素数组，则保持不变
reverse	这个方法反转指定数组中元素的顺序。默认情况下，它会修改原始数组。可以传入一个可选的内联方法参数，如果设置为 false，那么该方法将返回反转数组的克隆
size	这个方法返回 Array.length 属性
toArray	这个方法根据 Enumerable 对象返回一个数组
toJSON	这个方法将数组转换成 JSON 字符串
uniq	这个方法在将原始数组中的所有重复元素移除之后创建一个新的数组。如果没有发现重复元素，就会返回原始数组
without	这个方法创建一个新的数组，其中不包含任何具有指定值的元素

注意：

在大型数组上使用上面这些方法时，并不建议使用匿名函数作为迭代器，因为这会牺牲性能。如果有一个大型数组，那么最好是编写一个普通的使用数字索引的 for 循环，在这个循环内部执行所需的操作。

5.5.4　引入 Hash 类

Hash 类是 Prototype 引入的一个辅助类。可以将散列视为一种关联数组。所有的 JavaScript 对象都是关联数组，因此 Hash 类定义了几个额外的方法，使得关联数组的处理变得更加简单。既可以使用关键字 new 创建 Hash 类的一个新实例，也可以使用别名$H()并传入任何 JavaScript 对象。如果没有传入 JavaScript 对象，就会创建一个新的空 Hash 对象。

1. clone 方法

clone 方法的工作方式与 Array.clone 方法相同，它返回 Hash 对象的相等副本。

2. each 方法

这个方法遍历 Hash 对象并针对其中的每个元素调用指定的函数。当调用该函数时，将 key 和 value 作为前两个参数传入该函数。与使用 for...in 循环不同的是，这个方法将跳过在对象的原型中找到的方法。并不保证按照任何特定的顺序调用元素。

3. get 方法

get 方法返回由传给该方法的 key 所指定的元素中的 value。

4. inspect 方法

inspect 方法返回该 Hash 对象的调试格式版本。

5. keys 和 values 方法

这两个方法都返回包含指定值的数组。它们的工作方式与扩展 Object 对象上的 keys 和 values 方法相同。

6. merge 方法

merge 方法将一个 Hash 对象的值与另一个已有的 Hash 对象合并。这并不是破坏性的操作，它在合并之前会克隆原始 Hash 对象。如果目标 Hash 对象含有与原始 Hash 对象相同的键，那么对应的值将被覆盖。

7. set 方法

set 方法将指定的值赋给 key 所指定的元素。

8. toJSON 和 toQueryString 方法

这两个方法按照预期的方式执行操作，它们的工作方式与其他 toJSON 和 toQueryString 方法一样。

9. toObject 方法

这个方法返回 Hash 对象的简单对象副本。

10. unset 方法

这个方法将指定 key 表示的元素删除，并将它的值返回给调用程序。

11. update 方法

update 方法与 merge 方法的操作方式类似，它将指定对象的键和值插入到 Hash 对象中。但是，与 merge 方法不同的是，这个方法属于破坏性操作，原始 Hash 对象实例将被修改。

5.6　处理数值和日期

由于 JavaScript 具有动态性，因此日期和数值非常难以处理。Prototype 使得我们可以更容易地把数值和日期转换成 JSON 格式，同时还提供了几个处理数值的数学方法。

5.6.1　数值

Prototype 扩展了 JavaScript 中的数值，从而使我们可以使用类似于 Ruby 中的数值范围，还可以使用数值指定反复执行特定操作多少次。Prototype 也提供了一些基本的数学函数。

1. abs 方法

这个方法返回数值的绝对值。

2. cell 方法

这个方法返回大于或等于指定数值的最小整数。

3. floor 方法

floor 方法的作用与 cell 方法相反，它返回小于或等于指定数值的最大整数。

4. round 方法

这个方法返回最接近的整数，它采取四舍五入的方式。

5. succ 方法

这个方法返回后继，也就是指定数值在数轴上的下一个数值。

6. times 方法

这个方法是编写简单循环的一种快捷方法。它反复地执行指定函数，具体的执行次数由其中给出的数值确定。

```
(3).times(function(n) { alert("hiya " + n)});
```

7. toColorPart 方法

这个方法带有一个十进制数值参数，返回该数值的十六进制表示。当需要将一个十进制颜色

值转换成 CSS 十六进制值时，这个方法就非常有用。

8. toJSON 方法

这个方法返回一个 JSON 格式的字符串。

9. toPaddedString 方法

这个方法返回一个使用足够多个 0(让整个字符串的长度等于 n)填充的数值。

```
(5).toPaddedString(3) // returns '005'
```

5.6.2　日期

Prototype 只提供了一个用于处理日期的方法。它扩展 Date 对象，使其具有 toJSON 方法，该方法将 Date 对象作为 JSON 字符串返回。这个方法对于使用 JSON 把日期传递到服务器来说非常有用。

5.7　本章小结

在本章中，我们了解到 Prototype 通过引入 Enumerable 和 Hash 类并扩展原生 Array 对象，使得在 JavaScript 中处理集合和数组变得更加简单。我们还学习了如何使用 Class.create 和 Class.addMethods 方法创建和维护对象层次结构。Prototype 的模板引擎可用来重用常用的 HTML 标记并绑定数据对象，使得构建用户界面变成一件轻而易举的事情。

PDG

第 6 章

扩展 Prototype

Prototype 是创建新的 JavaScript 框架的极佳基础。它负责解决不同 DOM 实现之间的不兼容问题，并使我们可以将注意力放在编写框架的有趣部分上。这个库的轻量级特性表明，即使网页必须包括 Prototype 以及自己的库，用户也不必担心这些网页的下载大小。

本章将介绍 3 个基于 Prototype 库的 JavaScript 框架：

- Script.aculo.us
- Moo.fx for Prototype
- Rico

6.1 Script.aculo.us

如图 6-1 所示，Script.aculo.us(请访问网站 http://script.aculo.us/)是在 Prototype 的基础上构建的一个效果库，它最初作为 Ruby on Rails 中包含的默认 JavaScript 库而出名。该效果库提供了极佳的视觉效果，并在 Prototype 的 Ajax.Request 对象的优秀基础上构建了一些 AJAX 控件。

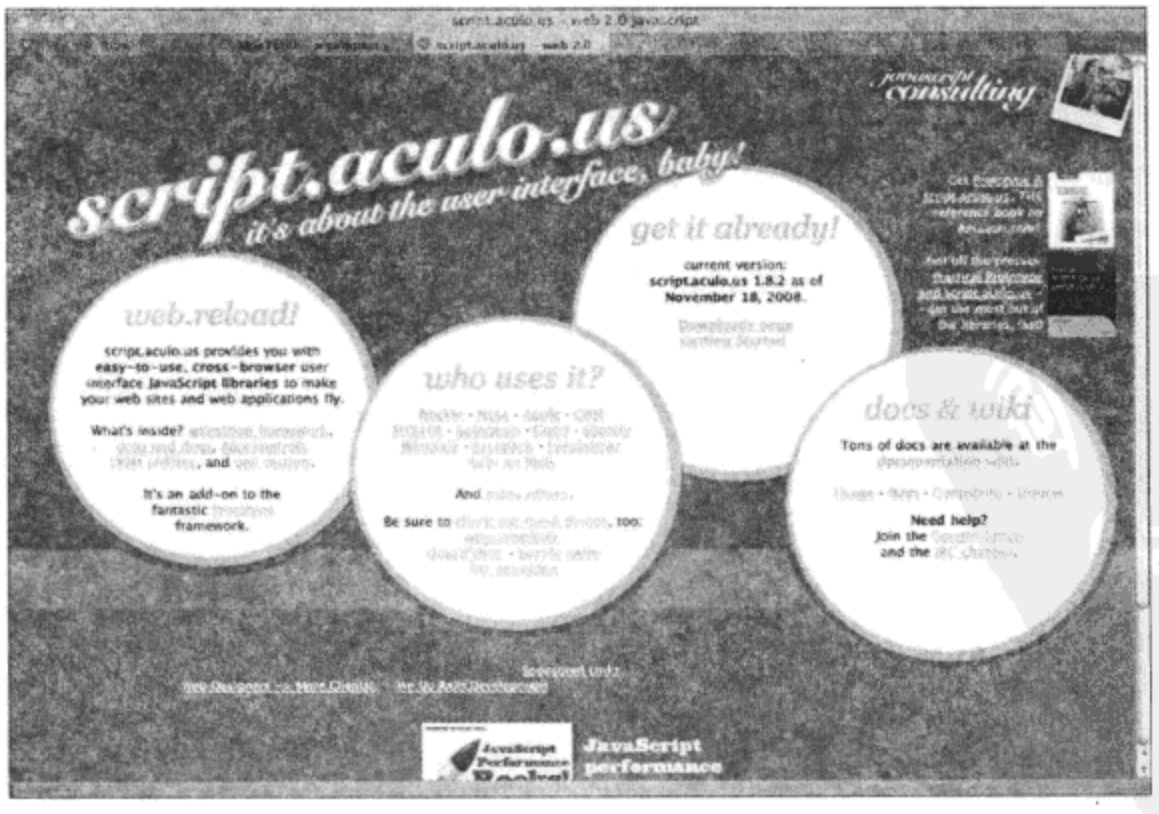


图 6-1

效果

Script.aculo.us 提供了一个视觉效果库，可以将这些效果添加到网页中以使网页变得更加绚丽和易用，也可以用来构建新的视觉效果。

该库提供的核心视觉效果有：

- Effect.Opacity
- Effect.Scale
- Effect.Morph
- Effect.Move
- Effect.Highlight
- Effect.Parallel
- Effect.Tween

通过合并两个或更多核心效果，该库中还创建了以下 16 个效果：

- Effect.Appear 和 Effect.Fade
- Effect.Toggle
- Effect.Puff
- Effect.DropOut
- Effect.Shake
- Effect.SwitchOff
- Effect.BlindDown 和 Effect.BlindUp
- Effect.SlideDown 和 Effect.SlideUp
- Effect.Pulsate
- Effect.Squish
- Effect.Fold
- Effect.Grow
- Effect.Shrink

使用所有这些效果的基本方式就是，传入希望应用这个效果的元素、这个效果指定的所有必要的参数以及任何希望传给这个效果的选项，以此创建这个效果的一个新实例。如果希望临时地将某个效果应用于某个元素，那么还可以使用 Effect.Toggle 对象。

```
new Effect.EffectName(element, required-params, [options]);
```

假设我们希望在元素中的文本发生变化时将该元素高亮显示。此时可以使用 Effect.Highlight 对象：将想要高亮显示的元素或它的 id 传入即可。

```
<body>
<div id="MainDiv"> Change the text </div>
<script type='text/javascript'>
Event.observe(window, "load", function(e) {
     setTimeout(function() { //simulate an AJAX request
          new Effect.Highlight("MainDiv");
          $("MainDiv").update("updated the content");
       }, 1000);
```

```
});
</script>
</body>
```

1. 控件

Script.aculo.us 有几个控件可用来简化 Web 开发。它有两个自动完成(autocomplete)控件，一个是服务器控件；另一个是本地控件；此外还有一个带有拖放功能的滑块控件以及一个就地编辑(in-place editing)控件。

Slider 控件

这是一个用来从一组数值中选择一个值的标准滑块控件。这个滑块控件比较有趣的地方在于它可用来定义多个处理程序。可以将它的轴线设置成水平或垂直。要使用该控件，就要创建这个类的一个新实例，并传入一个元素或元素 id 用作句柄，或者传入一组元素/id(如果需要多个句柄的话)。

```
new Control.Slider('handles','track', [options]);
```

Ajax.InPlaceEditor 控件

Ajax.InPlaceEditor 控件为网页上的数据字段提供了单击编辑功能，这是一种增强网站的极佳方式。如果您使用过 Flickr，就会用过就地编辑器。在文本“Click here to add a description”(“单击这里添加描述”)上单击时，该文本会变成一个输入框，可以输入所需的任何的文本。将光标悬停在该文本上时，Prototype 效果就会起作用，背景色淡入成黄色(或者是通过将选项对象传给构造函数选择的颜色)。

要创建一个新的 Ajax.InPlaceEditor 对象，只需要创建该类的一个新实例，并传入应该变成可编辑状态的元素/id 以及传入该控件 POST 其值时要使用的 URL 和任何想要设置的选项。指定的 URL 应该在响应正文中返回一个值，这个值将替换指定元素中的默认文本。

```
<body>
<div id="inPlaceEditorHolder"> Click here to edit </div>
<script type='text/javascript'>
Event.observe(window, "load", function(e) {
   var ipe = new Ajax.InPlaceEditor("inPlaceEditorHolder","/");
});
</script>
</body>
```

这里创建了一个含有文本“Click here to edit”(“单击这里编辑”)的 div 容器，并将其赋给一个新的 Ajax.InPlaceEditor 对象(参见图 6-2)。当在该文本中单击时，该文本就会替换成一个输入框、一个 OK 按钮以及一个 cancel 链接(参见图 6-3)。输入一个值，然后单击 OK 按钮，这会把输入的值 POST 到指定的 URL，并将 div 容器中的文本替换成服务器的响应正文中返回的值。

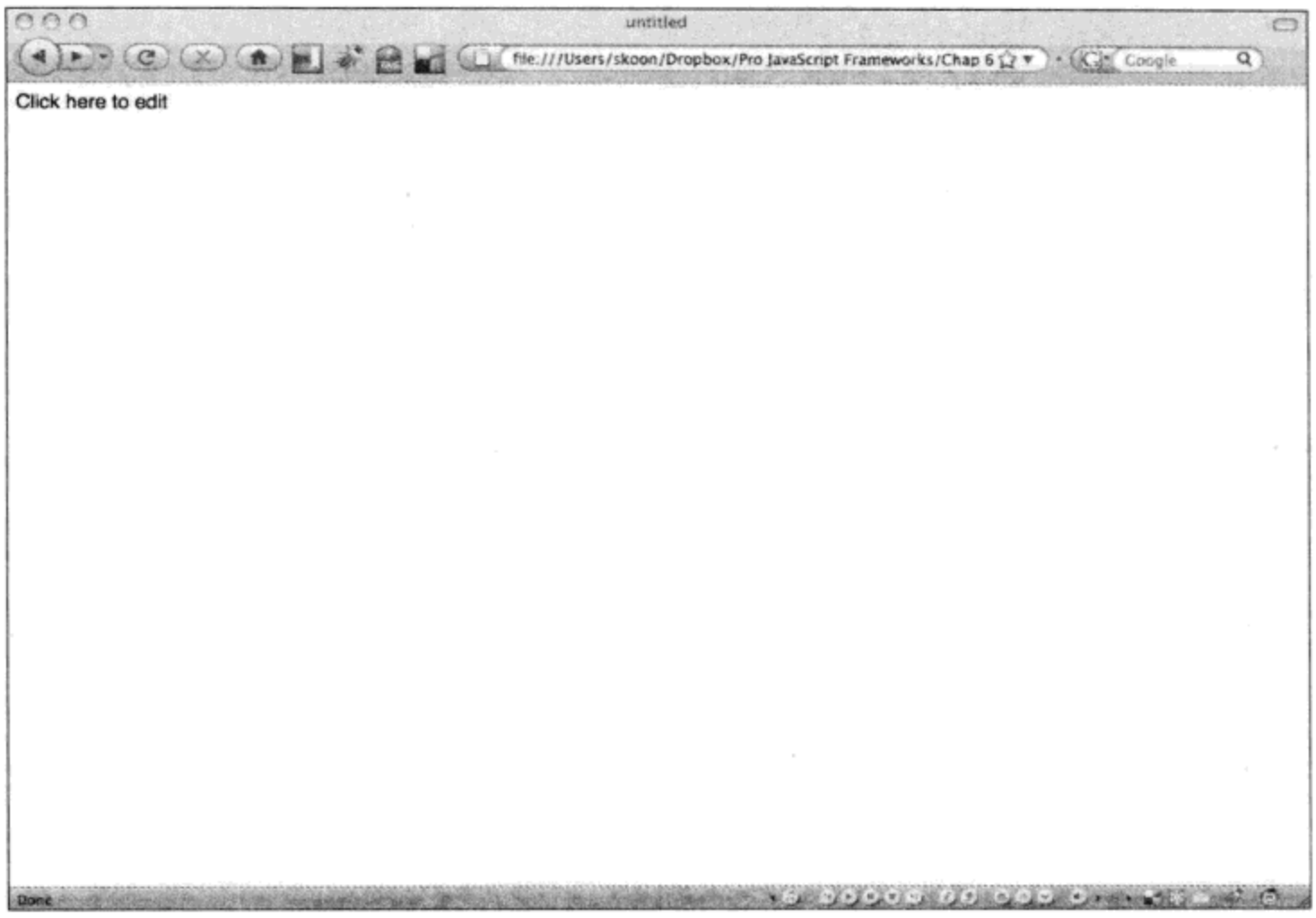

untitled
file:///Users/skoon/Dropbox/Pro JavaScript Frameworks/Chap 6
Google
Click here to edit
Done

图　6-3

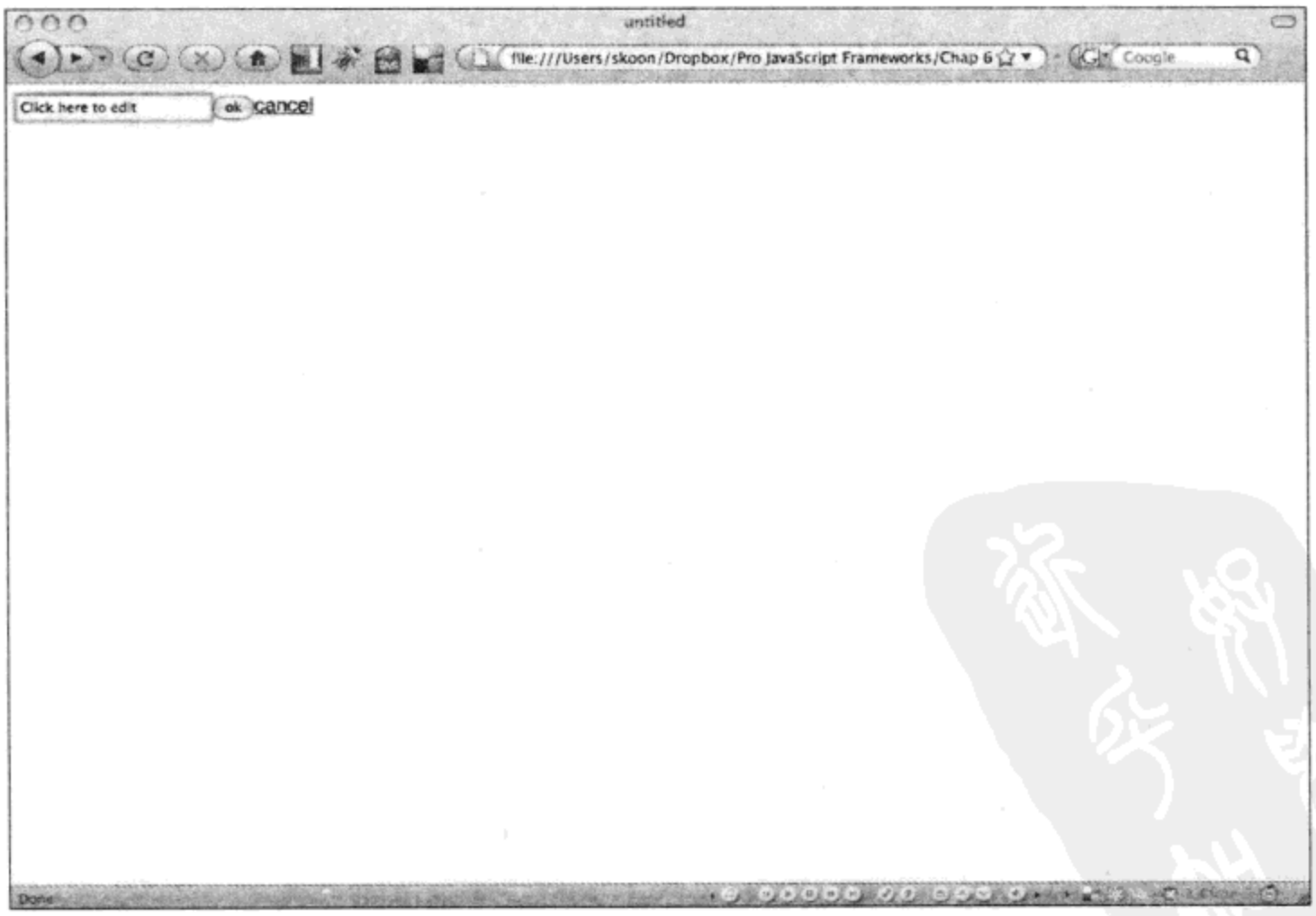

untitled
file:///Users/skoon/Dropbox/Pro JavaScript Frameworks/Chap 6
Google
Click here to edit
ok
cancel
Done

图　6-3

Ajax.InPlaceCollectionEditor 控件

这个控件的工作方式与 Ajax.InPlaceEditor 控件一样，但是它将指定元素替换成一个选择框而不是文本框。传入一个值数组以用于这个选择框。

```
new Ajax.InPlaceCollectionEditor( element, url,
                                  { collection: [array], [moreOptions] } );
```

Ajax.Autocompleter 和 Autocompleter.local 控件

Script.aculo.us 中的自动完成控件是网页中表单的优秀补充。Ajax.Autocompleter 控件利用 Prototype 中的 AJAX 对象，使我们可以方便地向输入元素添加自动完成功能。

首先，必须为自动完成控件将使用结果进行填充的 div 容器设置样式。一种较好的做法是在开始时将这个 div 容器的样式设置为 display:none(这样在使用结果填充它之前不会将其显示出来)，并将其 class 属性设置为 autocomplete。接下来，为 div 容器的选中状态设置某种指示器。

```
<style>

   div.autocomplete {
      margin:0px;
      padding:0px;
      width:250px;
      background:#fff;
      border:1px solid #888;
      position:absolute;
   }

   div.autocomplete ul {
      margin:0px;
      padding:0px;
      list-style-type:none;
   }

   div.autocomplete ul li.selected {
      background-color:#ffb;
   }

   div.autocomplete ul li {
      margin:0;
      padding:2px;
      height:32px;
      display:block;
      list-style-type:none;
      cursor:pointer;
   }

   </style>
</head>
<body>
<input type="text" id="autocompleteInput">
<div id="autcompleteResults" class="autocomplete" style="display:none;"> </div>
```

```
<script type='text/javascript'>
Event.observe(window, "load", function(e) {
     var stringArray = ["Alix", "Alice", "Amelia", "Amy"]
   var ac = new Autocompleter.Local(
      "autocompleteInput",
      "autcompleteResults",
      stringArray);

   });
   </script>
   </body>
```

最后，创建 Autocompleter 类的新实例来建立 Autocompleter 控件。Autocompleter 控件有两种不同的风格。第一种风格将使用文本框中输入的值查询 URL，而第二种风格则搜索传入的字符串数组以查找输入的文本。上面的示例建立了一个本地的 Autocompleter 控件，它将搜索一个含有女性姓名的数组，如图 6-4 所示。

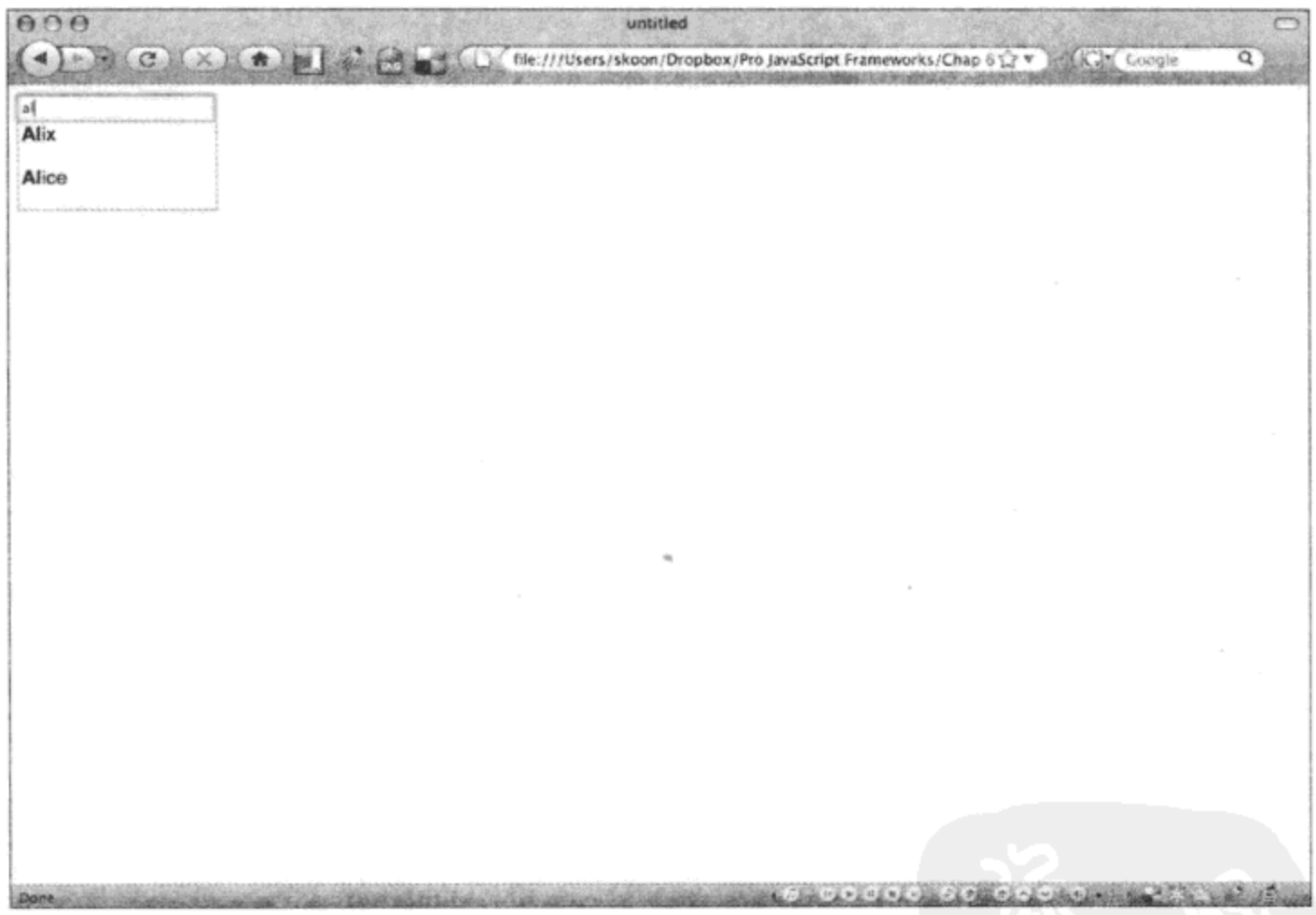


图　6-4

2. 行为

Script.aculo.us 定义了 3 种可以应用于元素的行为：

- **可拖动**　让用户能够在页面内拖动该元素。
- **可放置**　定义了一个区域，用户可以将一个可拖动元素放置在其中并触发一个事件。
- **可排序**　让用户重新排序容器中的元素。

拖放

在 JavaScript 中编写拖放代码是一件复杂的事情，可能会花费大量时间，而我们本来可以利用这些时间编写逻辑代码。Script.aculo.us 提供了两个类，使得我们可以定义一个用户可拖动的元素，或者定义一个可以放置元素的区域。要想让某个元素可拖动，只需要创建 Draggable 类的一个新实例，并传入希望可拖动的元素或其 id 以及其他选项。Script.aculo.us 维护一个可拖动元素的集合，要创建一个新的可拖动元素，只需要将其 id 添加到当前集合中即可。

```
new Draggable('id_of_element', [options]);
Droppables.add('id_of_element',[options]);
```

可排序

让一组元素可排序是一件简单的事情。只需要调用 Sortable.create 方法，并传入待排序的元素或其 id。

注意：

几乎可以使用任何容器元素作为可排序元素的容器，但是 TABLE、THEAD、TBODY 和 TR 这些元素除外，这是由于当前浏览器中的技术限制所导致的结果。

```
<body>
<div id="MainDiv">
     <ul id="sortableContainer">
          <li> first </li>
          <li> second </li>
          <li> third </li>
     </ul>
</div>
<script type='text/javascript'>
Event.observe(window, "load", function(e) {
     Sortable.create("sortableContainer");
});
</script>
</body>
```

6.2　Moo.fx for Prototype

Moo.fx(请访问网站 http://moofx.mad4milk.net/，如图 6-5 所示)的主要目标是成为一个小型的轻量级效果库。Moo.fx 库可以与 JavaScript 库 Prototype 或 MooTools 结合使用。Moo.fx 库中的基本效果有如下几种：

- **Fx.Tween**　可用来将任何 CSS 属性从一个值逐渐地修改成另一个值。
- **Fx.Morph**　可用来一次性改变多个 CSS 属性的值。
- **Fx.Transitions**　定义一组可在转换元素属性时使用的方程式。
- **Fx.Slide**　可用来让一个元素沿着垂直方向或水平方向滑动。

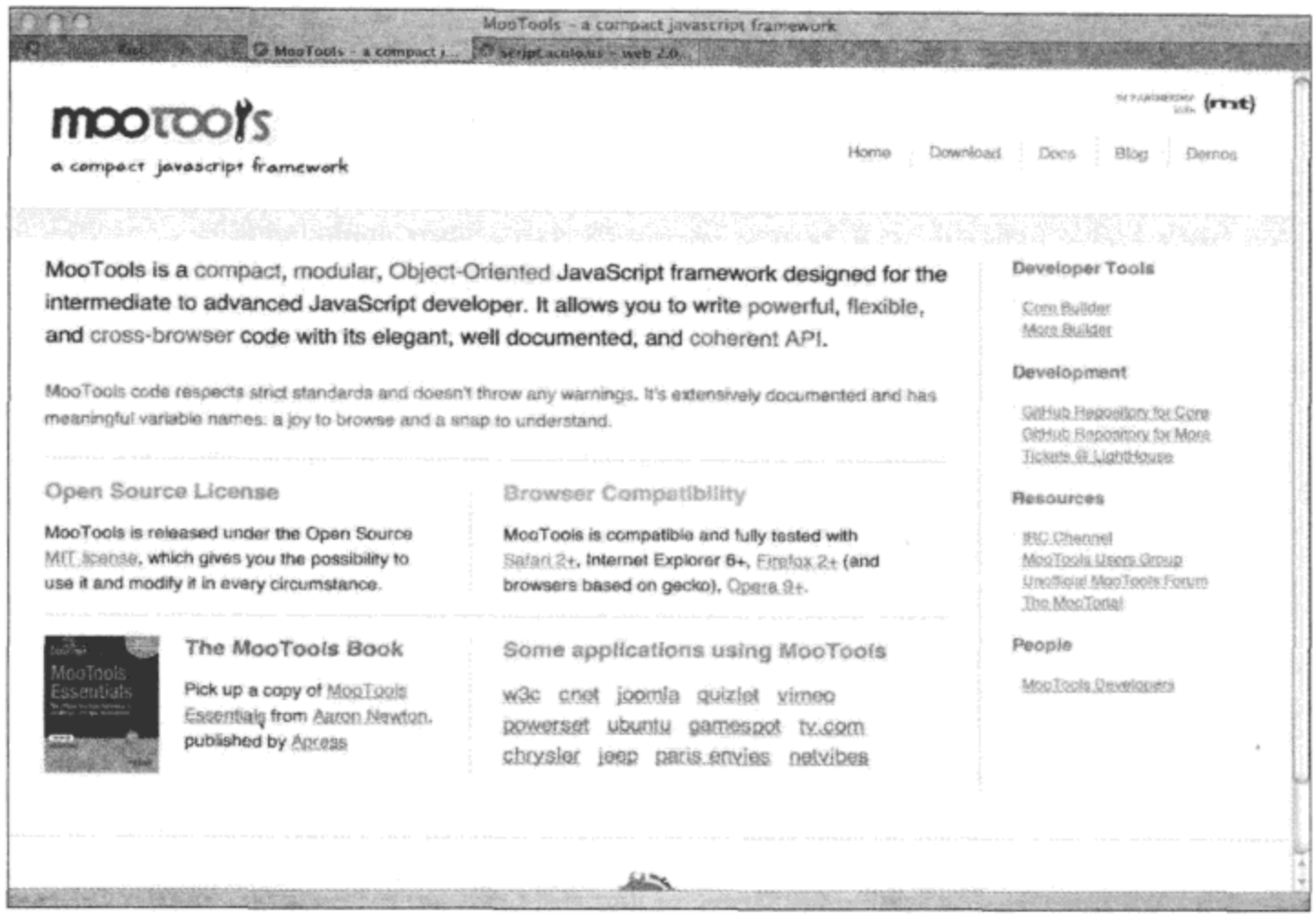


图　6-5

6.2.1　Fx.Tween

Fx.Tween 将指定 CSS 属性的值改为指定的最终值，它有两个方法。set 方法会立即修改 CSS 属性的值，而 start 方法逐渐地将 CSS 属性的值修改成传入的值。在调用 start 方法时可以选择提供一个最终值，并定义一个值范围。

```
var myFx = new Fx.Tween(element, [, options]);
myFx.set(property, value);
myFx.start(property, from, [to]);
```

6.2.2　Fx.Morph

Fx.Morph 一次性变换多个 CSS 属性。当在 Fx.Morph 实例上调用 start 方法时，必须将一个含有 CSS 属性及其值的对象传给它。可以使用 CSS 属性传入一个数组来指定该属性的修改范围。

```
var myFx = new Fx.Morph(element[, options]);
myFx.start({height: [100, 300], width:[100,300]});
```

6.2.3　Fx.Transitions

这是一个散列，其中包含了一组方程式，当转换元素的 CSS 属性时可以使用这些方程式。表 6-1 给出了部分方程式。

表　6-1

方　程　式	说　　明
Linear	线性转换
Quad	二次方程转换
Cubic	三次方程转换
Quartic	四次方程转换
Bounce	回弹转换
Elastic	弹性曲线转换
Circ	循环转换

6.2.4　Fx.Slide

Fx.Slide 可用来让一个元素沿着垂直方向或水平方向逐渐移动。根据想要完成的操作，可以调用 slideIn 或 slideOut 方法。当调用这些方法时，可以传入 mode 参数(默认值为 vertical)。

```
var myFx = new Fx.Slide(element[, options]);
myFx.slideIn("horizontal");
myFx.slideOut("vertical");
```

6.3　Rico

Rico(请访问网站 http://openrico.org/，如图 6-6 所示)框架有一些预先构建的优秀组件，还有一些用来创建诸如圆角之类效果的样式辅助工具。Rico 方面的开发已经停顿了一段时间，但是于 2009 年 5 月 3 日发布了 2.1 版本。

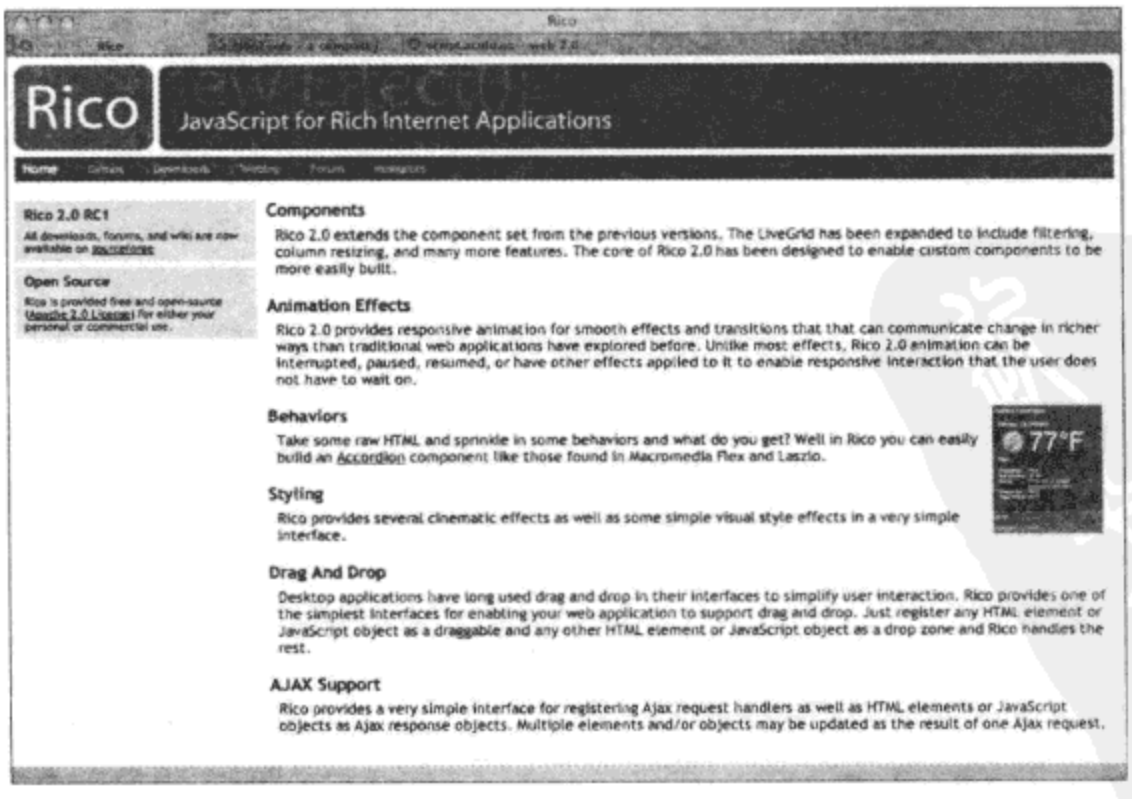


图　6-6

6.3.1　组件

Rico 组件是一些可以用来增强 JavaScript 应用程序的预先构建的窗口部件。Rico 有两种网格组件，可用来显示和排序表格型数据。此外，它还含有可折叠组件、日历以及弹出窗口等组件。

1. LiveGrid 和 SimpleGrid 组件

这两种网格组件有一些共同的特性：列标题始终固定在网格的顶部，网格的第一列位置固定，各列的大小可调整。它们可以从 JavaScript 数组、XML 源或者 AJAX 调用的结果中加载数据。Rico 提供了多种采用各种编程语言(包括.NET 和 PHP)编写的插件，可通过 SQL 查询把数据加载到网格中。简单网格是 Rico 2.0 版本的新功能，它是 LiveGrid 表的静态版本，通过使用某种插件或者对网页中预先填充的 HTML 表执行 XSLT 变换来填充数据。我们过去能够在客户端中执行 XSLT 变换，但是 Prototype 库的一处改动导致我们必须在服务器上执行该变换。

利用 LiveGrid 组件，使用 JavaScript 数组可以快速、方便地建立一个可排序的、可调整大小的网格，如图 6-7 所示。

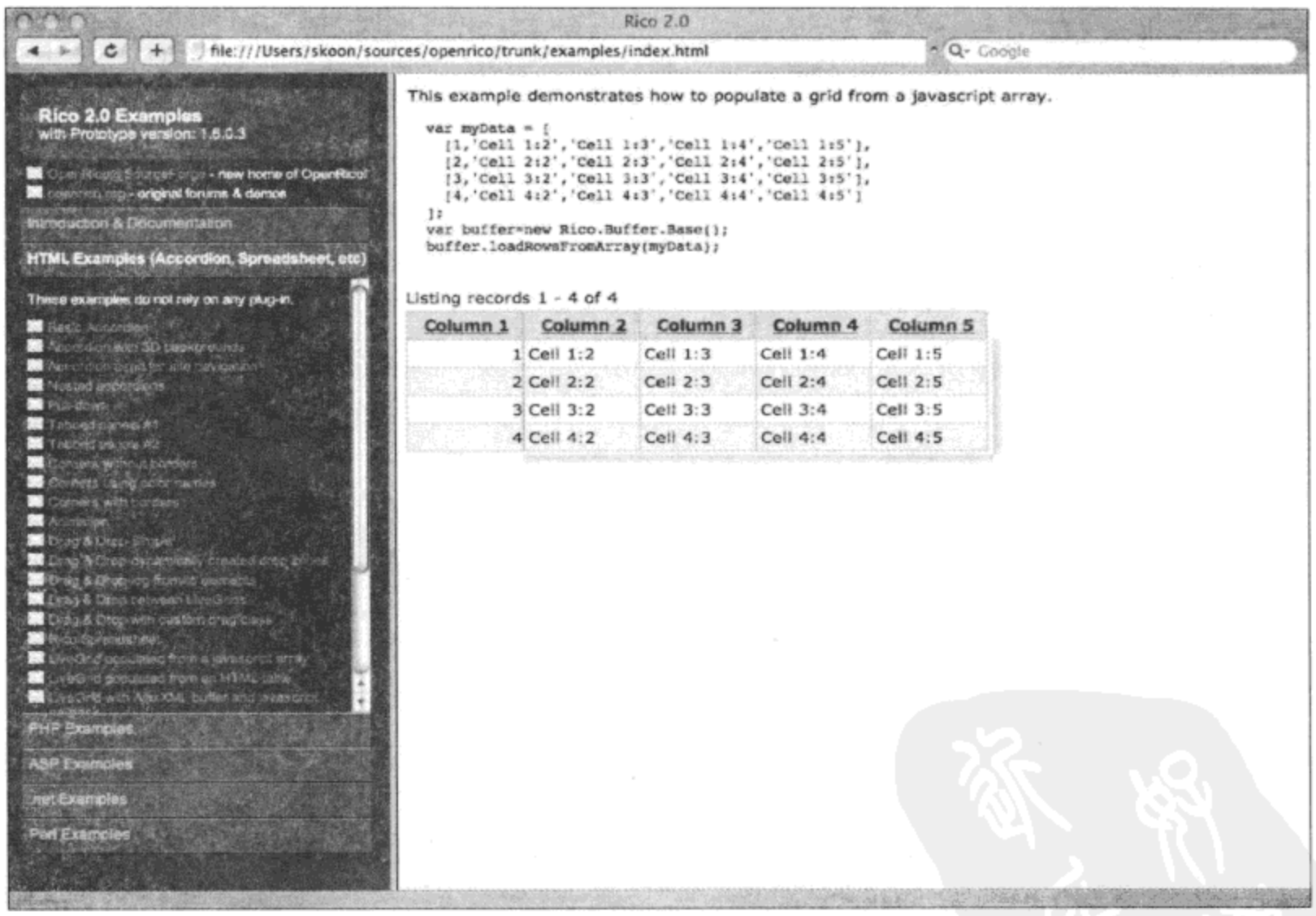


图　6-7

2. 可折叠组件

可折叠组件(参见图 6-8)将多个元素彼此堆叠在一起显示，用户可以切换打开其中的一个元素。

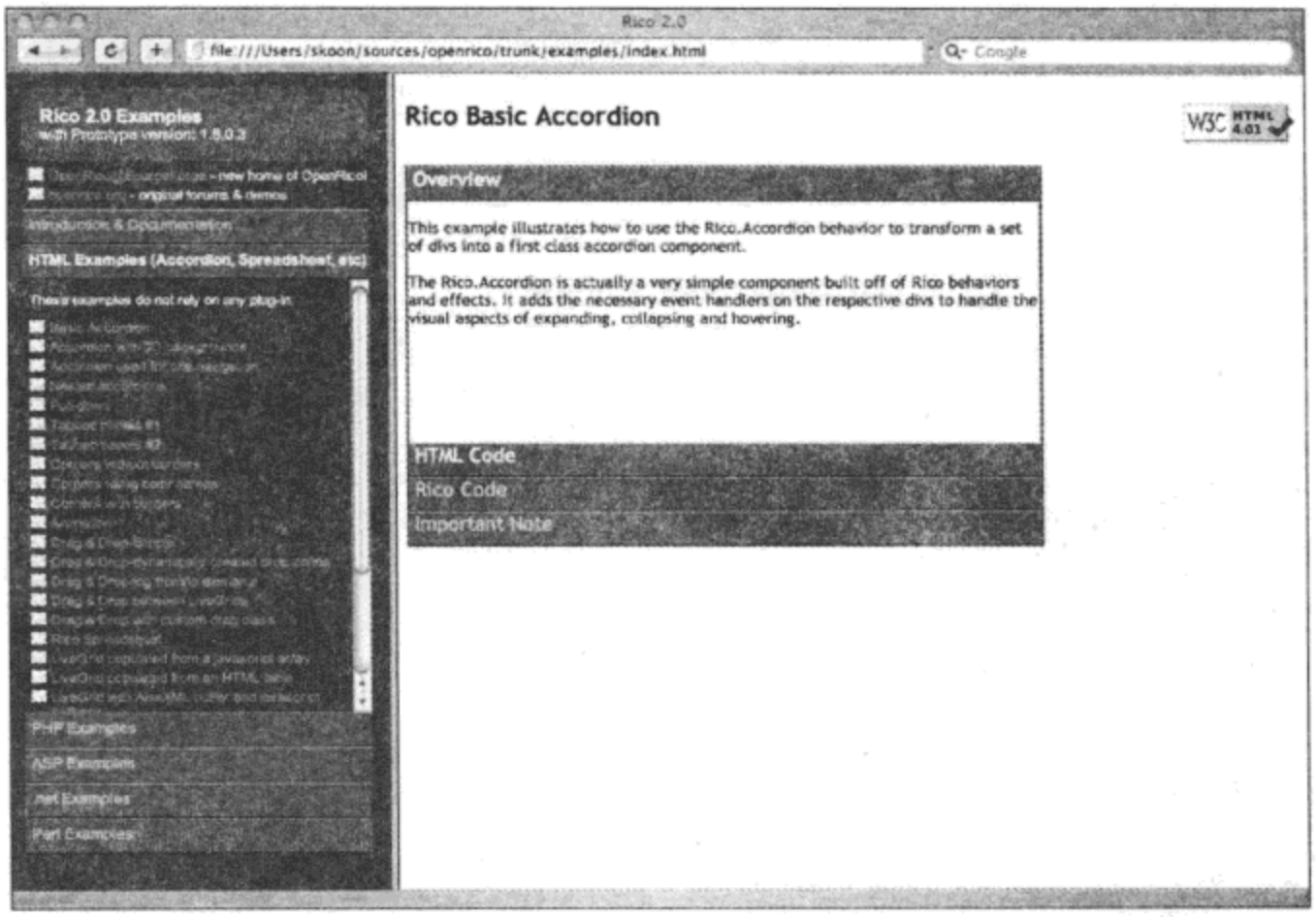


图　6-8

3. 弹出窗口

Rico 有两种弹出窗口，如图 6-9 所示。一种弹出窗口可以使用鼠标单击操作关闭，而另一种弹出窗口不能这样关闭。

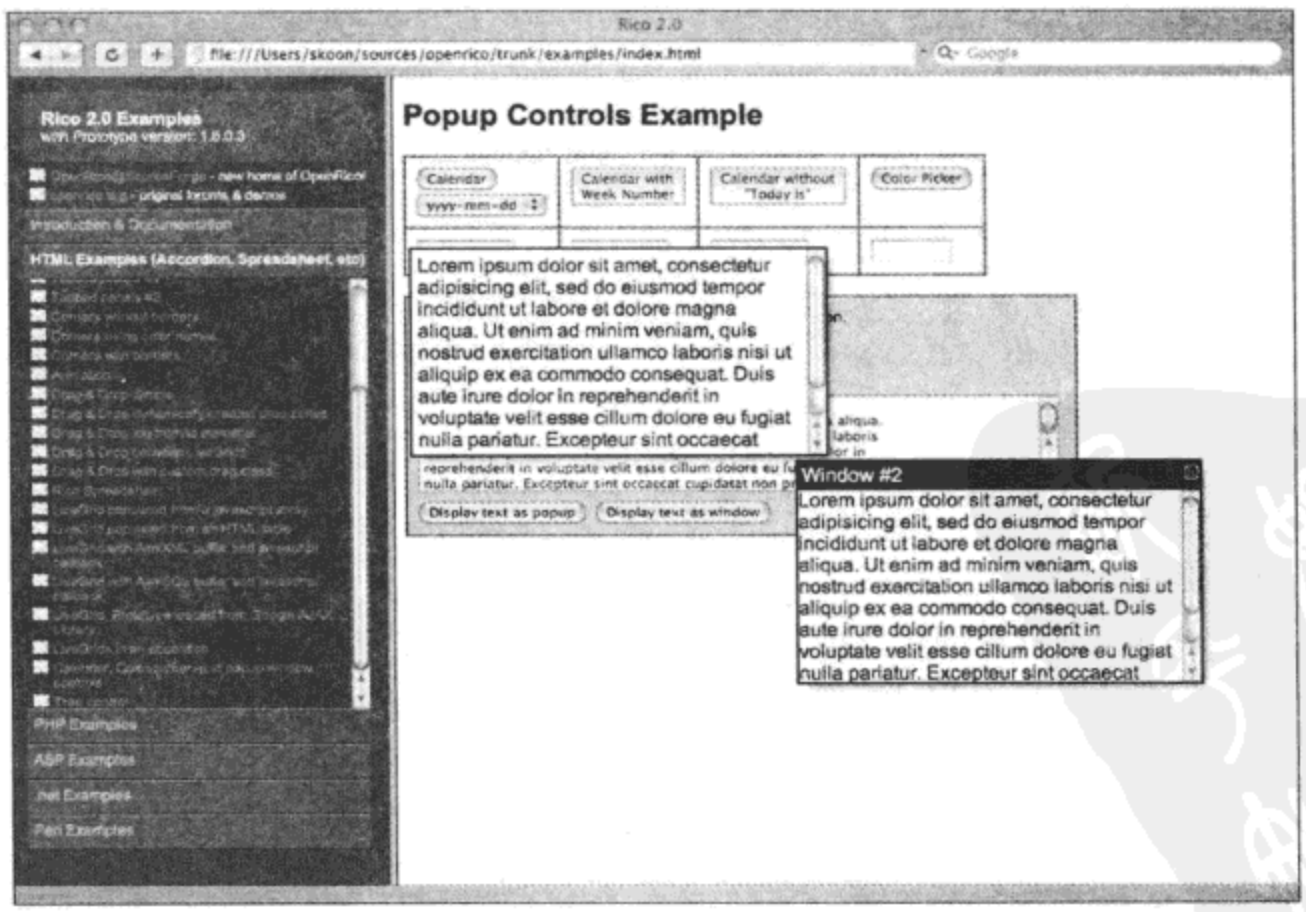


图　6-9

4. 颜色拾取器

可以创建一个简单的颜色拾取器(参见图 6-10)，它将返回用户选中的值。

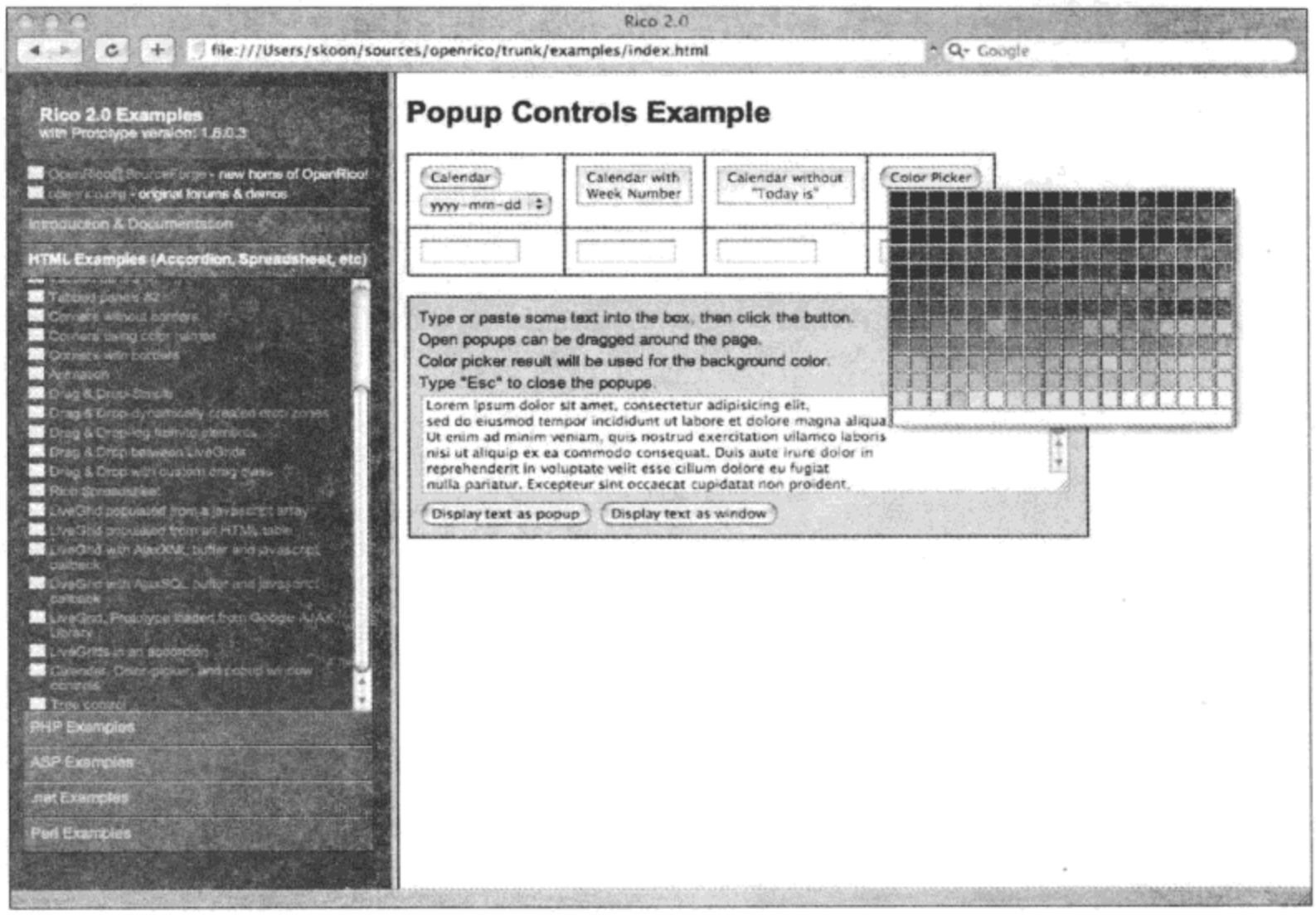


图　6-10

6.3.2　动画效果

Rico 有 3 种预定义效果：淡入/淡出、改变大小以及改变位置。可以将这 3 种效果组合起来使用，让元素在页面中逐渐放大，最终消失。

6.3.3　圆角

在网页设计中比较难以实现的效果之一就是圆角。使用 CSS 或切片图像实现这种效果的方式有多种。而利用 Rico，只需要编写一行 JavaScript 代码即可(参见图 6-11)。

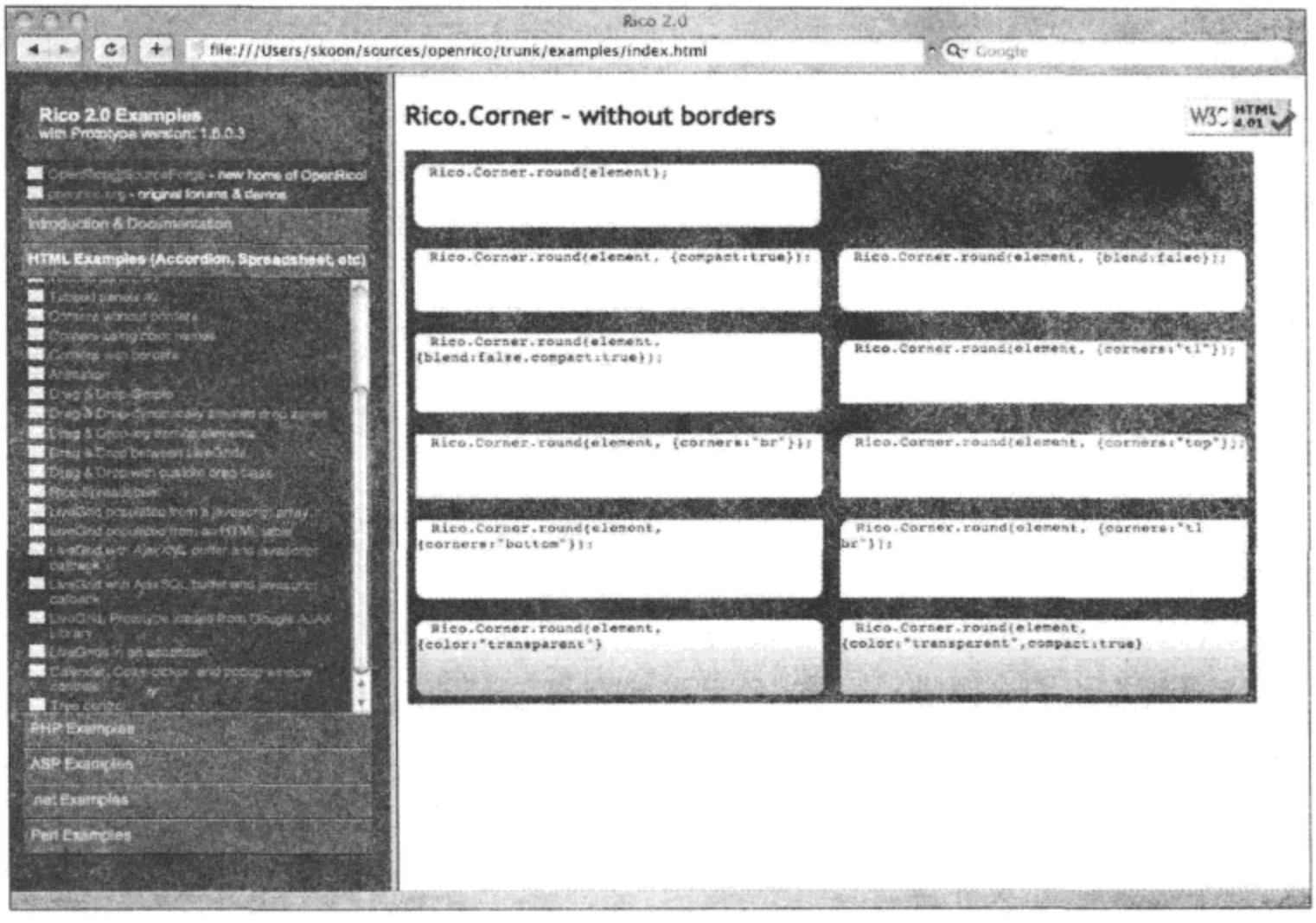


图　6-11

6.3.4　拖放

Rico 支持使元素可拖动和可放置，此外还可以定义某个元素在可以放置到可放置区域之前必须具有的属性。

6.4　本章小结

本章简要地介绍了 3 种基于 Prototype 库的 JavaScript 框架，即 Script.aculo.us、Moo.fx 和 Rico。这 3 种框架都提供了可用于网页中的极佳视觉效果。Script.aculo.us 有大量预先构建的效果。Moo.fx 可用来方便地在已有方程式的基础上创建新的转换，以及通过组合在不同 CSS 属性之间的转换来创建新的效果。Rico 提供了多个有用的预先构建的窗口部件，包括静态和动态网格控件、树状控件以及可折叠控件。

第 II 部分

YUI 库

整个雅虎用户界面库(Yahoo! User Interface Library，YUI)解压缩之后接近 50MB，包括素材、示例、测试、文档和包含主库的 3 种类型的文件。可以在 http://developer.yahoo.com/yui/中找到 YUI 库。

YUI 库划分为如下 4 个主要的组：

- YUI 核心
- 实用工具
- 控件/窗口部件
- CSS 工具

每个组又划分为个别组件，可以根据需要使用这些组件，而无须在站点上包含整个库。所有组件都依赖于 YAHOO Global Object，它负责一些必需的基础性工作。此外，大多数组件还需要 DOM Collection 和 Event Utility。每个组件都有精简版、标准版以及调试版。调试版将所有类型的信息记录到日志记录器，可用来跟踪组件的内部工作情况。调试版的大小总是最大，而且绝对不应该用在生产环境中。就此而言，也不应该使用标准版，这是因为正如其名称所暗示的那样，它是一个充满空白和注释(没有日志记录)的标准 JavaScript 文件。

将调试版和标准版中的任何一个放在生产环境中都将意味着，为了使组件能够工作，必须强迫访问者下载至少两倍于必要文件大小的数据。这正是创建精简版的原因。精简版只用于生产环境中的部署，这是因为它已经将所有空白和注释删除，并且在其上执行了一些用于节省空间的操作。精简版的效果与使用代码混淆技术的效果差不多，而 YUI 团队并不相信代码混淆技术，这是因为它可能引入程序错误。尽管如此，YUI 组件精简版的大小仍然要比标准版和调试版小很多。

此外，Yahoo!还通过版本化的 URL 提供了这些文件的托管版本。可以在 http://developer.yahoo.com/yui/articles/hosting/中找到所有版本化 URL 的详细的(以及维护的)列表。

下面是 YUI 托管文件的版本化 URL 示例：

```
< script type="text/javascript"
   src="http://yui.yahooapis.com/2.7.0/build/yahoo/yahoo-min.js" > < /script >
```

使用 YUI 托管版本的好处就是性能。这些文件都是通过低延迟服务器提供服务，并且进行全局缓存。

最后，这个库提供了最常用文件的聚合版本，从而减少 HTTP 请求次数。有两个这样的文件：yahoo-dom-event.js 和 utilities.js。很明显，前者包含 YAHOO Global Object、DOM Collection 和 Event Utility。后者包含这些组件以及 Element Utility、Connection Manager、Drag & Drop Utility 和 Animation Utility。

关于名称空间的名称的注意事项

YUI 的基本原则是全局变量有一定缺陷，无论从性能的角度还是它们容易与其他全局变量冲突来看均是如此。这就是整个 YUI 库只使用了一个全局变量 YAHOO(将其他所有组件都组织在其中)的原因。在嵌套对象中组织组件，这些嵌套对象的名称清晰地描述了各自的功能。因此，DOM Collection 位于 YAHOO.util.Dom 对象中，而 Event Utility 位于 YAHOO.util.Event 对象中。全局变量 YAHOO 采用大写是为了进一步降低与其他具有相同名称的全局变量产生冲突的可能性。

JavaScript变量是区分大小写的，因此YAHOO可以与Yahoo和yahoo共存，而不会有任何问题。

人们广泛认可的一个事实是，尽管命名空间的名称具有很好的描述性，但是却比较长，而且难以输入。因此，人们引入了快捷方式：

```
var Y = YAHOO.util.Dom;
var foo = Y.get("foo"); // instead of YAHOO.util.Dom.get("foo");
```

为了演示该技术，下面几个代码示例使用了快捷方式：

```
var YD = YAHOO.util.Dom,
    YE = YAHOO.util.Event,
    YEl = YAHOO.util.Element,
    YS = YAHOO.util.Selector;
```

此外，遵循尽可能少地使用全局变量的精神，代码示例将倾向于包装在自执行的匿名函数中。这项技术的目的是提供某种“沙箱”，所有示例变量的作用域都限制在匿名函数内部：

```
(function () {
     var foo ="This variable is not globally available.";
})();
```

当然，还可以使用namespace函数在YAHOO Global Object下创建自定义命名空间，例如：

```
YAHOO.namespace("foo.bar.baz");
```

这段代码创建了命名空间YAHOO.foo.bar.baz。如果将YAHOO作为命名空间字符串的一部分传入，那么它会被忽略，这是因为YAHOO是命名空间的隐含部分。因此，下面的代码片段也将创建YAHOO.foo.bar.baz名称空间：

```
YAHOO.namespace("YAHOO.foo.bar.baz");
```

第 7 章

利用 YUI 库遍历和操作 DOM

开发人员经常要担心陷入不同 Web 浏览器的特性所造成的困境中。在一种浏览器上遇到麻烦就意味着要浪费时间进行调试，而不是继续进行开发。这还可能意味着，为了绕开浏览器实现的缺陷，程序中会有大量代码分支。因此，YUI 团队建立了 DOM 集合，这是一组“用于 DOM 交互的便利方法”。这当然不会解决代码分支问题，但是所有的分支以及缺陷补丁都是在一个集中的便于维护的库中进行处理。而且，YUI 团队的部分任务是要测试它们的库在所有主流浏览器上的稳定性，而这并不是一般开发人员总是有时间去做的工作。其结果就是，尽管我们或许可以编写在 YUI DOM 集合中能够找到的大量函数，但是 YUI 团队已经替我们完成这项工作，而且已经确保它能够跨所有主流浏览器运行。

本章内容简介：

- 遍历 DOM 以及查找元素
- 操作内容
- 处理 CSS 和页面样式

7.1 遍历 DOM 以及查找元素

在遍历 DOM 时可以找到如下的元素。

7.1.1 get 方法

可以将其视为 getElementById 方法，但是功能更强大。DOM 方法 getElementById 工作得很好，但是它欠缺一些灵活性。顾名思义，getElementById 方法只附带一个参数，也就是一个表示元素 ID 的字符串。而 get 方法将 getElementById 方法包装起来，可以传入一个字符串 ID、HTMLElement 或者一个包含这两种参数之一的数组。如果接收到的是字符串 ID，那么它执行简单的 getElementById 方法，并返回找到的第一个元素。如果接收到的是 ID 数组，那么它返回匹配这些 ID 的 HTMLElement 所构成的数组。如果遇到元素作为参数，那么它只是立即返回同样的元素。

当这个参数是传递给构造函数的参数时，最后一种行为非常有用，它具有一定的灵活性，可

以从外部接收参数。

下面是一个示例：

```
function MyConstructor(id) {
        this.el = YAHOO.util.Dom.get(id);
};

var obj1 = new MyConstructor("foo");

var obj2 = new MyConstructor(document.body);
```

MyConstructor 函数的第一个实例将一个 ID 为 foo 的元素存储起来。第二个实例将 body 元素传给构造函数，然后将其传给 get 方法。get 方法识别出它是一个 HTMLElement 对象，只是将其立即返回。因此，MyConstructor 函数的第二个实例将 body 元素存储到它的 el 属性中。这就是 get 方法所提供的灵活性。

如果需要返回一组 HTMLElement 对象，那么还可以为 get 方法传入一个 ID 数组。在某些批处理操作中，如果希望确定某个数组的内容全部是 HTML 元素，那么这个行为非常有用。同样，get 方法会在所有的字符串上执行 getElementById 方法，并为每个字符串返回一个元素。因此，下面这段代码的运行结果就是一个含有 3 个元素的数组 arr，第一个参数实际上会被忽略，原因就在于它已经是一个元素。

```
var foo = document.getElementById("foo");
var arr = YAHOO.util.Dom.get([foo, "bar", "baz"]);
```

7.1.2　getElementsByClassName 方法

W3C DOM 规范中一个较为显而易见的疏漏之处是没有一个通过类名来获取元素的方法。既然有了 getElementById、getElementsByTagName 甚至 getElementsByName 方法，那么人们理所当然地认为还应该有一个 getElementsByClassName 方法，但是并没有这样的方法。因此，YUI 提供了它自己的 getElementsByClassName 方法。它的功能与 getElementsByTagName 方法类似，不仅可以在 document 元素上调用它，而且可以在任何其他元素上调用，这使得它成为搜索的起点。因此，下面的代码示例都是有效的：

```
var foo = YAHOO.util.Dom.getElementsByClassName("foo");

var bar = document.getElementById("bar");
var baz = YAHOO.util.Dom.getElementsByClassName("baz", null, bar);
```

既然这是一种 JavaScript 解决方法，而不是浏览器本身的方法，那么需要考虑它的运行速度。为了达到快速运行的目的，还有第三个参数可供使用。传入正在搜索的节点类型，例如：

```
var foo = YAHOO.util.Dom.getElementsByClassName("foo", "a");
```

这个示例获取文档中所有具有类名 foo 的锚点元素。由于 getElementsByClassName 方法的构造方式的缘故，必须进行这种优化。它在内部执行 getElementsByTagName(*)方法(这会匹配所有的元素)，然后遍历每个元素来查找指定的类名。如果给定了一个节点名称，那么它需要遍历的节点集合将迅速变小，从而获得更佳的性能。

最后一个参数可用来在找到的每个节点上执行一个函数。与其再编写一个循环对返回的元素集合应用一个函数，不如利用目前这个用于查找这些元素的循环。这个附加的参数的作用同样也是为了优化性能。通过使用这个参数，就不再需要第二个循环，这在较大的节点集合中可以获得性能上的显著提升。

下面是一个演示如何使用该方法的示例：

```
function addClick(el) {
       el.onclick = function () {
            alert("Click!");
       }
};
var nodes = YAHOO.util.Dom.getElementsByClassName("foo", "a", document, addClick);
```

在内部，唯一传给 addClick 函数的参数是循环中的当前元素(由变量名 el 表示)。

7.1.3　getFirstChild 和 getLastChild 方法

一个用来说明跨浏览器特性的优秀示例就是 DOM 方法 firstChild。在 Internet Explorer 中，它返回的是给定元素的属于 HTMLElement 类型的第一个子节点。而在 Firefox、Safari 和 Opera 浏览器中，情况并不是这样。在标准模式中，这些浏览器均严格遵循 W3C 规范，返回给定元素内的第一个文本节点。这是因为(根据 W3C 规范)DOM 中每个类型为 HTMLElement 的节点(也就是 HTML 标记)在其任意一侧都有一个空白的文本节点作为其兄弟节点。这是一个比较麻烦的问题，因为当需要第一个子元素时，所需要的总是 HTMLElement 而不是文本节点。getFirstChild 方法确保总是返回 HTMLElement 类型的节点，从而将这种行为规范化。也就是说，它跳过了第一个文本节点。

下面的示例演示了该方法的用法：

```
<html>
     <head>
        <title> YUI getFirstChild / getLastChild Test </title>
     </head>
     <body>
        <p> Hello World </p>
        <p> This is not the first child. </p>
     <p> Neither is this. </p>
     <script src="yahoo-dom-event.js"> </script>
     <script>
          var helloWorld = YAHOO.util.Dom.getFirstChild(document.body);
          var lastScript = YAHOO.util.Dom.getLastChild(document.body);
     </script>
   </body>
</html>
```

这段代码将返回在文档的正文中找到的第一个 HTMLElement。在这里，它就是包含文本“Hello World”的段落元素。

类似地，getLastChild 方法确保在 Firefox 中返回倒数第二个节点，从而将关于跨浏览器文本

节点的古怪行为规范化。在上面的示例中，最后一个子节点刚好是包含该示例的 JavaScript 代码的 script 标记。因此，这个变量的名称为 lastScript。

7.1.4 getFirstChildBy 和 getLastChildBy 方法

有时我们需要的节点并不是节点集合中的第一个或最后一个节点，而是节点集合中特定类型的第一个或最后一个节点。但是，没有 W3C DOM 方法可用于这种节点检索形式。请不要担心，YUI DOM 集合刚好实现了这些方法。而且，YUI DOM 集合并没有创建特定于某些条件的方法(例如 getElementById)，它将条件的确定权交给程序员。实现这种灵活性的就是下面这两个方法：getFirstChildBy 和 getLastChildBy 方法。这两个方法都附带一个起始节点作为它们的第一个参数，并且附带一个函数作为第二个参数。这个函数接收一个元素作为参数，并且必须返回 true 或 false。下面是这些方法的实际用法：

```
<html>
    <head>
        <title> YUI getFirstChildBy / getLastChildBy Test </title>
    </head>
    <body>
        <p> Hello World </p>
        <p class="intro"> Hello Planet! </p>
        <p class="intro"> This is not the first child. </p>
        <p> Neither is this. </p>
        <p class="outtro"> This one is somewhere in the middle. </p>
        <p class="outtro"> So is this. </p>
        <script src="yahoo-dom-event.js"> </script>
        <script>
            var YD = YAHOO.util.Dom,
                YX = YAHOO.example;
            YX.intro = YD.getFirstChildBy(document.body,
                function (el) {
                    return (el.className === "intro");
                });
            YX.outtro = YD.getLastChildBy(document.body,
                function (el) {
                    return (el.className === "outtro");
                });
        </script>
    </body>
</html>
```

在上面的示例中，intro 变量存放着含有文本“Hello Planet!”的段落元素，而 outtro 变量则包含带有文本“So is this.”的段落元素。这些段落元素均不是 body 元素的第一个子节点或最后一个子节点。但是，它们都是各自所属节点类型的第一个子节点和最后一个子节点。

7.1.5 getChildren 和 getChildrenBy 方法

通常，在处理 DOM 时，正在处理的节点的期望类型是元素节点。但是，document.body.

childNodes 方法会返回文本节点、注释节点以及所有其他类型的节点。这可能是一件麻烦的事情，因为在处理最终的集合时还需要额外的步骤来过滤元素节点之外的所有节点。但是，getChildren 方法会针对元素节点预先过滤结果。它还会返回一个数组，这是一种重要的行为，因为原生的 DOM 方法 childNodes 不会这样做。该方法看似会这样做，但是它实际上返回一个 NodeList，该 Nodelist 类似于一个数组，但是不具备数组的几个方法。因此，尽管可以通过 for 循环遍历该 NodeList，但是不能对其执行相应的操作。

不仅 getChildren 方法会针对元素节点进行过滤，而且它的相关方法 getChildrenBy 也可以通过一个用户自定义的 Boolean 函数来执行进一步的过滤。

```
<html>
    <head>
        <title> getChildrenBy </title>
    </head>
    <body>
        <p id="foo">
            Lorem ipsum dolor sit <em> amet </em> , consectetuer adipiscing elit.
            <a href="/vivamus/"> Vivamus </a> sed nulla. <em> Donec </em> vitae
            pede. <strong> Nunc </strong> dignissim rutrum nisi.
        </p>
        <script src="yahoo-dom-event.js"> </script>
        <script>
            var YD = YAHOO.util.Dom,
                YX = YAHOO.example;
            YX.isEm = function (el) {
                if (el.nodeName.toUpperCase() === "EM") {
                    return true;
                }
                return false;
            };
            YX.init = function () {
                var foo = YD.get("foo");
                var ems = YD.getChildrenBy(foo, YX.isEm);
                var msg = "";
                for (var i = 0; ems[i]; i += 1) {
                    var em = ems[i];
                    msg += "<em>"+ (em.innerText || em.textContent) + " </em> , ";
                }
                msg = msg.substring(0, msg.length - 2);
                alert("The following elements are emphasized: " + msg);
            }();
        </script>
    </body>
</html>
```

上面的示例获取 ID 为 foo 的段落元素的所有 em 子元素。isEm 函数可用作 Boolean 过滤器，它接收一个元素作为参数，检查它的节点名称是否为“EM”，然后返回 true 或 false。一旦找到所有 em 元素，就会显示一条警告消息来列出匹配的元素(参见图 7-1)。

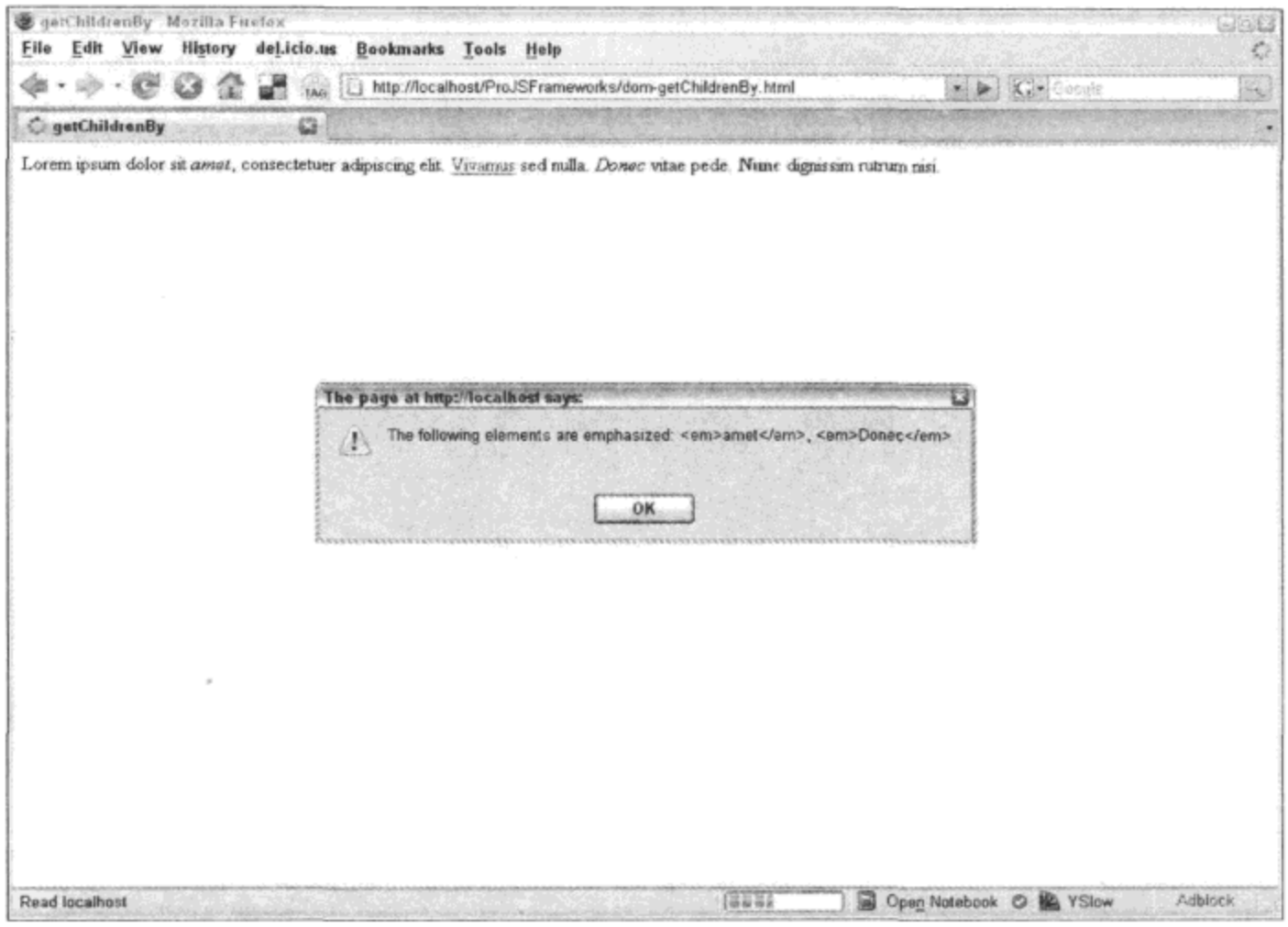


图　7-1

7.1.6　getElementsBy 方法

YUI DOM 集合还提供了一个 getElementsBy 方法，它为执行自定义执行了很大的便利。getElementsBy 方法并不限制特定类型的过滤(也就是“按照类名”、“按照 ID”)，而是将过滤条件交给开发人员确定。getElementsBy 方法接收的第一个参数是一个 Boolean 函数，将待过滤的元素传给这个 Boolean 函数进行测试。如果该元素通过测试(即该函数返回 true)，那么它将这个元素存储到最终返回的元素数组中。如果该元素未能通过测试，那么它就会移向下一个元素。此外，与 getElementsByClassName 方法类似，可以指定一种节点类型和一个根节点作为起点。最后，可以传入一个函数，该函数会应用于通过 Boolean 函数测试的每个节点。

```
<html>
    <head>
        <title> getElementsBy </title>
    </head>
    <body>
        <ul id="nav">
            <li> <a href="/"> Home </a> </li>
            <li>
                <a href="/products/"> Products </a>
                <ul>
                    <li> <a href="/products/widget/"> Widget </a> </li>
```

```
                <li> <a href="/products/gadget/"> Gadget </a> </li>
                <li> <a href="/products/whatzit/"> Whatzit </a> </li>
            </ul>
        </li>
        <li>
            <a href="/partners/"> Partners </a>
            <ul>
                <li> <a href="http://www.widgetsinc.com/"> Widgets Inc </a> </li>
                <li> <a href="http://www.gadgetsinc.com/"> Gadgets Inc </a> </li>
                <li> <a href="http://www.whatzitsinc.com/"> Whatzits Inc </a> </li>
            </ul>
        </li>
        <li> <a href="/about/"> About Us </a> </li>
        <li> <a href="/contact/"> Contact </a> </li>
    </ul>
    <script src="yahoo-dom-event.js"> </script>
    <script>
        var YD = YAHOO.util.Dom,
            YX = YAHOO.example;
        YX.isExternal = function (el) {
            if (el.href.substring(0, 4) === "http") {
                return true;
            }
            return false;
        };
        YX.makePopup = function (el) {
            el.onclick = function () {
                window.open(el.href);
                return false;
            };
        };
        YX.externalLinks = YD.getElementsBy(
            YX.isExternal, "a", "nav", YX.makePopup);
    </script>
</body>
</html>
```

上面的示例将函数 isExternal 应用于元素内 ID 为 nav 的所有锚点节点。然后将函数 makePopup 依次应用于通过这个测试的每个元素。这段代码的运行结果不言自明：其 href 特性以“http”开头的任何链接都会在新窗口中打开。

7.1.7　getAncestorByTagName 方法

有时候，充当容器的父元素需要受到在它的一个子元素上执行的操作的影响。但是麻烦的地方在于，子元素并不知道容器距离自己有多远，因此无法只是使用 parentNode 指针来指向该容器。

```
var parent = this.parentNode;
```

实际上，这里需要的操作是沿着 DOM 向上遍历，直到找到正在查找的节点。这正是 getAncestor 系列方法发挥作用的地方。getAncestorByTagName 方法从一个给定的节点开始向上沿着 DOM 遍历，并找到具有指定标记名称的父节点(即祖先节点)。

```
<html>
    <head>
        <title> getAncestorByTagName </title>
    </head>
    <body>
        <ul id="nav">
            <li> <a href="/" id="home"> Home </a> </li>
            <li> <a href="/contact/" id="contact"> Contact </a> </li>
            <li> <a href="/about/" id="about"> About Us </a> </li>
        </ul>
        <script src="yahoo-dom-event.js"> </script>
        <script>
            var YD = YAHOO.util.Dom,
                YX = YAHOO.example;
            YX.contact = document.getElementById("contact");
            YX.contact.onmouseover = function () {
                var ancestor = YD.getAncestorByTagName(this, "ul");
                if (ancestor) {
                    ancestor.style.border = "solid 1px red";
                }
            };
            YX.contact.onmouseout = function () {
                var ancestor = YD.getAncestorByTagName(this, "ul");
                if (ancestor) {
                    ancestor.style.border = "none";
                }
            };
        </script>
    </body>
</html>
```

在这个示例中，ID 为 contact 的锚点节点有两个事件处理程序，分别是 onmouseover(参见图 7-2)和 onmouseout(参见图 7-3)，它们均调用 getAncestorByTagName 方法，并将该锚点节点传入作为起点。它们还规定应该在遇到第一个 UL 父元素时停止遍历。一旦找到该元素(即祖先节点)，就通过 style 对象来修改它的边框。

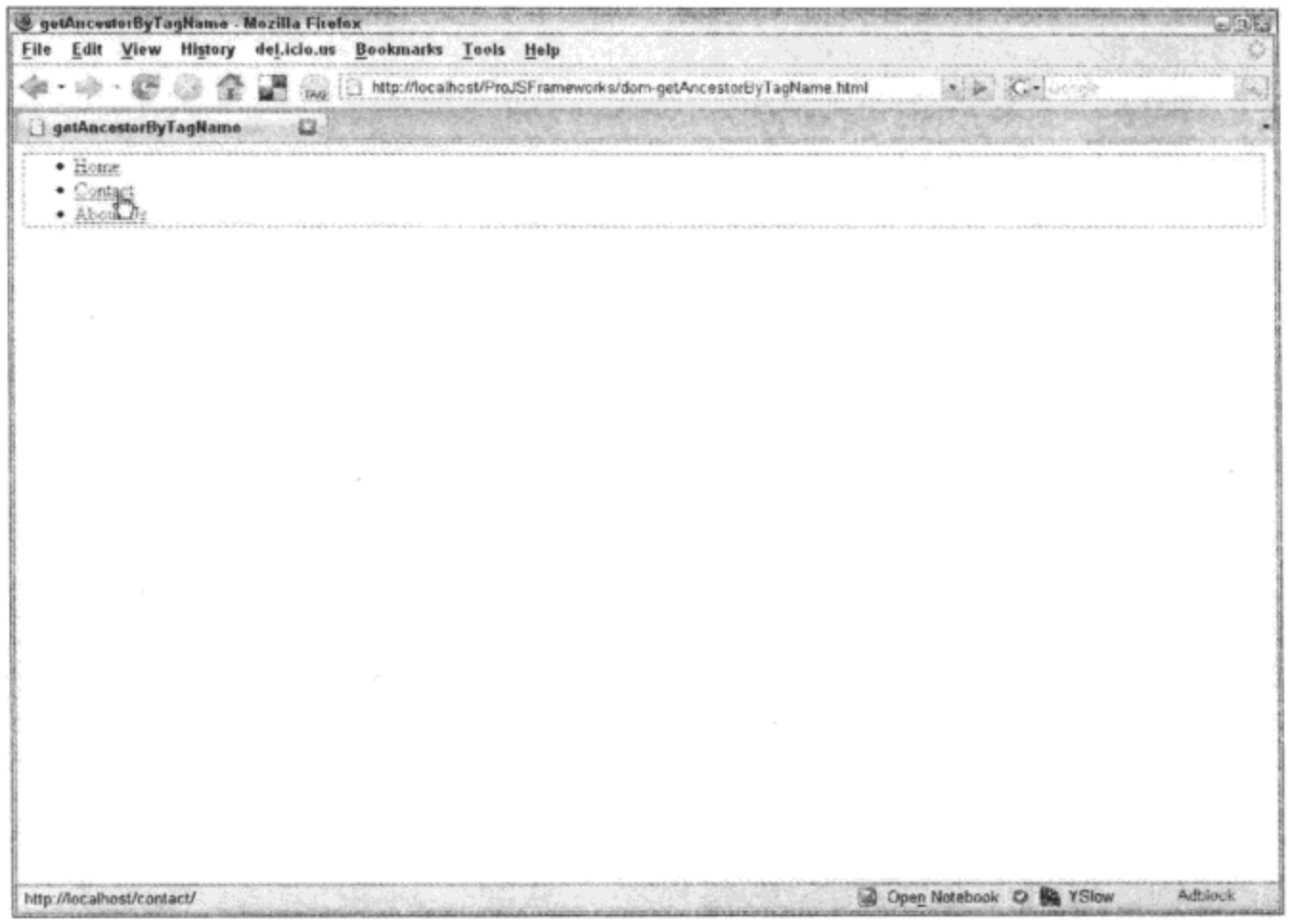

getAncestorByTagName - Mozilla Firefox
File Edit View History del.icio.us Bookmarks Tools Help
http://localhost/ProJSFrameworks/dom-getAncestorByTagName.html
getAncestorByTagName
Home
Contact
About Us
http://localhost/contact/
Open Notebook
YSlow
Adblock

图　7-2

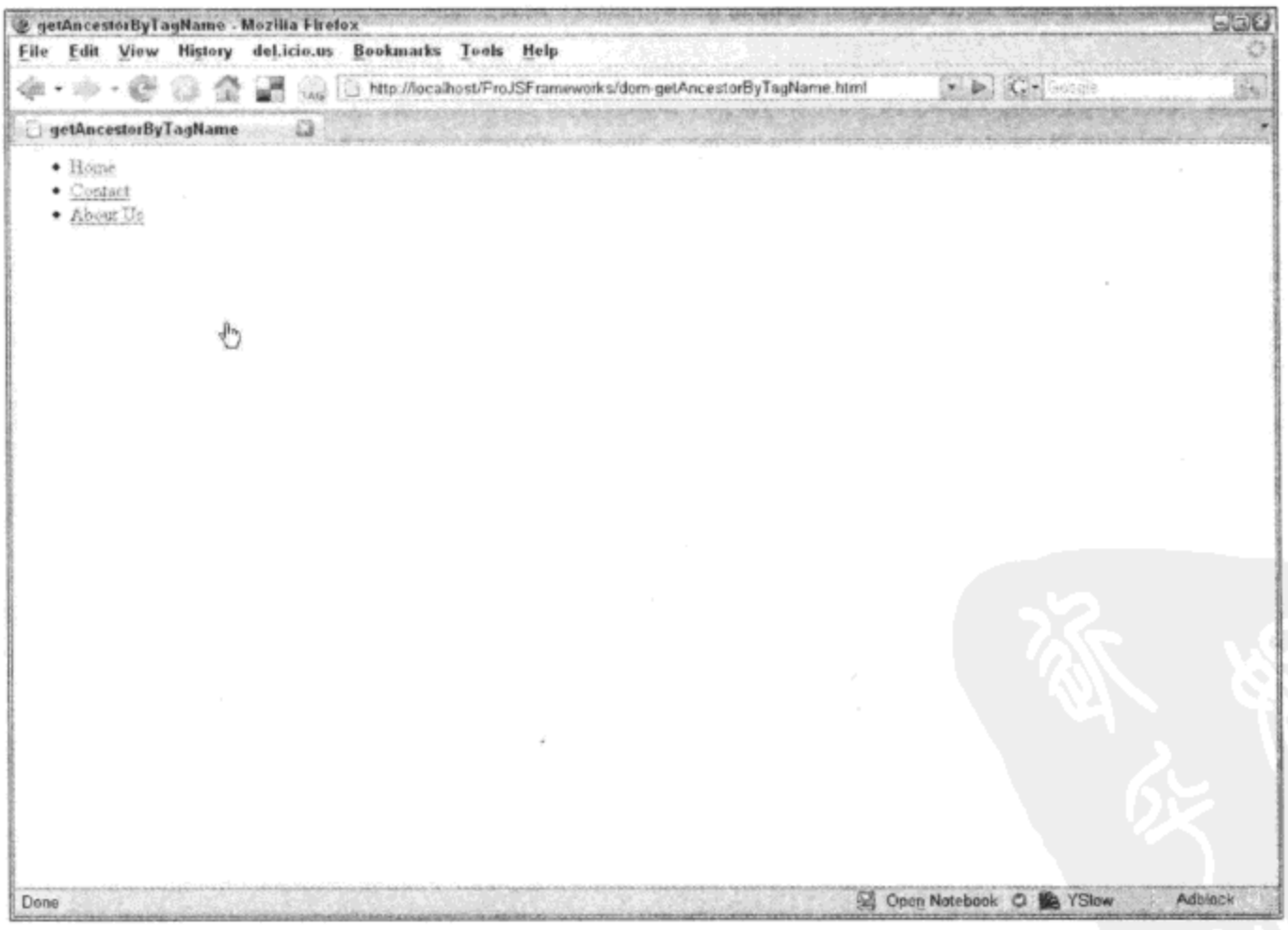

getAncestorByTagName - Mozilla Firefox
File Edit View History del.icio.us Bookmarks Tools Help
http://localhost/ProJSFrameworks/dom-getAncestorByTagName.html
getAncestorByTagName
Home
Contact
About Us
Done
Open Notebook
YSlow
Adblock

图　7-3

7.1.8 getAncestorByClassName 方法

类似地，getAncestorByClassName 方法从一个给定的节点开始，沿着 DOM 向上搜索具有给定类名的父元素。

```
<html>
    <head>
        <title> getAncestorByClassName </title>
        <style type="text/css">
            .main {
                border: solid 1px white;
            }
        </style>
    </head>
    <body>
        <div class="main">
            <div class="header">
            <ul id="nav">
                <li> <a href="/" id="home"> Home </a> </li>
                <li> <a href="/contact/" id="contact"> Contact </a> </li>
                <li> <a href="/about/" id="about"> About Us </a> </li>
            </ul>
        </div>
        <div class="body">
            <p> The page's main content block goes here </p>
        </div>
        <div class="footer">
            <p> <small> The page's footer goes here </small> </p>
        </div>
    </div>
    <script src="yahoo-dom-event.js"> </script>
    <script>
        var YD = YAHOO.util.Dom,
            YX = YAHOO.example;
        YX.contact = document.getElementById("contact");
        YX.contact.onmouseover = function () {
            var ancestor = YD.getAncestorByClassName(this, "main");
            if (ancestor) {
                ancestor.style.border = "solid 1px red";
            }
        };
        YX.contact.onmouseout = function () {
            var ancestor = YD.getAncestorByClassName(this, "main");
            if (ancestor) {
                ancestor.style.border = "solid 1px white";
            }
        };
    </script>
  </body>
</html>
```

这是一个与 getAncestorByTagName 类似的示例。但是在该示例中，当光标悬停在 contact 链接上时，就会把最外围的、ID 为 main 的 div 节点作为目标，并通过 style 对象修改它的边框。

7.1.9　getAncestorBy 方法

最后，使用 getAncestorBy 方法可以按照任何能够想到的条件来定位祖先节点，只需要将该条件编写进一个函数即可。该函数需要接收一个元素作为参数，然后针对该元素进行计算，并返回 true 或 false。因此，在 getAncestorBy 方法从它的起点开始向上遍历 DOM 的过程中，它会将遇到的每个父元素传给这个 Boolean 函数进行计算。如果该函数返回 false，它就会继续移向链式结构中的下一个元素。如果该函数返回 true，它就会停止遍历并将该元素返回给它的调用程序。

```
<html>
    <head>
        <title> getAncestorBy </title>
    </head>
    <body>
        <div class="main">
            <div class="header">
            <ul id="nav">
                <li> <a href="/" id="home"> Home </a> </li>
                <li> <a href="/contact/" id="contact"> Contact </a> </li>
                <li> <a href="/about/" id="about"> About Us </a> </li>
            </ul>
        </div>
        <div class="body">
            <p> The page's main content block goes here </p>
        </div>
        <div class="footer">
            <p> <small> The page's footer goes here </small> </p>
        </div>
</div>
<script src="yahoo-dom-event.js"> </script>
<script>
    var YD = YAHOO.util.Dom,
        YX = YAHOO.example;
    YX.hasBGColor = function (el) {
        if (el.nodeName.toUpperCase() === "DIV" & &
            el.style.backgroundColor) {
            return true;
        }
        return false;
    };
    YX.hasNoBGColor = function (el) {
        if (el.nodeName.toUpperCase() === "DIV" & &
            !el.style.backgroundColor) {
            return true;
        }
        return false;
};
```

```
        YX.contact = document.getElementById("contact");
        YX.contact.onmouseover = function () {
             var ancestor = YD.getAncestorBy(this, YX.hasNoBGColor);
             if (ancestor) {
                    ancestor.style.backgroundColor = "red";
             }
        };
        YX.contact.onmouseout = function () {
            var ancestor = YD.getAncestorBy(this, YX.hasBGColor);
            if (ancestor) {
                   ancestor.style.backgroundColor = "";
            }
        };
    </script>
  </body>
</html>
```

在这里，光标在 contact 链接上悬停或者离开时都会调用 getAncestorBy 方法，并分别传入 Boolean 函数 hasNoBGColor 和 hasBGColor。hasNoBGColor 函数查找第一个没有设置背景色的 div 祖先节点。然后，onmouseover 事件处理程序将该节点的背景色设为红色。类似地，hasBGColor 函数查找第一个设置了背景色的 div 祖先节点。然后，onmouseout 事件处理程序将该节点的背景色清除。

7.1.10 Element 实用工具

YUI Element 实用工具是 HTMLElement 的包装器。比较值得注意的地方在于，它允许延迟分配事件监听程序。它的实现方法是，监听何时把 HTMLElement 添加到 DOM，只有在这个时候才尝试将事件附加到这些对象上。当 HTMLElement 变成可用时，就会触发 contentReady 事件。最终，它简化了属性的设置和获取。

要想使用 Element 实用工具，需要像下面这样实例化 Element 对象：

```
var el = new YAHOO.util.Element("foo");
```

在使用上面示例中实例化的对象时，可以把一个事件处理程序添加到 el 元素，即使 ID 为 foo 的 HTMLElement 尚未添加到 DOM 中。在下面的示例中，当 ID 为 foo 的元素可用时，就会触发一个警告框。然后，它将一个 mouseover 事件处理程序附加到该元素，该处理程序负责把元素的文本颜色修改为红色。触发所有这些行为的就是一个两秒的定时器，它向 DOM 中插入一个 ID 为 foo 的 div 元素(参见图 7-4)。

```
<html>
     <head>
        <title> YUI Element Utility </title>
     </head>
     <body>
          <script src="yahoo-dom-event.js"> </script>
          <script src="element-min.js"> </script>
          <script>
```

```
            var YE = YAHOO.util.Event,
                YEl = YAHOO.util.Element,
                YX = YAHOO.example;
            YX.el = new YEl("foo");
            YX.el.on("contentReady", function () {
                alert("foo is here!");
            });
            YX.el.on("mouseover", function () {
                this.setStyle("color", "red");
            });
            setTimeout(function () {
                var foo = document.createElement("div");
                foo.id = "foo";
                foo.appendChild(document.createTextNode("foo"));
                document.body.appendChild(foo);
            }, 2000);
        </script>
    </body>
</html>
```

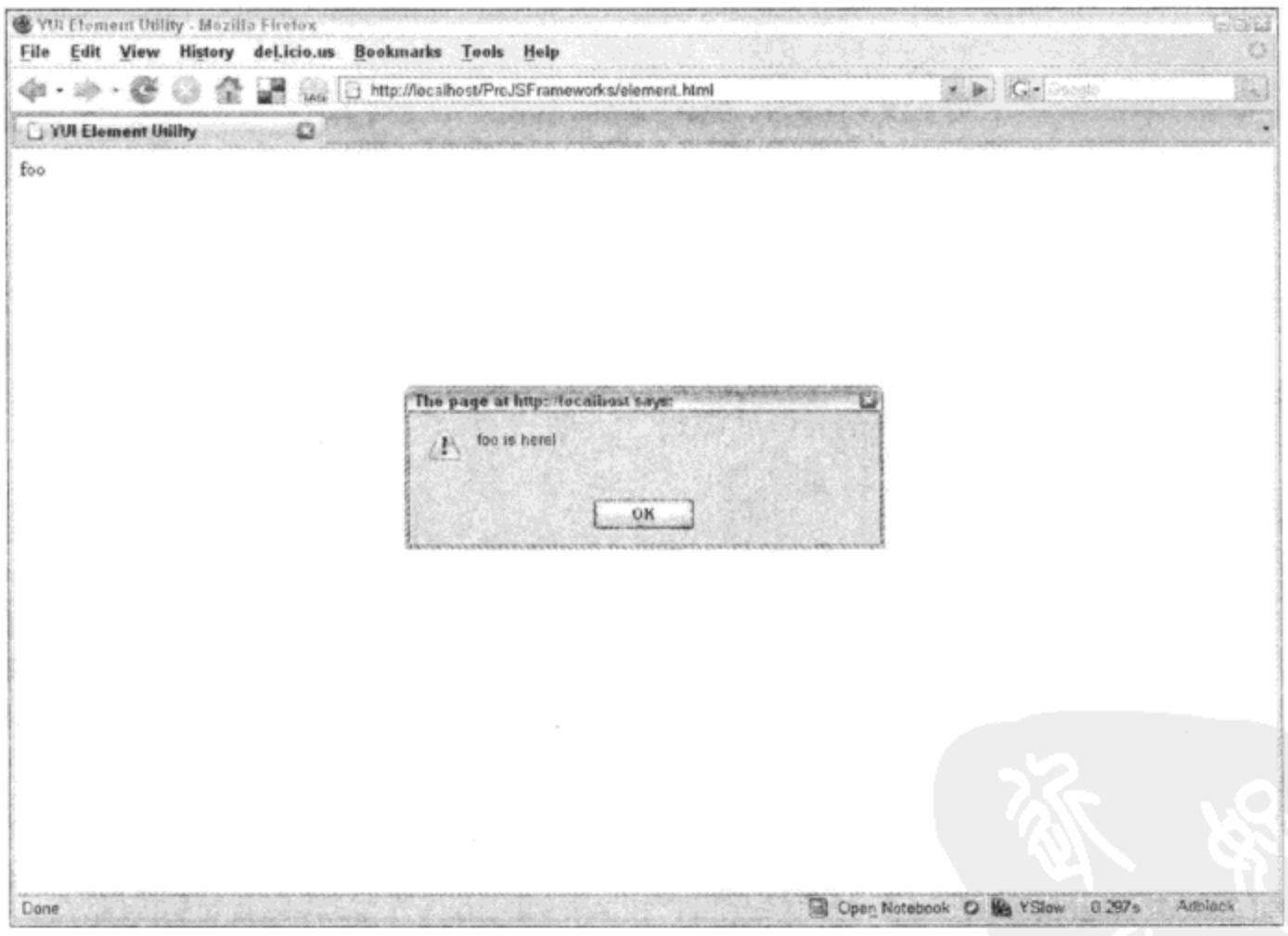


图　7-4

7.1.11　Selector 实用工具

有时候，使用 DOM 方法获取 HTMLElement 可能显得比较笨拙，常常需要几行代码来过滤位于特定节点层次中或者具有特定属性的某个元素。但是，如果熟悉 CSS 选择器语法，就会知道只需要一行简单的代码即可完成该操作。

例如，.foo 可以定位所有具有 CSS 类名 foo 的元素，而#foo .bar 则可以定位具有 CSS 类名 bar 的所有元素，并且这些元素是 ID 为 foo 的元素的子元素。

根据这个思想，对于下面的 JavaScript 代码块：

```
<html>
    <head>
        <title> YUI Selector Utility </title>
    </head>
    <body>
        <h1> YUI Selector Utility </h1>
        <div id="foo">
            <ul>
                <li class="first"> This is the first list item </li>
                <li class="second"> Here's the second </li>
                <li class="third"> And this one's the third </li>
            </ul>
        </div>
        <script>
            var foo = document.getElementById("foo");
            var listItems = foo.getElementsByTagName("li");
            for (var i = 0; listItems[i]; i += 1) {
                if (listItems[i].className === "second") {
                    listItems[i].style.color = "red";
                }
            }
        </script>
    </body>
</html>
```

使用 Selector 实用工具就可以方便地将其编写成如下的代码：

```
<script src="yahoo-dom-event.js"> </script>
<script src="selector-min.js"> </script>
<script>
    var YS = YAHOO.util.Selector,
        YX = YAHOO.example;
    YX.second = YS.query(".second", null, true);
    YX.second.style.color = "red";
</script>
```

YUI Selector 实用工具采用了 CSS 选择器语法，返回一组匹配的元素。在上面的示例中，传入一个附加的 Boolean 参数来告诉引擎只返回它找到的第一个元素。这样就可以避免在一个已经知道只含有一项的数组中选择第一项，这是一个多余的步骤。第二个参数是可以传入的起始节点 ID 或引用，这就允许通过减少需要检查的节点数量实现更好的性能。可以将第二个参数设置为 null 或 undefined，这样它就默认为 Selector.document，而 Selector.document 实际上就是指向文档自身的引用，但是不能将其作为参数传入，因为代码首先检查传入的值是否具有有效的标记名称(document 就无效)。

7.2　操作内容

仅仅从 DOM 中检索元素并没有什么用处，除非对这些元素进行一些处理。对于移动元素、修改元素或者只是添加新的元素这样的操作，也同样可以通过 YUI 的 DOM 集合来完成。

7.2.1　insertBefore 方法

使用 DOM 提供的 insertBefore 方法时，不仅要求给出一个新节点和一个引用节点，而且还需要在相同引用节点的父节点处调用该方法。

```
var newNode = refNode.parentNode.insertBefore(newNode, refNode);
```

YUI 的 insertBefore 方法不需要两次涉及引用节点，而且允许两个参数都是 ID 字符串类型或实际的节点引用。如果现有的参数只属于其中一种类型，那么这就会使编码变得简单一些。与原生 DOM 方法类似，YUI 的 insertBefore 方法也返回一个指向新节点的引用。

```
var newNode = YAHOO.util.Dom.insertBefore(newNode, refNode);
```

7.2.2　insertAfter 方法

虽然 DOM 提供了 insertBefore 方法，但是没有提供 insertAfter 方法。要在现有的节点之后插入一个新节点，就需要编写额外的代码。

```
var newNode = refNode.parentNode.insertBefore(newNode, refNode.nextSibling);
```

如果引用节点有一个 nextSibling(并非总是这种情况)，那么这段代码可以工作。因此，YUI 将这种额外的检查和引用全部包装进 insertAfter 方法中。

```
var newNode = YAHOO.util.Dom.insertAfter(newNode, refNode);
```

7.2.3　处理类名

在如今的复杂 Web 应用程序中，节点只有一个 CSS 类名的情形已经成为历史。下面是一个简单的示例：

```
<ul id="nav">
    <li class="home selected hover"> <a href="/index.html"> Home </a> </li>
    <! — More list items here -->
</ul>
```

上面的示例演示了一个元素如何能够拥有多个 CSS 类名，并且每个 CSS 类名都是有效的和必需的。然而，W3C DOM 规范并没有提供用于添加、替换和删除选择的 CSS 类名的原生方法，唯一用于处理 CSS 类名的方式就是通过赋值。

```
el.className = "home";
```

我们可以方便地添加另一个 CSS 类名。

```
el.className += " selected";
```

但是，无法知道某个 CSS 类名是否已经存在。YUI 的 DOM 集合提供了一组方法用于安全地操作 CSS 类名。而且，这些方法均接受元素、元素 ID 或元素数组作为待处理对象。因此，如果有一整批的元素都需要在其上执行某种 CSS 类名操作，那么可以将整个数组直接作为 el 参数传入。

1. hasClass 方法

检查一个元素是否具有某个特定的 CSS 类名并不像看起来那么简单。元素并非总是只有一个 CSS 类名：

```
<li class="home selected hover">
```

上面的代码使得类似于如下的检查变得不确定：

```
if (el.className === "selected") { // returns false
```

该元素的 CSS 类名之一可能就是 selected。但是，在这里它有多个 CSS 类名，这使得该 if 语句返回 false。检查 indexOf 也同样可能会变得不确定：

```
if(el.className.indexOf("select") !== -1) {
```

这不可能成为检查某个 CSS 类名是否存在的可靠方式，因为即使部分匹配，它也会返回 true。即使这个元素具有 CSS 类名 selected、selection 或 selectable，检查 select 也会返回 true。这里需要一种算法将 className 字符串划分成独立的 CSS 类名并检查每个 CSS 类名。这就是 hasClass 方法完成的工作。

```
if (YAHOO.util.Dom.hasClass(el, "select")) {
```

只有这个元素的 className 特性中出现完整的单词 select 时，这行代码才会返回 true。

2. addClass 方法

前面曾经提到，可以方便地添加 CSS 类名：

```
el.className += " hover";
```

但是麻烦的地方在于，无法知道刚才分配的 CSS 类名是否已经存在，而且 className 特性是字符串，因此可能只是连接该 CSS 类名。因此，上面的代码可能导致如下的结果：

```
<li class="hover hover">
```

在试图移除 CSS 类名 hover 时，这可能会导致程序错误，因为这两个 CSS 类名中只有一个会被删除，而该元素还剩下一个不希望具有的 CSS 类名 hover。YUI 的 addClass 方法首先进行检查以确保正在添加的 CSS 类名并不存在。如果该 CSS 类名并不存在，该方法就会将其添加进去，并返回 true。如果该 CSS 类名已经存在，该方法就会返回 false 而不添加任何 CSS 类名。

```
YAHOO.util.Dom.addClass(el, "hover");
```

3. removeClass 方法

将一个 CSS 类名从元素中删除可能是处理类名时比较困难的任务之一。这里必须进行实际的字符串处理。不仅要找到表示该 CSS 类名的整个单词，而且需要重新构造 className 属性的内容以减去要删除的 CSS 类名。一旦完成了这些操作，就需要将这个新的字符串重新分配给 className 属性，有效地使用新字符串覆盖旧字符串。YUI 将整个过程缩减为一个简单的方法 removeClass。

```
YAHOO.util.Dom.removeClass(el, "hover");
```

4. replaceClass 方法

有时候并不需要将 CSS 类名删除，而是只要将其替换。这正是 YUI 的 replaceClass 方法发挥作用的地方。如果没有找到需要替换的 CSS 类名，该方法就会将新的 CSS 类名添加进去并返回 true。如果该方法确实找到要替换的 CSS 类名，它就会将其替换并返回 true。如果某个元素刚好有多个相同的 CSS 类名，那么该方法会将它们全部替换成新的值。

```
YAHOO.util.Dom.replaceClass(el, "open", "closed");
```

7.2.4　setStyle 方法

设置 DOM 元素的样式值相对简单。但是，Internet Explorer 6 通过特殊的 filter 规则来管理透明度，而 Firefox、Opera 和 Safari 并不是这样操作。此外，浮动元素的使用也有些麻烦，这是因为 float 属于保留字。因此，Internet Explorer 将其称为 styleFloat，而其他浏览器则将其称为 cssFloat。这些特性不仅令人讨厌，而且可能会造成程序错误。必须记住并容忍它们可能也是一件痛苦的事情。此外，CSS 规则名称语法要求使用连字符分隔单词，但是 JavaScript 使用驼峰式大小写(camel case)的形式。

```
<style type="text/css">
#foo {
     font-size: 2em;
     font-family: arial;
     float: right;
}
</style>

<script>
     foo.style.fontSize = "2em";
     foo.style.fontFamily = "arial";
     foo.style.cssFloat = "right";
</script>
```

YUI 的 DOM 集合方法 getStyle 和 setStyle 将这些问题全部规范化。

```
<script>
   YAHOO.util.Dom.setStyle(foo, "font-size", "2em");
   YAHOO.util.Dom.setStyle(foo, "font-family", "arial");
   YAHOO.util.Dom.setStyle(foo, "float", "right");
   YAHOO.util.Dom.setStyle(foo, "opacity", "0.5");
```

```
</script>
```

注意：

一定要将这段脚本放在DOM中ID为foo的元素之后，最好是位于结束标记</body>之前。否则就无法找到该元素，并且这段代码不会运行。

7.2.5 getStyle方法

获取一个元素的样式可能比较棘手，这是因为JavaScript的style对象只保存通过JavaScript设置的值。换句话说就是，通过常规CSS代码设置的样式值不会由style对象获取。

```
<script>
    el.style.width = "100px";
    alert(el.style.width); // returns "100px" because it was just set;
    alert(el.style.height); // returns "";
</script>
```

获取元素的样式值(不管是否是在style对象中设置)的方式是使用getComputedStyle方法，但是该方式不适用于Internet Explorer。在这种情况下，要使用的方法是currentStyle，但是该方法没有考虑YUI方法setStyle中所涵盖的全部规范化操作。同样，YUI的DOM集合方法getStyle为解决这个问题提供了一种跨浏览器解决方案。

```
<script>
    YAHOO.util.Dom.getStyle(foo, "font-size");
</script>
```

7.2.6 setXY方法

设置元素的x和y坐标并非总是简单的事情，但是使用setXY方法确实使其变得简单。在大部分情况下，该方法运行良好。但是，如果元素深度嵌套在某个反常的布局内部，那么setXY方法不会输出想要的结果。然而，它确实会尝试第二次运行，并通过第三个Boolean参数来允许控制是否再次尝试执行。在大多数情况下，该方法还是相当可靠的。

```
// set foo's x to 10px and y to 100px and don't retry
YAHOO.util.Dom.setXY(foo, [10, 100], true);
```

还有单独的setX和setY方法，它们的语法与setXY方法相同，但x和y值参数是作为数值(而非数值)传入的。因此，可以将上面的示例改写为：

```
YAHOO.util.Dom.setX(foo, 10);
YAHOO.util.Dom.setY(foo, 100);
```

YUI 3中的新增功能

除了新的名称空间(现在是YUI而不是YAHOO)之外，YUI 3还引入了一种全新的DOM遍历和操作方式。它将DOM节点看作YUI节点对象。节点对象具有方法和属性，可用于更加方便地访问DOM并与其交互。一个重大的变化就是YUI使用了新的get方法，它与YUI 2.x版本的selector方法类似。下面这个示例使用了get方法(请注意如何在新的脚本依赖项加载器use中包装它)。

```
YUI().use('node', function(Y) {
    var node1 = Y.get('#main');
    var node2 = Y.get(document.body);
});
```

来源：这个代码示例来自 YUI 3 Node 页面。

7.3　本章小结

通过学习本章可以了解到，DOM 的处理存在不一致性问题，而且浏览器制造商追求自行解释和实现的倾向使得这个问题变得更加复杂。YUI 在弥合这些情况所造成的分歧方面取得了成功。

使用 YUI 实现对跨浏览器问题的规范化是一种很好的做法，而且出于一致性考虑，使用 YUI 也是非常明智的选择。浏览器在不断地成熟和进化。程序错误得以修复，新功能添加进来，又有更多的新程序错误被引入。例如，如果将来某个时候浏览器开始对当前尚不存在的 DOM 方法 getElementsByClassName 提供支持，那么 YUI 团队只需要将它们的代码替换成这个 DOM 方法自身即可，从而使得使用该库的项目可以无缝地过渡。否则，我们就要搜索整个项目中的不同代码片段来完成同样的修改。

第 8 章

处理跨浏览器事件

事件处理是现代 Web 开发的重要组成部分。但是，现代浏览器的 W3C DOM 事件模型的实现远远没有统一化，尤其是 Internet Explorer 处理事件的方式与市面上的其他主流浏览器相当不同。此外，就某些以性能为目标的事件处理程序(目前并不存在)而言，该规范相当薄弱，本章稍后将更多地讨论这一点。YUI Event Utility 不仅在规范化跨浏览器事件处理方面表现突出，而且它扩展了现有的事件模型，从而实现了一些采用其他方式不能实现的功能。更加引人注意的是 Event Utility 的事件处理程序，Event Utility 能够向一个元素添加和删除多个事件监听程序。

本章内容简介：

- 注册页面事件和元素准备就绪事件
- 向元素添加事件监听程序
- 处理键盘和鼠标输入
- 处理自定义事件
- 管理浏览器历史并修正后退按钮

8.1 注册页面事件和元素准备就绪事件

在 DOM 准备就绪之前，用于获取、设置和修改 DOM 元素的 JavaScript 函数将不能执行。如果此时调用这些函数，那么当它们试图对尚不存在的元素执行操作时就会抛出错误。

```
<html>
    <head>
        <title>Hello World - Broken</title>
        <script>
            var hello = document.getElementById("hello");
            var msg = hello.innerText || hello.textContent;
            alert(msg);
        </script>
    </head>
    <body>
```

```
        <p id="hello">Hello World</p>
    </body>
</html>
```

这段 JavaScript 代码将中断执行，这是因为它在 ID 为 hello 的元素创建之前就试图访问它的属性。在这种情况下，getElementById 方法返回 null，表明该元素没有任何属性。之所以发生这种情况，是因为 JavaScript 代码在读取的同时就会执行。绕开这个问题的做法就是将代码包装进一个函数中并在以后(该元素准备就绪之后)调用该函数。传统上，这是通过 window 对象的 onload 事件处理程序来处理的。

```
<html>
    <head>
        <title>Hello World</title>
        <script>
            window.onload = function () {
                var hello = document.getElementById("hello");
                var msg = hello.innerText || hello.textContent;
                alert(msg);
            }
        </script>
    </head>
    <body>
        <p id="hello">Hello World</p>
    </body>
</html>
```

在这个示例中，这种处理方式可以很好地工作。但是在更大型的页面上，在触发 onload 事件之前会有显著的延迟。这是因为在页面的依赖项全部完成加载之后才会将该页面视为加载完毕。这包括页面中的所有图像(还有其他内容)。但是在大多数时候，待执行的 JavaScript 代码并不会用到这些图像，它所需要的只是 DOM 准备就绪，也就是说，它只需要某些元素准备就绪即可。但是，没有原生的 W3C DOM 方法可用于在 DOM 准备就绪时触发该事件。

关于术语的注意事项

在本节中，当使用术语事件时，它指的是属于某个 DOM 元素的“关注时刻”，可以将一个处理程序附加到该事件。事件处理程序又称为监听程序(或者称为回调函数，我们会交换地使用这些术语)，它实际上是一个函数，每当它所附加的事件触发时，都会执行这个函数。我们也会交换地使用术语附加和分配来描述事件与处理程序的关联。

8.1.1 onDOMReady 事件处理程序

下面讲解 onDOMReady 事件处理程序，一旦 DOM 准备就绪，就会引发这个 YUI Event Utility 的自定义事件处理程序。基于 Dean Edwards、John Resig 和 Matthias Miller 的工作，该事件处理程序检查并了解文档是否已经就绪，至少对于 Internet Explorer 和 WebKit 的早期版本(Apple 的 Safari 浏览器)是如此。Firefox、Opera 和 WebKit 的后期版本实际上有一个专有的事件会被触发，它名为 DOMContentLoaded，而 onDOMReady 事件处理程序正是利用了该事件。但是，Internet Explorer 并没有这样的专有事件可触发。而且在所有浏览器中，Internet Explorer 正是受益最大的一款浏览

器，这是因为在某些情况下，在 DOM 准备就绪之前就在 IE 中修改它可能会导致页面实际地终止，并从浏览器中弹出一条表明操作终止的消息(参见图 8-1)。这并非对用户友好的处理方式。

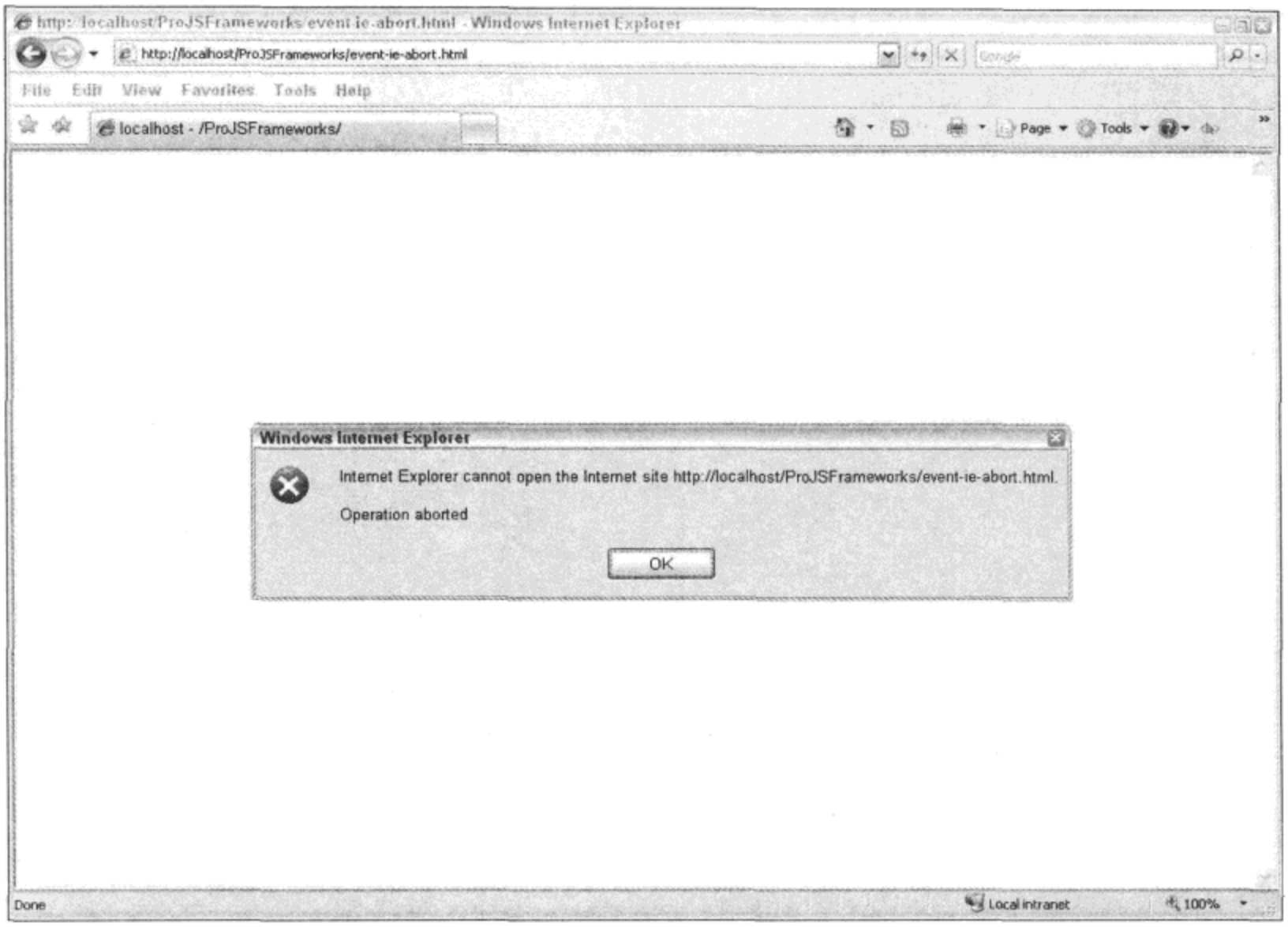


图　8-1

有些代码会导致 Internet Explorer 浏览器抛出“Operation aborted(操作终止)”消息，下面就是一个示例。出现这个问题的原因在于，如果 script 标记并不是 body 标记的直接子元素，那么 IE 就不能处理 body 标记的 appendChild 方法。

```
<html>
    <head>
        <title>Internet Explorer "Operation aborted"</title>
    </head>
    <body>
        <div>
            <script>
                var p = document.createElement("p");
                document.body.appendChild(p);
            </script>
        </div>
    </body>
</html>
```

通过将代码包装成一个匿名函数并将其传给 YUI 的 onDOMReady 事件处理程序，就可以方

便地解决这个问题。

```
<html>
    <head>
        <title>Internet Explorer "Operation aborted" - FIXED!</title>
    </head>
    <body>
        <div>
            <script src="yahoo-dom-event.js"></script>
            <script>
                var YE = YAHOO.util.Event;
                YE.onDOMReady(function () {
                    var p = document.createElement("p");
                    document.body.appendChild(p);
                });
            </script>
        </div>
    </body>
</html>
```

但是前面曾经提到，更为常见的问题是必须等待页面的所有依赖项全部下载完成。在含有大量图像的页面中，这一点尤其会成为问题。在下面这个示例中，在图像下载完成之前 JavaScript 代码需要等待很长时间。

```
<html>
    <head>
        <title>onDOMReady</title>
        <script src="yahoo-dom-event.js"></script>
        <script>
            var YE = YAHOO.util.Event;
            YE.onDOMReady(function () {
                var images = document.getElementsByTagName("img");
                alert("There will be " + images.length +
                    " images loaded on this page");
            });
        </script>
    </head>
    <body>
        <ol>
            <li><img src="image1.jpg" /></li>
            <li><img src="image2.jpg" /></li>
            <!-- Hundreds of really heavy images go here -->
        </ol>
    </body>
</html>
```

8.1.2 执行作用域和参数传递

通常情况下，事件处理程序的执行作用域是 window 对象，this 关键字引用的是其事件触发该处理程序的元素。下面的代码给出了一个示例。

```
var el = document.getElementById("foo");
el.onmouseover = function () {
    this.style.color = "red";
};
```

在这里，我们把一个匿名函数分配给元素的 onmouseover 事件。当触发该函数后，它通过 this 关键字来访问主调元素。然后，它通过该元素的 style 对象来修改其颜色。对于这样的简单示例来说，这是一种非常好的做法。但是在较为复杂的构造中，这种方式就会有问题。

```
<html>
    <head>
        <title>Execution Scope - Broken</title>
    </head>
    <body>
        <a href="/foo/" id="foo">Foo</a>
        <script>
            function DoSomething(id) {
                var foo = document.getElementById(id);
                this.bar = "baz";
                foo.onmouseover = function () {
                    alert(this.bar);
                };
            };
            var doSomething = new DoSomething("foo");
        </script>
    </body>
</html>
```

在这个示例中，我们在函数 DoSomething 中实例化一个对象，该对象有一个名为 bar 的属性，其值为 baz。还有一个 onmouseover 事件处理程序，在 ID 为 foo 的锚点元素上建立该事件处理程序，它修改 bar 属性的值。问题在于，事件处理程序的执行作用域并不是 doSomething 对象，而是 window 对象。因此，this 关键字指向的是调用该处理程序的元素而不是 doSomething。其结果就是，alert(this.bar);返回的值是 undefined，因为该锚点元素并没有 bar 属性(从这个属性中返回值)。这个事件处理程序所需要的就是修正它的执行作用域。换句话说就是，需要告诉它“在 doSomething 对象中运行，而不是在 window 对象中运行”。

Event Utility 的参数模式

YUI 的 Event Utility 函数(addListener、onAvailable、onContentReady 以及 onDOMReady)都允许将一个可选的数据对象传给回调函数。它们也都允许改变回调函数的执行作用域，这是通过名为 obj 和 override 的两个参数来完成的(在 onDOMReady 中，override 参数名为 scope，这是一种命名不一致的情况)。

```
YAHOO.util.Event.onDOMReady(callback, obj, scope/override);
```

这 3 个参数可以按照不同的方式使用以产生不同的结果：

(1) 如果只有数据对象，那么只把这个数据对象传给回调函数。

(2) 如果存在数据对象，而且 override 参数设置为 Boolean 值 true，那么该数据对象就成为回

调函数的执行作用域，而且它会传给回调函数。

(3) 如果存在数据对象，而且 override 参数也是一个对象，那么该数据对象就会传给回调函数，而 override 对象将成为该函数的执行作用域。

下面这个示例用来演示如何修正事件处理程序的执行作用域。

```
<html>
    <head>
        <title>Execution Scope - Fixed</title>
    </head>
    <body>
        <a href="/foo/" id="foo">Foo</a>
        <script src="yahoo-dom-event.js"></script>
        <script>
            var YE = YAHOO.util.Event,
                YX = YAHOO.example;
            YX.DoSomething = function (id) {
                var foo = document.getElementById(id);
                this.bar = "baz";
                YE.addListener(foo, "mouseover", function () {
                    alert(this.bar);
                }, this, true);
            };
            YX.doSomething = new YX.DoSomething("foo");
        </script>
    </body>
</html>
```

这个示例并没有使用一种直接的赋值技术，而是使用 YUI 的 addListener 函数(该函数的更多细节稍后将讨论)。第三个参数是回调函数，这是一个直接内联编码的匿名函数。第四个参数是数据对象，而第五个参数是 Boolean 值 true，它告诉 addListener 方法使这个数据对象成为该回调函数的执行作用域。因为 this 关键字引用的是这个数据对象，所以告诉该回调函数在 doSomething 对象的作用域内执行。这可以让 mouseover 事件处理程序访问 bar 变量的值。

8.1.3 onAvailable 函数

有时页面需要花费大量时间才能将它的内容全部加载完毕，否则它们的内容无法一次性加载完毕。在这些情况中，能够检测到某一块想要的内容何时变成可用就是一件非常重要的事情。如果具备这种能力，就可以让程序立即开始与元素交互，而无须等待页面加载甚至 DOM 准备就绪。YUI 的 onAvailable 函数在 DOM 构建期间就进行监视，观察它要关注的元素的状态。当它找到一个元素时，就会执行回调函数。但是，如果一旦页面加载完成就调用 onAvailable 函数，那么它监视 DOM 一段时间(可配置值，默认为 10 秒)之后就停止监视。下面是 onAvailable 函数的一个示例，它捕获在页面已经加载完成之后向 DOM 中插入新节点的操作。

```
<html>
    <head>
        <title>onAvailable</title>
    </head>
    <body>
```

```
        <p id="hello">Hello World!</p>
        <script src="yahoo-dom-event.js"></script>
        <script>
            var YD = YAHOO.util.Dom,
                YE = YAHOO.util.Event,
                YX = YAHOO.example;
            YX.insertNewContent = function () {
                var p = document.createElement("p");
                p.id = "how";
                p.appendChild(document.createTextNode("How's it going?"));
                YD.insertAfter(p, "hello");
            };
            YX.detected = function () {
                alert("New content detected!");
            };
            YE.onAvailable("how", YX.detected);
            setTimeout(YX.insertNewContent, 2000);
        </script>
    </body>
</html>
```

在这个示例中，当页面完成加载两秒钟之后，将一个新的包含文本“How's it going?”的段落元素插入到 DOM 中(参见图 8-2)。这就是 setTimeout 方法的作用。添加到 DOM 中的新段落元素的 ID 为 how。我们设置 onAvailable 函数来监听 ID 为 how 的元素，一旦发现这个元素就执行回调函数 detected。因此，当把新的段落元素添加到 DOM 中时，它就会捕获这个元素。

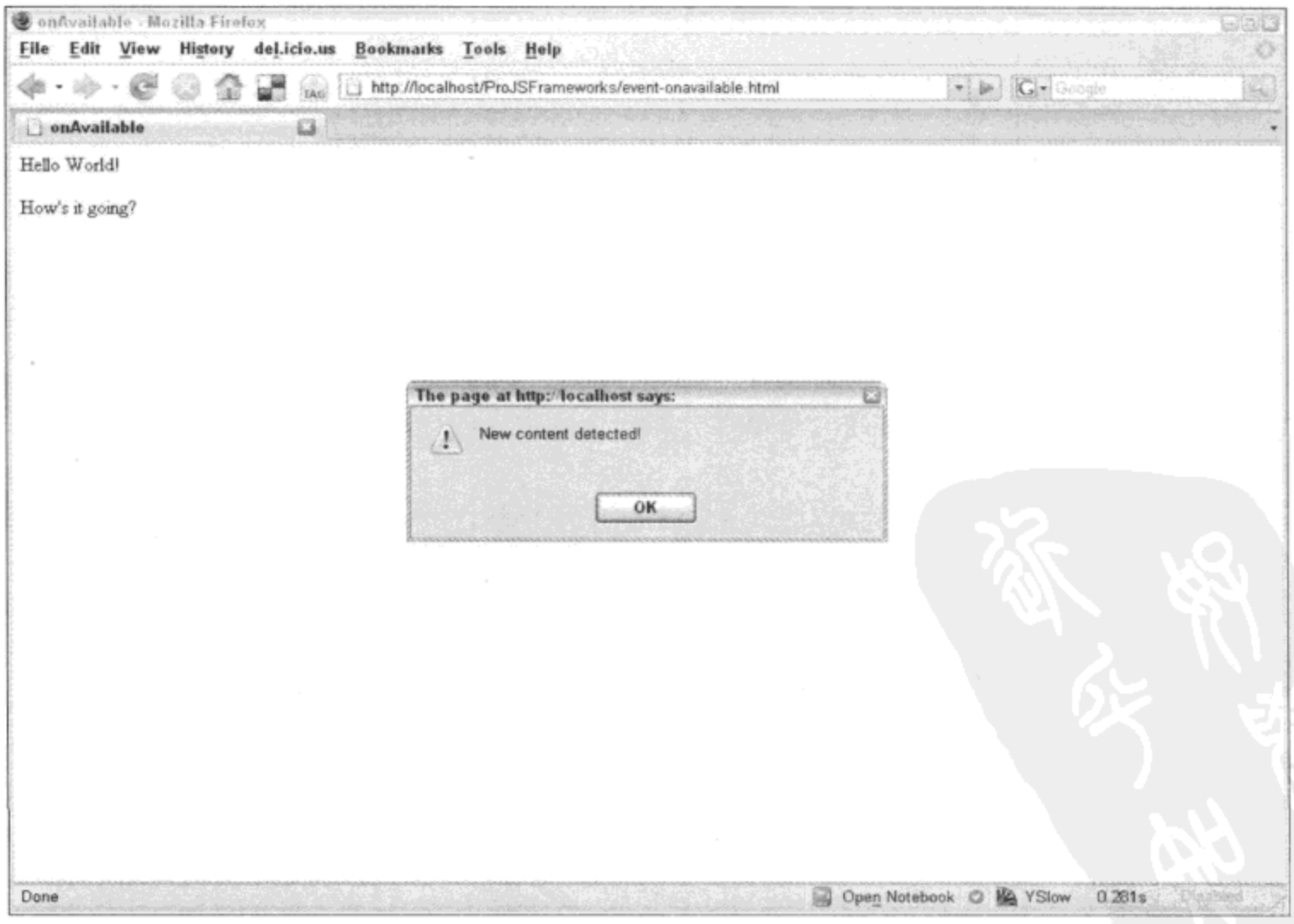


图　8-2

8.1.4 onContentReady函数

有时候，仅仅知道一个元素已经可用并不足够。有时候，需要在元素上执行的操作还会要求该元素的内容也要可用。onAvailable函数的第五个Boolean类型参数名为checkContent，如果将这个参数设置为true，那么onAvailable函数就会检查该元素的子节点准备就绪情况。它会检查该元素的兄弟节点，这是因为如果浏览器已经移向下一个兄弟节点，就可以推断出已经对当前节点(包括它的所有子节点)完成操作。onContentReady函数就是onAvailable函数将其checkContent参数永久设置为true的别名。

向元素添加事件监听程序是一件非常简单的事情，但是添加多个监听程序然后将其删除就要困难得多，至少很难跨浏览器一致地实现该操作。YUI利用下面给出的函数来解决这个难题。

8.1.5 on/addListener函数

向DOM元素添加一个事件处理程序是一件非常简单的事情，通过直接赋值即可。在下面的示例中，我们将一个匿名函数分配给某个元素的onclick事件。

```
var el = YAHOO.util.Dom.get("foo");
el.onclick = function () {
    // do something
};
```

类似地，可以像下面这样分配一个命名函数：

```
function doSomething() {
    // do something
};
var el = YAHOO.util.Dom.get("foo");
el.onclick = doSomething;
```

这两种赋值方法对于简单的事件处理而言可以很好地工作，但是很快就会出现问题。例如，一个需要分配事件的元素可能已经有一个分配给它的事件。第一个事件可能来自于另一位程序员，而且可能完全不知道现在这位开发人员要通过为其分配一个新的处理程序来覆盖它。这就是问题所在，因为标准事件模型只允许向一个事件分配一个函数。向一个事件添加多个事件处理程序是一件不可能完成的工作。

YUI的addListener函数可用来完成其名称所暗示的功能：向一个元素添加多个事件处理程序。

```
<html>
    <head>
        <title>addListener</title>
    </head>
    <body>
        <p>
            <a href="/dosomething/" id="foo">Do something</a>
        </p>
        <script src="yahoo-dom-event.js"></script>
        <script>
            var YE = YAHOO.util.Event,
```

```
            YX = YAHOO.example;
        YX.msg = "Doing something";
        YX.doSomething = function () {
            alert(YX.msg + "!");
        };
        YX.doSomethingElse = function () {
            alert(YX.msg + " else!");
        };
        YE.addListener("foo", "click", YX.doSomething);
        YE.addListener("foo", "click", YX.doSomethingElse);
    </script>
  </body>
</html>
```

注意：

在 IE 7 中，doSomethingElse 事件处理程序在 doSomething 之前运行。而在 Firefox 2 和 Safari 3.1 中，事件是按照添加它们的顺序触发的。

在这个示例中，我们将两个事件处理程序分配给 ID 为 foo 的元素的 onclick 事件。它们会按照分配时的顺序依次执行。与 YUI 的大多数函数和构造函数的情形一样，addListener 函数允许把字符串 ID、HTMLElement 或者这两种类型构成的数组作为它的第一个参数。第二个参数是处理程序(或监听程序)要附加的事件。这个函数并不关心该参数是否为有效的事件名称，不管其是否有效，它都会试着把回调函数附加到该事件。因此，程序员要负责确保自己提供的名称所对应的事件确实存在。换句话说就是，addListener 将采用单词 click 并试着将我们提供的回调函数附加到 onclick 事件。而如果传递的是单词 scream，那么这个函数会试着将回调函数附加到 onscream 事个把，很明显该事件并不存在。

注意，在这个示例中，事件会按照添加的顺序依次触发，但 Internet Explorer 是例外情况。因为 YUI 将 addListener 函数映射到 IE 的原生方法 attachEvent，所以事件实际触发的顺序是由 IE 来决定的。此外，由于这个原因，它们并不会按照先进先出的顺序触发。YUI 3 纠正了这个问题，它把事件处理程序包装起来并将它们看作自定义事件。换句话说就是，YUI 3 自行处理事件触发。

addListener 函数还允许传递任意的数据对象：

```
<html>
    <head>
        <title>addListener 2</title>
    </head>
    <body>
        <p>
            <a href="/johndoe/" id="jd">John Doe</a>
        </p>
        <script src="yahoo-dom-event.js"></script>
        <script>
            var YE = YAHOO.util.Event,
                YX = YAHOO.example;
            YX.Person = function (fname, lname) {
                this.fname = fname;
                this.lname = lname;
```

```
            this.sayFName = function (e, greeting) {
                alert(greeting.txt + this.fname);
            };
        };
        YX.johndoe = new YX.Person("John", "Doe");
        YX.init = function () {
            var greeting = {txt: "Hi, my name is "};
            YE.addListener(
                "jd", "mouseover", YX.johndoe.sayFName, greeting, YX.johndoe);
        }();
    </script>
</body>
</html>
```

这个示例演示可以将一个对象传给事件处理程序。首先，将一个 Person 对象实例化到变量 johndoe 中。然后，在 init 函数中建立一个事件处理程序，这样每当光标滑过 ID 为 jd 的锚点元素时，都会调用 johndoe 对象的 sayFName 方法。这个方法接收两个参数：第一个参数是事件对象，浏览器将该对象传给该回调函数(除 Internet Explorer 之外的所有浏览器)，而第二个参数是包含问候文本的自定义对象。

注意：

上面示例中的 init 函数是一个自调用函数，由位于结束大括号末尾的一对括号来实现自调用。一旦 JavaScript 分析器完成该函数的分析工作，它就会遇到这对括号，从而被告知立即执行该函数。

此外，YUI 为 addListener 函数提供了一个别名 on。这表示可以将下面这行代码：

```
YAHOO.util.Event.addListener("foo", "click", doSomething);
```

简化成如下代码：

```
YAHOO.util.Event.on("foo", "click", doSomething);
```

8.1.6 removeListener 函数

有时为了避免造成内存泄漏问题，需要把事件处理程序从 DOM 中删除。例如在 Internet Explorer 6 中，如果把一个事件处理程序分配给某个元素，然后销毁该元素，那么该事件处理程序仍然位于内存中。如果在 Web 应用程序上下文中多次执行这个操作，并且在此过程中用户并没有离开该页面，而是与该应用程序进行长时间的交互，那么这可能会导致大量的内存泄漏。因此，使用 YUI 函数 removeListener 移除事件处理程序是一种明智的做法。

```
<html>
    <head>
        <title>removeListener</title>
    </head>
    <body>
        <p>
            <a href="/dosomething/" id="foo">Do something</a>
```

```
        </p>
        <script src="yahoo-dom-event.js"></script>
        <script>
            var YE = YAHOO.util.Event,
                YX = YAHOO.example;
            YX.msg = "Doing something";
            YX.doSomething = function () {
                alert(YX.msg + "!");
            };
            YX.doSomethingElse = function () {
                alert(YX.msg + " else!");
            };
            YE.addListener("foo", "click", YX.doSomething);
            YE.addListener("foo", "click", YX.doSomethingElse);
            YE.removeListener("foo", "click", YX.doSomething);
        </script>
    </body>
</html>
```

这是与 addListener 函数相同的示例。将这两个处理程序中的第一个移除，这样就只有一个事件处理程序附加到 ID 为 foo 的锚点元素。如果有多个事件处理程序附加到一个元素，而且没有指定回调函数，那么 removeListener 函数将把所有属于指定类型(在这里就是“click”)的处理程序从该元素中移除。

```
<html>
    <head>
        <title>removeListener 2</title>
    </head>
    <body>
        <p>
            <a href="/dosomething/" id="foo">Do something</a>
        </p>
        <script src="yahoo-dom-event.js"></script>
        <script>
            var YE = YAHOO.util.Event,
                YX = YAHOO.example;
            YX.msg = "Doing something";
            YX.doSomething = function () {
                alert(YX.msg + "!");
            };
            YX.doSomethingElse = function () {
                alert(YX.msg + " else!");
            };
            YE.addListener("foo", "click", YX.doSomething);
            YE.addListener("foo", "click", YX.doSomethingElse);
            YE.removeListener("foo", "click");
        </script>
    </body>
</html>
```

在这个示例中，ID为foo的锚点元素最终将没有事件处理程序。

8.2 处理键盘和鼠标输入

有时候，YUI库的作用并不是解决跨浏览器问题，而是让开发工作变得更加轻松。例如，尽管在不同的浏览器中键盘处理支持相对标准，但是使用它完成有用的工作却需要编写庞大而笨拙的代码。在这里，YUI将这个过程缩减成Event Utility中始终可以找到的一个通用模式，从而简化了该过程。

8.2.1 KeyListener实用工具

YUI的KeyListener实用工具模仿了addListener函数，因为它可以用来将keyup或keydown事件的事件处理程序附加到一个DOM元素。区别在于，对于KeyListener，需要实例化一个新对象，而addListener并不需要这样做。此外，语法也略有不同。但是，KeyListener的优点在于可以使用方法通过相同的名称来启用或禁用这个实例化的对象。这使得键盘处理程序的管理变得非常简单。此外，它还简化了修改键(Ctrl、Alt和Shift)的捕获。

```
<html>
    <head>
        <title>keyListener</title>
    </head>
    <body>
<script src="yahoo-min.js"></script>
<script src="event-min.js"></script>
<script>
    var YX = YAHOO.example;
    YX.cut = function () {
        alert("Cut!");
    };
    YX.copy = function () {
        alert("Copy!");
    };
    YX.paste = function () {
        alert("Paste!");
    };
    YX.cutKey = new YAHOO.util.KeyListener(
        document,
        {ctrl: true, keys: 88},
        YX.cut);
    YX.cutKey.enable();
    YX.copyKey = new YAHOO.util.KeyListener(
        document,
        {ctrl: true, keys: 67},
        YX.copy);
    YX.copyKey.enable();
    YX.pasteKey = new YAHOO.util.KeyListener(
```

```
                document,
                {ctrl: true, keys: 86},
                YX.paste);
            YX.pasteKey.enable();
        </script>
    </body>
</html>
```

这个示例捕获常见的 Ctrl+X、Ctrl+C 和 Ctrl+V 按键组合，并为每种组合触发不同的回调函数。因此，将一个新的 KeyListener 对象实例化到变量 cut 中，它的第一个参数是 document 对象。这意味着不管用户在哪里触发事件，都可以捕获该事件。但是，这个参数还可以是任意 DOM 元素，因此可以限制捕获事件的范围。与 YUI 中的模式一样，这个参数既可以是表示元素 ID 的字符串，也可以是 HTMLElement 引用。第二个参数是一个称为 keyData 的对象，它基本上包含关于在什么按键和/或按键组合上调用该事件处理程序的信息。这就是 KeyHandler 与 addListener 的不同之处，因为对于后者而言，每次触发一个事件(例如每个 onclick 事件)时都会引发事件处理程序；另一方面，KeyHandler 默认捕获 keydown 事件，只有在按键和/或按键组合等于 keyData 对象中指定的值时才会触发事件处理程序。

keyData 对象只是一个简单的对象字面值。换句话说就是，我们并不需要在其他地方的现有类中实例化这个对象，它只需要包含由 KeyHandler 识别的名/值对即可。下面是 keyData 对象可以采用的一些示例形式。

```
{keys: 88}
```

这会捕获 X 键，88 是 X 键的字符代码。

```
{keys: [88, 67, 86]}
```

这会捕获 X、C 和 V 键。

```
{ctrl: true, keys: 88}
```

这会只有在按下 Ctrl 键时才捕获 X 键。

```
{ctrl: true, alt: true, shift: true, keys: [88, 67, 86]}
```

这会只有在同时按下 Ctrl、Alt 和 Shift 键时才捕获 X、C 和 V 键。

因为对象属性是顺序无关的，所以上面的示例与下面的编写方式作用相同，而且仍然有效：

```
{shift: true, keys: [88, 67, 86], alt: true, ctrl: true}
```

这里的唯一区别在于，keyData 对象的属性顺序不同。

当然，与所有 YUI 事件处理函数都具有的模式类似，可以将数据对象传给 KeyListener 回调函数，而且可以改变它的执行作用域。但是与 addListener 相比，使用 KeyHandler 执行操作的方式稍有不同。模式是相同的，因为可以传入一个对象，而且可以重写作用域，将该对象设置为执行作用域。但是，这些工作均是在对象内部完成的，而不是作为 KeyHandler 构造函数的独立参数。

下面是前面的 KeyHandler 代码示例的简化版本，其中将一个数据对象传给 cut 回调函数。请注意 object 参数令人误解的命名(在这里称为 scope)。

```
<html>
    <head>
        <title>keyListener 2</title>
    </head>
    <body>
        <script src="yahoo-min.js"></script>
        <script src="event-min.js"></script>
        <script>
            var YX = YAHOO.example;
            YX.cut = function (eType, codeAndEv, dataObj) {
                var msg = "The event type responsible for triggering ";
                msg    += "this function is '" + eType + "'.\n";
                msg    += "The key code for the key that was pressed ";
                msg    += "is '" + codeAndEv[0] + "'.\n";
                msg    += "And the message passed through the data object ";
                msg    += "is '" + dataObj + "'.";
                alert(msg);
            };
            YX.cutKey = new YAHOO.util.KeyListener(
                document,
                {ctrl: true, keys: 88},
                {fn: YX.cut, scope: "Cutting!"});
            YX.cutKey.enable();
        </script>
    </body>
</html>
```

在JavaScript中，字符串也是对象，因此将字符串"Cutting! "作为数据对象传递也是完全有效的。此外，还要注意这个回调函数接收3个参数。第一个参数是表示事件类型的字符串，该字符串来自于YUI的Custom Event对象，本章稍后将讨论它。这里，我们传递的是字符串值“keyPressed”。当分配回调函数以响应多种类型的事件时，要想确定什么操作要为该回调函数的调用负责，那么这个参数可能非常有用。第二个参数是一个数组，该数组的第一项是所按下键的键代码，第二项是浏览器生成的实际的事件对象。最后，第三个参数是数据对象。这可以是任意对象，无论是字符串还是函数均可。当同时按下Ctrl和X键时，上面的回调函数产生如下的消息：

```
The event type responsible for triggering this function is 'keyPressed'.
The key code for the key that was pressed is '88'.
And the message passed through the data object is 'Cutting!'.
```

修改执行作用域非常简单，只要使用另一个参数即可。

```
var cut = new YAHOO.util.KeyListener(
    document,
    {ctrl: true, keys: 88},
    {fn: cut, scope: newScope, correctScope: true}
);
```

类似地，这个数据对象可以保持不变，并可以通过correctScope参数传入一个新的作用域对象。

```
var cut = new YAHOO.util.KeyListener(
    document,
    {ctrl: true, keys: 88},
    {fn: cut, scope: "Cutting!", correctScope: newScope}
);
```

8.2.2　getCharCode 函数

如果知道要捕获的按键的键(或字符)代码，那么利用 KeyHandler 函数建立事件处理程序就是很好的做法。与平时一样，浏览器制造商似乎喜欢让开发人员的生活变得困难。因此，就算是完成从事件对象中获取 keyCode 值这样简单的事情也可能比较麻烦。YUI 的 Event Utility 有一个有用的函数 getCharCode。它检查事件对象以获取 keyCode 值，如果在其中没有找到正在寻找的内容，它就会在 charCode 中查找。这个函数也会针对 Safari 执行一些特殊处理，这是因为该浏览器将键代码存储到一个完全不同的位置中。下面是一个有用的程序，它输出在键盘上按下的任何按键的键代码(参见图 8-3)。

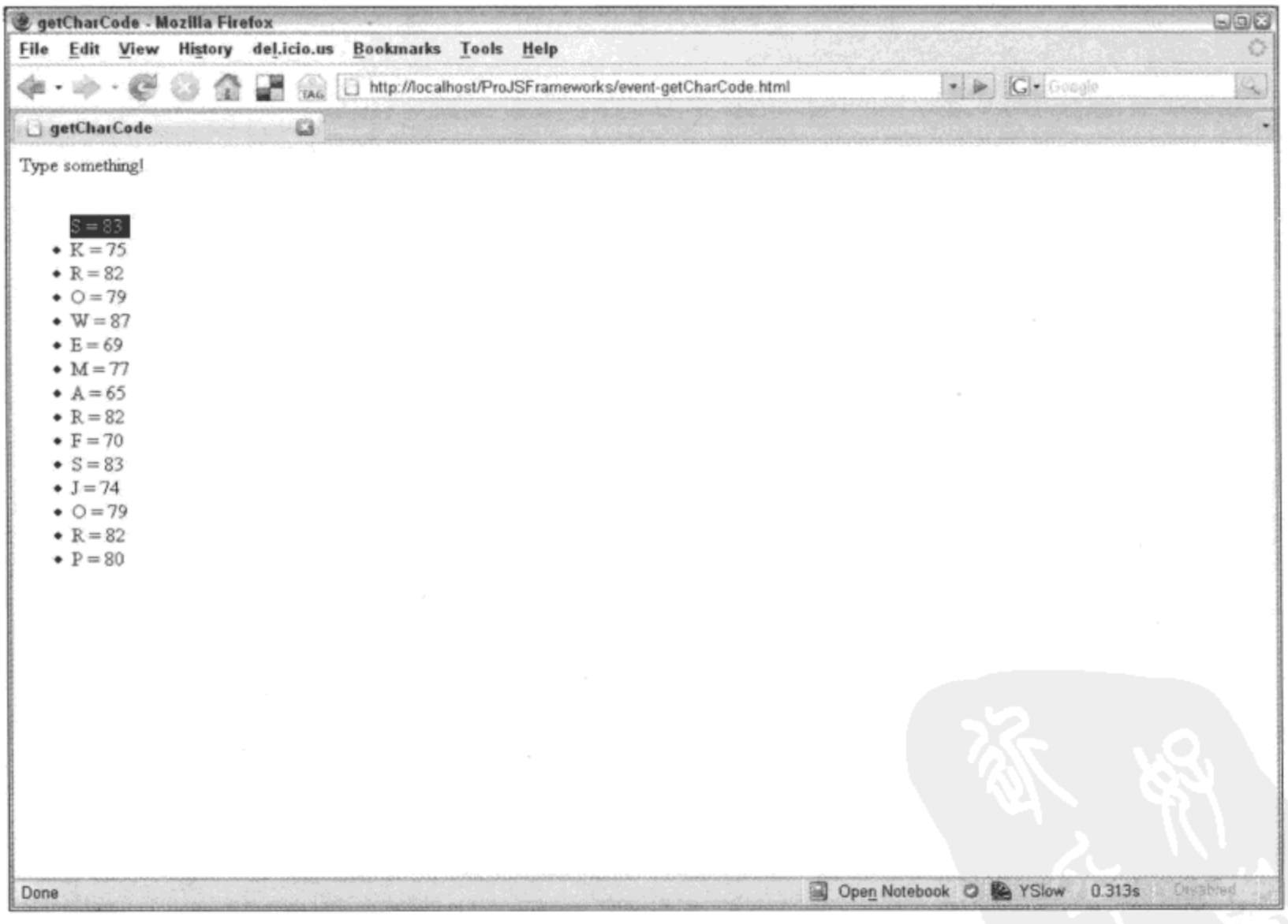


图　8-3

自然地，该程序使用了 getCharCode 函数。

```
<html>
    <head>
```

```
    <title>getCharCode</title>
    <style type="text/css">
        ul {
            float: left;
        }
        .first {
            color: #fff;
            background: #000;
        }
    </style>
</head>
<body>
    <p>Type something!</p>
    <ul id="output">
    </ul>
    <script src="yahoo-dom-event.js"></script>
    <script>
        var YD = YAHOO.util.Dom,
            YE = YAHOO.util.Event,
            YX = YAHOO.example;
        YX.output = YD.get("output");
        YE.addListener(document, "keyup", function (e) {
            e = e || event;
            var li = document.createElement("li");
            var cCode = YE.getCharCode(e);
            var txt = document.createTextNode(
                String.fromCharCode(cCode) + " = " + cCode);
            li.appendChild(txt);
            li.className = "first";
            var firstChild = YD.getFirstChild(YX.output);
            if (firstChild) {
                firstChild.className = "";
                YD.insertBefore(li, firstChild);
            } else {
                YX.output.appendChild(li);
            }
        });
    </script>
</body>
</html>
```

注意，针对字母的大写形式和小写形式所返回的键代码值是一样的。这是因为不管是大写还是小写，负责产生字母的按键均是相同的按键。不同之处在于使用了 Shift 键，而它是一个修饰键。Shift 键也有自己的键代码(16)，在该示例程序的输出结果中，它的键代码将会占据单独一行。

基本上，这个程序的功能就是在 document 元素上建立一个事件处理程序，每当引发 keyup 事件时，就会触发一个匿名函数。这个匿名函数创建一个列表项节点，并使用刚刚按下的按键的键代码来填充该节点。这个键代码是根据传给 YUI 函数 getCharCode 的事件对象(e)而得到的。然后，它将这个列表项附加到 DOM 中已经存在的 ID 为 output 的无序列表。这个程序还将字符代码转换成字母，并将该字母添加到列表项中的文本。这对于字母数字键来说可以工作得很好，但对

于其他键并不合适。例如，向左、向右、向上和向下方向键分别产生%、'、&和(，这与键盘按键上印出的内容不同。

8.2.3　getXY

事件对象中包含鼠标坐标数据，该数据存储在两个单独的变量中。所有浏览器均将这些坐标称为 pageX 和 pageY，但是 Internet Explorer 除外，它将这些坐标称为 clientX 和 clientY，更不必说甚至这些数值在 IE 中也不精确，因为还需要考虑页面滚动的情况。getXY 函数做了许多工作来规范化这些问题，并使得我们可以通过一个函数调用来方便地报告鼠标的 x 和 y 坐标。

```
<html>
    <head>
        <title>getXY</title>
    </head>
    <body>
        <p id="output"></p>
        <script src="yahoo-dom-event.js"></script>
        <script>
            var YD = YAHOO.util.Dom,
                YE = YAHOO.util.Event,
                YX = YAHOO.example;
            YX.output = YD.get("output");
            YE.addListener(document, "mousemove", function (e) {
                e = e || event;
                YX.output.innerHTML = YE.getXY(e);
            });
        </script>
    </body>
</html>
```

这个函数建立了一个简单的事件处理程序，用来在光标滑过 document 元素上方时跟踪鼠标。它报告鼠标的 x 和 y 坐标，方法是将 getXY 函数返回的值输出到 ID 为 output 的段落元素。返回的值是一个数组，x 坐标值是这个数组的第一项，而 y 坐标值是第二项。

还有两个函数用于分别报告 x 和 y 值。它们分别称为 getPageX 和 getPageY，该名称服从更加流行的变量名 pageX 和 pageY。实际上，getXY 函数只是一个调用 getPageX 和 getPageY 函数的别名函数，它获得这两个函数的返回值，将这些返回值放入一个数组中，然后返回该数组。

8.2.4　getTarget 函数

事件对象包含一个名为 target 的变量，该变量中包含一个指向鼠标单击的元素的引用。当执行诸如事件委托(一个事件处理程序负责 DOM 中多个对象的单击事件)之类的任务时，该变量就非常有用。但是，遵循“我们喜欢与众不同”的精神，Internet Explorer 将它的目标变量称为 srcElement。getTarget 函数检查变量 target，如果没有找到该变量，它就获取 srcElement 变量的值，从而解决了这个跨浏览器问题。

```
<html>
```

```
<head>
    <title>getTarget</title>
</head>
<body>
    <p id="output"></p>
    <ul>
        <li>
            <span>Hello World!</span>
        </li>
    </ul>
    <script src="yahoo-dom-event.js"></script>
    <script>
        var YD = YAHOO.util.Dom,
            YE = YAHOO.util.Event,
            YX = YAHOO.example;
        YX.output = YD.get("output");
        YE.addListener(document, "click", function (e) {
            e = e || event;
            YX.output.innerHTML = YE.getTarget(e).nodeName;
        });
    </script>
</body>
</html>
```

这个示例在 document 对象上建立一个 onclick 事件处理程序，并报告文档中单击的元素的节点名称。getTarget 函数可以访问 nodeName 值，这是因为 JavaScript 只是将 YAHOO.util.Event.getTarget(e)替换成它返回的值，然后检查 nodeName。这实际上是一些库中称为链式调用的相当流行的模式。

注意，当单击页面的下半部分时，上面的示例将针对不同的浏览器返回不同的值。这是因为在诸如 Internet Explorer 这样的浏览器中，body 元素跨越页面的整个高度；而在诸如 Firefox 这样的浏览器中，body 元素只占用可读取内容的高度。因此，在 IE 中单击页面底部会返回 body；而在中，它会返回 html。

8.2.5　getRelatedTarget 函数

当跟踪鼠标的移动时(特别是在执行拖放操作期间)，能够知道鼠标最终的位置会比较有用。该工作并不像看起来这么简单。探测鼠标指针运动的传统方式是将 onmouseover 事件分配给所有相关元素。但是这项技术非常笨拙，因为需要复杂的编程从一个元素切换到另一个元素。这意味着一个元素的 onmouseout 事件需要与另一个元素(鼠标指针刚离开这个元素)的 onmouseover 事件进行某种沟通。通过事件对象的 relatedTarget 属性可以探测这种元素之间的交互。但是我们再次会遇到同样的问题，Internet Explorer 并没有使用相同的名称称呼该属性。而且，这属性无法智能地执行如下操作：当事件是 mouseout 时，它应该报告鼠标要到达的地方；而如果是 mouseover 事件，它就应该报告鼠标来自哪里。YUI 使用 getRelatedTarget 函数解决了这个问题。下面的程序使用 getRelatedTarget 函数弄清楚鼠标的运动方向。如果鼠标向上滑过列表项，就把一个向上的箭头附加到该列表项；而如果鼠标向下移动，该列表项就会得到一个向下的箭头。

```
<html>
    <head>
        <title>getRelatedTarget</title>
    </head>
    <body>
        <ul>
            <li id="widget">Widget<span class="dir"></span></li>
            <li id="gadget">Gadget<span class="dir"></span></li>
            <li id="whoozit">Whoozit<span class="dir"></span></li>
            <li id="foozit">Foozit<span class="dir"></span></li>
            <li id="barzit">Barzit<span class="dir"></span></li>
            <li id="bazzit">Bazzit<span class="dir"></span></li>
        </ul>
        <script src="yahoo-dom-event.js"></script>
        <script>
            var YD = YAHOO.util.Dom,
                YE = YAHOO.util.Event,
                YX = YAHOO.example;
            YX.li = document.getElementsByTagName("li");
            YE.addListener(YX.li, "mouseout", function (e) {
                e = e || event;
                var rTarget = YE.getRelatedTarget(e);
                var dir = YD.getElementsByClassName(
                    "dir", "span", this);
                dir = (dir[0]) ? dir[0] : dir;
                var arrow = "";
                var previous =  YD.getPreviousSibling(this);
                var next = YD.getNextSibling(this);
                var first = YD.getFirstChild(this.parentNode);
                var last = YD.getLastChild(this.parentNode);

                if (rTarget === previous) {
                    arrow = "&uarr;";
                } else if (rTarget === next) {
                    arrow = "&darr;";
                } else if (this === first) {
                    arrow = "&uarr;";
                } else if (this === last) {
                    arrow = "&darr;";
                }
                dir.innerHTML = arrow;
            });
        </script>
    </body>
</html>
```

基本上，这个程序的功能就是将一个 onmouseout 事件处理程序附加到页面中的所有 li 元素。然后，该处理程序确定相关的目标是什么，并根据结果为每个 li 元素分配一个向上或向下的箭头。这些箭头插入到每个 li 元素中类名为 dir 的 span 内。使用 span 只是为了便于操作，这样就不需要处理 li 的文本内容(事件处理程序中的变量只是局部变量，不需要在 YAHOO.example 名称空间内

声明它们)。结果如图 8-4 所示。

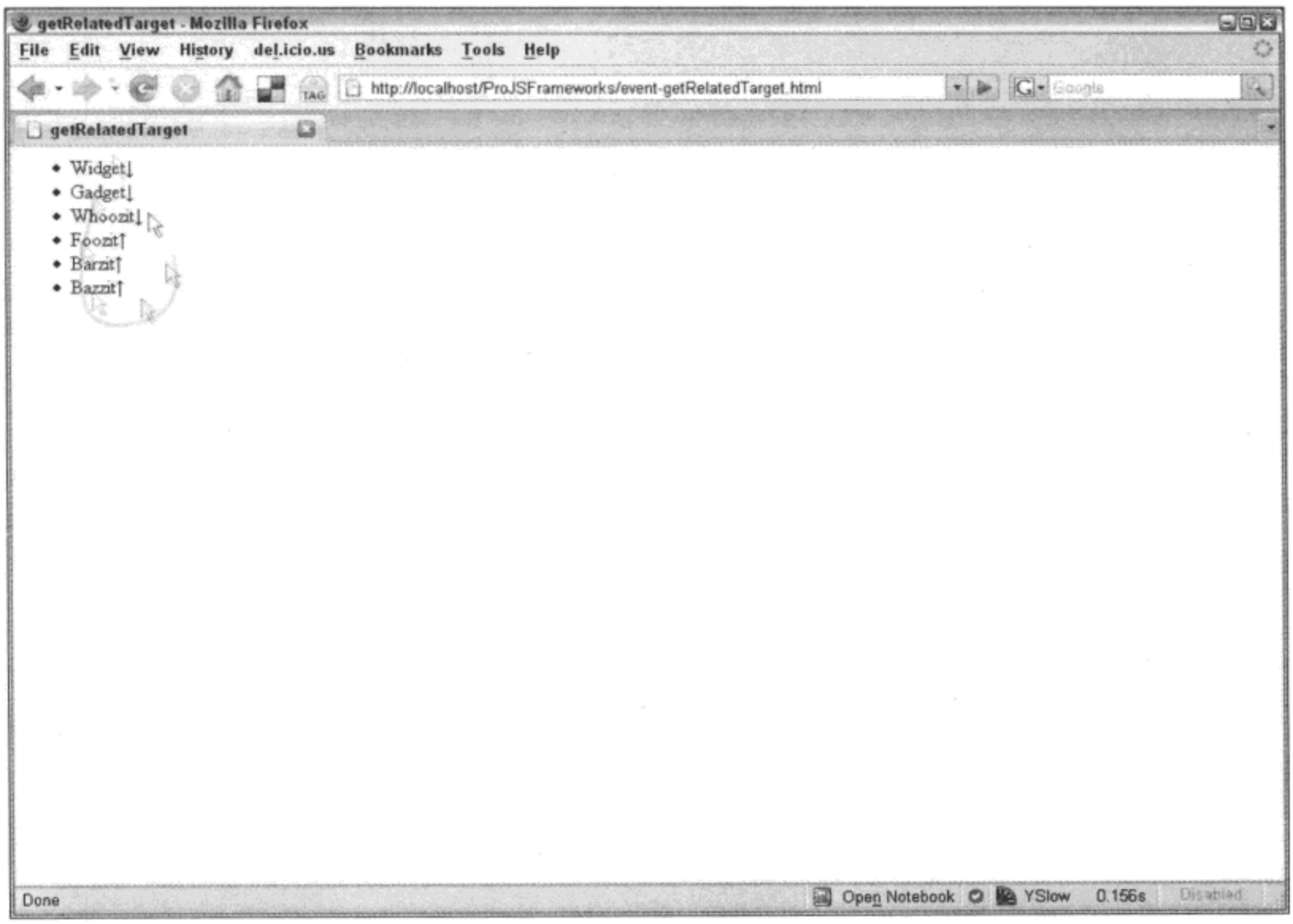


图　8-4

注意：

图 8-4 中的每个列表项均跨越浏览器窗口的整个宽度。因此，它们能够捕获鼠标指针的移动，即使可见的文本内容并没有占据整个窗口宽度。

8.2.6　preventDefault 函数

锚点元素通常都分配有 onclick 事件处理程序。然而，锚点元素会带来一些问题，因为它们的默认操作是跟随其 href 属性的值。因此，尽管锚点元素的事件处理程序已经引发，但是用户看不到它的结果，因为在单击操作完成之后就开始加载一个新页面。在传统上，向该事件返回 false 就会停止默认操作。

```
var a = document.getElementById("foo");
a.onclick = function () {
    // do something
    return false;
};
```

还有另一种通过事件对象直接停止该事件的默认行为的方式。在 Firefox 和其他兼容标准的浏

览器中，事件对象具有一个名为 preventDefault 的方法，调用这个方法就会把事件的默认操作取消。当然，Internet Explorer 处理这种情况的方式又有所不同，它要求在 returnValue(其 event 对象的一个属性)上设置 false 值。YUI 将这种差异包装进规范化的 preventDefault 函数中。

```
<html>
    <head>
        <title>preventDefault</title>
    </head>
    <body>
        <a href="/about/" id="about">About us</a>
        <script src="yahoo-dom-event.js"></script>
        <script>
            var YD = YAHOO.util.Dom,
                YE = YAHOO.util.Event,
                YX = YAHOO.example;
            YX.doSomething = function (e) {
                e = e || event;
                YE.preventDefault(e);
            };
            var a = YD.get("about");
            YE.addListener(a, "click", YX.doSomething);
        </script>
    </body>
</html>
```

在某些不可能返回 false 值的场合以及其他场合中，它就不能工作，例如在上面的示例中就是如此。由于 YUI 的 addListener 函数将事件处理程序附加到元素的方式，返回 false 值并不会取消单击操作的默认行为。因此，使用 preventDefault 函数取消事件的默认操作总是可取的做法。

8.2.7　stopPropagation 函数

在 DOM 中的一个元素上单击不仅仅会触发它的 onclick 事件，还会触发位于鼠标指针下方的所有元素的 onclick 事件。因此，如果有一个当单击 document 元素时触发的事件处理程序，而且在 document 元素内的锚点上还有另一个事件处理程序，那么当单击该锚点时就会同时引发这两个事件处理程序。

```
<html>
    <head>
        <title>stopPropagation</title>
    </head>
    <body>
        <a href="/about/" id="about">About us</a>
        <script src="yahoo-dom-event.js"></script>
        <script>
            var YD = YAHOO.util.Dom,
                YE = YAHOO.util.Event,
                YX = YAHOO.example;
            YX.msg = function (e, txt) {
```

```
            e = e || event;
            alert(txt);
            YE.preventDefault(e);
        };
        YX.about = YD.get("about");
        YE.addListener(YX.about, "click", YX.msg, "About!");
        YE.addListener(document, "click", YX.msg, "Document!");
    </script>
</body>
</html>
```

在这个示例中，在“About us”锚点上单击将两次触发 msg 函数，一次是针对该锚点的单击事件，另一次是针对 document 元素的单击事件。为了停止这种事件传播，兼容标准的浏览器在事件对象上提供了一个名为 stopPropagation 的方法。当调用该方法时，它会阻止触发其事件的操作继续深入 DOM。Internet Explorer 的事件对象没有 stopPropagation 方法，而是有一个 cancelBubble 属性，将其值设置为 false 就可以停止事件传播。YUI 再次将这种差异包装进一个便利的 stopPropagation 函数，从而将这种跨浏览器行为规范化。

```
<html>
    <head>
        <title>stopPropagation</title>
    </head>
    <body>
        <a href="/about/" id="about">About us</a>
        <script src="yahoo-dom-event.js"></script>
        <script>
            var YD = YAHOO.util.Dom,
                YE = YAHOO.util.Event,
                YX = YAHOO.example;
            YX.msg = function (e, txt) {
                e = e || event;
                alert(txt);
                YE.stopPropagation(e);
                YE.preventDefault(e);
            };
            YX.about = YD.get("about");
            YE.addListener(YX.about, "click", YX.msg, "About!");
            YE.addListener(document, "click", YX.msg,
                "Clicking 'About us' won't trigger this");
        </script>
    </body>
</html>
```

8.2.8　stopEvent 函数

在上面的示例中，我们同时使用了 stopPropagation 和 preventDefault 函数，目的是为了隔离锚点上的单击事件并停止它的默认行为，即跟随该锚点的 href 属性中的 URL。YUI 提供了一个名为 stopEvent 的便利函数，它依次调用这两个函数，从而将实现这种行为所需要编写的代码行数从两

行减为一行。

```
<html>
    <head>
        <title>stopEvent</title>
    </head>
    <body>
        <a href="/about/" id="about">About us</a>
        <script src="yahoo-dom-event.js"></script>
        <script>
            var YD = YAHOO.util.Dom,
                YE = YAHOO.util.Event,
                YX = YAHOO.example;
            YX.msg = function (e, txt) {
                e = e || event;
                alert(txt);
                YE.stopEvent(e);
            };
            YX.about = YD.get("about");
            YE.addListener(YX.about, "click", YX.msg, "About!");
            YE.addListener(document, "click", YX.msg,
                "Clicking 'About us' won't trigger this");
        </script>
    </body>
</html>
```

8.3　处理自定义事件

诸如 onclick、onmouseover 和 onload 这样的 JavaScript 事件非常有用。实际上，它们是现代 Web 开发的关键部分。基于事件的编程方式已经成为现今网站的主要创建方式，以至于 JavaScript 将从更多事件中受益，但是应该添加哪些事件呢？问题随之而来，人们编写了许多不同类型的应用程序，因此它们可能触发的事件的类型非常多，以至于按照这种方式来扩展当前的事件模型并不可行。本节将介绍 YUI 的自定义事件。利用自定义事件，我们就可以在 JavaScript Web 应用程序中的许多地方创建和触发事件。实际上，YUI 自身大量使用自定义事件。例如，一旦 onDOMReady 函数(通过多种方式)判断出 DOM 确实准备就绪，它就触发 DOMReady 自定义事件。

虽然 JavaScript 是一门可塑性和可扩展性极强的语言，但是它不允许创建自定义事件。因此，要实现这种功能，就必须采用不同的语法。换言之，不能像下面这样编写代码：

```
el.onmycustomevent = doSomething; // not possible
```

但是，我们可以将多个函数放入一个数组中，然后遍历该数组并依次触发这些事件，而这正是 YUI Custom Events 的真正功能所在。Custom Events 通过大量的代码来确保可以订阅和退订自定义事件，确保处理程序在正确的作用域中触发，并确保处理程序中的错误会被捕获和管理。

YUI Custom Events 背后的基本原理就是允许函数或对象将“关注的时刻”提供给其他的订阅者函数。换言之，为了将一个事件处理程序附加到一个自定义事件，它需要“订阅”该事件。这与 addListener 函数的工作方式类似，可以将多个事件处理程序附加到(或订阅)同一个事件。

8.3.1 创建和订阅自定义事件

创建一个自定义事件是一件简单的事情，只需要实例化一个对象即可。可以在任何地方完成这个操作，但是按照惯例，这个对象是函数的一个成员。

```
function Widget(name) {
    this.name = name;
    this.onNameChange = new YAHOO.util.CustomEvent("namechange", this);
};
```

在这里，Widget 类的构造函数包含一个名为 onNameChange 的自定义事件。onNameChange 是一个 YUI Custom Event 对象，它有一个 subscribe 方法，其他函数可以通过该方法来订阅该事件。

```
var gadget = new Widget("foo");
gadget.onNameChange.subscribe(doSomething);
```

当 onNameChange 事件引发时(通过它的 fire 方法)，所有订阅者函数都将执行。

```
gadget.onNameChange.fire();
```

此处的技巧是在关键的时刻引发自定义事件。如同上面示例中的自定义事件名称所暗示的那样，应该在 Widget 对象的名称发生变化时引发该事件。否则，如果不在正确时间和正确地点引发事件处理程序，它就没有多大用处。

```
<html>
    <head>
<title>customEvent</title>
</head>
<body>
    <a href="/change/" id="change">Change name to "baz"</a>
    <script src="yahoo-dom-event.js"></script>
    <script>
        var YE = YAHOO.util.Event,
            YX = YAHOO.example;
        YX.Widget = function (name) {
            this.name = name;
            this.onNameChange = new YAHOO.util.CustomEvent("namechange", this);
            this.setName = function (newName) {
                this.name = newName;
                this.onNameChange.fire();
            }
        };
        YX.changeName = function (e, params) {
            params.obj.setName(params.name);
            YE.preventDefault(e);
        };
        YX.announce = function () {
            alert("The object's name is '" + this.name + "'");
        };

        YX.gadget = new YX.Widget("foo");
        YX.gadget.onNameChange.subscribe(YX.announce);
        YX.gadget.setName("bar");
```

```
            YE.addListener("change", "click", YX.changeName,
                {obj: YX.gadget, name: "baz"});
        </script>
    </body>
</html>
```

在这个示例中，将 Widget 类实例化到一个名为 gadget 的变量中。Widget 有一个 name 属性、一个 YUI Custom Event 对象 onNameChange 以及一个 setName 方法。尽管可以通过像 gadget.name = "new name";这样直接访问 name 属性来手动设置它，但是这样就无法触发自定义事件。这就是 setName 方法存在的原因。这样一来，一旦设置新的 name 值，就可以随之触发 onNameChange 自定义事件。

注意：

虽然在典型意义上 JavaScript 并不是面向对象的语言，但是为了清晰起见，这里使用了构造函数、类和方法等术语。实际上，JavaScript 的行为足够类似于一门典型的面向对象语言，因此这些术语具有相对意义。

将 Widget 对象实例化，它的 name 属性的值为“foo”。然后分配 announce 函数作为它的 onNameChange 事件的订阅者。换言之，当引发 onNameChange 事件时，就会执行 announce 函数。然后，通过 setName 方法将它的名称修改成“bar”。这会导致执行 announce 函数，页面加载之后立即显示警告消息“The object's name is 'bar'”(参见图 8-5)。之后，将 onclick 事件处理程序附加到 ID 为 change 的锚点元素。这个处理程序是一个名为 changeName 的函数，它接收一个对象作为参数，该对象中含有一个指向待修改 name 属性的对象的引用以及要修改成的文本。

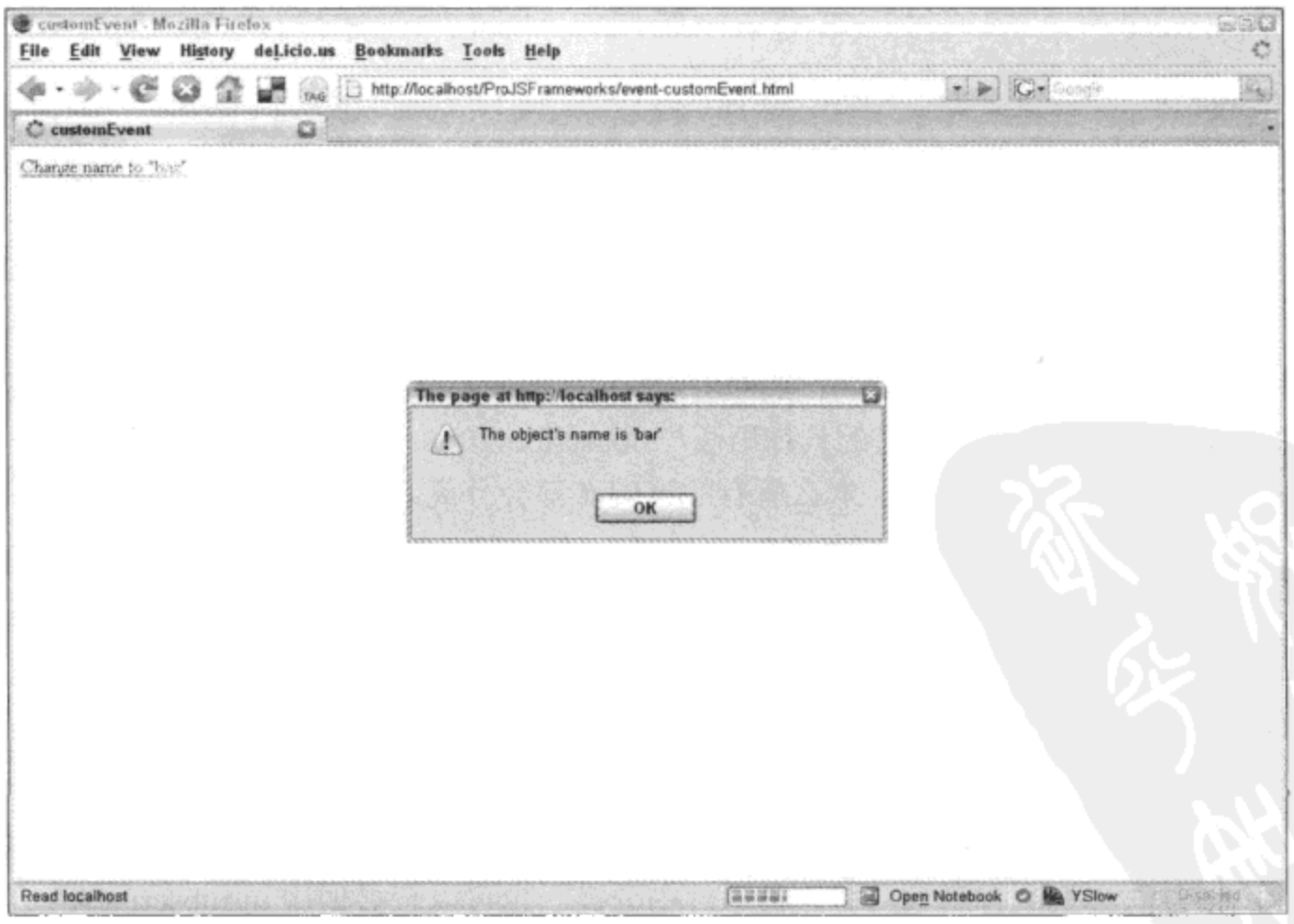


图　8-5

因此，当单击“Change name to 'baz'” 链接时，它触发 changeName 函数，该函数在 gadget 对象中设置一个新的 name 值。这又会触发自定义事件 onNameChange，该事件执行它的所有订阅者(这里只有一个订阅者，即 announce 函数)。最后，announce 函数将 name 的变动显示给用户(参见图 8-6)。

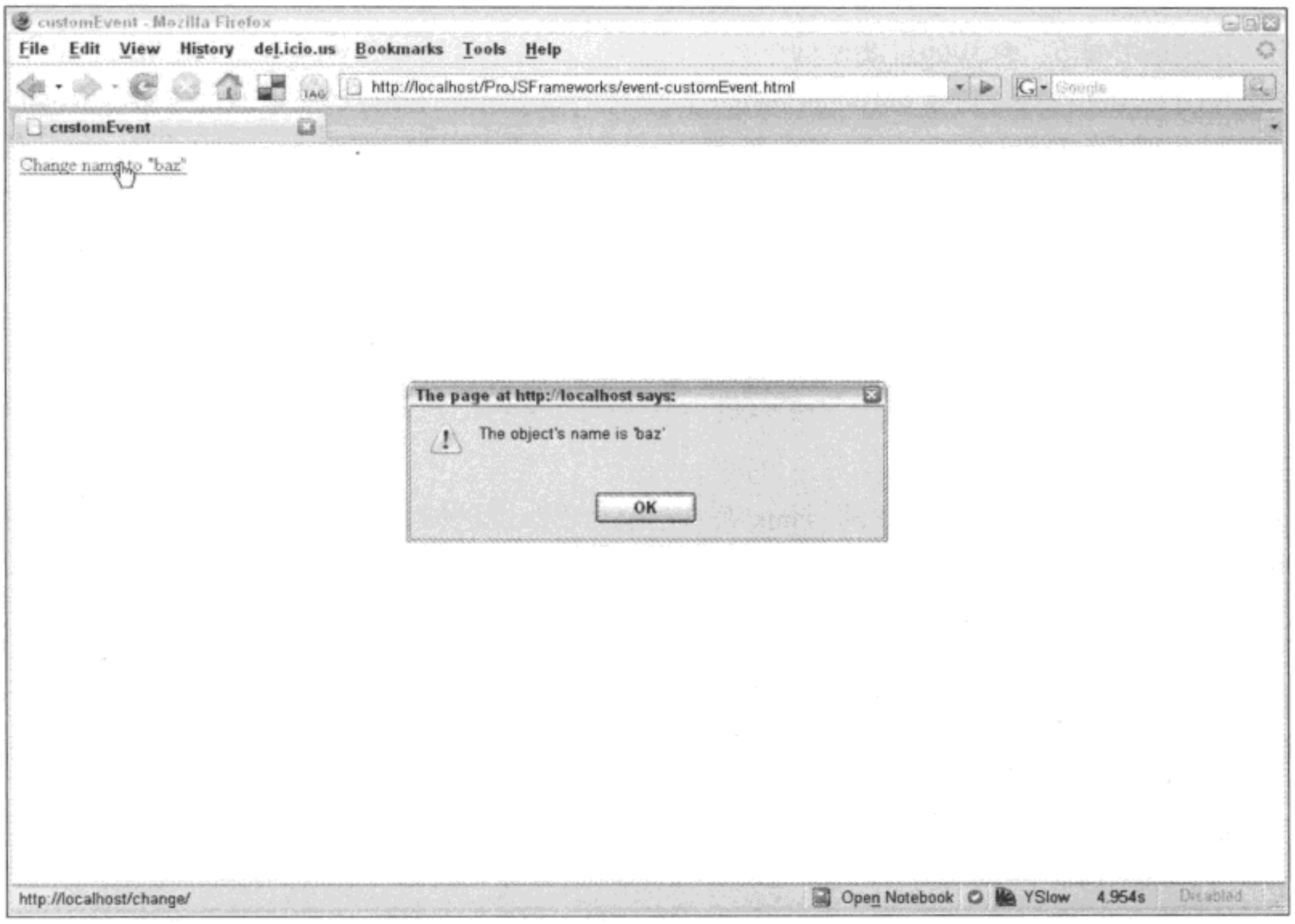


图　8-6

8.3.2　退订自定义事件

如果能够订阅自定义事件，那么自然能够退订它。unsubscribe 方法可以使已经附加到某个自定义事件的事件处理程序分离，它的具体语法与 subscribe 方法一样，但是不同之处在于，如果没有把要分离的函数作为参数传入，那么所有的订阅者都会分离。

```
gadget.onNameChange.unsubscribe(announce);
```

该代码会把 announce 函数从 onNameChange 自定义事件中分离。

```
gadget.onNameChange.unsubscribe();
```

该代码会把所有函数从 onNameChange 自定义事件中分离。

实际上，后面这行代码(在发现没有传入函数时)调用了 unsubscribeAll 方法，正如其名称所暗示的那样，该方法会把所有函数从自定义事件中分离。也可以直接调用 unsubscribeAll 方法。

```
gadget.onNameChange.unsubscribeAll();
```

unsubscribe(不带任何参数)和 unsubscribeAll 的不同之处(以及这两个选项为什么完成同样的任务)就在于前者是作为一个自动化过程的一部分调用的，尽管可以为其传递一个变量，但它可能不包含任何内容，这会提示它调用后者。

8.3.3　subscribeEvent 方法

YUI Custom Events 自身实际上也有一个自定义事件。每当一个新的订阅者订阅该事件时，就会引发 subscribeEvent 方法。这样就可以处理诸如以下的情况：捕获只会引发一次(而且已经引发)的事件的迟到订阅者。

8.4　管理浏览器历史并修正后退按钮

现在越来越多的网站运用了 Flash 和 AJAX。然而，这两者的流行已经带来了一些问题：浏览器后退按钮和收藏功能出现错误。通常，完全在一个 Flash 对象内构成的网站从来不会改变页面。因此，即使 Flash 对象内的页面发生变化，该站点的 URL 仍然保持不变。大量使用 AJAX 的站点也同样如此。当一个用户已经在某个 AJAX 驱动的站点上访问了一段时间后，他们看到的页面与最初加载的页面完全不同(但是 URL 没有改变)。在这两种情况下，无论用户在什么地方、什么时候创建书签，收藏该页面都会把他们带回页面初始时的状态。

YUI Browser History Manager 提供了一个方法来解决这两个问题。实际上，这个管理器会将页面中的关键变化记录在模块(module)中，并通过它的 hash 属性将该信息存储到页面的 URL 中，就像下面这样：

```
http://localhost/ProJSFrameworks/history.html#step=configure
```

模块就是 History Manager 用来划分需要记录的不同数据块的方式。每个模块是由存储在 URL 中的键/值对来表示的。在这里，History Manager 已经将名为 step 的模块的状态保存起来，该模块的值为 configure。

我们可以保存任何类型的特定数据，没有任何要求，只要它是键/值对即可。因此，状态可以表示某个过程中的一个步骤(如同前面的示例所示)，它的值可用作一个指示器，用来说明当页面初始加载时应该显示什么内容。因为 History Manager 支持一次性存储多个模块，所以它们可用来存储一个页面中多个组件的状态。它们还可以存储一个组件所需的多个数据块。因此，如果页面是一个地图绘制应用程序，那么可以使用一个模块表示经度，第二个模块表示纬度，第三个模块表示缩放级别。

现在，对于 AJAX 驱动的应用程序，应用程序将在关键的时间点调用这个管理器并保存状态。对于 Flash 应用程序，Flash 对象将必须调用这个页面中的 JavaScript 函数，并将相关的数据传给这些函数。

下面是一个简单的 AJAX 驱动的旅行预订流程示例(参见图 8-7)。这里的 AJAX 是带引号的，因为这个代码示例并没有包含任何 AJAX 代码。它只是向屏幕输出一些文本，并假定从服务器那里获取这些内容。这样做是为了简化这个示例。

```
<html>
    <head>
```

```
    <title>History</title>
    <style type="text/css">
      #yui-history-iframe {
        position:absolute;
        top:0; left:0;
        width:1px; height:1px;
        visibility:hidden;
      }
    </style>
</head>
<body>
    <iframe id="yui-history-iframe" src="history-iframe.html"></iframe>
    <input id="yui-history-field" type="hidden">
    <h1>Book a vacation</h1>
    <ul id="steps">
        <li id="step1">
            <a href="/choose/">Step 1 - Choose your vacation</a>
        </li>
        <li id="step2">
            <a href="/configure/">Step 2 - Configure it!</a>
        </li>
        <li id="step3">
            <a href="/pay/">Step 3 - Payment</a>
        </li>
        <li id="step4">
            <a href="/confirmation/">Step 4 - Confirmation</a>
        </li>
</ul>
<div id="step"></div>
<script src="yahoo-dom-event.js"></script>
<script src="history-min.js"></script>
<script>
    var YD = YAHOO.util.Dom,
        YE = YAHOO.util.Event,
        YH = YAHOO.util.History,
        YX = YAHOO.example;
    YX.initialize = function () {
        YX.loadContents("step", initialState);
    };
    YX.loadContents = function (containerId, state) {
        var step = YD.get(containerId);
        step.innerHTML = "<h2>" + state + "</h2>";
        step.innerHTML += "This is where the AJAX loaded contents of the ";
        step.innerHTML += "step '<strong>" + state + "</strong>' ";
        step.innerHTML += "would be.";
    };
    YX.stateChangeHandler = function (state) {
        YX.loadContents("step", state);
    };
    YX.stepClicked = function (e) {
```

```
            var state = this.href.replace(/\/$/, "");
            state = state.substring(state.lastIndexOf("/") + 1);
            try {
                YH.navigate("step", state);
            } catch (e) {
                //History manager not initialized
            }
            YE.preventDefault(e);
        };
        var bookmarkedState = YH.getBookmarkedState("step");
        var initialState = bookmarkedState || "choose";
        YH.register("step", initialState, YX.stateChangeHandler);
        try {
            YH.initialize("yui-history-field", "yui-history-iframe");
        } catch (e) {
            //Browser not supported
        }
        var links = YD.get("steps").getElementsByTagName("a");
        YE.addListener(links, "click", YX.stepClicked);
        YH.onReady(YX.initialize);
    </script>
</body>
</html>
```

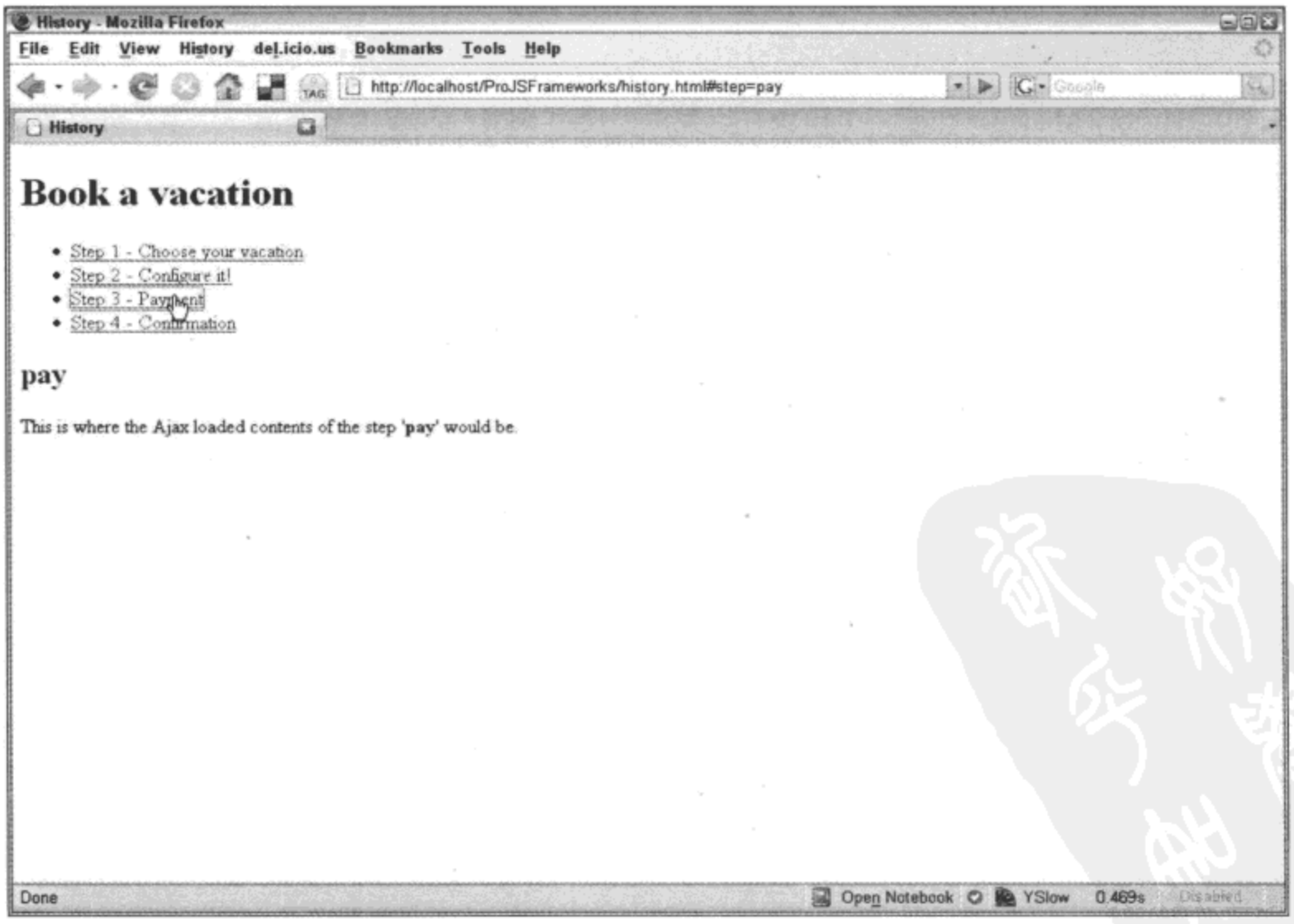


图　8-7

History Manager 需要把下面这两个 HTML 标记放入标记文件中。

```
<iframe id="yui-history-iframe" src="history-iframe.html"></iframe>
<input id="yui-history-field" type="hidden">
```

这两个标记用来存储当前状态的值。iframe 的来源不必是任何特殊的内容，但它确实必须是一个有效的 HTML 文档。

这里需要做的第一件事情就是确定这个页面的初始状态。为此，首先要弄清楚用户是否正在从书签中加载这个站点。可以通过查看 URL 来进行这项检查。如果该 URL 中含有一个带有状态名称和值的散列，那么这个访问者必定是通过收藏的链接来访问该站点。完成这项检查的最佳做法是使用 getBookmarkedState 方法。

```
var bookmarkedState = YH.getBookmarkedState("step");
```

一旦 getBookmarkedState 方法返回它的值，下一件要做的事情就是接收这个返回值，或者设置一个默认值作为备用值。

```
var initialState = bookmarkedState || "choose";
```

如果事实证明该访问者并非通过收藏的链接访问网站，就设置默认值 choose。否则，就使用收藏的链接包含的状态。我们已经确定了页面的状态，接下来就需要使用 History Manager 注册一个或多个模块。需要重点注意的是，一旦该管理器初始化完毕，就不能再注册模块，因此现在就要做这项工作。

```
YH.register("step", initialState, YX.stateChangeHandler);
```

使用 History Manager 注册一个模块有点类似于使用 addListener 函数添加一个事件处理程序。第一个参数是用来标识模块的名称(这个名称最终将用作 URL 名/值对中的名称)。第二个参数是该模块的初始状态(也就是 URL 名/值对中的值)。第三个参数是一个回调函数，每当模块的状态改变时，History Manager 都会调用这个函数。最后，与 addListener 一样，我们可以通过第四个和第五个参数来传递任意数据对象以及设置回调函数的执行作用域。

接下来，我们需要初始化 History Manager。这里的麻烦在于，如果浏览器不支持，那么它可能抛出异常。因此，YUI 团队建议将这个初始化过程包装进 try/catch 语句中，以避免可能出现的错误消息。实际上，如果浏览器不支持，那么可以将另一个可选的操作过程以编程方式写入 catch 子句中。对于这个示例而言，我们只是静默地失败。

```
try {
    YH.initialize("yui-history-field", "yui-history-iframe");
} catch (e) {
    //Browser not supported
}
```

既然已经设置好 History Manager，接下来就需要把页面中实际的导航链接连接起来以使用它。因此，使用一个简单的 addListener 调用，预订流程中的所有导航链接都连接到 stepClicked 函数。该函数为 History Manager 获取状态，然后通过它的 navigate 函数来设置该状态。

```
YE.addListener(links, "click", YX.stepClicked);
```

在这里，stepClicked 读取所单击链接的 href 属性值，并使用它找到的 URL 的过滤版本作为 step 模块的状态。

```
YX.stepClicked = function (e) {
    var state = this.href.replace(/\/$/, "");
    state = state.substring(state.lastIndexOf("/") + 1);
    try {
        YH.navigate("step", state);
    } catch (e) {
        //History manager not initialized
    }
    YE.preventDefault(e);
};
```

需要再次强调的是，如果没有成功初始化，那么 navigate 函数可能会失败并抛出异常，这正是使用 try/catch 块将其包装起来的原因。

YUI 3 中的新增功能

YUI 3 处理事件的方式与以前的版本不同。YUI 3 并没有映射到浏览器的原生事件处理方案，而是将事件处理程序包装起来并自行触发它们。这样做带来的一个好处就是，我们曾经提到的 Internet Explorer 中的先进先出问题就会自动得以解决。YUI 3 还允许模拟常见的事件，例如单击、双击、鼠标滑过/释放/按下/滑出/移动。换言之，我们可以使用代码模拟在一个元素上的鼠标单击操作，而并不需要用户实际完成这个操作。

8.5　本章小结

像 YUI 这样的库不仅为程序员解决了棘手的跨浏览器问题，而且它还扩展了 JavaScript 在基于事件编程方面的有限功能。首先，我们能够将多个事件处理程序附加到元素中并随意地将它们分离，这要领先于 JavaScript 目前提供的功能。类似地，我们能够创建并触发自定义事件，这为我们以前没有考虑过的各种可能性打开了便利之门。

最重要的是，在使用基于 JavaScript 并大量运用 AJAX 的用户界面已经屡见不鲜的今天，能够以一种可靠的方式管理浏览器历史是非常有意义的。否则，在浏览器后退按钮上的一次简单的、无意识的单击就会破坏一个极佳的 Web 应用程序。

第 9 章

使用动画和拖放

对于用户在与应用程序交互的过程中期望看到什么以及能够做些什么，现代操作系统的图形用户界面(以及桌面应用程序)已经形成了约定。例如，人们本能地期望能够拖放屏幕上的元素。

现代网站越来越向这些使用约定看齐。然而到目前为止，动画仍然不是 JavaScript 的强项。这并不是因为它不能实现动画，而是因为它缺乏原生的动画函数。例如，在 style 对象中就没有 move、fade 或 bounce 方法。这使得程序员(以及设计师)不太可能利用这些约定来构建 Web 应用程序。然而，JavaScript 不只是能够完成这项任务。但是，我们真正需要的(同时也正是 YUI 库所提供的)功能是，将原始功能包装进对程序员更友好的类和对象中。

例如，声明一个元素从一个位置移动一定像素到另一个位置就要比编写一个自定义例程来完成这项任务容易得多。YUI 的 Animation 组件提供了一组通用但可定制性极强的类，使得 DOM 元素的动画制作变成一件可以轻易完成的事情。

本章内容简介：

- 组合基本的动画序列
- 平滑动画路径和运动
- 带有拖放功能的交互动画

9.1 组合基本的动画序列

YUI Animation 组件划分成 4 个主要的类：Anim、Motion、Scroll 和 ColorAnim。实际上，Anim 是动画的基类，而其他 3 个类则是为处理不同类型的动画而设计的子类。Anim 基类完成了绝大部分的基本工作，而这些子类基本上利用该引擎，添加了一些新方法。因此，这个动画引擎实际上负责所有基本的细节，例如确保动画在任何给定时刻都能报告自己的位置；确保提供关键事件，这样就可以使用回调函数订阅这些事件；确保提供一个用来管理定时和帧的框架；以及确保动画受到控制而且准时，这样它就能够在预期的时间完成。

最终，Anim 基类所做的工作就是根据特定的缓动公式(easing formula)在特定的时间帧内递增或递减数值。因此在理论上，浏览器中任何依靠数值表示显示属性的元素都能够制作动画。这正是为什么我们不仅可以对元素的尺寸和位置制作动画，而且可以对其颜色制作动画。

9.1.1 Anim 类

要想使用 YUI 制作 DOM 元素的动画，首先需要实例化一个动画对象，这个对象可用于针对元素 CSS 样式的基于尺寸的属性制作动画(通过 style 对象)。下面是一个简单的示例：

```
var anim = new YAHOO.util.Anim("foo");
```

这段代码创建一个动画对象，但它并没有完成其他任务。要想使其真正变成动画，还需要调用 animate 方法。

```
anim.animate();
```

但是，此时仍然不会有任何动画效果，因为没有指明需要针对元素的什么属性制作动画。由于动画在深度和广度上存在着各种可能性，因此通过一个对象字面值来传递这些选项。这样，我们就可以指定一个或多个不同的动画，而不必让 API 滥用大量无用的参数。这个对象字面值的深度有两级。第一级定义要对元素的什么方面制作动画，例如高度、宽度和位置，第二级定义制作动画时实际使用的单位(例如多少像素)。

例如，在下面的示例中，我们将一个元素的宽度通过制作动画改成 100 像素：

```
var anim = new YAHOO.util.Anim("foo",
{
    width: {
        to: 100
    }
});
anim.animate();
```

默认动画单位是像素(px)，但是可以像下面这样重写该单位：

```
var anim = new YAHOO.util.Anim("foo",
{
    width: {
        to: 100,
        unit: "em"
    }
});
anim.animate();
```

下面的示例演示了基本的高度和宽度动画。

```
<html>
    <head>
        <title>Anim</title>
        <style type="text/css">
            #banner {
                background: #f00;
                border: solid 10px #ff0;
```

```
            color: #fff;
            text-align: center;
            width: 350px;
            height: 50px;
            overflow: hidden;
        }
        #banner h1 {
            font-size: 64px;
            margin-top: 42px;
        }
        #banner p {
            font-size: 32px;
        }
    </style>
</head>
<body>
    <div id="banner">
        <h1>YUI Wrox!</h1>
        <p>A simple YUI Animation</p>
    </div>
    <script src="yahoo-dom-event.js"></script>
    <script src="animation-min.js"></script>
    <script>
        YAHOO.example.animation = function () {
            var anim = new YAHOO.util.Anim("banner",
                {
                    width: {to: 976},
                    height: {to: 230}
                },
                1.5,
                YAHOO.util.Easing.easeOutStrong);
            YAHOO.util.Event.addListener(
                "banner",
                "mouseover",
                anim.animate,
                anim,
                true);
        }();
    </script>
</body>
</html>
```

在这个示例中，我们定义了一个 ID 为 banner 的 div 元素。分别将它的高度和宽度设置为 50 像素和 350 像素。将动画对象绑定到这个元素，并指示它通过制作动画把元素宽度设置为 976 像素，将高度设置为 230 像素。这个动画由 mouseover 事件触发(参见图 9-1)。

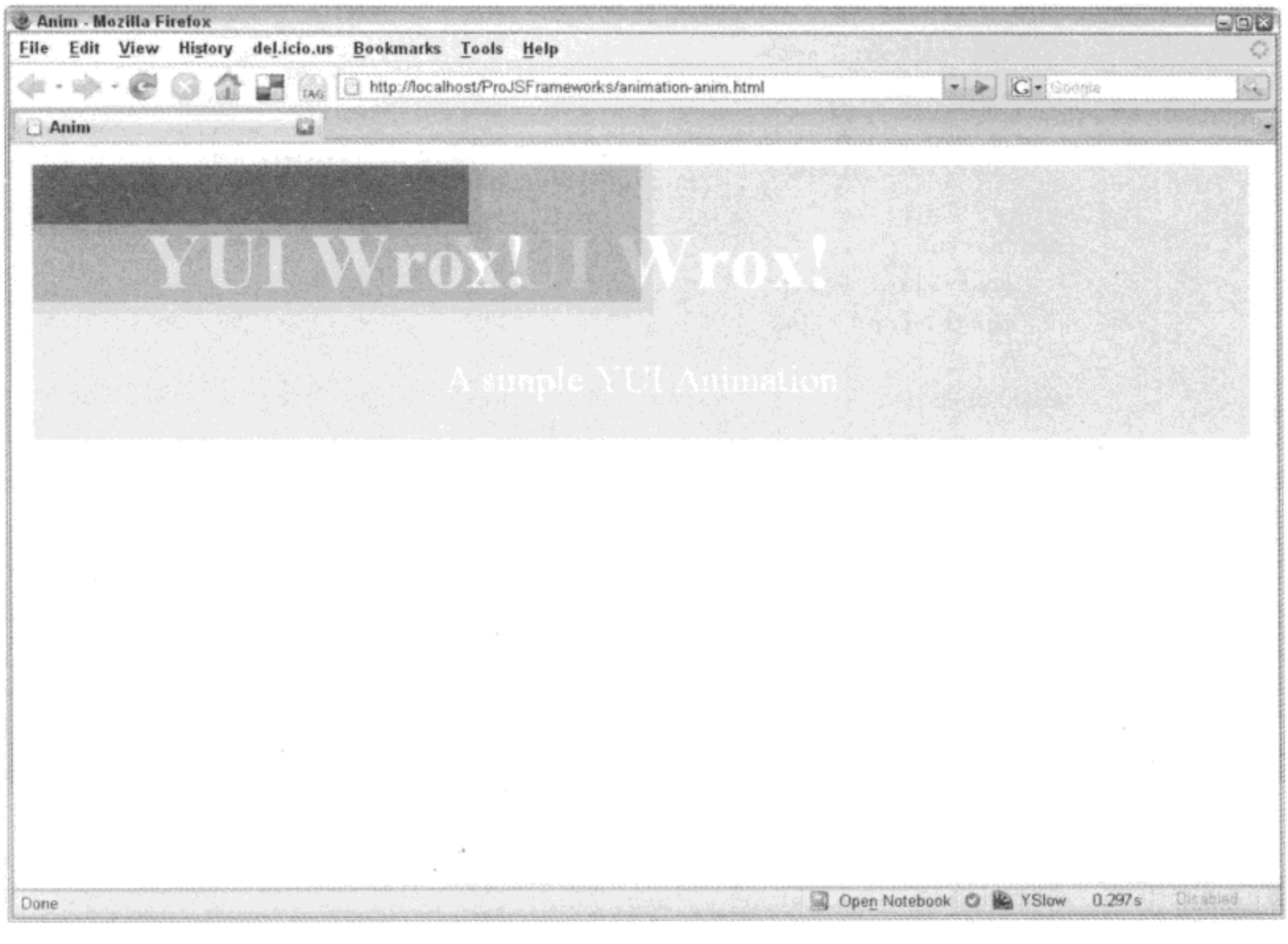


图　9-1

动画对象计算出从它的初始起点到它的结束终点需要多少帧，然后每一帧相应地提升宽度和高度值。如果宽度值递增的量要远多于高度，那么每一帧将使宽度值提升比高度值更大的量。

还有另外两个参数传给 Anim 构造函数。其中一个参数是值 1.5，这是该动画的预期持续时间，单位为秒；另一个参数是要应用于该动画的缓动效果。

事件

Anim 类有一些内置自定义事件，我们可以使用回调函数订阅这些事件。这样在动画序列中的关键时刻就可以触发这些代码。可用的事件有 3 个，分别为 onStart、onTween 和 onComplete。每个事件的名称可以说明它们各自的作用：当动画开始时，就会调用所有订阅 onStart 事件的回调函数；每次动画向前步进一帧时，就会调用所有订阅 onTween 事件的回调函数；而当动画结束时，就会调用所有订阅 onComplete 事件的回调函数。所有动画子类以及所有继承自 Anim 基类的子类都可以使用这些自定义事件。

一个订阅 onTween 自定义事件的回调函数的示例是探测元素冲突。每次动画把元素向前移动一帧时，这个回调函数都会将该元素的位置与它周围的其他对象进行核对。如果检测到冲突，那么它可以进一步触发与其冲突的对象中的另一个函数，这样就能够将其移开。

9.1.2　Motion 类

尽管使用 Anim 类可以通过操作元素的 top、right、bottom 和 left 值来四处移动元素，但这些值是相对于元素自身的。使用 Motion 子类可以相对于页面来指定对象要移向的点。这样，元素所在的位置就变得无关紧要，它最终将位于页面上指定的 x 和 y 坐标处。下面是一个示例：

```
<html>
    <head>
        <title>Motion</title>
        <style type="text/css">
            #doc {
                width: 200px;
                position: absolute;
                top: 0;
                right: 0;
                background: #000;
                padding: 10px;
            }
            #box {
                background: #f00;
                border: solid 10px #ff0;
                color: #fff;
                width: 100px;
                text-align: center;
                font-weight: bold;
            }
        </style>
    </head>
    <body>
        <div id="doc">
            <p id="box">YUI Wrox!</p>
        </div>
        <script src="yahoo-dom-event.js"></script>
        <script src="animation-min.js"></script>
        <script>
            YAHOO.example.motion = function () {
                var anim = new YAHOO.util.Motion("box",
                    {
                        points: {to: [350, 100]}
                    },
                    1.5,
                    YAHOO.util.Easing.easeOutStrong);
                YAHOO.util.Event.addListener(
                    "box",
                    "mouseover",
                    anim.animate,
                    anim,
                    true);
            }();
        </script>
```

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

在这个示例中，我们在 div 元素中设置了一个段落元素。通过绝对定位方式把这个 div 元素放在浏览器视口的右上角。自然地，该 div 元素会带上内部的段落元素(ID 为 box)。然后，在这个段落元素上设置一个 Motion 动画，这样当鼠标滑过它上方时，这个段落元素就会移到页面上的坐标(350，100)处(参见图 9-2)。不管开始位于何处，最终该元素都会出现在这里。因此，修改 CSS 规则，将“right: 0;”改为“left: 0;”，在开始时就将这个元素放到屏幕最左侧。即使是这样，一旦鼠标滑过该元素上方，最终它仍然会移到坐标(350，100)处。

图　9-2

我们通过 points 参数来实现 Motion 动画。这里并没有指定 height、width、top、right、bottom 或 down 值，而是只使用 points 参数。而且，我们也没有传递一个应用于特定属性的值，而是通过一个数组来传递 x/y 坐标。

虽然在 Anim 对象中以如下方式使用相应的参数：

```
{
    left: {to: 350},
    top: {to: 100}
}
```

但是在 Motion 对象中以如下方式使用参数。

```
{
    points: {to: [350, 100]}
}
```

9.1.3　Scroll 类

Scroll 动画子类用于制作溢出元素的动画，特别是溢出元素的自动化滚动。正常情况下，如果 CSS 中设置的元素 height 值小于它的自然高度，而且对其应用了 CSS 规则 overflow，那么 Scroll 子类就能够对其制作动画。下面的示例演示 div 元素内的文本段落如何滚动。

```
<html>
    <head>
        <title>Scroll</title>
        <style type="text/css">
            #box {
                height: 150px;
                width: 200px;
                overflow: auto;
            }
        </style>
    </head>
    <body>
        <div id="box">
            <h1>YUI Wrox!</h1>
            <p>Lorem ipsum dolor sit amet, consectetuer
            adipiscing elit. Donec fermentum, neque et
            ultrices dignissim, est est scelerisque erat,
            sed ornare nisl orci quis tellus. Donec quis
            lacus quis nibh consectetuer dictum. Duis
            fermentum, ligula condimentum congue bibendum,
            dui dolor vehicula lorem, sed congue nisl
            tellus id diam. Sed nulla sem, lacinia vel,
            ullamcorper at, auctor ut, dolor. Praesent
            turpis quam, posuere a, posuere sed, fermentum
            eu, risus. Integer luctus. Maecenas dolor.
            Donec vehicula.<p>
    </div>
    <script src="yahoo-dom-event.js"></script>
    <script src="animation-min.js"></script>
    <script>
        YAHOO.example.scroll = function () {
            var anim = new YAHOO.util.Scroll("box",
                {
                    scroll: {to: [0, 400]}
                },
                1.5,
                YAHOO.util.Easing.easeOutStrong);
            YAHOO.util.Event.addListener(
                "box",
                "mouseover",
```

```
                    anim.animate,
                    anim,
                    true);
            }();
        </script>
    </body>
</html>
```

这里，在 YAHOO.example 名称空间下建立了一个名为 Scroll 的自执行函数。一旦执行该函数，就会创建一个新的 Scroll 动画。这个动画对象附加到 ID 为 box 的元素。这个对象的属性参数是一个包含动画指令的对象，它指示动画在 x 轴上滚动 0 像素，在 y 轴上滚动 400 像素。同时，设置这个动画运行 1.5 秒，并使用缓动方法 easeOutStrong 完成缓动。最后，将这个对象的 animate 方法附加到 div 元素作为 mouseover 事件的事件处理程序。因此，当光标在 div 元素上悬停时，该元素就会滚动 400 像素(参见图 9-3)。

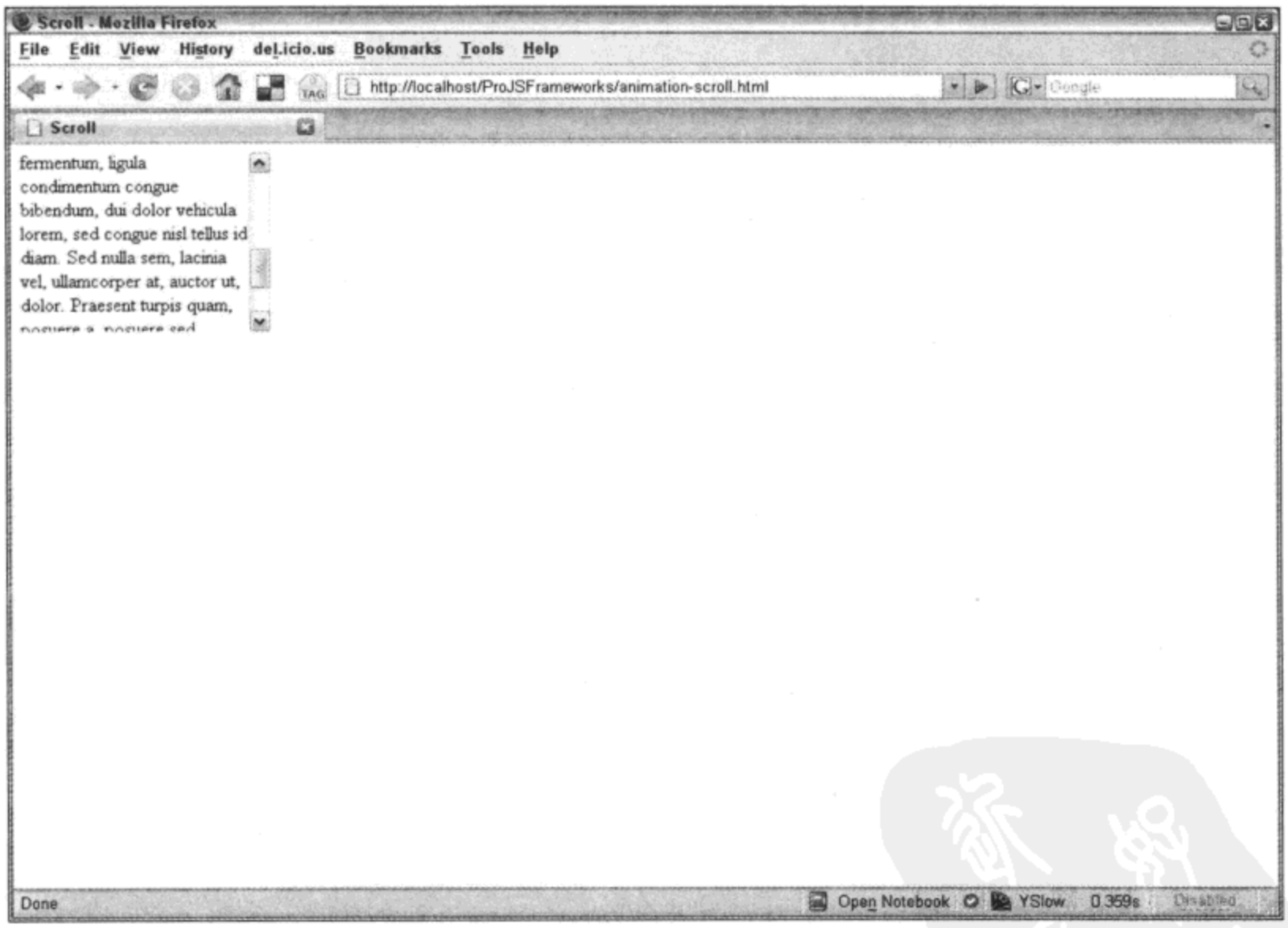


图 9-3

9.1.4 ColorAnim 类

对颜色制作动画曾经为网站增加了亮丽的外表，尽管很多时候这种动画制作得过于夸张，而且华而不实。但是，在“AJAX 驱动”的 Web 上，对颜色制作动画是构建易用界面的关键。尤其是通过 AJAX 将文档中修改的内容高亮显示，然后将其淡出，这已经成为提示访问者页面上的内

容已经发生变化的视觉信号。这种设计的最常见实现方式是将改变的内容以黄色高亮显示，然后在 1~2 秒之后淡出为白色(或页面的背景色)。但是，我们可以直接通过 Anim 对象来淡出元素的透明度，因为透明度是一个可以通过 style 对象访问的属性。

ColorAnim 子类利用了 Anim 类的主引擎，而且只为其传入一个用来将一种颜色值递增或递减到另一种颜色值的方法。CSS 颜色值有两种不同的形式，即 RGB 和 Hex。下面是 CSS 中的 RGB 值示例：

```
#foo {
    background-color: rgb(255, 255, 255); /* white */
}
```

下面是同一个示例，但是采用 Hex 形式：

```
#foo {
    background-color: #ffffff; /* white */
}
```

在为 ColorAnim 子类提供颜色值时可以采用以下 4 种形式之一：ffff、#ffff、[255, 255, 255]或 rgb(255, 255, 255)。获取这些值并将其传给一个公有方法 parseColor，该方法将试图转换颜色值并以 rgb 三元组的形式(也就是[255, 255, 255]这种形式)返回。这种规范化允许在内部以同样的方式处理颜色值，而不用担心它们是以什么形式传入的。

下面是一个将会用到颜色动画的实际用例，它是一个 AJAX 驱动的评论表单(为了节省代码所占篇幅，这里伪造了 AJAX 部分)。访问者提交的评论将动态地添加到页面上现有的评论列表中。为了指出评论内容确实已经添加，在把新评论插入到页面中之前为其设置了黄色背景色。插入该评论一秒之后，将其淡出为白色，从而融入背景。

```
<html>
    <head>
        <title>ColorAnim Comment Demo</title>
        <link rel="stylesheet" type="text/css" href="reset-fonts-grids.css" />
        <link rel="stylesheet" type="text/css" href="base-min.css" />
    </head>
    <body>
        <div id="doc">
            <div id="hd">
                <h1>ColorAnim Comment Demo</h1>
            </div>
            <div id="bd">
                <form id="leave-a-comment" method="post" action="/leaveComment/">
                    <label for="name">Name</label><br />
                    <input type="text" id="name" /><br />
                    <label for="comment">Comment</label><br/>
                    <textarea id="comment"></textarea><br />
                    <input type="submit" value="Submit comment!" />
                </form>
                <h2>Comments</h2>
                <ol id="comments">
                    <li>First!!! -- Anonymous Coward</li>
```

```
            <li>This site is awesome! -- John Doe</li>
            <li>Yahoo! -- Me</li>
        </ol>
    </div>
</div>
<script src="yahoo-dom-event.js"></script>
<script src="animation-min.js"></script>
<script>
    YAHOO.example.colorAnim = function () {
        function addComment(e) {
            var li = document.createElement("li");
            li.appendChild(
                document.createTextNode(
                    comment.value + " -- " + name.value));
            YAHOO.util.Dom.setStyle(li, "background-color", "#ffff00");
            comments.appendChild(li);
            name.value = "";
            comment.value = "";
            var cAnim = new YAHOO.util.ColorAnim(li, {
                    backgroundColor: {
                        to: "#ffffff"
                    }
                }, 1, YAHOO.util.Easing.easeOut);
            setTimeout(function () {
                cAnim.animate();
            }, 1000);
                // AJAX code goes here
                YAHOO.util.Event.preventDefault(e);
            };

            var name = YAHOO.util.Dom.get("name");
            var comment = YAHOO.util.Dom.get("comment");
            var comments = YAHOO.util.Dom.get("comments");
            YAHOO.util.Event.addListener(
                "leave-a-comment", "submit", addComment);
        }();
    </script>
</body>
</html>
```

这段代码所做的第一件事情就是在 YAHOO.example 名称空间下建立一个自执行函数 colorAnim。这将确保该代码创建的所有变量的作用域都是 colorAnim 函数，而不是全局作用域。然后，该代码建立 addC nent 函数，每次添加新评论时都会调用这个函数。该函数负责创建一个新的列表项，将其背景 设置为黄色，填充访问者的评论和姓名，然后将其添加到评论列表中。该代码将这个列表项添加到有序列表之后，它把表单字段清空并建立一个颜色动画。它设置该动画从当前背景色(因为这里没有指定何种颜色)变成白色。然后设置 setTimeout，在一秒钟之后触发这个动画对象的 animate 方法。

可以根据需要指定从何种颜色开始制作动画，例如从红色到白色。

```
backgroundColor: {
    from: "#ff0000",
    to: "#ffffff"
}
```

还可以在同一时刻调整多种颜色。在下面的代码中，将背景色设置为白色，而将前景色设置为红色：

```
backgroundColor: {
    to: "#ffffff"
},
color: {
    to: "#ff0000"
}
```

很明显，还可以向动画对象传入多个可选属性。可以同时设置 to 和 from 属性，也可以只设置 to 属性。可以同时调整 color 和 backgroundColor 属性，也可以只调整其中的一个属性。实际上，我们甚至可以通过 borderTopColor、borderRightColor、borderBottomColor 和 borderLeftColor 属性来设置边框颜色。必须单独设置各个边框(顶部、右侧、底部和左侧)，这是因为没有一个简单的包含所有边框的 borderColor 属性。此外，我们只能在设置了边框样式值的元素上设置边框颜色(参见图 9-4)。

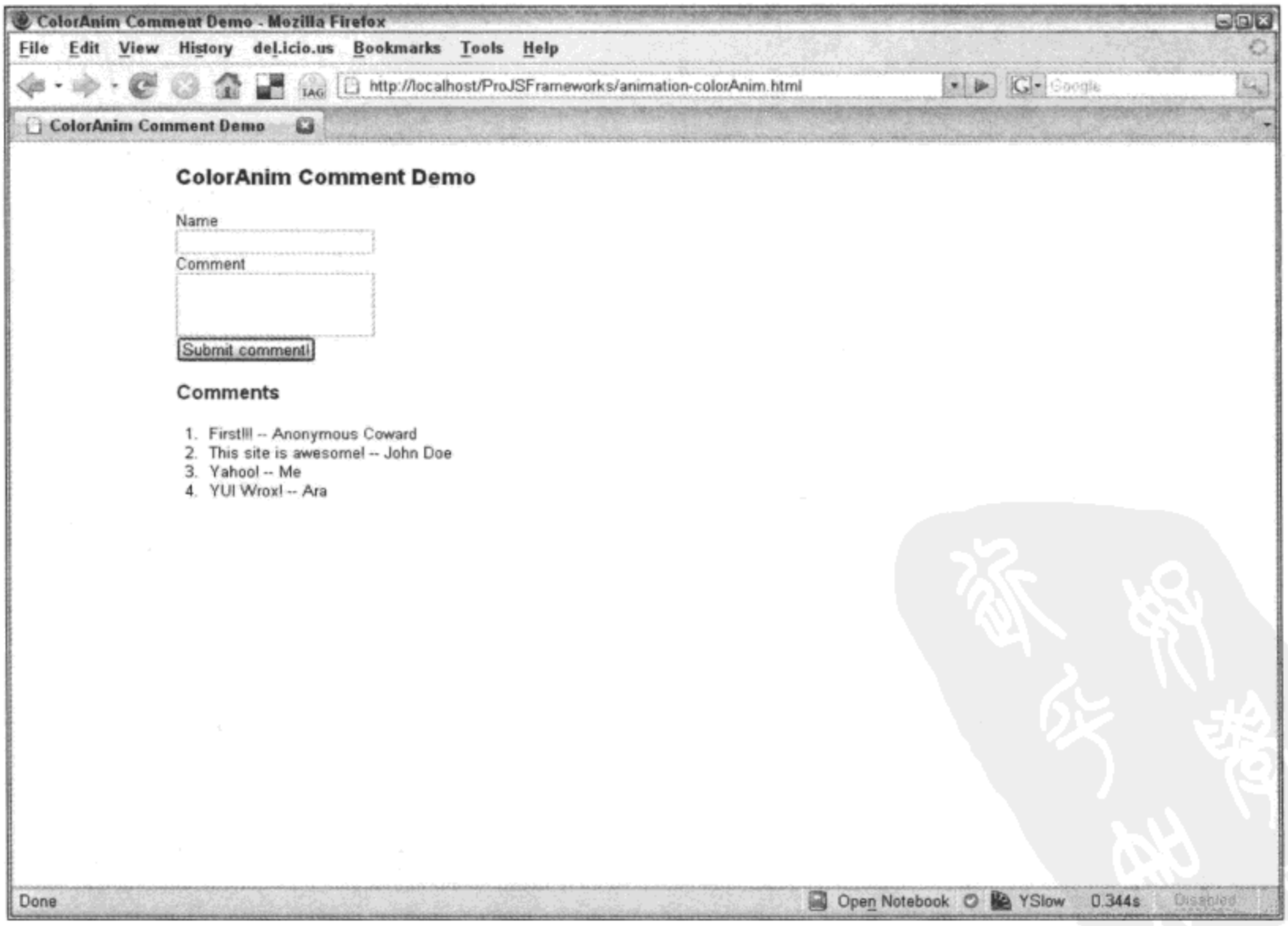


图　9-4

9.2 平滑动画路径和运动

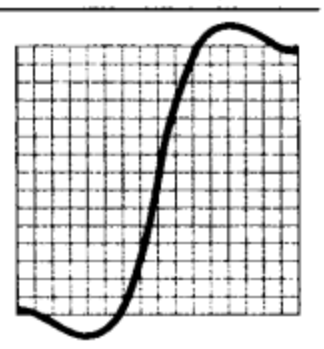
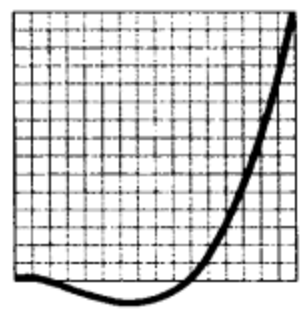
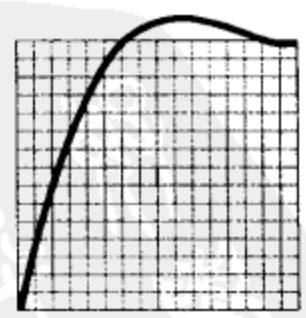
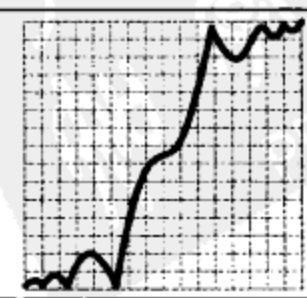
以相同的速度从起点运行到终点的动画在人们看来显得不够自然。例如，人类运动开始时比较缓慢，之后越来越快，然后在停止前慢下来。当然，还有其他类型的自然运动，例如物体落在下时弹回，或者在停止运动前来回弹跳。人类的大脑习惯了这类运动，因为它们是自然发生的运动。而计算机化的、从起点到终点始终如一的运动就显得很不自然。

9.2.1 缓动

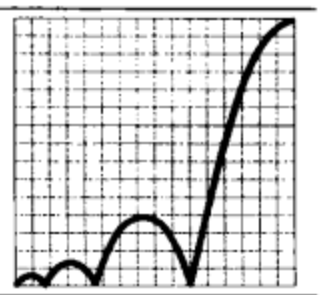
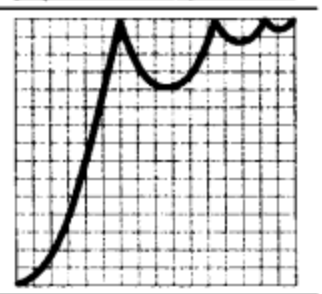
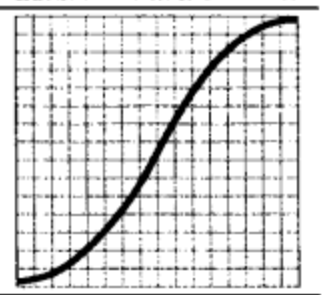
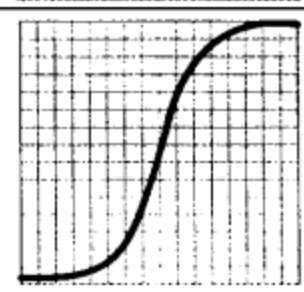
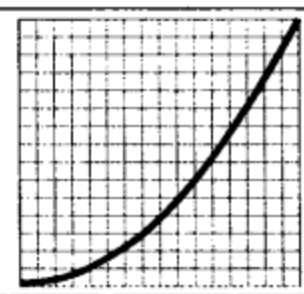
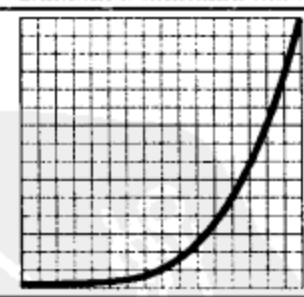
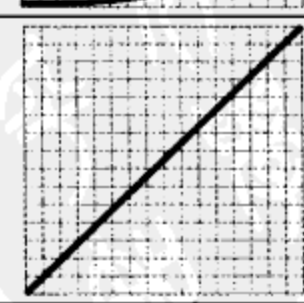
下面开始讲解缓动。缓动是计算机通过将数学公式应用于移动物体来模拟自然发生的运动的一种方式。如果为动画对象指定了一个缓动函数，那么每一帧的位置计算过程都会通过这个缓动函数，该函数会根据自己的公式来调整这一帧的位置。

表 9-1 给出了所有可用的缓动效果，每种效果都带有一段简单的说明(摘自 API 文档)和一个正在应用该公式的动画的图形表示。

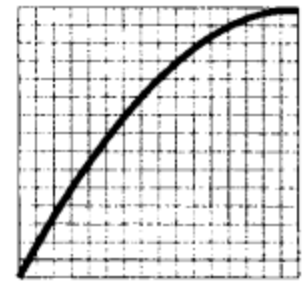
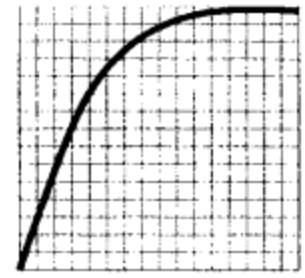
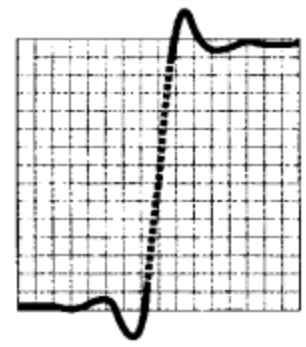
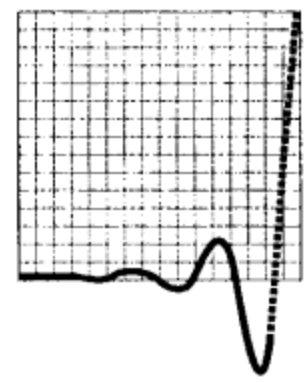
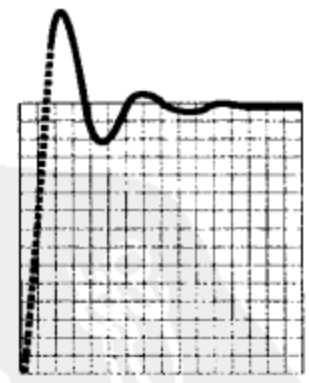
表 9-1

效　果	说　明	图形表示
backBoth	轻微地后退，然后调转方向，超越终点，然后再调转方向并返回到终点	
backIn	轻微地后退，然后调转方向并运动到终点	
backOut	超越终点，然后调转方向并返回到终点	
bounceBoth	在起点和终点都做弹回运动	

(续表)

效　果	说　明	图 形 表 示
bounceIn	在起点做弹回运动	
bounceOut	在终点做弹回运动	
easeBoth	缓慢地开始，然后减速进入终点(二次曲线)	
easeBothStrong	缓慢地开始，然后减速进入终点(四次曲线)	
easeIn	缓慢地开始，然后加速进入终点(二次曲线)	
easeInStrong	缓慢地开始，然后加速进入终点(四次曲线)	
easeNone	在两点之间匀速前进	

(续表)

效　果	说　明	图 形 表 示
easeOut	快速开始，然后减速进入终点(二次曲线)	
easeOutStrong	快速开始，然后减速进入终点(四次曲线)	
elasticBoth	在起点和终点都做弹性运动	
elasticIn	在起点做弹性运动	
elasticOut	在终点做弹性运动	

来源：YUI API 文档

9.2.2　曲线路径(贝塞尔曲线)

有时我们需要 Motion 动画不仅是沿着一条直线制作动画，贝塞尔曲线使这种情况成为可能。

这些曲线在诸如 Adobe Illustrator 这类的矢量绘图程序中常常称为路径(path)。向 YUI Motion 动画添加曲线路径只需要向现有的属性参数添加另一个属性。给定 from 和 to 属性，我们通过 control 参数来指定曲线路径。

下面是一个简单的 Motion 动画示例：

```
var anim = new YAHOO.util.Motion("obj", {
    points: {
        to: [400, 450]
    }
}, 3);
```

下面是一个带有曲线路径的 Motion 动画示例：

```
var anim = new YAHOO.util.Motion("obj", {
    points: {
        to: [400, 450],
        control: [
            [10, 175],
            [650, 30],
            [750, 250],
            [450, 500],
            [125, 350]
        ]
    }
}, 3);
```

在这个动画中，ID 为 obj 的元素将从它在屏幕上的原始位置行进到坐标(400，450)处。在行进过程中，它将受到 control 数组内指定的控制点的“影响”。该数组中有一系列子数组，每个子数组表示一个点的 x 坐标和 y 坐标。

下面是使用该动画的代码示例：

```
<html>
    <head>
        <title>Bezier Demo</title>
        <link rel="stylesheet" type="text/css" href="reset-fonts-grids.css" />
        <link rel="stylesheet" type="text/css" href="base-min.css" />
        <style type="text/css">
            #obj,
            .point,
            .ctrl,
            .end {
                font-size: 77%;
                height: 1.25em;
                width: 1.25em;
                overflow: visible;
                position: absolute;
```

```
            }
            #obj {
                z-index: 1;
            }
            #obj,
            .point {
                background: #f00;
            }
            #obj,
            .ctrl,
            .end {
                border: solid 2px #000;
            }
            .end {
                background: #000;
            }
        </style>
    </head>
    <body>
        <div id="doc">
            <div id="hd">
                <h1>Bezier Demo</h1>
            </div>
            <div id="bd">
                <p>Once clicked, the red square at the end of this sentence will
                travel to its end point (black square) while being influenced by
                control points (hollow squares) along the way.
                <span id="obj"></span></p>
            </div>
        </div>
        <script src="yahoo-dom-event.js"></script>
        <script src="animation-min.js"></script>
        <script>
            YAHOO.example.bezier = function () {
                function setPoint(xy, cn, txt) {
                    var pt = document.createElement("div");
                    pt.className = cn || "point";
                    if (typeof txt !== "undefined") {
                        pt.appendChild(document.createTextNode(txt));
                    }
                    document.body.appendChild(pt);
                    YAHOO.util.Dom.setXY(pt, xy);
                    if (pt.className === "point") {
                        YAHOO.util.Dom.setStyle(pt, "opacity", "0.25");
                    }
```

```
};
var anim = new YAHOO.util.Motion("obj", {
    points: {
        to: [400, 450],
        control: [
            [10, 175],
            [650, 30],
            [750, 250],
            [450, 500],
            [125, 350]
        ]
    }
}, 3);
anim.onTween.subscribe(function () {
    setPoint(YAHOO.util.Dom.getXY(this.getEl()));
});
for (var i = 0; anim.attributes.points.control[i]; i += 1) {
    setPoint(anim.attributes.points.control[i], "ctrl", i + 1);
}
setPoint(anim.attributes.points.to, "end");
YAHOO.util.Event.addListener(
    "obj",
    "click",
                anim.animate,
                anim,
                true);
        }();
    </script>
  </body>
</html>
```

这段代码创建一个 Motion 动画，并将其 animate 方法设置为 ID 是 obj 的 span 元素的事件处理程序。这样一旦单击该元素，就可以对其制作动画。这里的代码利用动画对象的 onTween 事件来调用 setPoint 函数，该函数基本上是在创建动画的 span 元素后面留下一条踪迹。该代码还遍历控制点数组，同时在屏幕上设置几个用来表示它们的点。其作用就是显示这些点在图 9-5 和 9-6 中的位置。一旦单击 span 元素，这个动画就会把 span 元素(红色正方形)从它的起点(页面中第一句话的末尾)带到坐标(400，450)处(黑色正方形)。在这个过程中，它的路径将受到控制点(空白正方形)的影响。

图 9-5 给出了这个动画的外观。

单击 span 元素可以再次调用该动画。但是，这一次 span 元素的起点改变。因此，它行进的路径也将不同(参见图 9-6)。它将从一个点开始运动，最终又回到这个点，并且在整个运动过程中都会受到控制点的影响。

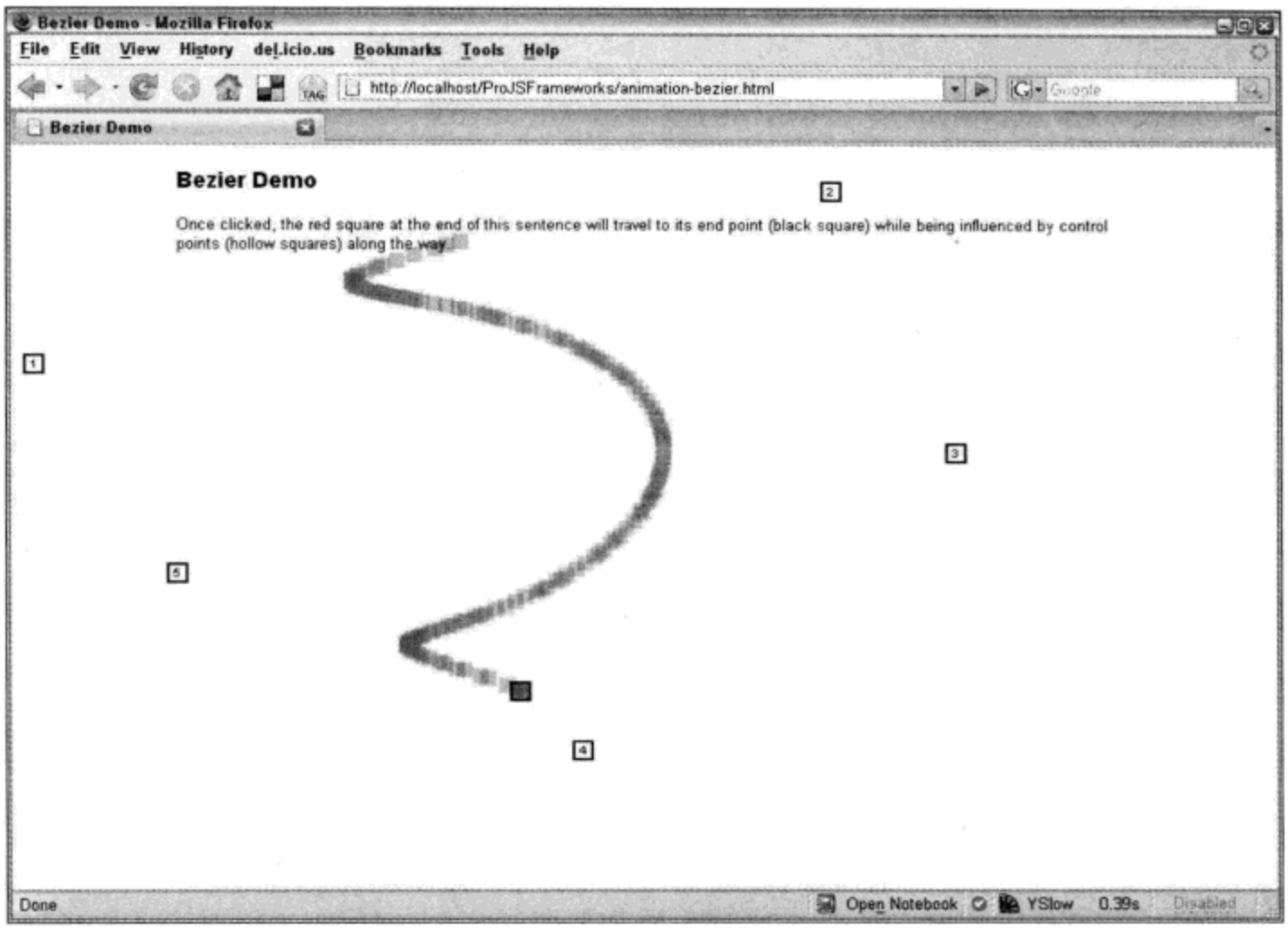

Bezier Demo - Mozilla Firefox
File Edit View History del.icio.us Bookmarks Tools Help
http://localhost/ProJSFrameworks/animation-bezier.html
Bezier Demo
Bezier Demo
Once clicked, the red square at the end of this sentence will travel to its end point (black square) while being influenced by control points (hollow squares) along the way.
1
2
3
4
5
Done
Open Notebook
YSlow
0.39s
Disabled

图　9-5

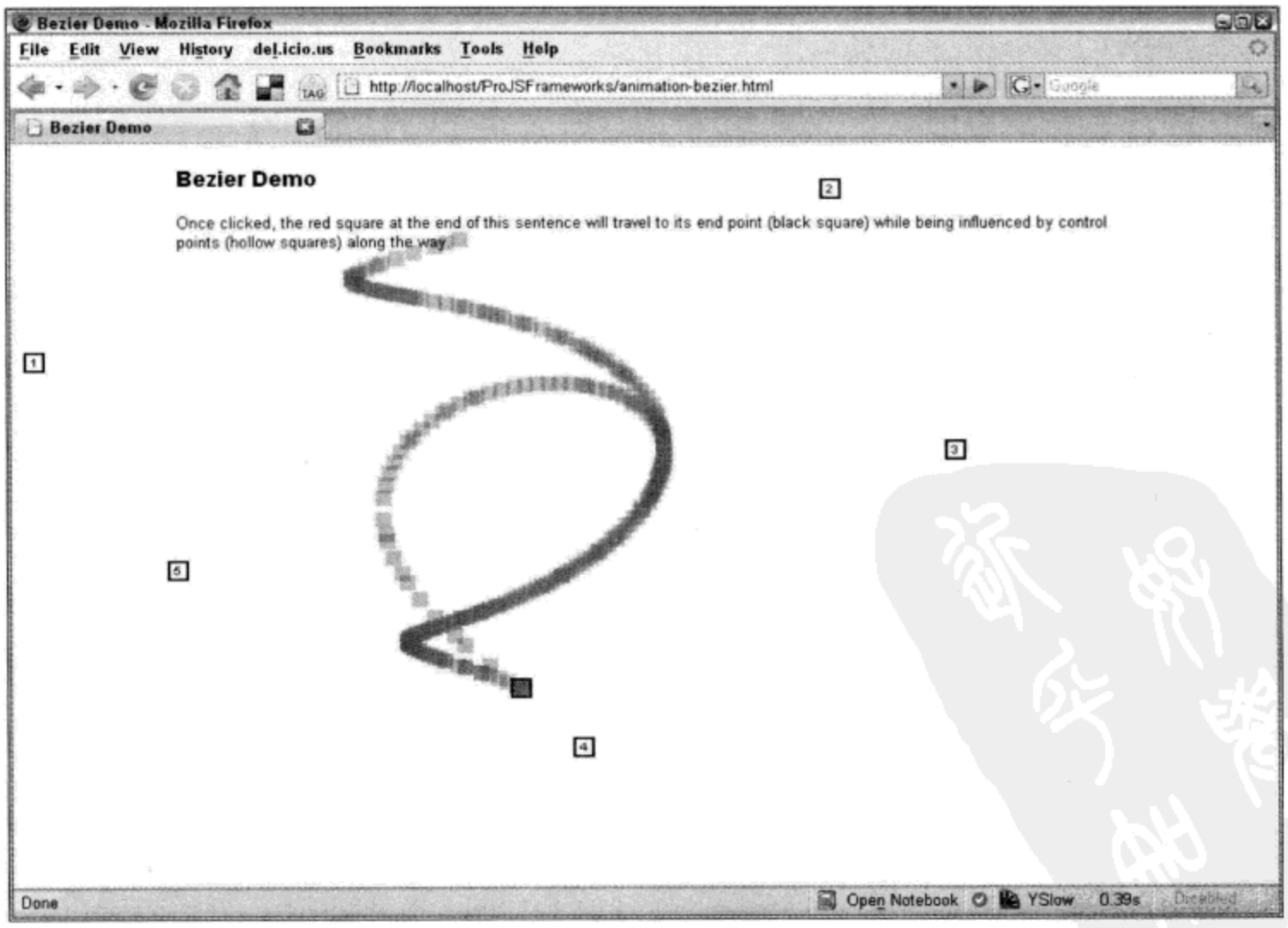

Bezier Demo - Mozilla Firefox
File Edit View History del.icio.us Bookmarks Tools Help
http://localhost/ProJSFrameworks/animation-bezier.html
Bezier Demo
Bezier Demo
Once clicked, the red square at the end of this sentence will travel to its end point (black square) while being influenced by control points (hollow squares) along the way.
1
2
3
4
5
Done
Open Notebook
YSlow
0.39s
Disabled

图　9-6

9.3　带有拖放功能的交互动画

图形用户界面的一个主要特征是可以拖放其中的项目。所有现代操作系统都遵循这个约定，这样它们的用户就可以随意地在屏幕上四处移动元素。现代网站也利用这个约定，这样它们的访问者就可以自由地交互，就像在操作系统中所做的那样。问题在于，JavaScript 没有内置拖放功能，而且编程实现它也是一项非常困难的任务。在这方面，使用库可以简化开发工作。YUI 有一个完整的 Drag & Drop Utility 组件，可用于帮助我们简化向网站中添加拖放功能时所需完成的工作。

9.3.1　DD

创建一个可拖动对象实际上非常简单，只需要使用将要成为可拖动对象的元素的 ID 来实例化一个 DD 对象即可。

```
var el = new YAHOO.util.DD("foo");
```

这一行代码会使 ID 为 foo 的元素成为可拖动对象。

YUI Drag & Drop Utility 的 DragDrop 基类的作用是作为进一步扩展的基础。换言之，直接通过 DragDrop 类实例化会返回一个没有多少功能的框架。这就是在上面的代码中通过 DD 类实例化的原因。DD 对象扩展 DragDrop 类并提供功能更加完整的对象。

9.3.2　DDProxy

默认情况下，Drag & Drop Utility 使正在拖动的元素随着光标在屏幕上移动。这种做法的麻烦之处在于，如果该元素的布局比较复杂，或者它的内容比较多，那么移动操作的性能最终将受到负面影响。这就是 DDProxy 发挥作用的地方。它与 DD 的工作方式类似，但是创建了一个独立的元素，用于在 mousemove 操作期间代表可拖动元素。这个动态生成的代理元素是 document.body 的直接子节点，它不包含任何内容。因此，对于只要拖动自身就会使用户体验变差的元素，就可以使用这个代理元素代表它们。该代理元素只会在开始拖动元素时出现，而且一旦操作完成就会隐藏。

下面的代码是一个简单的 Drag & Drop 示例：

```
<html>
    <head>
        <title>Drag &amp; Drop Demo</title>
        <link rel="stylesheet" type="text/css" href="reset-fonts-grids.css" />
        <link rel="stylesheet" type="text/css" href="base-min.css" />
        <style type="text/css">
            .box {
                border: solid 5px #000;
                height: 5em;
                width: 5em;
                margin: 1em;
                color: #fff;
                font-weight: bold;
```

```
            font-size: 123.1%;
            text-transform: uppercase;
            text-align: center;
        }
        .move {
            cursor: move;
        }
        #box1 {
            background: #300;
        }
        #box2 {
            background: #600;
        }
        #box3 {
            background: #900;
        }
        #box4 {
            background: #c00;
        }
        #box5 {
            background: #f00;
        }
    </style>
</head>
<body>
    <div id="doc">
        <div id="hd">
            <h1>Drag &amp;amp; Drop Demo</h1>
        </div>
        <div id="bd">
            <h2>Boxes</h2>
            <div class="box" id="box1">Box 1</div>
            <div class="box proxy" id="box2">Box 2</div>
            <div class="box" id="box3">Box 3</div>
            <div class="box proxy" id="box4">Box 4</div>
            <div class="box" id="box5">Box 5</div>
        </div>
    </div>
    <script src="yahoo-dom-event.js"></script>
    <script src="dragdrop-min.js"></script>
    <script>
        YAHOO.example.dd = function () {
            var boxes = YAHOO.util.Dom.getElementsByClassName("box");
            for (var i = 0; boxes[i]; i += 1) {
                var box = boxes[i];
                if (YAHOO.util.Dom.hasClass(box, "proxy")) {
                    var ddItem = new YAHOO.util.DDProxy(box);
                    box.innerHTML += " (proxy)";
                } else {
                    var ddItem = new YAHOO.util.DD(box);
```

```
                    }
                    ddItem.onMouseDown = function () {
                        var el = this.getEl();
                        if (el) {
                            YAHOO.util.Dom.addClass(el, "move");
                        }
                    };
                    ddItem.onMouseUp = function () {
                        var el = this.getEl();
                        if (el) {
                            YAHOO.util.Dom.removeClass(el, "move");
                        }
                    };
                }
            }();
        </script>
    </body>
</html>
```

这个代码示例获取页面中所有带有 CSS 类名 box 的元素，然后遍历它们并为每个元素实例化一个对象。但是在此之前，它检查该元素是否还具有 CSS 类名 proxy。如果有该类名，就为其实例化一个 DDProxy 对象。否则，就实例化 DD 对象。将 CSS 类名用作挂钩参数会使得这种程序的维护和配置变得更加简单。对于需要 DDProxy 对象的元素，代码还向元素的内容中添加了单词“(proxy)”，用于指示(出于演示的目的)哪些元素的行为有所不同。前面已经提及，DD 类扩展 DragDrop 类，后者包含了大量本身并不完成任何工作的事件处理程序。在拖放操作期间的关键时刻，就会调用这些函数。如果重写这些函数，那么它们就可以执行自定义代码。对于上面的示例而言，我们同时重写了 onMouseDown 和 onMouseUp 事件处理程序，分别用来向元素中添加和移除 move 类名。

表 9-2 列出了所有可用事件以及它们的 DragDrop、DD 和 DDProxy 说明(摘自 API 文档)：

表　9-2

事　　件	说　　明
b4DragDrop	在 dragDropEvent 事件之前引发
b4Drag	在 dragEvent 事件之前引发
b4DragOut	在 dragOutEvent 事件之前引发
b4DragOver	在 dragOverEvent 事件之前引发
b4EndDrag	在 endDragEvent 事件之前引发。如果返回 false，则取消 endDrag 事件
b4MouseDown	在引发 mouseDownEvent 事件之前提供访问 mousedown 事件的机会。如果返回 false，则取消拖动操作
b4StartDrag	在 startDragEvent 事件之前引发。如果返回 false，则取消 startDrag 事件
dragDrop	当把拖动的对象放到另一个对象上时引发该事件
dragEnter	当拖动的对象首次与另一个可停放的拖放对象交互时发生该事件
Drag	在拖动期间，每次引发 mousemove 事件都会发生该事件
dragOut	当一个拖动对象不再位于某个已经引发 onDragEnter 事件的对象上方时引发该事件

(续表)

事　　件	说　　明
dragOver	当在一个拖放对象上方时，每次引发 mousemove 事件都会发生该事件
endDrag	当已经初始化一次拖动操作之后(已经引发 startDrag 事件)，mouseup 事件会引发该事件
invalidDrop	当把拖动对象停放到某个没有可停放目标的位置时引发该事件
mouseDown	提供对 mousedown 事件的访问。mousedown 事件并非总是会导致拖动操作
mouseUp	当一次拖动操作结束时从 DragDropMgr 内部引发该事件
startDrag	在按下鼠标按键且已经达到拖动阈值时发生该事件。拖动阈值默认为鼠标移动 3 像素或者将鼠标按键持续按下 1 秒钟以上

图 9-7 给出了该示例实际运行时的情形：前 3 个方框已经移动完成，而第 4 个方框(DDProxy 类型的方框)正在拖动中。

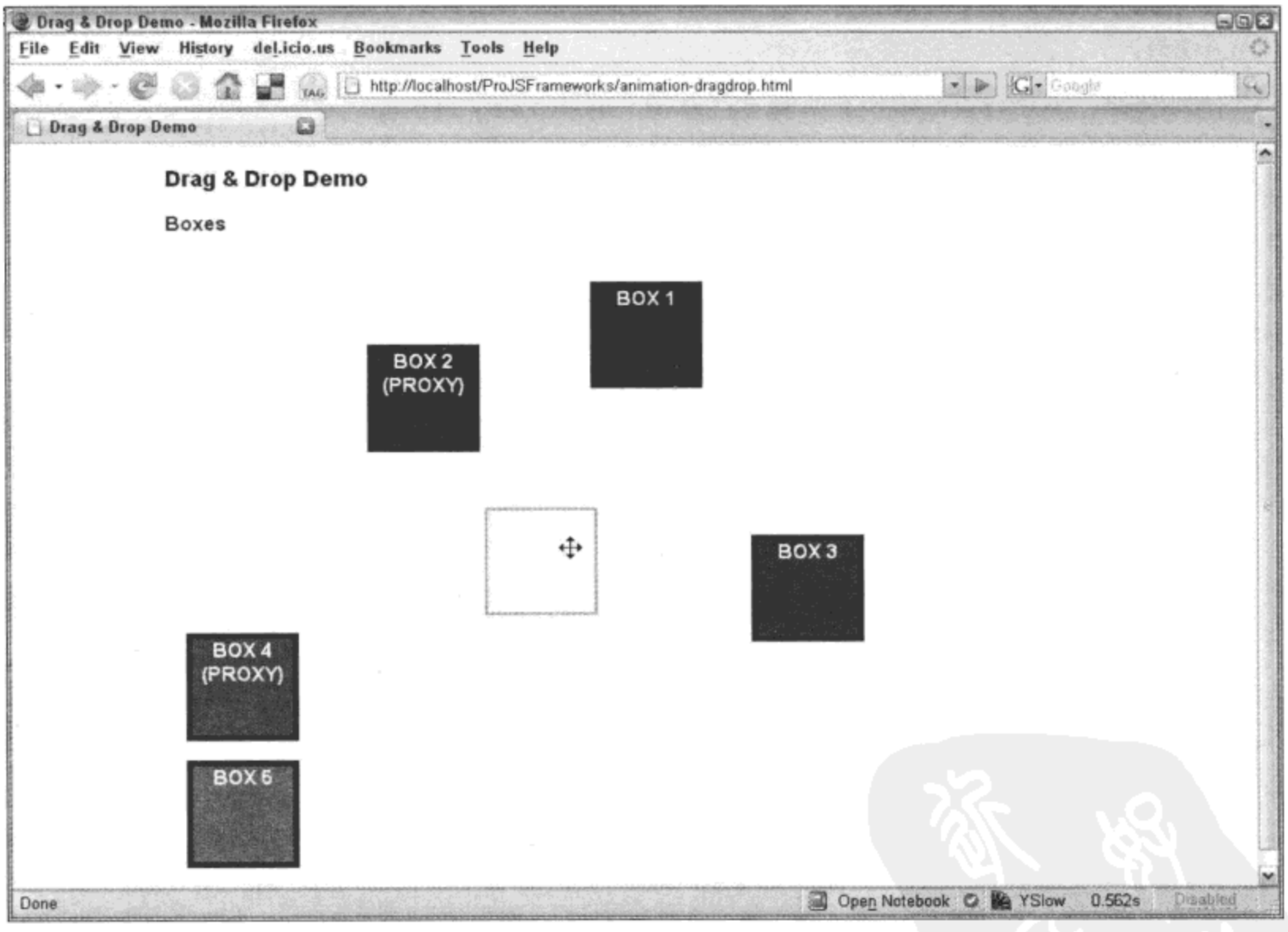


图　9-7

YUI 3 中的新增功能

YUI 3 的 Animation Utility 中的一项新功能是 to 和 from 值实际上可以是函数。该函数接收一个实参，该实参就是正在对其制作动画的节点对象，因此可以执行一些计算工作并返回结果值。对于需要引用节点的当前宽度的情况，这一新功能尤其有用。此外，animate 方法也改名为 run。

9.4　本章小结

虽然整合动画和拖放功能属于桌面应用程序的规范，但是并没有现成的 Web 技术能够用来实现这些功能。然而，诸如 YUI 之类的库提供了易于使用、功能强大而且扩展性极佳的组件，使得在 Web 中制作动画成为一件简单的事情。关于 Animation Utility 和 Drag & Drop Utility 需要重点记住的是，它们的开放式架构允许将自定义代码绑定到其执行过程中的关键事件。这一功能与这些实用工具高度可配置和可扩展的特性结合起来，为我们打开了通往无限可能世界的大门。

第10章

简化 AJAX 和动态加载

传统上，浏览器向服务器发出 HTTP 请求，获取一个页面，分析该页面，然后发出更多 HTTP 请求来获取诸如图像、JavaScript 文件等资源。这个模型从 Web 诞生开始就一直发挥作用。但是，最近随着 AJAX 以及有关页面优化的新最佳实践的出现，这个模型正在开始演变。不再通过第一个 HTTP 请求来传送页面的全部内容，也不再在分析该页面之后立即下载所有需要的资源。相反，Web 中出现了一个获取所需数据和资源的“按需”模型。

本章内容简介：

- 请求和获取数据
- 加载库和组件

10.1 建立 HTTP 请求并获取数据

Web 2.0 风靡一时。在询问业界人士“Web 2.0”的真正含义是什么时，我们将会得到很多答案，但是主要答案指向 AJAX。尽管 Web 2.0 并不仅仅指 AJAX，但是现在开发的大量网站都使用该技术。因此，能够跨浏览器实现 AJAX 已经成为一项关键的任务。

AJAX 其实并不是一种新技术。虽然 AJAX 这个名称是新的，但是其底层技术已经出现有 10 年时间，而如果算上 iframe 的使用，那么时间就更早。微软首先于 1999 年在 Internet Explorer 5 中实现了 XMLHTTP。然后，Mozilla 项目在 Mozilla 1.0 中原生地支持该技术，将其称为 XMLHttpRequest (XHR)。

对于浏览器不原生支持 XHR 的情况，例如 Internet Explorer 6，我们需要通过 MSXML ActiveX 对象来调用 XMLHTTP。问题在于，MSXML 在不同计算机上的可用性各不相同，它的版本也是如此。因此，棘手的问题就是跨浏览器和跨操作系统支持 AJAX 需要一些技巧，例如代码分支。

一旦创建 XHR 对象，使用它就需要一些技巧。有一个需要浏览器制造商解决的缺陷，就是 XHR 并不原生支持超时机制(Internet Explorer 8 的原生 XHR 对象中已经包含了超时属性)。因此，如果向服务器发出一个请求，而由于某种原因服务器始终没有应答，那么浏览器将停在那里等待

响应，从而给用户留下浏览器已经锁定的印象。当然，如果服务器确实有响应，那么这并不总是意味着该事务是成功的。在这里，我们需要的功能是能够解释服务器的响应，从而判断是否存在问题或者事务是否成功。YUI Connection Manager 负责处理所有这些问题，并对 XHR 功能的跨浏览器规范化和增强大有帮助。

10.1.1　asyncRequest 函数

YUI Connection Manager 完成的主要功能包装进一个名为 asyncRequest 的函数中。这个函数带有 4 个参数：method、uri、callback 和 postData。method 参数是字符串类型，它指定请求类型为 GET 或 POST。uri 参数是资源的完全限定路径。这可以是任何能够产生响应的资源，例如文本文件、HTML 文档、CGI 应用程序或者服务器端脚本(例如 PHP、JSP、ASP 或其他类似资源)。callback 参数是一个自定义对象，它包含了几个回调函数，用来处理待建立请求可能的输出结果。该参数至少需要包含一个用于处理调用成功的处理程序和一个用于处理调用失败的处理程序。最后，postData 参数是为 POST 类型的请求准备的。它用于将 POST 数据传给服务器，这是因为不能将该数据附在 URL 路径中发送(如同在 GET 请求中那样)。下面是一个样本 GET 调用：

```
var callback = {
    success: function (o) {/* do something! */},
    failure: function (o) {/* do something else */}
}
var request = YAHOO.util.Connect.asyncRequest('GET', '/order/', callback);
```

变量 request 接收 asyncRequest 创建的连接对象。如果服务器事务成功，就会调用成功事件处理程序；否则，就会调用失败事件处理程序。有 6 种可能的回调事件(参见表 10-1)可供连接。

表　10-1

回 调 事 件	说　　明
Start	该事件在事务开始时引发，并把该事务的 ID 传给它的已订阅处理程序
Complete	该事件在事务响应已经完成时引发，并将该事务的 ID 传给它的已订阅处理程序
Success	该事件在事务响应已经完成并确定响应状态是 HTTP 2xx 时引发。响应对象将传给 successEvent 的已订阅处理程序。这个事件类似于回调成功处理程序。注意：对于文件上传事务，不会引发这个事件
Failure	该事件在事务响应已经完成并确定响应状态是 HTTP 4xx/5xx 或 HTTP 状态不可用时引发。响应对象将传给 failureEvent 的已订阅处理程序。这个事件类似于回调失败处理程序
Upload	该事件在文件上传事务完成时引发。这个事件只针对文件上传事务引发，取代 successEvent 和 failureEvent。响应对象将传给 uploadEvent 的已订阅处理程序。这个事件类似于回调上传处理程序
Abort	这个事件在事务的 callback.timeout 触发一个终止操作时引发。还可以通过 YAHOO.util.Connect.abort() 来明确地引发

来源：YUI Connection Manager 网页

下面是 asyncRequest 函数的实际使用示例。

```
<html>
    <head>
        <title>Connection Demo</title>
        <link rel="stylesheet" type="text/css" href="reset-fonts-grids.css" />
        <link rel="stylesheet" type="text/css" href="base-min.css" />
    </head>
    <body>
        <div id="doc">
            <div id="hd">
                <h1>Connection Demo</h1>
            </div>
            <div id="bd">
                <h2>Status</h2>
                <ul id="status">
                </ul>
                <h2>Output</h2>
                <div id="output">
                </div>
            </div>
        </div>
        <script src="yahoo-dom-event.js"></script>
        <script src="connection-min.js"></script>
        <script src="animation-min.js"></script>
        <script>
YAHOO.namespace("example.proJavaScriptFrameworks");
YAHOO.example.proJavaScriptFrameworks = function () {
    var status = YAHOO.util.Dom.get("status");
    var output = YAHOO.util.Dom.get("output");

    // Utility functions
    function setStatusMessage(msg) {
        var li = document.createElement("li");
        li.appendChild(document.createTextNode(msg));
        status.appendChild(li);
        indicateNewContent(li);
    };
    function indicateNewContent(el) {
        el = YAHOO.util.Dom.get(el);
        el.style.backgroundColor = "#ff0";
        setTimeout(
            function () {
                var anim = new YAHOO.util.ColorAnim(
                    el, {backgroundColor: {to: "#ffffff"}});
                anim.animate();
            }, 1000);
    };
```

```
// XHR event handlers
function successHandler(o) {
    setStatusMessage("Success!");
    var id = YAHOO.util.Dom.generateId();
    var content = o.responseText.replace("new-content", id);
    output.innerHTML += content;
    indicateNewContent(id);
};
function failureHandler(o) {
    setStatusMessage("Failed");
};
var callback = {
    success: successHandler,
    failure: failureHandler
};

// Timed async requests to the server
setTimeout(function () {
    var request1 = YAHOO.util.Connect.asyncRequest(
        'GET', 'connection-htmlFragment.html', callback);
}, 1000);
setTimeout(function () {
    var request2 = YAHOO.util.Connect.asyncRequest(
        'GET', 'bogus.html', callback);
}, 3000);
setTimeout(function () {
    var request3 = YAHOO.util.Connect.asyncRequest(
        'GET', 'connection-htmlFragment.html', callback);
}, 5500);
})();
</script>
</body>
</html>
```

这个示例基本上建立了 3 个不同的 asyncRequest。它们按照时间顺序来模拟用户交互，并确保它们不会在同一时间完成。这 3 个请求均是 GET 请求，其中第二个请求故意失败，目的是为了说明如何使用 failure 回调函数。添加到 DOM 中的每一块新内容也都是通过 indicateNewContent 函数传入的，该函数会使新增内容短暂地以黄色高亮显示，然后淡出为白色。这是前面曾经提及的一种用来提示用户页面中某些内容发生变化的技术。

在成功事件处理程序中，注意来源内容中的 ID 如何修改为生成的 ID。因为这个示例使用的内容均来自同一个来源，所以最终有两个元素具有相同的 ID。将 ID 替换为生成的 ID 可以确保 DOM 具有唯一 ID，而没有重复的 ID。

图 10-1 给出了该示例的实际运行情况。

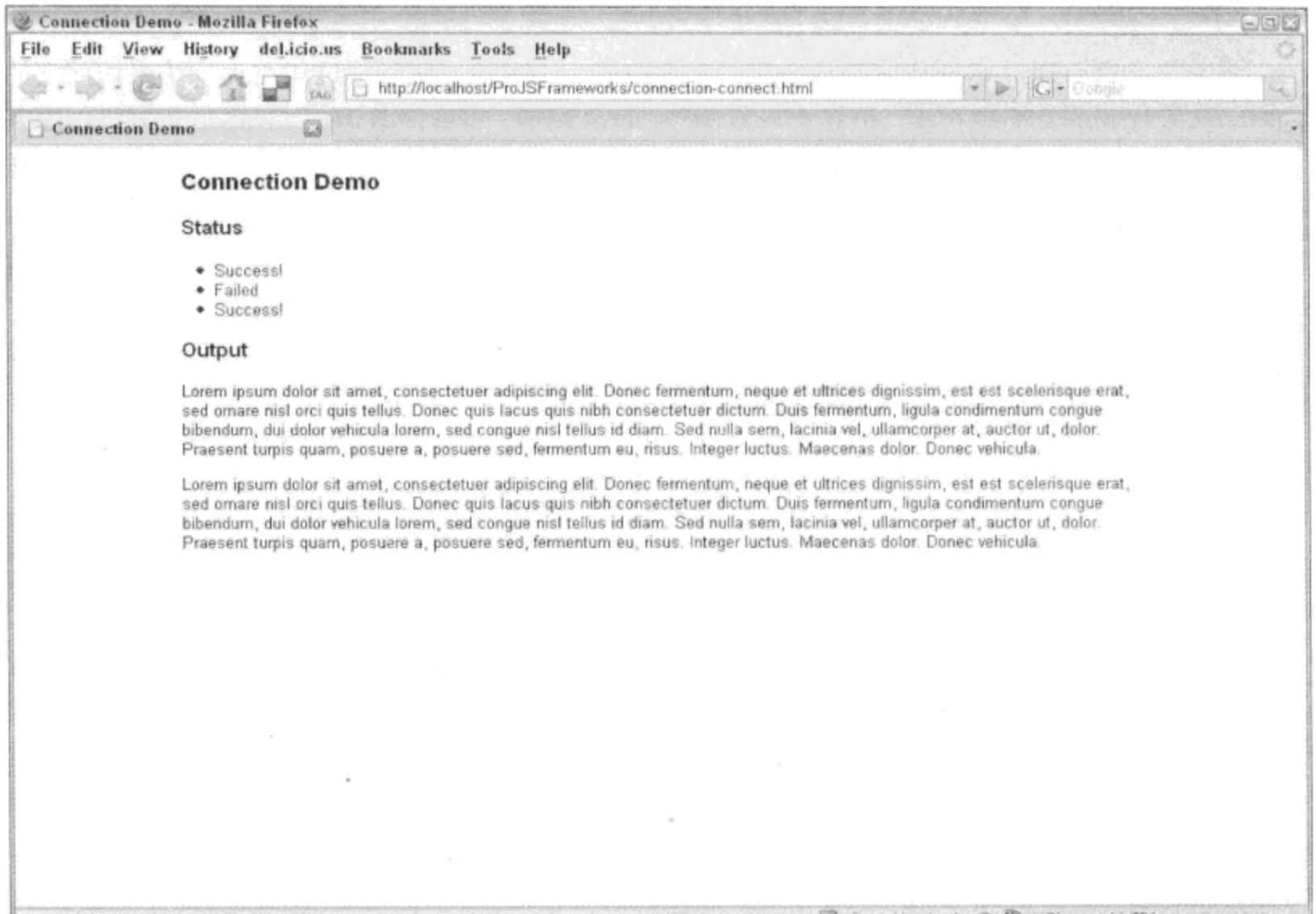


图　10-1

注意：

Connection Manager 示例需要在 Web 服务器上执行。也可以使用本地主机，只要它是 Web 服务器即可，这是因为浏览器严格遵循“同源”策略。如果从文件系统中运行这个示例，就不会解析域名，代码将失败。

10.1.2　JSON

JavaScript Object Notation(也称为 JSON)由 Douglas Crockford 在搜索一种理想的数据交换格式时发现。JSON 将 JavaScript 的原生对象字面值表示法简化成一种能够由其他语言和系统产生和使用的数据形式。这种最小公分母方式正是 JSON 要求名/值对中的名称是字符串类型的原因，虽然在 JavaScript 中它们本来可以采用其他类型。

下面是一段简单的 JSON 代码：

```
{"greeting": "Hello World!"}
```

一旦将这个字符串传给 JavaScript 的 eval 方法，它就会变换成一个实际的对象。因此，下面给出了一个简单的示例来说明如何完成该操作：

```
var personString = "{'fname': 'John', 'lname': 'Doe'}";
var person = eval("(" + personString + ")");
```

```
alert("Hello, my name is " + person.fname + " " + person.lname);
```

基本上，这里通过 eval 方法将这个字符串变换成一个对象，然后通过句点表示法来访问它的内容。这段代码非常简单直接，但是有一点例外：eval 函数。eval 函数调用 JavaScript 解释程序并把一个字符串传给它。然后解释程序获取这个字符串，进行分析之后执行它，就像该字符串是从可信 HTML 或 JavaScript 文件中读取的一样。这里并没有运用任何安全模型。因此，即使该字符串中有恶意代码，相应的代码也会得以执行。这会在网站中造成巨大的安全漏洞，并且是当今跨站脚本(XSS)攻击中经常利用的一种漏洞。

过滤恶意代码是一项艰难的工作，而且如果没有正确实现，它就没有任何用处。YUI 的 JSON Utility 提供了一组工具来验证 JSON 字符串的安全性，然后通过 eval 函数来安全地继续运行它。

1. isValid 方法

isValid 方法由 JSON Utility 在内部使用，同时也提供给外部使用。它接收一个字符串作为参数，并使其通过 4 项独立的测试：

(1) 移除所有转义序列。

(2) 移除所有安全值(true、false、null、回车等)。

(3) 移除所有开放的方括号(只有[，没有])。

(4) 验证剩余字符串中是否存在可能导致其成为无效 JSON 的特殊字符。

注意，第(3)步允许数组通过测试，这是因为数组属于 JSON 规范(RFC 4627)的一个有效部分，尽管它们容易被利用。这就要求开发人员负责确保传递到客户端的数据是安全的，而这很可能会要求忽略数组。

下面是一个导致 isValid 方法返回 false 的无效 JSON 字符串示例：

```
var foo = YAHOO.lang.JSON.isValid("{'foo': 'bar'}");
```

下面是一个导致 isValid 方法返回 true 的有效 JSON 字符串示例：

```
var foo = YAHOO.lang.JSON.isValid('{"foo": "bar"}');
```

注意两者的差异。尽管 JavaScript 并没有对单引号和双引号的使用做出明确规定，但是 JSON 有相应的规定。尽管 YUI 使用 JavaScript 分析 JSON 数据，但是 JSON 并不仅限于 JavaScript，因此 JSON 的规则要比 JavaScript 的规则更加严格。例如，在 JSON 中，名/值对中的名称部分必须是字符串，而 JavaScript 中的对象字面值并不要求这样。与此类似，JSON 中的字符串采用双引号表示。

2. parse 方法

将 JSON 数据字符串转换成可用的 JavaScript 对象的传统做法是将该字符串传给 eval 函数。这种方式(前面曾经提及)高度危险，因为该字符串可能包含恶意代码。因此，我们要把该字符串传给一个验证器来确保数据是合法的。当然，额外增加的这一步操作容易产生错误，因为开发人员可能忘记在通过 eval 运行该数据之前要进行验证。接下来介绍 parse 方法。parse 方法首先通过把将要执行 eval 函数的字符串传给 isValid 方法来检查它是否包含恶意代码。如果认为它是有效的 JSON，就通过 eval 运行它。它还带来另一个好处，就是它会把返回的对象交给一个由用户定义的过滤器函数。这个过滤器是可选的，可用于对新转换的 JSON 数据进行后期处理。

因此，下面给出一块准备好分析的 JSON 数据：

```
{
 "author": "Ara Pehlivanian",
 "version": "2.1.3",
 "status": "release",
 "pubDate": "2008-04-07",
 "content": "Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Donec
     fermentum, neque et ultrices dignissim, est est scelerisque erat, sed
     ornare nisl orci quis tellus. Donec quis lacus quis nibh consectetuer dictum.
     Duis fermentum, ligula condimentum congue bibendum, dui dolor vehicula lorem,
     sed congue nisl tellus id diam. Sed nulla sem, lacinia vel, ullamcorper at,
     auctor ut, dolor. Praesent turpis quam, posuere a, posuere sed, fermentum
     eu, risus. Integer luctus. Maecenas dolor. Donec vehicula."
}
```

下面的代码用来分析和过滤上面的 JSON 数据：

```
<html>
     <head>
          <title>JSON parse</title>
     </head>
     <body>
       <h1>JSON parse</h1>
       <ul id="output"></ul>
       <script src="yahoo-dom-event.js"></script>
<script src="connection-min.js"></script>
<script src="json-min.js"></script>
<script>
    YAHOO.namespace("example.ProJSFrameworks.lang.JSON");
    YAHOO.example.ProJSFrameworks.lang.JSON = function () {
        function versionFilter(key, val) {
            if (key === "version") {
                return undefined;
            } else {
                return val;
            }
        };
        var callback = {
            success: function (o) {
                var parsed = YAHOO.lang.JSON.parse(
                        o.responseText, versionFilter);
                var output = YAHOO.util.Dom.get("output");
                for (var key in parsed) {
                    if (parsed.hasOwnProperty(key)) {
                        var li = document.createElement("li");
                        var strong = document.createElement("strong");
                        strong.appendChild(
                                document.createTextNode(key + ": "));
                        li.appendChild(strong);
                        li.appendChild(
```

```
                                    document.createTextNode(parsed[key]));
                            output.appendChild(li);
                        }
                    }
                }
            };
            var transaction = YAHOO.util.Connect.asyncRequest(
                'GET', 'lang-json2.json', callback);
        }();
    </script>
</body>
</html>
```

这个代码示例非常简单。它由一个异步事务组成，该事务从文件 lang-json2.json 那里加载 JSON 数据。然后，它调用 success 回调函数，在该函数中分析在返回对象的 responseText 属性中找到的返回数据。一旦分析过 JSON 数据，就会将其交给 versionFilter 函数，该函数会将对象的 version 名/值对过滤出来。

值得注意的是，这个过滤器函数必须返回一个值。这是因为 parse 函数会把所有使过滤器函数返回 undefined 值的节点从原始 JSON 数据中移除。因此，如果这个节点未被修改，那么过滤器函数必须返回与它所接收的同样的值。

一旦将数据从字符串转换成一个对象并进行过滤，就简单地使用一个 for in 循环遍历它并输出到页面中的一个无序列表元素内。

10.2 动态加载库和组件

性能正成为 JavaScript 领域的一个热门话题，这很大程度上是因为采用该语言编写越来越多的 Web 应用程序。人们不再仅仅把 JavaScript 用作网页上的一种简单的脚本技巧。现如今，随着桌面应用程序不断迁移到浏览器，对采用最佳方式编写的 JavaScript 的需要正成为一项关键任务。延迟加载就是一种改进性能或者改善性能感觉的技术。这种技术将所需组件的加载推迟到真正需要用到它们的时刻。这样一来，如果从来没有用到某个组件，那么下载和执行它的依赖项给页面带来的负担就会减轻。另一种提高性能的技术是减少页面发出的 HTTP 请求次数。将实际的数据传输排除在外，我们仍然需要花费时间来建立每个请求。而且，根据网络状况和服务器负载，所花费的时间可能比较显著，累积起来就会造成明显的速度下降。为此，将资源压缩并组成“聚合”文件可以减少页面发出的 HTTP 请求次数。

10.2.1 Get Utility

YUI 提供了 Get Utility，可用来动态地加载脚本和 CSS 文件。这样就可以在需要用到组件的时候加载它们，而不是在页面加载时就预测它们的使用。它还可用来加载跨域数据，而这是不可能通过 Connection Utility 实现的功能，因为 XMLHttpRequest(XHR)严格遵循同源策略。

XHR 的同源策略假设只有来自于与加载页面相同的服务器的脚本才可信。这既有好处，又带来了麻烦。很明显，它的好处在于其带来的保护，它限制了把第三方脚本注入到站点中的可能性。但是，它带来的麻烦之处在于，现如今涌现出大量第三方 API 以及随之而来的数据混合。例如，不可能打开一个到第三方数据源的 XHR 连接。执行该操作的唯一的方式就是通过一个代理，从而遵守同源策略。然而，并不总是能够设置代理。Get Utility 完全绕开了同源策略，它动态地将新的脚本标记插入到页面中，而不会受到同源策略的约束。这样，该实用工具就可以从所需的域中加载脚本。

注意：

不应该将 Get Utility 用来从不可信源那里加载 JavaScript 文件。任何通过 Get Utility 加载的脚本都将以完全的特权执行，而不能像通过 Connection Utility 那样过滤为 JSON 数据。

Get Utility 的功能不仅仅是将 script 或 CSS link 标记转储到页面中。它提供了一个事务包装器，可用来分配成功和失败事件的回调函数，还可以定义这些函数执行的作用域。可以把一个文件名数组传给它，这样调用一次 GetUtility 就可以加载多个文件。它允许指定将 script 或 link 标记加载到哪个目标窗口中。GetUtility 允许指定一个全局变量，它将轮询该变量，只有当该变量可用时(对 Safari 2.x 来说，这一点尤其重要，因为它不能为新添加到页面中的 script 和 link 标记触发 loaded 事件)才继续执行。最后，它还允许在这些标记写入到文档之后自动将其清除。这对于 script 标记有益，因为一旦它们执行完毕，即使这些标记被移除，它们的代码在内存中仍然可用。但是，CSS link 标记提供了指向样式表的活跃链接，当把它们移除之后，它们提供的规则也一并被移除，页面布局将会相应地发生变化。

Get Utility 提供用来加载依赖项的两个主要方法是 script 和 css。

```
YAHOO.util.Get.script("wrox.js");
YAHOO.util.Get.css("wrox.css");
```

这两个方法在形式上是一样的，只是它们输出的标记不同。两个方法都可以接收可选的参数。下面是一个完全加载的 script 方法示例：

```
function successHandler(o) {
    // do something
};
function failureHandler(o) {
    // do something
};
var newWin = window.open("/ProJSFrameworks/newPage.html");
YAHOO.util.Get.script("wrox.js", {
    onSuccess: successHandler,
    onFailure: failureHandler,
    win: newWin, // window to insert script into
    scope: newWin, // scope of event handler
    data: "Hello World!", // arbitrary data to pass the event handler
    varName: ["wroxReady"], // var to check in wrox.js to call handler in Safari 2
    autoPurge: true
```

```
});
```

这个代码片段将 wrox.js 加载到 newPage.html 页面中，该页面将在新窗口中打开。一旦页面加载完毕，Get Utility 就检查 wroxReady 变量的可用性，然后引发它的 onSuccess 回调函数。

可以将回调函数分配给两个不同的事件 onSuccess 和 onFailure。回调函数接收一个含有事务唯一标识符的对象(tId)、一个用来向该回调函数传递任意数据的自定义 Data 对象(data)、一个由 Get Utility 方法创建的节点数组(nodes)、一个在其中创建这些节点的 Window 对象的引用(win)，以及一个用来移除刚刚创建的节点的清除方法(purge)。

值得注意的是，对于已经加载的脚本，虽然标记被清除，但是脚本仍然保留在内存中，这是因为它已经执行。但是对于 CSS 来说，一旦将标记移除，那么样式规则也随之清除，这将立即反应到页面中。

下面是一个更加简单的 Get Utility 实际运用的示例：

```
<html>
    <head>
        <title>Get Utility</title>
    </head>
    <body>
        <h1>Get Utility</h1>
        <script src="yahoo-min.js"></script>
        <script src="get-min.js"></script>
        <script>
            YAHOO.namespace("example.ProJSFrameworks.get");
            YAHOO.example.ProJSFrameworks.get = function () {
                var successHandler = function (o) {
                    var h1 = YAHOO.util.Selector.query("h1")[0];
                    var txt = document.createTextNode(o.data);
                    h1.insertBefore(txt, h1.firstChild);
                };
                var failureHandler = function (o) {
                    alert("Failed to load Selector Utility");
                };
                var transaction = YAHOO.util.Get.script("selector-min.js", {
                        onSuccess: successHandler,
                        onFailure: failureHandler,
                        data: "The YUI "
                    });
            }();
        </script>
    </body>
</html>
```

这个脚本的实际作用是延迟 Selector Utility 的使用，直到它已经动态加载。Get.script 方法获取 selector-min.js 文件，而且只有在该文件已经加载之后才引发 successHandler 函数。在 successHandler 函数内部，我们假设想要的 JavaScript 文件已经成功加载，因此可以安全地执行 Selector.query 方法。该查询获取页面的第一个 h1 标记，创建一个文本节点(其文本内容由 data 属性传入)，然后将该节点附加到这个 h1 元素的现有文本之前。其结果就是 h1 标记的内容从“Get

Utility”变成了“The YUI Get Utility”。

10.2.2　YUI Loader Utility

YUI Get Utility 用来动态加载外部脚本和 CSS 文件。但是，对于加载 YUI 文件而言，还需要进一步考虑其他事项。大多数 YUI 文件依赖其他文件，需要按照特定的顺序加载才能运行。随着组件数量不断增加以及它们之间存在众多依赖关系，要记住每个组件的需求是一件困难的事情。如果一个页面中有多个 YUI 组件、滑块、面板或自动完成组件，每个组件都有自己的需求，并与其他组件至少共享一些需求，这个问题就会变得更加复杂。

最终，YUI 提供了汇总(或聚合)文件。这些文件由最常用的文件组成，并将其汇总成一个文件。这样做的原因在于性能。Yahoo! Exceptional Performance 团队提出的性能规则之一是减少页面发出的 HTTP 请求数。将 yahoo.js、dom.js 和 event.js 文件汇总成一个文件，与此同时将这些文件最小化(减少文件大小)，从而形成了这 3 个最常用 YUI 组件的更高效的实现。

YUI Loader 是一个用来动态加载 YUI 组件的实用工具。它知道该库有多个组件以及它们之间的依赖关系，还知道所有这些汇总资源，因此在加载一个组件时，它能够选择最佳的文件组合。它能够检测已经加载的 YUI 部分，因此它不会无谓地尝试重新加载已有的组件。

YUI Loader 的使用非常简单。实例化一个新的 YUILoader 对象，并使用参数告诉它加载 YUI 的什么部分以及如何加载它们。例如，下面这行代码加载 container.js 文件以及它的依赖组件和可选的依赖组件。

```
var loader = new YAHOO.util.YUILoader({require: [container], loadOptional: true});
```

表 10-2 列出了 YUI Loader 知道的所有 YUI 组件以及通过什么名称来调用这些组件。name 值传给 require 数组，该数组然后会加载必要的文件以及依赖组件。如果将 loadOptional 参数设置为 true，那么还会加载可选依赖组件。当然，前面曾经提到过，如果某个依赖组件(例如 dom 或 event)刚好已经加载，就不会再次加载它们。

表　10-2

组 件 名 称	文　　件	依 赖 组 件	可选依赖组件
animation	animation-min.js	dom、event	
autocomplete	autocomplete-min.js	dom、event	connection、animation
base	base-min.css		
button	button-min.js	Element	menu
calendar	calendar-min.js	event、dom	
charts	charts-experimental-min.js	element、json、datasource	
colorpicker	colorpicker-min.js	slider、element	animation
connection	connection-min.js	Event	

(续表)

组件名称	文件	依赖组件	可选依赖组件
container	container-min.js	dom、event	dragdrop、animation、connection
containercore	container_core-min.js	dom、event	
cookie	cookie-beta-min.js	Yahoo	
datasource	datasource-beta-min.js	Event	connection
datatable	datatable-beta-min.js	element、datasource	calendar、dragdrop
dom	dom-min.js	Yahoo	
dragdrop	dragdrop-min.js	dom、event	
editor	editor-beta-min.js	menu、element、button	animation、dragdrop
element	element-beta-min.js	dom、event	
event	event-min.js	Yahoo	
fonts	fonts-min.css		
get	get-min.js	Yahoo	
grids	grids-min.css	Fonts	reset
history	history-min.js	Event	
imagecropper	imagecropper-beta-min.js	dom、event、dragdrop、element、resize	
imageloader	imageloader-min.js	event、dom	
json	json-min.js	Yahoo	
layout	layout-beta-min.js	dom、event、element	animation、dragdrop、resize、selector
logger	logger-min.js	event、dom	dragdrop
menu	menu-min.js	containercore	
profiler	profiler-beta-min.js	Yahoo	
profilerviewer	profilerviewer-beta-min.js	yuiloader、element	
reset	reset-min.css		
reset-fonts-grids	reset-fonts-reset-fonts-grids.css		

(续表)

组 件 名 称	文　　件	依 赖 组 件	可选依赖组件
reset-fonts	reset-reset-fonts.css		
resize	resize-beta-min.js	dom、event、dragdrop、element	animation
selector	selector-beta-min.js	yahoo、dom	
simpleeditor	simpleeditor-beta-min.js	Element	containercore、menu、button、animation、dragdrop
slider	slider-min.js	Dragdrop	animation
tabview	tabview-min.js	Element	connection
treeview	treeview-min.js	Event	
uploader	uploader-experimental.js	Yahoo	
utilities	utilities.js		
yahoo	yahoo-min.js		
yahoo-dom-event	yahoo-dom-yahoo-dom-event.js		
yuiloader	yuiloader-beta-min.js		
Yuitest	yuitest-min.js	Logger	

在下面的示例中，YUI Loader 获取样式表和 container 组件，然后在所有必要组件加载完毕之后立即构建一个面板。

```
<html>
    <head>
        <title>YUI Loader</title>
    </head>
    <body class="yui-skin-sam">
        <h1>YUI Loader</h1>
        <script src="yuiloader-beta-min.js"></script>
        <script>
            YAHOO.namespace("example.ProJSFrameworks.yuiloader");
            YAHOO.example.ProJSFrameworks.yuiloader = function () {
                var loader = new YAHOO.util.YUILoader({
                    require: ['reset-fonts', 'base', 'container'],
                    loadOptional: true,
                    onSuccess: function () {
                        var pnl = new YAHOO.widget.Panel("hello", {
                            visible: true,
                            modal: true,
                            fixedcenter: true,
                            close: true
                        });
                        pnl.setHeader("Hello World");
```

```
                    pnl.setBody(
                        "The code for this panel was dynamically loaded");
                    pnl.render(document.body);
                }
            });
            loader.insert();
        }();
    </script>
</body>
</html>
```

整个程序的全部功能均位于传给 YUILoader 构造函数的参数中。注意 onSuccess 处理程序。当然，该处理程序可以指向一个命名函数，但是在这里通过内联方式提供了一个匿名函数。在该函数中，创建一个新面板，并设置一些内容。当然，在 YUI Loader 下载并加载必要的组件之前不可能完成这些工作。

还可以通过 addModule 方法来使用 YUI Loader 加载非 YUI 模块。

```
var loader = new YAHOO.util.YUILoader(/* some params here */);
loader.addModule({
    name: "formValidator",
    type: "js",
    fullpath: "http://domain.com/js/form-validator.js",
    varName: "FORMVALIDATOR",
    requires: ['yahoo-dom-event']
});
```

传入 addModule 方法的对象字面值实际上与 YUI Loader 内部用来定义自身模块的语法相同。name 值必须唯一，不能与任何现有的 YUI 组件名称相同。type 值告诉 Loader 将什么类型的标记写入 DOM、CSS 或 JavaScript。fullpath 值指向要加载的文件，而 varName 是一个全局变量名，在 Safari 2.x 及更早版本中，Loader 将轮询该变量来确定模块是否已经加载完成。

此外，如果该模块并不属于第三方库(因而不能访问)，那么可以使用 YAHOO.register 方法而不是 varName 变量。库中的所有 YUI 组件都使用这个方法来注册自身。因为模块中的最后一行代码是调用 register 方法，所以它完成的工作是通知 YUI 该模块已经加载完成，因此不再需要 varName 变量。

下面这个 YAHOO.register 示例直接取自 connection.js 文件的代码：

```
YAHOO.register("connection",YAHOO.util.Connect, {version: "2.5.0", build: "895"});
```

YUI 3 中的新增功能

在 YUI 3 中，Connection Manager 已经被重写。它现在名为 IO，与其前身的主要差别在于，它现在支持跨域请求或 XDR。为了实现这个功能，它利用了 Flash 的功能来建立 XDR 调用。此外，还需要在该请求要访问的资源站点上部署跨域策略文件。否则该事务就会失败。最后，YUI 3 将 IO 划分成更小的块，这样可以只加载必需的文件而不是整个实用工具。

10.3　本章小结

简单地使用 HTTP 主请求加载 JavaScript 并且只使其充当页面上装饰对象的情况已经一去不复返。现如今，JavaScript 用于获取数据、依赖项甚至更多其他 JavaScript 代码。它用来向服务器建立更小的后期加载 HTTP 请求，还用来动态修改页面功能和内容(即使第一个 HTTP 有效载荷已经传送完毕)。YUI 的实用工具套件不仅将跨浏览器问题规范化，而且在扩展该语言提供的基础功能方面也取得了长足的进步。它为高级 Web 应用程序(仅仅在几年前人们都未曾想过这样的 Web 应用程序)的开发打下了坚实的基础。

第11章

利用窗口部件构建用户界面

(第一部分)

自从 Web 诞生之日起，Web 用户就一直希望他们的丰富桌面体验能够在网站上得以体现。在 Web 上实现这种富用户界面的困难之处在于，人们最初在设计 Web 技术时就没有考虑这个目标。Web 最初的意图是发表链接的科学文档，因此带有超链接，并因而得名网(web)。

但是在 Web 创建之后不久，人们就试图让网页变得更加漂亮。随着 Web 不断流行并开始攻占越来越多的桌面，人们对网页能够处理更多内容的愿望也越来越强烈。人们不再满足于只有各种大小的彩色文本和少量的链接。很快，可交互性成为一个焦点，这促成了 Java applet 的发明，而且很快 Netscape 的 Brendan Eich 创造了 JavaScript(与 Java 并没有任何关系)。当然，当时正值 20 世纪 90 年代中期所谓的浏览器大战时期，因而微软自然地以 JScript 应战，这实际上是 JavaScript 的一个副本，尽管它们接近一致，但仍然有些差异，而且 DOM 在两种浏览器上所表现的行为也稍有不同。到现在为止，各种浏览器之间的差异仍然存在，这使得所有向网页中添加 JavaScript 代码的努力都成为非常谨慎和务实的工作。

考虑到浏览器的不友好环境，YUI 在 YAHOO.widget 名称空间下提供了几个专业的跨浏览器组件，它们将浏览器中的交互性概念一下子提升到 21 世纪。

本章内容简介：

- 使用 AutoComplete 实用工具
- 构建容器
- 使用选项卡和树状视图

11.1 AutoComplete 实用工具与表单字段结合使用

表单是 Web 的基本构建块，仅次于超链接。人们普遍知道，表单获取其字段中的数据并将这些数据发送给服务器。但是，表单的发展到此为止。

而在另一方面，易用性和用户交互设计实践在不断地进展，它们引入的其中一项措施就是自

动完成。虽然这是一个极好的想法，但它尚未正式列入到浏览器制造商遵循的最流行的“规范”中。因此，像 YUI 这样的库就肩负起填补这项空白的责任。

AutoComplete 和 DataSource

要想使 YUI 的 AutoComplete 实用工具能够给出输入建议，需要为其配置数据源。这样，它就能够将输入字段中的值与数据集进行比对，并相应地给出建议。AutoComplete 实际上功能非常丰富，它可以连接到数组、函数、XMLHttpRequest 或 script 节点。

这 4 种类型的数据源定义为 4 个扩展 YAHOO.utilDataSourceBase 类的子类：LocalDataSource、FunctionDataSource、ScriptNodeDataSource 和 XHRDataSource。

下面这个示例演示如何将一个数组设置为数据源：

```
var data = [
    "Rock", "Rocket", "Rockets", "Rocket Man", "Rocketeer",
    "Rocketing", "Rocks", "Rocky", "Rocky and Bullwinkle"
];

var dataSource = new YAHOO.util.LocalDataSource(data);
```

一旦创建数据源，就可以将其传给 AutoComplete 构造函数。使用类似如下的代码：

```
var autoComplete = new YAHOO.widget.AutoComplete(inputEl, resultsEl, dataSource);
```

AutoComplete 构造函数需要 3 个参数：输入字段(将其绑定到该字段)、用于输出结果的 div 元素以及数据源。下面是上述代码的实际使用示例：

```
<html>
    <head>
        <title>AutoComplete</title>
        <link rel="stylesheet" type="text/css" href="reset-fonts-grids.css" />
        <link rel="stylesheet" type="text/css" href="base-min.css" />
        <link rel="stylesheet" type="text/css"
              href="autocomplete/assets/skins/sam/autocomplete.css" />
        <style type="text/css">
            #autocomplete {
                width: 20em;
                height: 1.5em;
            }
            label,
            #autocomplete,
            #search {
                float: left;
            }
        </style>
    </head>
    <body class="yui-skin-sam">
        <div id="doc">
            <div id="hd">
                <h1>AutoComplete</h1>
```

```
    </div>
    <div id="bd">
        <form method="get" action="http://search.yahoo.com/search">
            <label for="p">Search for:</label>
            <div id="autocomplete">
                <input type="text" id="p" />
                <div id="results"></div>
            </div>
            <input type="submit" value="Search!" id="submit" />
        </form>
    </div>
    <div id="ft">
    </div>
</div>

<script src="build/yahoo-dom-event/yahoo-dom-event.js"></script>
<script src="build/datasource/datasource-min.js"></script>
<script src="build/autocomplete/autocomplete-min.js"></script>
<script>
    (function () {
        var data = [
            "Rock", "Rocket", "Rockets", "Rocket Man", "Rocketeer",
            "Rocketing", "Rocks", "Rocky", "Rocky and Bullwinkle"
        ];

        var dataSource = new YAHOO.util.LocalDataSource(data);
        var autoComplete = new YAHOO.widget.AutoComplete(
                "p", "results", dataSource);
    })();
</script>
</body>
</html>
```

注意，AutoComplete 实用工具要求使用一个 div 父元素将输入字段和显示结果的 div 元素包装起来。这样做的目的是为了让显示结果的 div 元素能够正确地放置到输入字段下方。实际上，当把 AutoComplete 实用工具绑定到输入字段时，该实用工具会把 CSS 类名 yui-ac 分配给 div 包装器。这会将包装器的 position 属性设置为 relative 以容纳用来显示结果的 div 元素(随后包装器会通过 CSS 类名 yui-ac-container 将其 position 属性设为 absolute)。这些 CSS 类名的规则定义位于 YUI 自带的 autocomplete.css 文件中。

运行这个示例所需要的唯一CSS文件就是autocomplete.css，其他文件只是为了便于页面布局。即使是这样，autocomplete.css 也只是用来装饰自动完成窗口部件。实际上，自动完成功能来自于 autocomplete-min.js 和 datasource-min.js 的代码。一旦这些依赖项就位，那么实例化自动完成窗口部件就简单到只需要编写几行代码。

图 11-1 给出了自动完成窗口部件的实际运行情况。

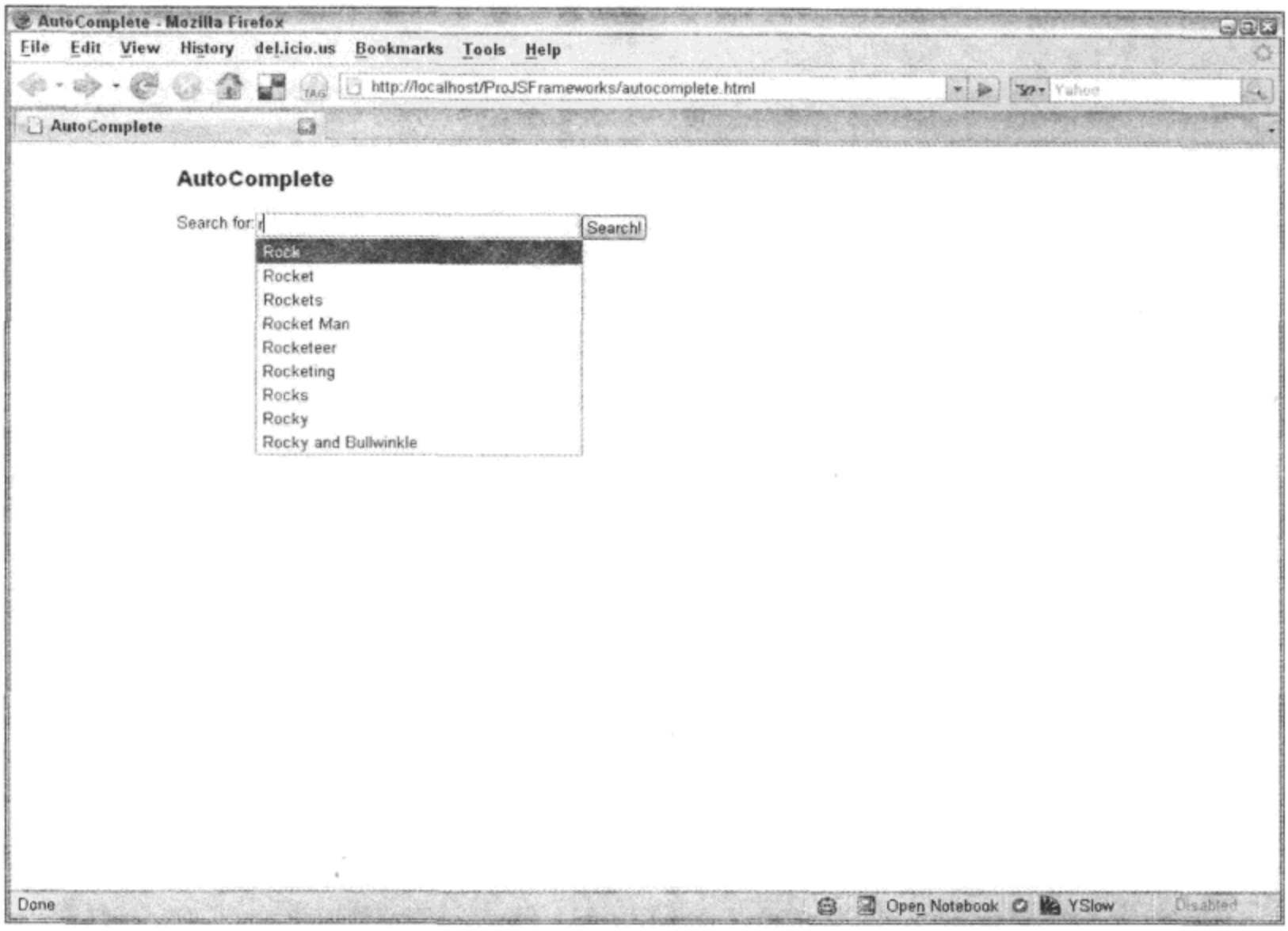


图　11-1

当然，对于搜索示例，将自动完成窗口部件绑定到活跃的搜索引擎更有意义。前面曾经提及，AutoComplete 提供了多种连接到数据源的方式。在这种场合下，显而易见的选择似乎是使用 XMLHttpRequest(XHR)，但有一个严重的问题：XHR 的“同源”策略。同源策略是浏览器厂商自动设置的安全限制，目的是为了防止跨站点脚本攻击。其思想是，如果将 JavaScript 代码限制到只能与该代码所在的服务器通信，那么恶意(被劫持的)JavaScript 代码将数据发送给欺骗性服务器的机会就会大大减少。但在这里，严格的安全措施实际上妨碍了合理的跨域通信需求。一个恰当的示例就是，只有来自 Yahoo! Search 域的脚本才能通过 XHR 访问 Yahoo! Search。为了能够让第三方访问 Yahoo! Search Web 服务，需要建立一个代理。该代理(在后端服务器上运行的程序)负责访问 Yahoo! Search Web 服务，获取响应，然后将其传给 JavaScript 代码。这种方式非常笨拙。YUI 的 DataSource 实用工具提供了另一种技术，可以通过 script 标记连接到外部服务器。尽管 XHR 受到限制，但 script 标记却可以从 Internet 上的任何地方加载 JavaScript 文件。在充分利用这种灵活性的同时不要忘记一个警告：需要完全信任要访问的数据源。否则，盲目地连接第三方服务器将会在 Web 应用程序中打开一个巨大的安全漏洞。

在通过 script 标记连接到脚本时，YUI 在底层使用的是 Get Utility(在第 10 章中描述过)。这可以确保对动态创建的 script 标记进行统一的、稳健的管理。既然所有的细节都已经由 YUI 处理完毕，那么具体实现就变得非常简单。

```
<script src="build/yahoo-dom-event/yahoo-dom-event.js"></script>
```

```
<script src="build/datasource/datasource-min.js"></script>
<script src="build/autocomplete/autocomplete-min.js"></script>
<script>
    (function () {
        var dataSource = new YAHOO.util.ScriptNodeDataSource(
                "http://search.yahooapis.com/WebSearchService/V1/" +
                "webSearch?appid=YahooDemo&results=10&output=json");
            dataSource.responseSchema = {
                resultsList: "ResultSet.Result",
                fields: ["Title"]
            };
        var autoComplete = new YAHOO.widget.AutoComplete(
                "p", "results", dataSource);
    })();
</script>
```

与前一个示例相比，这里有两处不同：一处是 ScriptNodeDataSource 数据源(而不是前面的 LocalDataSource)的实例化；另一处是响应模式属性(responseSchema)的添加。除此之外，其他内容保持不变。注意，在这里把参数 output=json 传给 Yahoo! Search API。这会指定 Yahoo! Search 引擎以 JSON 字符串形式返回结果。当然，DataSource 无法知道在该对象中的何处查找结果。它需要一种模式来告诉它到哪里寻找所有结果，以及每条结果的哪个属性匹配它的 AutoComplete 查询。

ScriptNodeDataSource 构造函数附带的参数是数据源的地址。接下来，为其分配一个模式，这样它就知道在所接收的有效负载中的什么地方寻找期望的数据：

```
dataSource.responseSchema = {
    resultsList: "ResultSet.Result",
    fields: ["Title"]
};
```

这段代码告诉 DataSource，该数据集的根节点名为 ResultSet，并且可以在 Result 节点中找到结果。该代码还告诉 DataSource 在每条结果的 Title 属性上匹配查询。下面是 Yahoo! Search API 发回的该模式所定义的数据样本(出于布局考虑，使用省略号将部分数据截断)：

```
{
 "ResultSet":{
       "type":"web",
       "totalResultsAvailable":26700000,
       "totalResultsReturned":10,
       "firstResultPosition":1,
       "moreSearch":"\/WebSearchService\/V1\/webSearch?query=yui&amp;appid=Ya...",
       "Result":[
          {
             "Title":"The Yahoo! User Interface Library (YUI)",
             "Summary":"The YUI Library also includes several core CSS...",
             "Url":"http:\/\/developer.yahoo.com\/yui\/",
             "ClickUrl":"http:\/\/uk.wrs.yahoo.com\/_ylt=A9iby40zEihIIDEAdS...",
             "DisplayUrl":"developer.yahoo.com\/yui\/",
             "ModificationDate":1209798000,
```

```
            "MimeType":"text\/html",
            "Cache":{
                "Url":"http:\/\/uk.wrs.yahoo.com\/_ylt=A9iby40zEihIIDEAdiz...",
                "Size":"29225"
            }
        },
        {
            "Title":"Yui - Wikipedia, the free encyclopedia",
            "Summary":"Yui (tribe), a small historical Native American tri...",
```

在自动完成窗口部件的结果窗格中显示的数据来自于 Title 属性。将模式定义改为如下内容，就可以方便地使该数据来自于 DisplayUrl 属性：

```
dataSource.responseSchema = {
    "ResultSet.Result",
    fields: ["DisplayUrl"]
};
```

图 11-2 给出了将自动完成窗口部件插入 Yahoo! Search API 时的实际运行情况。

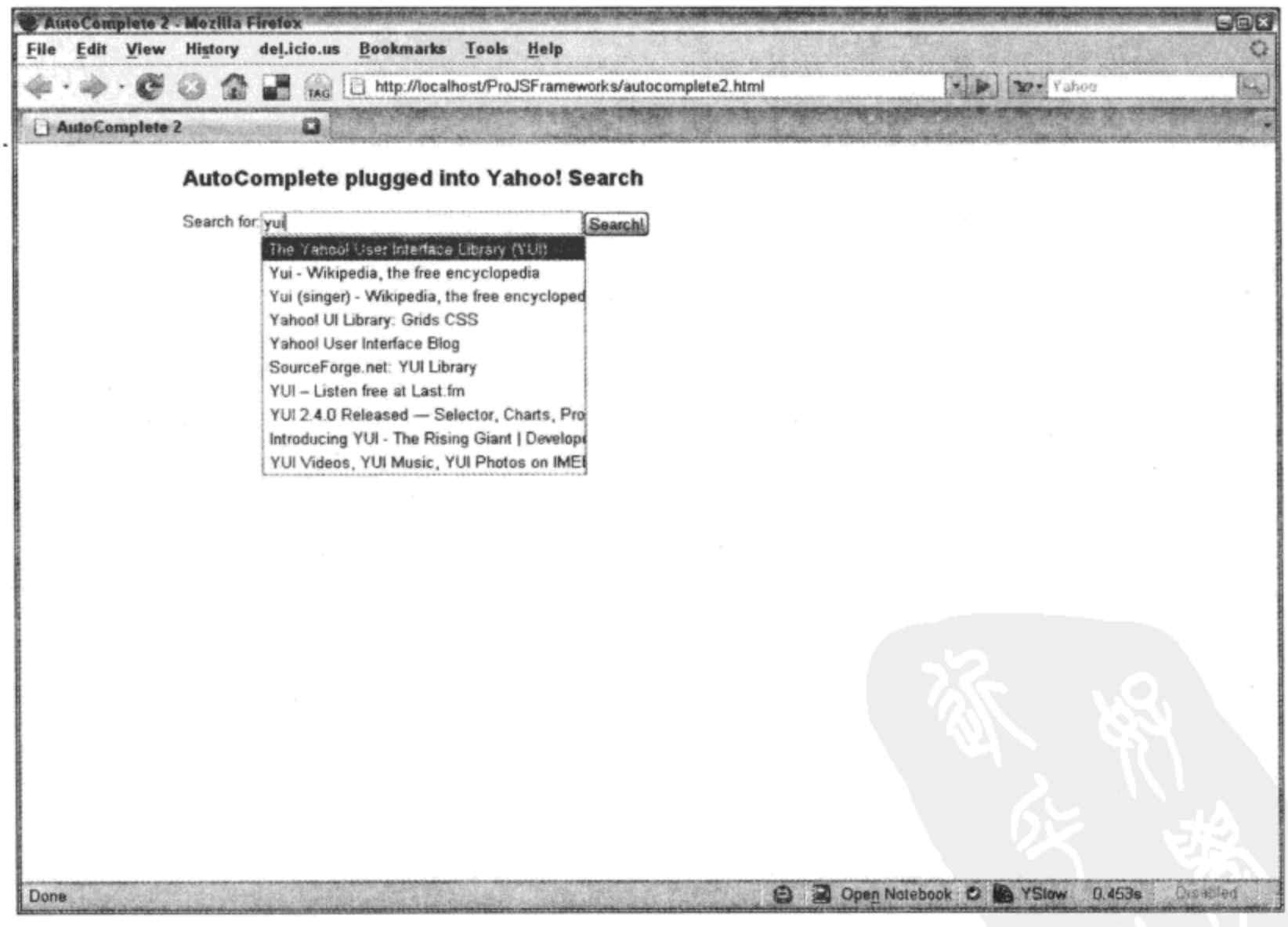


图 11-2

11.2　为内容构建容器

div 元素就是 div 元素，难道它就没有其他用处吗？如果一直将其作为静态 div 元素，那么它就是一个 div 元素。但是，有可能使 div 元素更像是一个可交互模块，而不仅仅是一个静态的 DOM 元素。YUI 的 Container 系列控件提供了一组容器，涉及范围从复杂的 SimpleDialog(A)一直到最简单的 Module(B)，如图 11-3 所示。

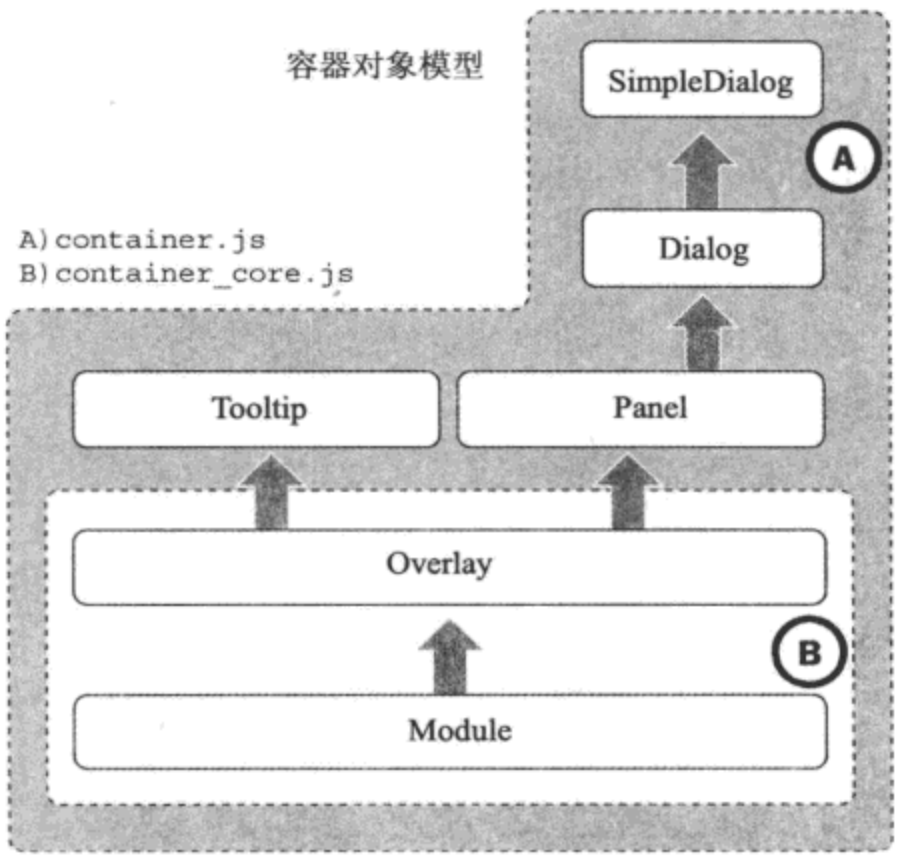


图　11-3

注意，这里有两个基础类 Module 和 Overlay，它们位于较小的 container_core.js 文件中。这两个类(以及剩余的类)也位于较大的 container.js 文件中。此外还要注意的是，如果正在使用 container.js 文件，那么就不再需要 container_core.js 文件。

构建容器的方式有两种，一种是使用 HTML 标记；另一种是纯粹采用 JavaScript 编写。两种方法都有其用处。有时候，我们需要让搜索引擎索引容器的内容，或者让容器能够优雅地降级，这时就需要使用 HTML 呈现容器的内容。还有些时候，如果没有 JavaScript 代码，那么容器的内容就没有什么意义，因此纯粹使用 JavaScript 代码生成该容器就是最有意义的方式。

11.2.1　Module

Module 控件是构建其他控件的基础性组件。很少将 Module 用作独立的控件，这是因为它的主要作用是作为构建其他控件的基础。Module 的标记的最简单形式如下所示：

```
<div id="foo">
    <div class="hd"></div>
    <div class="bd"></div>
    <div class="ft"></div>
</div>
```

注意 CSS 类名，这是 YUI 中反复使用的命名模式：hd 表示头部(header)，bd 表示正文(body)，而 ft 表示尾部(footer)。基于这个标记实例化 Module 对象的代码如下所示：

```
var module = new YAHOO.widget.Module("foo");
```

要让 Module 能够运行，并不需要该标记存在。如果传给构造函数的 ID 并没有指向 DOM 中的某个元素，那么 Module 将创建一个元素，并将传给它的 ID 分配给这个元素。但新建的元素不会自动添加到 DOM 中，还需要使用 Module 的 render 方法完成这项工作：

```
module.render(document.body);
```

请注意传给 render 方法的参数。这个参数告诉 Module 将其自身作为文档的 body 元素的子元素呈现。可以将元素的直接引用(例如在上面的示例中)或者表示元素 ID 的字符串作为参数传入。如果忽略该参数，那么 render 方法将失败并返回 false 值。

1. 插入或附加

在第 8 章中曾经讨论过，Internet Explorer 在 DOM 准备就绪之前向 DOM 中添加内容会导致一个相当严重的问题。当在 script 节点(它并不是 body 元素的直接子元素)中使用 DOM 方法 appendChild 把一个元素附加到文档的 body 元素的末尾时，尤其会发生这个问题。为了在纯粹通过 JavaScript 构建 Module 时解决这个问题，Module 默认使用 YUI 的 insertBefore 方法将其内容添加到 body 元素的起始处。

但是，如果断定 appendChild 方法不会在 IE 中引发任何问题，那么可以简单地通过向 Module 构造函数传入一个配置参数来改变它的这种行为。

```
var module = new YAHOO.widget.Module("foo", {appendtodocumentbody: true});
```

2. 显示和隐藏

可以设置 Module 初始时可见或隐藏。这样一来，就可以在视图中把内容隐藏起来，直到用户单击“show”按钮才将其显示出来。因此，visible 参数的默认值是 true。换句话说就是，Module 的实例在呈现时默认为可见。但是，我们可以通过向构造函数传入一个参数来重写该行为。

```
var module = new YAHOO.widget.Module("foo", {visible: false});
```

一旦实例化 Module 对象，就可以简单地通过调用该对象的 show 和 hide 方法来显示和隐藏它。

```
// Instantiate a hidden module
var module = new YAHOO.widget.Module("foo", {visible: false});

// Show the module
module.show();

// Now hide it
module.hide();
```

3. 配置对象

配置对象隐藏在 Module 控件内部，因此由所有其他容器控件继承。配置对象是一个功能强

大但未做大肆宣传的 YUI 组件，它的作用是充当其父对象属性的监督者。它可用来让属性具有默认值，将属性排队并一次性应用于给定元素，重置属性，检测属性变化并触发自定义事件。在 Module(以及所有其他容器控件)中，配置对象名为 cfg，它是容器对象的直接子对象。

```
// The configuration object
module.cfg
```

setProperty

显示或隐藏一个模块只需要通过配置对象将 visible 属性的值设为 true 或 false，这实际上就是 show 和 hide 模块方法所做的所有工作。

```
// Instantiate a hidden module
var module = new YAHOO.widget.Module("foo", {visible: false});

// Show the module
module.cfg.setProperty("visible", true);

// Now hide it
module.cfg.setProperty("visible", false);
```

getProperty

检索属性值也非常简单。下面这行代码用来检验该模块是否可见：

```
var isVisible = module.cfg.getProperty("visible");
```

在这里，visible 属性返回一个 Boolean 值。

由于 Module 可以配置的属性数量有限，因此在下面介绍更复杂的容器的过程中，我们将进一步了解配置对象的强大功能。

11.2.2　Overlay

Overlay 是增强的 Module。Overlay 控件是最初就很有用的容器控件，它是现成的已经实现的控件，不需要任何额外的编程。虽然 Module 控件的功能非常有限，但是 Overlay 控件提供了大量的有用属性和方法。

1. 定位

对于初学者而言，可以通过 Overlay 控件的 X、Y 和 XY 属性来定位该控件(除非特别指定，否则定位值所采用的单位均是像素)：

```
// Instantiate an Overlay
var overlay = new YAHOO.widget.Overlay("foo");

// Set the overlay's x position to 250
overlay.cfg.setProperty("x", 250);

// Set the overlay's y position to 100
overlay.cfg.setProperty("y", 100);
```

```
// Set both x and y positions at the same time
overlay.cfg.setProperty("xy", [250, 100]);
```

除了指定坐标之外，还可以使用 center 方法或者将配置对象的 fixedcenter 属性设置为 true，从而让 Overlay 控件自动居中。

下面的代码演示如何调用 center 方法：

```
overlay.center();
```

下面的代码演示如何设置 fixedcenter 属性：

```
overlay.cfg.setProperty("fixedcenter", true);
```

有一点非常重要：center 方法告诉 Overlay 将自身居中，而 fixedcenter 属性则告诉该控件持续地保持居中位置，即使当浏览器窗口大小发生改变或者页面滚动时也是如此。

另一种不使用坐标定位 Overlay 控件的方式是通过 context 属性将其固定到页面的某个元素上。

```
overlay.cfg.setProperty("context", ["hd", "tl", "br"]);
```

这里的代码告诉 Overlay 控件，将其左上角(tl)固定到 ID 为 hd 的元素的右下角(br)。

11.2.3 Panel

这里正是容器对象真正变得有趣的地方。在弹出窗口令人厌恶的今天，能够快速地部署一种“仿弹出窗口”非常有价值。与前两个示例一样，使用下面这行代码就可以实例化一个新的 Panel 对象：

```
var panel = new YAHOO.widget.Panel("foo");
```

既然 Panel 只是 Overlay 的扩展，因此我们可以期望它们具有相同的 API 结构。换言之，第一个参数是包含要显示在 Panel 中的内容的元素 ID，或者是动态创建时分配给 Panel 的 ID。第二个参数是包含用户可配置键值对的对象，这些键值对用来重写 Panel 的默认配置值。

```
var config = {
    close: true,
    draggable: true,
    underlay: shadow,
    height: "250px",
    fixedcenter: true,
    modal: true
};
var panel = new YAHOO.widget.Panel("foo", config);
```

Panel 还为一系列容器引入了键盘事件监听程序支持。利用 Panel，可以分配键盘快捷方式支持，这将直接影响 Panel 的行为。因此，下面的代码演示如何将 Esc 键分配给 Panel，这样当按下该键时，Panel 就会关闭。

```
<html>
    <head>
```

```
    <title>Container--Panel</title>
    <link rel="stylesheet" type="text/css" href="reset-fonts-grids.css" />
    <link rel="stylesheet" type="text/css" href="base-min.css" />
    <link rel="stylesheet" type="text/css"
       href="container/assets/skins/sam/container.css">
</head>
<body class="yui-skin-sam">
    <div id="doc">
        <div id="hd">
            <h1>Container--Panel</h1>
            <p>Hello world, this page contains a modal YUI Panel with a
            close button which also happens to be fixed to the center of
            the viewport.</p>
            <p>Note: Pressing the <kbd>Esc</kbd> key also closes the Panel.</p>
        </div>
        <div id="bd">
            <div id="foo">
                <div class="hd">Foo Header (from markup)</div>
                <div class="bd">Foo Body</div>
                <div class="ft">Foo Footer</div>
            </div>
        </div>
        <div id="ft">
        </div>
    </div>
    <script src="yahoo-dom-event.js"></script>
<script src="animation-min.js"></script>
<script src="dragdrop-min.js"></script>
<script src="container-min.js"></script>
<script>
    (function () {
        // Instantiate and render a Panel from markup
        //   (panel is deliberately global to break out of sandbox)

        var config = {
            close: true,
            width: "300px",
            fixedcenter: true,
            modal: true
        };
        panel = new YAHOO.widget.Panel("foo", config);

        var keylistener = new YAHOO.util.KeyListener(
               document,
               {keys: 27},
               {fn: panel.hide, scope: panel, correctScope: true});

        panel.cfg.queueProperty("keylisteners", keylistener);

        panel.render();
```

```
        })();
      </script>
    </body>
</html>
```

在这里，实例化 Panel 并将一个名为 config 的对象传给它，该对象包含的一些自定义参数定义了 Panel 的行为。在创建 Panel 对象之后，实例化一个 KeyListener 对象，并告诉它监听 Esc 键的事件。将 Panel 对象的 hide 方法分配为这个 KeyListener 对象的回调函数。这样一来，每当按下 Esc 键时，就会将 Panel 隐藏起来。

最初看起来，似乎只要通过对象 config 把 keylistener 对象包含到 Panel 中即可。但是，在这里这种做法却行不通，因为 KeyListener 需要一个指向待执行函数的引用，而该函数在 Panel 实例化之前并不存在。因此，这里要做的工作是首先实例化一个 Panel 对象，然后在实例化 KeyListener 对象时将其用作一个引用，最后通过 queueProperty 方法将其传回给 Panel。

queueProperty 方法名副其实，它将属性排队，然后在关键时刻将其引发。在这里，当调用 Panel 对象的 render 方法时，keylistener 对象就会绑定到 Panel。

图 11-4 给出了前面的 Panel 的外观。注意，Panel 后方的页面并不是白色的。这是因为代码将该 Panel 设置为模态，这意味着在激活该面板时，其后方的所有内容均会变灰。

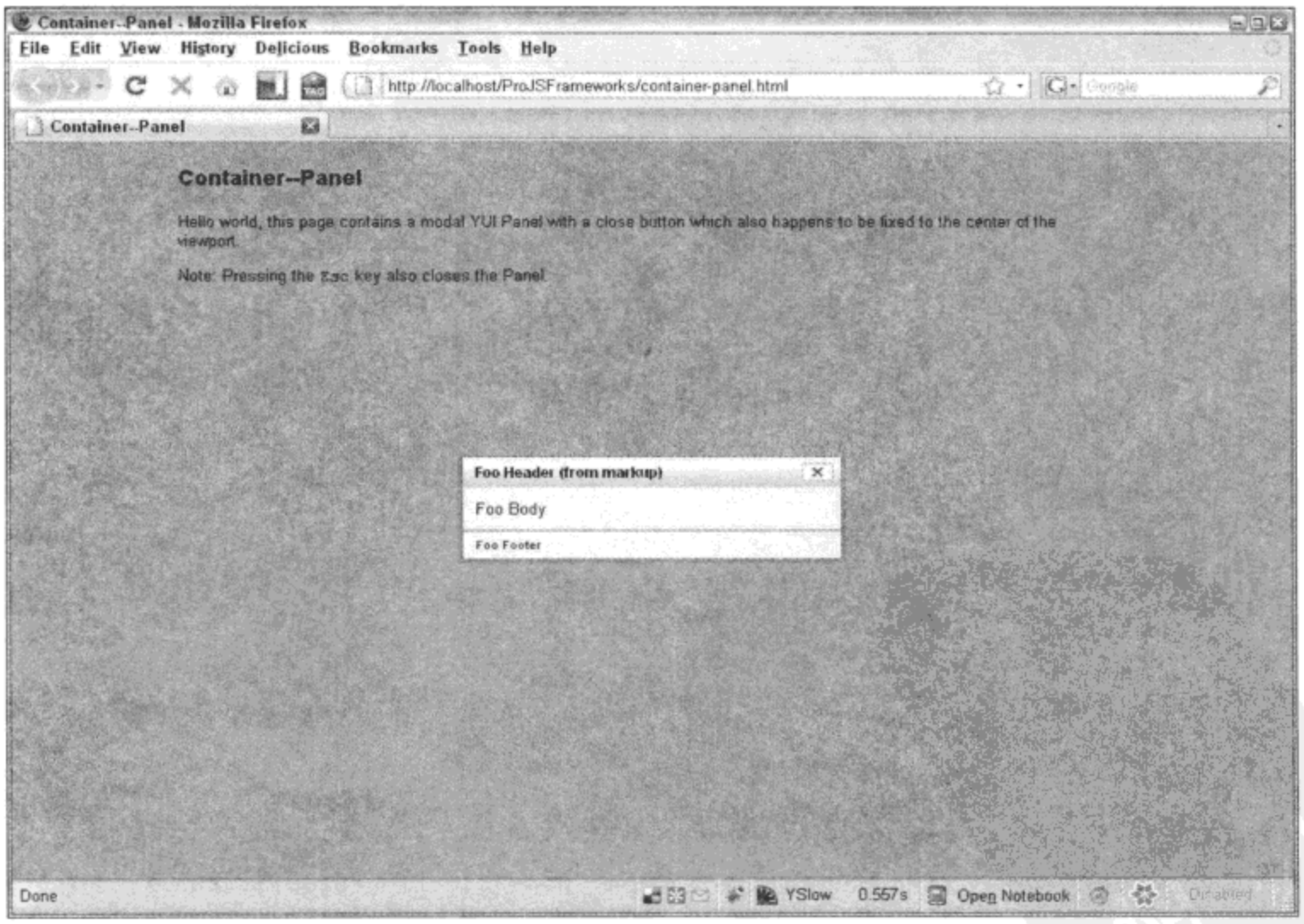


图　11-4

11.3　使用选项卡和树状视图呈现内容

屏幕上只有固定的有限的可用面积，当填满之后就不能再放入其他内容。因此，并不总是能够将全部的可用信息放在一个庞大的块中显示。处理这种空间紧缺的最公认的方式之一是将信息组织成多个较小的块，然后使用可单击的标题表示它们。单击一个标题就会将隐藏的内容显示出来，而之前查看的内容通常会隐藏。从本质上讲，该信息现在分时共享可用的屏幕面积。用来实现这种共享的两种方式是 TabView 和 TreeView。前者最适合于组织信息的简单集合，而后者能够处理层次组织。

11.3.1　TabView

有些时候，需要显示的信息非常长，需要滚动几页才能全部显示。这样带来的问题是大多数人不会通读该信息，呈现的大量数据会使他们感到畏惧。为了帮助他们获取信息，可以将页面的内容划分成更小的、更易于消化的信息块。然后可以使用选项卡表示信息块，让用户能够选择他们希望阅读的部分内容。

YUI 既可以将现有页面数据转换到 TabView 中，也可以纯粹采用 JavaScript 创建 TabView。TabView 的 HTML 结构非常简单。我们把选项卡组织成一个无序列表，每个选项卡是一个列表项，而将内容划分为若干个 div 元素。然后，使用 div 父元素将这两种元素包装起来。

1. 采用 HTML 标记创建 TabView

在下面的示例中，使用 HTML 标记创建 TabView：

```
<html>
    <head>
        <title>Container--Panel</title>
        <link rel="stylesheet" type="text/css" href="reset-fonts-grids.css" />
        <link rel="stylesheet" type="text/css" href="base-min.css" />
        <link rel="stylesheet" type="text/css"
            href="assets/skins/sam/tabview.css"/>
    </head>
    <body class="yui-skin-sam">
        <div id="doc">
            <div id="hd">
                <h1>TabView--Markup</h1>
            </div>
            <div id="bd">

                <!-- START TABVIEW MARKUP -->
                <div id="recipes" class="yui-navset">
                    <ul class="yui-nav">
                        <li>
                            <a href="#seafood">
                                <em>Seafood</em>
                            </a>
                        </li>
```

```
            <li class="selected">
                <a href="#bbq">
                    <em>BBQ</em>
                </a>
            </li>
            <li>
                <a href="#pasta">
                    <em>Pasta</em>
                </a>
            </li>
        </ul>
        <div class="yui-content">
            <div id="seafood">
                <p>Before you prepare any seafood you
                need to go fishing!</p>
            </div>
            <div id="bbq">
                <p>Fire up the grill, it's time to BBQ!</p>
             </div>
            <div id="pasta">
                <p>Will it be spaghetti or lasagna?</p>
            </div>
        </div>
    </div>
    <!-- END TABVIEW MARKUP -->
            </div>
            <div id="ft">
            </div>
        </div>
        <script src="yahoo-dom-event.js"></script>
        <script src="element-min.js"></script>
        <script src="tabview-min.js"></script>
        <script>
            (function () {
                var recipes = new YAHOO.widget.TabView("recipes");
            })();
        </script>
    </body>
</html>
```

实际上，唯一需要的CSS类名是容纳选项卡的ul元素上的yui-nav以及容纳内容块的div元素上的yui-content。TabView自动地将CSS类名yui-navset添加到div主容器中。它还向div主容器中添加第二个CSS类名yui-navset-top。

2. 选项卡方向

TabView之所以在默认情况下会向div主容器中添加CSS类名yui-navset-top，是因为我们可以配置TabView，在内容块的任何一边显示它的选项卡。这个CSS类名可用来根据期望的方向样

式化选项卡。下面的示例演示如何修改 TabView 的方向。

```
// tabs on top
var recipes = new YAHOO.widget.TabView("recipes", {orientation: "top"});

// tabs on right
var recipes = new YAHOO.widget.TabView("recipes", {orientation: "right"});

// tabs on bottom
var recipes = new YAHOO.widget.TabView("recipes", {orientation: "bottom"});

// tabs on left
var recipes = new YAHOO.widget.TabView("recipes", {orientation: "left"});
```

YUI 内置了一组用于定位和样式化各个方向的 TabView 的 CSS 规则，可以在文件夹 assets/skins/sam 内的 tabview.css 文件中找到这些规则。图 11-5~图 11-8 相应地给出了所有 4 个方向的 TabView 的外观。

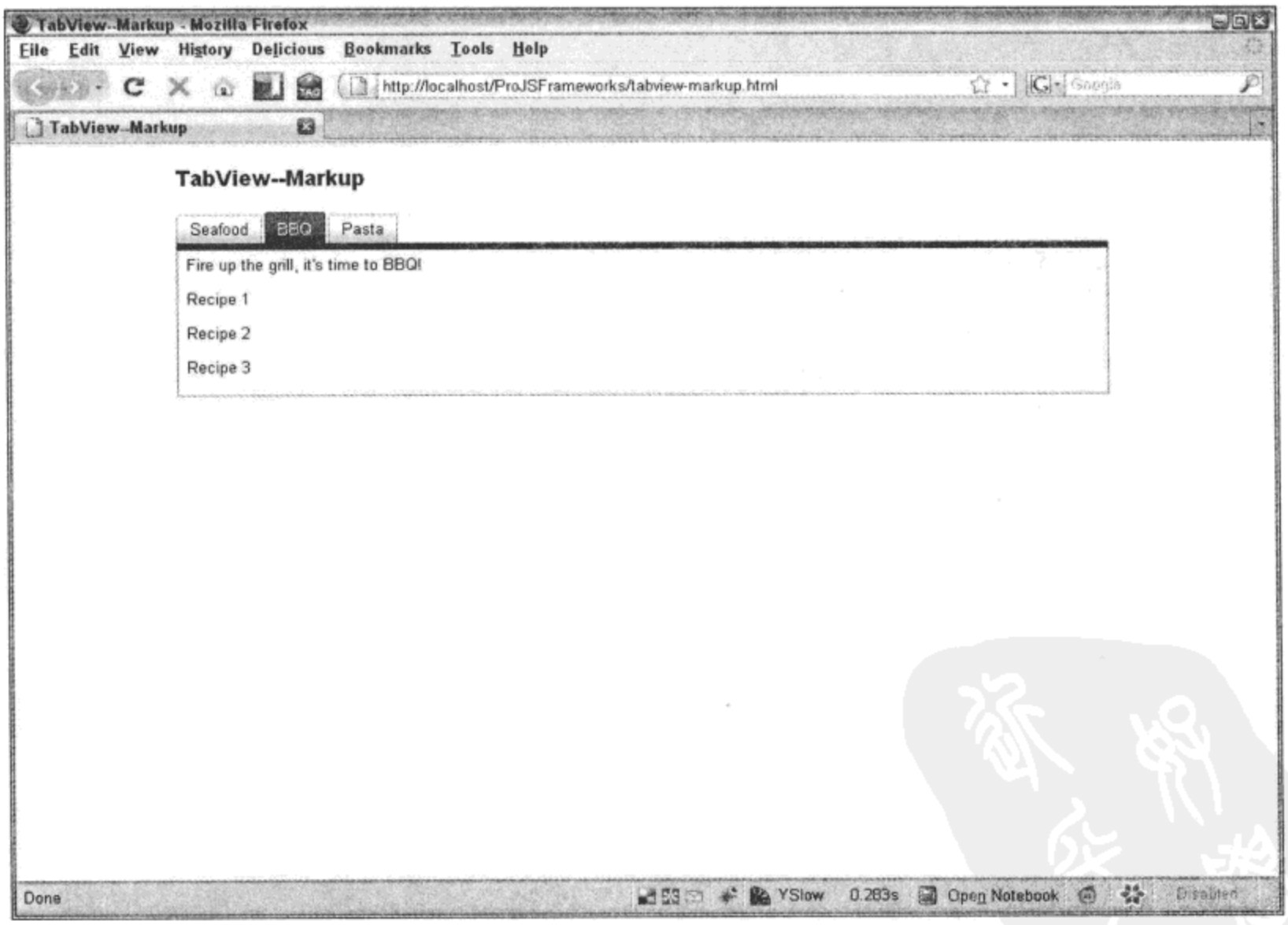


图　11-5

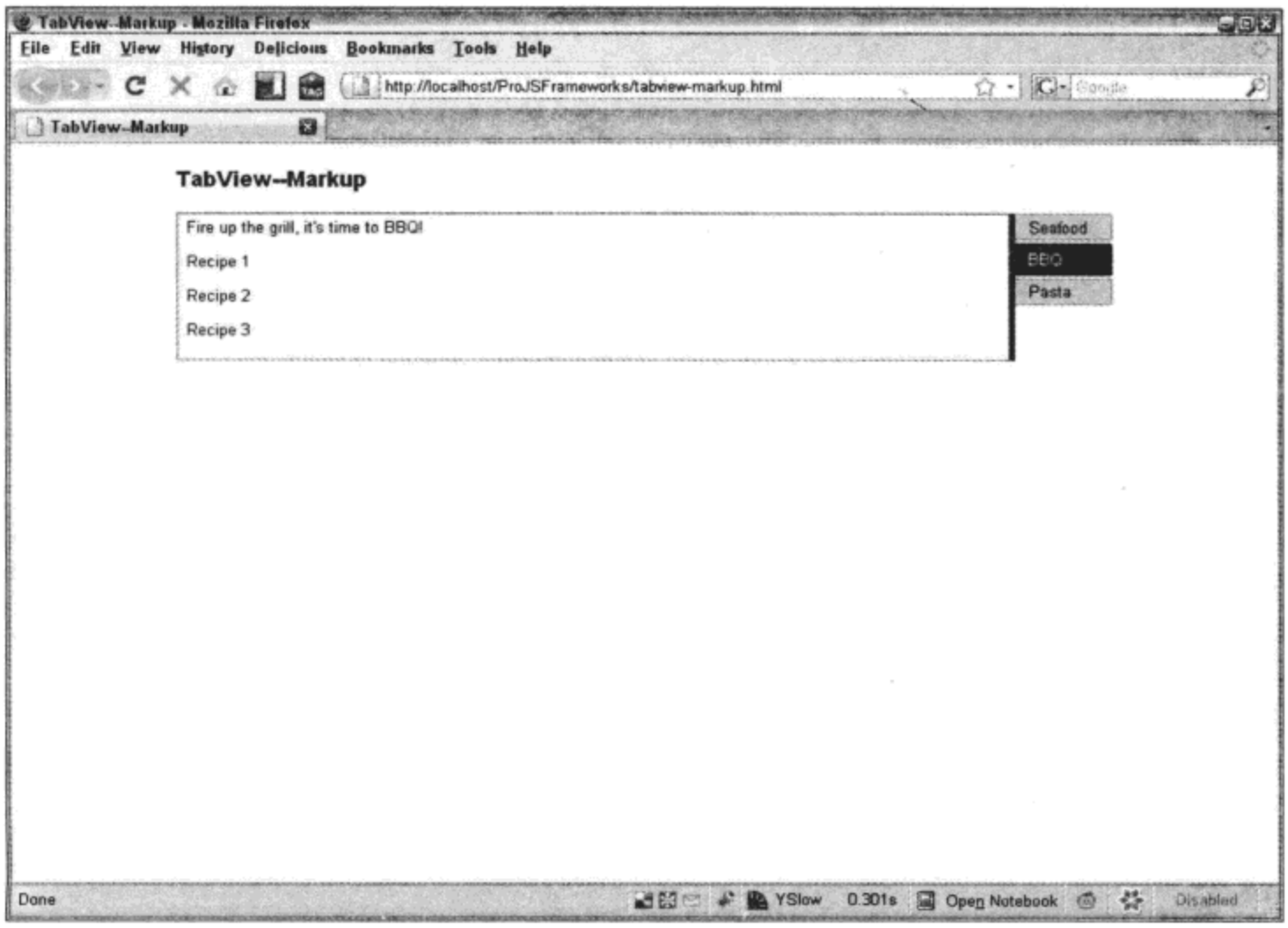

TabView--Markup - Mozilla Firefox
File Edit View History Delicious Bookmarks Tools Help
http://localhost/ProJSFrameworks/tabview-markup.html
TabView--Markup
TabView--Markup
Fire up the grill, it's time to BBQ!
Recipe 1
Recipe 2
Recipe 3
Seafood
BBQ
Pasta
Done
YSlow 0.301s Open Notebook Disabled

图 11-6

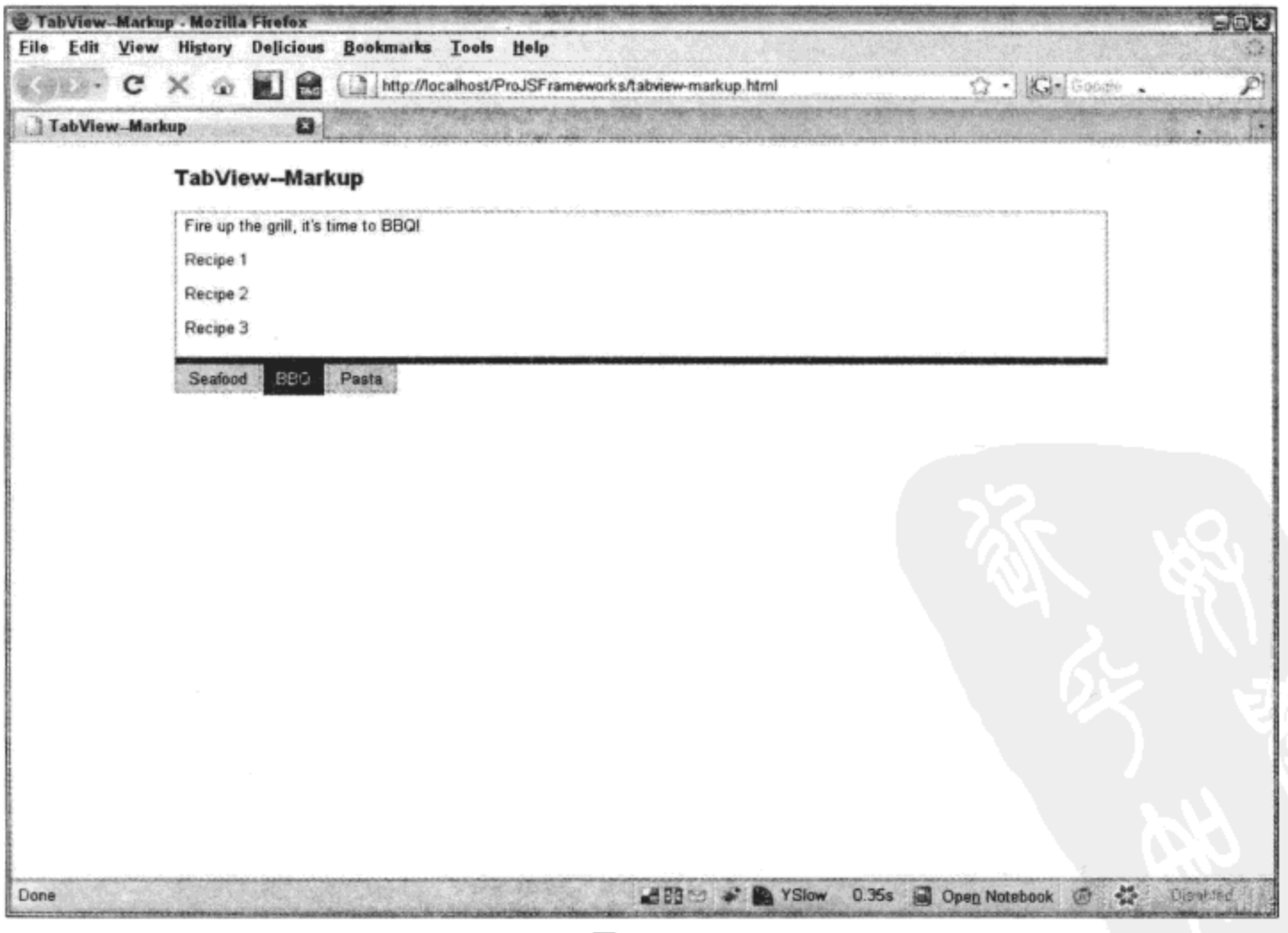

TabView--Markup - Mozilla Firefox
File Edit View History Delicious Bookmarks Tools Help
http://localhost/ProJSFrameworks/tabview-markup.html
TabView--Markup
TabView--Markup
Fire up the grill, it's time to BBQ!
Recipe 1
Recipe 2
Recipe 3
Seafood
BBQ
Pasta
Done
YSlow 0.35s Open Notebook Disabled

图 11-7

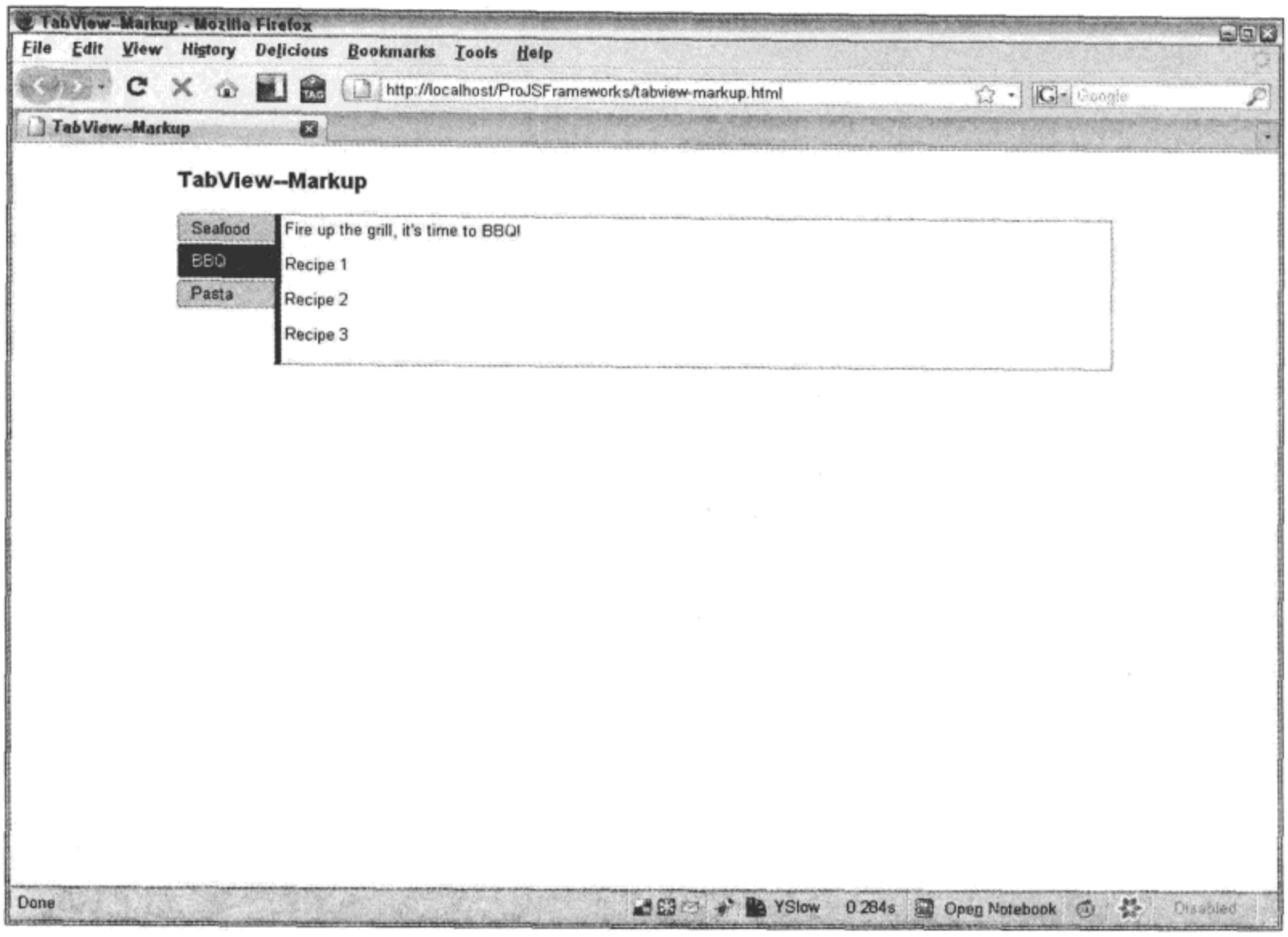


图　11-8

3. 纯粹采用 JavaScript 创建 TabView

前面曾经提到，还可以完全采用 JavaScript 创建 TabView。下面给出了具体的实现代码：

```
<html>
    <head>
        <title>TabView--JavaScript</title>
        <link rel="stylesheet" type="text/css" href="reset-fonts-grids.css" />
        <link rel="stylesheet" type="text/css" href="base-min.css" />
        <link rel="stylesheet" type="text/css"
            href="assets/skins/sam/tabview.css"/>
    </head>
    <body class="yui-skin-sam">
        <div id="doc">
            <div id="hd">
                <h1>TabView--JavaScript</h1>
            </div>
            <div id="bd">
                <div id="recipes"></div>
            </div>
            <div id="ft">
            </div>
```

```
</div>
<script src="yahoo-dom-event.js"></script>
<script src="element-min.js"></script>
<script src="tabview-min.js"></script>
<script>
    (function () {
        var recipes = new YAHOO.widget.TabView("recipes");
        recipes.addTab(new YAHOO.widget.Tab({
            label: "Seafood",
            content: "<p>Before you prepare any seafood you need to " +
                    "go fishing!</p><p>Recipe 1</p><p>Recipe 2</p>" +
                    "<p>Recipe 3</p>"
        }));
        recipes.addTab(new YAHOO.widget.Tab({
            label: "BBQ",
            content: "<p>Fire up the grill, it's time to BBQ!</p>" +
                    "<p>Recipe 1</p><p>Recipe 2</p><p>Recipe 3</p>",
            active: true
        }));
        recipes.addTab(new YAHOO.widget.Tab({
            label: "Pasta",
            content: "<p>Will it be spaghetti or lasagna?</p>" +
                    "<p>Recipe 1</p><p>Recipe 2</p><p>Recipe 3</p>"
        }));
    })();
</script>
    </body>
</html>
```

注意，div 元素 recipes 现在是空的，这是因为它的全部标记现在均由 JavaScript 生成。虽然在前面用于 TabView 的 JavaScript 代码在 TabView 对象实例化之后就结束，但在这里使用 TabView 的 addTab 方法向其中添加选项卡。addTab 方法接受的参数是 YAHOO.widget.Tab 对象。还可以传入第二个参数来指示将该选项卡添加到的位置的索引。如果没有提供索引，那么新选项卡就会附加到现有选项卡集合的末尾。

创建一个新的 Tab 对象是一件相当简单的事情。它可以基于某个 HTML 元素(通过引用或 ID)，也可以动态创建该元素。注意，如果采用 HTML 标记指定选项卡，那么它的 HTML 标记需要遵循前面的 HTML 示例中指定的相同模式。换言之，该选项卡应该是一个含有锚点的列表项，而且它里面最好有一个强调(em)标记。这是因为该选项卡要加入的 TabView 将会把该元素附加到其选项卡列表的末尾，从而将其移到 DOM 中。它将不再位于最初为其定义的位置，当然除非它已经放到了正确的位置。此外，锚点中最好有一个强调标记是因为默认的 YUI CSS 规则期望如此。否则，不需要强调标记，但确实需要锚点。

4. 处理事件

可以将事件处理程序分配给 TabView 或 Tab 对象。无论是 TabView 对象还是 Tab 对象，都支持基于 DOM 的事件(例如 onmouseover 和 onclick)和自定义事件。强烈建议不要直接将事件处理程序分配给表示 TabView 或 Tab 的 HTML 元素。这将确保这些对象上的事件会按照正确的顺序引

发，同时还能够保持使用事件委托所带来的好处。相反，应该使用每种对象原生的 addListener(或其别名 on)事件处理程序。表 11-1 列出了 TabView 和 Tab 提供的自定义事件。

表　11-1

TabView	Tab
activationEventChange	activeIndexChange
activeChange	activeTabChange
beforeActivationEventChange	beforeActiveIndexChange
beforeActiveChange	beforeActiveTabChange
beforeCacheDataChange	beforeOrientationChange
beforeContentChange	beforeTabsChange
beforeContentElChange	orientationChange
beforeContentVisibleChange	tabsChange
beforeDataLoadedChange	
beforeDataSrcChange	
beforeDataTimeoutChange	
beforeDisabledChange	
beforeHrefChange	
beforeLabelChange	
beforeLabelElChange	
beforeLoadMethodChange	
cacheDataChange	
contentChange	
contentElChange	
contentVisibleChange	
dataLoadedChange	
dataSrcChange	
dataTimeoutChange	
disabledChange	
hrefChange	
labelChange	
labelElChange	
loadMethodChange	

在下面的示例中，同时向 Tab 和 TabView 中添加了事件处理程序：

```
<html>
```

```
<head>
    <title>TabView--Events</title>
    <link rel="stylesheet" type="text/css" href="reset-fonts-grids.css" />
    <link rel="stylesheet" type="text/css" href="base-min.css" />
    <link rel="stylesheet" type="text/css"
       href="assets/skins/sam/tabview.css"/>
</head>
<body class="yui-skin-sam">
    <div id="doc">
        <div id="hd">
            <h1>TabView--Events</h1>
        </div>
        <div id="bd">

            <!-- START TABVIEW MARKUP -->
            <div id="recipes">
                <ul class="yui-nav">
                    <li>
                        <a href="#seafood">
                            <em>Seafood</em>
                        </a>
                    </li>
                    <li class="selected">
                        <a href="#bbq">
                            <em>BBQ</em>
                        </a>
                    </li>
                    <li>
                        <a href="#pasta">
                            <em>Pasta</em>
                        </a>
                    </li>
                </ul>
                <div class="yui-content">
                    <div id="seafood">
                        <p>Before you prepare any seafood you
                        need to go fishing!</p>
                        <p>Recipe 1</p>
                        <p>Recipe 2</p>
                        <p>Recipe 3</p>
                    </div>
                    <div id="bbq">
                        <p>Fire up the grill, it's time to BBQ!</p>
                        <p>Recipe 1</p>
                        <p>Recipe 2</p>
                        <p>Recipe 3</p>
                     </div>
                    <div id="pasta">
                        <p>Will it be spaghetti or lasagna?</p>
                        <p>Recipe 1</p>
```

```
                    <p>Recipe 2</p>
                    <p>Recipe 3</p>
                </div>
            </div>
        </div>
        <!-- END TABVIEW MARKUP -->

    </div>
    <div id="ft">
    </div>
</div>
<script src="yahoo-dom-event.js"></script>
<script src="element-min.js"></script>
<script src="tabview-min.js"></script>
<script>
    (function () {
        function clickHandler() {
            tab0.set("label", "Seafood: Breaking News!");
            tab0.set("content", "The ocean is closed. No fishing today.");
        };
        function labelChangeHandler(o) {
            alert("The label changed from '" + o.prevValue +
                  "' to '" + o.newValue + "'");
        };
        function overHandler() {
            // an alert over here would be really annoying
        };
        recipes = new YAHOO.widget.TabView("recipies");
        recipes.on("mouseover", overHandler);
        var tab0 = recipes.getTab(0);
        tab0.on("click", clickHandler);
        tab0.on("labelChange", labelChangeHandler);
    })();
</script>
</body>
</html>
```

在这里，我们将一个 mouseover 事件处理程序 overHandler 添加到 TabView 控件 recipes。此外，还向选项卡集合中的第一个选项卡分配了一个 click 处理程序 clickHandler。最后，有一个 Tab 对象特有的非 DOM 事件 labelChange，我们将处理程序 labelChangeHandler 分配给它。实际上，当单击该选项卡时，它会调用 clickHandler，这会修改它的标签和内容。标签变化会触发 labelChange 事件，这会调用 labelChangeHandler，该处理程序又会引发一个警告框，给出一条消息来显示该标签的旧值和新值。

注意，当把事件对象传给它们的处理程序时，DOM 原生事件的行为符合预期。相反，Tab 和 TabView 特有的非 DOM 事件(表 11-1 中已经列出)将传入一个自定义对象作为它的第一个参数，该对象由属性 newValue、prevValue 和 type 组成，可用来检测值的变化。

图 11-9 显示了单击选项卡集合中第一个选项卡时的情形。

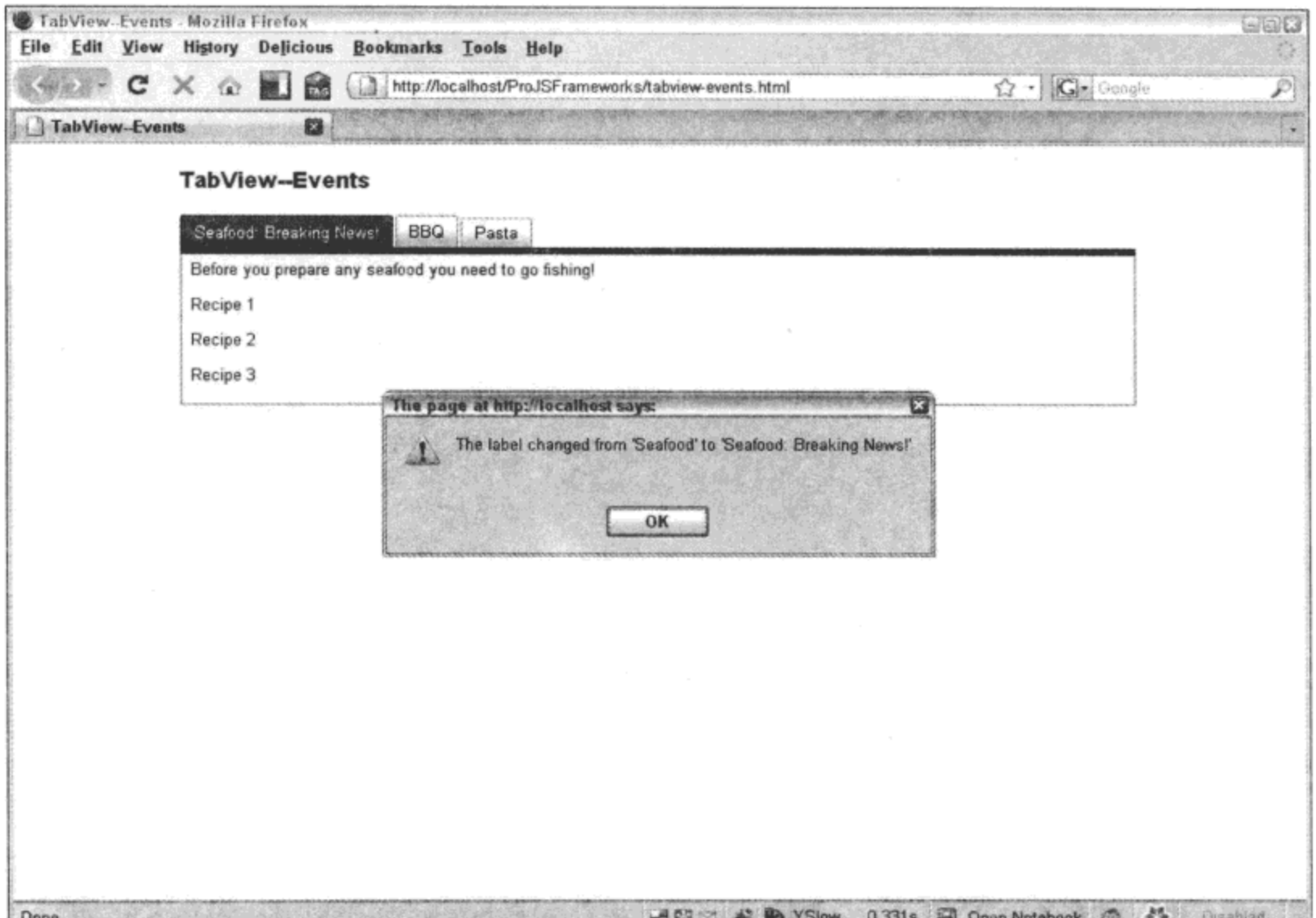


图 11-9

请注意第一个选项卡的标签已经发生变化，但是其中的内容并未发生变化。这是因为我们首先在 clickHandler 处理程序和 alert 方法中调用修改标签的操作，这会暂停所有 JavaScript 操作，从而阻止修改内容操作的执行直到单击 OK 按钮。

11.3.2 TreeView

TreeView 完全由 JavaScript 生成，没有“通过 HTML 标记生成”选项。然而，创建 TreeView 的任务确实非常简单。首先，需要实例化一个 TreeView 对象：

```
var tree = new YAHOO.widget.TreeView("tree");
```

然后，需要将 TreeView 对象的根节点放入一个变量，这样就可以将分支和叶节点附加到根节点上。

```
var root = tree.getRoot();
var chocolate = new YAHOO.widget.TextNode("Chocolate", root, true);
```

在这里，我们创建了一个带有标签“Chocolate”的 TextNode 对象，并将其添加到树状视图的根节点。第三个参数告诉该节点当向其添加子节点时它是否一直打开。换言之，是将其呈现为展

开状态还是折叠状态？在这里，当把子节点添加到该节点中时，它将始终保持打开。在下面的示例中，正在向“Chocolate”节点中添加一个子节点。

```
var sugarless = new YAHOO.widget.TextNode("Sugarless", chocolate, true);
```

请注意如何把这个新节点添加到 TreeView 中。它的第二个参数是该节点的变量名，我们将把新节点添加到该变量中，在这里就是 chocolate。

虽然在这些示例中第一个参数均是字符串类型，但它还可以是一个包含关于该节点更多信息的对象。因此，要想创建一个可单击且带有 ID 的节点，可以采用如下代码：

```
var dark = new YAHOO.widget.TextNode(
                {
                    label: "Dark",
                    href: "http://search.yahoo.com/search?p=Dark+Chocolate"
                }, chocolate, false);
```

在这里，我们创建了一个 TextNode 对象，其 label 属性为“Dark”，href 属性为打开 Yahoo! Search 搜索关键字“Dark Chocolate”的链接。

TextNode 对象将这个信息存储在 data 属性中。但值得注意的是，data 既可以是一个字符串，也可以是一个对象，这取决于它里面所包含的信息。如果标签所包含的信息不仅仅是标签，那么 data 就应该是一个对象。否则，data 就是一个字符串。因此，相应的示例如下所示：

```
sugarless.data // returns "Sugarless"

dark.data // returns an object
dark.data.label // returns "Dark"
```

还可以将任意值传给 data 对象，代码如下：

```
var dark = new YAHOO.widget.TextNode(
                {
                    label: "Dark",
                    href: "http://search.yahoo.com/search?p=Dark+Chocolate",
                    oldmacdonald: "had a farm"
                }, chocolate, false);
```

可以采用与访问标准属性 label 和 href 的值相同的方式来访问任意值：

```
dark.data.oldmacdonald // returns "had a farm"
```

下面是一个完整的 TreeView 示例以及它的呈现结果(参见图 11-10)。

```
<html>
    <head>
        <title>TreeView</title>
        <link rel="stylesheet" type="text/css" href="reset-fonts-grids.css" />
        <link rel="stylesheet" type="text/css" href="base-min.css" />
        <link rel="stylesheet" type="text/css"
            href="assets/skins/sam/treeview.css"/>
```

```
</head>
<body class="yui-skin-sam">
    <div id="doc">
        <div id="hd">
            <h1>TreeView</h1>
        </div>
        <div id="bd">
            <div id="tree"></div>
        </div>
        <div id="ft">
        </div>
    </div>
    <script src="yahoo-dom-event.js"></script>
    <script src="element-min.js"></script>
    <script src="treeview-min.js"></script>
    <script>
        (function () {
            var tree = new YAHOO.widget.TreeView("tree");
            var root = tree.getRoot();

            var trees = new YAHOO.widget.TextNode("Trees", root, true);
            var oak = new YAHOO.widget.TextNode(
                {
                    id: "oaktree",
                    label: "Oak",
                    href: "http://search.yahoo.com/search?p=Oak+Trees",
                    oldmacdonald: "had a farm"
                }, trees, false);
            var birch = new YAHOO.widget.TextNode("Birch", trees, false);
            var pine = new YAHOO.widget.TextNode("Pine", trees, false);
            var spruce = new YAHOO.widget.TextNode("Spruce", trees, false);
            var cedar = new YAHOO.widget.TextNode("Cedar", trees, false);

            var flowers = new YAHOO.widget.TextNode("Flowers", root, true);
            var rose = new YAHOO.widget.TextNode("Rose", flowers, true);
            var carnation = new YAHOO.widget.TextNode("Carnation",
                    flowers, false);
            var tulip = new YAHOO.widget.TextNode("Tulip", flowers, false);

            tree.subscribe("labelClick", function (node) {
                console.log("label clicked", node, node.data, node.data.id);
            });

            tree.draw();
        })();
    </script>
</body>
</html>
```

图　11-10

动态加载

TreeView 对象真正有用的功能是它允许在树层或节点层动态加载节点数据。

具体的实现方式是使用 setDynamicLoad 方法将一个回调函数直接分配给 TreeView 对象或它的节点。

```
tree.setDynamicLoad(loadDataForNode); // assigned to the treeview

textNode.setDynamicLoad(loadDataForNode); // assigned to a text node
```

该回调函数接收两个参数，其中一个是指向当前节点的引用；另一个是所有处理工作完成时执行的函数。后者将告诉 TreeView 该回调函数的工作已经完成。

```
function loadDataForNode(node, nodesAreReady) {
    var newNode = new YAHOO.widget.TextNode("New Node", node, false)
    nodesAreReady();
}
```

在这里，我们将 loadDataForNode 函数分配给 TreeView，针对 TreeView 中的每个节点都会调用这个函数。否则，只针对它所分配的节点调用该函数。

在下面的示例中，我们演示如何动态加载整个树状视图的节点，以及如何呈现该树状视图(参见图 11-11)。

```
<html>
```

```
<head>
    <title>TreeView--Dynamic (Tree)</title>
    <link rel="stylesheet" type="text/css" href="reset-fonts-grids.css" />
    <link rel="stylesheet" type="text/css" href="base-min.css" />
    <link rel="stylesheet" type="text/css"
        href="assets/skins/sam/treeview.css"/>
</head>
<body class="yui-skin-sam">
    <div id="doc">
        <div id="hd">
            <h1>TreeView--Dynamic (Tree)</h1>
        </div>
        <div id="bd">
            <div id="tree"></div>
        </div>
        <div id="ft">
        </div>
    </div>
    <script src="yahoo-dom-event.js"></script>
    <script src="element-min.js"></script>
    <script src="treeview-min.js"></script>
    <script src="json-min.js"></script>
    <script src="connection-min.js"></script>
    <script>
        (function () {
            function loadDataForNode(node, nodesAreReady) {
    var callback = {
        success: function (o) {
            var data = YAHOO.lang.JSON.parse(o.responseText);
            for (var key in data) {
                if (data.hasOwnProperty(key)) {
                    var newNode = new YAHOO.widget.TextNode(
                            {
                                label: data[key].label,
                                id: key
                            }, node, false)
                }
            }
            nodesAreReady();
        }
    };
    var transaction = YAHOO.util.Connect.asyncRequest(
            "GET",
            "treeview-dynamic-" + node.data.id + ".json",
            callback);
}
var tree = new YAHOO.widget.TreeView("tree");
tree.setDynamicLoad(loadDataForNode);
var root = tree.getRoot();
var links = new YAHOO.widget.TextNode({
```

```
                label: "Links",
                id: "links"}, root, false);
            var morelinks = new YAHOO.widget.TextNode({
                label: "More Links",
                id: "morelinks"}, root, false);
            var evenmorelinks = new YAHOO.widget.TextNode({
                label: "Even More Links",
                id: "evenmorelinks"}, root, false);
            var silly = new YAHOO.widget.TextNode({
                label: "OK, this is getting silly",
                id: "silly"}, root, false);
            tree.draw();
        })();
    </script>
</body>
</html>
```

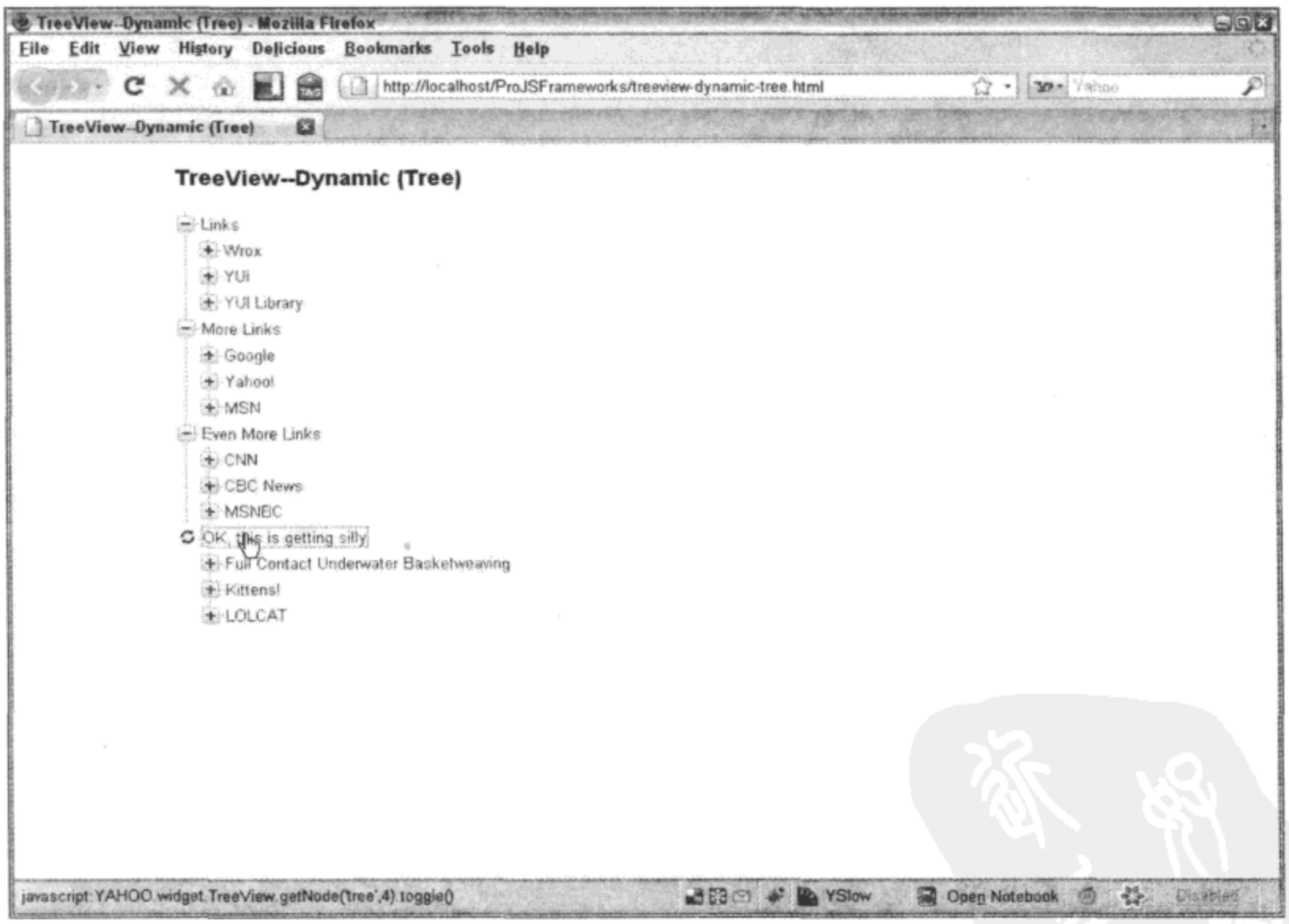


图　11-11

这个示例做的第一件事情(在定义回调函数 loadDataForNode 之后)就是创建一个 TreeView 对象 tree。在此之后，使用该对象的 setDynamicLoad 方法为其分配 loadDataForNode 函数。接下来，创建一些根级节点，动态加载的节点将会附加到这些节点上。

这个示例中的 loadDataForNode 函数稍微有些复杂，它利用 YUI Connection Utility 建立 XHR

调用来获取它的数据。这将演示我们可以从任何地方获取数据。首先，定义 Connection Utility 自己的 callback 对象。出于简洁性考虑，该对象只包含一个用于处理成功情形的函数(换言之，没有错误处理)。因为要处理的每个节点都传给了 loadDataForNode，而且每个节点在创建时都带有 id 参数，所以在收集正确数据时使用了节点 id 的值(这样做就避免了向真正的后端传递参数，从而让该代码能够独立运行)。这个示例使用了 4 个文本文件作为数据源：

- treeview-dynamic-**links**.json
- treeview-dynamic-**morelinks**.json
- treeview-dynamic-**evenmorelinks**.json
- treeview-dynamic-**silly**.json

注意每个文件名中加粗的部分，这就是插入节点 id 值的地方。一旦接收到数据，而且执行了 Connection Utility 的回调函数，那么接下来的事情就比较简单，只要遍历新数据，创建新的节点，最后执行 nodesAreReady 函数来告诉 TreeView 工作已经完成。

下面是上述示例所使用的数据：

treeview-dynamic-links.json

```
{
    "wrox": {
        "label": "Wrox",
        "href": "http://www.wrox.com"
    },
    "yui": {
        "label": "YUI",
        "href": "http://developer.yahoo.com/yui"
    },
    "yuilibrary": {
        "label": "YUI Library",
        "href": "http://yuilibrary.com"
    }
}
```

treeview-dynamic-morelinks.json

```
{
    "google": {
        "label": "Google",
        "href": "http://www.google.com"
    },
    "yahoo": {
        "label": "Yahoo!",
        "href": "http://www.yahoo.com"
    },
    "msn": {
        "label": "MSN",
        "href": "http://www.msn.com"
    }
```

treeview-dynamic-evenmorelinks.json

```
{
    "cnn": {
        "label": "CNN",
        "href": "http://www.cnn.com"
    },
    "cbcnews": {
        "label": "CBC News",
        "href": "http://www.cbcnews.ca"
    },
    "msnbc": {
        "label": "MSNBC",
        "href": "http://www.msnbc.com"
    }
}
```

treeview-dynamic-silly.json

```
{
    "basketweaving": {
        "label": "Full Contact Underwater Basketweaving",
        "href": "http://search.yahoo.com/search?p=full+contact+underwater+basket"
    },
    "kittens": {
        "label": "Kittens!",
        "href": "http://search.yahoo.com/search?p=kittens"
    },
    "lolcat": {
        "label": "LOLCAT",
        "href": "http://search.yahoo.com/search?p=lolcat"
    }
}
```

注意，这个示例的范围限于以下事实：它调用普通文件作为其数据。因此，一旦单击 Links 节点并加载它的子节点，就没有更多数据要显示。

前面曾经提到过，还有更精确的动态加载方式，而不是此处给出的“地毯式轰炸”示例。可以为创建的每个节点分配一个加载器，而不是为整个树状视图分配一个笼统的加载器。下面的示例演示了节点层的动态加载：

```
<html>
    <head>
        <title>TreeView--Dynamic (Node)</title>
        <link rel="stylesheet" type="text/css" href="reset-fonts-grids.css" />
        <link rel="stylesheet" type="text/css" href="base-min.css" />
        <link rel="stylesheet" type="text/css"
            href="assets/skins/sam/treeview.css"/>
    </head>
    <body class="yui-skin-sam">
        <div id="doc">
            <div id="hd">
```

```
            <h1>TreeView--Dynamic (Node)</h1>
        </div>
        <div id="bd">
            <div id="tree"></div>
        </div>
        <div id="ft">
        </div>
    </div>
    <script src="yahoo-dom-event.js"></script>
    <script src="element-min.js"></script>
    <script src="treeview-min.js"></script>
    <script src="json-min.js"></script>
    <script src="connection-min.js"></script>
    <script>
        (function () {
            function loadDataForNode(node, nodesAreReady) {
                var callback = {
                    success: function (o) {
                        var data = YAHOO.lang.JSON.parse(o.responseText);
                        for (var key in data) {
                            if (data.hasOwnProperty(key)) {
                                var newNode = new YAHOO.widget.TextNode(
                                        {
                                            label: data[key].label,
                                            id: key
                                        }, node, false);
                            }
                        }
                        nodesAreReady();
                    }
                };
                var transaction = YAHOO.util.Connect.asyncRequest(
                        "GET", "treeview-dynamic-flowers.json", callback);
            }

            var tree = new YAHOO.widget.TreeView("tree")
            var root = tree.getRoot();
            var oak = new YAHOO.widget.TextNode("Oak", root, false);
            var pine = new YAHOO.widget.TextNode("Pine", root, false);
            var ash = new YAHOO.widget.TextNode("Ash", root, false);
            var flowers = new YAHOO.widget.TextNode("Flowers", root, false);
            flowers.setDynamicLoad(loadDataForNode);
            tree.draw();
        })();
    </script>
</body>
</html>
```

这个示例与前一个示例相比并没有什么不同。与前面一样，它也设置了一个回调函数 loadDataForNode，但在这里它只将其分配给文本节点 flowers。最终的行为是，只有带有标签

“Flowers”的节点才动态加载子节点，如图 11-12 所示。

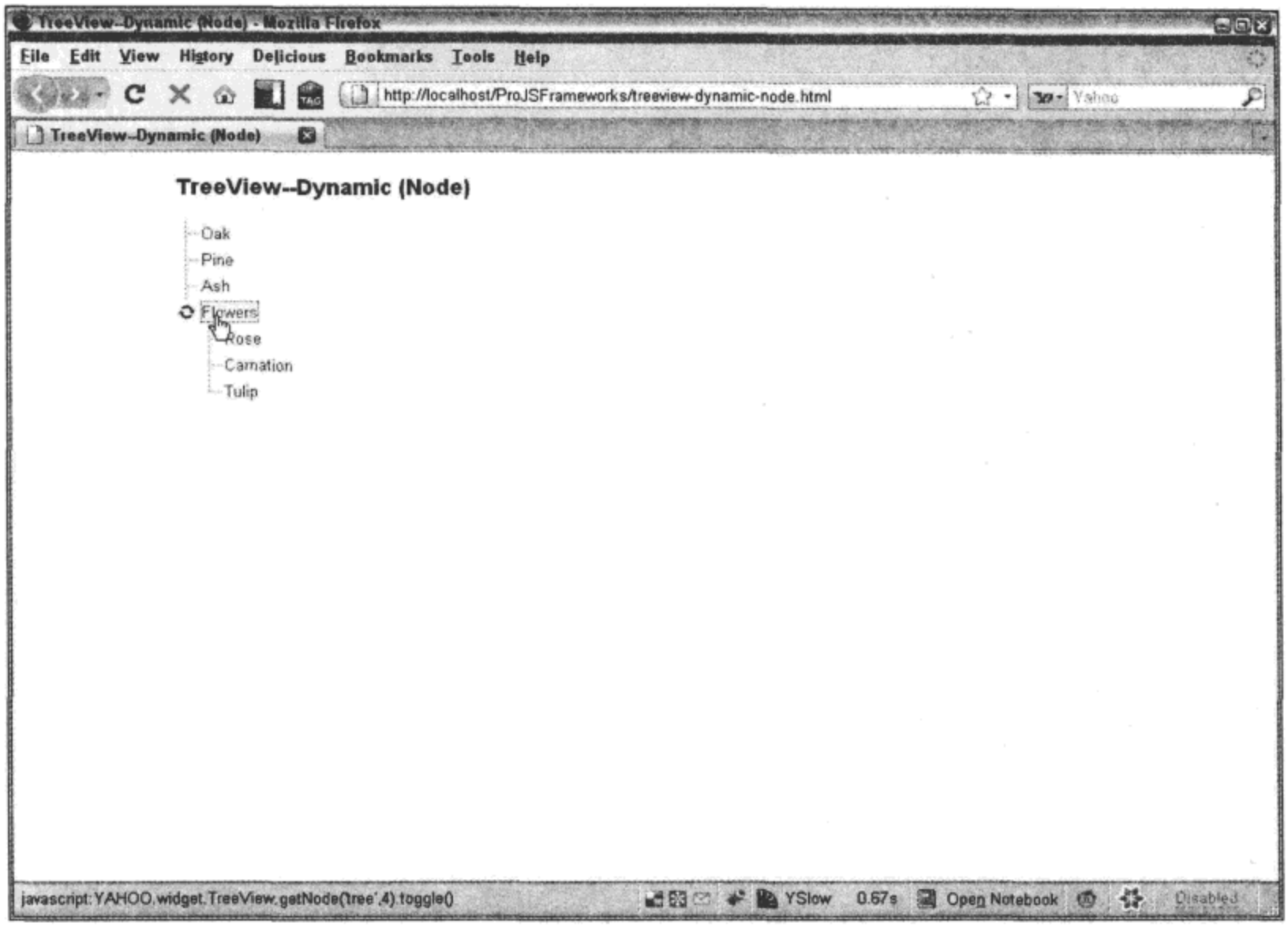


图　11-12

最后，还可以在树状视图中移动节点或移除节点。下面的示例移动几个节点，并将一个节点彻底移除(参见图 11-13)。它通过 appendTo、insertBefore 和 removeNode 方法完成了这些操作。注意，在调用树状视图的 draw 方法之前，任何修改都不会在屏幕上反映出来。类似地，可以通过调用节点的 refresh 方法以只刷新一个节点(例如将 pine 节点附加到 oak 节点)。

```
<html>
    <head>
        <title>TreeView--Move/Remove</title>
        <link rel="stylesheet" type="text/css" href="reset-fonts-grids.css" />
        <link rel="stylesheet" type="text/css" href="base-min.css" />
        <link rel="stylesheet" type="text/css"
            href="assets/skins/sam/treeview.css"/>
    </head>
    <body class="yui-skin-sam">
        <div id="doc">
            <div id="hd">
                <h1>TreeView--Move/Remove</h1>
            </div>
            <div id="bd">
```

```
            <div id="tree"></div>
        </div>
        <div id="ft">
        </div>
    </div>
    <script src="yahoo-dom-event.js"></script>
    <script src="element-min.js"></script>
    <script src="treeview-min.js"></script>
    <script src="json-min.js"></script>
    <script src="connection-min.js"></script>
    <script>
        (function () {
            var tree = new YAHOO.widget.TreeView("tree")
            var root = tree.getRoot();
            var oak = new YAHOO.widget.TextNode("Oak", root, false);
            var pine = new YAHOO.widget.TextNode("Pine", root, false);
            var ash = new YAHOO.widget.TextNode("Ash", root, false);
            var spruce = new YAHOO.widget.TextNode("Spruce", root, false);
            var birch = new YAHOO.widget.TextNode("Birch", root, false);
            var willow = new YAHOO.widget.TextNode("Willow", root, false);
            // Draw the tree with its original nodes
            tree.draw();

            // Remove Oak and append it to Pine as a child
            tree.removeNode(oak);
            oak.appendTo(pine);

            // Remove Birch and insert it before Spruce
            tree.removeNode(birch);
            birch.insertBefore(spruce);

            // Remove Willow altogether
            tree.removeNode(willow);

            // Re-render the tree
            tree.draw();
        })();
    </script>
</body>
</html>
```

YUI 3 中的新增功能

YUI 3 中的窗口部件继承自 Widget 基类，该类将所有窗口部件标准化，使它们都具有 render、init 和 destroy 方法。Widget 类使用“抽象呈现方法，从而在窗口部件之间支持一致的 MVC 结构，并拥有一组共同的窗口部件基本属性、一致的类名生成支持以及插件支持”。

来源：YUI 3 Widget Page。

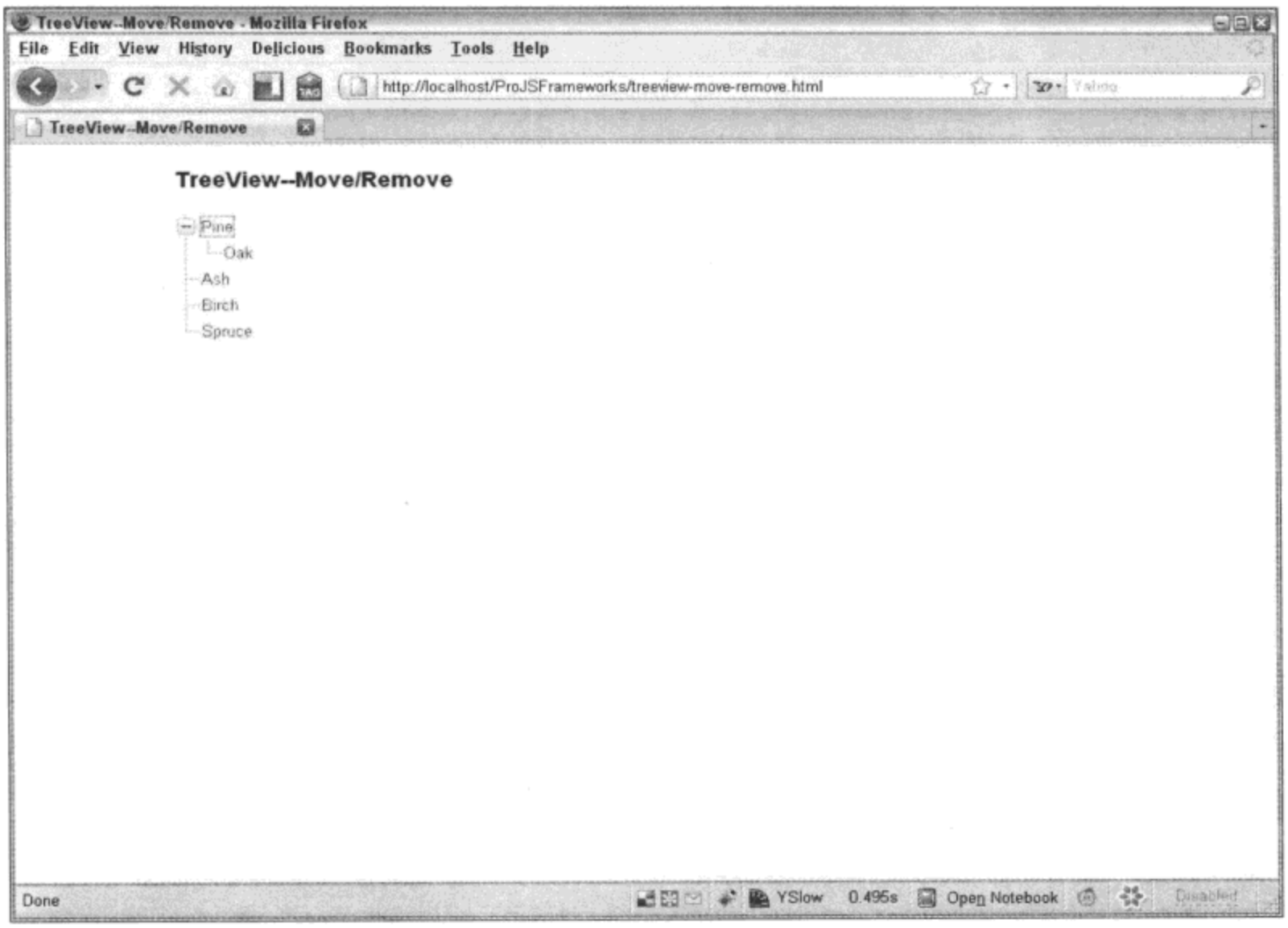


图　11-13

11.4　本章小结

浏览器并不原生地支持自动完成、面板和覆盖图、选项卡视图或树状视图，而所有这些都是现代 Web 设计所需的组件。YUI 利用跨浏览器的、可扩展的窗口部件集圆满地完成了这项任务，从而填补了这个鸿沟。

第12章

利用窗口部件构建用户界面

(第二部分)

有时候，浏览器自带的控件并不能完全满足特殊项目的需要，还需要完成一些额外的工作。

本章内容简介：

- 使用控件
- 编写菜单
- 向表单中添加日期
- 编辑富内容

12.1 装配按钮、滑块和菜单

YUI Library 提供了几个控件来填补浏览器现有控件与现代项目需求之间的鸿沟。

12.1.1 按钮

按钮、单选按钮以及复选框的表示能力和行为能力都相对有限。例如，普通按钮不能表现为单选按钮或复选框，也不能表现为拆分按钮或菜单按钮(分别参见图 12-7 和图 12-8)。这些控件的功能无法使用普通 HTML 按钮实现。但是，YUI Button 控件使得这一切成为可能。YUI Button 不仅功能比普通按钮更强大，而且它们可以具有不同的样式。就行为本身而言，它们表现出与其默认浏览器对应控件相似的行为。换言之，YUI 提交按钮将提交它的父表单，而 YUI 重置按钮将重置它的父表单。

12.1.2 样式化

每个 YUI Button 的根元素有两个 CSS 类，通用的.yui-button 和特有的.yui-[type]-button，其中

[type]是按钮类型(完整的按钮类型列表请参见本章后面的相关内容)。按钮根元素还可以接收两个CSS 类名以进行状态管理，分别是通用的.yui-button-[state]和特有的.yui-[type]-button-[state]。有效的状态有 focus、hover、active 和 disabled。注意，可以一次性向一个元素分配多个表示状态的 CSS 类，例如 focus 和 hover。

1. 按钮类型

下面是使用 YUI Button 控件能够创建的按钮类型，图 12-1 到图 12-8 演示了这些按钮的默认样式。但是，我们可以使用 CSS 修改所有这些按钮的外观。

名称：下压按钮
类型：push

图　12-1

名称：链接按钮
类型：link

图　12-2

名称：复选框
类型：checkbox

图　12-3

名称：单选按钮
类型：radio

图　12-4

名称：重置按钮
类型：reset

图　12-5

名称：提交按钮
类型：submit

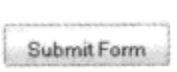


图　12-6

名称：拆分按钮
类型：split

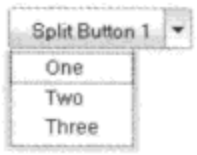


图　12-7

名称：菜单按钮

类型：menu

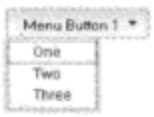


图　12-8

2. 使用 HTML 标记创建按钮

在基于 HTML 标记创建按钮时，<input>、<button>或<a>是有效的标记，而且是预定义的 YUI Button HTML 标记。下面的示例给出了预定义的 HTML 标记的代码：

```
<span id="yuilinkbutton" class="yui-button">
    <span class="first-child">
        <a href="http://developer.yahoo.com/yui">YUI</a>
    </span>
</span>
```

一旦标记准备就绪并在 DOM 中可用，实例化一个 Button 对象并通过该对象的 ID(在这里就是 yuilinkbutton)将其指向某个元素就是非常简单的事情：

```
var linkbutton1 = new YAHOO.widget.Button("yuilinkbutton");
```

在这里没有将上面的锚点嵌套到两个 span 标记中，这个锚点本身也可以只是引用：

```
<a href="http://developer.yahoo.com/yui" id="yuilinkbutton">YUI</a>
```

这就是为什么不真正推荐第一个标记示例(预定义 YUI Button HTML)的原因，因为指向一个简单的锚点已经足够。YUI Button 的构造函数在实例化时创建必要的标记。

3. 使用 JavaScript 创建按钮

在采用 JavaScript 创建 YUI Button 对象时，唯一需要的标记是提供一个元素引用，创建的按钮将放在该元素中呈现。

```
var linkbutton2 = new YAHOO.widget.Button({
    id: "yuilinkbutton2",
    type: "link",
    label: "YUI",
    container: "foo"
});
```

在这里，Button 构造函数只带一个参数：一个含有在创建按钮时要用到的信息的对象字面值。第一个值是 ID，在按钮创建时会把该值分配给它。第二个值是类型(要了解可用类型的完整列表，请参见本章前面相关内容)。第三个值是要在按钮中显示的文本。第四个值是元素 ID，它说明在

哪个元素中呈现这个按钮。注意，在基于现有标记创建 Button 对象时，构造函数会自动确定需要创建什么类型的按钮(根据它指向的元素)。但是，在这里有必要指定一个类型。

4. 按钮组

单选按钮的 YUI Button 对等控件是按钮组(Button Group)。可以通过容器元素的 ID 来指向一组单选按钮，或者只使用 JavaScript 从头开始生成这些按钮。

```
<div id="radiogroup1">
    <input type="radio" id="radio1" name="radio" value="one" checked="checked" />
    <input type="radio" id="radio2" name="radio" value="two" />
    <input type="radio" id="radio3" name="radio" value="three" />
    <input type="radio" id="radio4" name="radio" value="four" />
</div>
```

可以使用下面这行代码将这一组单选按钮转换成一个按钮组：

```
var radio1 = new YAHOO.widget.ButtonGroup("radiogroup1");
```

下面的代码演示如何只使用 JavaScript 代码生成一个由 4 个单选按钮组成的按钮组：

```
var radio1 = new YAHOO.widget.ButtonGroup({
    id: "radiogroup1",
    name: "somechoice",
    container: "radiogroup1"
});

radio1.addButtons([
    {label: "One", value: "1"},
    {label: "Two", value: "2"},
    {label: "Three", value: "3"},
    {label: "Four", value: "4"}
]);
```

5. 菜单按钮和拆分按钮

最后，拆分按钮也是一种常见的桌面应用程序约定，在单击这种按钮时，它会表现出普通按钮的行为；但在单击它的向下箭头图标时，还会出现多个可选的选项(参见图 12-7)。菜单按钮(参见图 12-8)与拆分按钮之间的主要差别在于，前者并没有默认的单击值，而后者有。

这两种类型的按钮的基本标记是一样的，均由一个按钮和一个选择元素组成。菜单按钮和拆分按钮控件将输入和选择元素折叠成一个按钮。下面给出了拆分(或菜单)按钮的标记示例：

```
<input type="submit" id="menubutton1" name="menubutton1" value="Menu" />
<select id="menubutton1select" name="menubutton1select" multiple="multiple">
    <option value="1">One</option>
    <option value="2">Two</option>
    <option value="3">Three</option>
</select>
```

实例化一个对象非常简单，就是指向这些元素并指定要构建按钮的类型：

```
var menubutton1 = new YAHOO.widget.Button(
    "menubutton1",
```

```
    {
        type: "menu",
        menu: "menubutton1select"
    });
```

6. 事件处理

与普通表单元素一样，也可以将事件处理程序绑定到 YUI Button。该操作实际上非常简单。除了像 onclick 这样的标准 DOM 事件之外，还有其他几个 YUI Button 特有的事件可供绑定。

下面的示例演示如何将事件处理程序分配给 YUI Button 的标准 DOM 事件：

```
function clickHandler () {
    alert("click");
};
var pushbutton1 = new YAHOO.widget.Button("pushbutton1");
pushbutton1.on("click", clickHandler);
```

表 12-1 列出了所有可供绑定的 YUI Button 特有事件。

表　12-1

事　　件	说　　明
beforeCheckedChange	在配置属性'checked'的值发生变化之前引发。返回 false 可以取消该属性修改
beforeContainerChange	在配置属性'container'的值发生变化之前引发。返回 false 可以取消该属性修改
beforeDisabledChange	在配置属性'disabled'的值发生变化之前引发。返回 false 可以取消该属性修改
beforeFocusmenuChange	在配置属性'focusmenu'的值发生变化之前引发。返回 false 可以取消该属性修改
beforeHrefChange	在配置属性'href'的值发生变化之前引发。返回 false 可以取消该属性修改
beforeLabelChange	在配置属性'label'的值发生变化之前引发。返回 false 可以取消该属性修改
beforeLazyloadmenuChange	在配置属性'lazyloadmenu'的值发生变化之前引发。返回 false 可以取消该属性修改
beforeMenuChange	在配置属性'menu'的值发生变化之前引发。返回 false 可以取消该属性修改
beforeMenuclassnameChange	在配置属性'menuclassname'的值发生变化之前引发。返回 false 可以取消该属性修改
beforeNameChange	在配置属性'name'的值发生变化之前引发。返回 false 可以取消该属性修改
beforeOnclickChange	在配置属性'onclick'的值发生变化之前引发。返回 false 可以取消该属性修改
beforeSelectedMenuItemChange	在配置属性'selectedMenuItem'的值发生变化之前引发。返回 false 可以取消该属性修改
beforeSrcelementChange	在配置属性'srcelement'的值发生变化之前引发。返回 false 可以取消该属性修改
beforeTabindexChange	在配置属性'tabindex'的值发生变化之前引发。返回 false 可以取消该属性修改
beforeTargetChange	在配置属性'target'的值发生变化之前引发。返回 false 可以取消该属性修改
beforeTitleChange	在配置属性'title'的值发生变化之前引发。返回 false 可以取消该属性修改
beforeTypeChange	在配置属性'type'的值发生变化之前引发。返回 false 可以取消该属性修改
beforeValueChange	在配置属性'value'的值发生变化之前引发。返回 false 可以取消该属性修改
blur	当菜单项失去输入焦点时引发。传回一个表示原始 DOM 事件对象(当该事件引发时由事件实用工具 YAHOO.util.Event 传回该对象)的对象
checkedChange	当配置属性'checked'的值发生变化时引发
containerChange	当配置属性'container'的值发生变化时引发
disabledChange	当配置属性'disabled'的值发生变化时引发

(续表)

事　　件	说　　明
focus	当菜单项接收输入焦点时引发。传回一个表示原始 DOM 事件对象(当该事件引发时由事件实用工具 YAHOO.util.Event 传回该对象)的对象
focusmenuChange	当配置属性'focusmenu'的值发生变化时引发
hrefChange	当配置属性'href'的值发生变化时引发
labelChange	当配置属性'label'的值发生变化时引发
lazyloadmenuChange	当配置属性'lazyloadmenu'的值发生变化时引发
menuChange	当配置属性'menu'的值发生变化时引发
menuclassnameChange	当配置属性'menuclassname'的值发生变化时引发
nameChange	当配置属性'name'的值发生变化时引发
onclickChange	当配置属性'onclick'的值发生变化时引发
option	当用户调用按钮的选项时引发。传回一个表示导致引发这个"option"事件的原始 DOM 事件("mousedown"或"keydown")的对象
selectedMenuItemChange	当配置属性'selectedMenuItem'的值发生变化时引发
srcelementChange	当配置属性'srcelement'的值发生变化时引发
tabindexChange	当配置属性'tabindex'的值发生变化时引发
targetChange	当配置属性'target'的值发生变化时引发
titleChange	当配置属性'title'的值发生变化时引发
typeChange	当配置属性'type'的值发生变化时引发
valueChange	当配置属性'value'的值发生变化时引发

来源：YUI Button API 文档

除了 blur、focus 和 option 事件之外，所有其他事件都会接收一个包含两个属性的事件对象：prevValue 和 newValue。因为当属性改变时所有这些事件都会触发，所以在跟踪按钮变化时，prevValue 和 newValue 属性可能非常有用。

下面的示例演示如何设置和触发非 DOM 事件：

```
function changeHandler (e) {
    alert("Changing from " + e.prevValue + " to " + e.newValue);
};
var pushbutton1 = new YAHOO.widget.Button("pushbutton1");
pushbutton1.on("labelChange", changeHandler);
pushbutton1.set("label", "Something new");
```

12.1.3　滑块

滑块(例如音量控件和颜色滑块)是一种明确地不会在任何浏览器上成为标准组件的控件。到目前为止，获取一个返回值的可拖动控件的唯一方式是使用一些精心设计的鼠标跟踪 JavaScript 代码。但是，编写实现拖动和鼠标跟踪功能的代码非常困难。然而，YUI Slider 组件为我们提供了可以工作的垂直、水平以及区域滑动控件。

1. 水平滑块和垂直滑块

建立 YUI Slider 对象是非常简单的事情。首先需要的就是一些由两个嵌套容器和一幅图像组成的基本 HTML 代码。第一个容器用作容纳第二个容器滑动的区域，而图像用作拖动的实际“滑块”元素。下面给出了 HTML 代码示例：

```
<div id="sliderbg">
    <div id="sliderthumb"><img src="sliderthumb.gif" /></div>
</div>
```

一旦 HTML 代码就绪，那么只需要一行 JavaScript 代码就可以实例化滑块对象：

```
var slider = YAHOO.widget.Slider.getHorizSlider("sliderbg", "sliderthumb", 0, 275);
```

在这里并没有使用 new 关键字完成对象实例化，而是使用 getHorizSlider 函数返回一个新的滑块对象。前两个参数用来说明到哪里查找两个滑块容器元素，接下来的两个参数用来说明如何滑动。在这个示例中，滑块不能移出左端(0 值)，可以滑动 275 像素到达右端。在下面这个完整的示例中，图像 sliderthumb.gif 的宽度为 75 像素，而 sliderbg 元素的宽度为 350 像素。因此，将滑块的水平滑动距离限制为 275 像素意味着，滑块图像将始终位于 sliderbg 元素内部。

一旦实例化滑块控件，就可以为其指定一个初始值，只要按照下面的代码设置即可(图 12-9 中的初始值为 0)：

```
slider.setValue(50);
```

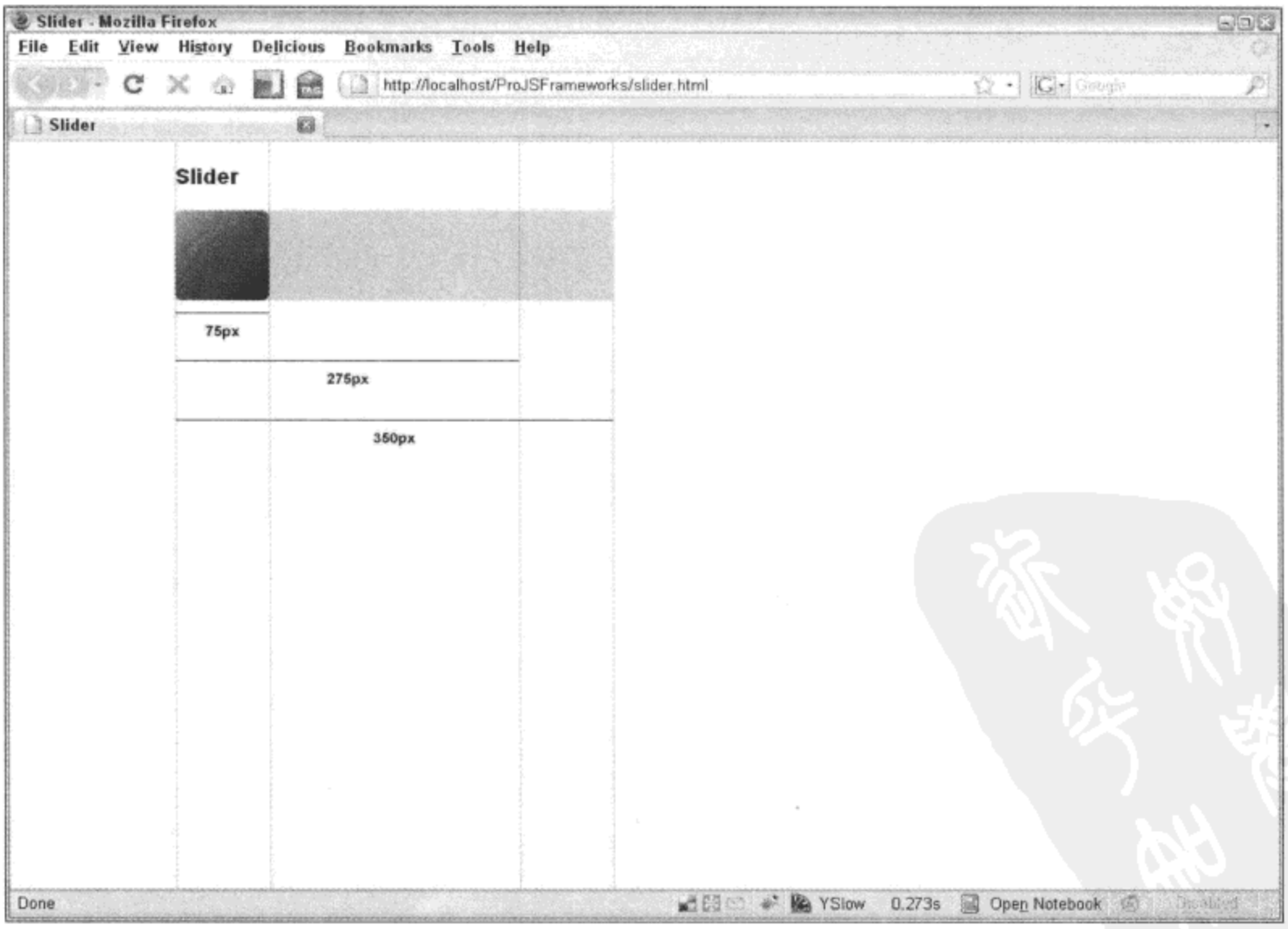


图　12-9

下面是图 12-9 对应的代码清单：

```
<html>
    <head>
        <title>Slider</title>
        <link rel="stylesheet" type="text/css" href="reset-fonts-grids.css" />
        <link rel="stylesheet" type="text/css" href="base-min.css" />
        <style type="text/css">
            #sliderbg {
                background: #ccc;
                width: 350px;
            }
            #sliderthumb {
                width: 75px;
            }
        </style>
    </head>
    <body class="yui-skin-sam">
        <div id="doc">
            <div id="hd">
                <h1>Slider</h1>
            </div>
            <div id="bd">
                <div id="sliderbg">
                    <div id="sliderthumb"><img src="sliderthumb.gif" /></div>
                </div>
                <div id="ft">
                </div>
            </div>
            <script src="yahoo-dom-event.js"></script>
            <script src="dragdrop-min.js"></script>
            <script src="slider-min.js"></script>
            <script>
                (function () {
                    var slider = YAHOO.widget.Slider.getHorizSlider("sliderbg",
                            "sliderthumb", 0, 275);
                })();
            </script>
        </body>
    </html>
```

我们也可以使用 getVertSlider 函数方便地创建垂直滑块。这个函数的构造函数附带的参数以及产生的结果与 getHorizSlider 函数相同，只是使用上和下方向换掉了左和右方向。

实际上，getHorizSlider 和 getVertSlider 函数的构造函数均附带第五个参数，它的作用是为滑块添加刻度。换言之，如果将值 25 作为第五个参数传入，那么滑块控件中的滑块在每次水平或垂直拖动时跳跃 25 像素。如果要求用户选择的值必须以步进 x(而不是 a 和 b 之间的任意值)递增，

那么这个参数就非常有用。

YUI Slider 的功能之一是：在 sliderbg 元素内的 sliderthumb 图像的任何一侧单击都会让滑块朝着该侧移动。实际上，这是滑块控件正常的、符合预期的行为(例如浏览器的滚动栏)。如果使用了 YUI Animation Utility，那么滑块将以动画的形式缓慢移动到单击的地方，而不是简单地跳跃到那里。如果这并不是想要的行为，那么可以像下面这样将其禁用：

```
var slider = YAHOO.widget.Slider.getHorizSlider("sliderbg", "sliderthumb", 0, 275);
slider.animate = false;
```

当然，如果不能访问滑块产生的信息，那么它就没有什么用处。YUI Slider 提供 3 种可以附加处理程序的事件，分别是 slideStart、slideEnd 和 change。slideStart 和 slideEnd 均不会向回调函数传递任何实参。然而，change 确实传递一个实参，即滑块的当前像素位置。因此，在上面的示例中，100%的值就是 275。

下面的示例演示如何将事件处理程序附加到滑块控件：

```
var slider = YAHOO.widget.Slider.getHorizSlider("sliderbg", "sliderthumb", 0, 275);
slider.subscribe("slideStart", function () {
    // slide action has started
});
slider.subscribe("slideEnd", function () {
    // slide action has ended
});
slider.subscribe("change", function (val) {
    // val contains the current slider value
});
```

2. 区域滑块

还可以使用一种二维滑块，其滑块元素既可以沿着垂直方向移动，也可以沿着水平方向移动。实现这种二维滑块(即区域滑块)也非常简单，只需要使用不同的构造函数，然后多添加几个参数。

```
var slider = YAHOO.widget.Slider.getSliderRegion("sliderbg", "sliderthumb",
       0, 275, 0, 275);
```

前两个参数仍然是 DOM 中滑块元素的 ID。第三个和第四个参数分别是左右像素限制，第五个和第六个参数分别是上下像素限制(参见图 12-10)。与一维滑块中的对应组成部分一样，这种滑块也使用了相同的尺寸，滑块元素仍为 75 像素宽和高，而滑块控件自身仍为 350 像素宽和高。

可以像下面这样设置区域滑块的初始值：

```
slider.setRegionValue(50, 100);
```

其中，第一个值是 x 偏移量，而第二个值是 y 偏移量。

除了接收坐标值以外，setValue(用于一维滑块)和 setRegionValue 还可以附带 3 个可选的 Boolean 参数。第一个参数用来确定是否对滑块操作制作动画(假设有 animation-min.js)；第二个参数用来确定如果滑块值锁定，那么是否强制设置该值；第三个参数用来确定是否应该静默地设置值。如果第三个参数设置为 true，那么这个操作将不会引发滑块的任何事件。

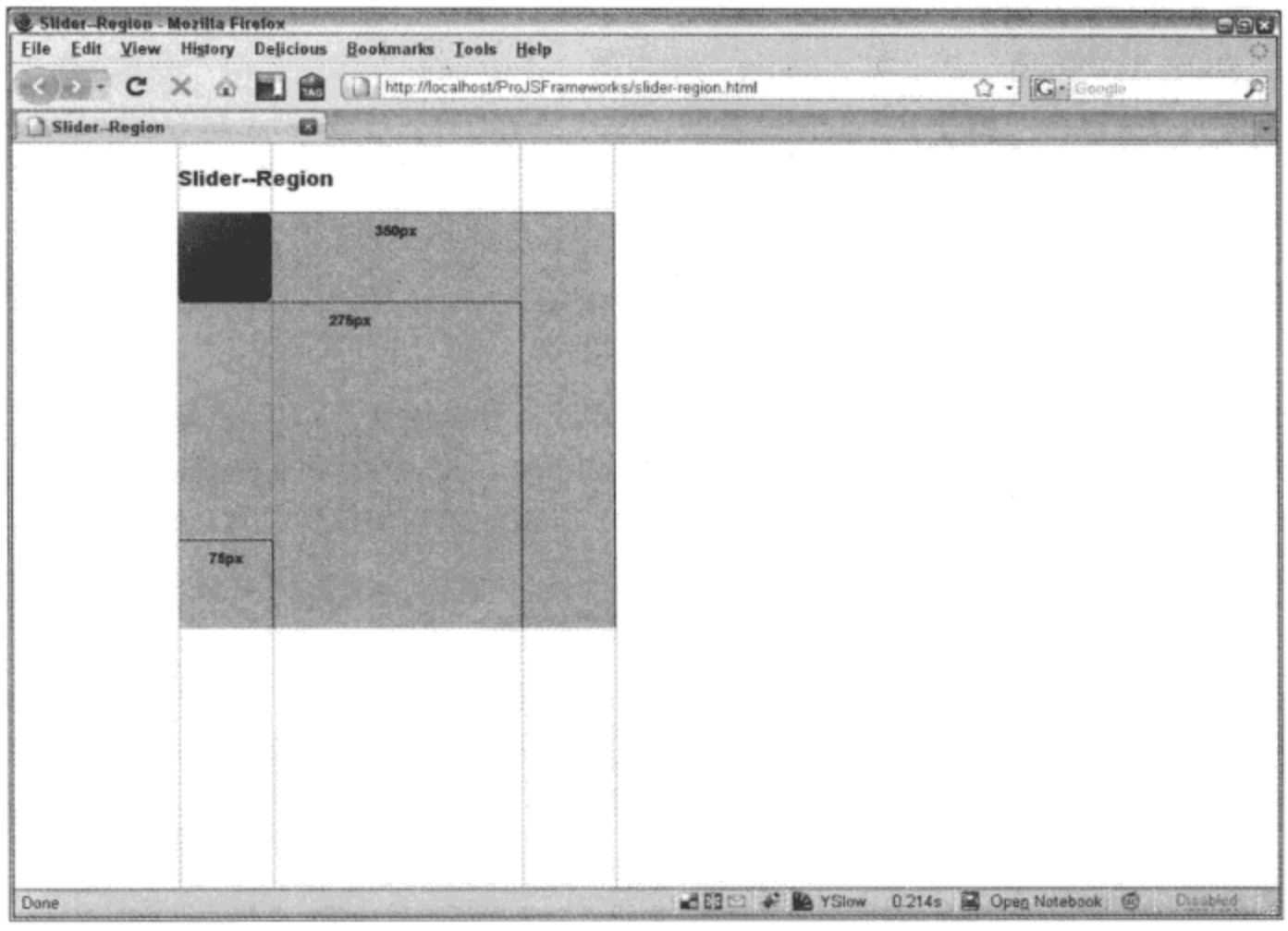


图 12-10

因为区域滑块的滑块元素沿着 x 轴和 y 轴移动，所以 change 事件处理程序会接收到两个值(以对象的形式传入)。因此，区域滑块的 change 事件处理程序并不会像下面这样接收值：

```
function changeHandler(val) {
    // handler code here
}
```

而是像下面这样同时接收 x 和 y 值：

```
function regionChangeHandler(vals) {
    var x = vals.x;
    var y = vals.y;
}
```

3. 双重滑块

最后，还有一种双重滑块，它实际上是两个共享相同背景容器的垂直滑块或水平滑块。双重滑块操作的并不是一个值，相反，它可用来操作最小值和最大值。

图 12-11 演示双重滑块的实际运行情况，在滑块下方更新显示每个滑块元素的值。

关于定位

在上面的示例中，没有必要明确地定位滑块背景或它的滑块元素。换言之，滑块背景仍然是 static(所有未定位元素的 CSS 默认值)，而 Slider 代码会自动将滑块元素设为 relative。这样做给双重滑块带来了一个问题：当相对定位时两个滑块元素不能位于同一行(参见图 12-12)。因此，有必要明确地将这两个滑块元素定位为 absolute。但是，这样一来就要求把滑块背景设为 relative，以

便正确容纳采用绝对定位的滑块元素。

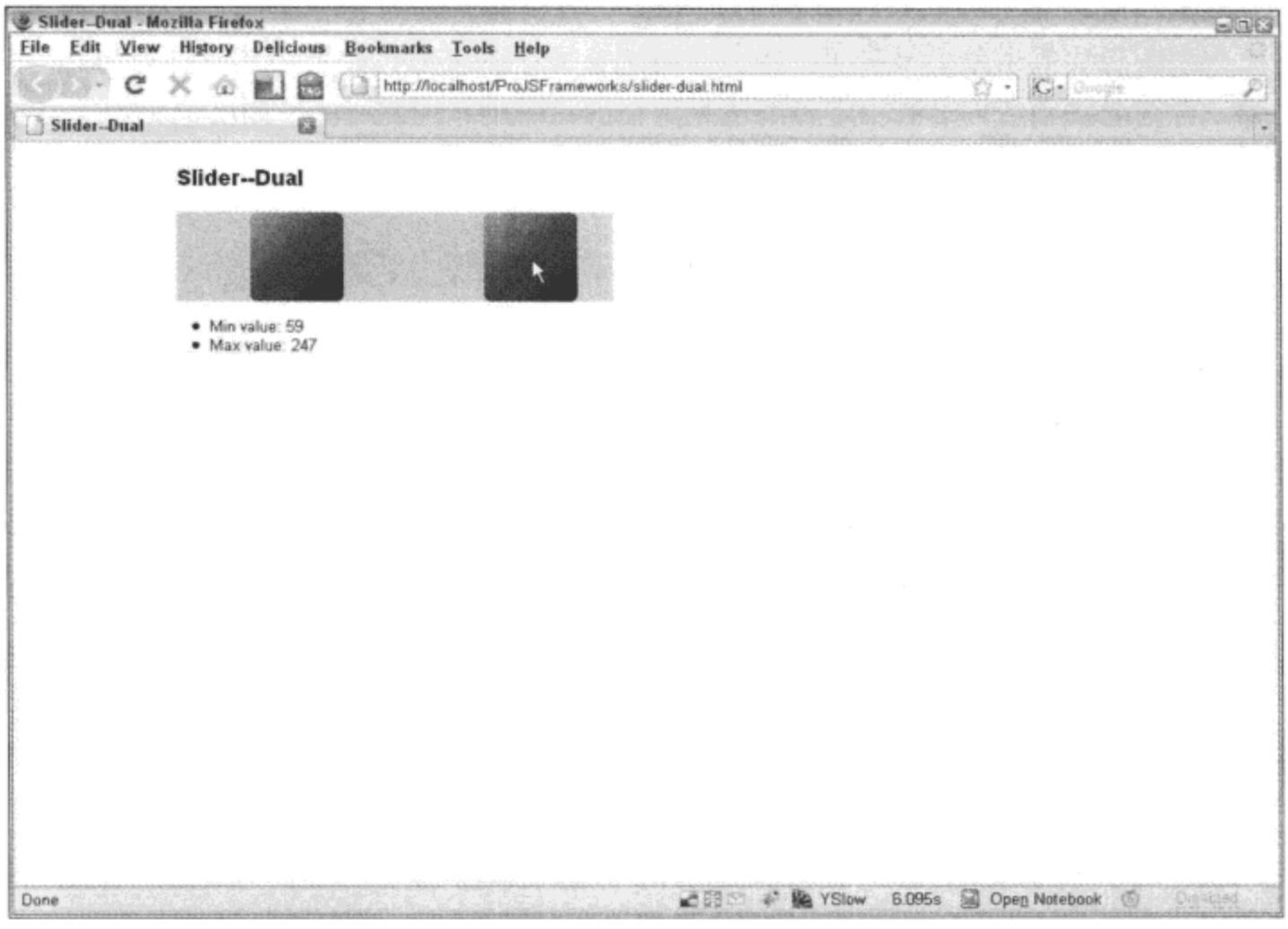


图　12-11

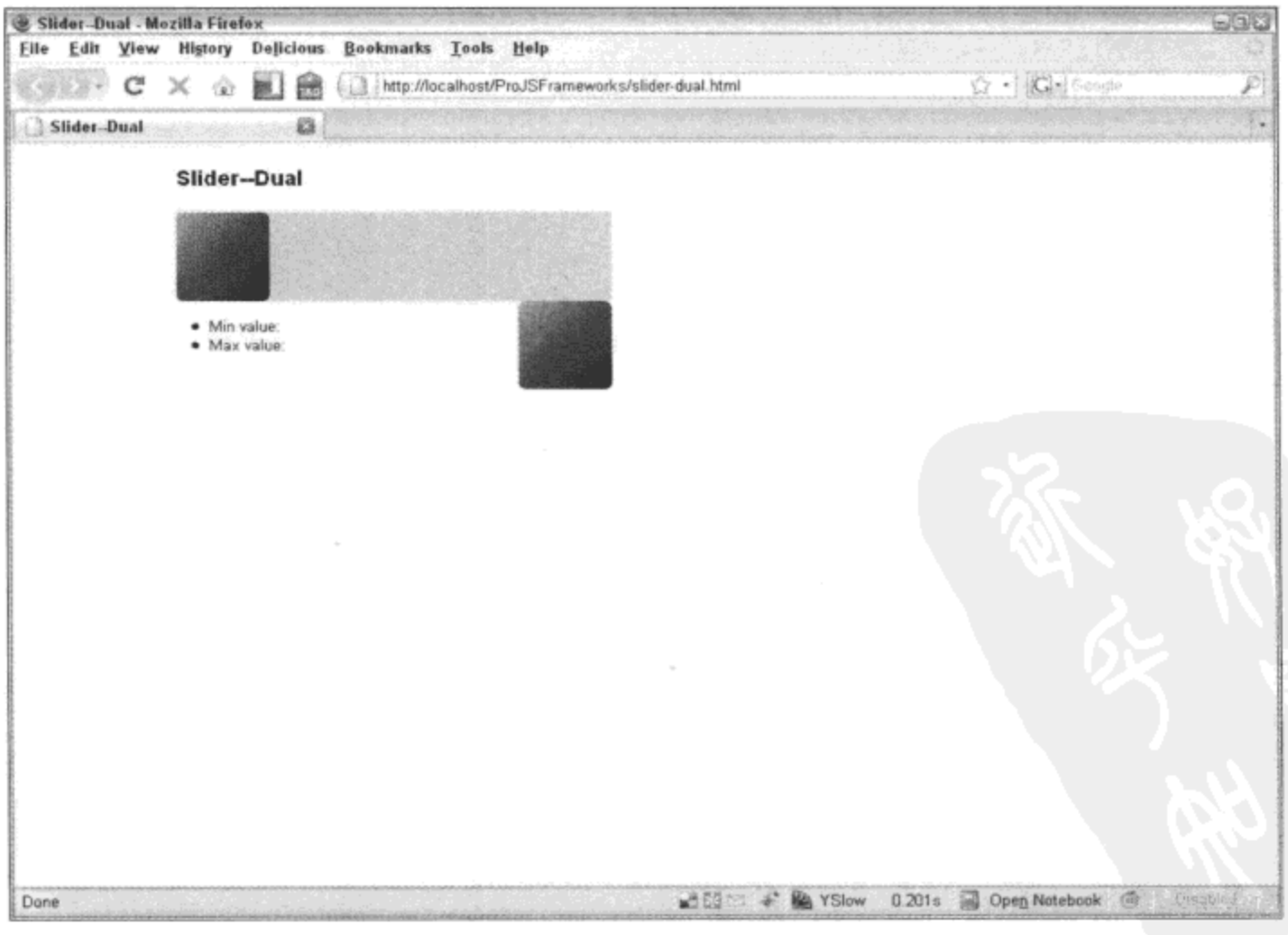


图　12-12

实例化双重滑块时附带 4 个必需的参数和两个可选参数。前 3 个必需的参数均是 ID：分别是滑块背景元素的 ID、最小值滑块元素的 ID 以及最大值滑块元素的 ID。第四个必需的参数是范围值，它用来说明滑块的值最大是多少。第一个可选参数是刻度值。如果将其设置为 0，那么滑块平滑滑动，否则滑块元素将沿着指定刻度值整数倍的点跳动。因此，如果刻度值为 50，那么滑块元素将会跳跃到 50、100、150 等位置。最后，第二个可选参数是一个由两个值构成的数组，它们分别是最小值滑块元素和最大值滑块元素的初始值。

与其他滑块一样，可以手动设置这两个滑块元素的值。但为了容纳两个滑块元素，方法名稍有不同：

```
slider.setMinVal(50); // sets the min thumb's value to 50
slider.setMaxVal(125); // sets the max thumb's value to 125
slider.setValues(50, 125); // does the same thing as the first to lines of code
```

访问双重滑块的滑块元素值并不比访问普通滑块的相应值困难。与其他滑块一样，它们引发相同的事件，并且访问滑块元素值的工作也是在 change 事件处理程序中完成的。主要的不同之处在于，双重滑块的 change 处理程序并不传递一个滑块元素的值，而是传递滑块对象本身的引用。实际上，处理程序中的 this 关键字也指向滑块对象，因此甚至不必将这个引用作为参数传入。滑块对象包含两个有价值的属性，即 minVal 和 maxVal，很明显，它们分别表示最小值滑块元素的值和最大值滑块元素的值。下面的代码清单给出了实际运行的代码，而前面的图 12-11 演示了运行情况。

```
<html>
    <head>
    <title>Slider--Dual</title>
    <link rel="stylesheet" type="text/css" href="reset-fonts-grids.css" />
    <link rel="stylesheet" type="text/css" href="base-min.css" />
    <style type="text/css">
        #sliderbg {
            position: relative;
            background: #ccc;
            width: 350px;
            height: 75px;
        }
        #minthumb,
        #maxthumb {
            position: absolute;
            width: 75px;
        }
    </style>
</head>
<body class="yui-skin-sam">
    <div id="doc">
        <div id="hd">
            <h1>Slider--Dual</h1>
        </div>
        <div id="bd">
            <div id="sliderbg">
```

```
            <div id="minthumb"><img src="sliderthumb.gif" /></div>
            <div id="maxthumb"><img src="sliderthumb.gif" /></div>
        </div>
        <ul>
            <li>Min value: <span id="minVal" /></li>
            <li>Max value: <span id="maxVal" /></li>
        </ul>
    </div>
    <div id="ft">
    </div>
</div>
<script src="yahoo-dom-event.js"></script>
<script src="dragdrop-min.js"></script>
<script src="slider-min.js"></script>
<script>
    (function () {
        var slider = YAHOO.widget.Slider.getHorizDualSlider("sliderbg",
                "minthumb", "maxthumb", 275);

        var minVal = YAHOO.util.Dom.get("minVal");
        var maxVal = YAHOO.util.Dom.get("maxVal");

        slider.subscribe("change", function () {
            minVal.innerHTML = this.minVal;
            maxVal.innerHTML = this.maxVal;
        });
    })();
</script>
</body>
</html>
```

12.1.4　菜单

菜单(尤其是弹出式菜单)是一种常见的用户界面设计，无论是桌面应用程序、操作系统还是网站，到处都可以看到它们的身影。菜单变得流行的原因在于，它们将子导航项和命令隐藏起来，直到用户需要时才显示出来，这让 UI 变得更加整洁。

然而，尽管弹出式菜单非常有用而且按需显示，但是在 Web 浏览器中编写它们却是相当困难的事情。主要的问题在于事件处理，特别是 mouseover 和 mouseout 事件。不同的浏览器在不同的时刻和不同的情况下引发这些事件，而且并不总是在想要的时候引发。以下面的 HTML 代码片段为例(参见图 12-13)：

- Home
- Products
  - Widget
  - Gadget
  - Gizmo

图　12-13

```
<ul>
    <li><a href="/home/">Home</a></li>
    <li>
        <a href="/products/">Products</a>
        <ul>
            <li><a href="/products/widget/">Widget</a></li>
```

```
            <li><a href="/products/gadget/">Gadget</a></li>
            <li><a href="/products/gizmo/">Gizmo</a></li>
        </ul>
    </li>
</ul>
```

将这段标记转换成一个弹出式菜单，从而当鼠标在 Products 项上悬停时就出现产品列表，这一操作实现起来相对容易。但是，检测什么时候关闭这个弹出式菜单却是一件比较棘手的事情。按照常识，关闭弹出式菜单的合适时机应该是当鼠标离开该菜单所占据的子菜单区域时，换言之，“当引发 mouseout 事件时关闭它”。这是一个很好的想法，但有一个问题：当鼠标在这个子菜单的子项上悬停时，它也会触发 mouseout 事件。因此，在子菜单的链接上悬停这样的简单操作也会使菜单关闭。这就是问题所在。为了解决这个问题，需要进行不同的检查来确保子菜单只在适当的时刻关闭。

另一方面，还有一些更为简单的完全采用 CSS 的解决方案，但这些解决方案的缺点在于可配置性较差。一旦实现这些解决方案，通常客户很快就会要求在显示或隐藏菜单和子菜单项时延迟片刻。这种需要的原因不言自明：人类不像计算机那么精确，在导航到子菜单项时偶尔会离开菜单一到两秒。如果没有延迟，那么当鼠标暂时离开菜单时子菜单就会关闭，从而给用户造成不必要的挫败经历。因此，CSS 解决方案已经过时。

YUI 的 Menu 组件系列考虑到这些问题，并提供了相对简单的带有大范围可配置项的实现。有 3 种风格的菜单可用：Menu、ContextMenu 和 MenuBar。

1. Menu

Menu 组件实际上属于弹出式菜单。它与 MenuBar 组件之间的主要区别在于，Menu 垂直呈现(如图 12-14 所示)，而 MenuBar 水平呈现(如图 12-15 所示)。

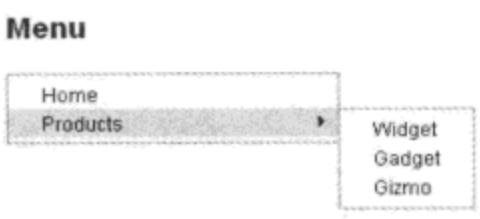


图　12-14

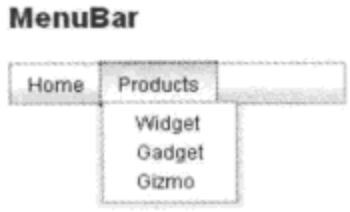


图　12-15

菜单的基本结构源自一个嵌套的无序列表：

```
<ul>
    <li><a href="/home/">Home</a></li>
    <li>
        <a href="/products/">Products</a>
        <ul>
            <li><a href="/products/widget/">Widget</a></li>
            <li><a href="/products/gadget/">Gadget</a></li>
            <li><a href="/products/gizmo/">Gizmo</a></li>
        </ul>
    </li>
</ul>
```

YUI Menu 系列控件扩展 YUI Overlay。这样一来，每种弹出式菜单均继承了 YUI Overlay 内

置的所有优点，例如定位、IE6 的 iframe 保护以及对齐。为了满足 Menu 组件的 Overlay 用法，需要稍微修改上面的示例中列出的标记：

```
<div id="nav">
    <div class="bd">
        <ul>
            <li><a href="/home/">Home</a></li>
            <li>
                <a href="/products/">Products</a>
                <div id="products">
                    <div class="bd">
                        <ul>
                            <li><a href="/products/widget/">Widget</a></li>
                            <li><a href="/products/gadget/">Gadget</a></li>
                            <li><a href="/products/gizmo/">Gizmo</a></li>
                        </ul>
                    </div>
                </div>
            </li>
        </ul>
    </div>
</div>
```

实际上，每个无序列表(ul)都会被两个 div 元素包装起来，然后变成一个 Menu 对象(参见图 12-16)。这是基本的 YUI Overlay 的结构。第一个 div 元素用于定义 Overlay 容器，第二个 div 元素用于定义它的正文(可以从它的 CSS 类名 bd 推断出这一点)。两个 div 元素都有 ID。第一个 ID 是必需的，因为它在 Menu 对象实例化时会传给 Menu 的构造函数，而其他的 ID 是为了便利(这样程序员就可以直接访问菜单，而不需要遍历菜单的层次结构)。如果嵌套 Overlay 的 div 元素中没有 ID，那么 YUI 就生成这些 ID 并分配给它们。

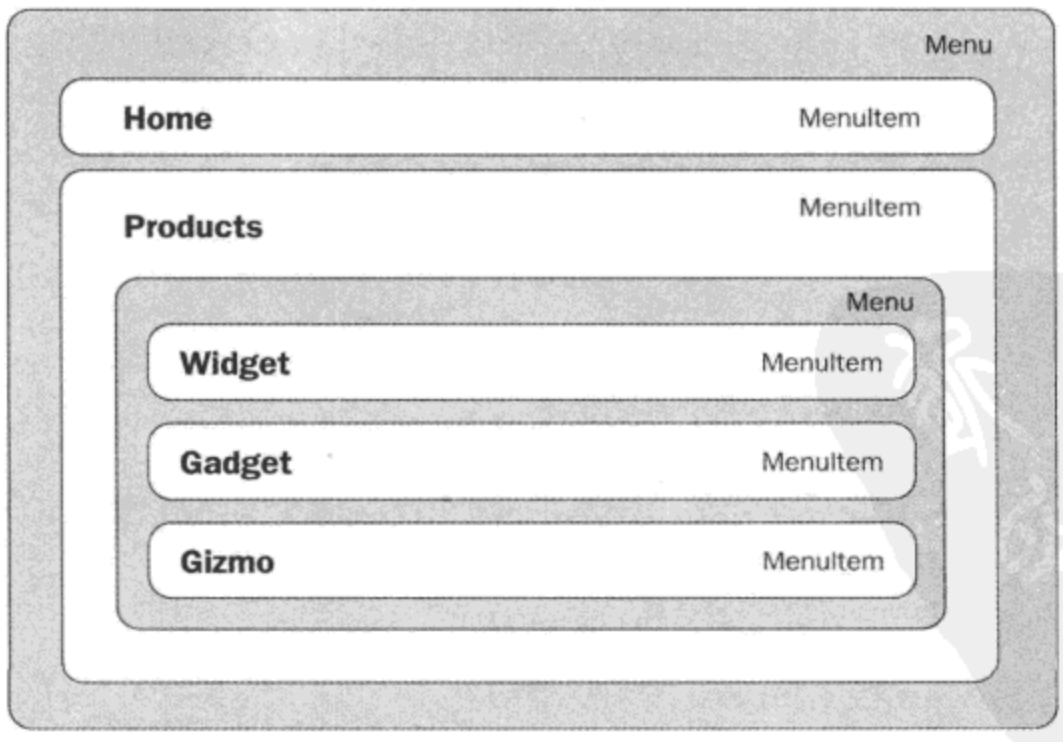


图　12-16

一旦定义了基本的标记，那么创建菜单就非常简单，只需要实例化 Menu 对象即可：

```
var menu = new YAHOO.widget.Menu("nav");
```

还需要完成的最后一件工作是将所有组成部分联系在一起：使用 CSS。YUI Menu 控件系列自带了一个基本的 CSS 皮肤文件(在图 12-14 和图 12-15 中可以看到其效果)。但是，除了让菜单变得漂亮之外，menu.css 文件还用于设置大量重要的定位规则。最后，容器 div 元素(nav)的宽度值规定了容器的宽度。下面就是图 12-14 的完整代码清单：

```
<html>
    <head>
        <title>Menu</title>
        <link rel="stylesheet" type="text/css" href="reset-fonts-grids.css" />
        <link rel="stylesheet" type="text/css" href="base-min.css" />
        <link rel="stylesheet" type="text/css"
              href="menu/assets/skins/sam/menu.css" />
        <style type="text/css">
            #nav {
                width: 200px;
            }
        </style>
    </head>
    <body class="yui-skin-sam">
        <div id="doc">
            <div id="hd">
                <h1>Menu</h1>
            </div>
            <div id="bd">
                <div id="nav">
                    <div class="bd">
                        <ul>
                            <li><a href="/home/">Home</a></li>
                            <li>
                                <a href="/products/">Products</a>
                                <div>
                                    <div class="bd">
                                        <ul>
                                            <li>
                                                <a href="/products/widget/">
                                                    Widget
                                                </a>
                                            </li>
                                            <li>
                                                <a href="/products/gadget/">
                                                    Gadget
                                                </a>
                                            </li>
                                            <li>
                                                <a href="/products/gizmo/">
                                                    Gizmo
                                                </a>
                                            </li>
```

```
                                    </ul>
                                </div>
                            </div>
                        </li>
                    </ul>
                </div>
            </div>
        </div>
        <div id="ft">
        </div>
    </div>
    <script src="yahoo-dom-event.js"></script>
    <script src="container_core-min.js"></script>
    <script src="menu/menu-min.js"></script>
    <script>
        (function () {
            var menu = new YAHOO.widget.Menu("nav");
        })();
    </script>
</body>
</html>
```

使用同样的标记和基本的 CSS 程序呈现 MenuBar 也非常简单，只要修改构造函数即可：

```
var menu = new YAHOO.widget.MenuBar("nav", {
    autosubmenudisplay: true
});
```

请注意新增的属性 autosubmenudisplay。这样当鼠标在 Products 项上悬停时就会弹出子菜单，而不需要等待单击，这是因为 Products 项有一个指向“/products/”的链接，单击它会加载该页面。因此，将 autosubmenudisplay 设置为 true 以确保用户可以访问子菜单。如果顶层 MenuBar 项没有链接，那么使用#就足够了。YUI Menu 检测到#并确保单击该链接时不会尝试加载它，因为它就是一个占位符。

注意：

之所以菜单项为链接而且实际上必须是链接，这是出于可访问性考虑。锚点元素也可以采用选项卡方式显示，从而让整个菜单都可以通过键盘导航。

不使用 HTML 标记

也可以不使用任何 HTML 标记，而只使用 JavaScript 代码呈现与图 12-14 中给出的菜单相同的 Menu 对象。假设页面的正文(ID 为 bd 的 div 元素)为空，下面的 JavaScript 代码将呈现与第一个示例中一样的 Menu 对象。

```
// Instantiate a Menu to hold the product links
var productsMenu = new YAHOO.widget.Menu("products");

// Populate the products Menu with items
productsMenu.addItems([
    {
```

```
            "text": "Widget",
            "url": "/products/widget/"
        },
        {
            "text": "Gadget",
            "url": "/products/gadget/"
        },
        {
            "text": "Gizmo",
            "url": "/products/gizmo/"
        }
]);

// Instantiate the main Menu and set its position to "static"
var menu = new YAHOO.widget.Menu("nav", {
    position: "static"
});

// Populate the main Menu with links and the submenu
menu.addItems([
    {
        "text": "Home",
        "url": "/home/"
    },
    {
        "text": "Products",
        "url": "/products/",
        "submenu": productsMenu
    }
]);

// Render the menu in the main body div
menu.render("bd");

// Show the menu
menu.show();
```

将 Menu 改成 MenuBar 非常简单，只需要修改构造函数即可。

```
var menu = new YAHOO.widget.MenuBar("nav", {
    position: "static",
    autosubmenudisplay: true
});
```

注意，与前面一样，将 autosubmenudisplay 属性添加到配置对象。

事件

表 12-2 列出了 Menu 和 MenuBar 继承的事件，它们来自 YUI Module 和 YUI Overlay(实际上，Menu 是 Overlay 的子类，而 Overlay 是 Module 的子类；MenuBar 是 Menu 的子类)。Menu 有一组特有的事件(MenuBar 继承了这些事件)。

表　12-2

事　　件	说　　明
click	当用户单击菜单时引发。将 DOM Event 对象作为实参传回
itemAdded	当向菜单中添加一项时引发
itemRemoved	当将某个菜单项从菜单中移除时引发
keydown	当菜单的某个菜单项拥有焦点时，如果用户按下键盘按键，就会引发该事件。将 DOM Event 对象作为实参传回
keypress	当菜单的某个菜单项拥有焦点时，如果用户按下字母数字键，就会引发该事件。将 DOM Event 对象作为实参传回
keyup	当菜单的某个菜单项拥有焦点时，如果用户释放键盘按键，就会引发该事件。将 DOM Event 对象作为实参传回
mousedown	当用户在菜单上单击时引发该事件。将 DOM Event 对象作为实参传回
mouseout	当鼠标已经离开菜单时引发该事件。将 DOM Event 对象作为实参传回
mouseover	当鼠标已经进入菜单时引发该事件。将 DOM Event 对象作为实参传回
mouseup	当鼠标位于菜单上方时，如果用户释放鼠标左键，就会引发该事件。将 DOM Event 对象作为实参传回

来源：API 文档

类似地，表 12-3 描述 MenuItem 和 MenuBarItem 共享的自定义事件。

表　12-3

事　　件	说　　明
blur	当菜单项失去输入焦点时引发该事件
click	当用户单击菜单项时引发该事件。将 DOM Event 对象作为实参传回
destroy	当菜单项的 元素从它的 父元素中被移除时引发该事件
focus	当菜单项接收焦点时引发该事件
keydown	当菜单项拥有焦点时，如果用户按下键盘按键，就会引发该事件。将 DOM Event 对象作为实参传回
keypress	当菜单项拥有焦点时，如果用户按下字母数字键，就会引发该事件。将 DOM Event 对象作为实参传回
keyup	当菜单项拥有焦点时，如果用户释放键盘按键，就会引发该事件。将 DOM Event 对象作为实参传回
mousedown	当用户在菜单项上单击时引发该事件。将 DOM Event 对象作为实参传回
mouseout	当鼠标已经离开菜单项时引发该事件。将 DOM Event 对象作为实参传回
mouseover	当鼠标已经进入菜单项时引发该事件。将 DOM Event 对象作为实参传回
mouseup	当鼠标位于菜单项上方时，如果用户释放鼠标左键，就会引发该事件。将 DOM Event 对象作为实参传回

来源：API 文档

下面几个示例演示如何将事件与 Menu 对象及其菜单项连接起来。

```
menu.subscribe("beforeShow", function () {
    alert("beforeShow"); // beforeShow is inherited from YAHOO.widget.Module
```

```
});
menu.subscribe("mouseover", function () {
    alert("This will get annoying really fast.");
});
```

在这里，我们附加了两个事件处理程序，一个绑定到 beforeShow，而另一个绑定到 mouseover。值得注意的是，分配给菜单的事件处理程序也会分配给它的所有子菜单。

遍历菜单层次结构

Menu 和 MenuItem 系列对象提供了一些用于遍历菜单层次结构的方法和属性。前面曾经提及，为了便于操作，Menu 构造函数也会为 Menu 和 MenuItem 元素分配 ID，这样就不再需要手动遍历层次结构。使用 ID 可以更简单地直接定位元素。

要检索 Menu 对象的菜单项，可以使用 getItem 或 getItems 方法，也可以使用配置方法 getProperty。一旦检索到想要的对象，那么访问它在 DOM 中的元素就是一件非常简单的事情，只要引用它的 element 属性即可。还可以使用 parent 属性和 getRoot 方法沿着相反的方法遍历 Menu 对象。下面的代码清单演示如何遍历前面构建的 nav 菜单。

```
// Instantiate a Menu object (based on the root HTML element with the ID "nav")
var menu = new YAHOO.widget.Menu("nav");

// Retrieve the Menu object's element
var menuElement = menu.element;

// Retrieve the Menu object's first item, "Products"
var productsItem = menu.getItem(1);

// Retrieve the Products item's element
var productsItemElement = productsItem.element;

// Retrieve the Products item's label text
var productsItemText = menu.getItem(1).cfg.getProperty("text");

// Retrieve the Products item's submenu object
var productsSubMenu = menu.getItem(1).cfg.getProperty("submenu");

// Retrieve the Products item's submenu object's element
var productsSubMenuElement = menu.getItem(1).cfg.getProperty("submenu").element;

// Retrieve the menu items belonging to the Products submenu (returns an array)
var productsSubMenuItems = menu.getItem(1).cfg.getProperty("submenu").getItems();

// Retrieve the first Products submenu item (widget)
var widget = menu.getItem(1).cfg.getProperty("submenu").getItem(0);

// Retrieve the widget menu item's label text ("Widget")
var widgetElement = widget.element;

// Retrieve the parent menu object of the widget element
var parent = widget.parent;
```

```
// Retrieve the main menu object
var root = productsSubMenu.getRoot();
```

前面曾经提及，菜单中的关键元素会被自动分配ID(如果它们还没有ID的话)，这样访问Products子菜单元素就非常简单，如下所示：

```
var productsSubMenuElement = document.getElementById("products");
```

或者，使用 YUI DOM Collection 也一样简单。

```
var productsSubMenuElement = YAHOO.util.Dom.get("products");
```

2. 上下文菜单

创建上下文菜单(如图 12-17 所示)与创建 Menu 或 MenuBar 一样简单。主要的差别在于，在 ContextMenu 构造函数中需要指定目标。作为目标参数传入的值可以是指向 DOM 中元素的引用、ID 或两者组成的数组。

```
var contextMenu = new YAHOO.widget.ContextMenu("cmenu", {trigger: document});
```

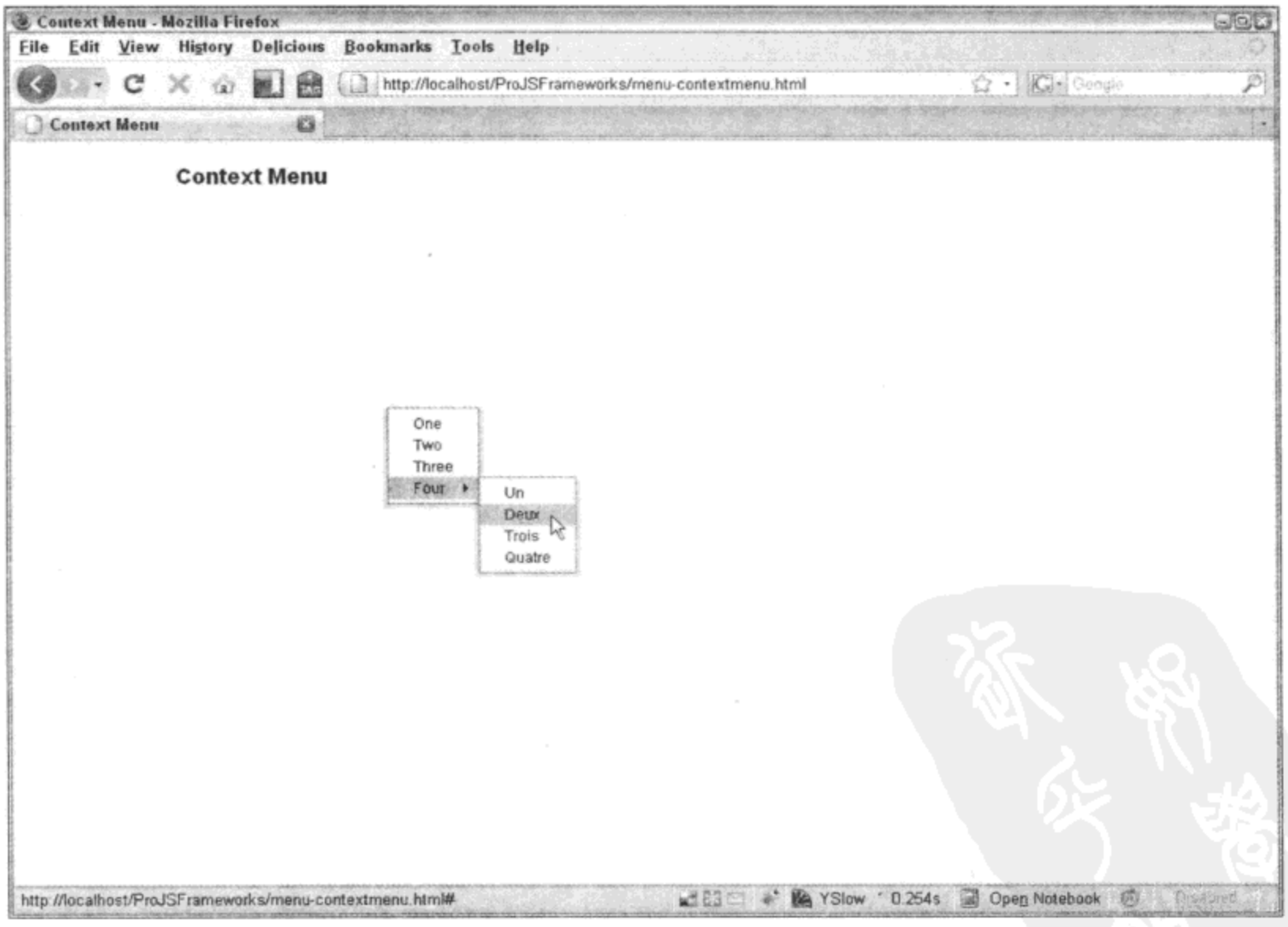


图　12-17

在这里，将触发器设置成 DOM 文档。因此，在文档中的任何地方右击都会使上下文菜单出

现。一旦实例化上下文菜单对象，就可以像前面那样添加菜单项和子菜单项。下面是一个完整的 ContextMenu 示例：

```
<html>
    <head>
        <title>Context Menu</title>
        <link rel="stylesheet" type="text/css" href="reset-fonts-grids.css" />
        <link rel="stylesheet" type="text/css" href="base-min.css" />
        <link rel="stylesheet" type="text/css"
              href="menu/assets/skins/sam/menu.css" />
    </head>
    <body class="yui-skin-sam">
        <div id="doc">
            <div id="hd">
                <h1>Context Menu</h1>
            </div>
            <div id="bd">
            </div>
            <div id="ft">
            </div>
        </div>
        <script src="yahoo-dom-event.js"></script>
        <script src="container_core-min.js"></script>
        <script src="menu/menu-min.js"></script>
        <script>
            (function () {
var contextMenu = new YAHOO.widget.ContextMenu("cmenu",
    {trigger: document});

var sub = new YAHOO.widget.Menu("sub");
sub.addItems(["Un", "Deux", "Trois", "Quatre"]);

contextMenu.addItems(
    [
        "One",
        {
            text: "Two",
            url: "http://www.wrox.com"
        },
        {
            text: "Three",
            onclick: {
                fn:function () {
                    alert("Three!")
                }
            }
        },
        {
            text: "Four",
            submenu: sub
```

```
                    }
                ]
            );
            contextMenu.render(document.body);
        })();
    </script>
</body>
</html>
```

注意 Opera 浏览器目前并不支持触发上下文菜单的右击事件。为了弹出上下文菜单，使用 Opera 浏览器的 Windows 用户需要按住 Ctrl 键，然后单击。使用 Opera 浏览器的 OSX 用户需要按住 Command 键，然后单击。

12.2　提供日期选择功能

可以形象地把填写表单描述为家常事务，而必须根据特定的模式输入信息使得这种体验更加令人讨厌。例如，当在一个旅行网站上输入出发时间时，该日期是否必须遵循 YYYY-MM-DD、DD-MM-YY 或 MM-DD-YYYY 格式？是否将年、月、日使用连字符、空格符或者斜杠隔开？大多数网站会在输入字段的旁边规定必要的格式，而有些网站并没有给出格式。不管怎样，人们并不习惯以这样的格式考虑日期。相反，人们认为 2008-07-01 就是“2008 年 7 月 1 日”。要求人们将其转换成数字只会增加填写表单的挫败感。

下面讲解日期的选择。在日期字段旁边提供一个日历图标，用户可以从一个弹出式日历中选择一个日期，这就使得在表单中填写日期的整个体验变得更加令人愉快。

但是，编写日历可能是一件困难的事情，特别是要采用客户端编程语言(容易出现本地化问题)来计算日期。YUI Calendar 组件彻底解决了这个问题。它是一种完全可定制的解决方案，提供了易于实现、易于使用的日历，并且可以在这些日历的基础上构建更加复杂的应用程序。

12.2.1　简单的日历

建立简单的日历是一件直观的事情。首先需要加载一些依赖项，这些依赖项主要有样本皮肤文件 calendar.css 以及 JavaScript 文件 yahoo-dom-event.js 和 calendar-min.js。加载这些文件之后，创建日历对象就是编写两行代码的事情：

```
var cal = new YAHOO.widget.Calendar("calendar");
cal.render();
```

在这里，我们正在实例化一个新的 Calendar 对象。我们在它的构造函数中传入 calendar，这是一个元素 ID，日历对象将呈现在这个元素中。但是，实例化 Calendar 对象并不会自动让其呈现出来。这样一来，在对象实例化和其呈现之间就可以对它进行配置。render 方法存在的另一个理由是，我们可以在后面通过日历对象的配置对象调整它的外观并再次调用它的 render 方法。

在上面的代码中，使用日历对象的默认配置来呈现它。图 12-18 给出了 Calendar 对象的默认呈现效果。当天的日期高亮显示并带有一个黑色方框，选中的日期具有浅蓝色背景，而鼠标悬停在其上的日期则具有深蓝色背景。为了应用 YUI 提供的样本皮肤，需要将 CSS 类名 yui-skin-sam

分配给一个元素，这个元素在 DOM 中处于较高的层次，它属于在其中呈现日历的元素的祖先元素。YUI 团队通常建议将这个 CSS 类名放到 body 标记上。

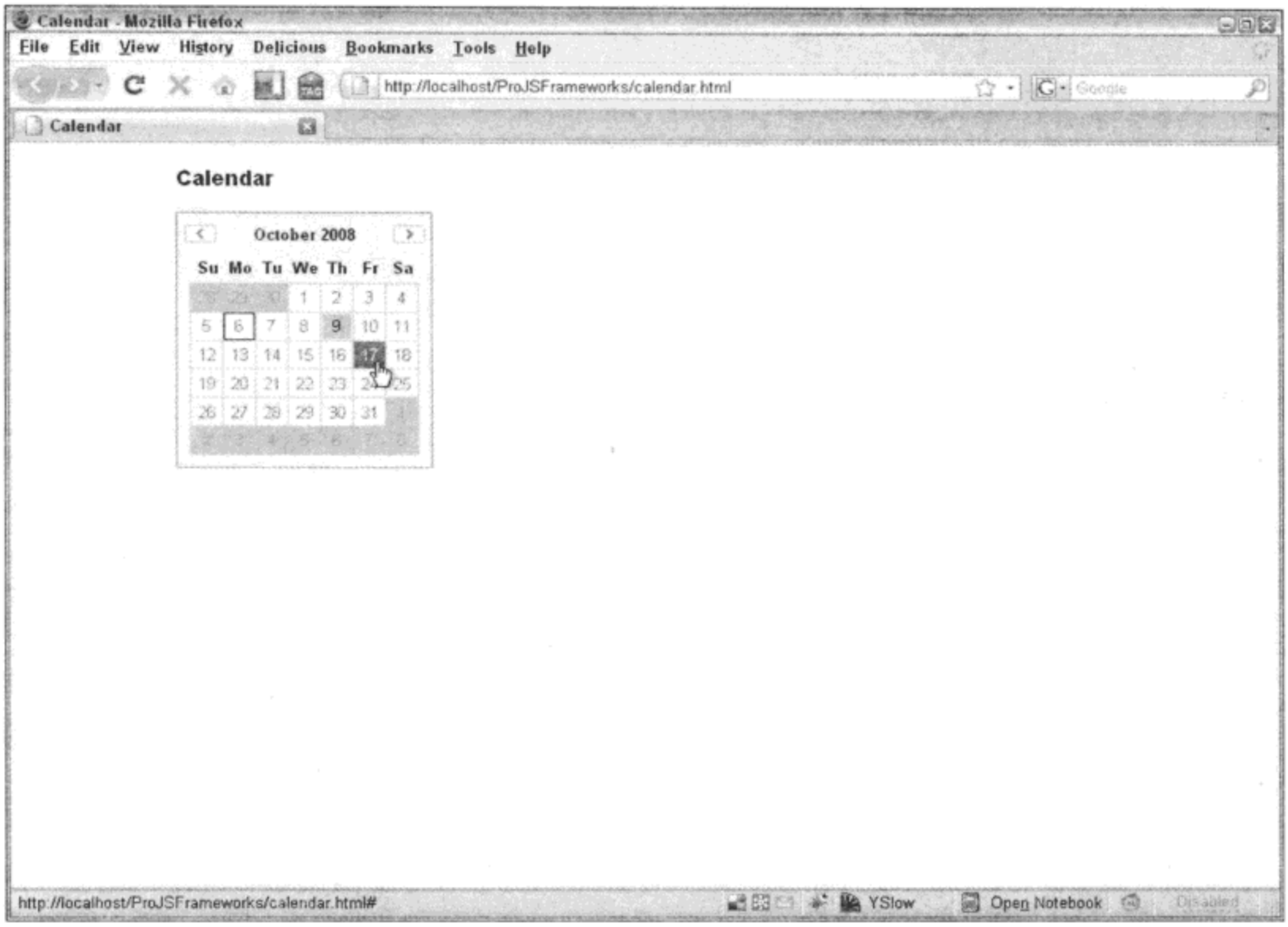


图　12-18

设置 Calendar 对象的配置属性有 3 种方式：通过它的构造函数、queueProperty 方法或 setProperty 方法。下面给出了这 3 种技术的示例：

```
// Constructor
var cal = new YAHOO.widget.Calendar("calendar", {title: "Pick a date"});
cal.render();

// Using queueProperty
var cal = new YAHOO.widget.Calendar("calendar");
cal.cfg.queueProperty("title", "Pick a date");
cal.cfg.fireQueue();
cal.render();

// Using setProperty
var cal = new YAHOO.widget.Calendar("calendar");
cal.cfg.setProperty("title", "Pick a date");
cal.render();
```

每种技术都有其存在的理由。第一种技术是在实例化阶段完成所有设置的简单方式。

queueProperty 方法使得我们可以分配多个值，而且一次性将其设置完毕(例如在一个循环中)。最后，setProperty 方法只允许立即设置一个值。注意在第二个示例中，调用 fireQueue 方法并不是必需的操作，这是因为 render 方法会自动应用所有已经排队的属性，但在这个示例中调用 fireQueue 是为了清晰起见。此外还要注意，大多数用于设置或修改日历的视觉方面的配置属性都将需要调用 render 方法，目的是为了让改动在屏幕上可见(参见图 12-18)。

表 12-4 列出了所有可用的配置属性，其中有一列指定在调整该属性时是否需要调用 render 方法。

表 12-4

名　称	类　型	默认值	说　明	是否需要调用 render 方法
pagedate	String/Date	当前月份	设置日历的可见月份和年份。如果使用字符串设置，那么默认的字符串格式为"mm/yyyy"	是
selected	String	null	设置日历的已选中日期。内置的默认日期格式为 MM/DD/YYYY。日期范围的定义格式为 MM/DD/YYYY - MM/DD/YYYY。月/日组合的定义格式为 MM/DD。通过使用逗号分隔字符串可以将这些格式组合起来(例如"12/24/2005,12/25/2005,1/18/2006 -1/21/2006")	是
mindate	String/Date	null	设置日历的最小可选择日期，既可以采用 JavaScript Date 对象的形式，也可以采用字符串日期(例如"4/12/2007")	是
maxdate	String/Date	null	设置日历的最大可选择日期，既可以采用 JavaScript Date 对象的形式，也可以采用字符串日期(例如"4/12/2007")	是
Title	String	null	设置显示在容器顶部的日历标题	否
Close	Boolean	false	如果设置为 true，就会显示一个可用来消除日历的关闭图标	否
iframe	Boolean	true	在日历底部放置一个 iframe 垫片以防止 select 元素“渗出”	否
multi_select	Boolean	false	确定日历是否允许选择多个日期	否
navigator	Boolean/Object	null	为日历配置 CalendarNavigator(年份选择器)功能。如果设置为 true，就启用日历的年份选择器功能。可以通过将该属性设置为一个对象字面值(其定义位于 Navigator Configuration Object 文档中)来自定义 CalendarNavigator 的配置	是
show_weekdays	Boolean	true	确定是否显示工作日首部	是

(续表)

名　称	类　型	默认值	说　明	是否需要调用 render 方法
locale_months	Array	"long"	要显示的月份标题的格式。可选的值有"short"、"medium"和"long"	是
locale_weekdays	Array	"short"	要显示的工作日标题的格式。可选的值有"1char"、"short"、"medium"和"long"	是
start_weekday	Integer	0	0~6，表示一周从哪一天开始	是
show_week_header	Boolean	false	确定是否显示行首	是
how_week_footer	Boolean	false	确定是否显示行尾	是
hide_blank_weeks	Boolean	false	确定是否隐藏完全超出当前月份的额外周	是

来源：YUI Calendar 页面

12.2.2 事件

通过绑定事件可以在关键时刻与 Calendar 对象交互。可以通过自定义事件对象的 subscribe 方法来实现该操作，如下所示：

```
cal.selectEvent.subscribe(function (eventType, args) {
});
```

自定义事件返回的实参遵循 YUI Custom Event 模式，第一个属性是表示刚刚发生的事件类型的字符串，第二个属性是传给处理程序的实参的集合。

对于 select 事件(当选中一个日期时引发该事件)，第二个实参包含一个由日期构成的数组(这是因为 Calendar 也支持选择多个日期)。下面的示例(参见图 12-19)演示如何访问传入事件处理程序的第一个日期的年、月、日值：

```
cal.selectEvent.subscribe(function (eventType, args) {
    var datesArray = args[0],
        firstDate = datesArray[0],
        year = firstDate[0],
        month = firstDate[1],
        day = firstDate[2];
    alert(
        "Event Type: " + eventType + "\n" +
        "Year: " + year + "\n" +
        "Month: " + month + "\n" +
        "Day: " + day);
});
```

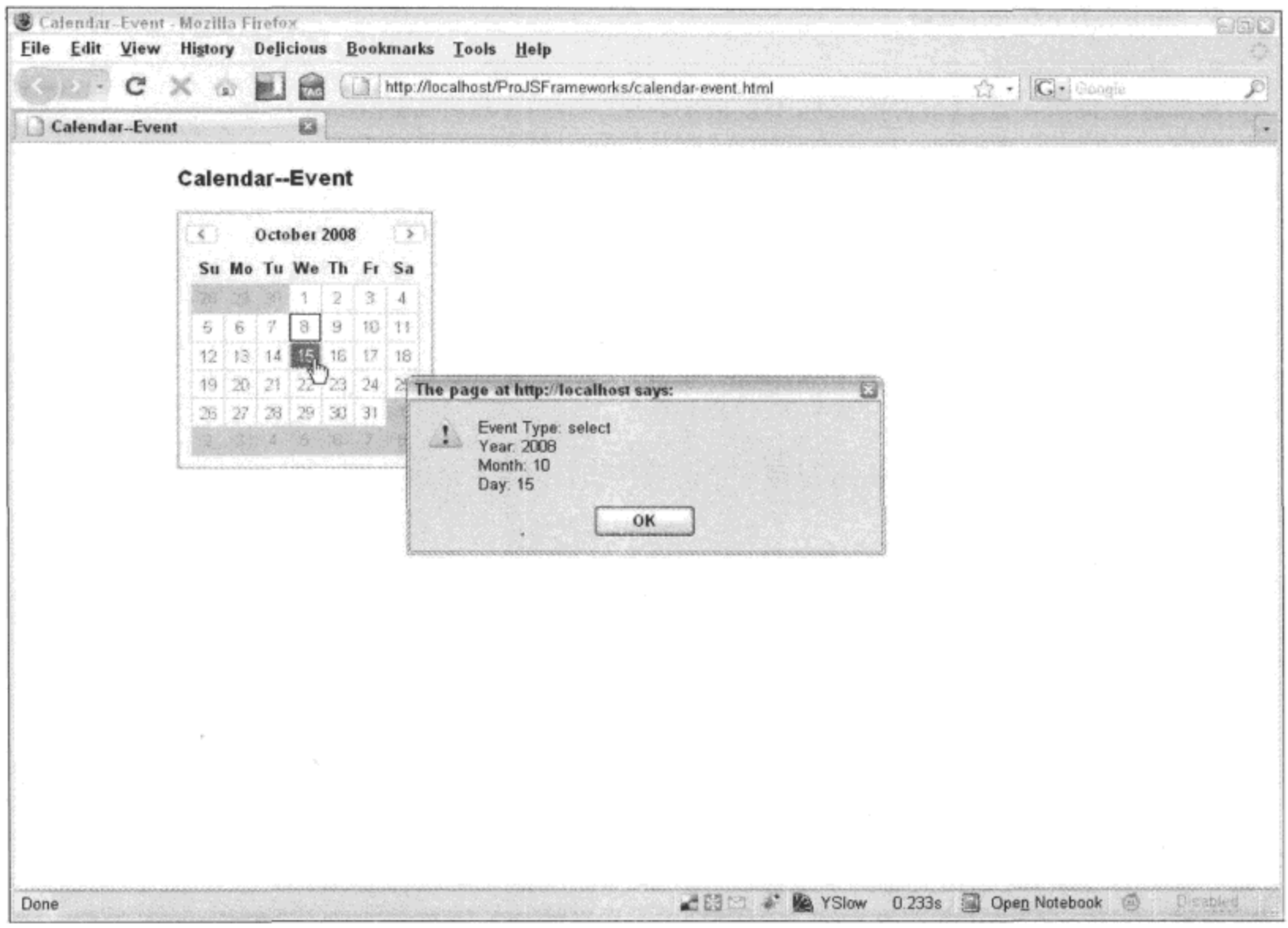


图　12-19

另一个传递实参给事件处理程序的自定义事件是 deselectEvent，正如它的名称所示，当取消选择一个日期时就会引发该事件。以下是 Calendar 的自定义事件的完整列表：

- selectEvent
- beforeSelectEvent
- deselectEvent
- beforeDeselectEvent
- renderEvent
- beforeRenderEvent
- changePageEvent
- clearEvent
- resetEvent
- beforeHideEvent
- hideEvent
- beforeShowEvent
- showEvent
- beforeShowNavEvent
- showNavEvent

- beforeHideNavEvent
- hideNavEvent

前面曾经提及，这些事件的名称不言自明。

12.2.3 多页日历

有时，有必要向用户展现一个含有多个月份(或多页)的日历。这种日历实现起来非常容易，只要调用 CalendarGroup 构造函数(而不是 Calendar 构造函数)即可。

```
var cal = new YAHOO.widget.CalendarGroup("calendar");
cal.render();
```

在默认情况下，CalendarGroup 呈现一个两页日历。但是，可以指定页数，只要通过构造函数的配置对象传入该值即可。

```
var cal1 = new YAHOO.widget.CalendarGroup("calendar1");
cal1.render();
var cal2 = new YAHOO.widget.CalendarGroup("calendar2", {pages: 3});
cal2.render();
```

在这里，在同一个页面上呈现了两个日历组。第一个日历组是一个标准的两页日历，而第二个日历组是一个三页日历(参见图 12-20)。

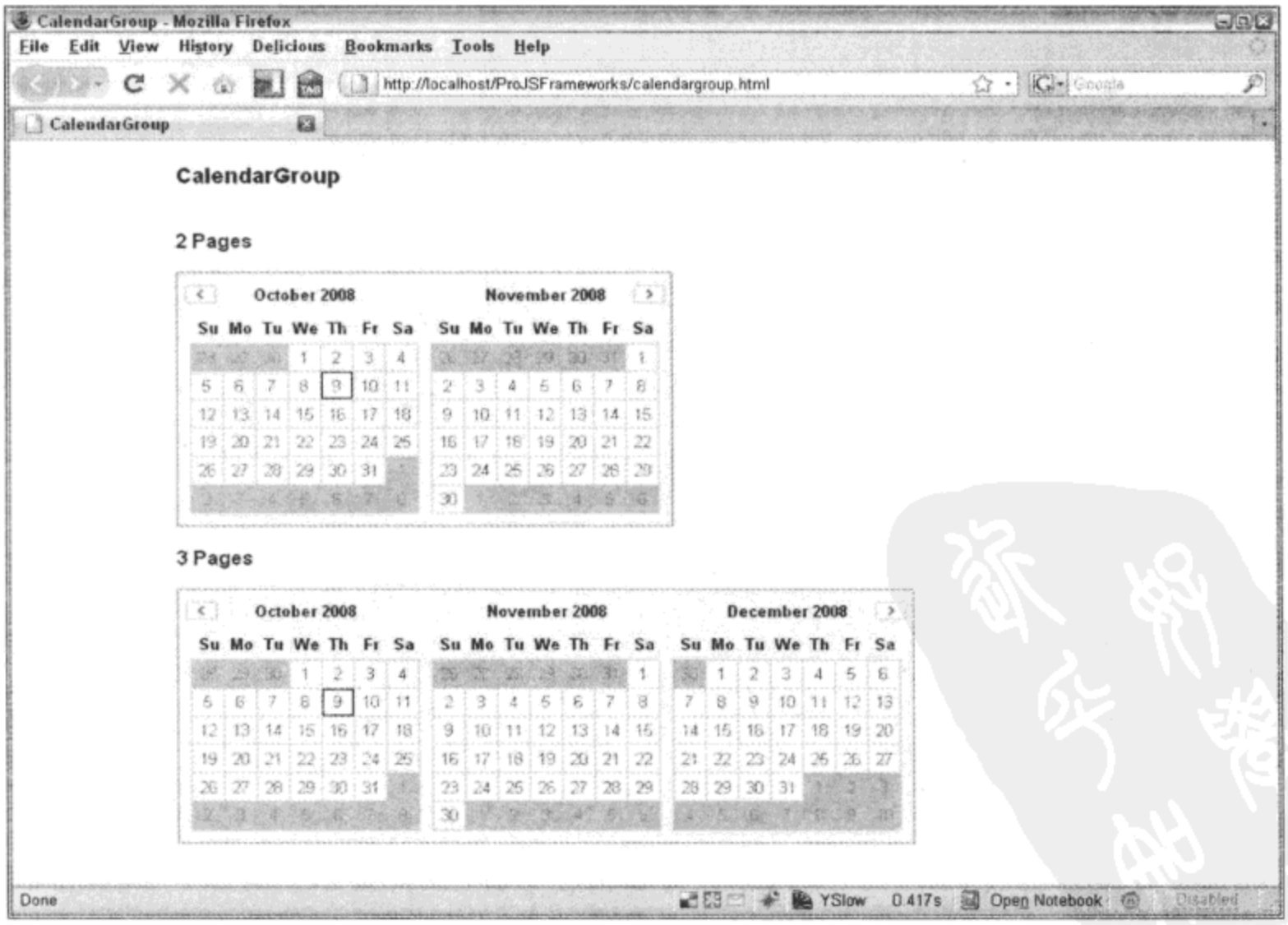


图 12-20

12.3　启用富内容编辑

textarea 表单字段只允许向其中输入基本的文本内容，它并不能进行任何格式化。然而，有些应用程序可能需要的并不只是简单的 textarea，而有些应用程序则需要富文本编辑器。但是，现代浏览器并不提供富文本编辑器。然而，有一种方式是让文档充当编辑器，可以通过将文档的 designMode 属性设为 on 来实现该功能。因此，这里的技巧是将在 iframe 中加载的文档的 designMode 属性设为 on，这样一来只有该文档是可编辑的，而不是整个宿主页面可编辑

当然，在这里就算是设置属性值这样简单的事情也比较麻烦。Internet Explorer 有一个知名的程序错误，就是必须将这个值设为 true 而不是 on，而且这一点到现在都没有任何改观。每一款现代浏览器(从 Firefox 到 Opera，再到 Adobe AIR)都存在某种古怪行为，从而让我们没有方法运用这个技巧来实现一个简单的、跨浏览器的解决方案。然而，YUI Rich Text Editor 实现了这个功能。它针对这些浏览器问题，通过修正、绕开问题和规范化为 Rich Text Editor 提供了一个简单的、易于使用的 API。最终的结果是一个易于实现、具有所有功能、没有任何副作用的稳定窗口部件。

首先，YUI Editor 有两种风格：SimpleEditor 和 Editor。后者扩展前者，提供了更多的功能，但必须加载更多的依赖项。

SimpleEditor 依赖的组件有：YAHOO、DOM、Event、Element 以及可选的 Container_Core、Animation、DragDrop 和 Resize。

Editor 依赖的组件有：YAHOO、DOM、Event、Element、Container_Core、Menu、Button 以及可选的 Animation、DragDrop 和 Resize。

虽然可以为 Editor 设置皮肤，但是本章的示例使用默认的 YUI 样本皮肤。这里使用的默认 CSS 在功能和视觉上已经足够。

一旦加载 CSS 和 JavaScript 文件，那么建立 Editor 就是相当简单的事情。首先，需要将 textarea 元素作为基础来提供呈现 Editor 的位置，同时将该元素作为预先加载到 Editor 中的富文本内容的来源。

```
<textarea name="message" id="message" cols="60" rows="20">
    <strong>Testing</strong>, <em>testing</em>, 1, 2, 3.
</textarea>
```

注意，虽然上一个示例中的 textarea 包含 HTML 内容，但是 textarea 并不能原生地呈现它。使用 textarea 作为基础元素，YUI Rich Text Editor 就变成了一种渐进增强，因为它通过 JavaScript 来增强标准的、易于访问的 HTML 元素。因此，textarea 的内容仍然可提供给没有启用 JavaScript 或没有 JavaScript 功能的用户。

一旦标记准备就绪，就可以实例化 Editor 对象，只要将 textarea 的 ID 传给 Editor 或 SimpleEditor 的构造函数即可。

```
// Render a SimpleEditor
var simpleEditor = new YAHOO.widget.SimpleEditor("message");
simpleEditor.render();

// Or render a full Editor
var editor = new YAHOO.widget.Editor("message");
editor.render();
```

虽然这些示例都呈现全功能的编辑器，但它们的外观不能调整。对于初学者而言，它们的高度和宽度值将由 textarea 的尺寸决定。可以通过将配置对象作为构造函数的第二个参数传递来实现高度和宽度值以及其他项的设置。

```
var editor = new YAHOO.widget.SimpleEditor("message", {
    height: "250px",
    width: "650px",
    dompath: true,
    autoHeight: true
});
editor.render();
```

在这里，我们设置编辑器的高度为 250 像素，宽度为 650 像素(参见图 12-21 和图 12-22)。我们还设置编辑器，在其底部显示 DOM 路径栏，DOM 路径栏的作用是显示光标在 DOM 中的当前位置。因此，如果光标位于单词“Testing”(它在 strong 标记中)上方，那么 DOM 路径栏将显示 body <strong。最后，设置 Editor，将它的高度调整到与其中的内容一样高，而不是显示一个滚动栏。

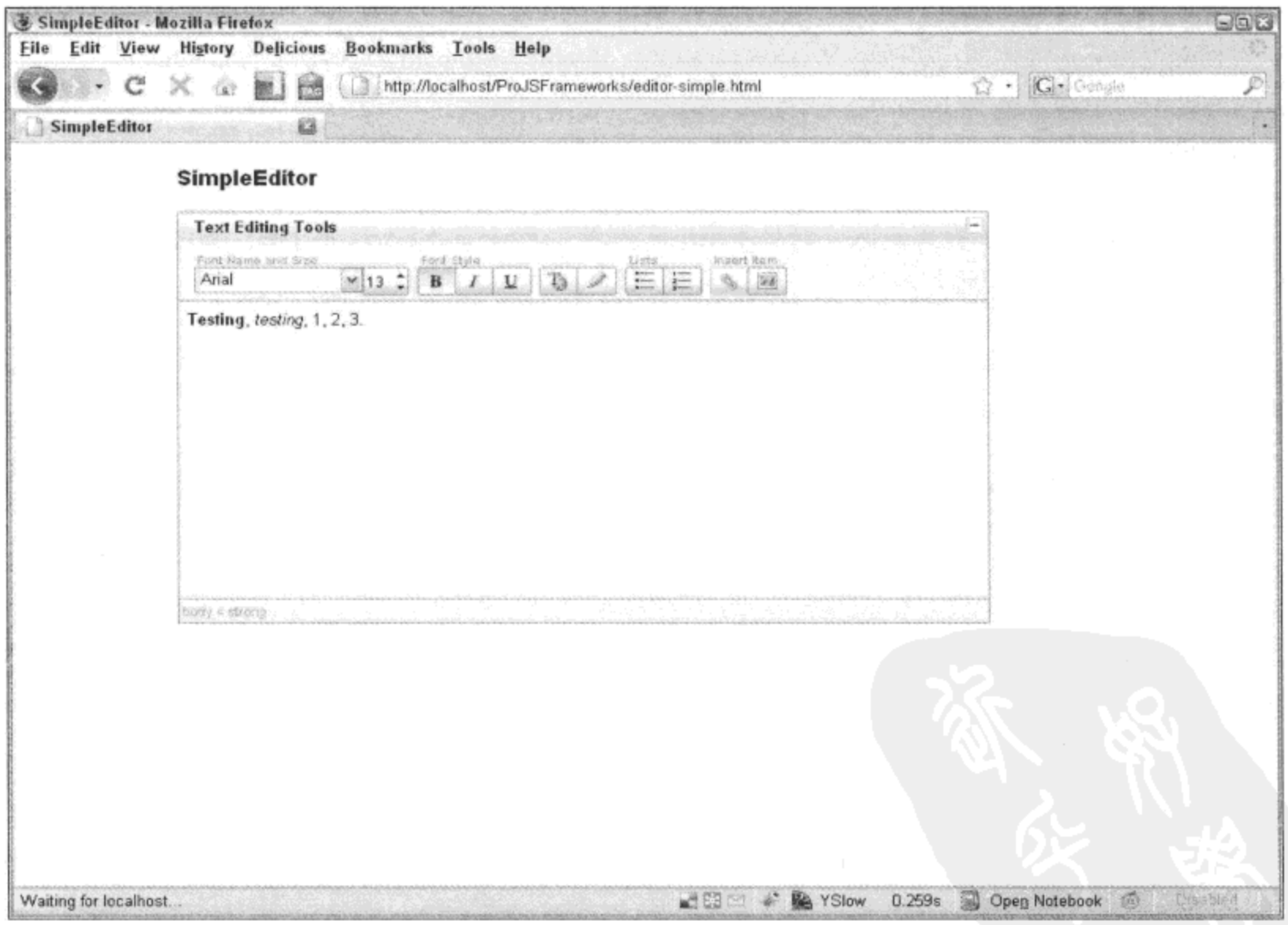


图　12-21

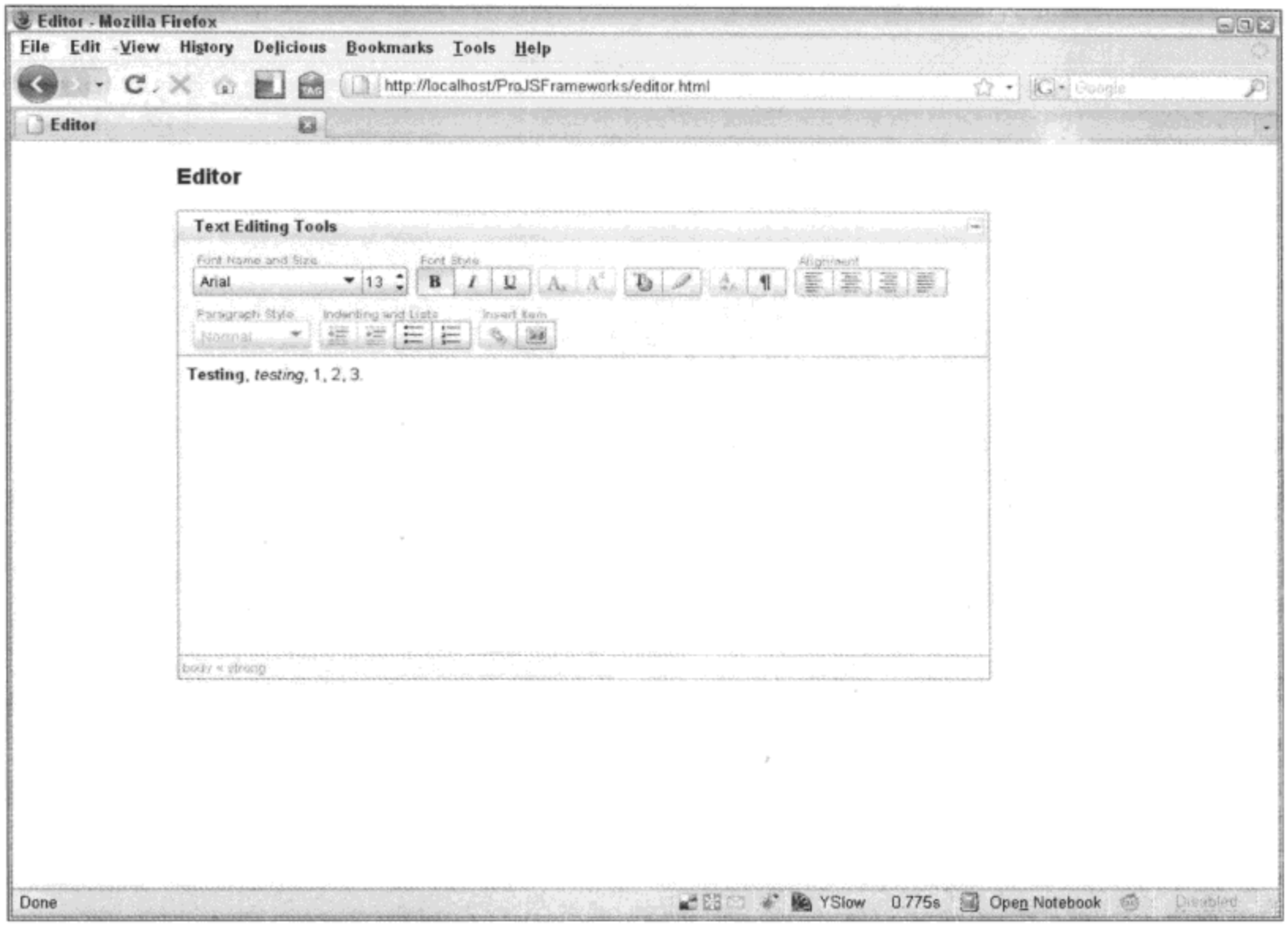


图　12-22

表 12-5 列出了可以在 Editor 或 SimpleEditor 对象上设置的所有不同的可配置属性。

表　12-5

属　性	说　明	默 认 值
allowNoEdit-Boolean	编辑器检查非编辑字段。值得注意的是，这个技术并不完善。如果用户做法正确的话，那么仍然能够进行修改，例如将内容下面和上面的元素高亮显示，单击工具栏按钮或快捷键	默认值：false
animate-Boolean	Editor 是否应该对窗口移动制作动画	Editor 应该对窗口移动制作动画
autoHeight-Boolean ||Number	将滚动栏从编辑区域移除并调整该区域大小，使其适应内容尺寸	默认值：false
blankimage-String	图像占位符(用在插入图像时)的 URL	默认值：当前版本的 yahooapis.com 地址+'assets/blankimage.png'

（续表）

属　性	说　明	默 认 值
css-String	用来格式化 Editor 内容的基本 CSS	默认值：

```
html {
    height: 95%;
}
body {
    height: 100%;
    padding: 7px;
    background - color: #fff;
    font:13px/1.22 arial,helvetica,clean,sans - serif;
    *font - size:small;
    *font:x - small;
}
a {
    color: blue;
text - decoration: underline;
cursor: pointer;
}
    .warning - localfile {
border - bottom: 1px dashed red !important;
}
.yui - busy {
    cursor: wait !important;
}
img.selected { //Safari image selection
    border: 2px dotted #808080;
}
img {
    cursor: pointer !important;
    border: none;
}
```

disabled-Boolean	是否禁用 Editor。当禁用 Editor 时，designMode 被关闭，并在 iframe 上方放置一个遮罩，这样就不会出现交互。同样，所有的工具栏按钮也被禁用，这样就不能使用它们	默认值：false
dompath-Boolean	是否在编辑器下方显示当前 DOM 路径	默认值：false
element_cont-HTMLElement	Editor 容器的内部配置	默认值：false
extracss-String	在加载默认的 SimpleEditor CSS 文件之后还要加载的额外的用户定义 CSS	默认值："
focusAtStart-Boolean	当内容准备就绪时 Editor 是否应该让窗口获得焦点	默认值：false
handleSubmit-Boolean	配置处理是否将 Editor 附加到 textarea 父表单的 submit 事件处理程序。如果将其设置为 true，Editor 将试图把一个 submit 事件监听程序附加到 textarea 父表单。然后，它会触发 Editor 的 save 事件处理程序，并在提交表单前将新内容放回到文本区域	默认值：false

(续表)

属　性	说　明	默 认 值
height-String	Editor iframe 容器的高度，不包括工具栏	默认值：textarea 的最佳猜测大小。为了获得最佳效果，最好使用 CSS 指定 textarea 的高度或将其作为一个实参传入

html-String	在加载内容之前写入到 iframe 文档的默认 HTML 代码(注意，会在呈现项上添加 DOCTYPE attr)	默认值：如果打算重写这段 HTML，那么要注意它的一些条件：需要有{TITLE}、{CSS}、{HIDDEN_CSS}、{EXTRA_CSS}和{CONTENT}。它们将传给 YAHOO.lang.substitute，由其他字符串替换。
		onload="document.body._rteLoaded= true;"：必须有这条 onload 语句，否则 Editor 将不会完成加载
		`<html>`
		`<head>`
		`<title> {TITLE} </title>`
		`<meta http - equiv="Content - Type"content= "text/html; charset=UTF - 8"/>`
		`<style>`
		`{CSS}`
		`</style>`
		`<style>`
		`{HIDDEN_CSS}`
		`</style>`
		`<style>`
		`{EXTRA_CSS}`
		`</style>`
		`</head>`
		`<body onload= "document.body._rteLoaded = true; ">`
		`{CONTENT}`
		`</body>`
		`</html>`

limitCommands-Boolean	Editor 是否应该把 execCommands 限制到工具栏中可用的命令：如果为 true，那么未在工具栏中定义的 execCommand 和键盘快捷方式将失效	默认值：false
markup-String	Editor 是否应该试图为以下类型调整标记：semantic、CSS、default 或 xhtml	默认值："semantic"
nodeChangeThreshold -Number	在 nodeChange 处理之间需要等待的秒数	默认值：3
panel-Boolean	指向用于窗口的面板的引用	默认值：false

(续表)

属　性	说　明	默 认 值
plaintext-Boolean	将初始的 textarea 数据当做普通文本处理。考虑空格、制表符和换行符	默认值：false
removeLineBreaks -Boolean	编辑器在清理文本时是否要把换行符和额外的空格移除	默认值：false
toolbar-Object	默认的工具栏配置	
toolbar_cont-Boolean	工具栏容器的内部配置	默认值：false
width-String	Editor 容器的宽度	默认值：textarea 的最佳猜测大小。为了获得最佳效果，最好使用 CSS 指定 textarea 的宽度或将其作为一个实参传入

来源：API 文档

12.3.1 事件

可以通过 Editor 在关键时刻触发的事件来与 Editor 交互。Rich Text Editor 使用较新的自定义事件处理的 Event Provider 模型，它将类似 obj.customEventName.subscribe(callback)这样的语句转换成类似 obj.on("customEventName", callback)这样的语句，从而简化了语法。因此，为了检测是否加载 Editor 的工具栏，可以使用下面的代码：

```
editor.on("toolbarLoaded", function () {
    alert("Toolbar is loaded!");
});
```

类似地，可以通过下面的 toolbar 对象来访问工具栏事件：

```
editor.on("toolbarLoaded", function () {
    this.toolbar.on("buttonClick", function () {
        alert("Click!");
    });
});
```

注意如何在 toolbarLoaded 回调内部完成 toolbar 事件的分配。这是因为 toolbar 的初始化需要一定的时间，而在它准备就绪之前试图为其分配一个事件处理程序将导致抛出一个错误。此外还要注意，这里通过 this 关键字来访问 toolbar 对象。在这里，this 关键字指向 Editor 对象自身。

表 12-6 列出了更多常见的 Editor 事件。在 YUI 的压缩文件中包括的 API 文档内可以找到完整的列表(包括传给每个事件的实参)。

表　12-6

SimpleEditor 事件	Editor 事件
编辑器呈现事件： • toolbarLoaded • afterRender • editorContentLoaded 编辑器 HTML 事件映射： • editorMouseUp • editorMouseDown • editorDoubleClick • editorKeyUp • editorKeyPress • editorKeyDown 编辑器命令执行事件： • beforeNodeChange • afterNodeChange • beforeExecCommand • afterExecCommand 工具栏事件(通过 EditorObj.toolbar.on()访问)： • toolbarExpanded • toolbarCollapsed • colorPickerClicked • cmdClick(动态事件) • menucmdClick(动态事件) • buttonClick	编辑器呈现事件： • toolbarLoaded • afterRender • editorContentLoaded 编辑器 HTML 事件映射： • beforeEditorMouseUp • editorMouseUp • beforeEditorMouseDown • editorMouseDown • beforeEditorClick • editorClick • beforeEditorDoubleClick • editorDoubleClick • beforeEditorKeyUp • editorKeyUp • beforeEditorKeyPress • editorKeyPress • beforeEditorKeyDown • editorKeyDown 编辑器命令执行事件： • beforeNodeChange • afterNodeChange • beforeExecCommand • afterExecCommand 编辑器窗口事件： • beforeOpenWindow • afterOpenWindow • closeWindow • windowCMDOpen(动态事件) • windowCMDClose(动态事件) • windowRender • windowInsertImageRender • windowCreateLinkRender 工具栏事件(通过 EditorObj.toolbar.on()访问)： • toolbarExpanded • toolbarCollapsed • colorPickerClicked • cmdClick(动态事件) • menucmdClick(动态事件) • buttonClick

12.3.2 实际使用编辑器

一旦建立好 Editor 或 SimpleEditor，下面就要设法访问用户输入到其中的文本。从 Editor 中获取数据有几种不同的方式。一种方式是将配置属性 handleSubmit 设为 true。这会让 Editor 尝试将自己绑定到它的父表单，当单击提交按钮时将它的内容传送给 textarea。还可以通过首先调用 Editor 的 saveHTML 方法(用于将它的内容传送给 textarea)来手动访问它的内容，然后只需要引用 textarea 的 value 属性即可。

YUI 3 中的新增功能

本章前面曾经提及，YUI 3 中的窗口部件继承自 Widget 基类，该类将所有窗口部件标准化，使它们均具有 render、init 和 destroy 方法。Widget 类使用“抽象呈现方法，从而在窗口部件中支持一致的 MVC 结构，并支持一组共同的窗口部件基本属性、一致的类名生成支持以及插件支持”。

来源：YUI 3 Widget 页面。

在本书即将付印之际，YUI 3 中唯一完全重写的窗口部件是 Slider。下面这个示例取自 YUI 3 网站，它通过呈现两个滑块(一个水平滑块和一个垂直滑块)来演示新滑块的运行情况。

```
// Create a YUI instance and request the slider module and
// its dependencies
YUI({base:"../../build/", timeout: 10000}).use("slider",
       function (Y) {

// store the node to display the vertical Slider's current
// value
var v_report = Y.get('#vert_value'),
   vert_slider;

// instantiate the vertical Slider.  Use the classic thumb
// provided with the Sam skin
vert_slider = new Y.Slider({
   axis: 'y', // vertical Slider
   value: 30, // initial value
   railSize: '10em', // range the thumb can move through
   thumbImage: Y.config.base+
         '/slider/assets/skins/sam/thumb-classic-y.png'
});

// callback function to display Slider's current value
function reportValue(e) {
   v_report.set('innerHTML', 'Value: ' + e.newVal);
}

vert_slider.after('valueChange', reportValue);

// render the slider into the first element with
// class vert_slider
vert_slider.render('.vert_slider');
```

```
// instantiate the horizontal Slider, render it, and
// subscribe to its valueChange event via method chaining.
// No need to store the created Slider in this case.
new Y.Slider({
        railSize: '200px',
        thumbImage: Y.config.base+
                '/slider/assets/skins/sam/thumb-classic-x.png'
    }).
    render('.horiz_slider').
    after('valueChange',function (e) {
        Y.get('#horiz_value').set('innerHTML', 'Value: ' +
                e.newVal);
    });

});
```

有几点需要注意：YUI 3 通过 use 方法来加载它的依赖项，此外它还大量使用了方法的链式调用(例如 Object.method1().method2().method3())以及节点包装器方法 get 和 set。对于 get 方法而言，它的作用基本上类似于 YUI 2.x 的选择器(根据 CSS 选择器值来获取 DOM 元素)。

12.4　本章小结

虽然 Web 浏览器控件从开始到现在并没有多大程度的发展，但是诸如 YUI 这样的 JavaScript 库已经介入并很好地填补了这一空白，第 11 章和第 12 章的内容已经证明了这一点。这些控件还有许多不同的配置和组合。利用 YUI 提供的可配置性、可扩展性以及模块性，我们几乎可以实现任何控件。

第13章

利用 YUI 核心增强开发

本章内容简介：

- 应用名称空间和模块性
- 检测浏览器环境和可用模块
- 日志记录和调试

13.1 应用名称空间和模块性

YUI Library 的一个强项是它的组织方式。每个组件(不管是实用工具还是窗口部件)都是以 YUI 存档中的独立程序包的形式发布的。这样一来，如果一个项目只需要几个组件，那么用户不必下载所有的组件。而且，为了避免复杂的命名约定或潜在的函数名冲突，整个 YUI 库均采用名称空间机制。YUI 核心(在文件 yahoo.js 中加载)由几项关键功能组成，包括名称空间功能以及对 JavaScript 语言本身的扩展。

13.1.1 名称空间

JavaScript 本身并不支持原生的名称空间机制，但它确实支持可以通过句点表示法访问的嵌套对象字面值，这使得创建名称空间变得非常简单。例如，编写下面的代码就可以非常容易地创建名称空间 TEST.testing.tester：

```
var TEST = {
    testing: {
        tester: {
        }
    }
};
```

请注意大写的根变量名 TEST，就像 YUI 根变量名是 YAHOO 一样。这样做的目的是为了降低与全局变量空间中已有的变量名产生冲突的可能性。变量名通常并不会采用全部大写的形式，

因此与名称 yahoo 相比，存在名为 YAHOO 的变量的可能性要低很多。

YUI 有一个名为 namespace 的核心函数，它可用来动态创建名称空间。但是，它主要针对 YUI Library，因此它产生的名称空间总是以根变量名 YAHOO 开头。

```
YAHOO.namespace("YAHOO.test");  // Creates the namespace YAHOO.test
YAHOO.namespace("test");        // Creates the namespace YAHOO.test
```

注意，这两条语句均使用 namespace 函数生成同样的结果。

namespace 函数将跳过已有的对象，它只会创建尚不存在的对象。

```
YAHOO.namespace("YAHOO.wrox");
YAHOO.wrox.important = "Don't delete me!";
YAHOO.namespace("YAHOO.wrox.books.ProJSFrameworks");
```

这里的代码从 YAHOO 变量那里创建了一个 wrox 对象。然后又从 wrox 对象那里创建了一个包含字符串"Don't delete me!"的变量，接下来继续从已有的 wrox 对象那里创建 books.ProJSFrameworks 对象。换言之，第二次调用 namespace 函数并不会重新创建完整的对象链，因此不会销毁变量 important。

13.1.2 语言扩展

JavaScript 并不是一门完美的语言，至少它不具备一些“有了会更好”的功能，YUI 团队取得进展并将这些扩展功能放在名称空间 YAHOO.lang 下方。例如，有一组完整的类型验证函数用来修正或包装原生 JavaScript 代码的程序错误，或者实现某种并不存在但却是必需的类似功能(参见表 13-1)。这些功能中有些是为了便于编码，而有些则是为了满足实际的需求。例如，YAHOO.lang.isFunction 函数纠正了 Internet Explorer 有时对某些函数是对象的错误断言。

表 13-1

函　数	说　明
hasOwnProperty	包装现有的 hasOwnProperty 函数，增加了对 Safari 1.3 的支持
isArray	测试所提供对象的类型是否为 Array。在 JavaScript 中不能原生支持该功能，因此 YUI 转而针对已知的 Array 属性进行测试
isBoolean	测试提供的对象是否为 Boolean 值
isFunction	测试提供的对象是否为函数。
isNull	测试提供的对象是否为 null
isNumber	测试提供的对象是否为数值
isObject	测试所提供对象的类型是否为 Object 或 Function
isString	测试所提供对象的类型是否为 String
isUndefined	测试提供的对象是否为未定义
isValue	测试提供的对象是否包含非空值

13.1.3　模拟类继承关系

与 C++这样的常规面向对象编程语言不同，JavaScript 使用原型继承而不是类继承。尽管这个主题超出了本书的范畴，但是使用 JavaScript 模拟类继承需要一些精心设计的代码。YUI 有一些函数可用来模仿类继承机制，例如 extend 和 augment，而 JavaScript 原生的原型继承机制并不支持这一功能。

1. extend 函数

YUI extend 函数可用来实现构造函数和方法的链式调用。注意，不能继承静态成员。下面的代码清单演示如何使用 extend 函数，通过从 Person 类继承来扩展 Employee 类。

```
// Define a Person "class"
var Person = function (config) {
    this.name = config.name;
    this.gender = config.gender;
    this.height = config.height;
    this.weight = config.weight;
    alert(
        "Person: " + this.name + ", " + this.height + ", " + this.weight);
};

// Define a method for Person called init
Person.prototype.init = function () {
    alert("Person Initialized!");
};

// Define an Employee "class"
var Employee = function (config) {
    Employee.superclass.constructor.call(this, config);
    this.employeeId = config.employeeId;
    this.position = config.position;
    this.seniority = config.seniority;
    alert(
        "Employee: " + this.employeeId + ", " + this.position + ", " +
        this.seniority + "\n" +
        "Person: " + this.name + ", " + this.gender + ", " +
        this.height + ", " + this.weight);
};

// Employee inherits from Person
YAHOO.lang.extend(Employee, Person);

// Define a method for Employee called init
Employee.prototype.init = function () {
    // Call Person's init function
    Employee.superclass.init.call(this);
    alert("Employee Initialized");
};
```

```
// Instantiate an Employee object
var john = new Employee({
    name: "John",
    gender: "Male",
    height: "5'11\"",
    weight: "190lbs",
    employeeId: "424242",
    position: "Senior Basket Weaver",
    seniority: "17 years"
});

// Initialize the Employee object
john.init();
```

实际上，extend 所做的工作就是将基类的成员复制到另一个类，并创建一个指向基类的引用(通过关键字 superclass)。这样一来，从继承自基类的子类中就可以调用该基类的构造函数和方法。Employee 构造函数和 init 方法说明了这一点。

上面的代码示例创建了一个基类 Person。Person 基类接收一个名为 config 的实参，这是一个含有 name、gender、height 和 weight 属性值的对象。可以认为这些属性足够通用，能够描述任何人。最后，构造函数触发一个警告框来告诉用户它所接收到的值。在构造函数后面立即定义它的 init 方法，这个简单的方法只是触发一个警告框来说明该构造函数已经执行。

接下来，创建 Employee 类，Employee 类要继承自 Person 类。为此，在 Employee 类定义的后面立即调用 YAHOO.lang.extend 函数。这就确保当执行 Employee 构造函数时，Employee 中的 superclass 引用(Person 类)存在。

一旦已经运行 extend，Employee 类就会通过 superclass 关键字拥有一个指向 Person 的引用，因此能够调用 Person 构造函数(当它执行时立即调用)。类似地，属于 Employee 的方法能够通过关键字 superclass 来访问 Person 类的方法。这样一来，当调用 Employee 类的 init 方法时，它要做的第一件事情就是触发 Person 类的 init 方法。实际上，这些方法采用了链式调用(就像通过同样的原理链式调用构造函数一样)。

对于这个代码示例而言，创建 Employee 对象 john 会触发 Employee 构造函数，而这又会调用 Person 构造函数。这会向屏幕上抛出 Person 类的警告框，之后又会弹出 Employee 类的警告框。一旦构造函数执行完毕，就会调用 john 的 init 方法，而这会再次调用 Person 类的 init 方法，以及之后调用 Employee 类的 init 方法。这些调用再次先从 Person 类抛出警告消息，然后从 Employee 类抛出警告消息。

2. augmentObject 和 augmentProto 函数

有时，我们只需要利用另一个对象的成员来扩充现有的对象，这就是 augment 和 augmentObject 函数的作用。

augmentObject

下面的代码清单演示了 augmentObject 函数的实际运行情况(参见图 13-1)。

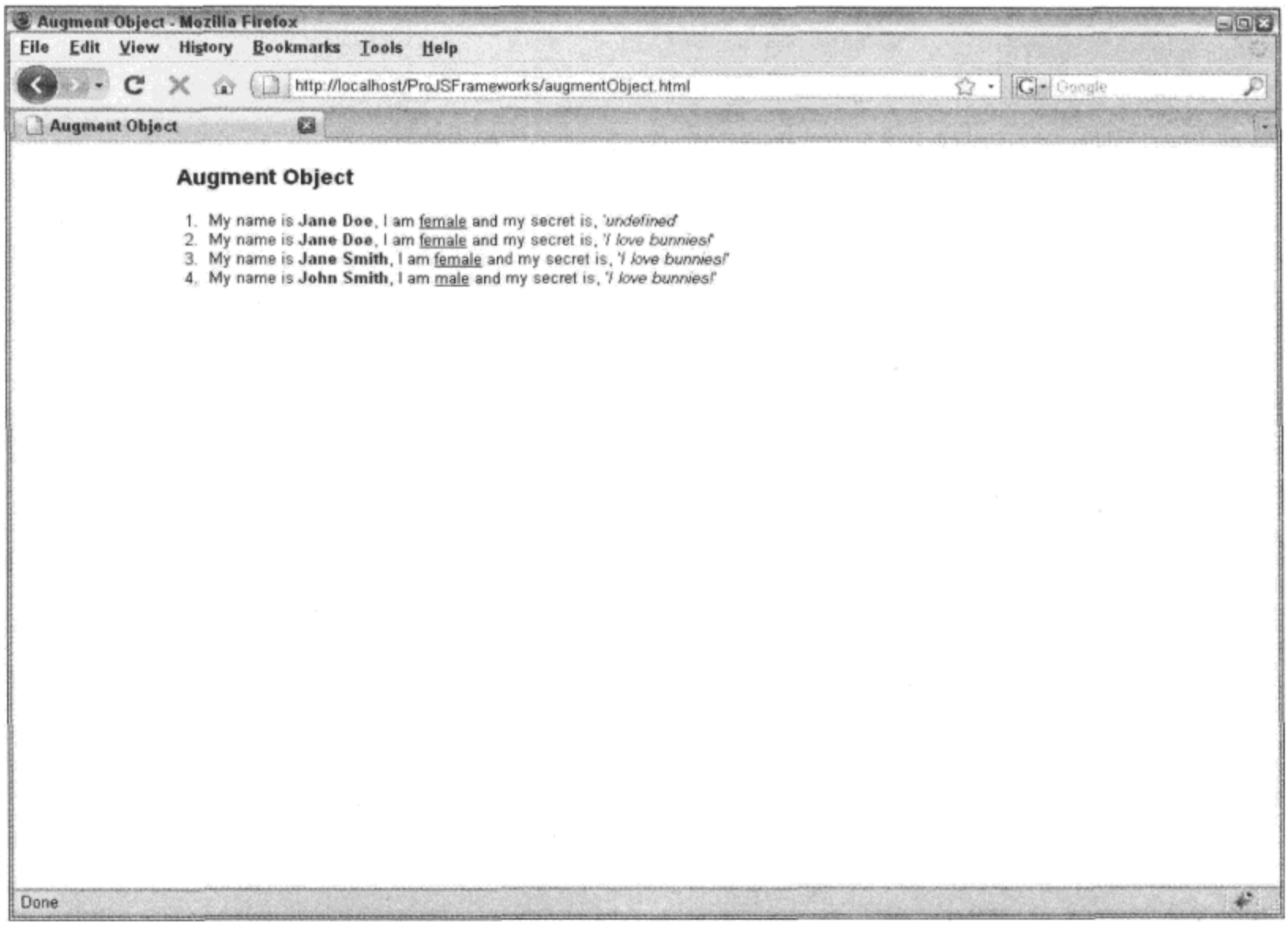


图　13-1

在这里实例化两个 Person 类型的对象：一个名为 john，另一个名为 jane。这里使用 augmentObject 函数逐步地把 john 的属性提供给 jane。

```
(function () {
    var Person = function (fname, lname, gender) {
        this.fname = fname;
        this.lname = lname;
        this.gender = gender;
    };

    Person.prototype.speak = function (elId, nodeName) {
        var el = document.getElementById(elId);
        el = el || document.body;
        nodeName = nodeName || "p";
        el.innerHTML += "<" + nodeName + ">" +
                        "My name is " +
                        "<strong>" +
                            this.fname + " " + this.lname +
                        "</strong>" +
                        ", I am " +
                        "<u>" +
                            this.gender +
```

```
                        "</u> " +
                        "and my secret is, " +
                        "'<em>" +
                            this.secret +
                        "</em>'" +
                        "</" + nodeName + ">";
};

// Instantiate a Person object named John Smith
var john = new Person("John", "Smith", "male");

// Give John a secret
john.secret = "I love bunnies!";

// Instantiate a Person object named Jane Doe
var jane = new Person("Jane", "Doe", "female");

/*
 * Output jane's properties to the screen.
 * Yields: "My name is Jane Doe, I am female and my secret is, 'undefined'"
 */
jane.speak("output", "li");

/* Copy properties from john that jane doesn't already have
 *   and output jane's properties to the screen.
 * Yields: "My name is Jane Doe, I am female and my secret is,
 * 'I love bunnies!'"
 */
YAHOO.lang.augmentObject(jane, john);
jane.speak("output", "li");

/*
 * Copy the value of john's "lname" property to jane
 *   whether she has a value in her "lname" porperty or not.
 *   Output jane's properties to the screen.
 * Yields: "My name is Jane Smith, I am female and my secret is,
 * 'I love bunnies!'"
 */
YAHOO.lang.augmentObject(jane, john, "lname");
jane.speak("output", "li");

/*
 * Overwrite all of jane's properties with john's and
 *   output jane's properties to the screen.
 * Yields: "My name is John Smith, I am male and my secret is,
 * 'I love bunnies!'"
 */
YAHOO.lang.augmentObject(jane, john, true);
jane.speak("output", "li");
})();
```

首先，创建 Person 类，并定义了 speak 方法。speak 方法附带两个可选参数：一个目标输出元素 ID 和一个用来包装它的输出结果的节点名称。如果两个参数都没有提供，那么该方法会直接在文档正文中呈现段落标记。

接下来，使用名“John”、姓“Smith”以及性别“male”(分别存储在 fname、lname 和 gender 属性中)来实例化一个 Person 对象。最后，为 John 定义一个 secret 属性。secret 属性并不是 Person 类的原生属性，而是一个动态添加的属性，其目的是为了演示 augmentObject 的默认行为。然后创建了对象 jane，它也是从 Person 类实例化的对象，其属性分别为：名“Jane”、姓“Doe”以及性别“female”。但是，Jane 并没有 secret 属性。

当调用 Jane 的 speak 方法时，向屏幕上输出如下文本：

```
My name is Jane Doe, I am female and my secret is, 'undefined'
```

注意，Jane 的 secret 属性值为“undefined”。这自然是因为 jane 对象并没有一个名为 secret 的属性。但是，一旦运行 augmentObject，jane 就会接收 john 的 secret 属性。这是因为 augmentObject 的默认行为就是利用另一个对象中目标对象所不具备的成员来扩充目标对象。这样，它就不会意外地不经明确声明就覆盖目标对象中的成员。

下一次调用 Jane 的 speak 方法时，就会向屏幕上输出如下文本：

```
My name is Jane Doe, I am female and my secret is, 'I love bunnies!'
```

现在已经证明 Jane 接收到 John 的 secret 属性。接下来，再次运行 augmentObject，但是这一次带上第三个参数“lname”。该参数将指示 augmentObject 覆盖目标对象中 lname 成员的值，而不论它是否已经包含值。

这一次 Jane 证明她的姓已经变成了“Smith”：

```
My name is Jane Smith, I am female and my secret is, 'I love bunnies!'
```

最后，再次运行 augmentObject，这一次将第三个属性设为 true。这将指示 augmentObject 覆盖目标对象中的所有属性。

现在调用 Jane 的 speak 方法，她实际上名为“John Smith”，是“男性”且“喜爱轻音乐”：

```
My name is John Smith, I am male and my secret is, 'I love bunnies!'
```

augmentProto(又名 augment)

有时，我们需要的是继承另一个类的原型的成员，但不需要其他的成员。下面的示例演示了这种情况，它允许 Animal 类从 Person 类那里继承 speak 方法。但是，它并没有从 Person 那里继承 fname、lname 和 gender 属性，这是因为没有将这些属性分配给它的原型。下面的示例使用 augment(augmentProto 的别名)来实现这个需求。

```
(function () {
    var Person = function (fname, lname, gender) {
        this.fname = fname;
        this.lname = lname;
        this.gender = gender;
    };

    Person.prototype.speak = function (msg) {
```

```
        var n = (this.fname) ? this.fname + " " + this.lname : this.name;
        alert(n + " says '" + msg + "'");
    };

    var Animal = function (name, species) {
        this.name = name;
        this.species = species;
    };

    YAHOO.lang.augment(Animal, Person);

    var john = new Person("John", "Smith", "male");
    john.speak("Hello world!");

    var spot = new Animal("Spot", "dog");
    spot.speak("Woof!");
})();
```

这个示例所做的第一件事情是创建一个具有 fname、lname 和 gender 属性的 Person 类。然后创建一个名为 speak 的方法，并将其添加到 Person 的原型中。最后，创建一个带有 name 和 species 属性的 Animal 类。注意，现在 Animal 并没有 speak 方法。YUI augment 函数用来让 Animal 具有 speak 方法。一旦这个继承操作完成，john 和 spot 对象就都能够运行 speak 方法。

3. 更多语言扩展：merge、dump 和 substitute

下面的代码示例给出了另外 3 种有用的语言扩展，即 merge、dump 和 substitute。前两个扩展有助于操作对象，而第三个扩展是一个用来重用文本字符串的工具。

```
(function () {
    function out(tag, msg) {
        var html = "<" + tag + ">" + msg + "</" + tag + ">";
        document.getElementById("bd").innerHTML += html;
    };

    var heading = "The following is a dump of the <em>{objName}</em> data object:";

    var addressData = {
        streetNumber: "711",
        streetName: "de la Commune W.",
        city: "Montreal",
        province: "Quebec",
        country: "Canada",
        postalCode: "H3C 1X6"
    };

    var personData = {
        firstName: "John",
        middleName: "Connor",
        lastName: "Doe",
```

```
        age: 32
    };

    var combined = YAHOO.lang.merge(addressData, personData);

    out("h2", YAHOO.lang.substitute(heading, {"objName": "address"}));
    out("p", YAHOO.lang.dump(addressData));

    out("h2", YAHOO.lang.substitute(heading, {"objName": "person"}));
    out("p", YAHOO.lang.dump(personData));

    out("h2", YAHOO.lang.substitute(heading, {"objName": "combined"}));
    out("p", YAHOO.lang.dump(combined));
})();
```

这个代码示例的主要作用是将 addressData 和 personData 这两个对象合并在一起(参见图 13-2)。dump 函数用来把对象数据序列化并输出到屏幕上，而 substitute 函数用来使用适当的对象名称更新标题文本以便重用它。这里创建了一个实用工具函数 out 来简化向屏幕输出内容并确保代码有效的任务。

图　13-2

4. trim 函数

虽然大部分现代编程语言都包含 trim 函数，但奇怪的是，JavaScript 中并没有这个函数。YUI 使用自己的 trim 函数填补了这一空白。该函数接收一个字符串并将它找到的开头和结尾空白符移除。如果接收的不是字符串，那么它只是将该参数原样返回。

下面的示例演示了 trim 函数的实际运行情况，该示例分析一个单词数组，每个单词的开头和结尾都有空白符。第一次将该单词数组输出到屏幕上(再次使用前一个示例中引入的实用工具函数 out)，并将所有空白符保留。但是，第二次将所有空白符删除(参见图 13-3)。

```
(function () {
    function out(tag, msg) {
        var html = "<" + tag + ">" + msg + "</" + tag + ">";
        document.getElementById("bd").innerHTML += html;
    };

    function makeMsg(arr, trim) {
        var msg = "", word;
        for (var i = 0; arr[i]; i += 1) {
            if (trim) {
                word = YAHOO.lang.trim(arr[i]);
            } else {
                word = arr[i];
            }
            msg += word + " ";
        }
        return msg;
    };

    var words = [
        "   These   ",
        " words         ",
        "   are        ",
        "  really     ",
        "       spaced   ",
        "                apart!     "
    ];

    out("h2", "Untrimmed");
    out("pre", makeMsg(words));
    out("h2", "Trimmed");
    out("pre", makeMsg(words, true));
})();
```

图　13-3

5. later 函数

最后，YUI 团队将 JavaScript 原生函数 setTimeout 和 setInterval 组合成一个包装器 later。这个函数的主要作用是提供如下能力：定义关键字 this 在 setTimeout 或 setInterval 调用的函数中所指向的对象。

下面的示例演示了如何使用 YUI 的 later 函数来延迟执行函数和重复执行函数。两种情况下的语法相同，但后者还有一个额外的 Boolean 值 true，用来指示该调用必须重复执行。此外还要注意的是，将 later 的第一次使用分配给变量 repeat。这样做的目的是可以在后面将其取消(在第二次使用 later 时)。

```
(function () {
    // Set scoped vars with initial values
    var message = "Warning, this message will self-destruct in {sec} seconds!";
    var counter = 5;

    // Message class constructor
    function Message(targetEl, msg) {
        this.target = YAHOO.util.Dom.get(targetEl);
        this.msg = msg;
    };

    // Method to output message to screen
    Message.prototype.say = function () {
        this.target.innerHTML = this.msg;
    };
```

```
    // Message to clear output
    Message.prototype.clear = function () {
        this.target.innerHTML = "";
    };

    // Instantiate a Message object, set initial message and output it
    var destruct = new Message("output");
    destruct.msg = YAHOO.lang.substitute(message, {sec: counter});
    destruct.say();

    // Set up a repeater that decrements counter and updates message every second
    var repeat = YAHOO.lang.later(1000, destruct, function () {
        counter -= 1;
        this.msg = YAHOO.lang.substitute(message, {sec: counter});
        this.say();
    }, null, true);

    // Clear the output and cancel the repeater after 5 seconds
    YAHOO.lang.later(counter * 1000, destruct, function () {
        this.clear();
        repeat.cancel();
    });
})();
```

这个示例显示消息“Warning, this message will self-destruct in 5 seconds!”(该消息 5 秒钟之后将自动销毁)，然后继续读秒，将 5 替换成 4，然后替换成 3，以此类推(参见图 13-4)，直到达到 0，此时将从屏幕上移除该消息。

图　13-4

为了实现这个目标，使用了 later 的两个实现。第一个实现每秒将屏幕上的消息更新一次，而第二个实现将消息清除并取消重复执行的 later 函数。

13.2　检测浏览器环境和可用模块

高效 DOM 编程取决于 JavaScript 代码是否能够感知它所运行的环境。这不仅包括它能够确定所需的 DOM 成员是否可用，还包括它正在什么浏览器(也称为用户代理)中运行。JavaScript 代码还应该能够确定当前加载的库组件以及可用的库组件。

13.2.1　YAHOO.env.ua

对于环境感知来说，最好的做法是*功能检测*(feature detection)，就是在使用某项功能(或 DOM 成员)之前要先进行测试。这要好于浏览器检测，原因在于确定脚本运行所在的浏览器并不能保证所需的 DOM 成员一定存在。虽然浏览器(或用户代理)检测并不是推荐的做法，但它有时候还是有一些用处的。有时，浏览器行为差异并不可测试，例如 IE 6 中的选择框问题：除了 iframe 之外，所有 HTML 内容都不能位于选择框元素上方。在与此类似的情况下，如果知道代码在什么浏览器上运行，就可以针对它运行特殊的例程来解决已知的问题。

通常情况下，检测浏览器意味着分析 userAgent 字符串，而这是一个比较棘手的操作。YUI 将这个过程简化，它分析 userAgent 字符串并使用分析结果填充 YAHOO.env.ua 对象。这个对象包含如下属性：air、gecko、ie、mobile、opera 和 webkit。每个属性(除了 mobile)都包含一个用来表示已检测版本的浮点值或 0，后者表示完全没有检测到给定浏览器。因为移动市场如此多样化，所以 mobile 属性包含一个带有相关用户代理信息的字符串。目前，能够检测到的移动用户代理只有 Safari、iPhone/iPod Touch、Nokia N 系列基于 WebKit 的浏览器以及 Opera Mini。

下面的示例演示如何使用 YUI 的用户代理检测机制：

```
if (YAHOO.env.ua.ie> 5 && YAHOO.env.ua.ie <7) {
    // iframe code for IE6
}
```

13.2.2　YAHOO.env.getVersion

对于环境感知，YUI 还提供了检测所有已加载 YUI 模块的版本以及内部版本号的功能。getVersion 方法检验给定 YUI 模块是否加载，如果找到该模块，就返回一个包含该模块版本信息的对象。下面这个示例给出了 getVersion 的实际使用方式：

```
<html>
    <head>
        <title>YAHOO.env.getVersion</title>
        <link rel="stylesheet" type="text/css" href="reset-fonts-grids.css" />
        <link rel="stylesheet" type="text/css" href="base-min.css" />
```

```
</head>
<body class="yui-skin-sam">
    <div id="doc">
        <div id="hd">
            <h1>YAHOO.env.getVersion</h1>
        </div>
        <div id="bd">
        </div>
        <div id="ft">
        </div>
    </div>
    <script src="build/yahoo-dom-event/yahoo-dom-event.js"></script>
    <script src="build/json/json-min.js"></script>
    <script>
        (function () {
            function verify(moduleName) {
                var module = YAHOO.env.getVersion(moduleName);
                var msg = "";
                if (module) {
                    msg += "Module Name: " + module.name + " [LOADED]\n";
                    msg += "Version: " + module.version + "\n";
                    msg += "Build: " + module.build + "\n";
                    msg += "--\n";
                } else {
                    msg += "Module Name: " + moduleName + " [NOT LOADED]\n";
                    msg += "--\n";
                }
                return msg;
            };

            var status = "";
            status = verify("json");
            status += verify("yahoo");
            status += verify("container");
            status += verify("event");
            status += verify("dom");
            status += verify("history");

            alert(status);
        })();
    </script>
</body>
</html>
```

这个代码示例基本上就是通过一个名为 verify 的包装器函数 6 次调用 getVersion 函数。每次调用 verify 时，它都使用 getVersion 计算传给它的模块名，然后根据给定模块是否加载，它返回一条适当的消息。将多次调用 verify 返回的消息聚集到变量 status 中之后，就通过一个警告框将它们输出到屏幕上(参见图 13-5)。

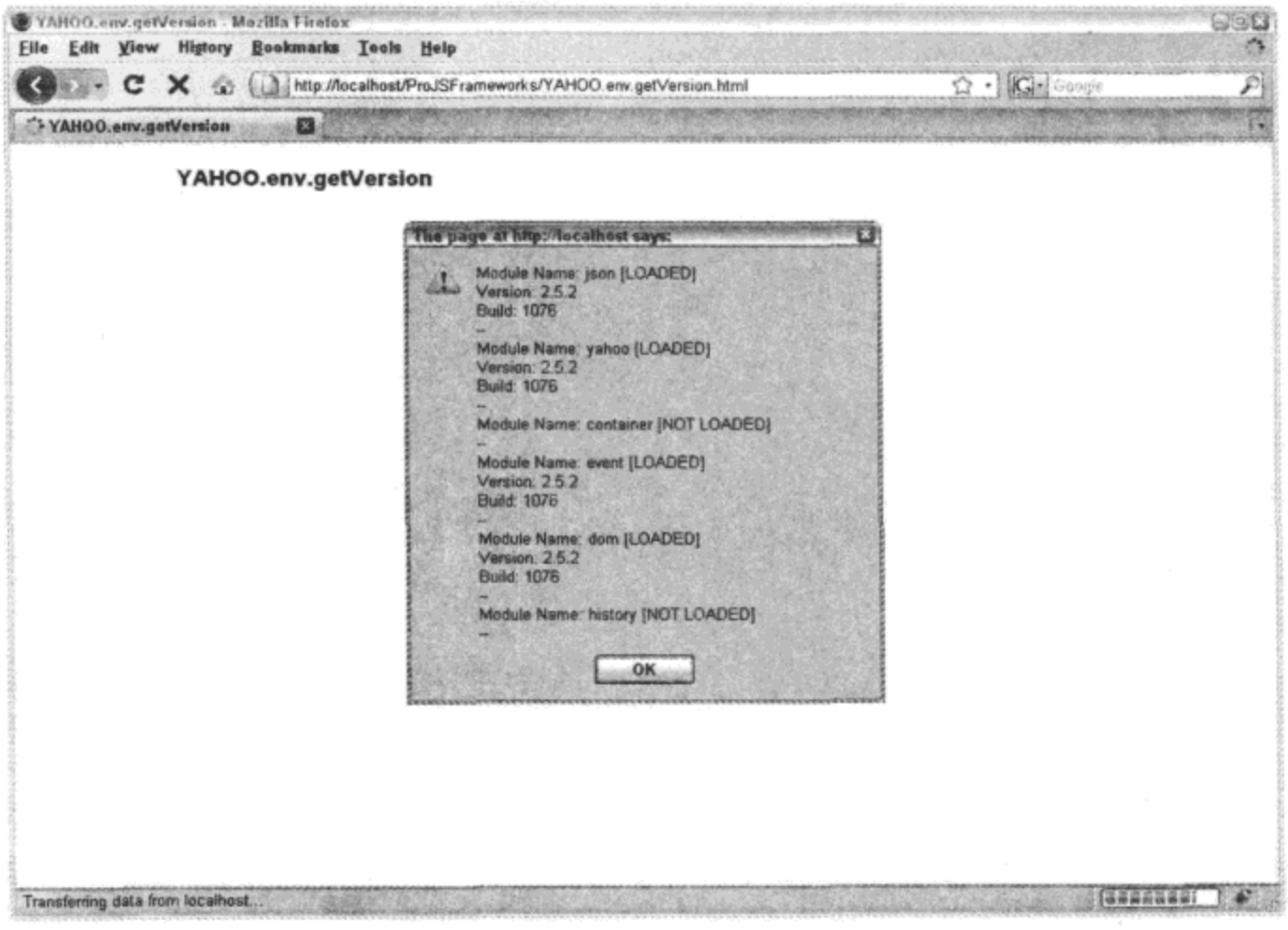


图　13-5

13.2.3　YAHOO_config

最后，在有些情况下，加载 YUI 模块的时机并不确定。全局对象 YAHOO_config 可用来检测加载模块的时刻。

```
<html>
    <head>
        <title>YAHOO_config</title>
        <link rel="stylesheet" type="text/css" href="reset-fonts-grids.css" />
        <link rel="stylesheet" type="text/css"
            href="container/assets/container.css" />
        <link rel="stylesheet" type="text/css" href="base-min.css" />
        <script>
            YAHOO_config = {
                listener: function (module) {
                    if (module.name === "container") {
                        YAHOO.util.Dom.addClass("yui-skin-sam", document.body);
                        var panel = new YAHOO.widget.Panel("hello", {
                            width: "350px",
                            fixedcenter: true,
                            modal: true
                        });

                        var msg = "The code for this panel was defined " +
                                  "at the top of this page but it was only " +
                                  "called once the container module was loaded, " +
                                  "at the end of this page.";
                        panel.setHeader("YAHOO_config");
```

```
                    panel.setBody(msg);
                    panel.render("bd");
                };
            }
        };
    </script>
</head>
<body class="yui-skin-sam">
    <div id="doc">
        <div id="hd">
            <h1>YAHOO_config</h1>
        </div>
        <div id="bd">
        </div>
        <div id="ft">
        </div>
    </div>
    <script src="build/yahoo-dom-event/yahoo-dom-event.js"></script>
    <script src="build/container/container-min.js"></script>
</body>
</html>
```

这个示例分配一个回调函数 YAHOO_config.listener，每次加载模块时都会调用它。这里的代码只检查 container 模块是否存在，实际上它总共会调用 5 次，针对 yahoo、dom 和 event 各调用一次，然后针对 yahoo-dom-event 调用一次，最后针对 container 调用一次。如果检测到 container 模块存在，就会创建一个新的面板并呈现到屏幕上(参见图 13-6)。

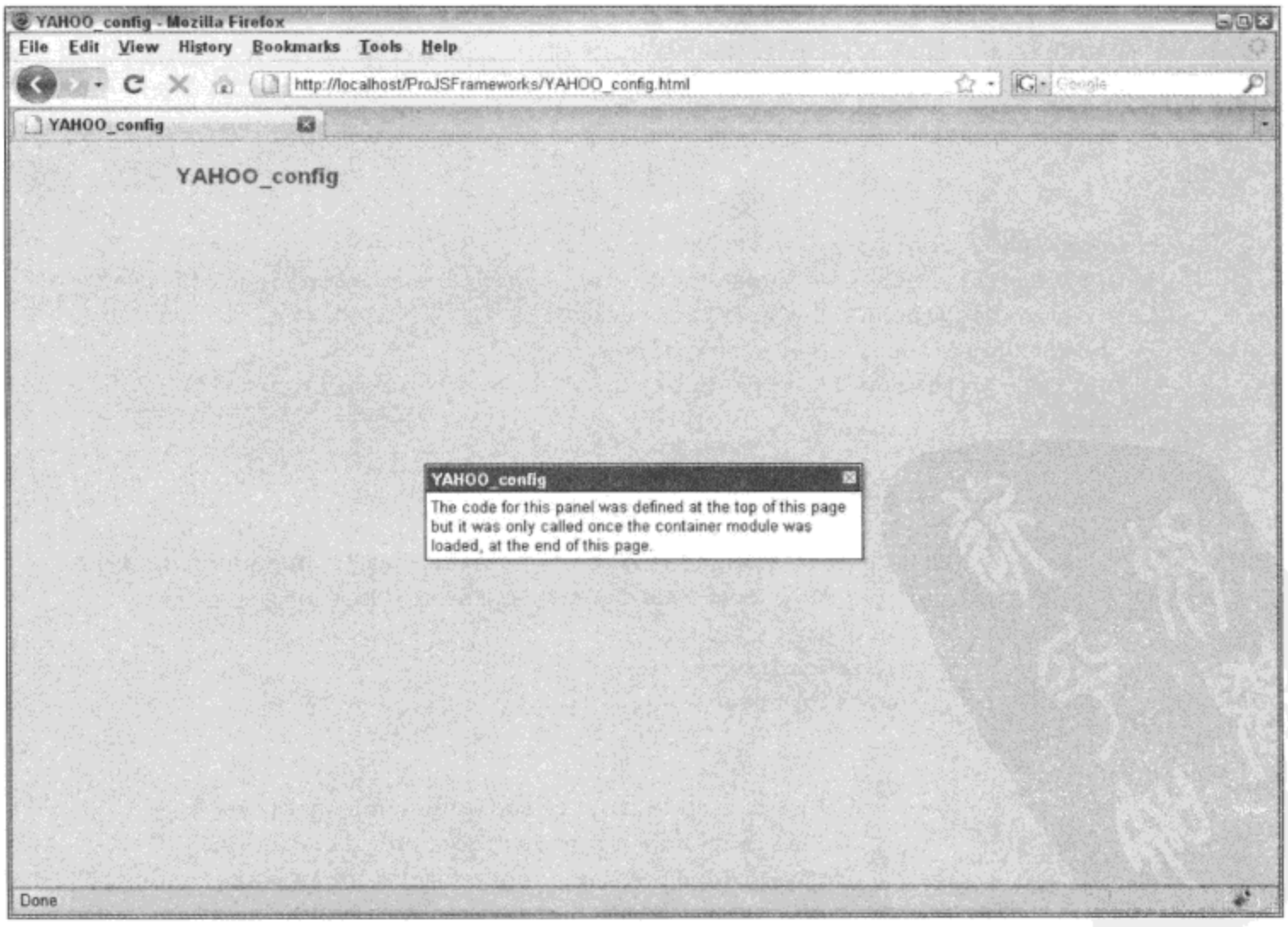


图 13-6

13.3　日志记录和调试

一直以来，JavaScript 程序员的调试技术就是通过 alert 函数输出调试数据。

```
var apples = 100;
var oranges = 200;
alert(apples + oranges); // outputs 300
```

对于类似前面示例的小型的、孤立的实例来说，这样做可以工作良好。但是当引入循环时，使用 alert 很快就会变得不可行。

```
var fruit = 300;
for (var i = 1; i <= fruit; i += 1) {
    alert(i);
}
```

这个示例将导致 300 个警告框相继出现，这就使得该代码的调试工作变得非常让人厌烦。这里需要的是一种方式来把 i 的值输出到某个不会中断程序执行流程的地方。在本节前面的示例中，我们编写自定义函数(例如 out)来将消息输出到屏幕。尽管这种方式可行，但在调试环境中并不实用。相反，我们需要的是一个能够轻易打开和关闭且不会干扰正在调试的页面的控制台。麻烦在于，虽然大多数浏览器都有内置的 JavaScript 控制台(例如 Firefox 和 Safari)，但其他浏览器并没有该控制台。此外，每种控制台支持的语法并不相同。这可能导致代码中试图访问并不存在的控制台的 JavaScript 错误。

下面介绍 YUI 的 Logger 控件。加载 YUI Library 之后，调用 YAHOO.log 并不会引起任何错误，即使并不存在 Logger 对象。YUI 足够智能，能够自动地取消它不能输出的调用，尽管它确实将这些调用记录下来。因此，对于没有加载 Logger 脚本文件的场合，调用 YAHOO.log 并不会抛出“log is not defined”(日志未定义)错误。

下面的示例演示如何实现 LogReader，该对象负责显示日志消息。

```
<html>
    <head>
        <title>Logger</title>
        <link rel="stylesheet" type="text/css" href="reset-fonts-grids.css" />
        <link rel="stylesheet" type="text/css" href="base-min.css" />
        <link rel="stylesheet" type="text/css"
            href="build/logger/assets/skins/sam/logger.css" />
    </head>
    <body class="yui-skin-sam">
        <div id="doc">
            <div id="hd">
                <h1>Logger</h1>
            </div>
            <div id="bd">
            </div>
            <div id="ft">
            </div>
        </div>
```

```
        <script src="build/yahoo-dom-event/yahoo-dom-event.js"></script>
        <script src="build/logger/logger-min.js"></script>
        <script>
            (function () {
                var logger = new YAHOO.widget.LogReader("bd");
                YAHOO.log("Hello World!");
            })();
        </script>
    </body>
</html>
```

注意 log 方法并没有与 logger 对象相关联。这样就可以把调试消息与日志记录器解除关联，这意味着可以将它们安全地留在产品代码中，而不用担心调试消息会出现在网站上。唯一真正的影响是 YUI 仍将记录消息，因此会影响性能。更多相关内容请参见第 16 章。

在上面的代码清单中，我们所做的第一件事情是包括 logger.css 文件，它包含了 LogReader 的基本 CSS 规则。但是，除非把 CSS 类名 yui-skin-sam 关联到 HTML 标记，否则该代码并不能运行。这个 CSS 类名将添加到 body 标记。最后，实例化一个 LogReader 对象，并把在其中呈现日志消息的元素 ID 作为它的唯一参数传给它。注意，“Hello World!”日志消息在 LogReader 实例化之后出现。这并不是必需的顺序。实际上，如果将这两行代码的顺序颠倒，那么只要日志记录器完成自身的呈现，它就立即输出存储的日志消息，如图 13-7 所示。

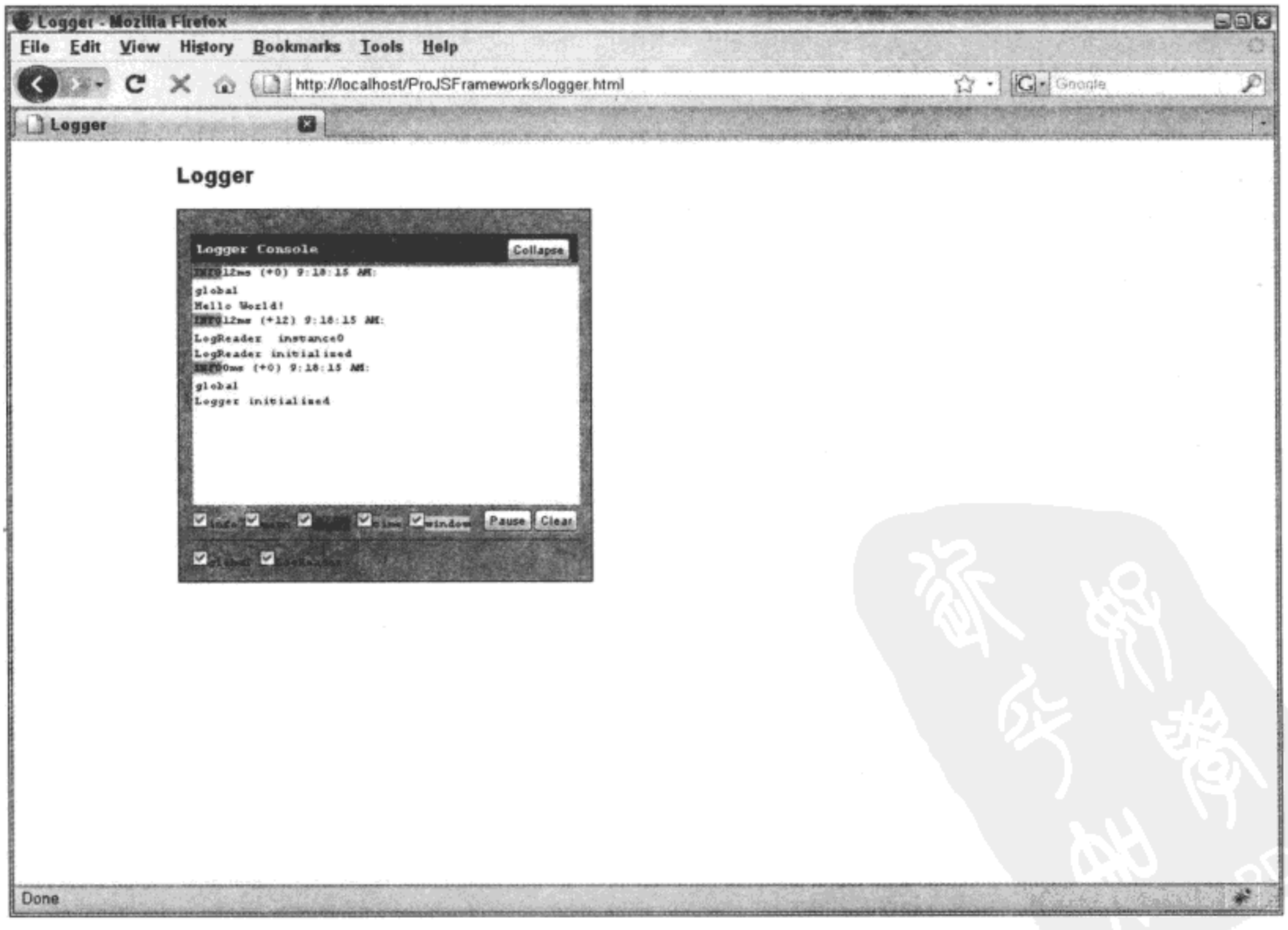


图　13-7

LogReader 能够输出 5 种不同类型的消息：info、warn、error、time 和 window(参见图 13-7)。它们可用来定制在不同的情况下显示不同类型的消息。因此，如果某条 try 语句失败，代码进入 catch 语句，那么将消息作为 warn 类型(可以轻易地通过其独特的颜色来识别它)记录到日志中就是一种良好的做法。下面给出了定义日志消息类型的语法：

```
YAHOO.log("Oops, something went wrong!", "warn");
```

13.4　本章小结

JavaScript 和 DOM 的原生功能可能比较有限，但是像 YUI 这样的库已经建立起来，从而填补了原生功能与实际需求之间的鸿沟。这些进展使得构建和调试现代网站变得容易很多。

第14章

处理数据、表和图表

本章内容简介：

- 格式化日期和数字
- 获取数据源
- 呈现表数据
- 绘制图表和图形

14.1 格式化日期和数字

在显示内容时，日期和数字的格式化非常关键。但是，JavaScript 并不原生支持日期和数字的格式化。YUI 的 DataSource Utility 中有几个隐藏的工具，可用作 DataTable 控件的格式化辅助器。第一个工具等同于 strftime，而第二个工具提供了格式化数字的灵活方式。可以在 YAHOO.util.Date 和 YAHOO.util.Number 下分别找到这两个工具。诚然，仅仅是为了格式化就要加载多达 30KB 的 datasource-min.js 文件并不合理，但是如果已经加载该文件，那么使用它就很有意义！

14.1.1 日期

表 14-1 列出了 YAHOO.util.Date.format 支持的格式化选项。

表 14-1

格式化选项	说　明
%a	根据当前区域设置的缩写工作日名称
%A	根据当前区域设置的工作日全称
%b	根据当前区域设置的缩写月份名称
%B	根据当前区域设置的月份全称
%c	当前区域设置的首选日期和时间表示法
%C	世纪数字(年份除以 100 并取整，范围是 00~99)

(续表)

格式化选项	说　明
%d	本月第几天，十进制数字(范围是 01~31)
%D	同%m/%d/%y
%e	本月第几天，十进制数字，只有个位的数字添加空格前缀(范围是' 1'~'31')
%F	同%Y-%m-%d(ISO 8601 日期格式)
%g	与%G 类似，但没有世纪数字
%G	对应到 ISO 周数的 4 位数字年份
%h	同%b
%H	小时，十进制数字，使用 24 小时计时法(范围是 00~23)
%I	小时，十进制数字，使用 12 小时计时法(范围是 01~12)
%j	本年第几天，十进制数字(范围是 001~366)
%k	小时，十进制数字，使用 24 小时计时法(范围是 0~23)，只有个位数字的小时添加空格前缀(另参见%H)
%l	小时，十进制数字，使用 12 小时计时法(范围是 1~12)，只有个位数字的小时添加空格前缀(另参见%I)
%m	月份，十进制数字(范围是 01~12)
%M	分钟，十进制数字
%n	换行符
%p	根据给定时间值添加'AM'或'PM'，或者当前区域设置的对应字符串
%P	与%p 类似，但是字符串为小写
%r	采用 a.m.和 p.m.表示法表示的时间，等同于%I:%M:%S %p
%R	采用 24 小时表示法表示的时间，等同于%H:%M
%s	自新纪元以来的秒数，也就是从 1970-01-01 00:00:00 UTC 开始计算的秒数
%S	秒数，十进制数字
%t	制表符
%T	当前时间，等同于%H:%M:%S
%u	工作日，十进制数字[1,7]，其中 1 表示星期一
%U	本年的周数，十进制数字，从第一个星期天(作为第一周的第一天)开始计算
%V	本年的 ISO 8601:1988 周数，十进制数字，范围是 01~53，其中第一周是第一个至少有 4 天属于本年的一周，而且将星期一作为本周的第一天
%w	本周第几天，十进制数字，星期天为 0
%W	当年的周数，十进制数字，从第一个星期一(作为第一周的第一天)开始计算
%x	当前区域设置的首选日期表示法，不含时间
%X	当前区域设置的首选时间表示法，不含日期
%y	年份，十进制数字，不含世纪数字(范围是 00~99)
%Y	年份的十进制数字，包括世纪数字
%z	时区的数字表示
%Z	时区名称或其缩写
%%	%字符的字面值

该表中的说明来自于 datasource-debug.js

下面这个简单示例演示如何使用 format 方法：

```
YAHOO.util.Date.format(
    new Date()
    {
        format: "Today's date is %A, %B %C, %G. The time is currently %l:%m%p."
    });
    /*
     * Returns:
     *   Today's date is Tuesday, February 20, 2009. The time is currently 9:02PM.
     */
```

实际上，除了传给格式化器的日期对象之外，还有通过配置对象传入的格式化指令。配置对象包含一个参数“format”以及一个包含格式代码(将会被正确的数据替换)的字符串。因此，在上面的代码清单中，%A 会被替换成 Tuesday，%B 会被替换成 February 等。

14.1.2　数字

YUI 的数字格式化器在格式化数字方面提供了一定的灵活性。例如，它可以附加前缀(例如在格式化货币时添加前缀$)、四舍五入的小数位数、小数点分隔符、千位分隔符以及后缀。

下面的代码清单同时为英语读者和法语读者输出格式化后的美国国债(参见图 14-1)。它获取数字 10773608518631.0509482739，输出如下内容：

```
The U.S. national debt at the time of this book's writing was
approximately $10,773,608,518,631.05 USD
La dette nationale des États-Unis au moment d'écriture de ce livre été
environs 10 773 608 518 631,05$ USD
```

```
<html>
    <head>
        <title>Number Formatting</title>
        <link rel="stylesheet" type="text/css" href="reset-fonts-grids.css" />
        <link rel="stylesheet" type="text/css" href="base-min.css" />
        <style type="text/css">
        </style>
    </head>
    <body>
        <div id="doc">
            <div id="hd">
                <h1>Number Formatting</h1>
            </div>
            <div id="bd">
            </div>
            <div id="ft">
            </div>
        </div>
        <script src="build/yahoo-dom-event/yahoo-dom-event.js"></script>
        <script src="build/datasource/datasource-min.js"></script>
        <script>
            (function () {
                var num = 10773608518631.0509482739;
                var bd = document.getElementById("bd");
```

```
            // In English

            var msg = "The U.S. national debt at the time of this book's " +
                    "writing was approximately ";
            msg += YAHOO.util.Number.format(num, {
                prefix: "$",
                decimalPlaces: 2,
                decimalSeparator: ".",
                thousandsSeparator: ",",
                suffix: " USD"
            });
            bd.innerHTML = "<p>" + msg + "</p>";

            // In French

            msg = "La dette nationale des États-Unis au moment d'écriture " +
                  "de ce livre été environs "
            msg += YAHOO.util.Number.format(num, {
                prefix: "",
                decimalPlaces: 2,
                decimalSeparator: ",",
                thousandsSeparator: " ",
                suffix: "$ USD"
            });
            bd.innerHTML += "<p>" + msg + "</p>";
        })();
    </script>
</body>
</html>
```

图 14-1

14.2　获取数据源

在当今的 Web 应用程序世界中，数据源的处理已经变成一个不可避免的现实问题。如果不能预先知道数据的类型和格式，或者数据的类型和格式会随着时间改变，那么构建依赖外部数据的可重用组件就会成为一个问题。能够抽象连接是处理这个问题的必要方式。YUI 的 DataSource Utility 使得我们只需要很少的努力就可以完成这个抽象过程(参见图 14-2)。

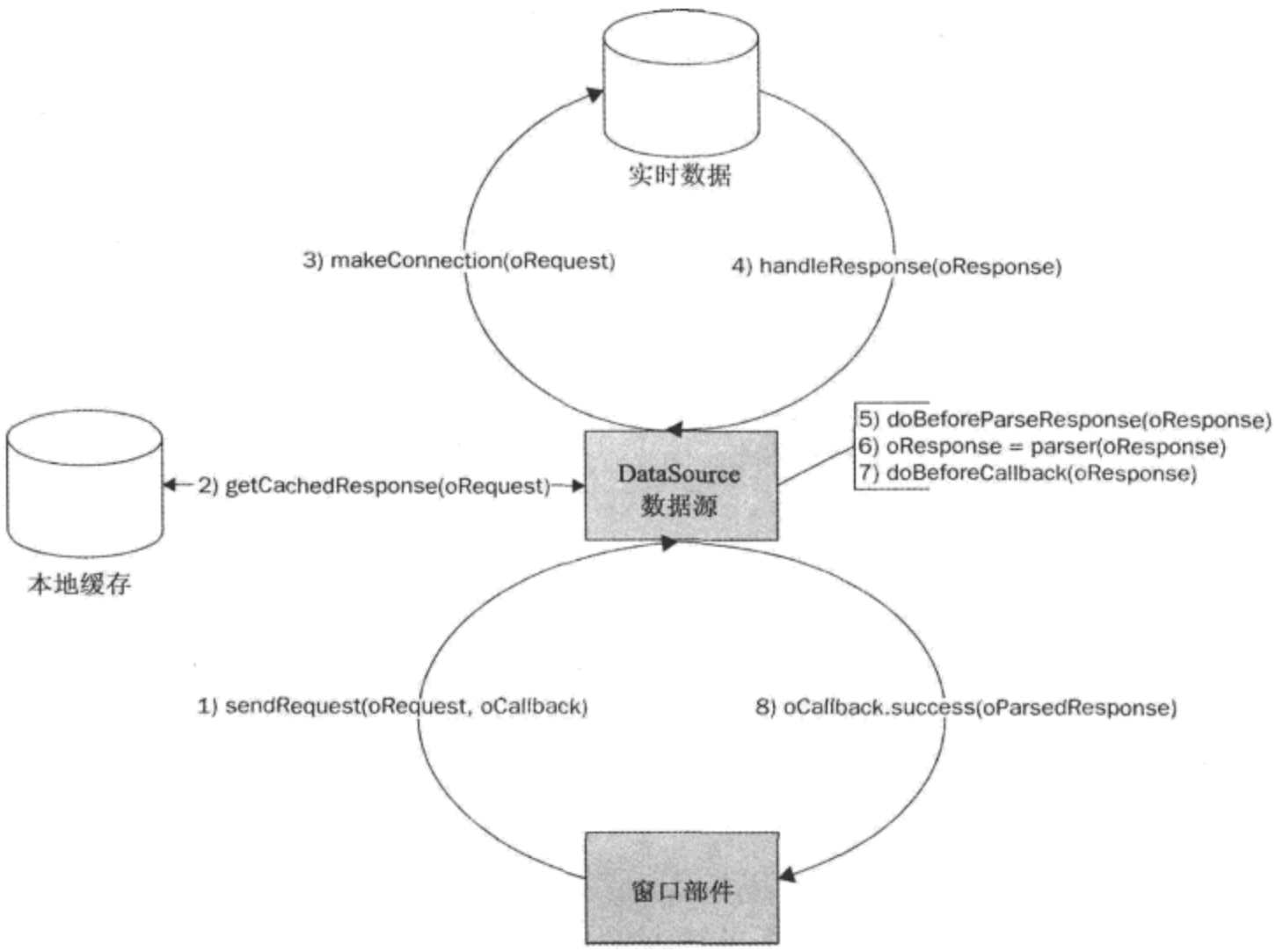


图　14-2

(1) 通过 DataSource 对象的 sendRequest 方法对该对象发出请求。

sendRequest 方法带有两个参数。第一个参数是与该请求相关的字符串。因此，对于远程数据，它将是类似"id=abc & show=all"的字符串；而对于本地数据，它可能会更简单，例如"xyz"。这个参数可能实际上与该调用无关，因此可以将其设为 null。

sendRequest 的第二个参数是一个用来为 success 函数、failure 函数、回调执行作用域 scope 以及任意实参对象 argument 指定回调信息的对象字面值。这个回调对象的参数名称分别为 success、failure、scope 和 argument。

(2) DataSource 对象检验它的缓存中是否已经存储请求的数据。

(3) 如果请求的数据不在缓存中，那么 DataSource 对象调用实时数据源。

(4) 以原始形式返回请求的数据。

(5) 此时可以通过 doBeforeParseResponse 方法访问和修改返回的数据。

(6) DataSource的主要工作在这里完成，它根据指定的dataType和模式分析原始数据。

(7) doBeforeCallback方法允许在把经过分析的数据缓存起来之前修改它。

(8) 使用如下的实参将经过分析的数据返回给发出请求的窗口部件：oRequest、oParsedResponse和oPayload。

第一个实参映射到发送给sendRequest方法的第一个值，而最后一个实参映射到oCallback对象。但oParsedResponse实参是一个新对象，它包含如下属性：tId、results、error、cached、meta。

下面的示例演示了如何将DataSource与一个简单的数组关联。

```
<html>
    <head>
        <title>DataSource--Simple Array</title>
        <link rel="stylesheet" type="text/css" href="reset-fonts-grids.css" />
        <link rel="stylesheet" type="text/css" href="base-min.css" />
    </head>
    <body>
        <div id="doc">
            <div id="hd">
                <h1>DataSource--Simple Array</h1>
            </div>
            <div id="bd">
            </div>
            <div id="ft">
            </div>
        </div>
        <script src="build/yahoo-dom-event/yahoo-dom-event.js"></script>
        <script src="build/datasource/datasource-min.js"></script>
        <script>
            (function () {
                var simple = ["red", "orange", "yellow", "green", "blue"];
                var ds = new YAHOO.util.LocalDataSource(simple);
                ds.responseType = YAHOO.util.LocalDataSource.TYPE_JSARRAY;
                ds.responseSchema = {fields: ["color"]};

                function out(target, msg) {
                    target = YAHOO.util.Dom.get(target);
                    if (target) {
                        target.innerHTML += "<p>" + msg + "</p>";
                    }
                };

                function dataLoadedCallback(request, payload) {
                    var tId = payload.tId;
                    var results = payload.results;
                    var meta = payload.meta;

                    var msg = "request: " + request;
                    msg += "<br \/>tId: " + tId;
```

```
                msg += "<br \/>results: " + YAHOO.lang.dump(results);
                msg += "<br \/>meta: " + YAHOO.lang.dump(meta);
                out("bd", msg);
            };

            // Get simple array
            ds.sendRequest(null, dataLoadedCallback);

            // Add a new color and get array again
            out("bd", "Adding indigo");
            simple.push("indigo");
            ds.sendRequest(null, dataLoadedCallback);

            // Add a new color and get array again
            out("bd", "Adding violet");
            simple.push("violet");
            ds.sendRequest(null, dataLoadedCallback);
        })();
    </script>
</body>
</html>
```

这个代码清单的核心功能实际上只有几行代码。首先，创建一个简单的数组，其中包含 5 种不同颜色的名称。该数组名为 simple。接下来，使用该数组作为 LocalDataSource 对象实例化的唯一参数。这会让该对象绑定到该数组。最后，定义一个响应模式，目的是为了告诉 DataSource 对象在它接收的有效负载中的何处查找所需数据。响应模式之所以使 YUI DataSource 功能如此多样，是因为它允许接收到的数据能够独立于最终使用者的需求来结构化。响应模式告诉 DataSource 在有效负载中的何处查找期望的数据，然后将标识名映射到期望的数据，从而让接收数据的窗口部件能够非常容易地访问它。在这里，数据存储到一个简单的数组中，因此映射非常简单。名称“color”分配给在该数组中找到的值。换言之，从该数组中检索的数据将像下面这样返回：

```
    {
        color: "red"
    },
    {
        color: "orange"
    },
    {
        color: "yellow"
    },
    {
        color: "green"
    },
    {
        color: "blue"
    }
]
```

这个数组中的每个条目都会转换成一个对象，在其中是一个名/值对。值就是在数组中找到的项，而名称就是响应模式中指定的名称。

对于稍微复杂一些的数据集(这次是两种语言的颜色集)，响应模式也变得稍微复杂一些，但并不是十分复杂。采用下面的数据集：

```
var simple = [
    "red", "rouge",
    "orange", "orange",
    "yellow", "jaune",
    "green", "vert",
    "blue", "bleue"
];
```

在这里，轮流采用英语和法语表示颜色。它们的位置代表了采用的语言，因此在这里位置非常重要。响应模式的名称位置必须匹配数组中找到的值所在的位置：

```
ds.responseSchema = {fields: ["color", "couleur"]};
```

这样，DataSource 就知道将“color”分配给偶数位置的值，而把“couleur”分配给奇数位置的值，从而产生下面的数据集：

```
[
    {
        couleur => red,
        color => red
    },
    {
        couleur => rouge,
        color => rouge
    },
    {
        couleur => orange,
        color => orange
    },
    {
        couleur => orange,
        color => orange
    },
    {
        couleur => yellow,
        color => yellow
    },
    {
        couleur => jaune,
        color => jaune
    },
    {
        couleur => green,
```

```
        color => green
    },
    {
        couleur => vert,
        color => vert
    },
    {
        couleur => blue,
        color => blue
    },
    {
        couleur => bleue,
        color => bleue
    }
]
```

访问该数据相当简单。首先，可以简单地实例化 DataSource 并传给另一个 YUI 窗口部件，让它自己负责访问数据：

```
// Set up a DataSource
var ds = new YAHOO.util.LocalDataSource(simple);
ds.responseSchema = {fields : ["color"]};

// Set up an AutoComplete
var ac = new YAHOO.widget.AutoComplete("myInput", "myContainer", ds);
...
```

但如果该数据并不是提供给 YUI 窗口部件，那么与 DataSource 交互就相当简单。与 DataSource 对象的交互是异步完成的，因此总是需要回调函数，前面的代码清单已经演示了这一点。下面是相同代码的裁剪版本：

```
(function () {
    var simple = ["red", "orange", "yellow", "green", "blue"];
    var ds = new YAHOO.util.LocalDataSource(simple);
    ds.responseType = YAHOO.util.LocalDataSource.TYPE_JSARRAY;
    ds.responseSchema = {fields: ["color"]};

    ...

    function dataLoadedCallback(request, payload) {
        ...
    };

    // Get simple array
    ds.sendRequest(null, dataLoadedCallback);

    ...
})();
```

这样一来，无论数据源的类型是什么，都会查询实时数据源，一旦数据准备就绪，就执行回调函数。

下面的示例演示如何使用DataSource检索JSON数据。注意，为了允许DataSource对象远程调用数据，现在需要connection-min.js和json-min.js这两个文件。

```
<html>
    <head>
        <title>DataSource--XHR</title>
        <link rel="stylesheet" type="text/css" href="reset-fonts-grids.css" />
        <link rel="stylesheet" type="text/css" href="base-min.css" />
    </head>
    <body>
        <div id="doc">
            <div id="hd">
                <h1>DataSource--XHR</h1>
            </div>
            <div id="bd">
                <h2>Employees</h2>
                <ul id="employees"></ul>
            </div>
            <div id="ft">
            </div>
        </div>
        <script src="build/yahoo-dom-event/yahoo-dom-event.js"></script>
<script src="build/datasource/datasource-min.js"></script>
<script src="build/connection/connection-min.js"></script>
<script src="build/json/json-min.js"></script>
<script>
    (function () {
        var ds = new YAHOO.util.XHRDataSource("employees.json?");
        ds.responseType = YAHOO.util.DataSource.TYPE_JSON;
        ds.connXhrMode = "queueRequests";
        ds.responseSchema = {
            resultsList: "resultset.results",
            fields: ["fname", "lname", "title"]
        };

        function out(target, msg) {
            target = YAHOO.util.Dom.get(target);
            if (target) {
                target.innerHTML += "<li>" + msg + "</li>";
            }
        };

        function callback(request, response) {
            for (var i = 0; response.results[i]; i += 1) {
                var employee = response.results[i];
                out("employees", employee.fname + " " + employee.lname +
```

```
                        " (" + employee.title + ")");
                }
            };

            var callbackObj = {
                failure: callback,
                success: callback
            };

            ds.sendRequest("count=all", callbackObj);
        })();
    </script>
</body>
</html>
```

下面给出了 employees.json 文件中的数据：

```
{"resultset":{
    "results":[
        {
            "fname": "John",
            "lname": "Doe",
            "title": "CEO"
        },
        {
            "fname": "Jane",
            "lname": "Doe",
            "title": "CFO"
        },
        {
            "fname": "Jack",
            "lname": "Smith",
            "title": "CIO"
        },
        {
            "fname": "Jen",
            "lname": "Smith",
            "title": "CTO"
        }
    ]}
}
```

这个示例在很大程度上与前一个示例相同，但在这里定义了 resultsList，用于告诉 DataSource 对象在它接收到的 JSON 数据中的什么地方查找结果。此外，因为这个数据是一个 JSON 对象而不是数组，所以在响应模式中找到的值的顺序不必匹配数据的顺序。换言之，除了定义为“fname”、“lname”、“title”，还可以轻易地定义为“lname”、“title”、“fname”，所有代码仍然能够运行。一旦获取数据，就会调用回调函数(在这里遵循 YUI Connection 对象的模式，分别针对调用结果 failure 或 success 定义回调函数)。出于简洁性考虑，在这个示例中，它们指向同一个回调函数。一旦接

收到数据，那么遍历该数据并将其输出到屏幕上就会非常简单(参见图 14-3)。

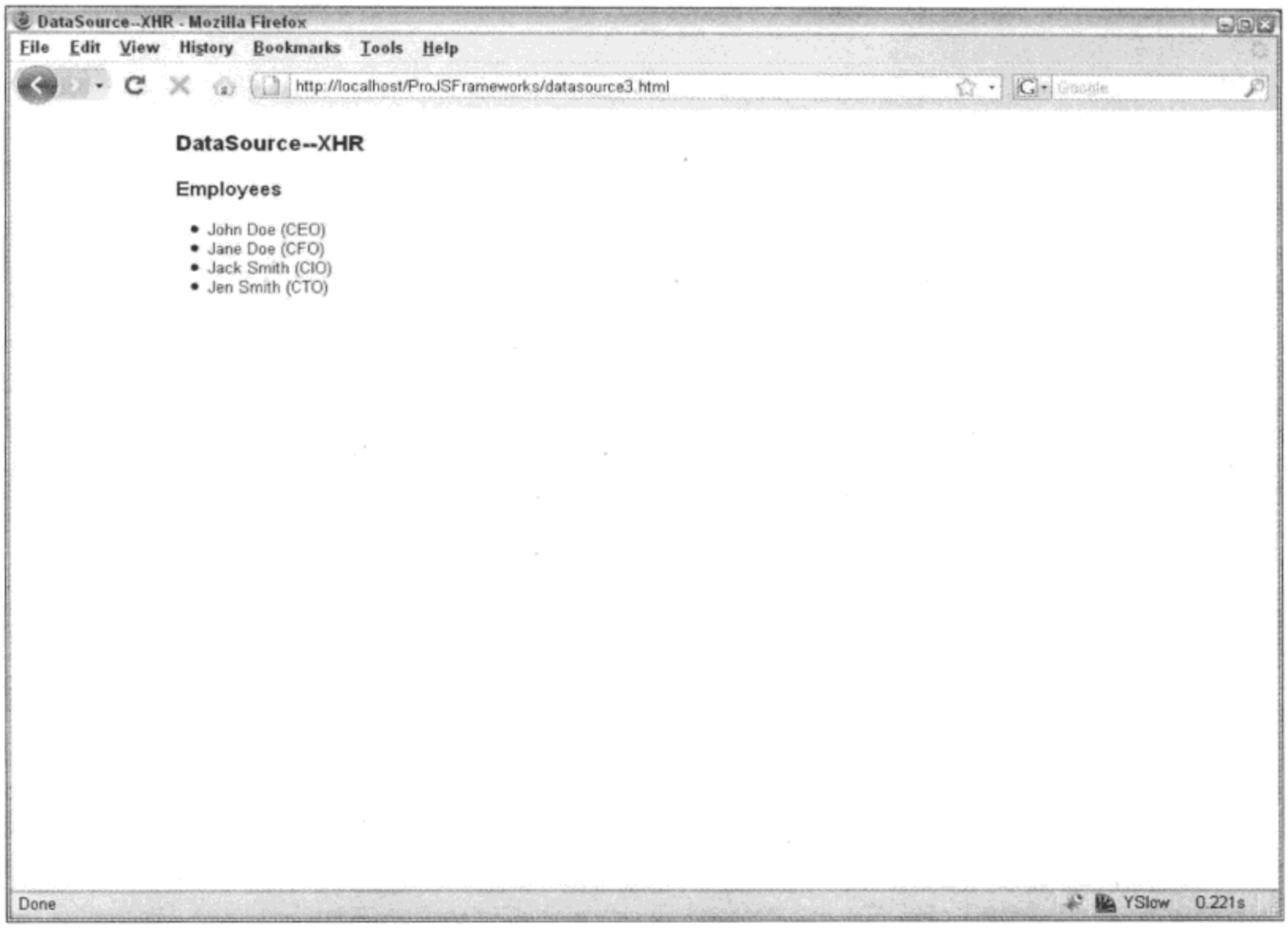


图　14-3

14.3　呈现表数据

使用 HTML 标记构建表是一件非常简单的事情。甚至于有很长一段时期，人们在 Web 上所做的事情就是构建表，即使在并不是显示表数据的场合中也是如此。然而，简单的 HTML 表在当今的 Web 中不再被推崇。在把数据集呈现给网站访问者时，他们现在已经期望一定的丰富程度和交互性。有时候采用数据可视化技术(本章稍后的图表部分将更深入讨论这方面的内容)，而有时候呈现原始数据。对于后者，让访问者能够与他们面前的数据进行交互(这可能意味着对数据进行排序或者过滤，甚至实时更新它)是一件有意义的事情。YUI 的 DataTable 可用来实现这些功能以及其他更多功能。

DataTable 的一个优秀功能是我们能够通过 DataSource 对象将其插入到数据源中，并将原始数据转换成丰富的、可交互的数据表(参见图 14-4)。

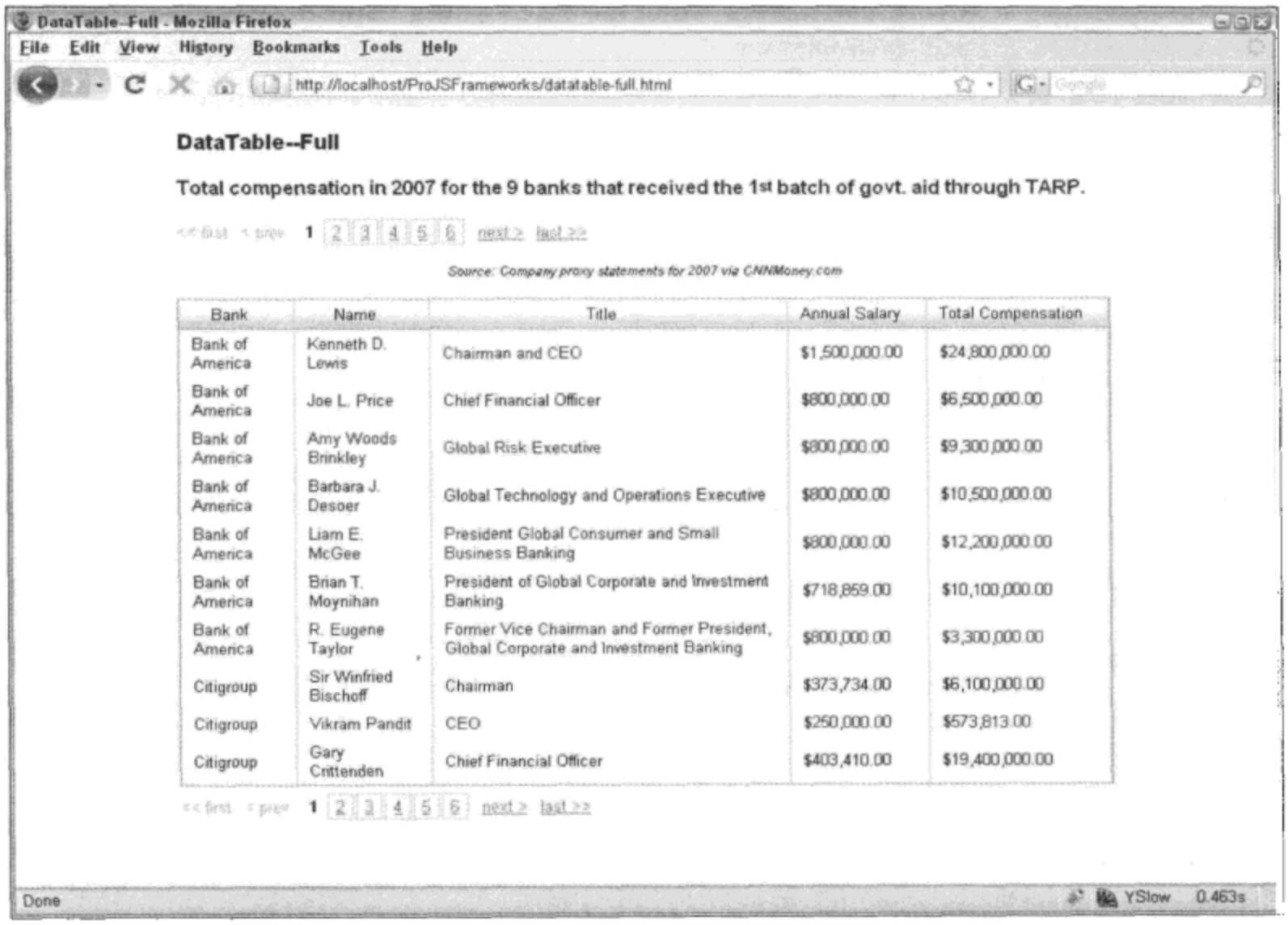


图 14-4

下面是具体的实现代码：

```
<html>
    <head>
        <title>DataTable--Full</title>
        <link rel="stylesheet" type="text/css"
            href="reset-fonts-grids.css" />
        <link rel="stylesheet" type="text/css"
            href="base-min.css" />
        <link rel="stylesheet" type="text/css"
            href="build/paginator/assets/skins/sam/paginator.css" />
        <link rel="stylesheet" type="text/css"
            href="build/datatable/assets/skins/sam/datatable.css" />
    </head>
    <body class="yui-skin-sam">
        <div id="doc">
            <div id="hd">
            </div>
            <div id="bd">
                <h1>DataTable--Full</h1>
                <h2>
                    Total compensation in 2007 for the 9 banks
```

```
            that received the 1<sup>st</sup> batch of govt.
            aid through TARP.
        </h2>
        <div id="data">
        </div>
    </div>
    <div id="ft">
    </div>
</div>
<!-- Required -->
<script type="text/javascript"
    src="build/yahoo-dom-event/yahoo-dom-event.js"></script>
<script type="text/javascript"
    src="build/element/element-min.js"></script>
<script type="text/javascript"
    src="build/datasource/datasource-min.js"></script>

<!-- Optional -->
<script type="text/javascript"
    src="build/paginator/paginator-min.js"></script>
<script type="text/javascript"
    src="build/json/json-min.js"></script>
<script type="text/javascript"
    src="build/connection/connection-min.js"></script>
<script type="text/javascript"
    src="build/get/get-min.js"></script>
<script type="text/javascript"
    src="build/dragdrop/dragdrop-min.js"></script>

<!-- Required -->
<script type="text/javascript"
    src="build/datatable/datatable-min.js"></script>

<script>
    (function () {
        /*
         * Sort function for currency values
         */
        function totalSort(a, b, desc) {
            return moneySort("total", a, b, desc);
        };

        function salarySort(a, b, desc) {
            return moneySort("salary", a, b, desc);
        };

        function moneySort(val, a, b, desc) {
            a = a.getData()[val];
            b = b.getData()[val];
```

```
    if (desc) {
        return b - a;
    } else {
        return a - b;
    }
};

/*
 * Instantiate data source
 */
var dataSource = new YAHOO.util.XHRDataSource("bonuses.json?");
dataSource.responseType = YAHOO.util.DataSource.TYPE_JSON;

dataSource.connXhrMode = "queueRequests"; /* handles requests
                                             synchronously */
dataSource.responseSchema = {
    resultsList: "resultset.results",
    fields: ["bank", "name", "title", "salary", "total"]
};

/*
 * Define columns
 */
var columnDefs = [
    {key: "bank", label: "Bank", sortable:true, resizeable:true},
    {key: "name", label: "Name", sortable:true, resizeable:true},
    {key: "title", label: "Title", sortable:true, resizeable:true},
    {key: "salary", label: "Annual Salary", sortable:true,
        resizeable:true,
        formatter:YAHOO.widget.DataTable.formatCurrency,
        sortOptions:{sortFunction:salarySort}},
    {key: "total", label: "Total Compensation", sortable:true,
        resizeable:true,
        formatter:YAHOO.widget.DataTable.formatCurrency,
        sortOptions:{sortFunction:totalSort}}
];

/*
 * Instantiate data table
 */
var config = {

        caption: "Source: Company proxy statements for 2007 via " +
                "CNNMoney.com",
        paginator: new YAHOO.widget.Paginator({
            rowsPerPage: 10
        }),
        draggableColumns:true
};
```

```
                var dtable = new YAHOO.widget.DataTable("data", columnDefs,
                        dataSource, config);
            })();
        </script>
    </body>
</html>
```

在将所有必要的 CSS 文件和脚本文件包含进来之后，将数据源绑定到数据表的过程就相当简单。在这里，为了演示该窗口部件的功能，这个 DataTable 做了大量不同的工作。

DataTable 做的第一件事情就是创建一个自定义排序函数。这样做的目的在于确保把带有货币信息的列按照数字而不是字符串排序。换言之，它是为了确保不会把 15、4、1、7 这样的一组值最终排列成 1、15、4、7，而不是正确的 1、4、7、15。因为无法知道哪一列调用这个排序算法，所以为了让算法知道要对哪些数据排序，必须为每一列创建一个唯一的排序函数。接着，这些唯一的排序函数又根据进行排序的列的名称来调用主排序函数。

接下来，实例化一个 DataSource 对象，将其指向 bonuses.json 文件。这个文件包含类似于如下的数据：

```
{"resultset":{
    "results":[
        {
            "bank": "Bank of America",
            "name": "Kenneth D. Lewis",
            "title": "Chairman and CEO",
            "salary": 1500000,
            "total": 24800000
        },
        {
            "bank": "Bank of America",
            "name": "Joe L. Price",
            "title": "Chief Financial Officer",
            "salary": 800000,
            "total": 6500000
        },
        ...
    ]}
}
```

既然已经创建了数据源，那么剩下的工作就是实例化一个 DataTable 对象，并将两者连接起来。没有以内联方式完成 DataTable 对象的所有配置，而是使用了几个变量，即 columnDefs 和 config。该练习的最终结果是一个可排序、带有分页功能、可调整列宽度且可拖动列的数据表。

14.4　绘制图表和图形

利用表呈现数据固然很好，但是没有什么方式能够实现像视觉呈现那样直接的反应。将数据

转换成图表曾经是属于电子表格软件的领域，现在情况已经改变。YUI 的 Charts 控件可以插入到 DataSource 中，将它的数据转换成可视的、极具吸引力的图表和图形。Charts 控件能够显示线形图、条形图、柱形图、饼图、堆叠条形图以及堆叠柱形图。通过选择下面的构造函数就可以选择这些图表：

- YAHOO.widget.LineChart
- YAHOO.widget.BarChart
- YAHOO.widget.ColumnChart
- YAHOO.widget.PieChart
- YAHOO.widget.StackedBarChart
- YAHOO.widget.StackedColumnChart

以下面的部分数据集为例：

```
{"resultset":{
    "results":[
        {
            "bank": "Merrill Lynch",
            "name": "E. Stanley O'Neal",
            "title": "Former Chief Executive Officer",
            "salary": 584231,
            "total": 24300000
        },
        {
            "bank": "Merrill Lynch",
            "name": "Ahmass L. Fakahany",
            "title": "Former Co-President and Co-Chief Operating Officer",
            "salary": 350000,
            "total": 4600000
        },
        {
            "bank": "Merrill Lynch",
            "name": "Dow Kim",
            "title": "Former Executive Vice President",
            "salary": 309615,
            "total": 14500000
        },
        ...
    ]}
}
```

使用下面的代码可以将这个数据集转换成图表(参见图 14-5)：

```
<html>
    <head>
        <title>Charts</title>
        <link rel="stylesheet" type="text/css" href="reset-fonts-grids.css" />
```

```
    <link rel="stylesheet" type="text/css" href="base-min.css" />
    <style type="text/css">
        #chart {
            height: 450px;
            width: 750px;
        }
    </style>
</head>
<body class="yui-skin-sam">
    <div id="doc">
        <div id="hd">
        </div>
        <div id="bd">
            <h1>Charts</h1>
            <h2>
                Total compensation in 2007 for the 9 banks
                that received the 1<sup>st</sup> batch of govt.
                aid through TARP. (partial list)
            </h2>
            <div id="chart">
            </div>
        </div>
        <div id="ft">
        </div>
</div>
<!-- Required -->

<script type="text/javascript"
       src="build/yahoo-dom-event/yahoo-dom-event.js"></script>
<script type="text/javascript"
       src="build/element/element-min.js"></script>
<script type="text/javascript"
       src="build/datasource/datasource-min.js"></script>
<script type="text/javascript"
       src="build/json/json-min.js"></script>

<!-- Optional -->

<script type="text/javascript"
       src="build/connection/connection-min.js"></script>

<!-- Required -->
<script type="text/javascript" src="build/charts/charts-min.js"></script>

<script>
    (function () {
        /*
         * Instantiate data source
```

```
 */

var dataSource = new YAHOO.util.XHRDataSource(
       "bonuses-short.json?");
dataSource.responseType = YAHOO.util.DataSource.TYPE_JSON;

dataSource.connXhrMode = "queueRequests"; /* handles requests
                                          synchronously */
dataSource.responseSchema = {
    resultsList: "resultset.results",
    fields: ["bank", "name", "title", "salary", "total"]
};

/*
 * Instantiage chart
 */
function formatCurrencyAxisLabel( value )
{
    return YAHOO.util.Number.format( value,
    {
        prefix: "$",
        thousandsSeparator: ",",
        decimalPlaces: 2
    });
}
            var currencyAxis = new YAHOO.widget.NumericAxis();
            currencyAxis.labelFunction = formatCurrencyAxisLabel;

            var seriesDef =
            [
                 { displayName: "Salary", yField: "salary" },
                 { displayName: "Total Compensation", yField: "total" }
            ];


            var chart = new YAHOO.widget.StackedColumnChart( "chart",
            dataSource,
            {
                xField: "name",
                yField: "total",
                yAxis: currencyAxis,
                series: seriesDef
            });
        })();
    </script>
</body>
</html>
```

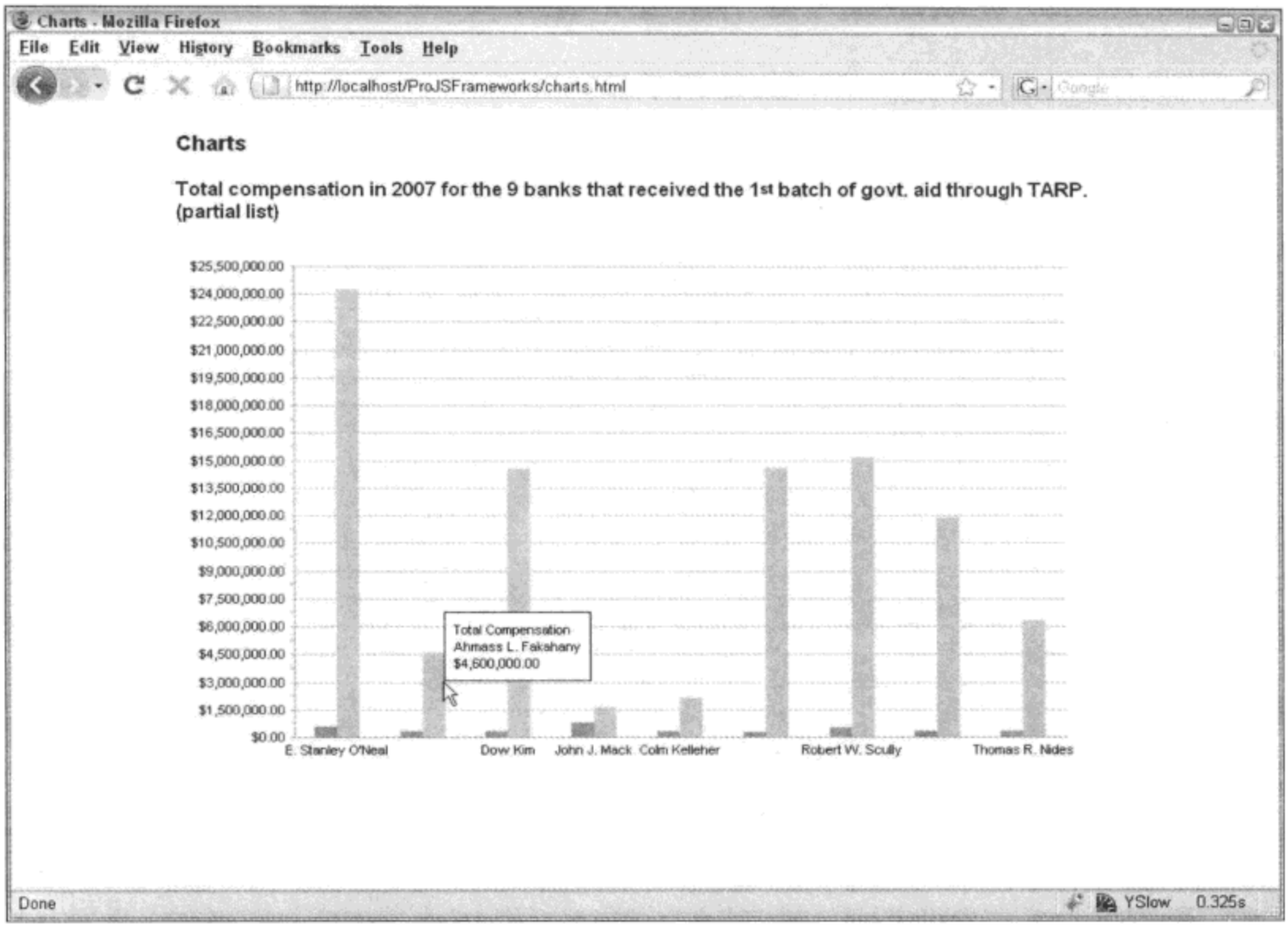


图　14-5

一旦所需的文件就位，实例化一个数据源并将其绑定到图表对象就是一件非常简单的事情。这里进行了一些定制工作，通过 NumericAxis 对象的 labelFunction 属性修改了 y 轴标签，以便将货币格式应用于这些标签。此外，要想沿着坐标轴显示基于时间的值，还可以创建一个 TimeAxis 对象。

通过配置参数将一个 style 对象传给构造函数以自定义图表对象，该参数采用如下形式：

```
var seriesDef =
[
    {
        displayName: "Salary",
        yField: "salary",
        style: {
            color: 0xff0000
        }
    },
    {
        displayName: "Total Compensation",
        yField: "total",
        style: {
            color: 0x00ff00
        }
```

```
        }
    ];
```

style 对象中的新值将把 Salary 条形变成红色，将 Total Compensation 条形变成绿色(0xff0000 是红色的十六进制值，而 0x00ff00 是绿色的十六进制值)。

表 14-2 列出了所有可用的样式化属性。

表　14-2

样　式	子 样 式	说　明
padding		一个用来指定围绕图表内容边缘的间距的数字值。与 HTML 中的 CSS 内边距不同，图表的内边距不会增加图表的尺寸
animationEnabled		一个用来指定是否启用标记符动画的 Boolean 值。默认值为 true，这意味着当数据改变时标记符将以动画方式显示
font		可以声明一种字体样式来定制默认的坐标轴文本，包括字体名称、大小、颜色以及其他样式。使用一个含有几个子样式的 Object 值表示该样式
	name	该样式可以接受一个字符串，该字符串既可以是字体的名称，也可以是由逗号分隔的字体名称列表，这与 CSS 中的 font-family 的工作方式类似
	color	类似 0xff0000 的十六进制值
	size	该样式可以接受一个数字值作为字体磅值。没有其他字体大小单位可用
	bold	如果要以粗体显示字体，就设置这个 Boolean 值
	italic	如果要以斜体显示字体，就设置这个 Boolean 值
	underline	如果要以下划线显示字体，就设置这个 Boolean 值
border		开发人员可以利用 border 样式在图表周围添加彩色的边框。图表本身的尺寸会缩小，以便容纳边框。使用一个含有几个子样式的 Object 值表示该样式
	color	一个十六进制格式化字符串或数字值，例如"ff0000"或 0xff0000
	size	边框的厚度，单位为像素
background		background 样式可用来定制背景色或图像。使用一个含有几个子样式的 Object 值表示该样式
	color	指定背景填充色。如果有一幅图像，那么这个填充色将出现在该图像的后面。该样式接受一个十六进制格式化字符串或数字值，例如"ff0000"或 0xff0000
	alpha	一个范围是 0.0~1.0 的值，表示背景色的透明度。当用于数据提示背景上时，这个样式非常有用
	image	JPG、PNG、GIF 或 SWF 图像文件的 URL，可以是相对 URL 或绝对 URL。相对 URL 相对于嵌入该图表的 HTML 文档
	mode	用来显示背景图像的方法。值可以是"repeat"(默认值)、"repeat-x"、"repeat-y"、"no-repeat"或"stretch"
legend		legend 样式可让开发人员定制图例的外观。使用一个含有几个子样式的 Object 值表示该样式
	display	指定在哪个位置绘制图例。可接受的值包括"none"、"left"、"right"、"top"和"bottom"。默认值为"none"

(续表)

样　式	子 样 式	说　明
legend	spacing	指定在图例中所示的各项之间的像素数
	padding	与前面描述的 padding 样式相同
	border	与前面描述的 border 样式相同
	background	与前面描述的 background 样式相同
	font	与前面描述的 font 样式相同
dataTip		dataTip 样式可让开发人员定制数据提示的外观。使用一个含有几个子样式的 Object 值表示该样式
	padding	与前面描述的 padding 样式相同
	border	与前面描述的 border 样式相同
	background	与前面描述的 background 样式相同
	font	与前面描述的 font 样式相同
xAxis 和 yAxis		xAxis 和 yAxis 样式可用来定制坐标轴的外观。使用含有几个子样式的 Object 值表示这些样式
	color	坐标轴本身的颜色。它可接受十六进制格式化的字符串或数字值，例如 "ff0000"或 0xff0000
	size	表示坐标轴本身厚度的数字值。0 值表示隐藏坐标轴(但不隐藏标签)
	showLabels	如果为 true，就显示标签。如果为 false，则将标签隐藏
	hideOverlappingLabels	指示是否隐藏重叠的标签。当 calculateCategoryCount 为 false 时，这个样式用在 Category Axis 上。当用户指定 majorUnit 时，这个样式用在 TimeAxis 和 NumericAxis 上。否则，坐标轴会放置标签，不让它们彼此重叠
	labelSpacing	坐标轴上标签之间的距离(单位为像素)。默认值为 2
	labelDistance	标签与坐标轴之间的距离(单位为像素)。默认值为 2
	titleRotation	指示是否沿着坐标轴旋转标题
	titleDistance	标题与坐标轴标签之间的距离(单位为像素)。默认值为 2
	majorGridLines	参见后面的描述
	minorGridLines	参见后面的描述
	zeroGridLine	参见后面的描述
	majorTicks	参见后面的描述
	minorTicks	参见后面的描述
majorGridLines 和 minorGridLines		majorGridLines 和 minorGridLines 样式有几个子样式需要额外的解释。前面已经介绍过，majorGridLines 和 minorGridLines 属于 xAxis 和 yAxis 样式的子样式
	color	网格线的颜色。它可接受十六进制格式化的字符串或数字值，例如 "ff0000"或 0xff0000

(续表)

样　式	子 样 式	说　明
majorGridLines 和 minorGridLines	size	一个表示网格线厚度的数字值。要想隐藏网格线，可以将 size 子样式设为 0(零)。如果默认情况下隐藏了网格线，就必须将其厚度设为大于 0 的值以显示它们
zeroGridLine		zeroGridLine 样式用于当零网格线位于坐标轴原点下方时突出显示它。zeroGridLine 样式具有如下子样式
	color	零网格线的颜色。它可接受十六进制格式化的字符串或数字值，例如"ff0000"或 0xff0000
	size	一个表示零网格线厚度的数字值。要想隐藏该网格线，可以将 size 子样式设为 0
majorTicks 和 minorTicks		majorTicks 和 minorTicks 样式有几个需要额外解释的子样式。前面已经介绍过，majorTicks 和 minorTicks 属于 xAxis 和 yAxis 样式的子样式
	color	刻度的颜色。它可接受十六进制格式化的字符串或数字值，例如"ff0000"或 0xff0000
	size	一个表示刻度厚度的数字值。如果默认情况下没有显示刻度，就需要把这个样式设置为一个大于 0 的有效数字值
	length	刻度从坐标轴上延伸出来的长度，单位为像素。如果默认情况下没有显示刻度，就需要把这个样式设置为一个大于 0 的有效数字值
	display	指定如何绘制刻度。可接受的值包括"none"、"inside"、"outside"和"cross"。在许多情况下，默认值为"none"

该表取自 YUI 网站中的 Charts 页面

14.5　本章小结

使用 YUI 的 DataSource 组件(能够轻易地插入到其他 YUI 窗口部件中)处理数据是一件相当简单的事情。DataSource 还使得开发自定义数据驱动窗口部件变得非常简单，这是因为它将数据抽象成易于管理的对象。此外，利用 DataTable 和 Charts 组件，DataSource 可以轻易实现数据可视化。在没有出现这些组件之前，实现这些功能非常困难。图 14-2 基于 YUI DataSource 页面中的图。

第15章

使用 YUI CSS 工具

本章内容简介：

- 建立跨浏览器一致性
- 控制字体
- 利用网格构建布局

15.1 建立跨浏览器一致性

每种浏览器制造商都构建了自己的布局算法，虽然这些算法非常接近，但并不完全相同。因此，在不同的浏览器上，页面中对象之间的距离甚至字体间距调整都可能稍有不同。实际上，对于浏览器目前的状态，不可能在不同的浏览器上实现完全相同的呈现效果。

以下面的 HTML 标记为例：

```
<h1>Base Render</h1>
<p>
    Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec lectus.
    Curabitur malesuada purus vitae tellus. Quisque feugiat volutpat enim.
    Donec tempor mauris et nunc.
</p>
<ul>
    <li>
        Etiam nunc turpis, placerat a, aliquam non, vestibulum in, odio.
    </li>
    <li>
        Nulla scelerisque, nisl in tincidunt iaculis, quam ante malesuada
        purus, ac interdum tortor elit sed sapien.
    </li>
    <li>
        Ut auctor, diam at bibendum accumsan, felis nisi porttitor enim,
        sed feugiat nibh mi sagittis dolor.
```

```
    </li>
</ul>
<h2>Nulla fringilla turpis ac nibh.</h2>
<blockquote>
    <p>
        Vivamus tempus turpis adipiscing nibh. Ut nec orci. Etiam vitae
        ante nec nunc ornare tincidunt. Ut tortor nunc, adipiscing vel,
        semper at, tincidunt et, lectus.
    </p>
</blockquote>
<table>
    <thead>
        <tr>
            <th>Donec</th>
            <th>non orci</th>
        </tr>
    </thead>
    <tbody>
        <tr>
            <td>
                ut sem dapibus mollis. Donec nunc ipsum, pellentesque
                consectetur, congue non, faucibus bibendum, lectus.
            </td>
            <td>
                In hac habitasse platea dictumst. Sed fringilla.
                Quisque tristique leo eu risus.
            </td>
        </tr>
    </tbody>
</table>
```

图 15-1 并排展示了这段标记在 IE7、Firefox 3 和 Chrome 中的呈现结果(为了展示区别，将结果重叠在一起)。

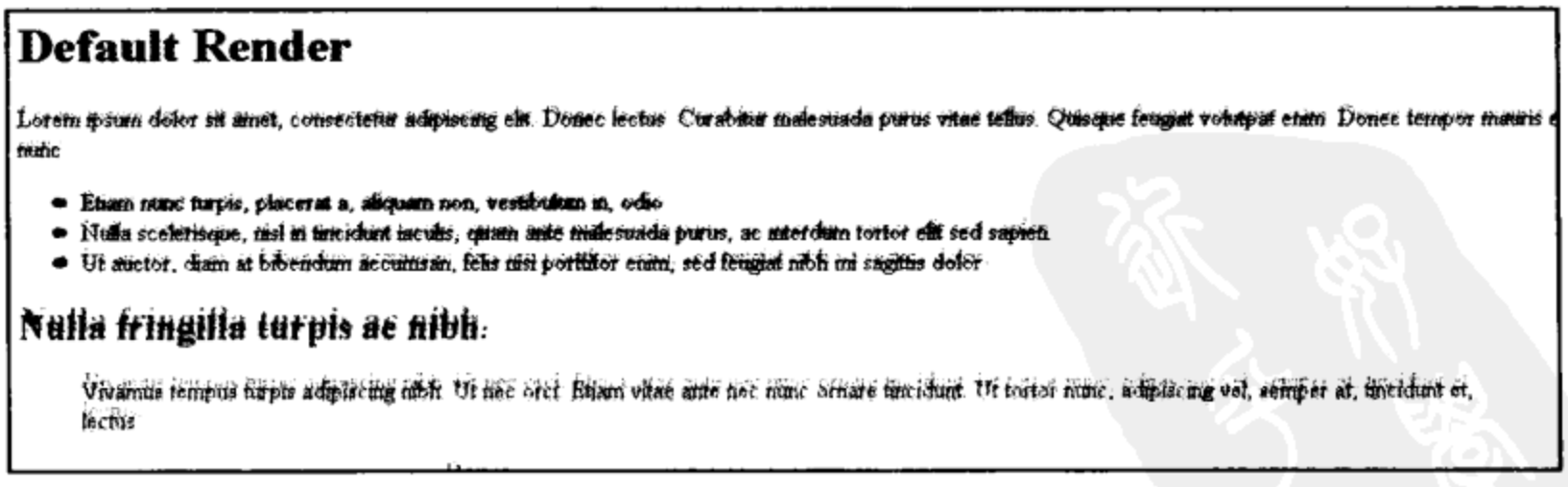


图 15-1

YUI 的 CSS 文件试图尽可能地统一呈现引擎，首先就是使用 reset.css 文件。这个文件将浏览器的默认呈现值去除，从而所有元素的所有字体大小变得相同，所有的外边距和内边距值都设为

0 等。尽管结果不是很完美(参见图 15-2)，但它确实建立了一个统一的基础，在此基础之上可以设置一个默认值。如果没有其他设置，那么它将确保开发人员不会认为默认值理所当然。换言之，如果一个列表项需要项目符号，而且要有 1em 的左内边距，那么就要由开发人员指定该值，而不是假设所有浏览器都是同样的设置。

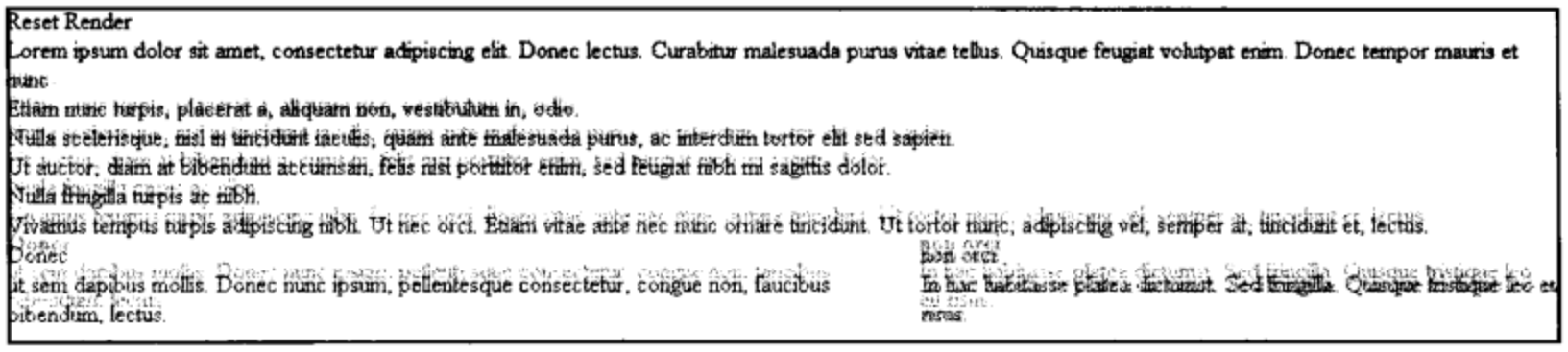


图　15-2

然而，在图 15-2 中非常明显的是，即使将大小和定位值的默认呈现去除，不同浏览器的输出结果仍然不能做到完全一致。但是，向重置的呈现分配一致的新外边距、内边距和字体大小值确实可以真正地让所有浏览器产生更加类似的呈现效果(参见图 15-3)。注意，分配给图 15-3 的字体值实际上来自于 fonts.css 文件(后面将讨论)。

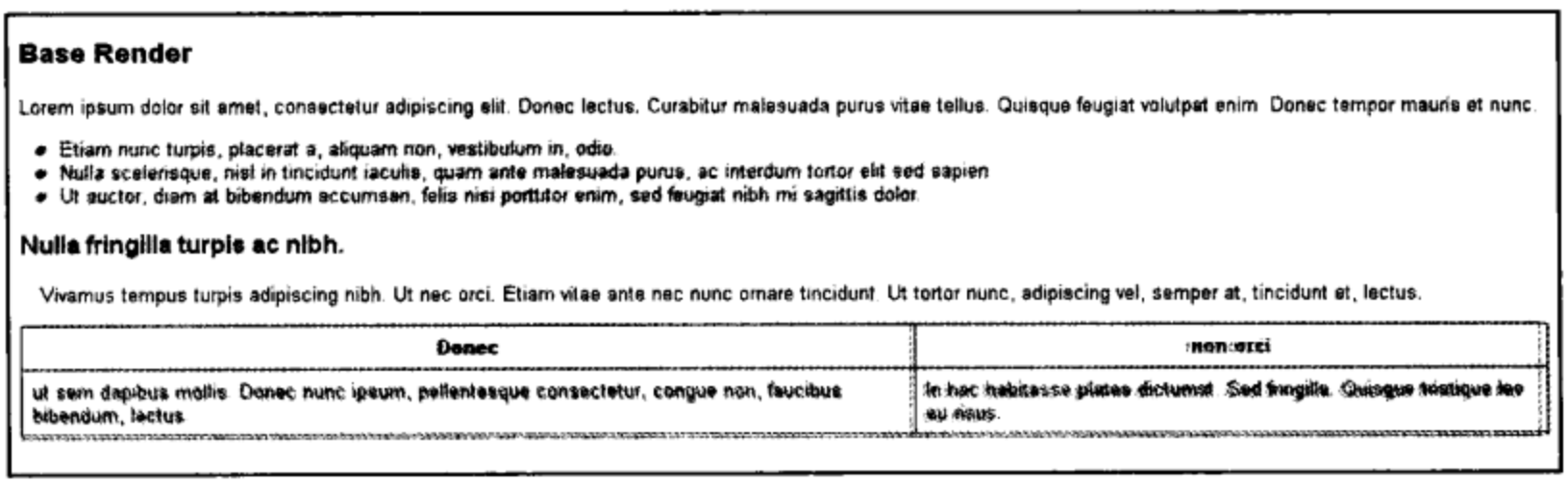


图　15-3

尽管 base.css 文件能够很好地校正样式，但是并不真正推荐将其用于生产环境，这是因为大多数生产网站都有特定的布局需要，这就要求使用特定的 CSS 规则。事实上，使用 reset.css 和 fonts.css(下面会讨论)将所有样式重置到一个基准，然后再向页面添加大量的规则才能实现有用的状态。此外，使用网站的特定 CSS 重写 base.css 并没有多大意义。

15.2　控制字体

字体一直是 Web 出版的弱项。浏览器依赖于存放在本地的字体满足呈现需求。换言之，不能将字体文件随着 HTML、CSS 和 JavaScript 文件一起传送给浏览器。这样，页面的呈现效果就取

决于它在什么平台上呈现。更糟糕的是，在处理字体间距调整和其他计算时，不同的浏览器处理同一种字体的方式也不同。

为了缓解这个问题，YUI 提供了一个 fonts.css 文件，它将不同浏览器之间的字体差异尽可能规范化。它将默认字体系列设为 Arial，并为常见的字体系列建立降级路径(如果一种字体不可用，就请求它的已知的等价字体)。同时，它还将基准字体大小设为 13 像素，将行高设为 16 像素。

下面的字体系列将跨多个操作系统很好地降级：

```
#demo1 {}
#demo2 {font-family:monospace;}
#demo3 {font-family:georgia;}
#demo4 {font-family:verdana;}
#demo5 {font-family:times;}
```

注意：
这个示例来自于 YUI 字体页面。

将这种字体系列设置好之后，就可以使用下面的查找表(参见表 15-1)足够可靠地跨浏览器设置字体大小。

表 15-1

像　素	百　分　比
10	77
11	85
12	93
13	100
14	108
15	116
16	123.1
17	131
18	138.5
19	146.5
20	153.9
21	161.6
22	167
23	174
24	182
25	189
26	197

因此，为了实现 12 像素的字体大小，需要在 CSS 中设置的值是 93%。但是，考虑到在 CSS

中字体值具有继承关系，设置嵌套百分比将产生非预期的结果。因此，将容器元素的字体大小设置成93%，然后同样将子元素的字体大小设置成93%，那么最终将把子元素的字体大小设置成93%的93%，也就是11.2像素。因此，重点是明智地设置字体大小，确保绝对不会嵌套设置，除非是故意这么做。图15-4和图15-5给出了fonts.css文件如何将前面提及的3款浏览器的呈现效果统一起来。

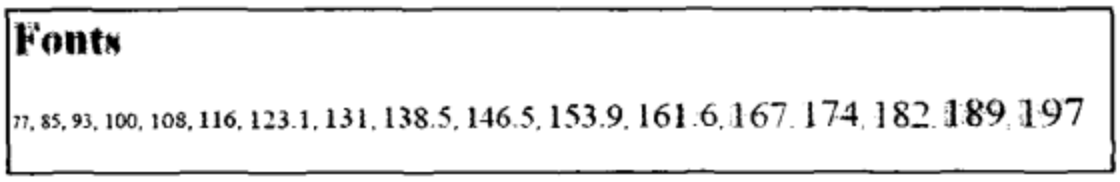


图　15-4

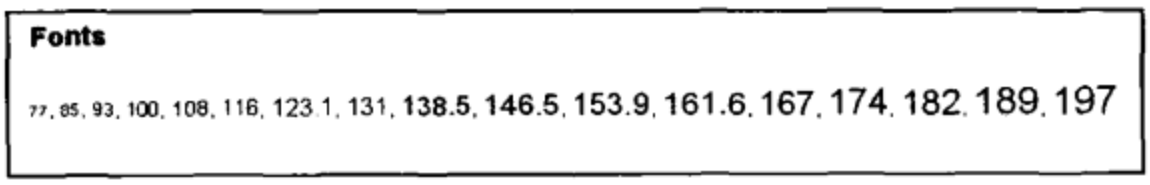


图　15-5

15.3　利用网格构建布局

布局一直是 Web 开发世界中产生摩擦的来源。关于是否使用表实现布局的争论已经尘埃落定，现在只剩下CSS可用于在页面上正确地放置内容。这应该算是好消息，但是浏览器有一个非常不好的传统，就是实现布局的方式彼此不同。如果Web开发人员希望只使用一种统一的布局，那么这就会带来问题。

YUI的grids.css文件能够以跨浏览器的、健壮的、灵活的方式来实现页面布局。网格系统提供的基本尺寸与美国互动广告局(Interactive Advertising Bureau，IAB)的Ad Unit Guidelines的常见广告尺寸相一致。换言之，可以选择与IAB标准广告尺寸相一致的“列宽”或“单元格宽度”(使用table语法)。YUI团队甚至提供了一个“网格生成器(Grid Builder)”工具(http://developer.yahoo.com/yui/grids/builder/)，只需要在窗口部件上单击几次就可以把全部标记(后面有具体的描述)整合在一起，如图15-6所示。

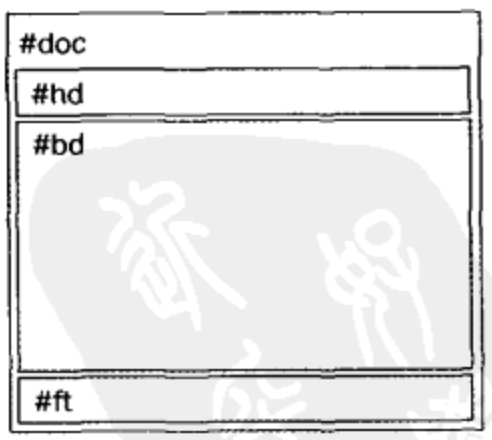


图　15-6

YUI网格有4种主要的风格，由4个根ID表示这些风格：

- **div#doc**　建立750像素的页面宽度。
- **div#doc2**　建立950像素的页面宽度。
- **div#doc3**　建立100%的页面宽度(注意，100%的页面宽度也会设有10像素的左右外边距，这样页面内容与浏览器边框之间就会有一些空间)。
- **div#doc4**　建立974像素的页面宽度，这是YUI 2.3.0版本中的网格的新增风格。

注意：
这个项目列表的内容直接取自 YUI(750 像素页面宽度)示例页面。

100%页面宽度布局的两侧均有 10 像素的外边距，这样就可以防止内容渗透到浏览器边框中。但是，可以使用下面的 CSS 规则重置外边距：

```
#doc3 {margin: auto}
```

定制网格布局的宽度是一件简单的事情。因为 YUI 的网格布局基于 em 值，而该值又与基准字体大小相联系，所以将想要的宽度转换成 em 值只需要少量简单的计算即可。宽度是按照 em 单位设置的，因为 em 会随着基准字体大小缩放。换言之，当用户调整字体大小时，整个网格也会随之作出响应。由于基准字体大小设为 13 像素，因此将像素宽度转换成 em 值时，只要将宽度除以 13 即可，至少对于所有非 IE 浏览器来说是如此。对于 IE 而言，必须将宽度除以 13.3333。下面的示例演示如何实现一个重写值：

```
<style>
#custom-doc {
      margin:auto;text-align:left; /* leave unchanged */
      width:46.15em;/* non-IE */
      *width:45.00em;/* IE */
      min-width:600px;/* optional but recommended */
}
</style>
```

注意：
这个示例直接取自 YUI Grids 页面。

在本节的示例中，我们已经给出了基于网格的文档的基本结构，如下所示：

```
<html>
    <head>
        <title></title>
        <link rel="stylesheet" type="text/css" href="reset-fonts-grids.css" />
    </head>
    <body>
        <div id="doc">
            <div id="hd">
                <!-- Header content goes here -->
            </div>
            <div id="bd">
                <!-- Body content goes here -->
            </div>
            <div id="ft">
                <!-- Footer content goes here -->
            </div>
        </div>
    </body>
</html>
```

首先，在文档的 head 标记中包括 grids.css 文件。然后创建带有想要的 doc ID 的 div 元素，在这里就是“doc”，这会产生 750 像素宽度的布局。在该元素内部有 3 个 div 元素，ID 分别为 hd、bd 和 ft，分别表示页眉、正文和页脚(参见图 15-6)。

15.3.1　模板

大多数网站都有列式布局，其中的一列表示主要内容，另一列表示次要内容。YUI Grids 有一组围绕这个概念构建的预制模板(尺寸基于 IAB 指导原则)。最初的两列有两个带有 CSS 类名 yui-b 的 div 元素，b 代表“block(块)”。为了实现源代码顺序无关性(换言之，主要内容或次要内容首先出现在源代码中，但在布局中并不是如此)，使用 ID 为 yui-main 的 div 元素将主要内容块包装起来。最后，为了触发 6 种预设模板中的一种，要把如下 CSS 类名(参见表 15-2)之一分配给根 div 元素。

表　15-2

模　板　类	预 设 说 明
yui-t1	左列宽度 160 像素
yui-t2	左列宽度 180 像素
yui-t3	左列宽度 300 像素
yui-t4	右列宽度 180 像素
yui-t5	右列宽度 240 像素
yui-t6	右列宽度 300 像素

下面是使用预设 yui-t4 实现基本模板布局的代码清单(参见图 15-7)：

```
<div id="doc" class="yui-t4">
    <div id="hd">
        <! — Header content goes here -->
    </div>
    <div id="bd">
       <div id="yui-main">
          <div class="yui-b">
             <!-- Primary content goes here -->
          </div>
       </div>
       <div class="yui-b">
           <!-- Secondary content goes here -->
        </div>
    </div>
    <div id="ft">
        <! — Footer content goes here -->
    </div>
</div>
```

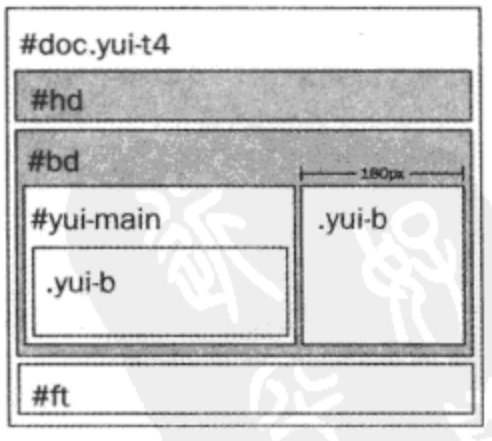


图　15-7

15.3.2 嵌套网格

可以使用 YUI 网格进一步将内容块划分成两部分。网格(在有限的程度上)等同于 HTML 表中的行元素。在网格中放置两个单元元素，它们等同于表的单元格。网格通过 CSS 类名 yui-g 来识别，而单元通过 CSS 类名 yui-u 来识别(参见图 15-8)。下面是一个网格中有两个单元的代码：

```
...
<div class="yui-g">
    <div class="yui-u first">
        <!-- Unit content here -->
    </div>
    <div class="yui-u">
        <!-- Unit content here -->
    </div>
</div>
...
```

实现 4 个单元的代码如下：

```
<div class="yui-g">
    <div class="yui-g first">
        <div class="yui-u first">
            <!-- Unit content here -->
        </div>
        <div class="yui-u">
            <!-- Unit content here -->
        </div>
    </div>
    <div class="yui-g">
        <div class="yui-u first">
            <!-- Unit content here -->
        </div>
        <div class="yui-u">
            <!-- Unit content here -->
        </div>
    </div>
</div>
```

注意，外层网格(yui-g)包含另外两个网格。因为每个网格只有两个单元，所以总共有 4 个单元。还请注意 CSS 类名 first。每当有两个或更多的 yui-u 或 yui-g 这样的一系列项时，CSS 类名 first 将告诉 YUI 特殊对待这一系列中的第一项(例如调整外边距等)。简而言之，CSS 类名 first 确保元素能够正确地布局。

网格单元(yui-u)将可用空间划分成相同的两部分。但在有些情况下，需要让其中一个单元更大一些，或者需要有 3 个单元。这就是特殊网格发挥作用的地方，参见表 15-3。

表　15-3

特殊网格类	说　　明
yui-gb	1/3-1/3-1/3
yui-gc	2/3-1/3
yui-gd	1/3-2/3
yui-ge	3/4-1/4
yui-gf	1/4-3/4

通过结合使用网格和特殊网格，我们可以构建非常复杂的布局。下面的代码就实现了这样的一个复杂布局(参见图 15-8)。

```
<html>
    <head>
        <title></title>
        <link rel="stylesheet" type="text/css" href="reset-fonts-grids.css" />
        <link rel="stylesheet" type="text/css" href="base-min.css" />
    </head>
    <body>
        <div id="doc" class="yui-t4">
            <div id="hd">
                <h1>Header</h1>
</div>
<div id="bd">
    <div id="yui-main">
        <h2>Primary Col</h2>
        <div class="yui-b">
            <div class="yui-g">
                <div class="yui-g first">
                    <div class="yui-u first">
                        <h3>Row 1, Col 1</h3>
                        <p>Lorem ipsum dolor sit amet, consectet...</p>
                    </div>
                    <div class="yui-u">
                        <h3>Row 1, Col 2</h3>
                        <p>Lorem ipsum dolor sit amet, consectet...</p>
                    </div>
                </div>
                <div class="yui-g">
                    <div class="yui-u first">
                        <h3>Row 1, Col 3</h3>
                        <p>Lorem ipsum dolor sit amet, consectet...</p>
                    </div>
                    <div class="yui-u">
                        <h3>Row 1, Col 4</h3>
                        <p>Lorem ipsum dolor sit amet, consectet...</p>
```

```
                    </div>
                </div>
            </div>
            <div class="yui-gc">
                <div class="yui-u first">
                    <h3>Row 2, Col 1</h3>
                     <p>Lorem ipsum dolor sit amet, consectet...</p>
                </div>
                <div class="yui-u">
                    <h3>Row 2, Col 2</h3>
                    <p>Lorem ipsum dolor sit amet, consectet...</p>
                </div>
            </div>
        </div>
    </div>
    <div class="yui-b">
        <h2>Secondary Col</h2>
        <p>Lorem ipsum dolor sit amet, consectetur adipiscing... </p>
    </div>
</div>
<div id="ft">
    <p>Footer</p>
</div>
</div>
</body>
</html>
```

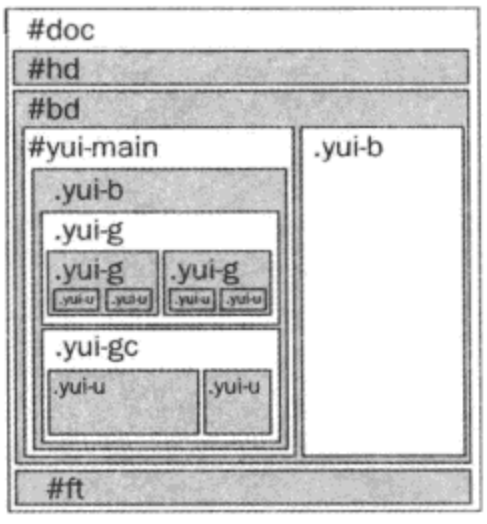


图 15-8

YUI 3 中的新增功能

除了出于清晰起见而将 CSS 文件夹和文件重命名以添加“css”前缀(它们现在名为 cssreset、cssfonts 和 cssgrids)之外，YUI 3 CSS 文件现在也包括上下文。这意味着我们并不会被迫将 CSS 文件应用于整个页面，而是可以通过库文件的-context.css 版本只将它们应用于页面的某个部分。这使得改进没有使用 YUI 的现有页面变得简单得多。此外，不再是一个 ID 定义一个网格文档，而是每个页面上可以有多个布局并存，所有基于 ID 的 CSS 选择器已经替换成 CSS 类名。

15.4　本章小结

不同浏览器的 CSS 实现更多属于艺术而不是科学。其结果就是，不同浏览器的呈现引擎并不总是能产生完全相同的布局。YUI Library 的一系列 CSS 文件在统一浏览器呈现引擎并将它们的输出规范化成更加可预测的输出结果方面取得了进展。这些文件涵盖的基础帮助开发人员和设计师立即展开工作，并让他们专注于网站设计而不是呈现引擎之间的差异。

PDG

第16章

构建和部署

JavaScript 是一种非编译语言，它以明文形式通过网络传送给接收端中的浏览器进行解释和执行。以编写形式存在的代码(保留所有空白符、注释和缩排格式)对于需要阅读它们的开发人员而言非常便利。但对于计算机而言，这些代码却会减慢它们的运行速度。空白符和注释对于浏览器而言完全无用，它们可能占据 JavaScript 文件规模的 60%。这可能就是造成在处理比较大的库文件时计算机性能低下的原因。

YUI 提供了几个解决方案来处理 JavaScript 的规模和传送问题。

本章内容简介：

- 如何使用来自 Yahoo!的共享 YUI 文件
- 如何减少和优化加载时间

16.1 来自 Yahoo!的共享 YUI 文件

缓存文件一直以来都是浏览器用来改进页面加载时间的一种有效技术。尽管缓存一个文件并不能帮助提高第一次加载该文件的时间，但是一旦该文件进入浏览器的缓存，那么访问它的速度就会得到极大的提高。这基本上是一种缩短距离的措施；将资源带到距离浏览器更近的地方，从而减少加载它们所需的时间。很明显，硬盘是浏览器获取文件最近的位置，但是我们还可以实现其他级别的接近程度。除了从主机托管服务器(hosting server)上传送文件之外，我们还可以从在地理位置上距离访问者更近的服务器那里将这些文件传送给访问者。这类服务器网络称为内容分发网络(Content Delivery Network，CDN)。作为一家大型公司，Yahoo!拥有自己的 CDN。在 2007 年 2 月，随着 YUI 2.2.0 的发布，该公司选择在 CDN 上托管 YUI 文件。顺便提及一下，Google 也将 YUI 文件以及其他各种库放在自己公司的服务器中进行托管(更多信息请参见 http://code.google.com/apis/ajaxlibs/documentation/#AjaxLibraries)。Yahoo!的 CDN 与 Google 的 CDN 之间的主要差别在于，Yahoo!支持组合处理(combo-handling)，而 Google 支持 SSL。

在 Yahoo! CDN 上托管的文件不仅距离访问者更近，而且它们的访问速度也要比一般服务器提供的服务速度快得多。这是因为 CDN 服务器专门配置用于传送静态的缓存文件，它们均针对该需求进行了优化。与处理大量服务器端脚本且可能访问数据库的网站相比，简化的 CDN 服务

器的速度要快很多。

通过使用托管 YUI 文件获得的另一个性能方面的好处是，如果访问者已经位于某个使用托管 YUI 文件的网站，那么该文件已经在他的计算机中(因为文件的签名取决于它的 URL，而该 URL 来自同一个来源)。换言之，访问者已经将该文件下载到自己的计算机上，因此他们不需要再次下载它。

使用 YUI 托管文件的最简单方式是访问 YUI Dependency Configurator(http://developer.yahoo.com/yui/articles/hosting/)。利用这个页面上的工具，通过一组易于使用的按钮，可以选择所有必要的依赖项，并且这个页面已经将必要的代码准备好，可供复制和粘贴到自己的页面中。使用这个配置器带来的一个好处就是，它能够感知特定库组件所需的依赖项。例如，单击“AutoComplete Control”按钮将生成代码来加载 autocomplete.css、yahoo-dom-event.js、datasource-min.js 和 autocomplete-min.js(参见图 16-1)。还可以调整配置器输出的最小化文件。通过单击另一个按钮，可以选择输出最小化的、原始的或调试版本的文件。

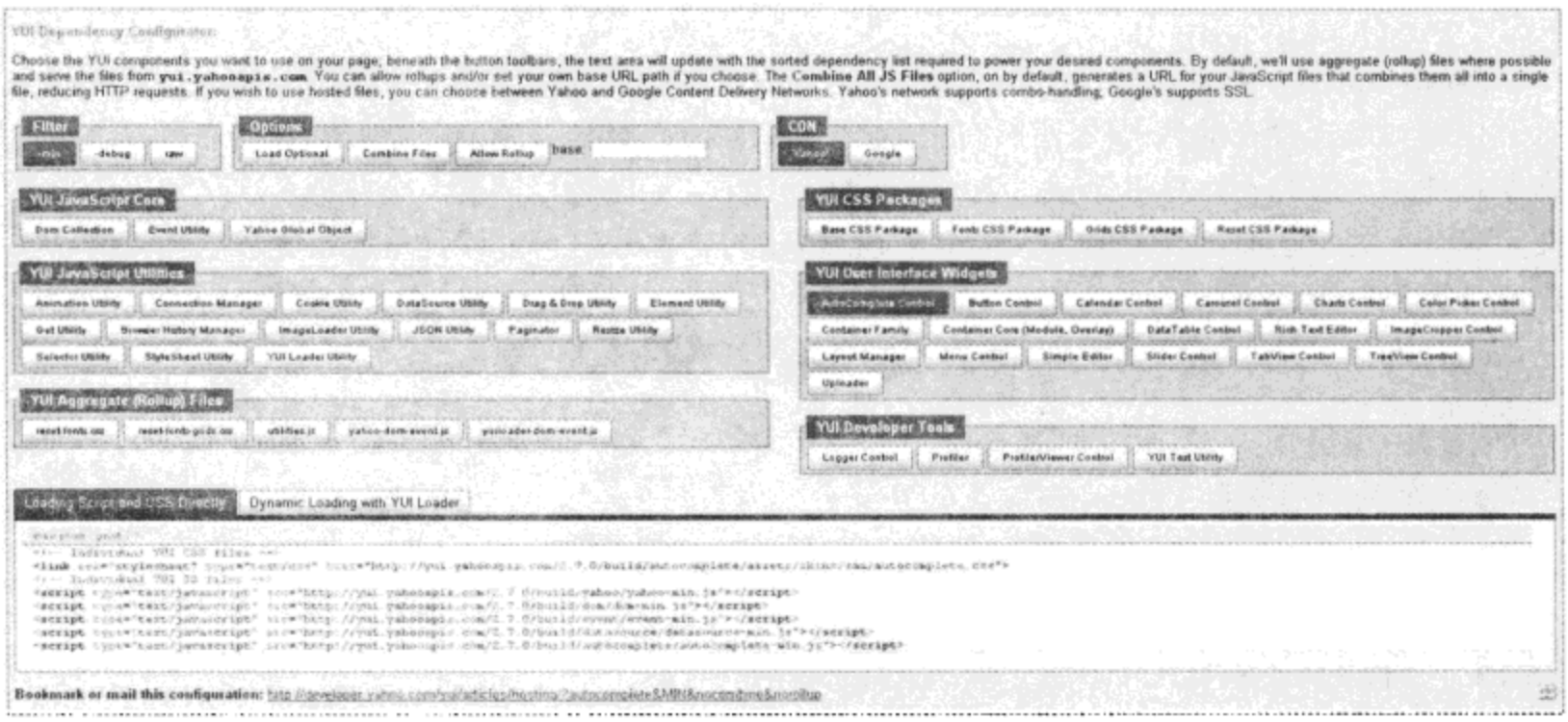


图　16-1

下面是一个快速示例，演示如何加载核心 Yahoo、DOM 和事件文件。确保所有的 src=表达式都在一行中声明。

```
<script type="text/javascript" src=
"http://yui.yahooapis.com/2.7.0/build/yahoo/yahoo-min.js"></script>
<script type="text/javascript" src="
http://yui.yahooapis.com/2.7.0/build/dom/dom-min.js"></script>
<script type="text/javascript" src=
"http://yui.yahooapis.com/2.7.0/build/event/event-min.js"></script>
```

还可以像下面这样从汇总文件 yahoo-dom-event.js 中加载所有 3 个文件：

```
<script type="text/javascript" src=
"http://yui.yahooapis.com/2.7.0/build/yahoo-dom-event/yahoo-dom-event.js"></script>
```

托管文件带有版本号(在这里它们属于 2.7.0 版本)，因此不可能出现不小心修改它们的情况。新版本的库只会放在新版本号的文件夹下面托管。与可下载库一样，托管文件也有 3 种风格：最

小化、调试和原始。可以简单地修改请求 URL 中的文件名，向基文件名中添加-min 或-debug，或者不添加任何内容，就可以访问这些风格的文件。下面是采用 3 种不同方式加载 yahoo.js 文件的示例：

最小化

```
<script type="text/javascript" src=
"http://yui.yahooapis.com/2.7.0/build/yahoo/yahoo-min.js"></script>
```

调试

```
<script type="text/javascript" src=
"http://yui.yahooapis.com/2.7.0/build/yahoo/yahoo-debug.js"></script>
```

原始

```
<script type="text/javascript" src=
"http://yui.yahooapis.com/2.7.0/build/yahoo/yahoo.js"></script>
```

组合文件

尽管通过 CDN 来提供文件服务可以提高浏览器访问这些文件的速度，但是这需要建立多个请求来下载文件，从而产生另一个性能瓶颈。每个 HTTP 请求都要建立到服务器的请求并等待响应，这会造成时间损失。更糟糕的是，浏览器在任一时刻最多只能建立两个到同一台服务器的 HTTP 连接，因此会有多个调用必须等待这些连接可用。此外，额外的 HTTP 报头数据以及文件头数据(以图像为例)也增加了需要往返传送的有效负载。

解决方案就是将所需的文件合并到聚合文件中，这样就可以尽可能减少 HTTP 请求数量。对于图像而言，这意味着创建子图形(这一点超出本书的介绍范畴)。对于 JavaScript 和 CSS 文件而言，所需的工作就是把这些文件连接成一个大型的 JavaScript 或 CSS 文件。YUI 已经有这样的文件，它的 yahoo-dom-event.js 文件就是由 yahoo.js、dom.js 和 event.js 文件组成的聚合文件。但是，其他的依赖项应该如何处理呢？为此，YUI 引入了组合处理程序，可用来动态地完成 YUI 文件的连接。下面就是使用组合处理程序加载 AutoComplete 脚本时的 JavaScript 代码：

```
<script type="text/javascript" src=
"http://yui.yahooapis.com/combo?2.7.0/build/yahoo-dom-event/yahoo-dom-event.js&2.7.
0/build/datasource/datasource-min.js&2.7.0/build/autocomplete/autocomplete-min.js"></sc
ript>
```

尽管组合处理程序建立的 URL 比较长，但是最终结果只有一个 HTTP 请求。对于 CSS 文件也可以进行同样的处理。下面的示例演示如何通过组合处理程序同时加载 AutoComplete 控件和 Button 控件的 CSS 文件：

```
<link rel="stylesheet" type="text/css" href=
"http://yui.yahooapis.com/combo?2.7.0/build/autocomplete/assets/
skins/sam/autocomplete.css&2.7.0/build/button/assets/skins/sam/button.css">
```

16.2 减少和优化加载时间

除了 CDN 和文件连接，还可以把减小传输文件的大小作为另一项性能优化措施。JavaScript 或 CSS 文件大小的 40%~60%来自于空白符、注释和冗长的变量名。

虽然并不属于 YUI 分发文件，但 YUI Compressor 是 YUI 用来解决 JavaScript 和 CSS 文件大小问题的解决方案。它之所以不属于可分发文件，是因为分发文件本身已经是最小化文件，但是任何希望利用 YUI 优化自己代码的人都可以使用这个工具。在 YUI Library 首页上可以找到该压缩器的链接。

YUI Compressor 将 JavaScript 和 CSS 文件中的所有注释和不必要的空白符去除。对于 JavaScript，它甚至还能够将局部变量重命名成单字母名称，从而进一步减小文件大小(之所以只重命名局部变量，是因为这不会让代码的实际 API 变得含糊不清，进而无意中引入程序错误)。

运行 YUI Compressor 需要 Java。它使用 Java 分析 JavaScript 文件。下面的示例演示如何运行该工具(x.y.z 代表 Compressor 的版本号)：

```
java -jar yuicompressor-x.y.z.jar myfile.js -o myfile-min.js
```

下面是 Compressor 提供的所有可用选项的列表：

```
$ java -jar yuicompressor-x.y.z.jar
Usage: java -jar yuicompressor-x.y.z.jar [options] [input file]

  Global Options
    -h, --help                Displays this information
    --type <js|css>           Specifies the type of the input file
    --charset <charset>       Read the input file using <charset>
    --line-break <column>     Insert a line break after the specified column number
    -v, --verbose             Display informational messages and warnings
    -o <file>                 Place the output into <file>. Defaults to stdout.

  JavaScript Options
    --nomunge                 Minify only, do not obfuscate
    --preserve-semi           Preserve all semicolons
    --disable-optimizations   Disable all micro optimizations

GLOBAL OPTIONS

  -h, --help
     Prints help on how to use the YUI Compressor

  --line-break
     Some source control tools don't like files containing lines longer than,
     say 8000 characters. The linebreak option is used in that case to split
     long lines after a specific column. It can also be used to make the code
     more readable, easier to debug (especially with the MS Script Debugger)
     Specify 0 to get a line break after each semi-colon in JavaScript, and
     after each rule in CSS.
```

```
  --type js|css
      The type of compressor (JavaScript or CSS) is chosen based on the
      extension of the input file name (.js or .css) This option is required
      if no input file has been specified. Otherwise, this option is only
      required if the input file extension is neither 'js' nor 'css'.

  --charset character-set
      If a supported character set is specified, the YUI Compressor will use it
      to read the input file. Otherwise, it will assume that the platform's
      default character set is being used. The output file is encoded using
      the same character set.

  -o outfile
      Place output in file outfile. If not specified, the YUI Compressor will
      default to the standard output, which you can redirect to a file.

  -v, --verbose
      Display informational messages and warnings.

JAVASCRIPT ONLY OPTIONS

  --nomunge
      Minify only. Do not obfuscate local symbols.

  --preserve-semi
      Preserve unnecessary semicolons (such as right before a '}') This option
      is useful when compressed code has to be run through JSLint (which is the
      case of YUI for example)

  --disable-optimizations
      Disable all the built-in micro optimizations.
```

This text comes from the YUI Compressor page and is an output of the actual YUI Compressor application.

注意：

这段文本取自于 YUI Compressor 页面，它是实际 YUI Compressor 应用程序的输出结果。

运行时机和场合

最好将 YUI Compressor 用于构建过程中。换言之，在开发环境中，JavaScript 和 CSS 文件应该保持编写时的完整形式。但是，当把这些文件移动到准生产环境或 QA 环境时，就应该使用 Compressor 处理这些文件。这样一来，源文件保持不变，但交付的版本得到优化。理想情况下，不应该压缩这些文件并直接部署到生产环境中(活跃的网站)，因为压缩过程中总是有可能出现错误。相反，应该压缩该文件并部署到一个能够进行测试和验证的环境中。一旦确认这些文件没有问题，就可以安全地将其部署到活跃的网站上。

16.3　本章小结

随着时间的过去，JavaScript 库大小的自然增长会造成文件规模和性能方面的问题，YUI 团队灵巧地解决了这些问题。他们不仅提供了库文件的轻量级版本，而且提供了免费的 CDN 服务来托管这些文件，此外还提供了工具来优化并不属于他们的库的代码。

第 III 部分

Ext JS

2006 年初，YUI Library 发布，其重点是实现强大、稳定而且文档齐全的通用 JavaScript 功能库。YUI 团队成功实现了最初制定的目标。但是，这个库在控件库方面提供的选择却很少。浏览器现在已经能够呈现丰富且极具吸引力、足以媲美传统桌面的界面。而 Internet 客户和用户同样也期盼拥有丰富的、高性能的 Web 体验。

一些历史信息

2006 年中期，Jack Slocum 登上了 YUI 的历史舞台，他基于 YUI 基础实现了一个非常吸引人的数据网格。这个网格对用户响应灵敏，提供了桌面用户一直期待的一些高级功能，而且在视觉上非常具有吸引力。Slocum 的代码基于 YUI 基础的面向对象设计，开发人员发现它非常易于学习和定制。

大约一年之后，Slocum 正式建立 YUI Library 的一个分支，并提供了一个名为 Ext JS 1.0 的控件库。Slocum 并不希望把自己的库绑定到 YUI 基础，因此他决定创建一个“适配器”抽象层，让开发人员自己在多种成熟的现代 JavaScript 库之间做出选择。Ext JS 现在有一个大型的控件库可供开发人员使用 jQuery、Prototype，当然还有 YUI。通过使用简单的名称空间技术，开发人员可以确信 Ext JS 控件库不会与他们选择的核心 JavaScript 库相冲突。在 Ext JS 1.0 发布几个月之后，它的 1.1 版本发布，其中提供了一个仅基于 Ext JS 代码(不依赖外部的库)的新“适配器”。从此之后，Ext JS 团队一直在维护“适配器”抽象层，同时不断地向其丰富的控件库添加新成员。

虽然 Ext JS 库最初只是一个数据网格，但该库并不是典型的 JavaScript 函数库。当然，Ext JS 具有开发人员一直期待的 JavaScript 库的通用功能(高性能的 DOM 遍历、CSS 和 HTML 操作、简化的 AJAX 执行、XML 和 JSON 数据处理等)。但 Ext JS 的目标是为开发人员提供完整的、带有主题的控件库，它易于扩展，易于使用，而且在视觉上极具吸引力，以至于成为事实上的标准。到目前为止，Ext JS 已经超越了所有这些目标。

在开始学习之前需要注意的要点

首先必须理解 JavaScript 是一门面向对象的语言。当前 Internet 上已有的大多数 JavaScript 代码都没有充分挖掘面向对象设计的全部潜力和灵活性。虽然即使初级开发人员也能够轻易地理解诸如继承和模块性这样的概念，但要理解多态(作用域)和封装(私有成员)的概念可能比较困难。Ext JS 库大量地使用了所有这些面向对象概念并受益匪浅。

还有一点值得注意：Ext JS 库的学习曲线比较陡峭。但是，这种学习曲线多多少少是由于开发人员一开始就贸然进入 Ext JS 窗口部件库“最酷的”部分造成的。当然，Ext Grid 功能异常强大。从简单的 HTML 表一下子跳到如此复杂的控件很明显并不是明智的做法。在这一部分中，在深入介绍“最酷的”部分之前，我们将花费相当一部分时间来理解 Ext JS 库的构建块。

第17章

架构和库约定

本章内容简介：

- 何时使用 Ext JS
- 如何使用 Ext JS
- Ext JS 的面向对象设计
- 功能强大的实用工具函数
- Ext JS 的基于事件的设计

17.1 何时使用 Ext JS

Ext JS 明确地瞄准下一代企业级 Web 应用程序。就这一点而言，Ext JS 是一个大型的综合性库。对于新手开发人员来说，它看起来相当让人畏惧。但确信无疑的是，它是一个经过精心设计且模块化的库。不管是创建简单的内容网站(博客、CRM、文档储存库)，还是创建完全交互的数据库应用程序，将 Ext JS 裁剪成适合自己的大小都是一件容易的事情。

在本章中将会看到，Ext JS 是可扩展的。如果不喜欢 Ext JS 执行某项特定任务的方式，那么可以将其重写、增强或者替换。

Ext JS 可以很好地与其他库协作。如果已经在使用另一个框架，那么 Ext JS 被设计成能够与其他表现良好的框架一起运行。事实上，我们可以将 Ext JS 的核心库替换成其他流行的框架(这既可以减少开发人员培训的时间以及已有代码的维护工作，还可以节省终端用户带宽)。

Ext JS 并没有使用任何整体类来解决许多问题。Ext JS 中的每个类均设计成解决少数几个问题。将想要的功能独立出来非常容易，而排除不需要的功能也同样容易。而且，这些类均是构建在彼此的基础之上，因此很少会遇到代码重复的问题。

将这些因素集成在一起就促成了一个得到良好维护的、紧密结合的、能够适合当今 Web 上几乎每个 HTML 项目的代码主体。

17.2 如何使用 Ext JS

要使用 Ext JS，需要下载 Ext JS 库。对于本节而言，我们将使用完整的 Ext JS 2.2 SDK，可以在 Ext JS 网站(http://extjs.com/)上找到它。Ext JS 网站还提供了“Build-Your-Own”功能，可用来选取自己所需的功能。完整的功能列表太长，因此不在此列出，但有一点值得注意的是，“Ext Core”功能是这个库唯一必需的部分。您并不会使用为本书定制的库版本，但如果选择使用“Build-Your-Own”功能，那么您将得到一个专门的、只占用非常小空间的脚本文件。

完整的 Ext JS SDK 带有文档、可运行的示例库、所有 Ext JS 资源(图像、CSS 文件以及主题)、第三方适配器(jQuery、Prototype 以及 YUI)、所有原始的源代码文件(内容完整且带有注释)、所有源代码文件的最小化版本以及几个汇编文件。这些汇编文件如下：

- ext-all.css 最小化形式的所有 CSS 文件。
- ext-all.js 最小化形式的完整库。
- ext-all - debug.js 不含注释(但不是最小化)的完整库。
- ext-core.js 最小化形式的“核心”文件组。
- ext-core-debug.js 不含注释(但不是最小化)的“核心”文件组。

Ext JS 库所使用的 CSS 文件要求如图 17-1 所示的典型网站文件夹层次结构。

图 17-1

在 images 文件夹中，每个 Ext JS 主题都有一个子文件夹。建立新主题是一件非常简单的事情，只要创建新 CSS 文件即可(在 CSS 文件夹中)。如果需要重写图像，那么就在 images 文件夹中建立一个新文件夹来容纳替换图像。虽然关于如何创建新主题的内容超出本书的讨论范围，但是有必要提醒一下，Ext JS SDK 中包含“default”和“gray”主题。这两个主题都是全功能的主题，可用作学习如何创建新主题的工具。

该 SDK 还带有一个 INCLUDE_ORDER.txt 文件，它描述了需要包含哪些文件，这取决于选择的适配器。对于本节而言，我们将使用 Ext JS 核心适配器。因此，我们需要包含如下 3 个文件：

```
<link type="text/css" rel="stylesheet" href="css/ext-all.css" />
<!-- the Ext adapter (as opposed to a 3rd Party adapter) -->
<script type="text/javascript" src="scripts/ext-base.js"></script>
<script type="text/javascript" src="scripts/ext-all.js"></script>
<script type="text/javascript">
    Ext.onReady(function() { // the web-page is now loaded
        alert("Hello, World!");
    });
</script>
```

利用这些代码，我们已经将完整的 Ext JS 库包含进来，并使用全局的 Ext.onReady 事件处理程序安全地启动应用程序。在深入研究更高级的功能之前，我们将在下一节讨论该库的一些较为基础的方面。

样式约定

Ext JS 库采用相当普遍的编码样式约定，这种一致性有助于开发人员的学习。虽然每个开发人员肯定会有自己的习惯和约定，但库自身风格的一致性是非常有帮助的：

- 类名首字母大写(GridPanel、Observable 等)。
- 事件名称小写(click、dblclick 等)。
- 常量大写(DAY、HOUR 等)。
- 所有其他标识符均采用驼峰式大小写(ext、doSomething、myValue 等)。

如果研究 Ext JS 源代码，那么将会看到下面的代码约定：

- 起始大括号出现在前一行代码的末尾。
- 所有的控制块均明确标出。

```
if (condition) code(); // this will not be seen
if (condition) { // this is preferable
    code();
}
```

Ext JS 库基于事件。任何“可以取消的”事件的名称均采用“before*Event*”这样的命名约定。在处理这些事件时，开发人员只要在事件处理程序中返回 false，就可以阻止事件处理流程剩余部分继续执行。

在 Ext JS 库中还有许多“单例”类。单例类不需要开发人员实例化，通常由一个有组织的方法库组成。因此，单例类通常是实用工具类(Ext.DomHelper)或相似类的集合的管理器(Ext.WindowMgr)。

下面是 Ext JS 库中创建单例类的典型技术：

```
MyClass = function() {
    var arg = 5; // a private variable
    return {
        myMethod: function() { // a public method
            return arg;
        }
    }
}();
// no need to instantiate
var myValue = MyClass.myMethod();
```

17.3　Ext JS 的面向对象设计

Ext JS 最初的目标之一是维持与任何其他负责任的 JavaScript 库共存的能力，并始终保持可扩展性。为此，Ext JS 类提供了几个简单的方法来帮助开发人员组织、创建和扩展类。尽管这些方法均是针对 Ext JS 库而编写的，但是它们也能够在所有 JavaScript 对象上运行。

17.3.1 Ext.namespace

```
namespace( String namespace1, String namespace2, String etc ) : void
```

在 JavaScript 中，名称空间就是一个对象，它只包含表示对象定义的属性。使用正确的名称空间可确保封装代码，使其不会与其他库产生冲突。当然，正确的名称空间很快就会变得非常冗长。像 Ext.ux.graphing.GraphPanel 这样经过慎重命名的类并不鲜见。利用 Ext.namespace 便利方法，只需要一次调用就可以定义所有的名称空间。这不仅有助于编写能够与其他代码共存的代码，而且减小了脚本的大小。

下面是传统的建立名称空间层次结构的 JavaScript 代码：

```
var Ext = Ext || {};
Ext.ux = Ext.ux || {};
Ext.ux.graphing = Ext.ux.graphing || {};
Ext.ux.soundFx = Ext.ux.soundFx || {};
Ext.ux.maps = Ext.ux.maps || {};
```

只需要调用一次 Ext.namespace 方法就可以实现同样的结果：

```
Ext.namespace("Ext.ux.graphing", "Ext.ux.soundFx", "Ext.ux.maps");
```

17.3.2 Ext.override

```
override( Object origclass, Object overrides ) : void
```

JavaScript 是一门灵活性很强的语言，要重写一个方法，只需要重新分配该方法的名称即可：

```
// let's create a function
function doesStuff() {
    alert("I do it!");
}
// now, let's override that function
doesStuff = function() {
    alert("I did it!");
};
```

要将方法分配给类定义，只需要使用该类的原型即可：

```
function MyClass() {
    // class constructor
}
MyClass.prototype.myMethod = function() {
    alert("hey");
};
var x = new MyClass();
x.myMethod(); // displays "hey"
MyClass.prototype.myMethod = function() {  // replace the method
    alert("hello");
};
x.myMethod(); // displays "hello"
```

使用同样的方法，开发人员还可以将自己的自定义函数添加到已有的类中。

Ext.override 方法是一个用来帮助开发人员即刻重新定义整个类的便利方法。下面的示例替换一个类定义中的两个方法。遵循传统的 JavaScript 约定，如果方法不在该类的原始定义中，就将该方法添加进去。第一个参数是要修改的类，第二个参数是包含正在重写的成员的 JavaScript 对象。

```
Ext.override(MyClass, {
    myMethod: function() {
        // do something different
    }
});
```

这个简单的方法在保持代码维护关注点最小化方面具有巨大的价值。例如，有经验的 JavaScript 开发人员熟悉如何使用和定制第三方库。但是，每当这些第三方库有新的发布版本时，都需要重新进行定制。使用 Ext.override 方法，开发人员可以让原有的第三方文件原封不动。可以很容易地把它们的自定义代码放到一个单独的自包含文件中。

17.3.3　Ext.extend 和构造函数约定

```
extend( Function superclass, Object overrides ) : Function
extend( Function subclass, Function superclass, [Object overrides] ) : Function
```

Ext.override 方法用来实现替换功能，而 Ext.extend 方法则是 Ext JS 实现面向对象继承机制的解决方案。有两种调用该方法的方式，每种方式都会产生一个新的类定义，该类继承自参数 superclass 定义的类，并重写参数 overrides 定义的方法。Ext.extend 方法有 3 个副作用，它们会作用到新类的定义中。首先，向新类添加 override 方法，该方法只是调用 Ext.override。其次，override 方法添加到新类的每个实例中。这个方法附带一个参数，并将该参数指向对象的所有成员复制到当前实例中(重写任何同名的现有成员)。最后，新类的每个实例中都有一个 superclass 属性，可用来访问父对象的定义。

前面曾经介绍过，Ext JS 有几个用来提高该库一致性的易于接受的编码约定。其中一个约定是该库的构造函数接口。几乎所有的 Ext JS 构造函数都只有一个参数，这个参数就是一个包含适合于实例化类的配置选项的对象。这个编码约定打破了传统的 OOP 语言约定。传统上，所有的方法签名(包括构造函数)都有明确的参数名称和明确的参数类型。这种传统约定利用了编译器的功能和安全性。但 JavaScript 参数不需要有数据类型，而且没有编译器。对于现代浏览器中的 JavaScript 而言，传统约定带来的好处并不存在。

事实上，传统约定确实妨碍 JavaScript 代码维护。例如，methodA 版本 1 期望第一个参数是 String 值。3 个月之后，methodA 版本 2 发布，它的第一个参数现在是 Date 值。由于没有编译器，开发人员被迫筛查所有代码，试图找出并修正本不应该是问题的代码。而且，虽然我们都知道 classA 的作者造成了所有这些不必要的痛苦，但是我们却束手无策。

这是一个非常常见的、现实的 JavaScript 开发问题。因此，Ext JS 选择了一种简单的、只带一个参数的方法签名约定，这种约定利用了 JavaScript 本身就具备的能力，并使代码维护变得轻松。此外，开发人员可以方便地派生 Ext JS 对象，而不需要担心基类的构造函数将来发生变化，从而破坏继承的对象。

除了上述副作用之外，Ext.extend 还有第四个可能的副作用：创建 constructor 方法。第一个调用 Ext.extend 的方法只接受两个参数。第一个参数是 superclass，而第二个参数(overrides)是一个对象，该对象包含要添加到 superclass 中的方法或重写 superclass 的方法。如果 overrides 参数包含 constructor 的定义，那么这个方法将用作新类定义的 constructor。如果没有定义 constructor 属性，那么 Ext.extend 将遵循已经描述过的约定创建一个 constructor。这个自动创建的 constructor 也将调用 superclass 的 constructor，并传入所有的参数。这种执行 superclass 的 constructor 的做法与传统 OOP 约定相当吻合，而且在使用 Ext JS 的单一配置对象约定时，它还能够向前兼容。

下面的示例演示了 Ext.extend 的所有用法，每种用法都实现了相同的目标。注意 superclass 的使用以及定义 constructor 的不同方法：

```
var NewClass = Ext.extend(MyClass, {
    constructor: function(config) { // explicit constructor
        config.name = config.name || "Bob";
        NewClass.superclass.constructor.call(this);
        this.name = config.name;
    },
    myMethod: function() {
        NewClass.superclass.myMethod.call(this);
        alert(this.name);
    }
});
function NewClass(config) { // constructor
    config.name = config.name || "Bob";
    NewClass.superclass.constructor.call(this);
    this.name = config.name;
}
Ext.extend(NewClass, MyClass, {
    myMethod: function() {
        NewClass.superclass.myMethod.call(this);
        alert(this.name);
    }
});
```

由于几乎所有 Ext JS 对象都有相同的构造函数签名，这就使得每个构造函数执行非常类似的操作。事实上，所有构造函数的主要功能就是将 config 参数的每个成员复制到正在构造的新对象中。Ext JS 提供了两个可以完成这项任务的便利方法。

17.3.4 Ext.apply

```
apply( Object obj, Object config, [Object defaults] ) : Object
```

Ext.apply 将 config 参数的所有成员复制到 obj 参数中。可以选择指定 defaults 参数，在复制 config 的值之前会先将其复制到 obj 中。

17.3.5 Ext.applyIf

```
applyIf( Object obj, Object config ) : Object
```

Ext.applyIf 将 config 参数的成员复制到 obj 参数中，但是只复制 obj 中没有的成员。

当开发人员继承一个对象时，他的目的通常是添加功能。增加的功能可能需要更多的配置。开发人员可以利用 Ext JS 的构造函数约定，调用 superclass 的构造函数(同时保留必要的配置信息)将变成一件简单的事情：

```
MyClass = Ext.extend(MyBase, {
    constructor: function(config) {
        // provide some default values, but only if they do not exist in 'config'
        config = Ext.applyIf(config, {
            someValue: 5
        });
        // provide some default values, overriding any that are already in 'config'
        config = Ext.apply(config, {
            someOtherValue: 10
        });
        // call the superclass's constructor, passing our modified configuration
        MyClass.superclass.constructor.call(this, config);
    }
});
```

apply 和 applyIf 方法的功能经常被人们忽视。虽然这些方法对于对象构造函数而言非常有帮助，但在整个库中有多处也用到了这些方法。

17.4　功能强大的实用工具函数

本节将要讲解的方法都已经添加到 Function 对象的原型中，这意味着开发人员能够将这些方法用在所有 JavaScript 函数上，从而修改它们的原始行为。

17.4.1　Function.createCallback

```
createCallback() : Function
```

事件和事件处理程序是 Ext JS 的主要部分。Function.createCallback 函数主要用于分配事件处理程序。例如，下面是一个简单的、只带有一个参数的 sayHi 函数：

```
function sayHi(name) {
    alert("Hi " + name);
}
```

假设在整个应用程序中的多个地方用到这个函数，其中一个地方就是在处理 Button 的单击事件时。当单击 Button 时，希望把“Fred”传递给 sayHi 函数。但是，事件处理程序配置选项期望的是一个函数指针。下面的代码不能运行：

```
new Ext.Button({
    text: "Say Hi",
    renderTo: Ext.getBody(),
    handler: sayHi("Fred") // this won't work!
});
```

这段代码使用 Function.createCallback 函数实现预期结果：

```
new Ext.Button({
    text: "Say Hi",
    renderTo: Ext.getBody(),
    handler: sayHi.createCallback("Fred") // much better
});
```

Function.createCallback 返回的函数能够处理任意数量的参数，而且总是在浏览器的 window 对象作用域中执行。

通过“设置作用域”，我们就能够控制 this 关键字引用的对象。前面提及，Function.createCallback 并不设置作用域，它使用的是浏览器的 window 对象的全局作用域。

17.4.2 Function.createDelegate

```
createDelegate( Object obj, [Array args], [Boolean/Number appendArgs] ) : Function
```

虽然 Function.createCallback 函数非常有用，但在有些场合中，受影响的函数需要在 window 对象之外的作用域中执行。这就是 Function.createDelegate 函数发挥作用的地方。Function.createDelegate 方法创建一个新的函数，当执行该函数时，将 this 关键字设成 obj 参数的值。在下面的示例中，请注意 this 引用对象的变化：

```
function MyClass(color) {
    this.color = color;
};
MyClass.prototype.myMethod = function(name) {
    alert("My name is " + name + ", and I like the color " + this.color);
};

// here, we instantiate a new object, which has its own scope
var x = new MyClass("red");
x.myMethod("Fred"); // displays "My name is Fred, and I like the color red"

// here, we create an object that is NOT derived from MyClass
//  but it has a 'color' property
var inlineObject = {color: "green"};
var func = MyClass.prototype.myMethod.createDelegate(inlineObject);
func("Sally"); // displays "My name is Sally, and I like the color green"

// this version of createDelegate uses an array of pre-defined parameters
var func2 = MyClass.prototype.myMethod.createDelegate(inlineObject, ["Sam"]);
func2(); // displays "My name is Sam, and I like the color green"
```

可以看到，Function.createDelegate 让开发人员能够方便地重用代码，而不需要修改类定义或完全遵循类定义。

17.4.3 Function.createInterceptor

```
createInterceptor( Function fcn, [Object scope] ) : Function
```

利用 Function.createInterceptor 方法，开发人员可以拦截方法调用，并根据条件来确定是否继续执行。只拦截方法而不派生类，这是相当有用的功能。例如，代码审核在任何语言中都是相当困难的工作。但是，使用 Function.createInterceptor 方法审核特定的方法却是非常快捷的做法。

```
// a VERY simple Logger class
function Logger() {
    this.messages = [];
}
// here's the method we want to audit
function myMethod(color) {
    alert(color);
}
// create an instance of the logger
var myLogger = new Logger();
// create an interceptor to be used in place of the original method
myMethod = myMethod.createInterceptor(function(color) {
    this.messages[this.messages.length] = color;  // write value to log
    return color != "yellow"; // if the color is yellow, we don't want to continue
}, myLogger);
myMethod("red"); // alert is displayed
myMethod("yellow"); // alert is NOT displayed
```

17.4.4　Function.createSequence

```
createSequence( Function fcn, [Object scope] ) : Function
```

另一个可用于实现简单代码审核的方法是 Function.createSequence。在这里，开发人员创建一个新函数，它执行原始函数，然后执行通过 fcn 参数传入的函数。例如：

```
function repeat() {
    var s = "";
    for (var x = 0; x < arguments.length; x++) {
        s += arguments[x] + ",";
    }
    alert("The arguments: " + s);
}
repeat(1, 2, 3); // displays "The arguments: 1,2,3,"
repeat = repeat.createSequence(function() {
    alert("There were " + arguments.length + " arguments.");
});
repeat(1, 2, 3); // displays "The arguments: 1,2,3,", and then
                 // displays "There were 3 arguments."
```

还可以使用可选的 scope 参数(就像在 Function.createInterceptor 示例中一样)。虽然 Function.createInterceptor 可以用来记录每次方法调用，但是 Function.createSequence 也可用来记录每次方法调用，并且能够在不抛出未处理异常的情况下完成该操作。有了这两个方法，只需要少量的工作就可以实现相当有用的代码审核器。

17.4.5 Function.defer

```
defer( Number millis, [Object obj], [Array args], [Boolean/Number appendArgs] ) :
    Number
```

Function.defer 方法调度原始函数在指定的毫秒数之后执行。可以选择设置函数的作用域并传入一个实参数组。这个方法返回一个 Number 值，它与浏览器的 clearTimeout 函数兼容。一个范例就是自动完成文本框。当用户输入内容时，我们希望检查数据存储以找出可能完成用户正在输入内容的字符串。但如果用户输入非常快，那么当我们正忙于检查字符串时，用户可能已经结束输入。相反，可以在文本框的 keyup 事件触发之后 1 秒钟调用 getPossibleStrings 方法。如果在执行 getPossibleStrings 之前用户按下另一个键，那么可以通过调用 clearTimeout 来取消第一次调用。

```
function getPossibleStrings(value) {
    alert("checking for " + value);
}
var x = null;
var txt = Ext.get("textbox");
txt.on("keyup", function() {
    if (x) {
        clearTimeout(x); // abort the previously deferred call
    }
    x = getPossibleStrings.defer(1000, window, [this.getValue()]);
}, txt);
```

17.5 Ext JS 的基于事件的设计

除了提供一致的面向对象设计之外，Ext JS 还提供了简单的、深思熟虑的基于事件的设计。Ext JS 基于事件的设计的核心是 Ext.util.Observable 类，它形成了所有需要引发事件的类的抽象基类。

17.5.1 Ext.util.Observable.addEvents

```
addEvents( Object object ) : void
```

通常在 Observable 的派生类的构造函数内部调用 addEvents 方法，该方法定义了该类可能引发的所有事件的集合。但是，如果开发人员试图针对一个未定义事件调用 addListener 或 fireEvent 方法，那么该事件将自动添加到这个集合中。因此，这个方法现在主要用于清晰易懂的代码。

```
MyClass = Ext.extend(Ext.util.Observable, {
    constructor: function() {
        this.addEvents({
            click: true,
            keyUp: true,
            keyDown: true
        });
    }
});
```

17.5.2　Ext.util.Observable.addListener /.on

```
addListener( String eventName, Function handler, [Object scope], [Object options] )
    : void
on( String eventName, Function handler, [Object scope], [Object options] ) : void
```

这就是典型的 addListener 方法，它附带参数 eventName 和 handler，懒惰的程序员可以使用 on 方法来减少输入的字符。可以选择指定一个 scope 对象。第四个参数也有一些选项：

- **delay—Number**　在事件引发后延迟指定毫秒数后再调用处理程序(参见 Function.defer)。
- **single—Boolean**　如果设为 true，就处理事件的下一次引发，然后自动将该处理程序移除。
- **buffer—Number**　行为与 delay 选项类似。但如果在指定的毫秒数内多次引发该事件，那么该处理程序只执行一次。

下面是该方法最简单的形式：

```
var instance = new MyClass();
instance.on("click", onClick);
```

还可以一次性添加多个处理程序：

```
instance.on({
    click: onClick,
    keyUp: onKeyUp
});
```

此外，还可以添加多个带选项的处理程序：

```
instance.on({
    click: {fn: onClick, single: true}, // handle the click event once
    keyUp: {fn: onKeyUp, buffer: 1000} // handle the keyUp event after a second of
                                       // no other keyUp events
});
```

17.5.3　Ext.util.Observable.removeListener /.un

```
removeListener( String eventName, Function handler, [Object scope] ) : void
un( String eventName, Function handler, [Object scope] ) : void
```

这个方法停止指定事件 eventName 的指定处理程序 handler 的执行。在 Observable 类的内部，它维护一个以 eventName、handler 和 scope 为键的处理程序集合。因此，如果曾经为特定的 scope 指定过一个处理程序，那么在移除该处理程序时必须指定这个特定的 scope。

17.5.4　Ext.util.Observable.fireEvent

```
fireEvent( String eventName, Object... args ) : Boolean
```

这个方法引发指定的事件 eventName 并将指定的参数传递给每个处理程序。按照每个处理程

序添加的顺序依次执行所有的处理程序。如果某个处理程序返回 false，就不会执行剩余的处理程序，并且 fireEvent 方法返回 false。下面的示例代码执行 click 事件(带有两个参数)：

```
if (!instance.fireEvent("click", 1, 2)) {
    alert("a handler returned false");
}
```

17.5.5 Ext.util.Observable.hasListener

```
hasListener( String eventName ) : Boolean
```

这个方法检查该对象是否具有针对指定事件 eventName 的处理程序。

17.5.6 Ext.util.Observable.purgeListeners

```
purgeListeners() : void
```

这个方法将对象的所有处理程序移除。

17.5.7 Ext.util.Observable.relayEvents

```
relayEvents( Observable o, Array events ) : void
```

这个方法将指定事件从该对象转发到指定的 Observable 对象。

在下面的示例中，每次 anotherInstance 对象引发 click 或 keyUp 事件时，instance 对象也会引发同样的事件：

```
instance.relayEvents(anotherInstance, ["click", "keyUp"]);
```

注意：

在 relayEvents 方法中用到的这两个实例对象都不必属于相同类型。

17.5.8 Ext.util.Observable.suspendEvents /.resumeEvents

```
suspendEvents() : void
resumeEvents() : void
```

suspendEvents 方法阻止事件引发，而 resumeEvents 方法恢复正常的执行。一旦已经调用 suspendEvents，那么调用 fireEvent 将没有任何效果。

17.5.9 Ext.util.Observable.capture /.releaseCapture

```
capture( Observable o, Function fn, [Object scope] ) : void
releaseCapture( Observable o ) : void
```

这两个方法都是静态的实用工具方法，它们位于名称空间 Ext.util.Observable 中，但并不属于 Observable 的实例。capture 方法用于在指定的 Observable 对象上为所有事件添加一个处理程序。这个处理程序将在所有普通的处理程序之前执行。还可以使用一个可选的 scope 对象。实际上，

capture 方法在指定对象的 fireEvent 方法上创建一个拦截器函数(参见 Function.createInterceptor)。

17.6　本章小结

本章快速介绍了隐藏在 Ext JS 背后的一些核心概念。在下一章中，我们将继续了解 Ext JS 库的构建块。浏览器平台和 HTML DOM 并不总是像我们希望的那样运转，而 Ext JS 具备一个牢固的基础，可以让开发人员的工作变得更加轻松。

元素、DomHelper 和模板

本书假设读者是经验丰富的 Web 开发人员，因此不会浪费时间讲解当今 HTML 世界中存在的大量问题。CSS 和 HTML 规范在不断地变化，每个季度都有新的浏览器版本发布，每位 Web 开发人员都不可避免地被要求以多种不同方式来解决同样的问题。

无论是遍历 DOM 来查找某段文本，还是仅仅要弄清楚用户单击的是哪个鼠标按键，对于 Web 开发人员而言，这些都是棘手的问题。但是，Ext JS 库解决了这些跨浏览器/跨版本问题。

此外，Ext JS 为开发人员构建丰富的 UI 提供了一个坚实的基础。本章主要演示 Ext JS 如何解决跨浏览器问题并为 Ext JS Component System 打下坚实的基础。

本章内容简介：

- 元素操作
- DOM 遍历
- DOM 操作
- CSS 操作

18.1 元素操作

HTML 文档由 HTML 元素组成。传统上，Web 服务器负责建立一个完整的 HTML 文档，然后浏览器下载该文档。HTML 文档包含浏览器所需要的所有信息。如果需要改变屏幕显示，浏览器就会向服务器请求新的 HTML 文档。但如果变化很小(例如启用一个按钮)，那么从服务器那里获取新文档的代价就显得太高。显而易见且代价较低的解决方案是在浏览器中修改 HTML 文档。为此，Ext JS 库提供了一个类，它可以表示每种 HTML 元素。

18.1.1 Ext.Element

因为 HTML 元素是 HTML 文档的核心，所以浏览器厂商一直努力围绕这些元素来建立 DOM API。为此，每种浏览器都提供了 HTMLElement 类。但是，这些浏览器提供不同的技术来解决相

同的问题，因此大多数 JavaScript 库的作者都开始增强 HTMLElement 类。然而，Ext JS 库并没有修改浏览器内置的 HTMLElement 类。相反，Ext JS 库建立了 Ext.Element 类来集中并简化修改元素所需的功能。保持 HTMLElement 类的“纯正性”的原因之一是可以让其他库(并不知道 Ext JS)继续运行，如同 Ext JS 不存在一样。通过这种低调的方式，我们可以将 Ext JS 库添加到现有的项目中，而不会对现有的基本代码产生较大的影响。

Ext.Element 是一个跨浏览器的抽象类，它提供了原生 HTMLElement 类提供的所有常见功能。经验丰富的 JavaScript 开发人员会立即理解 Ext.Element 类提供的方法。如果有必要，那么也可以利用 Ext.Element 的 dom 属性来访问未经修改的 HTMLElement。下面的代码示例构造了 Ext.Element 的一个实例：

```
// here's the old-fashioned way to get an HTMLElement
var domElement = document.getElementById("someId");
var el;
// here, we're constructing an Ext.Element by ID
el = new Ext.Element("someId");
// here, we're constructing an Ext.Element by dom node
el = new Ext.Element(domElement);
// let's prove we got the right HTMLElement
if (domElement == el.dom) {
    alert("We got it!");
} else {
    alert("We don't got it.");
}
```

还有一个辅助函数(get)可用来构造和检索 Element。get 方法与构造函数的工作方式类似，但它将所有“获得的” Element 缓存起来。这种简单的缓存确保开发人员每次都会接收到同一个 Element 对象(而不是每次都要构造新实例，从而消耗内存)。对这种缓存定期地执行垃圾回收(连同 Element 上的所有事件处理程序)，这样就可以避免浏览器内存泄漏。

```
// the helper function works with IDs and dom nodes too
el = Ext.get("someId");
el = Ext.get(domElement);
```

事实上，Ext.Element 对象并没有自己的属性(它只是原生 HTMLElement 的跨浏览器包装器)。即使这样，当操作大量 HTMLElement(例如表中的每个单元格)时，为每个 HTMLElement 构造一个 Element 实例也是非常昂贵的操作。为了缓解这个问题，还可以使用另一个辅助函数(Ext.fly)，它维护 Ext.Element 的“轻量级”实例。

```
// the 'fly' method works just like 'get'
el = Ext.fly("someId");
el = Ext.fly(domElement);
```

这个“轻量级”Ext.Element 实例利用了它的“包装器”架构，只交换它的 dom 属性值。Element 的所有方法继续按照预期运行，而不用承担成百上千个实例化对象所造成的系统开销。但开发人员必须知道，在整个应用程序中都在使用轻量级对象。这意味着 dom 属性将交换出去。换言之，在诸如密集循环这种性能至关重要的场合中使用 fly 方法。

```
// make all the inputs transparent without constructing
//  unnecessary Element instances
var col = document.getElementsByTagName("input");
for (var i = 0, el; el = Ext.fly(col[i++]);) {
    el.setOpacity(0);
}
```

“轻量级”对象实际上是一个 Ext.Flyweight 实例，开发人员可以实际地创建多个轻量级对象。对于将自己的代码与 Ext JS 的代码分离(并提高自己的轻量级元素的可预测性)来说，这是非常有用的做法。要创建并访问自己的 Ext.Flyweight 实例，只需要指定一个名称即可：

```
var el = Ext.fly("someId", "myFlyweight");
```

Ext.Element 类提供了几个 DOM 操作方法(createChild、replace、replaceWith、insertHtml、insertFirst 等)，每个方法都使用 Ext.DomHelper 类，我们将在本章稍后讨论该类。

1. DOM 事件处理

Element 类并不继承自 Ext.util.Observable。但是，Element 内部使用 Ext.EventManager 类，而且具有熟悉的 addListener/on 和 removeListener/un 方法。Ext.EventManager 类是另一个用来帮助开发人员解决棘手问题的跨浏览器类。所有浏览器共享同一个 DOM 接口，并且所有引发的事件都将产生共同的、跨浏览器的对象(Ext.EventObject)。Ext.Element.addListener 的参数与 Ext.util.Observable 相同。options 参数仍然允许采用 scope、delay、single 和 buffer 选项，但它还提供了如下的选项：

- delegate　一个简单的选择器，用来查找目标的祖先元素(如果未能找到匹配的祖先元素，就使用原来的目标元素)。
- normalized　如果为 false，则将浏览器事件传给处理程序函数而不是 Ext.EventObject。
- preventDefault　如果为 true，则阻止浏览器的默认操作。
- stopEvent　如果为 true，则停止事件。与 preventDefault: true 和 stopPropagation: true 相同。
- stopPropagation　如果为 true，则阻止事件传播(事件冒泡)。

这些额外的选项是 DOM 事件特有的选项，因此对 Ext.util.Observable 而言并不需要有它们。如果使用 preventDefault、stopEvent 或 stopPropagation，那么仍然会调用处理程序。

虽然普通 Ext JS 事件处理程序的函数签名是特有的，但 DOM 事件处理程序都具有相同的签名(Ext.EventObject 或浏览器的事件对象、Ext.Element 和 options 对象)。

Ext.EventObject 类中有几个辅助方法用来获得关于事件的跨浏览器信息(getKey、getWheelDelta、getPageX、getPageY 等)。但更为重要的是，browserEvent 属性提供了原有的原始浏览器事件对象。

2. 元素动画

Ext.Element 类的许多方法都接受一个可选的 animate 参数。设置这个参数可以在该方法执行过程中执行可见的动画。animate 参数可设为 true 以显示默认的行为，或者利用传入一个 options 对象来进行精细的控制。该 options 对象具有如下属性：

- duration　确定动画持续的时间(单位为秒，默认为 0.35)。

- **easing** 确定动画的行为(有效值如下：easeNone、easeIn、easeOut、easeBoth、easeInStrong、easeOutStrong、easeBothStrong、elasticIn、elasticOut、elasticBoth、backIn、backOut、backBoth、bounceIn、bounceOut、bounceBoth)。
- **callback** 当动画完成时执行该函数。
- **scope** 确定回调的作用域。

下面是操作带有动画的 Element 的示例：

```
// by default, no animation
el.moveTo(50, 50);
// here, we use the default animation settings
el.moveTo(100, 100, true);
// here, we slow down the animation (default duration is .35)
el.moveTo(150, 150, {
   duration: .5
});
```

因为假设使用的是完整的 Ext JS 库，所以 Ext.Element 类带有完整的动画效果库。这个函数库放在 Ext.Fx 类中，包括诸如 fadeIn、fadeOut、highlight、scale 之类的效果。与其他 Ext.Element 方法一样，这些效果方法也可以链式调用：

```
el.hide(); // hide the element
el.slideIn().puff(); // the element drops into view from the top and then
                     // slowly fades while growing larger (like smoke)
```

在将视觉效果方法与非视觉效果方法进行链式调用时必须特别小心。Ext.Fx 方法立即返回(即使效果尚未完成)。在内部，每种效果都添加到一个定序器中(这样效果之间就不会彼此干扰)。而非视觉方法就不会添加到这个序列中。

```
el.fadeIn().setVisible(false); // the element is hidden before
                               // the fadeIn method can complete
el.fadeIn({
   callback: function() { // callbacks are called when the animation is complete
      el.setVisible(false);
   }
}); // the user now gets the intended effect
```

3. 操作元素集合

Ext.Element 可用来操作单个 HTMLElement 对象，而 Ext.Flyweight 通过使用更少的对象来节省宝贵的浏览器资源。但维护一个元素集合是一种常见的需求。Ext.Flyweight 用来保存单个元素，而 Ext.Element 的系统开销太大。这就是 Ext.select 方法发挥作用的地方：

```
var col = Ext.select("div"); // grab ALL div tags on the page
```

Ext.select 方法返回 Ext.CompositeElementLite 类的一个实例。这个类在内部保存 Ext.Flyweight 的实例以及一个 DOM 节点数组。该类提供 Ext.Element 类的所有方法，并在数组中的每个 DOM 节点上执行所有的方法调用：

```
var col = Ext.select("div"); // grab ALL div tags on the page
```

```
col.setStyle({border: "solid 1px green"}); // give them all a border
```

Ext.CompositeElementLite 类继承自较少用到的 Ext.CompositeElement 类。CompositeElement 存储一个 Ext.Element 数组，并没有利用轻量级技术。要创建一个 CompositeElement 实例，可使用 Ext.select 方法，并将它的第二个参数设为 true：

```
var col = Ext.select("div", true); // true for unique elements (non-flyweight)
```

CompositeElement 及其子类 CompositeElementLite 均有几个实用工具方法用来操作元素集合。这些方法是标准的集合形式方法，例如 add、contains、clear、each、item 等。

18.1.2　Ext.Element 方法

Element 类的几乎所有方法都返回 Element 自身，从而可以采用链式调用技术：

```
// set height, then set color
el.setHeight(50).setStyle({color: "red"});
```

下面是 Ext.Element 类的所有方法分类之后的清单(没有额外的参数描述)。

1. 通用实用工具

这些方法可用来以低级方式访问 HTMLElement 的任何属性。

```
getAttributeNS( String namespace, String name ) : String
getValue( Boolean asNumber ) : String/Number
set( Object o, [Boolean useSet] ) : Ext.Element
```

2. DOM 结构化

前面曾经讨论过，DOM 结构是 HTML 文档的重要组成部分。这些方法帮助开发人员移动、添加、移除和替换 HTML 文档结构中的元素。在内部，这些方法使用 Ext.DomHelper 和 Ext.DomQuery 类(都将在本章中讨论)。

```
appendChild( String/HTMLElement/Array/Element/CompositeElement el ) : Ext.Element
appendTo( Mixed el ) : Ext.Element
child( String selector, [Boolean returnDom] ) : HTMLElement/Ext.Element
clean( [Boolean forceReclean] ) : void
contains( HTMLElement/String el ) : Boolean
createChild( Object config, [HTMLElement insertBefore], [Boolean returnDom] )
    : Ext.Element
createProxy( String/Object config, [String/HTMLElement renderTo],
    [Boolean matchBox] ) : Ext.Element
createShim() : Ext.Element
down( String selector, [Boolean returnDom] ) : HTMLElement/Ext.Element
findParent( String selector, [Number/Mixed maxDepth], [Boolean returnEl] )
    : HTMLElement
findParentNode( String selector, [Number/Mixed maxDepth], [Boolean returnEl] )
    : HTMLElement
first( [String selector], [Boolean returnDom] ) : Ext.Element/HTMLElement
insertAfter( Mixed el ) : Ext.Element
```

```
insertBefore( Mixed el ) : Ext.Element
insertFirst( Mixed/Object el ) : Ext.Element
insertHtml( String where, String html, [Boolean returnEl] )
    : HTMLElement/Ext.Element
insertSibling( Mixed/Object/Array el, [String where], [Boolean returnDom] )
    : Ext.Element
is( String selector ) : Boolean
last( [String selector], [Boolean returnDom] ) : Ext.Element/HTMLElement
next( [String selector], [Boolean returnDom] ) : Ext.Element/HTMLElement
parent( [String selector], [Boolean returnDom] ) : Ext.Element/HTMLElement
prev( [String selector], [Boolean returnDom] ) : Ext.Element/HTMLElement
query( String selector ) : Array
remove() : void
replace( Mixed el ) : Ext.Element
replaceWith( Mixed/Object el ) : Ext.Element
select( String selector, [Boolean unique] ) : CompositeElement/CompositeElementLite
up( String selector, [Number/Mixed maxDepth] ) : Ext.Element
update( String html, [Boolean loadScripts], [Function callback] ) : Ext.Element
wrap( [Object config], [Boolean returnDom] ) : HTMLElement/Element
```

3. 定位

这些方法帮助开发人员修改 Element 的视觉位置。

```
alignTo( Mixed element, String position, [Array offsets],
    [Boolean/Object animate] ) : Ext.Element
anchorTo( Mixed element, String position, [Array offsets],
    [Boolean/Object animate], [Boolean/Number monitorScroll], Function callback )
    : Ext.Element
autoHeight( [Boolean animate], [Float duration], [Function onComplete],
    [String easing] ) : Ext.Element
center( [Mixed centerIn] ) : void
clearPositioning( [String value] ) : Ext.Element
getAlignToXY( Mixed element, String position, [Array offsets] ) : Array
getAnchorXY( [String anchor], [Boolean local], [Object size] ) : Array
getBox( [Boolean contentBox], [Boolean local] ) : Object
getCenterXY() : Array
getComputedHeight() : Number
getComputedWidth() : Number
getHeight( [Boolean contentHeight] ) : Number
getLeft( Boolean local ) : Number
getOffsetsTo( Mixed element ) : Array
getPositioning() : Object
getRegion() : Region
getRight( Boolean local ) : Number
getScroll() : Object
getSize( [Boolean contentSize] ) : Object
getTextWidth( String text, [Number min], [Number max] ) : Number
getTop( Boolean local ) : Number
getViewSize() : Object
getWidth( [Boolean contentWidth] ) : Number
```

```
getX() : Number
getXY() : Array
getY() : Number
move( String direction, Number distance, [Boolean/Object animate] ) : Ext.Element
moveTo( Number x, Number y, [Boolean/Object animate] ) : Ext.Element
position( [String pos], [Number zIndex], [Number x], [Number y] ) : void
scroll( String direction, Number distance, [Boolean/Object animate] ) : Boolean
scrollIntoView( [Mixed container], [Boolean hscroll] ) : Ext.Element
scrollTo( String side, Number value, [Boolean/Object animate] ) : Element
setBottom( String bottom ) : Ext.Element
setBounds( Number x, Number y, Number width, Number height,
    [Boolean/Object animate] ) : Ext.Element
setBox( Object box, [Boolean adjust], [Boolean/Object animate] ) : Ext.Element
setHeight( Number height, [Boolean/Object animate] ) : Ext.Element
setLeft( String left ) : Ext.Element
setLeftTop( String left, String top ) : Ext.Element
setLocation( Number x, Number y, [Boolean/Object animate] ) : Ext.Element
setPositioning( Object posCfg ) : Ext.Element
setRegion( Ext.lib.Region region, [Boolean/Object animate] ) : Ext.Element
setRight( String right ) : Ext.Element
setSize( Number width, Number height, [Boolean/Object animate] ) : Ext.Element
setTop( String top ) : Ext.Element
setWidth( Number width, [Boolean/Object animate] ) : Ext.Element
setX( Number The, [Boolean/Object animate] ) : Ext.Element
setXY( Array pos, [Boolean/Object animate] ) : Ext.Element
setY( Number The, [Boolean/Object animate] ) : Ext.Element
translatePoints( Number/Array x, [Number y] ) : Object
```

4. 外观

这些方法帮助开发人员修改 Element 的外观。

```
addClass( String/Array className ) : Ext.Element
addClassOnClick( String className ) : Ext.Element
addClassOnFocus( String className ) : Ext.Element
addClassOnOver( String className ) : Ext.Element
applyStyles( String/Object/Function styles ) : Ext.Element
boxWrap( [String class] ) : Ext.Element
clearOpacity() : Ext.Element
clip() : Ext.Element
enableDisplayMode( [String display] ) : Ext.Element
getBorderWidth( String side ) : Number
getBottom( Boolean local ) : Number
getColor( String attr, String defaultValue, [String prefix] ) : void
getFrameWidth( String sides ) : Number
getMargins( [String sides] ) : Object/Number
getPadding( String side ) : Number
getStyle( String property ) : String
getStyles( String style1, String style2, String etc. ) : Object
hasClass( String className ) : Boolean
hide( [Boolean/Object animate] ) : Ext.Element
```

```
isBorderBox() : Boolean
isDisplayed() : Boolean
isMasked() : Boolean
isScrollable() : Boolean
isVisible( [Boolean deep] ) : Boolean
radioClass( String/Array className ) : Ext.Element
removeClass( String/Array className ) : Ext.Element
repaint() : Ext.Element
replaceClass( String oldClassName, String newClassName ) : Ext.Element
setDisplayed( Mixed value ) : Ext.Element
setOpacity( Float opacity, [Boolean/Object animate] ) : Ext.Element
setStyle( String/Object property, [String value] ) : Ext.Element
setVisibilityMode( visMode Element.VISIBILITY ) : Ext.Element
setVisible( Boolean visible, [Boolean/Object animate] ) : Ext.Element
show( [Boolean/Object animate] ) : Ext.Element
toggle( [Boolean/Object animate] ) : Ext.Element
toggleClass( String className ) : Ext.Element
unclip() : Ext.Element
```

5. 事件和行为

这些方法辅助开发人员完成事件处理并控制 Element 的行为。在内部，每个订阅的 DOM 事件均由 Ext.EventManager 类处理。每次浏览器引发一个 DOM 事件时，它都会传递一个对象，该对象的属性指示了浏览器的当前状态。Ext JS 将这个对象包装到一个名为 Ext.EventObject 的跨浏览器类中。这两个类建立了一个可预测、可重用且可扩展的跨浏览器平台，该平台以事件为基础。本章稍后将更加深入地讨论这些类。

```
addKeyListener( Number/Array/Object/String key, Function fn, [Object scope] )
    : Ext.KeyMap
addKeyMap( Object config ) : Ext.KeyMap
addListener( String eventName, Function fn, [Object scope], [Object options] )
    : void
blur() : Ext.Element
focus() : Ext.Element
hover( Function overFn, Function outFn, [Object scope] ) : Ext.Element
mask( [String msg], [String msgCls] ) : Element
on( String eventName, Function fn, [Object scope], [Object options] ) : void
relayEvent( String eventName, Object object ) : void
removeAllListeners() : Ext.Element
removeListener( String eventName, Function fn, [Object scope] ) : Ext.Element
swallowEvent( String eventName, [Boolean preventDefault] ) : Ext.Element
un( String eventName, Function fn ) : Ext.Element
unmask() : void
unselectable() : Ext.Element
```

6. 动画

这些辅助方法在前一节中已经讨论过，它们只是用来包装 Ext.Fx 类的功能。

```
animate( Object args, [Float duration], [Function onComplete], [String easing],
```

```
    [String animType] ) : Ext.Element
// the following methods are added from the Ext.Fx class
fadeIn( [Object options] ) : Ext.Element
fadeOut( [Object options] ) : Ext.Element
frame( [String color], [Number count], [Object options] ) : Ext.Element
ghost( [String anchor], [Object options] ) : Ext.Element
hasActiveFx() : Boolean
hasFxBlock() : Boolean
highlight( [String color], [Object options] ) : Ext.Element
pause( Number seconds ) : Ext.Element
puff( [Object options] ) : Ext.Element
scale( Number width, Number height, [Object options] ) : Ext.Element
sequenceFx() : Ext.Element
shift( Object options ) : Ext.Element
slideIn( [String anchor], [Object options] ) : Ext.Element
slideOut( [String anchor], [Object options] ) : Ext.Element
stopFx() : Ext.Element
switchOff( [Object options] ) : Ext.Element
syncFx() : Ext.Element
```

7. 服务器通信

这些方法帮助开发人员与服务器进行 AJAX 形式的通信。

```
getUpdater() : Ext.Updater
load() : Ext.Element
```

8. 拖放

拖放功能是 Web 开发人员面临的棘手问题。为了简化该操作，提供了下面 3 个便利方法。

```
initDD( String group, Object config, Object overrides ) : Ext.dd.DD
initDDProxy( String group, Object config, Object overrides ) : Ext.dd.DDProxy
initDDTarget( String group, Object config, Object overrides ) : Ext.dd.DDTarget
```

18.2　DOM 遍历

当谈到在 Web 应用程序中使用 JavaScript 时，即使最复杂的任务也都能够归结为动态修改结构化 HTML 文档。这意味着操作文档对象模型。修改 HTML 文档涉及如下 3 个步骤：

(1) 定位新内容的位置。

(2) 构建新内容。

(3) 插入新内容。

传统上，在浏览器中进行 DOM 的修改是一个复杂的过程：确定正在使用哪一种浏览器，弄清楚使用的是该浏览器的什么版本，然后执行该浏览器的 document 对象上的正确方法。我们已经知道如何定位一个元素(使用 Ext.get 或 Ext.fly)，而且已经简单了解如何定位一组元素(使用 Ext.select)。当使用 Ext.select 方法创建一组元素时，可以使用 DOM 节点数组、Ext.Element 数组或 DOM 节点 id 数组。但如果使用的是字符串，那么 Ext.select 方法将使用 Ext.DomQuery 类。

18.2.1 Ext.DomQuery

Ext.DomQuery 是一个单例类，它包含用于创建和执行高性能选择器函数的实用工具方法。在整个 Ext JS 库中到处都用到了 DomQuery 类，该类可以接受几种形式的选择器(其中有许多尚未得到现代浏览器的支持)，其中包括 XPath 和 CSS。下面是每种选择器的简要介绍。

1. 元素选择器

*——任意元素。
E——带有标记 E 的元素。
E F——E 的所有带有标记 F 的子元素。
下面是 4 个“并不简单的”选择器。
E > F 或 E/F——E 的所有带有标记 F 的直接子元素。
E + F——紧靠元素 E 之后的所有带有标记 F 的元素。
E ~ F——元素 E 之后的所有带有标记 F 的兄弟元素。

2. 属性选择器

@和引号的使用是可选的。例如，div[@foo='bar']也是有效的属性选择器。
E[foo]——具有属性“foo”。
E[foo=bar]——具有属性“foo”，其值等于“bar”。
E[foo^=bar]——具有属性“foo”，其值以“bar”开头。
E[foo$=bar]——具有属性“foo”，其值以“bar”结尾。
E[foo*=bar]——具有属性“foo”，其值包含“bar”子字符串。
E[foo%=2]——具有属性“foo”，其值能够被 2 整除。
E[foo!=bar]——具有属性“foo”，其值不等于“bar”。

3. 伪类

E:first-child——E 是其父节点的第一个子节点。
E:last-child——E 是其父节点的最后一个子节点。
E:nth-child(n)——E 是其父节点的第 n 个子节点(从 1 开始)。
E:nth-child(odd)——E 是其父节点的奇数子节点。
E:nth-child(even)——是其父节点的偶数子节点。
E:only-child——E 是其父节点的唯一子节点。
E:checked——E 是 checked 属性值为 true 的元素(例如单选按钮或复选框)。
E:first——结果集中的第一个 E 元素。
E:last——结果集中的最后一个 E 元素。
E:nth(n)——结果集中的第 n 个 E 元素(从 1 开始)。
E:odd——:nth-child(odd)的快捷方式。
E:even——:nth-child(even)的快捷方式。
E:contains(foo)——E 的 innerHTML 包含子字符串“foo”。
E:nodeValue(foo)——E 包含一个其 nodeValue 等于“foo”的 textNode。

E:not(S)——一个不匹配简单选择器 S 的 E 元素。
E:has(S)——元素 E 的子节点匹配简单选择器 S。
E:next(S)——元素 E 的下一个兄弟节点匹配简单选择器 S。
E:prev(S)——元素 E 的上一个兄弟节点匹配简单选择器 S。

4. CSS 值选择器

E{display=none}——CSS 值“display”等于“none”。
E{display^=none}——CSS 值“display”以“none”开头。
E{display$=none}——CSS 值“display”以“none”结尾。
E{display*=none}——CSS 值“display”包含子字符串“none”。
E{display%=2}——CSS 值“display”能够被 2 整除。
E{display!=none}——CSS 值“display”不等于“none”。

每种选择器都可供开发人员使用，而且可以与其他选择器以任何能够想到的方式组合使用。按照开发人员指定的顺序依次计算所有选择器。这些功能让开发人员能够具有全部的控制权，从而针对其特有的 DOM 结构需求来优化选择器定义。在内部，选择器要执行大量的正则表达式计算。通过使用 compile 方法，开发人员可以提升查询的性能(这对于常用查询来说非常有用)。

18.2.2　Ext.DomQuery 方法

在本节中将描述如下几个方法。

1. compile

```
compile( String selector, [String type] ) : Function
```

这个方法将一个选择器或 xpath 查询编译成一个可重用的函数。这个方法返回一个函数，该函数附带一个参数(root)，并返回一个 DOM 节点数组。参数 root(默认为 document)是作为查询开始位置的上下文节点。

```
var fn = Ext.DomQuery.compile("input:checked");
var a = fn(); // check the entire document
alert("I found " + a.length + " elements!");
```

参数 type 接受的值有“select”或“simple”，默认值为“select”。这个参数指示如何搜索 ID 和 tagName 属性。“simple”的速度稍快，但它不支持下面的元素选择器：

- E > F 或 E / F
- E + F
- E ~ F

2. filter

```
filter( Array el, String selector, [Boolean nonMatches] ) : Array
```

这个方法接受一个元素数组作为参数，并返回该数组中匹配指定的 simple 选择器的元素。可以选择将 true 值传给参数 nonMatches(默认为 false)，从而获得不匹配指定选择器的元素。

3. is

```
is( String/HTMLElement/Array el, String selector ) : Boolean
```

如果传入的元素匹配指定的 simple 选择器，那么这个方法返回 true。

4. select

```
select( String selector, [Node root] ) : Array
```

这个方法返回一个由匹配指定选择器的 DOM 节点构成的数组。这个方法中的 selector 参数可以包含多个选择器定义(由逗号分隔)。还可以选择指定一个参数，让搜索从指定的 DOM 节点(默认为 document)开始。

```
// grabs all the text boxes on the document
var col = Ext.DomQuery.select("input[type=text],input:not([type]),textarea");
```

5. selectNode

```
selectNode( String selector, [Node root] ) : Element
```

这个方法返回 Ext.DomQuery.select 中的第一个元素，并接受相同的参数。

6. selectNumber

```
selectNumber( String selector, [Node root], [Number defaultValue] ) : Number
```

这个方法执行 selectValue，并试图将它的返回值分析成一个 Number。如果没有找到值，那么可以指定一个 defaultValue(默认为 0)。还可以选择指定一个参数，让搜索从指定的 DOM 节点(默认为 document)开始。

7. selectValue

```
selectValue( String selector, [Node root], [String defaultValue] ) : String
```

这个方法选择由选择器定义找到的第一个节点的值。如果没有值，那么可以通过可选参数 defaultValue 提供一个值(默认为 null)。还可以选择指定一个参数，让搜索从指定的 DOM 节点(默认为 document)开始。

18.3 DOM 操作

现在我们已经知道如何查找元素并操作它们。为了全面控制 DOM，下一步就是创建和插入新的 DOM 节点。Ext.DomHelper、Ext.Template 和 Ext.XTemplate 均是用来帮助开发人员实现这个步骤的类。

18.3.1 Ext.DomHelper

Ext.DomHelper 单例类实现了创建新元素并将其放进 HTML 文档的任务。DomHelper 中的每个方法均接受一个含有 HTML 代码片段的参数，或者一个含有新元素定义的对象。这个对象的一

个示例如下所示。

```
var html = {
    tag: "div",
    children: [
        {
            tag: "input",
            type: "submit",
            value: "Click Me!"
        },
        {
            html: "some <i>inner</i> HTML"
        }
    ],
    cls: "someCssClass"
};
```

下面是定义对象中几个需要知道的特殊属性：

- **tag**　这是待创建的元素类型(默认为 div)。
- **children**　这是一个由将要放进新元素的对象定义组成的数组。
- **cls**　这是 CSS 类名(这个特殊的名称是为了避免与保留字“class”产生冲突)。
- **html**　这是 HTML 代码片段，该值将分配给新元素的 innerHTML 值。
- **style**　参见后面介绍的 Ext.DomHelper.applyStyles。

如果同时使用了 children 和 html，那么创建 children 而抛弃 html。

DomHelper 类还有一个 useDom 属性(默认为 false)。默认情况下，DomHelper 使用字符串连接构建 HTML 代码片段并插入这一个字符串(这通常会导致最佳的性能)。但如果 useDom 属性设为 true，那么 DomHelper 构建 HTML DOM 节点并使用浏览器的 DOM 方法插入所有节点。

1. Ext.DomHelper.append

```
append ( Mixed el, Object/String o, [Boolean returnElement] )
    : HTMLElement/Ext.Element
```

这个方法将新元素(由 o 定义)插入到指定元素(由 el 定义)的结束标记之前。el 可以是 DOM 节点的 id、DOM 节点或 Ext.Element。该方法返回一个 DOM 节点(默认情况下)或一个 Ext.Element 对象(如果 returnElement 为 true)。

2. Ext.DomHelper.applyStyles

```
applyStyles ( String/HTMLElement el, String/Object/Function styles ) : void
```

这个方法接受 DOM 节点的 id 或 DOM 节点(el)作为参数，并将指定的 CSS 样式(styles)应用于该节点。可以通过字符串、对象或者一个返回字符串或对象的函数来指定样式。例如：

```
function getStyles(useObj) {
    if (useObj) {
        return {width: "100px", color: "#000000"};
    } else {
        return "width:100px;color:#000000;";
```

```
    }
}
var style = getStyles(false);
Ext.DomHelper.applyStyles(myEl, style); // set styles using a string
var styleObj = getStyles(true);
Ext.DomHelper.applyStyles(myEl, styleObj); // set styles using an object
Ext.DomHelper.applyStyles(myEl, getStyles); // set styles using a function
```

3. Ext.DomHelper.createTemplate

```
createTemplate ( Object o ) : Ext.Template
```

这个方法调用 Ext.DomHelper.markup，然后返回一个新的 Ext.Template。有关 Ext.Template 的更多信息，请参见本章后面的内容。

4. Ext.DomHelper.insertAfter

```
insertAfter ( Mixed el, Object o, [Boolean returnElement] )
    : HTMLElement/Ext.Element
```

这个方法将新元素(由 o 定义)附加到指定元素(由 el 定义)的结束标记之后。el 可以是 DOM 节点的 id、DOM 节点或 Ext.Element。该方法返回一个 DOM 节点(默认情况下)或一个 Ext.Element 对象(如果 returnElement 为 true)。

5. Ext.DomHelper.insertBefore

```
insertBefore ( Mixed el, Object/String o, [Boolean returnElement] )
    : HTMLElement/Ext.Element
```

这个方法将新元素(由 o 定义)附加到指定元素(由 el 定义)的开始标记之前。el 可以是 DOM 节点的 id、DOM 节点或 Ext.Element。该方法返回一个 DOM 节点(默认情况下)或一个 Ext.Element 对象(如果 returnElement 为 true)。

6. Ext.DomHelper.insertFirst

```
insertFirst ( Mixed el, Object/String o, [Boolean returnElement] )
    : HTMLElement/Ext.Element
```

这个方法将新元素(由 o 定义)附加到指定元素(由 el 定义)的开始标记之后。el 可以是 DOM 节点的 id、DOM 节点或 Ext.Element。该方法返回一个 DOM 节点(默认情况下)或一个 Ext.Element 对象(如果 returnElement 为 true)。

7. Ext.DomHelper.insertHtml

```
insertHtml ( String where, HTMLElement el, String html ) : HTMLElement
```

这个方法将一个 HTML 代码片段插入到文档中，插入位置相对于指定的 DOM 节点(el)。参数 where 可以有如下值："beforeBegin"、"afterBegin"、"beforeEnd" 或 "afterEnd"。该方法返回一个新的 DOM 节点。当 useDom 为 false 时，append、insertAfter、insertBefore 和 insertFirst 在内部均使用这个方法。

8. Ext.DomHelper.markup

```
markup ( Object o ) : String
```

这个方法附带一个元素定义对象作为参数，并返回一个 HTML 代码片段。

9. Ext.DomHelper.overwrite

```
overwrite ( Mixed el, Object/String o, [Boolean returnElement] )
    : HTMLElement/Ext.Element
```

这个方法利用新元素(由 o 定义)来覆盖指定元素(由 el 定义)的内容。el 可以是 DOM 节点的 id、DOM 节点或 Ext.Element。该方法返回一个 DOM 节点(默认情况下)或一个 Ext.Element 对象(如果 returnElement 为 true)。这个方法会忽略 useDom 属性。

18.3.2　Ext.Template

DomHelper 中的跨浏览器方法当然要好于处理所有浏览器特有的问题。而且通过将 HTML 定义为一个定义对象，这个简单的方法提升了工作效率和性能。但是，Ext JS 库更进一步。该库引入 Ext.Template 类，让开发人员创建可重用的 HTML 模板，每次执行时可以将不同的值插入到模板中(例如，表行有不同的值和属性，但 HTML 结构是相同的)。

然而，Ext.Template 类只是一个字符串生成器。虽然它本来的目标是构建 HTML 代码片段，但它的用途可以更加广泛得多(例如，生成文本段落)。

此外，Ext.Template 类识别 Ext.util.Format 类。Ext.util.Format 是一个包含用于字符串格式化的实用工具函数的单例类。例如，要格式化日期字符串，可以使用下面的代码：

```
var d = new Date(); // current date/time
var s = Ext.util.Format.date(d, "m/d/Y");
alert(s); // you'll see a user-friendly date string
```

下面是一个使用 Ext.Template 的示例：

```
var t = new Ext.Template("{myDate:date('m/d/Y')}");
var s = t.apply({myDate: new Date()});
alert(s); // you'll see the same user-friendly date string
```

很明显，这个简单的示例没有太多实际意义。Ext.Template 的真正作用在于生成大量 HTML 代码片段并将这些代码片段绑定到复杂的数据。

1. Template.from

```
Template.from ( String/HTMLElement el, [Object config] ) : Ext.Template
```

这个静态方法获取指定元素的文本值(value 属性或 innerHTML 值)，并调用 Ext.Template 构造函数。

2. 构造函数

```
Template ( String/Array html, [Object config] )
```

这个构造函数根据参数 html 来创建 Ext.Template 类的一个实例。参数 config 的每个属性都会复制到新实例(参见 Ext.apply)。只有一个配置选项 compiled。compiled 配置选项强制在构造期间调用 compile 方法。其他所有配置选项都会复制到类的实例中。

构造函数也可以接受无限数量的字符串参数(用来提高可读性)。在内部，这些字符串将连接起来。

```
var t;
// each of the following constructor calls gets identical results
t = new Ext.Template("<div>{text}</div>");
t = new Ext.Template(
    "<div>",
        "{text}",
    "</div>"
);
// this line compiles the template
t = new Ext.Template("<div>{text}</div>", {compiled: true});
```

3. append

```
append( Mixed el, Object/Array values, [Boolean returnElement] )
    : HTMLElement/Ext.Element
```

将值应用于模板(参见 Ext.Template.applyTemplate)并调用 Ext.DomHelper 方法将新 HTML 插入到指定元素(由 el 定义)的结束标记之前。el 可以是 DOM 节点的 id、DOM 节点或 Ext.Element。该方法返回一个 DOM 节点(默认情况下)或一个 Ext.Element 对象(如果 returnElement 为 true)。

4. applyTemplate/apply

```
applyTemplate( Object/Array values ) : String
```

这个方法返回一个包含带有指定值的模板的 HTML 代码片段。可以将该值指定为定义对象或数组。

5. compile

```
compile() : Ext.Template
```

与 Ext.DomQuery 类相似，Ext.Template 在内部使用了大量的正则表达式运算。而且与 Ext.DomQuery 相似，这个类提供了 compile 方法，以便提供更佳的性能。这个方法只返回该类的当前实例(而不是 Ext.Template 的一个新实例)。

6. insertAfter

```
insertAfter( Mixed el, Object/Array values, [Boolean returnElement] )
    : HTMLElement/Ext.Element
```

将值应用于模板(参见 applyTemplate/apply)并调用 Ext.DomHelper 将新 HTML 插入到指定元素(由 el 定义)的结束标记之后。el 可以是 DOM 节点的 id、DOM 节点或 Ext.Element。该方法返回一个 DOM 节点(默认情况下)或一个 Ext.Element 对象(如果 returnElement 为 true)。

7. insertBefore

```
insertBefore( Mixed el, Object/Array values, [Boolean returnElement] )
    : HTMLElement/Ext.Element
```

将值应用于模板(参见 applyTemplate/apply)并调用 Ext.DomHelper 将新 HTML 插入到指定元素(由 el 定义)的开始标记之前。el 可以是 DOM 节点的 id、DOM 节点或 Ext.Element。该方法返回一个 DOM 节点(默认情况下)或一个 Ext.Element 对象(如果 returnElement 为 true)。

8. insertFirst

```
insertFirst( Mixed el, Object/Array values, [Boolean returnElement] )
    : HTMLElement/Ext.Element
```

将值应用于模板(参见 applyTemplate/apply)并调用 Ext.DomHelper 将新 HTML 插入到指定元素(由 el 定义)的开始标记之后。el 可以是 DOM 节点的 id、DOM 节点或 Ext.Element。该方法返回一个 DOM 节点(默认情况下)或一个 Ext.Element 对象(如果 returnElement 为 true)。

9. overwrite

```
overwrite( Mixed el, Object/Array values, [Boolean returnElement] )
    : HTMLElement/Ext.Element
```

将值应用于模板(参见 applyTemplate/apply)并完全将指定元素(由 el 定义)的 innerHTML 值替换成新 HTML。el 可以是 DOM 节点的 id、DOM 节点或 Ext.Element。该方法返回一个 DOM 节点(默认情况下)或一个 Ext.Element 对象(如果 returnElement 为 true)。

10. set

```
set( String html, [Boolean compile] ) : Ext.Template
```

开发人员可以使用这个方法，通过提供新 HTML 并有选择地编译模板来重置模板。与构造函数不同，html 参数必须是一个字符串。

18.3.3　Ext.XTemplate

功能强大的 Ext.Template 类能够将复杂的数据绑定到 HTML 代码片段，只需要少量几行代码即可。这节省了开发人员的时间，易于维护，而且提高了可读性。但 Ext JS 库提供了 Ext.XTemplate 类，让开发人员可以更进一步。XTemplate 派生自 Ext.Template 类，它具备所有功能，易于使用，而且能够处理分层的数据。

注意：
不应该在 Ext.XTemplate 实例上使用 set 方法，这样做实际上会破坏模板。

所有 XTemplate 方法都派生自 Template，因此不需要在此赘述。但是，模板语法现在支持一种新的标记：tpl。开发人员可以利用 tpl 标记进行循环、条件判断和代码执行。例如：

```
var t;
t = new Ext.XTemplate(
```

```
    "<div>",
        "<span>{name:capitalize}</span>", // use method from Ext.util.Format
        "<tpl for='scores'>", // looping
            "<p>",
                "{#}: {[this.round(score)]}", // execute code in square brackets
                "<tpl if='score &lt; 60'>", // conditional
                    " AWFUL!",
                "</tpl>",
            "</p>",
        "</tpl>",
    "</div>"
, {
    round: function(value) {
        return Math.round(value);
    }
});
```

tpl 标记有 3 个属性，不能将这 3 个属性连用：

```
<tpl for="string"> - The contents of the tpl tag execute for each member of the
    named child array.
<tpl if="condition"> - If the condition returns true, the contents of the tpl tag
    execute.  The basic math operators (+, -, /, and *) can also be used here.
<tpl exec="executable code"> - Executes the specified code (ignores the contents
    of the tpl tag).
```

在 tpl 标记内，还可以使用几个关键字：

```
{string} - Binds the named value of the current object.
{parent.string} - When looping, binds the named value of the parent object.
{#} - When looping, binds the index of the current object (1 based).
{.} - When looping through an array of values (not objects), binds the current
    value.
{[executable code]} - Executes the specified code in the scope of the current
    XTemplate instance.  The following variables are available for use (these
    are also available to <tpl if> and <tpl exec>):
    this - the current XTemplate instance
    values - the current object - ex: values.score
    parent - the current object's parent object - ex: parent.name
    xindex - when looping, the current object's index
    xcount - when looping, the total number of items
    fm - shortcut to Ext.util.Format - ex: fm.trim(values.name)
```

由于{[...]}块在当前 XTemplate 实例的作用域中执行，因此该实例的所有方法和属性都可供使用。这会非常有用，在前面的代码示例中已经看到，我们使用 round 方法自定义格式。

这些功能使得 Ext.XTemplate 成为 Ext JS 库中最为强大的类之一。

18.4 CSS 操作

如果不能控制层叠样式表，那么 DOM 操作控制就算不上完整，而 Ext JS 也同样为此提供了

解决方案。

Ext.util.CSS

这个简单的单例类可以让开发人员跨浏览器控制附加到 HTML 文档的 CSS。在一些浏览器中访问 CSS 信息的系统开销可能比较大，因此 Ext JS 为了提高性能而将该信息缓存起来。这个缓存只用于该类(它并不会影响 Ext.DomQuery 或 Ext JS 库的任何其他部分)。如果动态地移除 CSS 信息，就要确保在必要的时候调用 refreshCache。

1. Ext.util.CSS.createStyleSheet

```
createStyleSheet( String cssText, String id ) : StyleSheet
```

这个方法根据一段规则文本(cssText)建立样式表。这些规则将包装到 style 标记中，附加到文档头，并具有指定的 id。

2. Ext.util.CSS.getRule

```
getRule( String/Array selector, [Boolean refreshCache] ) : CSSRule
```

这个方法返回第一次匹配指定选择器之一的 CSSRule。开发人员可以选择重新构建内部的 CSSRule 缓存。如果没有找到指定的选择器，那么这个方法返回 null。

3. Ext.util.CSS.getRules

```
getRules( [Boolean refreshCache] ) : Object
```

这个方法返回一个包含该文档的所有 CSSRule 的对象。这个对象具有如下的结构：

```
{
    selector1: definition1,
    selector2: definition2
}
```

开发人员可以选择重新构建内部的 CSSRule 缓存。

4. Ext.util.CSS.refreshCache

```
refreshCache() : Object
```

这个方法刷新规则缓存(如果动态添加样式表)，它等同于 getRules(true)。

5. Ext.util.CSS.removeStyleSheet

```
removeStyleSheet( String id ) : void
```

这个方法使用指定的 id 移除 style 或 link 标记。如果没有找到指定的 id，那么什么事情也不会发生。

6. Ext.util.CSS.swapStyleSheet

```
swapStyleSheet( String id, String url ) : void
```

这个方法执行 removeStyleSheet(使用指定的 id)，然后使用指定的 id 和 url 创建一个 link 标记。

7. Ext.util.CSS.updateRule

```
updateRule( String/Array selector, String property, String value ) : Boolean
```

这个方法更新 CSSRule 的 property，这会使用指定的 value 匹配指定的选择器 selector。

```
Ext.util.CSS.updateRule([".invalid", ".error"], "color", "red");
```

18.5　本章小结

本章简要地讲解了 Ext JS 的 DOM 遍历和 DOM 操作技术。Ext JS 具有牢固的对象模型、经过深思熟虑的事件系统以及规范化的浏览器接口，我们现在已经为解决构建 Internet 应用程序的实际业务做好准备。有一点值得注意的是，大多数 JavaScript 库并没有提供 HTML 生成功能。实际上，我们已经具备生成大型的有效 HTML 代码片段并将其精确地放到 DOM 中所需的所有构建块。在下一章中，我们将深入研究 Ext Component 系统，它是 Ext JS 库中的所有 UI 窗口部件的重要组成部分，而且它直接构建在本章所概述的基础之上。

第19章

组件、布局和窗口

本章将讲解 Ext JS 库的实质部分。到目前为止，我们涉及的仅仅是 Ext JS Component 系统的构建块。HTML 元素本身不错，但即使是最基本的“窗口部件”，DOM 都太笨拙，而且处理起来非常费时间。例如，要创建自动完成组合框(常被开发人员视为基本控件)，需要把几个 HTML 元素准确放置，这些元素必须在视觉上和行为上相互协调。当然，需要把组合框放到一个表单上，并置于其他几个数据输入控件的上下文中。所有的控件必须与标签和验证标记汇集起来。对于顾客来说，这只是基本的控件。但对于 Web 开发人员来说，这很快就会变成一个相当棘手的问题。Component 系统为这个问题的彻底解决提供了非常优雅的解决方案。

本章内容简介：

- Ext JS Component 系统
- Ext JS Component 生命周期
- Ext.Viewport
- Ext.Container 布局
- 面板和窗口

19.1 Ext JS Component 系统

下面描述 Ext JS Component 系统。

19.1.1 Ext.Component

Ext JS Component 从 Ext.Component 类开始。它是所有窗口部件的基础，而且是 Ext JS 工具集中所有“窗口部件”的基类。无论开发人员需要的是一个简单的复选框，还是一个复杂的带有嵌套网格的 3 列布局，Ext.Component 类都是所有组件的核心。在内部，每个 Component 都有一个 Ext.Element 实例(该实例又含有一个指向旧式 DOM HTMLElement 的引用)。

Ext.Component 类为开发人员构建代码提供了非常好的基础。对于嵌套组件，Component 类提

供了几个用于遍历组件层次结构的方法(要遍历组件层次结构，请参见 19.1.4 节)。Component 还实施了一个简单的生命周期策略，这将辅助完成事件监听程序清理任务并减少浏览器内存泄漏问题。通过简单的方法就可以完全支持诸如显示/隐藏、启用/禁用以及保存状态/恢复状态之类的常见任务。

为了进一步提高可扩展性，Component 系统还添加了一个强大的插件架构。每个 Component 都接受一个插件集合(配置选项 plugins)。一个插件就是一个具有 init 方法的类。插件的 init 方法将接收一个指向 Component 的引用。插件可以利用这个引用来执行完成任务所需的任何操作。这是不需要通过继承就可以增强现有 Component 的极其简单的方法。

除此之外，Ext.Component 派生自 Ext.util.Observable，这就让所有 Component 能够引发它们自己的自定义事件。记住，这些事件并没有必要绑定到 HTML 元素。有几个事件是成对出现的(例如，beforerestorestate 和 restorestate)。在这些事件中，第一个事件会在指定操作发生之前引发。通过从事件处理程序中返回 false，开发人员可以取消指定的操作。在指定的操作发生之后，就会引发第二个事件。

有些开发人员可能会质疑，在原生 HTML 之上添加另一个抽象层是否物有所值。但是，请考虑一个简单的 ComboBox“窗口部件”。按照 HTML 规范，ComboBox 通常由一个 input 标记(用于显示选中的文本)、一个隐藏的 input 标记(用于存储选中的值)、一个用于显示所有选项的滚动 div 标记、每个选项对应的 div 标记和隐藏 input 标记、一个 button 标记(用于显示滚动 div 标记)以及一个用于把这些元素包装起来的 div 标记组成。这是一个典型的解决方案，当然可能还有更好的解决方案。但问题是一个这样的“窗口部件”就非常容易激增到成百上千个 HTMLElement 对象。再加上浏览器不兼容问题，遍历、操作以及与所有这些 HTMLElement 对象交互将很快变成一个棘手的问题。

显然，创作原始的、原生的 HTML 标记已经成为快速开发的障碍。利用“窗口部件”接口进行开发可以显著地提升工作效率，而且 Ext.Component 类提供了简单的、可扩展的基础。

19.1.2 Ext.ComponentMgr

当创建每个 Component 对象时，它们都会注册到 Ext.ComponentMgr 单例类(Ext.ComponentMgr.register)。与此类似，当销毁每个 Component 对象时，也会从 ComponentMgr 集合中移除该对象(Ext.ComponentMgr.unregister)。Component 构造函数确保每个 Component 对象都有一个唯一的 ID(并不是 DOM 节点 ID)。通过该 ID，Ext.ComponentMgr 提供了对所有 Component 对象的简单访问。

```
var cmp = Ext.ComponentMgr.get("myComponent");
cmp = Ext.getCmp("myComponent"); // short-hand
```

ComponentMgr 类也提供了一个只读属性(Ext.ComponentMgr.all)，它是所有实例化的 Component 对象的集合。all 是 Ext.util.MixedCollection 类的一个实例。MixedCollection(派生自 Observable)是一个名-值形式的集合。每当该集合出现改动时，都会引发相关的事件。这可以让开发人员完全了解应用程序中所有实例化的 Component 对象。

需要明确的是，Ext Component 系统最好用于单页面应用程序。强制终端用户浪费时间等待 Web 服务器为许多不同的屏幕页面产生屏幕布局是一种带宽浪费。与其使用不必要的回送，Ext

开发人员可以按需创建 Component 对象，并在需要的时候重绘屏幕页面的选中部分。这并不是说 Ext Component 不能用于传统的网页中。Tree 和 Grid 类属于在任何情况下都会表现上佳的组件。但为了充分利用所有窗口部件并让它们按照最佳方式协作，首选单页面应用程序。

为此，每个派生自 Component 的类都有一个 xtype 配置选项。xtype 可用于嵌套组件的延迟实例化和延迟呈现。对于提高大型应用程序的性能，这是一个非常有用的选项。例如，大型单页面应用程序(包含大量屏幕页面)的用户可能只需要打开一个屏幕页面。利用延迟实例化可以只创建必要的 Component，而利用延迟呈现可以防止浏览器浪费处理资源绘制绝不会被用户看到的 DOM 节点。这项技术可以让 Ext 应用程序能够正确地扩展。单页面应用程序不会占用大量内存，从而不会严重减慢用户系统的运行速度。

为了进一步阐述 xtype 概念，下面给出一个带有一个表单 Panel 和 3 个 TextField 的 Window 对象示例：

```
var window = new Ext.Window({
    items: [
        new Ext.Panel({
            layout: "form",
            items: [
                new Ext.form.TextField({
                    fieldLabel: "First Name"
                }),
                new Ext.form.TextField({
                    fieldLabel: "Middle Name"
                }),
                new Ext.form.TextField({
                    fieldLabel: "Last Name"
                })
            ]
        })
    ],
    title: "Sample Window"
});
window.show(); // display the window
```

下面是使用 xtype 的相同示例：

```
var window = new Ext.Window({
    items: [
        {xtype: "panel",
            layout: "form",
            items: [
                {xtype: "textfield",
                    fieldLabel: "First Name"
                },
                {xtype: "textfield",
                    fieldLabel: "Middle Name"
                },
                {xtype: "textfield",
                    fieldLabel: "Last Name"
```

```
                }
            ]
        }
    ],
    title: "Sample Window"
});
window.show(); // display the window
```

如果这个窗口显示出来，那么所有这些示例都会向用户产生相同的结果。但是，如果该窗口没有显示，那么 xtype 示例占用的内存会小很多。此外，该 xtype 示例不会把时间浪费在绝不会用到的对象的实例化上。对于较大规模的应用程序，这种对内存和性能的节省累积起来就会得到理想的结果。

注意：

在上面的代码示例中，并没有使用 xtype 值“window”。xtype 实例化操作发生在容器中。在正常情况下，窗口应该浮动在所有其他内容之上，而且不必有容器。

xtype 也会注册到 ComponentMgr 类(Ext.ComponentMgr.registerType)。如果 Component 类的 xtype 没有注册到 ComponentMgr，那么它就不能利用延迟实例化和延迟呈现功能。

```
Ext.ComponentMgr.registerType("textfield", Ext.form.TextField);
// now, the "textfield" xtype will be recognized
```

所有内置 Component 都会注册自己的 xtype。但如果开发自己的 Component 并希望它们利用延迟实例化和延迟呈现功能，那么就必须注册它们。

19.1.3 Ext.BoxComponent

有几个类直接继承自 Component(Ext.Button、Ext.ColorPalette、Ext.DatePicker 等)，但 Ext.BoxComponent(也派生自 Component)是另一个增加功能的基础基类。基本上，BoxComponent 增加了一些用于 Component 对象定位、移动和改变大小的简单功能。任何派生自 BoxComponent 的类都可以调用诸如 getSize、setPosition、setHeight 之类的方法。BoxComponent 类确保所有 Component 对象精确、一致地定位。

19.1.4 Ext.Container

Ext.Container 类派生自 BoxComponent，它增加了嵌套 Component 对象的功能。在一个调用中创建完整的 Component 层次结构并非难事：指定一个 Component 数组作为配置选项 items 即可。一旦创建完毕，就可以通过只读属性 items(Ext.util.MixedCollection 的实例)来访问嵌套的 Component 对象。但 Container 类还增加了几个方法，用于进一步简化嵌套 Component 对象的查找。下面是这些方法的简要介绍。

1. getComponent

```
getComponent(String/Ext.Component id) : Ext.Component
```

这个方法搜索 items 属性以查找指定的 id 或 Component 对象。这个方法不会递归搜索。

2. findBy

```
findBy(Function func, [Object scope]) : Ext.Component[]
```

这个方法递归地使用指定的函数来确定应该返回哪些嵌套 Component 对象：

```
// find all the hidden components
var hiddenComponents = comp.findBy(function(innerComp) {
    return innerComp.hidden;
});
```

3. find

```
find(String name, Object value) : Ext.Component[]
```

这个方法递归地使用指定的属性 name 和 value 来确定应该返回哪些嵌套 Component 对象：

```
// find all the hidden components
var hiddenComponents = comp.find("hidden", true);
```

4. findById

```
findById(String id) : Ext.Component
```

这个方法递归地使用指定的 id 来确定应该返回哪些嵌套 Component 对象：

5. findByType

```
findByType(String/Class xtype, []) : Ext.Component[]
```

这个方法使用 findBy 方法(递归方法)查找所有具有指定 xtype(或 Component 类类型)的嵌套 Component 对象：

```
// find all TextFields
var texts = comp.findByType("textfield");
// or
texts = comp.findByType(Ext.form.TextField);
```

6. 创建嵌套组件

可以使用 add 或 insert 方法将项添加到 Container 对象(或它的派生对象)中，但最简单的方式是在构造时使用配置选项 items。配置选项 defaults 也可用来一次性在 items 上指定常见的、可重写的配置选项：

```
var window = new Ext.Window({
    defaults: {
        allowBlank: false, // all fields are required
        xtype: "textfield" // all nested items are text boxes by default
    },
    items: [
        {fieldLabel: "First Name"},
```

```
            {fieldLabel: "Last Name", allowBlank: true}, // not a required field
            {fieldLabel: "Has Hair", xtype: "checkbox"} // not a textfield
        ],
        layout: "form"
    });
    window.show();
```

通过使用配置选项 items 并利用每一项的配置选项 xtype，开发人员节省了宝贵的浏览器内存，并提高了性能。

19.2 Ext JS Component 生命周期

Component Life Cycle 控制着每个 Component 对象的初始化、呈现以及销毁。根据设计，最终开发人员不需要关心 Component Life Cycle 的内部工作原理。在默认情况下，Ext.Container(以及它的子类)将自动销毁所有嵌套的 Component 对象，而每个 Component 对象负责处理自己创建的所有 Element 对象。尽管生命周期主要是为典型用途而设计的，但是深入了解它仍然非常有用。

19.2.1 初始化

一旦 Component 实例化，其生命周期的初始化阶段就会开始。

首先，如果没有在配置选项中指定 Component ID，就为其生成一个 ID。然后将该 Component 对象注册到 ComponentMgr，并调用基类(Ext.util.Observable)构造函数(创建在配置选项中指定的所有事件监听程序)。

接下来调用 initComponent 方法，这是初始化阶段的核心。在默认情况下，initComponent 不执行任何操作，Component 的子类应该重写该方法。但是在派生类时，一定要调用基类的 initComponent 方法。这个方法应该执行构造函数的典型操作(分配集合类、附加默认的事件监听程序等)。通过在这里(而不是在构造函数中)执行这些操作，Component 实施了一个可预测的架构，可以让以后的子类有机会重写该架构。

现在初始化插件架构。依次调用配置选项 plugins 中每个插件的 init 方法，并把 Component 对象作为唯一的实参。

如果 Component 对象参与状态管理，那么此时恢复它的状态。引发 beforestaterestore 事件(可以让开发人员有机会取消整个状态恢复过程)，恢复状态，然后引发 staterestore 事件。通过将 Component 对象的配置选项 stateful 设为 true，就可以开启状态管理。通过单例类 Ext.state.Manager(稍后讨论)来访问状态。

如果已经设置配置选项 renderTo 或 applyTo，就会立即开始呈现阶段。否则，就会按需进行呈现。renderTo 用来将 Component 对象呈现为一个 DOM 节点(将其添加到现有的内容之后)。applyTo 用来指定 DOM 节点，该 DOM 节点已经包含有效的标记。applyTo 优先于 renderTo，如果同时使用了这两个配置选项，renderTo 就会被忽略。

19.2.2 呈现

当调用 render 方法时，呈现阶段就会开始。render 方法附带两个可选参数：container(可以是

Ext.Element、DOM 节点或 DOM 节点 id，用来为 Component 对象指定容纳它的 DOM 节点)和 position(指定将 Component 对象添加到容器中的什么位置)。一旦 Component 对象呈现完毕，这个 Component 对象的 rendered 属性就会返回 true。如果发现需要重新呈现某个 Component 对象，就需要将这个属性设为 false。

下面是一个示例：

(1) render 方法首先调用 beforerender 事件。开发人员可以通过处理这个事件并返回 false 值来取消整个阶段。

(2) 通常，调用 render 方法时会带上 container 实参(这个 Component 对象将用于容纳新呈现的 Component 对象)。但是，如果没有设置 container 实参，就要求 Component 对象有一个 Element 引用(在它的 el 属性中)。如果没有 container 实参，这个 Element 的 parentNode 就会用作容器(通常，el 属性已经在 initComponent 方法中完成设置)。不管是哪种情况，此时 Component 对象的 container 属性已经设置完毕。

(3) 即使 DOM 操作尚未发生，rendered 属性此时仍然将设为 true。

(4) 现在执行 onRender 方法。与 initComponent 方法类似，应该重写这个方法(确保调用基类的 onRender 方法)并在其中完成绝大部分工作。onRender 方法应该正式将 Component 对象放进 DOM 中。

(5) 根据 Boolean 类型配置选项 autoShow 的值，Component 对象现在取消隐藏(也就是将 CSS 类 x-hidden 从底层元素中移除)。添加配置选项 cls 并应用配置选项 style。

(6) 现在可以认为呈现已经完成，引发 render 事件。在此之后，调用 afterRender 方法。在默认情况下，afterRender 不执行任何操作，应该由子类重写该方法。

(7) 应用配置选项 hidden 和 disabled。这些选项用于设置 Component 对象的初始状态。很明显，设置 hidden 和 autoShow 将不必要地增加系统的负担，最终让 Component 对象隐藏起来。

(8) 最后，创建 stateEvents。配置选项 stateEvents 用于指定一组应该调用 Component 的 saveState 方法的 Component 事件。例如，GridPanel 类可以存储每一列的宽度(以及其他信息)。如果希望每次调整列宽时存储该信息，那么应该指定如下代码：

```
stateEvents: ["columnresize"]
```

19.2.3　销毁

当执行 destroy 方法时，销毁阶段就会开始。通常，destroy 方法是从框架内部调用的，并不需要直接执行。下面是一个相关的示例：

(1) 引发 beforedestroy 事件，从而允许开发人员取消整个阶段。

(2) 调用 beforeDestory 方法。在默认情况下，这个方法不做任何事情，应该由子类重写该方法。

(3) 将所有事件监听程序从 Component 对象的底层 Element 中移除并将该 Element 从 DOM 中移除。

(4) 现在底层 Element 真正被销毁，接下来调用 onDestroy 方法。在默认情况下，这个方法同样不做任何事情，应该由子类重写该方法。

(5) 将该 Component 对象从 ComponentMgr 中注销，并引发 destroy 事件。

(6) 最后，将所有事件监听程序从该 Component 对象中移除。

请注意，Component 对象只能有一个基本的 Element。如果创建了自己的 Component 子类，该类有多个 Element，那么就需要正确地将这些 Element 销毁。可以通过 beforeDestroy 或 onDestroy 方法完成该操作。

19.3 Ext.Viewport

为了便于下面几个示例的讲解，本节提前讨论 Ext.Viewport 类。Viewport 类是一个特殊化的 Ext.Panel(本章稍后将讨论它)，它应该在整个浏览器的视区中呈现自己。Viewport 直接在文档的 body 部分中呈现自己，而且当浏览器调整大小时也会自动调整自己的大小。

下面是一个完整的示例：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
    <head>
        <title>Viewport Sample</title>
        <meta content="text/html; charset=UTF-8" http-equiv="Content-Type" />
        <link type="text/css" rel="stylesheet" href="css/ext-all.css" />
    </head>
    <body>
        <form>
            <script type="text/javascript" src="js/ext-base.js"></script>
            <script type="text/javascript" src="js/ext-all.js"></script>
            <script type="text/javascript">
TheApp = {
    init: function() {
        Ext.BLANK_IMAGE_URL = "images/default/s.gif";
        new Ext.Viewport({
            items:[
                {
                    xtype: "panel",
                    border: false,
                    html: "Hello World!"
                }
            ]
        });
    }
};
Ext.onReady(TheApp.init, TheApp, true);
            </script>
        </form>
    </body>
</html>
```

19.4　Ext.Container 布局

Ext.Container 类不仅允许嵌套项，它还处理棘手的布局概念。只有当使用 Container 处理各种布局时，Ext 的 Component System 的全部功能才真正显现出来。

Ext.Container 的配置选项 layout 支持几种值，它们指定如何在 Container 中布局 item。每种可能的 layout 值代表不同的类，当呈现 Container 时实例化该类(这些类不需要由开发人员来实例化)。根据 layout 的类型，Container 中的每个 item 都可能支持额外的配置选项。此外，开发人员可以创建自己的布局类来支持自定义布局和复杂的多个 Component 对象的布局。

图 19-1 显示了内置布局类型的列表。

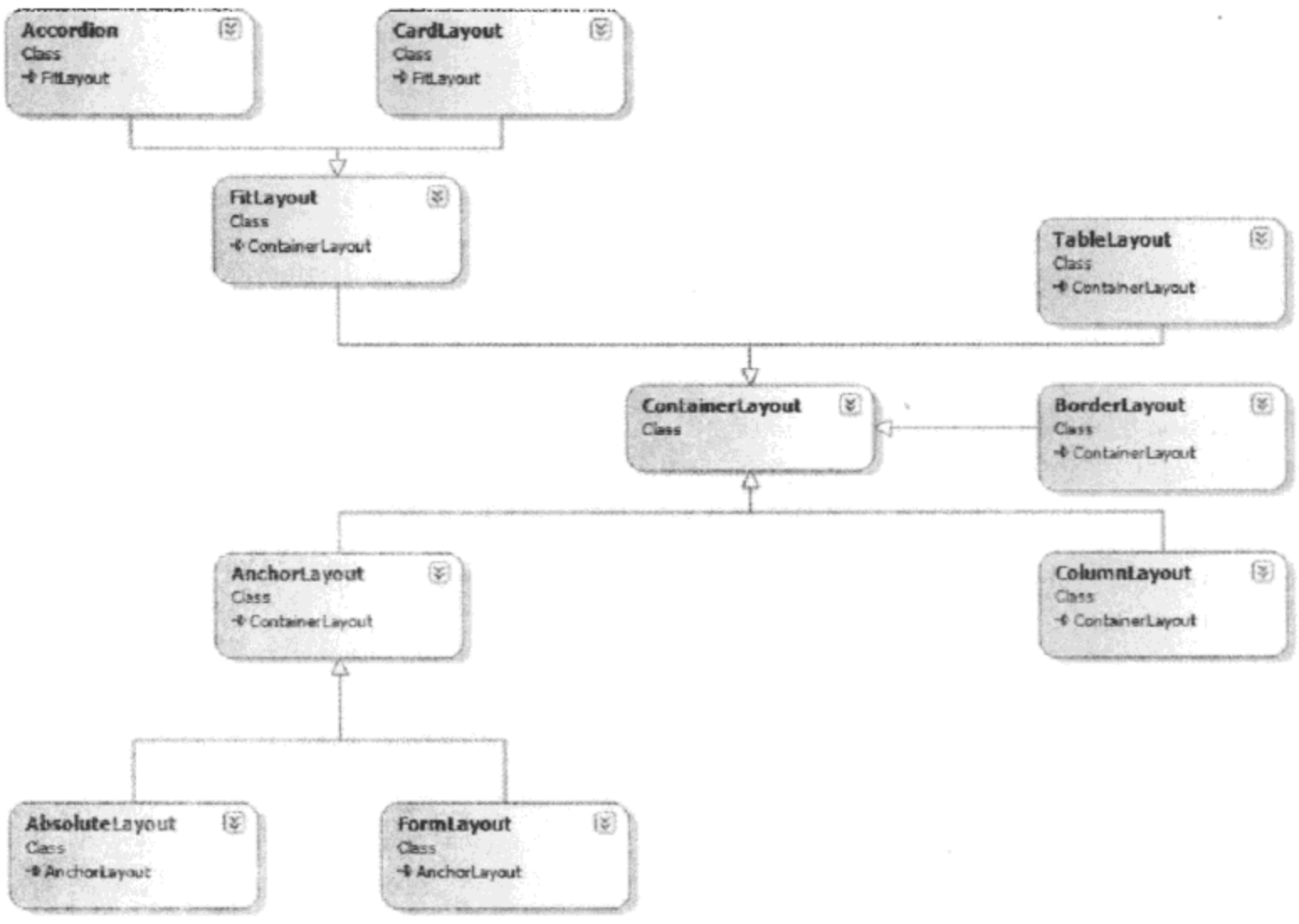


图　19-1

19.4.1　Ext.layout.ContainerLayout

如果没有指定 layout 值，就会实例化一个 Ext.layout.ContainerLayout 对象。ContainerLayout 只是按照原样显示它的全部项，它并不会试图重新格式化这些项。Container 类也支持配置选项 layoutConfig。因为 ContainerLayout 并不是由开发人员实例化，所以配置选项 layoutConfig 用来向布局传递配置选项。ContainerLayout 支持配置选项 extraCls(添加到布局中所有项的 CSS 类)和 renderHidden(用来确定在首次呈现时是否应该隐藏布局的 items 的 Boolean 值)。这个类还有一个只读属性 activeItem，但是仅用于 Accordion 和 CardLayout 类。

```
// the default layout
```

```
new Ext.Viewport({
    layout: "auto",
    layoutConfig: {
        renderHidden: true   // each nested item is hidden when rendered
    },
    items:[{  // three nested panels
        title: "Test 1",
        xtype: "panel"
    }, {
        title: "Test 2",
        xtype: "panel"
    }, {
        title: "Test 3",
        xtype: "panel"
    }]
});
```

19.4.2 Ext.layout.BorderLayout

Ext.layout.BorderLayout(派生自 Ext.layout.ContainerLayout)使用 5 个区域(北、南、东、西和中)来呈现其 Container 的各个项。每一项可以指定自己的配置选项 region(默认为 center)。在内部，BorderLayout 类为它的每一项创建一个 Ext.layout.BorderLayout.Region 对象。并不支持添加多个指定相同 region 的项，这样做将导致未知的最终行为。换言之，BorderLayout 最多只应该有 5 项(每个 Region 对应一项)。当然，每个 Region 可以有任意多个项(以及嵌套项)。

```
new Ext.Viewport({
    layout: "border",
    items:[{  // three nested panels
        title: "Center Region",
        xtype: "panel",
        region: "center"
    }, {
        title: "North Region",
        xtype: "panel",
        region: "north",
        height: 150,
        minHeight: 100,
        split: true  // this region is resizable
    }  {
        title: "East Region",
        xtype: "panel",
        collapsible: true,
        region: "east",
        width: 200
    }]
});
```

19.4.3　Ext.layout.ColumnLayout

Ext.layout.ColumnLayout(派生自 Ext.layout.ContainerLayout)允许开发人员建立多列布局。配置选项 items 中的每一项均表示一列。每一项可以指定自己的宽度(通过配置选项 width，其单位是像素)或百分比(使用配置选项 columnWidth)。百分比宽度的值介于 0 和 1 之间。在呈现时，首先考虑绝对值宽度，然后让百分比宽度占据剩余的可用空间。为了获得可预测的结果，必须将所有基于百分比宽度的列合并成一列。没有指定 width 或 columnWidth 值的项将使用其底层 Element 的 getWidth 方法(该方法返回像素数量)。然后，将该项视为具有 width 值。

```
new Ext.Viewport({
    layout: "column",
    items:[{  // three columns
        title: "Test 1",
        xtype: "panel",
        width: 250  // this column is exactly 250 pixels
    }, {
        title: "Test 2",
        xtype: "panel",
        columnWidth: .2  // this column is 20% of the remaining total width
    }, {
        title: "Test 3",
        xtype: "panel",
        columnWidth: .8  // this column is 80% of the remaining total width
    }]
});
```

19.4.4　Ext.layout.TableLayout

Ext.layout.TableLayout(派生自 Ext.layout.ContainerLayout)使用标准的 HTML 表提供传统的 rowspan 和 colspan 功能。TableLayout 类要求使用配置选项 columns，这个选项指定表将具有的列数，并通过 Container 的配置选项 layoutConfig 传给 TableLayout 构造函数。开发可以根据需要在每一项上指定 rowspan 和 colspan 值。当呈现每一项时，就会呈现一个 HTML td 元素来容纳该项。开发人员可以使用配置选项 cellCls(应该包含一个 CSS 类名)来调整表单元格的外观(而不是项本身的外观)。要通过编程控制表单元格，开发人员可以使用配置选项 cellId。

```
new Ext.Viewport({
    layout: "table",
    layoutConfig: {
        columns: 3  // this option is passed to the TableLayout class
    },
    items:[{ // three columns, two rows
        title: "Test 1",
        xtype: "panel",
        cellId: "theFirstCell",  // for easy recall
        rowspan: 2,  // this item takes up column 1 of rows 1 and 2
        html: "rowspan = 2"
    }, {
```

```
            title: "Test 2",
            xtype: "panel",
            colspan: 2,  // this item takes up columns 2 and 3 of row 1
            html: "colspan = 2"
        }, {
            title: "Test 3",  // this item takes up column 2 of row 2
            xtype: "panel"
        }, {
            title: "Test 4",  // this item takes up column 3 of row 2
            xtype: "panel"
        }]
});
var myCell = Ext.get("theFirstCell");
```

19.4.5 Ext.layout.AnchorLayout

Ext.layout.AnchorLayout(派生自 Ext.layout.ContainerLayout)可以让开发人员创建非常易于改变的设计。每一项都固定到容器的左侧并垂直堆叠。每一项的宽度和高度均可以使用配置选项anchor(含有以空格隔开的两个值的字符串)来指定。配置选项 anchor 中的值可以是距离容器右下侧的偏移量(以像素数量指定)，也可以是占容器总宽度的百分比。要在容器内建立一个“虚拟”空间，开发人员可以使用配置选项 anchorSize。anchor 值现在基于 anchorSize 计算(而不是基于容器的实际大小)。

```
new Ext.Viewport({
    layout: "anchor",
    anchorSize: {  // a virtual "box"
        width: 800,
        height: 800
    },
    items:[{  // three nested panels
        title: "Test 2",
        xtype: "panel",
        anchor: "-200 -600" // 200 pixels from the right, 600 from the bottom
    }, {
        title: "Test 1",
        xtype: "panel",
        anchor: "50% 25%" // 50% of the width, 25% of the height
    }, {
        title: "Test 3",
        xtype: "panel",
        anchor: "-10 5%" // 10 pixels from the right, 5% of the height
    }]
});
```

19.4.6 Ext.layout.AbsoluteLayout

Ext.layout.AbsoluteLayout(派生自 Ext.layout.AnchorLayout)结合使用每一项的配置选项 x 和 y 以及配置选项 anchor。

```
new Ext.Viewport({
    layout: "absolute",
    anchorSize: {  // a virtual "box"
        width: 800,
        height: 800
    },
    items:[{  // two nested panels
        title: "Test 1",
        xtype: "panel",
        anchor: "-200 50%",
        x: 50,
        y: 50
    }, {
        title: "Test 2",  // height and width are based on the panel's contents
        xtype: "panel",
        x: 100,
        y: 100
    }]
});
```

19.4.7　Ext.layout.FormLayout

Ext.layout.FormLayout(派生自 Ext.layout.AnchorLayout)让开发人员控制传统的表单数据录入屏幕页面(标签-值对)的外观。为了实现这种控件，已经把配置选项添加到布局、容器以及每一项中。

下面是布局的配置选项：

- elementStyle　允许开发人员指定 CSS 样式规范字符串(该样式将会应用于所有字段值)。
- labelSeparator　允许开发人员指定在标签和值之间显示的文本(默认为冒号)。
- labelStyle　允许开发人员指定 CSS 样式规范字符串(该样式将会应用于所有字段标签)。

下面是添加到容器的配置选项：

- hideLabels　这个 Boolean 值决定是否将所有标签和标签分隔符隐藏起来。
- itemCls　这个选项允许开发人员指定一个 CSS 类，这个类将应用于用来包装标签和值的特殊 div 标记。
- labelAlign　在默认情况下，标签呈现在值的左边。这个选项可以接受“top”值，从而在值的上方呈现标签。
- labelPad　与 labelWidth 结合使用，这个选项指定标签和它们的值之间的像素距离(默认为 5)。如果 labelAlign 设为“top”，这个选项就不会起作用。
- labelWidth　这是标签的总像素宽度(默认为 100)。如果 labelAlign 设为“top”，这个选项就不会起作用。

下面是添加到每一项的配置选项：

- clearCls　允许开发人员指定一个 CSS 类，该类将应用于字段左侧的特殊空 div 标记。
- fieldLabel　包含该项的标签的文本。
- hideLabel　这个 Boolean 值将决定是否隐藏该项的标签和标签分隔符。

- **itemCls** 允许开发人员指定一个 CSS 类，该类将应用于用来包装标签和值的特殊 div 标记(如果指定的话，就会重写容器的配置选项 itemCls)。
- **labelSeparator** 允许开发人员指定标签和值之间显示的文本(如果指定的话，就会重写布局的配置选项 labelSeparator)。
- **labelStyle** 允许开发人员指定一个 CSS 样式规范字符串，该样式将应用于标签(如果指定的话，就会重写布局的配置选项 labelStyle)。

```
new Ext.Viewport({
    layout: "form",
    layoutConfig: {
        labelSeparator: ","
    },
    labelAlign: "top",
    items:[{  // three nested text boxes
        fieldLabel: "Test 1",
        xtype: "textfield"
    }, {
        fieldLabel: "Test 2",
        xtype: "textfield"
    }, {
        fieldLabel: "Test 3",
        xtype: "textfield",
        labelSeparator: ":" // overrides layoutConfig
    }]
});
```

虽然能够进行所有这些细粒度的控制，但是 FormLayout 类仅仅垂直地堆叠所有项。如果开发人员需要更复杂的布局，那么只需要将各种布局组合起来使用即可。

```
new Ext.Viewport({
    layout: "column",
    items: [{  // two columns of text boxes with labels
        xtype: "panel",
        title: "Column 1",
        columnWidth: .5,
        layout: "form",  // this column has a form layout
        items: [{
            xtype: "textfield",
            fieldLabel: "Textbox 1"
        }, {
            xtype: "textfield",
            fieldLabel: "Textbox 2"
        }]
    }, {
        xtype: "panel",
        title: "Column 2",
        columnWidth: .5,
        layout: "form",  // this column has a form layout
        items: [{
```

```
            xtype: "textfield",
            fieldLabel: "Textbox 3"
        }, {
            xtype: "textfield",
            fieldLabel: "Textbox 4"
        }]
    }]
});
```

还应该注意，Ext.layout.FormLayout 类并没有实现表单的行为(数据绑定、输入验证等)。这些任务独立于它们的逻辑位置，我们将在以后的章节中讲解。

19.4.8　Ext.layout.FitLayout

Ext.layout.FitLayout(派生自 Ext.layout.ContainerLayout)强制容器只显示一项，并且扩展该项以填充容器的整个宽度和高度。如果添加了多项，那么剩余的项也会呈现，但是绝对不会显示出来：

```
new Ext.Viewport({
    layout: "fit",
    items:[{
        title: "Test 1",
        xtype: "panel"
    }]
});
```

19.4.9　Ext.layout.Accordion

Ext.layout.Accordion(派生自 Ext.layout.FitLayout)允许开发人员将多项添加到容器中，但是每次只显示一项。其他项的标题可以垂直地堆叠。每一项的标题栏上都添加了一个折叠工具栏按钮(这个按钮位于其他按钮的右侧)。ContainerLayout 的 activeItem 属性设为选中的新项。

布局支持如下配置选项以定制外观和行为：

- activeOnTop　这个 Boolean 值指示新选中的项是否应该移到可折叠页面的顶部(默认为 false)。
- animate　这个 Boolean 值指示当选中新项时是否显示动画(默认为 false)。
- autoWidth　这个 Boolean 值指示是否强制把所有项的宽度设为容器的宽度(默认为 true)。
- collapseFirst　这个 Boolean 值指示是否在其他按钮的左侧呈现折叠工具栏按钮(默认为 false)。
- fill　这个 Boolean 值指示是否强制把所有项的高度设为容器的高度(默认为 true)。
- hideCollapseTool　这个 Boolean 值指示是否隐藏折叠工具栏按钮(默认为 false)。
- titleCollapse　这个 Boolean 值指示是否允许用户单击每一项的工具栏上的任意位置(默认为 true)。如果 hideCollapseTool 和 titleCollapse 均为 false，那么用户将不能交换项。

```
new Ext.Viewport({
    layout: "accordion",
    layoutConfig: {
        animate: true
```

```
    },
    items:[{  // three nested panels
        title: "Test 1",
        xtype: "panel"
    }, {
        title: "Test 2",
        xtype: "panel"
    }, {
        title: "Test 3",
        xtype: "panel"
    }]
});
```

19.4.10 Ext.layout.CardLayout

Ext.layout.CardLayout(派生自 Ext.layout.FitLayout)允许添加多项，但每次只显示一项。但是，要在两项之间进行交换，开发人员必须执行 setActiveItem 方法。这个类构成了向导形式界面或定时幻灯片展示的极佳基础。开发人员必须确保设置 Container 的配置选项 activeItem。否则，在默认情况下不会显示任何内容。

```
var viewport = new Ext.Viewport({
    layout: "card",
    layoutConfig: {
        deferredRender: true  // render each card as it is selected
    },
    activeItem: 0, // make sure there is default active item
    items:[{  // three nested panels
        title: "Test 1",
        xtype: "panel",
        html: "Content Page 1"
    }, {
        title: "Test 2",
        xtype: "panel",
        html: "Content Page 2"
    }, {
        title: "Test 3",
        xtype: "panel",
        html: "Content Page 3"
    }]
});
viewport.layout.setActiveItem(1);  // display the second card
```

19.4.11 创建自定义布局

除了这些内置布局类之外，创建自定义布局类也是一件简单的事情。为了实现这种高效的、可扩展的呈现机制，Ext.Container 和 Ext.ContainerLayout 类紧密联系。容器布局由 Ext.Container 的 render 方法实例化和执行。与 Component 生命周期非常相似，布局类也有一种可预测的工作模式。但是，我们还需要了解一些内部工作原理：

- constructor(layoutConfig)　Container 的配置选项 layoutConfig 传给 Layout 的构造函数。
- setContainer(container)　这个方法初始化 Layout。通常，这个方法由 Layout 用来向 Container 附加事件监听程序。
- setActiveItem(item)　如果设置 Container 的配置选项 activeItem，Container 就会调用这个方法。但是，当前只有 CardLayout 支持这个方法。

此时，Container 的 render 方法调用 doLayout 方法。doLayout 重新计算 Container 的布局以及所有嵌套 Container，直到所有 Container 都重新计算完毕为止。这个方法对于刷新屏幕页面中的选中部分非常有用。doLayout 方法调用 Layout 的如下方法。

- layout()　这个方法调用 Container 的 getLayoutTarget()方法，然后调用 Layout 的 onLayout 方法。当这个过程完成之后，引发 afterlayout 事件。
- onLayout(container, target)　这个方法只是调用 renderAll(container, target)方法，而该方法又调用 renderItem(container, position, target)方法。这些呈现方法负责在每一项(Ext.Component)上执行 render 方法，这实际上是将它们放入 DOM 中。通过在项外部包装自己的 HTML 代码片段，我们可以轻易地定制最终的布局。

最后，Container 的销毁阶段将执行 Layout 的 destroy()方法(完成所有必要的清理工作)。

可以看出，创建自己的自定义布局是一件简单的事情：派生 Ext.layout.ContainerLayout 类(或任何其他内置布局类型)，然后重写必要的方法。为了让自己的自定义布局能够被 Container 的配置选项 layout 识别，我们还需要创建一个名称，并将其添加到 Ext.Container.LAYOUTS 集合中：

```
// this makes "mine" a valid layout value
Ext.Container.LAYOUTS['mine'] = Ext.ux.layout.MyLayout;
```

19.5　面板和窗口

下面讨论面板和窗口。

19.5.1　Ext.Panel

Ext.Panel 类派生自 Ext.Container，添加了用户期望富 GUI 应用程序具有的几个有用功能。面板可以拖动和调整大小，带有工具栏和圆角，而且带有页眉、页脚和主体。此外，作为 Container，它们具备前面讨论嵌套 Component 时提及的所有布局以及它们的所有功能。

Panel 类也带有几个内置“工具”按钮。这些工具按钮在默认情况下不执行任何操作，开发人员必须提供一个包含相关功能的处理程序。但是，该类提供这些按钮的原因是，这样开发人员就有一个方便的储存库来无缝地适合选中 UI 主题的其余部分。

此外，我们还能够更新 Panel，在第 20 章中将讲解如何执行这种更新。现在，我们先来查看实现一个复杂的 Panel 是多么简单的一件事情。下面的示例展示了一个可拖动和可折叠的面板，该面板带有页眉(包含标题和所有的工具)，在主体上方有一个工具栏，而在主体下方有一个工具栏和页脚(带有一个 Ext.Button)：

```
var toolClick = function(event, toolEl, panel) {
    // do something here
};
```

```
var layout = new Ext.Viewport({
    layout: "absolute",
    items:[{
        bbar: ["Bottom Toolbar"],
        buttons: [ // this creates the footer automatically
            {text: "Footer"}
        ],
        collapsible: true,
        draggable: true,
        frame: true, // adds a frame around the entire panel
        height: 300,
        html: "Here's the body",
        tbar: ["Top Toolbar"],
        title: "Header", // this creates the header automatically
        tools: [
            // the "toggle" tool is created automatically when collapsible=true
            // {id: "toggle", qtip: "toggle", handler: toolClick},
            {id: "close", qtip: "close", handler: toolClick},
            {id: "minimize", qtip: "minimize", handler: toolClick},
            {id: "maximize", qtip: "maximize", handler: toolClick},
            {id: "restore", qtip: "restore", handler: toolClick},
            {id: "gear", qtip: "gear", handler: toolClick},
            {id: "pin", qtip: "pin", handler: toolClick},
            {id: "unpin", qtip: "unpin", handler: toolClick},
            {id: "right", qtip: "right", handler: toolClick},
            {id: "left", qtip: "left", handler: toolClick},
            {id: "up", qtip: "up", handler: toolClick},
            {id: "down", qtip: "down", handler: toolClick},
            {id: "refresh", qtip: "refresh", handler: toolClick},
            {id: "minus", qtip: "minus", handler: toolClick},
            {id: "plus", qtip: "plus", handler: toolClick},
            {id: "help", qtip: "help", handler: toolClick},
            {id: "search", qtip: "search", handler: toolClick},
            {id: "save", qtip: "save", handler: toolClick},
            {id: "print", qtip: "print", handler: toolClick}
        ],
        width: 500,
        x: 100,
        xtype: "panel",
        y: 100
    }]
});
```

除了从 Container、BoxComponent 和 Component 继承所有的属性、方法和事件之外，Ext.Panel 类还增加了开发人员期望的几个事件和方法。诸如 collapse、expand、addButton 和 setTitle 这样的方法可以帮助开发人员处理日常任务。但是，还有 getTopToolbar、getInnerHeight、getFrameWidth 等方法，它们为开发人员提供了一个自然的对象模型来满足最直接的需求。Panel 的大部分功能在本章中都已经介绍过。

19.5.2　Ext.Window

Ext.Window 类直接构建在 Ext.Panel 的基础之上。Window 类更加特殊化，它具有的内置功能也稍微多一些。Window 可以是模态窗口，可以调整大小，并且限制在浏览器视区中。还有几个内置工具具有默认的功能(最小化、最大化、关闭等)。

当关闭 Window 时，默认的操作是销毁该 Window 对象并将其从 DOM 中移除。但是，许多应用程序的 Window 会重复使用。在这种情况下，开发人员可以简单地将 Window 的配置选项 closeAction 设置为“hide”。这个选项不会销毁 Window，而只是将其隐藏起来。show 方法将恢复该 Window。这可以极大地减少重新创建大量控件所需的系统开销。

因为 Window 传统上是浮动的，所以 Ext.Window 类有几个与定位有关的方法。alignTo、anchorTo、center、toBack 和 toFront 这几个辅助函数更适合于传统 Window。

或许最让终端用户感到兴奋的是，Ext.Window 类有一个配置选项 manager。manager 配置选项接受一个 Ext.WindowGroup 实例(默认情况下，该项设为 Ext.WindowMgr)。

19.5.3　Ext.WindowGroup

WindowGroup 只是 Window 实例的集合类。当创建一个 Window 对象时，它自动添加到 WindowGroup 中。当销毁 Window 对象时，它将从该组中移除。WindowGroup 类管理该组的“活跃”Window，以及该组中所有 Window 的叠放顺序(z-order)。还有几个辅助函数，分别用于遍历该组(each)，处理叠放顺序(sendToBack 和 bringToFront)，以及检索特定 Window(get 和 getBy)。所有这些方法都与之前讨论的方法类似。

19.5.4　Ext.WindowMgr

Ext.WindowMgr 类是 Ext.WindowGroup 的全局单例实例。如果开发人员决定不创建自己的 WindowGroup，那么可以将该组用作所有 Window 对象的默认管理器。通过创建自己的 WindowGroup，我们可以轻易地在同一个浏览器页面中创建多个“桌面”。

19.6　本章小结

在前面几章中，我们学习了建立 Ext JS 库基础的构建块，而本章是针对该库自身的介绍。虽然大部分 JavaScript 库仅能作为某种考虑周全的基础库，但是 Ext JS 却真正做了一些事情。通过使用诸如 Ext.DomQuery、Ext.DomHelper 和 Ext.Template 这样的类，Ext Component 系统为完整的窗口部件库建立了健壮而简单的基础。我们已经在诸如 Ext.Panel 和 Ext.Window 这样的类中看到了简洁的、有组织的对象层次结构带来的全部好处。

在下一章中，我们将开始深入讨论能够帮助开发人员向服务器发送数据、从服务器获取数据以及在浏览器中进行数据建模的类。

第20章

数据处理以及服务器通信

HTML 和 Internet 的主要作用通常总是关于分发数据。无论是科学文档、家庭影集，还是视频新闻剪辑，几乎所有的数据形式都可以在 Web 上找到其身影。随着数据的增长，Web 的性能不断降低。为了打开家庭影集的第二页，我们就要等待 5 秒，这个时间过于漫长。虽然有关在不重新加载整个页面的情况下加载数据的技术已经出现将近十几年的时间，但是直到最近几年这些技术才真正充分发挥其作用。

现在，Internet 上的每种 JavaScript 框架都增加了一些用于执行这些 AJAX 任务的辅助函数，Ext JS 也不例外。但 Ext JS 的作者意识到 AJAX 技术并没有解决所有的数据访问问题。获取数据只是问题的一部分。诸如数据格式、数据模式以及数据反序列化之类的问题均在 Ext JS 中得到解决。

本章内容简介：

- 获取数据
- 重新建模数据
- 本地存储数据
- 集成所有类

我们还将了解 Ext JS 如何简化整个数据访问问题。用来解决主要数据访问问题的类有 3 种。我们将研究用来检索数据的类(代理)、用来反序列化并建模数据的类(读取器)以及缓存重新建模的数据的类(存储)。所有这些类都彼此紧密关联，并且协同工作以实现高性能且灵活的数据引擎。

20.1 获取数据

前面曾经提到，AJAX 技术并没有解决所有的数据访问问题。确切地说，AJAX 依赖浏览器的 XMLHttpRequest 对象。AJAX 技术不会从当前页面所在服务器之外的其他服务器上检索数据。但是，开发人员希望所有类型的数据访问技术都有一致的接口。为了实现这种一致性，Ext JS 的作者创建了 Ext.data.DataProxy 抽象基类。

20.1.1 Ext.data.DataProxy

DataProxy 类本身不完成任何工作(参见图 20-1)。它只是提供了两个可供子类引发的事件(beforeload 和 load)。DataProxy 的所有子类各自实现特定的数据访问技术。当 DataProxy 子类引发这些事件时，它们应该遵循如下的签名：

- beforeload(this, params) this 是 Ext.data.DataProxy 的当前实例，而 params 通常是发送给服务器的参数。可以通过在事件监听程序中返回 false 来取消该事件。
- load(this, o, arg) this 是 Ext.data.DataProxy 的当前实例，o 是从服务器上加载的数据，而 arg 是开发人员提供的对象。

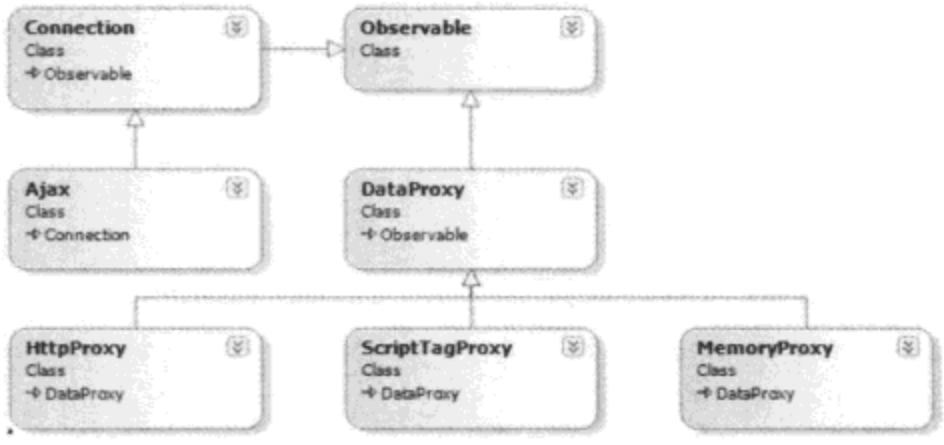


图 20-1

20.1.2 Ext.data.HttpProxy

既然 AJAX 是大多数开发人员使用的技术，因此我们首先查看 Ext.data.HttpProxy 类。

HttpProxy 类提供了一个方法：

load : function(params, reader, callback, scope, arg)——params 是发送给服务器的参数，reader 是 Ext.data.DataReader 类的一个实例，callback 是与服务器的通信结束时调用的方法，scope 确定这个回调方法的执行作用域，而 arg 则是传给这个回调方法的开发人员自定义对象。这个回调方法具有如下 3 个参数: result(成功反序列化时返回的对象)、arg(load 方法的可选参数 arg)和 success(一个 Boolean 值，指示整个加载过程是否成功完成)。

HttpProxy 类提供了一个新事件：

loadexception : function(this, options, response, e) ——this 为 Ext.data.HttpProxy 的当前实例，options 是一个对象，该对象包含所有要发送给 HttpProxy 的 load 方法的值；response 是真正执行服务器通信的浏览器 XMLHttpRequest 对象；而 e 要么是 null，要么是在反序列化过程中遇到的 Error。

当调用 load 方法时，HttpProxy 通过 Ext.data.Connection 类的一个实例来完成它的所有服务器通信任务。HttpProxy 构造函数有选择地接受 Connection 类的一个实例或一个对象，该对象将传给 Ext.Ajax.request 方法。Ext.Ajax 是 Ext.data.Connection 的一个全局实例，在没有明确的 Connection 对象的场合中可以使用该实例。

一旦调用 HttpProxy 的 load 方法，就会引发它的 beforeload 事件，从而让开发人员有机会取

消加载过程。如果没有取消，那么加载过程就会继续，从而执行 Connection 对象的 request 方法。request 方法是对服务器的异步调用，我们将在下面的小节中更加深入地讲解它。如果 request 方法执行成功，HttpProxy 就会继续执行，调用指定的 reader(Ext.data.DataReader 的一个实例)来反序列化数据。如果 DataReader 执行成功，就会引发 HttpProxy 的 load 事件，否则就会引发 HttpProxy 的 loadexception 事件。最后，执行指定的 callback 方法。

```
var proxy = new Ext.data.HttpProxy({  // we're not using an explicit Connection
    url: "getData.html"
});
proxy.on({
    beforeload: {
        fn: function(proxy, params) {
            // here's a chance to cancel the load process
        }
    },
    load: {
        fn: function(proxy, data, arg) {
            // loading has completed successfully
        }
    },
    loadexception: {
        fn: function(proxy, options, response, e) {
            if (e) {
                // there has been a deserialization error (examine the e argument)
            } else {
                // there has been a server error (examine the response argument)
            }
        }
    }
});
proxy.load(params, reader, callback, scope, arg);
```

1. Ext.data.Connection 和 Ext.Ajax

Ext.data.Connection 类是一个标准的 AJAX 类，它是 Ext JS 的服务器连接功能的基础。Ext.data.HttpProxy 类在内部使用该类访问服务器。但是，Connection 类还可以单独用于对数据访问进行更多的控制。所有发送到服务器的请求都是通过一个简单而灵活的方法来异步完成的，该方法用于附加回调。Ext.Ajax 类是 Connection 类的全局实例，它为开发人员提供了便利。下面是一个简单的单行服务器调用：

```
Ext.Ajax.request({
    url: "getData.html",
    callback: function(options, success, response) {
        if (success) {
            alert("Yea!");
        } else {
            alert("Boo!");
        }
    }
});
```

可以看到，response 参数总是传给 callback 方法。在前面的示例中，开发人员检查 success 标志。除了通用的 callback 方法之外，还有 success 方法和 failure 方法(不需要 success 标志)。

Connection 类可用于多次请求，而且可以重用。其构造函数接受的配置选项可重复用于对请求方法的所有调用中。下面是该构造函数的所有配置选项的清单：

- autoAbort　该 Boolean 值指示是否终止所有正待处理的请求(默认为 false)。
- defaultHeaders　该对象包含要添加到每次请求中的报头(默认为 undefined)。
- disableCaching　该 Boolean 值指示在建立请求时是否向 QueryString 中添加一个时间戳(这会让浏览器缓存失效)。只有当 method 选项设为“GET”时才使用该配置选项(默认为 true)。
- disableCachingParam　QueryString 时间戳名称的字符串值，它会禁用缓存(默认为“_dc”)。
- extraParams　该对象包含要添加到每次请求中的值(默认为 undefined)。
- method　每次请求使用的 HTTP 方法(默认为“GET”，但是如果有 params，那么默认为“POST”)。
- timeout　在终止请求前允许等待的时间(单位为毫秒，默认为 30 000，也就是 30 秒)。
- url　服务器资源的位置。该资源必须位于当前页面的起源域中。

request 方法也接受一个配置选项对象。request 方法将重写构造函数中所有产生冲突的配置选项。下面是 request 方法的选项：

- headers　一个包含 HTTP 报头值的对象(默认为 undefined)。
- disableCaching　用来重写构造函数中的 disableCaching 选项。
- params　这个对象包含要添加到请求中的值(默认为 undefined)。
- method　用来重写构造函数中的 method 选项。
- timeout　用来重写构造函数中的 timeout 选项。
- url　用来重写构造函数中的 url 选项。
- callback　当请求完成或失败时调用该方法。
- success　当请求完成时调用该方法。
- failure　当请求失败时调用该方法。
- scope　该对象是 callback、success 和 failure 方法的作用域。
- form　表示需要序列化的 HTML 表单标记的 Ext.Element、HTMLElement 或 DOM 节点 ID。
- isUpload　将其设为 true 以允许文件上传，更多相关信息请参见下一节。
- xmlData　在请求的正文中发送的 XML 文档(如果同时使用 params，那么它们将附加到 QueryString)。该选项不能与 jsonData 结合使用。
- jsonData　在请求的正文中发送的 JavaScript 字符串或对象(如果同时使用 params，那么它们将附加到 QueryString)。该选项不能与 xmlData 结合使用。

这个配置对象总是与构造函数的配置对象合并，然后传给回调方法(callback、success 和 failure)。当回调方法需要一些不必发送给服务器的信息时，这个配置对象就非常有用：

```
var conn = new Ext.data.Connection();
conn.request({
   myValues: { // these values are NOT sent to the server
```

```
        value1: "test"
    },
    url: "getData.html",
    params: { // these values ARE sent to the server
        dataType: "user"
    },
    callback: function(options, success, response) {
        if (success) {
            // all the options are still here
            alert(options.myValues.value1);
        }
    }
});
```

Connection 类也派生自 Ext.util.Observable，并提供了 3 个事件。所有从 Connection 的当前实例中执行的请求都会引发这些事件：

- beforerequest(Connection conn, Object options)　在即将把请求发送给服务器之前引发该事件。在事件监听程序中返回 false，将导致该事件被取消(不会把请求发送给服务器)。
- requestcompleted(Connection conn, Object response, Object options)　当从服务器那里接收到成功响应时立即引发该事件。
- requestexception(Connection conn, Object response, Object options)　当从服务器那里接收到失败响应时立即引发该事件。

在所有这 3 个事件中(以及 3 个回调方法中)，response 参数均是浏览器的原生 XMLHttpRequest 对象。这个对象包含开发人员能够访问的所有原始信息。request 方法还返回一个事务 ID，可以将该事务 ID 传回给 Connection 的 isLoading 或 abort 方法。与往常一样，开发人员可以完全控制该行为。

2. 文件上传

执行文件上传是 AJAX 的薄弱环节。由于一些原因，XMLHttpRequest 对象并不支持文件附件。为了绕开这个问题，Ext JS 创建了一个隐藏 IFRAME，通过该 IFRAME 来发送数据(以及所有文件附件)，一旦请求完成，就立即销毁该 IFRAME。Ext JS 还创建了一个仿造的 XMLHttpRequest 对象，并使用服务器返回的数据填充该对象。这为开发人员提供了一个针对所有请求的一致性对象模型(无论它们通过 AJAX 还是 IFRAME 发送数据)。

但是，这种折中方法可能需要开发人员给予特别的注意。当 IFRAME 请求返回 JSON 数据时，有必要采用 MIME 类型“text/html”发送数据。这将告诉浏览器不要处理数据，而是由 Ext JS 进行必要的分析。此外，上传文件功能要求浏览器使用“multipart/form”内容类型，而这对于某些 Web 服务器来说可能引起问题。这些都是浏览器 HTTP 平台必然存在的问题，可以在 Web 服务器上轻易地解决它们。

20.1.3　Ext.data.MemoryProxy

MemoryProxy 类目前是最简单的代理类，它的唯一作用就是用来支持数据访问的一致接口。MemoryProxy 访问保存在本地 JavaScript 对象中的数据。尽管 MemoryProxy 非常简单，但是从第

三方 JavaScript 或 Flash 库中加载数据时，它可能非常有用。

MemoryProxy 的构造函数接受 JavaScript 对象参数(需要加载的对象)。这个类还提供了 load 方法和 loadexception 事件，两者的签名与 HttpProxy 中相同。MemoryProxy 类的 load 方法不需要 params 实参，它的 loadexception 事件也不需要 response 实参，但是出于一致性和可扩展性的目的，它们仍然可用。然而，该类并没有 beforeload 和 load 事件。

```
var data = [
    {
        FirstName: "Bob",
        LastName: "Smith"
    }, {
        FirstName: "Gary",
        LastName: "Robertson"
    }
];
var proxy = new Ext.data.MemoryProxy(data);
proxy.on({
    loadexception: {
        fn: function(proxy, options, response, e) {
            // response will always be null
            if (e) {
                // there has been a deserialization error
            }
        }
    }
});
proxy.load(null, reader, callback, scope, arg);
```

20.1.4 Ext.data.ScriptTagProxy

Ext.data.ScriptTagProxy 解决了第三个数据访问问题：从远程服务器检索数据。在后台中，ScriptTagProxy 类创建了一个隐藏的回调方法并在 QueryString 上将回调方法的名称传给远程服务器。然后，远程服务器生成有效的 JavaScript 代码来执行这个回调方法，并把 JSON 格式化数据传给它。下面是一些服务器端的伪代码样本：

```
Response.ContentType = "text/javascript";
String theData = getData().toJson();
String callbackName = Request.QueryString["callback"];
Response.Write(callbackName + "(" + theData + ");");
```

当然，在浏览器端使用 ScriptTagProxy 与使用其他代理非常类似：同样有 load 方法(与 HttpProxy 一样)和 loadexception 事件，beforeload 和 load 事件也以预期的方式表现。唯一的特别之处在于 loadexception 事件将 response 参数替换成 arg 参数(该参数将传给 load 方法)。一旦检索到数据(或者发生超时)，隐藏的回调方法就会被销毁。

ScriptTagProxy 构造函数接受一个带有如下选项的配置对象：

- url　远程服务器资源的位置。
- timeout　代理放弃之前允许等待的时间(单位为毫秒，默认为 30 000)。
- callbackParam　QueryString 参数的名称，它将包含隐藏 JavaScript 回调方法的名称(默认为“callback”)。
- nocache　该 Boolean 值指示是否向 QueryString 中添加一个时间戳以使该请求唯一，这会让浏览器缓存失效(默认为 true)。

load 方法接受的参数与 HttpProxy 的 load 方法相同，而且两者的行为方式也相同。唯一的不同之处在于，在发送到远程服务器时，params 实参将序列化成 JSON 并放到 QueryString 中(实际上，这个数据请求必须是 HTTP GET)。

```
var proxy = new Ext.data.ScriptTagProxy({
    url: "www.remoteServer.com"
});
proxy.on({
    beforeload: {
        fn: function(proxy, params) {
            // here's a chance to cancel the load process
        }
    },
    load: {
        fn: function(proxy, data, arg) {
            // loading has completed successfully
        }
    },
    loadexception: {
        fn: function(proxy, options, arg, e) {
            if (e) {
                // there has been a deserialization error (examine the e argument)
            } else {
                // there has been a server error
            }
        }
    }
});
proxy.load(params, reader, callback, scope, arg);
```

20.2　重新建模数据

前面曾经讨论过，每个 DataProxy 子类都有一个 load 方法，该方法接受一个 reader 实参，这个 reader 实参是用来完成数据反序列化的对象。为了进行反序列化，读取器需要了解正在处理的数据模型。这就是 Ext.data.Record 类发挥作用的地方。

20.2.1　Ext.data.Record

Ext.data.Record 类也是一个抽象基类，但它使用了工厂模式来创建新的 Record 类型。Record

类具有双重作用。它可用于定义数据行(建模)，还可以用于存储单个数据行的原始值和修改后的值(反序列化)。此处我们将讨论建模，而 Record 类的反序列化机制留待本章稍后讨论。

重点是要理解 Record 定义非常通用。所有的定义都可以定制，但是它们必须具备相似的行为方式。由于这个原因，使用工厂模式创建 Record 类型是有意义的行为。要创建新的自定义 Record，应该执行 Ext.data.Record.create 方法，该方法接受一个字段定义数组作为参数。

1. Ext.data.Field

在内部，字段定义用来创建一个 Ext.data.Field 实例数组。除了在 Ext.data.Record 中用到 Field 类之外，在 Ext 库中的其他地方并不使用它，但是每个 Field 都包含一个自定义的 convert 方法来把原始数据值转换成一些更加合适的值。这个 convert 方法通常基于数据的 type 构造，但是在字段定义中可以轻易地将其改写：

```
var userRecordDefinition = Ext.data.Record.create([
    {name: "FirstName", mapping: "FName", type: "string"},
    {name: "LastName", mapping: "LName", type: "string"},
    {name: "DateOfBirth", mapping: "DOB", type: "date", dateFormat: "m/d/Y"},
    {name: "FavoriteColor", defaultValue: "Green"},
    {name: "ShirtSize", mapping: "size", convert: function(value, row) {
        switch (value) {
            case 1:
                return "X-Large";
            break;
            case 2:
                return "Large";
            break;
            case 3:
                return "Medium";
            break;
            default:
                // we have access to the original data row
                if (!row["DOB"]) {
                    return "Small";
                }
            break;
        }
    }}
]);
```

在上面的示例中可以看到，每个 Field 定义都有如下的属性(name 是唯一必需的值)：

- **name** 开发人员在自己的代码中使用该名称访问这个字段的值。
- **mapping** 这个方法获取相对于其记录的根的值，这个值特定于读取的数据的类型。例如，XmlReader 类期望具有路径表达式(由 Ext.DomQuery 对象理解)，如“firstName”或“name/first”(注意，支持层次结构数据)。如果 mapping 值与 name 值相同，那么可以排除 mapping。

- type　这是用于默认转换的数据类型，可用的选项有 auto(不转换)、string、int、float、boolean 和 date。
- sortType　该属性定义在根据当前 Field 排序时应该如何对记录进行排序。Ext.data.SortTypes 类中包含了所有有效值(更多相关细节请参见下一节)。默认值取决于 type 值。
- sortDir　ASC(升序)或 DESC(降序)，默认为 ASC。
- convert　获取可显示值的自定义转换函数。

2. Ext.data.SortTypes

Ext.data.SortTypes 是一个简单的辅助类，它所包含的函数可用来在出于排序目的而对原始数据值进行比较之前执行转换。重点是要理解，为了排序而进行的数据转换与为了显示而进行的数据转换有所不同。例如，对于终端用户来说，需要使用非常特殊的方式格式化日期数据，而同样的格式化不会作为逻辑命令用于排序。下面给出了内置的排序转换函数：

- none　不要转换值。
- asText　将所有 HTML 标记从值中去除。
- asUCText　将所有 HTML 标记去除并转换为大写(进行不区分大小写的排序)。
- asUCString　转换为大写(进行不区分大小写的排序)。
- asDate　返回一个日期(或 0)。
- asFloat　返回一个浮点值(或 0)。
- asInt　返回一个整数(或 0)。

20.2.2　Ext.data.DataReader

Ext.data.DataReader 是另一个用来提供更多代码一致性的抽象基类。DataReader 产生的 JSON 对象高度可定制(使用 Ext.data.Record)，而且性能极佳，即使有大量数据也是如此。Ext 库提供了几种数据缓存类(本章稍后讨论)，它们使用格式化的 JSON 对象填充 Ext 窗口部件库中的几乎所有 UI 组件。

有 3 个内置子类：Ext.data.XmlReader、Ext.data.JsonReader 和 Ext.data.ArrayReader。每个子类的构造函数都接受如下两个实参：

- meta　该子类特有的信息。
- recordType　发送给 Ext.data.Record.create 方法的自定义 Ext.data.Record 定义。

每个子类还有一个 readRecords 方法，它只接受一个实参，也就是数据。这个方法将数据反序列化并为每行数据创建一个 Record 实例。返回值是一个具有如下结构的对象：

- records　Record 实例的数组(使用配置选项 recordType 指定类型)。
- totalRecords　records 数组中的总行数或服务器上的总行数(用于数据分页)。
- success　该 Boolean 值指示表单提交是否成功(在第 33 章中将更深入地讨论这一点)。

对于这 3 个内置的 DataReader 子类，本章中的示例中使用了类似结构的数据(参见图 20-2)。注意，给定选中的 DataReader，只需要修改少量的 Ext 代码即可。

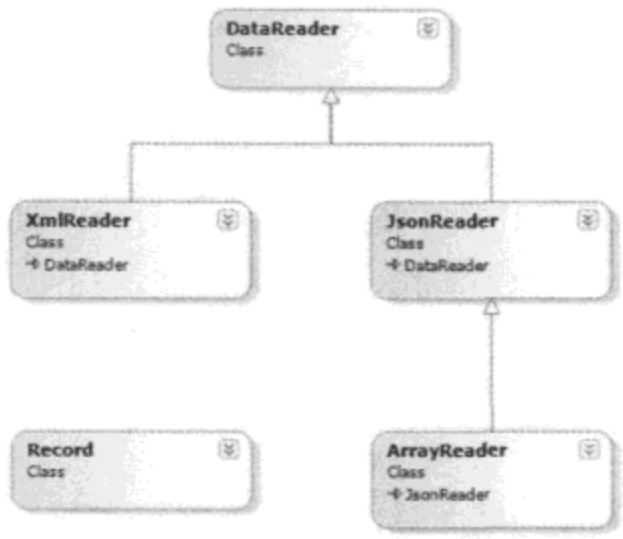


图　20-2

1. Ext.data.XmlReader

Ext.data.XmlReader 用于读取 XML 文档。此处的代码示例使用下面的 XML 文档：

```
<?xml version="1.0" encoding="UTF-8"?>
<table>
    <count>4</count>
    <person>
        <firstName>Bob</firstName>
        <lastName>Smith</lastName>
        <age>16</age>
        <userId>bsmith</userId>
    </person>
    <person>
        <firstName>Gary</firstName>
        <lastName>Johnson</lastName>
        <age>22</age>
        <userId>gjohnson</userId>
    </person>
    <person>
        <firstName>Susan</firstName>
        <lastName>Brown</lastName>
        <age>32</age>
        <userId>sbrown</userId>
    </person>
    <person>
        <firstName>Leslie</firstName>
        <lastName>Harris</lastName>
        <age>18</age>
        <userId>lharris</userId>
    </person>
</table>
```

meta 配置对象具有下面 5 个属性(均为可选属性)：

- **totalRecords**　一个指向总行数值的 Ext.DomQuery 路径表达式。在执行分页时，这个值非常有用。

- record　一个指向循环元素(表示数据行)的 Ext.DomQuery 路径表达式。
- id　一个指向每一行的唯一标识符字段的 Ext.DomQuery 路径表达式(相对于 record 表达式)。
- fields　发送给 Ext.data.Record.create 方法的 Record 定义(借助这个属性，可以将 DataReader 构造函数的 recordType 实参排除)。
- success　一个指向表示表单提交成功(更详细的相关内容请参见第 33 章)的值的 Ext.DomQuery 路径表达式。

```
var reader = new Ext.data.XmlReader({
    totalRecords: "count",
    record: "person",
    id: "userId",
    fields: [
        {name: "firstName"},
        {name: "lastName"},
        {name: "age", type: "int"}
    ]
});
```

2. Ext.data.JsonReader

Ext.data.JsonReader 用于读取 JSON 对象。此处的代码示例使用下面的 JSON 对象：

```
var data = {
    count: 4,
    person: [
        {firstName: "Bob", lastName: "Smith", age: 16, userId: "bsmith"},
        {firstName: "Gary", lastName: "Johnson", age: 22, userId: "gjohnson"},
        {firstName: "Susan", lastName: "Brown", age: 32, userId: "sbrown"},
        {firstName: "Leslie", lastName: "Harris", age: 18, userId: "lharris"}
    ]
};
```

meta 配置对象具有下面 5 个属性(均为可选属性)：

- totalProperty　一个指向总行数值的 JavaScript 表达式。在执行分页时，这个值非常有用。
- root　一个指向包含所有数据行的数组的 JavaScript 表达式。
- id　一个指向每一行的唯一标识符字段的 JavaScript 表达式(相对于 root 表达式)。
- fields　发送给 Ext.data.Record.create 方法的 Record 定义(借助这个属性，可以将 DataReader 构造函数的 recordType 实参排除)。
- successProperty　一个指向表示表单提交成功(更详细的相关内容请参见第 33 章)的值的 JavaScript 表达式。

```
var reader = new Ext.data.JsonReader({
    totalProperty: "count",
    root: "person",
    id: "userId",
    fields: [
        {name: "firstName"},
```

```
        {name: "lastName"},
        {name: "age", type: "int"}
    ]
});
```

JsonReader 还提供了 onMetaChange 方法(在内部执行)。当已经从 DataProxy 那里接收到数据时，数据对象可能包括一个 metaData 属性，该属性可以为当前 JsonReader 重新定义底层的 Record 结构以及所有的配置选项。当接收到 metaData 属性时，就会执行 onMetaChange 方法。下面就是这样一个对象(从 DataProxy 中返回)的示例：

```
{
    metaData: {
        totalProperty: "length", // changes the totalProperty value
        root: "person",
        id: "userId",
        fields: [ // redefines the Record structure
            // reverses the mapping of the name fields
            {name: "firstName", mapping: "lastName"},
            {name: "lastName", mapping: "firstName"},
            {name: "age", type: "int"}
        ]
    }
}
```

如果运用得当，那么这就是一项非常强大的功能。例如，如果用户将网格中的某一列隐藏起来，那么服务器就可以不发送这些值，从而节省了宝贵的带宽。而随着 Record 结构的重新定义，浏览器不用为不再存在的值浪费处理能力。

3. Ext.data.ArrayReader

Ext.data.ArrayReader 是一个特殊化的 JsonReader，用来读取复杂的 JavaScript 数组。下面的代码示例给出一个 JavaScript 数组：

```
var data = [
    ["Bob", "Smith", 16, "bsmith"],
    ["Gary", "Johnson", 22, "gjohnson"],
    ["Susan", "Brown", 32, "sbrown"],
    ["Leslie", "Harris", 18, "lharris"]
];
```

meta 配置对象有一个属性(可选)：id，这是数据行的唯一标识符的序号。

```
var reader = new Ext.data.ArrayReader({id: 3}, [
    {name: "firstName", mapping: 0},
    {name: "lastName", mapping: 1},
    {name: "age", mapping: 2, type: "int"}
]);
```

20.3　本地存储数据

在此之前，本章已经讲解了检索数据的各种方法，并且介绍了如何重新建模和反序列化数据。现在，需要将数据存储到本地，这样就可以用在网格、树、表单等对象中。Ext.data.Store 类使用了本章目前讲解的所有类和技术(参见图 20-3)。

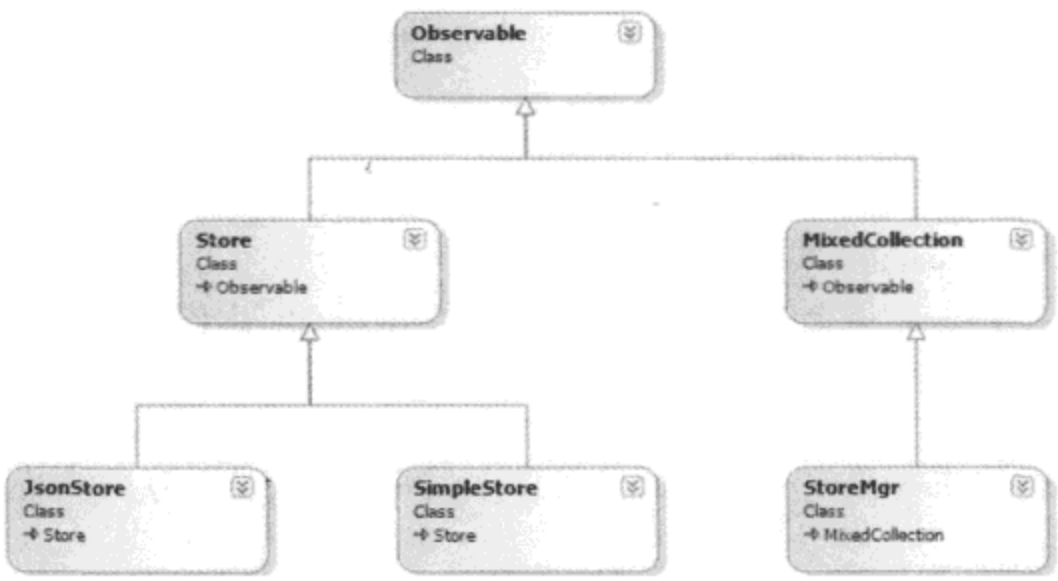


图　20-3

20.3.1　Ext.data.Store

Store 类为几乎所有数据访问提供了一站式解决方案。它不知道底层的结构(Ext.data.Record)、最初的外观(Ext.data.DataReader)，也不知道它来自何处(Ext.data.DataProxy)。但是，通过在内部使用这些类，Store 类能够以极其一致的方式来处理任意数据。

1. Ext.data.JsonStore

Ext.data.JsonStore 类派生自 Ext.data.Store，它只是一个带有内置 Ext.data.JsonReader 的辅助类。为了正确地构造 JsonReader，JsonStore 构造函数接受 Ext.data.Store 的所有选项，此外还有一个 fields 选项(将传给 JsonReader 构造函数)。

2. Ext.data.SimpleStore

Ext.data.SimpleStore 类派生自 Ext.data.Store，它只是一个带有内置 Ext.data.ArrayReader 的辅助类。为了正确地构造 ArrayReader，SimpleStore 构造函数接受 Ext.data.Store 的所有选项，此外还有 fields 选项和 id 选项(将传给 ArrayReader 构造函数)。

下面介绍 Store 类(及其子类)的全面细节。

3. Ext.data.Store 的选项

Store 构造函数具有如下配置选项：

- storeId　Ext.StoreMgr 使用的唯一值(缓存数据的集合，更详细的相关内容请参见 20.3.3 节)。

- url 用来构造 Ext.data.HttpProxy 的内部实例的 URL(如果同时提供配置选项 proxy，就忽略该选项)。
- autoLoad 一个 Boolean 值(用来确定在构造后是否应该立即执行 Store 的 load 方法)或一个传给 Store 的 load 方法的对象(在构造后立即执行)。如果同时提供了配置选项 data，就忽略该值(默认为 false)。
- proxy 用于数据访问的 Ext.data.DataProxy 实例。
- data 构造后立即加载的内联数据。Ext.data.DataProxy 实例完全没有必要，这是因为数据已经就位。
- reader 用于反序列化数据的 Ext.data.DataReader 实例。
- baseParams 一个对象，其中包含将要发送给 DataProxy 的 load 方法的值(将与 DataProxy 的 load 方法的 params 实参合并)。
- sortInfo 用来确定数据的默认排序的选项(示例：{field: "fieldname", direction: "DESC"})。如果 remoteSort 设为 true，那么这个信息将发送给 DataProxy。
- remoteSort 一个用来确定是否要由 DataProxy 处理排序的 Boolean 值。如果 remoteSort 为 true，就会把 sort 值和 dir 值同时发送给 DataProxy 的 load 方法(将与 load 方法的 params 实参合并)。
- pruneModifiedRecords Store 类能够跟踪对底层 Record 执行的修改。如果该值设为 true，就会尽可能快地将修改过的 Record 销毁(通过删除原始 Record 或重新加载 DataProxy)。

4. Ext.data.Store 的事件

Store 派生自 Ext.util.Observable，因此也具有几个事件：

- datachanged(store) 当底层 Record 数据因为重新加载、过滤或排序而发生变化时引发该事件。
- metachange (store, metadata) 当 Store 的 DataReader 执行它的 onMetaChange 方法时引发该事件。
- add(store, records) 当向 Store.records 中添加 Record 时引发该事件。records 是一个 Ext.data.Record 数组。
- remove(store, record, index) 当从 Store 中移除 Record 时引发该事件。
- update(store, record, operation) 当 Record 的值更新时引发该事件。operation 实参可以是下列值之一："edit"(Ext.data.Record.EDIT)、"reject"(Ext.data.Record.REJECT)或"commit"(Ext.data.Record.COMMIT)。
- clear(store) 当执行 Store 的 RemoveAll 方法时引发该事件。
- beforeload(store, options) 在执行 DataProxy 的 load 方法之前引发该事件。开发人员可以通过在事件监听程序中返回 false 来绕开加载过程。options 实参包含要发送给 Store 的 load 方法的 options 对象。
- load(store, records, options) 当成功执行 DataProxy 的 load 方法之后引发该事件。records 实参包含一个 Ext.data.Record 数组，该数组将添加到 Store 中。options 实参包含要发送给 Store 的 load 方法的 options 对象。
- loadexception 转发 DataProxy 的 loadexception 事件。

5. Ext.data.Store 的方法

Store 也具有一些有用的方法，下面就讨论它们。

手动添加和移除记录

- add(Records[])　添加到 Store 中的 Ext.data.Record 数组。引发 add 事件。
- addSorted(Record)　利用当前排序信息将 Ext.data.Record 实例添加到 Store 中。调用 insert 方法来把 Record 添加到适当位置。
- insert(index, Record[])　将 Record 数组插入到指定索引 index 处。引发 add 事件。
- remove(Record)　移除指定的 Record。引发 remove 事件。
- removeAll()　从 Store 中移除所有 Record。引发 clear 事件。

过滤和定位记录

- filter(field, value, anyMatch, caseSensitive)　根据指定 Field(field)并与 value(可以是一个字符串或 RegEx)比较来过滤 Record。anyMatch 是一个 Boolean 值，它用来确定在进行比较时是匹配 Field 值内部的任何地方，还是匹配该值的开头。caseSensitive 是一个 Boolean 值，它用来确定在比较时是否区分大小写。
- filterBy(func, scope)　Store 中的每个 Record 都会传给指定的函数(func)，该函数的作用域为可选参数 scope。对于在过滤器中包含的 Record，该函数必须返回 true。
- clearFilter(suppressEvent)　将所有过滤的数据恢复。suppressEvent 实参是一个 Boolean 值，用来确定是否引发 datachanged 事件。
- isFiltered(), Boolean　返回一个 Boolean 值，指示当前是否有过滤器在运行。
- find(field, value, startIndex, anyMatch, caseSensitive), Number　与 filter 类似，但返回第一个匹配的索引。
- findBy(func, scope, startIndex), Number　与 filterBy 类似，但返回第一个匹配的索引。
- query(field, value, anyMatch, caseSensitive), Record[]　与 filter 类似，但返回一个匹配 Record 的数组。
- queryBy(func, scope), Record[]　与 filterBy 类似，但返回一个匹配 Record 的数组。
- getAt(index), Record　返回指定索引 index 处的 Record。
- getById(id), Record　返回具有指定唯一标识符的 Record。
- getRange(startIndex, endIndex), Record[]　返回一个 Record 数组。
- indexOf(Record), Number　返回指定 Record 的索引。
- indexOfId(id), Number　返回具有指定唯一标识符的 Record 的索引。

排序记录

- getSortState(), Object　返回一个包含当前排序字段和排序方向(ASC 或 DESC)的对象。例如，{field: "name", dir: "ASC"}。
- setDefaultSort(field, direction)　为下一次加载过程设置排序信息。
- sort(field, direction)　排序数据。

加载记录

- load(options), Boolean　options 实参将传给 DataProxy 的 load 方法(本章前面详细讨论过该方法)。这些 options 选项将存储起来，以供 reload 方法在后面使用。引发 beforeload 事件，从而让开发人员有机会取消整个加载过程。如果加载过程被取消，那么这个方法返回 false。
- loadData(data, append)　这个方法使用 Store 的 Ext.data.DataReader 建模数据 data，然后填充 Store。Boolean 值 append 指示在填充前是否清空当前数据。引发 load 事件，如果执行附加操作，那么还会引发 datachanged 事件。
- reload(options)　这个方法与 load 方法的工作方式完全相同，但它重用了前一次调用 load 时的 options 选项。开发人员可以根据需要重写 options。

其他方法

- collect(dataIndex, allowNull, bypassFilter)　返回一个由 Record 指定列中的所有唯一值构成的数组。字符串 dataIndex 指定了待搜索的 Ext.data.Field 的名称。Boolean 值 allowNull 用来确定是否应该包括 null 值。Boolean 值 bypassFilter 用来确定是否应该包括过滤的 Record。
- each(func, scope)　Store 中的每个 Record 都会传给指定的函数(func)，该函数的作用域为可选参数 scope。如果从该函数返回 false，则会停止遍历。
- getCount(), Number　返回 Store 中 Record 的数量。
- getTotalCount(), Number　返回总计值(由 DataReader 返回)。
- sum(field, startIndex, endIndex), Number　返回指定索引范围内的所有 Record 的总和值(范围可选)。

20.3.2 Ext.data.Record(回顾)

当讲解数据建模和反序列化时，我们曾经讨论过 Ext.data.Record 类。然而，当与 Ext.data.Store 类结合使用时，Record 类就不只是一个简单的定义。

每一行由 Ext.data.DataReader 反序列化的数据都存储在自定义类型的 Record 实例中。Record 不仅存储数据，它还跟踪哪些值正在被修改。当与 Store 结合使用时，Record 类的方法可以引发 Store 中的事件，反过来也是如此。每个 Record 类都提供一组用来辅助改动管理的方法。

- get(name), Object　返回指定字段的值。
- isModified(name), Boolean　如果指定字段已经修改，就返回 true。
- getChanges(), Object　返回一个只由修改过的 Field 组成的散列对象。
- set(name, value)　将值应用于指定字段。如果 Record 并未处于“编辑”模式，那么这个方法会使用“编辑”操作引发 Store 的 update 事件。
- beginEdit()　将 Record 置于“编辑”模式。
- endEdit()　将 Record 移出“编辑”模式。如果任何 Field 值已经修改，就会立即使用“编辑”操作引发 Store 的 update 事件。
- cancelEdit()　将 Record 从“编辑”模式移出，并阻止 reject 方法恢复原始值。

- commit(silent)　将 Record 移出“编辑”模式，并接受所有修改的值。根据 Boolean 值 silent 来确定是否使用“提交”操作引发 Store 的 update 事件。
- reject(silent)　将 Record 移出“编辑”模式，并恢复原始值。根据 Boolean 值 silent 来确定是否使用“拒绝”操作引发 Store 的 update 事件。

Store 类还有一些辅助方法，用来一次性处理所有 Record：

- commitChanges()　在每个 Record 上执行 commit 方法。
- rejectChanges()　在每个 Record 上执行 reject 方法。
- getModifiedRecords(), Record[]　返回已修改的 Record 的数组。

如果使用这些便利的方法，那么确定哪些 Record(甚至哪些 Field)已经改变将是一件简单的事情。

```
var records = store.getModifiedRecords();
var line = [];
line[line.length] = "The following Records have been modified...";
for (var i = 0, rec; rec = records[i++];) {
    line[line.length] = "Record " + rec.id + ": ";
    var fields = rec.getChanges();
    for (var name in fields) {
        line[line.length] = "  Field: " + name + " Value: " + fields[name];
    }
}
alert(line.join("\n"));
```

20.3.3　Ext.StoreMgr

单例类 Ext.StoreMgr 应该用作缓存数据的储存库。如果设置了 Ext.data.Store 的配置选项 storeId，那么 Store 就会自动注册到 StoreMgr。一旦完成注册，就可以在任何时候使用 storeId 访问该 Store。

Ext.StoreMgr 类提供了 register 方法，它接受一个 Ext.data.Store 实例以及 unregister 方法(该方法接受 Ext.data.Store 实例或 storeId 作为参数)作为参数。还有一个 lookup 方法，该方法接受 storeId 作为参数，并返回 Ext.data.Store 实例。

这个简单的类以直观而有意义的方式简化了创建本地数据缓存的工作。

20.4　集成所有类

下面这个相当经典的示例演示如何将所有这些类集成起来。

```
var storeId = "adminUsers";
var store = new Ext.data.Store({
    storeId: storeId, // we'll cache this store instance for later
    baseParams: {
        userType: "admin" // we only want Admin users
    },
```

```
    sortInfo: {field: "lastName"}, // default sort by lastName
    // here's the DataProxy!
    proxy: new Ext.data.HttpProxy({url: "getUsers.html"}),
    // here's the DataReader!
    reader: new Ext.data.JsonReader({
        root: "person",
        id: "userId",
        // here's the Ext.data.Record structure!
        fields: [
            {name: "firstName", type: "string"},
            {name: "lastName", type: "string"},
            {name: "age", type: "int"}
        ]
    })
});
// let's use some events of the Store
store.on({
    load: {
        fn: function(store, records, options) {
            // when loading is done, get some Records
            var family = store.queryBy(function(record, id) {
                if (record.get("lastName") == "Jones") {
                    return true;
                }
                return false;
            });
            // then, filter the grid
            store.filter("firstName", "Bob");
            // and re-sort on age
            store.sort("age");
        }
    },
    loadexception: {
        fn: function(proxy, options, response, error) {
            alert("Something bad happened while loading");
        }
    }
});
// let's load this sucker up!
store.load();

// even if we delete the store variable, it's still available in the Ext.StoreMgr
//  this happens because we created a store with a storeId config option
delete store;
var store2 = Ext.StoreMgr.lookup(storeId);
```

20.5　本章小结

本章简要地讨论了几个以数据为中心的 Ext 类。我们讲解了诸如公共数据访问接口、数据反序列化这样的高级主题，并且讲解了自定义的数据建模和简单的数据缓存。

在下一章中，我们将了解这些类如何简化诸如呈现 Grid 和 Tree 这样的常见(而复杂)的任务。

第21章

DataView 和网格

本章讲解用于显示大量数据的 Ext JS 组件。DataView 和 GridPanel 均利用了前几章中介绍过的类。

这些类依靠第 20 章中讨论的数据类来收集数据并建模，而且它们延续了在整个 Ext JS 库中采用的对象与基于事件的架构。

本章内容简介：

- Ext.DataView
- Ext.grid.GridPanel

21.1 Ext.DataView

第 18 章介绍过 Ext.XTemplate 类，它能够根据层次结构数据来呈现任意 HTML。第 19 章讲解了 Ext Component 系统，并揭示了它的灵活性和易用性。在第 20 章中，我们讨论了 Ext.data.Store 类的数据处理功能。在使用这 3 种技术时，Ext JS 库已经提供了比普通 JavaScript 库更多的功能。当所有这些基础就位之后，Ext JS 的增量改进方法所带来的好处就会开始显露出来。

Ext.DataView 类派生自 Ext.BoxComponent 类，利用 Ext.XTemplate 呈现从 Ext.data.Store 那里获取的数据。下面的代码用来呈现用户列表：

```
var theData, theStore, theTemplate, theView;

theData = [
    ["Bob", "Smith", 16, "bsmith"],
    ["Gary", "Johnson", 22, "gjohnson"],
    ["Susan", "Brown", 32, "sbrown"],
    ["Leslie", "Harris", 18, "lharris"]
];
theStore = new Ext.data.Store({
    proxy: new Ext.data.MemoryProxy(theData),
```

```
    reader: new Ext.data.ArrayReader(
        {id: 3},
        [
            {name: "id", mapping: 3, type: "string"},
            {name: "firstName", mapping: 0, type: "string"},
            {name: "lastName", mapping: 1, type: "string"},
            {name: "age", mapping: 2, type: "int"}
        ]
    ),
    sortInfo: {field: "lastName"}
});
theTemplate = new Ext.XTemplate(
    "<tpl for=\".\">",
        "<div id=\"{id}\">Name: {lastName}, {firstName}",
            // do not show age for anyone over 21
            "<tpl if=\"age &lt; 21\"> - Age: {age}</tpl>",
        "</div>",
    "</tpl>"
);
theView = new Ext.DataView({
    itemSelector: "div", // every DIV is a unique and selectable item
    singleSelect: true,
    store: theStore,
    tpl: theTemplate
});
theStore.load();
new Ext.Viewport({items: [theView]});
```

因为 Store 类派生自 Ext.util.Observable，所以 DataView 类能够监听事件并在必要的时候重绘自身。在前一章中曾经介绍过，DataView 类也支持列表项选择。在默认情况下，选中项被分配 CSS 类 x-view-selected，但这个 CSS 类被 Ext JS 设计人员有意忽略。将下面的代码段添加到该代码示例中：

```
<style type="text/css">
    .x-view-selected {
        color: blue;
    }
</style>
```

现在，我们可以单击某一项并看到它的外观变化。

DataView 非常简单，但是具有多种用来呈现任意数据并允许用户与这些数据进行交互所需的功能强大的方法。在第 19 章中曾经讨论过，Ext.BoxComponent 基类提供了一种状态持久化机制(使用配置选项 stateEvents)和自定义扩展性(使用配置选项 plugins)，还提供对底层 HTMLElement 的访问(利用 getEl 方法)。

除了从 BoxComponent 类继承的功能之外，我们还可以利用下面的配置选项来定制 DataView 的行为：

- **autoAbort**　一个 Boolean 值，指示是否终止所有待处理的请求(默认为 false)。

- deferEmptyText　一个 Boolean 值，指示是否将 emptyText 的显示延迟到 Store 的第一次加载操作(默认为 true)。
- emptyText　一个字符串值，如果 Store 对象为空，就会显示其中包含的文本(默认为"")。
- itemSelector　(必需)一个简单的 CSS 选择器，指示哪些 Template 节点表示选中项(默认为 undefined)。
- loadingText　一个字符串值，当 Store 正在加载时显示其中包含的文本(默认为 undefined)。
- multiSelect　一个 Boolean 值，指示是否允许多选(默认为 false)。
- overClass　一个 CSS 类值，当光标在每一项上方悬停时应用该类(默认为 undefined)。
- selectedClass　一个 CSS 类值，当选中每一项时应用该类(默认为 x-view-selected)。
- simpleSelect　一个 Boolean 值，指示是否要求用户在进行多选时按下 Shift 或 Ctrl 键(默认为 false)。
- singleSelect　一个 Boolean 值，指示是否限制用户一次只能选取一项(默认为 false)。如果同时将该选项和 multiSelect 选项设为 true，那么该选项将被忽略。
- store　(必需)用于绑定的 Ext.data.Store(默认为 undefined)。
- tpl　(必需)用于呈现的 Ext.XTemplate(默认为 undefined)。
- trackOver　一个 Boolean 值，指示是否为每一项引发 mouseenter 和 mouseleave 事件(默认为 false)。

注意：

配置选项 itemSelector 是必需的。Ext JS 设计人员总是希望最终开发人员使用 DataView 执行列表项操作以及终端用户交互。itemSelector 确定操作的具体项。如果不需要进行操作或交互，那么 Ext.XTemplate 类可能是更好的选择。

21.1.1　操作 DataView

一旦呈现出来，DataView 中的每个节点就都是一个简单的 HTMLElement。而且每个节点在呈现时均带有模板中定义的属性。在之前给出的示例模板中，每个节点均是一个 div 标记，而且每个 div 标记都有一个确定的 id 属性。在呈现 DataView 之后再操作它非常简单，只需要选择节点并按照要求修改这些节点。因为开发人员能够完全控制呈现的内容，所以能够使用最适合自己目标的选择方法，在第 18 章中已经讨论过这一点(Ext.DomQuery、Ext.DomHelper 和 Ext.Element)。

1. 查找 DataView 的节点

除了直接使用 Ext.DomQuery 之外，还有几种方法可以帮助开发人员查找 DataView 节点。这些方法均针对 DataView 进行了优化。

findItemFromChild

```
findItemFromChild( HTMLElement node ) : HTMLElement
```

对于复杂的呈现模板，这个方法用于定位包含指定 HTMLElement 的 DataView 节点。

getNode

```
getNode( HTMLElement/String/Number nodeInfo ) : HTMLElement
```

这个方法用来定位 DataView 节点(使用节点的 HTMLElement、它的 ID 或节点索引)。

getNodes

```
getNodes( [Number start], [Number end] ) : Array
```

这个方法根据指定的节点索引范围返回一个 DataView 节点数组。

indexOf

```
indexOf( HTMLElement/String/Number nodeInfo ) : Number
```

这个方法返回指定节点(使用节点的 HTMLElement、它的 ID 或节点索引)的索引。

2. 选择 DataView 的节点

DataView 类也提供了几个支持通过编程方式选择节点的方法。

clearSelections

```
clearSelections( [Boolean suppressEvent] ) : void
```

这个方法清除所有选中项(将 CSS 类 selectionClass 从所有节点中移除)。可选实参 suppressEvent 用来防止引发 selectionchange 事件。

deselect

```
deselect( HTMLElement/Number node ) : void
```

这个方法取消选择指定的节点(通过 HTMLElement 或节点索引)。引发 selectionchange 事件。

getSelectedIndexes

```
getSelectedIndexes() : Array
```

这个方法返回由所有选中节点的索引构成的数组。

getSelectedNodes

```
getSelectedNodes() : Array
```

这个方法返回由所有选中节点构成的数组。

getSelectedRecords

```
getSelectedRecords() : Array
```

这个方法返回一个由所有选中的 Ext.data.Record 构成的数组(由 DataView 的配置选项中的 Ext.data.Store 定义)。

getSelectionCount

```
getSelectionCount() : Number
```

这个方法返回选中节点的计数。

isSelected

```
isSelected( HTMLElement/Number node ) : Boolean
```

这个方法返回一个 Boolean 值，指示指定的节点(通过 HTMLElement 或节点索引)当前是否选中。

select

```
select( Array/HTMLElement/String/Number nodeInfo, [Boolean keepExisting],
    [Boolean suppressEvent] ) : void
```

这个方法选择一个节点(利用该节点的 HTMLElement 或节点索引)。可以有选择地选中一个节点数组。使用 keepExisting 实参防止当前选中的节点被取消选择。使用 suppressEvent 实参防止引发 selectionchange 事件。

selectRange

```
selectRange( Number start, Number end, [Boolean keepExisting] ) : void
```

这个方法选择位于节点索引范围从 start 到 end (包括 start 和 end)的所有节点。引发 selectionchange 事件。

3. 操作 DataView 的底层数据

当然，HTMLElement 操作只是实际数据绑定 Component 的一部分。当把一个 Ext.data.Store 实例附加到 DataView 时，该 DataView 就会监听这个 Store 实例的几个事件(分别是 beforeload、datachanged、add、remove、update 和 clear)。

这些事件处理程序确保 DataView 反映出所有的数据变化。但是，有时开发人员需要进行完全的控制。下面的方法对于访问底层的 Ext.data.Record 非常有用。

getRecord

```
getRecord( HTMLElement node ) : Record
```

这个方法返回指定 DataView 节点的 Ext.data.Record(使用该节点的 HTMLElement)。

getRecords

```
getRecords( Array nodes ) : Array
```

这个方法返回指定 DataView 节点数组对应的 Ext.data.Record 数组(利用每个节点的 HTMLElement)。

refresh

```
refresh() : void
```

这个方法重新加载所有 Ext.data.Record 并利用模板彻底重新呈现 DataView。在 DataView 的 store 上执行 reload 方法也会导致 DataView 刷新。

refreshNode

```
refreshNode( Number index ) : void
```

这个方法重新呈现单个节点(利用节点索引指定)。当修改单个 Record 的数据时，这个方法非常有用。例如：

```
// 'dv' is an instance of Ext.DataView
var index = dv.indexOf("someElementId");
var node = dv.getNode(index);
var rec = dv.getRecord(node);
rec.set("someField", "newValue");
dv.refreshNode(index);
```

但是，由于 DataView 监听 Ext.data.Store 的 update 事件并在修改节点的 Record 时自动重新呈现它们，因此通常没有必要使用该方法。

setStore

```
setStore( Store store ) : void
```

这个方法将 DataView 附加到一个新的 Ext.data.Store。自动调用 refresh 方法。

21.1.2 DataView 事件

除了操作 DataView 的节点之外，还有几个事件可供开发人员利用，让系统对列表项选择和典型的鼠标事件(click、mouseenter、mouseleave、selectionchange 等)进行响应。下面的示例在前一个示例的基础上添加了一个单击事件监听程序：

```
var theData, theStore, theTemplate, theView;
theData = [
    ["Bob", "Smith", 16, "bsmith"],
    ["Gary", "Johnson", 22, "gjohnson"],
    ["Susan", "Brown", 32, "sbrown"],
    ["Leslie", "Harris", 18, "lharris"]
];
theStore = new Ext.data.Store({
    proxy: new Ext.data.MemoryProxy(theData),
    reader: new Ext.data.ArrayReader(
        {id: 3},
        [
            {name: "id", mapping: 3, type: "string"},
            {name: "firstName", mapping: 0, type: "string"},
            {name: "lastName", mapping: 1, type: "string"},
```

```
            {name: "age", mapping: 2, type: "int"}
        ]
    ),
    sortInfo: {field: "lastName"}
});
theTemplate = new Ext.XTemplate(
    "<tpl for=\".\">",
        "<div id=\"{id}\">Name: {lastName}, {firstName}",
            // do not show age for anyone over 21
            "<tpl if=\"age &lt; 21\"> - Age: {age}</tpl>",
        "</div>",
    "</tpl>"
);
theView = new Ext.DataView({
    itemSelector: "div", // every DIV is a unique and selectable item
    singleSelect: true,
    store: theStore,
    tpl: theTemplate,
    listeners: {
        click: function(view, index, node, e) {
            alert(theView.getRecord(node).data.lastName);
        }
    }
});
theStore.load();
new Ext.Viewport({items: [theView]});
```

21.2　Ext.grid.GridPanel

虽然 DataView 为在任何布局中显示数据提供了极大的灵活性，但是 Ext.grid.GridPanel 类解决了显示表格数据这个常见的问题。GridPanel 派生自 Ext.Panel，但是不需要布局配置设置。相反，GridPanel 按照传统的网格样式显示数据(参见图 21-1)。

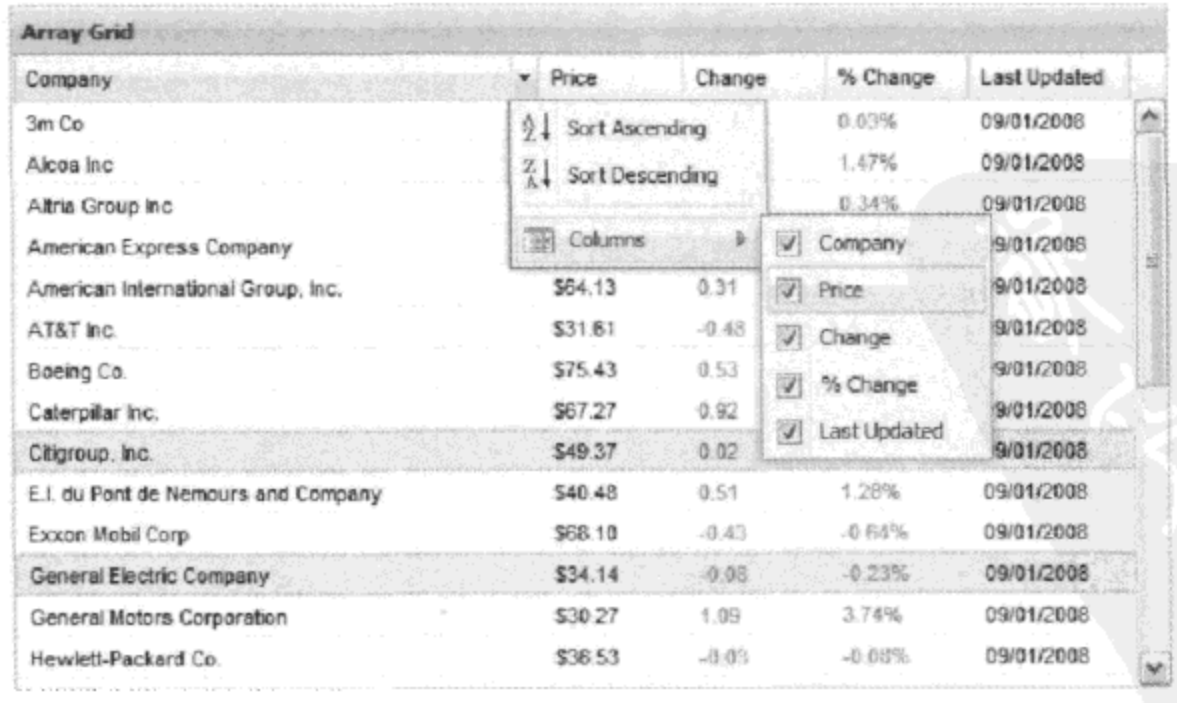


图　21-1

GridPanel 提供了许多配置选项。大部分配置选项都是自描述的标志，它们用来确定网格的不同行为(disableSelection、enableColumnHide、enableColumnResize 等)。但是，有几个配置选项需要额外的关注。很明显，store 选项非常重要，因为它指向 Ext.data.Store，并为网格提供数据(第 20 章讨论了 Store 类)。其他几个需要注意的配置选项有 colModel、selModel 和 view。

21.2.1 Ext.grid.ColumnModel

下面是图 21-1 中给出的网格的定义(出于简洁性考虑，这里已经将 store 对象忽略)：

```
var store = new Ext.data.Store(...);
// a function to help render the pctChange column
function pctChange(val, metadata, record, rowIndex, colIndex, store) {
    var color;
    if (val>= 0) {
        color = "green";
    } else {
        color = "red";
    }
    return "<span style=\"color:" + color + ";\">" + val + "%</span>";
}
// a function to help render the change column
function change(val, metadata, record, rowIndex, colIndex, store) {
    if(val>= 0){
        return "<span style=\"color:green;\">" + val + "</span>";
    } else {
        return "<span style=\"color:red;\">" + val + "</span>";
    }
    return val;
}
var grid = new Ext.grid.GridPanel({
    autoExpandColumn: "company",
    columns: [
        {
            header: "Company",
            width: 160,
            sortable: true,
            dataIndex: "company",
            id: "company"
        }, {
            header: "Price",
            width: 75,
            sortable: true,
            renderer: "usMoney",
            dataIndex: "price"
        }, {
            header: "Change",
            width: 75,
            sortable: true,
            renderer: change,
```

```
            dataIndex: "change"
        }, {
            header: "% Change",
            width: 75,
            sortable: true,
            renderer: pctChange,
            dataIndex: "pctChange"
        }, {
            header: "Last Updated",
            width: 85,
            sortable: true,
            renderer: Ext.util.Format.dateRenderer("m/d/Y"),
            dataIndex: "lastChange"
        }
    ],
    height: 350,
    store: store,
    stripeRows: true,
    title: "Array Grid",
    width: 600
});
new Ext.Viewport({items: [grid]});
```

配置选项 columns 包含列定义数组。在内部，该数组将传给 Ext.grid.ColumnModel 类的构造函数。ColumnModel 类告诉网格每一列的外观和行为。

列定义是一个支持如下选项的简单对象：

- align　一个用来设置列的 CSS 属性 text-align 的字符串(默认为 undefined)。
- css　一个用来为列中的所有表单元格(但不包括列标题单元格)设置 CSS 的字符串(默认为 undefined)。
- dataIndex　一个字符串，它指出该列要绑定的 Ext.data.Field。在默认情况下，该列将绑定到具有相同序号的 Field(例如，Column 1 绑定到 Field 1)。
- editor　在列中编辑值时用到的 Ext.form.Field(默认为 undefined)。我们将在第 22 章中讨论表单控件。
- header　一个字符串，其中包含的文本将在列标题单元格中显示(默认为 undefined)。
- hidden　一个 Boolean 值，指示在首次呈现时是否应该将该列隐藏起来(默认为 false)。
- hideable　一个 Boolean 值，指示该列是否能够隐藏(默认为 true)。
- id　一个字符串，在构造 CSS 类名时用到它，这个 CSS 类名将应用于该列中的每个单元格。最终的 CSS 类名看上去类似于如下：“x-grid3-td-{id}”。在默认情况下，将使用该列的序号(例如，“x-grid3-td-1”)。
- menuDisabled　一个 Boolean 值，指示是否应该禁用该列的菜单(默认为 false)。
- renderer　一个自定义函数，它用于生成该列每个单元格中的 HTML 标记。在默认情况下，将显示原始数据值。有关自定义呈现函数的完整讨论，请参见下面的小节。
- resizable　一个 Boolean 值，指示该列是否可调整大小(默认为 true)。
- sortable　一个 Boolean 值，指示该列是否可排序(默认为 GridPanel 的 defaultSortable 属性)。由 Ext.data.Store 确定是否使用本地排序或远程排序。

- tooltip 一个字符串，其中包含的文本将作为列标题的提示信息(默认为 undefined)。
- width 一个数字，给出首次呈现时该列的宽度(单位为像素，默认为 undefined)。

注意，网格列没有必需的属性。如果有一个带有 3 个已定义字段的 DataReader，而且只是希望依次显示这 3 个字段(不带列标题)，那么这个 ColumnModel 定义如下所示：

```
columns: [{}, {}, {}]
```

使用自定义函数呈现数据

Ext JS 让开发人员能够完全控制网格中的每个单元格的呈现，通过自定义 renderer 函数完成该操作。每个 renderer 函数都具有如下实参：

- value 原始数据值。
- metadata 一个包含两个属性的对象。
- css 一个字符串，它包含要添加到当前单元格的 td 标记中的 CSS 类名。
- attr 一个字符串，它包含要添加到当前单元格的 td 标记中的 HTML 属性标记(例如，'id="cell1" style="color: red;"')。
- record 包含当前行的所有 Ext.data.Field 的 Ext.data.Record 对象。
- rowIndex Ext.data.Record 在 Ext.data.Store 内的索引。
- colIndex 网格列的索引(而不是 Ext.data.Field 的索引)。
- store 用来绑定整个 GridPanel 的 Ext.data.Store 对象。

在上面的代码示例中，您可能已经注意到 Change 列中的值要么是红色，要么是绿色。查看该列的定义，可以看到它使用了自定义 renderer 函数。

下面是其代码：

```
function change(val, metadata, record, rowIndex, colIndex, store) {
    if(val>= 0){
        return "<span style=\"color:green;\">" + val + "</span>";
    } else {
        return "<span style=\"color:red;\">" + val + "</span>";
    }
    return val;
}
```

除了自定义呈现函数之外，还有几个内置呈现函数。在之前给出的代码示例中，Price 列使用名为 usMoney 的呈现函数。这些预先构建的呈现函数放在 Ext.util.Format 类中。当呈现网格数据并指定一个字符串值作为呈现器(例如 usMoney)时，网格就会利用该名称在 Format 类中查找。这意味着添加能够在整个应用程序中使用的自定义呈现器非常简单，如下所示：

```
Ext.util.Format.change = function(val) {
    if(val>= 0){
        return "<span style=\"color:green;\">" + val + "</span>";
    } else {
        return "<span style=\"color:red;\">" + val + "</span>";
    }
    return val;
};
```

现在，要使用新的自定义 renderer 函数，只需要在列定义中指定该函数的名称即可：

```
renderer: "change"
```

Ext.util.Form.dateRenderer 函数(在 Last Updated 列中出现过)是一个工厂函数，它使用指定的日期格式创建一个自定义的 renderer 函数。

利用这些简单的技术，创建真正自定义的、极具吸引力的网格将不再是一件困难的事情。

21.2.2　Ext.grid.AbstractSelectionModel

用户很可能需要在网格内部进行选择，GridPanel 的配置选项 SelectionModel 满足了这项需求。Ext JS 提供 3 种不同的预制选择模型(单元格、行和基于行的复选框)。每种选择模型均派生自简单的 Ext.grid.AbstractSelectionModel。除了从 Ext.util.Observable 类继承的成员之外，AbstractSelectionModel 类还提供了 3 个方法：isLocked、lock 和 unlock。

1. isLocked

```
isLocked() : Boolean
```

这个方法返回一个 Boolean 值，指示当前用户选择是否被锁定。

2. lock

```
lock() : void
```

该方法阻止用户再次进行任何选择。

3. unlock

```
unlock() : void
```

该方法允许用户再次进行选择。

21.2.3　Ext.grid.CellSelectionModel

CellSelectionModel 类可让用户选择单个单元格(典型的传统电子表格应用程序)。该类的功能完全符合开发人员的预期，而且提供了用于操作选择项的简单方法(clearSelections、getSelectedCell、hasSelection 和 select)。

21.2.4　Ext.grid.RowSelectionModel

RowSelectionModel 类可让用户选择 GridPanel 中的行。与 CellSelectionModel 类相似，该类的功能完全符合开发人员的预期，而且提供了几个用于操作选择项的相似方法(clearSelections、deselectRow、each、getCount、getSelections、selectRows 等)。

21.2.5　Ext.grid.CheckboxSelectionModel

CheckboxSelectionModel 类派生自 Ext.grid.RowSelectionModel，并且工作方式也一样。但是，

CheckboxSelectionModel 还能够用作列定义，该列定义会导致生成一个复选框列(为用户提供类似的、有用的 UI 约定)：

```
// create the selection model
var sm = new Ext.grid.CheckboxSelectionModel();

var grid = new Ext.grid.GridPanel({
    selModel: sm, // apply the selection model
    columns: [
        sm, // add the selection model as a column definition
        {
            header: "Last Name",
            dataIndex: "lastName"
        }, {
            header: "First Name",
            dataIndex: "firstName"
        }, {
            header: "Age",
            dataIndex: "age"
        }
    ]
    ... more config settings
});
```

其结果如图 21-2 所示。

	Last Name	First Name	Age
☐	Brown	Susan	32
☑	Harris	Leslie	18
☐	Johnson	Gary	6
☐	Smith	Bob	16

图 21-2

21.2.6 Ext.grid.GridView

ColumnModel 类处理单个单元格内容的呈现，而 Ext.grid.GridView 类处理的则是整个网格的呈现。表、表头、每一行以及每个单元格的外观均由 GridView 控制。GridView 派生自 Ext.util.Observable，并在呈现 GridPanel 的过程中引发多个事件(例如 refresh、rowremoved 等)。

内置 GridView 类在呈现网格方面做得非常好，其效果非常具有吸引力。但是，人们经常需要采用表的形式，以非传统的方式呈现数据。GridView 的 templates 属性包含了各种用来呈现 GridPanel 的模板(有关 Ext.XTemplate 类的讨论请参见第 18 章)。

21.2.7 Ext.grid.GroupingView

Ext.grid.GroupingView 类扩展默认的 GridView 类，增加了数据分组的层次，如图 21-3 所示。网格中的每个组均是可展开的，可用来为大量数据显示简单的摘要。

Company	Price	Change	Industry	Last Updated
Industry: Automotive (1 Item)				
General Motors Corporation	$30.27	$1.09	Automotive	04/03/2009
Industry: Computer (5 Items)				
Industry: Finance (3 Items)				
American Express Company	$52.55	$0.01	Finance	04/08/2009
Citigroup, Inc.	$49.37	$0.02	Finance	04/04/2009
JP Morgan & Chase & Co	$45.73	$0.07	Finance	04/02/2009
Industry: Food (2 Items)				
McDonald's Corporation	$36.76	$0.86	Food	04/02/2009
The Coca-Cola Company	$45.07	$0.26	Food	04/01/2009
Industry: Manufacturing (9 Items)				
3m Co	$71.72	$0.02	Manufacturing	04/02/2009
Alcoa Inc	$29.01	$0.42	Manufacturing	04/01/2009
Altria Group Inc	$83.81	$0.28	Manufacturing	04/03/2009
Boeing Co.	$75.43	$0.53	Manufacturing	04/08/2009
E.I. du Pont de Nemours and Company	$40.48	$0.51	Manufacturing	04/01/2009

图　21-3

21.2.8　其他的定制方法

ColumnModel 和 SelectionModel 架构已经为开发人员提供了相当大的控制权。但是，始终存在一些尚未实现的需求。幸运的是，我们还有一些定制 GridPanel 的技术。

修改 GridPanel 的最明显的技术就是继承它并添加自己的自定义功能。Ext.grid.EditorGridPanel 类就是如此，它只是添加了一个简单的编辑 API。Ext.grid.PropertyGrid 进一步进行扩展，为典型的属性网格(可编辑的由名/值对组成的网格)需求提供了一个解决方案。

还可以利用插件架构，我们曾经在第 19 章中提到过这种架构。任何派生自 Ext.Component 类的子类均具备插件架构。对于 GridPanel 来说，这是封装自定义的、可重用的逻辑的极佳方式。一旦构造好网格，就会执行所有插件的 init 方法并传入该网格的引用。现在每个插件都可以监听所有网格事件，而且基本上可以采用任何能够想到的方式来改变网格的行为。

Ext JS 库的优雅之处在于所有成员都有自己的合理定位。例如，EditorGridPanel 对输入控件知之甚少。在 Ext JS 中，有一个名称空间专门用于输入控件(我们将在第 22 章中讨论该名称空间)。因此，没有必要在名称空间 Ext.grid 中添加大量的逻辑(该名称空间在其他地方定义，而且可以重用)。对于网格标题行中的上下文菜单也同样如此：名称空间 Ext.menu 完全独立，并且可以重用。

扩展网格的各个部分是一件简单的事情：标识需求，并找到需要扩展的类。虽然这看似常识，但是扩展大多数 JavaScript 库都是困难的事情。而且对于大多数 JavaScript 窗口部件而言，如果将其用于最初预期的用途之外，它们就不再工作。

21.3　本章小结

本章简要浏览了 DataView 和 GridPanel 类，这两个类为呈现大量数据提供了解决方案。DataView 类让开发人员能够完全控制数据的呈现，而 GridPanel 类为开发人员提供了大量经常需要用到的工具。这两种解决方案均是可扩展的，即使在处理大量数据时也能够表现出极佳的性能。

PDG

第22章

表单控件、验证及其他功能

我们已经进入本书第III部分的冲刺阶段，并且将介绍 Ext JS 库中最有趣和最具视觉吸引力的部分。实际上，应用程序能否取得成功在很大程度上取决于细节方面。用户期待表单字段验证机制的立即反馈、自动完成文本框等。开发人员终于开始需要考虑周全的框架，而不是许多不成系统的函数库。本章将快速浏览整个表单窗口部件集合并了解它们如何验证，由于没有足够的篇幅来深入研究一些更有趣的部分，因此在本章结尾时我们只能给出部分示例。

本章内容简介：

- 表单控件
- 表单字段和表单验证

22.1 表单控件介绍

Ext JS 库旨在用作企业级应用程序的前端。因此，所有典型的原始控件均需要接受检查。表单控件是任何企业级应用程序的必需品。但是，大多数 JavaScript 库依靠基本的 HTML 控件，几乎忽略了这个问题的最关键部分。尽管一些 JavaScript 库提供了验证 API，但是大多数库仍然把正确的跨浏览器呈现的复杂任务交给开发人员完成。

对于试图解决一致性呈现问题的库来说，它们的解决方案通常就是内存和处理器密集型 API。开发人员认为这些 API 非常简单，但是当应用程序的设计变得越来越庞大、越来越复杂时，用户体验就会变得越来越差。Ext JS 的作者已经认真地考虑这项困难的任务并提供了相当令人满意的 API。

下面的样本代码给出了带有 3 个表单字段和一个 Save 按钮并具备验证功能的全功能 Ext.form.FormPanel 示例。虽然要到本章稍后部分才会深入了解其中的细节，但令人印象深刻的是，我们可以在不牺牲用户体验的前提下实现这些功能。实际上，这个样本只是一个简单的对象。在将该对象传给 Ext.form.FormPanel 构造函数并呈现之前，浏览器只需要存储该对象即可，它并没有引用 DOM 元素或其他大型对象，因此其内存消耗并不是问题。

```
{
    xtype: "form",
    defaults: {
        msgTarget: "side"
    },
    monitorValid: true,
    buttons: [
        {
            formBind: true,
            text: "Save",
            handler: function() {
                alert("Saved!");
            }
        }
    ],
    items:[
        {
            fieldLabel: "Checkbox",
            xtype: "checkbox"
        }, {
            fieldLabel: "TextField",
            xtype: "textfield",
            allowBlank: false
        }, {
            fieldLabel: "DateField",
            xtype: "datefield"
        }
    ]
}
```

下面快速浏览每种表单控件。

22.1.1　Ext.form.Label

通常，我们不会手动创建一个 Label 对象。但是，如果遇到这种情况，那么 Label 类只有少数几个配置选项需要设置(Ext.form.Label 派生自 Ext.BoxComponent，因此继承了该类的所有属性和功能)。

- **text**　待呈现的简单字符串。
- **forId**　用于 htmlFor 属性的 HTML 元素的 ID 属性。
- **html**　待呈现的 HTML 代码片段(如果同时设置了配置选项 text，那么该选项将被忽略)。

```
var myLabel = new Ext.form.Label({text: "First Name", forId: "fname"});
```

22.1.2　Ext.form.Field

与 Label 类一样，我们通常也不会手动创建 Field 对象。所有其他表单控件均继承自 Field 类，正常情况下最好直接使用这些类。在默认情况下，Field 类会呈现为普通的 HTML 文本框。但是，Field 类提供了 autoCreate 配置选项(一种 DomHelper 元素规范，参见第 18 章)，如果在某些特殊场

合下希望利用该类，但又需要完全控制 HTML，就可以使用该配置选项。Field 类处理获得焦点和丢失焦点事件，还提供了一个非常不错的验证 API。

下面的示例是一个采用 JavaScript 最佳实践的完整 Ext JS 应用程序。这个示例虽然简单，但是它演示了真正轻量级的 Ext JS。

```
// We create a single object to contain our entire application.
//  (No need to clutter the global namespace.)
TheApp = (function() {
    // Private variables can go here and be used throughout our application.
    //  These variables are global to our application, but hidden from the outside.
    var myLabelText = "Field Label";
    return {
        // The init method initiates our application.
        init: function() {
            // Remember, always set this value.
            Ext.BLANK_IMAGE_URL = "images/default/s.gif";

            var theForm = new Ext.form.FormPanel({
                renderTo: Ext.getBody(),
                // By using xtypes, we avoid creating heavy objects.
                //  The objects are created when they are actually needed.
                xtype: "form",
                buttons: [
                    // Our FormPanel has one button.
                    {
                        text: "Save",
                        handler: function() {
                            alert("Saved!");
                        }
                    }
                ],
                items:[
                    // Our FormPanel has only one element - a Field.
                    {
                        // No need to create a Label manually, the Field
                        //  class does it for us.
                        fieldLabel: myLabelText, // our private variable!
                        xtype: "field"
                    }
                ]
            });
        }
    };
})();
// The onReady method will fire our app's init method when the HTML document is
//  ready.
Ext.onReady(TheApp.init, TheApp);
```

表单字段和标签如图 22-1 所示。

图 22-1

上面的示例只是把 Field 显示出来，它并没有展示 Field 的任何功能。要想了解 Field 类的验证 API 的实际运行情况，还需要编写一些代码。但是，既然 Ext JS 已经提供了如此丰富的库，因此我们并不需要“重新发明轮子”。

22.1.3 Ext.form.TextField

TextField 类派生自 Field 类，并添加了一些真正有用的功能，例如自动增长以匹配用户输入，规定最小长度，实施最大长度，以及使用正则表达式限制只显示有效字符。

接下来，在前面的示例中添加一个 TextField(此处移除了注释)：

```
TheApp = (function() {
    return {
        init: function() {
            Ext.BLANK_IMAGE_URL = "images/default/s.gif";
            var theForm = new Ext.form.FormPanel({
                renderTo: Ext.getBody(),
                xtype: "form",
                buttons: [
                    {
                        text: "Save",
                        handler: function() {
                            alert("Saved!");
                        }
                    }
                ],
                items:[
                    {
                        fieldLabel: "Field Label",
                        xtype: "field"
                    }, {
                        // Here's our TextField!
                        fieldLabel: "Text Field",
                        xtype: "textfield",
                        // This TextField cannot be blank.
                        allowBlank: false
                    }
                ]
            });
        }
    };
})();
Ext.onReady(TheApp.init, TheApp);
```

图 22-2 给出了文本框的显示结果。

图　22-2

在执行这个示例时，我们会看到两个文本框。当通过 Tab 键在文本框之间循环遍历时(让它们为空)，第二个文本框(TextField)就会变成红色并带有下划线。在该文本框中输入一个值，按下 Tab 键，这个 TextField 现在变为有效。

22.1.4　Ext.form.FormPanel 和 Ext.form.BasicForm

在进一步深入了解各种 Field 类之前，首先查看 FormPanel 类。FormPanel 类派生自 Ext.Panel 类，并使用 Ext.layout.FormLayout 呈现它的组件(有关容器及其布局的讨论，请参见第 19 章)。除了绘制漂亮的图片之外，FormPanel 还增加了一个 Ext.Button 集合(参见前面的示例)和一些基本的表单验证监听机制，并且封装了 Ext.form.BasicForm 类。

BasicForm 类(派生自 Ext.util.Observable)包含加载和提交典型表单数据所需的所有机制。简而言之，FormPanel 在内部创建一个 BasicForm 对象并将所有 Field 对象添加到该 BasicForm 对象中。当 BasicForm 提交数据时，它将 Field 对象序列化并使用 AJAX 请求向服务器提交(如果有必要的话，可以使用一个配置选项，按照旧式的 FORM 提交来提交数据)。为了简化操作，FormPanel 接受 BasicForm 的所有配置选项并在创建 BasicForm 对象时将它们传入。

AJAX 数据加载和数据提交任务均通过各种 Ext.form.Action 类完成。Ext.form.Action.Load 类和 Ext.form.Action.Submit 类可以直接开箱即用，这些类的行为与 HttpProxy 类(参见第 20 章)非常相似，但是增加了几个配置选项来实现更精细的控制。每种 Action 类都有一个 run 方法，由该方法执行 AJAX 请求。但这两个类均完全由 BasicForm 类封装，通常并不需要直接使用它们。

因此，即使在 FormPanel 的后台(BasicForm 及其 Action 类)有大量的工作要完成，我们仍然不需要编写太多代码。

现在返回到该示例。新的需求如下：

- 在每个无效 Field 的旁边显示一个带有错误消息的验证图标。
- 只要表单无效，就将 Save 按钮禁用。
- 通过 Save 按钮实际地提交数据。

```
TheApp = (function() {
    return {
        init: function() {
            Ext.BLANK_IMAGE_URL = "images/default/s.gif";
            // a utility class for displaying fancy tooltips
            Ext.QuickTips.init();
            var theForm = new Ext.form.FormPanel({
                renderTo: Ext.getBody(),
                // see Chapter 19 about the defaults option
                defaults: {
                    // Validation messages are now on the left.
                    msgTarget: "side"
                },
```

```
            // This makes the FormPanel monitor all Field's validity.
            monitorValid: true,
            // This tells the FormPanel where to submit to
            url: "http://ourserver/ourpage",
            buttons: [
                {
                    // This makes this button aware of the Form's
                    //  validity.
                    formBind: true,
                    text: "Save",
                    handler: function() {
                        // Get the BasicForm and submit!
                        var formPanel = this.findParentByType("form");
                        if (formPanel) {
                            formPanel.getForm().submit();
                        }
                    }
                }
            ],
            items:[
                {
                    fieldLabel: "Field Label",
                    xtype: "field",
                    name: "Field1"
                }, {
                    fieldLabel: "Text Field",
                    xtype: "textfield",
                    name: "Field2",
                    allowBlank: false
                }
            ]
        });
    }
};
})();

Ext.onReady(TheApp.init, TheApp);
```

执行结果如图 22-3 所示。

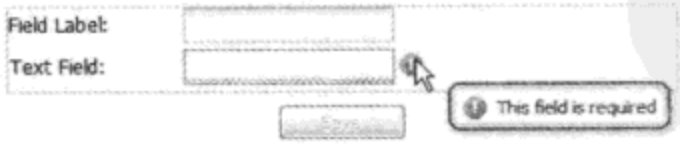


图　22-3

这样就完成了所有工作！只需要添加几个配置选项即可，剩余工作将交给 Ext JS 完成。需要注意的是，HTTP 要求每个 Field 均有一个 name 属性(但是，具体处理提交的服务器端代码并不在本书讨论范围之内)。

PDG

```
{xtype: "datefield"}
```

图　22-7

图　22-8

22.1.10　Ext.form.ComboBox

ComboBox 类也是一个特殊化的 TriggerField，它提供绑定到 Ext.data.Store 所需的全部功能(就像 Ext.grid.GridPanel 一样)。ComboBox 提供了可选的自动完成功能和可选的自定义文本输入功能，如图 22-9 所示。

```
new Ext.form.ComboBox();
```

或

```
{xtype: "combo"}
```

图　22-9

22.1.11　Ext.form.TimeField

TimeField 类是一个特殊化的 ComboBox，它不会绑定到 Ext.data.Store。所有有关时间格式化的方面均可以定制，如图 22-10 所示。

```
new Ext.form.TimeField();
```

或

```
{xtype: "timefield"}
```

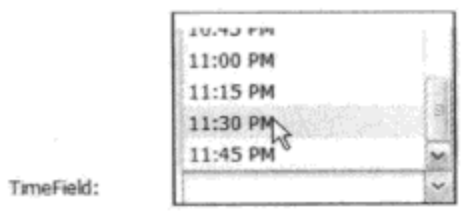


图　22-10

22.1.12　Ext.form.Checkbox

Checkbox 类的功能完全符合我们的预期(参见图 22-11)。

```
new Ext.form.Checkbox();
```

或

```
{xtype: "checkbox"}
```

图　22-11

22.1.13　Ext.form.Radio

Radio 类(派生自 Checkbox 类)表示一个单选按钮(参见图 22-12)。通过为多个单选按钮指定同一个 name 属性，它们就会变成互斥的按钮组。

```
new Ext.form.Radio();
```

或

```
{xtype: "radio"}
```

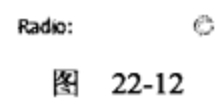


图　22-12

22.1.14　Ext.form.CheckboxGroup

CheckboxGroup 类是一个用来将多个 Checkbox 对象组合在一起的便利类。这些 Checkbox 对象将沿着水平或垂直方向呈现在一张表中。下面的示例创建了一个 3×3 大小的复选框表，如图 22-13 所示。

```
new Ext.form.CheckboxGroup({
    fieldLabel: "CheckboxGroup",
    columns: 3,
    defaults: {
        labelStyle: "{width: 5px}"
    },
    items: [
        {fieldLabel: "one"},
        {fieldLabel: "two"},
        {fieldLabel: "three"},
        {fieldLabel: "four"},
        {fieldLabel: "five"},
        {fieldLabel: "six"},
        {fieldLabel: "seven"},
        {fieldLabel: "eight"},
        {fieldLabel: "nine"}
    ]
});
```

或

```
{
    xtype: "checkboxgroup",
    fieldLabel: "CheckboxGroup",
    columns: 3,
    defaults: {
        labelStyle: "{width: 5px}"
    },
    items: [
        {fieldLabel: "one"},
        {fieldLabel: "two"},
        {fieldLabel: "three"},
        {fieldLabel: "four"},
        {fieldLabel: "five"},
        {fieldLabel: "six"},
        {fieldLabel: "seven"},
        {fieldLabel: "eight"},
        {fieldLabel: "nine"}
```

```
    ]
}
```

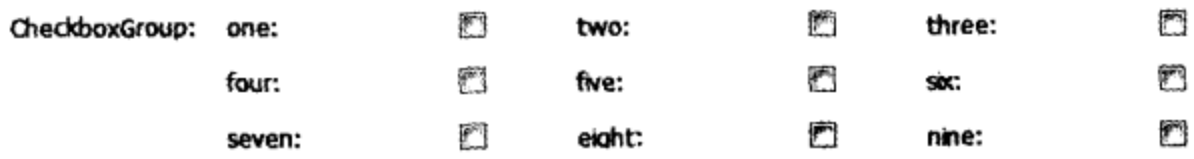


图 22-13

22.1.15 Ext.form.RadioGroup

RadioGroup类(派生自CheckboxGroup类)是一个用来将多个Radio对象组合在一起的便利类。这些Radio对象将沿着水平或垂直方向呈现在一张表中。下面的示例创建了一个3×3大小的单选按钮表，如图22-14所示(这里采用的是垂直布局)。

```
new Ext.form.RadioGroup({
    fieldLabel: "RadioGroup",
    columns: 3,
    vertical: true,
    defaults: {
        labelStyle: "{width: 5px}"
    },
    items: [
        {fieldLabel: "one"},
        {fieldLabel: "two"},
        {fieldLabel: "three"},
        {fieldLabel: "four"},
        {fieldLabel: "five"},
        {fieldLabel: "six"},
        {fieldLabel: "seven"},
        {fieldLabel: "eight"},
        {fieldLabel: "nine"}
    ]
});
```

或

```
{
    xtype: "radiogroup",
    fieldLabel: "RadioGroup",
    columns: 3,
    vertical: true,
    defaults: {
        labelStyle: "{width: 5px}"
    },
    items: [
        {fieldLabel: "one"},
        {fieldLabel: "two"},
        {fieldLabel: "three"},
        {fieldLabel: "four"},
```

```
            {fieldLabel: "five"},
            {fieldLabel: "six"},
            {fieldLabel: "seven"},
            {fieldLabel: "eight"},
            {fieldLabel: "nine"}
        ]
    }
```

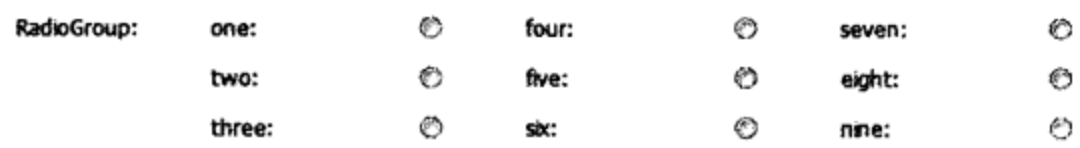


图　22-14

22.1.16　Ext.form.HtmlEditor

HtmlEditor 类是一个轻量级的、可定制的富文本编辑器。它的值将以格式化 HTML 的形式返回，如图 22-15 所示。

```
new Ext.form.HtmlEditor();
```

或

```
{xtype: "htmleditor"}
```

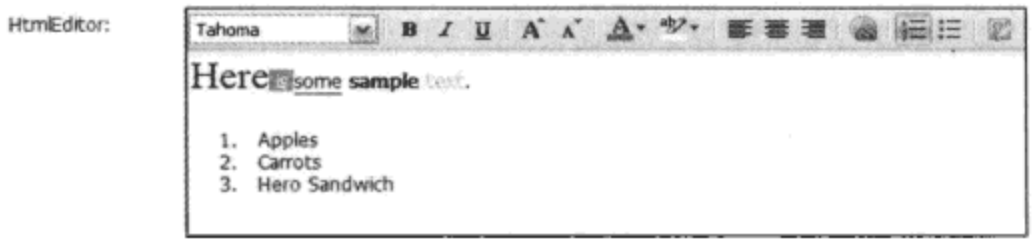


图　22-15

22.1.17　Ext.form.Hidden

Hidden 类是典型的隐藏表单字段。它不会呈现，但是它的值会与表单元素的其他部分一起提交(只要正确设置它的 name 属性)。

```
new Ext.form.Hidden();
```

或

```
{xtype: "hidden"}
```

22.2　表单字段和表单验证

大多数表单控件都内置有典型的验证技术。例如，NumberField 允许开发人员指定 minValue

和 maxValue，以及是否允许小数点和负号。DateField 允许对可选择的日期和工作日进行精细的控制。TextField 提供了 allowBlank、minLength 和 maxLength 选项。所有这些技术都非常简单，而且不言自明。

22.2.1　验证消息

通常，每种验证技术都会显示 invalidText 选项的文本(在默认情况下是“This value in this field is invalid.”)。但是，每种验证技术都还有一个选项可用来设置更具体的消息。例如，NumberField 的验证选项 minValue 有一个与之匹配的选项 minText。如果 minValue 验证失败，就会显示 minText。

所有这些消息都可定制，而且完全可选。此外，如何将这些消息报告给用户也是可定制的。Ext.ux.Field 类提供了 msgTarget 属性，这个属性用来确定 Field 的验证错误消息的显示位置。msgTarget 选项有 5 个内置的值，首先是 qtip 值(如图 22-16 所示)：

```
{xtype: "textfield", allowBlank: false, msgTarget:
"qtip"}
```

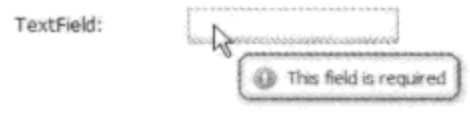


图　22-16

请注意，qtip 值要求调用 Ext.QuickTips.init 方法。

接下来是 title 值(如图 22-17 所示)：

```
{xtype: "textfield", allowBlank: false, msgTarget:
"title"}
```

图　22-17

然后是 under 值(如图 22-18 所示)：

```
{xtype: "textfield", allowBlank: false, msgTarget:
"under"}
```

图　22-18

请注意，under 值将会让错误消息直接在该表单字段下方呈现出来，并将其他内容下移。

第四个值是 side(如图 22-19 所示)：

```
{xtype: "textfield", allowBlank: false, msgTarget: "side"}
```

图　22-19

请注意，side 值要求调用 Ext.QuickTips.init 方法。

第 5 个选项是 HTML 元素的 ID。错误消息文本将简单地替换该元素的 innerHTML，这是一种完全的替换。换言之，如果希望使用一个元素包含所有表单字段的所有错误消息，那么不要使用这个方法。

当然，未支持的第 6 个选项是进行完全的控制。最简单的实现方式就是使用 title 值，然后监听 valid 和 invalid 事件：

```
{xtype: "textfield", allowBlank: false, msgTarget: "title",
    listeners: {
        invalid: function(field, msg) {
            field.el.dom.title = ""; // undo the built-in mechanism
            // this field is now invalid
            //  do whatever you want
        },
        valid: function(field) {
            // this field is now valid
            //  do whatever you want
        }
    }
}
```

22.2.2　高级验证技术

除了各种内置的验证方法之外，TextField(以及它的所有派生控件)还提供了 3 个可扩展选项：regex、validator 和 vtype。

首先，regex 选项是现代 TextField 控件的典型选项。当测试 TextField 的值的有效性时，将根据 regex 表达式测试该值。

```
{xtype: "textfield", regex: /^[0-9a-zA-Z]*$/ // alphanumerics ONLY
```

其次，validator 选项只是一个由开发人员提供的函数。该函数接受一个参数(TextField 的值)并返回一个指示该值是否有效的 Boolean 值。

```
// the value must be "red" or "green"
{xtype: "textfield", validator: function(value) {
    return (value == "red" || value == "green";
}}
```

最后，vtype 用作常见验证函数的快捷方式。一些基本的 vtype 函数已经内置，但我们可以轻易创建自己的函数。例如，下面的 TextField 必须是一个有效的 URL：

```
{xtype: "textfield", vtype: "url"}
```

这种方式非常简单，而且能够节省大量录入工作。

创建在整个应用程序中都可使用的新 vtype 函数也非常容易。例如，要创建一个简单的 vtype 函数来验证美国社会保障号码，可以使用如下的代码：

```
Ext.apply(Ext.form.VTypes, {
    ssn:  function(v) {
        return /^\d{3}\-\d{2}\-\d{4}$/.test(v);
    },
    ssnText: "Must be a valid SSN."
});
```

现在使用新的 vtype：

```
{xtype: "textfield", vtype: "ssn"}
```

这样就完成了所有工作！现在，我们有了一个自定义验证函数，它的自定义错误消息将在整个应用程序中都可用。

22.2.3 表单级验证

前面介绍的均是“表单字段级”验证(在单个字段上执行验证)。Ext JS 库并不提供任何开箱即用的“表单级”验证(在整个表单上执行验证)功能。然而，只需要少量工作就可以实现这种验证。

实现“表单级”验证的最简单方式就是重写 BasicForm 的 isValid 方法：

```
isValid: function() {
    var isValid = Ext.form.BasicForm.prototype.isValid.call(this);
    if (isValid) {
        // perform your custom "form-level" validation here
        // you'll also have to report any errors
    }
    return isValid;
}
```

当然，向用户报告表单级验证错误消息的工作必须由开发人员完成。我们要把该文本插入到表单的顶部吗？要显示一个模态对话框吗？最合适的做法取决于特定的需求。

22.3 其他功能

在 Ext JS 库中，我们可以找到非常多的小工具和便捷技术，即便是一本专门讲解 Ext JS 的书籍也很难将它们全部涵盖。在本章的最后一节中，我们将快速浏览一些极具吸引力的有用功能。

22.3.1 状态管理

一旦企业级应用程序达到特定的成熟度，用户就开始期望应用程序能够记住某些事情(网格排序顺序、可移动列的位置等)。但是，Ext JS 设计人员无法了解对于具体用户来说哪些是重要的方面，因此他们只向 GridPanel 中添加了状态管理功能。

但是，与该库的其他功能一样，我们也可以使用一种简单而健壮的技术来扩展基础库的功能。实际上，每个派生自 Ext.Component 类的窗口部件均具有如下的配置选项、方法和事件。

1. 配置选项

配置选项如下：

- stateEvents 一个由触发 saveState 方法的事件名称构成的数组。
- stateId 一个用于保存和恢复当前组件状态的唯一名称，如果已经为该组件指定一个唯一的 id，那么就没有必要设置这个配置选项。

- stateful　一个标志，用来确定当构造组件时是否应该恢复它的状态。构造组件过程是恢复状态的唯一时刻。

2. 方法

- applyState(state)　在恢复状态时，调用该方法以执行实际状态恢复。例如，当恢复网格中的隐藏列时，这个方法将检查状态对象并把适当的列隐藏起来。
- getState(), state　当保存状态时，调用该方法以收集状态信息(需要保存)。状态对象的结构由组件作者确定。这个结构可以是任何有效对象。

3. 事件

- beforestaterestore(component, state)　当恢复组件的状态之前引发该事件。返回 false 会阻止状态的恢复(从而不会调用 applyState 方法)。
- beforestatesave(component, state)　当保存组件的状态之前引发该事件。返回 false 会阻止状态的保存(从而不会调用 getState 方法)。
- staterestore(component, state)　在恢复组件的状态之后引发该事件。
- statesave(component, state)　在保存组件的状态之后引发该事件。

现在我们已经知道如何保存和恢复状态，但是将它保存到何处呢？Ext JS 只有一个内置在该库中的状态提供程序实现。Ext.state.CookieProvider 将状态保存到浏览器 cookie 中。CookieProvider 派生自 Ext.state.Provider(它提供编码和解码所需的所有机制)。很明显，CookieProvider 并不能适用于所有应用程序，但它演示了创建一个更为健壮的状态提供程序是非常简单的一件事情。

22.3.2　浏览器历史

除了 Component State Management 之外，Ext JS 还提供了一种用于处理浏览器状态管理(也就是浏览器的历史)的简单机制。Ext.History 类可用来执行让用户在浏览历史中前进或后退所需的所有任务。换言之，Ext JS 并不会完全记录应用程序的当前状态，而只是提供了一种记录简单记号(表示应用程序的当前状态)的机制。

1. 方法

- add(token, preventDuplicates)　这个记号是一个表示浏览器当前状态的字符串。该值可以是对应用程序有意义的任何内容。preventDuplicates 参数(默认为 true)的作用不言自明。
- back()　在浏览历史中后退一步。
- forward()　在浏览历史中前进一步。
- getToken(), token　返回代表浏览器当前状态的记号。

2. 事件

- change(token)　每当历史记号改变时都会引发该事件。有必要指出的是，每次记号发生变化时(即使是添加记号)都会引发 change 事件。

22.3.3 视觉效果

Ext.Fx 名称空间包含几种内置的视觉效果。与 Ext JS 库中的大部分名称空间一样，这个名称空间是可选的。如果不需要使用视觉效果，那么务必将 Ext.Fx 名称空间排除在应用程序之外。但是，这里给出的视觉效果执行起来非常简单，只有很低的性能开销。

当包含这个名称空间时，该名称空间中的方法将自动添加到 Ext.Element 类中。Ext.Fx 方法可以链式调用，从而为创建一些给人留下深刻印象的展示效果提供了非常简单的方式。

```
myElement.highlight(); // a simple highlight animation
```

请注意，Ext.Fx 方法可以执行链式调用，而 Ext.Element 方法也可以执行链式调用。但是，在将 Ext.Fx 方法和 Ext.Element 方法放在一起执行链式调用时必须小心。所有的 Ext.Fx 方法天生都是异步的，因此必须排队(这样它们就能够按照正确的顺序执行，而不会彼此重叠)。但 Ext.Element 方法是同步的，没有必要进行排队。

例如，Ext.Element 类中的 focus 方法将输入焦点放到指定元素中(假设为 TextField)。highlight 方法属于 Ext.Fx 类。

```
// I want to highlight the field and then set focus to it
myTextField.highlight("00FF00").focus(); // this won't work
myTextField.highlight("00FF00", { // this WILL work
    callback: function(el) {
        el.focus();
    }
});
```

第一个示例并不会工作，这是因为在高亮动画完成之前不会设置焦点。第二个示例使用了一个简单的回调函数，当动画播放完成时就会执行该回调函数。

22.3.4 拖放

虽然对于用户而言，拖放操作听起来非常简单(“我希望将这个组件抓起来并拖动到那里”)，但是对于开发人员来说，它却是一件比较棘手的事情。幸运的是，Ext JS 已经为我们提供了非常完善的基础来标识拖动区域和停放区域。既有用于完成简单的拖放操作的辅助类，也有用来呈现代理元素(该元素在拖动期间跟随光标)的辅助类。

还有几个类用来简化较为常见的必需功能：拖动到网格、表单、面板等或从其中拖出。

22.3.5 工具栏和菜单

工具栏已经融入到 Ext JS 库的每种 Panel 中(包括网格、选项卡、表单等)。但是，由于 Ext.Toolbar 类派生自 Ext.BoxComponent，这意味着实际上可以在任何地方呈现它。有几个 Toolbar 专用控件(特殊化的按钮和其他类似控件)，这使得一些常见任务变得容易。此外，还有一些特殊化的 Toolbar：

Ext.StatusBar 和 Ext.PagingToolbar。

当然，每种成熟的 JavaScript 窗口部件库都提供菜单控件。Ext.menu.Menu 类功能齐全，而且相当健壮，它提供图标、无限嵌套的子菜单、单选项、复选框等。它获得良好的支持，而且易于实现。当然，可以向工具栏中添加菜单，从而创建一个完备的菜单系统。

但是，Ext.menu.Menu 类并非派生自 Ext.Component 类。这其实是好事情，因为它避免了不必要的代码膨胀，并且不应该是问题。

22.3.6　主题

Ext JS 的整体外观是通过 CSS 和图像子图形完成的。所有的窗口部件类都允许开发人员针对几乎所有可以想到的情况指定 CSS 类名(selectedClass、invalidClass、hoverClass 等)。很明显，Ext JS 设计人员完成了大量的工作，以便利用 CSS 级联功能。然而，创建自己的主题并非易事。

通过一次只选择一个窗口部件作为目标(首先专注基础窗口部件)，我们能够完成这项工作。而且，Ext JS 社区已经为我们提供了大量预制的主题。

22.3.7　树

树是另一种经常用到的控件(导航树、数据树)。Ext JS 为树的展示提供了一种简单的、可扩展的机制，而且给出了几个示例来演示不同的树加载和呈现技术。很明显，Ext.tree.TreePanel 使用 Store 类来完成数据的收集和建模。此外，树的选择模型与网格的选择模型非常类似。

22.3.8　键盘导航

Ext.KeyMap 和 Ext.KeyNav 类为键盘导航问题提供了简单的解决方案。KeyNav 类是定向的解决方案，开发人员可以利用它来满足典型的键盘导航需求(左、右、上、下、向上翻页、向下翻页、回车、取消等)。而 KeyMap 类通过更通用的解决方案提供了一种简单的机制。

例如，KeyNav 类适合于处理组件(选项卡条、网格等)内的导航，而 KeyMap 类适合于更为通用的组件(菜单、编辑器等)。因为它如此通用，所以 Ext.Element 类中的一些辅助方法可以创建 KeyMap(addKeyListener 和 addKeyMap)。

22.3.9　其他更多技术

当然，还可以继续列出其他技术。有诸如 Ext.MessageBox 类和 Ext.Slider 控件这样的常见窗口部件，还有像名称空间 Ext.air(用于支持 Adobe AIR)和名称空间 Ext.sql(用于支持 SQLite)这样的先进窗口部件。还有大量的辅助类，它们有助于每天的开发工作：Ext.Layer、Ext.Shadow、Ext.Resizable、Ext.ColorPalette、Ext.util.ClickRepeater、Ext.util.DelayedTask、Ext.util.TaskRunner 等。

22.4　本章小结

本章讲解了完整的 Ext JS 表单窗口部件库。我们了解这些表单控件如何验证。当然，验证并不仅仅是一个简单的 Boolean 值，用户还需要知道哪个表单字段无效以及无效的原因。此外，我们还了解了向用户显示该信息的内置机制。

我们快速浏览了 Ext JS 库中一些知名度较低的类。对于像 Ext JS 这样具有如此丰富功能的库，它的规模却并不大，这实在是一件令人感到惊异的事情。但是，正确的面向对象的基本代码使得代码得以重用。通过利用 JavaScript 最佳实践和技术，整个库在所有浏览器上的性能都非常好。

毫无疑问，Ext JS 库在小型项目上表现非凡，甚至在最大型的项目中也具有极佳的扩展性。

第Ⅳ部分

Dojo

第 23 章：利用 Dojo 核心增强开发

第 24 章：操作 DOM

第 25 章：处理事件

第 26 章：编排动画

第 27 章：处理 AJAX 和动态数据

第 28 章：利用窗口部件构建用户界面

第 29 章：构建和部署 Dojo

第 30 章：扩展 Dojo

Dojo 是一个开源框架，其最初的目标是统一几个独立 DHTML 工具集和 JavaScript 实用工具库。就这一点而言，Dojo 工具集提供了现代 JavaScript 框架应该具备的所有便利工具。我们将看到针对基本语言的增强以及用于处理 DOM 操作和 AJAX 请求的常用工具。

但是，Dojo 的真正闪光之处在于它用于构建用户界面的便利工具，而且整个应用程序都使用声明式 HTML 标记进行构建。Dojo 并不是使用大段对象初始化代码拼凑应用程序并连接 DOM 元素，而是能够通过扫描页面自身来设置 Dojo 特有的属性和约定。尽管我们仍然能够选择不采用这种声明式方法，但它确实能够加快开发速度。

这个工具集的另一个主要支柱是它的封装系统，从而能够将丰富的库、组件、窗口部件和扩展有机组织起来。所有的模块都有清晰布局的命名空间，这一点与 Java 和 Python 中的标准库非常相似。

而且，提供一个清晰的结构确实是好事情，这是因为 Dojo 提供了大量的功能。但我们不必一次性使用所有这些功能：结合封装系统的有机组织，Dojo 提供了实用程序来实现动态加载模块以及它声明的所有依赖项(所有这些都可以在后面进行优化，通过定制内部版本将所需的代码打包成一个精简的 JS 包含文件)。此外，Dojo 的封装系统还可以用来组织和查找 CSS、图像以及其他与窗口部件和项目代码相关联的素材。

最后，在介绍 Dojo 时，有必要提及非盈利的 Dojo 基金会(Dojo Foundation)，建立这个基金会的目的是赞助该项目，同时让所有涵盖的贡献都能够具有清晰一致的授权。该项目的每个贡献者都要签署所谓的 Contributor License Agreement 协议。这份协议明确希望贡献方允许将自己的工作作为 Dojo 框架的一部分提供。

随后，Dojo 框架可以采用 BSD 授权或 Academic Free License，这两种授权(以及其他方面)允许人们在商业项目中安心地使用 Dojo，而不需要单独的授权或费用，从而有助于促进开发社区的无责任贡献。

第23章

利用 Dojo 核心增强开发

在本章中，我们将学习基本 JavaScript 语言增强的 Dojo 核心，它为该工具集提供的所有其他工具和系统建立了基础。但是，我们首先将学习如何获取 Dojo 副本(当前发行版本或最新的开发版本)。然后，我们将浏览基本的封装机制、标记声明以及面向对象编程功能。

本章内容简介：

- 声明、加载和提供依赖项
- 定义类和使用继承
- 在标记中声明对象

23.1 获取 Dojo

现在深入了解 Dojo，并开始亲自使用它。完成该操作有几种方式，每种方式都有各自的优点。

23.1.1 通过 AOL CDN 使用 Dojo

要开始使用 Dojo，最简单的方式就是通过 AOL 的内容分发网络(CDN)。

简而言之，AOL CDN 是一种广泛分布、高度可用的 Web 服务器网络，该网络用来托管静态素材(以及 Dojo 工具集的代码和附属素材)。这意味着我们可以使用类似下面的一条包含语句开始使用 Dojo：

```
<script type="text/javascript" src="http://o.aolcdn.com/dojo/1.1.1/dojo/dojo.xd.js">
</script>
```

这条 JavaScript 包含语句将获取 Dojo 核心的优化内部版本，该内部版本对于起步来说已经足够，这一点得益于 Dojo 封装系统和动态模块加载机制。

23.1.2 下载最新的 Dojo 发行版本

如果不喜欢这种依靠其他人的服务器来托管 JS 框架的形式(即使对方是像 AOL 这样的大公司)，那么可以下载完整的 Dojo 框架来自行托管。

可以如下网址中找到最新的 Dojo 发行版本：

```
http://dojotoolkit.org/downloads
```

在这个页面中，我们可以找到所有 Dojo 的分发包(以及几个其他支持工具，例如 Dojo ShrinkSafe，对于日后构建自己最终使用的 Dojo 压缩版本来说，该工具使用起来非常方便)。

一旦下载了分发压缩包，就需要将其解压缩到 Web 服务器中的某个便利的地方。注意该分发包的 URL，我们在本书这一部分的剩余内容中都将使用该 URL。

23.1.3 尝试尚处于开发阶段的 Dojo

如果您喜欢冒险，那么可以跳过发行版本，直接使用尚处于开发阶段的最新代码。为此，最简单的方式是签出一个内部测试版本(nightly build)，网址如下：

```
http://archive.dojotoolkit.org/nightly/
```

但是，如果希望更近距离地跟踪 Dojo 开发，那么可以直接从该项目的 Subversion 储存库签出代码。在写作本书期间，可以在下面的网址中找到有关该储存库的详细信息：

```
http://dojotoolkit.org/book/dojo-book-0-9/part-4-meta-dojo/get-code-subversion
```

如果有一个命令行 Subversion 客户端，那么(在写作本书期间)使用下面的命令就可以收集并签出所有子项目：

```
svn co http://svn.dojotoolkit.org/src/view/anon/all/trunk dojodev
```

在运行这条命令之后，应该会看到一个 dojodev/目录，其组织方式与分发包一样。本章后面的示例将引用 dojodev/目录(以本地方式使用 Dojo)，但是要记住：如此密切地使用最新的 Dojo 代码副本将不得不经常性地跟进开发进度，这可能并不是您想要的结果。

23.2 尝试使用 Dojo

如果获取了 Dojo 本地副本，无论是下载发行版本还是从开发储存库签出，那么应该注意，在包目录的根下面有几个目录，包括如下：

- dojo/ 这里存放的是 Dojo 的核心，包括基本的 JS 扩展、动态模块加载、DOM 和 AJAX 实用工具以及许多其他有用的工具。
- dijit/ 这个目录中存放的是 Dijit，即 Dojo 窗口部件系统。
- dojox/ 在这个路径下可以找到功能强大的可选 Dojo 扩展，它们提供了制图实用程序、加密例程、脱机应用程序支持等。

一个简单的 Dojo 示例

尝试使用 Dojo 的最佳方式可能就是立即浏览一个快速示例页面，加载工具集并尝试一些功能：

```
<html>
    <head>
        <title>Hello Dojo</title>

        <style type="text/css">
            @import "../dojodev/dojo/resources/dojo.css";
            @import "../dojodev/dijit/themes/tundra/tundra.css";
        </style>

        <style type="text/css">
            .accord { margin: 1em; height: 300px; }
        </style>

        <script type="text/javascript" src="../dojodev/dojo/dojo.js"
            djConfig="isDebug: true, parseOnLoad: true"></script>
```

注意，上面代码中的 CSS 和 JavaScript 路径假设这个页面位于本章专用的文件夹(ex-dojo-core/)中，该文件夹与 dojodev/文件夹在同一个目录中。

这里的代码目前非常简单，该页面打开时带有一个标题，并包括几个 Dojo 样式表。但值得注意的是，Dojo 确实包括一个基准“重置”CSS，这一点与 YUI 以及其他库提供的 CSS 类似，尽管 Dojo 的目标并没有那么远大。

第二条 CSS 导入语句更加值得注意：tundra.css 为 Dijit 窗口部件定义了完整的主题，可以将其替换成其他文件。例如，可以试着替换成 soria/soria.css。有关该功能的更多信息，请参见第 28 章。

最后的 CSS 内容只是对这个页面中出现的一个元素进行小幅度调整。

了解 CSS 之后，接下来查看该页面中的第一条 JavaScript 导入语句，这条语句加载 Dojo 核心。注意，在<script>标记中有一个自定义属性 djConfig(这是第一条提示，说明下面会有一些不同的内容出现)。djConfig 属性可以包含一些标志和设置，用来全局地配置 Dojo 工具集。在这里，isDebug 标志启用了调试日志记录和消息，而 parseOnLoad 标志告诉 Dojo，一旦该页面完成加载就扫描 DOM 以获得 Dojo 的设置提示。

下面是最后的<head>元素的标记：

```
        <script type="text/javascript">
            dojo.require("dojo.parser");
            dojo.require("dijit.form.TextBox");
            dojo.require("dijit.form.CheckBox");
            dojo.require("dijit.form.DateTextBox");
            dojo.require("dijit.form.NumberSpinner");
            dojo.require("dijit.form.Slider");
            dojo.require("dijit.layout.AccordionContainer");
        </script>
    </head>
```

这个<script>块中的 dojo.require()调用利用了 Dojo 模块系统。对于每次 dojo.require()调用，如果指定的模块尚未加载，那么 Dojo 核心会尝试找到并动态加载它，以满足所有的依赖项。如果这些模块声明了进一步的需求(此处的这些模块就是如此)，那么 Dojo 也会获取并加载它们。

实际上，如果使用 Firefox 作为自己的浏览器，而且安装了 Firebug 扩展，那么可以查看网络活动选项卡，监控指定的需求所生成的所有请求。图 23-1 给出了一个可能会看到的示例。这里有大量的请求，但要记住的是，我们后面会介绍一些用于优化这种情况的工具。

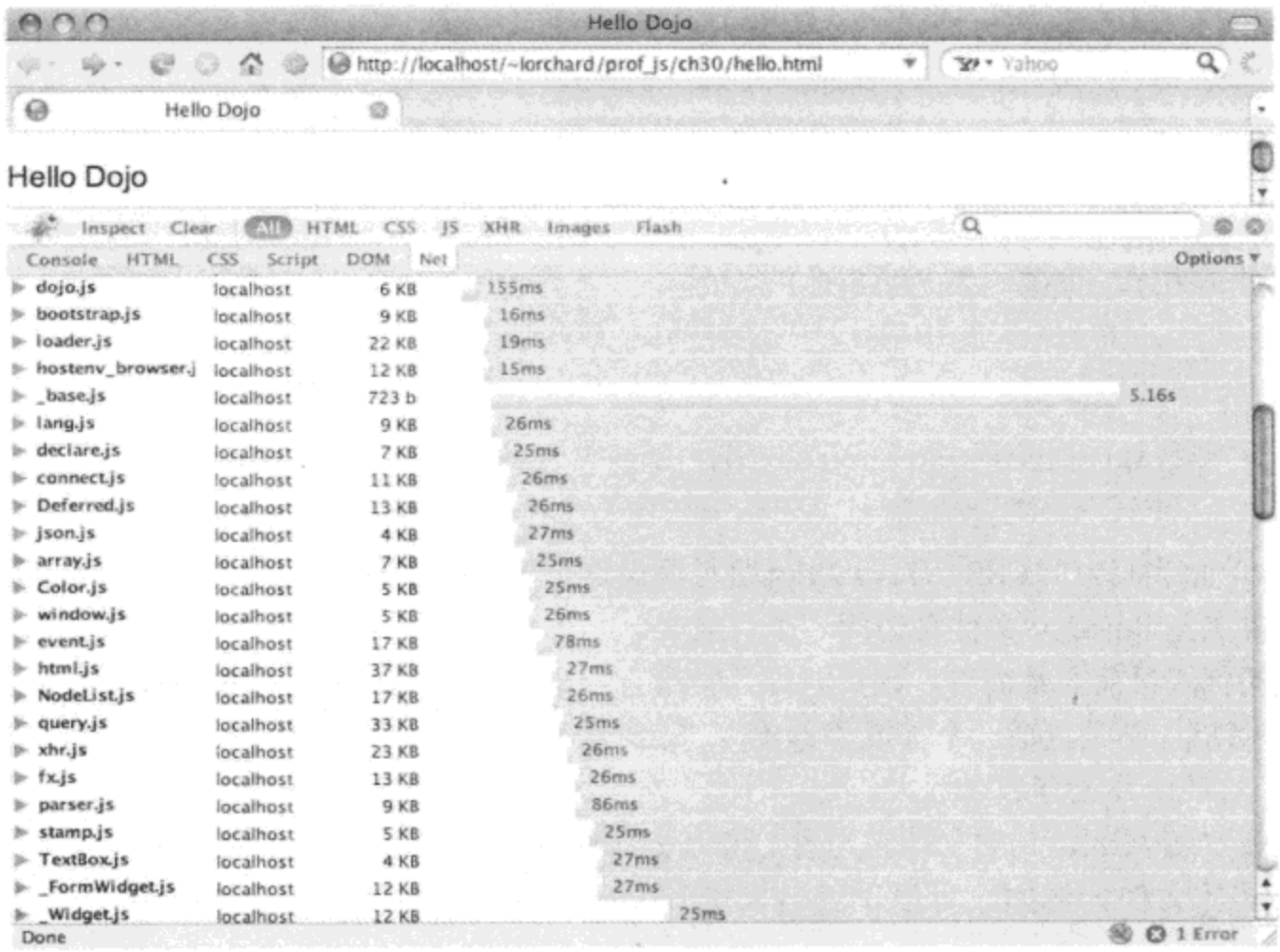


图　23-1

使用 Firebug

Firebug 是专门为 Web 开发人员设计的极为有用的 Firefox 扩展。

Firebug 对于开发工作的方方面面都非常有用，它可用来检查页面中的元素，查看并操作现场 CSS，访问 JavaScript 外壳以尝试运行小型代码片段，还可以查看代码中的调试消息日志。Firebug 的网站为 http://getfirebug.com。

还有一点值得注意的是，Dojo(至少在开发版本中)附带了 Firebug Lite 的非 Firefox 浏览器专用副本，可以通过 Dojo 配置设置 isDebug 来激活它。有关 Firebug Lite 的更多信息，请参见 http://getfirebug.com/lite.html。

接下来查看该页面的<body>标记：

```
<body class="tundra">

    <h1>Hello Dojo</h1>
```

注意，<body>标记有一个属性 class="tundra"，这对应于前面的 CSS 代码中加载的主题。现

在我们开始了解一些有趣的内容。

```
<div class="accord"
    dojoType="dijit.layout.AccordionContainer"
    duration="200">

   <div dojoType="dijit.layout.AccordionPane" selected="true"
      title="Intro" class="apane">
      <p>Hello world - welcome to Dojo!</p>
   </div>
```

这个页面中的 JavaScript 代码不算太多，但上面示例中的 HTML 标记将引出相当多的 JavaScript 代码。由于<head>标记中加载的需求模块(以及用来加载 Dojo 的<script>标记中出现的属性 djConfig="parseOnLoad: true")，当页面加载时，这里的标记会解释成用于对象初始化的声明式提示。

具体来说，这个标记声明要构造一个可折叠式布局窗口部件。尽管会产生一些警告，但可以将这个窗口部件设置成与页面上任何 HMTL 标记相似的样式，通过之前的 CSS 中出现的.accord 选择器(放在<head>元素中)可以看到这一点。

注意，每个 元素都带有一个自定义的 dojoType 属性，该属性定义了要实例化哪些 Dijit 类并包装这些元素。 父元素变成了 AccordionContainer，而第一个 子元素变成了 AccordionPane。此外，每个元素都以自定义属性的形式附带了窗口部件的配置信息。例如，duration="200"指定了可折叠面板开启和关闭时所播放的动画的速度，而 title="Intro"指定了面板的标题。

图 23-2 给出了前面的代码产生的结果示例。

图　23-2

为了进一步理解这些理念，接下来查看在标记中声明的其他几个窗口部件：

```
<div dojoType="dijit.layout.AccordionPane"
    title="Form #1" class="apane">
   <form>
```

```
<label for="fname">First Name</label><br />
<input name="fname" type="text" length="10"
   trim="true" propercase="true"
   dojoType="dijit.form.TextBox" /><br />

<label for="lname">Last Name</label><br />
<input name="lname" type="text" length="10"
   trim="true" propercase="true"
   dojoType="dijit.form.TextBox" /><br />
```

这也是一个 AccordionPane，这一次它包含一个<form>元素。这个表单开头有两个输入字段，它们也声明为 TextBox 窗口部件。这两个窗口部件都包括形如 trim="true"和 propercase="true"属性的参数(这两个属性分别指示修剪空白符以及将首字母大写)。

接下来尝试一个稍微更加有趣的窗口部件：

```
<label for="bday">Birthday</label><br />
<input name="bday" type="text" length="10"
   dojoType="dijit.form.DateTextBox" /><br />
```

尽管这个 DateTextBox 的标记要比前面两个窗口部件的标记简单，但是它完成的工作要多得多。图 23-3 给出了当用户单击这个新的输入字段时会发生什么情况。我们可以将一个功能齐全的日历日期选择窗口部件整合到简单的 HTML 表单中。

图 23-3

最后，查看完成设置页面外观工作的脚本：

```
<label for="fnum">Favorite Number</label><br />
<input name="fnum" id="fnum" type="text" length="15"
   dojoType="dijit.form.TextBox" /><br />

<div dojoType="dijit.form.HorizontalSlider"
   id="horiz1" name="horiz1" value="10"
```

```
                    maximum="100" minimum="0"
                    intermediateChanges="true" showButtons="false"
                    onChange="dojo.byId('fnum').value=arguments[0]"
                    style="width:10em; height: 20px;">
                </div>

            </form>

        </div>
    </div>
</body>
</html>
```

如果不能理解脚本处理，那么查找 HorizontalSlider 窗口部件的 onChange 属性。这段代码将该窗口部件的更新内容作为一个数字复制到它前面的 TextBox 中。除此之外，这里用到的两个窗口部件并不比我们在样本页面中看到的其他窗口部件复杂。图 23-4 给出了最终结果。

图　23-4

Dojo 主要通过 HTML 来引入 JavaScript 框架，虽然这看上去有点奇怪，但却是 Dojo 魔法的主要部分。在接下来的一章中，我们将看到 Dojo 的许多方面都可以通过声明方式或编程方式使用。我们可以完全严格地通过 JavaScript 来完成工作，但是可能最终会发现，充分利用该系统并构建自己的窗口部件和声明式驱动模块会非常有用。

23.3　研究 Dojo 核心

既然我们已经开始深入 Dojo，因此接下来更近距离地查看 Dojo 核心为 JavaScript 开发人员提供了哪些功能。在本书这一部分的剩余内容中，我们还会再次接触到最初的示例中演示的其他方面。

23.3.1　声明、加载和提供依赖项

定义模块、声明依赖项和动态加载代码是 Dojo 的关键方面，我们可以在自己的项目中利用这些便利功能。

1. 使用 dojo.require()声明和加载依赖项

在前面的示例所使用的 dojo.require()中，我们已经看到这个系统的部分运行情况，如下所示：

```
dojo.require("dojo.parser");
dojo.require("dijit.form.TextBox");
dojo.require("dijit.form.CheckBox");
dojo.require("dijit.form.DateTextBox");
dojo.require("dijit.form.NumberSpinner");
dojo.require("dijit.form.Slider");
dojo.require("dijit.layout.AccordionContainer");
```

这些语句的运行结果是动态构造<script>并插入到页面的<head>标记中。虽然还有其他的细节，但这是它的基本思想。为了推导要包含的 URL，Dojo 需要完成如下工作：将模块名中的句点替换成斜杠，添加.js 后缀，然后根据 dojo.js 所在位置构造基础的 URL 前缀。

这些模块大多数都进一步包括 dojo.require()语句，这又会导致更多的脚本包含。但是，最终这些 dojo.require()语句中有许多请求相同的一些模块。dojo.require()足够智能，它知道某些模块已经加载，从而不会尝试再次加载。

2. 使用 dojo.provide()提供依赖项

实际上，Dojo 的智能化并不仅仅体现在 dojo.require()中。相反，它们依赖于使用 dojo.provide()的约定。Dojo 中的每个模块均从调用 dojo.provide()开始，这会将该模块注册为已经加载。例如，dijit/form/TextBox.js 文件的开头是如下代码：

```
dojo.provide("dijit.form.TextBox");
```

但是，在自己的代码中使用 dojo.provide()还会带来另一个好处。回顾自己是否曾经看到或编写过类似下面的代码：

```
if (typeof window.decafbad == 'undefined')
    window.decafbad = {};
if (typeof decafbad.util == 'undefined')
    decafbad.util = {};
if (typeof decafbad.util.foo == 'undefined')
    decafbad.util.foo = {};

decafbad.util.foo.aMethod = function() {
    // method body
}
```

这是许多现代 JavaScript 应用程序建立名称空间时使用的约定，在尝试使用名称空间之前确保它的每一部分都存在。有时候只需要创建名称空间的最后一部分，这是因为所有的父名称空间都已经在先前加载的依赖项中创建完毕。但是，这只是一种通用形式的解决方案。

可以将这段代码替换成一种更为精确且更具描述性的 dojo.provide()调用，就像下面这样：

```
dojo.provide("decafbad.util.foo")

decafbad.util.foo.aMethod = function() {
    // method body
}
```

这个调用不仅会把该名称空间注册为已经加载(对应到 dojo.require())，而且确保完整的名称空间本身也会得以创建(如果必要的话)。这就为代码的实现和可读性方面带来了益处。

3. 将模块位置告诉 Dojo

关于模块依赖项还有一点需要了解：Dojo 如何找到我们编写的代码？在默认情况下，dojo.require()试图从 dojo.js 所在的父 URL 那里加载所有的模块。这意味着 Dojo 将根据安装 Dojo 及其子项目的位置自行确定相对路径(如果采用 AOL CDN 托管方式，那么该路径就可能是 AOL CDN)。

因此，dojo.require("decafbad.util.foo")可能解析成如下的网址：

```
http://o.aolcdn.com/dojo/1.1.1/decafbad/foo/bar.xd.js
```

由于将项目托管到 AOL CDN 的可能性并不大，因此我们希望 Dojo 在其他地方查找我们的代码。并且，即使正在使用本地下载的 Dojo 安装，将自己编写的代码放到 Dojo 安装目录之外的地方也会让人觉得更加清晰(假设计划在未来升级 Dojo)。因此，这就是 dojo.registerModulePath()发挥作用的地方：

```
dojo.registerModulePath('decafbad', '../../ex-dojo-core/decafbad');
```

如果正在使用 Dojo 的本地安装，那么这条语句将导致针对 decafbad 模块以及它的任何子模块的 dojo.require()调用从相对于 dojo 模块目录的给定路径中加载它们。例如，假设项目的目录结构如下：

- dojodev/
  - dojo/dojo.js
- ex-dojo-core/
  - hello.html
  - decafbad/foo/bar.js

既然 dojo.js 位于 dojodev/dojo/目录下，那么注册的相对模块路径../../ex-dojo-core/decafbad 将位于 Dojo 安装本身所在的目录之外。

如果正在使用 AOL CDN 托管版本 Dojo(XDomain 内部版本)，那么事情会变得稍微复杂一些。这意味着这个特殊的 Dojo 内部版本要从另一个不同的域中(也就是 o.aolcdn.com)加载它的资源，而不是从包含它的目录(也就是我们指定的目录)加载资源。由于跨域模块加载实现的特殊性，这里的情况与本地 Dojo 安装所允许的加载方式稍有不同。

我们基本上不需要太担心这一点带来的问题，但是在这里需要在 djConfig 中为自己的模块包括一个基础 URL(相对于当前页面而不是本地 Dojo 安装)：

```
<script type="text/javascript"
   src="http://o.aolcdn.com/dojo/1.1.1/dojo/dojo.xd.js"
```

```
    djConfig="isDebug: true, parseOnLoad: true, baseUrl: './'"></script>

<script type="text/javascript">
    dojo.registerModulePath('decafbad', './decafbad');
    dojo.require("dojo.parser");
    dojo.require("dijit.layout.AccordionContainer");
    dojo.require("decafbad.foo.bar");
</script>
```

这会为所有采用相对路径加载的模块设置基础 URL，从而将属于 Dojo XDomain 内部版本的模块排除在外(这是因为在构建这个版本的 Dojo 的过程中，XDomain 内部版本中的模块都经过处理以使用指向 AOL CDN 的绝对路径)。

此外，既然 djConfig 中现在已经有两项设置，那么值得注意的是，还可以在 JavaScript 代码块(而不是属性)中更加完整地定义 djConfig：

```
<script type="text/javascript">
    djConfig = {
        isDebug: true,
        parseOnLoad: true,
        baseUrl: './',
        modulePaths: {
            "decafbad": "./decafbad",
        },
    };
</script>
<script type="text/javascript"
    src="http://o.aolcdn.com/dojo/1.1.1/dojo/dojo.xd.js"></script>
```

注意，可以替换 dojo.registerModulePath()调用，转而在 djConfig 中定义 modulePaths。如果知道自己将要定义一系列路径或完成一些高级的 Dojo 配置，那么这种方式更加清晰和高效。

> 有关 Dojo XDomain 内部版本的更多内容
>
> 为了避免太多的离题内容，本章并没有深入研究 Dojo 生成系统。本书后面有一章内容专门讲解生成系统，但如果希望了解 XDomain 内部版本，那么可以到下面的网站中阅读更多内容：
>
> http://dojotoolkit.org/book/dojo-book-0-9/part-4-meta-dojo/package-system-and-custom-builds

23.3.2 定义类和使用继承关系

要理解 JavaScript 语言，重点是要知道虽然该语言确实能够处理对象，但它是一门基于原型的语言。它并不能像其他关注面向对象编程的语言那样原生地提供类和继承机制。在许多方面，基于原型的语言更具表达力，这是因为它们可以实施多种形式的代码重用(包括类和继承)，但问题是要自己实现这些功能。

1. 使用 dojo.declare()定义类

认识到这一点之后，接下来查看下面的代码，它按照通常的方式定义了一个 JavaScript 类：

```
decafbad.school.PersonClassic = function(name) {
    this.name = name;
};
decafbad.school.PersonClassic.prototype = {
    getName: function() {
        return this.name;
    }
};
```

如下功能类似的代码来自于 decafbad/school.js 中定义的一个模块：

```
dojo.provide("decafbad.school")

dojo.declare("decafbad.school.Person", null, {
    constructor: function(name) {
        this.name = name;
    },
    getName: function() {
        return this.name;
    }
});
```

dojo.declare()方法负责完成创建一个带有可继承构造函数和方法的 JavaScript 类所需的所有后台工作。这个方法的实参如下：

- 类名，带有完整的名称空间。
- 要继承的父类(如果有的话)。此处该实参为 null，说明这个类没有父类。
- 定义该类的对象字面值，包括它的所有数据成员和方法。

在 Dojo 类所定义的方法中，每当创建该类的新实例时都会调用 constructor 方法。这个方法的功能与原生 JS 基于原型的类中的初始化函数定义相同。

2. 声明子类和重写方法

进一步考虑在标准 JavaScript 原型之外 Dojo 完成什么功能，首先查看 Person 子类的声明：

```
dojo.declare("decafbad.school.Student", decafbad.school.Person, {
    constructor: function(name, grade) {
        // Note that the inherited constructor is automatically called.
        this.grade = grade;
    },
    getGrade: function() {
        return this.grade;
    }
});
```

新的 Student 类是 Person 的子类，借助 Dojo 的继承机制，Student 继承了 Person 的所有方法，包括构造函数。当创建 Student 的新实例时，在调用 Student 的 constructor 方法之前会自动调用

Person 类的 constructor 方法。

这就是其他语言所支持的基于类的编程方式，但是这需要在 JavaScript 的基于原型的环境中做一些额外工作，以支持继承和其他典型的 OOP 功能。这就是 dojo.declare()提供的功能。

现在考虑下面的代码，查看如何重写从父类继承的方法：

```
dojo.declare("decafbad.school.MaleStudent", decafbad.school.Student, {
    getName: function() {
        var name = this.inherited(arguments);
        return "Mr. " + name;
    }
});
```

MaleStudent 类从 Student 那里继承了 getName()方法，但是将其重写。新的实现使用 Dojo 提供的 this.inherited(arguments)来调用父类方法，并将它自己添加的前缀放在该调用返回的值中。除了 constructor 这一特例之外，在子类中不会自动调用重写的父类方法。

注意，this.inherited()的 arguments 参数是一个内置的 JavaScript 功能。这种调用约定可用来方便地把最初传给当前方法的实参向前传递。

3. 通过混入使用多重继承

Dojo 还支持混入(mixin)形式的多重继承。看看下面这个新的示例：

```
dojo.declare("decafbad.school.DoorUnlocker", null, {
    canUnlockDoors: true,
    constructor: function() {
        this.doorsUnlocked = [];
    },
    unlockDoor: function(door) {
        this.doorsUnlocked.push(door);
        return door + " now unlocked";
    }
});

dojo.declare("decafbad.school.DormAssistant",
    [ decafbad.school.Student, decafbad.school.DoorUnlocker ], { });
```

这里定义了两个类：DoorUnlocker 和 DormAssistant。

第一个类 DoorUnlocker 并不继承任何父类，而是定义了一个属性 canUnlockDoors、一个构造函数和一个方法 unlockDoor。

第二个类 DormAssistant 使用一个数组字面值来声明同时继承自 Student 和 DoorUnlocker。这两个类均称为混入类。这意味着 Dojo 将它们混入，也就是将每个混入类中的所有属性和方法添加到 DormAssistant 类中(按照它们在继承列表中出现的顺序)。这个规则有一个例外，就是构造函数：它们累积到一个内部列表中，用于创建新类，而且按照创建新实例的顺序来调用每个构造函数。

因此，在这个示例中，DormAssistant 是一个被赋予执行 unlockDoor()这个额外附加能力的 Student。Student 类(位于继承列表中的第一位)是 DormAssistant 的正式父类。DoorUnlocker 类则被视为 DormAssistant 具备的额外能力。

4. 使用 dojo.extend()扩展已有类

在声明新类时，使用多重继承混入并匹配功能非常方便，但更方便的是我们能够增强现有类。这就是 dojo.extend()发挥作用的地方：

```
dojo.extend(decafbad.school.Person, {
    _studying: null,
    study: function(subject) {
        this._studying = subject;
        return "Now studying "+subject;
    }
});
```

上面的代码增强 Person 基类，添加了一个新的 study()方法和一个新的数据成员以跟踪正在学习什么内容。该功能特别优秀的地方在于，对基类的增强会传给子类。以 DormAssistant 对象的创建为例：

```
var bar = new decafbad.school.DormAssistant('kim', 'senior');
bar.study('physics');
```

使用 dojo.extend()方法，我们可以在现有窗口部件和类上以一种强大的方式添加自定义功能。这种对现有类的声明后增强为我们对自己的类(以及属于 Dojo 自身的类)进行修改和调整提供了一种方式。

前面给出的示例中的 dojo.extend()调用甚至不需要与原始 Person 类位于同一个模块中。在使用 dojo.require()利用外部包的过程中可以包括 dojo.extend()语句，而不需要预先与其他现有的包和类进行协调。

23.3.3　在 HTML 标记中声明对象

在本章前面我们已经看到，Dojo 的部分魅力就是在 HTML 标记中声明对象。对于利用一些额外功能包装现有 DOM 元素的窗口部件而言，这种对象声明方式几乎立即就能用得上。但是，这项功能并不仅限于窗口部件。使用 Dojo 分析器，我们可以通过 HTML 标记来声明任何 Dojo 类的实例化。

dojo.parser 模块提供了 Dojo 分析器，可以通过在 djConfig 中确保 parseOnLoad 为 true 并在页面脚本中使用 dojo.require("dojo.parser")来启用该分析器。对于本章开头提供的示例，这种启用方式通常也可行。

当加载页面时，分析器扫描整个 DOM 以查找含有 dojoType 属性的元素，出现这个属性表示该元素是一个对象声明，而这个属性的值指定了待实例化对象所属的类。

1. 声明对象

下面研究对象声明，查看下面的 HTML 代码：

```
<html>
    <head>
        <title>Hello Dojo Parser</title>
```

```
    <script type="text/javascript">
        djConfig = {
            isDebug: true,
            parseOnLoad: true,
            modulePaths: {
                "decafbad": "../../ex-dojo-core/decafbad",
            },
        };
    </script>

    <script type="text/javascript"
        src="../dojodev/dojo/dojo.js"></script>

    <script type="text/javascript">
        dojo.require("dojo.parser");
        dojo.require("decafbad.things");
    </script>

</head>
<body>

    <h1>Hello Dojo Parser</h1>

    <div dojoType="decafbad.things.thingA" jsId="decafbad.stuff.someThingA"
        class="someThingA" alpha="true" beta="three, four"
        foo="bar" baz="123" xyzzy="hello">
            <p>Alpha: <span class="alpha">default</span></p>
            <p>Beta:  <span class="beta">default</span></p>
            <p>Foo:   <span class="foo">default</span></p>
            <p>Baz:   <span class="baz">default</span></p>
            <p>Xyzzy: <span class="xyzzy">default</span></p>

    </div>

    </body>
</html>
```

这里的大部分代码与在本章中目前已经看到过的代码相似。第一个<script>块设置 djConfig，然后在接下来的<script>元素中加载 Dojo 核心。然后是两个 dojo.require()调用，用于加载 Dojo 分析器和新模块 decafbad.thingA。

在这个页面的<body>元素中，可以看到一个使用 dojoType="decafbad.thingA"声明的对象实例，该对象有几个自定义属性，而且其中还包含一些段落元素。

在这几个属性中，jsId 是第一次出现的属性。这个属性的值标识了一个全局名称空间中的变量，分析器应该将新实例化的对象存储到这个变量中。在这里，新对象将存储到 decafbad.stuff.someThingA 中。对于引用和连接标记中声明的多个对象而言，这项功能非常有用。有关这方面的更多内容，将在稍后的章节中讨论 Dijit 窗口部件时讲解。

2. 定义类以支持标记中的声明

现在，查看模块 decafbad/thingA.js 的实现。

```
dojo.provide("decafbad.things");

dojo.declare("decafbad.things.thingA", null, {

    alpha: false,
    beta:  [ 'one', 'two' ],
    foo:   'default',
    baz:   456,

    constructor: function(args, node) {

        dojo.mixin(this, args);

        dojo.query('span', node).forEach(function(ele) {

            var name = ele.className;
            var val  = this[name];

            ele.innerHTML = val ?
                '[' + (typeof val) + "] " + val :
                'undefined';

        }, this);

    }

});
```

这里没有多少代码，但是其中发生了很多事情，这也是贯穿 Dojo 的一种特色。

首先，通过调用 dojo.provide()建立模块，然后开始声明类 decafbad.things.thingA。

这个类首先定义了几个属性，每个属性的 JavaScript 类型都不同。这一点非常重要，因为分析器会调查类，然后根据默认值的原始类型对属性字符数据执行适当的类型转换。查看这项功能的实际运行情况之后，我们就更加能够认识到它的作用。

在属性定义之后是构造函数，它的参数由分析器提供：

- args　一个从标记中收集属性的对象。
- node　一个指向声明该对象实例的 DOM 元素的引用。

在这个构造函数实现中，所做的第一件事情就是调用 dojo.mixin(this, args)。这是从 args 那里获取接收到的所有已转换属性并将它们分配给对象属性(类似于混入类，dojo.mixin()将给定的属性混入到给定的对象中)的快速方式。

下一部分是 dojo.query()和 forEach()的链式调用，我们将在第 24 章更详细地讨论这种构造。简而言之，它搜索 node 的内容以查找元素，并将一个匿名函数应用到所有这些元素中。最后一个参数(this)使匿名函数在正在构造的对象的上下文中执行。

对于每个元素，该匿名函数检查 CSS 类名并尝试使用同名对象属性的表示更新它的

innerHTML 属性。因此，如果有一个元素，那么应该将其修改为 this.alpha 的类型和内容表示(如果这个属性在该对象中有对应的值)。否则，应该读取"undefined"。

3. 标记对象声明的实际运行情况

图 23-5 给出了把这里的代码结合起来的最终效果。

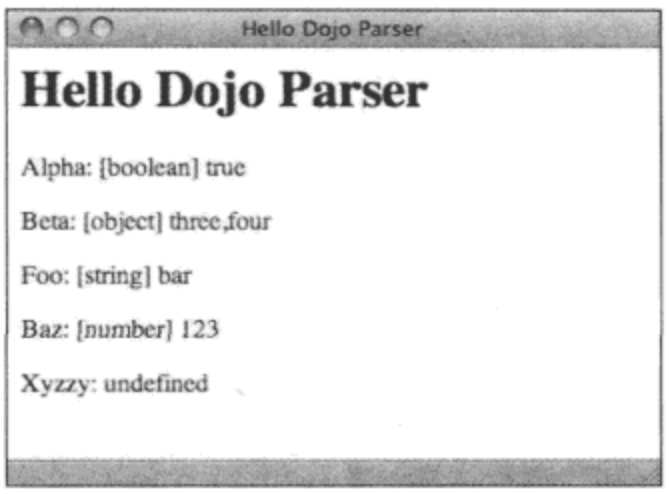


图 23-5

下面给出了有关该输出结果以及所发生事情的注意事项：

- 由于标记中出现 dojoType="decafbad.things.thingA"属性，因此在加载页面时 Dojo 分析器创建一个 decafbad.things.thingA 实例。
- 由于出现 jsId="decafbad.stuff.someThingA"，因此其他代码可以使用 decafbad.stuff.someThingA 来引用这个新实例化的对象。
- 元素中的每个属性都会传给一个 JavaScript 对象中的 decafbad.things.thingA 构造函数，并执行类型转换以匹配类声明中已有的属性默认值的类型。
- 在构造 decafbad.things.thingA 实例的过程中，声明对象的 元素的内容将被修改以反映新实例的属性。
- 还有一个注意事项：属性 xyzzy 显示为 undefined。

最后一项成立的原因是，并不是标记中的所有属性都经过转换并传给构造函数。再次查看 decafbad.things.thingA 的实现，我们会注意到类声明中的属性列表并不包含 xyzzy。

由于分析器调查类声明中定义的属性及其类型，因此它只转换声明中指定的属性并为它们提供数据。因此，尽管 xyxxy(甚至还有 class)均以属性的形式出现在对象声明 元素中，但它们并未传给构造函数，因为它们并不属于类声明。换言之，如果类不需要，那么分析器就不会提供数据。

4. 同时针对声明式用法和编程式用法设计类

我们也可以绕开分析器，转而通过编程方式创建一个针对标记中的声明而设计的类的新实例。例如，可以编写类似下面的代码：

```
var new_thing = new decafbad.things.thingA(
    { alpha: true },
    dojo.byId('someDiv')
);
```

毕竟，这基本上就是 dojo.parser 所做的事情。但在某些情况下，更好的做法是拥有一个更简单的用于代码中的构造函数，同时仍然支持标记中的声明。这就是特殊的“静态”类方法 markupFactory 发挥作用的地方。

```
dojo.declare("decafbad.things.thingB", null, {

    alpha: false,
    beta:  [ 'one', 'two' ],
    foo:   'default',
    baz:   456,

    constructor: function(alpha, beta, foo, baz) {
        this.alpha = alpha;
        this.beta  = beta;
        this.foo   = foo;
        this.baz   = baz;
    },

    markupFactory: function(args, node, thisClass) {
        var instance = new thisClass(
            args.alpha, args.beta, args.foo, args.baz
        );
        return instance;
    }

});
```

在这个新的 decafbad.things.thingB 类中，我们可以看到它同时具有构造函数 constructor 和 markupFactory 方法。Dojo 分析器使用构造函数 constructor 实例化对象，除非它发现有一个 markupFactory 方法可用。

markupFactory 方法的工作方式与其他语言中的“静态”类方法相似，也就是说，我们并不是在类的单个对象实例上调用它，而是在类原型上调用它。这个方法的参数(与前面看到的 constructor 的签名类似)包括：

- **args**　从标记属性中收集的一组参数。
- **node**　一个指向声明对象实例的 DOM 节点的引用。
- **thisClass**　一个指向被创建实例所属类的引用。

当调用时，这个方法负责创建并返回 thisClass 类的一个新实例，而且可以在此期间完成它所需要做的任何工作。这样，我们就可以基于编程式上下文或在标记中声明的方式来维护两种彼此独立的创建类实例的方式。

考虑到混入类和 dojo.mixin()方法，markupFactory 的存在为我们增强并非为在标记中使用而设计的现有类提供了可能性(也就是通过新混入的 markupFactory 实现)。

5. 在标记中声明对象与验证 HTML

在本节中讲解 Dojo 分析器以及在标记中声明对象时，我们会遇到一个问题：如果正在使用这些自定义属性，那么页面如何通过 HTML 验证测试呢？

简而言之，页面不能通过验证，但可能并没有问题。

具体来说，我们需要在下面这两种方式之间进行选择，也就是选择 Dojo 分析器的便利性和不能通过验证的自定义属性，或者选择编程式对象创建方式和有效的标记。

自定义属性能够在浏览器中运行，而且人们认为 Dojo 分析器目前的实现是在便利性和性能之间做出的一种折中。引入 XML 名称空间或使用一些 CSS 类命名约定(两者似乎均为常见的推荐做法)均会造成性能损失，因此 Dojo 团队采取了实用的做法，并不打算在这个充满争议的领域中找出一个能够满足所有人需求的解决方案。

因此，如果打算放弃严格的 HTML 有效性验证，就可能发现在标记中声明对象和窗口部件体系带来的好处是值得的。

另一方面，如果确实坚持严格验证所有标记的有效性，那么 dojo.parser 并不适合您。这是 constructor/markupFactory 方案非常有用的另一个原因，因为所有能够在标记中声明的对象也同样能够通过普通旧式 JS 代码实例化。此外，可以研究如何扩展或替换 dojo.parser 模块，以满足自己可接受的约束条件。

如果感兴趣的话，那么可以到下面给出的网址中阅读一些围绕这个问题的背景知识：

```
www.dojotoolkit.org/book/dojo-porting-guide-0-4-x-0-9/widgets/general
```

23.4 本章小结

在本章中，我们快速浏览了 Dojo 核心，包括如何让 Dojo 在自己的页面中运行，如何管理模块和依赖项，以及如何在 JavaScript 基于原型的系统之上构建 Dojo 的类系统和页面分析器。这为 Dojo 框架的其他所有功能提供了基础，但是我们在自己的代码中同样也可以利用这个基础。

在下一章中，我们将讲解如何使用 Dojo 的 DOM 操作工具。

第24章

操 作 DOM

直到浏览器开始引入动态 HTML(DHTML)时，JavaScript 在构建 Web 应用程序方面才真正变得有用。当然，在早期可以使用它执行一些简单的表单验证，生成简陋的弹出窗口，以及在状态栏中滚动文本；但是当 JavaScript 开发人员真正能够把浏览器窗口当作一张画布(尽管仍然比较简陋)并使用它开发客户端用户界面时，该语言才逐渐开始受到重视。

现如今很难找到一款正在开发的或已经成熟的框架，其核心不努力实现跨浏览器一致性并进行抽象，以减轻在导航和操作浏览器文档对象模型以及处理页面上用户操作产生的事件时的工作量。当然，Dojo 也不例外。实际上，尽管 Dojo 工具集(本身与 DHTML 并没有直接的关联)的功能覆盖面非常广泛，但 Dojo 在它的整个文档中都被称为 DHTML 工具集。

而且，在 Dojo 中我们将找到用于查找、修改和创建 DOM 元素的方式。但除了基本的方式之外，我们还会找到大量的便利方法和约定，既有原创的，也有受到其他 JS 库启发的，它们可用于编写简洁而清晰的代码来完成大规模的修改。Dojo 还提供了一组非常灵活的事件处理抽象机制来完成所有事件的处理，既包括简单的按钮单击事件的处理，也包括使用自定义事件将整个应用程序结合起来。

本章内容简介：

- 查找 DOM 元素
- 处理 DOM 元素列表
- 处理节点列表

24.1 查找 DOM 元素

大多数 DHTML 和 DOM 脚本处理均从一个完整的网页开始，在这个基础之上修改和增强该页面。因此，DOM 脚本处理的一个基础构建块就是能够根据要执行的期望操作找出页面中的元素。

24.1.1 利用 dojo.byId 查找 DOM 元素

查找 DOM 元素的最简单方式是，在标记中提供一个字面值 ID，然后使用 JavaScript 中的 document.getElementById()来获取它。

在 Dojo 中，dojo.byId()方法是构建在这个基本 DOM 功能之上的快捷方式。查看下面的 HTML 代码：

```
<div id="sect1">
    <h2>Section 1</h2>
    <p id="para1">
        Lorem ipsum dolor sit amet, consectetuer adipiscing elit.
    </p>
</div>
```

为了获取前面示例中 和<p>元素上的句柄，可以编写如下的代码：

```
var sect1 = dojo.byId('sect1');
var para1 = dojo.byId('para1');
```

除了简洁性之外，dojo.byId()和 document.getElementById()之间的主要区别在于，如果向其传递一个非字符串数据作为其参数(也就是已经获取的一个元素)，那么 dojo.byId()简单地返回该参数。例如：

```
var sect1_dup = dojo.byId(sect1); // sect1_dup == sect1
```

有时候，我们很容易获取一个已知的 ID 或根据某种命名约定推导出 ID。还有些时候，我们在导航 DOM 的过程中已经获得某个元素的引用。为了构建能够适应这两种情况的灵活的函数，用来处理元素的代码可以将接收的参数包装在 dojo.byId()调用中并继续进行处理，就像下面这样：

```
function getTagName(id_or_ele) {
    var ele = dojo.byId(id_or_ele);
    return ele.tagName;
}
```

24.1.2 利用 dojo.query 查找元素

通过 ID 加载一个元素固然有用，但是很少有如此简单的任务。更常见的情况是，操作数量很大的一组元素。但是，为了确保页面中所有相关的元素都已经拥有一个 ID，需要完成大量的协调工作并编写杂乱的标记。

为此，DOM 提供了诸如 document.getElementsByTagName()和 document.getElementsByName()这样的方法来获取一组元素。此外，个别元素还提供诸如 childNodes 和 parentNode 这样的属性来导航页面层次结构。通过在精心构造的循环中使用这些方法和属性，并且在标记中查找诸如标记名称和 CSS 类名这样的语义挂钩，我们可以很好地定位并操作任何元素布局。

但问题是用于处理 DOM 导航的代码的规模和复杂性会快速增长，而且在此之前我们甚至还没有开始进行最初希望对元素执行的操作。

但是，标识页面上的一组元素的问题不是已经由 Web 开发套件中的另一部分(也就是 CSS 选择器)解决了吗？使用 CSS 选择器声明查找元素的模式，并将该模式应用到 CSS 样式中。如果能

够在 JavaScript 中查找和处理 DOM 节点时执行同样的操作，那么情况会怎样？

事实证明，这就是 dojo.query()提供的功能。考虑下面的 HMTL 代码片段：

```
<div id="ex1">
    <ul>
        <li><a href="http://decafbad.com">Item #1</a></li>
        <li class="seconditem dojolink">
            <a href="http://dojoproject.org">Item
                <span class="num">#2</span></a>
        </li>
        <li>
            <span>Sub-list</span>
            <ul>
                <li><a href="http://delicious.com">Item
                    <span class="num letter">B</span></a></li>
                <li><a href="http://mozilla.org">Item C</a></li>
            </ul>
        </li>
        <li><a href="http://wrox.com">Item
            <span class="num">#5</span></a></li>
    </ul>
</div>
```

现在，查看如何艰难地找出一些元素：

```
var num_spans = [];
var root_ele = document.getElementById('ex2');
var spans = root_ele.getElementsByTagName('span');
for (var i=0,span; span=spans[i]; i++) {
    if (/\s?num\s?/.test(span.classname)) {
        num_spans.push(span);
    }
}
```

首先，需要花费一些时间解释这段代码到底在做些什么工作，而且我们并不是有意要编写出如此混乱的代码。

前面的代码通过 ID 查找元素 ，然后在该元素中搜索元素。接下来，处理元素集合，搜索带有 CSS 类名'num'的元素。注意，这里使用了正则表达式，因为 CSS 类名可能是以空格分隔的列表，而且样本 HTML 中的一个元素刚好就有一个带有多个 CSS 类的属性。

一旦找到匹配的元素，就把它们收集到一个数组中。突出显示的行给出了找到这些项的地方：

```
<div id="ex1">
    <ul>
        <li><a href="http://decafbad.com">Item #1</a></li>
        <li class="seconditem dojolink">
            <a href="http://dojoproject.org">Item
                <span class="num">#2</span></a>
```

```
        </li>
        <li>
            <span>Sub-list</span>
            <ul>
                <li><a href="http://delicious.com">Item
                    <span class="num letter">B</span></a></li>
                <li><a href="http://mozilla.org">Item C</a></li>
            </ul>
        </li>
        <li><a href="http://wrox.com">Item
            <span class="num">#5</span></a></li>
    </ul>
</div>
```

现在，查看使用 dojo.query()完成相同工作的代码：

```
var num_spans = dojo.query('span.num', 'ex2');
```

代码量急剧减少，而且更加清晰。dojo.query()的第一个参数是一个描述待查找元素的 CSS 3 选择器，而可选的第二个参数可以是一个元素或 ID，可以将搜索范围限制到该元素的子元素中。

dojo.query()还支持更高级的选择器。例如：

```
var dot_org_links = dojo.query('a[href$=".org"]', 'ex2');
var li_the_first  = dojo.query('li:first-child', 'ex2');
var li_the_third  = dojo.query('li:nth-child(3)', 'ex2');
```

第一条语句查找所有 URL 以.org 结尾的链接。

第二条语句查找所有在父列表中第一次出现的<li>元素。

最后一条语句查找在各自列表中第三次出现的<li>元素。

dojo.query()实现并不支持完整的 CSS3 规范，但它确实支持一个有用的子集。作为参考，表 24-1 列出了在编写本书时已知能够运行的选择器。

表 24-1

选 择 器	作 用
*	匹配所有元素
.class	匹配 CSS 类中含有'class'的元素
#foo	匹配 id="foo"的元素
E	匹配<E>元素
E F	匹配属于<E>元素的子节点的<F>元素
E > F	匹配属于<E>元素的直接子节点的<F>元素
E[foo]	匹配带有属性 foo 的<E>元素
E[foo="bar"]	匹配<E foo="bar">元素
E[foo~="bar"]	匹配属性 foo(由空白符隔开的列表)中包含值 bar 的<E>元素，该选择器对于类似 CSS 类名构造的属性非常有用
E[foo^="bar"]	匹配属性 foo 以 bar 开头的<E>元素

(续表)

选　择　器	作　　用
E[foo$="bar"]	匹配属性 foo 以 bar 结尾的<E>元素
E[foo*="bar"]	匹配属性 foo 包含 bar 的<E>元素
E:nth-child(n)	匹配属于父元素第 n 个子节点的<E>元素
E:nth-child(odd)	匹配属于父元素的奇数子节点的<E>元素
E:nth-child(even)	匹配属于父元素的偶数子节点的<E>元素
E:nth-child(3n+1)	匹配父元素的第 3n+1 个<E>元素(从第一个子节点开始)
E:first-child	匹配属于父元素第一个子节点的<E>元素
E:last-child	匹配属于父元素最后一个子节点的<E>元素
E:not(...)	忽略所有上述选择器

24.2　处理 DOM 元素列表

使用 dojo.query()有些类似于使用 SQL 表达数据库查询——不是通过编程方式遍历集合并进行搜索，而是能够声明需要什么数据以及想要完成什么操作。到目前为止，我们已经看到如何声明自己需要什么数据，现在就查看如何使用 Dojo 优化结果并声明自己希望对这些结果执行什么操作。

这里的核心就是 dojo.NodeList 类，调用 dojo.query()返回的结果就是这个类的一个实例。dojo.NodeList 类提供了一组有用的方法，可用来过滤、优化以及处理返回结果中包含的节点。而且，这些方法中的大多数本身也返回一个 dojo.NodeList 引用，这样就能够将这些方法调用链接起来，从而它们读起来就像是一种声明式查询语言。

24.2.1　过滤和优化节点列表

一旦了解如何使用 CSS 选择器和 dojo.query()查找元素的基础知识，就可以简单调用 NodeList 的方法来进一步优化和过滤结果。

1. 利用.filter()过滤 NodeList 结果

NodeList 中最简单的方法是.filter()。利用它可以将一些节点从列表中排除，可通过 CSS 选择器来完成该操作，或者应用一个回调函数，根据该回调函数的返回值来决定是否将该节点包括在输出列表中。

查看下面的标记：

```
<ul id="ex1">
    <li><a href="http://decafbad.com">Item #1</a></li>
    <li class="seconditem dojolink">
        <a href="http://dojoproject.org">Item
            <span class="num">#2</span></a>
    </li>
    <li class="thirditem">
```

```
        <a href="http://w3.org">Item #3</a>
    </li>
    <li>
        <span>Sub-list</span>
        <ul>
            <li><a href="http://yahoo.com">Item A</a></li>
            <li><a href="http://delicious.com">Item
                <span class="num letter">B</span></a></li>
            <li><a href="http://mozilla.org">Item C</a></li>
        </ul>
    </li>
    <li id="item4"><a href="http://getfirebug.com">Item #4</a></li>
    <li><a href="http://wrox.com">Item
        <span class="num">#5</span></a></li>
</ul>
```

记住这一点，然后考虑下面的代码将会执行什么操作：

```
var items = dojo
    .query('a')
    .filter('a[href$=".org"]');
```

因为 JavaScript 允许在链式函数调用的句点表示法中间出现空白符，所以可以让代码格式更加清晰，从而更容易说明应该发生的操作，并在 JavaScript 中模仿声明式查询语言。

第一个.query()调用查找页面中的所有链接，然后.filter()调用将结果的范围缩小到 URL 以.org 结尾的链接。

```
<ul>
    <li><a href="http://decafbad.com">Item #1</a></li>
    <li class="seconditem dojolink">
        <a href="http://dojoproject.org">Item
          <span class="num">#2</span></a>
    </li>
<li class="thirditem">
    <a href="http://w3.org">Item #3</a>
</li>
<li>
    <span>Sub-list</span>
    <ul>
        <li><a href="http://yahoo.com">Item A</a></li>
        <li><a href="http://delicious.com">Item
            <span class="num letter">B</span></a></li>
        <li><a href="http://mozilla.org">Item C</a></li>
    </ul>
</li>
<li id="item4"><a href="http://getfirebug.com">Item #4</a></li>
<li><a href="http://wrox.com">Item
        <span class="num">#5</span></a></li>
</ul>
```

当然，这并不是一个非常实用的示例。本来在开始时可以直接将过滤器选择器作为查询自身。

当我们把.filter()与其他调用(这些调用用来修改超出最初的.query()调用能够找到的结果)链接在一起(我们将在稍后的.map()调用中看到这一点)时，它的效用就会变得更加明显。

接下来，查看下面的.filter()调用会对上述示例中的标记做些什么：

```
var items = dojo
    .query('a[href$=".org"]')
    .filter(function(el) {
        return dojo.hasClass(el.parentNode, 'dojolink');
    }, this);
```

与之前一样，.query()调用的结果返回页面中的所有链接。然后，链式调用中的.filter()调用将结果的范围缩小到父列表项的 CSS 类名包括 'dojolink' 的链接。在前面的 HTML 代码中，这样的链接就是下面这一个结果：

```
<ul id="ex1">
    <li><a href="http://decafbad.com">Item #1</a></li>
    <li class="seconditem dojolink">
         <a href="http://dojoproject.org">Item
             <span class="num">#2</span></a>
    </li>
    <li class="thirditem">
        <a href="http://w3.org">Item #3</a>
</li>
  <li>
    <span>Sub-list</span>
    <ul>
         <li><a href="http://yahoo.com">Item A</a></li>
        <li><a href="http://delicious.com">Item
            <span class="num letter">B</span></a></li>
     <li><a href="http://mozilla.org">Item C</a></li>
    </ul>
 </li>
 <li id="item4"><a href="http://getfirebug.com">Item #4</a></li>
 <li><a href="http://wrox.com">Item
      <span class="num">#5</span></a></li>
</ul>
```

此外，下面这件事情也是.query()调用不能做到的工作：

```
var items2 = dojo
    .query('a')
    .filter(function(el) {
        return (Math.random() > 0.5);
    }, this);
```

这段代码返回页面中所有链接的随机样本，任何一项都有 50%的机会出现在这个列表中。尽管这个过滤器相当简单，但是我们实际上可以在.filter()调用中使用任何选择条件，并且只会在代码中增加少许复杂度。

但是，在提到复杂度时(如果发现在自己所处的情况下需要稍微复杂一些)，.filter()方法实际上

为我们的回调函数提供了另外两个参数。

```
var items3 = dojo
    .query('a[href$=".com"]')
    .filter(function(item, index, arr) {
        return index % 2;
    }, this);
```

这段代码返回页面上其他 URL 以.com 结尾的所有链接，这就演示了传给回调的完整参数集：

- item　正在处理的当前 DOM 节点。
- index　当前节点在列表中的索引(从 0 开始)。
- arr　由.filter()调用考虑的所有节点所组成的数组，因此下面这个表达式永远成立：arr[index] === item。

此外，我们可以使用提供的索引和完整的数组执行一些操作，例如从当前位置向前或向后查找以判断是否包含当前项。

2. 利用.query()执行子查询

另一种缩小或扩展 NodeList 结果范围的方式是使用.query()方法，以 NodeList 中的每个元素为根来执行 dojo.query()并收集返回的结果。查看下面的标记：

```
<div>
    <h2>Example HTML Code</h2>
    <ul id="ex2">
        <li class="foo bar baz"><strong>One</strong></li>
        <li>2
            <span>Sublist</span>
            <ul>
                <li>Alpha</li>
                <li>Beta</li>
            </ul>
        </li>
        <li>
            <span>Yet another sublist</span>
            <ul>
                <li>Foo</li>
                <li>Bar</li>
            </ul>
        </li>
        <li>
            <span>An ordered sublist</span>
            <ol>
                <li>One</li>
                <li>Two</li>
            </ol>
        </li>
    </ul>
</div>
```

给定之前的 HTML 代码，考虑如下的代码：

```
var items = dojo.query('div > ul').query('ul > li');
```

这个查询返回本身也是列表成员的<ul>列表中的所有<li>元素。

第一个.query()调用查找属于 元素的直接子节点的<ul>元素，这段 HTML 代码中只有一个这样的元素。

注意，第二个链式调用的.query()在第一个调用返回的结果上进行操作，搜索<li>元素(作为<ul>元素的子节点)。第二个.query()调用并没有包括第一个调用的结果，因为原始结果中的每个元素均用作子查询的基础，它们本身并不属于子查询的结果。

在调用链末尾的 NodeList 中，包含以下突出显示的项：

```
<div>
    <h2>Example HTML Code</h2>
    <ul id="ex2">
        <li class="foo bar baz"><strong>One</strong></li>
        <li>
            <span>Sublist</span>
            <ul>
                <li>Alpha</li>
                <li>Beta</li>
            </ul>
</li>
<li>
    <span>Yet another sublist</span>
     <ul>
        <li>Foo</li>
        <li>Bar</li>
     </ul>
</li>
<li>
   <span>An ordered sublist</span>
    <ol>
       <li>One</li>
       <li>Two</li>
    </ol>
  </li>
 </ul>
</div>
```

此处将不会返回
One
和
Two
项，这是因为它们位于<ol>有序列表中，而其他项则全部位于要求的<ul>列表中。

这是一个人为的示例，但并不是简单的 CSS 选择器能够实现的功能。此外，要记住.query()只是 NodeList 的一个方法(而且 NodeList 的大多数方法还会返回 NodeList)。因此，子查询可以出现在 NodeList 调用链中的任何地方，而不仅仅是紧跟着另一个.query()调用。

这意味着我们不仅可以在查询结果上执行子查询，而且可以在其他的列表修改操作(在本章剩余内容中讲解)所产生的列表结果上执行子查询。

3. 利用.map()转换 NodeList

NodeList 提供的.map()方法可用来把一个回调函数应用到列表中的每个节点上，该函数的返回值将替换在执行.map()调用之后返回的 NodeList 中的节点。这样就可以对整个列表执行逐个节点的转换。

再次查看用来演示.filter()方法的标记。给定这些标记，考虑下面的代码的运行结果：

```
var items = dojo
    .query('a[href$=".org"]')
    .map(function(el) {
        return el.parentNode;
    }, this);
```

dojo.query()调用中的选择器在本章前面曾经用到过，它查找页面中 URL 以.org 结尾的所有链接。链式.map()调用的执行结果就是将每个这样的链接替换成一个指向各自的<li>父节点的引用。

对于示例 HTML 代码，下面是该代码将要找到的项：

```
<ul id="ex1">
    <li><a href="http://decafbad.com">Item #1</a></li>
    <li class="seconditem dojolink">
        <a href="http://dojoproject.org">Item
            <span class="num">#2</span></a>
    </li>
    <li class="thirditem">
        <a href="http://w3.org">Item #3</a>
    </li>
    <li>
        <span>Sub-list</span>
        <ul>
            <li><a href="http://yahoo.com">Item A</a></li>
            <li><a href="http://delicious.com">Item
                <span class="num letter">B</span></a></li>
            <li><a href="http://mozilla.org">Item C</a></li>
        </ul>
    </li>
    <li id="item4"><a href="http://getfirebug.com">Item #4</a></li>
    <li><a href="http://wrox.com">Item
        <span class="num">#5</span></a></li>
</ul>
```

回调函数完成的这种转换可用来对文档中选中的节点执行优化。可以检查节点的任何属性，对每个节点执行复杂的查找和比较，并可以完成所有其他不容易使用 CSS 表达的功能。

注意，.map()方法还将一个索引和完整的数组作为参数传给回调函数，就像.filter()所做的那样：

```
var items2 = dojo
    .query('a[href$=".org"]')
    .map(function(el, index, arr) {
        return (index % 2) ? el : el.parentNode;
    }, this);
```

这段代码将一个链接列表映射成一个由链接和它的父列表项交替组成的集合。

4. 利用.concat()连接 NodeList

.filter()调用用来减少列表中的节点数量，而.concat()方法则用于增加节点数量。查看下面这段 HTML 代码：

```
<ul id="ex3">
    <li class="foo bar baz"><strong>One</strong></li>
    <li class="bar baz"><em>Two</em></li>
    <li class="foo baz"><u>Three</u></li>
    <li class="baz"><small>Four</small></li>
    <li class="foo">Five</li>
</ul>
```

考虑下面的查询链(包括一系列.concat()调用)的结果：

```
var items = dojo
    .query("strong")
    .concat(dojo.query("em"))
    .concat(dojo.query("u"))
    .concat(dojo.query("small"))
    .map(function (el) {
        return el.parentNode;
    });
```

每个.concat()调用将嵌套的 dojo.query()调用的结果连接到 NodeList 中。在执行.map()转换之后，该结果包含下面突出显示的节点：

```
<ul id="ex3">
    <li class="foo bar baz"><strong>One</strong></li>
    <li class="bar baz"><em>Two</em></li>
    <li class="foo baz"><u>Three</u></li>
    <li class="baz"><small>Four</small></li>
    <li class="foo">Five</li>
</ul>
```

利用 dojo.extend()增强 NodeList

如果经常发现自己在代码中使用.concat()合并 NodeList 或者进行相似的操作，那么可能会希望定制 Dojo 环境，让这种处理变得更加优雅。

相比于使用.concat(dojo.query())调用，下面的代码能够更好地完成任务：

```
dojo.extend(dojo.NodeList, {
    also: function(queryStr) {
        return this.concat(dojo.query(queryStr))
    }
});

var items2 = dojo
    .query("strong")
    .also("em")
```

```
    .also("u")
    .also("small")
    .map(function (el) {
        return el.parentNode;
    });
```

dojo.extend()调用将一个名为.also()的新方法安装到 dojo.NodeList 中，该方法可以缩短前面看到的.concat(dojo.query())调用。

可将这种方式视为 dojo.query()和 NodeList 系统的一个扩展点。它是一种前一章中介绍的语言增强，而且这些增强本身可用于 Dojo 工具集，从而可以根据需要完成自己的改进和定制工作。

5. 利用 slice()提取 NodeList 的子集

NodeList 提供的.slice()方法可用来指定列表中的确切起始和结束位置，以便提取子集。在诸如为较大型的节点集合构建分页窗口这样的场合中，这个方法可能非常有用。返回到有关.map()的讨论中给出的 HTML 列表，考虑下面代码的运行结果：

```
var items = dojo
    .query('a')
    .map(function(el) {
        return el.parentNode;
    }, this)
    .slice(1,4);
```

.slice()的参数是数组索引(从 0 开始)。第一个参数指示从哪里开始包括节点，而第二个可选参数指示在哪里结束。第一个参数指定的索引处的节点将被提取，而结束索引处的节点则不会被提供。

前面的示例包括在页面中找到的第二个到第四个链接，在下面的示例中突出显示了这些链接：

```
<ul id="ex1">
    <li><a href="http://decafbad.com">Item #1</a></li>
    <li class="seconditem dojolink">
        <a href="http://dojoproject.org">Item
            <span class="num">#2</span></a>
    </li>
    <li class="thirditem">
        <a href="http://w3.org">Item #3</a>
    </li>
    <li>
        <span>Sub-list</span>
        <ul>
            <li><a href="http://yahoo.com">Item A</a></li>
            <li><a href="http://delicious.com">Item
                <span class="num letter">B</span></a></li>
            <li><a href="http://mozilla.org">Item C</a></li>
        </ul>
    </li>
    <li id="item4"><a href="http://getfirebug.com">Item #4</a></li>
```

```
    <li><a href="http://wrox.com">Item
        <span class="num">#5</span></a></li>
</ul>
```

6. 利用 splice()就地编辑 NodeList

NodeList 的.splice()方法可用于就地编辑列表，并将列表的一个项子集完全地分离出来。

这个方法与其他方法略有不同。尽管.splice()与其他方法一样返回一个 NodeList，但它还会对接收到的 NodeList 进行修改。例如，考虑前面的.slice()示例的一个变体：

```
var orig_items = dojo
    .query('a')
    .map(function(el) {
        return el.parentNode;
    }, this);

var sub_items = orig_items
    .splice(1,4);
```

执行这段代码后，sub_items 中的最终列表看上去与.slice()的结果类似。然而，虽然在执行.splice()之前 orig_items 列表包含了所有的链接列表项，但是该列表的内容被.splice()修改，就像下面这样：

```
<ul id="ex1">
    <li><a href="http://decafbad.com">Item #1</a></li>
    <li class="seconditem dojolink">
        <a href="http://dojoproject.org">Item
            <span class="num">#2</span></a>
    </li>
    <li class="thirditem">
        <a href="http://w3.org">Item #3</a>
    <li>
        <span>Sub-list</span>
        <ul>
            <li><a href="http://yahoo.com">Item A</a></li>
            <li><a href="http://delicious.com">Item
                <span class="num letter">B</span></a></li>
            <li><a href="http://mozilla.org">Item C</a></li>
        </ul>
    </li>
    <li id="item4"><a href="http://getfirebug.com">Item #4</a></li>
    <li><a href="http://wrox.com">Item
        <span class="num">#5</span></a></li>
</ul>
```

换言之，.slice()只提取复制的项子集，而.splice()则是将项子集从原始列表中完全地移除。

.splice()调用在起始和结束索引参数之后也接受数量可变的可选参数，这些参数的值将插入以替换移除的项。下面给出了先前代码的修改版本：

```
var orig_items = dojo
```

```
    .query('a')
    .map(function(el) {
        return el.parentNode;
    }, this);

var replacements = dojo.query('span');

var sub_items = orig_items
    .splice(1, 4, replacements[0], replacements[1]);
```

这段代码将会把链接项从列表中移除，并将其替换成一对元素：

```
<ul id="ex3">
    <li><a href="http://decafbad.com">Item #1</a></li>
    <li class="seconditem dojolink">
        <a href="http://dojoproject.org">Item
            <span class="num">#2</span></a>
    </li>
    <li class="thirditem">
        <a href="http://w3.org">Item #3</a>
    <li>
        <span>Sub-list</span>
        <ul>
            <li><a href="http://yahoo.com">Item A</a></li>
            <li><a href="http://delicious.com">Item
              <span class="num letter">B</span></a></li>
            <li><a href="http://mozilla.org">Item C</a></li>
        </ul>
    </li>
    <li id="item4"><a href="http://getfirebug.com">Item #4</a></li>
    <li><a href="http://wrox.com">Item
        <span class="num">#5</span></a></li>
</ul>
```

24.2.2 处理节点列表

既然我们已经了解如何使用 dojo.query()和 NodeList 提供的类似声明式的语法来查找、过滤、优化和转换 DOM 节点列表，因此下面研究能够对这些列表中的节点执行什么操作。

1. 利用 forEach 将函数应用于每个节点

对于能够用来操作 NodeList 内容的方法，最基本的就是.forEach()方法。这个方法接受一个回调函数作为参数，并将该参数应用于列表中的每个节点。以下面的代码为例：

```
dojo.query("a[href$='.org']")
    .forEach(function(el) {
        el.style.backgroundColor = "#8f8";
    });
```

前面的代码查找URL 以.org 结尾的每个链接，并将每个这样的链接的背景颜色修改为浅绿色。

.forEach()接受的第二个参数定义了一个上下文，将在这个上下文中调用匿名函数。考虑如下的代码：

```
dojo.declare("decafbad.exForEach", null, {

    the_color: "#f88",

    doTheThing: function() {

        dojo.query('a')
            .map(function(el) {
                return el.parentNode;
            })
            .forEach(function(el) {
                el.style.backgroundColor = this.the_color;
            }, this);

    }

});
var obj = new decafbad.exForEach();
obj.doTheThing();
```

在这段代码中，我们定义了一个新类 decafbad.exForEach，它有一个名为 the_color 的数据成员和一个名为 doTheThing()的方法。

在 doTheThing()的实现中，有一个.forEach()调用，它的回调函数实参需要使用 the_color 的值。这就是.forEach()的第二个参数的作用：通过传递这个方法的 this 值，就会在相同的 this 作用域上下文中执行.forEach()中的匿名函数。如果忽略.forEach()的这个参数，那么匿名函数的 this 值将指向浏览器的 window 对象而不是 exForEach 实例。

注意，这种方式对于传递对象方法(而不是匿名函数)的引用也非常有用，就像下面这样：

```
dojo.declare("decafbad.exForEach2", null, {

    the_colors:  [ "#f88", "#8f8" ],

    mapNode: function(el, index, arr) {
        return el.parentNode;
    },

    processNode: function(el, index, arr) {
        el.style.backgroundColor =
            this.the_colors[ (index % 2) ? 0 : 1 ];
    },

    doTheThing: function() {
        dojo.query('a', 'ex1')
            .map(this.mapNode, this)
            .forEach(this.processNode, this)
    }
```

```
});
var obj2 = new decafbad.exForEach2();
obj2.doTheThing();
```

在这个示例中，doTheThing()方法调用单独的对象方法 processNode()来完成.forEach()调用中的节点处理工作。如果发现自己在编写类似的回调作为内联匿名函数，并发现有机会创建一个新的共享方法来处理所有这些情况，这个方法就非常有用。我们可以看到.map()方法也支持相同的做法。

还要注意的是，在这个示例中，.forEach()方法会向回调函数传递索引和数组参数，就像.filter()和.map()所做的一样。在前面的示例中定义的 processNode()方法引用了一个颜色数组，这些颜色将用作节点的交替样式。

.forEach()还有一个有趣的功能，它也接受一个更加简洁的代码片段，这是一个字符串(而不是函数引用)类型的参数：

```
dojo.query("a:not([href$='.org'])")
    .forEach('item.style.backgroundColor="#ff8"');
```

这段代码查找 URL 不是以.org 结尾的所有链接，并将所有这些链接的背景颜色设为浅黄色。

既然希望执行的操作是简单的单行代码，因此很容易将该代码作为一个字符串传给.forEach()。然后.forEach()方法在如下预定义变量提供的上下文中执行该代码：

- item　正在处理的当前 DOM 节点。
- index　当前节点在列表中的索引(从 0 开始)。
- arr　由.forEach()调用考虑的所有节点所组成的数组，因此下面这个表达式永远成立：arr[index] === item。

这些预定义变量实际上与传入回调函数的参数相同，但可以让代码片段使用它们。

2. 利用.style()操作节点样式

.forEach()可用来对 DOM 节点执行任何操作，但是有些特定的任务要比其他操作更加经常需要用到。对于这些任务，NodeList 类提供了几个更加快捷的方法，即.style()、.attr()和.addContent()。

直接查看演示，考虑下面的标记：

```
<ul id="ex4">
    <li><a href="http://decafbad.com">Link #1</a></li>
    <li><a href="http://dojoproject.org">Link #2</a></li>
    <li><a href="http://getfirebug.com">Link #3</a></li>
    <li><a href="http://delicious.com">Link #4</a></li>
    <li><a href="http://mozilla.org">Link #5</a></li>
    <li><a href="http://wrox.com">Link #6</a></li>
    <li><a href="http://w3.org">Link #7</a></li>
    <li id="mary">
        <span>Mary</span>
        <span>had</span>
        <span>a</span>
        <span>lamb</span>
    </li>
```

```
</ul>
```

可以使用.style()方法调整 CSS 样式属性：

```
dojo.query('a[href$=".org"]', 'ex4')
    .map(function(el) { return el.parentNode })
    .style('border', '2px solid #000');
```

这段代码查找 URL 以.org 结尾的所有链接，并为它们的列表项父节点设置一个边框。在前面的示例中我们可以看到，.style()方法带有两个参数：

- CSS 样式属性的名称
- 要为该样式属性设置的值

3. 利用.attr()操作节点属性

可以使用.attr()方法修改节点属性：

```
dojo.query('a[href$=".com"]', 'ex4')
    .attr('target', '_new');
```

在上面的代码中，URL 以.com 结尾的所有链接都设置成将在新窗口中打开。与.style()方法一样，.attr()也接受两个参数：

- 属性名称
- 属性值

4. 利用.addContent()添加节点内容

.addContent()是一个更复杂且更有趣的方法，可以使用这个方法向 DOM 中插入新内容：

```
dojo.query('a[href^="http://de"]', 'ex4')
    .addContent(' [New!] ', 'after');
```

这段代码查找 URL 以“http://de”开头的所有链接，并在每个这样的链接后面插入[New!]作为父列表项的子节点。.addContent()方法接受两个参数：

- 待插入的内容
- 要插入该内容的位置

这两个参数都有一些有趣的功能：

```
var img = document.createElement('img');
img.src = 'http://decafbad.com/images/globe.jpg';

dojo.query('a[href^="http://de"]', 'ex4')
    .addContent(img, 'first');
```

.addContent()方法既可以接受字符串形式的原始 HTML 内容，也可以接受预先构造的 DOM 节点作为插入的内容。前面的代码已经演示了这一点，我们创建了一个新的图像元素并提供该元素作为传给.addContent()调用的内容。

对于第二个参数，值“first”会让该内容作为选中元素的第一个子节点插入。这个参数还支持如下几个指定的值。

- **before** 在选中节点之前插入(作为父节点的子节点)。
- **after** 在选中节点之后插入(作为父节点的子节点)。
- **last 或 end** 作为选中节点的最后一个子节点插入。
- **first 或 start** 作为选中节点的第一个子节点插入。

这个参数还可以是一个数字索引，计数选中节点中的子节点：

```
dojo.query('li#mary', 'ex4')
    .addContent('<span>little</span> ', 6);
```

注意，尽管在标记中只有 4 个元素构成了“Mary had a lamb”，但元素的子节点列表还包括文本节点，因此在索引计数时也会将文本节点计算在内。

图 24-1 给出了演示.style()、.attr()和.addContent()方法的代码样本的运行结果。

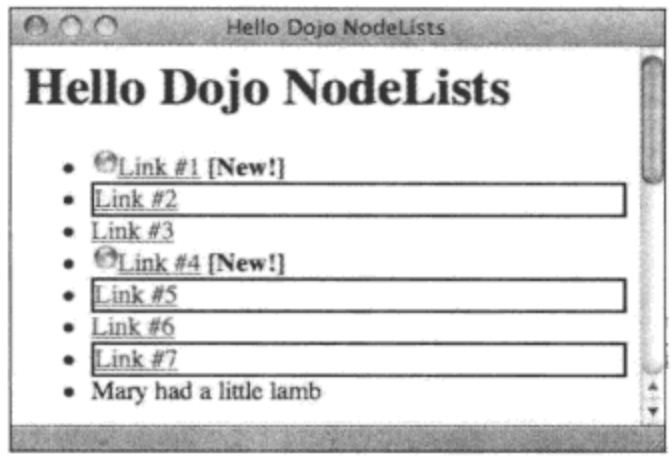


图 24-1

5. 利用.addClass()和.removeClass()操作 CSS 类

前面给出的一些示例直接操作元素上的样式属性，而在重要的 Web 应用程序中，这实际上应该属于谨慎使用的方式(如果确实需要使用的话)。相反，应该在 CSS 中使用基于类的选择器单独声明这些样式。然后，可以操作元素中的 CSS 类名来选择要应用哪些预定义样式。这就在标记、样式和行为之间建立了松散耦合关系。

要实行上面概述的措施，具体方式就是使用.addClass()和.removeClass()方法。以下面的标记为例：

```
<style type="text/css">
    .foo      { list-style-type: none }
    .baz      { font-family: monospace; font-weight: bold; font-size: 200% }
    .warning   { background-color: #ffc; border: 3px solid #000;  }
    .selected  { background-color: #cfc; border: 3px dotted #000; }
</style>

<ul id="ex3">
    <li class="foo bar baz"><strong>One</strong></li>
    <li class="bar baz"><em>Two</em></li>
    <li class="foo baz"><u>Three</u></li>
    <li class="baz"><small>Four</small></li>
    <li class="foo">Five</li>
</ul>
```

现在，考虑将如下的代码应用于前面的代码：

```
var list_kids = dojo.query('li > *', 'ex3');

list_kids
    .filter(function(el) { return /^T/.test(el.innerHTML); })
    .map(function(el) { return el.parentNode; })
    .addClass('warning');

list_kids
    .filter(function(el) { return /^F/.test(el.innerHTML); })
    .map (function(el) { return el.parentNode; })
    .addClass('selected');
```

首先，dojo.query()调用将<ul id= "ex3">下的列表项的所有第一个子节点收集起来，并将结果保存到一个中间变量 list_kids 中。接下来，第一个子节点的内容以字母“T”开头的所有列表项均被设置 CSS 类“warning”。然后，将 CSS 类“selected”添加到内容以字母“F”开头的列表项中。

图 24-2 给出了这段代码在浏览器中的运行结果。请注意，列表项“Five”并没有添加 CSS 类“selected”(因此缺少虚线边框，也没有任何背景颜色)。这是因为选择器“li > *”匹配子元素节点，而不匹配文本节点。这只是一个需要引起注意的“陷阱”。

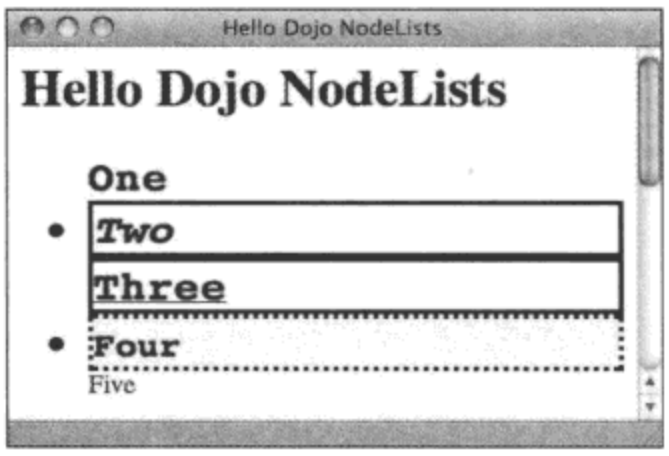


图　24-2

此外还要注意的是，addClass()适当地添加了 CSS 类，而没有干扰其他类，这一点在带有 CSS 类“foo”的 3 个列表项中表现非常明显，该 CSS 类借助“.foo”选择器下的样式属性声明“list-style-type: none”将列表的项目符号隐藏起来。

最后，考虑将所有上述内容应用于.removeClass()方法，它的功能刚好与.addClass()方法相反。例如，下面的代码将遍历所有具有 CSS 类“bar”的列表项并将 CSS 类“baz”移除。

```
dojo.query('li.bar', 'ex3').removeClass('baz');
```

请查看图 24-3 并注意前两个列表项(它们均被标记为同时带有 CSS 类“bar”和“baz”)不再被“baz”涵盖，它们对应的内容也不再使用等宽字体或大字号。

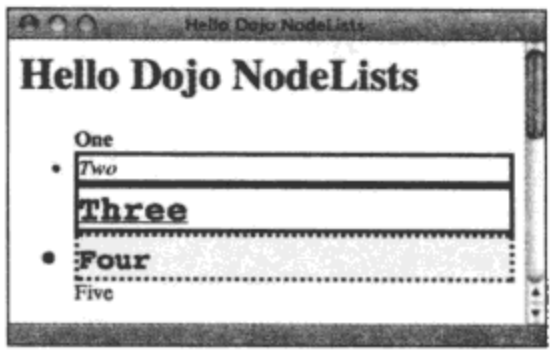


图 24-3

6. 重新定位 DOM 节点

到目前为止，所有用来处理节点的方法所关注的均是属性和样式修改，并没有修改 DOM 结构。但是，除了这些简单的就地调整之外，NodeList 方法.orphan()、.adopt()和.place()还分别提供了在 DOM 结构中删除、移动和插入节点的途径。与目前已经描述过的所有列表处理方式结合使用，借助这些方法，我们只需要少量的代码就可以提供一些非常强大的功能，对文档进行大规模的修改。

查看下面的标记样本：

```
<ul id="ex5">
    <li>
        <h2>.com links</h2>
        <ul id="dotcom"></ul>
    </li>
    <li>
        <h2>.org links</h2>
        <ul id="dotorg"></ul>
    </li>
    <li><a href="http://decafbad.com"><span>Link #1</span></a></li>
    <li><a href="http://dojoproject.org"><span>Link #2</span></a></li>
    <li><a href="http://getfirebug.com"><span>Link #3</span></a></li>
    <li><a href="http://delicious.com"><span>Link #4</span></a></li>
    <li><a href="http://mozilla.org"><span>Link #5</span></a></li>
    <li><a href="http://wrox.com"><span>Link #6</span></a></li>
    <li><a href="http://w3.org"><span>Link #7</span></a></li>
</ul>
```

现在考虑.orphan()方法的演示：

```
var dot_orgs = dojo.query('a[href$=".org"]', 'ex5')
    .map(function(el) { return el.parentNode; })
    .orphan();
```

上面示例中使用的.orphan()方法将所有包含以.org 结尾的链接的列表项从文档中移除，但把列表中现在处于孤立状态的节点放到了 dot_orgs 变量中：

```
<ul id="ex5">
    <li>
        <h2>.com links</h2>
        <ul id="dotcom"></ul>
    </li>
```

```
    <li>
        <h2>.org links</h2>
        <ul id="dotorg"></ul>
    </li>
    <li><a href="http://decafbad.com"><span>Link #1</span></a></li>
    <li><a href="http://dojoproject.org"><span>Link #2</span></a></li>
    <li><a href="http://getfirebug.com"><span>Link #3</span></a></li>
    <li><a href="http://delicious.com"><span>Link #4</span></a></li>
    <li><a href="http://mozilla.org"><span>Link #5</span></a></li>
    <li><a href="http://wrox.com"><span>Link #6</span></a></li>
    <li><a href="http://w3.org"><span>Link #7</span></a></li>
</ul>
```

对于 dot_orgs 中的孤立节点，可以使用.adopt()方法为它们设置存放位置，就像下面这样：

```
dojo.query('#dotorg').adopt(dot_orgs);
```

这段代码将全部的孤立节点插入到 ID 为“dotorg”的列表中，最终的文档结构如下所示：

```
<ul id="ex5">
    <li>
        <h2>.com links</h2>
        <ul id="dotcom"></ul>
    </li>
    <li>
        <h2>.org links</h2>
        <ul id="dotorg">
            <li><a href="http://dojoproject.org"><span>Link #2</span></a></li>
            <li><a href="http://mozilla.org"><span>Link #5</span></a></li>
            <li><a href="http://w3.org"><span>Link #7</span></a></li>
        </ul>
    </li>
<li><a href="http://decafbad.com"><span>Link #1</span></a></li>
<li><a href="http://getfirebug.com"><span>Link #3</span></a></li>
<li><a href="http://delicious.com"><span>Link #4</span></a></li>
<li><a href="http://wrox.com"><span>Link #6</span></a></li>
</ul>
```

最后，考虑.place()调用，它的功能基本上与前面示例中的.adopt()方法相反：

```
dojo.query('a[href$=".com"]', 'ex5')
    .map(function(el) { return el.parentNode; })
    .place('#dotcom');
```

这个调用将所有包含.com 链接的列表项移入 ID 为“dotcom”的列表中，最终的文档结构如下所示：

```
<ul id="ex5">
    <li>
        <h2>.com links</h2>
        <ul id="dotcom">
            <li><a href="http://decafbad.com"><span>Link #1</span></a></li>
            <li><a href="http://getfirebug.com"><span>Link #3</span></a></li>
```

```
            <li><a href="http://delicious.com"><span>Link #4</span></a></li>
            <li><a href="http://wrox.com"><span>Link #6</span></a></li>
        </ul>
    </li>
    <li>
        <h2>.org links</h2>
        <ul id="dotorg">
            <li><a href="http://dojoproject.org"><span>Link #2</span></a></li>
            <li><a href="http://mozilla.org"><span>Link #5</span></a></li>
            <li><a href="http://w3.org"><span>Link #7</span></a></li>
        </ul>
    </li>
</ul>
```

还有一点值得注意，.place()和.adopt()均接受可选的第二个参数，即与.addContent()方法相同的位置实参。这包括指定的值 after、before、first 和 last 以及数字索引位置。例如：

```
dojo.query('ul#ex5 > li:first-child')
    .place('ul#ex5 > li:last-child', 'after');
```

这段代码将前面标记中的.com 和.org 链接列表交换。对于.adopt()方法，尝试下面这个示例：

```
var node = dojo.query('ul#ex5 > li:last-child').orphan();
dojo.query('ul#ex5').adopt(node, 'first');
```

上面的代码与前一个示例相反，将这两个列表的最后一个子节点移到第一个子节点的位置。

24.2.3 使用 NodeList 的其他方法

NodeList 还支持许多其他在本章中没有深入研究的数组方法，包括如下几个方法：

- .indexOf() 查找给定节点的第一个数字索引。
- .lastIndexOf() 查找给定节点的最后一个数字索引。
- .every() 将一个回调函数应用于列表中的每个节点，并返回该函数是否在每个节点上都返回 true。
- .some() 将一个回调函数应用于列表中的每个节点，并返回该函数是否在某些节点上返回 true。
- .coords() 返回每个节点在页面上的坐标。

与往常一样，要查找关于这些方法以及更多尚未在此讨论的方法的资料，最好的查找位置是官方 Dojo API 文档：

```
http://dojotoolkit.org/docs/api
```

24.3 本章小结

本章浏览了 Dojo 提供的 DOM 操作方法，包括使用 dojo.query()轻松查找元素，然后优化列表并利用 dojo.NodeList 提供的可链式调用的方法来操作它们，以修改和重新构造页面内容。

在下一章中，我们将学习 Dojo 用来绑定页面事件处理程序以响应用户交互的各种工具。

第25章

处 理 事 件

到目前为止，我们已经学习了如何使用dojo.query()查找文档对象模型中的元素，利用NodeList的方法优化和过滤节点列表，以及对DOM内容和结构进行批量修改。但是，没有来自用户的输入，这些操作方式在很大程度上一直是自发的和即时的。

虽然这些操作方式都非常有用，但它们并没有形成构建响应灵敏的用户界面的全景。在Dojo的底层，Dojo 事件系统提供了灵活的可跨浏览器的方式来连接用户输入和操作(它们触发来自按钮和链接单击、其他鼠标操作以及键盘活动的DOM事件)。

但是 Dojo 在这个基础之上构建并提供了进一步的抽象和灵活性，从而不仅能够将处理程序连接到DOM事件，而且能够连接到其他处理程序和对象方法。这样就能够构建复杂的、松散耦合的响应程序来响应应用程序中的输入和事件。Dojo还提供了涉及发布者和订阅者的广播消息系统以进一步松散耦合事件。

此外，为了辅助构造用户界面中所需的事件处理程序模式，Dojo 还提供了额外的 NodeList方法，让事件处理程序连接也能够实现在第24章中讨论的dojo.query()链式调用。除此之外，Dojo还提供了一个“行为”工具，可用于以类似CSS样式表的声明方式来映射事件和处理程序。

本章内容简介：

- 响应页面加载和卸载事件
- 连接到DOM事件
- 连接到方法
- 发布与订阅

25.1 响应页面加载和卸载事件

响应页面成功加载和卸载事件是在建立和拆除用户界面过程中最方便的两个基本事件挂钩。Dojo提供了两个方法来注册这些事件的处理程序，分别名为dojo.addOnLoad()和dojo.addOnUnload()，其用法如下所示。

```
dojo.addOnLoad(function() {
```

```
    console.log("** Inline onload handler fired!");
});

dojo.addOnUnload(function() {
    alert("** Inline onunload handler fired!");
});
```

如果尚未安装 Firebug(http://getfirebug.com，对于所有其他浏览器，确保在 djConfig 中将 isDebug 设为 true)，那么最好现在就完成这项工作。这将确保我们能够看到 console.log()语句的输出结果，在本章中，我们将使用这些语句来演示 Dojo 事件处理系统的某些不是那么显而易见的方面。

当加载页面和离开页面时，均会相应地调用前面代码中的每个内联匿名函数。在页面加载事件处理程序中使用 console.log()将会导致在 Firebug 日志(或 Dojo 自带的 Firebug Lite 版本所提供的页面调试日志)中出现一条消息。

页面卸载事件处理程序选择使用 alert()，这是因为它可以临时中断前往下一个页面的转换操作(由于浏览器加载下一个页面，因此在我们能够看到 console.log()语句的输出结果之前，它们已经消失)。一旦理解这里的工作原理，可能就会希望注释这个 alert()调用，主要是因为在调试该代码的过程中，它会让每次页面重新载入变得有些乏味。

注册页面加载和卸载的事件处理程序的另一种方式是使用对象方法，就像下面这样：

```
dojo.declare('decafbad.eventsanim.events.pageHandlers', null, {
    handleOnLoad: function() {
        // this == handlers
        console.log("** Object onload handler fired!");
    },
    handleOnUnload: function() {
        // this == handlers
        //alert("** Object onunload handler fired!");
        console.log("** Object onunload handler fired!");
    }
});
var handlers = new decafbad.eventsanim.events.pageHandlers();

dojo.addOnLoad(handlers, 'handleOnLoad');
dojo.addOnUnload(handlers, 'handleOnUnload');
```

注意，这些函数中 this 的上下文自动设为指向 handlers 的引用，从而让它们仍然处于最初的对象上下文中。

window.onload 问题

关于 dojo.addOnLoad()非常有趣的方面是，它在成功加载 DOM、Dojo 以及它的一组初始 dojo.require()语句之后(但在载入页面中所包含的所有 CSS 和图像之前)引发注册的处理程序。

这一点非常重要，因为页面上每个引用的素材到达之后均会引发简单的 window.onload 处理程序。

这个差别非常有用，因为我们可以利用它让页面初始化更早完成。在其他视觉样式生效之前可以构造窗口部件、绑定事件、应用页面修改，从而可以优先考虑所有令人困惑的一闪而过的内

容或其他过渡伪像(在缓慢载入需要长时间加载的图像的过程中出现)。

另一件值得强调的事情是，dojo.addOnLoad()在 Dojo 加载完毕之后引发。

这一点非常重要，因为 Dojo 可能仍然在完成一些加载任务，即使是在原生的 window.onload 事件引发之后。Dojo 可能仍然在继续解析剩余的最后几条 dojo.require()语句，或者在做其他事情。出于这个原因，在 Dojo 上下文中，dojo.addOnLoad()可以真正替代 window.onload。

25.2 连接到 DOM 事件

在概念上，Dojo 利用连接和订阅机制解决了响应用户界面事件的问题。这个差别非常重要，因为我们很快就会看到，连接和订阅机制均可以应用于 DOM 事件领域之外的事件。

为了进行演示，我们首先查看下面的标记，它提供了一些 CSS 和 HTML 代码，以供本章后面要演示的事件处理程序使用：

```
<style type="text/css">
    #ex1 .clickme {
        padding: 0.5em; margin: 0.5em;
        display: block; background-color: #ddf;
        border: 2px dashed #fff;
        color: #000;
    }
    #ex1 .clickcolor {
        background-color: #dfd;
        border: 2px dashed #000;
    }
</style>

<div id="ex1">
    <a id="link1" class="clickme color0" href="#">Click me!</a>
    <a id="link2" class="clickme color0" href="#">Click me!</a>
    <a id="link3" class="clickme color0" href="#">Click me!</a>
    <a id="link4" class="clickme color0" href="#">Click me!</a>
    <a id="link5" class="clickme color0" href="#">Click me!</a>
</div>
```

图 25-1 给出在引入事件处理程序之前该标记在浏览器中的运行结果。

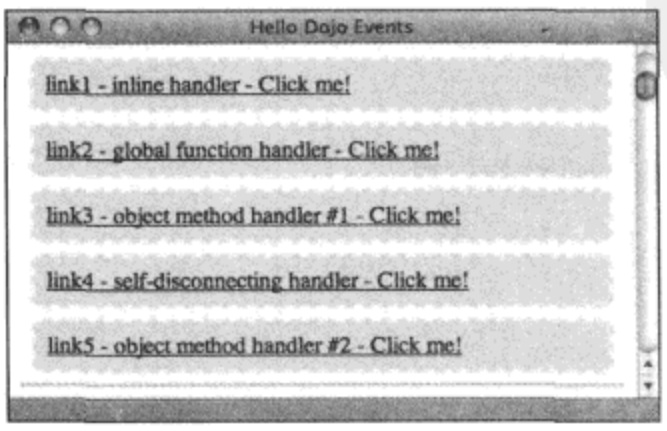


图 25-1

25.2.1 将内联处理程序连接到事件

为了将一些事件处理代码连接到第一个链接，请考虑下面的代码示例：

```
dojo.connect(dojo.byId('link1'), 'click', function(ev) {
    dojo.stopEvent(ev);
    dojo.toggleClass(ev.target, 'clickcolor');
    // this == ev.target
    console.log(this);
    console.log('Clicked link #1');
});
```

在上面的示例中，我们可以看到一个 dojo.connect()调用，这是 Dojo 的 DOM 事件处理机制的核心。它的第一个参数是指向 DOM 节点的引用，这里是通过内联调用 dojo.byId('link1')来获取该引用。接下来是 DOM 事件名称“click”，它捕获该链接上的所有单击事件。

最后，有一个内联匿名回调函数，它处理该链接上的所有单击事件。在这个函数中，调用了 dojo.stopEvent(ev)。这是一个快捷方式，它同时调用了 ev.preventDefault()和 ev.stopPropagation()，这些方法将分别阻止 DOM 事件的默认操作(也就是离开该页面)以及阻止事件传播到任何其他带有监听该事件的处理程序的 DOM 节点。

处理程序的剩余部分非常简单：调用 dojo.toggleClass()，这样每次单击操作都会导致交替地添加和删除 CSS 类“clickcolor”，并将几条消息发送到日志。其中一条消息将报告该函数的 this 变量的当前上下文(在这里就是事件的目标元素)。图 25-2 给出了连接事件处理程序并单击第一个链接之后的情形。

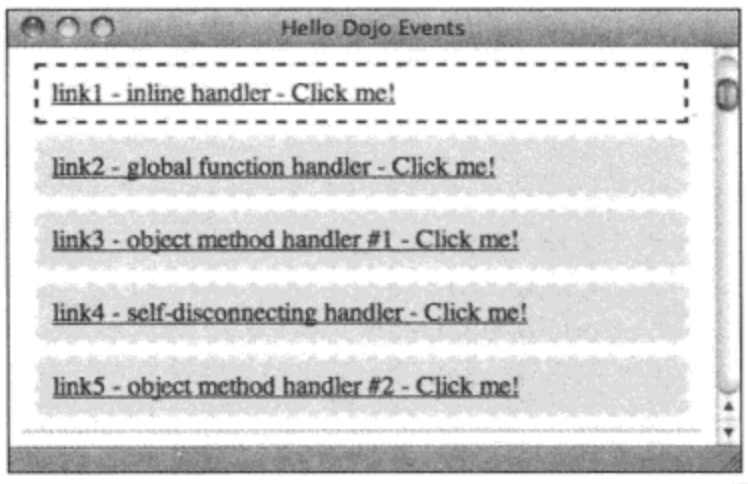


图 25-2

25.2.2 将全局函数连接到事件

接下来，对前面的代码进行细微的调整：

```
function clickFunc(ev) {
    dojo.stopEvent(ev);
    dojo.toggleClass(ev.target, 'clickcolor');
    // this == ev.target
    console.log(this);
    console.log(ev);
```

```
}

var link2 = dojo.byId('link2');
dojo.connect(link2, 'click', clickFunc);
```

这个事件处理程序的设置与前一个相同。唯一的区别在于，连接到事件的处理程序函数是一个全局声明的函数，而不是一个内联定义的函数。函数的功能以及 this 的上下文均相同。

与前面的形式相比，这种 dojo.connect()形式的优点在于，我们可以定义一个能够连接到多个节点的事件的可重用事件处理程序。图 25-3 给出了连接到各个链接的第二种处理程序的效果。

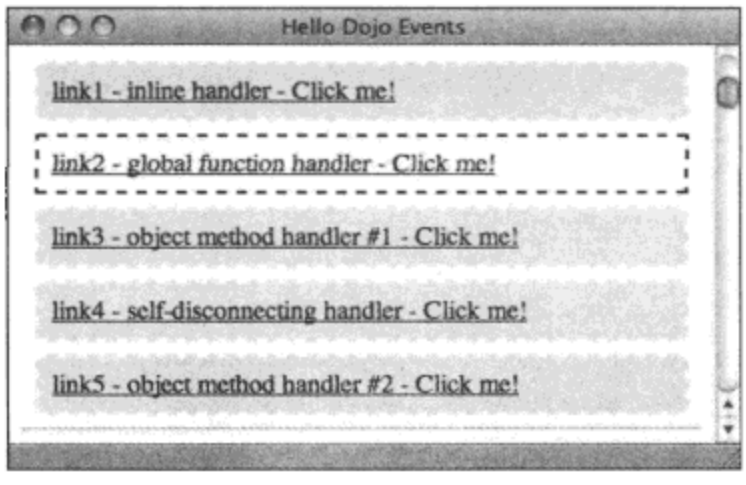


图　25-3

25.2.3　将对象方法连接到事件

继续探索可重用性，查看下面的代码：

```
dojo.declare('decafbad.eventsanim.events.connectHandlers', null, {
    color_toggle: 'clickcolor',

    clickMethod: function(ev) {
        dojo.stopEvent(ev);
        // this == handlers
        dojo.toggleClass(ev.target, this.color_toggle);
        console.log(this);
        console.log(ev);
    }
});
var handlers = new decafbad.eventsanim.events.connectHandlers();

var link3 = dojo.byId('link3');
dojo.connect(link3, 'click', handlers, 'clickMethod');
```

在这里，我们创建了一个包含处理程序的对象，它有一个属性 color_toggle 和一个方法 clickMethod。而且，在 dojo.connect()语句中，第三个参数变成了两个参数：一个指向该对象的引用，另一个为连接该事件的处理程序方法的名称。

这里的最大不同之处在于，在 clickMethod 处理程序中，this 的上下文已经发生改变以指向 handlers，该处理程序方法属于这个对象。因此，它能够访问该对象的 color_toggle 属性。第三种

处理程序实现了与前两种处理程序相同的效果，如图 25-4 所示。

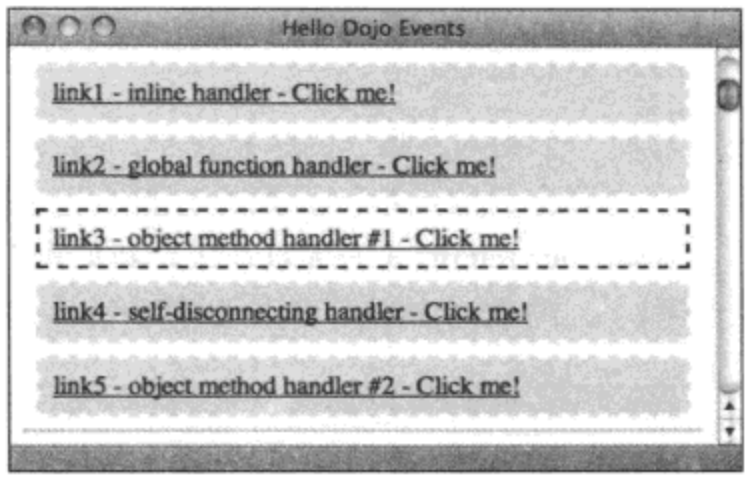


图 25-4

25.2.4 断开与事件的连接

最后，请注意 dojo.connect()有一个配对的 dojo.disconnect()方法，dojo.connect()方法返回的句柄可用在 dojo.disconnect()中以取消事件连接，就像下面这样：

```
var conn1 = dojo.connect(dojo.byId('link4'), 'click',
    function(ev) {
        dojo.stopEvent(ev);
        dojo.toggleClass(ev.target, 'clickcolor');
        console.log("This connection will now self-destruct");
        dojo.disconnect(conn1);
    });
```

这里的代码提供了自引用：以内联形式声明的 click 事件处理程序函数是一个访问 conn1 变量的闭包。conn1 变量的值是在连接事件时定义的。因此，在处理程序内部，它能够将自身从触发它的 DOM 事件中断开。

图 25-5 显示最后这种处理程序完成了与前面 3 种处理程序相同的工作。但是，在图 25-5 中并不能看到的是，一旦单击这个链接，处理程序就会断开连接，因此后续的单击将不会再切换样式。

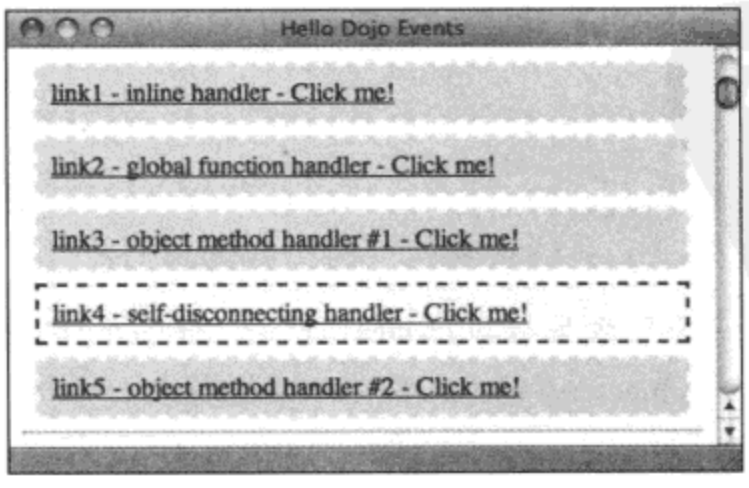


图 25-5

25.2.5　特殊的事件处理与事件对象

在使用 dojo.connect()时，Dojo 支持浏览器提供的所有 DOM 事件，这些事件在不同浏览器上略有不同，但有一个公共的规范化的集合，如表 25-1 所示。

表　25-1

事　　件	作　　用
click	当用户单击鼠标按键时产生该事件
keydown	当用户开始按下某个键时产生该事件
keypress	当用户按下和释放某个键时产生该事件
keyup	当用户释放某个键时产生该事件
mousedown	当用户开始按住某个鼠标按键时产生该事件
mouseenter	当用户将光标移入页面上某个元素的边界内时产生该事件
mouseleave	当用户将光标移出页面上某个元素的边界外时产生该事件
mousemove	当用户将光标移入页面上的某个元素时产生该事件
mouseup	当用户释放鼠标按键时产生该事件

显然还有其他 DOM 事件。但是表 25-1 给出的是一个非常有用的列表，本章剩余内容通篇都在使用该列表。在下面的网页中可以找到有关各种浏览器中可用事件的更多细节：

- http://en.wikipedia.org/wiki/DOM_Events
- http://developer.mozilla.org/en/docs/DOM:event
- http://msdn2.microsoft.com/en-us/library/ms533022.aspx
- www.w3.org/TR/DOM-Level-2-Events/

关于 Dojo 对这些事件的处理以及它们的名称，还有几点需要注意：

- 事件名称中的前缀“on”是可选的，将由 Dojo 根据需要添加或删除。
- 对有些事件(尤其是“keypress”)进行了特殊的跨浏览器处理以规范化行为。
- 尝试解释并清除诸如 Internet Explorer 这样浏览器中显著的内存泄漏问题。

还有一件重要的事情：事件处理程序中包含一个具有规范化属性和方法的跨浏览器事件对象，如表 25-2 所示。

表　25-2

规范化属性/方法	作　　用
target	产生该事件的元素
currentTarget	正在处理该事件的处理程序所属的元素
layerX	鼠标的 x 坐标，相对于 currentTarget
layerY	鼠标的 y 坐标，相对于 currentTarget
pageX	鼠标的 x 坐标，相对于视区
pageY	鼠标的 y 坐标，相对于视区
relatedTarget	针对 mouseout 和 mouseover 事件，分别对应于鼠标移入或移出的对象

(续表)

规范化属性/方法	作　用
charCode	检测到的按键的字符代码
preventDefault()	该方法阻止执行元素的默认事件处理方法
stopPropagation()	该方法阻止将事件传播到父节点

在传给处理程序的 Dojo 事件对象中还有其他属性可用，它们由来自于浏览器的原生 DOM Event 对象的不可修改属性组成。表 25-3 包含了一些能够跨越不同浏览器使用的额外属性。

表　25-3

非规范化属性	作　用
type	指出引发的事件的类型(例如"click")
ctrlKey	指出是否按 Ctrl 修改键
shiftKey	指出是否按 Shift 修改键
altKey	指出是否按 Alt 修改键
metaKey	指出是否按元修改键(例如 Mac OS X 计算机上的 Command 键)
button	指出检测到的鼠标按键
keyCode	指出键盘事件中非字符键的 unicode 值

25.3　连接到对象方法

既然我们已经了解 Dojo 支持 DOM 事件与处理程序回调之间的简单连接，因此下面介绍 dojo.connect()的另一种形式——将回调处理程序连接到一般的对象方法。查看下面的代码示例：

```
dojo.connect(handlers, 'clickMethod', function(ev) {
    // this == handlers
    console.log(this);
    console.log("Hey, me too!  I'm anonymous!");
});
```

在前面曾经定义了一个 handlers 对象，它有一个名为 clickMethod()的方法。这段代码将另一个处理程序附加到这个处理程序上。因此，每当调用 handlers.clickMethod()时，前面示例中的函数都会在此之后立即被调用。

这一点非常有用，因为可以将 clickMethod 连接到多个 DOM 方法，就像下面这样：

```
dojo.connect(dojo.byId('link3'), 'click', handlers, 'clickMethod');
dojo.connect(dojo.byId('link5'), 'click', handlers, 'clickMethod');
```

在这里，当单击链接 link3 或 link5 时，clickMethod 以及在前面代码中定义的内联回调都会引发。由于这个连接的目标是方法而不是 DOM 事件，因此所有引发该方法的 DOM 事件连接(甚至是直接的方法调用)也都会引发额外的附加处理程序。

按照连接 DOM 事件的形式，dojo.connect()可以使用全局函数以及像下面这样的对象方法。

```
function clickMeTooFunc(ev) {
    // this == handlers
    console.log(this);
    console.log("Hey, me too!  I'm a function!");
}
dojo.connect(handlers, 'clickMethod', clickMeTooFunc);

dojo.extend(decafbad.eventsanim.events.connectHandlers, {
    clickMeTooMethod: function(ev) {
        // this == handlers
        console.log(this);
        console.log("Hey, me too!  I'm a method!");
    }
});
dojo.connect(handlers, 'clickMethod', handlers, 'clickMeTooMethod');
```

在上面的代码中，一个全局函数和一个新定义的对象方法均连接到现有的处理程序方法中。如同可以将多个处理程序连接到同一个元素的 DOM 事件一样，我们也可以将多个处理程序连接到同一个方法，这些处理程序将按照连接顺序依次引发。

实际上，如果已经安装了 Firebug 或者启用了 djConfig.isDebug(确实应该使用其中的一种方式或者同时使用两种方式)，那么尝试前面示例中的代码。如果单击链接 link3 或 link5，那么这段代码就将产生类似于下面的一系列日志消息：

```
Object color_toggle=clickcolor
click clientX=170, clientY=254
Object color_toggle=clickcolor
Hey, me too! I'm anonymous!
Object color_toggle=clickcolor
Hey, me too! I'm a function!
Object color_toggle=clickcolor
Hey, me too! I'm a method!
```

注意，在所有指向 handlers.clickMethod()方法的连接中，this 的上下文均为 handlers 对象。

由于能够连接到处理程序以及 DOM 事件，因此这为我们开发自己的扩展和增强提供了极大的灵活性，只需要少量预先计划的协调即可。我们能够发现窗口部件和处理程序已经在页面中的多处进行注册，只要连接它们即可，这就将所需的连接数量降到最少，而且不需要待连接代码的预先批准。图 25-6 给出了最后一个连接，它演示了对象方法处理程序的第二次应用。

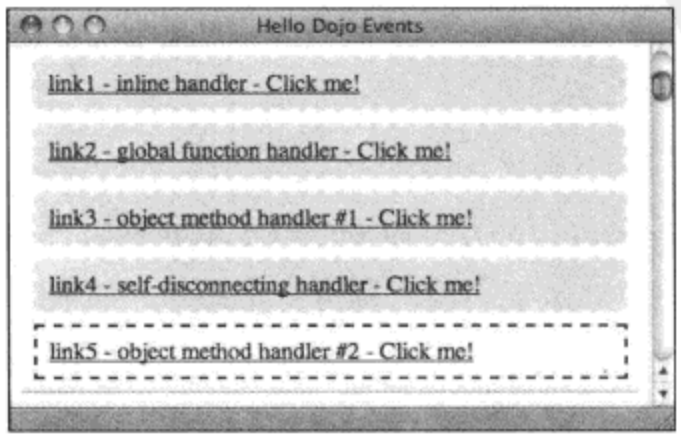


图 25-6

25.4 利用 NodeList 建立连接

一旦开始利用 dojo.byId()调用进行大量的 dojo.connect()调用，事情就会变得越来越缺乏优雅性，这是因为在利用原生的导航方法定位 DOM 节点时，它们确实是在倒退。使用 dojo.query()和 NodeList 方法有助于改善这种情况，而且事实证明，它们也有助于改善事件连接。

为接下来的演示做好准备，考虑下面的 HTML 标记：

```
<style type="text/css">
    .canvas {
        padding:.1em; margin: 1em;
        width: 100px; height: 100px;
        background-color: #ddf;
        border: 2px dashed #fff;
        float: left;
    }
    .entered, .selected {
        background-color: #dfd;
        border: 2px dashed #000;
    }
</style>

<form id="coord_canvas">
    <label for="coord_x">x:</label>
    <input name="coord_x" id="coord_x" type="text" size="5" />
    <label for="coord_y">y:</label>
    <input name="coord_y" id="coord_y" type="text" size="5" />
    <label for="status">status:</label>
    <input name="status" id="status" type="text" size="35" />

    <br />

    <div id="c1" class="canvas"> </div>
    <div id="c2" class="canvas"> </div>
    <div id="c3" class="canvas"> </div>
    <div id="c4" class="canvas"> </div>

    <br style="clear: both" />

</form>
```

图 25-7 给出了上面代码中的标记的运行效果，它是由几个表单字段和一组方块组成的简单布局。在这个演示中，我们为方块绑定了一组鼠标事件处理程序并使用所处理事件的属性来更新表单字段。

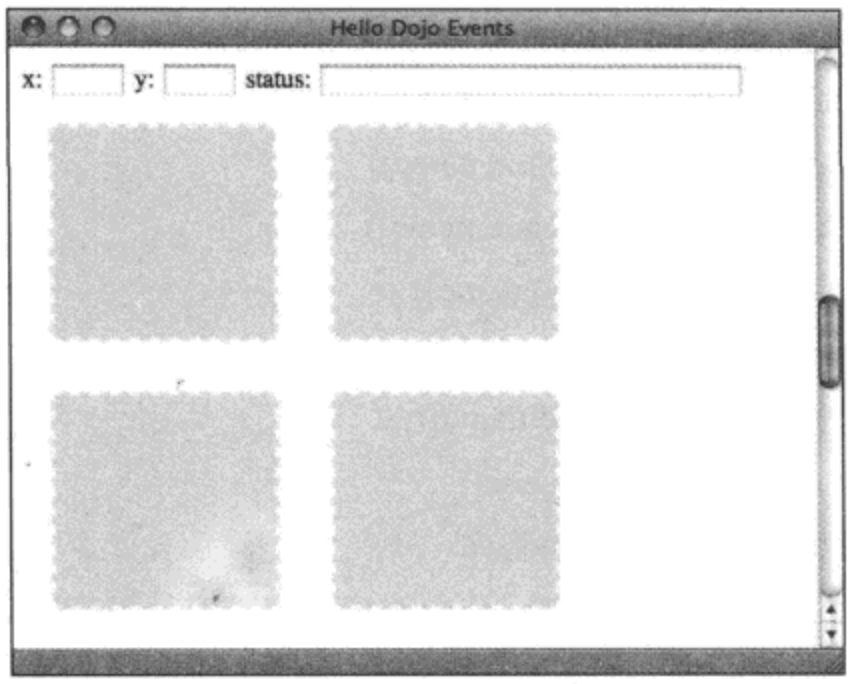


图 25-7

现在，查看下面的代码：

```
var canvases = dojo.query('#coord_canvas .canvas');

canvases
    .onmousemove(function(ev) {
        dojo.byId('coord_x').value = ev.pageX;
        dojo.byId('coord_y').value = ev.pageY;
    })
    .onclick(function(ev) {
        var el = ev.target;
        // this == el
        dojo.byId('status').value =
            'Mouse clicked '+el.id+' at '+ev.pageX+' x '+ev.pageY;
    });
```

这段代码使用 dojo.query()查找所有带有 CSS 类“canvas”的元素，它们包含了事件处理程序需要连接的方块。

NodeList 类提供了多个辅助函数来将 dojo.connect()调用应用于列表中的每个元素。具体来说，这里使用的两个辅助函数是.onmousemove()和.onclick()。第一个函数处理光标滑过其中一个方块时的事件，它利用该事件报告的坐标更新文本字段 coord_x 和 coord_y。第二个函数处理方块中的单击事件，它使用一条消息更新 status 字段。

这段简短的代码替换了一系列成对的 dojo.byId()和 dojo.connect()调用。使用 NodeList 辅助函数，可以清晰地声明以 dojo.query()返回的结果作为目标的事件处理程序连接，并让代码保持简洁易懂。

NodeList 提供的事件连接辅助函数包含如下方法：

- connect(event, func or object[, method name])
- onclick(func or object[, method name])

- onmousedown(func or object[, method name])
- ,onmouseenter(func or object[, method name])
- onmouseleave(func or object[, method name])
- onmousemove(func or object[, method name])
- onmouseup(func or object[, method name])
- onkeydown(func or object[, method name])
- onkeypress(func or object[, method name])
- onkeyup(func or object[, method name])

可以看到，这些辅助方法的模式相当简单。

第一个.connect()方法的工作方式与 dojo.connect()类似，但是第一个参数被移除，而由 Dojo 填充为列表中的每个节点。

剩下的方法还把第二个参数移除，这是因为事件名称可由辅助方法本身的名称推断出来。

所有这些方法的最后两个参数可以是处理程序回调函数或对象以及用作回调的方法的名称。

下面这段代码利用属于某个对象的处理程序来演示.onmouseenter()和.onmouseleave()的用法：

```
dojo.declare('decafbad.eventsanim.events.nodelistHandlers', null, {

    highlight_class: 'entered',

    handleOnMouseEnter: function(ev) {
        var el = ev.target;
        // this == handlers
        dojo.addClass(el, this.highlight_class);
        dojo.byId('status').value = 'Mouse entered '+el.id;
    },

    handleOnMouseLeave: function(ev) {
        var el = ev.target;
        // this == handlers
        dojo.removeClass(el, this.highlight_class);
        dojo.byId('status').value = 'Mouse left '+el.id;
    }

});
var handlers = new decafbad.eventsanim.events.nodelistHandlers();

canvases
    .onmouseenter(handlers, 'handleOnMouseEnter')
    .onmouseleave(handlers, 'handleOnMouseLeave');
```

这段代码将标记中的所有方块都使用事件绑定起来以跟踪光标进入和离开元素，适当地添加或移除高亮显示 CSS 类并更新 status 文本字段。在图 25-8 中可以看到这段代码执行时的快照。

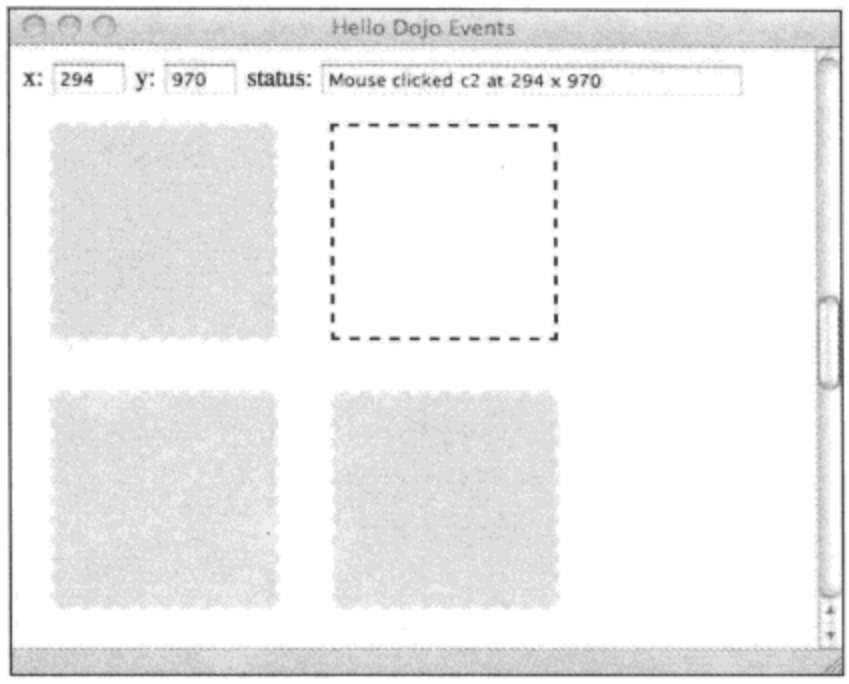


图 25-8

25.5 发布与订阅事件主题

在本章前面我们已经看到，将方法连接到方法提供了一定程度的事后扩展和松散协调性。此外，Dojo 提供了一种通用的事件发布/订阅工具，它以 dojo.publish()、dojo.subscribe()和 dojo.unsubscribe()方法的形式提供了一种更加强大的机制。

25.5.1 结合使用事件主题与 DOM 事件处理程序

首先查看演示，考虑如下标记：

```
<style type="text/css">
    .canvas {
        padding: 1em; margin: 1em;
        width: 100px; height: 100px;
        background-color: #bbf;
        float: left;
    }
    .entered, .selected {
        background-color: #bfb;
    }
</style>

<form id="broadcaster">

    <label for="source">select:</label>
    <select name="source" id="source">
        <option value="none">None</option>
        <option value="sel1">Square 1</option>
        <option value="sel2">Square 2</option>
        <option value="sel3">Square 3</option>
        <option value="sel4">Square 4</option>
```

```
</select>

<label for="source_status">status:</label>
<input name="source_status" id="source_status" type="text" size="35" />


<br />

<div id="sel1" class="canvas"> </div>
<div id="sel2" class="canvas"> </div>
<div id="sel3" class="canvas"> </div>
<div id="sel4" class="canvas"> </div>

<br style="clear: both" />
```

</form>

如图 25-9 所示，这段 HTML 标记包含一个下拉式选择框，其中的选项对应于页面上每个方块的 ID。

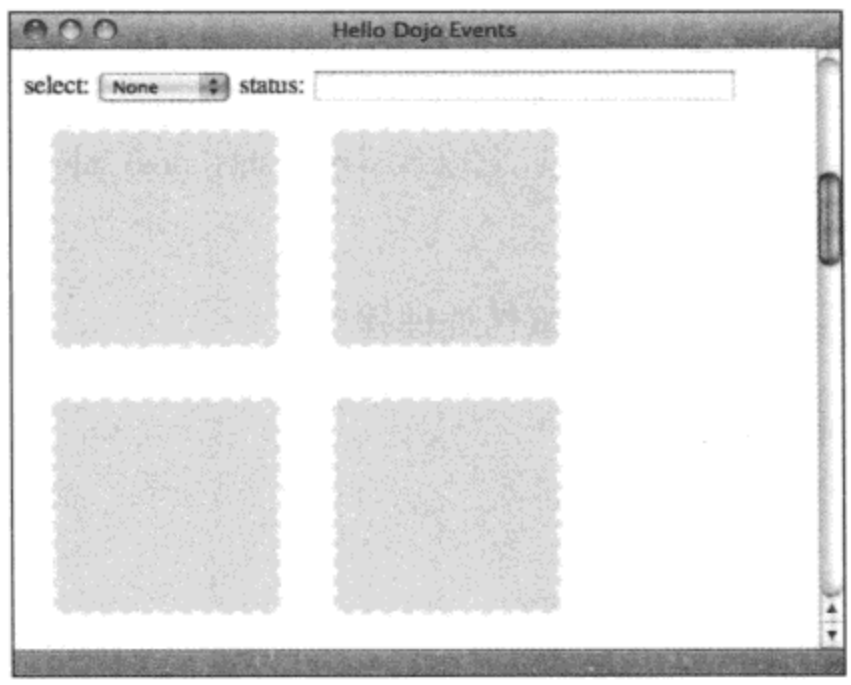


图 25-9

下面的代码将一个处理程序连接到选择框的 change 事件，该事件将广播一条关于任何选项值变化的消息：

```
var select_topic = '/decafbad/eventsanim/squareselected';

dojo.connect(dojo.byId('source'), 'change', function(ev) {
    var el  = ev.target;
    var idx = el.selectedIndex;
    var val = el.options[idx].value;
    console.log("Announcing selection of " + val);
    dojo.publish(select_topic, [ val ]);
});
```

首先要做的事情是建立一个共享的事件主题(event topic)，这是一个用来标识可以在页面上引

发的自定义事件的任意字符串。所有事件发布者与订阅者均围绕这个共享的字符串常量进行操作：发布者利用它发送消息，而订阅者利用它注册自己感兴趣的消息。

接下来，dojo.connect()将一个处理程序连接到下拉式选择框的 change 事件。这个处理程序弄清楚选中什么选项，然后使用 dojo.publish()以及建立的事件主题广播一条消息以指示选中的值。注意该消息的内容(作为 dojo.publish()的第二个参数)是一个数组。

现在要响应前面给出的代码中所发布的消息，查看下面的代码：

```
dojo.subscribe(select_topic, function(sel_id) {
    console.log("Selection changed to " + sel_id);
    dojo.byId('source_status').value =
        "Selected changed to " + sel_id;
});
```

这段代码使用一个内联函数回调来订阅下拉式选择框改变时发布的事件主题。注意，dojo.publish()发送的数组转换成订阅函数的参数。如果该数组包含两个或更多元素，那么订阅回调就可以预期有两个或更多参数。

在这个回调中，我们将接收到的选项发送到日志，并使用一条报告当前选项的消息更新文本字段 source_status 的值。

前面的代码非常简单，而下面的代码则有些复杂：

```
function respondToSelection(el_id, sel_id) {
    var el = dojo.byId(el_id);
    if (el_id == sel_id) {
        console.log("Selecting " + el_id);
        dojo.addClass(el, 'selected');
    } else {
        console.log("Deselecting " + el_id);
        dojo.removeClass(el, 'selected');
    }
}

dojo.forEach([ 'sel1', 'sel2', 'sel3', 'sel4' ], function(el_id) {
    dojo.subscribe(select_topic, function(sel_id) {
        respondToSelection(el_id, sel_id);
    });
});
```

在前面的代码中，我们定义了 respondToSelection()函数，它的参数为元素 ID 和选项 ID。该函数获取标识的元素，把它的 ID 与选项 ID 进行比较，然后适当地调整该元素的 CSS 类“selected”。

在此之后，dojo.forEach()调用遍历可选方块的已知 ID。对于每个方块，它创建一个新的订阅处理程序来响应选项框事件主题。因为这里的内联函数是一个闭包，所以它可访问 dojo.forEach()中的 el_id 参数。同样，由于回调处理程序的第一个参数预期是选项 ID，因此这些就是该处理程序中引发的 respondToSelection()调用所需的所有参数。

将这些操作结合起来，当选择项改变时发生多件事情并发布一条结果消息。图 25-10 给出了选择一个方块时的效果。

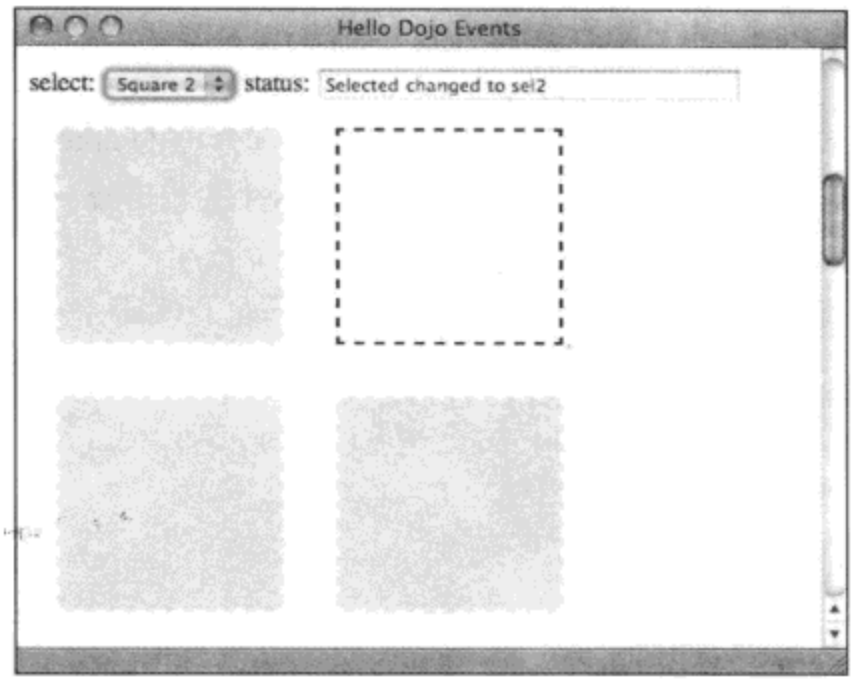


图　25-10

除了可见的用户界面变化之外，还应该在 Firebug 日志中看到类似下面的文本记录：

```
Announcing selection of sel2
Selection changed to sel2
Deselecting sel1
Selecting sel2
Deselecting sel3
Deselecting sel4
```

上面所有代码的目的就是创建一个松散耦合的系统，选项变化导致广播针对共享主题的消息。有一个状态表单字段用来响应该消息，并且每个可选择的方块均订阅到该共享主题，并独立地确定是高亮显示还是去除高亮显示。

所有这些方面都不需要预先进行确定：可以按照需要添加任意多个订阅者以响应选项框改变，选项框本身不需要进行修改就可以看到系统的其他部分得到更新。此外，我们可以根据需要规划任意多个事件主题来处理用户界面操作中的重要事件。

25.5.2　将对象方法用作订阅者

前面的示例演示如何使用内联函数处理程序构建一个带有事件广播功能的松散耦合系统。出于完整性考虑，值得注意的是，对象方法也可以运行(从而促进了重用性)：

```
console.log("*** Adding subscribers from object...");

var the_topic = '/decafbad/eventsanim/sample';
var subs = [];

dojo.declare('decafbad.eventsanim.events.pubsubHandlers', null, {
    foo: function(a, b) {
        console.log("foo: received event with args "+a+" and "+b);
    },
    bar: function(a, b) {
        console.log("bar: received event with args "+a+" and "+b);
```

```
    },
    baz: function(a, b) {
        console.log("baz: received event with args "+a+" and "+b);
    }
});
var handlers = new decafbad.eventsanim.events.pubsubHandlers();

dojo.forEach(['foo', 'bar', 'baz'], function(name) {
    var sub = dojo.subscribe(the_topic, handlers, name);
    subs.push(sub);
});

dojo.publish(the_topic, ['hello again', 'pubsub']);
```

这段代码声明了一个带有 foo()、bar()和 baz()方法的新类。然后，通过 dojo.forEach()来订阅每个方法以响应事件主题“/decafbad/eventsanim/sample”。接下来，针对该事件主题进行广播。最终的 Firebug 日志文本消息应该如下所示：

```
*** Adding subscribers from object...
foo: received event with args hello again and pubsub
bar: received event with args hello again and pubsub
baz: received event with args hello again and pubsub
```

25.5.3 取消订阅已发布的消息

此外，注意有一个 dojo.unsubscribe()调用，它接受 dojo.subscribe()的返回值作为句柄并取消订阅，如下所示：

```
console.log("*** Unsubscribing every other subscriber...");

for (var i=0; i<subs.length; i++) {
    if (i % 2) dojo.unsubscribe(subs[i]);
}

dojo.publish(the_topic, ['private', 'pubsub']);
```

在运行该代码之后，Firebug 文本消息将改变以反映消失的第二个订阅者：

```
*** Adding subscribers from object...
foo: received event with args hello again and pubsub
baz: received event with args hello again and pubsub
```

25.5.4 将对象方法转换成发布者

另一个有用的技巧就是利用发布的事件监听已有的对象方法调用。Dojo 利用实用工具 dojo.connectPublisher()提供了一种简单的实现方式。查看下面的演示代码：

```
console.log("*** Connecting publishers to object methods...");

dojo.declare('decafbad.eventsanim.events.methodPub', null, {
```

```
    quux: function(a, b) {
        console.log("quux says "+a+" and "+b);
    },
    xyzzy: function(a, b) {
        console.log("xyzzy mentions "+a+" and "+b);
    },
    foobar: function(a, b) {
        console.log("foobar announces "+a+" and "+b);
    }
});
var some_obj = new decafbad.eventsanim.events.methodPub();

var method_topic = '/decafbad/eventsanim/methodCalled';

dojo.forEach(['quux', 'xyzzy', 'foobar'], function(name) {
    var conn = dojo.connectPublisher(method_topic, some_obj, name);
});

var sub = dojo.subscribe(method_topic, function(a, b) {
    console.log("subscriber overheard "+a+" and "+b);
});

some_obj.quux('hello', 'world');
some_obj.xyzzy('again', 'hello');
some_obj.foobar('what', 'where');
```

上面的示例创建了一个带有一组简单方法的新类并创建这个新的 methodPub 类的一个实例。每个方法只产生一条总结方法名称以及它的两个参数的日志消息。

接下来，建立事件主题/decafbad/eventsanim/methodCalled。在 dojo.forEach()循环中，dojo.connectPublisher()调用使用该事件主题将每个对象方法转换成发布者。dojo.connectPublisher()的方法签名如下：

```
dojo.connectPublisher(event topic, object, method name)
```

注意，这个实用工具返回一个句柄，dojo.disconnect()可使用该句柄与发布者断开连接。

在启用发布者之后，dojo.subscribe()调用建立该事件主题的监听者。这个处理程序只报告自己成功监听到方法调用。

最后，调用该对象方法。将这些操作放在一起，就会看到文本消息中出现类似下面的日志消息：

```
*** Connecting publishers to object methods...
quux says hello and world
subscriber overheard hello and world
xyzzy mentions again and hello
subscriber overheard again and hello
foobar announces what and where
subscriber overheard what and where
```

25.6 使用 Dojo 行为

在 dojo.query()和 NodeList 方法中使用 CSS 选择器绑定事件处理程序，这种方式具有声明式编码的风格。而 Dojo 行为则更近一步，它提供了一种方式，使得向元素附加代码就像是在使用样式表声明样式一样。

考虑对前面给出的 HTML 标记略作修改之后的变体：

```
<form id="coord_canvas_2">

    <label for="coord_x_2">x:</label>
    <input name="coord_x_2" id="coord_x_2" type="text" size="5" />
    <label for="coord_y_2">y:</label>
    <input name="coord_y_2" id="coord_y_2" type="text" size="5" />
    <label for="status_2">status:</label>
    <input name="status_2" id="status_2" type="text" size="35" />

    <br />

    <div id="c1_2" class="canvas"> </div>
    <div id="c2_2" class="canvas"> </div>
    <div id="c3_2" class="canvas"> </div>
    <div id="c4_2" class="canvas"> </div>
    <br style="clear: both" />

</form>
```

在上面的标记中，ID 与之前的示例不同，这样它们就能够同时放在同一个页面上，以便进行并排比较。相应地，有几个表单文本字段用于报告发生的事件，此外还有几个 元素，它们的样式将设置为彩色方块，而且事件处理程序将会连接到这些元素上。

25.6.1 利用行为查找节点并建立连接

比 NodeList 链式方法调用更优雅的方式是，dojo.behavior 模块提供了一种方式来清晰地声明并映射 Web 应用程序所需的所有事件处理程序和事件主题。查看下面的代码，它演示了该模块提供的一些功能：

```
dojo.require('dojo.behavior');

dojo.behavior.add({

    '#coord_canvas_2 label': function(el) {
        console.log("Found label "+el.getAttribute('for'));
    },

    '#coord_canvas_2 .canvas': {
```

```
        found: function(el) {
            console.log("Found canvas "+el.id);
        },

        onmousemove: function(ev) {
            dojo.byId('coord_x_2').value = ev.pageX;
            dojo.byId('coord_y_2').value = ev.pageY;
        },

        onclick: function(ev) {
            var el = ev.target;
            // this == el
            dojo.byId('status_2').value =
                'Mouse clicked '+el.id+' at '+ev.pageX+' x '+ev.pageY;
        }

    }

});

dojo.behavior.apply();
```

首先，由于 dojo.behavior 并不属于基本 dojo 模块，因此需要使用 dojo.require()调用加载它。然后调用 dojo.behavior.add()，这里只附带一个对象字面值作为唯一的实参。这个对象字面值设计了一组 CSS 选择器和将要应用于匹配这些选择器的元素的规则。最后一个 dojo.behavior.apply()调用启动 Dojo 行为引擎，它将应用在 add()调用中定义的所有规则。

传入 dojo.behavior.add()的数据结构是一组规则，与 CSS 样式表类似。顶层属性通过使用 CSS 选择器命名，而每个选择器的值可以是一个函数，也可以是嵌套的数据结构。

如果该值是一个函数(内联定义或来自某个变量)，那么 Dojo 行为引擎会针对每个匹配 CSS 选择器的元素来调用该函数一次。因此，在前面的示例中调用 apply()方法时，我们将会在 Firebug 日志文本消息中看到类似下面的消息：

```
Found label coord_x_2
Found label coord_y_2
Found label status_2
```

前面代码中的第二条规则构造一个嵌套的数据结构作为 CSS 选择器规则的值。该结构中每个属性的值反映了 DOM 事件，在之前给出的示例中，我们使用了 onmousemove 和 onclick 事件。每个这样的事件都会导致一次 dojo.connect()调用，将指定函数作为处理程序连接到 CSS 选择器发现的每个元素的指定事件上。注意，在定义行为时，前缀“on”并不是可选项，因为它处在 dojo.connect()调用中。

这里的一个例外情况就是属性“found”，这个属性与前面的规则的工作方式类似，对于每个找到的元素都会调用指定的函数。这段代码在 Firebug 中产生如下的日志消息。

```
Found canvas c1_2
Found canvas c2_2
Found canvas c3_2
Found canvas c4_2
```

应用这个行为集所产生的最终结果就是之前曾经提到的日志消息，同时把事件处理程序连接到页面上的 方块的 click 和 mousemove 事件。

25.6.2 利用行为连接对象方法

更为复杂但却更加有用的功能是将处理程序从一个对象连接到 DOM 事件，就像下面这样：

```
dojo.declare('decafbad.eventsanim.events.behaviorHandlers', null, {

    highlight_class: 'entered',

    handleOnMouseEnter: function(ev) {
        var el = ev.target;
        // this == handlers
        dojo.addClass(el, this.highlight_class);
        dojo.byId('status_2').value =
            'Mouse entered '+el.id;
    },

    handleOnMouseLeave: function(ev) {
        var el = ev.target;
        // this == handlers
        dojo.removeClass(el, this.highlight_class);
        dojo.byId('status_2').value =
            'Mouse left '+el.id;
    }

});
var handlers = new decafbad.eventsanim.events.behaviorHandlers();

dojo.behavior.add({
    '#coord_canvas_2 .canvas': {
        onmouseenter:
            dojo.hitch(handlers, 'handleOnMouseEnter'),
        onmouseleave:
            dojo.hitch(handlers, 'handleOnMouseLeave')
    }
});

dojo.behavior.apply();
```

这个代码示例声明了一个新类 behaviorHandlers。该类的实例为 mouseenter 和 mouseleave 事件提供了两个处理程序方法。dojo.behavior.add()调用适当地将这两个处理程序连接到它们各自在方块画布 上的事件。

Dojo 文档提到，应该可以使用下面的构造声明行为规则，从而将对象连接为处理程序：

```
onmouseenter: { targetObj: handlers, targetFunc: 'handleOnMouseEnter' }
```

但在编写本书时，这种做法似乎并不能工作。因此，这里的代码调用的是 dojo.hitch()，这个 Dojo 实用工具返回一个包装器函数，该函数调用给定对象上的适当方法。这是额外的一步，但工作方式基本相同。在阅读到这里时，您或许希望尝试这种更具声明性的方式而不是调用 dojo.hitch()，从而查看该方式是否可行。

25.6.3 利用行为发布事件主题

对于 dojo.behavior 的最后一种用法，请查看下面的代码：

```
dojo.behavior.add({
 '#coord_canvas_2 div': '/decafbad/eventsanim/div',
 '#coord_canvas_2 input': {
    'found': '/decafbad/eventsanim/foundInput',
    'onchange': '/decafbad/eventsanim/inputChanged',
    'onkeypress': '/decafbad/eventsanim/inputKeypress'
   }
});

dojo.subscribe('/decafbad/eventsanim/div', function(el) {
   console.log("Found a div "+el.id);
});

dojo.subscribe('/decafbad/eventsanim/div', function(el) {
   console.log("Found a div (again) "+el.id);
});

dojo.behavior.apply();

dojo.subscribe('/decafbad/eventsanim/foundInput', function(el) {
   console.log("Found an input "+el.id);
});

dojo.subscribe('/decafbad/eventsanim/inputChanged', function(ev) {
   console.log("Detected change on "+ev.target.id);
});

dojo.subscribe('/decafbad/eventsanim/inputKeypress', function(ev) {
   console.log("Detected keypress on "+ev.target.id);
});
```

正如这段代码所演示的那样，在行为规则中，我们可以使用事件主题字符串而不是函数。其中部分规则会根据找到的元素发布事件，而其他规则将自动把处理程序连接到 DOM 事件(向给定的主题发布事件)。

但是请注意，“/decafbad/eventsanim/foundInput”主题的订阅出现在 dojo.behavior.apply()调用之后。因此，它绝不会接收到行为规则针对该主题发布的任何消息。当然，我们可以在任何时候

订阅主题(即使在调用apply()之后)，但在调用apply()之后，所有与选择器找到的元素相关的事件(如found事件)都已经发布。

对于应用程序，这些规则以及之后发布的事件将产生类似下面的Firebug日志文本消息：

```
Found a div c1_2
Found a div (again) c1_2
Found a div c2_2
Found a div (again) c2_2
Found a div c3_2
Found a div (again) c3_2
Found a div c4_2
Found a div (again) c4_2
```

对于最后几次订阅，操作页面上的表单输入字段将触发类似下面的日志消息：

```
Detected keypress on coord_x_2
Detected keypress on coord_x_2
Detected change on coord_x_2
Detected keypress on coord_y_2
Detected keypress on coord_y_2
Detected keypress on coord_y_2
Detected change on coord_y_2
Detected keypress on status_2
Detected change on status_2
```

25.7 本章小结

在本章中，我们介绍了Dojo提供用来把处理程序连接到DOM事件的工具，包括简单的dojo.connect()调用，使用dojo.NodeList方法批量地连接事件，以及采用类似于CSS样式表的方式使用dojo.behavior声明事件和处理程序的映射。此外，我们还研究了发布和订阅抽象事件主题的系统，它提供了另一种把处理程序与DOM事件解除关联的灵活方式。

在下一章中，我们将介绍Dojo提供的动画系统。我们将看到Dojo中各种动画基本要素的演示，以及链接和合并动画并把缓动效果应用于动画流的各种方式。我们将介绍扩展的NodeList方法，显示如何使用dojo.query()构造特定节点集合的动画。

PDG

第26章

编排动画

接收用户输入并据此执行操作是构建响应灵敏的用户界面的关键步骤，但是不仅限于此。同样重要的是能够提供反馈并高亮显示结果和新信息，以吸引用户的注意力。根据“展示而不只是表述”的原则，动画是最容易也是最有效的方式之一：可以在状态消息中使用淡入淡出或擦除效果，可以通过将元素滑入新位置来描述操作结果，甚至可以编排一个完整的运动和转换链来讲述一个故事。

Dojo 工具集提供了一个动画系统，它兼具使用方面的简洁性和调整方面的灵活性。Dojo 核心中有完整的系统和一些动画基本要素，此外还有几个可加载模块用来按需提供额外的功能。

动画引擎本身针对性能做了调优，仅使用一个 JavaScript 定时器来管理多个并发的动画，并让转换和运动保持平滑。该引擎管理的动画基本要素全部基于一个共用类，它的实例描述了 CSS 样式属性(包括位置、颜色和大小)之间的转换。

这些动画对象还提供了一组方法和综合事件来动态修改动画并对动画生命周期中的不同时间点做出响应。最后，Dojo 提供了几种方式将动画端到端地链接在一起以及以并行方式组合动画。

本章内容简介：

- 对 CSS 样式属性制作动画
- 使用淡入淡出和擦除转换
- 使用滑动动画

26.1 对 CSS 样式属性制作动画

全部 Dojo 动画的基础均建立在 CSS 样式属性之间的转换之上。由于这些属性包括了位置、不透明度、颜色、大小以及数个其他呈现方面，因此这是一种非常强大的运动和转换表达方式。

为了进行第一个演示，考虑下面的 HTML 标记：

```
<style type="text/css">
    #toAnim, .toAnim {
        position: relative;
        width: 100px;
```

```
            text-align: center;
            padding: lem;
            border: 2px dotted #000;
            background-color: #ddf;
        }
    </style>

    <div id="anim_property">

        <form>
            <button id="doAnim1">Anim Out</button>
            <button id="doAnim2">Anim Back</button>
        </form>

        <div id="toAnim">Watch me!</div>

        <br style="clear: both" />

    </div>
```

在这个示例中，我们编写了一段 CSS 代码来为即将制作动画的 元素提供样式。这段标记的其余部分还包括两个按钮作为用户界面： 元素本身以及用于完成这个小型布局的换行符。图 26-1 给出了这段标记的显示结果。

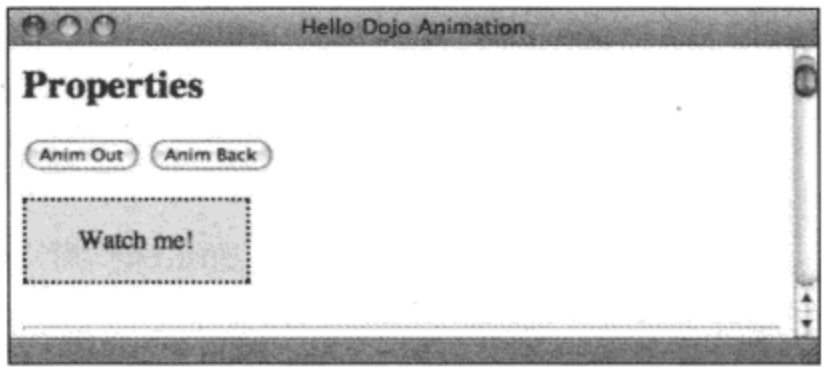


图 26-1

现在，查看第一段动画代码：

```
dojo.connect(dojo.byId('doAnim1'), 'click', function(ev) {
    dojo.stopEvent(ev);
    var anim = dojo.animateProperty({
        node: dojo.byId('toAnim'),
        duration: 1000,
        properties: {
            left:  { start: '0', end: '300', unit: 'px' },
            width: { start: '100', end: '300', unit: 'px' },
            backgroundColor: { start: '#dfd', end: '#ddf' }
        }
    });
    anim.play();
});
```

```
dojo.connect(dojo.byId('doAnim2'), 'click', function(ev) {
    dojo.stopEvent(ev);
    var anim = dojo.animateProperty({
        node: dojo.byId('toAnim'),
        duration: 1000,
        properties: {
            left:  { start: '300', end: '0', unit: 'px' },
            width: { start: '300', end: '100', unit: 'px' },
            backgroundColor: { start: '#ddf', end: '#dfd' }
        }
    }).play(500); // delay 500ms before starting animation
});
```

利用之前介绍的有关事件处理的知识，在这里两次调用 dojo.connect()来绑定页面上的按钮事件，从而在单击时，每个按钮都会启动一段涉及布局中 元素的动画。

在每次调用中，均使用 dojo.animateProperty()创建一个动画对象，并将其分配给变量 anim。本章稍后将深入研究这些对象的具体方法和属性，但现在请注意这些动画在创建时并没有启动，而是需要调用.play()方法来启动该过程。

此外还要注意的是，.play()方法返回动画对象本身，这样就能够进行链式调用。另外，该方法接受一个单位为毫秒的参数，它定义了在启动实际动画之前应该等待的时间。

进一步深入研究代码，请注意每个 dojo.animateProperty()调用都接受一个对象字面值作为其参数。该参数将作为一组属性用来实例化这个新动画对象，表 26-1 给出了这些属性。

表　26-1

动 画 属 性	作　　用
node	将对其制作动画的节点
duration	一段时间(单位为毫秒)，动画应该在此时间内结束
delay	动画启动之前等待的时间(单位为毫秒)
repeat	重复播放该动画的次数
easing	用来将线性过程转换成动态运动的函数
properties	声明 CSS 属性及其转换

在前面的代码给出的每个动画中，dojo.byId('toAnim')负责提供节点，而 duration 设置为 1 秒。由于没有定义 delay 和 repeat，因此这些属性均使用默认的 0 值。

此外，也没有定义 easing，因此会使用默认的值(本章稍后将详细讨论缓动，因此现在不要担心该属性)。这里并没有提及额外的属性(包括事件处理程序的分配，在本章快结束时将讲解它们)，而其他属性则可以在 Dojo API 文档中找到。

最后，有一个嵌套数据结构用来定义 properties，这是一个与 start、end 和 unit 属性相关联的 CSS 属性名称列表。

注意，在使用 JavaScript 表达 CSS 属性名称时遵循一个通用的产生式规则，即把连字符分隔转换成 CamelCase(驼峰式大小写)，因此属性“background-color”转换成 JS 名称“backgroundColor”。

因此，随着 duration 指定时间的推进，Dojo 将弄清楚如何将每个 CSS 属性以给定的 unit 类型由 start 值转换成 end 值。因此，在前面的代码中，每个动画将沿着水平方向移动节点，改变它的宽度，并一次性混合其背景颜色。Dojo 的代码在内部处理任何类型 CSS 属性的任何一对值之间的转换过程，因此我们可以随意地尝试各种组合，以便体验 Dojo 动画的功能。

26.2 使用淡入淡出转换

虽然使用 dojo.animateProperty()可以实现任何想要的转换，但有一些转换特别有用或常见。因此，Dojo 提供了一些辅助函数来简化使用它们的过程。Dojo 核心提供的一对辅助函数是 dojo.fadeOut()和 dojo.fadeIn()，两者均是对元素的不透明度制作动画以提供淡入淡出效果。

与前一个示例中的标记一样，下面的这些 HTML 标记给出了一个基本的界面：

```
<style type="text/css">
    #toFade {
        border: 2px dotted #000;
        background-color: #ddf;
    }
</style>

<div id="anim_fade">
    <form>
        <button id="fadeItOut">Fade Out</button>
        <button id="fadeItIn">Fade In</button>
    </form>
    <div id="toFade">
        <h3>Watch me fade in and out!</h3>
    </div>
</div>
```

在这个示例中，总共有两个启动动画的按钮，还有一个对其制作动画的 元素。图 26-2 给出了该示例在浏览器中的呈现结果。

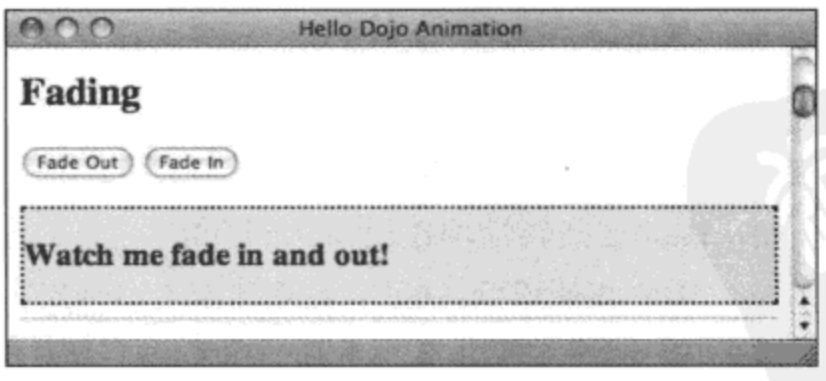


图 26-2

下面的代码将按钮事件与简化的淡入淡出动画辅助函数连接起来：

```
dojo.connect(dojo.byId('fadeItOut'), 'click', function(ev) {
    dojo.stopEvent(ev);
    var anim = dojo.fadeOut({
```

```
        node: dojo.byId('toFade'), duration: 1000
    }).play();
});

dojo.connect(dojo.byId('fadeItIn'), 'click', function(ev) {
    dojo.stopEvent(ev);
    var anim = dojo.fadeIn({
        node: dojo.byId('toFade'), duration: 1000
    }).play();
});
```

该代码的作用是，单击第一个按钮将导致该元素淡入背景，但仍然占据原有的空间。单击第二个按钮则会让该元素返回到完全不透明的状态。

请注意，尽管该代码要比使用 dojo.animateProperty()简单得多，但是它真正实现的功能却更多：为操作不透明度做准备，这些辅助函数调整了其他几个样式，以确保这个效果能够跨浏览器运行。

除了 properties 值(由这些辅助函数自行管理)以外，这些辅助函数的参数与使用 dojo.animateProperty()实用工具时的参数完全一样，包括 node、duration、delay、repeat、easing。

26.3 使用擦除转换

前面介绍的淡出的元素仍然占据原有的空间，而我们下面将要讨论的一对动画辅助函数则不同，它们分别名为 dojo.fx.wipeOut()和 dojo.fx.wipeIn()，均来自 dojo.fx 模块。

这两个函数分别减少和增加元素的高度，看上去就像是把该元素插入到页面中，或者将其卷起并消除。与淡出相反，这个效果将元素从视图以及布局流中一并移除。这对于短暂显示的警告面板或其他在短暂出现之后就消失的瞬时消息来说非常方便。请注意，该转换效果并不会真正将元素从 DOM 中删除(如果希望这样做，就需要单独进行操作)。

下面的标记看上去应该与目前演示的其他动画相似：

```
<style type="text/css">
    #anim_fx_wipe .toAnim div {
        padding: 1em;
        border: 2px dashed #000;
        background-color: #dfd;
    }
</style>

<div id="anim_fx_wipe">

    <form>
        <button id="doWipeOut">Wipe Out</button>
        <button id="doWipeIn">Wipe In</button>
    </form>

    <div class="toAnim">
        <div>Watch me!</div>
```

```
    </div>

</div>
```

这个示例中的 HTML 标记包括两个驱动动画的按钮，还有一个作为这些动画作用对象的元素。图 26-3 给出了这段标记的呈现结果。

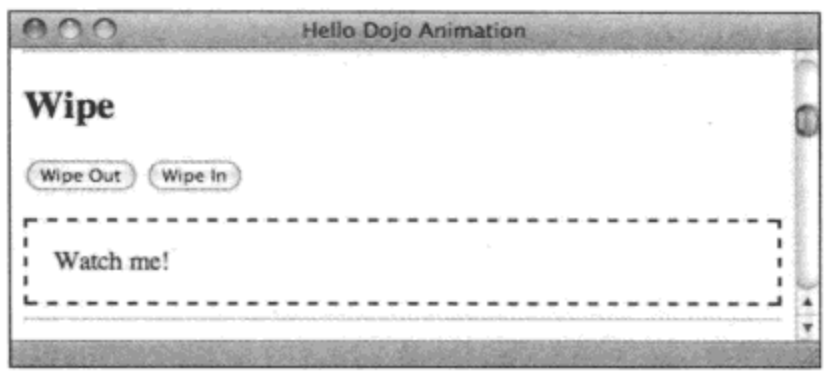


图 26-3

考虑下面这段演示 dojo.fx.wipeOut()和 dojo.fx.wipeIn()辅助函数的代码：

```
dojo.require('dojo.fx');

dojo.connect(dojo.byId('doWipeOut'), 'click', function(ev) {
    dojo.stopEvent(ev);
    var anim = dojo.fx.wipeOut({
        node: dojo.query('.toAnim', 'anim_fx_wipe')[0], duration: 250
    }).play();
});

dojo.connect(dojo.byId('doWipeIn'), 'click', function(ev) {
    dojo.stopEvent(ev);
    var anim = dojo.fx.wipeIn({
        node: dojo.query('.toAnim', 'anim_fx_wipe')[0], duration: 250
    }).play();
});
```

遵循目前建立的模式，这段代码将界面中两个按钮的单击事件分别连接到两个处理程序，这两个处理程序将分别实例化 wipe-in 和 wipe-out 动画。其效果就是，对其制作动画的元素将沿着垂直方向收缩直到从页面中的可见区域消失，或者沿着垂直方向扩展以将自己插入到布局中。

尽管它们的用法与 dojo.fadeOut()和 dojo.fadeIn()非常相似，但其主要差别在于，它们来自一个独立于 Dojo 代码的模块 dojo.fx，需要使用 dojo.require()语句加载这个模块。dojo.fx 模块提供了几个增强功能(包括本节讨论的辅助函数)，我们将在下面看到这些功能。

26.4 使用滑动动画移动元素

淡入淡出和擦除均属于同一类动画，但它们并没有给予元素任何运动。这就是 dojo.fx.slideTo()辅助函数发挥作用的地方。

考虑下面的标记，同时请注意样式中的 position: relative：

```
<style type="text/css">
    .toAnim {
        position: relative;
        width: 100px;
        text-align: center;
        padding: 1em;
        border: 2px dotted #000;
        background-color: #ddf;
    }
</style>
<div id="anim_fx_slide">
    <form>
        <button id="doSlideOut">Slide Out</button>
        <button id="doSlideBack">Slide Back</button>
    </form>
    <div class="toAnim">
        <div>Watch me!</div>
    </div>
    <br style="clear: both" />
</div>
```

与前面一样，这段标记给出了两个按钮来演示两个动画。带有 CSS 类“toAnim”的 元素等待应用运动效果。图 26-4 给出了这段标记的呈现效果。

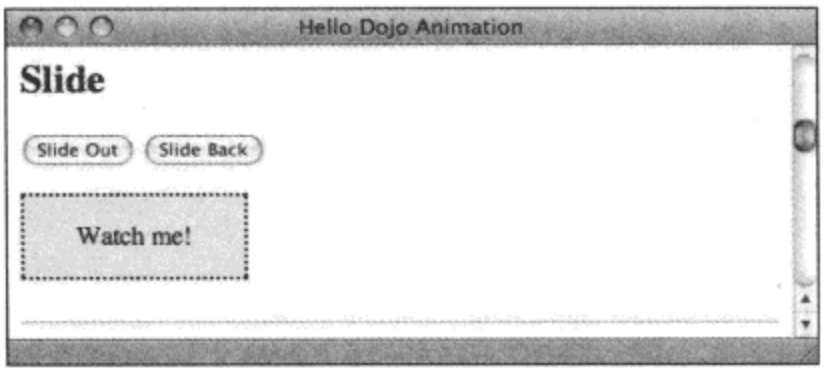


图　26-4

该运动是由如下代码实现的：

```
dojo.require('dojo.fx');

dojo.connect(dojo.byId('doSlideOut'), 'click', function(ev) {
    dojo.stopEvent(ev);
    var anim = dojo.fx.slideTo({
        node: dojo.query('.toAnim', 'anim_fx_slide')[0],
        duration: 1000,
        left: '200', top: '100', unit: 'px'
    }).play();
});

dojo.connect(dojo.byId('doSlideBack'), 'click', function(ev) {
    dojo.stopEvent(ev);
    var anim = dojo.fx.slideTo({
```

```
            node: dojo.query('.toAnim', 'anim_fx_slide')[0],
            duration: 1000,
            left: '0', top: '0', unit: 'px'
        }).play();
    });
```

第一个使用 dojo.fx.slideTo()的动画将该元素向左移动 200 像素，同时向下移动 100 像素。第二个动画将该元素移回原位。

如果回顾前面给出的 position: relative，那么需要重点注意的是 dojo.fx.slideTo()不仅可以移动页面上采用绝对定位的元素，它还可以移动采用相对定位的元素。如果元素没有采用这两种定位方式之一，那么 Dojo 将计算出该元素的位置，并将其 position 属性设为 absolute，然后再进行处理。

与擦除动画辅助函数一样，dojo.fx.slideTo()位于 dojo.fx 模块中，因此也需要加载。

实际上，如果之前已经使用 dojo.require()语句加载 dojo.fx 模块，就不需要再次加载它。但因为 dojo.require()和 dojo.provide()足够智能，能够记录已经加载了哪些模块，所以重复声明依赖项并不会带来任何副作用，实际上，有时候确实需要重复声明，特别是在彼此不相互依赖的不同模块中。

26.5 使用缓动控制运动

线性运动和转换并不能真正具有趣味。它们并不能非常好地吸引眼球，而且观看起来并不有趣。相反，最好能够在动画播放过程中改变事物。例如，与其均匀地从 A 点滑到 B 点，可以想像一个元素以极快的速度射出，然后逐渐慢下来驶入目的地。或者，可以想像一个元素回弹到某个位置，并在其路径上进行小范围的弹跳运动。只需要添加一点点趣味的内容就能够给用户界面的使用带来极大的乐趣。

最简单(同时也是最无趣)的缓动公式就是生成一条线性的、无变化的线路直至完成：

```
var linear_easing = function(n) { return n; }
```

我们之前提到，所有的动画均有一个名为 easing 的属性，如果没有指定该属性的值，就使用默认值。事实证明，这就是默认公式：

```
dojo._defaultEasing = function(/*Decimal?*/ n){
    // summary: The default easing function for dojo._Animation(s)
    return 0.5 + ((Math.sin((n + 1.5) * Math.PI))/2);
}
```

默认的缓动公式产生的运动看起来更加自然，带有惯性和动量：运动缓慢开始，逐渐提速，然后逐渐减速，直到终点停止。

为了帮助您彻底地理解该效果，可以认为 Dojo 通过跟踪总体完成百分比来让一个动画中的一切事物同步。Dojo 使用一个均匀地从 0.0 增长到 1.0 的小数来衡量在 duration 的值所指定时间中所占的百分比。Dojo 负责处理如何执行动画所指定的一个或多个 CSS 属性的转换。任何给定点中的每个转换的进度均由整体百分比的倍数计算而来。

实际上，这并不是事情的全部：easing 表示在交给每个转换之前在总体完成百分比之上进行

的某种变换。因此，虽然 Dojo 跟踪的完成百分比确实是从 0.0 均匀地增长到 1.0，但缓动函数利用这个过程中的值为我们带来了乐趣，并因此带来了动画过程中的视觉运动和转换效果。

理解缓动工作方式的最简单途径可能就是构建一个用来尝试它们的实验室。首先，考虑下面这段用作用户界面框架的标记：

```
<div id="anim_easing">

    <form>
        <select class="slideEase"></select>
        <button class="doSlideOut">Ease Out</button>
        <button class="doSlideBack">Ease Back</button>
    </form>

    <div class="toAnim">
        <div>Watch me!</div>
    </div>

    <br style="clear: both" />

</div>
```

在上面标记定义的表单中，第一个元素就是一个空白的<select>。在后面将要给出的代码中，我们将利用缓动函数来动态地填充这个元素。选中的缓动函数将用于两个动画中(这些动画将连接到界面中的按钮)。如果感兴趣的话，那么可以查看图 26-5 给出的这段标记的显示结果。

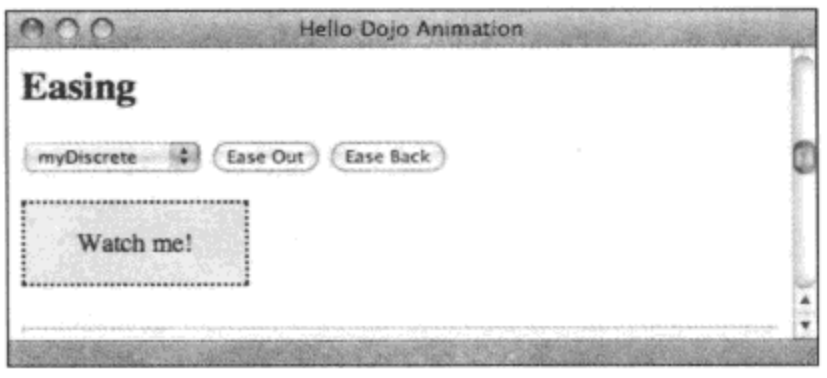


图　26-5

对于可用的缓动函数，下面的代码定义了几种这样的函数：

```
// Build a set of simple easings.
var my_easings = {
    myPlainLinear: function(n) {
        return n;
    },
    myDiscrete: function(n) {
        return Math.floor( n * 10 ) / 10;
    },
    myPower: function(n) {
        return (n==0) ? 0 : Math.pow(2, 10 * (n - 1));
    }
};
```

my_easings 中定义了 3 种缓动函数：

- myPlainLinear 该缓动函数非常简单且无趣。
- myDiscrete 该缓动函数将动画严格限制为离散步进方式。
- myPower 该缓动函数逐渐加快动画的速度直到突然停止。

为了探索更多选项，我们可以转向 DojoX 模块：

```
// Sprinkle in some more experimental easings from dojox.
dojo.require('dojox.fx.easing');
dojo.mixin(my_easings, dojox.fx.easing);
```

这段代码载入 dojox.fx.easing 模块并将其内容合并到 my_easings 中，它可提供 30 多种缓动函数。有几个缓动函数非常引人注目，包括 elasticOut、bounceOut 和 backOut，当我们把“实验室”建立好之后要留心它们。

下面的代码让我们能够在用户界面中选择这些缓动函数：

```
// Populate the selection box with available easings.
dojo.query('.slideEase', 'anim_easing')
    .forEach(function(el) {
        for (var name in my_easings) {
            if (dojo.isFunction(my_easings[name])) {
                el.options[el.options.length] =
                    new Option(name, name);
            }
        }
    }, this);
```

该代码定位空白的选择框，并遍历可用的缓动函数，将所有这些缓动函数作为该菜单中的新选项添加进去。接下来，我们要使用选中的缓动函数实现一个动画：

```
dojo.require('dojo.fx');

// Simple helper to perform a slide to some horizontal coord
function performSlideTo(left_coord) {
    var sel  = dojo.query('.slideEase', 'anim_easing')[0];
    var name = sel.options[sel.selectedIndex].value;
    console.log("Using selected easing " + name);

    var anim = dojo.fx.slideTo({
        node:     dojo.query('.toAnim', 'anim_easing')[0],
        duration: 1000,
        easing:   my_easings[name],
        left:     left_coord
    }).play();
}
```

在这段代码中，我们首先使用 dojo.require()载入 dojo.fx 模块，以防止之前没有载入该模块。然后，定义 performSlideTo()函数。

performSlideTo()函数的前半部分查找缓动函数选择框，弄清楚当前选中的值，并且向日志中写入一条消息来报告该选项。然后，创建 dojo.fx.slideTo()动画，这一次使用提供的缓动属性(通过

在选择框选项指定的 my_easings 中查找缓动函数)。

最后，该代码将事件处理程序连接到两个按钮来启动动画：

```
// Wire up the buttons to each perform a slide.
dojo.query('.doSlideOut', 'anim_easing')
    .onclick(function(ev) {
        dojo.stopEvent(ev);
        performSlideTo(300);
    });
dojo.query('.doSlideBack', 'anim_easing')
    .onclick(function(ev) {
        dojo.stopEvent(ev);
        performSlideTo(0);
    });
```

第一个按钮将元素向右移动 300 像素，而第二个按钮将其移回原位。这两个按钮均使用 performSlideTo()函数在一个共同的地方应用选中的缓动效果。

这个实验室框架可用来试验各种缓动函数，但是纯粹采用文本很难描述它们。组合起来试验每种可用的缓动函数，查看是否能够找到一种或多种缓动来替换默认的缓动。

26.6　顺序链接动画

到目前为止，我们已经看到如何启动各种单一的动画。但是，Dojo 还在 dojo.fx 模块中提供了一个工具来构建按照顺序启动的动画链。

现在就研究这项功能，我们将下面的 HTML 标记作为它的测试平台：

```
<style type="text/css">
    #anim_chain .fader {
        margin: 0.25em;
        width: 100px;
        float: left;
        border: 2px dotted #000;
        background-color: #ddf;
    }
    #anim_chain span {
        display: block;
        padding: 1em;
    }
</style>


<div id="anim_chain">

    <div class="fader" id="play1"><span>Click me to fade</span></div>
    <div class="fader" id="play2"><span>Click me to fade</span></div>
    <div class="fader" id="play3"><span>Click me to fade</span></div>
    <div class="fader" id="play4"><span>Click me to fade</span></div>
```

```
    <br style="clear: both" />

</div>
```

这个用户界面由 4 个方块组成，每个方块在单击时都会淡入淡出(由下面给出的代码实现)。图 26-6 给出了该用户界面的显示效果。

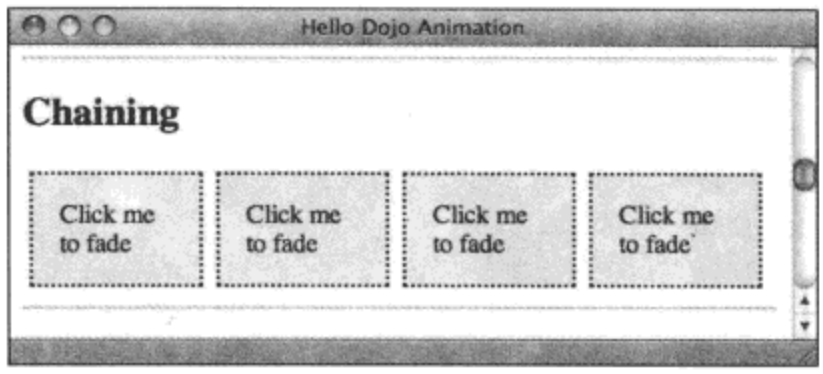


图 26-6

这个计划实施起来相当简单，使用如下的代码即可：

```
dojo.require('dojo.fx');

dojo.query('.fader', 'anim_chain')
    .onclick(function(ev) {
        dojo.fx.chain([
            dojo.fadeOut({ node: ev.currentTarget, duration: 1000 }),
            dojo.fadeIn({ node: ev.currentTarget, duration: 1000 })
        ]).play();
    });
```

上面的代码使用 dojo.query()查找所有的淡入淡出方块，然后链式调用.onclick()，将给定的处理程序连接到所有方块。这个内联处理程序调用 dojo.fx.chain()函数，该函数接受的唯一参数就是一个动画数组。dojo.fx.chain()调用本身会返回一个动画，然后.play()调用启动该动画。

可以使用 dojo.fx.chain()构建任意数量的动画链。按照顺序启动每个动画，每个动画均在前一个动画结束时启动。利用该工具，我们可以构建元素移动的路径和循环以及类似于前面曾经演示过的淡出/淡入效果组合。

26.7 以并行方式组合动画

除了能够顺序启动动画之外，dojo.fx 模块还提供了一种利用 dojo.fx.combine()并行播放动画的方式。

为了准备快速演示，请查看下面的标记：

```
<div id="anim_combine">
    <div class="toAnim" id="toAnim1"><span>Click me</span></div>
    <br style="clear: both" />
    <div class="toAnim" id="toAnim2"><span>Watch me</span></div>
```

```
    <br style="clear: both" />
</div>
```

这段 HTML 标记给出了两个方块，其中一个可单击，并且两者都将制作动画。图 26-7 给出了该标记的呈现结果。

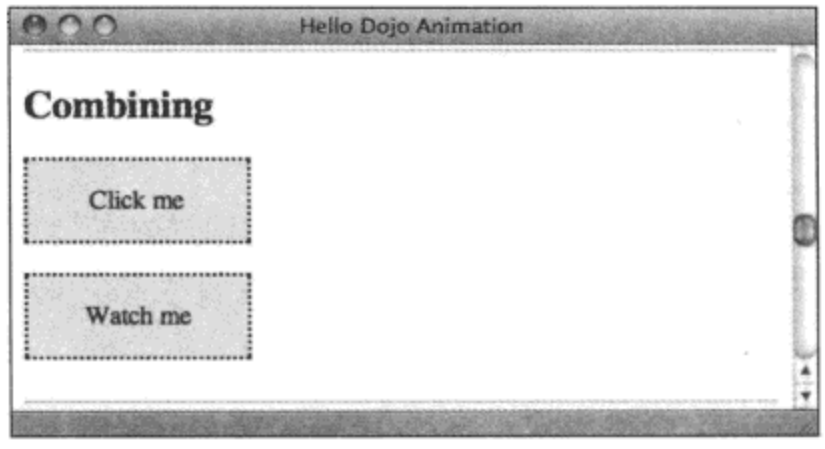


图　26-7

考虑下面的代码，它同时构建链式动画和组合动画：

```
dojo.require('dojo.fx');

dojo.connect(dojo.byId('toAnim1'), 'click', function(ev) {
    dojo.fx.chain([
        dojo.fx.combine([
            dojo.fadeOut({
                node: dojo.byId('toAnim1'), duration: 1000
            }),
            dojo.fx.slideTo({
                node: dojo.byId('toAnim2'), duration: 1000, left: 300
            })
        ]),
        dojo.fx.combine([
            dojo.fadeIn({
                node: dojo.byId('toAnim1'), duration: 1000
            }),
            dojo.fx.slideTo({
                node: dojo.byId('toAnim2'), duration: 1000, left: 0
            })
        ])
    ]).play();
});
```

在上面的代码中发生了许多事情，但是可以归纳如下：当单击第一个元素时，它首先淡出，然后又淡入；而其下方的元素将滑动到右侧，然后又滑动回来。

仔细查看代码，初始的 dojo.connect()调用将内联处理程序连接到第一个元素。在这个处理程序内部，dojo.fx.chain()调用建立一组要播放的动画。

所有这些链接的动画都是由 dojo.fx.combine()调用返回的，下面具体地讨论它。这个方法接受一个由其他动画组成的列表(由在本章中目前已经讨论过的动画辅助函数来提供这些动画)作为参

数。虽然这些动画均各自应用于不同的元素，但是借助 dojo.fx.combine()函数，它们将并行地启动。

利用 dojo.fx.combine()函数，我们可以将任意数量的动画混合到个别元素(甚至相同的元素)上。在淡入一个新数据时擦除一条消息，甚至构建出复杂的并行移动路径。这又为我们提供了另一个利用可用动画基本要素和辅助函数来组合复杂动画的工具。

26.8 使用 NodeList 动画方法

我们已经学习了 Dojo 动画工具在编排单个动画方面提供的功能，现在就来介绍 dojo.NodeList 扩展中有什么方法可用于将动画应用到由 dojo.query()定位的节点集合中。

下面的标记基于前一章中讲解 dojo.query()时曾经使用过的示例：

```
<div id="anim_nodelist">

    <form>
        <button class="doAnim">Animate!</button>
    </form>

    <ul>
        <li><a href="http://decafbad.com">Item #1</a></li>
        <li class="seconditem dojolink">
            <a href="http://dojoproject.org">Item
                <span class="num">#2</span></a>
        </li>
        <li class="thirditem">
            <a href="http://w3.org">Item #3</a>
        <li>
            <span>Sub-list</span>
            <ul>
                <li><a href="http://yahoo.com">Item A</a></li>
                <li><a href="http://delicious.com">Item
                    <span class="num letter">B</span></a></li>
                <li><a href="http://mozilla.org">Item C</a></li>
            </ul>
        </li>
        <li id="item4"><a href="http://getfirebug.com">Item #4</a></li>
        <li><a href="http://wrox.com">Item
            <span class="num">#5</span></a></li>
    </ul>

</div>
```

尽管之前已经看到过，但是图 26-8 仍然给出该标记在浏览器中的显示结果。

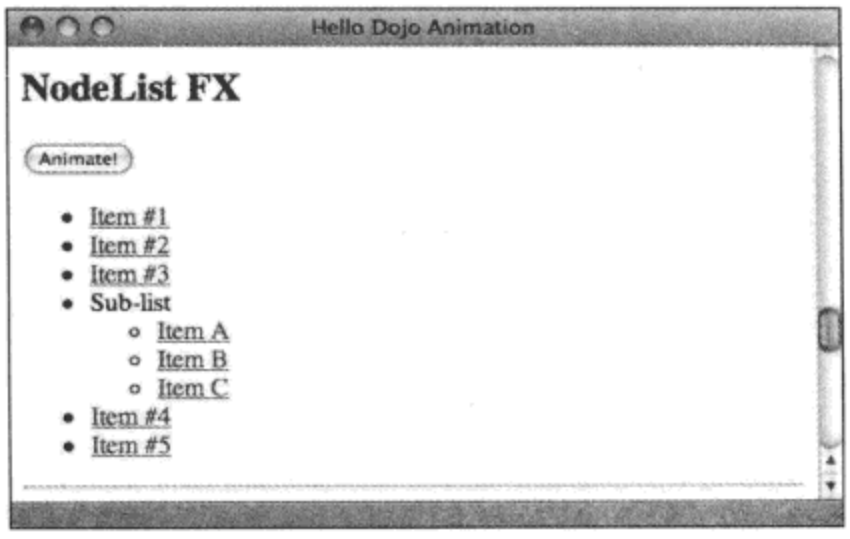


图　26-8

考虑将下面的代码应用于前一个示例中的 HTML 标记：

```
dojo.require('dojo.NodeList-fx');

dojo.query('.doAnim', 'anim_nodelist').onclick(function(ev) {
    dojo.stopEvent(ev);

    dojo.fx.chain([

        dojo.query('a[href^="http://d"]')
            .fadeOut({ duration: 500 }),

        dojo.query('a[href^="http://d"]')
            .fadeIn({ duration: 500 }),

        dojo.fx.combine([

            dojo.query('a[href$=".com"]')
                .map(function (el) { return el.parentNode })
                .animateProperty({
                    duration: 500,
                    properties: {
                        backgroundColor: { start: '#fff', end: '#ff8' }
                    }
                }),

            dojo.query('a[href$=".org"]')
                .map(function (el) { return el.parentNode })
                .animateProperty({
                    duration: 500,
                    properties: {
                        backgroundColor: { start: '#fff', end: '#8f8' }
                    }
                })

        ])
```

```
        }).play();
    });
```

首先，加载 dojo.NodeList-fx 模块。在加载时，这个模块将 dojo.NodeList 类扩展，添加了几个新方法，包括如下几个熟悉的方法：

- .animateProperty()
- .wipeIn()
- .wipeOut()
- .slideTo()
- .fadeIn()
- .fadeOut()

所有这些方法的操作方式均与本章目前已经介绍过的非 NodeList 对等方法相似。区别在于，所有这些方法均遍历 NodeList，并依次应用到该列表的所有节点中。

这些方法与 NodeList 的其他可链接方法不同，它们的返回值是一个动画对象。这个动画对象是使用 dojo.fx.combine()方法构建的，它编排了一组应用于集合中所有节点的并行动画。虽然这结束了 NodeList 上下文中的方法链式调用，但是允许我们把 NodeList 构建动画的结果用于其他的组合动画和链接动画中。

前一个示例中的代码将所有内容结合起来。连接到动画按钮的处理程序编排了一个由多个步骤组成的动画链。在这个动画链中，第一步就是让 URL 以“http://d”开头的所有链接淡出，然后执行相反操作的动画将这些链接淡入。

然后，编排一组并行的动画：将所有含有.com 链接的列表项的背景颜色从白色渐变成浅黄色。与此同时，所有含有.org 链接的列表项的背景颜色均从白色渐变到浅绿色。

这个示例的目的就是为了演示：我们可以只使用少量的代码就将声明式 dojo.query()查询的功能与创建、链接以及组合任意数量和排列的动画结合起来。

26.9 研究动画对象

到目前为止，本章中的所有动画示例均作为由各种动画辅助函数和生成函数返回的带有一个.play()方法的黑盒对象。实际上，所有动画对象均提供了几个方法，可以用于动态地操作它的播放进度，包括下面的这些方法：

- .play(delay, gotoStart) 在可选的延迟之后开始播放动画。如果 gotoStart 设置为 true，就从头开始播放该动画，否则就会从它上一次退出或暂停的地方开始播放(默认值)。
- .pause() 暂停动画。
- .gotoPercent(percent, andPlay) 将动画跳转到给定的完成百分比 percent(一个介于 0.0 和 1.0 之间的小数)。如果 andPlay 为 true，就将从指定位置处开始播放该动画。
- .stop(gotoEnd) 停止播放动画。如果 gotoEnd 为 true，那么在停止时会将该动画跳转到结尾处。
- .status() 返回动画的当前状态，例如下列值之一：paused、playing 或 stopped。

此外，还有几个综合事件可供连接，在动画生命周期的不同时间点将会报告这些事件。它们

包括如下：

- beforeBegin　在开始播放动画之前引发该事件。
- onBegin　已经开始播放动画。
- onAnimate　在动画的播放过程中，每个时钟周期都会引发该事件。
- onPause　当调用.pause()方法时引发该事件。
- onStop　当调用.stop()方法时引发该事件。
- onEnd　当完成动画播放时引发。

利用这些方法和事件，我们可以构建进度条来跟踪实际的播放过程，利用相互关联的线索将复杂的动画串接起来，或者构建动画定时器等，这里只给出了少数几个想法而已。

我们之前讨论缓动时曾经提到，构建一个小型实验室可能是感受这些方法和事件的工作方式的最简单途径。考虑下面的标记，开始这个项目：

```
<div id="anim_events">

    <form>

        <button class="reset">Reset</button>
        <button class="play">Play</button>
        <button class="pause">Pause</button>
        <button class="stop">Stop</button>
        --
        Jump to:
        <button class="go0">0%</button>
        <button class="go25">25%</button>
        <button class="go50">50%</button>
        <button class="go75">75%</button>

        <br style="clear: both" />

        <label for="status">Status:</label>
        <input class="status" type="text" size="40" />

        <br style="clear: both" />

        <div class="toAnim">Watch me!</div>

    </form>

</div>
```

为了练习动画对象提供的方法，这个界面提供了一组按钮，绑定它们以触发这些方法。还有一个文本字段用来在事件引发时报告它。Firebug 日志在这里非常方便。最后，在末尾处还有一个 元素，这将是这个实验中对其制作动画的目标。图 26-9 给出了这段 HTML 标记的呈现结果。

图 26-9

接下来研究代码，下面的代码首先定义一个新类来驱动该界面：

```
dojo.declare('decafbad.ch4.animation.withEvents', null, {

    root_id: '',
    anim: null,

    constructor: function(root_id) {
        this.root_id = root_id;
        this.resetAnim();
        this.wireUpButtons();
    },
```

上面的代码首先声明一个名为 withEvents 的类，它将封装运行该界面所需的所有连接和处理程序。

注意，这个类的构造函数接受一个 root_id 参数，这个参数的值由前面示例标记中的 anim_events 提供。这个类并不是一个合适的窗口部件，但由于 root_id 被参数化，因此我们可以尝试制作这个标记的多个带有不同 ID 的副本，并为每个副本实例化这些对象之一。我们可以并行地运行许多事件并彼此独立地处理它们。

构造函数的另外两个职责是调用.resetAnim()重置动画以及调用.wireUpButtons()绑定所有按钮。首先，.resetAnim()方法开头的代码如下：

```
    resetAnim: function(ev) {
        console.log("Resetting animation");

        // If there's an existing animation, stop it.
        if (this.anim) this.anim.stop();

        // Build the new animation object.
        var root_id = this.root_id
        this.anim = dojo.animateProperty({
            node: dojo.query('.toAnim', root_id)[0],
            duration: 10000,
            properties: {
                left: { start: '0', end: '300', unit: 'px' },
            },
            beforeBegin: function() {
                console.log(root_id + ': beforeBegin fired!');
```

```
        },
        onBegin: function() {
            console.log(root_id + ': onBegin fired!');
        },
        onAnimate: function() {
            // No console.log here - too noisy!
        },
        onPause: function() {
            console.log(root_id + ': onPause fired!');
        },
        onStop: function() {
            console.log(root_id + ': onStop fired!');
        },
        onEnd: function() {
            console.log(root_id + ': onEnd fired!');
        }
    });
```

在.resetAnim()方法中，首先要做的工作就是检查在 this.anim 中是否已有动画。如果发现有动画，就停止该动画。然后，开始构建一个新的动画。

在这里，该动画只是一个使用 dojo.animateProperty()实用工具创建的、为期 10 秒的、从左向右的滑动过程。这里演示的新功能是，这个实用工具接受的属性包含了一些指定的处理程序，用来处理返回的动画对象所提供的所有事件。由于在本章中我们到目前为止从来没有使用过这些处理程序，因此它们当前当然都是可选的。但为了讲解基础知识，我们为每个事件提供了一个处理程序来生成日志消息。

这里的一个例外情况就是 onAnimate 处理程序：这个处理程序引发了很多事件(实际上是每秒多次引发事件)。这个速率可能太高，而且它生成的日志文本消息有些过多，因此在前面的代码中，这个处理程序并没有做任何工作。这里之所以给出它，只是为了演示也可以使用它。

但在.resetAnim()方法的另一半代码中，我们可以看到另一种连接到动画对象所提供事件的方式：

```
    // Wire up connections for each event fired by the animation.
    var anim_events = [
       'beforeBegin', 'onBegin', 'onPause',
       'onStop', 'onAnimate', 'onEnd'
    ];
    dojo.forEach(anim_events, function(ev_name) {
        var handler = dojo.hitch(this, function(val) {
            this.setStatus(ev_name, val);
        });
        dojo.connect(this.anim, ev_name, handler);
    }, this);
},

setStatus: function(ev_name, val) {
    var str = ev_name + ' ' + this.anim.status();
    if (val) str += ' ' + dojo.toJson(val);
    dojo.query('.status', this.root_id)[0].value = str;
},
```

这里的代码有些密集，但是可以归纳如下：.resetAnim()的后半部分使用 dojo.connect()将处理程序连接到动画对象提供的每个方法中，每个处理程序调用新的.setStatus()方法来更新用户界面中的文本字段，使用一条消息报告最近引发的事件、该动画的.status()方法的当前结果以及关于传递给处理程序的参数(如果有的话)的描述。

这里比较有趣的部分是 dojo.toJson()实用工具的使用，我们将在下一章中更加深入地讨论它。这个函数的主要功能是获取一个 JavaScript 数据结构并将其转换成 JSON 字符串表示，从而便于观察在传给每个事件处理程序的参数中发生的情况。

因此，除了在创建动画时提供处理程序之外，我们还可以在后面使用 dojo.connect()方法将处理程序附加到该动画引发的一系列综合事件。

接下来是前面给出的.wireUpButtons()方法的定义：

```
wireUpButtons: function() {

    dojo.forEach(['reset', 'play', 'pause', 'stop'], function(name) {
        dojo.query('.'+name, this.root_id)
            .onclick(this, 'handle' + name);
    }, this);

    dojo.forEach(['0', '25', '50', '75'], function(perc) {
        dojo.query('.go'+perc, this.root_id).onclick(
            dojo.hitch(this, function(ev) {
                return this.handlego(ev, perc);
            })
        );
    }, this);

},
```

.wireUpButtons()方法的工作分为两个部分。

第一部分将 reset、play、pause 和 stop 按钮的单击事件连接到带有 handle 前缀和按钮名称后缀的处理程序方法。

第二部分构建并连接中间处理程序，每个处理程序调用.handlego()方法，其参数为预制的百分比参数和接收到的事件。这有助于减少需要实现的不同处理程序的数量，因为只有百分比不同而已。对于所有其他处理程序也进行相同的处理，但是为了演示，将这些处理代码明确写出来似乎更有用。

对于处理程序，下面的代码提供了它们的实现并将其连接到 withEvents 类：

```
handlereset: function(ev) {
    dojo.stopEvent(ev);
    this.resetAnim();
},

handleplay: function(ev) {
    dojo.stopEvent(ev);
    this.anim.play();
},
```

```
        handlepause: function(ev) {
            dojo.stopEvent(ev);
            this.anim.pause();
        },

        handlestop: function(ev) {
            dojo.stopEvent(ev);
            this.anim.stop();
        },

        handlego: function(ev, perc) {
            dojo.stopEvent(ev);
            this.anim.gotoPercent( perc / 100.0, true);
        }

    });

    var obj = new decafbad.ch4.animation.withEvents('anim_events');
```

在上面的代码中，我们定义了处理按钮事件所需的处理程序。首先，.handlereset()包装了对.resetAnim()方法的调用，从头开始播放动画。其余处理程序均包装了对动画对象上的某个方法的调用，分别包括.play()、.pause()、.stop()和 gotoPercent()。

.handlego()处理程序稍微有些特殊，因为它从之前连接的中间处理程序那里接受一个 0~100 的百分比值。它还将 andPlay 参数硬编码为 true，使得界面中的所有“Jump to(跳转到)”按钮把动画跳转到指定位置后播放，而无须单击播放按钮。

最后，在完成类的声明之后，使用 ID anim_events 作为参数创建该类的一个实例，将该实例绑定到本节开头提供的标记。您可能已经注意到，在整个代码中，所有尝试为事件和类似对象获取节点的操作均结合使用 CSS 类名和 dojo.query()调用，它们均以这个 ID 为基础。这就实现了前面提出的让该代码可以在标记的多个实例之间移植和重用的功能。

这个研究用界面应该能够让您对如何与动画事件和方法交互有了直观的感受。观察状态字段和日志消息来获取线索，并使用按钮尝试各种方法调用组合。

26.10　本章小结

在本章中，我们介绍了 Dojo 提供用来编排动画和视觉转换的工具。我们演示了 Dojo 中的各种动画基本要素，以及链接和组合动画并将缓动应用于动画流的方式。然后，我们介绍了 NodeList 扩展方法，并给出如何使用 dojo.query()构造特定节点集合的动画。最后，我们深入研究了动画对象的工作原理，并给出了一个简单的界面进行试验。

在第 27 章中，我们将向该组合中添加外部数据源。其中将介绍 Dojo 提供的各种实用工具，这些实用工具用于通过使用 AJAX 和 IFrame 来访问服务器端数据，以及在各种基于 Web 的服务上执行远程过程调用。

第27章

处理 AJAX 和动态数据

与浏览器状态栏中不断滚动的跑马灯提示相比，第一批采用动态 HTML 和 DOM 脚本编程的 Web 应用程序当然要有趣得多，但在浏览器中，它们本质上都是封闭的系统。也就是说，用户界面所需的一切内容在页面加载时都必须就绪，而且只能通过表单提交才能将数据传回服务器。因此，将客户端的用户交互活动与服务器端的处理程序结合起来的唯一方式就是定期采用页面刷新重新加载整个经过精心构造的界面。

为了改善这种情况，有几种方式已经逐渐兴起，使得 Web 客户端可以与服务器进行更小规模的就地交互。尽管这些技术多种多样，但它们通常都归类到首字母缩写词“AJAX”下。

这个术语的原始展开形式(Asynchronous JavaScript and XML)后来有些被误用，这个概念的几乎每个部分(可能除 JavaScript 自身之外)都已经或多或少替换成其他技术和数据格式。无论如何，基本的概念保持不变：不是重新加载整个页面，而是只执行给定用户界面交互所需的小型的、专注的 Web 客户端-服务器对话。

在 Dojo 中，可以任意使用几种方式来请求和处理动态数据，既可以使用在其中服务页面的相同域，也可以跨越来自第三方数据提供商的不同域。我们将在本章稍后部分看到，Dojo 工具集为使用 XMLHttpRequest 对象、IFrame 和 JSON 数据馈送提供抽象机制，还有几种方法用来自动分析和处理这些方式提供的数据。

本章内容简介：

- 建立简单的 Web 请求
- 处理 Web 响应
- 增强表单

27.1 建立简单的 Web 请求

最简单的 AJAX 事务之一就是通过 HTTP GET 获取资源。借助大多数现代浏览器中可用的 XMLHttpRequest 对象，我们可以建立连接从中加载页面的服务器域的大部分 HTTP 请求类型。这项著名限制的作用在于限定给定网页的安全作用域和流量影响，但在本章后面我们将看到几种绕开跨域限制获取数据的方式。

无论如何，为了研究 Dojo 的最简单的 XMLHttpRequest 工具，请考虑下面的代码：

```
<style type="text/css">
    .success {
        background-color: #dfd;
        border: 2px solid #000;
    }
    .error {
        background-color: #fdd;
        border: 2px dotted #000;
    }
</style>

<div id="xhrget1">
    <h2>Simple xhrGet</h2>
    <p class="content_found">Loading...</p>
    <p class="content_not_found">Loading...</p>
</div>
```

我们在前几章中已经看到，Dojo 提供了大量的工具用于操作这段标记，包括动画和修改内容。在本章中，我们关注的对象就是内容。

对于内容，我们假设下面的数据位于服务器上名为“data.txt”的文件中，该文件位于包含前面示例中标记的页面文件所在的目录中：

```
Hello world, this is some text available for load.
```

本章剩下的所有示例都会引用这个简单的文本文件。

27.1.1 建立简单的请求并处理响应

XMLHttpRequest 的使用天生就是异步操作，因此这里讲解 AJAX 中的第一个字母“A”。这意味着建立请求和接受响应这两个操作在时间和执行上下文方面均是独立的，也就是说请求通过立即返回的方法调用启动，而响应则由回调函数(并非与处理 DOM 所使用的回调完全不同)处理。

这就让我们可以并行地启动多个同时发出的请求，而且在 Web 请求仍然运行各自的处理过程时继续保持用户界面的灵敏响应；而同步的 Web 请求会在浏览器等待请求完成的过程中将页面上的一切内容冻结。虽然有可能这样做，但这通常并不是我们想要的操作方式，因此现代 Web UI 一直在致力于处理异步流。

查看下面的代码：

```
dojo.xhrGet({
    url: 'data.txt',
    timeout: 5000,
    load: function(resp, io_args) {
        dojo.query('.content_found', 'xhrget1')
            .addClass('success')
```

```
            [0].innerHTML = resp;
    },
    error: function(error, io_args) {
        dojo.query('.content_found', 'xhrget1')
            .addClass('error')
            [0].innerHTML = error.message;
    }
});
```

该代码试图使用 dojo.xhrGet()调用获取 data.txt 资源，该调用立即返回，并在后台引发 Web 请求。下面是用到的参数：

- url　建立请求所使用的 URL。
- timeout　浏览器应该等待响应多长时间(单位为毫秒)。
- load　当接收到成功的响应时调用该回调函数。
- error　当遇到失败或错误响应时调用该回调函数。

如果该请求最终产生了一个成功的响应，就会调用 load 参数定义的函数。对于前面的标记而言，这个函数将把第一个 元素的 CSS 类切换成“success”，并将其内容替换成获取的数据。在传给回调的第一个参数 resp 中包含了 Web 请求所返回的响应数据。

第二个回调参数 io_args 的有趣之处在于，它提供了该请求的额外上下文。这是一个内部 Dojo 类的实例，它的属性包括：

- args　传给 dojo.xhrGet()的原始参数的一个副本，其中还包括未被 dojo.xhrGet()理解的自定义参数。可以使用这个属性将数据直接传给响应处理程序。
- xhr　请求中使用的 XMLHttpRequest 对象，对于查询响应的报头以及其他细节信息非常有用。
- url　该请求使用的最终的 URL，由于重定向和其他变化，该 URL 通常不同于最初的 url 参数。
- query　一个表示在请求中发送的查询参数的对象。

第二个回调名为 error，当请求遇到问题时就会调用该回调。此时， 元素的 CSS 类将切换成“error”，而且最终的错误消息将替换该 元素的内容。

对于 error 处理程序，第一个参数将是一个类型为 Error 的对象，这是一个包含如下属性的原生 JavaScript 对象：

- message　关于该错误的人类可阅读的描述。
- filename　出现该错误的文件的名称。
- lineNumber　该错误在上述文件中出现时所在的行号。

在该上下文中，只有 message 属性真正有用。现在，我们已经解释了 dojo.xhrGet()调用，接下来查看图 27-1 以了解成功请求在页面上产生的效果。

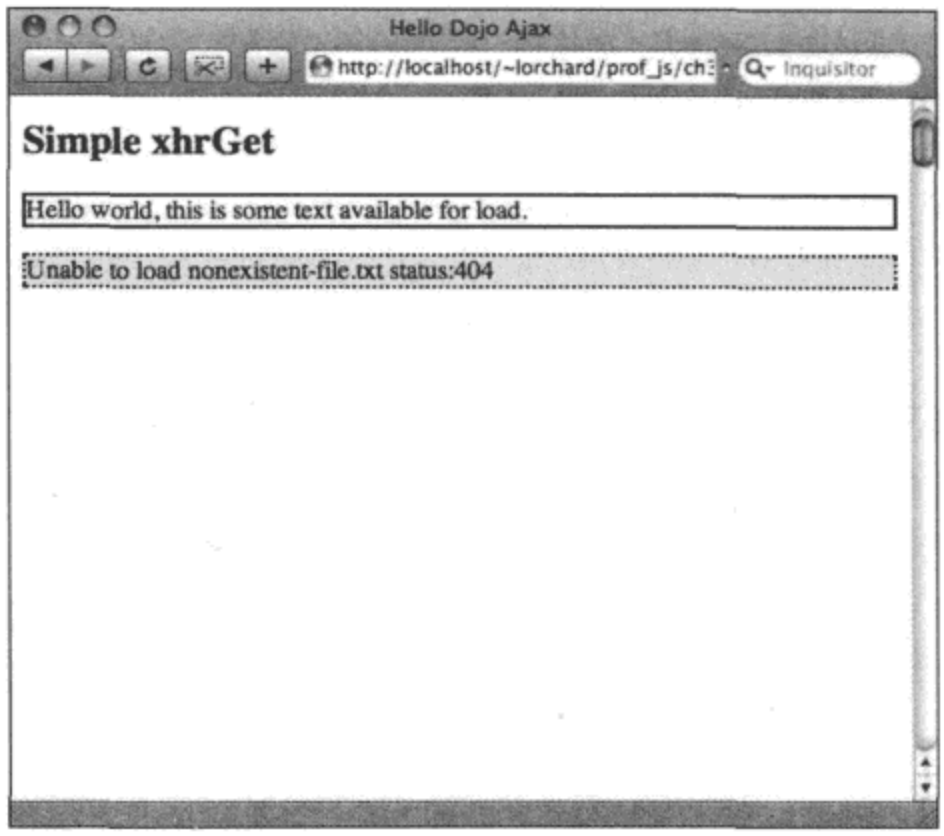


图　27-1

27.1.2　使用一个处理程序同时处理错误和成功响应

在有些情况下，为处理错误和成功情况分别定义两个单独的处理程序回调有些多余。因此，dojo.xhrGet()还允许提供一个函数来处理这两种情况，就像下面这样：

```
dojo.xhrGet({
    url: 'nonexistent-file.txt',
    timeout: 5000,
    handle: function(result, io_args) {
        if (result instanceof Error){
            dojo.query('.content_not_found', 'xhrget1')
                .addClass('error')
                [0].innerHTML = result.message;
        } else {
            dojo.query('.content_not_found', 'xhrget1')
                .addClass('success')
                [0].innerHTML = result;
        }
    }
});
```

该代码基本上与前一个示例相同，只是 load 和 error 处理程序已经替换成一个名为 handle 的回调函数。该函数检查它的第一个参数的类型，如果请求失败，该参数就是 Error 的一个实例。因此，一个处理程序就可以负责处理错误和成功两种情况。

注意，图 27-1 也描绘了第二个调用的结果，即故意尝试获取应该不存在的资源，从而导致错误。

27.2　利用 Deferred 处理 Web 响应

尽管 dojo.xhrGet()使得在 Dojo 中处理请求响应变得简单，但还需要其他方面的一些操作。实际上，dojo.xhrGet()调用返回一个名为 Deferred 的类的对象。该对象表示后台任务最终的成功或错误结果，并且接受针对不同的情况注册的多个回调。

为了精确地了解其中的含义，我们首先考虑下面的 HTML 标记，后面给出的代码示例将会更新该标记：

```
<div id="xhrget2">
    <h2>Deferred xhrGet</h2>
    <p class="content_found1">Loading...</p>
    <p class="content_found2">Loading...</p>
    <p class="content_not_found1">Loading...</p>
    <p class="content_not_found2">Loading...</p>
</div>
```

27.2.1　为成功和错误响应注册处理程序

当我们向 dojo.xhrGet()提供 load 和 error 参数时，Dojo 就会自动完成等同于下面代码的功能：

```
var defer1 = dojo.xhrGet({
    url: 'data.txt', timeout: 5000
});

defer1.addCallback(function(resp) {
    var io_args = defer1.ioArgs;
    dojo.query('.content_found1', 'xhrget2')
        .addClass('success')
        [0].innerHTML = resp;
    return resp;
});

defer1.addErrback(function(error) {
    var io_args = defer1.ioArgs;
    dojo.query('.content_found1', 'xhrget2')
        .addClass('error')
        [0].innerHTML = error.message;
    return error;
});
```

在 Dojo 内部，创建 Deferred 对象并调用.addCallback()和.addErrback()方法来注册由参数提供的 load 和 error 处理程序。在上面的代码中，通过直接使用 dojo.xhrGet()返回的 Deferred 对象明确这个过程。

请注意，成功和错误响应回调均接收一个参数，分别为响应内容和 Error 对象(以前 io_args 是作为传入 dojo.xhrGet()调用的处理程序的第二个参数，而现在可以通过 Deferred 对象本身的 ioArgs

属性来访问 io_args)。

还要注意的是，每个这样的处理程序均返回作为参数传入的值。这一点非常重要，因为我们可以按照下面的方式依次注册额外的处理程序：

```
defer1.addCallback(function(resp) {
    dojo.query('.content_found2', 'xhrget2')
        .addClass('success')
        [0].innerHTML = resp;
    return resp;
});

defer1.addErrback(function(error) {
    dojo.query('.content_found2', 'xhrget2')
        .addClass('error')
        [0].innerHTML = error.message;
    return error;
});
```

在这个示例中注册的处理程序将会在之前注册的处理程序之后引发。此外，它们接收到的参数将是前面的处理程序返回的参数。这样一来，我们就可以构建回调链，它们在向后续处理程序传递的过程中可以选择对结果进行过滤。

这一点值得额外关注是因为，在基于 Deferred 对象的回调末尾遗漏 return 语句是一种简单易犯的错误，这会导致链式回调无法访问响应数据。

27.2.2 在一次调用中注册错误和成功处理程序

如果发现自己正在使用 Deferred 接口，而且经常同时注册成功和错误响应处理程序，那么下面的便利方法非常有用：

```
var defer2 = dojo.xhrGet({
    url: 'nonexistent-file.txt', timeout: 5000
});

defer2.addCallbacks(
    function(resp) {
        dojo.query('.content_not_found1', 'xhrget2')
            .addClass('success')
            [0].innerHTML = resp;
        return resp;
    },
    function(error) {
        dojo.query('.content_not_found1', 'xhrget2')
            .addClass('error')
            [0].innerHTML = error.message;
        return error;
    }
);
```

在上面的代码中，单个 Deferred 方法.addCallbacks()同时取代了.addCallback()和.addErrback()方法。传给.addCallbacks()方法的第一个参数应该是成功回调处理程序，而错误处理程序则应该是第二个参数。

27.2.3　注册一个同时处理错误和成功响应的处理程序

如果发现自己希望注册一个回调来同时负责错误和成功情况的处理，那么下面的代码基本上就等同于在 dojo.xhrGet()中使用 handle 参数：

```
defer2.addBoth(function(result) {
    if (result instanceof Error){
        dojo.query('.content_not_found2', 'xhrget2').addClass('error')
            [0].innerHTML = result.message;
    } else {
        dojo.query('.content_not_found2', 'xhrget2').addClass('success')
            [0].innerHTML = result;
    }
    return result;
});
```

传给.addBoth()的唯一参数应该是在处理错误和成功情况时均调用的处理程序函数。与传给 dojo.xhrGet()方法的 handle 参数一样，该函数根据它的唯一参数的类型来确定要处理的是错误情况还是成功情况。

图 27-2 给出了这个 Deferred 操作在示例 HTML 页面中产生的结果。

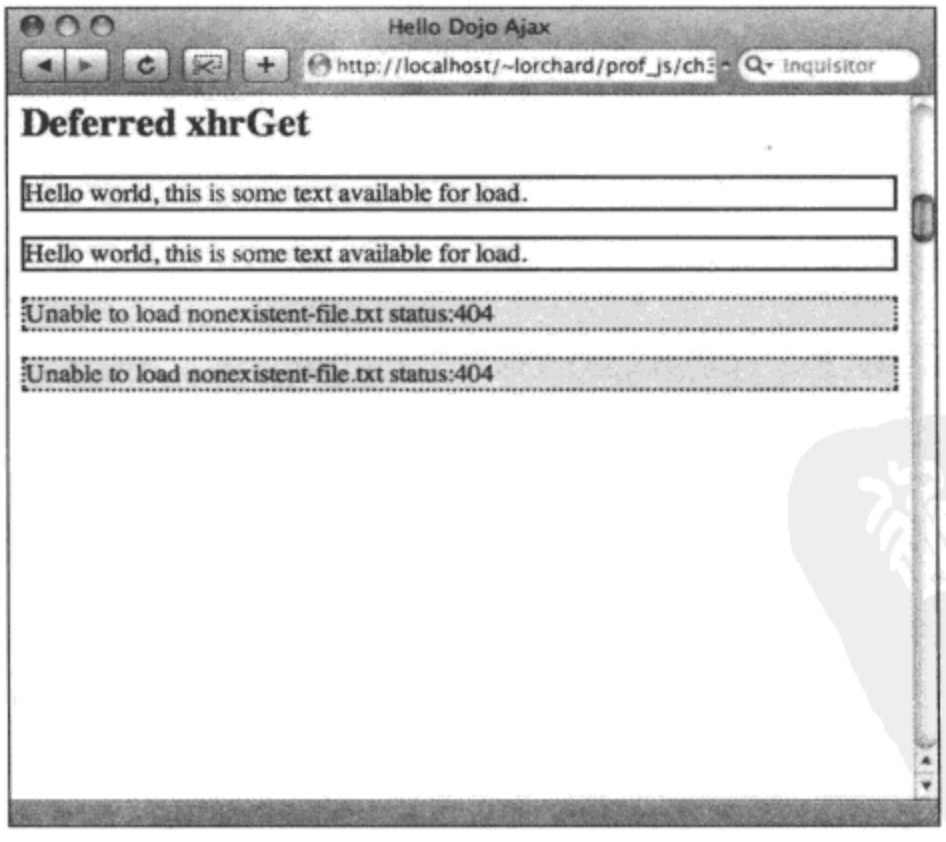


图　27-2

27.3 处理响应格式

到目前为止，在建立简单的 HTTP GET 请求时，本章中的成功响应处理程序均一直将响应内容作为普通文本对待。而且在默认情况下，这是 Dojo 处理响应的默认方式。但 Dojo 也理解其他几种对于 Web 响应非常有用的数据格式，而且能够自动将原始响应数据转换成更有用的对象。

将下面的标记作为一个显示本节后面几个示例的输出结果的框架：

```
<div id="xhr_handle_as">
    <h2>xhrGet with handleAs</h2>
    <p class="content_text">Loading...</p>
    <p class="content_xml">Loading...</p>
    <p class="content_json">Loading...</p>
    <p class="content_js">Loading...</p>
    <p class="content_json_comment">Loading...</p>
</div>
```

此外，记住下面这个用来更新该标记内容的辅助函数：

```
function updateContent(result, handle_as, sel) {
    if (result instanceof Error){
        dojo.query(sel, 'xhr_handle_as').addClass('error')
            [0].innerHTML = result.message;
    } else {
        dojo.query(sel, 'xhr_handle_as').addClass('success')
            [0].innerHTML = handle_as + ' [' + (typeof result) + '] ' + result;
    }
}
```

这段代码将处理 Web 请求的成功和错误情况，利用错误消息或数据处理报告和请求返回的内容来更新 CSS 类标识的给定元素。

27.3.1 处理文本响应

将响应数据作为文本处理，这是 dojo.xhrGet()的默认情形，我们已经看到过这一点。为了明确这个过程，下面的代码引入 handleAs 参数，并取其默认值“text”。

```
dojo.xhrGet({
    url:      'data.txt',
    handleAs: 'text',
    timeout:  5000,
    handle: function(result, io_args) {
        return updateContent(result, 'text', '.content_text');
    }
});
```

假设成功地获取本章目前使用的 data.txt 文件的相同内容，这段代码将使用 updateContent()辅助函数将<p class="content_text">的内容替换成类似下面的信息。

```
text [string] Hello world, this is some text available for load.
```

这里没有什么让人感到意外的地方：将该内容作为 text 类型处理，响应数据参数的 JavaScript 类型是 string，而内容自身看起来符合预期。

27.3.2　处理 XML 响应

此处有趣的事情是修改 handleAs 的值，并获取不同类型的数据。在这里，我们考虑 XML 类型，它就是 AJAX 中的“X”：

```
<?xml version="1.0"?>
<foo>
    <bar>hello</bar>
    <baz>world</baz>
</foo>
```

假设这个简单的 XML 文档位于一个名为 data.xml 的文件中，考虑下面用来获取该文件的代码示例：

```
dojo.xhrGet({
    url:      'data.xml',
    handleAs: 'xml',
    timeout:  5000,
    handle: function(result, io_args) {
        console.log("Fetched XML data, root tag is:");
        console.log('<' + result.firstChild.tagName + '>');
        return updateContent(result, 'xml', '.content_xml');
    }
});
```

假设请求成功，下面将是放入到“content_xml”段落中的结果：

```
xml [object] [object XMLDocument]
```

此外，当该回调运行时会记录下面的消息：

```
Fetched XML data, root tag is:
<foo>
```

请注意这里与 handleAs: 'text'的不同之处：Dojo 并不是简单地将数据传入回调，而是将 handleAs: 'xml'作为一个提示，首先试着将数据作为 XML 文档进行分析，并将该对象作为响应传给回调。

关于 XML 响应有些比较有趣的事情值得注意：在规范化这种行为的背后，有一些跨浏览器问题需要解决。尽管原生 XMLHttpRequest 对象自带 responseXML 属性，但有些浏览器拒绝提供 XML 文档，除非 Content-Type 报头正确显示为 XML 内容。

相反地，Dojo 的 handleAs 行为确保，无论 Content-Type 报头是什么，只要内容可以分析成 XML 内容，就会有一个 XML 文档可用。

27.3.3 处理 JSON 响应

XML 文档响应并不是唯一的选择。JSON(JavaScript Object Notation，http://json.org)是另一个可用来包装结构化数据的选择。

查看下面的示例，它位于 data.json 文件中：

```
{"url":"http:\/\/decafbad.com","title":"0xDECAFBAD"}
```

简而言之，这是一种采用 JavaScript 语法子集来表达数据结构的方式。尽管实际情况可能有所不同，但是根据定义，这种格式是一种可执行 JavaScript 代码。

为了获取该数据，handleAs 参数取值“json”，如下所示：

```
dojo.xhrGet({
    url:      'data.json',
    handleAs: 'json',
    timeout:  5000,
    handle: function(result, io_args) {
        console.log("Fetched JSON data:");
        console.dir(result);
        return updateContent(result, 'json', '.content_json');
    }
});
```

对于成功的响应，下面的内容将会出现在<p class="content_json">元素中：

```
json [object] [object Object]
```

此外，我们还会在日志文本消息中看到下面的输出结果，它提供了关于获取的数据在数据结构方面的更详细信息：

```
Fetched JSON data:
title
      "0xDECAFBAD"
url
      "http://decafbad.com"
```

虽然处理 XML 数据的工具集非常丰富，但 JSON 数据处理起来通常更快，而且更简单。毕竟，这种数据格式采用 JavaScript 自身表达，因此分析器就不用再调用 eval()。

27.3.4 处理注释过滤 JSON 响应

使用 eval()分析 JSON 可能会带来问题，除非我们对 Web 应用程序中的服务器和客户端具有完全的控制权，毕竟它是一种可执行代码，而 eval()并不区分简单的 JSON 数据与更复杂的 JavaScript 代码(可能包含潜在的威胁应用程序的漏洞)。

此外，让 Web 主机轻易地使用 eval()提供数据可能会把数据暴露给位于应用程序客户端和服务器之外的非预期代理使用。可以使用一种称为 JavaScript 劫持(JavaScript Hijacking)的技术包含一个脚本标记，将应用程序的 JSON 数据加载到另一个域的页面中(通过重写某些本地对象构造函数)并在应用程序之外访问该数据。

这种技术带来的危险可能被过分夸大，但如果应用程序仅仅依靠 cookie 作为通过简单 JSON 获取的私有数据的身份验证保护措施，那么就可能会遇到问题。有关 JavaScript 劫持的更多内容，请参见下面的链接：

```
www.fortify.com/security-resources/javascripthijacking.jsp
```

为了对抗应用程序遇到的这种威胁，Dojo 提供了名为注释过滤(Comment-Filtered)JSON 的 JSON 变体。就像下面这样：

```
/*{"url":"http:\/\/decafbad.com","title":"0xDECAFBAD"}*/
```

唯一真正的变化就是 JSON 数据包装到 JavaScript 注释中，从而导致不可能跨域使用<script>标记来访问该数据，或者不可能在不首先将注释符去除的情况下直接调用 eval()来使用它。Dojo 提供了一个名为“json-comment-filtered”的 handleAs 类型，它就用来实现该功能。

假设该数据已经可以通过 data.json-comment-filtered 访问，下面的代码用于获取它：

```
dojo.xhrGet({
    url:      'data.json-comment-filtered',
    handleAs: 'json-comment-filtered',
    timeout:  5000,
    handle: function(result, io_args) {
        console.log("Fetched JSON data:");
        console.dir(result);
        return updateContent(result, 'json-comment-filtered',
            '.content_json_comment');
    }
});
```

与前面的 JSON 一样，成功的响应将下面的内容插入到“content_json_comment”段落中：

```
json [object] [object Object]
```

而且在日志文本消息中会出现与原始 JSON 相同的消息：

```
Fetched JSON data:

title
      "0xDECAFBAD"
url
      "http://decafbad.com"
```

27.3.5　处理 JavaScript 响应

与注释过滤 JSON 约定寻求预防攻击针锋相对的是，Dojo 还能够动态加载并执行任意 JavaScript 代码。只要我们已经控制服务器从域中发回的内容，那么该功能就没有听起来那么令人担忧，而且它对于根据服务器端的决策按需将代码注入到页面中非常有用。

考虑下面的数据，假设它位于 data.js 文件中：

```
(function() {
    console.log("Executable JavaScript ahoy!");
```

```
    dojo.query('.content_js', 'xhr_handle_as')
        .addContent('<p>Look, a side-effect!</p>', 'after');
    return "Hello! Today is " + ( new Date() );
})();
```

请注意，除了在客户端构建的返回值，它还向日志控制台发送一条消息，并附带地向页面中插入一些内容。

按照与所有其他响应格式相同的方式，使用适当的 handleAs 值获取(并执行)这段内容：

```
dojo.xhrGet({
    url:      'data.js',
    handleAs: 'javascript',
    timeout:  5000,
    handle: function(result, io_args) {
        return updateContent(result, 'javascript', '.content_js');
    }
});
```

该获取操作的成功响应将导致 content_js 段落中出现类似下面的内容：

```
javascript [string] Hello! Today is Tue Mar 11 2008 21:33:23 GMT-0700 (PDT)
```

此外，紧跟着该段落元素之后将出现一个带有如下内容的新段落元素：

```
Look, a side-effect!
```

最后，下面这条消息将出现在日志中：

```
Executable JavaScript ahoy!
```

所有这些操作只是为了强调这种特殊的响应处理类型的灵活性。来自服务器的 JavaScript 响应可以对页面以及客户端上的用户界面执行任何操作，既可以修改页面，也可以更新驻留在内存中的类和代码。

图 27-3 给出了标记中已经完成的所有响应格式处理程序。

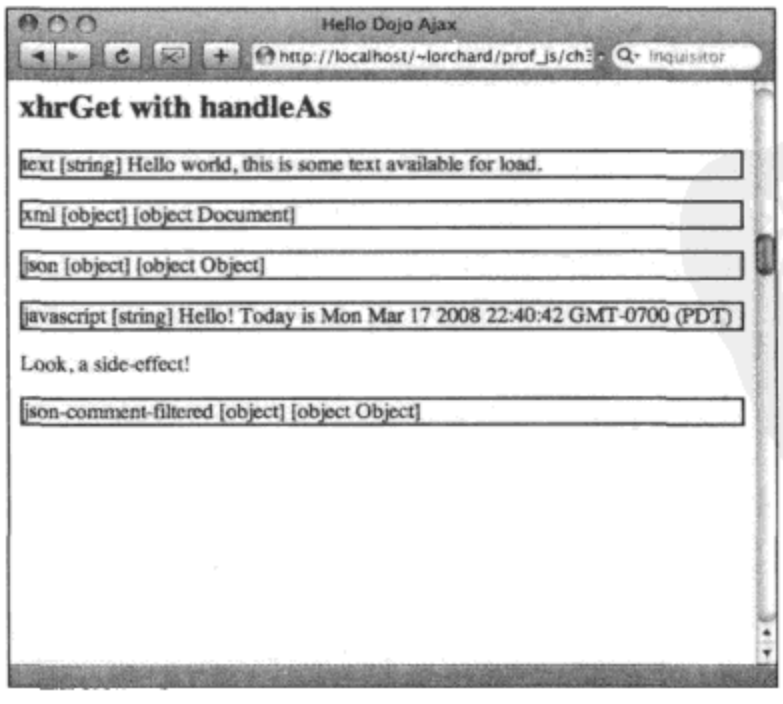


图 27-3

27.4　指定请求方法

在本章中，到目前为止只演示了 HTTP GET 请求(使用 dojo.xhrGet()实用工具)。但对于 HTTP 来说，不仅仅只有 GET 方法：在浏览器中使用表单时至少会遇到 POST 方法，而在研究 REST 形式的 Web 服务时就需要用到其他方法，如 PUT、DELETE、HEAD 和 OPTIONS。

对于这些额外的方法，Dojo 提供了如下的实用工具函数：

- dojo.xhrPost(params)
- dojo.xhrPut(params)
- dojo.xhrDelete(params)

这些函数的工作方式基本上与 dojo.xhrGet()相同：通过请求参数传递的数据结构应该包含同样的信息，例如 url、timeout、handleAs、load、error、handle。这些函数也会返回 Deferred 的一个实例，就像 dojo.xhrGet()一样。

实际上，所有这些方法特有的实用工具均是下面这个函数的包装器：

```
dojo.xhr(method, params, has_body)
```

第一个参数 method 应该是 HTTP 方法(例如 GET、POST、PUT)。第二个参数应该是描述该请求的相同数据结构(就像目前在 dojo.xhrGet()示例中已经使用的数据结构一样)。

第三个也是最后一个参数 has_body 是一个 Boolean 标志，指示该请求是否应该在发送时带有请求正文。方法 POST 和 PUT 应该将这个标志设为 true，但实际的发送请求正文的过程将在下一节中讲解。

27.4.1　构建一个服务器端请求回应工具

现在，在对这些调用进行试验之前，将注意力暂时先放在一段服务器端代码上或许会有所帮助。到目前为止给出的所有示例均很好地处理静态数据文件，但现在我们要查看 HTTP 请求/响应对话之外的一点内容。

由于其流行性，您很可能已经访问过运行 PHP 的 Web 服务器。因此，将下面的代码作为研究工具 echo.php 的开始：

```
<?php
// Assemble a report of what was received in the request.
$out = array(
    'method' => $_SERVER['REQUEST_METHOD'],
    'post_params' => $_POST,
    'query_params' => $_GET,
    'accept' => $output_type,
    'x_requested_with' => $_SERVER['HTTP_X_REQUESTED_WITH'],
);

// If a request body was present include it in the response.
```

```
if ($_SERVER['CONTENT_LENGTH']) {
    $out['content_type']   = $_SERVER['CONTENT_TYPE'];
    $out['content_length'] = $_SERVER['CONTENT_LENGTH'];
    $out['request_body']   = file_get_contents('php://input');
}
```

这段代码汇集了一些描述访问该脚本的请求的关键数据点，这为我们在响应中把这些数据以某种形式反馈回去做好了准备。它们包括：

- 请求中使用的 HTTP 方法
- 在请求正文中发送的所有 POST 参数
- 在 URL 中传递的所有查询参数
- 随请求发送的 Accept:报头
- X-Requested-With 报头的值(Dojo 将其设为“XMLHttpRequest”，这可能有助于在服务器端代码中区分 AJAX 请求与其他页面请求)

最后，如果在请求正文中带有内容，那么它的长度、类型以及内容本身将会包含在为响应收集的数据中。

接下来，为了进一步练习 Dojo 的 AJAX 辅助函数的一些异步功能，下面的脚本包含了一个选项，可以仿造一个长时间运行过程：

```
// If a ?wait parameter passed, wait that number of seconds
// This lets us pretend there's work going on to allow animated
// spinners to spin for a little while.
if ($_GET['wait']) {
    sleep(intval($_GET['wait']));
}
```

有了上面的代码，向该脚本提供?wait=10 参数将会导致它等待 10 秒时间。一般情况下，在服务器处理工作期间，客户端代码可以显示正在加载消息，给出一个带有不断旋转的指示器的动画 GIF，或者执行任何其他想做的操作。不管怎样，添加一个延迟将让执行速度变慢，从而可以了解都发生了什么事情。

最后，该脚本汇编一个响应结果：

```
// Accept either a ?type query parameter or an Accept: header
if ($_GET['type']) {
    $output_type = $_GET['type'];
} else {
    $output_type = $_SERVER['HTTP_ACCEPT'];
}

// Now, decide what output format to use.
switch($output_type) {

    case 'text':
    case 'text/plain':
        header('Content-Type: text/plain');
```

```
        echo var_export($out, true);
        break;

    case 'json-comment-filtered':
    case 'text/json-comment-filtered':
        header('Content-Type: text/json-comment-filtered');
        echo '/*'.json_encode($out).'*/';
        break;

    case 'iframe-text':
        header('Content-Type: text/html');
        ?><html><body><textarea><?php
            echo htmlentities(var_export($out, true));
        ?></textarea></body></html><?php

    case 'iframe-json':
        header('Content-Type: text/html');
        ?><html><body><textarea><?php
            echo htmlentities(json_encode($out));
        ?></textarea></body></html><?php

    default:
        header('Content-Type: application/json');
        echo json_encode($out); break;

}
```

上面的代码期望在请求中发送 Accept:报头，或者在 URL 查询字符串中包含名为 type 的参数。这大致上对应于 dojo.xhr()接受的 handleAs 参数提供的选项：

- **text 或 text/plain**　以文本格式发送响应。
- **json-comment-filtered 或 text/json-comment-filtered**　使用“comment-filtered”JSON 格式。
- **iframe-text**　将文本响应包装到一个 HTML <textarea>元素中，我们将在本章结束时讨论该选项。
- **iframe-json**　将 JSON 响应包装到一个 HTML <textarea>元素中，我们将在本章结束时讨论该选项。

任何没有涵盖在上述列表中的选项均将产生 JSON 格式响应。无论如何，响应结果只是请求中收集到的数据点的快速转储(通过 PHP 函数 json_encode()或 var_export())。如果愿意的话，那么可以给出更美观的报告，但是这里的做法应该适用于大多数场合。

这个实用工具可用来尝试参数与 HTTP 方法的各种组合，响应结果则可以向我们提供在实际运行的服务器端应用程序代码中可能看到的细节。如果无法直接在 PHP 中运行该代码，那么希望您能够从该代码中获得一些认识(如果需要或希望在选择的服务器端环境中实现类似的功能的话)。

27.4.2 尝试多种请求方法

与其仅仅列出使用各种 HTTP 方法建立请求的代码示例，不如构建一个小型实验室来交互地尝试各种方法并查看服务器的响应，这样会更加有趣。

首先，考虑下面的标记：

```
<div id="xhr_methods">
    <h2>xhr methods</h2>

    <form>
        <button class="GET">GET</button>
        <button class="POST">POST</button>
        <button class="PUT">PUT</button>
        <button class="DELETE">DELETE</button>
        <button class="HEAD">HEAD</button>
        <button class="OPTIONS">OPTIONS</button>

        <br />

        <textarea class="results" cols="70" rows="25"></textarea>
    </form>

</div>
```

这里的代码提供了一个用户界面，它包含了几个用来试验 HTTP 方法的按钮，后面是一个<textarea>元素，用来显示请求接收到的响应结果。

接下来，查看 JavaScript 代码，它定义了一个用来驱动用户界面的类：

```
dojo.declare('decafbad.ajax.ajax.MethodPlay', null, {

    root_id: '',

    url: 'echo.php?type=text&wait=1',

    constructor: function(root_id) {
        this.root_id = root_id;
        this.wireUpButtons();
    },

    wireUpButtons: function() {
        dojo.forEach(this.methods, function(method) {
            dojo.query('.'+method, this.root_id)
                .onclick(this, 'handleButtonClick');
        }, this);
    },
```

这段代码声明了一个新类 MethodPlay。它的构造函数的参数为前面 HTML 标记的根 ID，构造函数将利用它把所有 HTTP 方法按钮与一个处理程序方法.handleButtonClick()连接起来。此外，

还有一个属性指向 echo.php 脚本的位置。

下面的代码定义了.handleButtonClick()方法：

```
handleButtonClick: function(ev) {
    var method = ev.currentTarget.className;

    dojo.stopEvent(ev);

    dojo.query('.results', this.root_id)
        .style('backgroundColor', '#ffd')
        [0].value = 'loading...';

    dojo.xhr(method, {
        url:      this.url,
        handleAs: 'text',
        load:     dojo.hitch(this, 'handleResponse'),
        error:    dojo.hitch(this, 'handleError')
    });
},
```

这个按钮单击事件处理程序方法首先使用目标按钮的 CSS 类名找出它应该使用哪一个 HTTP 方法。然后，它使用一条表明正在加载的消息来更新<textarea>显示元素，并将其背景颜色设为浅黄色(注意，之前指定的 echo.php 脚本 URL 有一个 wait=1 参数，这会导致这条加载消息有希望停留足够长的时间，以便能够看到它)。

这个方法最后调用 dojo.xhr()，并设置合适的 HTTP 方法和 URL。这里有一件有趣的事情需要注意，我们使用 dojo.hitch()将对象方法连接为响应的回调处理程序。与 Dojo 中的其他事件处理程序代码不同，这里不能直接使用对象引用和方法名，因此需要使用 dojo.hitch()实用工具提供一个包装器函数，以便满足 dojo.xhr()以及它所使用的 Deferred 对象的需要。

下面的代码定义了用来处理成功响应的.handleResponse()方法：

```
handleResponse: function(resp, io_args) {
    var out =
        "Response Headers:\n\n" +
        io_args.xhr.getAllResponseHeaders() + "\n" +
        "Response body:\n\n" +
        resp;
    dojo.query('.results', this.root_id)
        .style('backgroundColor', '#dfd')
        [0].value = out;
},
```

当接收到一个成功响应时，.handleResponse()方法收集一份响应报头以及正文内容的简单副本。然后，它将<textarea>元素的颜色切换到浅绿色，并将其内容替换成这份报告。

最后，在这个类定义的末尾定义了.handleError()方法，如下所示：

```
handleError: function(error, io_args) {
```

```
            dojo.query('.results', this.root_id)
                .style('backgroundColor', '#fdd')
                [0].value = error.message;
        }

});

var method_play = new decafbad.ajax.ajax.MethodPlay('xhr_methods');
```

最后这个方法用来在接收到响应失败消息时将<textarea>转换成浅红色，并显示错误提示消息。然后，在类声明的末尾，使用本节开头介绍的标记的根 ID 来创建该类的一个新实例。

图 27-4 显示了结果页面。

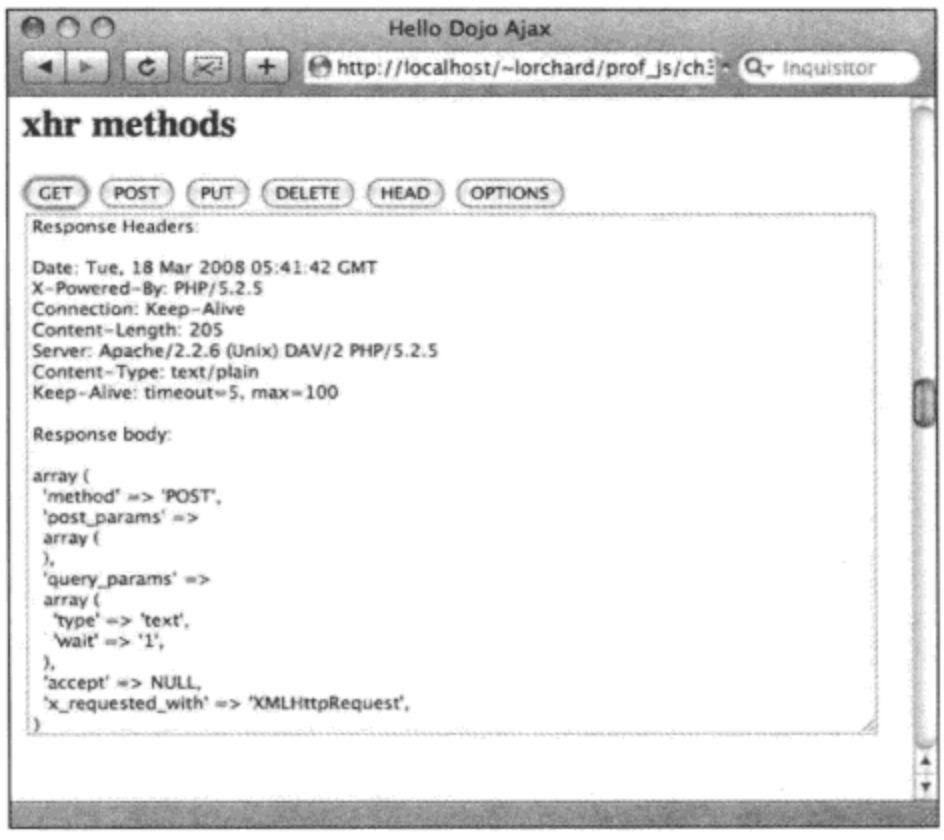


图 27-4

27.5 使用请求参数和内容

既然我们已经看到如何以各种方式建立请求和处理相应内容，并改变在请求中用到的 HTTP 方法，现在就来研究如何发送查询参数以及请求正文内容。与建立 HTTP 请求并处理响应的其他方面一样，Dojo 也提供了一些便利的功能，让我们能够更方便地传递请求中的数据。

27.5.1 建立带有查询参数的 GET 请求

将参数放入到请求中的最简单方式之一就是通过 URL 的查询字符串部分。您可能已经熟悉这种方式，而且我们在前面传给 echo.php 脚本的 wait 和 type 参数中已经见过这种方式。当然，这些参数是通过硬编码方式写入到请求 URL 中的，而且刚好不需要任何转义处理。

考虑下面的示例，它包含的参数的复杂程度要高得多：

```
dojo.xhrGet({
    url: 'echo.php?type=text',
    content: {
        alpha:     'one two three',
        beta:      '!@#$%^&*()',
        'splat[]': ['uno', 'dos', 'tres'],
        glob:      { what:1, who:2, where:23 }

    },
    handle: function(resp, io_args) {
        console.log("Response from xhrGet with request parameters:");
        console.log(resp);
    }
});
```

在上面的代码中，URL 本身已经包含了一个查询字符串参数。此外，在传入 dojo.xhrGet()的 content 参数下又提供了一组请求参数。注意，这些参数包含了带有空白符的较长字符串、需要 URL 编码的字符，甚至为了增加趣味，其中还有一个值数组。

在构造请求的过程中，Dojo 将使用一个名为 dojo.objectToQuery()的函数，根据 url 和 content 参数来建立下面的 URL：

```
echo.php?type=text&alpha=one%20two%20three&beta=!%40%23%24%25%5E%26*()
&splat%5B%5D=uno&splat%5B%5D=dos&splat%5B%5D=tres&glob=%5Bobject%20Object%5D
```

注意，参杂在这个参数混合体中的所有特殊情况都已经得到处理：空白符以及其他对 URL 而言相对奇异的字符都已经得到正确编码，而值数组已经转换成多个参数。值得注意的是，列表的命名(splat[])并非依照 Dojo 的约定，而是遵循 PHP 约定，它指示给定参数应该具有多个值。

但 dojo.objectToQuery()提供的功能还是有一定局限性的，最后一个参数 glob 呈现为%5Bobject %20Object%5D，这可能在一定程度上并不符合我们的预期。但这并不是 Dojo 的问题，因为这是使用 URL 查询字符串表达参数所带来的限制。

最后，运行上面的代码，获得成功的响应之后就应该在 Firebug 日志中看到如下的信息：

```
Response from xhrGet with request parameters:
array (
  'method' => 'GET',
  'post_params' =>
  array (
  ),
  'query_params' =>
  array (
    'type' => 'text',
    'alpha' => 'one two three',
    'beta' => '!@#$%^&*()',
    'splat' =>
    array (
```

```
        0 => 'uno',
        1 => 'dos',
        2 => 'tres',
      ),
      'glob' => '[object Object]',
    ),
    'accept' => NULL,
    'x_requested_with' => 'XMLHttpRequest',
  ),
```

根据本节的讲解，我们应该能够了解像 PHP 这样的服务器端语言如何接收并处理这些参数。

27.5.2 建立带有响应正文参数的 POST 请求

另一种将参数放进请求中的方式是通过 POST 请求中的正文内容。其实现方式与前一个示例类似，但将 dojo.xhrGet()替换成 dojo.xhrPost()，如下所示：

```
dojo.xhrPost({
    url: 'echo.php?type=text',
    content: {
        alpha:     'one two three',
        beta:      '!@#$%^&*()',
        'splat[]': ['uno', 'dos', 'tres'],
        glob:      { what:1, who:2, where:23 }
    },
    handle: function(resp, io_args) {
        console.log("Response from xhrPost with request parameters:");
        console.log(resp);
    }
});
```

为了将这个 dojo.xhrPost()调用与前面 dojo.xhrGet()的调用进行比较，查看在执行一次成功请求之后日志中显示的内容：

```
Response from xhrPost with request parameters:
array (
  'method' => 'POST',
  'post_params' =>
  array (
    'alpha' => 'one two three',
    'beta' => '!@#$%^&*()',
    'splat' =>
    array (
      0 => 'uno',
      1 => 'dos',
      2 => 'tres',
    ),
    'glob' => '[object Object]',
  ),
  'query_params' =>
```

```
    array (
      'type' => 'text',
    ),
    'accept' => NULL,
    'x_requested_with' => 'XMLHttpRequest',
    'content_type' => 'application/x-www-form-urlencoded',
    'content_length' => '127',
    'request_body' => 'alpha=one%20two%20three&beta=!%40%23%24%25%5E%26*()
      &splat%5B%5D=uno&splat%5B%5D=dos&splat%5B%5D=tres
      &glob=%5Bobject%20Object%5D',
)
```

首先注意到的是，尽管仍然有一个查询参数 type，但其余参数已经分开放入到 post_params 中。这是因为 PHP 将 URL 指定的参数和请求正文中传入的参数分别放入不同的集合中。

还要注意的是，这个请求现在有了正文内容，内容类型、长度以及正文数据的出现反映了这一点。正文数据本身看上去就像是 URL 中的查询字符串，这基本上就是 application/x-www-form-urlencoded 内容 MIME 类型的定义方式。

但是，这意味着参数编码技术也将受到与 GET 请求一样的限制。然而，由于在不同的浏览器中对 URL 长度的限制各有不同，而 POST 请求的正文长度不受限制，因此使用 POST 请求至少在这方面能给您带来更大的灵活性。

27.5.3　建立带有原始正文内容的 POST 请求

在了解普通的参数传递方式的限制之后，现在就查看一种用来改善这种情况的“不太常用的”方式。我们可以使用自己选择的编码将原始数据放在请求正文中发送。这些编码可以包括文本、XML 文档、二进制数据或 JSON；使用 dojo.rawXhrPost()调用，就像下面这样：

```
dojo.rawXhrPost({
    url: 'echo.php?type=text',
    headers: {
        'Content-Type': 'application/json'
    },
    postData: dojo.toJson({
        alpha: 'one two three',
        beta:  '!@#$%^&*()',
        splat: ['uno', 'dos', 'tres'],
        glob:  { what:1, who:2, where:23 }
    }),
    handle: function(resp, io_args) {
        console.log("Response from rawXhrPost with request parameters:");
        console.log(resp);
    }
});
```

这一次代码使用 dojo.toJson()将数据结构编码成一个 JSON 字符串，并通过 postData 参数传给 dojo.rawXhrPost()。该数据不必是 JSON 格式，但这个示例使用它与前面的两个示例进行了简易的

对比。

在 Firebug 日志中，当请求成功时应该出现如下的信息：

```
Response from rawXhrPost with request parameters:
array (
  'method' => 'POST',
  'post_params' =>
  array (
  ),
  'query_params' =>
  array (
    'type' => 'text',
  ),
  'accept' => NULL,
  'x_requested_with' => 'XMLHttpRequest',
  'content_type' => 'application/json',
  'content_length' => '113',
  'request_body' => '{"alpha":"one two three","beta":"!@#$%^&*()",
    "splat":["uno","dos","tres"], "glob":{"what":1,"who":2,"where":23}}',
)
```

尽管 echo.php 脚本并没有使用 json_decode()函数，但 PHP 5 提供了这个函数。在自己的脚本中可以使用这个函数解码 request_body 的内容。大多数现代 Web 应用程序环境都提供了类似的工具，因此 JSON 可以成为一种极佳的选择：它为数据结构从客户端 JavaScript 代码向服务器端代码的无缝传送提供了很好的机制，克服了普通 URL 编码查询字符串的各种限制。

最后，值得注意的是，伴随 dojo.rawXhrPost()的还有一个 dojo.rawXhrPut()函数，它的功能与前面的代码实现的功能一样，但使用的是 HTTP PUT 方法而不是 POST。它们的参数和使用模式完全相同。对于其他诸如 GET、DELETE 或 HEAD 这样的 HTTP 方法来说，它们并没有“rawXhr”对等方法，也不支持请求正文。

27.6 利用就地请求增强表单

对于获取专门为在客户端用户界面中使用而构建的服务器端资源(或者构建为 REST 形式的 Web 服务的服务器资源)来说，上述在发送给服务器的请求中获取参数和数据的所有方式都非常优秀。但现代 Web 开发的目标之一就是为页面引入一定程度的非侵入式增强，可以优雅地回退到非 AJAX 功能。

关于这一点，有些站点使用的一项技术就是构造一个表单，尽可能向服务器提交它的数据并就地显示结果，而不干扰整个页面。如果不能实现这种增强方式，那么该表单就会回退，重新加载整个页面。

为了预览 Dojo 如何帮助实现该机制，请查看下面的代码：

```
dojo.xhrPost({
    form:   dojo.query('form')[0],
```

```
    handle: function(resp, io_args){ ... }
});
```

这段代码的有趣之处在于 dojo.xhrPost()(实际上是所有围绕 dojo.xhr()包装的辅助函数)接受一个名为 form 的参数，该参数应该是一个 ID，或者是一个指向 DOM 中<form>节点的引用。

为此，Dojo 使用了一个名为 dojo.formToObject()的函数来遍历表单并将它包含的所有数据转换成一个 JavaScript 对象。这个过程会跳过按钮和禁用的表单字段，查找当前选中的复选框以及下拉列表项，并收集文本字段的内容。然后，将该对象传入 dojo.objectToQuery()进行处理，目的是获取 POST 请求的正文内容。此外，这个请求的 URL 是从表单的 action 属性中提取的。

下面的标记开始演示该功能：

```
<style>
    form li {
        list-style-type: none;
        line-height: 3.5ex;
    }
    form li.submit {
        margin-left: 6ex;
    }
    form label {
        display: block;
        text-align: right;
        width: 5ex;
        padding-right: 1ex;
        float: left;
    }
    .shown {
        display: block;
    }
    .hidden {
        display: none;
    }
</style>
```

这段标记为表单以及一个隐藏的窗格(将为表单提交操作提供就地反馈)提供了一些简单的 CSS 样式规则。上面的代码就是为下面示例中出现的标记而设计的：

```
<div id="xhr_request_form">
    <h2>xhr request form</h2>

    <div id="magic_form1">

        <form class="form_post" method="post"
              action="echo.php?wait=1&amp;type=text">
            <h3>magic ajax form #1</h3>
            <ul>
                <li>
                    <label for="foo">foo:</label>
```

```
            <input type="text" name="foo" value="Hello" size="20" />
        </li>
        <li>
            <label for="disabled_thing">disabled:</label>
            <input type="text" name="disabled_thing"
                value="Ignore me" disabled="true" size="20" />
        </li>
        <li>
            <label for="bar[]">bar:</label>
            <input type="checkbox" name="bar[]" value="world" /> world
            <input type="checkbox" name="bar[]" value="planet" /> planet
        </li>
        <li>
            <label for="baz">baz:</label>
            <select name="baz">
                <option value="one">Number one</option>
                <option value="two">Number two</option>
                <option value="three">Number three</option>
            </select>
        </li>
        <li class="submit">
            <input type="submit" value="submit" />
        </li>
    </ul>
</form>
```

这里构造的表单并不是太复杂，但它确实包含了一些典型的元素：一个文本字段、一个禁用的元素、一对复选框以及一个下拉式选择器。再次提请注意，表单字段名称 bar[]反映出一条 PHP 约定：它指示这个表单字段应该含有多个值。图 27-5 给出了这个表单的外观。

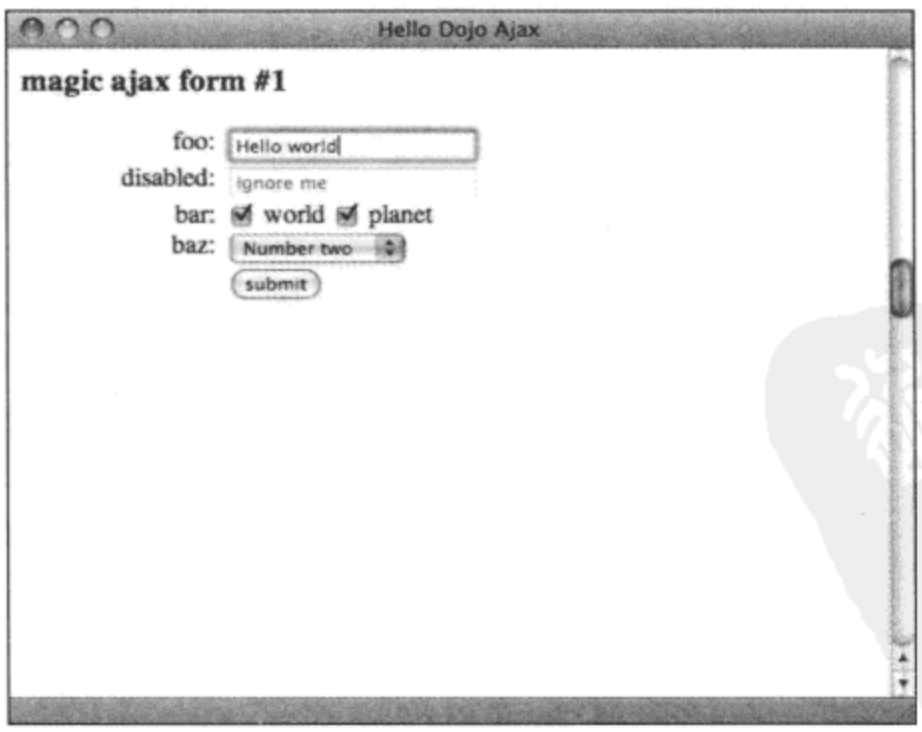


图　27-5

下面这部分代码起初会被隐藏，但它包含了一些用来报告表单提交情况所需的元素：

```
        <form class="form_results hidden">
            <h3>magic ajax form #1 results</h3>
            <a class="reset_form" href="#">click here to try the form again</a>
            <br />
            <textarea class="results" cols="70" rows="25"></textarea>
        </form>

    </div>

</div>
```

到这里为止，这个示例的所有标记均已经展示完毕。最后，有一个初始时隐藏起来的表单，它里面包含一个链接(当单击它时会把表单重置)以及一个<textarea>元素(响应的结果将插入到这个表单字段中)。

现在，开始研究为上面的标记实现功能的 JavaScript 代码：

```
dojo.require('dojo.NodeList-fx');

dojo.declare('decafbad.ajax.ajax.MagicForm', null, {

    root_id: '',

    constructor: function(root_id) {
        this.root_id = root_id;
        this.wireUpForm();
    },

    wireUpForm: function() {
        dojo.query('.form_post', this.root_id)
            .connect('submit', this, 'handleSubmit');
        dojo.query('a.reset_form', this.root_id)
            .onclick(this, 'resetForm');
    },
```

这段代码首先请求 dojo.NodeList-fx 模块，该模块将动画功能整合到 dojo.query()和 dojo.NodeList 系统中。接下来，开始声明一个名为 MagicForm 的类。这个类的构造函数带有一个参数：根 ID(例如 magic_form1)，该 ID 标识了一个容器，它将把处理程序连接到该容器以负责表单提交以及表单重置链接上的单击事件的处理。

接下来的代码定义了.handleSubmit()方法，它负责提交表单：

```
    handleSubmit: function(ev) {
        dojo.stopEvent(ev);

        // Disable the submit button while this is working.
        dojo.query('input[type="submit"]', this.root_id)
```

```
            .attr('value', 'loading...')
            [0].disabled = true;

        // Find the form, submit using XHR
        dojo.query('form.form_post', this.root_id)
            .forEach(function(form) {
                dojo.xhrPost({
                    form: form,
                    handle: dojo.hitch(this, 'displayResults')
                });
            }, this);
    },
```

.handleSubmit()方法首先使用 dojo.stopEvent()拦截表单提交事件并负责完成表单提交。注意这个 submit 事件既包括表单提交按钮上的单击事件，也包括用户在表单中的任何一个文本字段中按下 Enter 键时的事件。

接下来，该方法查找表单提交按钮，将其标签改为“loading...”，并将其设为禁用。这在大多数浏览器中会让该按钮以灰色显示，并阻止它被再次单击。

此后，该方法定位表单以及在指定表单上引发 dojo.xhrPost()调用，并且注册.displayResults()方法来处理所有响应。

下面的代码定义了.displayResults()方法：

```
    displayResults: function(result, io_args) {
// Hide the submitted form.
dojo.query('.form_post', this.root_id)
    .addClass('hidden').removeClass('shown');

var msg, clr;
if (result instanceof Error){
    // Display error message and flash red color.
    clr = '#f88'; msg = result.message;
} else {
    // Display response and flash green color.
    clr = '#8f8'; msg = result;
}

// Reveal the results pane container.
dojo.query('.form_results', this.root_id)
    .removeClass('hidden').addClass('shown');

// Inject the results into the results pane.
// .attr() doesn't seem to work with textareas
var pane = dojo.query('.results', this.root_id);
pane[0].value = msg;

// Make the textarea flash to attract attention.
```

```
        pane.animateProperty({
            duration: 750,
            properties: {
                backgroundColor: { start: clr, end: '#fff' }
            }
        }).play();
    },
```

.displayResults()方法并没有展示 Dojo 的更多功能。首先，当接收到响应时，该方法将该表单隐藏起来。然后，它判断该响应是一个错误消息或一条成功消息，然后挑选红色或绿色以及合适的消息。

此后，表单提交结果窗格变得可见。一旦可见，该窗格中的<textarea>元素的内容就会替换成错误消息或成功响应的正文。此外，<textarea>元素的背景颜色也会短暂地变成红色或绿色以指示错误或成功响应。

最后，下面的代码定义了.resetForm()方法，它将连接到表单重置链接：

```
    resetForm: function(ev) {
        // Show the form again.
        dojo.query('.form_post', this.root_id)
            .removeClass('hidden').addClass('shown');

        // Hide the results pane.
        dojo.query('.form_results', this.root_id)
            .addClass('hidden').removeClass('shown');

        // Re-enable the submit button.
        dojo.query('input[type="submit"]', this.root_id)
            .attr('value', 'submit')
            [0].disabled = false;

        dojo.stopEvent(ev);
    },

    EOF:null

});

var mf1 = new decafbad.ajax.ajax.MagicForm('magic_form1');
```

图 27-6 给出了成功表单提交之后的情形。在报告响应信息的同时，还会让表单重置链接变得可见。该链接的单击事件将由前面代码中定义的.resetForm()方法处理，这会让表单再次出现并隐藏结果窗格。它还会再次启用提交按钮，并将其标签改回到“submit”，这样就为再次尝试提交表单做好了重置工作。

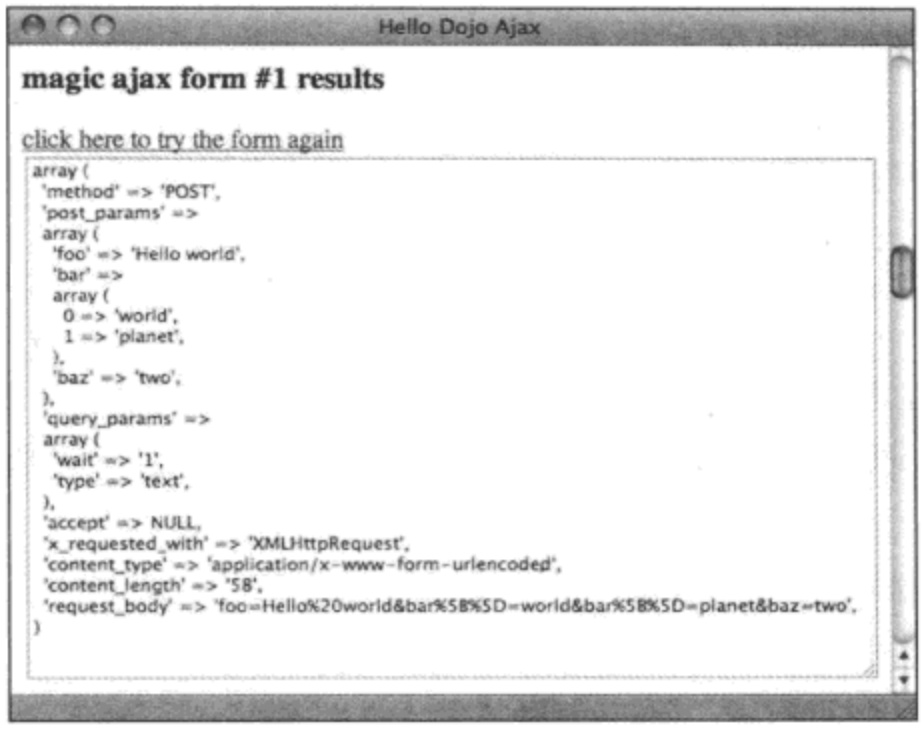


图　27-6

最后，在 MagicForm 类定义的末尾处，有一条语句使用 magic_form1 作为根 ID 来创建该类的一个新实例，从而完成了所有 JavaScript 代码并将标记连接起来。

虽然图27-6已经比较清晰，但在下面仍然给出了在这个示例请求中 echo.php 脚本返回的结果：

```
array (
  'method' => 'POST',
  'post_params' =>
  array (
    'foo' => 'Hello world',
    'bar' =>
    array (
      0 => 'world',
      1 => 'planet',
    ),
    'baz' => 'two',
  ),
  'query_params' =>
  array (
    'wait' => '1',
    'type' => 'text',
  ),
 'accept' => NULL,
 'x_requested_with' => 'XMLHttpRequest',
 'content_type' => 'application/x-www-form-urlencoded',
 'content_length' => '58',
 'request_body' => 'foo=Hello%20world&bar%5B%5D=world&bar%5B%5D=planet&baz=two',
)
```

现在，如果想知道这里的表单增强失败时会出现什么情况，那么可以试着注释上面示例中的最后一行代码，以阻止连接到该表单。此时提交表单，我们不会得到就地 Dojo 表单提交技术带来的好处，而是会被带到一个全新的包含如下输出信息的页面中：

```
array (
  'method' => 'POST',
  'post_params' =>
  array (
    'foo' => 'Hello world',
    'bar' =>
    array (
      0 => 'world',
      1 => 'planet',
    ),
    'baz' => 'two',
  ),
  'query_params' =>
  array (
    'wait' => '1',
    'type' => 'text',
  ),
  'accept' => NULL,
  'x_requested_with' => NULL,
  'content_type' => 'application/x-www-form-urlencoded',
  'content_length' => '56',
  'request_body' => 'foo=Hello+world&bar%5B%5D=world&bar%5B%5D=planet&baz=two',
)
```

需要注意的是，除了下面几个比较突出的区别之外，这个简单的表单提交与前面使用 dojo.xhrPost()调用的表单提交基本上是相同的：

- 未提供 X-Requested-With 报头。
- Content-Length 更简短。
- 前面已经预测到，正文内容本身更简短，这是因为浏览器选择使用“Hello+world”而不是 Dojo 选择的“Hello%20World”。这两种表示方式基本上是等同的，请求报告的剩余部分也指出了这一点。

这就是 X-Requested-With 报头比较有用的地方：例如，Web 应用程序框架可以进行某种自动化判断，如果没有发现该报头，就使用一种完整的页面模板；而如果检测到 X-Requested-With 报头的值为“XMLHttpRequest”，就生成一个更简单的标记片段。

总体而言，诸如本节中讲述的技术可以提供工具来构建令人愉悦的用户界面，同时在一些功能较弱的浏览环境中仍然能够工作。

27.7　使用跨域的 JSON 源

本章已经介绍了 Dojo 为了简化 AJAX 而围绕 XMLHttpRequest 对象所做的抽象以及提供的实用工具。但浏览器的 XMLHttpRequest 工具的一个突出限制就是，它仅限于向与加载当前页面所使用的域、协议和端口相同的组合建立请求。也就是说，假设页面地址为 http://example.com/foo/

bar.html，那么本章目前给出的代码将只限于基于 http://example.com 的 URL，而不能向 Web 的其他地方建立请求。

简而言之，这个限制的目的是为了将允许客户端代码访问 Web 所带来的风险加以约束。通过只允许 JavaScript 代码向它的原始服务器建立请求，基于浏览器的应用程序中的 JavaScript 代码向 Web 上的其他站点发动攻击(或者访问存放在防火墙和家用路由器后面的、仅供客户访问的信息)的可能性就会降低。

但这是一项过于严厉的限制措施，它限制了聚合服务以及跨服务集成的作用域。此外，通过代理可以突破这项限制。可以想象在原始 Web 服务器上放置一段 PHP 脚本，它接受某个 URL 作为请求参数，然后获取该 URL 的内容，接下来返回该内容——尽管这满足了同域限制，但它并非总是最安全或最便利的方法。

JSON 能够在这个领域发挥作用。JSON 实际上就是一种可执行 JavaScript 代码，而<script>标记不受跨域限制。因此，我们可以使用<script>标记从 Web 上的任何一个域加载 JSON 数据结构，唯一的技巧就是一旦 JSON 的惰性数组和对象字面值载入完毕，就执行某些操作。

Dojo 基本上为此提供了两种处理方式：轮询某个脚本加载变量，或者在加载时引发某个回调脚本。

27.7.1 通过轮询变量来加载 JSON

为了了解通过轮询某个变量是否存在能够处理什么类型的 JSON 源，请查阅如下 delicious.com 的帮助页面，其中描述了书签的 JSON 源：

```
http://delicious.com/help/json/posts
```

为了演示从这个源能够获取什么数据，查看下面给出的某个用户的书签列表：

```
http://feeds.delicious.com/feeds/json/deusx?raw
```

获取该 URL 应该产生一个类似下面的 JSON 数据结构：

```
[{"u":"http://blog.simonshea.com/2005/09/holly-shelf-unit-batman.html","n":"&quot;I
have always wanted to build a concealed room or have secret trapdoors or similar in
a house, well this is my first (very amateur) attempt.&quot;","d":"Per Vivere [To
Live]: Holly Shelf Unit, Batman!","t":["funny","home","projects","furniture"]},...]
```

这个源是一个对象字面值数组，每个对象字面值表示一个书签。每个对象字面值均提供如下的属性：

- u 该书签的 URL
- d 该书签的标题
- n 该书签的扩展备注
- t 附加到该标签的标记数组

最好能够使用 dojo.xhrGet()调用(参数 handleAs 的值为 json)加载这段数据并进行分析，但把我们的代码托管到 delicious.com 域中的可能性非常小。

因此，试着将?raw 参数从 URL 中删除，此时从该源那里获取的内容将发生改变，它包含了一个类似下面的包装器：

```
if(typeof(Delicious) == 'undefined') Delicious = {}; Delicious.posts = [...]
```

这里的重要之处在于变量赋值 Delicious.posts = [...]。当把这个 JSON 源包含到一个<script>标记中时，变量 Delicious.posts 将初始化为这个源中包含的数据。

换言之，只要之前没有定义 Delicious.posts 变量，那么在加载该脚本之前该变量将是未定义，而在加载脚本之后会变成已定义，因此它为我们提供了一种检测数据已经加载完毕的机制。这就是 Dojo 的第一种 JSON 加载技术的工作原理。

为了进行演示，请考虑下面的标记：

```
<h3>delicious posts (checkString)</h3>
<ul class="checkstring_posts"></ul>
```

现在请考虑下面的代码，它加载该源并观察 Delicious.posts 变量：

```
dojo.require('dojo.io.script');

// Update this with your delicious.com username
var del_user = 'deusx';

// Establish the Delicious namespace, so checkString works with Delicious.posts
Delicious = {};

dojo.io.script.get({
    url: 'http://feeds.delicious.com/feeds/json/' + del_user,
    checkString: 'Delicious.posts',
    handle: function(io_args) {

 var list = dojo.query('.checkstring_posts', 'xhr_request_script');
        dojo.forEach(Delicious.posts, function(post) {
            list.addContent(
                '<li>'+
                '<a href="' + post.u + '">' + post.d + '</a>'+
                ( (post.n) ? '<br /><span>' + post.n + '</span>' : '' ) +
                '</li>'
            );
        });

        delete Delicious.posts;
    }
});
```

首先，在上面的代码中是一个用来加载 dojo.io.script 模块的 dojo.require()调用。这个模块提供了通过<script>标记来加载和处理 JSON 源所需的功能。

在该模块中，dojo.io.script.get()方法的使用类似于实用工具 dojo.xhrGet()的使用，它们都接受下面的参数：

- **url**　将要作为 JSON 源脚本加载的 URL

- **handle** 当请求的脚本已经加载完毕时调用的函数

但是，驱动这个脚本加载技术的是下面这个参数：

- **checkString** 这个参数的值指定了一个全局名称空间变量，当 URL 载入到<script>标记中时将初始化该变量。

在调用 dojo.io.script.get()之后，Dojo 开始定期检查该变量(在上面的代码中，该变量名为 Delicious.posts)。这些检查中使用的测试的实现类似于下面的代码：

```
eval("typeof(" + checkString + ") != 'undefined'")
```

这意味着一旦指定的变量变得具备某种类型的定义，那么该测试将变成 true，并调用 handle 函数。

这个具体的实现细节非常重要：JSON 数据可能完全由 false 值构成，而仍然会调用 handle 回调。需要记住的另一点是，如果希望加载另一个同样类型的源，那么就必须使用 delete 语句将该变量清除。

上面代码中实现的处理程序回调只接受一个参数 io_args，它与 dojo.xhr()所使用的 io_args 参数类似。此外还要注意的是，dojo.io.script.get()调用返回了一个 Deferred 对象，可以按照讲解 dojo.xhrGet()时描述的方式来使用该对象。

在该回调内部，dojo.forEach()循环直接使用 Delicious.posts 变量。这个循环利用在源中找到的书签的 HTML 呈现来生成一个列表。请注意，按照前面给出的建议，在处理完数据之后针对 Delicious.posts 变量使用了一条 delete 语句，这样就可以重用该代码来获取更多源。

图 27-7 给出了运行这段代码的示例结果。

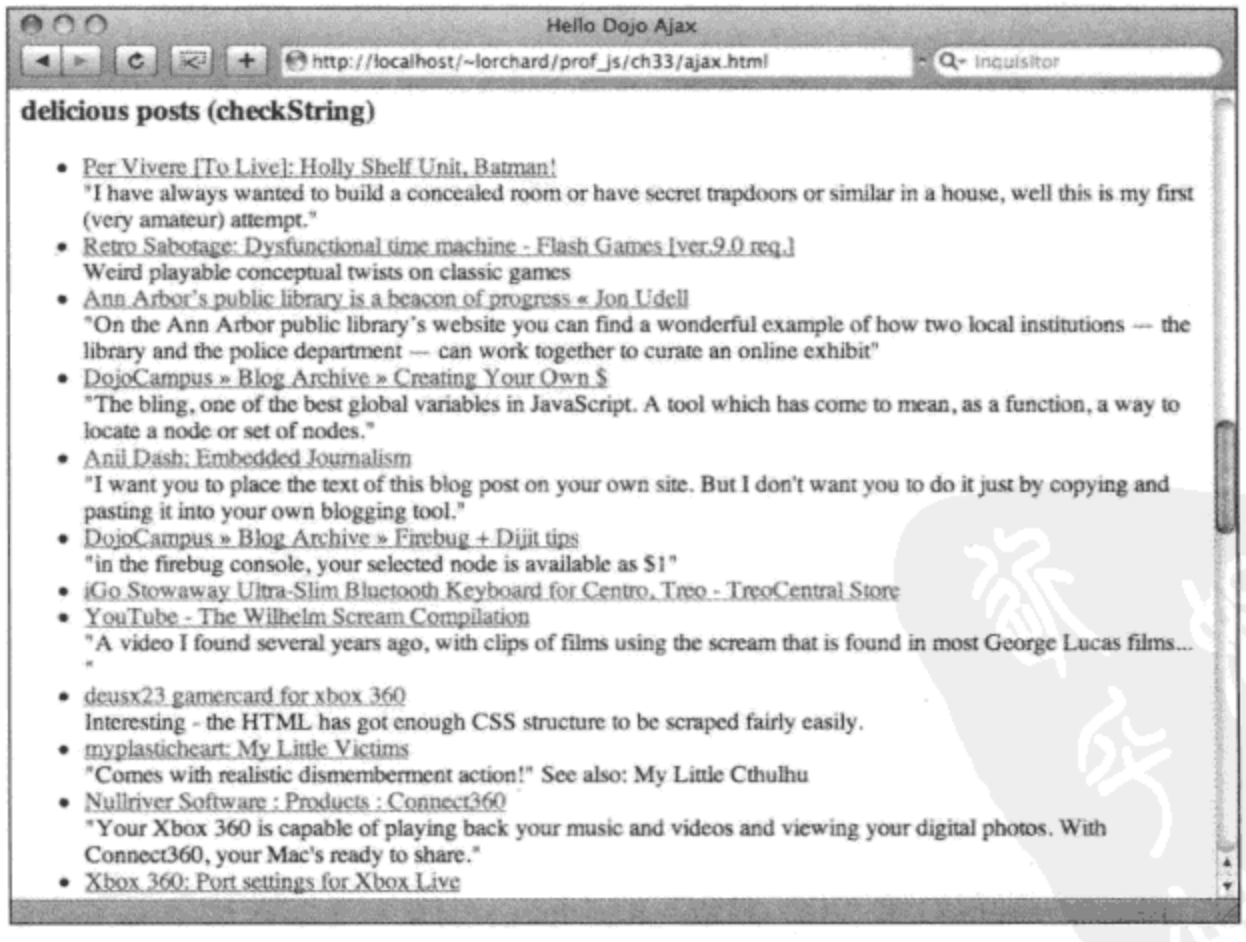


图 27-7

27.7.2　利用回调加载 JSON

不断地观察一个变量是否出现，然后引发一个回调函数，这是处理<script>标记加载的一种方式。但更好的方式是让加载脚本直接引发一个回调，这项技术称为 JSONP(带有填充的 JSON，JSON with padding)，在下面的链接中可以找到更多有关它的内容：

```
http://bob.pythonmac.org/archives/2005/12/05/remote-json-jsonp/
```

基本思想是向 JSON 源传递一个参数来指定一个本地回调函数。通过调用这个函数包装 JSON 数据，因此在加载时引发该回调。

作为示例，请查看 delicious.com 的另一个 JSON 源帮助页面：

```
http://delicious.com/help/json/tags
```

下面的 URL 是这个页面上描述的源的一个示例实例：

```
http://feeds.delicious.com/feeds/json/tags/deusx?raw
```

这个 URL 中的数据将与下面的 JSON 结构类似，它表示用户的标记使用计数：

```
{"webdev":1051,"funny":971,"gaming":947,"politics":777,"nifty":472,"osx":435,
"music":389,"metablogging":339,"odd":331}
```

关于 JSONP 模式，对于这个源可供使用的参数之一是指定的?callback。在上面的 JSON URL 中，将?raw 替换成?callback=display_tags 将产生下面的结果：

```
display_tags({"webdev":1051,"funny":971,"gaming":947,"politics":777,"nifty":472,
"osx":435,"music":389,"metablogging":339,"odd":331})
```

注意，JSON 数据结构已经附加了 display_tags 和一个圆括号作为前缀。在该数据的另一端还有另一个结束圆括号，这样就包括了该数据，从而完成了函数调用包装器。这意味着当使用一个<script>标记来载入该 URL 时，就会调用全局函数 display_tags()，并将这个 JSON 数据结构作为其参数。

Dojo 还提供了 dojo.io.script.get()的另一个变体来简化上面的使用场景。下面的标记为快速演示建立了基础：

```
<h3>delicious tags (callback)</h3>
<ul class="callback_tags"></ul>
```

接下来，下面的代码再次调用 dojo.io.script.get()：

```
dojo.require('dojo.io.script');

// Update this with your delicious.com username
var del_user = 'deusx';

dojo.io.script.get({
    url: 'http://feeds.delicious.com/feeds/json/tags/' + del_user,
    callbackParamName: 'callback',
    content: {
```

```
            count: 10,
            sort: 'count'
        },
        handle: function(resp, io_args) {
            var list = dojo.query('.callback_tags', 'xhr_request_script');
            for(tag in resp) {
                list.addContent(
                    '<li><a href="http://del.icio.us/deusx/'+tag+'">' +
                    tag + ' (' + resp[tag] + ')' +
                    '</a></li>'
                );
            }
        }
    });
```

与 dojo.xhr()调用类似，这里接受的参数包括 url、handle 和 content。在上面的代码中，content 用来向 JSON 源传递几个额外的查询参数，这会生成一个由前 10 个最常使用的标记构成的列表。

下面是启用回调的参数：

- **callbackParamName** URL 接受的这个参数的名称用于通过一个函数调用包装 JSON 数据。

这个参数让 dojo.io.script.get()与 dojo.xhr()调用的工作方式类似。Dojo 将自动地创建和管理一个回调函数，并使用 callbackParamName 指定的查询字符串参数将其传入请求。这个托管的回调调用内联定义的 handle 函数，但由于 dojo.io.script.get()也返回一个 Deferred 对象(就像 dojo.xhr()调用一样)，托管回调实际上是通过引发在这个 Deferred 对象上注册的适当成功和错误回调来实现的。

在上面的示例中，回调处理程序处理返回的标记列表，将每个标记呈现为一个列表项。与前面的变量轮询示例不同的是，这个方法不需要使用 delete 语句，因为没有涉及任何临时的全局变量。

图 27-8 给出了这段代码的运行示例。

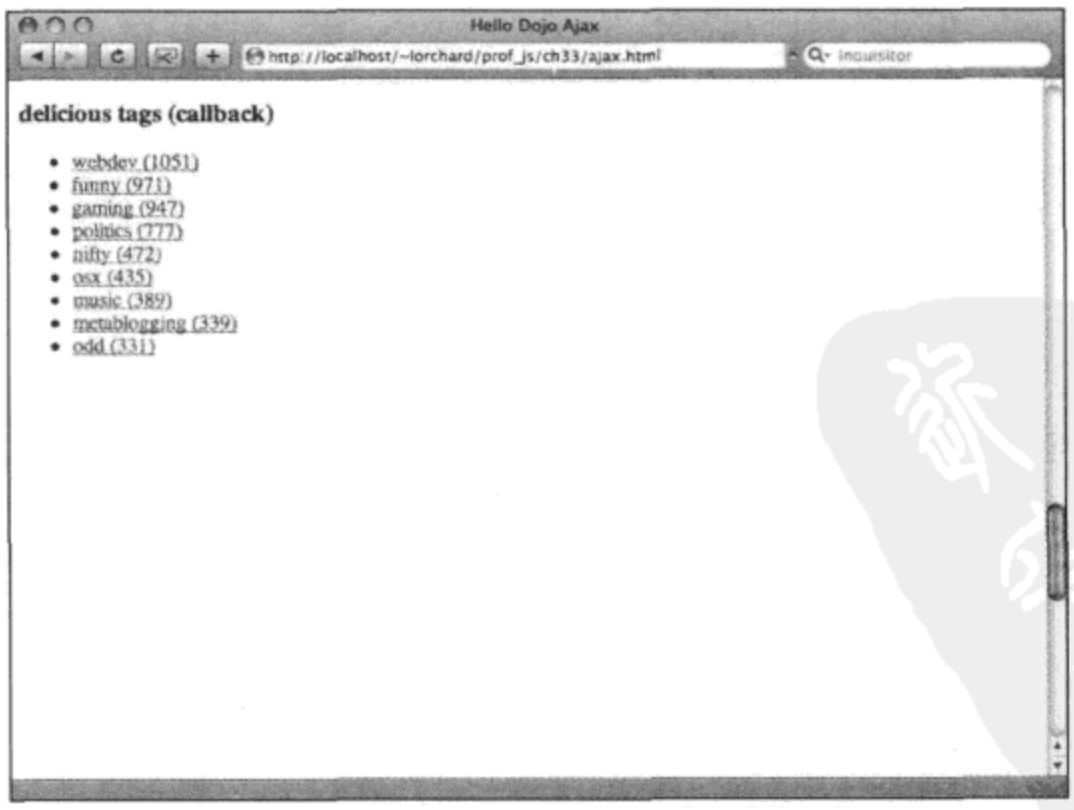


图 27-8

27.8　利用 IFrame 建立请求

当 XMLHttpRequest 对象在大多数现代浏览器中变得广泛可用之前，还有几种在页面加载之后向服务器建立请求的替代方式。其中之一就是使用<iframe>，使用 CSS 将其从视图中隐藏。这个不可见的<iframe>可通过它的 src 属性来方便建立 HTTP GET 请求，甚至可以通过成为表单提交的目标来帮助执行 HTTP POST 请求。

在大多数情况下，只要 XMLHttpRequest 对象可用，就不会使用隐藏的<iframe>元素。<iframe>不能绕开跨域限制，这是因为 JavaScript 不能访问从另一个域加载的<iframe>内容。此外，<iframe>并不能真正有助于执行采用除 GET 和 POST 之外的其他 HTTP 方法的请求。

但是，除了为 XMLHttpRequest 对象提供了一个最具功能性的后备技术之外，至少还有一个场合只能使用<iframe>技术：由于<iframe>可以作为表单提交的目标，因此它能够帮助增强包含带有就地反馈(以避免页面重新加载)的文件上传字段的表单。在这种场合中，只使用 XMLHttpRequest 对象并不能实现所有功能，这是因为 JavaScript 代码不能直接访问文件上传字段的内容。这对于像在电子公告栏或论坛中上传个人头像图像这类的操作来说非常有用。

因此，在了解这一点之后，本节将介绍 Dojo 对利用隐藏<iframe>元素建立请求的支持，其目的是为了尽可能像 dojo.xhr()那样工作。

27.8.1　利用代理脚本打包 IFrame 数据

使用 Dojo 的 IFrame 支持至少要面对一个主要问题：在<iframe>中不能可靠地处理各种格式的原始数据，因此 Dojo 要求将该数据包装到只含有一个<textarea>元素的 HTML 页面中。

再次依靠 PHP，考虑下面的脚本(名为 iframe_proxy.php)，我们使用它来讲解这种打包方案：

```
<?php
$path = $_GET['path'];
if (realpath($path) != realpath('./'.$path)) {
    header('Status: 403 Forbidden'); exit;
}
?>
<html><body><textarea><?php
    echo htmlentities(file_get_contents($path));
?></textarea></body></html>
```

这段代码接受一个名为?path 的查询字符串参数，它指定了一个位于相同目录下的文件作为脚本(由脚本起始处的 realpath()条件实施的要求)。

利用 file_get_contents()读入?path 指定的文件，并传给 htmlentities()以使数据得到适当的处理，以便能够包含到 HTML 中。然后使用一个简单的 HTML 页面中的<textarea>元素将其包装起来。

因此，试着获取 iframe_proxy.php?path=data.json 将得到下面的结果：

```
<html><body><textarea>{&amp;quot;url&amp;quot;:&amp;quot;http:\/\/decafbad.com&amp;quot;,&amp;quot;title
&amp;quot;:&amp;quot;0xDECAFBAD&amp;quot;}</textarea></body></html>
```

通过使用这种数据打包方案，Dojo 能够以一种一致的、可预测的方式来接收<textarea>内容并适当地处理选中的数据格式。

但上面的讲解有一个例外情况，就是使用 handleAs 参数值 html，此时<iframe>的 HTML DOM 本身也会作为响应传递，这样就可以通过 dojo.query()和 DOM 遍历执行任意的自定义处理。

27.8.2 利用 IFrame 处理响应格式

Dojo 试图让<iframe>元素的处理尽可能与 dojo.xhr()的处理相似。因此，它为 handleAs 参数提供了几个选项。下面的代码用来练习 Dojo 的<iframe>抽象模块：

```
dojo.require('dojo.io.iframe');

var resources = [
    [ 'data.txt',  'text' ],
    [ 'data.json', 'json' ],
    [ 'data.js', 'javascript' ],
    [ 'data.xml', 'html' ]
];

dojo.forEach(resources, function(resource) {
    var url       = resource[0];
    var handle_as = resource[1];

    var dfd = dojo.io.iframe.send({
        url: 'iframe_proxy.php',
        content: {
            path: url
        },
        handleAs: handle_as,
        handle: function(resp, io_args) {
            console.log("Response via an iframe, handled as " + handle_as + ":");
            console.log(resp);
        }
    });

});
```

上述代码中的第一行载入 dojo.io.iframe 模块。该模块提供 dojo.io.iframe.send()实用函数，我们稍后将用到该函数。

接下来的部分代码构建一个由多个列表构成的列表，每个子列表包含一个 URL 和一个 handleAs 值。这是 dojo.xhr()可用选项的一个子集，包括如下选项：

- text　作为普通文本内容进行处理。
- json　作为 JSON 数据分析成一个 JavaScript 对象。
- javascript　作为 JavaScript 代码执行。
- html　返回载入的<iframe>元素的 DOM 结构。请注意，这并不与 XML 分析相同，直接将 XML 文档载入到<iframe>中可能会产生混合结果。

在定义了这个列表之后，使用一个 dojo.forEach()循环来处理每个子列表。调用 dojo.io.iframe.send()函数，它返回一个 Deferred 对象(就像 dojo.xhr()一样)，而且接受类似的参数，包括 url、content、handleAs 和 handle。

在调用时，dojo.io.iframe.send()函数首先确保页面中有一个隐藏的<iframe>元素可用，既可以动态地向文档中插入一个<iframe>元素，也可以找出之前已经插入的<iframe>元素。然后，它将该请求放入队列中；尽管 Dojo 只使用一个<iframe>元素实现这种抽象，但它确实维护了一个由等待使用它的请求构成的内部队列。

当请求有机会让<iframe>元素提交时，Dojo 修改该元素的 src 属性来建立 GET 请求。如果该请求涉及使用表单，那么该表单的目标将修改为指向该<iframe>元素，并通过编程方式提交 POST 请求。不管是哪种情况，Dojo 都会观察<iframe>元素是否完成加载，以获取该请求的响应并引发适当的 Deferred 成功或错误回调分支。再次提请注意，如果使用了 handle、load 或 error 参数，那么内联定义的回调将自动注册为 Deferred 对象上的处理程序。

在运行上面的代码时，应该会看到类似下面的消息出现在 Firebug 日志文本消息中：

```
Response via an iframe, handled as text:
Hello world, this is some text available for load.
Response via an iframe, handled as json:
Object url=http://decafbad.com title=0xDECAFBAD
Executable JavaScript ahoy!
Response via an iframe, handled as JavaScript:
Hello! Today is Sun Mar 16 2008 21:53:31 GMT-0700 (PDT)
Response via an iframe, handled as HTML:
Document iframe_proxy.php
```

这应该能够让您快速地了解如何使用隐藏的<iframe>元素以及如何处理这类请求返回的结果。下一节将介绍<iframe>元素与使用 XMLHttpRequest 对象相比的主要优点之一。

27.8.3　利用表单和 IFrame 上传文件

本章前面曾经演示过，可以调用实用函数 dojo.xhrPost()并在该函数的 form 参数中指定<form>元素的引用或 ID 来模拟表单提交。

在大多数情况下，这种技术运行良好。但至少有一种表单字段不能从 JavaScript 中有效地读取或操作：文件选择输入字段。我们能够获取选中文件的文件名，但之后无法从本地文件系统读取文件内容，如果不能将文件数据上传到服务器，那么使用文件选择字段的目的何在？

这正是使用<iframe>元素进行 AJAX 形式请求的真正亮点所在：尽管不能从 JavaScript 那里访问选中文件的内容，但是可以将表单的目标属性指向隐藏的<iframe>元素并通过编程方式提交该表单。因此，它避免了整个页面的重新加载，而能够通过浏览器的普通表单提交机制将选中文件的内容上传到服务器。此外，一旦隐藏的<iframe>元素中的表单提交已经完成，我们就可以处理从其中接收到的响应。

为演示这项技术做好准备，请查看下面的标记：

```
<div id="magic_upload_form">

    <form class="form_post" method="post"
          enctype="multipart/form-data" action="upload_avatar.php">
        <h3>magic upload form</h3>
        <ul>
            <li>
```

```
                <label for="nickname">nickname:</label>
                <input type="text" name="nickname" value="biff" size="20" />
            </li>
            <li>
                <label for="avatar">avatar:</label>
                <input type="file" name="avatar" size="20" />
            </li>
            <li class="submit">
                <input type="submit" value="submit" />
            </li>
        </ul>
    </form>

    <form class="form_results hidden">
        <h3>magic upload form results</h3>
        <a class="reset_form" href="#">click here to try the form again</a>
        <br />
        <div class="results"></div>
    </form>

</div>
```

上面的标记应该看起来非常熟悉：它基本上与我们在前面的就地表单提交示例中提供的标记相同。表单字段替换成一个文本字段和一个文件选择字段，而用来显示文本结果的<textarea class= "results">元素现在变成了一个 元素。

此外，请注意<form>元素上的属性 enctype="multipart/form-data"。一般而言，为了启用文件上传功能，必须设置该属性，而忽略该属性是一个非常普遍的错误。图 27-9 给出了这段标记的运行情况。

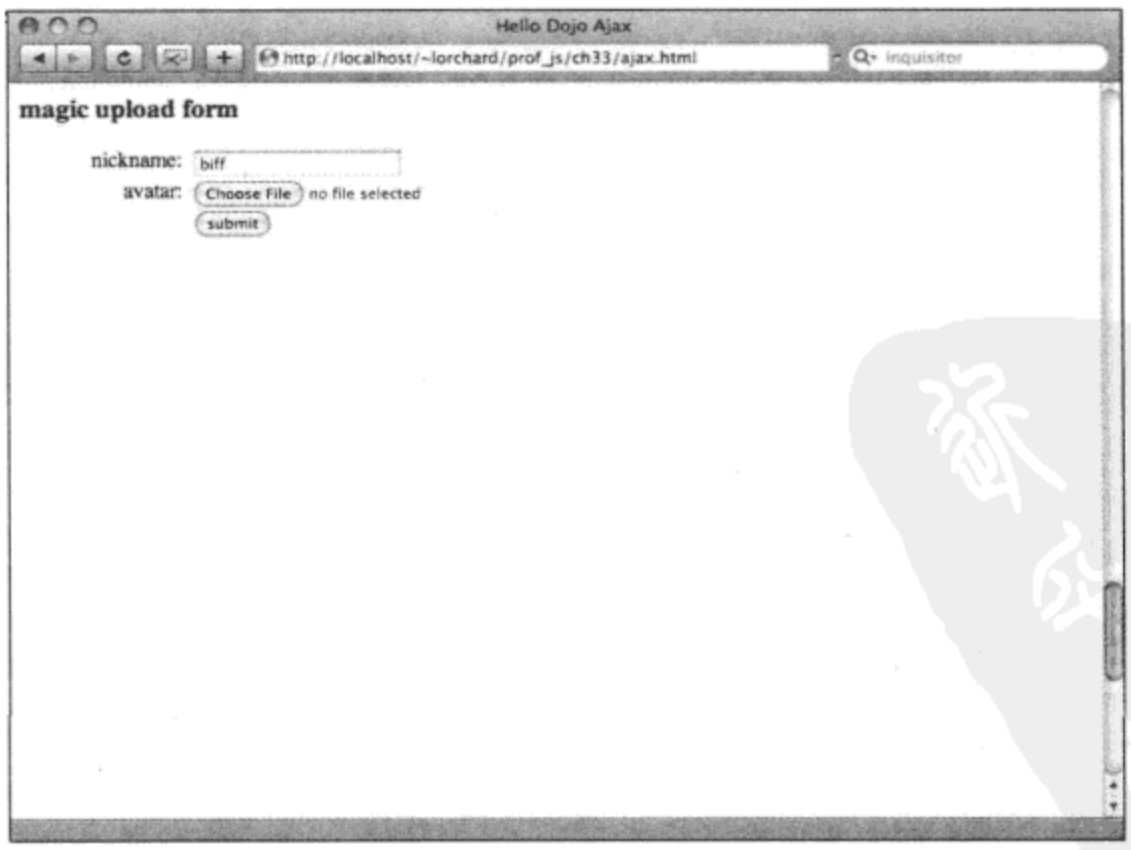


图 27-9

现在，在深入研究用来处理前面示例标记的 JavaScript 代码之前，更深入地研究表单操作的处理脚本(upload_avatar.php)应该会对我们有所帮助。下面的代码提供了该脚本的第一部分，它用来接受论坛头像图像或个人照片的上传。

```
<?php
// Find the location of this script and the uploads
// directory relative to it.
$BASE_DIR    = dirname(__FILE__);
$uploads_dir = $BASE_DIR . '/uploads';

// Determine if upload is of acceptable type and decide on
// extension, or decide if the type is not acceptable.
switch($_FILES['avatar']['type']) {
    case 'image/gif':
        $avatar_ext = 'gif'; break;
    case 'image/jpeg':
        $avatar_ext = 'jpeg'; break;
    case 'image/png':
        $avatar_ext = 'png'; break;
    default:
        $avatar_ext = FALSE; break;
}

if (!$avatar_ext) {

    // Since no acceptable type was found for the upload,
    // report that it was unacceptable.
    $out = array(
        'error' => true,
        'message' => "Avatar upload ".
            $_FILES['avatar']['name'].
            " is not an acceptable image type"
    );
```

这段脚本假设在它所处的同一个路径下有一个名为 uploads/的目录，而且该脚本具有这个目录中的文件写入权限。如果想移动这个目录，那么请注意这段脚本剩余部分对该目录的引用。

接下来，检查浏览器报告的文件上传内容的 MIME 类型，由表单字段 avatar 提交该文件上传。该脚本要求该类型匹配如下 3 种著名的图像类型：

- image/gif
- image/jpeg
- image/png

对照这些著名的图像类型，该脚本选取文件扩展名。如果上传的文件类型未能匹配这些类型中的任何一种，那么不选择任何扩展名，而是构造一条报告上传遭到拒绝的错误消息。

如果这一次上传通过上述代码的严格检验，那么下面就要实际地对文件数据进行操作：

```
} else {

    // Get the chosen nickname and come up with a safe
```

```
    // filename for the image based on the nickname.
    $nickname  = $_REQUEST['nickname'];
    $avatar_fn = 'avatar-'.md5($nickname).".".$avatar_ext;

    // Move the uploaded file into an appropriately named file
    // under the uploads directory.
    $rv = move_uploaded_file(
        $_FILES['avatar']['tmp_name'],
        $uploads_dir . '/' . $avatar_fn
    );

    if (!$rv) {
        // Report an error if the upload move failed.
        $out = array(
            'error' => 'Could not process avatar upload.'
        );
    } else {
        // Report details of success if the move worked.
        $out = array(
            'nickname' => $nickname,
            'avatar_fn' => $avatar_fn
        );
    }

}
```

为了给上传的图像挑选一个相对安全的文件名，这里使用了选中的别名的 MD5 散列值。这个值总是一个长度为 32 个字符、由字母和数字组成的字符串，它来源于用户提供的 nickname 字段，但绝不包含任何可能会影响到文件路径的危险字符(例如“.”和“/”)。这还带来了让数据库标识符变模糊(而不是选择使用人员的昵称作为标识符)的额外好处。除了基于别名的文件名之外，该代码还从 MIME 类型中选取文件扩展名附加到最终的文件名末尾。

接下来，使用 PHP 函数 move_uploaded_file()将上传的图像文件从一个临时位置移到位于选中文件名下的正确目录中。如果该操作执行成功，就会准备好一条报告消息，内容包括别名以及为此次上传构造的文件名。如果文件移动失败，那么就会准备好一条错误消息来报告该情况。

最后，实际产生一些输出信息：

```
switch($_REQUEST['type']) {

    case 'iframe-json':
        ?><html><body><textarea><?php
            echo htmlentities(json_encode($out))
        ?></textarea></body></html><?php
        break;

    default:
        ?><html>
            <body>
                <?php if ($out['error']): ?>
                    <h1>Error accepting avatar upload:</h1>
```

```
                <p><?php echo htmlentities($out['error']) ?></p>
            <?php else: ?>
                <h1>Avatar upload accepted</h1>
                <dl>
                    <dt>Nickname</dt>
                    <dd>
                        <?php echo htmlentities($out['nickname']) ?>
                    </dd>
                    <dt>Avatar</dt>
                    <dd>
                        <img src="uploads/<?php echo $out['avatar_fn'] ?>" />
                    </dd>
                </dl>
            <?php endif ?>
        </body>
    </html><?php
    break;

}
```

根据可选的表单字段 type 的值，上面的代码划分为两个主要的输出模板。如果该值为“iframe-json”，就采用 Dojo 所需的<iframe>数据打包格式将输出变量的简单 JSON 编码值包装起来。否则，就会根据输出变量来构造一个完整的 HTML 页面。

此时，给定标记和 upload_avatar.php 脚本，您可能希望尝试执行，查看会发生什么情况。由于尚未有 JavaScript 或 AJAX 的参与，因此上面描述的这些代码在运行时就是一个接受给定别名的图像上传的系统。图 27-10 给出了当给定一个有效的图像文件时上述脚本产生的结果。

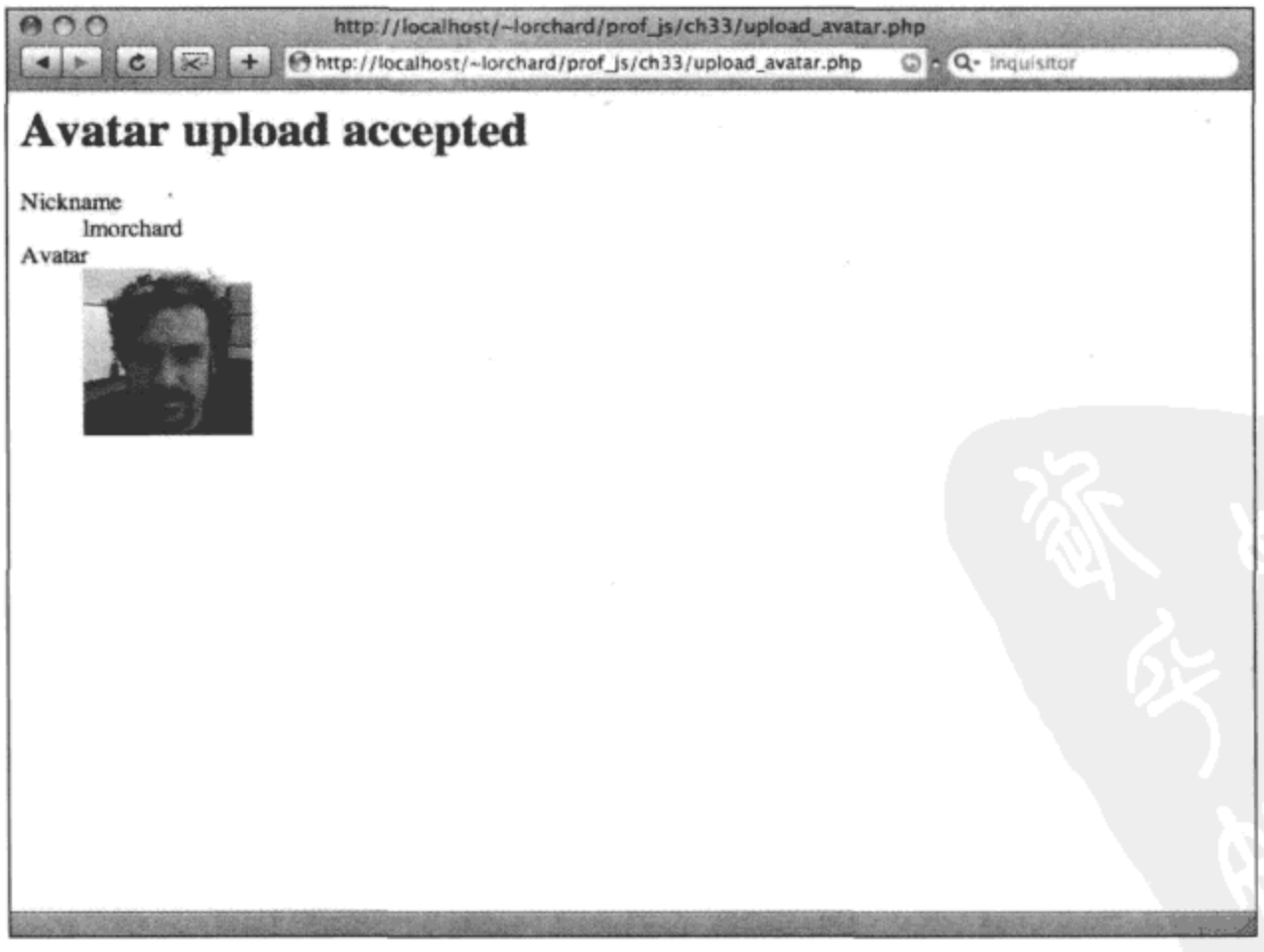


图　27-10

请注意图 27-9 和图 27-10 之间的区别：地址栏中的 URL 发生了变化。这是由于在普通表单提交中加载了全新的页面。

但是，我们可以通过某种方式增强用户界面，在不离开页面的情况下就地反映表单提交和文件上传的结果。下面的代码就是这种努力的开始：

```
dojo.require('dojo.io.iframe');

dojo.declare('decafbad.ajax.ajax.MagicUploadForm', decafbad.ajax.ajax.MagicForm, {

    handleSubmit: function(ev) {
        dojo.stopEvent(ev);

        // Disable the submit button while this is working.
        dojo.query('input[type="submit"]', this.root_id)
            .attr('value', 'loading...').attr('disabled', true)

        // Find the form, dig up its method and URL, submit using XHR
        dojo.query('form.form_post', this.root_id)
            .forEach(function(form) {
                dojo.io.iframe.send({
                    form: form,
                    content: {
                        type: 'iframe-json'
                    },
                    handleAs: 'json',
                    handle: dojo.hitch(this, 'displayResults')
                });
            }, this);
    },
```

上面的代码声明了一个名为 MagicUploadForm 的类，这是本章前面声明的 MagicForm 类的一个子类。

在这个子类中，首先要做的工作是替换.handleSubmit()方法：这里调用函数 dojo.io.iframe.send()，传入头像上传表单的引用，同时还使用另一个参数 content 将类型字段设为“iframe-json”。如果回顾 upload_avatar.php 脚本的另一个输出分支，那么可以知道这个值将导致产生 Dojo 打包格式的 JSON 响应，借助参数 handleAs 值“json”可以将其适当地分析成一个 JavaScript 数据结构。

这里有一点非常重要，尽管向 dojo.io.frame.send()方法传递了一个表单引用，但我们仍然可以指定额外的表单字段，将它们与该表单的表单字段一起包含到请求中。为了维护表单提交的一致性，该实用工具函数在提交请求之前将为每个额外指定的字段创建并插入一个隐藏的表单字段，跟踪这些插入的表单字段，然后在表单提交获得响应之后将这些表单字段移除。

为了获取该响应，dojo.io.frame.send()调用修改表单的 target 属性，将其指向托管的隐藏<iframe>元素，然后通过编程方式来提交该表单。至此，事情一般就会按照我们在目前给出的 dojo.io.frame.send()示例中看到的那样发展。但是，这种方式带来的好处是：由于提交了整个表单，因此允许像以往那样使用 avatar 文件选择字段上传它的数据。

当接收到响应时，下面的.displayResults()函数将处理它。

```
displayResults: function(result, io_args) {

    // Hide the submitted form.
    dojo.query('.form_post', this.root_id)
        .addClass('hidden').removeClass('shown');

    var msg, clr;
    if (result instanceof Error || result.error){
        // Display error message and flash red color.
        clr = '#f88';
        msg = '<p>' + result.message + '</p>';
    } else {
        // Display response and flash green color.
        var avatar_url = result.avatar_fn + '?' +
            (new Date()).valueOf();
        clr = '#8f8';
        msg = '<img src="uploads/' + avatar_url + '" />';
    }

    // Update the results content.
    dojo.query('.results', this.root_id)
        .empty().addContent(msg);

    // Reveal the results pane container.
    dojo.query('.form_results', this.root_id)
        .removeClass('hidden').addClass('shown')
            .animateProperty({
                duration: 750,
                properties: {
                    backgroundColor: { start: clr, end: '#fff' }
                }
            }).play();
    },

    EOF:null

});

var mf2 = new decafbad.ajax.ajax.MagicUploadForm('magic_upload_form');
```

这个实现与之前使用的 MagicForm 实现的不同之处在于，该实现并不只是更新<textarea>。相反，如果头像图像文件上传成功，该方法就会插入一个可以反映该上传文件的<img>元素。这样一来，如果在 upload_avatar.php 中移动了 uploads/目录，就需要调整以反映这个新位置。

此外，如果响应中出现 error 属性，那么这段代码将其视为一种错误情况。这可以用来与服务器端脚本进行协作以指出验证上传文件过程中出现的错误。

最后，在这个新子类的末尾，使用前面给出的标记中的根 ID 来创建一个实例。现在将所有方面结合起来，您应该能够提交该表单来上传新图像，并看到图 27-11 中所示的结果。

图　27-11

与图 27-10 相反，请注意图 27-11 中地址栏内的 URL 并没有发生变化。图像上传的结果已经就地显示出来，而且表单重置链接已经就绪，单击它就可以再次进行上传，而不需要离开该页面。

27.9　本章小结

在本章中，我们浏览了 Dojo 为获取服务器端资源数据提供的各种工具，这样我们就不需要执行完全的页面加载，也就不会担心破坏精心构造的客户端用户界面。在这些 AJAX 支持的功能中，我们了解了 Dojo 如何简化使用 XMLHttpRequest 对象执行各种 HTTP 请求的方式，如何抽离出跨域加载 JSON 源的方法，以及如何使用隐藏的<iframe>元素加载数据和提交带有文件上传字段的表单。

下一章将要介绍的 Dojo 方面将把目前已经讲解的所有内容结合起来：我们将看到 Dojo 窗口部件系统 Dijit 如何将 Dojo 的封装、继承、标记分析、事件和数据工具组合起来形成一个声明式的应用程序组装方式，它只需要编写非常少的代码，有时候只需要使用标记就能完成工作。

第28章

利用窗口部件构建用户界面

在前几章中，我们介绍了 Dojo 工具集的核心功能。这些模块和辅助函数提供了搭建一个具有响应灵敏的用户界面且能够访问服务器端资源的现代 Web 应用程序所需的所有组成部分。但相对于本书中呈现的其他 JavaScript 框架，这些功能并非真正与众不同。

但 Dojo 至少有一部分功能是相对有特色的：我们在第 23 章中曾经看到过，Dojo 提供了一个名为 dojo.parser 的模块，它可用来在 HTML 标记中声明 JavaScript 对象。该工具集的这项功能在 Dojo 的 Dijit 子项目中占据了中心位置，这个子项目包含了非常丰富的用户界面组件(或窗口部件)，这些组件包装浏览器原生 UI 元素，或者封装使用 HTML、CSS 和 JavaScript 构建的全新 UI 元素。

Dijit 窗口部件既可以在常规的代码中通过编程方式实例化，也可以声明为标记中现有节点的增强。这种二元性使得 Dijit 支持“非侵入式”范例，让 JavaScript 和 HTML 尽可能保持独立，同时引入更加集成化的方式，可以与用户界面元素一起定义和声明它们的行为和数据处理代码。

本章内容简介：

- 构建并验证表单
- 管理应用程序布局
- 创建应用程序控件和对话框

28.1 构建并验证表单

Dijit 最为实际的用途是构造带有内置的输入验证和格式化功能的表单。为了强调如何使用 Dojo 和 Dijit 构造 HTML 页面，下面的标记给出了一个准备用于表单构建的完整文档：

```
<html>
    <head>
        <title>Hello Dojo Forms</title>

        <style type="text/css">
            @import "../dojodev/dojo/resources/dojo.css";
            @import "../dojodev/dijit/themes/tundra/tundra.css";
            @import "forms.css";
        </style>
```

```
    <script type="text/javascript">
        djConfig = {
            isDebug:      true,
            parseOnLoad:  true,
            modulePaths:  { "decafbad": "../../ex-dojo-widgets/decafbad", },
        };
    </script>

    <script type="text/javascript" src="../dojodev/dojo/dojo.js"></script>

    <script type="text/javascript">
        dojo.require("dojo.parser");
        dojo.require("decafbad.widgets.forms");
    </script>

</head>
```

上面的标记可以划分为以下几部分：

加载 Dojo 的基本样式表，然后是定义“tundra”窗口部件主题的样式表。我们以前看到过这种标记，但在本章末尾处将更加深入地研究窗口部件主题。

构造 djConfig 变量，为 Dojo 工具集建立配置设置。记住，模块路径的定义相对于工具集加载位置所在的路径。

- 使用<script>元素加载 Dojo 工具集。
- 一旦 Dojo 自身加载完毕，就按照要求声明 dojo.parser 和 decafbad.widgets.forms 模块。

decafbad/widgets/forms.js 文件提供了第二个模块，它的开头内容如下：

```
dojo.provide("decafbad.widgets.forms");

dojo.require('dojo.behavior');

dojo.require("dijit.form.Form");
dojo.require("dijit.form.ValidationTextBox");
dojo.require("dijit.form.NumberTextBox");
dojo.require("dijit.form.NumberSpinner");
dojo.require("dijit.form.CurrencyTextBox");
dojo.require("dijit.form.DateTextBox");
dojo.require("dijit.form.TimeTextBox");
dojo.require("dijit.form.CheckBox");
dojo.require("dijit.form.Textarea");
dojo.require("dojo.data.ItemFileReadStore");
dojo.require("dijit.form.ComboBox");
dojo.require("dijit.form.FilteringSelect");
dojo.require("dijit.form.Slider");
dojo.require("dijit.Editor");
dojo.require("dijit._editor.plugins.TextColor");
dojo.require("dijit._editor.plugins.LinkDialog");

decafbad.widgets.forms = function() {
```

```
    return {

        init: function() {
            dojo.addOnLoad(this, 'onLoad');
            return this;
        },

        onLoad: function() {
            console.log("* Welcome to Chapter 28 and forms!");

            /**
             * Startup code goes here.
             */
        },

        EOF: null // I hate trailing commas.

    };

}().init();
```

这里的 dojo.require()语句列表载入本章剩余内容要用到的所有 Dijit 模块。我们本来可以将这些语句放在页面自身的<head>部分，但将它们放在这里有助于减少混乱。此外，在模块.onLoad()方法中可以添加任何必要的启动代码。

下面的标记提供了导入到页面顶部的 forms.css 的 CSS 声明：

```
form
    { float: left; clear: both; width: 400px; }
form fieldset.main
    { border: 1px solid #ccc; margin: 1em; padding: 0.5em 1em; }
form ul li
    { list-style-type: none; padding-top: 1em; clear: both; }
form ul li.first
    { padding-top: 0em; }
form ul li label
    { width: 10ex; margin-right: 1ex; text-align: right; float: left; }
form ul li label.inline
    { width: auto; margin-right: 0; text-align: left; float: none; }
form ul li div.RichTextEditable
    { margin-left: 11ex; width: 225px; height: 25em; border: 1px solid #ccc; }
form ul li.buttons
   { text-align: right; }
hr
   { clear: both }
```

这为简单的 HTML 表单标记提供了一些样式，并给出了一个由标签和表单字段组成的两列布局。结合主题样式表，这段 CSS 代码有助于说明 Dijit 窗口部件遵循普通的样式。

返回到 HTML 页面，现在开始研究表单自身。

```
<body class="tundra">
    <h1>Hello Dojo Forms</h1>

    <form id="form1" method="post" action="echo.php?type=text"
          dojoType="dijit.form.Form">

       <fieldset class="main">
          <legend>Book inventory management</legend>
          <ul>
```

这些标记似乎预示着一个相当标准的 HTML 表单，但这里有两个 Dojo 相关的内容：

- <body>标记上的 class="tundra"属性确定在整个页面中都应使用“tundra”窗口部件主题。这个主题的 CSS 文件在前面已经加载。
- <form>标记中的 dojoType="dijit.form.Form"属性将导致该元素被包装到 Dijit 窗口部件中。我们将在本章稍后部分更加充分地讲解这项设置的含义。

28.1.1 使用 JavaScript 实例化窗口部件

现在更深入地了解如何将窗口部件放入到表单中，首先使用 JavaScript 代码进行试验。考虑下面附加的表单标记：

```
<li class="first">
    <label for="title">Title</label>
    <input type="text" name="title" id="title" />
</li>
<li>
    <label for="author">Author</label>
    <input type="text" name="author" id="author" />
</li>
```

为了将这些元素连接成窗口部件，查看下面的代码(应该将其包含在 decafbad.widgets.forms 模块的.onLoad()方法中)：

```
dojo.behavior.add({
    '#title': function(el) {
       new dijit.form.ValidationTextBox({
          name:          el.name,
          required:      true,
          properCase:    true,
          promptMessage: "Enter the book title"
       }, el);
    },
    '#author': function(el) {
       new dijit.form.ValidationTextBox({
          name:          el.name,
          required:      true,
          properCase:    true,
          promptMessage: "Enter the author's full name"
       }, el);
```

```
    }
});
```

上面的代码使用 dojo.behavior 构建两个窗口部件(对应于页面上的表单字段)。这两个窗口部件均是 Dijit 窗口部件 dijit.form.ValidationTextBox 的新实例。传入构造函数的第一个参数是一个含有这个窗口部件的参数的对象，而第二个参数则传入一个指向 DOM 节点(该节点应该交给这个窗口部件托管)的引用。

如果回顾第 23 章的结尾部分，那么这段代码看起来应该比较熟悉。当在标记中声明对象时，dojo.parser()将使用这个构造函数方法签名。由于不需要借助这种构造函数用法带来的功能，因此这里采用 JavaScript 通过编程方式实例化窗口部件已经足够。

实际上，在本章剩余部分内容中，我们可能更希望使用前面代码中的 dojo.behavior 而不是声明式标记形式。记住，任何使用标记完成的示例也都可以使用类似上一个示例中的代码来实现。

图 28-1 给出了前一个示例中的代码的运行结果。应该看到，除了在页面的<head>部分中内联定义了表单样式之外，表单字段已经从“tundra”窗口部件主题中获得了一些微妙的渐变效果。

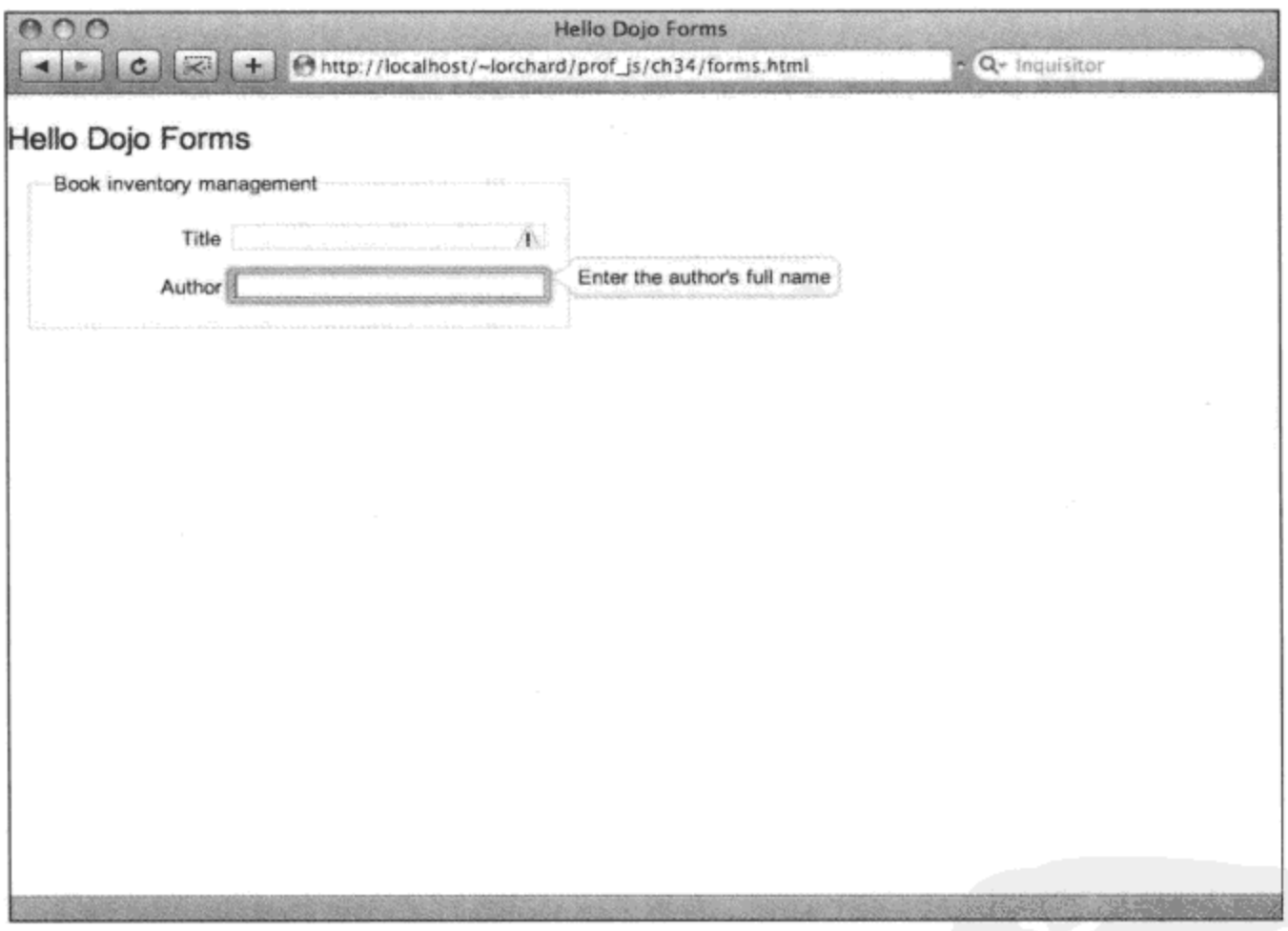


图　28-1

当选中其中一个表单字段时，就会使用 promptMessage 属性创建一个浮动到该表单字段右侧的工具提示。借助 required 属性，在选中其中一个表单字段之后再让该字段保持空白，这会产生一个警告性质的高亮显示效果。最后，properCase 属性将会导致当把输入焦点移出该字段时，输入到该字段的文本的每一个字母都变成大写，这对于书名或作者姓名来说非常有用。

28.1.2　在 HTML 标记中声明窗口部件

为了尝试将 dojo.parser 方法用于窗口部件的创建，我们将前一个示例中的第二个表单字段替

换成下面的标记：

```
<li>
    <label for="author">Author</label>
    <input type="text" name="author"
        dojoType="dijit.form.ValidationTextBox"
        required="true"
        properCase="true"
        promptMessage="Enter the author's full name" />
</li>
```

由于这里不必使用任何 JavaScript 代码，因此也可以将为这个表单字段调用的 dojo.behavior 代码删除。

我们在第 23 章中曾经描述过，dojo.parser 模块在载入时扫描页面并找出所有含有 dojoType 属性的元素，就像前一个示例中的元素一样。对于每个这样的元素，分析器实例化一个新对象，该对象的类型在 dojoType 属性中指定；并且将一组元素属性以及一个指向该元素自身的引用传给该对象的构造函数调用。

此外，dojo.parser 在页面加载时执行其扫描，在此之前，dojoType 属性引用的所有模块必须通过 dojo.require()语句载入。还应该看到，本章中使用的窗口部件模块均出现在之前提供的<head>标记中。

尽管这种集成化的方式看起来直接违反了许多 Web 开发人员推崇的“非侵入式”方式，但它确实有利于将所有上下文保存在一个地方。这段标记完整地描述了预期的窗口部件以及它应该具有的行为，而不需要在不同文件之间跟踪意图。而且它精简了代码；如果没有 dojo.parser，那么它能够优雅地降级，这是因为浏览器会忽略非标准 HTML 属性。

28.1.3　利用正则表达式验证输入

在下面的标记示例中，我们展示了这个 ValidationTextBox 窗口部件的另一个更有用的功能：

```
<li>
    <label for="sku">SKU</label>
    <input type="text" name="sku"
        dojoType="dijit.form.ValidationTextBox"
        required="true"
        regExp="^[0-9\-]+$"
        invalidMessage="Invalid stock ID (eg. 00-00000-000-0)"
        promptMessage="Enter the stock ID (eg. 00-00000-000-0)" />
</li>
```

通过 regExp 属性，我们可以指定一个描述这个表单字段可接受的有效输入的正则表达式。在上面的标记中，必须使用数字和短划线。每次按下键时，这个表单字段的内容都会接受这个测试；如果该内容未能通过验证测试，那么该表单字段将高亮显示并显示一条指示该错误的工具提示。

注意，在这个窗口部件中，同时设有 promptMessage 和 invalidMessage 属性。每当表单字段内容未能通过这个正则表达式定义的测试条件时，invalidMessage 属性就会出现在工具提示中，图 28-2 演示了这一点。该属性是可选的：如果忽略它，那么 Dojo 将使用一条通用的消息。

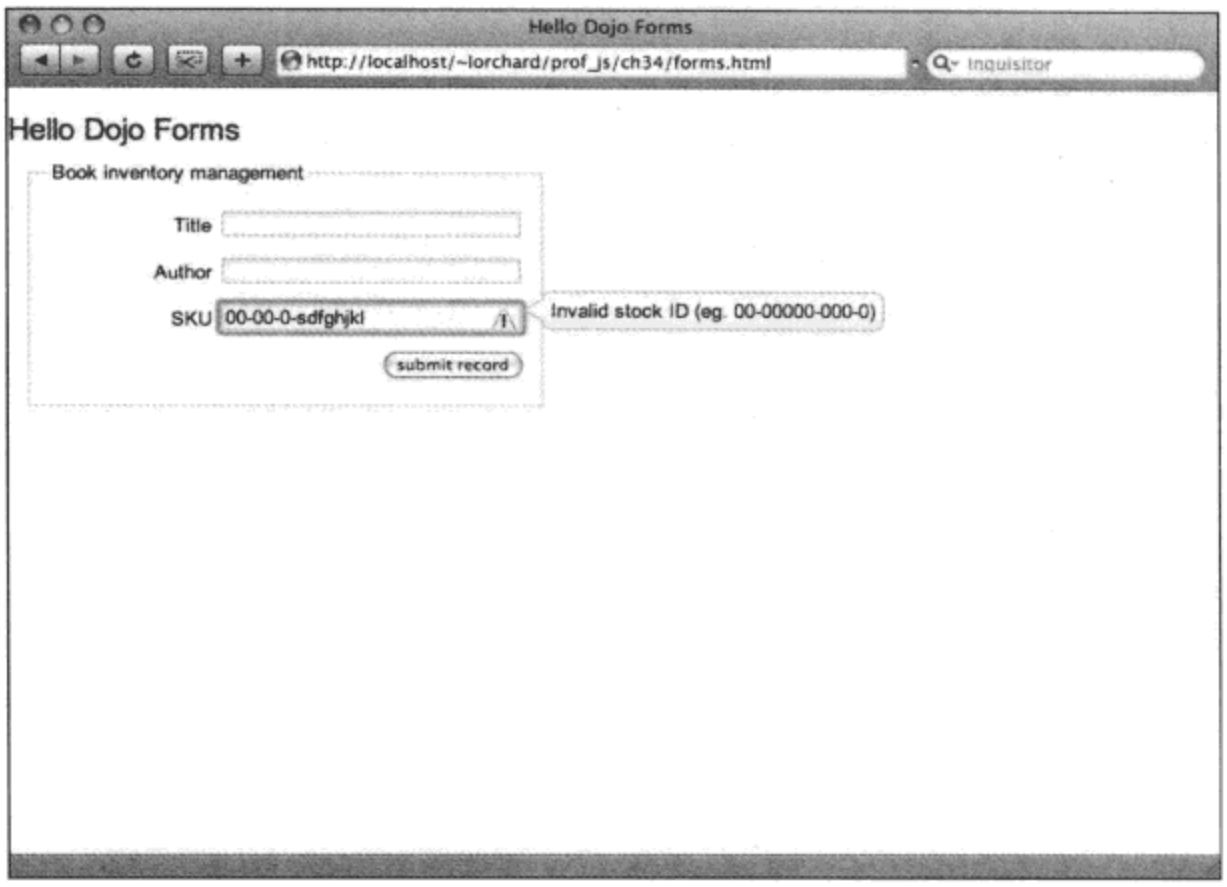


图　28-2

JavaScript 中的正则表达式

一份像样的 JavaScript 正则表达式指南远远超出了本章的介绍范畴，而且实际上它本身足以成为好几本大篇幅著作的主题。但在下面的网站中可以找到一份有关该主题的较好的初级读物：

```
http://developer.mozilla.org/en/docs/Core_JavaScript_1.5_Guide:Regular_Expressions
```

28.1.4　在提交时实施表单验证

尽管这个表单还有其他表单字段，但在进一步深入研究之前，似乎现在应该查看如何在表单提交时验证表单数据的有效性。考虑下面的标记，它完成了表单以及页面自身：

```
                <li class="buttons">
                    <input type="submit" value="submit record" />
                </li>

            </ul>
        </fieldset>

        <script type="dojo/method" event="onSubmit" args="ev">
            if (!this.validate()) {
                dojo.stopEvent(ev);
                alert("Form not yet valid! (defined in markup)");
            }
        </script>

    </form>
```

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

这个表单的最后一部分是一个普通的提交按钮，并没有包装成任何类型的窗口部件。本来可以在这里使用一个按钮窗口部件，但这里的目的是提醒您原生元素是与窗口部件共存的。

Dojo 的功能以带有自定义 dojo/method 类型的<script>元素的形式表现。这是一项交由 dojo.parser 处理的约定，因为它是一种无法识别的脚本类型，浏览器本身会将其忽略，但 dojo.parser 在它的页面扫描中会处理该元素。

这个<script>元素是<form>元素的直接子节点，它在 DOM 树中的位置非常重要。前面我们把这个<form>元素声明为一个 dijit.form.Form 窗口部件，而 dojo.parser 会将这个<script>子节点作为该窗口部件构造函数的函数参数。

<script>元素的 event="onSubmit"属性指示这个窗口部件实例的.onSubmit()方法应该替换成这里的代码。args="ev"属性则指定该方法本身应该接受一个名为 ev 的参数。由于这个窗口部件的.onSubmit()方法处理由它托管的<form>节点上的 submit 事件，因此这是实施表单输入验证的理想地方。

最后，该处理程序自身所做的工作就是触发 Form 窗口部件的.validate()方法。这个方法遍历表单内部的所有表单字段窗口部件，检查每个表单字段是否包含有效数据。如果任何一个表单字段报告数据无效，这个方法就会返回 false，这样 submit 事件就会终止，而且会使用一个警告对话框抛出错误消息。作为一个附带的结果，所有具有无效数据的表单字段都会高亮显示，而且光标输入焦点会放到第一个无效表单字段中。图 28-3 给出了无效表单提交事件的处理结果。

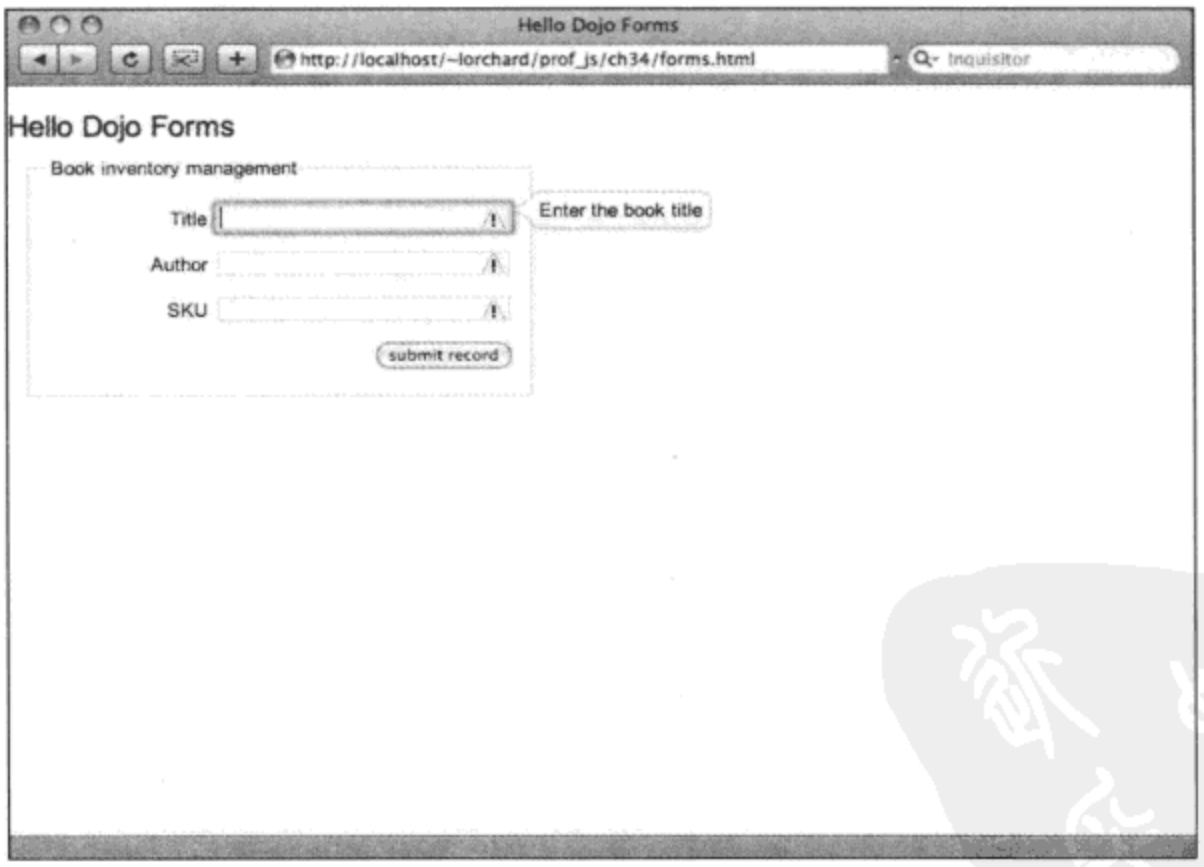


图 28-3

如果不喜欢这种自定义标记方式，那么还可以采用另一种方式：

```
dojo.behavior.add({

    '#form1': function(el) {
```

```
        new dijit.form.Form({
            onSubmit: function(ev) {
                if (!this.validate()) {
                    dojo.stopEvent(ev);
                    alert("Form not yet valid! (connected in JS)");
                }
            }
        }, el);
    }

});
```

记住，在上面的代码中使用 dojo.behavior 替换前面用于<form>标记的 dojoType 属性。

28.1.5　处理数字与货币值

到目前为止我们已经拥有了一个完整的表单，它有几个必填的表单字段，有一个采用基于正则表达式验证测试的表单字段，还有一个检验数据有效性的表单提交事件处理程序。

现在，查看一些更为专用的表单字段。第一个这样的表单字段是处理数字的窗口部件：

```
<li>
    <label for="pages">Pages</label>
    <input type="text" name="pages"
        dojoType="dijit.form.NumberTextBox"
        required="true"
        promptMessage="Enter the number of pages in the book"
        constraints="{min:1,max:999}"
        rangeMessage="Too many pages - we only carry books under 1000 pages" />
</li>
```

这个窗口部件名为 dijit.form.NumberTextBox，由它管理的输入字段用来接收数字。通过使用属性 constraints，可以建立该字段所允许输入的最小值和最大值。虽然有可能利用正则表达式来完成这项工作，但这种方式更加自然。与这种约束配对的是 rangeMessage 属性，可用来指定一条工具提示消息，当该字段的值处于指定的约束范围之外时就会显示该消息。

与 NumberTextBox 紧密相关的是下一个窗口部件，它专门用来处理货币金额：

```
<li>
    <label for="price">Price</label>
    <input type="text" name="price"
        dojoType="dijit.form.CurrencyTextBox"
        required="true"
        currency="USD"
        constraints="{min:1.00, max:199.00}"
        rangeMessage="Books must be priced between $1 and $199"
        promptMessage="Enter the book's price (in USD)" />
</li>
```

请注意，这个 CurrencyTextBox 窗口部件的属性与上一个示例中的 NumberTextBox 非常接近，它们的作用是相同的。constraints 和 rangeMessage 属性管理 min 和 max 值的有效性。

这里的新属性是 currency，它接受 ISO 4217 货币代码，例如 USD、CAD、GBP 和 EUR。在下面的 Wikipedia 页面中可以找到这些货币代码的列表：

```
http://en.wikipedia.org/wiki/ISO_4217
```

但这个属性的主要作用并不是为了检验数据的有效性。我们可以输入处于约束范围之内的任何数字值。相反，这个窗口部件将重新格式化这个数字值以匹配指定货币类型的表示约定。在这里可以尝试不同的货币类型，查看会发生什么情况。

下面进入更高级控件的讨论，查看下一个窗口部件：

```
<li>
    <label for="stock">Stock</label>
    <input type="text" name="stock"
        dojoType="dijit.form.NumberSpinner"
        required="true"
        value="0"
        smallDelta="5"
        constraints="{min:0,max:200}"
        rangeMessage="Shelves will only hold up to 200 books"
        promptMessage="How many of this book are in stock?" />
</li>
```

与前面的 NumberTextBox 窗口部件类似，这个 NumberSpinner 窗口部件管理着一个带有约束范围的数字值。但这里的新功能包括该表单字段末尾的一对向上和向下按钮。当单击这些按钮时，该数字值会显示出递增和递减的效果，递增和递减的量由 smallDelta 指定。当该字段获得输入焦点时，我们还可以使用键盘上的向上和向下方向键来执行同样的操作。

图 28-4 给出了本节中介绍的 3 种新型数字表单字段窗口部件的预览结果，并利用一个超出范围的值来演示工具提示消息。

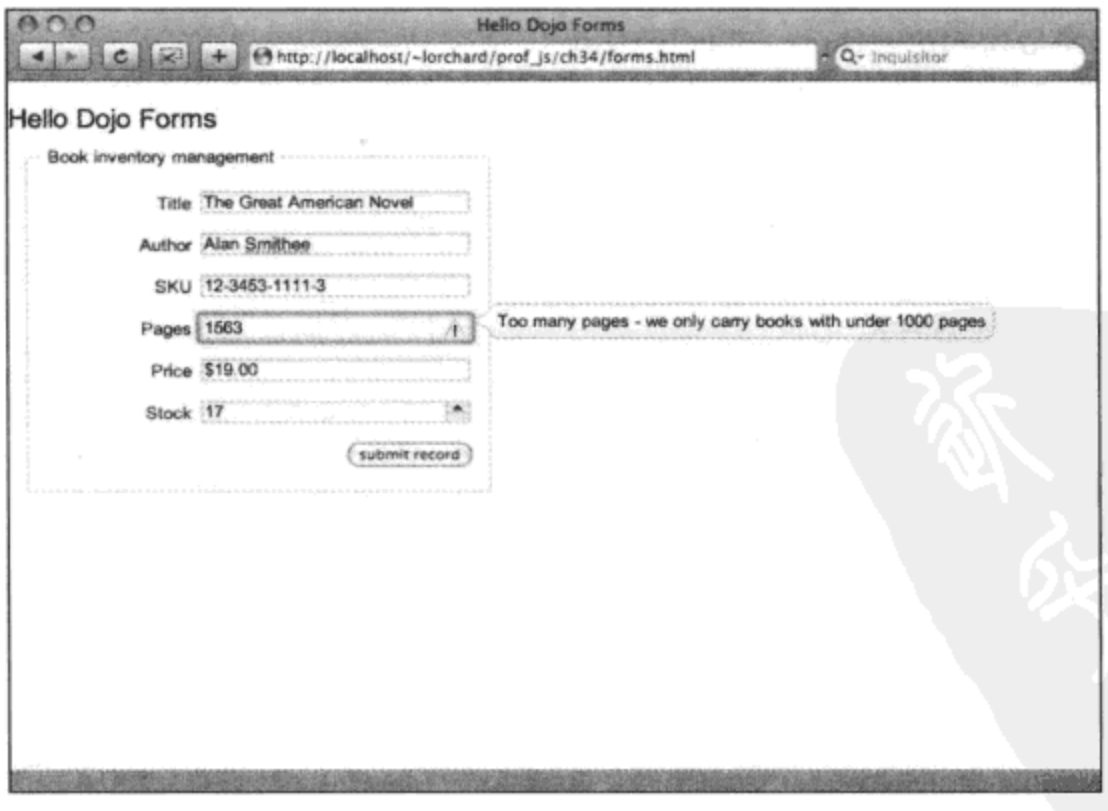


图 28-4

28.1.6　处理日期和时间字段

日期和时间要比数字稍微复杂一些，它们通常是构造表单所必不可少的部分。Dijit 在这一领域中提供了很好的支持，它提供了两个带有自定义界面的窗口部件。

第一个窗口部件为选择日期提供了一个日历界面：

```
<li>
    <label for="pubdate">Date</label>
    <input type="text" name="pubdate"
        dojoType="dijit.form.DateTextBox"
        required="true"
        constraints="{min:'2008-03-19', max:'2008-06-20'}"
        rangeMessage="Store is only open between March 19 and June 20"
        promptMessage="Select the book's publication date" />
</li>
```

这个名为 dijit.form.DateTextBox 的窗口部件提供了一个自定义的基于日历的 UI 元素，它由 HTML、CSS 和 JS 代码构建而成，用于选择一个将会出现在由其托管的输入字段中的日期。可以在图 28-5 中看到这个窗口部件的快速预览。

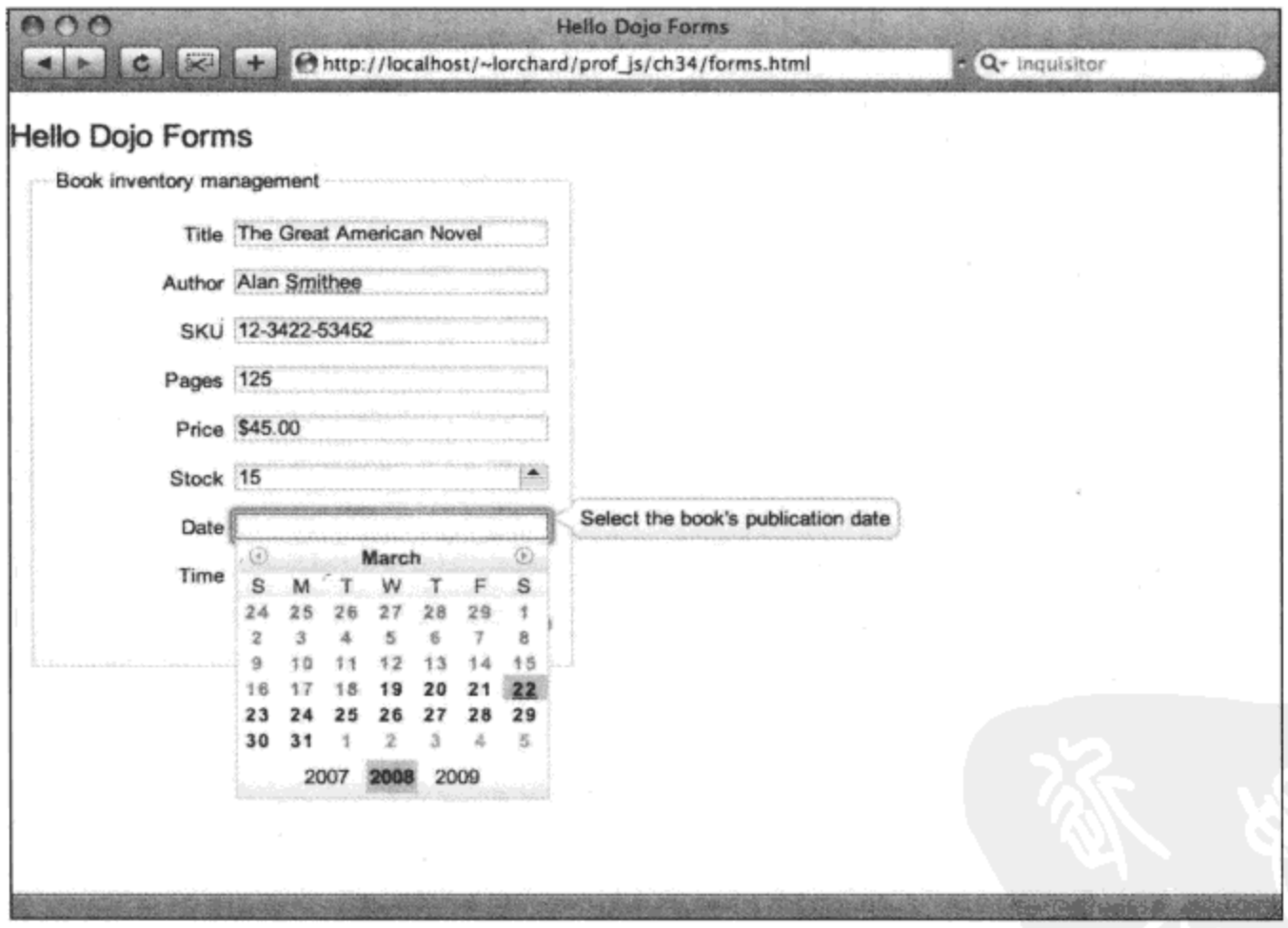


图　28-5

与之前的数字窗口部件类似，该窗口部件遵循该表单字段可接受值的约束条件的定义。这些约束条件采用 ISO8601 格式表达，该格式基本上包含如下的模式：

```
yyyy-MM-ddTHH:mm:ssZ
```

因此，当前日期和时间可以表示为如下。

```
2008-03-24T06:29:15-08:00
```

在 dojo.date.stamp.fromISOString 函数的文档中可以找到有关 Dojo 针对这种格式的具体实现的更多内容，可在下面的网址中找到该文档：

```
http://redesign.dojotoolkit.org/jsdoc/dojo/HEAD/dojo.date.stamp.fromISOString
```

在试验日历窗口部件的过程中，您应该还会注意到它遵循约束条件，将位于其有效范围之外的日期变灰，而且不允许选择。

下面的表单字段用来选择某一天的具体时间：

```
<li>
    <label for="reltime">Time</label>
    <input type="text" name="reltime"
        dojoType="dijit.form.TimeTextBox"
        required="true"
        constraints="{min:'T09:00', max:'T17:00'}"
        rangeMessage="Books are shelved only between 9AM and 5PM"
        promptMessage="Select the shelving time of the book" />
</li>
```

上面示例中的窗口部件名为 dijit.form.TimeTextBox，它提供了一种选择器，将某一天的时间分隔成几部分。选中的值最终进入该窗口部件托管的文本输入字段。我们在图 28-6 中可以看到该窗口部件运行时的情形。与日历窗口部件类似，它也接受采用 ISO8601 格式表达的时间值约束条件。但与日历窗口部件不同的是，在编写本书时，它不会阻止任何无效的选择。

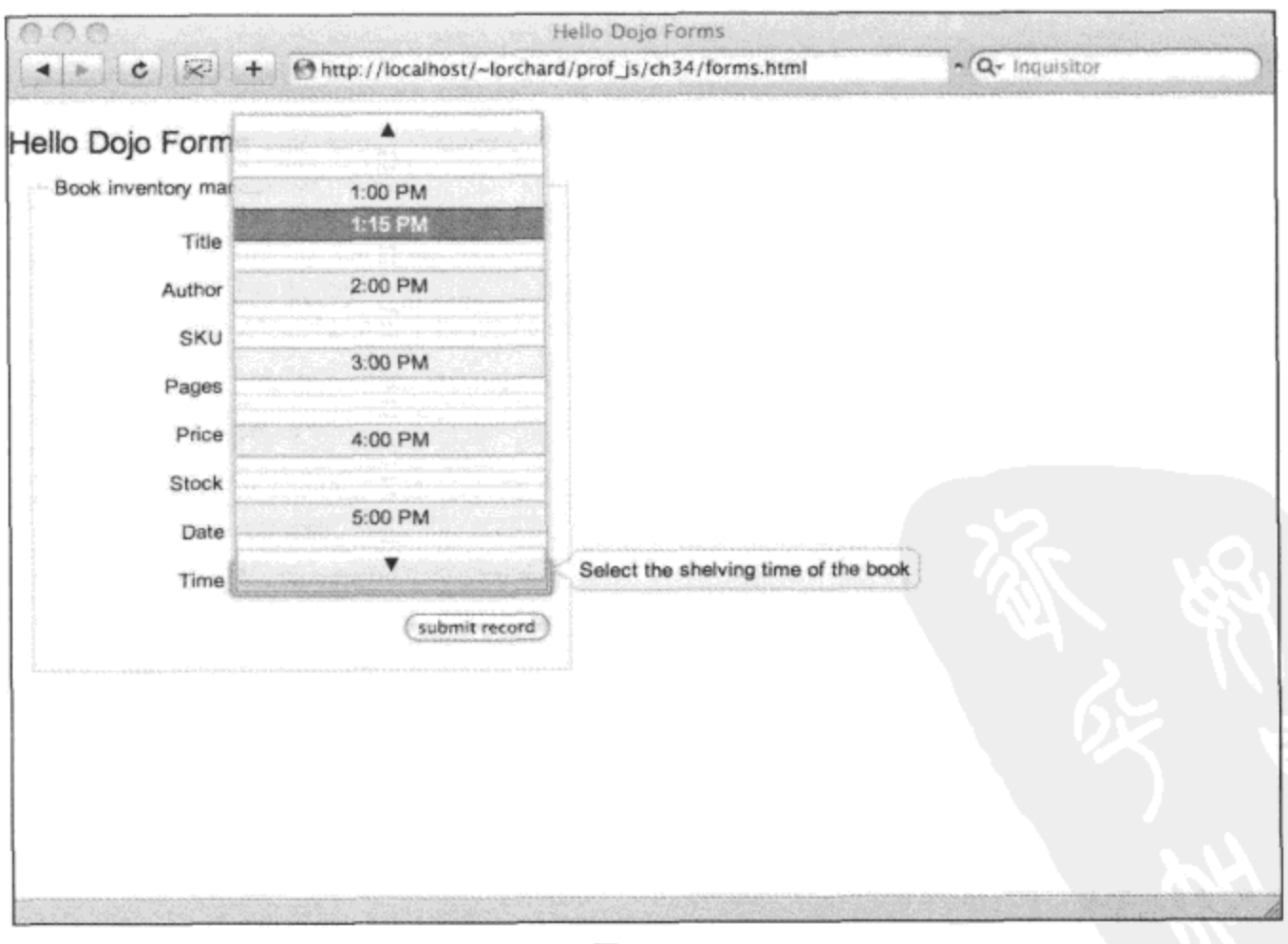


图　28-6

这两种基于时间的窗口部件均说明了 Dijit 可以实现哪些高级的窗口部件，这是因为它们均是

通过模板化的 HTML、CSS 和 JS 代码来实例化的，我们将在本章末尾部分更加深入地讲解这一点。这些窗口部件也非常有用，因为使用它们选择的数据最终将作为原生的浏览器文本输入字段的值，因此在提交表单时能够像任何其他字段一样发送出去。

28.1.7　增强单选按钮和复选框

Dijit 为单选按钮和复选框提供的窗口部件并不像日历和时间选择器那样引人注目，但它们确实提供了至少一项微妙的优势。考虑下面这段定义了一对单选按钮的标记：

```
<li>
    <label for="cover">Cover</label>

    <input type="radio" name="cover" id="cover-paperback"
        value="paperback"
        dojoType="dijit.form.RadioButton" />
    <label for="cover-paperback" class="inline">Paperback</label>

    <input type="radio" name="cover" id="cover-hardback"
        value="hardback" checked="checked"
        dojoType="dijit.form.RadioButton" />
    <label for="cover-hardback" class="inline">Hardback</label>
</li>
```

除了 dojoType 属性的值为 dijit.form.RadioButton 之外，这里并没有多少与普通单选按钮不同的地方。对于 Dijit 的复选框窗口部件也同样如此：

```
<li>
    <label for="features[]">Features</label>

    <input type="checkbox" name="features[]" id="features-slipcover"
        value="slipcover"
        dojoType="dijit.form.CheckBox" />
    <label for="features-slipcover" class="inline">Slipcover</label>

    <input type="checkbox" name="features[]" id="features-cdrom"
        value="cdrom"
        dojoType="dijit.form.CheckBox" />
    <label for="features-cdrom" class="inline">Computer media</label>
</li>
```

除了 dojoType 的值为 dijit.form.CheckBox 之外，这里也同样没有值得描述的内容。这些表单字段的名称 features[]是一种基于 PHP 的命名约定，它为服务器提供了一个线索，提示它们可能对这些字段提供多个值。

但使用这些窗口部件的主要原因在于，呈现这些通常非常难以样式化的表单元素，从而让它们参与到 Dijit 的窗口部件主题中。您应该能够在图 28-7 中看到这一点，单选按钮和复选框都有着与原生平台 UI 存在很大不同的外观。

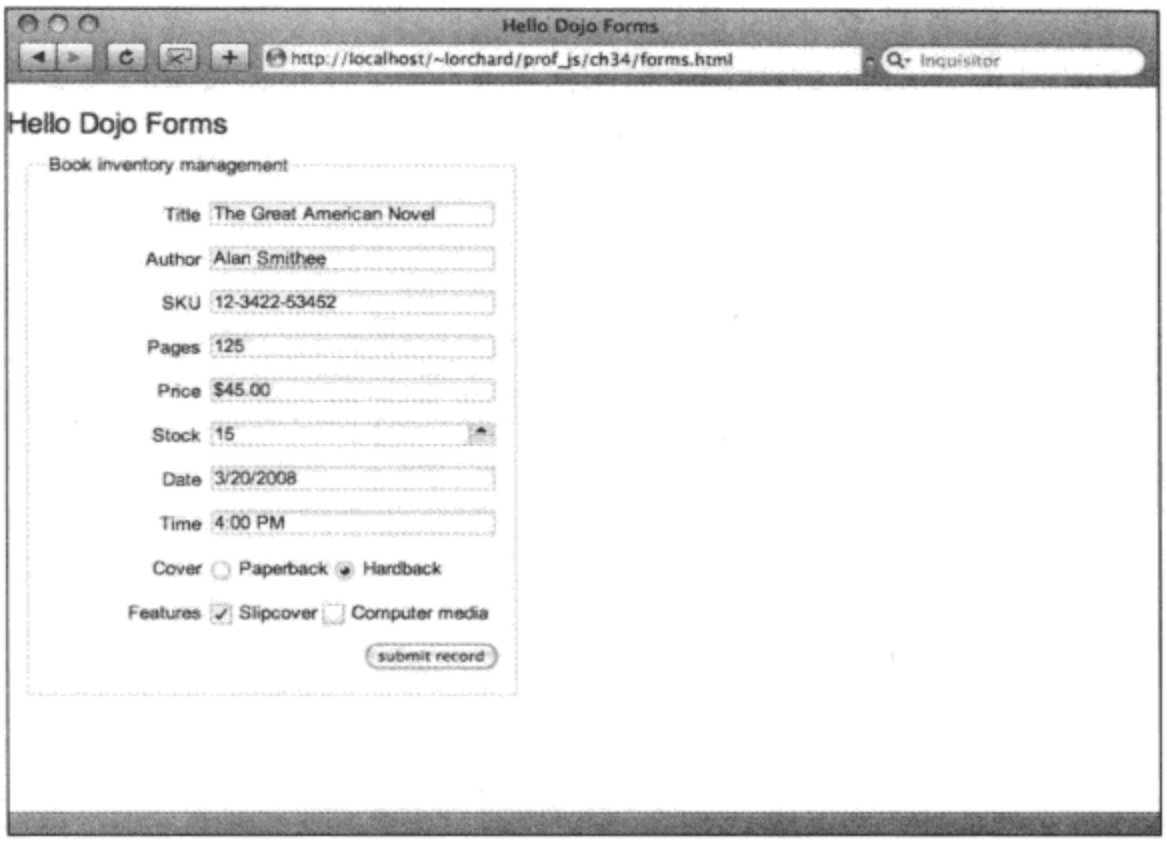


图 28-7

操作选项字段和数据源

下一个由 Dojo 增强的表单元素是下拉式选择框。这些元素通常是页面上最为数据密集的一些表单字段，因此 Dijit 为构造这些元素提供了一些有趣的选项。查看下面的标记：

```
<li>
    <div dojoType="dojo.data.ItemFileReadStore" jsId="genre_store"
        url="genres.json"></div>

    <label for="genre">Genre</label>
    <input name="genre" value="Misc"
        dojoType="dijit.form.ComboBox"
        store="genre_store"
        searchAttr="name"
        autoComplete="true"
        required="true" />
</li>
```

在这个示例中，首先请注意用来声明 dojo.data.ItemFileReadStore 类的一个实例的空白 元素，虽然从技术上来讲它本身并不是一个窗口部件，但 dojo.parser 会对其进行解释以创建一个对象。

要想彻底描述 dojo.data 程序包，我们需要使用一整章的篇幅，但简而言之，该程序包提供了一种以本地方式(并不完全像使用简单的数据库那样)访问服务器端数据资源的途径。这里的用法只涉及基础，如果希望了解有关 dojo.data API 的 Dojo 文档，请查阅下面的页面：

- http://redesign.dojotoolkit.org/jsdoc/dojo/HEAD/dojo.data
- www.dojotoolkit.org/book/dojo-book-0-9/part-3-programmatic-dijit-and-dojo/data-retrieval-dojo-data-0

尽管这里的示例中使用的数据源均基于静态 JSON 结构，但我们有可能实现一些子类，重复地查询远程服务器端资源来获取部分数据(例如从某个大型集合中搜索选项的子集，这些选项均以该表单字段中目前已经输入的字母开头)。

但为了尽量简化这个示例，这段标记创建一个数据源对象，它从一个名为 genres.json 的 JSON 资源那里获取内容。借助 jsId 属性，将一个指向该对象实例的引用放入全局 JS 变量 genre_store 中。

获取 genres.json 资源的结果应该如下所示：

```
{
    "identifier": "name",
    "items": [
        { "name": "Misc",  },
        { "name": "Scifi" },
        { "name": "Mystery" },
        { "name": "Romance" },
        { "name": "Tech" },
        { "name": "Self-Help" }
    ]
}
```

如果将这个数据看作是关系型数据库中的一张表，那么 identifier 属性基本上就是该数据集的主键，而 items 下的对象数组就是数据行。这个数据集中的每一项均是一个对象，该对象的属性组成了各个数据列。在这里，它是一张非常简单的表，实际上就是一个书籍类型名称的列表。

现在，考虑 dijit.form.ComboBox 窗口部件的声明。它有一个 store 属性，其值为“genre_store”，这对应于前面描述的数据存储的 jsId。它还有一个 searchAttr 属性，其值为“name”，这对应于数据集中包含类型名称的数据列。

当将这些代码结合起来的，就会得到图 28-8 中所示的结果：一个带有按钮的文本字段，当单击按钮时会根据这个数据源中的可用项来弹出一个选项列表。还可以向这个表单字段中输入一些文本，它会根据输入内容从列表中提供建议(如果有的话)。

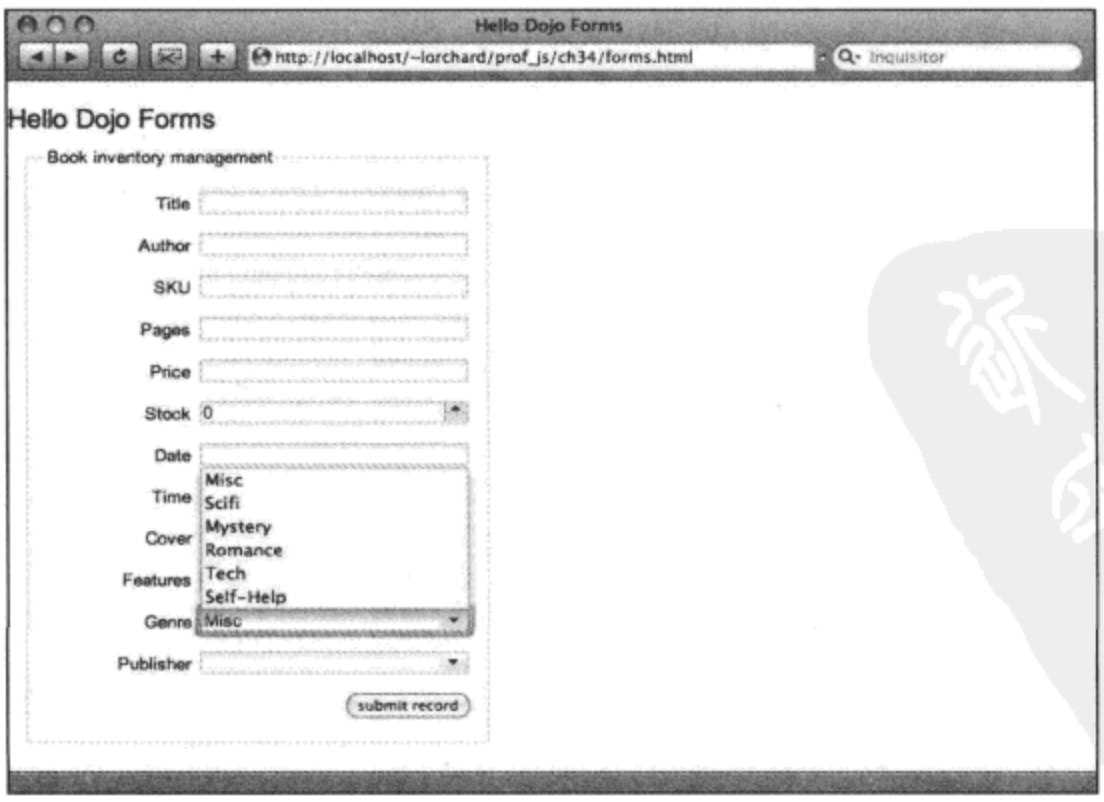


图　28-8

下面这段标记声明的窗口部件稍有不同(但是有一定关联):

```
<li>
    <div dojoType="dojo.data.ItemFileReadStore" jsId="publisher_store"
        url="publishers.json"></div>

    <label for="publisher">Publisher</label>
    <select name="publisher" value="Wiley"
        dojoType="dijit.form.FilteringSelect"
        store="publisher_store"
        searchAttr="name"
        labelAttr="label"
        labelType="html"
        required="true">
        <option value="001">Wiley</option>
    </select>
</li>
```

与前一个示例类似，这个表单字段借助一个数据源来提供数据，这一次是通过一个名为publishers.json 的资源来提供数据。但是，这一次的 JSON 数据有点复杂:

```
{
    "identifier": "code",
    "items": [

        { "code": "001", "name": "Wiley",
            "label": "<b>Wiley<\/b>"},

        { "code": "a08", "name": "Sybex",
            "label": "<i>Sybex<\/i>"},

        { "code": "fdd", "name": "Wrox",
            "label": "<u>Wrox<\/u>"},

        { "code": "010", "name": "Pfeiffer",
            "label": "<small>Pfeiffer<\/small>"},

        { "code": "00f", "name": "Frommer\'s",
            "label": "<big>Frommer\'s<\/big>"},

        { "code": "00a", "name": "CliffsNotes",
            "label": "<center>CliffsNotes<\/center>"},

        { "code": "101", "name": "Audel",
            "label": "<sup>Audel<\/sup>"}

    ]
}
```

dijit.form.FilteringSelect 窗口部件看上去就像是一个 ComboBox 窗口部件，但它的行为更像是一个标准的<select>元素。与 ComboBox 窗口部件的功能相反的是，FilteringSelect 窗口部件还允

许为下面的每一项进行单独的定义：

- 当输入内容时如何在弹出式菜单和提示中呈现数据项。
- 当选中或输入某一个数据项时出现在可见表单字段中的文本。
- 随表单提交的值(基于选中的数据项)。

可以看到在 JSON 数据源中分别使用 label、name 和 code 属性定义了这 3 项。可以在数据源中任意指定这些项，下面的条件确定如何使用它们：

- 窗口部件上的 labelAttr 属性指定了数据源中用来构造弹出式菜单的属性，而 labelType 属性既可以是 text，也可以是 html，它确定了如何呈现数据源中的值。
- 窗口部件上的 searchAttr 属性确定在文本字段中出现什么内容，就像它在 ComboBox 窗口部件中的功能一样。
- 数据源的 identifier 报头属性确定了在成功提交表单时将选中项的哪个属性发送给服务器。

从图 28-9 中可以看到，FilteringSelect 窗口部件允许在弹出式菜单显示的每个菜单项中使用样式和标记。此外，由于窗口部件的 searchAttr 属性可以不同于数据源的 identifier，因此可以使用比最终提交的数据(在这个示例中，这是一个对用户非常不友好的、需要修饰的数据库代码)可读性更好的文本来显示选项。

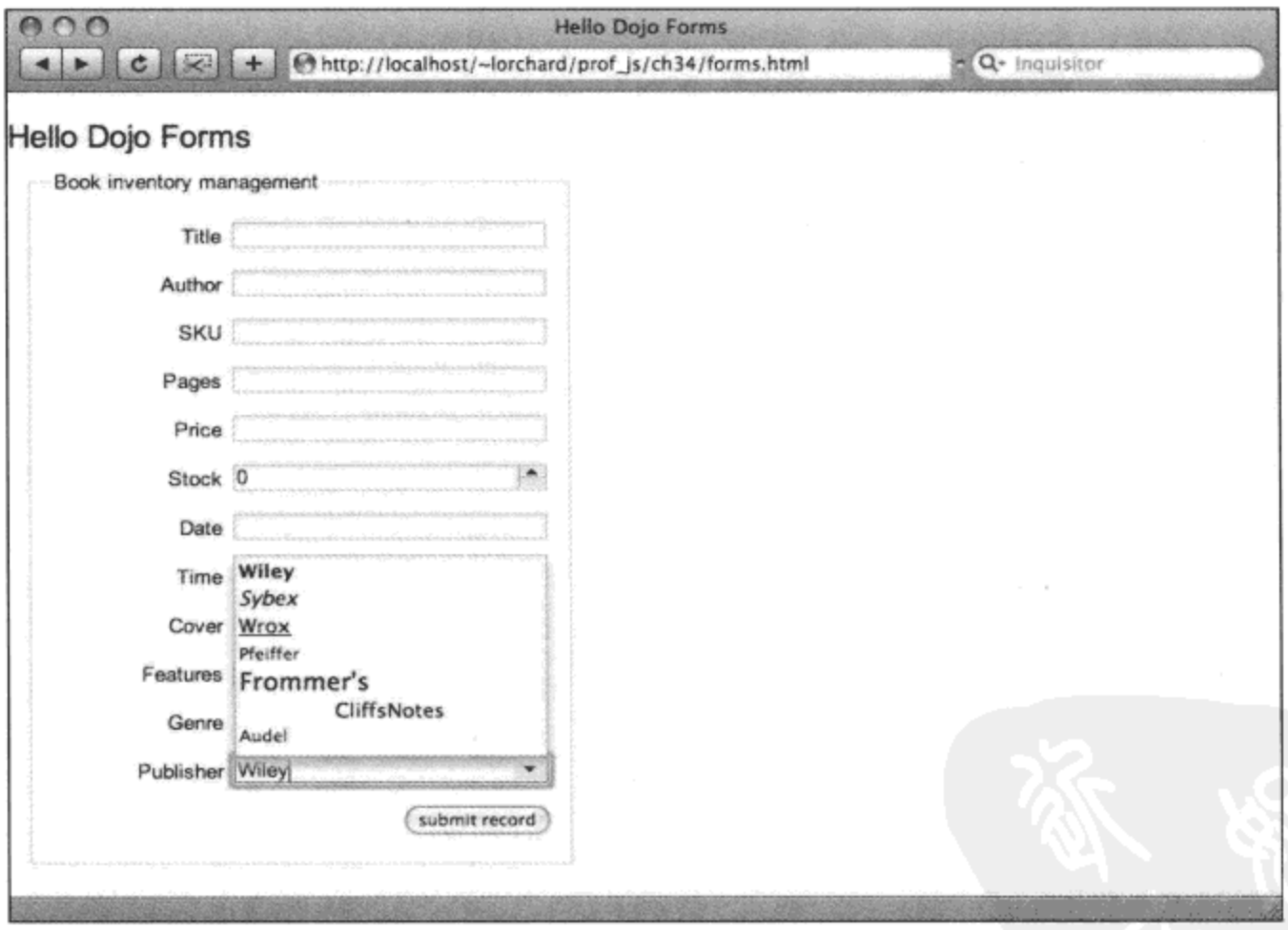


图　28-9

给定显示内容与提交值之间的关系之后，这个窗口部件还根据提供给它的列表实施了数据验证要求。图 28-10 描述了当用户输入一个并没有出现在数据源项中的值时发生的情况。

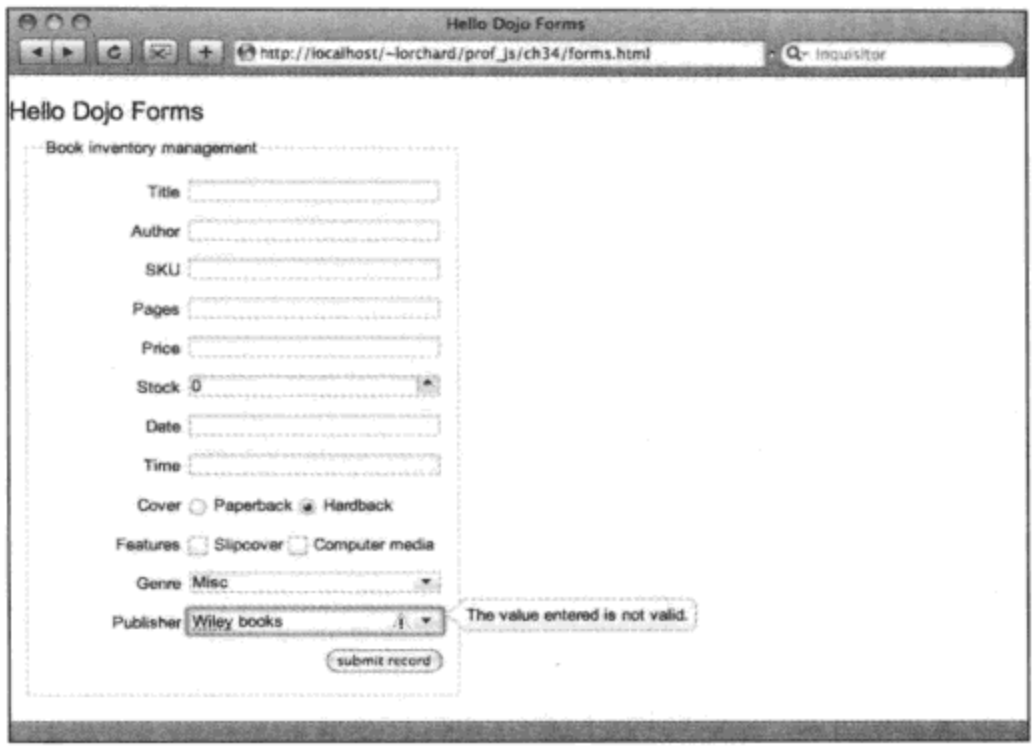


图　28-10

因此，ComboBox 窗口部件只允许利用数据源提供的列表中给出的建议来输入数据。在此基础之上，FilteringSelect 窗口部件允许根据采用 HTML 或普通文本构建的菜单项来输入数据，将选项映射到不可见值，并强制要求数据位于数据源中的项约束范围之内。

28.1.8　使用滑块进行离散值的选取

当有一组离散值时，在某些情况下，数字旋钮和选项框都不是可以使用的最佳 UI 元素。这就是滑块窗口部件的用武之地，下面的标记就定义了这样的一个滑块窗口部件：

```
<li>
    <label for="rating">Rating</label>
    <input type="hidden" name="rating" id="rating" value="0" />

    <div id="rating_slider" dojoType="dijit.form.HorizontalSlider"
        value="3" style="width: 225px;"
        minimum="0"
        maximum="5"
        discreteValues="11"
        showButtons="true">

        <script type="dojo/connect" event="onChange">
            dojo.byId('rating').value = this.getValue();
        </script>

        <ol dojoType="dijit.form.HorizontalRuleLabels"
            style="height:1em; font-size:75%; color:#333;">
            <li>awful</li>
            <li>bad</li>
            <li>so-so</li>
            <li>okay</li>
            <li>good</li>
            <li>great</li>
```

```
        </ol>

    </div>
</li>
```

滑块是一种带有手柄的用户界面元素，可以沿着水平方向(dijit.form.HorizontalSlider)或垂直方向(dijit.form.VerticalSlider)来回拖动手柄，跳到一组离散位置中的一个位置。

在上面的标记中，这些位置由 minimum、maximum 和 discreteValues 属性定义。属性 minimum 和 maximum 定义了滑块的上下数字边界值，而 discreteValues 属性指定了将这两个边界值之间的数字范围划分为多少个等长的部分。给定上面的标记，从 0 到 5 的数字值是可行的，将介于两者之间的范围按照 0.5 的半个步长进行划分，因此就可以看到 0、0.5、1、1.5 等这样的值。

注意，在上面的标记中，有一个名为 rating 的隐藏表单字段。这个表单字段之所以存在，是因为 dijit.form.HorizontalSlider 窗口部件本身并不是一个表单元素。相反，它完全是一个独立的 UI 元素，因此需要使用一点粘贴脚本代码将其状态变化与这个最终包含在表单提交中的隐藏表单字段绑定起来。

这样就再次出现了自定义<script>元素：这一次它的类型为“dojo/connect”，事件属性为“onChange”。这类脚本并不会完全把窗口部件的.onChange()方法替换，而是会由 dojo.parser 通过 dojo.connect()调用附加到该方法。在第 27 章中，我们已经了解如何将处理程序连接到方法调用(在这里也正是这样做的)：采用声明式标记表达，并由 dojo.parser 页面扫描实施。

对于处理程序代码，它只是通过调用窗口部件的.getValue()方法来获取滑块的当前值，而且每次滑块位置变化时都会更新这个隐藏的表单字段。

这段标记的最后一部分是在 HorizontalSlider 窗口部件内声明一个 dijit.form.HorizontalRuleLabels 窗口部件。这个窗口部件将滑块与几个定位标记(声明为一个有序列表)关联起来。这些列表项将沿着滑轨均匀地分布，而滑块将有几个位置来停放这些标签。请注意，这里只包括 6 个标签，考虑到我们定义了 11 个离散的位置，这个滑块只计数滑轨上的奇数位置。因此，该滑块将提供介于这些标签之间的位置。

图 28-11 描绘了这里给出的窗口部件在浏览器中的显示结果。

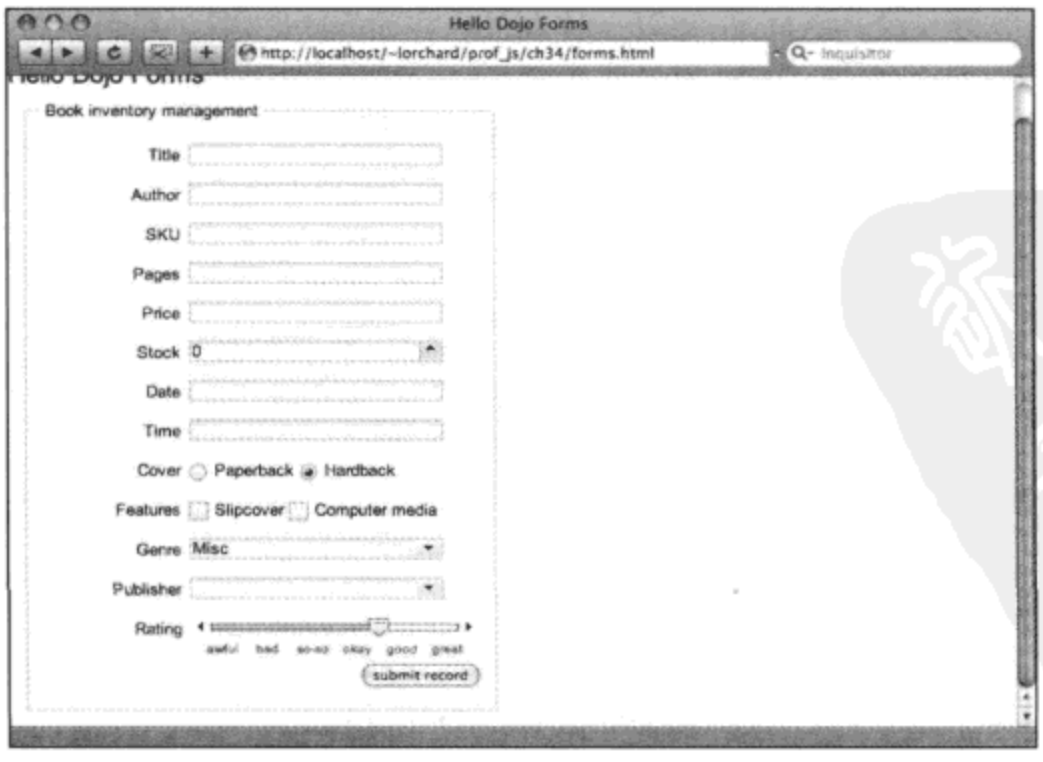


图　28-11

28.1.9 使用动态文本区域和富文本编辑器

Dojo 还对另一个原生浏览器元素 textarea 窗口部件进行了小幅度的升级，就像下面的标记中声明的那样：

```
<li>
    <label for="abstract">Abstract</label>
    <textarea name="abstract" style="width:50ex; height: auto"
        dojoType="dijit.form.Textarea"></textarea>
</li>
```

这个 dijit.form.Textarea 窗口部件并没有采用标准<textarea>元素的 cols 和 rows 属性，而是要求使用 CSS 样式 width 和 height。该窗口部件的这种交换所提供的功能是能够自动增大以适应输入到它里面的内容。

在上面的标记中，我们将宽度限定为 50ex，但是把高度设为 auto。这意味着当在这个文本区域中输入时，这个窗口部件的高度将逐行增加以容纳输入的文本。这可以让我们放置初始时比较紧凑的表单字段，但可以根据需要扩大它。

虽然有用，但这个 textarea 窗口部件不仅仅是一个文本编辑器。更加有趣的地方在于下面声明的窗口部件：

```
<li>
    <label for="review">Review</label>
    <input type="hidden" name="review" id="review" value="" />
    <div id="review_editor"
        dojoType="dijit.Editor"
        plugins="[
            'undo','redo','|',
            'bold','italic','underline','|',
            'indent', 'outdent', '|',
            'foreColor', 'hiliteColor', '|',
            'createLink', 'insertImage'
        ]"
        onChange="dojo.byId('review').value = dojo.trim(this.getValue());">

        Compose your book review here.

    </div>
</li>
```

与数据源类似，可能需要专门使用一章的篇幅来讲解这个窗口部件。dijit.Editor 窗口部件提供了类似微型字处理程序(可以处理字体、格式化、链接以及图像，生成的结果为 HTML 标记)的富文本编辑器。

在编写本书期间，这个窗口部件还在活跃开发中，而且有些功能可能还有一些不稳定，但它仍然是非常实用的窗口部件。在下面的 Dojo 文档页面上可以阅读更多有关这个窗口部件的信息：

- http://redesign.dojotoolkit.org/jsdoc/dijit/HEAD/dijit.Editor

- www.dojotoolkit.org/book/dojo-book-0-9/part-2-dijit/advanced-editing-and-display/editor-rich-text

在上面的标记中，我们对编辑器窗口部件的处理方式与滑块窗口部件并无不同：将一小段粘贴代码附加到 onChange 事件，将编辑器中的修改转发到一个隐藏的表单字段中。

此外，还有一个 plugins 属性，它列出了下列将要包含在窗口部件的工具栏中的项：

- undo、redo　提供用来取消和重做修改的按钮。
- bold、italic、underline　将格式化修改应用于文本。
- indent、outdent　操作文本的缩进层次。
- foreColor、hiliteColor　设置由 dijit._editor.plugins.TextColor 模块(前面将其声明为需求模块)提供的文本颜色和背景颜色。
- createLink、insertImage　将文本转换成超链接以及嵌入图像，由 dijit._editor.plugins.LinkDialog 模块(前面将其声明为需求模块)提供。

可以看到，该编辑器有一些开箱即用的插件，还有一些插件可以独立于模块在编辑器核心程序包之外加载。如果忽略 plugins 属性，那么所有的内置按钮都会包含进来，可以这样尝试以查看有哪些插件可用。

在这两个文本编辑窗口部件的后台完成了大量的工作，但它们在页面上的声明非常简单。图 28-12 给出了这两个窗口部件在页面上的预览。

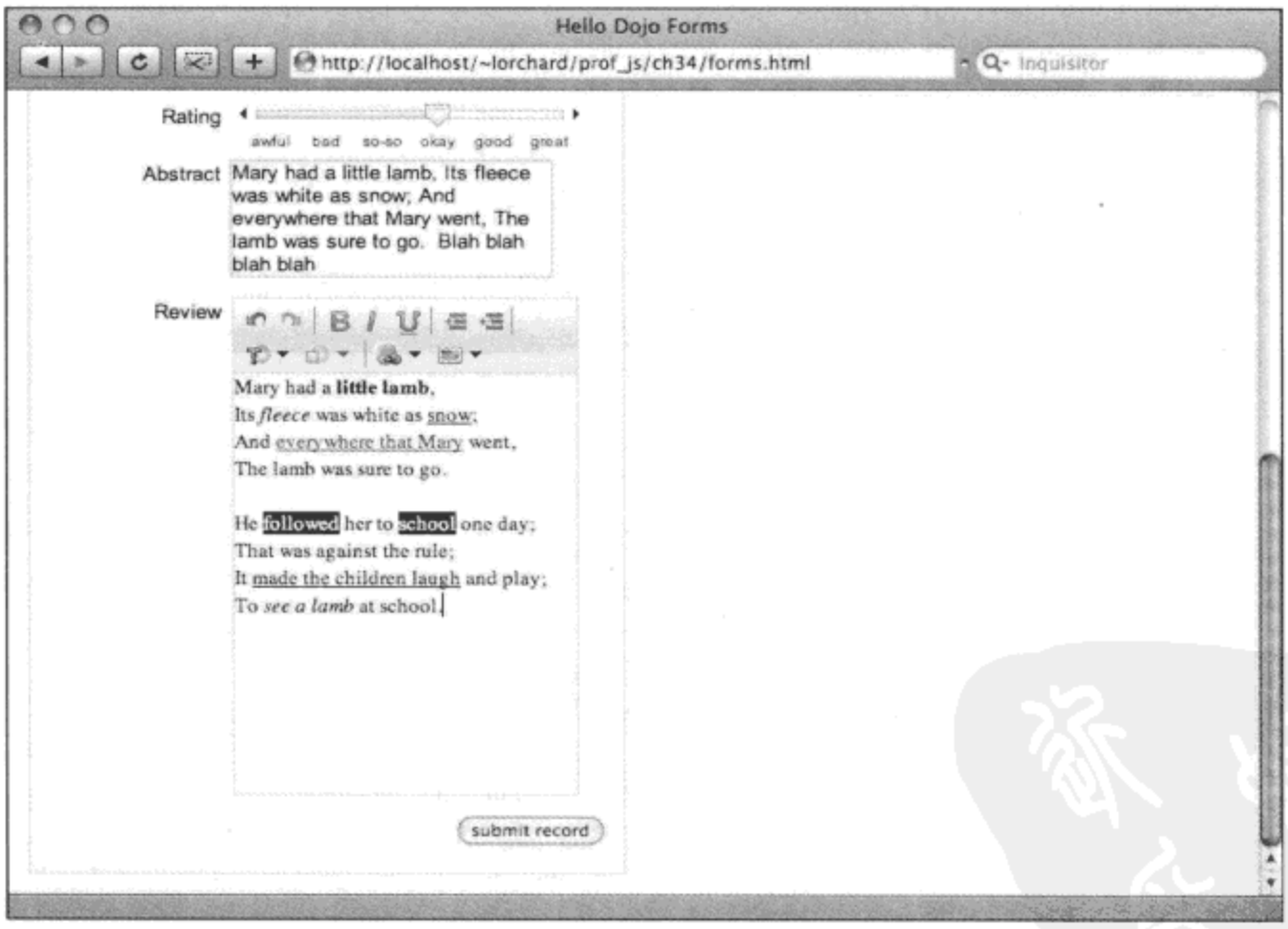


图　28-12

28.2　管理应用程序布局

一种看待网页创建的方式是将其作为文档，使用 CSS 样式构建列和部分，并依靠浏览器的主

滚动栏在多页内容之间导航。另一种方式是将浏览器看作是一个应用程序窗口，完全管理可用空间以将其用于自己的用户界面元素以及内容显示。

虽然可以使用基于内容的页面技术来构建全窗口应用程序，但这两种方式的差异很大，必须采用不同的方式来实现它们。这就是 Dijit 的布局窗口部件的用武之地。我们在构造用户界面时不用操心 CSS 样式以及底层的元素摆放这样的细粒度问题，Dijit 提供了一组窗口部件，这些窗口部件定义了一种不同类型的自顶向下的窗口空间管理方法。有些窗口部件将空间划分成灵活的区域，而其他的窗口部件则管理应用程序中的子面板的可见性。

此外，借助于 dojo.parser 提供的声明式窗口部件构造方式，我们可以创建相当复杂的用户界面布局，而不需要使用一行额外的 CSS 或 JavaScript 代码。

28.2.1 搭建应用程序布局页面

利用 Dijit 为一个全窗口应用程序搭建基本结构与构建其他页面并无太大不同，实际上它甚至要更加简单。下面的标记就是这样的一个页面的开头部分：

```
<html>
    <head>
        <title>Hello Dojo Layouts</title>

        <style type="text/css">
            @import "../dojodev/dojo/resources/dojo.css";
            @import "../dojodev/dijit/themes/tundra/tundra.css";
        </style>
```

```
<style type="text/css">
    html, body, #main {
        /* make the body expand to fill the visible window */
        width: 100%; height: 100%;
        padding: 0; margin: 0;
        /* erase window level scrollbars */
        overflow: hidden;
    }

    h1, h2, h3 { margin: 0.5em 0 1em 0 }
    p  { margin: 0 0 1em 0 }

    .box     { border: 1px #bbb solid; }
    .content { padding: 0.5em; overflow: auto }

    #header  { height: 40px; }
    #sidebar { width: 150px; }
    #content { padding: 1em; }
    #footer  { height: 25px; padding: 0.25em; text-align: right; }

    #sidebar ul { margin-left: -1em; }

    #mainstack { width: 75%; height: 75%; border: 1px #888 solid }
</style>
```

这个页面头部的第一部分主要关注 CSS。导入基础的 Dojo 样式表以及窗口部件主题“tundra”。

然后是页面特有的样式，其数量非常少：首先，样式基本上都应用于页面，指定页面以及包含用户界面的主 元素应该从一边到另一边地填充浏览器窗口。然后，还有一些样式用来调整边距并建立几个主要页面区域的大小。接着，几个完全采用标记声明的 Dijit 窗口部件管理该页面布局方面的剩余部分，不再需要其他 CSS 代码。

在下面这段标记中，我们完成了这个页面的整体搭建工作：

```
<script type="text/javascript">
    djConfig = {
        isDebug:      true,
        parseOnLoad:  true
    };
</script>

<script type="text/javascript"
    src="../dojodev/dojo/dojo.js"></script>

<script type="text/javascript">
    dojo.require("dojo.parser");

    dojo.require("dijit.layout.ContentPane");
    dojo.require("dijit.layout.BorderContainer");
    dojo.require("dijit.layout.StackContainer");
    dojo.require("dijit.layout.AccordionContainer");
    dojo.require("dijit.layout.SplitContainer");
    dojo.require("dijit.layout.TabContainer");
    dojo.require("dijit.layout.LinkPane");
</script>

</head>

<body class="tundra">
    <!-- content goes here -->
</body>

</html>
```

上面的代码给出了载入 Dojo 工具集并根据需要声明几个 Dijit 模块的通常做法。与使用 CSS 一样，这里除了把 dojo.parser 载入页面并加载将要用到的窗口部件之外，并不需要太多代码。

28.2.2　将 ContentPane 用作布局构建块

我们加载的第一个窗口部件(名为 dijit.layout.ContentPane)是在 Dijit 布局中显示内容的基本构建块。在最小配置情况下，它可以充当一个简单的容器，占据自己的一块空间，或者向父窗口部件提供挂钩来管理它的位置和大小。

但 ContentPane 窗口部件确实提供了至少一项值得注意的内置技术：

```
<div class="content"
```

```
        dojoType="dijit.layout.ContentPane"
        loadingMessage="Loading content..."
        errorMessage="Content load failed!"
        href="content.html"
        preventCache="true"
        refreshOnShow="true">

        <!-- content will be loaded dynamically -->

    </div>
```

利用 href 属性，ContentPane 窗口部件可以调用 dojo.xhrGet()从外部资源获取数据以填充内容。与使用<iframe>元素相反，这个内容通常应该是一个文档片段，它会立即注入到 ContentPane 窗口部件正文内部的页面 DOM 中。

这种内容加载机制在等待资源获取请求完成时演示 loadingMessage 属性，在成功时注入该内容，在失败时则显示 errorMessage 的内容。所有这些都是可选属性，如果某个属性没有设置，就会采用其默认值。

下面演示的属性是 preventCache，它将在获取的 URL 后面附加一个基于时间的参数，这样就可以让请求变得唯一，从而绕开缓存机制，以有助于确保获取的内容是最新的内容。这个属性默认为 false，因此可以让浏览器缓存正常发挥作用。

最后一个属性是 refreshOnShow，当把 ContentPane 作为另一个控制内部元素可见性的窗口部件的一部分进行管理时，这个属性非常有用。如果该属性设置为 true，那么每当把这个 ContentPane 窗口部件从视图中隐藏起来然后又显示时，它都将试着再次获取其内容的最新数据。

28.2.3 利用 BorderContainer 管理布局区域

页面头部中列出的下一个布局窗口部件是 dijit.layout.BorderContainer。这个窗口部件能够让我们轻易地构造一个带有页眉、页脚、左右侧边栏以及中心内容区域的布局。下面的标记开始声明这样的一个窗口部件：

```
<div id="main" dojoType="dijit.layout.BorderContainer"
    design="headline"
    persist="true"
    liveSplitters="false">
```

第一个可选属性 design 可以取下面两个值之一：

- **headline(默认值)** 顶部的页眉和底部的页脚横跨整个布局。可以将这个形状想象为字母“I”。
- **sidebar** 左右侧边栏扩展到布局的顶部和底部，压住页眉和页脚区域。可以将这个形状想象为字母“H”。

下一个属性 persist 关注的是拆分器。区域之间的每个边界均可以指定一个可拖动的分割器，允许用户自定义布局的比例。persist 属性会让这个窗口部件将拖动的拆分器位置记录到 cookie 中，这样当再次访问该应用程序时就可以恢复用户自定义的布局状态。

对于拆分器，liveSplitters 属性可在两种拆分器样式之间切换。当该属性设置为 true(默认值)时，拆分器将使布局在调整大小时实时更新自身。按比例调整大小的元素将变大或缩小，而文本

会在容器内流回。在处理简单内容时，这可以为用户提供响应非常灵敏的反馈。

但是，如果这些区域中包含复杂的、在调整大小时需要耗费大量计算的次级布局，那么应该将 liveSplitters 设置为 false。在这种模式下，只有等到用户停止拖动拆分器时才会调整布局，而在拆分器移动的过程中会提供它的一个替代对象。

下面为这个窗口部件定义几个内容区域：

```
<div id="header" class="box"
    dojoType="dijit.layout.ContentPane"
    region="top">
    <h1>Hello Dojo Layouts</h1>
</div>

<div id="footer" class="box"
    dojoType="dijit.layout.ContentPane"
    region="bottom">
    <p>This is a footer!</p>
</div>

<div id="maincontent" class="box"
    dojoType="dijit.layout.ContentPane"
    region="center">
    <h3>This is center content</h3>
    <p>
        Cras ut mauris vitae nisl mattis vulputate. Duis urna
        pede, iaculis vitae, tristique a, tempor eget, leo.
    </p>
</div>
```

在上面的标记中，BorderContainer 的每个内容区域都是 dijit.layout.ContentPane 的简单实例，但是在本章稍后将会看到，这些实例几乎可以是任何其他布局窗口部件。

对于这里的每个 ContentPane 窗口部件，关于 BorderContainer 的最重要的功能是 region 属性。它可以取如下的值：

- center　这种类型的区域至少要有一个，它将放在布局的中心位置，所有其他类型的区域都将围绕它。
- top　这个区域将在布局顶部形成页眉。
- bottom　这个区域将构成布局底部的页脚。
- left　这个区域将作为布局左侧上的侧边栏。
- right　这个区域将作为布局右侧上的侧边栏。
- leading　根据文本在布局中的流动方向(在英语中是从左到右，而在希伯来语中是从右到左)，这个区域将出现在文本开始的位置。
- trailing　根据文本在布局中的流动方向(在英语中是从左到右，而在希伯来语中是从右到左)，这个区域将出现在文本结束的位置。

除了 region 属性，BorderContainer 还会查看它内部的窗口部件上的一些属性：

```
<div id="sidebar" class="box"
    dojoType="dijit.layout.ContentPane"
```

```
            region="left"
            splitter="true"
            minSize="160"
            maxSize="300">

            <p>This is the sidebar!</p>

        </div>
    </div>
```

前面曾经提到过，BorderContainer 窗口部件能够在区域之间提供可拖动的拆分器。对于除 center 之外的任何 region 属性值，将 splitter 属性设置为 true 就可以实现这一点。与 splitter 属性相伴的还有 minSize 和 maxSize 属性。这些属性在拆分器上为该区域允许的最小尺寸和最大尺寸定义了约束条件。

图 28-13 演示了当把上一个示例中声明的 BorderContainer 窗口部件拖进页面的正文中时，它就会显示出来。

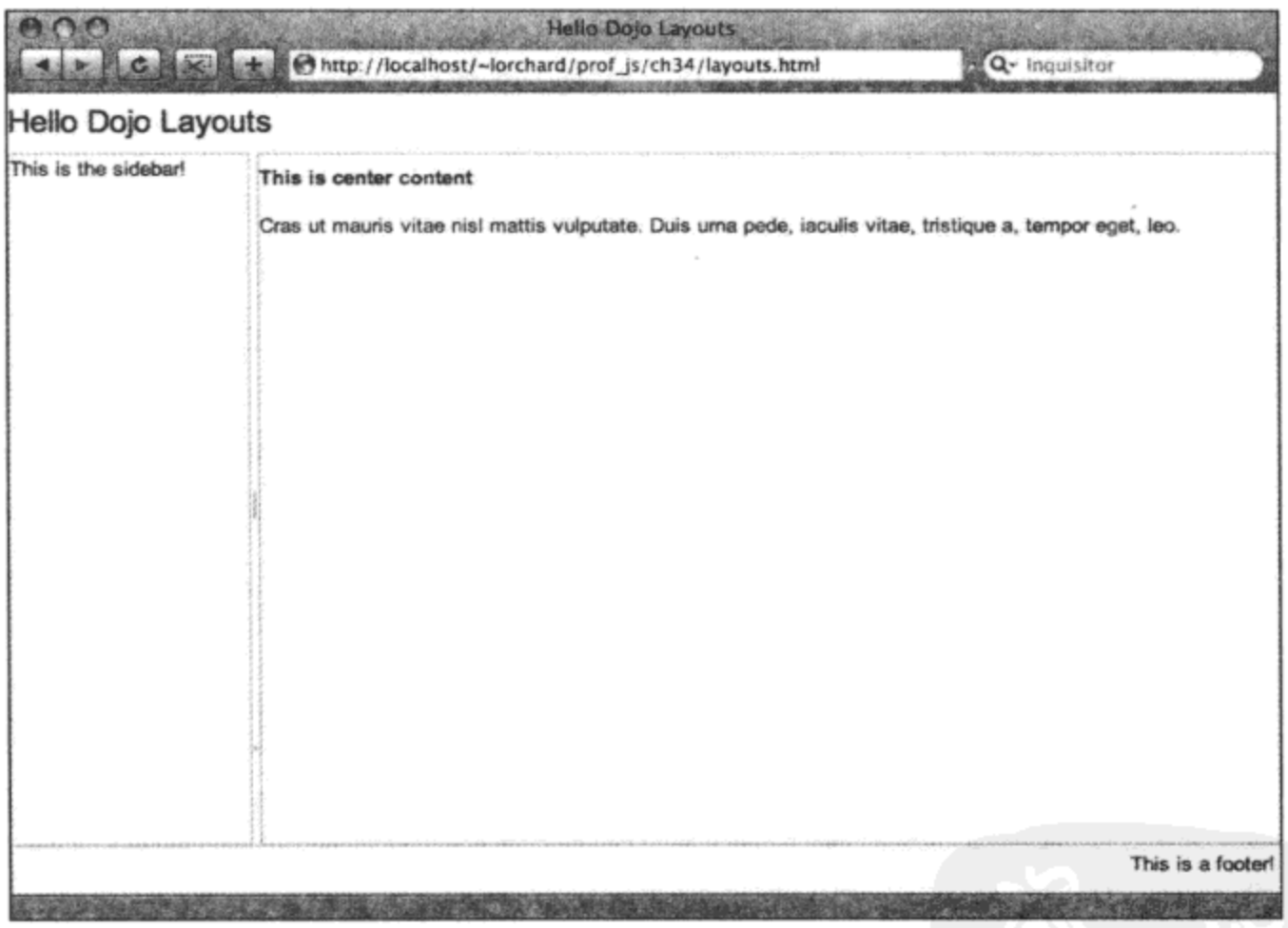


图 28-13

28.2.4 利用 StackContainer 管理内容可见性

BorderLayout 窗口部件对于组织带有页眉和侧边栏的布局非常有用。但除了将空间划分成若干区域之外，通常可以通过将不同的元素组整体换入和换出的方式来管理空间。

在 Dijit 中实现这种方式的基本构建块是 dijit.layout.StackContainer，该类可以作为多个提供相关功能的窗口部件的父类。但 StackContainer 可以自包含使用，如同下面的标记所示：

```
<div id="maincontent" class="box"
```

```
dojoType="dijit.layout.ContentPane"
region="center">

<h2>StackContainer ahoy!</h2>

<div id="mainstack"
    dojoType="dijit.layout.StackContainer">

    <div class="content"
        dojoType="dijit.layout.ContentPane"
        title="Page 1">
        <h3>Page 1</h3>
        <p>Proin suscipit gravida quam. Quisque nec enim.</p>
    </div>

    <div class="content"
        dojoType="dijit.layout.ContentPane"
        selected="true"
        title="Page 2">
        <h3>Page 2</h3>
        <p>Phasellus suscipit mollis turpis.</p>
    </div>

    <div class="content"
        dojoType="dijit.layout.ContentPane"
        title="Page 3">
        <h3>Page 3</h3>
        <p>Pellentesque consectetuer, ligula eget adipiscing pharetra</p>
    </div>

    <div class="content"
        dojoType="dijit.layout.ContentPane"
        title="Page 4">
        <h3>Page 4</h3>
        <p>Magna magna suscipit nibh, et fermentum massa ante a tortor.</p>
    </div>

</div>
```

与 BorderContainer 类似，StackContainer 管理几个 ContentPane 子窗口部件。每次只有其中的一个子窗口部件可见，如同一叠卡片中只有最上方的卡片能够显示一样。

StackContainer 并没有现成的导航控件，但有一个名为 dijit.layout.StackController 的伴随窗口部件，可以将它连接起来以提供按钮，这些按钮管理栈中的当前可见卡片：

```
<div dojoType="dijit.layout.StackController"
    containerId="mainstack">
</div>
```

StackController 是一个功能非常少的窗口部件，它主要的用途是作为更高级和更复杂的窗口部件的父类。它有一个名为 containerId 的属性，该属性引用的是计划控制的 StackContainer 的 ID。

但 StackController 本身只提供了一组简单的按钮，栈中的每个页面对应一个按钮。当单击适当的按钮时，对应的页面就会显示出来，而所有其他页面都会隐藏起来。当向该栈中添加新的子窗口部件时，StackController 就会自动添加新按钮。

为 StackContainer 构建分页控件

为了练习 StackController 窗口部件的其他几个功能，我们再使用一些原型用户界面来增强这个控件。以下面的标记作为开始：

```
<div id="stackpaging" class="content"
    containerId="mainstack"
    dojoType="dijit.layout.ContentPane">

    <a href="#" class="prev">&lt; prev</a> <span>--</span>

    <span class="currpage"></span> <span>--</span>

    <a href="#" class="next">next &gt;</a> <span>--</span>

    <a href="#" class="add">add child</a> <span>--</span>

    <a href="#" class="remove">remove child</a>
```

这段标记声明一个新的 ContentPane 窗口部件，它包含如下简单的元素：

- 一个用来导航到前一张卡片的链接
- 一个用来指示当前可见卡片的标题的元素
- 一个用来导航到下一张卡片的链接
- 一个用来添加新卡片的链接
- 一个用来移除当前卡片的链接

建立用户界面的各个部分之后，现在开始将它们连接起来：

```
<script type="dojo/method">

    var root_id      = this.srcNodeRef.id;
    var container_id = this.srcNodeRef.getAttribute('containerId');

    var stack = dijit.byId(container_id);

    dojo.query('.prev', root_id)
        .onclick(function(ev) {
            dojo.stopEvent(ev);
            stack.back();
        });

    dojo.query('.next', root_id)
        .onclick(function(ev) {
            dojo.stopEvent(ev);
            stack.forward();
        });
```

上述代码的开头是 dojo.parser 能够理解的另一种自定义<script>标记。这一次它的类型为“dojo/method”，但没有设置 event 属性：在创建包含该脚本的窗口部件之后，dojo.parser 就会立即执行包含在这类<script>标记内的代码。

虽然这一组用户界面元素本身并不是新的窗口部件类别，但这里的代码试图遵循一些窗口部件约定：从标记的属性中同时获取容器窗口部件自身的 ID 和 StackContainer 的 ID。

此外，注意我们使用 dijit.byId()获取一个指向窗口部件实例的引用，而不是使用 dojo.byId()，它返回的是 DOM 节点引用。

在脚本块内部，可以使用作用域变量 this 指向刚刚创建的窗口部件。基于这一点，Dijit 窗口部件的基础属性之一是 srcNodeRef，它提供了一个指向原始 DOM 节点的引用。用户界面元素本身通过类名来标识；通过使用窗口部件的根 ID 来调用 dojo.query()可以方便地找到这些元素。

这些安排组成了一种原型窗口部件，可以将其轻易地移植到自身的多个实例以及将来的其他 StackController 实例。如何充分地将这种原型窗口部件融进一个适当的 Dijit 窗口部件超出了本章的介绍范畴，但可以将其作为以后研究的一个高级主题。

结束前面的代码，向前和向后导航链接都已经连接起来，单击事件则可通过分别调用 StackContainer 的.back()和.forward()方法来处理。这两个方法遍历栈中的卡片，将当前卡片隐藏起来，并显示列表中的前一张或下一张卡片。

接下来，下面的代码引入了 StackContainer 窗口部件发布的事件主题：

```
dojo.subscribe(
    container_id + '-startup',
    function(params) {
        if (params.selected) {
            var title = params.selected.title;
            dojo.query('.currpage', root_id)
                [0].innerHTML = title;
            console.log("Starting up, selected child '" + title + "'");
        }
    }
);

dojo.subscribe(
    container_id + '-selectChild',
    function(page) {
        dojo.query('.currpage', root_id)
            [0].innerHTML = page.title;
        console.log("Selected child '" + page.title + "'");
    }
);
```

前面给出的代码块连接用户界面以使 StackContainer 逐页遍历它的内容，而这个示例中的代码则连接事件主题订阅者以对 StackContainer 初始唤醒并随后遍历其内容做出响应。这段代码中的两个事件订阅者都记录日志消息以响应 StackContainer 中的可见性变化，同时更新用户界面中的当前页面指示器。

StackContainer 窗口部件除了提供可供调用的方法来控制它之外，它还发布了几个事件主题来宣告它正在执行的操作。由于所有这些事件主题都是作为全局事件主题发布的，因此我们非常容

易将多个松散耦合的处理程序连接起来，对这个窗口部件中的变化做出响应。事件主题包括如下几种：

- {id}-startup　当窗口部件唤醒时，它发布这个事件主题，这个事件中包含了一个详细列举了该窗口部件的子卡片的对象，而且当前可选中。
- {id}-selectChild　当某张子卡片可见时，就会发布这个事件主题，并提供一个指向这个选中子卡片的引用。
- {id}-addChild　当添加一张新的子卡片时发布这个事件主题，并提供一个指向这张新的子卡片的引用以及插入位置的数字索引(作为参数)。
- {id}-removeChild　当移除某张子卡片时发布这个事件主题，并提供一个指向移除卡片的引用。

注意，上述列表中的字符串{id}应该替换成 StackContainer 窗口部件的 ID。

下面的代码处理卡片栈的添加和移除操作：

```
dojo.query('.add', root_id)
    .onclick(function(ev) {
        dojo.stopEvent(ev);
        var widget = new dijit.layout.ContentPane({
            title: "child " + parseInt(Math.random() * 1000),
            href: 'random.php'
        });
        stack.addChild(widget, 0);
        stack.selectChild(widget);
    });
```

```
dojo.query('.remove', root_id)
    .onclick(function(ev) {
        dojo.stopEvent(ev);
        var widget = stack.selectedChildWidget;
        stack.removeChild(widget);
    });
```

这段代码的第一部分将“添加”链接连接起来，以编程方式创建新的 ContentPane 窗口部件，从一个名为 random.php 的资源那里动态加载内容。此外，我们还为这个窗格提供了一个随机的唯一标题。我们使用 StackContainer 的.addChild()方法将这个新的窗格插入到列表的表头，并调用.selectChild()方法让这个新添加的窗格可见。

这段代码的第二部分将“移除”链接连接起来，利用 StackContainer 的 selectedChildWidget 属性获取当前可见卡片，然后调用.removeChild()方法将其从栈中移除。

下面的代码提供了几个订阅者来对上面的操作做出响应：

```
dojo.subscribe(
    container_id + '-addChild',
    function(page, index) {
        console.log("Added child '" + page.title + "' at " + index);
    }
);
```

```
        dojo.subscribe(
            container_id + '-removeChild',
            function(page) {
                console.log("Removed child '" + page.title + "'");
            }
        );

    </script>

</div>
```

前面曾经描述过，每当向 StackContainer 的栈中添加卡片或从中移除卡片时，它都会发布事件主题{id}-addChild 和{id}-removeChild。上面的代码简单地创建了几个订阅者来将这些活动记录到日志中。

为了使用一些不断变化的内容演示这些功能，下面给出了 href 属性引用的 random.php 资源的具体实现，在前面创建 ContentPane 窗口部件时曾经用到该资源：

```
<?php
// Wait for a number of seconds to fake work.
if ($_GET['wait']) {
    sleep($_GET['wait']);
}

// Take a list of words and shuffle them all up.
$words = explode(" ",
    "Lorem ipsum dolor sit amet, consectetuer adipiscing elit. ".
    "Maecenas porta, sapien bibendum adipiscing rutrum, ".
    "est nunc congue nibh, nec faucibus magna turpis sed arcu."
);
shuffle($words);
?>
<ul>
    <?php if ($_GET['dojo_preventCache']): ?>
        <li>preventCache: <?php echo $_GET['dojo_preventCache'] ?></li>
    <?php endif ?>
    <?php foreach ($words as $word): ?>
        <?php if (rand(0, 10) > 5 ): ?>
            <li><?php echo $word ?></li>
        <?php endif ?>
    <?php endforeach ?>
</ul>
```

这段 PHP 脚本接受一个参数?wait，它将导致一段几秒的延时，这样就可以在演示过程中伪造服务器处理时间并把加载消息显示出来。这段脚本的剩余部分构建一个从样本段落中随机挑选出来的单词组成的项目符号列表。将这段代码留作备用，在本章剩余内容中演示动态内容时它会非常有用。

图 28-14 给出了上述代码在浏览器中的运行结果的预览。在这个界面上执行一些操作之后，事件订阅者将会产生类似下面的日志文本消息。

```
Added child 'child 290' at 0
Selected child 'child 290'
GET http://localhost/~lorchard/prof_js/ch34/random.php
Added child 'child 904' at 0
Selected child 'child 904'
Selected child 'Page 2'
Removed child 'Page 2'
Selected child 'child 904'
GET http://localhost/~lorchard/prof_js/ch34/random.php
Added child 'child 596' at 0
Selected child 'child 596'
Selected child 'child 904'
Removed child 'child 904'
Selected child 'child 596'
```

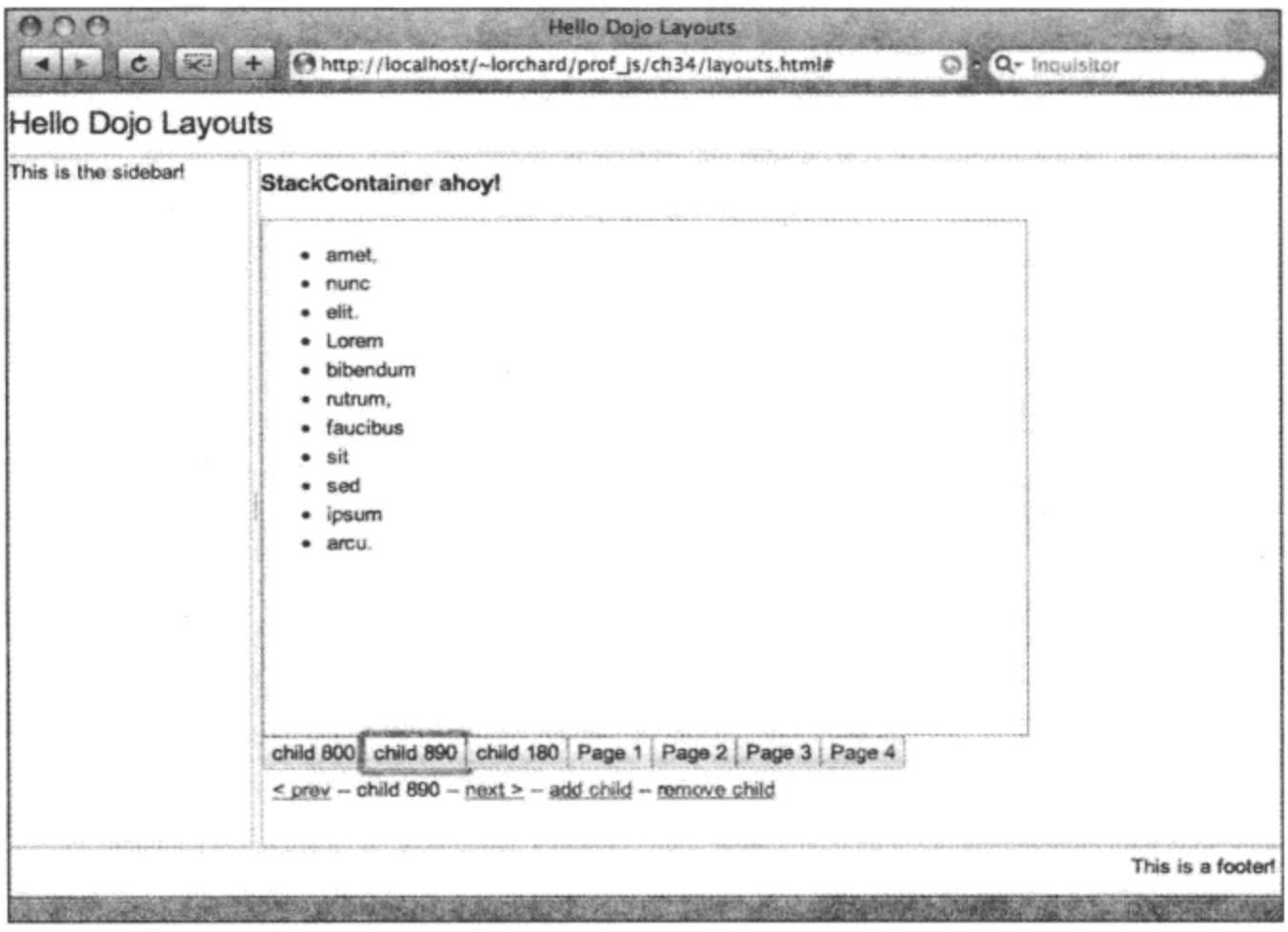


图　28-14

您应该能够很好地了解在导航和管理 StackContainer 的内容期间引发的一系列事件。

此外，在本章剩下的内容中阅读更多有关布局窗口部件的过程中，要记住 StackContainer 是其中几个窗口部件的父类，这也就意味着上面的内容也均适用于它们，包括操作方法以及可供订阅的事件主题。

28.2.5　利用 AccordionContainer 切换内容窗格

另一个可用来管理可见性的窗口部件是 dijit.layout.AccordionContainer。这是 StackContainer 的一个子类，它提供垂直的内容窗格栈，任意时刻这些窗格中只有一个可见。其他窗格则折叠成标题栏，当单击这些标题栏时，可见的窗格会隐藏起来，而单击的窗格则会在一段组合动画中展开。

考虑下面的标记，它用来替换前一个示例中 BorderLayout 窗口部件的左侧侧边栏的<p>内容。

```
<div id="sideaccord"
    dojoType="dijit.layout.AccordionContainer">

    <div id="side1" dojoType="dijit.layout.AccordionPane"
        title="Pane #1">
        <ul>
            <li>foo</li>
            <li>baz</li>
        </ul>
    </div>

    <div id="side2" dojoType="dijit.layout.AccordionPane"
        loadingMessage="Loading content..."
        href="random.php?wait=1"
        preload="true"
        preventCache="false"
        refreshOnShow="false"
        title="Pane #2 (dynamic content)">
    </div>

    <div id="side3" dojoType="dijit.layout.AccordionPane"
        title="Pane #3">
        <ul>
            <li>bar</li>
            <li>baz</li>
        </ul>
    </div>

    <div id="side4" dojoType="dijit.layout.AccordionPane"
        title="Pane #4"
        <ul>
            <li>foo</li>
            <li>bar</li>
            <li>baz</li>
        </ul>
    </div>

</div>
```

这个示例中给出的每个窗格的 title 属性用于可单击的折叠窗格标题栏。此外还要注意的是，dijit.layout.AccordionPane 窗口部件是 ContentPane 的一个子类，经过细微的调整，以便能够在这种折叠形式导航元素中使用。可以在这里包括其他非布局窗口部件(AccordionContainer 自身的一项限制)并使用 ContentPane 接受的所有其他基本属性。

另一件值得回顾的事情是用来加载动态内容的 href 属性。由于 AccordionContainer 窗口部件将内容整体地隐藏和显示，因此 ContentPane 的 refreshOnShow 属性就可以派上用场。因为折叠窗格的第二个窗格加载动态内容(refreshOnShow 设为 true)，因此我们应该能够在各种窗格之间导航，并观察到每次返回到第二个窗格时，它的内容都经过重新加载。

图 28-15 给出了 AccordionContainer 的呈现结果，这一次同样将 BorderContainer 父容器的 design 属性切换成“sidebar”，目的是为折叠窗格元素提供更多的垂直空间。注意，这里已经将页眉和页脚移走，以便让侧边栏展开到窗口的顶部和底部边框。

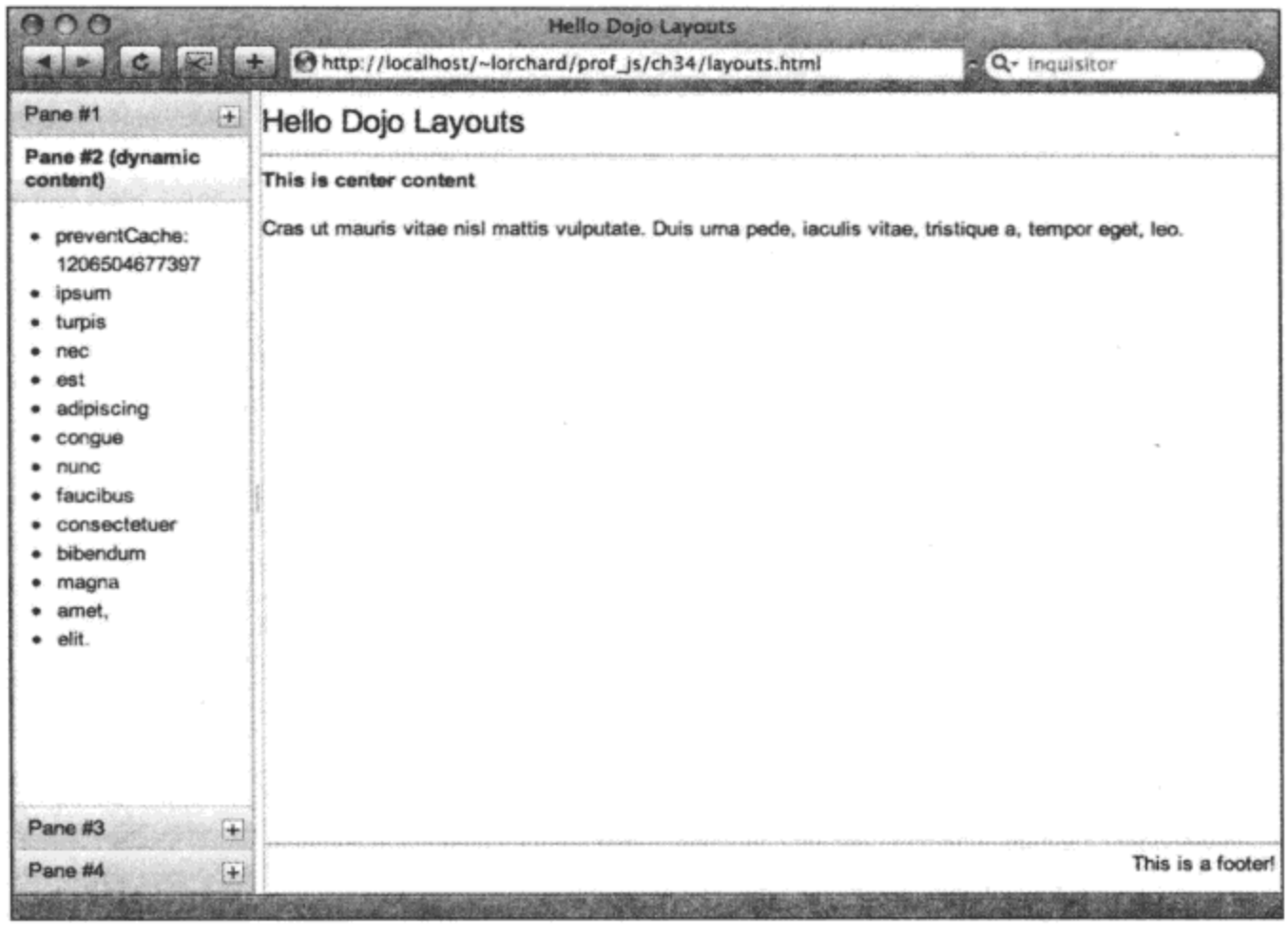


图 28-15

28.2.6 利用 TabContainer 构建选项卡式的内容窗格

AccordionContainer 最适合于管理像侧边栏这样的垂直空间中的窗口隐藏和显示，而 StackContainer 窗口部件的子类 dijit.layout.TabContainer 则更适合于管理布置在水平或垂直栏中的带有可单击选项卡的较宽空间。

考虑下面的标记，它声明了一个 TabContainer 窗口部件，用来替换本章前面的 BorderContainer 中的中心区域 ContentPane：

```
<div id="maincontent"
    dojoType="dijit.layout.TabContainer"
    tabPosition="bottom"
    region="center">

    <div class="content"
        dojoType="dijit.layout.ContentPane"
        title="Tab #1">
        <h3>Praesent aliquet</h3>
        <p>
            Cras ut mauris vitae nisl mattis vulputate. Duis urna
            pede, iaculis vitae, tristique a, tempor eget, leo.
        </p>
```

```
    </div>

    <div class="content"
        dojoType="dijit.layout.ContentPane"
        closable="true"
        selected="true"
        title="Tab #2 (closable)">
        <h3>Duis urna pede</h3>
        <p>
            Nam imperdiet, lectus sit amet feugiat adipiscing,
            turpis mi ornare enim,
        </p>
    </div>

    <div class="content"
        dojoType="dijit.layout.ContentPane"
        loadingMessage="Loading content..."
        href="random.php?wait=1"
        preventCache="true"
        refreshOnShow="true"
        title="Tab #3 (dynamic content)">
    </div>

</div>
```

注意，TabContainer(包含 region="center"属性)可以直接将 BorderContainer 中的 ContentPane 替换。此外，TabContainer 窗口部件接受一个名为 tabPosition 的属性，它可以取下列值之一：

- top　在窗口部件顶部沿着水平方向摆放选项卡。
- bottom　在窗口部件底部沿着水平方向摆放选项卡。
- left-h　一个垂直的选项卡条出现在左侧。
- right-h　一个垂直的选项卡条出现在右侧。

TabContainer 的选项卡子窗口部件是 ContentPane 窗口部件。每个选项卡窗口部件提供的 title 属性将变成显示在可单击选项卡中的标题。第一个选项卡非常简单，它只提供一个 title 和一些内容。

但第二个选项卡提供了一对新属性：

- closable　如果设置为 true，那么这个选项卡可以关闭并从视图中整体移除。
- selected　当加载页面时，第一个将该属性设置为 true 的选项卡将变得可见。否则，这个列表中的第一个选项卡将变得可见。

最后，选项卡集合中的第三个选项卡加载动态内容，就像前一节中的折叠窗格部分中的窗口部件一样。您应该可以观察到类似的行为：从动态内容窗格切换出去，然后再返回到其中时，它将重新加载其内容。

图 28-16 给出了前面的标记作为窗口部件产生的结果。

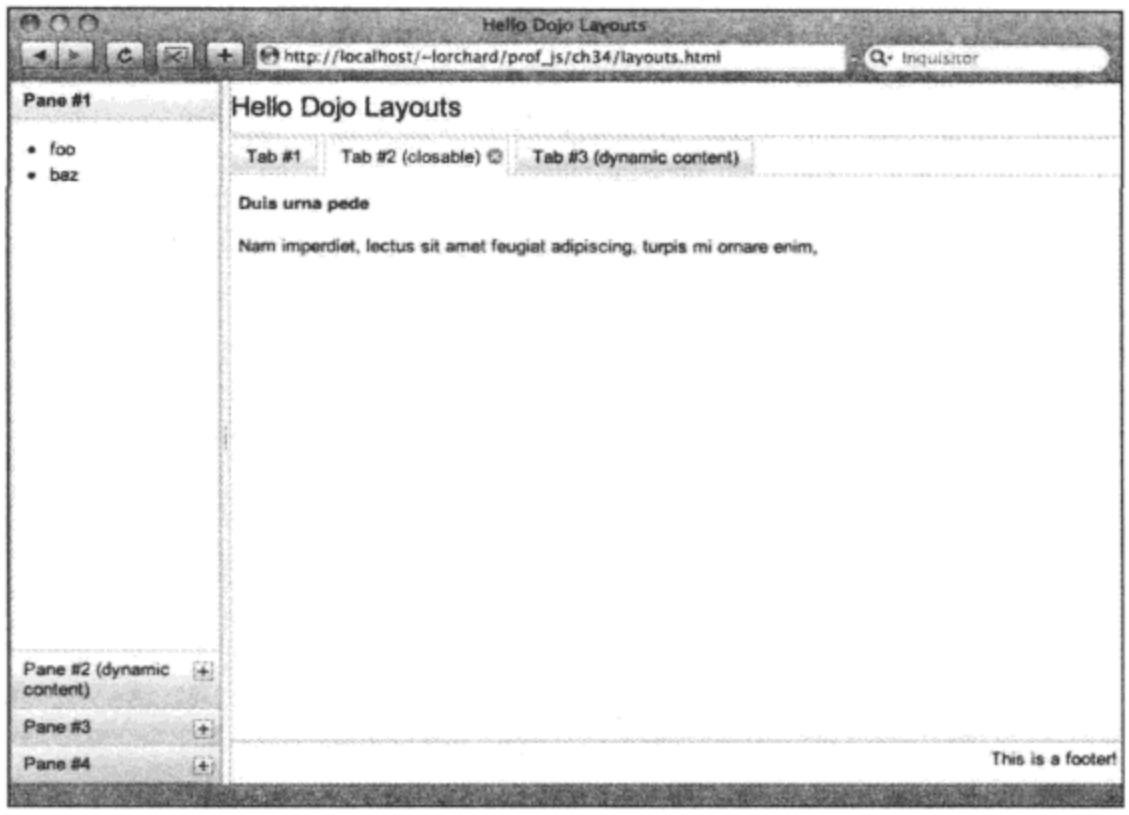


图　28-16

28.2.7　利用 SplitContainer 划分布局区域

如果您比较欣赏使用 BorderContainer 提供的可拖动区域边框，那么可能喜欢下面这个名为 dijit.layout.SplitContainer 的窗口部件，它唯一的作用就是提供可拖动的划分边框。考虑下面的标记，它的作用是为前一节的 TabContainer 窗口部件提供一个新的选项卡：

```
<div id="splittab"
    dojoType="dijit.layout.SplitContainer"
    orientation="vertical"
    activeSizing="true"
    persist="false"
    title="Tab #4 (SplitContainer)">

    <div class="content"
        dojoType="dijit.layout.ContentPane"
        sizeShare="15"
        sizeMin="65">
        <h3>Another section</h3>
        <p>Lorem ipsum dolor sit amet, consectetuer adipiscing elit.</p>
    </div>

    <div class="content"
        dojoType="dijit.layout.ContentPane"
        sizeShare="55"
        loadingMessage="Loading content..."
        href="content.html"
        preventCache="true">
    </div>

    <div class="content"
```

```
            dojoType="dijit.layout.ContentPane"
            sizeShare="30">
            <h3>Duis urna pede</h3>
            <p>
               Nam imperdiet, lectus sit amet feugiat adipiscing,
               turpis mi ornare enim,
            </p>
         </div>

      </div>
```

上面的标记中演示的 SplitContainer 窗口部件接受以下的属性来控制它的可拖动边框的行为和布局：

- orientation　这个值可以是 horizontal(默认值)或 vertical，它指定应该沿着哪个方向摆放划分区域。与 BorderContainer 不同的是，SplitContainer 只允许沿着一个方向摆放划分区域。
- activeSizing　与 BorderContainer 使用的 liveSplitters 属性类似，这个标志决定了是动态调整区域内容的大小，还是在拖放完成之前只显示一个替代对象。
- persist　如果该属性设为 true，就会把用户选中的区域大小保存到 cookie 中。

查看定义划分区域的子窗口部件，有几个属性可用来管理划分区域的相对大小：

- sizeMin　这是在调整大小时该区域所允许的最小尺寸。
- sizeShare　这个属性是一个比例值，在与所有其他划分区域的 sizeShare 值加在一起时，它代表了特定区域应该占用多少空间。

刚开始时，sizeShare 属性可能有点难以理解。在前面的标记中，该属性的所有值累加起来等于 100，因此它们类似于百分比。这是一种非常方便的管理方式，但从技术上讲，它只与总数和比例有关。除了初始页面加载之外，这个属性还允许在调整浏览器窗口大小时让 SplitContainer 以流动形式增大或缩小。

图 28-17 给出了上述标记中声明的窗口部件作为一个 TabContainer 窗格如何适应布局。

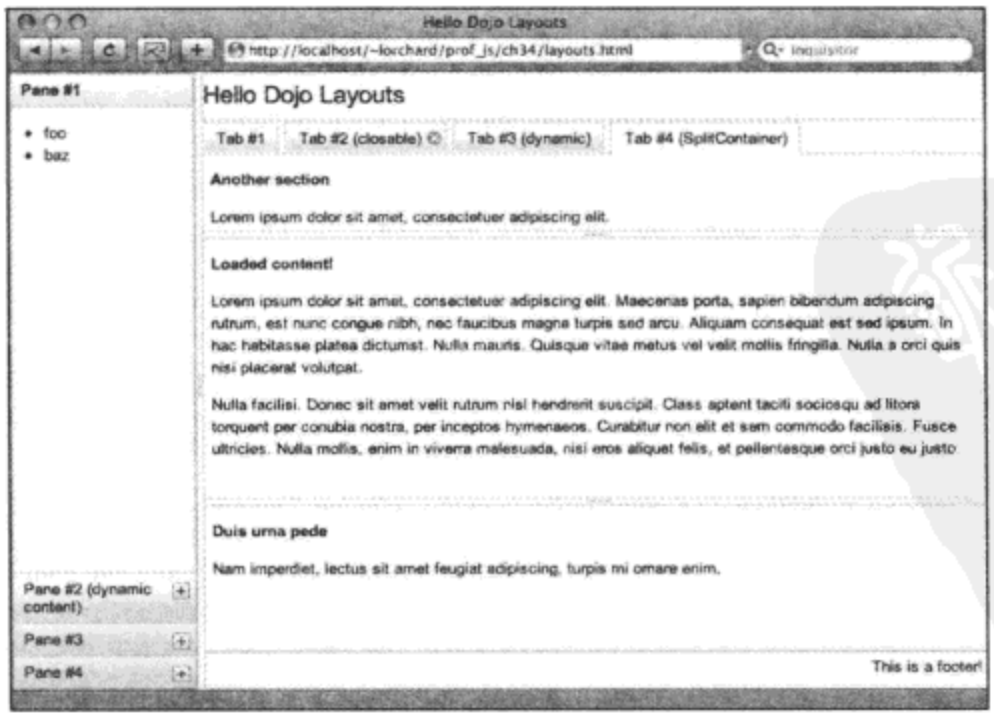


图　28-17

28.3 创建应用程序控件和对话框

利用 Dijit 窗口部件，我们已经看到如何构建带有内置验证处理机制和 CSS 样式挂钩的表单，并了解了如何使用自顶向下的布局窗口部件管理窗口空间。现在，我们要讲解剩下的与应用程序控制和反馈有关的窗口部件。

由于上一节中构建的布局已经变得相当复杂，因此我们使用下面的标记来提供一个新的起点，从而继续试验更多的窗口部件：

```
<html>
   <head>
      <title>Hello Dojo Widgets</title>

      <style type="text/css">
         @import "../dojodev/dojo/resources/dojo.css";
         @import "../dojodev/dijit/themes/tundra/tundra.css";
      </style>

      <script type="text/javascript" src="../dojodev/dojo/dojo.js"
         djConfig="parseOnLoad: true"></script>

      <script type="text/javascript">
         dojo.require("dojo.parser");
         dojo.require("dijit.Menu");
         dojo.require("dijit.Toolbar");
         dojo.require("dijit.form.Button");

         dojo.require("dijit.Dialog");
         dojo.require("dijit.ProgressBar");
         dojo.require("dijit.TitlePane");
      </script>

      <style type="text/css">
         h1, h2, h3 { margin: 0.5em 0 1em 0 }
         p  { margin: 0 0 1em 0 }
         hr { clear: both }
         #loader {
            padding: 0; margin: 0;
            top: 0; left: 0;
            position: absolute;
            width: 100%; height: 100%;
            background: #fff;
            z-index: 999;
         }
         .menu_target {
            padding: 1em; margin: 1em;
            width: 75px; float: left;
            border: 1px solid #333;
```

```
            background-color: #ddd;
        }
    </style>

</head>

<body class="tundra">

        <div id="loader"><h1>Loading widgets demo...</h1></div>
            <script type="text/javascript">
        dojo.addOnLoad(function() {
            var loader = dojo.byId('loader');
            loader.style.display = "none";
        });
    </script>

    <h1>Hello Dojo Widgets</h1>

    <!-- body content goes here -->

</body>

</html>
```

相对于其他使用 Dojo 和 Dijit 的页面，这段标记看上去相当简单。需要解释的一个新功能是这个页面的<body>部分中出现的第一个加载器元素。

根据加载速度的快慢不同，您可能开始注意到在窗口部件载入过程中出现“短暂的无样式内容”的情况。也就是说，您会看到页面上的内容处于应用 CSS 规则以及 dojo.parser 模块完成页面扫描以实例化窗口部件等不同阶段之间。

这个简单的表示正在加载的 元素展示了在加载阶段中的一种透明窗帘，直到 Dojo 通知所有内容加载完毕才会将这个加载器 元素变成不可见。我们本来可以让这种机制变得更加复杂：在页面加载结束时应用一种淡出效果，甚至可以在加载过程中给出某种加载进度提示信息，或者至少给出一个制作动画的旋钮；但出于演示的目的，我们让这个加载器尽可能保持简单。

28.3.1　构建可单击按钮并为其编写脚本

除了数据录入表单，任何用户界面的基础性构建块中都包括可单击按钮。回顾一下，在我们研究 Dijit 的表单窗口部件的过程中，为了简单起见以及演示如何绑定 onSubmit 处理程序来调用表单验证，我们只使用了普通的提交按钮。

事实证明，Dijit 确实提供了一组灵活的按钮，既可以用在表单中，也可以用于应用程序中。第一个这样的按钮以 dijit.form.Button 窗口部件为基础，其声明类似下面这样：

```
<button id="button1" dojoType="dijit.form.Button">
    Click me #1!
    <script type="dojo/method" event="onClick">
```

```
        console.log('Clicked '+this.id);
    </script>
</button>
```

这段标记声明了一个新的<button>元素，其 dojoType 设置为 dijit.form.Button。最好使用<button>元素而不是<input type="button">，这是因为前者要比后者更有利于实施 CSS 样式，在应用窗口部件主题时也是如此。

在<button>元素的正文中有一些文本，它们构成了这个按钮的标签。在此之后，有一个类型为“dojo/method”的<script>元素，它为该按钮的 onClick 方法实现了一个处理程序。这里的实现只是向日志中记录一条消息，但它作为演示已经足够。

这个按钮相当简单，但 Button 窗口部件确实接受其他一些属性：

```
<button id="button2" dojoType="dijit.form.Button"
    showLabel="false"
    iconClass="dijitEditorIcon dijitEditorIconCopy"
    onClick="console.log('Clicked '+this.id);">
    Click me #2!
</button>
```

在上面的标记中，iconClass 属性的值为富文本编辑器工具栏的窗口部件主题中图标的 CSS 类名。这会让“复制”操作对应的图标出现。此外，虽然在窗口部件声明正文中提供了一个标签，但如果将 showLabel 属性设置为 false，则会阻止显示该标签。

最终结果是一个仅由图标组成的、不含文本标签的按钮。此外还要注意的是，这一次在按钮内部并没有<script>元素，而是有一个 onClick 属性，它指定了当单击按钮时要执行的代码。

为了了解窗口部件主题的构造，下面给出了对应于前面使用的图标的“tundra”主题 CSS 代码：

```
.tundra .dijitEditorIcon {
      background-image: url('images/editor.gif'); /* editor icons sprite image */
      background-repeat: no-repeat;
      width: 18px;
      height: 18px;
      text-align: center;
}
.tundra .dijitEditorIconCopy { background-position: -72px; }
```

在本章后面我们将更详细地讨论窗口部件主题，但此处的代码提示我们在构建按钮时，我们能够做些什么事情来使用自己的图标。

下面的代码演示如何通过编程方式来创建一个新的按钮：

```
<button id="button3">Click me #3!</button>

<script type="text/javascript">
    dojo.addOnLoad(function() {
        var button3 = new dijit.form.Button({
            onClick: function(ev) {
                console.log('Clicked '+this.id);
            }
        }, dojo.byId('button3'));
```

```
    });
</script>
```

这里并没有新内容：可以在 JavaScript 代码中创建一个 dijit.form.Button 新实例(就像创建任何其他对象一样)，传入一组属性和一个指向 DOM 节点的引用，就像 dojo.parser 在扫描页面期间所做的工作一样。

图 28-18 显示了所有这 3 个按钮在浏览器中的外观。

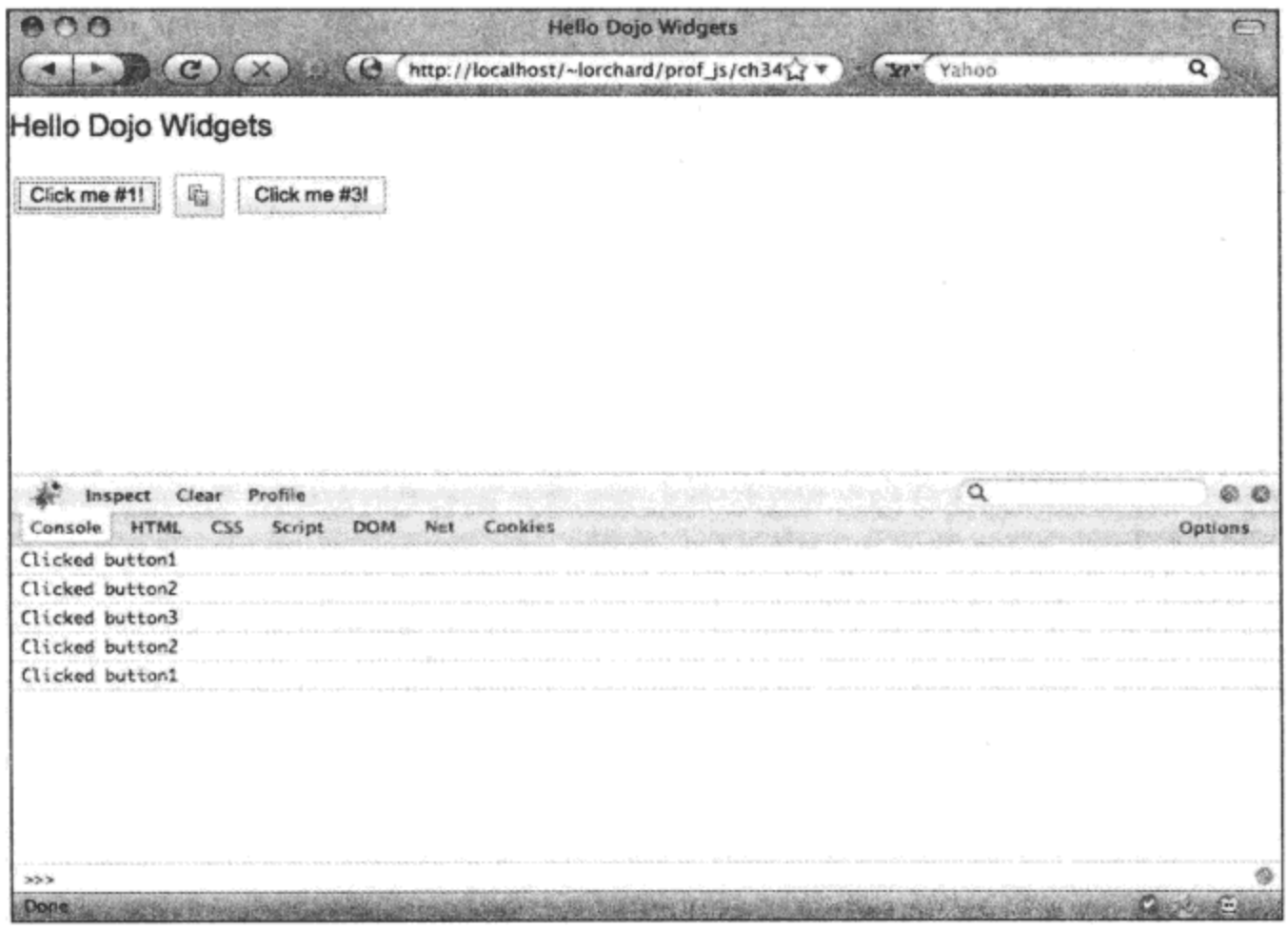


图　28-18

28.3.2　编排弹出式上下文菜单

按钮已经由浏览器以不同的原生形式提供，只是有的按钮带有窗口部件或 CSS 样式增强，而有的按钮没有。但到目前为止，浏览器尚未提供任何类型的弹出式菜单结构。因此，Dijit 提供了一个窗口部件来满足这种层次的命令菜单需求。

下面的标记演示了如何声明一个这样的窗口部件：

```
<div id="menu_target1" class="menu_target">Target #1</div>
<div id="menu_target2" class="menu_target">Target #2</div>
<div id="menu_target3" class="menu_target">Target #3</div>

<div dojoType="dijit.Menu"
    targetNodeIds="menu_target1, menu_target2, menu_target3"
    contextMenuForWindow="false"
    style="display: none">
```

```
<div id="foo_item" dojoType="dijit.MenuItem">
    <span>Foo</span>
    <script type="dojo/method" event="onClick">
        console.log('Menu clicked: '+this.id);
    </script>
</div>
```

在这段标记中首先要注意的是，这里有 3 个 元素，每个都带有 CSS 类 menu_target。紧跟着是一个窗口部件的声明，它的 dojoType 设置为 dijit.Menu，其第一个属性 targetNodeIds 列出了前 3 个 节点的 ID。

前面曾经演示过，targetNodeIds 属性接受一个由节点 ID 构成的列表，这个弹出式菜单将要附加到这些节点上。当用户在这些元素之一上右击时，声明的菜单就会激活。

在这个属性之后，contextMenuForWindow 属性设置为“false”。与 targetNodeIds 属性的行为相反，如果 contextMenuForWindow 属性设置为“true”，那么在任何地方右击都会弹出这个菜单，从而重写该窗口的浏览器原生上下文菜单。

作为这个属性列表中的最后一项(但绝非最不重要的一项)，您可能已经注意到这个元素的样式设置为不可见。如果没有这样做，那么这个上下文菜单在初始时将出现在页面布局中，而这通常并不是期望的行为。

一旦建立 Menu 窗口部件，下面就要声明一个 dijit.MenuItem 实例。注意，这个窗口部件与 dijit.Button 窗口部件非常类似，它接受标记中的一个标签或者作为一个属性，而且可以使用自定义<script>块或属性为其指定一个 onClick 处理程序。

一个 Menu 窗口部件可以包含任意多个 MenuItem 实例。但是，较深层次的菜单结构来自于 MenuItem 子类，就像下面这个菜单一样：

```
<div dojoType="dijit.PopupMenuItem">
    <span>Edit</span>

    <div dojoType="dijit.Menu" id="context_edit">

        <div dojoType="dijit.MenuItem" label="Copy"
            onClick="console.log('Menu clicked: '+this.id);"
            iconClass="dijitEditorIcon dijitEditorIconCopy"></div>

        <div dojoType="dijit.MenuItem" label="Paste"
            onClick="console.log('Menu clicked: '+this.id);"
            iconClass="dijitEditorIcon dijitEditorIconPaste"></div>

    </div>
</div>
```

这个 dijit.PopupMenuItem 实例引入了一个新项，但它不能单击，而是对鼠标翻转做出响应，将它包含的隐藏 dijit.Menu 窗口部件展现出来。此外与 Button 窗口部件类似，这个子菜单中包含的 MenuItem 窗口部件提供了两项内容，其中菜单项使用了 CSS 窗口部件主题提供的图像图标，

并通过 onClick 属性定义一个处理程序。

需要注意的是，这里的“复制”和“粘贴”菜单项均没有指定 ID，但它们的 onClick 处理程序引用了 ID。在创建时，这些 MenuItem 窗口部件将拥有它们各自的唯一 ID，因此当在这些菜单项上单击时，应该会在日志中看到它们。

继续研究下面稍微复杂的代码，这是一个具有自我修改功能的菜单。

```
<div dojoType="dijit.PopupMenuItem">
   <span>Windows</span>

   <div dojoType="dijit.Menu" id="context_windows">

      <div dojoType="dijit.MenuItem" id="new_window"
         label="New Window..."
         iconClass="dijitEditorIcon dijitEditorIconInsertImage">

         <script type="dojo/method" event="onClick">
            var rand = parseInt(Math.random() * 1000);
            console.log("Adding a new window item, window_"+rand);
            this.getParent().addChild(
               new dijit.MenuItem({
                  id: 'window_'+rand,
                  label: 'Window '+rand,
                  onClick: function() {
                     console.log('Removing clicked: '+this.id);
                     this.getParent().removeChild(this);
                  }
               })
            );
         </script>

      </div>
   </div>
</div>
</div>
```

与所有其他窗口部件类似，可以通过编程方式创建新的 MenuItem 窗口部件。此外，Menu 窗口部件提供了.addChild()和.removeChild()方法来修改菜单的内容。

上面示例中的标记可让用户通过单击“New Window ...”菜单项来添加新项，而当单击这些新项时，它们会将自己删除。尽管这里并没有进行练习，但我们还可以在 MenuItem 窗口部件上调用.setDisabled()方法，以设置该项是否可单击或者以灰色显示。

图 28-19 给出了完成之后的菜单在 Firefox 中的运行情况，这里同时还打开了 Firebug 日志文本消息来显示在与该菜单交互过程中产生的消息。

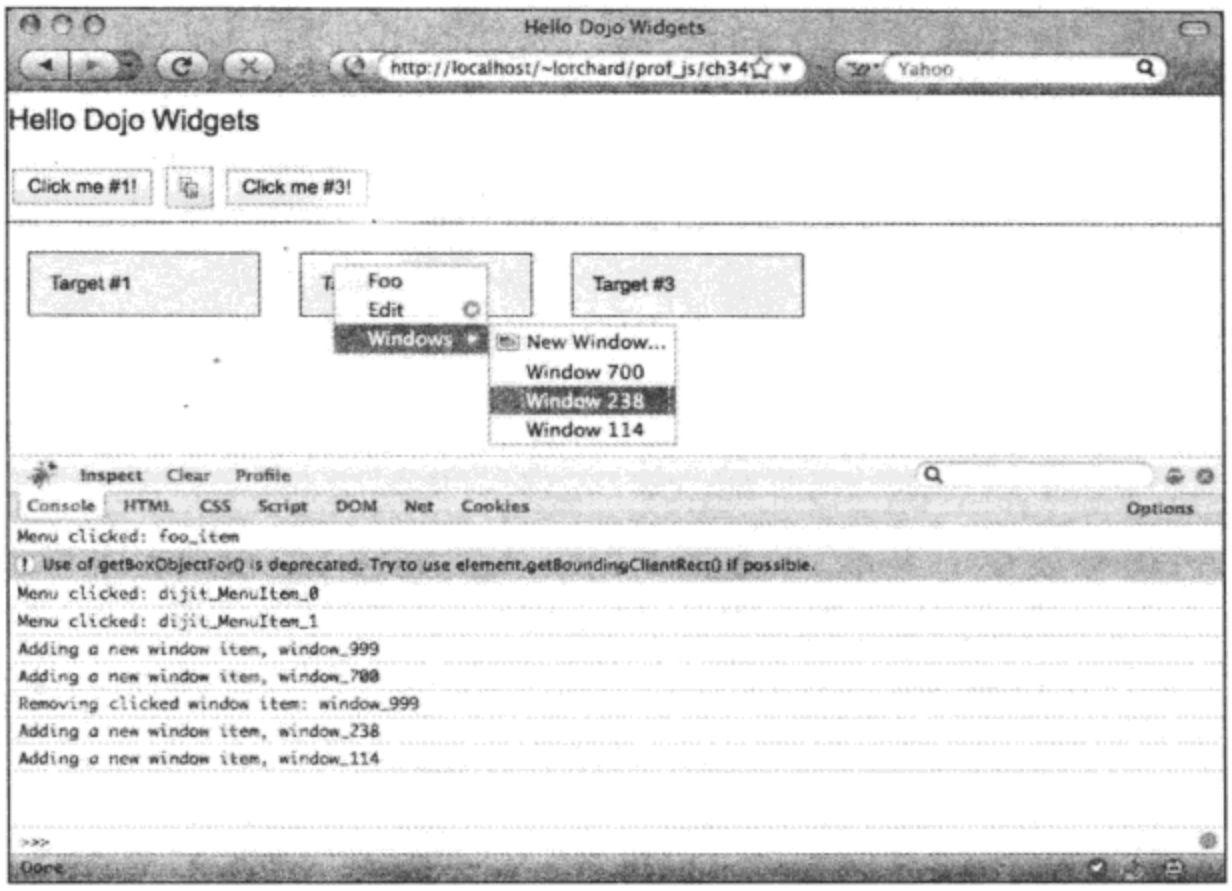


图　28-19

28.3.3　组合按钮和菜单

如果按钮和菜单都非常有用，那么能够生成菜单的按钮将会更有用。Dijit 提供了两种按钮来实现这个目的，下面的标记给出了其中的一种按钮：

```
<div dojoType="dijit.form.DropDownButton" id="edit_dropdown">
    <span>Edit</span>

    <div dojoType="dijit.Menu">

        <div dojoType="dijit.MenuItem" label="Copy"
            onClick="console.log('Menu clicked: '+this.id);"
            iconClass="dijitEditorIcon dijitEditorIconCopy"></div>

        <div dojoType="dijit.MenuItem" label="Paste"
            onClick="console.log('Menu clicked: '+this.id);"
            iconClass="dijitEditorIcon dijitEditorIconPaste"></div>

    </div>
</div>
```

当单击时，dijit.form.DropDownButton 窗口部件展开一个菜单，但它本身并不处理单击事件。下面的窗口部件确实处理单击事件，而且产生一个菜单：

```
<div dojoType="dijit.form.ComboButton" id="edit_combo"
    onClick="console.log('Menu clicked: '+this.id);">
    <span>Edit</span>
```

```
    <div dojoType="dijit.Menu">

        <div dojoType="dijit.MenuItem" label="Copy"
            onClick="console.log('Menu clicked: '+this.id);"
            iconClass="dijitEditorIcon dijitEditorIconCopy"></div>

        <div dojoType="dijit.MenuItem" label="Paste"
            onClick="console.log('Menu clicked: '+this.id);"
            iconClass="dijitEditorIcon dijitEditorIconPaste"></div>

    </div>

</div>
```

在上面的标记中，dijit.form.ComboButton 窗口部件的声明中的 onClick 属性证明，这个窗口部件处理它的主体内部的单击事件，就像普通的按钮一样。但它还在侧边提供一个区域，当单击该区域时，就会显示它包含的菜单。

28.3.4　利用按钮和菜单构建工具栏

并没有专门为提供水平样式应用程序菜单而构建的窗口部件。但有一个工具栏窗口部件，它可以包含目前给出的所有按钮：

```
<div dojoType="dijit.Toolbar">
<button dojoType="dijit.form.Button"
    showLabel="true">File</button>

<button dojoType="dijit.form.DropDownButton">
    <span>Edit</span>
    <div dojoType="dijit.Menu">

        <div dojoType="dijit.MenuItem" label="Copy"
            iconClass="dijitEditorIcon dijitEditorIconCopy"></div>

        <div dojoType="dijit.MenuItem" label="Cut"
            iconClass="dijitEditorIcon dijitEditorIconCut"></div>

        <div dojoType="dijit.MenuItem" label="Paste"
            iconClass="dijitEditorIcon dijitEditorIconPaste"></div>

    </div>
</button>

<div dojoType="dijit.ToolbarSeparator"></div>

<button dojoType="dijit.form.Button"
    iconClass="dijitEditorIcon dijitEditorIconBold"
    showLabel="true">Bold</button>
```

```
    <div dojoType="dijit.ToolbarSeparator"></div>

    <button dojoType="dijit.form.Button"
        iconClass="dijitEditorIcon dijitEditorIconCopy"
        showLabel="false">Copy</button>

</div>
```

这个 dijit.Toolbar 窗口部件要比简单的水平菜单栏窗口部件更加灵活，它能够容纳多个 Button 窗口部件。如果只是希望得到一个水平的应用程序菜单，那么可以通过仅仅使用 dijit.DropDownButton 窗口部件来构造一个这种菜单。还可以包含带有图标图像的按钮，并可以使用 dijit.ToolbarSeparator 窗口部件将类似的按钮组合在一起。

图 28-20 描绘了工具栏中的一组按钮，包括前一节中的 ComboButton 和 DropDown 按钮窗口部件。

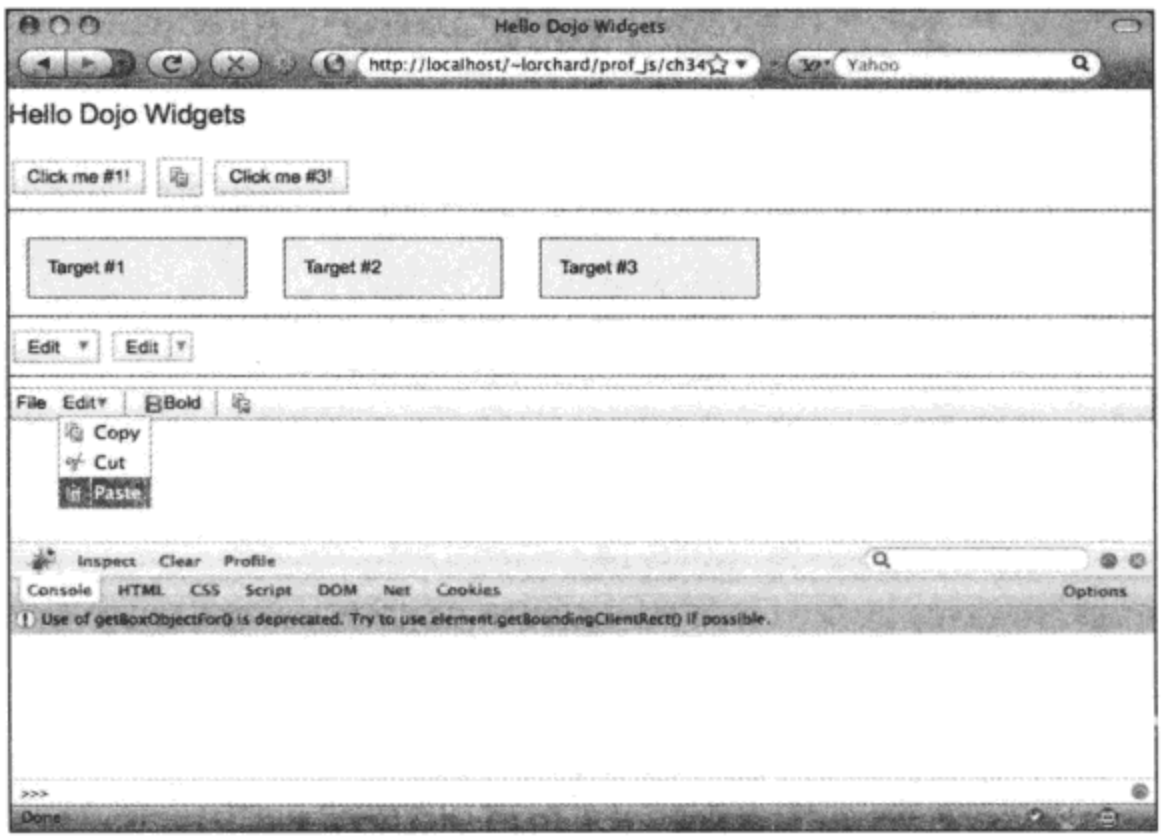


图 28-20

28.3.5 利用进度条提供完成进度反馈

当启动一个需要较长时间才能完成的操作或者从远程资源中加载一些数据时，如果能够提供某种制作动画的指示器来说明工作正在完成中，那么通常会比较有用。我们可以通过一个简单的动画 GIF 图像来完成这项工作，但更好的做法是提供某种精确的度量来展示完成的进度。为此，Dijit 提供了下面的窗口部件：

```
<div id="progress1" dojoType="dijit.ProgressBar"
    style="width: 250px;">
</div>
```

```
<button dojoType="dijit.form.Button">
    <span>Start thinking</span>
    <script type="dojo/method" event="onClick">
        dijit.byId('progress1').update({
            indeterminate: true
        });
    </script>
</button>
```

这段标记声明一个 dijit.ProgressBar 实例，这是一个用户界面元素，它应该与其他应用程序中的进度条类似。当单击上面示例中的按钮时，ProgressBar 窗口部件就会进入“不确定”状态，这意味着它将显示一个动画来指示任务正在完成过程中，但它并没有指示具体完成多少进度。这是许多应用程序中的“红白相间旋转招牌”样式。

下面这个窗口部件示例稍微复杂一些，这一次我们尝试对时间进度进行某种模拟度量：

```
<div id="progress2" dojoType="dijit.ProgressBar"
    style="width: 250px;">
</div>

<button dojoType="dijit.form.Button">
    <span>Start loading</span>
    <script type="dojo/method" event="onClick">

        var prog_done  = 0;
        var total_prog = 1000;

        this.setLabel("Loading...");
        this.setDisabled(true);

        var prog_int = setInterval(dojo.hitch(this, function() {
            prog_done += Math.random() * 100;

            dijit.byId('progress2').update({
                indeterminate: false,
                progress: prog_done,
                maximum:  total_prog,
                places:   2
            });

            if (prog_done >= total_prog) {
                clearInterval(prog_int);
                this.setLabel("Start loading");
                this.setDisabled(false);
            }
        }), 100);

    </script>
</button>
```

当单击这个示例中的按钮时，它启动一个以递增形式前进的定时器，经过一段时间之后就计

入一个随机量，直到它到达总值 1000。每次引发这个例程时，它都会调用 ProgressBar 窗口部件的.update()方法，并传入一组属性来报告以下信息：

- indeterminate　在这里该值为 false，这是因为进度是一个已知的范围。
- progress　这个值代表的是目前已经完成的工作。
- maximum　这个值代表的是结束时期望完成的工作。
- places　由于进度条显示的是一个 0~100 的数字百分比，因此这个值指定应该显示这个百分比的多少小数位。因此，如果给定值 2，则会看到类似 12.34%的百分比数字。

图 28-12 给出了 ProgressBar 窗口部件的两种形式的运行情况。

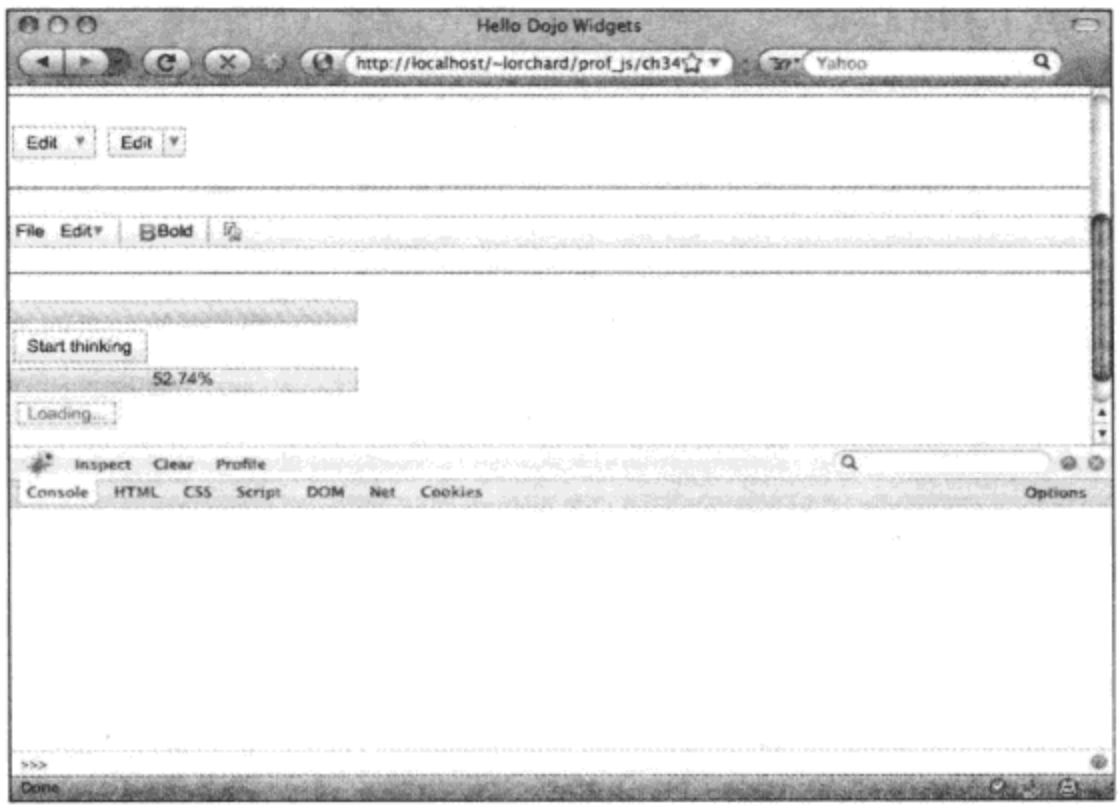


图　28-21

28.4　对窗口部件应用主题

Dojo 和 Dijit 提供了越来越多的基于 CSS 的主题选项，可用来在整个窗口部件选择中应用不同的外观。之所以能够做到这一点，是因为每个窗口部件都经过设计，可以提供一组丰富的 CSS 类名和标记模式，从而便于对它们的 CSS 外观进行完全控制。

28.4.1　检查窗口部件的 DOM 结构

例如，回顾 HTML 中一个按钮的声明标记：

```
<button id="button2" dojoType="dijit.form.Button"
    showLabel="false"
    iconClass="dijitEditorIcon dijitEditorIconCopy"
    onClick="console.log('Clicked '+this.id);">
    Click me #2!
</button>
```

一旦 dojo.parser 扫描完页面并实例化一个窗口部件来管理这个按钮，那么 DOM 结构将改写

成如下所示：

```
<div widgetid="button2"
    class="dijit dijitLeft dijitInline dijitButton dijitButton"
    dojoattachevent="onclick:_onButtonClick,onmouseenter:_onMouse,
      onmouseleave:_onMouse,onmousedown:_onMouse">
    <div class="dijitRight">
        <button aria-disabled="false" aria-valuenow="" title="Click me #2!"
            tabindex="0" id="button2" aria-labelledby="button2_label"
            role="button"
            class="dijitStretch dijitButtonNode dijitButtonContents"
            dojoattachpoint="focusNode,titleNode" type="button"
            wairole="button" waistate="labelledby-button2_label">
            <span class="dijitInline dijitEditorIcon dijitEditorIconCopy"
                dojoattachpoint="iconNode">
                <span class="dijitToggleButtonIconChar">› </span>
            </span>
            <span class="dijitButtonText dijitDisplayNone"
                id="button2_label" dojoattachpoint="containerNode">
                Click me #2!
            </span>
        </button>
    </div>
</div>
```

对于 Dijit 窗口部件而言，这是一个标准化的操作过程，将简单的标记声明转换成丰富的 DOM 结构以完成多种目标，其中包括帮助创建视觉主题。

在这种情况下，Firefox 中的另一项 Firebug 功能非常方便。如图 28-22 所示，我们可以检查这个按钮元素，不仅能够了解这个窗口部件在 DOM 方面完成的工作，而且可以获得一个有关 CSS 声明(Dojo 将这里的样式表应用于这个窗口部件在 DOM 中的不同部分)的报告。

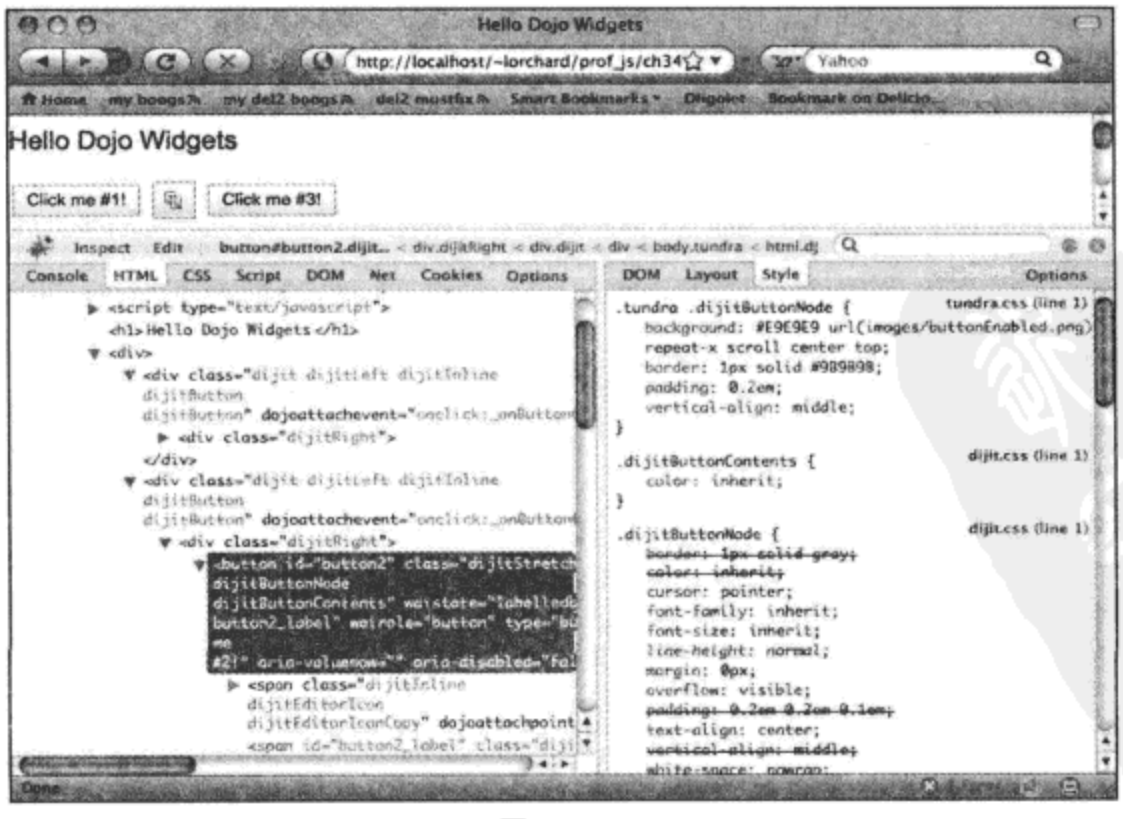


图　28-22

28.4.2 加载主题并将其应用于窗口部件

到目前为止，我们可以看到本书通篇都在使用“tundra”主题，但 Dojo 中的替代主题越来越多。下面的标记加载了两种不同的主题：

```
<html>
    <head>
        <style type="text/css">
            @import "../dojodev/dojo/resources/dojo.css";
            @import "../dojodev/dijit/themes/tundra/tundra.css";
            @import "../dojodev/dijit/themes/soria/soria.css";
        </style>
    </head>

    <body>
        <div class="tundra">
            <!-- tundra-themed widgets here -->
            <button id="button2" dojoType="dijit.form.Button"
                showLabel="false"
                iconClass="dijitEditorIcon dijitEditorIconCopy"
                onClick="console.log('Clicked '+this.id);">
                Click me #2!
            </button>
        </div>
        <div class="soria">
            <!-- soria-themed widgets here -->
            <button id="button3" dojoType="dijit.form.Button"
                showLabel="false"
                iconClass="dijitEditorIcon dijitEditorIconCopy"
                onClick="console.log('Clicked '+this.id);">
                Click me #3!
            </button>
        </div>
    </body>

</html>
```

如前所示，这是一个将图像图标应用于前一个示例中的 CSS 代码片段，可以在 Dojo 资源 dijit/themes/tundra/Editor.css 下找到它：

```
.tundra .dijitEditorIcon {
      background-image: url('images/editor.gif'); /* editor icons sprite image */
      background-repeat: no-repeat;
      width: 18px;
      height: 18px;
      text-align: center;
}
.tundra .dijitEditorIconCopy { background-position: -72px; }
```

请注意 CSS 中每个声明的.tundra 前缀。这个 CSS 类名应该出现在该按钮的父节点中的某个

地方(要么在<body>标记上，要么在沿着 DOM 树向下的某个地方，这样就可以将多个主题混合在一起，就像上面的示例一样)。

还有一个名为“soria”的主题，它在 dijit/themes/soria/Editor.css 文件中提供以下样式来替代前面示例中的标式：

```
.soria .dijitEditorIcon {
      background-image: url('images/editor.gif'); /* editor icons sprite image */
      background-repeat: no-repeat;
      width: 18px;
      height: 18px;
      text-align: center;
}
.soria .dijitEditorIconCopy { background-position: -72px; }
```

虽然这里给出的并不是与“tundra”主题有着本质不同的主题，但可以看到这里使用了.soria 类名选择器。加载这个主题，并将一个容器节点的 CSS 类切换到“soria”，这会让所有节点拥有微妙差别的外观。

28.4.3　定制并检查可用主题

关于如何从头开始创建一个全新的主题已经超出了本章的介绍范畴。与窗口部件匹配的 CSS 声明非常庞大，而且需要将现有示例与 Firebug 的 DOM 检查器结合起来以了解什么样式应用于哪个地方。

最好的做法是坚持使用自己的少数几个 CSS 声明来重写某个可接受主题的某些部分，其余部分则保留；否则，如果必须构建自己的全面外观，那么试着研究已安装的 Dojo 的 dijit/themes/目录，查看有哪些主题可以作为基础。不管采取哪种方式，窗口部件中的 CSS 挂钩数量都非常庞大，而且结构良好，创建或调整主题都只需要构建层叠样式表而已。

如果希望研究现有的主题，那么查看 Dojo 自带的演示页面 dijit/themes/themeTester.html。如图 28-23 所示，这个演示页面给出了绝大多数可用的窗口部件和可用主题的选择项，以便在它们之间进行切换。可以将这个工具与自己的页面组合起来试验各种主题，以便了解什么主题最适合自己。

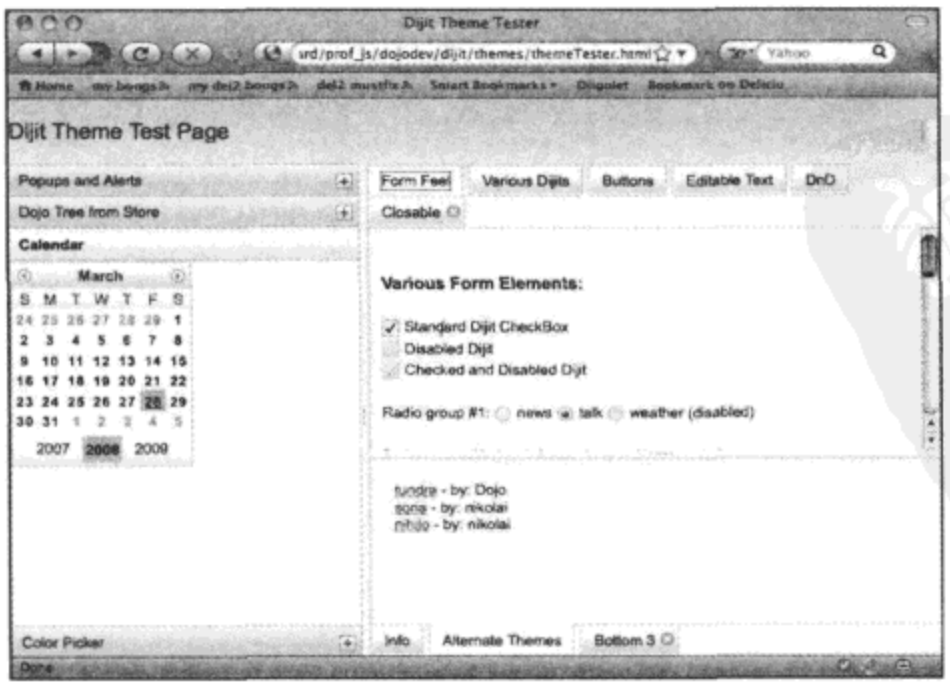


图　28-23

28.5　本章小结

在这一章中，我们研究了 Dojo 的 Dijit 子项目。Dijit 提供了范围很广的窗口部件，包括提供高级表单字段、布局管理器以及其他用户界面控件的窗口部件。每种窗口部件均可以以编程方式采用 JavaScript 代码来创建，或者本着 Dojo 的精神，采用由 dojo.parser 模块处理的声明式 HTML 标记来创建。一旦实例化，Dijit 系列窗口部件就提供了大量的选项来与 JavaScript 代码整合，并为扩展的 CSS 主题的应用提供了丰富的 CSS 挂钩。

在下一章中，我们将学习有关如何根据自己的项目来自定义 Dojo 框架内部版本，并了解 DojoX 子项目中潜在的高级功能和实验性功能。

第29章

构建和部署 Dojo

Dojo 工具集提供的自定义构建工具为开发人员提供了各种选项，使该工具集(以及您自己的打包代码)最小化并进行收缩，只保留部署的 Web 应用程序所需的部分。到目前为止我们已经看到，可以通过单个的脚本标记和多条 dojo.require()声明语句来加载所需的模块。

在底层，Dojo 管理 JavaScript 模块以及其他各种各样的素材(例如 HTML 窗口部件模板)的加载。但是，所有这些按需加载活动都会占用时间和带宽。为了缓解这种压力，Dojo 工具集的各种自定义构建试图将所有应用程序所需的最重要部分包含在一个精简的自包含包中。

本章内容简介：

- 创建自定义构建配置文件
- 生成自定义构建
- 检验并使用自定义构建

29.1 研究 Dojo 构建

Dojo 工具集有许多模块。在构建一个适当大小的 Web 应用程序时，非常有可能依赖数十个或更多单独的 Dojo 模块。

如果使用 Dojo 的“源代码”发行版本，或者从 Subversion 上检出最新最全的版本，那么模块的加载将转化为多次 Web 请求(通常每个模块要有多次请求，这是因为模块通常会进一步声明需求，甚至载入诸如 HTML 窗口部件模板之类的资源)。所有这些 HTTP 请求很快就会累积起来，从而增加应用程序加载延迟并占用更多带宽。

由于这些原因，Dojo 的自定义构建工具提供了一个更好的替代品：利用 Dojo 中的名称空间和模块结构以及依赖项声明，Dojo 中的模块非常容易把它们的源文件全部合并到一个大型的包含文件中。一旦合并在一起，JavaScript 代码就可以进一步进行优化和压缩，虽然不再适合于人类阅读，但其加载效率更高。

这个构建系统真正的优点在于，与 Dojo 提供的模块和类系统类似，它并不仅限于 Dojo 内部代码使用。我们可以将自己项目的模块和资源与用到的 Dojo 模块一起打包。此外，这个构建系统理解“层”的概念，可以选择性地将项目中用到的各个部分按照不同的排列进行打包。即使在创建高效加载的工具集编译文件时，也可以选择只加载在站点特定部分中用到的代码。

29.2 查找构建系统

为了了解构建系统，我们需要一个完整的 Dojo“源代码”发行版本，或者从 Subversion 储存库检出最新最全的版本。回顾第 23 章开头部分的内容，就会理解如何通过这两种方式之一获取完整的 Dojo 版本(如果没有使用其他方式的话)。

在 Dojo 发行版本的根目录下，我们可以找到类似下面的目录结构：

- dijit/ Dijit 子项目
- dojo/ Dojo 核心
- dojox/ DojoX 子项目
- util/ 构建、测试与文档编制工具
  - buildscripts/ 构建工具

在目录 util/buildscripts/下存放着所有的构建工具，具体来说，build.bat 脚本用于 Windows，而 build.sh 脚本用于类似 Linux 或 Mac OS X 的系统。

构建系统是一种命令行工具，因此这里并没有图形界面。但是，如果您已经习惯于使用终端窗口或外壳，那么这个过程不会太复杂。此外，构建系统要求安装 Java 1.4.2 或更新版本。除此之外，运行该工具所需的一切内容都应该已经包含在 Dojo 中。

29.3 创建自定义构建配置文件

Dojo 构建过程执行依赖性分析，检查在模块中找到的 dojo.require()语句以收集构造一个完备集合所需的所有模块的详细清单。但它需要知道从哪里开始分析，这就是构建配置文件的作用所在。构建配置文件制定模块名称空间和它们的位置，同时给出该过程产生的“层”的定义。

顺便提一下，层实际上并不比页面上通过<script>标记与 dojo.js 一起包含进来的额外.js 资源复杂。这些额外的包含文件将代码置于 dojo.js 的顶层，并预先加载这些模块和素材，否则当 dojo.require()语句出现时，它们将动态加载。

该构建过程允许产生多个层，每个层包含不同的依赖项集合，这样就可以加载站点具体部分将要使用的功能所在的个别层。这样有助于将每个页面上加载的代码降到最少。例如，如果知道站点的某个部分将会用到 dojox.dtl，而另一部分将会用到 dijit.form，那么在这种情况下可以创建两个不同的层，从而避免加载未用到的模块。

可以在 util/buildscripts/profiles 目录下找到一些样本配置文件，包括如下：

- layers.profile.js 演示该结构的工作原理的样本配置文件。
- base.profile.js 一个空白的配置文件，产生一个包含 Dojo 核心的层。
- offline.profile.js 包含使用 dojox.offline 模块所需的一切内容。
- standard.profile.js 这个配置文件包含了 Dijit 以及其他一些较为常用的模块。

仔细查看这些配置文件，以便对能够包含什么内容有一些最新的认识。考虑下面这个示例配置文件：

```
dependencies ={
    prefixes: [
        [ "dijit", "../dijit" ],
```

```
        [ "dojox", "../dojox" ],
        [ "decafbad", "../../ex-dojo-expand-deploy/decafbad" ]
    ],
    layers:  [
        {
            name: "decafbad-expand.js",
            dependencies: [
                "decafbad.expand.encodings",
                "decafbad.expand.templates",
                "decafbad.expand.jsonpath"
            ]
        },
        {
            name: "decafbad-io-common.js",
            dependencies: [
                "dojo.io.script",
                "dojo.io.iframe"
            ]
        },
        {
            name: "decafbad-fx-common.js",
            dependencies: [
                "dojo.fx",
                "dojox.fx.easing",
                "dojo.NodeList-fx",
                "dojo.behavior"
            ]
        },
        {
            name: "decafbad-form-common.js",
            dependencies: [
                "dijit.form.Form",
                "dijit.form.ValidationTextBox",
                "dijit.form.NumberTextBox",
                "dijit.form.NumberSpinner"
            ]
        }
    ]
};
```

按照约定，我们将配置文件定义为 JavaScript 对象。这个依赖项对象的第一个属性是 prefixes，它列出了在这个配置文件中除了 Dojo 核心自身之外用到的所有模块名称空间。这个列表中的每一项都提供了一个模块名称空间的名称以及一个相对于 Dojo 核心的目录路径。可以看到，Dijit 和 DojoX 子项目均包含在内，还有一个位于 Dojo 目录树之外的模块，它里面包含了本章的样本代码。

一旦描述了模块名称空间，下面就要定义层，通过一个名为 layers 的列表来完成该操作。这个列表的每一项都是一个结构，它指定了该层的名称并提供了进一步的列表，其中详细列举了该层中包含的基础模块。记住，我们不需要将所有内容都包含在这里：列出的模块应该只包括必需的部分。在构建层的过程中，Dojo 构建过程将对这些模块进行依赖性分析，不仅包括列出的模块，而且包括模块内部的 dojo.require()调用。

因此，decafbad.expand.encodings 的代码包含针对以下额外模块的 dojo.require()调用：

- dojo.io.script
- dojox.encoding.base64
- dojox.encoding.crypto.Blowfish
- dojox.encoding.digests.MD5

因此，只列出 decafbad.expand.encodings 作为层的一部分，这也会导致将这 4 个额外的模块(连同它们列出的进一步的依赖项)放入该层中。

29.4　生成自定义构建

构建过程是通过命令行脚本启动的。这个脚本引发一个基于 Java 的 Rhino JavaScript 解释器(www.mozilla.org/rhino/)，由它来执行待构建的 JavaScript 代码。如果感兴趣的话，那么可以查看该解释器的具体实现，它位于与 build.sh 相同的目录下的 build.js 文件中。

下面是一个构建脚本的调用示例，为了清晰起见，这里添加了换行符：

```
./build.sh \
   profileFile=../../../ex-dojo-expand-deploy/decafbad.profile.js \
   action=clean,release \
   releaseDir=../../../ex-dojo-expand-deploy/build/ \
   releaseName=prof_js
```

可以看到，这个脚本接受几个参数来控制它的行为以及输出。这个调用并没有包括所有可用选项，但下面是一些较为有用的选项：

- profile　这是 util/buildscripts/profiles 目录中配置文件的可选名称(删除.profile.js 后缀)。因此，profile=base 将从 base.profile.js 中读取。
- profileFile　这个选项替换配置文件，允许我们指定构建配置文件的完整路径和文件名，该路径可能位于 Dojo 目录树之外。
- action　这个选项用来确定构建过程是否将现有的文件删除并替换(clean, replace)、简单地将现有文件删除(clean)或者简单地覆盖文件(replace)。
- releaseDir　可以选择指定一条路径，构建过程将在这里创建发布目录。默认值为 Dojo 目录树根目录下名为 release 的目录，或者是相对于 buildscripts/目录的“../../release/”。
- releaseName　可以选择指定发布目录的名称。默认值为“dojo”。
- optimize　这个选项接受如下选项之一来进行代码的简化和压缩：
  - comments　将注释从代码中删除。
  - packer　使用 Dean Edwards 的 Packer(http://dean.edwards.name/packer/)压缩代码。这个系统对原始的 JS 源代码应用一些简单的技术来尽可能减小代码规模，但代价是代码的可读性极差。
  - shrinksafe(默认值)　使用 Dojo ShrinkSafe 系统(http://dojotoolkit.org/docs/shrinksafe)压缩代码。ShrinkSafe 系统利用 Rhino JS 解释器将 JS 代码分析成字节码，然后试图以字节码的形式进行精简处理。接下来，将该字节码重新还原成 JS 源代码，这样的代码应该规模更小、加载效率更高。

解释了这些选项之后，我们回过头来查看前面给出的调用示例，此时应该会理解它使用本章样本代码目录中的 decafbad.profile.js 配置文件生成了一个新的内部版本，并在目录../../../ex-dojo-expand-deploy/build/prof_js 下生成了一个新的发行版本。

在运行这条命令时，应该看到类似下面的示例输出结果：

```
$ ./build.sh profileFile=../../../ex-dojo-expand-deploy/decafbad.profile.js \
    action=clean,release releaseName=prof_js \
    releaseDir=../../../ex-dojo-expand-deploy/build/

clean:  Deleting: ../../../ex-dojo-expand-deploy/build/prof_js
release:  Using profile: ../../../ex-dojo-expand-deploy/decafbad.profile.js
release:  Using version number: 0.0.0.dev for the release.
release:  Deleting: ../../../ex-dojo-expand-deploy/build/prof_js
release:  Copying: ../../dojo/../dijit to:
    ../../../ex-dojo-expand-deploy/build/prof_js/dijit
release:  Copying: ../../dojo/../dojox to:
    ../../../ex-dojo-expand-deploy/build/prof_js/dojox
release:  Copying: ../../dojo/../../ex-dojo-expand-deploy/decafbad to:
     ../../../ch35/build/prof_js/decafbad
release:  Copying: ../../dojo to: ../../../ex-dojo-expand-deploy/build/prof_js/dojo
release:  Building dojo.js and layer files
...
release:  Files baked into this build:
dojo.js:
...

decafbad-expand.js:
...

decafbad-io-common.js:
...

decafbad-fx-common.js:
...

decafbad-form-common.js:
...
release:  Build is in directory: ../../../ex-dojo-expand-deploy/build/prof_js
Build time: 67.238 seconds
```

这段输出结果将为您提供有关构建期间所有已完成工作的长时间运行说明。

29.5　检验并使用自定义构建

现在，我们已经了解如何为自定义构建指定一个配置文件，以及如何根据配置文件来生成构建过程。但是，在这个过程结束时我们可以得到什么呢？

首先要注意的是，在构建过程的文本消息中，有几条消息宣告该过程正在“复制”各种文件

集合。这说明一个自定义构建包含了所有的单个模块(Dojo 通常包含的模块)，这意味着仍然可以调用 dojo.require()来加载并不属于任何层的模块，而且仍然可以动态加载。

但接下来需要注意的是，新构建的“层”文件写入到构建过程的根目录下的 dojo/子目录中。在前面给出的文本消息中，这些包括配置文件中定义的各个层对应的如下文件：

- dojo/dojo.js
- dojo/decafbad-expand.js
- dojo/decafbad-io-common.js
- dojo/decafbad-fx-common.js
- dojo/decafbad-form-common.js

上面这些文件都已经经过压缩，但是添加后缀.uncompressed.js 应该还可以找到对应的文件。这些文件是未经压缩的 JS 文件，它们对于可视的调试来说比较有用，因为它们仍然具有一定的可读性，而且仍然包含完全压缩版本中可用的全部代码。

要进一步了解在构建期间发生的事情，请查看 dojo/dojo.js 文件。您应该会注意到，这个文件已经从其源代码形式的 5KB 启动代码扩展到大约 76KB 的聚集压缩代码。

如果在.uncompressed.js 版本中搜索 dojo.provide()调用，那么可以找到 20 多个实例，而在原始的启动源文件中找不到这些实例。这些模块的代码现在全部属于 dojo/dojo.js 启动文件的一部分。当在页面的头部中使用<script>标记加载时，用来加载这些模块的 dojo.require()调用将不会导致产生进一步的动态获取请求。

因此，为了使用构建过程生成的所有这些新层，只需要在页面的<head>部分中添加类似下面的内容：

```
<script type="text/javascript"
   src="build/prof_js/dojo/dojo.js"></script>
<script type="text/javascript"
   src="build/prof_js/dojo/decafbad-expand.js"></script>
<script type="text/javascript"
   src="build/prof_js/dojo/decafbad-io-common.js"></script>
<script type="text/javascript"
   src="build/prof_js/dojo/decafbad-fx-common.js"></script>
<script type="text/javascript"
   src="build/prof_js/dojo/decafbad-form-common.js"></script>
```

这些包含语句将载入所有经过构建过程处理的模块，而且一旦浏览器加载它们，就不再需要额外的动态获取请求来满足 dojo.require()声明。另一方面，注意到这种设计的优点在于，如果将这些包含语句中的任何一条去除(只要保留 dojo/dojo.js)，那么模块需求系统仍将能够找到并获取任何想要的模块。

因此，构建系统是一种优化加载时间和效率的完全事后的方式，除了使用 dojo.require()和 dojo.provide()调用之外，不需要在开发期间进行任何预先的计划。但是，如果已经认可 Dojo 的工作方式，那么您可能已经在开发应用程序时通过使用模块系统获益，因此自然而然地为采用构建过程做好了准备。

要获取有关构建系统的更深入的信息，请参阅位于如下网址中的官方文档：

```
http://dojotoolkit.org/book/dojo-book-0-9/part-4-meta-dojo/package-system-and-custo
m-builds
```

29.6　本章小结

到这里，本书中专门讲解 Dojo 工具集的部分即将结束。我们快速浏览了 DojoX 子项目以及许多可用的不同实验性代码程序包。然后，我们看到了 Dojo 自定义构建过程如何帮助在部署的项目中更加高效地加载 Dojo 代码以及自己的代码。

第30章

扩展 Dojo

DojoX 子项目包含了 Dojo 不断扩展的前沿技术，包括新模块和实验性代码。虽然这个项目规模太大，而且变化速度太快，仅使用一章的篇幅很难做到面面俱到，但在这里我们计划浏览一些比较有趣的模块，并了解 Dojo 工具集的整体发展方向。DojoX 子项目中的可选代码尚未成为 Dojo 核心的一部分，不使用它当然也能够创建引人注目的 Web 应用程序。但是，这里介绍的模块肯定能够改进我们的开发工作，而且在某些情况下还能够开发出全新类型的应用程序。

本章内容简介：

- DojoX 子项目
- 试验高级窗口部件
- 利用模板生成内容

30.1 研究 DojoX 子项目

Dojo 工具集的核心几乎提供了开发响应灵敏的现代 Web 应用程序所需的所有关键组件。将 Dijit 加入这个组合为我们提供了非常丰富的用户界面和布局管理扩展组件。除了这个牢固的基础之外，DojoX 子项目引入了各种有趣的、创新的模块，扩展了浏览器和应用程序设计的领域。这些模块实现的功能包括如下：

- 高级用户界面和布局窗口部件
- 模板处理与呈现
- 矢量绘图和数据驱动图表
- AJAX 的替代技术
- 扩展的表单数据验证辅助函数
- 本地浏览器内存储器和脱机应用程序支持
- 编码和散列函数
- 高级数据结构

这个列表并不完整，而且我们不会在这里讲解这个列表，但是您或许可以在 DojoX 子项目中找到某项功能，它能够满足自己应用程序中某个不常见的或者实现起来比较困难的需求。

30.2 尝试高级窗口部件

在 DojoX 中，我们可以找到好几个尚未成为 Dijit 一部分的实验性 UI 窗口部件。它们处于不同的开发和文档编制阶段，但其中有几个值得我们了解其是否适合自己的项目。

与 DojoX 子项目中提供的大多数组件一样，有必要深入 Dojo 发行版本本身来了解有什么可用的窗口部件(特别是如果正在使用从 Subversion 储存库中检出的发行版本)。DojoX 窗口部件位于路径 dojox/widget/下，我们特别有必要浏览 dojox/widget/tests/目录，在这个目录中可以找到大部分(甚至全部)可用的 DojoX 窗口部件的演示。

30.2.1 构建鱼眼菜单

开始了解这些窗口部件，首先查看下面这段标记：

```
<div dojoType="dojox.widget.FisheyeList"
    itemWidth="50" itemHeight="50"
    itemMaxWidth="150" itemMaxHeight="150"
    orientation="horizontal"
    effectUnits="2"
    itemPadding="10"
    attachEdge="top"
    labelEdge="top"
    id="fisheye1">
    <div id="item1" dojoType="dojox.widget.FisheyeListItem"
        onclick="console.log('clicked '+this.id)"
        label="Item 1"
        iconSrc="../dojodev/dojox/widget/tests/images/icon_browser.png">
    </div>

    <div id="item2" dojoType="dojox.widget.FisheyeListItem"
        label="Item 2"
        onclick="console.log('clicked '+this.id)"
        iconSrc="../dojodev/dojox/widget/tests/images/icon_calendar.png">
    </div>

    <div id="item4" dojoType="dojox.widget.FisheyeListItem"
        label="Item 3"
        onclick="console.log('clicked '+this.id)"
        iconSrc="../dojodev/dojox/widget/tests/images/icon_email.png">
    </div>

</div>
```

上面的标记声明了一个 dojox.widget.FisheyeList 实例，它里面含有多个 dojox.widget.FisheyeListItem 实例。这个标记创建了一组沿着水平方向或垂直方向排列的图标(类似于 Mac OS X Dock)：当光标靠近这个列表中的某个点时，它就会变大并凸起。最接近光标的图标会变得更大，而它两侧的

图标变大的程度次之，因此创建了“鱼眼”效果。这些图标是可单击的，而且可以带有文本标签。

与前一个示例相比，下面这个窗口部件属于一种轻量级的实现：

```
<ul id="fisheye_list">
    <li>Foo bar</li>
    <li>Baz foo</li>
    <li>Quux xyzzy</li>
    <li>Plugh thud</li>
    <li>Hello world</li>
</ul>

<script type="text/javascript">
    dojo.addOnLoad(function() {
        dojo.query('li', 'fisheye_list').forEach(function(el) {
            new dojox.widget.FisheyeLite({
                durationIn:  350,
                easeIn:      dojox.fx.easing.bounceOut,
                durationOut: 600,
                easeOut:     dojox.fx.easing.elasticOut
            }, el);
        });
    });
</script>
```

dojox.widget.FisheyeList 窗口部件并没有声明一个带有子窗口部件的父列表，而是声明了多个独立的元素，每个元素在 mouseover 事件引发时都会变大。在上面的代码中，我们使用 dojo.query() 通过编程方式为 HTML 列表中的每个列表项创建一个该窗口部件的实例。

30.2.2　利用 Toaster 窗口部件建立动画通知

下面也是一个值得考虑的窗口部件：

```
<div id="toast1" dojoType="dojox.widget.Toaster"
    positionDirection="br-up"
    messageTopic="decafbad/toasterMessage"></div>

<div id="toast2" dojoType="dojox.widget.Toaster"
    positionDirection="tl-right"
    messageTopic="decafbad/toasterMessage"></div>

<button dojoType="dijit.form.Button">
    <span>Message</span>
    <script type="dojo/method" event="onClick">
        dojo.publish("decafbad/toasterMessage", ["Hello toaster!"]);
    </script>
</button>

<button dojoType="dijit.form.Button">
    <span>Warning</span>
    <script type="dojo/method" event="onClick">
```

```
        dojo.publish("decafbad/toasterMessage", [{
            message: "I'm warning you toaster!",
            type: "warning",
            duration: 1500
        }]);
        </script>
    </button>
```

这段标记声明了一对 dojox.widget.Toaster 窗口部件和配套的 dijit.form.Button 窗口部件。Toaster 窗口部件的工作方式与许多未读电子邮件通知系统类似，它会从窗口的某个角落滑入一个元素。可以配置前面示例中的窗口部件，让其监听全局事件主题并为各种类型的消息(包括提供信息的消息、警告和错误)提供一个动画通知功能。

图 30-1 描绘了本章中目前已经演示过的窗口部件。此外要记住，这些远远没有包括 DojoX 中的所有可用窗口部件，因此一定要在自己的 Dojo 目录中四处浏览，以便找到更多的模块和演示。

图 30-1

30.3 采用高级表单验证辅助函数

在前一章中我们看到，Dijit 子项目提供了众多带有内置表单验证选项的表单窗口部件。可以指定几种类型的数据，还可以基于正则表达式和值范围来定义约束。

DojoX 子项目在这些功能的基础上进行扩展，提供了一组附加的验证测试，还提供了基于标记的验证规则的替代技术。为了进行演示，应该将下面的模块作为需求模块进行声明。

```
<script type="text/javascript">
    dojo.require("dojo.parser");
```

```
    dojo.require("dijit.form.Form");
    dojo.require("dijit.form.ValidationTextBox");

    dojo.require('dojox.validate');
    dojo.require('dojox.validate.web');
    dojo.require('dojox.validate.us');
    dojo.require('dojox.validate.check');
</script>
```

接下来，我们部分地回顾第 28 章中构建的表单类型，查看下面这段标记：

```
<form id="form1" method="post" action="echo.php?type=text"
    dojoType="dijit.form.Form">

    <fieldset class="main">
        <legend>Registration form</legend>
        <ul>

            <li>
                <label for="homepage">Homepage</label>
                <input type="text" name="homepage"
                    dojoType="dijit.form.ValidationTextBox"
                    required="true"
                    validator="dojox.validate.isUrl"
                    promptMessage="Enter a URL" />
            </li>

            <li>
                <label for="email">Email</label>
                <input type="text" name="email"
                    dojoType="dijit.form.ValidationTextBox"
                    required="true"
                    validator="dojox.validate.isEmailAddress"
                    constraints="{ allowCC: false }"
                    promptMessage="Enter an email address" />
            </li>
```

这段标记声明了一个窗口部件类型为 dijit.form.Form 的表单。在这个表单中有两个类型为 dijit.form.ValidationTextBox 的表单字段窗口部件，每个字段都有一个新的属性 validator。

这个 validator 属性接受一个函数名，该函数将负责检验这个字段的有效性。按照这种方式使用的函数应该接受两个参数：该字段的值及其 constraints 属性的值。

因此，第一个表单字段要求它的值是一个有效的 URL。它并没有定义约束，因此任何有效的 URL 都可接受。但是，第二个表单字段的约束属性值为“{ allowCC: false }”，它将拒绝任何域名以国家代码结尾的电子邮件地址。通过 dojox.validate.isEmailAddress 函数背后的代码来实现该功能。这项功能的文档可能有些混乱，因此最好直接查阅 Dojo 发行版本，查看有哪些可用内容。

下面这段标记彻底避免使用窗口部件，退回到使用普通的输入字段。

```
            <li>
                <label for="homepage2">Homepage2</label>
                <input type="text" name="homepage2" />
            </li>
            <li>
                <label for="email2">Email2</label>
                <input type="text" name="email2" />
            </li>
            <li>
                <label for="password">Password</label>
                <input type="password" name="password" />
            </li>
            <li>
                <label for="passsword_check">Password (again)</label>
                <input type="password" name="password_check" />
            </li>
```

上面示例中的表单字段现在不再那么让人激动，它们不再是窗口部件，但表单验证涉及的事项已经完全移入表单提交处理程序中：

```
            <li class="buttons">
                <input type="submit" value="submit record" />
            </li>

        </ul>
    </fieldset>
```

```
    <script type="dojo/method" event="onSubmit" args="ev">
        var profile = {
            required: [
                'homepage2',
                'email2'
            ],
            constraints: {
                homepage2: dojox.validate.isUrl,
                email2:    dojox.validate.isEmailAddress
            },
            confirm: {
                password_check: 'password'
            }
        };

        var validate_result = window.foo =
            dojox.validate.check(dojo.byId('form1'), profile);
        if (!this.validate() || !validate_result.isSuccessful()) {
            dojo.stopEvent(ev);
            alert("Form not yet valid! (connected in markup)");

            console.log(validate_result.getMissing());
            console.log(validate_result.isMissing('email2'));
```

```
            console.log(validate_result.getInvalid());
            console.log(validate_result.isInvalid('password_check'));
        }

    </script>

</form>
```

与第 28 章中构建的表单类似，这个表单在其 submit 事件的处理程序中实现了表单验证检查。而且，与前面的表单类似，它要求这个页面上的所有表单字段窗口部件都报告有效性。但是，这里的新事物是验证配置文件以及使用它的辅助函数 dojox.validate.check()调用。

这个验证配置文件取代了大部分(如果不是全部的话)与表单字段值后期处理以及验证有关的表单窗口部件属性。这里给出的配置文件并没有充分运用这项功能，但可以看到它定义了一些必填的表单字段、验证处理程序函数以及哪些表单字段应该具有匹配的值(例如 password 和 password_check)。

dojox.validate.check()调用返回一个对象，该对象中有几个方法在确定表单的验证状态时非常有用。第一个调用的方法是.isSuccessful()，它依据通过配置文件验证的情况来报告表单的整体有效性。之后，有几个方法用来确定哪些表单字段未填写或包含无效数据。

与基于窗口部件的表单不同，这种验证方法没有指示或高亮显示无效的或缺少必填数据的表单字段。不管怎样，这种方法将所有这些工作交给开发人员完成，让开发人员按照自己满意的方式进行处理。因此，如果您不喜欢窗口部件，也不喜欢在自定义标记中声明表单需求，那么使用 dojox.validate.check()可以完全直接控制这些方面。

30.4　从模板生成内容

大多数服务器端的现代 Web 应用程序框架都有 Model-View-Controller(模型-视图-控制器)架构的概念。其基本思想是，在设计时应该使涉及的数据、业务逻辑和显示呈现保持独立，这样就可以灵活地修改或替换任何一个领域中的组件。

Django(采用 Python 编写)就是这样一个在服务器端实现这种模型的 Web 应用程序框架。而且事实证明，在 DojoX 子项目中有 Django Template Language 的克隆实现。从下面给出的网址中可以了解有关该原始系统的更多信息：

```
www.djangoproject.com/documentation/templates/
```

dojox.dtl 模块提供了 Django 的内置模板系统的完整克隆实现，此外还增加了一些功能，以便它能够更好地在浏览器中运行。为了进行演示，请考虑下面这个 Django 模板示例：

```
<li>
    Total results:
    <i>{{ ResultSet.totalResultsAvailable }}</i>
</li>
{% for result in ResultSet.Result %}
    <li>
        <h4><a href="{{ result.Url }}">{{ result.Title }}</a></h4>
```

```
            <div>
                <p>{{ result.Summary }}</p>
                <div class="meta">

                    <b>{{ result.DisplayUrl }}</b> -

                    {% ifnotequal result.MimeType 'text/html' %}
                        <i>{{ result.MimeType }}</i> -
                    {% endifnotequal %}

                    {{ result.ModificationDate|timesince }} -

                    {% if result.Cache.Size %}
                        {{ result.Cache.Size|filesizeformat }} -
                        <a href="{{ result.Cache.Url }}">Cached</a>
                    {% endif %}
                </div>
            </div>
        </li>
    {% endfor %}
```

要想完全理解这个示例，查阅 Django 模板语言文档会有所帮助，但是这个模板生成的标记看起来类似于 Web 搜索的结果。它接受一个名为 ResultSet 的数据结构，该数据结构包含了有关搜索结果的各种元数据以及搜索结果列表。

这个模板接受搜索结果数据并将它们呈现为 HTML 列表，包括每条结果的各种详细信息(例如作为页面链接的标题、页面的内容类型、页面上一次索引的时间以及一个指向搜索引擎缓存并提示该页面在缓存中的大小的链接)。该模板语言还提供了输出过滤器，用于把时间和文件大小转换成更加具有可读性的表示形式，而且各种数据都经过 HTML 转义以帮助堵塞 XSS 安全漏洞。

最后，这个模板生成了一些看起来类似于 Yahoo!搜索引擎搜索结果页面的内容。相应地，这个演示使用了 Yahoo! Web Search API 的 JSONP 版本来填充该模板。

现在，考虑下面的标记，它用来构建一个简单的执行 Web 搜索的界面：

```
<div id="ex_search">

    <form class="search_form">
        Search:
        <input type="text" size="30" class="search" />
        <input type="submit" value="search" />
    </form>

    <ul class="results">
    </ul>

</div>
```

接下来，下面的 JavaScript 代码为前面示例中的搜索表单提供功能：

```
dojo.require('dojo.io.script');
dojo.require('dojox.dtl');
```

```
dojo.require('dojox.dtl.Template');
dojo.require('dojox.dtl.Context');

// Fetch the external template source and instantiate a new DTL object.
var tmpl = null;
dojo.xhrGet({
    url: 'templates.dtl',
    handleAs: 'text',
    load: function(resp, io_args) {
        tmpl = new dojox.dtl.Template(resp);
    }
});
```

这里要做的第一件事情是声明一些模块需求。由于这段代码最终需要使用 Yahoo! Web Search API 的 JSONP 版本，因此需要载入 dojo.io.script。接下来是用来加载 dojox.dtl 名称空间各个部分的一些声明。

在编写需求声明之后，调用 dojo.xhrGet()来加载前面演示的模板。假设该模板位于 URL templates.dtl 处，应该将其保存到此处，或者应该修改该代码以指向具体的存放位置。一旦模板加载完毕，就使用它的文本来实例化 dojox.dtl.Template 对象。一旦数据可用，就使用这个对象来呈现模板。

现在连接搜索表单：

```
dojo.query('.search_form', 'ex_search')
    .connect('onsubmit', function(ev) {
        dojo.stopEvent(ev);

        // Don't work until the template has arrived.
        if (!tmpl) return;

        // Build up the JSON URL for search results.
        var search_url = 'http://search.yahooapis.com/'+
            'WebSearchService/V1/webSearch?'+
            dojo.objectToQuery({
                appid: 'appid_goes_here',
                output: 'json',
                results: 5,
                query:   dojo.query('.search', 'ex_search')[0].value
            });
```

上面的代码查找页面中的搜索表单并为其提供一个处理程序函数(用于引发表单提交)。在进行普通的处理之前，这段代码获取表单字段 search 的值，并准备好一个 URL 以供后面的 Yahoo! Web Search API 调用使用。

有关这个 API 工作原理的信息以及它接受哪些参数，请查阅下面的文档：

```
http://developer.yahoo.com/search/web/V1/webSearch.html
```

最后，获取数据并让模板运行：

```
        // Insert a loading message.
        dojo.query('.results', 'ex_search')
```

```
            .empty().addContent('<li>Loading...</li>');

        // Fetch the JSON feed for search results...
        dojo.io.script.get({
            url: search_url,
            callbackParamName: 'callback',
            handle: function(resp, io_args) {
                // HACK: Search result times are in seconds, but JS
                // times are in milliseconds.  Patch the data.
                dojo.forEach(resp.ResultSet.Result, function(r) {
                    r.ModificationDate *= 1000;
                });
                console.log(resp);
                var ctx = new dojox.dtl.Context(resp);
                var out = tmpl.render(ctx);
                dojo.query('.results', 'ex_search')
                    .empty().addContent(out);
            }
        });

    });
```

在最后这段代码中，我们将搜索结果的内容取出来并使用它替换正在加载的消息。然后，使用前面的代码清单中组成的 Yahoo! Web Search API URL 调用 dojo.io.script.get()。一旦数据到达，就调用 handle 参数中定义的回调函数。

最后一部分就是模板发挥作用的地方：首先，使用搜索 API 返回的数据结构创建一个 dojox.dtl.Context 对象。这个对象存放要传入模板进行呈现的数据。然后，调用之前实例化的模板对象的.render()方法，该方法返回最终要呈现的内容。然后将这段内容插入到页面中以替换正在加载的消息。

如图 30-2 所示，该代码的最终结果是动态呈现搜索结果，所有操作都是在客户端中完成的。

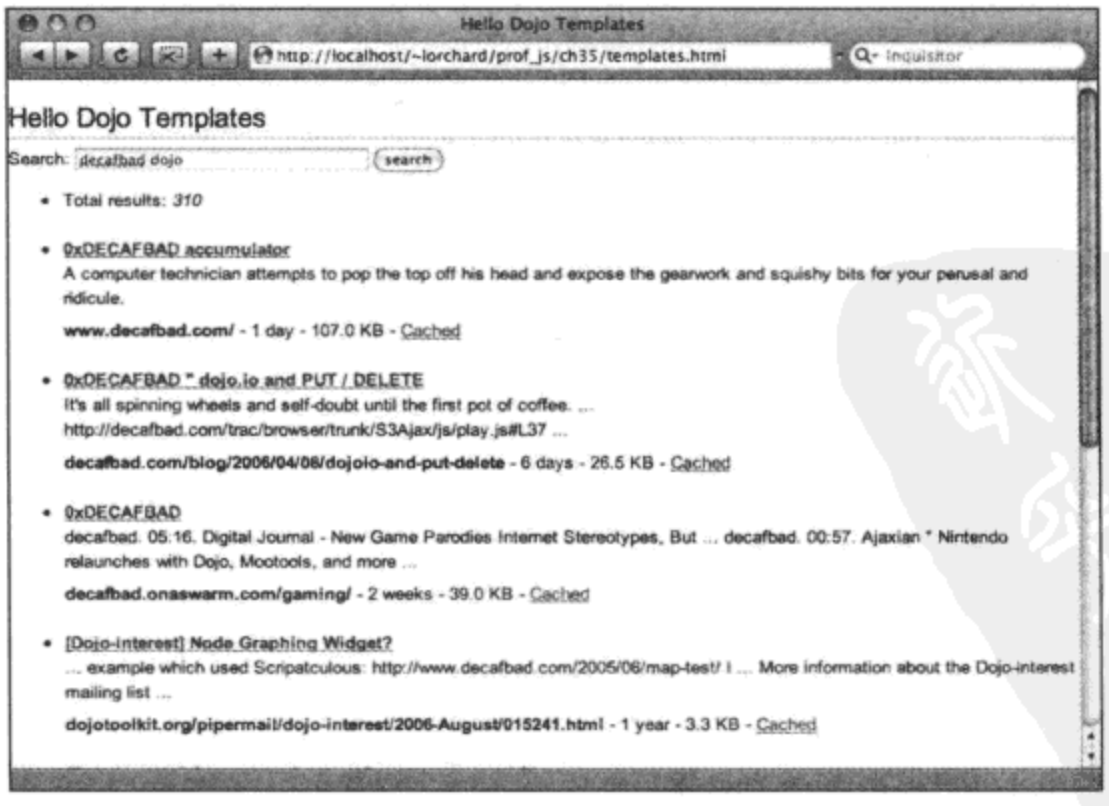


图 30-2

尽管大多数时候这类处理都会在服务器上执行，但是这个模板系统有很大的灵活性，可以让我们能够在客户端中生成或更新内容。如果开始创建自己的 Dijit 窗口部件，那么这个系统用起来也非常方便。

30.5　绘制形状以及呈现图表

使用普通 HTML、CSS 和 JavaScript 绘制任意形状和线是非常困难的事情，这通常会迫使人们把目光投向诸如 Java 小应用程序和 Flash 电影之类的基于插件的技术。但是，现代浏览器在这个领域中提供的功能越来越强大。这些功能尚未在所有浏览器中取得一致，但是它们非常相似，以至于我们能够将它们之间的差异进行抽象并放到一个实用的公共接口中。

基于这一点，dojox.gfx 程序包将可跨浏览器的绘图工具封装成一个可交换的呈现引擎(适当地使用 SVG、VML、Silverlight 或<canvas>标记)。在这些引擎的基础上，dojox.gfx 模块提供了各种声明式绘图基本要素来描述线、曲线、闭合的和填充的形状、图像以及文本。还支持各种变换和旋转以及用于提供交互性的事件处理程序连接。

此外，利用 dojox.gfx 工具以及几个其他的模块来构建 dojox.charting。这个程序包能够用来将来自各种源的数据绘制和展现为线形图、柱状图以及饼图。它包含了多个选项，可用来调整数据和标签轴呈现的外观，以及动态更新显示。此外，借助于 Dojo 中的其他可用技术，可以按照需要从服务器端来源那里获取数据，或者将数据作为声明式标记嵌入到页面内部。

30.5.1　绘制形状和线

为了了解 dojox.gfx 的功能，下面查看一个小示例，从下面的标记开始：

```
<div id="mycanvas" style="width: 400px; height: 400px"></div>
```

这段标记为 dojox.gfx 提供了一块空白的画布，通过如下的代码管理该画布：

```
dojo.require("dojox.gfx");

var g = dojox.gfx.createSurface(dojo.byId("mycanvas"), 400, 400);

g.createCircle({ cx: 200, cy: 200, r: 195 })
    .setFill([255, 255, 128, 1])
    .setStroke({ color: [0, 0, 0, 1], width: 2 });

g.createCircle({ cx: 125, cy: 120, r: 25 })
    .setFill('white')
    .setStroke({ color: 'black', width: 2 });

g.createCircle({ cx: 275, cy: 120, r: 25 })
    .setFill('#fff')
    .setStroke({ color: "#000", width: 2 });

g.createPath()
    .setStroke({color: "#000", width: 2})
```

```
.setFill('white')
.moveTo(100, 225)
.curveTo(100, 225, 200, 400, 300, 225)
.curveTo(300, 225, 205, 500, 100, 225);
```

这段代码只是绘制了一张笑脸，如图 30-3 所示。但是，它可以实现的功能要多于使用普通的 HTML、CSS 和 JavaScript。

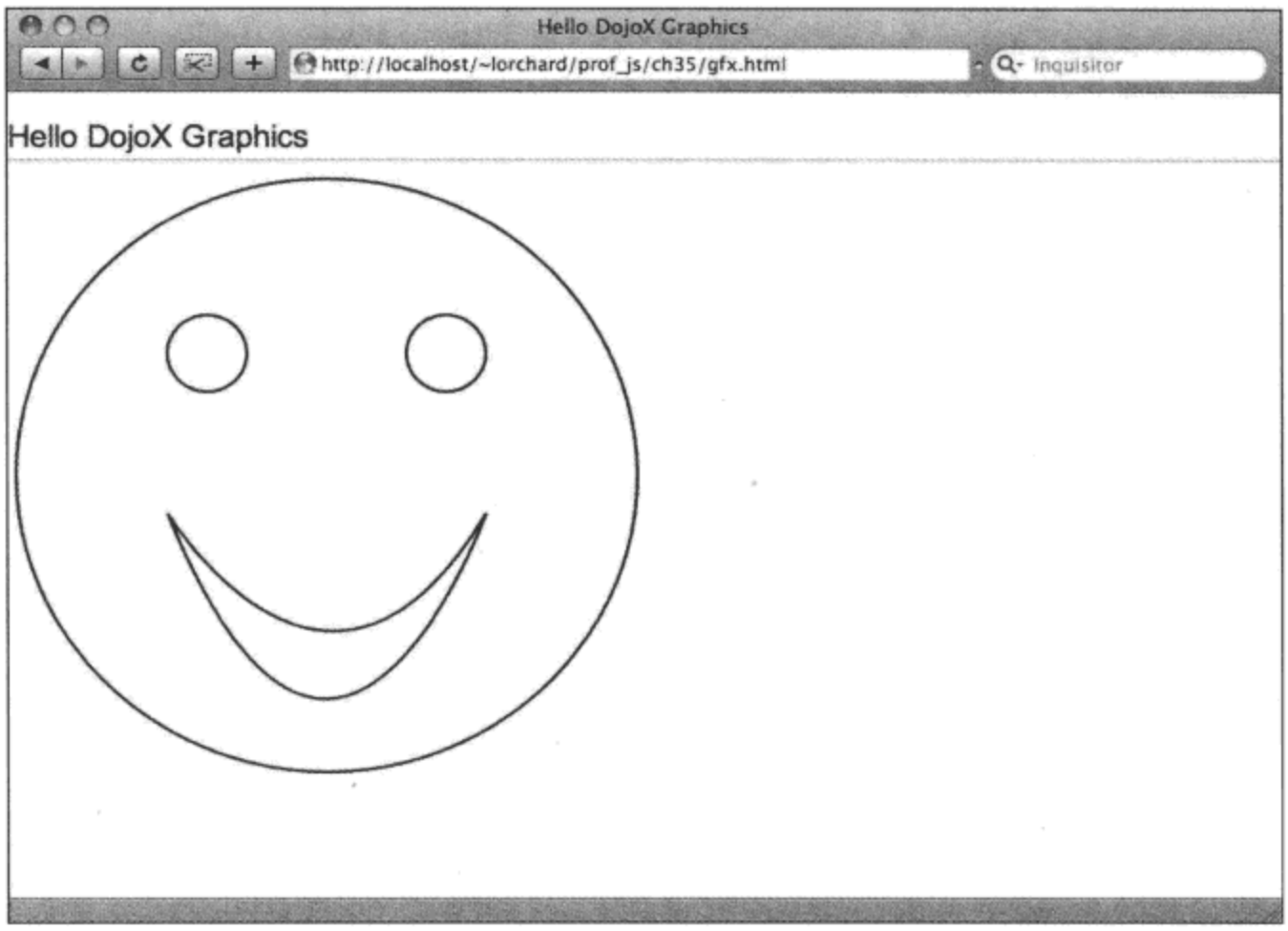


图　30-3

这里提供的示例只涉及基础，仅仅讲解 dojo.gfx 就需要专门的几章篇幅。为了真正感受 dojox.gfx 提供的功能，我们应该研究 Dojo 发行版本中的这个程序包。具体来说，在下面的路径中有一些比较有趣的演示和详细的测试可供仔细研读：

- dojox/gfx/demos/
- dojox/gfx/tests/

demos/目录下方有一些实用的交互性示例，它们由 dojox.gfx 的多个部分组合而成。此外，在 tests/目录下可以找到许多详尽的、专注的测试来检验该程序包的方方面面，对于查阅源代码和了解有什么组件可用以及如何使用这些组件而言，这些测试非常有用。

与 DojoX 中大多数可用的组件一样，该程序包的文档也仍然处于不断改进之中，因此深入研究这些演示和测试实际上是了解这些实验性的、非常有用的代码的最佳方式。

30.5.2　呈现图表和曲线图

从原始图形转移到结构化图表，考虑下面的标记：

```
<div dojoType="dojox.data.HtmlStore"
```

```
    dataId="tableExample" jsId="tableStore"></div>

<table id="tableExample" style="display: none;">
    <thead>
        <tr><th>value</th></tr>
    </thead>
    <tbody>
        <tr><td>6.3</td></tr>
        <tr><td>1.8</td></tr>
        <tr><td>3.0</td></tr>
        <tr><td>0.5</td></tr>
        <tr><td>4.4</td></tr>
        <tr><td>2.7</td></tr>
        <tr><td>2.0</td></tr>
    </tbody>
</table>
```

在上面这段标记中，我们再次看到了 Dojo 和 DojoX 中的数据功能。我们在第 28 章中曾经看到过，可以使用标记或通过编程方式来创建数据源，以及从外部 JSON 数据源获取数据。此处在标记中包含了所有相关数据。我们为数据窗口部件 dojox.data.HtmlStore 指定了一个隐藏表的 ID，并利用 DOM 中的表结构的行和列来填充数据源。

但是，这里并没有填充一个下拉式选择框窗口部件，而是使用该数据实现更具视觉趣味的目标：

```
<table><tr>
    <td>

        <div dojoType="dojox.charting.widget.Chart2D"
                style="width: 300px; height: 300px;">
            <div class="series" name="Series A" store="tableStore"
                    valueFn="Number(x)"></div>
        </div>

    </td><td>

        <div dojoType="dojox.charting.widget.Chart2D"
            theme="dojox.charting.themes.PlotKit.glue"
            fill="'#999'" style="width: 300px; height: 300px;">
            <div class="axis" name="x" font="italic normal bold 10pt Tahoma"></div>
            <div class="axis" name="y" vertical="true" fixUpper="major"
                includeZero="true" font="italic normal bold 10pt Tahoma"></div>
            <div class="plot" name="default" type="Areas"></div>
            <div class="plot" name="grid" type="Grid"></div>
            <div class="series" name="Series A" store="tableStore"
                valueFn="Number(x)*100" stroke="'#666666'" fill="'#b3b3b3'"></div>
        </div>

    </td>
</tr></table>
```

在前面的标记中，我们声明了两个窗口部件。第一个窗口部件要比第二个简单得多，它只声明了 dojox.charting.widget.Chart2D 窗口部件的一个实例，将其连接到前面定义的数据源。这个窗口部件将显示为一个简单的线形图，没有任何类型的坐标轴和标签修饰。

第二个窗口部件则相反，它有几个附加的修饰对象。它同时定义了水平坐标轴和垂直坐标轴，包括字体以及一些其他选项。这个图表还有一个网格背景，线下方也有一块经过填充的区域。除了为线和区域定义颜色之外，还有一个新的函数，当把数据传入并绘制到图形上之前，会首先将数据传给该函数。

图 30-4 给出了这两个窗口部件的呈现结果。

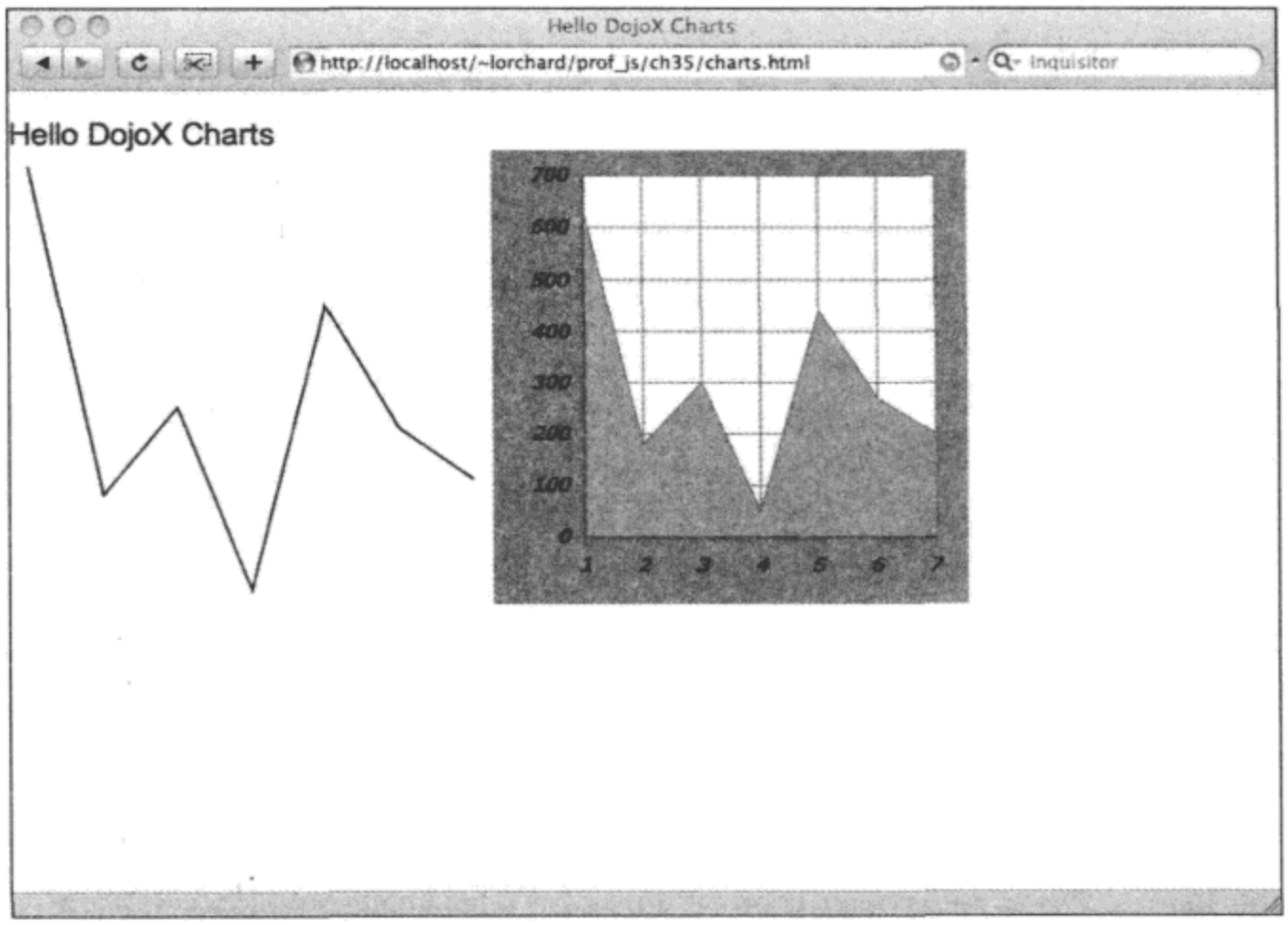


图　30-4

与 dojox.gfx 演示一样，这个图表窗口部件的快速示例只演示了 dojox.charting 提供的部分功能而已，而且超出了本书的介绍范畴。要查看示例和测试，一定要深入研究 Dojo 发行版本，尤其是下面目录中的内容：

```
dojox/charting/tests/
```

在其中可以找到一些演示 dojox.charting 方方面面的示例和测试，包括更多地通过编程方式构造图表以及使用三维图形实现更丰富的图表和数据展示。

30.6　使用编码和加密例程

在 DojoX 子项目中，有几个在数据编码、加密和压缩方面比较有用的模块。这些模块在基于浏览器的应用程序中看似作用有限，但是它们使用起来非常方便。

30.6.1　生成 MD5 散列值

dojox.encoding.digests.MD5 模块实现了 MD5 加密散列算法，它主要用于确信一组数据的完整性，还可以方便地为给定字符串生成一个安全的唯一标识符。

为了演示这一点，请考虑下面的标记和代码：

```
<div id="ex_md5">
    <h2>Hello MD5</h2>
    <form class="url_form">
        URL:
        <input type="text" size="50" class="url"
            value="http://decafbad.com/blog/" />
        <input type="submit" />
    </form>
    <dl class="urlinfo"></dl>
</div>
```

这段标记构建一个小型表单，它接受输入一个 URL，该地址将在下面的代码中用于从社会化书签网站 Delicious 那里获取 JSON 源：

```
dojo.require("dojox.encoding.digests.MD5");
dojo.require('dojo.io.script');

dojo.query('.url_form', 'ex_md5')
    .connect('onsubmit', function(ev) {
dojo.stopEvent(ev);

var url = dojo.query('.url', 'ex_md5')[0].value;

var url_md5 = dojox.encoding.digests.MD5(
    url, dojox.encoding.digests.outputTypes.Hex
);

var json_url =
    'http://feeds.delicious.com/feeds/json/url/data?hash='+url_md5;

dojo.query('.urlinfo', 'ex_md5')
    .empty()
    .addContent('<dt>Loading...</dt><dd>'+url_md5+'</dd>');

dojo.io.script.get({
    url: json_url,
    callbackParamName: 'callback',
    handle: function(resp, io_args) {
        dojo.forEach(resp, function(urlinfo) {
            dojo.query('.urlinfo', 'ex_md5').empty();
            for (key in urlinfo) {
                var val = urlinfo[key];
                if (key == 'top_tags') {
                    var tags = [];
```

```
                    for (tag in val)
                        tags.push(tag + ' (' + val[tag] + ')');
                    val = tags.join(", ");
                }
                dojo.query('.urlinfo', 'ex_md5')
                    .addContent("<dt>"+key+"</dt><dd>"+val+"</dd>");
            }
        });
    }
});

});
```

上面代码的关键之处在于 dojox.encoding.digests.MD5 的使用：Delicious 提供了一组 JSON 源来报告为指定的 URL 保存的书签数量以及这些标签中使用的热门标签的摘要信息。需要注意的是，需要将该 URL 使用 MD5 进行散列，以便形成 JSON 源 URL。

因此，使用 dojo.io.script.get()或 dojo.query()获取 JSON 源，并使用最终数据中这个 URL(用来计算 MD5 值)的详细信息来动态更新页面。

30.6.2 采用 Base64 编码数据

Base64 编码方案提供了只使用一组 64 种字符来表达完整的 8 位数据的方式。Base64 中使用的字符集由 A~Z、a~z、0~9、+和/组成，其中所有的字符都要能够避开特定的转义字符和序列，在容器和转换格式的结构和语句中使用这些转义字符和序列。

基本上，在任何可以安全传送普通的 7 位 ASCII 内容的地方，均可以使用 Base64 传送完整的 8 位数据，例如图像和非英语文本。这包括电子邮件、XML，甚至是几乎所有现代操作系统上的剪贴板内容。可以在如下网址中阅读更多有关 Base64 的内容：

```
http://en.wikipedia.org/wiki/Base64
```

下面的标记搭建了一个测试平台来试验 DojoX 中的 Base64 实现：

```
<div id="ex_base64">
    <h2>Hello Base64</h2>
    <form>
        Raw: <br />
        <textarea cols="70" rows="3" class="raw"></textarea>
        <br />
        Base64: <br />
        <textarea cols="70" rows="3" class="base64"></textarea>
        <br />
        <button class="encode">encode with base64</button>
        <button class="decode">decode from base64</button>
    </form>
</div>
```

上述标记的功能由下面的代码实现：

```
dojo.require("dojox.encoding.base64");
```

```
dojo.query('.encode', 'ex_base64')
    .onclick(function (ev) {
        dojo.stopEvent(ev);

        var raw = dojo.query('.raw', 'ex_base64')[0].value;

        var byte_array = [];
        for(var i=0; i<raw.length; i++) {
            byte_array.push(raw.charCodeAt(i));
        }

        dojo.query('.base64', 'ex_base64')
            [0].value = dojox.encoding.base64.encode(byte_array);
    });

dojo.query('.decode', 'ex_base64')
    .onclick(function (ev) {
        dojo.stopEvent(ev);

        var byte_array = dojox.encoding.base64.decode(
            dojo.query('.base64', 'ex_base64')[0].value
        );

        var raw_array = [];
        for(var i=0; i<byte_array.length; i++) {
            raw_array.push(String.fromCharCode(byte_array[i]));
        }

        dojo.query('.raw', 'ex_base64')
            [0].value = raw_array.join("");
    });
```

DojoX 在 dojox.encoding.base64 模块中提供了自己的 Base64 实现，而上面的代码将标记中的按钮连接起来，相应地编码和解码给定文本区域中的内容。

注意，Base64 编码器处理的是表示 8 位字节的数值数组，因此前面的编码函数和解码函数都必须相应地从字符串转换成数组以及从数组转换成字符串，这样才能与文本区域协调一致。

但是，在这个简单的代码中存在一个缺陷：它假设“原始”文本区域中的数据是普通的 ASCII，因此字符串中的每个字符都直接对应到一个 8 位的字节。但如果粘贴或输入非 ASCII Unicode 字符(例如日语文本)，那么这个演示就会失败，这是因为 Unicode 字符需要使用多个字节来表达，从而在这里 raw.charCodeAt()返回的值将超出 8 位的表达范围。

但是，这里并不会深究诸如 UTF-8 这样的 Unicode 编码，而是将其留给读者来完成。如果对此感兴趣，那么可以在下面这个公共域加密程序包中找到一个采用 JavaScript 实现的 UTF-8 编码例程：

```
www.fourmilab.ch/javascrypt/index.html
```

查找 utf-8.js 文件，它里面含有 Unicode/UTF-8 编码和解码例程，可以用于使前面的代码更加

健壮。

30.6.3　采用 Blowfish 加密数据

前两节处理单向的散列和双向的数据编码，而接下来的模块则处理利用密钥进行对称加密。下面的示例标记搭建了一个测试平台。

```
<div id="ex_blowfish">
    <h2>Hello Blowfish</h2>
    <form>
        Key:<br />
        <input type="text" value="asecretkey" width=20" class="key" />
        <br />
        Clear: <br />
        <textarea cols="70" rows="3" class="clear"></textarea>
        <br />
        Crypt: <br />
        <textarea cols="70" rows="3" class="crypt"></textarea>
        <br />
        <button class="encrypt">encrypt with blowfish</button>
        <button class="decrypt">decrypt with blowfish</button>
    </form>
</div>
```

这段标记由 3 个文本字段组成，它们用来输入以下信息：密钥、明文以及经过 Base64 编码的加密数据。然后，使用下面的代码为上面的标记提供功能：

```
dojo.require("dojox.encoding.crypto.Blowfish");

dojo.query('.encrypt', 'ex_blowfish')
    .onclick(function(ev) {
        dojo.stopEvent(ev);

        var key   = dojo.query('.key', 'ex_blowfish')[0].value;
        var clear = dojo.query('.clear', 'ex_blowfish')[0].value;

        dojo.query('.crypt', 'ex_blowfish')[0].value =
            dojox.encoding.crypto.Blowfish.encrypt(clear, key);
    });

dojo.query('.decrypt', 'ex_blowfish')
    .onclick(function(ev) {
        dojo.stopEvent(ev);

        var key   = dojo.query('.key', 'ex_blowfish')[0].value;
        var crypt = dojo.query('.crypt', 'ex_blowfish')[0].value;

        dojo.query('.clear', 'ex_blowfish')[0].value =
            dojox.encoding.crypto.Blowfish.decrypt(crypt, key);
    });
```

这段代码将 encrypt 和 decrypt 按钮与用来练习 Blowfish 算法实现(由 DojoX 中的 dojox.encoding.crypto.Blowfish 模块提供)的处理程序连接起来。我们应该能够将文本放入到第一个文本区域，然后在第二个文本区域中看到加密的结果，反之亦然，只要密钥一致即可。输入一个新的密钥，之前加密的数据就无法按照预期还原。

注意，如果试着使用非 ASCII 文本作为明文或密钥，那么这个演示也同样会失败。这是因为 Blowfish 的 DojoX 实现要求字符串由 8 位字节组成，就像前面的示例一样。我们可以通过以下方式来解决这个问题：在传入 encrypt()函数之前将字符串编码成 UTF-8，然后在从 decrypt()函数返回之后将字符串从 UTF-8 解码成 Unicode。

在 dojox.encoding 名称空间下有几个可用例程。一定要记得查看这些例程，从中可以找到压缩算法以及其他几个可能有用的数据操作工具。

30.7　导航 JSON 数据结构

CSS 和 dojo.query()使用选择器来辅助标识 DOM 中的元素，而 dojox.jsonPath 提供了一种声明性表达式语言来标识 JavaScript 中的数据以及 JSON 数据结构。这个程序包由前面描述的独立项目贡献而成，因此可以在下面的 URL 中找到一些有关这种类似 XPath 的表达式语言的资料：

- http://code.google.com/p/jsonpath/
- http://goessner.net/articles/JsonPath/

为了帮助理解这个程序包，下面的标记提供了之前用来演示 Django 模板支持的搜索表单的一个变体：

```
<div id="ex_search">

    <form class="search_form">
        Search:
        <input type="text" size="30" class="search"
            value="decafbad" />
        <br />

        Path:
        <input type="text" size="30" class="path"
            value="$.ResultSet.Result..Title" />
        <br />

        <input type="submit" value="search" class="search_button" />
        <br />

        <textarea cols="70" rows="15" class="results"></textarea>
    </form>

</div>
```

这个表单的主要变化在于增加了另一个表单字段，应该在其中输入一个 JSONPath 表达式。这个表达式的结果将出现在该表单底部的结果 textarea 元素中。

现在，下面的代码提供了使用 Yahoo! Web 搜索源的一个变体来实现表单的功能。

```
dojo.require('dojo.io.script');
dojo.require("dojox.jsonPath");

dojo.query('.search_button', 'ex_search')
    .onclick(function(ev) {
        dojo.stopEvent(ev);

        // Build up the JSON URL for search results.
        var search_url = 'http://search.yahooapis.com/'+
            'WebSearchService/V1/webSearch?'+
            dojo.objectToQuery({
                appid: 'appid_goes_here',
                output: 'json',
                results: 5,
                query:   dojo.query('.search', 'ex_search')[0].value
            });

        // Insert a loading message.
        dojo.query('.results', 'ex_search')
            [0].value = 'Loading...';

        // Fetch the JSON feed for search results...
        dojo.io.script.get({
            url: search_url,
            callbackParamName: 'callback',
            handle: function(resp, io_args) {
                var matches = dojox.jsonPath.query(
                    resp, dojo.query('.path', 'ex_search')[0].value
                );
                                dojo.query('.results', 'ex_search')
                    [0].value = dojo.toJson(matches, true);

            }
        });
    });
```

一旦运行这段代码，就可以尝试几个不同的 JSONPath 表达式，如下所示：

- 引用数据结构的根：

```
$
```

- 查找具体属性的值：

```
$.ResultSet.totalResultsAvailable
```

- 查找数据结构中所有名为 Title 的属性：

```
$..Title
```

- 查找具有指定 Title 值的 Result 对象：

```
$.ResultSet.Result[?(@.Title=="0xDECAFBAD")]
```

进一步的示例和文档请参阅这个程序包自带的文件。例如，请查看如下文件中的一些示例：

```
dojox/jsonPath/README
```

这个程序包提供了一个强大的工具，可用来在客户端中筛选数据，而不是为了自定义视图和内容片段多次向服务器发送请求。可以让服务器交付完整的数据集合，然后使用 JSONPath 表达式进行查询和扩展。

30.8　研究 DojoX 的其他功能

本章只给出了 DojoX 子项目中潜藏的丰富功能的一个示例。我们无法充分地使用文本讨论所有功能；您可以深入到各个目录，查看其中有什么可用功能。尽管在您阅读本书时可能会有更多的进展，但下面给出了在编写本书时实验性代码中包含的部分额外程序包：

- dojox.analytics　用户行为和交互的实时浏览器到服务器度量。
- dojox.av　用于处理音频和视频资源的抽象层和包装器。
- dojox.collections　高级 JavaScript 数据类型和迭代器。
- dojox.color　用来转换和处理各种颜色表达方式的实用工具。
- dojox.cometd　用来处理 Cometd(AJAX 的一种持久化连接替代技术)的代码。
- dojox.data　用来处理来自 XML 源、Web 服务、DOM 等位置中的数据的实用工具。
- dojox.date　日期操作实用工具。
- dojox.flash　实现 JavaScript 与 Flash 对象之间的双向通信的工具。
- dojox.gfx3d　用于处理 3D 图形的模块。
- dojox.grid　一种用于导航数据集的基于网格和表的强大窗口部件。
- dojox.highlight　一种语法高亮显示引擎。
- dojox.image　用于处理图像和图像集合的窗口部件和实用工具。
- dojox.lang　JavaScript 的概念性扩展，包括功能性编程辅助函数。
- dojox.layout　附加的实验性布局管理器。
- dojox.math　高级数学函数。
- dojox.off　Dojo offline 程序包用来帮助创建在连接网络和断开网络连接时均可用的 Web 应用程序。
- dojox.presentation　Dojo Presentation 程序包提供了幻灯片演示引擎的起点。
- dojox.sketch　一种跨浏览器绘图编辑器。
- dojox.storage　一个用来处理各种跨浏览器的本地客户端存储机制的模块。它远不止是 cookie。
- dojox.string　字符串操作，包括 sprintf 和 tokenize。
- dojox.timing　高级的定时构造和排序。
- dojox.uuid　按照 RFC 4122 标准生成通用唯一标识符字符串。
- dojox.wire　一种通用的数据绑定和服务调用库。
- dojox.xml　XML 操作实用工具，包括一个采用 JavaScript 实现的 XML 分析器。

作为功能和实验性功能的孵化器，DojoX 包含大量的程序包。其中一些程序包提供一到两个简单的新函数(例如 dojox.string)，而其他程序包则提供完整的应用程序(例如 dojox.sketch)或者构建 Web 应用程序的全新方式(例如 dojox.off)。如果有最新的 Subversion 检出版本，那么一定会在 dojox 子目录中看到更多的程序包。

30.9 本章小结

这是本书第Ⅳ部分的最后一章。我们快速浏览了 DojoX 子项目以及许多可用的不同实验性代码程序包。

需要理解的是，这一特殊的章节将很快过时，如果对 Dojo 项目的最新进展感兴趣，那么一定要经常在 Dojo Project 网站(http://dojotoolkit.org/)上查阅有关这些开发的最新消息，还可以在 SitePen Labs 网站(http://sitepen.com/labs/dojo.php)上找到一些即将推出的工具集。

第Ⅴ部分

MooTools

第 31 章：利用 MooTools 增强开发

第 32 章：操作 DOM 以及处理事件

第 33 章：简化 AJAX 以及处理动态数据

第 34 章：构建用户界面以及使用动画

正如 MooTools 的主页上所宣称的那样，MooTools 是一个简洁的、模块化的、面向对象的 JavaScript 框架。与本书已经描述的所有其他框架相比，MooTools 更是一种原生的 JavaScript 工具集。jQuery 实现了自己的领域特有的语言，与之不同的是，MooTools 坚定地以标准 JavaScript 为基础并对其进行增强。与 YUI 提供的类似于 Java 的隔离模块和按需依赖项系统(Dojo 也应用了这样的系统)相反，MooTools 通过 HTML 脚本标记加载并依赖于 JavaScript 的内置命名空间和模块化工具(通过对象字面值和闭包)。

人们通常将 MooTools 与 Prototype 相比，因为它们都采用了同样的方式将功能注入到 JavaScript 环境中原生对象的原型。但 MooTools 远不仅仅是 Prototype 的克隆，它提供了一个清晰的、强大的、自始至终均保持牢固的一致性标准的 API。在此基础之上，MooTools 提供了动画和 UI 工具(类似于 Scriptaculous 为 Prototype 提供的工具)，而且一直致力于提供更加轻量级的替代品。

MooTools 的模块化和可扩展设计允许只把自己需要的部分框架包含进来，同时允许在运行时创建子类或把自己的函数注入到现有的 MooTools 原型中，从而引入自定义功能。在这方面，MooTools 充分利用了 JavaScript 作为一门基于原型的动态语言的优势，而不是设法将其塑造成其他语言的模式。在接下来的几章内容中，我们将从该框架的 CSS 选择器代码、DOM 属性操作以及自定义事件等方面提供的可扩展性中看到这一点。

最后，MooTools 的增量式构造特性意味着，所有用于实现更高级的 DOM 和 AJAX 工具的便利工具以及构建块本身也都可用作独立的实用工具，可以用在自己的代码中。总而言之，MooTools 是 JavaScript 的一个扩展包，不管是环境基础还是现代框架的华丽功能，从头到尾都是如此。

MooTools 中的“Moo”代表“My Object Oriented(我的面向对象的)”JavaScript Tools。

第31章

利用 MooTools 增强开发

MooTools 不仅仅是一种页面操作工具集或 AJAX 实用工具库，它还为 JavaScript 开发提供了基本的增强。这包括像散列这样的新数据结构以及对诸如数组、字符串和数字这样的现有类型进行增强。此外，MooTools 为实践面向对象编程概念(例如类、继承以及混入)提供了一组工具。

本章内容简介：

- 获取 MooTools
- MooTools Core

31.1 获取 MooTools

如果对 MooTools 感兴趣，那么下面有几种方式可用来获取该框架的稳定版本或开发版本。

31.1.1 下载最新的 MooTools 发布版本

最新的 MooTools 发布版本如图 31-1 所示，可在如下网址中找到该版本：

```
http://mootools.net/download
```

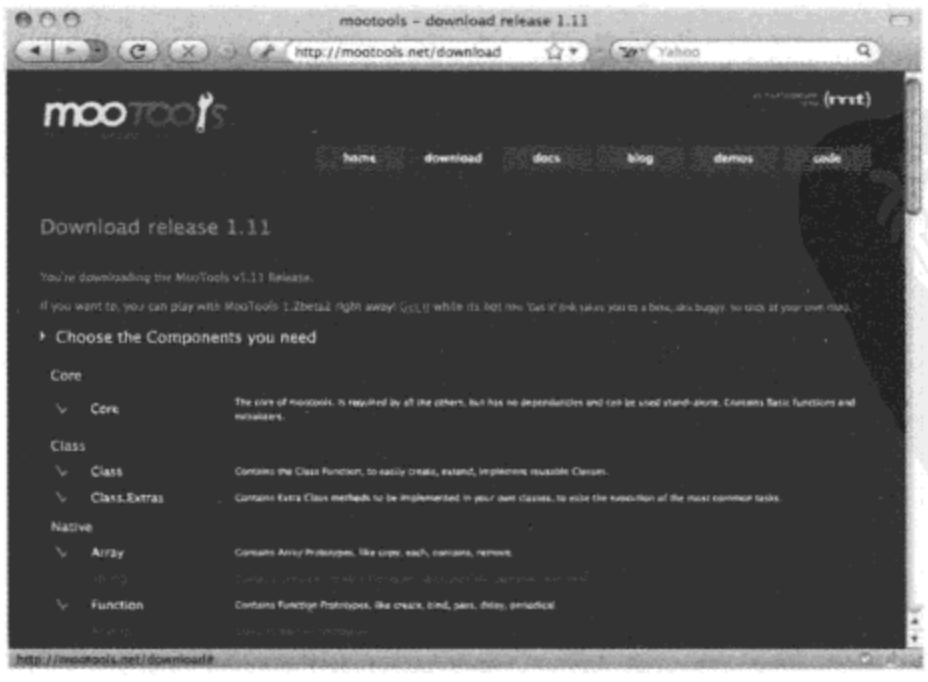


图 31-1

在这里可以找到 MooTools Core 库，该库已经应用了各种各样的压缩和代码缩减方案，从而在用于页面中时获得更快的下载速度。但是，可以在下面的 URL 中构建一个个性化的 MooTools 副本：

```
http://mootools.net/core
```

在这里可以看到图 31-2 所展示的模块化的框架结构。这个下载页面提供了该发布版本中可用的所有核心模块的可定制清单。可以只选取所需的部分，而相应的依赖项也将自动选中。这种动态下载构建程序还提供了一个选项来压缩接收到的库文件。

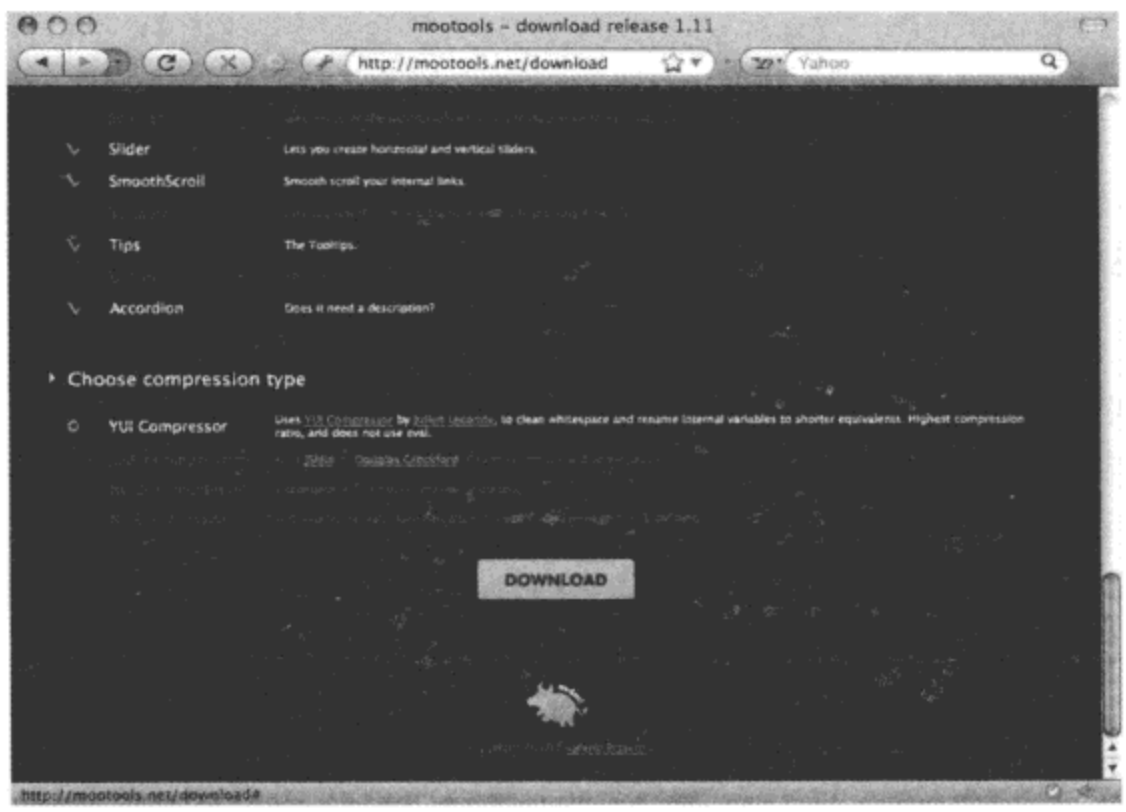


图 31-2

可以从 MooTools More 集合中获得额外的一组插件和模块，使用下面的另一个构建工具可以获得该集合：

```
http://mootools.net/more
```

这里给出了除 MooTools Core 之外的又一个适合于包含的 JavaScript 库，它包括了 MooTools 团队创建的所有扩展组件。在后面几章中，我们还将介绍这些组件中的大部分，因此可以将这些组件全部下载，以后如果需要为自己的项目定制一个缩减版本的库，那么可以返回到这个构建程序。

因此，与一些框架不同的是，我们可以在自定义 MooTools 构建版本出厂之前就对其进行组装并优化。如果希望在构建过程中进行定制，那么这可能或多或少是令人满意的方式。不管怎样，您可能希望下载几个版本的 MooTools。例如，一个版本包含所有的模块但不进行压缩(用于开发)，而另一个版本只包含几个手动选取的经过压缩的模块(用于生产站点)。

在页面的<head>中添加下面的标记就可以开始使用这些下载的文件：

```
<script type="text/javascript" src="../mootools-1.2.1-core-yc.js"></script>
<script type="text/javascript" src="../mootools-1.2-more.js"></script>
```

31.1.2　尝试正在开发的 MooTools 版本

如果希望跳过便利的自定义发布构建工具而直接钻研最新版本，那么 MooTools 提供了对正处于开发阶段的开发人员代码的公开访问。

1. 获得 Git 检出

通过使用命令行 Git 客户端，可以调用类似于下面的命令来获取所需的一切文件：

```
git clone git://github.com/mootools/mootools-core.git
git clone git://github.com/mootools/mootools-more.git
```

这将传送构成 MooTools 框架的所有文档、测试和原始源代码。但与网站提供的自定义构建过程不同，这个 Git 储存库不会提供该框架的单一包含版本，因此需要自行使用包含的 build.rb 脚本(采用 Ruby 编写)来构建它。这个脚本的操作将留给读者进行练习。

2. 下载最新内部版本

如果不想费力地从源代码构建 MooTools，那么下面的网址还提供了一个 MooTools Core 的每小时内部版本：

```
http://mootools.net/download/get/mootools-core-edge.js
```

按照这种方法下载的库可用作包含文件，就像下面这样：

```
<script type="text/javascript" src="../mootools-core-edge.js"> </script>
```

注意，这个每小时内部版本并不包含 MooTools More 提供的插件或模块。

31.2　研究 MooTools Core

MooTools Core 虽然很小，但它把大量有用的功能打包进来，这些功能的有用之处不仅表现在为该框架的其余部分搭建好测试平台，而且通常还可以用来增强自己的 JavaScript 开发。

为了浏览 MooTools Core 提供的功能以及其他方面，下面给出了两个用来产生日志消息的 MooTools 形式的全局快捷方式：

```
function $log(msg) {
    return console.log(msg);
}

function $log_json(obj) {
    return $log( JSON.encode(obj) );
}
```

在本章演示各种功能的过程中，我们将大量使用这些快捷方式来提供数据结构以及返回值的某种可见性。第一个快捷方式$log()只是 Firebug 的 console.log()调用的更简洁方式。第二个快捷方式$log_json()练习使用 MooTools 的 JSON.encode()工具，将对象以及它们的属性转化成具有描述性的字符串。

31.2.1 检查 MooTools 版本

MooTools 框架的一项相对较新的功能是能够检查已经加载的是什么版本：

```
$log("** Checking MooTools version");

if (MooTools) {
    switch(MooTools.build) {
        case '%build%':
            $log("*** SVN checkout, current version " + MooTools.version);
            break;
        default:
            $log("*** Version " + MooTools.version + ", build " + MooTools.build);
            break;
    }
} else {
    $log("*** Unknown version/build of MooTools");
}
```

可以遵循上面代码中使用的逻辑或类似的逻辑，在不同版本的 MooTools 之间进行选择，或者在尝试运行之前验证某个版本是否可用。通常，我们应该能够将期望的 MooTools 版本与自己的代码配对，但这种版本检测机制在某些场合下也比较实用。

31.2.2 确定类型

JavaScript 提供了原生的 typeof 操作符来确定各种值和对象的类型，但在许多场合中，它对一些截然不同的类型均后退到只报告“object”类型。这正是 MooTools 提供有用的$type()函数的原因之一：

```
$log( $type( null ) );                  // false
$log( $type( arguments ) );             // "arguments"
$log( $type( { foo: 'bar' } ) );        // "object"
$log( $type( [ 1, 2, 3 ] ) );           // "array"
$log( $type( "hello" ) );               // "string"
$log( $type( 123 ) );                   // "number"
$log( $type( true ) );                  // "boolean"
$log( $type( function() { } ) );        // "function"
$log( $type( /^hello(.*)World$/ ) );    // "regexp"
$log( $type( $('para1') ) );            // "element"
$log( $type( $('para1').firstChild ) ); // "textnode"
$log( $type( $('para1').childNodes ) ); // "collection"
```

可以看到，全局函数$type()可以报告一个值是否为未定义(false)，或者该“对象”实际上是否是一个数组、字符串、数字、函数或其他的类型。利用这个实用工具检测到的类型的组合最终将非常方便。

例如，可以在数组类型和其他类型之间进行检测，以便确定某个函数接收的是单个参数还是一个参数列表，并在必要的时候将这个值包装到一个数组中。

31.2.3　检查已定义的值

可以使用原生的 JavaScript 构造 typeof obj == 'undefined'或 MooTools 的$type(obj)==false 来检测某个给定变量是否含有一个已定义的值，但 MooTools 还为这种用途提供了另一个便利的函数：

```
var foo = null;
$log( $defined(foo) ); // false

foo = "Hi there";
$log( $defined(foo) ); // true
```

31.2.4　选择已定义的值

一旦能够检测到已定义的值，那么第一个运用该功能的场合就是从一系列参数和默认值中选择一个已定义的值。对于这项任务，MooTools 提供了全局函数$pick()：

```
$log( $pick(    1,    2, 3) ); // 1
$log( $pick( null,    2, 3) ); // 2
$log( $pick( null, null, 3) ); // 3
$log( $pick(false,    2, 3) ); // false
$log( $pick(    0,    2, 3) ); // 0
$log( $pick( "",    2, 3) ); // ""
```

在上面的代码中，可以看到$pick()函数接受多个参数。所有这些参数都通过$defined()进行计算，并返回第一个已定义的值。在这段代码之前曾经提到过，这个函数在从用户提供的一组参数、配置值和默认值中进行选择以提取该集合中第一个已定义的值时非常有用。

31.2.5　选取随机数

虽然这个表达式实现的是一项小功能，只是从 10 到 100 之间随机挑出一个整数，但该表达式有点冗长：

```
parseInt( Math.random() * (100 - 10) ) + 10
```

根据正在完成的工作情况，可以需要频繁地执行类似的任务(例如在游戏中或者在创建随机动画时)。为此，MooTools 定义了$random()函数：

```
$log( $random(0, 10) );     // Integer between 0 and 10
$log( $random(10, 100) );   // Integer between 10 and 100
$log( $random(100, 1000) ); // Integer between 100 and 1000
```

31.2.6　获取当前时间

尽管下面的代码行数与挑选随机数一样，但为了实现获取当前时钟时间(单位为毫秒)，这段代码也相当冗长：

```
Date.now(); // Not always available
(new Date()).getTime();
```

因此，MooTools 为快速访问当前时间提供了$time()函数：

```
$log( $time() );
```

这个函数在设置定时器以及度量程序性能方面非常方便。此外，在日志消息中包含一个时间戳可有助于浏览应用程序中引发的事件的顺序。

31.2.7 清除定时器和时间间隔

由于 JavaScript 从技术上讲并不是一种多线程环境，因此需要将多任务处理通过协同方式放进多个较小的步骤中，然后利用定时器执行或以一定时间间隔重复执行。为此，可使用原生的window.setTimeout()和 window.setInterval()方法。

这两个方法返回不同类型的句柄，可用于终止下一次执行。利用 window.clearTimeout()方法可以取消一次性的定时器，而利用 window.clearInterval()方法可取消定期的重复执行。

有时候很难弄清楚一个给定句柄到底属于一次性的定时器还是定期的重复执行，出于这个原因，MooTools 提供了一个统一的可用于两种情况的$clear()函数：

```
var timer = setTimeout(function() {
    alert('Alert never happens');
}, 1000);
timer = $clear(timer);

var interval = setInterval(function() {
    alert('Alert never happens');
}, 1000);
interval = $clear(interval);
```

31.2.8 合并和扩展对象

MooTools 支持面向对象编程，而继承关系正是这个编程范例的重要组成部分。但是有时候，我们并不是完全采用这种方式，而是从合并或扩展数据结构中获得继承关系。因此，我们可以建立一组默认配置值，然后在它的基础之上设置用户提供的值。

全局函数$merge()和$extend()在实现这种方式方面非常有用。为了进行演示，考虑下面的数据结构：

```
var alpha = {
    foo: 'bar',
    baz: 'quux',
    test:  { one: 1, two: 2 }
};
var beta  = {
    thud:  'splat',
    xyzzy: 'magic',
    test:  { three: 3, four: 4 }
};
```

注意，上面的所有代码片段都有两个不同的属性，并且有一个共同的对象属性，该对象属性

本身在每个结构中具有不同的属性。现在，查看$merge()函数对这些结构做了什么工作：

```
// Returns recursively merged copy of alpha and beta, no side-effects.
var gamma = $merge(alpha, beta);

$log_json(alpha);
// {"foo":"bar","baz":"quux","test":{"one":1,"two":2}}

$log_json(gamma);
// {"foo":"bar","baz":"quux","test":{"one":1,"two":2,"three":3,"four":4},
// "thud":"splat","xyzzy":"magic"}
```

上面代码中的 alpha 和 beta 结构都未经修改，但 gamma 中的返回值已经接收到递归合并 alpha 和 beta 之后的结果(test 属性下包含的对象)。

$extend()函数完成的工作稍有不同，它修改原始结构之一：

```
// Returns modified copy of alpha, with properties overwritten by beta.
var delta = $extend(alpha, beta);

$log_json(alpha);
// {"foo":"bar","baz":"quux","test":{"three":3,"four":4},"thud":"splat",
// "xyzzy":"magic"}

$log_json(delta);
// {"foo":"bar","baz":"quux","test":{"three":3,"four":4},"thud":"splat",
// "xyzzy":"magic"}
```

在上面的代码中可以看到，alpha 结构已经被修改，它的属性被 beta 重写。此时的 gamma 变量只包含 alpha 结构的一个副本。

在某种程度上，$extend()函数的工作方式类似于面向对象编程中的类继承机制，但可能有一点刚好相反：beta 的功能注入到 alpha 中，而不是由 beta 建立一个子类。

31.3　使用数组扩展

到目前为止，我们看到的 MooTools 核心功能均是全局的便利函数。从这里开始，我们将开始了解 MooTools 如何通过添加新方法来扩展原生浏览器对象，以便取得功能的跨浏览器一致性或引入新功能。

这里要介绍的第一项原生功能是 Array 对象。在现代版本的 JavaScript 中，这个对象有多个用于操作和处理其内容的便利方法。例如，我们可以从下面网址中了解 Mozilla 浏览器中有哪些可用方法：

```
http://developer.mozilla.org/en/docs/Core_JavaScript_1.5_Reference:Objects:Array
```

实际上，MooTools 添加的众多扩展中有许多是上述文档中描述的原生 Array 方法的冗余实现。之所以提供这些方法，是因为少数没有原生实现的浏览器可以通过安装补丁的方式来提供一致的功能。我们可以参考这份文档来从另一个角度审视这些方法。

此外，一定要查阅 MooTools 的文档，这是因为本节并没有把所有 Array 扩展方法都涵盖在内。

31.3.1 使用.each()和.forEach()处理数组项

对列表中的项进行的最有用的处理之一是在该列表中的每一项上执行有用的操作。为此，可以使用 Array 的.forEach()方法以及它的别名.each()：

```
var list = [
 'foo', 'bar', 'baz', 'quux',
 'xyzzy', 'plugh', 'thud', 'splat'
];

var someObj = {

    someThing: 'value of some thing',

    doIt: function() {

        list.forEach(function(item, idx) {
            $log(idx + ': ' + item + ' (' + this.someThing + ')' )
        }, this);

        list.each(function(item, idx) {
            $log(idx + ': ' + item + ' (' + this.someThing + ')' )
        }, this);
    }

};
someObj.doIt();
```

上述代码产生的日志文本消息输出应该是把以下内容重复两遍：

```
0: foo (value of some thing)
1: bar (value of some thing)
2: baz (value of some thing)
3: quux (value of some thing)
4: xyzzy (value of some thing)
5: plugh (value of some thing)
6: thud (value of some thing)
7: splat (value of some thing)
```

.forEach()和.each()的第一个参数是一个函数，将把列表中的项和当前索引传给该函数。在这个函数中，可以对列表项执行任何处理，而且返回值将被忽略。

.forEach()和.each()的第二个参数是一个可选的对象作用域。如果指定该参数，那么它将变成第一个参数传入的函数内的变量的本地值。上面的代码演示这个参数可让匿名函数访问在其中调用.forEach()和.each()循环的对象作用域。

1. 使用$each()快捷方式

还有一点值得注意的是，有一个全局快捷方式$each()可以应用于几乎所有类似于列表的数据：

```
$each(
    ['foo', 'bar', 'baz'],
    function(item, idx) {
        $log( idx + ': ' + item );
    }/*,
    optional_scope */
);
// 0: foo
// 1: bar
// 2: baz

$each(
    {oof:'foo', rab:'bar', zab:'baz'},
    function(value, key) {
        $log( key + ': ' + value );
    }/*,
    optional_scope */
);
// oof: foo
// rab: bar
// zab: baz
```

```
$each(
    document.getElementsByTagName('p'),
    function(el) {
        $log( el.firstChild.nodeValue );
    }/*,
    optional_scope */
);
// Open your Firebug console for log messages!
// Here's another paragraph for reference.
```

这个$each()快捷方式可以处理数组、对象甚至 DOM 查找方法返回的节点集合。

2. 使用$A()快捷方式创建数组

提及快捷方式，还有一个$A()快捷函数，在把几乎所有类似于列表的数据转换成原生数组时，这个函数非常有用：

```
function foo() {
    $A(arguments).each(function(item, idx) {
        $log(idx + ': ' + item);
    });
}
foo('bar', 'baz', 'quux', 'xyzzy');
// 0: bar
// 1: baz
```

```
// 2: quux
// 3: xyzzy
```

与$each()类似，$A()函数可应用于 DOM 节点列表和其他集合。

31.3.2 过滤和映射数组项

在.forEach()和.each()方法中，函数处理列表项之后的返回值会被忽略。但在接下来的两个方法.filter()和.map()中，它们的返回值将变得非常重要。

1. 使用数组方法.filter()

可以使用.filter()方法生成一个由所有匹配查询条件(在一个函数中实现)的列表项组成的列表子集：

```
$log(
    list.filter(
        function(item, idx) {
            return item.length <= 3
        } /*, optional_scope */
    )
    .join(', ')
);
// foo, bar, baz
```

在上面的代码中，只有长度为 3 及以下的项才会被选入子集。true 或 false 值相应地决定了列表中的特定项是否位于子集中。与.forEach()类似，这个函数的参数也是列表项和当前数组索引。尽管这里没有用到，但可以为这个函数传入一个对象作用域参数作为第二个参数。

2. 使用数组方法.map()

.filter()用来判断列表项是否存在，而.map()方法用来确定由哪些列表项的值形成最终列表：

```
$log(
    list.map(
        function(item, idx) {
            return item.length
        } /*, optional_scope */
    )
    .join(', ')
);
// 3, 3, 3, 4, 5, 5, 4, 5
```

在上面的代码中，原始列表中的每一项都通过指定函数变换成在最终数组中对应的字符串长度。此外，这里还接受了可选的对象作用域参数。

31.3.3 检查数组项的内容

在上面的两个方法中，处理每个数组项的函数的返回值用来影响最终的输出数组。但在下面

的函数中，输出值仅是一个 Boolean 值 true/false 或一个整数。

1. 使用数组方法.some()

考虑下面的.some()方法演示：

```
$log(
    list.some(
        function(item, idx) {
            return item.length <4
        }/*, optional_scope*/
    )
);
// true

$log(
    list.some(
        function(item, idx) {
            return item.length> 5
        }/*, optional_scope */
    )
);
// false
```

.some()方法的作用是判断列表中是否有一些项符合给定函数的条件。如果这个函数在处理列表中的任何一项时返回 true，那么.some()最终的返回值也将是 true。

2. 使用数组方法.every()

执行.some()方法的相反操作的是.every()方法，如下所示：

```
$log(
    list.every(
        function(item, idx) {
            return item.length> 2
        }/*, optional_scope */
    )
);
// true

$log(
    list.every(
        function(item, idx) {
            return item.length <4
        }/*, optional_scope */
    )
);
// false
```

对于.every()方法，要想返回值为 true，那么数组中的所有项都必须使该函数返回 true。如果任何一项的返回值为 false，那么.every()方法就会返回 false 值。

3. 使用数组方法.indexOf()、.contains()和.erase()

对于检查列表中是否存在特定项来说，下面这些方法非常有用：

```
$log( list.indexOf('xyzzy') );  // 4
$log( list.contains('xyzzy') ); // true

list.erase('xyzzy');

$log( list.indexOf('xyzzy') );  // -1
$log( list.contains('xyzzy') ); // false
```

.contains()方法返回是否在列表中找到某一项，而.indexOf()方法则返回给定值在列表中的位置(如果未能找到给定值，那么返回-1)。

这里介绍的另一个方法.erase()可用来将给定值的所有实例删除。

31.3.4 将数组项转换成对象属性

下面这个.associate()方法把给定列表的值作为列表(在其上调用该方法)中各项的键关联起来，从而将一个数组转换成一个对象。考虑下面的代码：

```
var obj = list.associate([
 'oof', 'rab', 'zab', 'xuuq'
]);
$log_json(obj);
// {"oof":"foo", "rab":"bar", "zab":"baz", "xuuq":"quux"}
```

可以看到，这个方法的第一个参数是一个数组。所有这些值都将作为键与在其上调用.associate()方法的数组中的每个值配对，最终形成一个返回对象。

31.3.5 扩展与合并数组

在了解如何处理数组项并查询其内容之后，现在研究如何混合数组的内容。

1. 使用数组方法.extend()

数组扩展方法.extend()可用来向数组中附加值：

```
var list = [
 'foo', 'bar', 'baz', 'quux', 'xyzzy', 'plugh', 'thud', 'splat'
];
list.extend([ 'foo', 'alpha', 'beta', 'gamma' ]);
$log( list );
// ["foo", "bar", "baz", "quux", "xyzzy", "plugh", "thud",
//    "splat", "foo", "alpha", "beta", "gamma"]
```

这个数组将作为.extend()方法的参数传入，该方法将这个数组的每个值都附加到在其上调用该方法的数组中。注意，这个列表会被就地修改，而.extend()的返回值将是原始列表自身。

2. 使用数组方法.include()

值得注意的是，.extend()方法并没有特意阻止重复值。下面这个.include()方法则不会附加数组中已有的值：

```
var list = [
 'foo', 'bar', 'baz', 'quux', 'xyzzy', 'plugh', 'thud', 'splat'
];
list.include('alpha').include('foo');  // Notice the chaining?
$log( list );
// ["foo", "bar", "baz", "quux", "xyzzy", "plugh", "thud",
//    "splat", "alpha"]
```

注意，尽管这里两次调用.include()方法，其中一次调用带有参数'alpha'，而另一次调用带有参数'foo'，但是只有第一次调用才会把值添加到列表中。这是因为在第二次调用中，'foo'值已经位于列表中。

3. 使用数组方法.combine()

.combine()方法提供了上述功能的基于列表的对等形式：

```
var list = [
    'foo', 'bar', 'baz', 'quux', 'xyzzy', 'plugh', 'thud', 'splat'
];
list.combine(['alpha', 'foo', 'bar', 'baz', 'beta']);
$log( list );
// ["foo", "bar", "baz", "quux", "xyzzy", "plugh", "thud",
//    "splat", "alpha", "beta"]
```

与.extend()方法类似，.combine()方法接受一个列表作为参数。但与.include()方法类似，这个方法并不允许向列表中添加重复的值。

31.3.6　展平嵌套数组

有时候，需要将一个由列表构成的列表展平为一个值数组。这就是.flatten()方法的用处所在：

```
var list = [ ];

list.push(['foo', 'bar', 'baz']);
list.push(['alpha', 'beta', 'gamma']);

$log( list );
// [["foo", "bar", "baz"], ["alpha", "beta", "gamma"]]

$log( list.flatten() );
// ["foo", "bar", "baz", "alpha", "beta", "gamma"]
```

可以看到，上面的代码构造了一个包含两个嵌套数组的数组。调用.flatten()方法将生成一个包含了嵌套列表中的所有值的单个列表。下面是一个更加复杂的示例：

```
$log(
    list.map(function(i) {
        return i.map(function(j) {
            return [ j, j.length ]
        })
    })
    .flatten()
);
// ["foo", 3, "bar", 3, "baz", 3, "alpha", 5, "beta", 4, "gamma", 5]
```

在这段代码中，使用一对嵌套的.map()调用来构建一个由列表项及其长度组成的嵌套列表。在末尾调用.flatten()方法，生成一个由列表项的值及其长度交替组成的简单列表。

31.3.7 利用.link()应用选择规则

数组扩展方法.link()可用来将一些相当复杂的选择规则应用到对象属性构建过程中：

```
var list = [
 'foo', 'bar', 'baz', 'quux',
 'xyzzy', 'plugh', 'thud', 'splat'
];
var obj = {
    'three1': function(key) { return key.length == 3 },
    'three2': function(key) { return key.length == 3 },
    'foo' : function(key) { return key == 'foo' },
    'x1'    : function(key) { return key.indexOf('x') != -1 },
    'x2'    : function(key) { return key.indexOf('x') != -1 }
};

var obj2 = list.link(obj);

$log_json(obj);
// {}

$log_json(obj2);
// {"three1":"foo", "three2":"bar", "x1":"quux", "x2":"xyzzy"}
```

.link()方法可将一组选择规则应用到一个列表上。最终返回的对象是应用所有规则之后的结果。

在上面的代码中，我们定义了一个列表，还有一个由指定函数组成的对象字面值。当在这个列表上调用.link()方法并将该对象作为参数传入时，这个数组中的每一项都将依次传入每个指定的函数中。

第一个返回 true 的函数将导致该列表项成为返回值对象中对应属性名的值。一旦匹配，就会将规则从集合中删除。由于这里有重复规则(例如 three1 和 three2)，因此每个规则匹配一个不同的 3 字母的值，这是因为每项规则在成功匹配之后就会被删除。

例如，如果希望某个函数以任意顺序接受多个参数，但根据参数类型(例如对象、字符串等)利用参数名称来匹配参数值，那么这个方法非常有用。

31.4　使用散列数据结构

尽管在 JavaScript 中可以将普通对象作为散列或关联数组结构使用，但是这样对象就会失去这些结构在其他语言中实现的一些有用方法。例如，获取仅由对象中所有的键或值组成的列表，或者判断给定值是否存在或给定值的键是什么，这些操作非常不自然。

31.4.1　定义散列和散列快捷方式

MooTools 在这里并没有扩展基础对象原型本身，而是提供了一个 Hash 类作为 JS 对象的包装器。下面的代码演示了创建 Hash 对象的几种方式：

```
var h1 = new Hash({
    foo: 'bar',
    baz: 'quux',
    xyzzy: 'thud',
    plugh: 'fred',
    x123: 'y456'
});

var h2 = $H({
    foo: 'bar',
    baz: 'quux',
    xyzzy: 'thud',
    plugh: 'fred',
    x123: 'y456'
});

var h3 = $H(h1);
```

创建 Hash 对象的第一种方式是只使用 new 关键字，而第二种方式则是使用一个全局快捷函数$H()，这两种方式在功能上是等同的。

第三个示例生成现有散列或对象的一个副本，但其有趣的地方在于该副本是独立的，这意味着对任何原生类型(数组、字符串、对象等)的值进行的修改都不会改变原始值的嵌套数据类型。通过使用全局函数$unlink()来实现该功能，它生成给定数据结构的一个嵌套副本。

31.4.2　设置与获取键和值

虽然可以像访问普通 JavaScript 对象那样访问 Hash 对象，但 Hash 对象提供了几个便利方法，可以让代码的意图更加清晰：

```
$log( h2.hasValue('fred') );    // true
$log( h2.hasValue('alpha') );   // false

$log( h2['foo'] );              // "bar"
$log( h2.get('foo') );          // "bar"
h2.erase('foo');
```

```
$log( h2['foo'] );              // typeof h2['foo'] == "undefined"
$log( h2.get('foo') );          // null

h2.set('foo', 'bar');
$log( h2.keyOf('bar') );        // "foo"

$log( h2.getKeys() );           // ["foo", "baz", "xyzzy", "plugh"]
$log( h2.getValues() );         // ["bar", "quux", "thud", "fred"]
```

在上面的代码中，使用.hasValue()方法确定在数据结构中给定值是否存在于某个键名之下。接着，使用.get()方法获取某个键的值，这与使用 h2['foo']或 h2.foo 作为直接访问表达式是等同的。然后，使用.set()方法为某个键设置值，使用.keyOf()方法报告给定值的键是什么。最后，使用.getKeys()和.getValues()方法分别报告散列中使用的键和值。

在这里值得注意的是，Hash 方法区别于使用普通 JavaScript 对象的主要方面是，每个 Hash 方法均使用了基础 JS 对象方法.hasOwnProperty()。这意味着如果在某个地方确实对基础 JS 对象进行了扩展(不同于 Hash 包装器)，那么这些扩展将出现幼稚的“for key in obj”循环。但 Hash 类的方法注意到这一点，它们只报告这个特殊数据结构中引入的特定键和值。

可以从下面的博客文章中阅读更多有关这种差异及其意义的内容：

```
http://yuiblog.com/blog/2006/09/26/for-in-intrigue/
```

31.4.3 映射和过滤散列

与 Array 扩展类似，Hash 实例提供了几个类似的方法，它们使用值和键，而不是使用值和数字索引。首先是.map()和.filter()方法：

```
$log_json(
    h1.map(
        function(value, key) {
            return value + " (" + key + ")";
        }/*, optional_scope */
    )
);
// {"foo":"bar (foo)","baz":"quux (baz)","xyzzy":"thud (xyzzy)",
//    "plugh":"fred (plugh)"}

$log_json(
    h1.filter(
        function(value, key) {
            return key.length <=3
        }/*, optional_scope */
    )
);
// {"foo":"bar","baz":"quux"}
```

在上面的代码中，我们可以看到.map()的工作方式与它的 Array 对等方法相似，通过该函数的返回值在散列的值上进行操作。注意，它只处理值；使用这个方法修改散列的键并不容易。

.filter()方法也与它的 Array 对等方法类似，它用来确定将哪些键和值放进返回值散列中。.map()

和.filter()都不会修改原始散列。

31.4.4　使用.every()和.some()检查散列

Hash 类提供的.every()和.some()方法也与它们的 Array 对等方法类似，其作用是查询所有的值和键是否满足函数中定义的查询条件：

```
$log(
    h1.every(
        function(value, key) {
            return /(\d+)/.test(key)
        }/*, optional_scope */
    )
); // false

$log(
    h1.some(
        function(value, key) {
            return /(\d+)/.test(key)
        }/*, optional_scope */
    )
); // true
```

在这个示例中再次可以看到，.every()和.some()使用的函数的参数是值和键，而不是 Array 对等方法中使用的值和索引。而且，.every()和.some()方法分别查询是否所有的或任意的键/值对匹配给定函数中实现的查询条件。

31.4.5　扩展与合并散列

与 Array 扩展类似，散列也提供了.extend()、.include()和.combine()方法。下面的代码演示了这些方法的工作方式：

```
h2.extend({
    foo: 'alpha', bar: 'beta', thingy: 'gamma'
});
$log_json( h2 );
// {"baz":"quux","xyzzy":"thud","plugh":"fred","x123":"y456",
// "foo":"alpha","bar":"beta","thingy":"gamma"}

h2.include('foo', 'ignored_value');
h2.include('stuff', 'delta');
$log_json( h2 );
// {"baz":"quux","xyzzy":"thud","plugh":"fred","x123":"y456",
// "foo":"alpha","bar":"beta","thingy":"gamma","stuff":"delta"}

h2.combine({
    foo: 'ignored',
    stuff: 'ignored',
    more: 'added value'
```

```
});
$log_json( h2 );
// {"baz":"quux","xyzzy":"thud","plugh":"fred","x123":"y456",
// "foo":"alpha","bar":"beta","thingy":"gamma","stuff":"delta",
// "more":"added value"}
```

.extend()方法接受一个散列或JS对象参数并修改原始散列，并使用它的参数传入的数据结构覆盖找到的键。.include()方法接受独立的键/值对，但拒绝替换散列中任何现有的数据。最后，.combine()方法的工作方式介于.extend()和.include()之间，它接受一个散列参数以混入原始散列，但拒绝篡改任何现有数据。

31.4.6 将散列转换成URL查询字符串

由于查询字符串在建立Web请求和构造URL的过程中占有如此重要的地位，因此.toQueryString()是Hash类中尤其有用的方法：

```
$log(
    $H({
        foo: "The quick",
        bar: "brown fox",
        baz: "@#$%^&*",
        thud: [ 'alpha', 'beta', 'gamma' ]
    })
    .toQueryString()
);
// foo=The%20quick&bar=brown%20fox&baz=%40%23%24%25%5E%26*&thud[0]=alpha
// &thud[1]=beta&thud[2]=gamma
```

在上面的代码中，我们使用快捷方式$H()快速构造一个Hash对象，然后调用Hash类的.toQueryString()方法。该方法的返回值是一个包含该散列对象的所有键和值的字符串，每一项都经过适当的URL转义。此外还要注意的是，嵌套数组已经展平为枚举参数，有点类似于PHP为给定的命名参数指定多个值的方式。

31.5 使用字符串扩展

MooTools还为原生的String类型提供了几个有用的扩展。这些方法对于执行几种测试、转换甚至一些基于模板的简单格式化操作非常有用。

31.5.1 检查字符串内容

第一个方法.contains()可用来判断给定的子字符串是否出现在某个字符串中：

```
var str2 = "brown, bluegreen, black, red";
$log( str2.contains("blue") );          // true
$log( str2.contains("blue", ", ") );     // false
$log( str2.contains("bluegreen", ", ") ); // true
```

.contains()的第一个参数是要搜索的子字符串，可选的第二个参数可用来指定一个在搜索子字符串过程中使用的分隔符。

在上面的代码中，待搜索的字符串包含一个由逗号分隔的列表。因此，第一个示例搜索简单的子字符串“blue”，而第二批的两个示例搜索特定的分隔项“blue”和“bluegreen”。当调查 CSS 类名中有哪些 CSS 类或者执行简单的列表搜索时，这种方式非常有用。

更为一般地，.test()方法能够查找某个子字符串或将正则表达式应用于字符串：

```
var str = "The quick brown fox jumps over the lazy dog";
$log( str.test("brown") );              // true
$log( str.test("bar") );                // false
$log( str.test(/^([a-zA-Z\s]+)$/) );    // true
$log( str.test(/^(\d+)$/) );            // false
```

前两次调用.test()方法都只是搜索子字符串。第二批的两次方法调用则应用正则表达式，其中第一次调用检查字母和空白符，而第二次调用检查所有数字。注意，正则表达式对象也提供了一个.test()方法，因此字符串原型中的这个方法基本上就是它的一个替代表示。

31.5.2　将字符串转换成数字和颜色

JavaScript 为字符串值到数字的转换提供了 parseInt()和 parseFloat()函数，但作为一项便利措施，MooTools 将这些函数包装起来作为字符串扩展.toInt()和.toFloat()，如下所示：

```
$log( "100".toInt() );     // 100
$log( "99.96".toInt() );   // 99
$log( "100".toFloat() );   // 100
$log( "99.96".toFloat() ); // 99.96
```

采用这种方式编写的代码要比使用原生 JavaScript 函数更清晰。

下面这一对方法.hexToRgb()和.rgbToHex()可用于操作元素和样式中的颜色值：

```
$log('*** String.hexToRgb and String.rgbToHex');
$log( "#aabbcc".hexToRgb() );           // "rgb(170,187,204)"
$log( "#abc".hexToRgb() );              // "rgb(170,187,204)"
$log( "rgb(170,187,204)".rgbToHex() );  // "#aabbcc"
$log( "rgb(186,218,85)".rgbToHex() );   // "#bada55"
```

通常，使用十六进制颜色值对应的十进制 RGB 数字来进行数学计算更加容易，但在编写标记和 CSS 时，十六进制颜色值往往更加容易记忆。因此，这两个方法对于根据条件来回转换颜色值的不同表示方式非常有帮助。

31.5.3　使用简单的替换模板

将一系列字符串和变量连接起来构建一个字符串通常是不适当的做法，而调用一种全功能的模板语言来完成这项工作则通常过于复杂。MooTools 的.substitute()方法提供了一个非常好的折中处理方式。

```
var str = "Hello, my name is {name}, and I am a {sign}.";
```

```
$log(
    str.substitute({
        name: "l.m.orchard", sign: "Scorpio"
    })
);
// Hello, my name is l.m.orchard, and I am a Scorpio.

var str = "Hello, my name is %%name%%, and I am a %%sign%%.";
$log(
    str.substitute(
        { name: "l.m.orchard", sign: "Scorpio" },
        /\\?%%([^%]+)%%/g
    )
);
// Hello, my name is l.m.orchard, and I am a Scorpio.
```

.substitute()方法接受一个对象或Hash对象作为它的第一个参数，该参数的值将用来替换字符串中的指定占位符。可选的第二个参数是一个正则表达式，它用于重新定义占位符模式。上面的代码演示了默认占位符的用法以及如何指定新模式。

在将格式化模板与它们的使用场合分开(以便实施国际化或者让代码更加清晰)时，这个方法非常有用。

31.5.4 执行其他的转换

尽管本节并不奢望涵盖所有的扩展，但下面的各种转换还是非常值得专门提出来：

```
$log( "    lots   of   space ".trim() );   // "lots   of   space"
$log( "    lots   of   space ".clean() );  // "lots of space"

$log( "this-is-a-test".camelCase() );      // thisIsATest
$log( "thisIsATest".hyphenate() );         // this-is-a-test

$log( "this is a test".capitalize() );     // This Is A Test
```

.trim()方法将字符串末尾的多余空白符删除，而.clean()方法完成同样的工作，而且将多个空格压缩成单个空格。

.camelCase()和.hyphenate()方法是一对互为补充的方法，它们用于在字符串的驼峰式大小写和连字符表达式之间进行转换。当在JavaScript代码和CSS中应用CSS属性时，这一对方法非常方便，因为这里的通用规则是，在JavaScript中属性采用驼峰式大小写，而在CSS中采用连字符命名方式。

最后，.capitalize()方法可用来确保标题和专有名词的大小写正确。

31.6 使用函数扩展

由于在JavaScript中函数本身就是带有原型的对象，因此可以将一些扩展方法附加到函数中。考虑到这一点，MooTools也在这个领域提供了一些改进。

31.6.1　将函数绑定到对象上下文

在 JavaScript 中，函数的一种有用但有些复杂的功能是作用域变量 this。一般而言，在函数执行期间，this 变量的可访问值反映出它被调用时所处的相关上下文。当作为某个对象的方法调用时，this 通常指向该对象。但当把函数作为某个事件的处理程序使用时，this 可以指向某个元素或者浏览器窗口自身。

但是在使用对象方法作为处理程序时，我们通常希望 this 仍然指向该方法所属的对象。关于这一点，请考虑下面的代码：

```
var SelfDestructTimer = {
    count:    5,
    interval: null,

    tick: function(msg) {
        $log(msg + " Tick... " + this.count);
        if (this.count-- <= 0) {
            $log("Boom!");
            $clear(this.interval);
        }
    },

    lightFuse: function(msg) {
        /* This doesn't work - 'this' for tick() will be the window itself:
        this.interval = setInterval(
            this.tick, 1000
        );
        */
        this.interval = setInterval(
            this.tick.bind(this, [ msg ]), 1000
        );
        $log("The fuse is lit!");
    }
};

SelfDestructTimer.lightFuse('Oh noes!');

// The fuse is lit!
// Oh noes! Tick... 5
// Oh noes! Tick... 4
// Oh noes! Tick... 3
// Oh noes! Tick... 2
// Oh noes! Tick... 1
// Oh noes! Tick... 0
// Boom!
```

在上面的代码中定义的对象实现了一个倒数读秒定时器，它是利用.bind()方法和 setInterval()函数来构建的。通常，传入 setInterval()作为处理程序的函数将看到 this 变量的值是一个指向浏览器窗口的引用。

但是，这里在这些函数上调用了这个新方法.bind()，它构建了一个包装器函数作为闭包，当调用这段代码时仍然能够访问原始对象的引用。因此，当将这个函数作为处理程序传入setInterval()时，该包装器将使用适当的上下文来调用这个对象方法以访问计数器和最终取消的时间间隔。还要注意的是，可以将一个参数数组作为可选的第二个实参传入.bind()，当调用包装器时，这些参数将传给绑定的方法。

注意，所有操作实际上与数组和散列上的.forEach()方法接受的第二个可选实参所完成的事情是一样的，只是可以用在其他的场合中。

31.6.2 间歇性地延迟和设置函数调用

由于前一个示例使用了定时器，因此下面这对扩展方法处理的是定时器和时间间隔。这些方法是.delay()和.periodical()，它们简化了 setTimeout()和 setInterval()函数的使用以及上面代码所示的.bind()方法的使用。

作为演示，请查看修改后的定时器对象：

```
var SelfDestructTimer_v2 = {
    count:    10,
    interval: null,

    tick: function(msg) {
        $log(msg + " Tick... " + this.count);
        if (this.count-- <= 0) {
            $log("Boom!");
            $clear(this.interval);
        }
    },

    cancel: function(msg) {
        $log(msg + " Self-destruct cancelled!");
        $clear(this.interval);
    },

    lightFuse: function(msg) {
        this.interval =

            this.tick.periodical(1000, this, [ msg ]);
        $log("The fuse is lit!");

        this.cancel.delay(5000, this, [ msg ]);
        $log("Considering cancellation!");
    }
};

SelfDestructTimer_v2.lightFuse('Banzai!');
```

这里并没有使用 setInterval()启动定时器，而是调用.periodical()函数方法。第一个参数是重复调用.tick()方法的间隔时间(单位为毫秒)。第二个参数用来绑定对象上下文，而最后一个参数是一

个参数列表，当调用.tick()方法时会传入这些参数。

类似地，这里使用一个新的函数方法.delay()(取代了 setTimeout()调用)来延迟调用.cancel()方法。.delay()方法的参数与.periodical()完全相同。此外还要注意，这两个方法均返回适当的句柄以供$clear()使用，以最终取消这类定时操作。

31.6.3　尝试带有潜在异常的函数调用

在有些情况下，我们需要试着执行可能会出现异常的操作，但要在异常出现时捕获并消除它：

```
function bad_function() {
    window.thisDoesntExist();
}
$log( bad_function.attempt() );
```

使用上面代码中的.attempt()方法，由试图调用不存在的窗口方法而导致的常见错误条件会被 MooTools 内部的 try/catch 代码块静默地捕获。

当用来在整个浏览器中查找可用的对象时(如下所示)，相关的全局函数$try()最为有用：

```
function(){
    return $try(
        function(){ return new XMLHttpRequest() },
        function(){ return new ActiveXObject('MSXML2.XMLHTTP') }
    );
};
```

$try()函数依次尝试执行传入的每个函数，直到其中一个函数成功地返回一个已定义的值。在上面的代码中，我们试着查找当前浏览器的 XMLHttpRequest 对象实现，该对象在不同的浏览器厂商之间以及不同的浏览器版本之间各不相同。

31.7　使用面向对象编程方法

尽管 JavaScript 确实支持面向对象编程，但它基于原型而不是类继承关系。由于使用第二种方式的程序员更多，因此前者可能稍微有点不好理解。但在基于原型的系统中，可以实现基于类的继承关系。

因此，MooTools 增强了 JavaScript 的基础原型系统，使其具有一点基于类的继承编程风格；还有几种方式用来实现现有类的事后增强。

31.7.1　构建类和子类

首先，下面这段代码构造一个简单的基类：

```
var Furniture = new Class({

    name: 'Furniture',

    isSelfAssembled: false,
```

```
    initialize: function() {
        $log("Furniture init!");
    },

    describe: function() {
        return '[' + this.name + '] ' +
            ( (this.isSelfAssembled) ? '(self assembled)' : '(pre-built)' );
    }

});
```

在 MooTools 中，所有类本身也都是一个名为 Class 的类的实例。当使用 new 关键字创建这个 Furniture 类的一个实例时，就会调用.initialize()方法。但除此之外，上面代码中并没有什么非常特别的地方。

现在，考虑如何声明 Furniture 的一个子类：

```
var CheapBookshelf = new Class({

    Extends: Furniture,

    name: 'CheapBookshelf',

    isSelfAssembled: true,

    initialize: function() {
        this.parent();
        $log("CheapBookshelf init!");
    }

});

$log('*** Creating a CheapBookshelf');
var a_bookshelf = new CheapBookshelf();
$log( a_bookshelf.describe() );

// *** Creating a CheapBookshelf
// Furniture init!
// CheapBookshelf init!
// [CheapBookshelf] (self assembled)
```

凭借这个类的定义中提供的 Extends 属性，CheapBookshelf 类首先继承 Furniture 类的一切内容，然后选择性地重写一些内容。这一点类似于前面介绍的全局函数$extend()。

因此，CheapBookshelf 类具有一个.describe()方法，而不需要声明一个该方法。但是，注意这个类重写它的父类的.initialize()实现，在这个方法实现的主体中，语句 this.parent()将导致执行已经重写的父类函数，因此可以让我们能够扩展父类的行为，而不是简单地将其替换。

为了进一步演示这一点，请考虑下面演示的另一个名为 Chair 的 Furniture 子类，它也提供了.describe()方法的新实现：

```
var Chair = new Class({

    Extends: Furniture,

    name: 'Chair',

    initialize: function() {
        this.parent();
        $log("Chair init!");
    },

    describe: function() {
        var desc = this.parent();
        return desc + " (I'm a chair!)";
    }

});

$log('*** Creating a Chair');
var a_chair = new Chair();
$log( a_chair.describe() );

// *** Creating a Chair
// Furniture init!
// Chair init!
// [Chair] (pre-built) (I'm a chair!)
```

可以看到，语句 this.parent()适用于任何方法(而不仅限于.initialize())，无论在什么地方使用该语句，它都会调用父类中的适当方法。在这个新的.describe()实现中可以看到，在返回值的过程中，我们可以将其捕获并进行处理。

借助上面代码中演示的行为，MooTools 中的 Class 对象构成了类继承关系以及重写父类行为的简单基础。

31.7.2　将方法和属性注入到现有类中

但是，除了普通的类继承关系之外，MooTools 还在 Class 对象中提供了.implement()方法，可以用来将新行为注入到现有类中。

```
Chair.implement({
    owner: null,
    assignSeating: function(name) {
        this.owner = name;
    },
    describe: function() {
        var desc = this.parent();
        return desc + " (owner: "+ this.owner +")";
    }
});
```

```
$log('*** Creating another Chair, with assigned seating');
var b_chair = new Chair();
b_chair.assignSeating("Alan Smithee");
$log( b_chair.describe() );

// *** Creating another Chair, with assigned seating
// Furniture init!
// Chair init!
// [Chair] (pre-built) (owner: Alan Smithee)
```

上面的代码说明，在 Chair 类上调用.implement()将会把指定的属性和方法混入到现有类中，从而允许后面实例化的对象访问新的数据和功能。但是有些事情需要注意：在新实现的.describe()方法中，this.parent()引用的是 Furniture 父类而不是 Chair 类。

这项功能非常有趣的一点在于，这是对现有类的事后增强。这个类可以属于 MooTools 框架，或者来自其他的第三方库，但不需要对原始的源代码进行修改，而且不需要与原来的作者进行协调。新的实现将插入到现有的类中并为现有的功能添加补丁。

扩展原生的浏览器对象

还有一个有趣的地方需要注意，.implement()方法的工作方式也与许多原生类型上的扩展方法类似，例如：

```
String.implement({
    join: function(list) {
        return list.join(this);
    }
});

$log( " -- ".join([1, 2, 3, 4, 5]) );
// 1 -- 2 -- 3 -- 4 -- 5
```

这基本上就是MooTools在Array、String和Function原型上安装所有原生扩展所使用的方式(前面已经介绍过)，但我们自己也可以使用这项功能。MooTools 试图避免的一件事情是将任何自定义代码安装到 Object 基类型的原型中(在该场合下选择 Hash 包装器类)。

31.7.3 实现混入类

因此，在定义子类时，指定 Extends 属性是声明父类的方式。但要注意的是，Extends 属性的期望值是一个 Class 实例，这意味着一个类只能继承一个父类。使用面向对象编程术语描述，这就是单一继承。

但是，使用多重继承将来自多个父类的功能作为“混入”功能累积起来通常非常有用。在 MooTools 中，可以通过在类定义中指定 Implements 属性来实现该功能，如下所示：

```
var Reclinable = new Class({
    recline: function() { $log('Reclining chair...'); }
});

var Massaging = new Class({
```

```
    massage: function() { $log('Initiating massage...'); }
});

var Heated = new Class({
    activateHeat: function() { $log('Warming up chair...'); }
});

var ComfyChair = new Class({
    Extends: Chair,
    Implements: [
        Reclinable, Massaging, Heated
    ]
});

$log('*** Creating a ComfyChair');
var c_chair = new ComfyChair();
$log( c_chair.describe() );
c_chair.recline();
c_chair.massage();
c_chair.activateHeat();

// *** Creating a ComfyChair
// Furniture init!
// Chair init!
// [Chair] (pre-built) (owner: null)
// Reclining chair...
// Initiating massage...
// Warming up chair...
```

与 Java 中 implements 关键字的概念不同的是，这里的混入类并不是抽象的接口类。在 Class 构造函数中将按照 MooTools 类定义中的 Implements 属性操作，使用前面描述的.implement()方法，将指定类的属性和方法的副本注入到新定义的类中。

但要记住.implement()方法的工作方式：复制过来的属性和方法会把类自身中以前定义的属性和方法覆盖，而不是将它们重写。所有的 this.parent()调用都将指向父类，而不是新的子类。这一点也同样适用于在类定义中使用 Implements 属性混入的一切内容。

1. 使用事件混入类

MooTools 提供的一个尤其有用的混入类就是 Events 类，它可以让我们在对象上实现事件作为处理程序函数的附着点，当引发该事件时，就会调用这些函数。考虑下面这个使用 Events 混入的类的实现：

```
var Firework = window.Firework = new Class({

 Implements: [ Events ],

    fuse_lit: false,
    delay:    null,
    color:    null,
```

```
    lightFuse: function(color) {
        this.fuse_lit = true;
        this.delay    = $random(500, 5000);
        this.color    = color;

        this.fireEvent('burst', this.color, this.delay);

        $log("Fuse lit @ {time}, {delay}ms delay, color {color}".substitute({
            time: $time(), delay: this.delay, color: this.color
        }));

    }
});
```

请注意上面代码中.fireEvent()方法的用法，它就是 Events 混入类所带来的方法。.fireEvent()方法接受事件名称作为参数，并且有一个可选的参数，可将其一起发送给事件处理程序；此外还有一个可选的延时(单位为毫秒)，表示在引发事件前等待指定的时间。

现在，查看如何将事件处理程序与.addEvent()方法连接起来：

```
var rocket1 = new Firework();
var rocket2 = new Firework();
var rocket3 = new Firework();

function watchFirework(color) {
    $log("Saw firework @ "+$time()+" burst "+color);
}

rocket1.addEvent('burst', watchFirework);
rocket2.addEvent('burst', watchFirework);
rocket3.addEvent('burst', watchFirework);
rocket1.addEvent('burst', function(color) {
    $log("Also saw firework @ {time} burst {color} after {delay}ms".substitute({
      time: $time(), color: this.color, delay: this.delay
    }));
});

rocket1.lightFuse('red');
rocket2.lightFuse('white');
rocket1.lightFuse('blue');

// Fuse lit @ 1207749778446, 3040ms delay, color red
// Fuse lit @ 1207749778447, 1195ms delay, color white
// Fuse lit @ 1207749778447, 3731ms delay, color blue
// Saw firework @ 1207749779638 burst white
// Saw firework @ 1207749781482 burst red
// Also saw firework @ 1207749781485 burst red after 3040ms
// Saw firework @ 1207749782174 burst blue
```

前 3 次.addEvent()调用将预定义的处理程序函数连接起来以响应引发的“burst”事件。第 4

次.addEvent()调用连接一个内联函数，在这个函数中可以看到 this 的值指向该事件引发时所在的对象。这个工具将在处理 DOM 事件时提供便利，但这里的演示说明，它是一个通用的工具，可以使用它来构建事件驱动的代码，而不仅限于用户界面中的事件处理。

2. 在类中提供选项和默认值

另一个比较有用的混入类是 Options 类。Options 混入类可以为对象建立一个默认选项集，从而可以非常方便地让用户在初始化新对象时指定值：

```
var NiftyThing = new Class({
    Implements: [ Options ],
    options: {
        rating:   50,
        color:    'red',
        duration: 1000
    },
    initialize: function(options) {
        this.setOptions(options);
    },
    describe: function() {
        return "[NiftyThing] " + JSON.encode(this.options);
    }
});
```

Options 混入类提供了一个.setOptions()方法，它的参数为保存在一个对象或 Hash 中的属性集合。这些属性将合并到对象的选项属性中。当按照如上代码所示在.initialize()方法中调用这个方法时，它为我们提供了一种快速、简单的初始化过程。

现在查看这项功能的实际使用情况：

```
var nifty1 = new NiftyThing();
$log( nifty1.describe() );
// [NiftyThing] {"rating":50,"color":"red","duration":1000}

var nifty2 = new NiftyThing({ rating: 100 });
$log( nifty2.describe() );
// [NiftyThing] {"rating":100,"color":"red","duration":1000}

var nifty3 = new NiftyThing({ rating: 25, quacks: false });
$log( nifty3.describe() );
// [NiftyThing] {"rating":100,"color":"red","duration":1000,"quacks":false}
```

如果没有给定选项，那么就全部使用默认值。在第二个对象实例中，只给出了 rating 属性，因此其余选项将使用默认值。在第三个对象实例中，有一个默认值被替换，而且将一个新的属性 quacks 合并进来。看过这些示例，您应该能够理解.setOptions()方法如何处理这些情况以及如何合并默认值与用户指定的属性。

3. 构建方法调用链

另一个值得注意的混入类是 Chain 类，在后面利用一系列方法调用编排动画时，这个类将变

得尤其重要。考虑下面这个使用了 Chain 类的类定义：

```
var Poet = new Class({

 Implements: [ Chain ],

   reader_name: null,

   initialize: function(reader_name) {
      this.reader_name = reader_name;

      // Compose the method call chain
      this.chain(this.line1);
      this.chain(this.line2);
      this.chain(this.line3);
      this.chain(this.line4);
   },

   read: function() {
      // Call the next method in the chain
       return this.callChain();
   },

   line1: function() {

      // The scope of 'this' is the Poet object, no need for .bind()
      $log(this.reader_name + ': Mary had a little lamb,');

      // The return value here is passed along by this.callChain()
      return true;

   },

   line2: function() {
      $log(this.reader_name + ': Its fleece was white as snow;'); return true;
   },
   line3: function() {
      $log(this.reader_name + ': And everywhere that Mary went,'); return true;
   },
   line4: function() {
      $log(this.reader_name + ': The lamb was sure to go.'); return true;
   }
});
```

Chain 混入类提供了两个新方法。第一个方法.chain()可用来向内部列表添加新方法，而第二个方法.callChain()调用列表中的第一个方法然后将其移除。有两点尤其要注意的是：在调用链中的方法时，this 指向这些方法所属的对象实例，而这个方法本身返回的值将由.callChain()方法传递。

下面的代码演示如何使用前面代码中的 Poet 类：

```
var poet1 = new Poet('Alan Smithee');
```

```
while (poet1.read()) {
    $log('next line...');
}
$log('done!');

poet1.read();  // Nothing happens; the chain is empty.

// Alan Smithee: Mary had a little lamb,
// next line...
// Alan Smithee: Its fleece was white as snow;
// next line...
// Alan Smithee: And everywhere that Mary went,
// next line...
// Alan Smithee: The lamb was sure to go.
// next line...
// done!
```

由于 Poet 类定义的调用链中的每个方法都返回 true 值，因此在.read()中.callChain()方法调用没有返回 true 的第一个实例将指示这个调用链的结束。因此，这里使用一个 while 循环就足以处理调用链。还可以使用一个时间间隔定时器，当调用链结束时将该定时器取消。

31.8　本章小结

本章深入研究了 MooTools Core，如何获得该框架的一个发布版本或开发版本副本，以及如何在页面中使用它。这里详细论述的这些类和扩展为接下来几章中要讨论的功能提供了基础。

在下一章中，我们将开始了解 MooTools 如何运用其核心中的基本功能来简化 DOM 操作，以及构建用于响应事件的用户界面。

第32章

操作 DOM 以及处理事件

通过扩展浏览器对象和 DOM 元素，MooTools 增强了基于浏览器的 JavaScript 的内置功能，使其拥有了众多用来查找和操作元素以及处理原生和自定义事件的便利方法。这种与 DOM 脚本处理的紧密集成使得我们能够使用简洁的、最少的代码来完成复杂的页面修改和用户界面构造工作。

在 MooTools 提供的增强中，我们可以找到一些便利的挂钩工具，可用来在框架内连接自己的增强代码。

本章内容简介：

- 查找元素
- 操作元素样式和属性

32.1 查找 DOM 元素

在本章中我们将看到，MooTools 提供了综合的便利工具集来查找元素以及修改页面的 DOM 结构。为了进行演示，请考虑下面的标记：

```
<div id="div2">
    <h2>Some list</h2>
    <ul id="list1" class="first-list">
        <li class="foo">Foo</li>
        <li class="bar">Bar</li>
        <li class="sublist">
            <span class="baz">Baz</span>
            <ol>
                <li>Alpha</li>
                <li>Beta</li>
                <li>Gamma</li>
            </ol>
            <span class="zab">Zab</span>
        </li>
        <li class="xyzzy">Xyzzy</li>
```

```
        <li class="quux">Quux</li>
    </ul>
</div>
```

这段标记提供了足够的复杂性来演示 MooTools 提供的一些便利工具。

32.1.1 使用$()和 ID 查找元素

MooTools 提供的第一个也是最简单的 DOM 导航工具是$()函数(其他许多 JavaScript 框架也提供了这个函数)，其用法如下：

```
$log( $('div2') );
// <div id="div2">
```

如果您在前一章中没有看到$log()函数，那么这里再回顾一下该函数，它并不是 MooTools 提供的函数。相反，它是一个定义用来包装 console.log()语句的快捷方式。

返回到代码示例，当使用一个字符串参数调用$()函数时，它可以按照指定 ID 查找 DOM 元素。但$()函数的参数还可以是一个指向现有 DOM 元素的引用，此时它充当一个传递操作，即直接返回基本上未经修改的元素：

```
$log( $( document.getElementsByTagName('div')[0] ) );
// <div id="div2">
```

但是，这里的“元素未经修改”并非完全正确：在检索元素时，使用原生浏览器方法与使用 MooTools 方法之间存在着一个重要的差别。

MooTools 对元素进行了一系列增强，为其提供了许多方法来处理 DOM 节点和结构；设置了一个处理程序，在卸载页面时处理元素引用以防止内存泄漏；并且为元素的元数据存储创建唯一的 ID。

这就是$()的第二种形式的方便之处，它为我们提供了一种方式来把所需的 DOM 元素从 MooTools 的方法中取出来并归入到工具集的管理范围。如果已经在使用 MooTools 的方法进行 DOM 遍历，那么通常并不需要考虑这一点。

32.1.2 使用$$()和 CSS 选择器查找元素

下一个用来查找元素的有用函数是$$()，它的用法如下：

```
$log( $$('li') ); // Only accepts tag names without Selectors
// [li.foo, li.bar, li.sublist, li, li, li, li.xyzzy, li.quux]
```

$$()函数的最基本形式可用于通过标记名称来查找元素列表。但如果在 MooTools 内部版本中包含了 Element.Selectors 模块，那么可以进一步使用$$()函数，利用 CSS3 选择器查找元素，如下所示：

```
$log( $$('ol>li') );
// [li, li, li]
```

MooTools 中的 Element.Selectors 和$$()实现并不支持完整的 CSS3 规范，但它们确实支持一个较大的、有用的子集。表 32-1 描述了一些已知能够用于 MooTools 的选择器。

表　32-1

选　择　器	作　　用
*	匹配所有元素
.class	匹配 CSS 类中含有'class'的元素
#foo	匹配 id="foo"的元素
E	匹配<E>元素
E F	匹配属于<E>元素的子节点的<F>元素
E>F	匹配属于<E>元素的直接子节点的<F>元素
E+F	所有直接位于<E>元素之前的<F>元素
E-F	所有跟在<E>元素之后的兄弟节点<F>元素
E[foo]	匹配带有属性 foo 的<E>元素
E[foo="bar"]	匹配<E foo="bar">元素
E[foo!="bar"]	匹配所有不含属性 foo="bar"的<E>元素
E[foo~="bar"]	匹配属性 foo(由空白符隔开的列表)中包含值 bar 的<E>元素，对于类似 CSS 类名构造的属性非常有用
E[foo^="bar"]	匹配属性 foo 以 bar 开头的<E>元素
E[foo$="bar"]	匹配属性 foo 以 bar 结尾的<E>元素
E[foo*="bar"]	匹配属性 foo 包含 bar 的<E>元素
E:nth-child(n)	匹配属于父元素第 n 个子节点的<E>元素
E:nth-child(odd)	匹配属于父元素的奇数子节点的<E>元素
E:nth-child(even)	匹配属于父元素的偶数子节点的<E>元素
E:nth-child(3n+1)	匹配属于父元素第 3n+1 个子节点的<E>元素(从第一个子节点开始)
E:first-child	匹配属于父元素第一个子节点的<E>元素
E:last-child	匹配属于父元素最后一个子节点的<E>元素
E:only-child	匹配属于父元素的唯一子节点的<E>元素
E:empty	所有的文本内容为空的<E>元素
E:contains(TXT)	所有的直接文本子节点包含 TXT 的<E>元素
E:not(...)	忽略所有上述选择器

使用和创建自定义 CSS 伪选择器

表 32-1 给出了 MooTools 中可用的 CSS3 选择器子集，但该工具集还提供了几个该规范尚未定义的额外伪选择器。表 32-2 给出了这些扩展的伪选择器。

表　32-2

选　择　器	作　　用
E:index(n)	匹配属于父元素第 n 个子节点的<E>元素
E:even	匹配属于父元素的偶数子节点的<E>元素
E:odd	匹配属于父元素的奇数子节点的<E>元素

由于这些伪选择器均是通过保存在名为 Selectors.Pseudo 的散列(由 Element.Selectors 模块提供)中的 JavaScript 函数来实现的，因此创建自己的新伪选择器并不困难。下面是一个可运行的示例：

```
Selectors.Pseudo.multicontains = function(text) {

    return text.split('|').some(function(part) {

        return (this.innerText || this.textContent || '').contains(part)

    }, this);

};
```

这段代码向 MooTools 散列集合 Selectors.Pseudo(它属于 Element.Selectors 模块)中添加一个新的名为“multicontains”的函数。

这几行代码中打包了大量的功能：该函数应该在一个元素上下文中被调用，并接受一个由管道字符(|)作为分隔符的字符串。如果这个上下文元素的文本内容中至少包含由管道字符作为分隔符的列表中的一个值，那么该函数返回 true。作为一个自定义选择器，其用法如下所示：

```
$log( $$('li:multicontains(Foo|Bar|Quux)') );
// [li.foo, li.bar, li.quux]
```

MooTools 中的 CSS 选择器分析器在散列 Selectors.Pseudo 中查找这个自定义伪选择器，提取参数字符串，并调用这个自定义函数来判断是否在结果中包含特殊的<li>元素。

32.1.3 导航 DOM 结构

原生 DOM 元素为导航文档树的父节点、子节点以及兄弟节点关系提供了几种基本方式。但是，这些只是基础方式。除了使用 ID 和 CSS 选择器查找元素之外，MooTools 还向元素中附加了一些增强方法，以简化从一个元素遍历到页面 DOM 结构中的其他单个元素或元素组的操作。

所有 DOM 节点均提供了一个名为 parentNode 的属性，它指向包含该节点的节点。可以构建循环，让这个指针从一个节点指向另一个节点来遍历子节点到父节点的关系，直到遍历到文档根节点。

考虑到这一点，查看下面使用 MooTools 提供的.getParent()和.getParents()方法的代码：

```
var ol = document.getElement('ol');

$log( ol.getParent() );
// <li class="sublist">

$log( ol.getParent('ul') );
// <ul id="list1" class="first-list">

$log( ol.getParents() );
// [li.sublist, ul#list1.first-list, div#div2, body, html]

$log( ol.getParents('div') );
// [div#div2]
```

可以看到，.getParent()方法返回一个父元素，并接受一个可选的参数来指定一个标记名称作为过滤条件。这个方法将继续在父元素层次结构中向上遍历，直到找到一个匹配查询条件的父元素，或者在到达文档根元素时返回空。

相应地，.getParents()方法返回多个父元素，它沿着父元素层次结构向上遍历文档并返回遍历过程中所有满足指定标记名称查询条件的元素(如果有的话)。

MooTools 提供了类似上面提到的两个方法的完整导航工具集合，这增强了已经出现在元素上的 DOM 导航指针的用法；每个工具都可接受一个可选的过滤条件，而且每个工具都可以在 DOM 结构中沿着不同的维度或方向进行遍历，如表 32-3 所示。

表　32-3

方　法	作　用
el.getElement(selector)	查找第一个匹配给定 CSS 选择器的子元素
el.getElements(selector)	查找所有匹配给定 CSS 选择器的子元素
el.getFirst(tag_name)	获取第一个匹配可选标记名称的子元素
el.getLast(tag_name)	获取最后一个匹配可选标记名称的子元素
el.getChildren(tag_name)	获取所有匹配可选标记名称的子元素
el.hasChild(child_el)	判断给定子元素是否为上下文元素的直接子元素
el.getParent(tag_name)	获取匹配可选标记名称的第一个父元素
el.getParents(tag_name)	获取匹配可选标记名称的所有父元素
el.getPrevious(tag_name)	获取匹配可选标记名称的前一个兄弟节点
el.getAllPrevious(tag_name)	获取匹配可选标记名称的前面的所有兄弟节点
el.getNext(tag_name)	获取匹配可选标记名称的后一个兄弟节点
el.getAllNext(tag_name)	获取匹配可选标记名称的后面的所有兄弟节点

通过标记名称来过滤元素列表

尽管可能非常明显，但仍然值得一提的是，到目前为止所描述的方法和辅助函数返回的元素列表基本上都是 JavaScript 数组。这意味着所有常见的数组方法都可以应用于 DOM 元素列表，包括诸如.each()、.filter()、.map()、.some()以及.every()这样的方法。

但更精确的陈述应该是：MooTools 中的元素列表是内置 JavaScript 数组的子类 Elements 的实例。这个子类提供的一项增强是：对于.filter()方法，既可以像通常那样使用一个实现选择查询条件的函数，也可以为其指定一个标记名称，就像前面描述的 DOM 导航方法一样，如下所示：

```
$log( $$('*').filter('li') );
// [li.foo, li.bar, li.sublist, li, li, li, li.xyzzy, li.quux]

var eles = $$('*')
    .filter('li')
    .filter(function(el) {
        return el.getParent().get('tag') == 'ol';
    })
    .map(function(el) {
        return el.get('text');
```

```
    });
$log( eles );
// ["Alpha", "Beta", "Gamma"]
```

在本章剩余部分中，我们将继续进一步了解 Elements 子类引入的数组方法，有些方法既能够用于单个元素，也可以用于元素列表。

32.2 操作元素样式和属性

一旦获得一个或多个 DOM 元素的句柄，下面就要实际地操作它们。首先要做的工作就是操作属于这些元素的显示样式、CSS 类以及其他属性。您可能已经猜到，MooTools 提供了一套齐备的工具来完成这些任务以及其他更多任务。

32.2.1 操作元素 CSS 类

由于 CSS 类名是在元素的类属性中指定的，因此可以直接使用原生的 className 属性或者.getAttribute()和.setAttribute()方法来访问和操作 CSS 类。但由于 CSS 类的值是由空格隔开的字符串，因此即便是检测是否存在某个 CSS 类这样的操作都可能相当复杂：

```
$log( /\s?first-list\s?/.test( $('list1').className ) );
// true
```

正是由于这个原因，MooTools 提供了几个扩展方法来负责检查、添加、删除以及切换一组 CSS 类。下面是前一个代码示例的 MooTools 实现：

```
$log( $('list1').hasClass('first-list') );
// true
```

元素上的.hasClass()方法的参数是一个字符串，返回值表示该 CSS 类是否出现在元素的集合中。

现在，针对本章开头给出的标记，下面的 CSS 代码提供了一些样式来演示 MooTools 提供的其他 CSS 方法：

```
<style type="text/css">
    .bordered { border: 2px dotted #000 }
    .grayed   { background-color: #ccc; margin: 0.25em 0 }
</style>
```

下面的代码演示如何使用.addClass()方法：

```
var first_item = document.getElement('ol>li');

first_item.addClass('bordered');

first_item.getAllNext().addClass('grayed');
```

这里使用 first_item 变量来存放标记中第一个有序列表的第一个子节点。在这个元素上，调用.addClass()方法将 CSS 类“bordered”添加到它的集合中。该操作会按照之前提供的 CSS 定义在

该元素周围放置简单的虚线边框。

.addClass()方法在这个示例中的第二次应用更有意思：它是在.getAllNext()方法返回值上的链式调用，这意味着既可以在单个元素上调用.addClass()，也可以在元素列表上调用它。顺便提一下，这是数组子类 Elements 提供的另一个数组扩展。

在图 32-1 中可以看出，其作用是紧跟第一个元素后面的所有兄弟节点都被赋予“grayed”CSS 类，这会让它们具有灰色的背景。

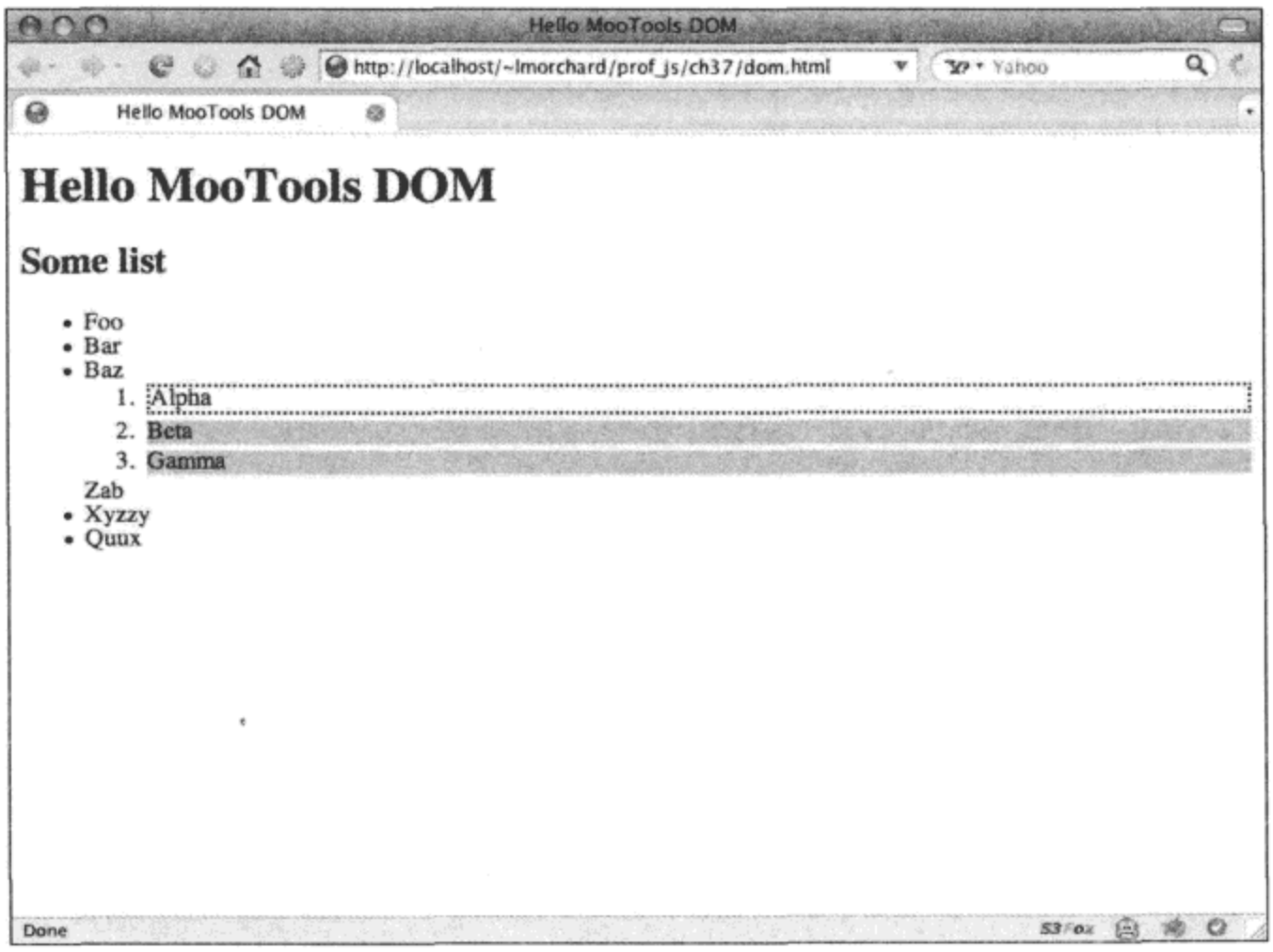


图　32-1

.addClass()的反面是.removeClass()方法，其用法如下：

```
$$('li.foo').addClass('bordered');
$$('li.foo').removeClass('bordered');
```

上面这两行代码的实际效果就是什么也不做：第一行代码将 CSS 类“bordered”添加到元素，而第二行代码将该 CSS 类移除。注意，与.addClass()方法一样，.removeClass()也可以用在元素列表上下文中。

最后，作为.addClass()和.removeClass()这两个方法的补充，MooTools 提供了.toggleClass()方法来交替地添加和移除 CSS 类：

```
var ele = document.getElement('li.quux');

ele.toggleClass('bordered');

var timer = ele.toggleClass.periodical(500, ele, [ 'bordered' ]);
```

```
$clear.delay(5000, ele, [ timer ]);
```

上面的代码首先直接调用元素方法.toggleClass()。然后设置一个重复执行的定时器，它每隔半秒将“bordered”CSS 类移除和添加，直到在 5 秒之后被另一个一次性的定时器取消。如果尝试运行该代码，那么它应该很容易显示出不断切换的视觉效果。

尽管图 32-2 并不是动画，但可以看到列表中的最后一个元素将获得不断切换的效果。

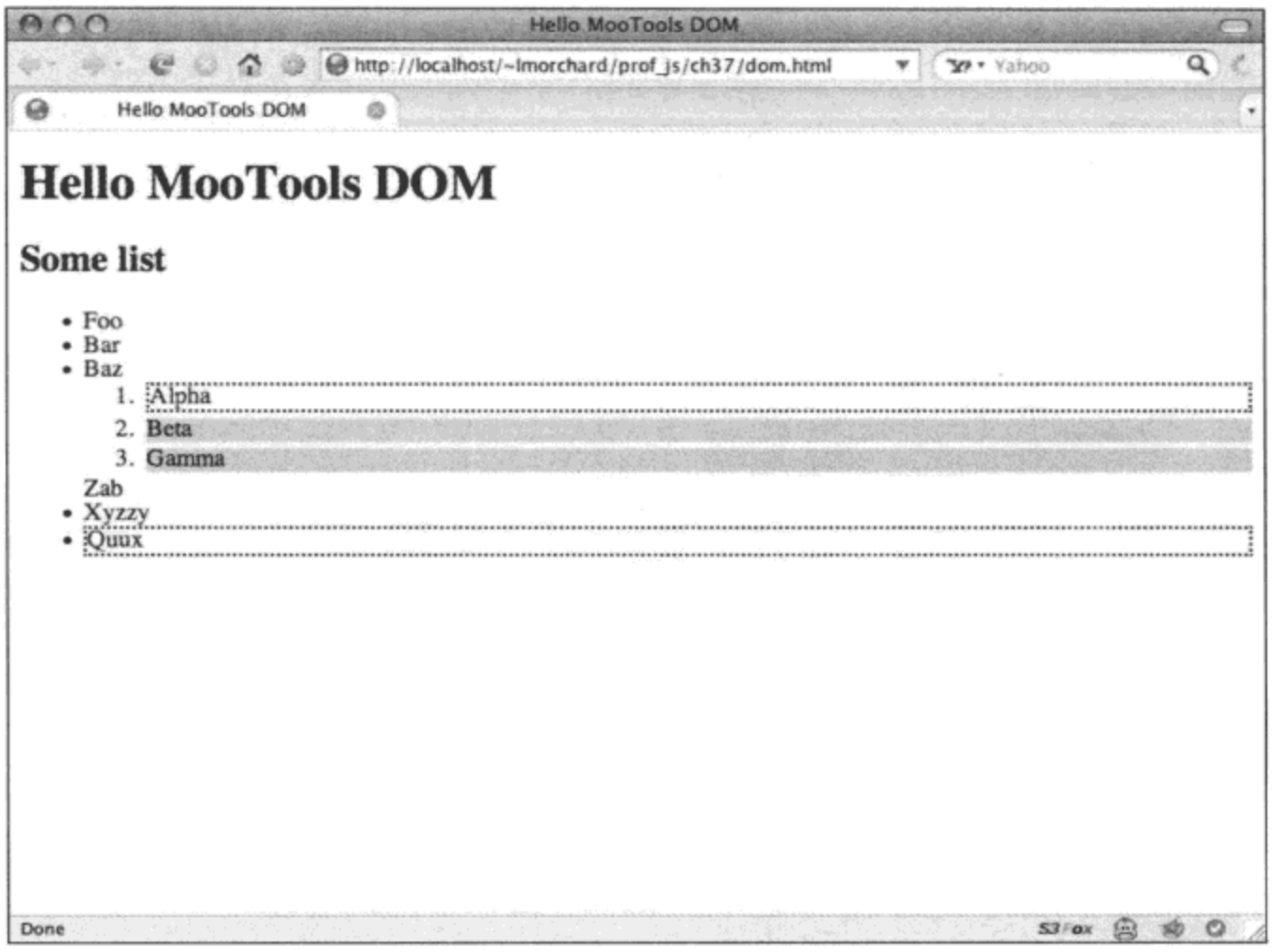


图　32-2

32.2.2　操作元素视觉样式

在一个单独的层叠样式表文件中定义样式，然后操作元素 CSS 类，这是在 JavaScript 中处理视觉样式最清晰且松散耦合性最好的方式，但有时候直接操作元素的视觉样式可能更加方便和有必要。而在这方面，MooTools 同样提供了一组非常方便的扩展方法。

第一个这样的方法是.setStyle()，其用法如下所示：

```
$$('li.bar').setStyle('font-weight: bold');
```

与前一节展示的.addClass()和.removeClass()方法类似，既可以在单个元素上调用.setStyle()方法，也可以在元素列表上下文中调用它。上述语句的作用是在带有 CSS 类“bar”的列表元素中呈现粗体文本。

虽然在一个或多个元素上设置一种样式确实有用，但更为常用的做法是一次性应用一组样式。为此，与.setStyle()方法相伴的是.setStyles()方法，这个方法接受一个类似下面的样式散列。

```
$$('li.bar').setStyles({
```

```
    fontWeight: 'bold',

    fontStyle: 'italic'

});
```

上面的代码将粗体与斜体组合呈现样式应用于所有带有 CSS 类“bar”的列表节点中包含的文本。这种调用功能强大，在一次高效的调用中就可以将一组样式应用于各种元素。还要注意的是，样式散列的键采用驼峰式大小写(camelCase)方式表示，而不是连字符分隔的 CSS 命名方式；这是一种相当一致的转换规则，以便让我们可以轻易地表达在 JavaScript 中命名的样式属性。

在了解如何设置样式之后，还要了解如何检查某个元素已经设置了什么样式。为此，MooTools 提供了.getStyle()和.getStyles()方法：

```
$log( $$('li.bar').getStyle('font-weight') );
// ["bold"]

$log_json( $$('li.bar').getStyles(

    'font-weight',

    'font-style'

));
// [{"font-weight":"bold","font-style":"italic"}]
```

这两个方法的用法都非常简单，下面查看它们的作用。

第一个方法.getStyle()获取单个样式的值。如果针对单个元素调用该方法，那么它的返回值是一个字符串。如果针对一个元素列表调用该方法，那么它的返回值是一个列出所有样式值的字符串列表。

第二个方法.getStyles()的参数是一个样式名称列表。如果针对单个元素调用该方法，那么它的返回值是一个由指定样式及其值组成的散列。如果针对一个元素列表调用该方法，那么它的返回值是一个由指定样式及其值组成的散列列表。

操作元素的不透明度

尽管元素的透明度或不透明度实质上只是另一种视觉样式，但由于它在不同的浏览器之间的实现差异，而且经常要用于淡入淡出变换和动画，因此形成了一个棘手的问题。为了帮助解决这个问题，MooTools 提供了一对方法，将元素不透明度的访问方式规范化：

```
$$('ol>li').setOpacity(0.75);

$log( $$('li').getOpacity() );
// [1, 1, 1, 0.75, 0.75, 0.75, 1, 1]
```

与.setStyle()和.getStyle()方法类似，既可以在单个元素上调用.getOpacity()和.setOpacity()方法，也可以在元素列表上下文中调用它们。参数值是一个介于 0 和 1 之间的小数，它表示元素介于透明(0)和不透明(1)之间的程度。

32.2.3 操作元素属性

除了CSS类和样式属性(property)之外，标记属性(attribute)通常是元素的下一个值得注意和增强的方面。MooTools 提供了一些有趣的扩展方法和跨浏览器规范化来辅助完成这个领域中的工作，稍后就会看到这方面的示例。

首先，考虑下面这段新的带有大量属性的标记示例：

```
<div id="div3">
    <h2>Some form</h2>
    <form id="form1">
        <ul>
            <li>
                <label for="alpha">Alpha</label>
                <input type="text" id="alpha" name="alpha" value="Some value" />
            </li>
            <li>
                <label for="beta">Beta</label>
                <input type="checkbox" id="beta" name="beta" />
            </li>
            <li>
                <label for="gamma">Gamma</label>
                <input type="text" id="gamma" name="gamma" readonly="readonly" />
            </li>
            <li>
                <label for="delta">Delta</label>
                <input type="checkbox" id="delta" name="delta" checked="checked" />
            </li>
        </ul>
    </form>
</div>
```

1. 获取元素属性值

记住上面的标记，查看 MooTools 向元素中引入的.getProperty()方法的演示：

```
$log( $('alpha').getProperty('type') );
// text

$log( $('alpha').getProperty('value') );
// Some value

$log( $('gamma').getProperty('readonly') );
// true

$log( $('alpha').getPrevious().getProperty('text') );
// Alpha

$log( $('alpha').getPrevious().getProperty('for') );
// alpha
```

.getProperty()方法与原生的 DOM 方法.getAttribute()或访问元素对象属性非常类似。实际上，.getProperty()方法提供的是一个经过合并的名称空间，每个属性名称自动映射到 DOM 对象属性访问或 DOM 对象.getAttribute()方法调用。

作为 DOM 对象属性直接访问处理的属性名称定义为 MooTools 内部对象 Element.Attributes.Props 的键，因此可以像下面这样产生自己的列表：

```
$log( $H(Element.Attributes.Props).getKeys() );

["html", "class", "for", "text", "compact", "nowrap", "ismap", "declare", "noshade",
"checked", "disabled", "readonly", "multiple", "selected", "noresize", "defer",
"value","accesskey","cellpadding","cellspacing","colspan","frameborder","maxlength",
"rowspan", "tabindex", "usemap"]
```

当调用.getProperty()时，上面代码中列出的每一个属性名称均映射成一个 DOM 对象属性。此外，对于其中一些 MooTools 属性，除了执行简单的名称映射之外还会进行特殊的内部处理(参见表 32-4)。

表 32-4

属　性	映　射
el.getProperty("html")	el.innerHTML
el.getProperty("text")	el.innerText 或 el.textContent，取决于浏览器
el.getProperty("class")	el.className
el.getProperty("for")	el.htmlFor，主要用于<label>元素

这些属性中还有一个子集，它们被当作 Boolean 值处理(根据某个值是否存在)。它们在 Element.Attributes.Bools 下定义，因此可以产生一个类似下面的列表：

```
$log( $H(Element.Attributes.Bools).getKeys() );
["compact", "nowrap", "ismap", "declare", "noshade", "checked", "disabled", "readonly",
 "multiple", "selected", "noresize", "defer"]
```

这就说明了为什么在上面给出的代码示例中，当获取 readonly 属性的值时得到的是一个 true 值，而不是标记提供的字面值“readonly”属性值。

最后，如果调用.getProperty()方法所使用的属性名称未能匹配之前描述的任何一种映射，那么将后退以调用该元素的.getAttribute()方法。在演示.getProperty()方法的代码示例中，当获取“type”属性时就会遇到这种情况。

2. 一次性获取多个属性

除了提供.getProperty()方法之外，MooTools 还提供了一个名为.getProperties()的伴随方法，它可用来在一次调用中获取多个属性：

```
$log_json( $$('label').getProperties('for', 'text') );
// [{"for":"alpha","text":"Alpha"},{"for":"beta","text":"Beta"},
// {"for":"gamma","text":"Gamma"},{"for":"delta","text":"Delta"}]
```

在这个示例中，我们在一个元素列表上调用.getProperties()方法，因此一次性获取了一组元素的多个属性。返回值是一个散列列表，每个散列提供了列表中每个元素的请求属性名及其对应的值。

3. 设置属性值

与.getProperty()方法互为补充的是 MooTools 附加到元素上的.setProperty()方法，它可用来设置属性值，如下所示：

```
$('gamma').setProperty('value', 'Another value');

$('gamma').setProperty('disabled', true);
```

注意，.getProperty()方法所使用的属性和属性映射也都适用于该方法：MooTools 属性 html 和 text 相应地映射，而像 disabled 这样的 Boolean 属性则会在分配给元素属性之前转换成 Boolean 值。

4. 移除属性值

可以通过将属性的值设置为 null、false 或一个空字符串来实际地移除它。但更简单的方式是使用.removeProperty()方法，如下所示：

```
// Equivalent to .setProperty('checked', false)
$('delta').removeProperty('checked');
```

与.getProperty()和.setProperty()方法类似，所有属性和属性映射都会被.removeProperty()方法接受。

5. 从表单中构建查询字符串

由于上面示例中的标记是一个表单，因此这里还可以演示.toQueryString()方法：

```
$('beta').setProperty('checked', true);

$log( $('form1').toQueryString() );
// alpha=Some%20value&beta=on
```

首先确保选中一个复选框以进行测试，然后调用.toQueryString()方法，该方法将根据表单的内容生成一个查询字符串。这个方法有一个非常方便的用途，就是将表单提交操作转换成一个较轻量级的 AJAX 请求，我们将在第 33 章中更加详细地讨论这一点。

32.2.4 操作扩展的元素属性

MooTools 提供的.getProperty()、.setProperty()和.removeProperty()方法将映射到 DOM 对象属性和 DOM 节点属性的属性合并成一个名称空间。

但是，在这些方法以及其他方法的基础之上，MooTools 通过似乎非常简单的.get()、.set()和.erase()方法提供了一种强大且更为抽象的属性访问系统。

1. 获取扩展属性值

如果不能映射到已经定义的某个 DOM 对象属性，那么.getProperty()方法将后退以使用原生方法.getAttribute()。与这种思想类似，当没有为给定名称定义扩展属性时，.get()方法将回退以使用.getProperty()方法。

因此在大多数情况下，.get()方法可用来替代.getProperty()方法。但是，它们的关键差别在于.get()方法依靠的是一种允许创建新的属性处理程序的可插入扩展方案。它使用这些处理程序，而不是调用.getProperty()方法，因此在某些情况下可能仍然需要调用.getProperty()来获得原始属性。如果上述表达看起来有点混乱，那么查看这个系统的实际运行情况可能更有意义。

.get()方法本身的用法非常简单，如下面的代码示例所示：

```
$log( $$('li').get('class') );
// ["foo", "bar", "sublist", "", "", "", "xyzzy",
// "quux", "", "", "", ""]

$log( $$('ol>li').get('text') );
// ["Alpha", "Beta", "Gamma"]

$log( $$('li>*').get('tag') );
// ["span", "ol", "span", "label", "input", "label",
// "input", "label", "input", "label", "input"]
```

在上面的代码中，最开始的两个.get()调用均将调用.getProperty()方法，它们分别用来获取元素列表的 class 和 text 属性。但第三个.get()调用使用名为 tag 的扩展属性处理程序来获取列表元素的标记名称。

虽然我们很快就会了解如何定义自己的扩展属性处理程序，但是有必要首先了解 MooTools 模块提供了什么扩展属性。表 32-5 列出了属性以及提供这些属性的相关模块，通过将该模块包含在内部版本中就可以使用相应的扩展属性。

表　32-5

属　　性	提供该属性的模块	作　　用
style	Element	获取和设置 CSS 样式
tag	Element	获取元素的标记名称
href	Element	从元素那里获取相对 URL
html	Element	接受一个字符串列表，将它们连接起来作为元素的 HTML
styles	Element.Style	映射到.setStyles()和.getStyles()调用
opacity	Element.Style	映射到.setOpacity()和.getOpacity()调用
events	Element.Event	映射到.addEvents()调用
send	Request	有关 AJAX 处理的更多内容，请参见第 33 章
load	Request.HTML	有关 AJAX 处理的更多内容，请参见第 33 章
tween	Fx.Tween	有关动画的更多内容，请参见第 34 章
morph	Fx.Morph	有关动画的更多内容，请参见第 34 章
slide	Fx.Slide	有关动画的更多内容，请参见第 34 章

为了强调.getProperty()和.get()之间的差别，请查看如下的标记：

```
<div id="div5">
    <a href="http://decafbad.com/">decafbad</a>
    <a href="/~lmorchard/prof_js/ch30/dom.html">this page</a>
</div>
```

注意，在表 32-5 中列出了一个扩展的 href 属性处理程序。这里使用的是这个处理程序而不是.getProperty()方法，因此下面的代码演示这些方法产生的结果稍有不同：

```
$log( $$('a').getProperty('href') );
// ["http://decafbad.com/", "http://localhost/~lmorchard/prof_js/ch30/dom.html"]

$log( $$('a').get('href') );
// ["http://decafbad.com/", "/~lmorchard/prof_js/ch30/dom.html"]
```

标记中的第二个链接的声明使用了相对 URL，但.getProperty()方法返回的值是绝对 URL，这是浏览器处理 href 属性的行为造成的结果。但当使用.get()方法时，这个扩展处理程序自动地将当前页面的 URL 从 href 值中分离，因此将其转换回相对值。

2. 设置扩展属性值

.set()方法是.get()方法的直接补充，它遵循前面描述的所有相同的属性映射特征。可以使用该方法设置单个属性，如下所示：

```
$$('#div2 .bar').set('class', 'grayed');
```

3. 一次性设置多个扩展属性

但是，.set()方法的真正有用之处在于一次性设置多个属性，特别是用在元素列表中时，如下所示：

```
$$('ol>li').set({

    // This falls back to .setProperty()
    text:  $time() + '-' + $random(0,1000),
    class: 'bordered',

    // Uses Element.Properties.html.set()
    html: '<b>Hello there</b>',

    // Uses setOpacity from Element.Style
    opacity: 0.75,

    // Uses setStyles from Element.Style
    styles: {
        lineHeight: '1.5em',
        textAlign: 'center'
    }

});
```

从某种意义上来讲，这种.set()用法是该扩展属性处理程序系统的结果。通过指定一个简单的指定属性和值的散列，可以使用最少的声明式代码来完成复杂的处理程序安排。

上面代码中的简单.set()调用将导致对.setProperty()、Element.Properties.html.set()、.setOpacity()和.setStyles()这几个方法的调用，所有操作都是通过一个统一的、一致的、从这些复杂性和个别调用中抽象出来的接口完成的。

在其他地方也会嵌入.set()方法调用，例如在 Element 构造函数中以及在 document.newElement()方法中，从而在操作元素的过程中进一步利用.set()方法提供的简洁性。

4. 删除扩展属性值

最后一个(但并非最不重要)的方法是.erase()方法。这个方法的操作方式与.removeProperty()方法类似，用于将某个属性删除，如下所示：

```
document.getElement('ol>li:first-child').erase('style');
```

但是，.erase()与.removeProperty()方法之间的主要差别仍然在于支持.get()、.set()和.erase()方法的扩展系统。

5. 创建自定义属性处理程序

我们已经了解.get()、.set()和.erase()方法如何使用扩展属性处理程序，下面查看如何定义自己的属性处理程序。

MooTools 定义了一个名为 Element.Properties 的散列集合，它的键指定了扩展属性。每个键的值也是一个散列，它可以包含名为 get、set 和 erase 的键(每一个键都是可选的)。所有这些键又分别指向.get()、.set()和.erase()方法所使用的最终完成任务的函数。

下面就是一个自定义属性处理程序的示例：

```
Element.Properties.color = {

    get: function() {
        $log("get color called");
        return this.style.color;
    },

    set: function(clr) {
        $log("set color called");
        this.style.color = clr;
    },

    erase: function() {
        $log("erase color called");
        this.style.color = '';
    }

};
```

上面的代码定义了一个名为“color”的新的自定义扩展属性，它支持所有 3 个扩展属性方法。所有这些方法都应该在待操作元素的上下文中调用。还要记住的是，每个方法都是可选的：如果

没有定义这些方法，就会回退以调用相应的.getProperty()、.setProperty()和.removeProperty()方法。

为了解这个新的属性处理程序的实际运行情况，请查看下面的示例代码：

```
var item = document.getElement('li');

item.set('color', 'green');
// set color called

$log( item.get('color') );
// get color called
// green

item.erase.delay(2000, item, ['color']);
// erase color called
```

由于在自定义处理程序中包含了日志记录语句，因此可以在日志文本消息中看到何时调用这些方法。

由于能够在 MooTools 中定义自己的属性处理程序，这就为我们扩展该工具集的核心便利工具提供了一种强大的机制。尽管 MooTools 提供了操作通常所需的所有属性的方式，但这个系统可以让我们实现对任何发现需要处理的不寻常方面的支持。

32.2.5 使用元素存储机制安全地管理元数据

对于不寻常方面的处理，MooTools 还提供了一个名为“*元素存储*(element storage)”的系统，把任意元数据和对象与元素关联起来。这个系统可能看起来显得多余，因为 DOM 元素能够在任意的可扩展对象(expando)属性上设置值，从而可以让我们将任何数据存储到元素上。

但使用 DOM 可扩展对象属性的问题在于，它们容易引起内存泄漏和其他问题。例如，如果将一个元素的引用存储到一个对象中，然后将这个对象的引用存储到该元素上，那么就创建了一个循环引用，当用户从页面上离开时，垃圾回收机制并不总是能很好地处理这种情况；换句话说就是，这造成了内存泄漏问题。然后，还会存在与 DOM 对象的现有属性产生名称空间冲突的问题。

MooTools 寻求通过将自身约束到单个可扩展对象属性(名为 uid)来解决这些问题。当 MooTools 的方法第一次遇到某个元素时，就会为该属性分配一个唯一标识符，并将其用作全局散列的键。然后，这个全局散列可以包含任意数据。当卸载页面时，MooTools 将自己的全局存储散列清空，从而中断了循环引用并防止了内存泄漏。

因此，在讲解了幕后工作之后，演示这个元素存储工具的用法就会相当简单：

```
$('list1').store('database_id', '12345');

$log( $('list1').retrieve('database_id') );
// 12345
```

每个元素都被赋予.store()和.retrieve()方法，分别用来在全局存储散列中存储和获取数据项。上面的代码使用键 database_id，但我们可以使用任何字符串作为键。

虽然正常情况下不需要做这件事情，但通过检查分配给元素的唯一 ID 可以进一步了解该机

制，如下所示：

```
var ele_uid = $('list1').uid;

$log( ele_uid );
// 3

$log_json( Element.Storage.get(ele_uid) );
// {"database_id":"12345"}
```

每个元素都被赋予一个.eliminate()方法，用来清除某个键的数据，其用法如下：

```
$('list1').eliminate('database_id');

$log( $('list1').retrieve('database_id') );
// null

$log_json( Element.Storage.get(ele_uid) );
// {}
```

MooTools 在其内部实现中大量使用了这个系统，因此我们也可以从中受益。

实际上，与其直接使用.store()和.retrieve()方法，不如创建新的扩展属性处理程序，将特殊用途的元数据需求整合到核心属性处理系统中。不管采用哪种方式，这种元素存储系统均会使我们更加方便地将信息和 DOM 对象绑定到元素，同时避免了常见的浏览器问题和内存泄漏问题。

32.3　修改 DOM 结构

到目前为止，本章所演示的是如何使用 MooTools 查找元素，以及一旦找到元素之后如何操作它们的属性。现在我们研究 MooTools 如何帮助创建元素自身以及修改 DOM 结构。

为了进行演示，请考虑下面这个简短的标记示例：

```
<div id="div4">
    <h2>Another list</h2>
    <ul>
        <li><span>First list item</span></li>
    </ul>
</div>
```

32.3.1　创建新元素

上面的列表几乎为空，因为我们首先要研究的操作之一就是如何使用 MooTools 创建新元素。在 MooTools 下创建新元素的一种方式就是利用 document.newElement()方法，如下所示：

```
var item1 = document.newElement('li', {
    class: 'bordered',
    text: 'Second list item'
});
```

document.newElement()方法的第一个参数是要创建的元素的名称，后跟一个可选的散列。如果提供该参数，那么在创建这个新元素之后，这个散列将会用于调用该元素的.set()方法，从而增强了前面描述的.set()方法的重要性和功能。

在内部使用.set()意味着在创建元素之后立即就可以执行多项工作，其范围从设置该元素的文本和 HTML 内容到附加事件以及设置 CSS 类和样式。当然，可以忽略第二个参数，而是通过其他方式来操作这个新元素，但这种方式提供了一种优雅的“一步到位的”创建和改进步骤。

除了 document.newElement()方法之外，还可以通过创建 MooTools 的 Element 类的一个实例来创建新元素：

```
var item2 = new Element('li', {
    style: 'font-weight: italic',
    html: '<a href="http://decafbad.com">A link</a>'
});
```

Element 构造函数的参数基本上与 document.newElement()方法类似，也包括将传给.set()内部使用的可选散列参数。

这两种方式各有其优点，但 Element 构造函数除了接受标记名称之外还可以接受一个原生 DOM 元素作为它的第一个参数。这样，新的 Element 实例就可以将现有的原生 DOM 元素包装起来，而 document.newElement()却不能这样使用。

32.3.2 复制元素

一旦创建一个元素，那么复制它几乎总是获取更多相同元素(特别是如果需要大量的元素)的最快捷方式。MooTools 简化了复制操作：

```
var item3 = item2.clone();
```

因此，如果需要构建一个列表或者其他某种带有大量相似元素的结构，那么一种较好的做法是利用 document.createElement()方法或者一个新的 Element 实例来创建一个初始的模型元素，然后从这个初始的模型元素创建大量副本。

.clone()方法接受两个可选的 Boolean 参数：

- 如果第一个参数为 false，则会阻止子元素的递归式复制。否则，在默认情况下新的副本将包含它的原型的所有内容，并且都是最新复制的内容。
- 如果第二个参数为 true，则会导致副本保留原型的 id 属性。否则，这个属性在默认情况下将在副本中被丢弃，因此有助于保持原型的 id 的唯一性。

对于.clone()方法，最后一个需要注意的警告是在元素存储中与原型关联的数据不会复制到新元素中。所有的原生属性都会被复制，除了唯一的元素存储 ID。因此，新复制的元素将收到一个新的元素存储 ID，但它不会获得原型的元素存储数据副本。

这就给出了一些有趣的暗示，因为 MooTools 使用元素存储机制附加事件以及执行其他几项常见的任务。因此，我们只能期望基本的 DOM 数据(例如属性和子元素)会复制进副本元素中。

这项限制实际上让我们因祸得福，因为事件以及其他与元素相关的对象的构造工作通常比较复杂，而且会引用其他的元素，这就使得重新构造要比试图复制更加容易处理。

32.3.3　获取元素

接下来要做的工作就是将这个新元素放进页面的 DOM 结构中。一种方式就是使用已经位于 DOM 中的元素的.grab()方法：

```
var list = $$('#div4>ul')[0];
list.grab(item1, 'inside'); // see also: .grab{Inside,Before,After,Bottom,Top}
```

MooTools 为元素提供了一个名为.grab()的方法，正如它的名称所暗示的那样，它获取一个给定的元素并将其放入可选的第二个参数指定的某个相对位置：

- before　前一个兄弟元素
- after　下一个兄弟元素
- top　第一个子元素
- bottom——最后一个子元素
- inside——最后一个子元素(默认值)，bottom 的别名

还有几个分别针对这些位置的指定函数，这样就不需要向更为通用的.grab()方法提供指定的位置：

- .grabBefore(el)
- .grabAfter(el)
- .grabTop(el)
- .grabBottom(el)
- .grabInside(el)

当创建延迟函数调用或者遇到其他传递参数可能比较困难的不常见场合时，这些特定位置函数用起来要更加简单。

32.3.4　注入元素

.grab()的逆向操作是.inject()方法：可以将一个元素注入到相对于 DOM 中另一个元素的位置中。

```
item2.inject(list, 'inside'); // see also: .inject{Before,After,Bottom,Top}
```

与.grab()方法类似，.inject()的第一个参数是一个目标元素，并且可接受一个可选的字符串参数来声明所需的相对位置。此外，还有下列特定位置的.inject()版本：

- .injectBefore(el)
- .injectAfter(el)
- .injectTop(el)
- .injectBottom(el)
- .injectInside(el)

.grab()和.inject()以及它们各自的相关方法组合在一起，构成了原生 DOM 提供的.appendChild()和.insertBefore()方法的替代方法。使用这些原生方法完成同样的元素放置工作则会涉及很多行的元素定位与原生调用代码。

32.3.5 创建并附加文本节点

尽管可以使用.set()方法来操作元素的文本内容，但我们还可以利用 document.newTextNode()方法将文本作为 DOM 节点进行处理，如下所示：

```
var txt1 = document.newTextNode(' - some random text');
item2.grab(txt1);
```

可以使用.appendText()方法将这两个步骤合并到一个步骤中：

```
item3.appendText('Prepended text - ', 'top');
```

注意，与.grab()方法类似，.appendText()方法接受可选的第二个参数来声明一个相对位置，新创建的文本节点将放到这个位置上；在这里，我们将其作为父元素的第一个子元素。

32.3.6 替换和包装元素

除了.grab()和.inject()之外，MooTools 还提供了几个更有趣的结构化工具。利用到目前为止所演示的工具，我们可以向页面中插入新内容，但.replaces()方法提供了把一个元素替换成另一个元素的途径：

```
var new_title = new Element('span', {
    text: 'New sublist'
});
new_title.replaces(span1);
```

.replaces()方法的参数就是将要删除并替换成上下文元素的元素。这个方法的返回值是该上下文元素，因此可以进行链式调用。

.wraps()方法基于.replaces()方法，可用来把一个元素放入一个新的父元素中，并将这个新的父元素插入到该元素所在的位置上，从而为一个元素创建一个新的就地包装器：

```
var span1 = $$('#div4>ul>li>span')[0];

var strong = new Element('strong');

strong.wraps(span1, 'top');
```

.wraps()方法的第一个参数应该是一个元素。调用.wraps()时所在的上下文元素立即成为替换元素并成为这个元素的新的父元素。在这个代码示例中可以看到，可以使用这个方法来完成诸如将文本节点或其他元素包装到一个新的格式化元素或其他容器中这样的操作。

32.3.7 接纳元素

.grab()方法非常适合于逐个地将新元素放进父节点中，但是.adopt()方法提供了一个更加容易的方式来一次性添加多个子节点：

```
var new_list = new Element('ul', {
```

```
    html: '<li>First item</li>'
});

$$('#div4>ul>li')[0].grabBottom(new_list);

var new_items = [ 1, 2, 3, 4 ].map(function(num) {
    return new Element('li', { text: 'Item ' + num });
});

var extra_item = new Element('li', {
    text: 'And one more item!'
});

new_list.adopt(new_items, extra_item);
```

在上面的代码中，我们创建了一个新的列表并将其作为子列表插入。然后，在一组数字上调用.map()方法，在一个列表中创建多个新的列表项。也使用一个变量创建一个元素。这些都将作为参数传入新的父列表的.adopt()方法。

这里比较有趣的事情是.adopt()方法接受多个参数，它们均可以是单个元素或元素列表。这些参数均会被展平为一个单一列表，然后作为子节点添加到上下文元素中。这为我们构建元素列表或元素的单个实例提供了极大的灵活性，从而在一个单步操作中就可以将全部元素插入到页面中。

32.3.8　销毁和清空元素

最后两个待描述的 DOM 操作方法是.destroy()和.empty()方法，它们分别用来删除元素和删除元素的所有子元素。

.destroy()方法的用法如下所示：

```
new_items[1].destroy();
```

这个方法将上下文元素从 DOM 中删除，并将 MooTools 所知道的所有相关资源(包括子元素以及附加的事件和函数)清除。由于存在这种额外的清理，因此更好的做法是使用.destroy()方法，而不是在父节点上调用原生 DOM 函数.removeChild()。

另一方面，如果想做的工作只是清空元素的内容，那么可以调用.empty()方法：

```
new_list.empty();
```

这个方法基本上是将.destroy()方法应用到元素所包含的每个子节点上。注意，.destroy()方法又会在内部将.empty()方法实施到子节点上进行递归式清除。

32.4　附加监听程序并处理事件

虽然调整元素和操作 DOM 结构非常有趣，但这些操作只有在服务于构建响应灵敏的用户界面的目标时才真正有用。这就是 MooTools 的 DOM 事件处理便利方法发挥作用的地方，这些方法缓和浏览器差异并简化向元素连接事件监听程序的整个过程。

32.4.1 响应页面加载和卸载事件

大多数基于浏览器的应用程序都是从加载页面开始的。由于浏览器支持将处理程序附加到页面加载和卸载事件，因此这些事件分别是启动初始化以及以后进行清理的最便利场合。

下面演示了在 MooTools 中如何注册处理程序来处理页面加载和卸载事件：

```
window.addEvent('load', function() {
    $log("** window load event fired");
    $log(this == window); // true
});

window.addEvent('unload', function() {
    alert("** window unload event fired");
    alert(this == window); // true
});
```

.addEvent()方法的第一个参数是事件的名称，第二个参数是将要作为处理程序调用的函数。事件名称“load”和“unload”分别用来标识页面加载和卸载事件。对于这两个事件，在调用处理程序函数时没有带有任何参数，它将在窗口对象自身的上下文中执行。

可以任意多次调用.addEvent()方法来添加任意多的处理程序。MooTools 负责在元素存储中维护内部列表，并使用它自己的真正的事件监听程序来调用它们。

响应 DOM 准备就绪事件

比简单页面加载和卸载事件更高级的事件是 DOM 准备就绪事件。页面加载事件本身只会在所有的图像以及内嵌对象全部加载之后才会引发，但在图像完成加载之前用户将看到页面已经出现并有机会与之交互。

这里的问题是在这个时间点之前 JavaScript 代码没有机会连接用户界面元素并执行其他初始化和页面调整工作。此外，根据页面上的图像以及嵌入的其他素材的数量不同，该操作可能会在一段相当长的延迟之后才会发生。因此，用户将有可能体验一个“破碎的”界面，因为该界面尚未完成初始化。

但在页面加载过程中，在某一个时间点上页面 DOM 结构已经完成加载，而且准备好由 JavaScript 代码进行操作，虽然此时还有图像和其他对象仍然在加载之中。在不同的浏览器中，检测页面加载过程中的这种里程碑事件并执行相应操作的方式各有不同，但在大多数现代浏览器中均有相应的方式来进行检测。

MooTools 将这种操作作为一个自定义事件提供了跨浏览器实现，在最坏的情况下，它就像一个窗口加载事件；而在最好的情况下，它会在正确的时间点捕获 DOM 事件。

这个概念讲解起来比较困难，但使用起来却非常容易，因为 MooTools 让其得以简化，就像注册一个窗口加载事件处理程序一样：

```
// Requires Domready module
window.addEvent('domready', function() {
    $log("** window domready event fired");
});
```

这里有一个警告，就是必须将 Domready 模块包含进来作为 MooTools 内部版本的一部分。如

果希望更多一般性地了解 DOM 准备就绪概念，那么可以阅读下面这篇由 Dean Edwards 撰写的博客文章“The window.onload Problem - Solved!”：

http://dean.edwards.name/weblog/2005/09/busted/

32.4.2　添加和删除事件处理程序

通过处理窗口加载、卸载以及 DOM 准备就绪事件，我们已经初步尝试 MooTools 简化的事件处理风格，但还有更多的内容需要研究。为演示 MooTools 的事件工具做好准备，请考虑下面的标记：

```
<style type="text/css">
    #ex1 .clickme {
        padding: 0.5em; margin: 0.5em;
        display: block; background-color: #ddf;
        border: 2px dashed #fff;
        color: #000;
    }
    #ex1 .clickcolor {
        background-color: #dfd;
        border: 2px dashed #000;
    }
</style>
<div id="ex1">
    <h2>Add / Remove events</h2>

    <a id="link1" class="clickme color0" href="#">Click me!</a>
    <a id="link2" class="clickme color0" href="#">Click me!</a>
    <a id="link3" class="clickme color0" href="#">Click me!</a>
    <a id="link4" class="clickme color0" href="#">Click me!</a>
    <a id="link5" class="clickme color0" href="#">Click me!</a>
</div>
```

上面的示例将各种处理程序附加到代码中定义的链接，并通过切换 CSS 类名和样式提供一些反馈。

1. 添加事件处理程序

MooTools 的事件处理工具非常简单，这是一件好事情，因为它们涵盖了大量的跨浏览器不一致性问题和怪异行为。下面演示了如何将一个快速的鼠标单击事件处理程序连接到一个链接元素：

```
var first_handler = function(ev) {

    ev.stopPropagation();
    ev.preventDefault();

    var el = ev.target;
    el.toggleClass('clickcolor');
```

```
    $log('Click handler #1');
    $log( this == ev.target ); // true

};

$('link1').addEvent('click', first_handler);
```

上面的代码演示了 MooTools 的 DOM 事件处理的许多主要功能：

- 事件处理程序函数的第一个参数是一个 Event 对象，这个自定义 MooTools 包装器将跨浏览器事件对象的差异规范化。
- Event 对象提供了.stopPropagation()和.preventDefault()方法，这些方法分别用来取消事件冒泡和事件的默认行为。
- 在 Event 对象的可用属性中，可以使用 target 属性直接访问事件中涉及的元素。
- 事件处理程序函数的上下文经过跨浏览器规范化以反映注册该事件的元素。
- 可以使用给定元素的.addEvent()方法来注册事件处理程序，该方法的第一个参数指定了所需的事件，而第二个参数提供了处理程序函数。

下面的代码提供了一个更加简洁而概括的示例：

```
$$('.clickme').addEvent('click', function(ev) {

    ev.stop().target.toggleClass('clickcolor');

    $log('Click handler #2');
    $log( this == ev.target ); // true

});
```

前一次.addEvent()调用是在单个元素的上下文中执行的，而这一次调用是在一个元素列表上执行的，因此将该事件处理程序附加到列表中的每个元素。

此外，通过使用快捷方法.stop()可以将.stopPropagation()和.preventDefault()这两个方法调用合并起来。由于这个方法返回事件对象本身，因此这个调用可以链接起来以访问目标元素，而这又允许调用.toggleClass()方法。这只是一个小型的演示，但它说明了 MooTools 提供的简洁性和强大功能。

2. 一次性添加多个事件处理程序

相比于向一组元素添加一个处理程序，更加强大的方式是将一组处理程序添加到一组元素上，就像下面这样：

```
$$('.clickme').addEvents({
    mouseover: function(ev) {
        $log("Mouseover " + ev.target.id);
    },
    mouseout: function(ev) {
        $log("Mouseout " + ev.target.id);
    }
});
```

addEvent()方法只接受一个事件名称和事件处理程序，而.addEvents()方法接受一个指定处理程序的散列，可以在单个元素或元素列表的上下文中调用这些处理程序。这样就可以一次性将复杂的处理程序映射注册到各种元素中。

此外，如果回顾元素上的.set()方法支持的扩展属性处理程序“events”，那么可以认为下面的代码是可行的：

```
$$('.clickme').set({

    html: '<strong>No, really click me!</strong>',

    styles: {
        color: 'green',
        border: '1px solid #ddd'
    },

    events: {
        mouseover: function(ev) {
            $log("Mouseover " + ev.target.id);
        },
        mouseout: function(ev) {
            $log("Mouseout " + ev.target.id);
        }
    }

});
```

通过使用.set()方法，我们可以在针对页面上一个或多个元素进行属性、样式和内容调整的同一个调用中注册一组事件处理程序。

此外，记住.set()方法可用于构造新元素，因此甚至可以在创建元素时注册事件。.set()方法是 MooTools 工具集中功能最强大的方面，这种说法一点也不夸张。

3. 添加对象方法作为事件处理程序

到目前为止，所有附加作为事件监听程序的函数都是匿名函数。但是，通常我们需要将对象的方法注册为事件监听程序。这里的问题是，MooTools 会自动纠正处理程序函数的上下文以反映出注册该事件的元素。

出于这个原因，在函数上提供了.bindWithEvent()方法来帮助保持正确的对象上下文，确保传入 MooTools 事件包装器，甚至允许传入可选参数，就像函数上的.bind()方法一样。下面的代码演示了.bindWithEvent()方法的用法：

```
var obj = {
    handler: function(ev, optional) {
        ev.stop();

        $log('Object method handler');
        $log( this == obj ); // true

        if (optional)
```

```
            $log("Optional parameter: " + optional);
        }
    };

    $('link3').addEvent('click',
        obj.handler.bindWithEvent(obj, "hello there"));
```

在运行上面的代码之后，在标记示例中的第三个链接上单击，这样就会在日志文本消息中产生如下的消息：

```
Object method handler
true
Optional parameter: hello there
```

this 的值已经绑定到该方法所属的对象，而且已经伴随 Event 对象传递可选参数。

4. 复制事件处理程序

一旦将一组事件分配给一个元素，那么通常更快且更便利的方式是只把这种安排复制到其他的元素。MooTools 的.cloneEvents()方法为此提供了便利，如下所示：

```
$('link2').cloneEvents($('link1'), 'click');
```

在运行这段代码之后，所有附加到第一个链接上的处理程序均将被复制并附加到第二个链接上。

还有一点需要记住，由于用来复制元素的.clone()方法并没有将事件复制过来，因此在该方法后面紧接着调用.cloneEvents()方法将有助于弥补这个缺陷。

5. 删除事件处理程序

最后，有时候需要在某个特定时间点之后将事件监听程序删除。该操作有两种完成方式，第一种方式是使用.removeEvent()方法将特定事件类型的特定处理程序移除，而第二种方式是使用.removeEvents()将特定事件类型的所有处理程序移除。这些方法的用法如下所示：

```
// Note that a reference to the function is needed
$('link1').removeEvent('click', first_handler);

// All click events removed, no references needed.
$$('.clickme').removeEvents('click');
```

因此，如果已经保留某个特定处理程序的引用(例如 first_handler 变量)，那么可以使用.removeEvent()方法选择性地移除一个处理程序；注意 MooTools 并没有提供类似.getEvents()这样的方法来获取附加的事件处理程序，因此保留处理程序的引用是关键所在。

在调用.removeEvent()方法之后，调用.removeEvents()方法将元素列表的所有 click 处理程序移除。与其他事件处理方法一样，均可以针对单个元素和整个元素列表来调用.removeEvent()和.removeEvents()方法。

6. 创建自定义事件类型

在更加深入地研究 Event 对象的领域之前，MooTools 事件处理机制还提供了另一种技巧，它使得我们能够有条件地根据现有的原生 DOM 事件来定义自定义事件。最好通过代码来讲解这项功能，因此查看下面的标记：

```
<style type="text/css">
    #shift_listener {
        padding: 0.5em; margin: 0.5em;
        display: block; background-color: #ddf;
        border: 2px dashed #fff;
        color: #000;
    }
    #ex3 .clickcolor {
        background-color: #dfd;
        border: 2px dashed #000;
    }
</style>

<div id="ex3">
    <h2>Custom events</h2>
    <div id="shift_listener">Shift-click on me!</div>
</div>
```

现在查看连接事件监听程序的代码：

```
Element.Events.shiftclick = {

    base: 'click',

    condition: function(ev) {
        return ev.shift;
    }

};

$('shift_listener').addEvent('shiftclick', function(ev) {

    ev.stop().target.toggleClass('clickcolor');

    $log(ev.target.id + " was shift clicked!");

});
```

需要注意的第一件事情是，MooTools 维护了一个名为 Element.Events 的散列，它可用来定义新的指定事件类型。这些事件类型本身也定义为散列，它里面包含键 base 和 condition 的值。第一个键 base 声明了该事件所基于的真正的 DOM 事件，而第二个键 condition 定义了一个函数，该函数将检查类型和事件条件来确定是否引发自定义事件。

为自定义事件类型注册处理程序实际上最终会为声明的 base 类型注册一个处理程序，但只有

当 condition 函数返回 true 时才会引发该事件。因此，在上面的代码中，我们创建了一个自定义事件类型 shiftclick，它的注册处理程序只有在按下 Shift 键并发生鼠标单击事件时才会被调用。

MooTools 内部使用了这种机制来提供跨浏览器规范化的 mousewheel 事件以及自定义的 mouseleave 和 mouseenter 事件，后两个事件不会将经过或离开子元素视为离开父元素。

32.4.3 检查事件包装器对象

到目前为止，MooTools 事件处理程序系统中尚未经过深入讨论的最后一部分是传给所有事件处理程序的 Event 对象包装器。但是，这方面的 MooTools 功能最好还是通过代码演示来进行说明。

因此，本章的最后一部分将专门构建一个演示应用程序，图 32-3 给出了它的预览。这个应用程序的构造过程将有助于讲解 Event 对象提供的功能，并有助于在您看到代码的运行情况之后演示它所提供的功能。

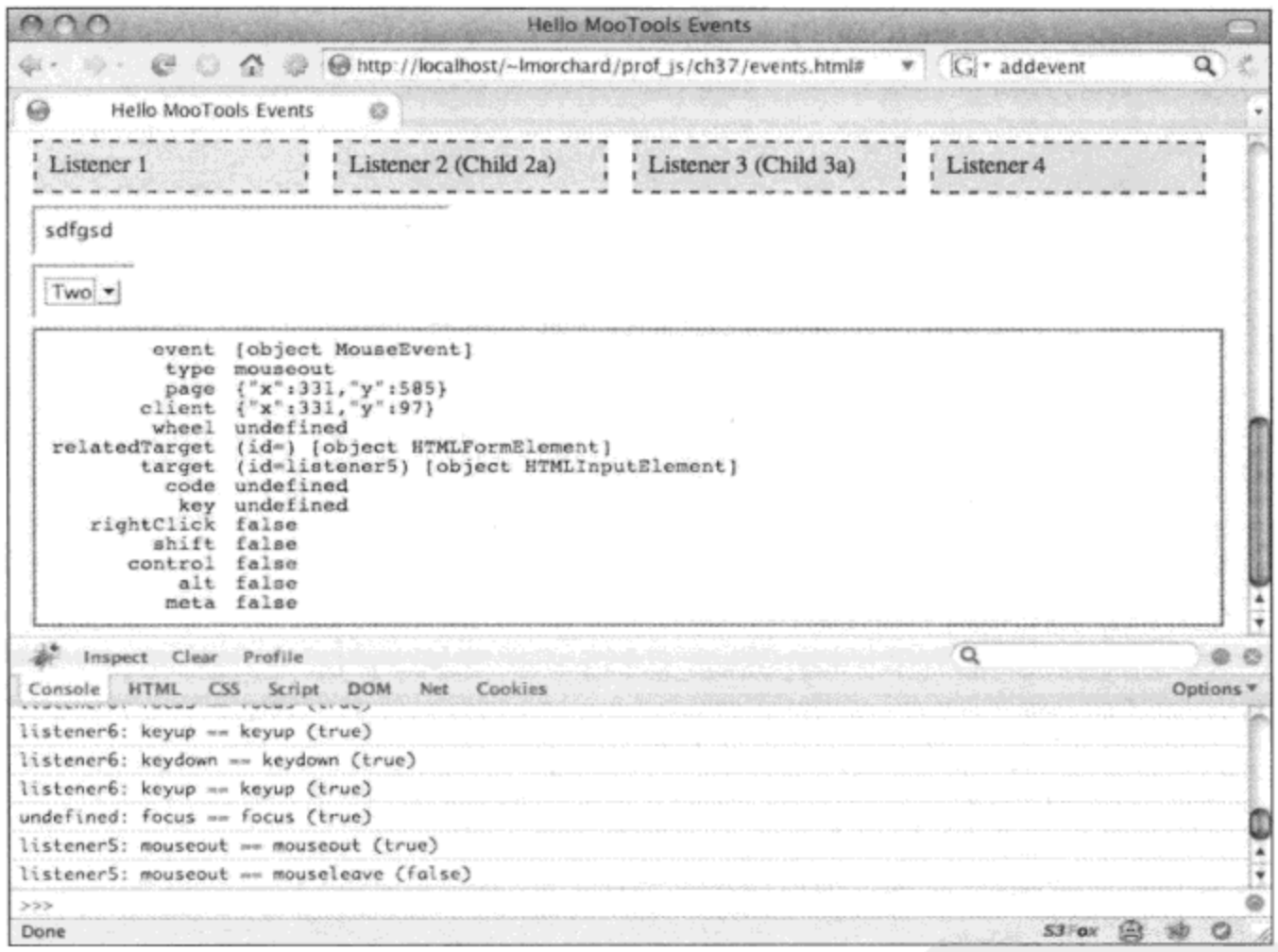


图 32-3

1. 构造应用程序页面

首先给出下面这段标记，它提供了示例元素和从中进行演示的框架：

```
<div id="ex2">
    <h2>Event wrapper</h2>

    <div id="listener1" class="listener">Listener 1</div>
    <div id="listener2" class="listener">
        Listener 2
        <span id="listener2a">(Child 2a)</span>
```

```
    </div>
    <div id="listener3" class="listener">
        Listener 3
        <span id="listener3a">(Child 3a)</span>
    </div>
    <div id="listener4" class="listener">Listener 4</div>

    <form>
        <input type="text" id="listener5" class="listener" size="30" />
        <select id="listener6" class="listener">
            <option>One</option>
            <option>Two</option>
            <option>Three</option>
        </select>
    </form>

    <dl id="listener_status"></dl>

</div>
```

页面上最初的几个 元素以及<form>元素将为要附加的事件监听程序提供锚点；注意所有这些元素都包含 CSS 类名“listener”，我们在后面选择要附加事件处理程序的元素时将要用到它。

在这些元素之后，最后的<dl>元素将作为信息读出器，它(结合 Firebug 浏览器日志)描述了事件引发状态。

接下来，查看下面这些让该应用程序更具吸引力且更加紧凑的 CSS 样式：

```
<style type="text/css">
    #ex2 .listener {
        padding: 0.5em; margin: 0.5em; display: block;
    }
    #ex2 div.listener {
        background-color: #ddf; border: 2px dashed #333;
        color: #000; width: 20%; float: left;
    }
    #ex2 input.listener {
        clear: both;
    }
    #listener_status {
        padding: 0.5em; margin: 0.5em; border: 2px solid #666;
        font-family: monospace;
    }
    #listener_status dt {
        float: left; text-align: right;
        width: 18ex;
    }
    #listener_status dd {
        margin-left: 20ex;
    }
</style>
```

这些样式形成了图 32-3 中所示的布局、边框和阴影。当然，对于这个演示的功能来说，CSS 样式是可选的，但它应该有助于协调页面上的所有内容。

2. 准备一个事件对象检查器

在这个演示应用程序中，下一步是构建某种在事件处理程序引发时能够检查事件对象的方式。下面的数组列出了 MooTools 的 Event 对象提供的几个重要属性：

```
// Properties made available by Mootools Event wrappers
var event_props = [ .
    'event',                        // Native browser event object
    'type',                         // Type of event, normalized by MooTools
    'page', 'client',               // Mouse position in page and viewport coordinates
    'wheel',                        // Normalized to [-3 ... 3] movement speed
    'relatedTarget', 'target',      // Elements involved in the event
    'code',                         // Original which or keyCode from key event
    'key',                          // Events.Keys used for translation of key event
    'rightClick',                   // Derived by MooTools from mouse state
    'shift', 'control',             // Keyboard modifiers
    'alt', 'meta'
];
```

代码中的注释应该有助于描述每个属性的作用，但只有在我们看到实际引发的事件并检查这些不同属性的最终值时才会更加清晰地了解它们的作用。

下面的代码将这个 Event 对象属性列表转换成一组元素，并添加到应用程序标记末尾的<dl>元素中：

```
// Inject status display fields for each Event property
var event_prop_fields = $H(event_props.map(
    function(name) {

        var label =
            document.newElement('dt', { text: name });

        var field =
            document.newElement('dd', { text: '...' });
        $('listener_status').grab(label).grab(field);

        // Note that MooTools purges all element wrapper
        // references on page unload, so this should be safe.
        return field;

    }
).associate(event_props));
```

作为一个定义列表，这种动态构造的标记排列将显示 Event 属性名称(作为条目)，并且很快会将其用于显示值(作为定义)。使用前面给出的 CSS 样式，这将显示为一个两列的表。

这种.map()和.associate()方法调用的链式排列将合适的标记添加到页面中(并形成一个属性名称与插入到页面中的<dd>元素相关联的散列)。该操作可能最初有点难以理解，但它是前一章中介

绍的数组和散列方法的简洁演示。

还要注意的是，借助 MooTools 的 Element 对象处理以及在卸载页面时进行的清理，散列中的这些元素引用相对而言不易形成内存泄漏问题。

3. 构建一个通用处理程序来检查事件对象

由于我们已经拥有了一个漂亮的散列结构，用于将 Event 属性绑定到 DOM 元素以显示它们的内容，因此下面将构建一个事件处理程序，它将完成在事件发生时填充这些字段的工作。下面就是完成这项工作的代码：

```
// Create a generic event handler to report on Event
var base_handler = function(event, expected_type) {

    // Update each status display field with Event properties.
    event_prop_fields.each(function(el, name) {

        var value = event[name];
        switch ($type(value)) {

            case 'element':
                // Display the ID of elements.
                value = '(id=' + value.id + ') ' + value; break;

            case 'object':
                // Show JSON for more complex objects.
                value = JSON.encode(value); break;

            default:
                // Coerce undefined and others to string.
                value = ''+value; break;

        }

        el.set('text', value);
    });

    // Report the event with a log message.
    $log(
        event.target.id + ': ' +
        event.type + ' == ' + expected_type +
        ' (' + (event.type == expected_type) + ')'
    );

};
```

上面示例中定义的事件处理程序使用前面代码块中构造的 event_prop_fields 散列，逐个地检查接收到的 Event 对象的属性。

检查每个属性值的类型以确定如何进行显示：

- 元素由一个包含该元素 ID 的字符串表示。

- 对象转换成 JSON 表示(出于可读性考虑)。
- 所有其他值类型均简单地转换成字符串。

一旦构造好表示该值的合适字符串，就使用该值更新相关的检查器字段的文本内容。在此之后，构造一条日志消息来报告目标元素的 ID、事件类型以及处理程序期待的事件类型。

总体来说，这个处理程序的作用是不断地使用它处理的 Event 对象的内容来更新定义列表，同时在浏览器日志文本消息中产生一个针对事件流的连续性说明。

4. 创建针对特定事件类型的处理程序

我们已经创建了一个通用事件处理程序，下面构建几个专门绑定到每种预期的事件类型的处理程序。这将有助于说明自定义事件类型和原生事件类型之间的差别。还可以借此机会列出 MooTools 处理的一些已知的事件类型。

在下面的代码中，我们使用.bindWithEvent()方法将预期的事件类型名称与特定的处理程序绑定起来：

```
// A subset of event types available for listeners/
var event_types = [

    'mouseup', 'mousedown',

    'click', 'dblclick', 'contextmenu',

    'mouseover', 'mouseout', 'mousemove',

    'mouseenter', 'mouseleave', 'mousewheel', // Custom event types

    'selectstart', 'selectend',

    'keydown', 'keypress', 'keyup',

    'focus', 'blur', 'change'

];

// Build a set of event handlers associated with types.
var event_handlers = event_types.map(
    function(type) {
        return base_handler.bindWithEvent(
            event_handlers, type
        );
    }
).associate(event_types);
```

与构造 Event 属性检查器字段一样，这段代码使用.map()和.associate()方法组合将一个简单的列表转换成事件处理程序函数散列(基于事件名称列表)。

5. 附加应用程序事件处理程序

最后，利用事件将事件处理程序整体连接起来，如下所示：

```
// Finally, wire up the event handler set to listener elements.
$$('#ex2 .listener').addEvents(event_handlers);

// Stop the form from submitting.
$$('#ex2 form').addEvent('submit', function(ev) {
    ev.stop();
});
```

这里利用事件列表以及方便的指定事件处理程序散列，通过调用.addEvents()方法，所有带有 CSS 类名“listener”的元素现在都应该已经与 event_types 列表中包含的每种事件类型连接起来。

6. 使用事件对象检查器应用程序

在实现了上述所有功能之后，当与标记中提供的各种元素交互时，将导致 Event 对象属性检查器不停地得到更新，而浏览器日志将随着事件流的记录而不断滚动。

这在开始时可能产生太多的信息，因而我们可能希望选择性地注释 event_types 列表中指定的某些事件类型，以便一次只关注少数几个事件类型。但在这个页面交互过程中，我们应该能够认识到 MooTools 的 Event 对象可以提供的页面交互活动信息。

我们可以观察到鼠标移动带来的坐标变化，看到报告的键盘修改键，并检查在各种不同的事件组合中涉及哪些元素。还要注意的是，当自定义事件引发时，它们的预期类型并不匹配 Event 对象报告的实际的事件类型。

这应该能够帮助您了解在自己的应用程序中使用 MooTools 提供的 Event 对象包装器可以实现什么功能。

32.5　本章小结

本章介绍了 MooTools 引入的大量工具和抽象，这些工具和抽象用于简化页面 DOM 结构中元素和节点的操作，以及向这些元素和节点连接事件处理程序以构建响应灵敏的用户界面。

在这些工具中不断重复一些主要主题，包括简洁而强大的表达方式：能够一次性操作大量元素，也能够影响复杂的元素修改与事件处理程序连接设置。第 31 章中描述的数组、散列以及函数方法均构建在本章中描述的这些工具的基础之上，只需要几行简短的代码就可具备丰富的表达能力。

在下一章中，我们将了解 MooTools 如何简化 AJAX、JSON 请求以及其他形式的动态数据请求的处理。

第33章

简化 AJAX 以及处理动态数据

本书介绍的每一种框架都以稍有不同的方式处理 AJAX，也就是在页面加载完毕之后通过建立小巧的 Web 请求来获取新数据，使用动态数据选择性地更新页面内容和界面元素，而无须彻底地重新加载整个页面。

例如，YUI 和 Dojo 框架提供了一组全功能的、异常丰富的工具来处理不同的建立请求方式以及处理各种数据格式。而在另一方面，Prototype 提供了一种更加简单直接的接口，它关注的是简单的请求，而不是试图涵盖所有可能性。

MooTools 框架提供了介于这两种方式之间的方法：可以执行各种相同域的 Web 请求，同时只对 HTML 内容和 JSON 数据提供了一点特殊处理。诸如 CSS、图像以及额外的 JavaScript 包含文件等页面素材很容易载入页面，而扩展的元素方法使得动态更新页面元素变得非常容易。

本章内容简介：

- 操作浏览器 cookie
- 动态加载页面素材

33.1 操作浏览器 cookie

虽然 cookie 并不能与 AJAX 和 JSON 数据交换相提并论，但它提供了一种有趣的机制，既能通过请求向服务器发送数据，又能在多次请求之间保存数据，从而在客户端上维持页面间的状态。

就浏览器支持而言，cookie 是一种相对古老的机制，它可用来采用 JavaScript 设置名/值对，这些名/值对存放在特殊格式的 document.cookie 字符串中。在客户端设置的名/值对将通过一个 Cookie:报头随着下一次服务器请求发送出去。反过来，服务器可以在 Set-Cookie:报头中包含客户端的新 cookie 数据，因此客户端和服务器可以进行通信并维持持久状态。

虽然您可能比较熟悉 cookie 的工作原理，但还是可以从下面的 URL 中阅读更多有关该技术的内容：

```
http://en.wikipedia.org/wiki/HTTP_cookie
http://wp.netscape.com/newsref/std/cookie_spec.html
```

33.1.1 使用 cookie 函数

使 cookie 的处理稍微有点困难的原因是 document.cookie 中保存的特殊格式的字符串：虽然它基本上就是一组名/值对，但其数据经过类似 URL 查询字符串的编码。此外，每个 cookie 还有一些元数据，可用来指定在 Web 上的什么地方能够使用该 cookie 以及它应该在什么时候失效。

MooTools 提供了一个简单的接口来自动处理这些琐碎的、错综复杂的事情。

1. 写入 cookie

下面是使用 MooTools 写入 cookie 的最简单方式：

```
Cookie.write('demo_cookie', $time());
```

这段代码设置一个名为“demo_cookie”的 cookie，并将其值设置为当前时间。这里使用了默认的元数据选项，这意味着当浏览器关闭时该 cookie 就会失效，而且该 cookie 将提供给当前页面使用。

但是，根据需要可以为 cookie 选项指定值，这需要使用一个对象或散列作为该方法的第三个参数，如下所示：

```
Cookie.write('demo_cookie_2', 'value #2', {
    domain: 'localhost',  // domain from which cookie accessible
    path: '/',            // path under which cookie accessible
    duration: 7,          // 7 days
    secure:   false,      // whether or not HTTPS is required
    document: document    // document for which to set cookie
});
```

上面的代码相对于默认值进行了几个不同选择。cookie 的可用域为 localhost，而且由于 path 的值(/)，任何来自这个站点的根下方的页面都可以访问。此外，由于 secure 的值，不需要使用 HTTPS 和 SSL，但我们可以将这个选项设置为 true，这样就强制要求使用 https://这样的 URL 访问该 cookie。

此外，duration 的值将导致该 cookie 持续 7 天时间。尽管 cookie 的过期时间和日期的表达方式有很多种，但它们通常比较混乱，而且难以记忆。因此，MooTools 通过使用距离当前时间的天数(而这通常正是我们想要的方式)来将这种情况规范化。这样做的不利之处在于 Cookie.write()函数不接受任何其他的过期时间表达方式，因此在某些罕见的情况下，这种便利措施会变成一种障碍。

2. 删除 cookie

删除 cookie 在 MooTools 中也同样简单：

```
Cookie.dispose('demo_cookie');
```

此外还要注意，类似于 Cookie.write()，Cookie.dispose()函数的第二个参数也可以接受选项，就像下面这样：

```
Cookie.dispose('demo_cookie_2', {
    domain: 'localhost', // domain from which cookie accessible
    path: '/',           // path under which cookie accessible
```

```
    secure:   false,          // whether or not HTTPS is required
    document: document        // document for which to set cookie
});
```

注意，相对于 Cookie.write()示例，这里忽略了 duration 选项。cookie 的删除是通过如下方式完成的：将其过期时间设成过去的时间，从而导致浏览器自动将其删除。

3. 读取 cookie

读取 cookie 要比写入或删除它们更简单：

```
var the_value = Cookie.read('demo_cookie');
```

在读取 cookie 时不涉及任何选项，在写入 cookie 时设置的详细信息将根据页面环境以及过期时间来确定是否能够访问给定 cookie，因此在读取数据时只需要提供 cookie 名称即可。

4. 使用 Cookie 对象和方法

Cookie 模块提供的 write、dispose 和 read 函数实际上只是 MooTools 定义的 Cookie 对象的便利的方法。可以直接使用该对象，如下所示：

```
var ck = new Cookie('cookie_obj_demo', {
    domain: 'localhost',  // domain from which cookie accessible
    path: '/',            // path under which cookie accessible
    duration: 7,          // 7 days
    secure:   false,      // whether or not HTTPS is required
    document: document    // document for which to set cookie
});

var val_read = ck.read();
$log("*** Obj cookie value: " + val_read );

var obj_val = $time();
ck.write( obj_val );

$log("*** New obj cookie value: " + obj_val );

$log("*** Disposing of object cookie");
ck.dispose();
```

使用面向对象接口相对于使用对应函数的优点在于，我们只需要在传给对象构造函数的参数中设置一次 cookie 名称和选项即可。如果最终需要执行一些复杂的逻辑来确定是否写入以及如何写入 cookie 的值，并且不希望每次都要重复相同的选项设置，那么这种方式就非常有用。

33.1.2　使用 cookie 支持的散列

除了为 cookie 设置简单的名/值对之外，MooTools 还提供了一种有趣的 Hash 子类，可以自动地或通过手动调用方式将它的内容保存为一个 cookie。这个类的作用仅限于一个 cookie 可以接受的数据量：在各种浏览器上，每个 cookie 最多有 4KB 数据。但是，这已经为实现有用的功能提

供了充足的空间：

```
var ch1 = new Hash.Cookie('hash_cookie_1');

ch1.set('alpha', 'foo');                    // cookie changed
ch1.set('beta', [ 'bar', 'baz', 'quux' ]); // cookie changed
```

上面的代码创建了 Hash.Cookie 类的一个新实例，它接受的参数与 Cookie 对象构造函数的参数完全一样。用来初始化这个新散列的所有数据都可以从这个 cookie 中读取。在完成构造之后，可以像处理普通 MooTools 散列一样使用.set()方法存储数据，但它提供了一个新的特殊方式：在默认情况下，每次数据修改都会写入到支持该散列的 cookie 中。

注意，在存储到这个散列的值中有一个数组。使用 cookie 支持的散列与设置单个的 cookie 相比所具备的首要优点就是：在更新相关的 cookie 之前，MooTools 将散列中的所有数据都编码成 JSON。因此，可以将结构化数据存储到一个 cookie 中，而再次获取该数据时并不需要进行任何分析或解释。

为了强调实际的 cookie 值发生了什么变化，试着使用 Cookie.read()查看存储的值，如下所示：

```
$log( Cookie.read('hash_cookie_1') );
// {"alpha":"foo","beta":["bar","baz","quux"]}
```

下面的代码演示了 Hash.Cookie 类的其他几个选项：

```
var ch2 = new Hash.Cookie('hash_cookie_2', {
    autoSave: false,  // must call .save() before changing.
    duration: 2       // 2 days
});

ch2.set('alpha', 'foo');
ch2.set('beta', [ 'bar', 'baz', 'quux' ]);

ch2.save();         // cookie changed
```

前面曾经提到，Hash.Cookie 类的构造函数接受的参数与 Cookie 类的构造函数接受的参数一样，此外还增加了一个名为 autoSave 的标志。如果该标志为 false，那么在每次调用.set()时不会再写入该 cookie。相反，需要调用.save()方法来把对散列的修改反映到 cookie 中。

当计划在散列上执行多种操作时，在最终决定是否更新 cookie 之前将 autoSave 标志关闭可能会非常方便。它还有助于节省把所有修改更新到 cookie 所带来的系统开销。

33.2 动态加载页面素材

尽管可以在标记中直接包含各种 JavaScript 库的引用、层叠样式表和图像，但通常更方便的做法是在页面加载之后再载入页面资源以响应用户交互。MooTools 提供了一系列有用的函数来完成这项工作，可以让我们按需加载 JavaScript 和 JSON 源，将新的层叠样式表链接到页面中，以及预先加载任何单幅图像或图像集合。

33.2.1　加载 JavaScript 和 JSON 源

与 Dojo 工具集类似，MooTools 也提供了一种按需加载 JavaScript 模块和 JSON 数据源的方式。但与 Dojo 工具集不同的是，MooTools 并没有将其作为该框架的核心功能，也没有提供依赖项跟踪系统。MooTools 只提供了一个名为 Asset.javascript()的便利函数，它在页面头部中创建一个新的<script/>元素。

1. 利用回调加载 JSON 源

在加载新的 JavaScript 模块或 JSON 数据源时，通常需要在完成资源加载时执行某种操作。一种越来越常见的做法是，向服务器提供一个回调参数以调用客户端 JavaScript 函数并把载入的数据作为其参数，如下面的代码所示：

```
var cb_name = 'feed_cb_' + $random(1, 1000);

window[cb_name] = function(data) {
    $log('*** Loaded ' + data.length + ' bookmarks');
};

$log('*** Loading delicious bookmarks JSON feed...');
Asset.javascript(
    'http://del.icio.us/feeds/json/deusx?callback=' + cb_name
);
```

在上面的代码中，我们为回调函数创建了一个唯一的名称，这个回调函数将负责处理来自 delicious.com 的书签 JSON 源数据。然后，我们使用 MooTools 函数 Asset.javascript()并结合这个回调函数名称来获取该源。当加载该源时，它将作为实参包装到该回调的调用中，这会利用该源中包含的书签计数生成一条日志消息。

日志文本消息中将出现类似下面的消息：

```
*** Loading delicious bookmarks JSON feed...
*** Loaded 15 bookmarks
```

前面曾经提到，这种 JavaScript 素材加载背后的机制基本上就是向页面的<head/>部分添加一个新的<script/>元素。可以使用该机制加载标记中最初没有包含的 JSON 源和额外的按需加载 JavaScript 模块。

2. 利用加载检查来加载 JSON 源

但在 Asset.javascript()函数背后还有一些故事。在服务器端没有提供回调功能(由于数据本身是一个静态文件，或者服务器的 Web API 刚好不支持回调)的场合中，MooTools 能够检测 JavaScript 代码或 JSON 数据是否成功加载。考虑如下代码：

```
Asset.javascript(
    'http://del.icio.us/feeds/json/tags/deusx',
    {
        check: function() {
            return $defined(window.Delicious) &&
```

```
            $defined( Delicious.tags );
        },
        onload: function() {
            var tags = $H(Delicious.tags);
            $log('*** Loaded ' + tags.getKeys().length + ' tags');
            delete Delicious.tags;
        }
    }
);
```

在运行这段代码时应该看到类似下面的日志文本消息：

```
*** Loading delicious tags JSON feed...
*** Loaded 4989 tags
```

Asset.javascript()函数的第二个参数可以接受一个定义了属性 check 和 onload 的对象或散列。第一个属性 check 应该是一个函数，一旦预期的 JavaScript 代码已经成功加载，该函数就会返回 true。第二个属性 onload 也应该是一个函数，当检测到成功加载时就应该运行该函数。

注意，有些浏览器在<script/>标记上原生地定义了 onLoad 事件，对于这些浏览器，可以使用一个针对该原生事件的监听程序来引发 onload 函数。但对于其他的浏览器来说，只有当检测到成功加载时才能定期调用 check 函数。

33.2.2 包含额外的 CSS 样式表

下面是一个简单的样式表：

```
body { font: 12px arial; }
```

假设它所在的文件名为 ajax.css，可以使用 MooTools 加载该文件，如下所示：

```
Asset.css('ajax.css');
```

这个名为 Asset.css()的 MooTools 实用函数只是向页面的<head/>部分中添加一个新的<link/>标记，根据需要载入指定的层叠样式表。

控制样式表包含属性

除了简单地构造并插入<link/>标记之外，Asset.css()还有第二个参数，可以接受一些元素属性选项。考虑下面的示例：

```
Asset.css('ajax.css', {
    rel:   'stylesheet',
    media: 'screen',
    type:  'type/css'
});
```

上面代码中使用的选项基本上匹配默认值。但是，可以针对不同的媒介类型(也就是 print)来附加新的样式表，或者调整页面关系或内容类型属性(但修改 rel 或 type 的值将偏离这个函数的指定用途，也就是加载 CSS 素材，而不是作为一种通用的<link/>元素注入器)。

33.2.3　获取图像和图像集合

在复杂的现代 Web 应用程序中，一项常见的任务是在加载页面时预先加载图像，这些图像将用于类似图像翻转元素或动画这样的用户界面元素中。虽然可以将所需的图像嵌入到标记中，但通常更好的做法是动态地确定需要什么图像文件，然后等待它们加载完毕后再构建用户界面，在此期间可能会显示一条加载消息，告诉用户估计还需要多久应用程序才能准备就绪。

1. 获取单幅图像

为了快速演示 MooTools 如何能够让我们根据需要获取单幅图像，请考虑包含以下 元素的标记：

```
<div id="images"></div>
```

可以使用 MooTools 加载一幅图像并将其放入这个 元素中，如下所示：

```
var img1 = Asset.image(
    'http://decafbad.com/images/decafbad-title.gif',
    {
        onload: function(el) {
            $log('*** Image loaded ' + el.src );
            $('images').grab(el);
        }
    }
);
$log( img1 );
// <img src="http://decafbad.com/images/decafbad-title.gif">
```

Asset.image()函数的第一个参数是一幅图像的 URL。它的第二个参数是一个用来定义事件处理程序的对象或散列，而在上面的代码中，我们只定义了 onload 处理程序。

在这个示例中，一旦加载该图像，就会将一个指向这幅图像的<img/>元素实例传给 onload 处理程序调用，然后该处理程序使用.grab()方法把这幅图像插入到页面内容中。

还可以定义几个其他的事件处理程序，如下面的示例所示：

```
var img2 = Asset.image(
    'http://example.com/nonexistent.jpg',
    {
        onload: function(el) {
            $log('*** Image loaded ' + el.src );
        },
        onabort: function(el) {
            $log('*** Image aborted ' + el.src );
        },
        onerror: function(el) {
            $log('*** Image error ' + el.src );
        }
    }
);
// *** Image error http://example.com/nonexistent.jpg
```

在上面的代码中，图像 URL 应该导致程序失败，从而导致调用 onerror 处理程序，并将未能找到其中图像的<img/>元素的引用传给该处理程序。这里还有一个 onabort 处理程序，它用来处理用户单击浏览器停止按钮或采用手动方式终止图像加载的情况。

2. 获取图像集合

按需加载一幅图像非常有用。但如果用户界面需要使用一幅图像，那么通常需要加载大量图像。要是为每幅图像设置一个获取函数，那么就会有点乏味。因此，MooTools 也提供了一种利用 Asset.images()函数来预先加载并跟踪一系列图像 URL 的机制。

下面的代码演示了 Asset.images()的用法：

```
var imgs = Asset.images(
    [1, 2, 3, 4, 5, 6, 7, 8, 9].map(function(n) {
        return 'http://decafbad.com/images/coffee-stains-'+n+'.gif'
    }),
    {
        onProgress: function(counter, idx) {
            // The function context is the image element.
            $('images').grab(this);

            $log(
                "*** Loaded #" + idx + ", " + (counter + 1) + " so far, "
                + this.src
                );
        },

        onComplete: function() {
            $log("*** Completed all images!");
        }
    }
);
```

在上面的代码中，我们使用数组方法.map()构造一个快速的编号图像 URL 列表，并将其作为第一个参数传给 Asset.images()函数。第二个参数是一个对象或散列，它定义了一些在处理图像加载过程中调用的事件处理程序。第一个事件处理程序名为 onProgress，每当这个图像集合中的一幅图像成功加载时都会调用该处理程序。第二个事件处理程序名为 onComplete，当整个集合全部完成加载后将调用该处理程序。

根据前一个示例中的定义，应该出现类似下面的日志文本消息：

```
*** Loaded #4, 1 so far, http://decafbad.com/images/coffee-stains-5.gif
*** Loaded #2, 2 so far, http://decafbad.com/images/coffee-stains-3.gif
*** Loaded #5, 3 so far, http://decafbad.com/images/coffee-stains-6.gif
*** Loaded #0, 4 so far, http://decafbad.com/images/coffee-stains-1.gif
*** Loaded #7, 5 so far, http://decafbad.com/images/coffee-stains-8.gif
*** Loaded #1, 6 so far, http://decafbad.com/images/coffee-stains-2.gif
*** Loaded #3, 7 so far, http://decafbad.com/images/coffee-stains-4.gif
```

```
*** Loaded #6, 8 so far, http://decafbad.com/images/coffee-stains-7.gif
*** Loaded #8, 9 so far, http://decafbad.com/images/coffee-stains-9.gif
*** Completed all images!
```

关于 onProgress 事件处理程序的调用，有以下几个有趣的地方：

- 上下文变量 this 设为指向表示已经完成加载的图像的<img/>元素。在前面的示例中，我们使用这个变量将图像添加到页面内容中。
- 第一个参数 counter 指示进度，也就是当前已经载入多少幅图像，从 0 开始计数。
- 第二个参数 idx 指示在提供给 Asset.images()函数的列表中哪一幅图像已经载入。

借助 Firebug，我们可以看到加载这些额外图像所导致的网络请求，如图 33-1 所示。

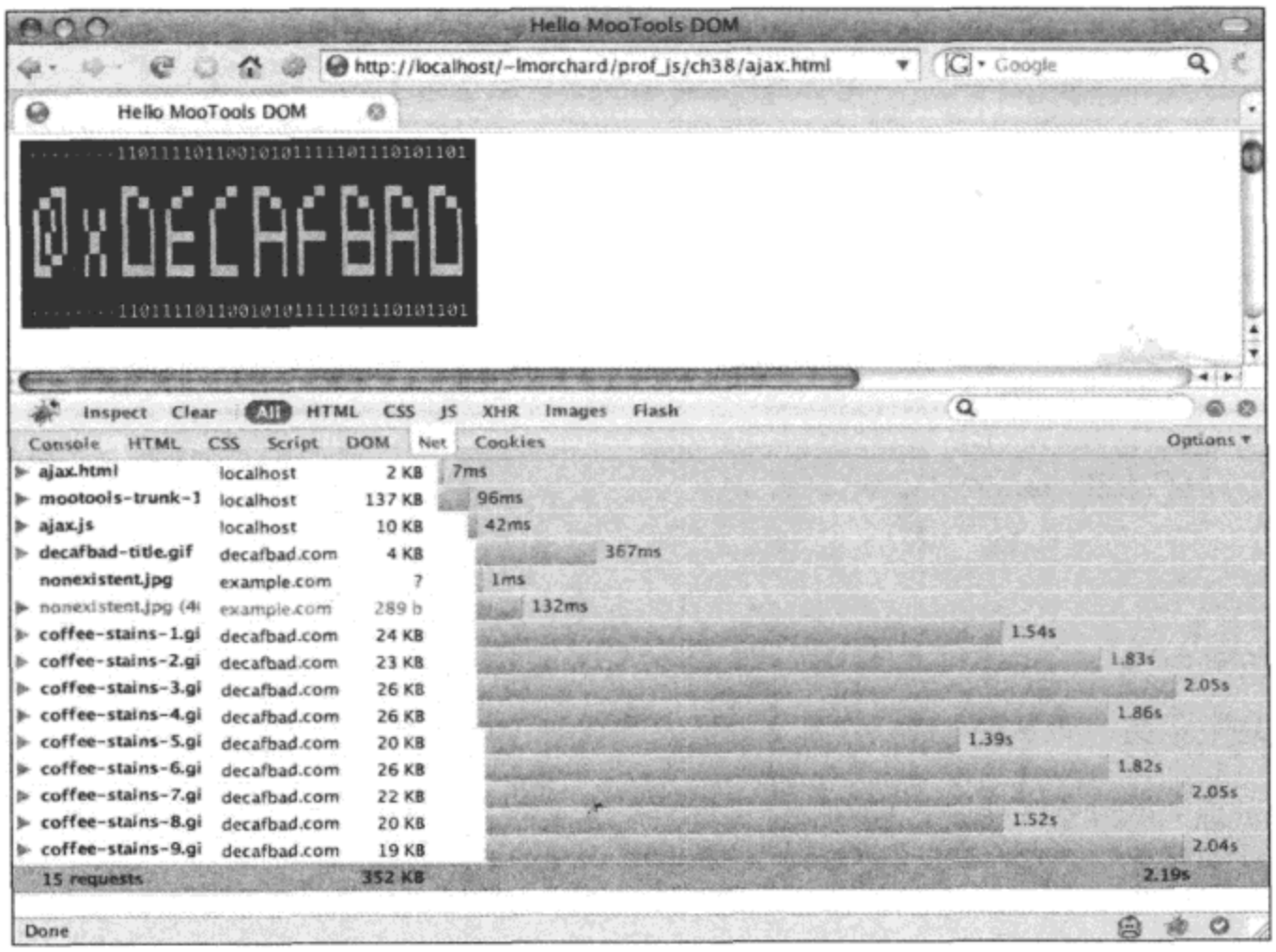


图　33-1

在首次加载时，这些图像可能按照随机顺序到达，如日志文本消息所示。在首次加载之后，这些图像应该已经被本地缓存，因此它们通常会按照各自加载的顺序到达。不管是哪种方式，counter 参数用起来都非常方便，因为我们不能依靠最初提供图像的顺序来确定加载进度。

33.3　建立 Web 请求

根据需要操作 cookie 和获取页面资源相当有用，但如果现代 Web 应用程序不使用功能非常丰富的 XmlHTTPRequest 对象建立相同域的 AJAX Web 请求以提供响应灵敏的、数据驱动的用户界面，那么就不能说它是一个完全专业的 Web 应用程序。

MooTools 提供了一个非常轻量级但具有足够功能的跨浏览器接口来按需建立 Web 请求，包括几个特殊的 MooTools 风格的快捷方式，它们用于将通过 AJAX 获取的动态内容整合到页面中。

33.3.1 执行基本的 Web 请求

MooTools 提供了 Request 类来建立基本的 Web 请求。在创建 Request 新实例时，需要传入一个散列或对象参数，该参数定义了在构造对象时使用的选项。下面的示例演示了一个简单的 HTTP GET 请求：

```
var req1 = new Request({
    method: 'GET',
    url:    'data1.txt',

    onRequest: function() {
        $log("*** Firing request for " + this.options.url);
    },
    onSuccess: function(result_text, result_xml) {
        $log("*** Loaded " + this.options.url);
        $log("*** Data: " + result_text);
    },
    onFailure: function() {
        $log("*** Request failed: " + this.status);
    },
    onException: function() {
        $log("*** Exception occurred!");
        $log(arguments);
    }
});

req1.send();
```

对于这个示例，如果请求成功，那么日志文本消息中就会出现类似下面的消息：

```
*** Firing request for data1.txt
*** Loaded data1.txt
*** Data: Hello there, have some sample data!
```

Request 构造函数的前两个选项指定了该请求的 URL(data1.txt)和 HTTP 方法(GET)，其余的选项定义了该请求的事件处理程序：

- **onRequest** 当请求发送时引发，只有当调用.send()方法时才会发生该事件。
- **onSuccess** 当请求成功完成时调用该处理程序。在响应中发回的文本和 XML 数据将作为参数传入该处理程序。
- **onFailure** 如果请求未能成功结束，就会调用该处理程序。如果该请求导致 404 Not Found 或其他 HTTP 错误状态码，就会出现这种情况。
- **onException** 如果在处理请求或响应的过程中发生任何 JavaScript 异常，就会调用该处理程序。

注意，对于所有这些事件处理程序，上下文变量 this 指向 Request 对象，如果事件处理程序是该对象本身的方法一样。在事件处理程序中可以看到，这可以让我们访问处理程序参数之外的数据，例如 HTTP 状态码以及传给该对象构造函数的初始选项。

1. 使用 Web 请求的事件监听程序

虽然前面的代码示例在一次调用中完成了 HTTP 请求及其事件处理程序的构造，但我们可以使用事件监听程序和 HTTP 对象方法，以一种更为渐进的方式来将所有操作组合起来，如下所示：

```
var req2 = new Request()
    .addEvent('success', function(txt, xml) {
        $log("*** Data loaded: " + txt);
    })
    .addEvent('failure', function() {
        $log("*** Request failed: " + this.status);
    });

req2.addEvent('success', function(txt, xml) {
    $log("*** Me too! Data loaded: " + txt);
});

req2.GET('data1.txt', { /* options */ })

/* req.{get,post,GET,POST,PUT,DELETE}() */
```

运行这段代码之后，在日志文本消息中应该看到诸如下面的消息：

```
*** Data loaded: Hello there, have some sample data!
*** Me too! Data loaded: Hello there, have some sample data!
```

注意，在上面的示例中创建这个 Request 对象时并没有向构造函数提供任何选项。这将产生一个通用实例，我们可以向 request、success、failure 和 exception 中的任何事件附加事件处理程序，如同它们是元素上的原生 DOM 事件一样。

还要注意的是，我们可以将多个事件处理程序附加到同一个事件，正如附加到 success 事件的第二个处理程序所演示的那样。

最后，可以调用.GET()方法来发送 Request 对象。Request 对象有几个方法，分别对应到 HTTP 方法 GET、POST、PUT 和 DELETE。这些方法的参数是请求的 URL，后面跟着一个可选的包含一些选项的散列或对象参数，如果没有提供该参数，那么这些选项将提供给对象构造函数。

由于到目前为止我们只演示了将 URL、HTTP 方法以及事件处理程序作为选项的情形，因此在前面的示例中选项散列为空，但在下一节中我们将会看到还有其他选项可用。

2. 发送报头和请求数据

到目前为止，使用 Request 的示例只是处理简单的 URL、HTTP 方法以及事件处理程序。但对于 HTTP 协议请求来说，还有更多相关操作：可以在请求正文中同时提供报头和额外数据。此外，MooTools 还提供了其他几个特殊功能来负责处理 HTTP 请求和响应。

下面这个示例展示了 Request 对象的额外功能：

```
var req3 = new Request({
    url: 'echo.php',
    method: 'PUT',
    emulation:  false, // default true
```

```
        urlEncoded: false, // default true
        encoding: 'utf-8',
        data: 'This is some sample data!',
        headers: {
            'Content-Type': 'text/plain',

            'X-Alpha': 'foo',
            'X-Beta': 'bar',
            'X-Gamma': 'baz'
        },
        onSuccess: function(txt, xml) {
            $log("*** Text: " + txt);
            $log(xml);
        }
    });
    req3.send();
```

第一个选项 url 设为“echo.php”，该文件将用于请求回应脚本(例如在第 27.4.1 节中提供的脚本示例)。虽然可以构建自己的脚本，但这个脚本的基本思想是产生一个响应，在响应内容中报告它接收到该请求。这将有助于研究 MooTools(以及任何其他框架)建立的 HTTP 请求中发生了什么情况。

第二个和第三个选项(method 和 emulation)多少有点相关：在这个示例中 HTTP 方法设为“PUT”，这是规范中的一个有效的 HTTP 方法(虽然并不是所有浏览器都支持 HTTP PUT 方法)。因此，如果仿真标志(emulation)设为 true(这是其默认值)，那么该请求中实际使用的 HTTP 方法将切换成“POST”。然后，通过添加一个参数的方式将最初请求的方法包含到请求数据的起始处，就像下面这样：

```
_method=PUT &
```

当然，为了正确地使用这种仿真技术，服务器端的应用程序必须支持如下约定：将 HTTP POST 请求中的_method 参数视为真正的预期方法。此外，请求数据必须采用 URL 编码格式，如同它是通过表单提交的一样(这妨碍了在请求中使用原始数据)。

urlEncoded 选项(默认为 true)将把请求中的 Content-Type 报头设置为 application/x-www-form-urlencoded，同时在该报头中包含 encoding 的值。这些默认值通常适合于 AJAX 形式的表单提交(这是非常常见的情形)，但如果希望发送原始数据，就需要对这些选项进行调整，就像上面的示例代码中所做的那样。

这个示例将 urlEncoded 选项的值设为 false，同时还通过 data 选项来提供原始文本。当然，这会与 emulation 标志的使用相冲突。

3. 通过 AJAX 提交表单

由于 Request 对象中的许多默认选项似乎已准备好为 AJAX 形式的表单提交提供便利，因此需要提供一个演示来说明 MooTools 如何处理这类事情。

首先，下面的标记建立了一个用于 AJAX 提交的表单：

```
<div id="ex_form" class="form_widget">
```

```
    <form method="put" action="echo.php?type=text">

        <label for="foo">Foo</label>
        <input type="text" name="foo" id="foo" />
        <br />

        <label for="bar">Bar</label>
        <input type="text" name="bar" id="bar" />
        <br />

        <input type="submit" value="submit this form!" />

    </form>

    <div class="results">
        <pre></pre>
        <a href="#">Reset form</a>
    </div>

</div>
```

这段标记建立了一个准窗口部件，它由一个简单的表单和一个结果显示部分(通过显示成功请求时服务器返回的结果来确认表单提交)组成。

接下来，将下面的 CSS 样式应用于这段标记：

```
<style type="text/css">
    .form_widget  .results { display: none }
    .show_results form     { display: none }
    .show_results .results { display: block }
</style>
```

当最初显示时，这个窗口部件只显示表单本身。在成功提交表单之后会添加 CSS 类“show_results”，这会将表单隐藏起来并显示结果确认信息和表单重置链接。

现在，将上面的所有操作使用代码连接起来，使其具备预期的功能：

```
function ajaxifyFormWidget(widget_id) {

    // Wire up the form submission with AJAX instead of native submit.
    $$("#"+widget_id+" form")
        .addEvent('submit', function(ev) {

            // Intercept the usual form submission process.
            ev.stop();
```

第一段代码定义一个名为 ajaxifyFormWidget()的函数，它的参数是一个元素 ID，这是构建其事件处理程序的基础。该函数所做的第一件事情就是构建一个处理程序来拦截窗口部件中的表单的 submit 事件。

继续实现这个函数，下面引入了所需的 AJAX 表单提交代码：

```
            // Set Request options for the form AJAX request
```

```
        this.set('send', {

            onSuccess: function(txt) {

                // Swap classes to hide the form and show results.
                $$('#'+widget_id).addClass('show_results');

                // Populate the results with server response content.
                $$('#'+widget_id+' .results pre').set('text', txt);

            }

        }).send();

    });
```

这里的代码比较新鲜而且有趣：借助于 Request 模块提供的扩展属性处理程序，表单元素上支持的“send”属性根据指定选项创建了一个隐式的 Request 对象。值得注意的是，这个隐式的 Request 对象存放在元素存储中，这是第 32 章中介绍的一项功能。

如果没有指定明确的选项，那么将自动根据标记中该表单的 action 和 method 属性分别推导出这个隐式 Request 对象的 url 和 method 选项值。这个对象的 data 选项值是一个指向该表单元素自身的引用。

由于表单元素本身已经作为 data 选项传入，因此 Request 的.send()方法将自动使用该表单的.toQueryString()方法。这会将表单字段的内容编码成查询字符串，为提交给服务器做好准备。还有一点值得注意的是，如果为 data 选项指定一个任意的对象，那么 MooTools 将使用 Hash 类的.toQueryString()方法。

为 Request 对象指定的 onSuccess 处理程序将 CSS 类“show_results”添加到窗口部件，这会把表单隐藏起来并展现结果显示区域。在此之后，我们利用从服务器那里获得的文本响应结果来生成结果显示区域的内容，从而为请求中发生的情况提供反馈。

除了“send”属性之外，元素还支持.send()方法。当调用该元素方法时，这个方法又会调用扩展属性处理程序隐式创建的 Request 对象的.send()方法。

因此，在上面的代码中，在为隐式的 Request 对象构造选项之后，立即调用该元素的.send()对象来引发 AJAX 请求。

最后，下面的代码完成了 ajaxifyFormWidget()方法的剩余代码并将其投入使用：

```
    // On click of the reset form link, swap back to form display.
    $$("#"+widget_id+" .results a")
        .addEvent('click', function(ev) {
            $$('#'+widget_id).removeClass('show_results');
        });

}

ajaxifyFormWidget('ex_form');
```

ajaxifyFormWidget()函数最后做的工作就是连接“reset form”链接。这个链接属于在表单成

功提交之后显示出来的结果显示区域的一部分。当单击时，上面代码中附加的事件处理程序将窗口部件上的 CSS 类“show_results”移除，因此将结果显示区域再次隐藏起来，并再次将表单显示出来以供再次尝试。

然后，在完成 ajaxifyFormWidget()函数之后，我们利用在前面给出的标记中曾经用过的 ID“ex_form”作为参数来调用这个函数。这会让表单窗口部件一切就绪。

如果正在使用第 27 章中的 echo.php 脚本，那么向表单中输入一些数据并将其提交应该会产生类似下面的结果：

```
array (
  'method' => 'POST',
  'post_params' =>
  array (
    '_method' => 'put',
    'foo' => 'sadf asdf as',
    'bar' => 'as dfasd f',
  ),
  'query_params' =>
  array (
    'type' => 'text',
  ),
  'x_requested_with' => 'XMLHttpRequest',
  'content_type' => 'application/x-www-form-urlencoded; charset=utf-8',
  'content_length' => '51',
  'request_body' => '_method=put&foo=sadf%20asdf%20as&bar=as%20dfasd%20f',
)
```

在有关该请求的报告详情中，有几点比较有趣的事情：

- X-Requested-With 报头被 MooTools 设置成 XMLHttpRequest，从而允许我们在服务器端检测基于 AJAX 的请求，从而自动修改响应中发回的内容的类型。
- Content-Type 报头设为 application/x-www-form-urlencoded(这是因为 urlEncoded 选项的默认值为 true)。
- 由于调用了表单的.toQueryString()方法，请求正文数据变成了 URL 编码的格式。
- 由于 emulation 选项的默认值为 true，因此通过 HTTP 方法 POST 发送请求，而不是使用标记中指定的 PUT 方法。此外，在请求正文数据开头插入了“_method=put &”以指出最初的 HTTP 方法。

在 MooTools 中通过 AJAX 提交表单是一件非常简单的事情，而且借助扩展属性和方法，该操作实际上已整合到表单元素自身中。实际上，请注意这段代码是如何为表单建立用户界面的，这里并不是基于表单来构建 AJAX 请求。

4. 加载和执行 JavaScript

Request 还有另两个更有趣的选项可用，这两个选项的默认值均为 false，而且它们均用于实现响应内容中的 JavaScript 代码的自动执行：

- **evalResponse**　如果该选项为 true，那么将响应正文的整个内容作为 JavaScript 代码执行。

- **evalScripts** 如果该选项为 true，那么提取响应正文中找到的<script/>标记并将它们的内容作为 JavaScript 代码执行。

因此，文件“data1.js”的内容如下：

```
alert("hello world!");
```

然后，下面的代码用来获取前面的内容：

```
var req_js = new Request({
    method: 'get',
    url:        'data1.js',
    evalResponse: true,
});
req_js.send();
```

如果成功获取，那么这将会弹出一条警告消息“hello world!”。
接下来是文件“data2.html”的内容：

```
<script>alert("hello world!");</script>

This is some sample content.
```

下面的代码用来获取这个文件的内容，它也将产生一条“hello world!”警告消息：

```
var req_js2 = new Request({
    method: 'get',
    url:        'data2.html',
    evalScripts:  true,
    onSuccess:    function(txt) {
        $log("content fetched was " + txt);
    }
});
req_js2.send();
```

但 evalScripts 标志也会导致执行脚本从结果内容中分离。因此，当执行 onSuccess 处理程序时，日志中将出现如下的消息：

```
content fetched was This is some sample content.
```

注意，在提供给 onSuccess 事件处理程序的文本数据中并没有<script/>标记的内容。

33.3.2 获取和更新 HTML 内容

在 AJAX 表单提交示例中所做的一件事情就是使用文本内容更新页面上某个元素的内容。MooTools 提供了一个与此有关的 Request 子类 Request.HTML，从而便于根据需要通过 AJAX 获取内容来更新页面上任何元素的内容。

1. 使用元素方法更新内容

下面是页面上的一段标记。

```
<div id="content"></div>
```

接下来的标记是 URL“data.html”中的可用内容：

```
<script>alert("Script executed!");</script>
<div>
    <h2>Some headline</h2>
    <p>Here is some sample text!</p>
</div>
```

要想根据需要将“data.html”的内容插入到前面的 元素中，最简单的方式是使用元素方法.load()，如下所示：

```
$('content').load('data.html');
```

该.load()调用创建了一个 Request.HTML 实例，利用指定的 URL 引发请求，并将成功请求返回的内容注入到上下文元素中。

但是，尝试执行这段代码就会发现，.load()调用并不只是将内容插入到页面中：同时还会执行<script/>标记中的代码，从而产生一条警告消息。前面曾经提到过，之所以执行响应中的脚本，是因为 Request 类理解 evalScripts 选项。这里的新内容在于 Request.HTML 子类在默认情况下将 evalScripts 标志设为 true。

因此，在后台中，元素的.load()方法在内部使用了一个 Request.HTML 实例。作为 Request 的子类，这个对象除了接受 Request 的所有选项之外还有其他几个选项可用。下面这个扩展示例演示如何在引发加载内容的请求之前设置这些选项中的某一部分：

```
$('content').set({
    load: {
        url:        'data.html',
        method: 'post',
        filter: 'p',
        evalScripts: false
    }
});

$('content').load();
```

可以使用.set()方法操作元素属性“load”，建立在 Request.HTML 对象(在后续的.load()方法调用中创建该对象)中使用的选项。这里可用的选项包括超类 Request 能够理解的所有选项，此外还包括以下的选项以及对默认值的修改：

- **filter**　当响应中的 HTML 被分析成 DOM 元素时，将这个 CSS 选择器过滤器应用于分析的结果。
- **evalScripts**　在 Request.HTML 中默认为 true，这个选项将导致执行响应中出现的<script/>标记。

因此，在上面的示例中不再会出现警告消息，而且在加载的 HTML 内容中只有段落元素才会注入到页面中。

2. 获取 HTML 和 JavaScript 内容

在了解后台中使用的 Request.HTML 对象之后，下面详细查看这个对象。查看下面的代码示例：

```
var req1 = new Request.HTML({

    url:        'data.html',
    evalScripts: false,

    onSuccess: function(tree, elements, txt, js) {
        $log('Document: ' + tree);
        $log('Elements: ' + elements.length);
        $log('Text: ' + text);
        $log('JS: ' + js);
    }

});
req1.send();
```

一旦运行，该代码将产生如下的日志文本消息：

```
Document: [object NodeList]
Elements: 3
Text:
<div>
    <h2>Some headline</h2>
    <p>Here is some sample text!</p>
</div>
JS: alert("Script executed!");
```

相对于 Request 超类，在这里提供给 onSuccess 处理程序的参数有所变化。新的参数如下所示：

- tree　表示获取内容的已分析 HTML 结构的文档片段。
- elements　HTML 内容中找到的所有元素组成的简单列表，适合于进行过滤。
- text　来自于响应的原始文本内容，包含 HTML 标记，但不包含脚本标记。
- js　从<script/>标记中分离的 JavaScript 代码，如果 evalScripts 为 true，该代码就会得以执行。

可以看到，对于 HTML 内容来说，这些经过修改的参数更为有用，当加载时，它们将在后台自动地分析成 DOM 节点。

3. 利用请求对象更新内容

元素的.set()和.load()组合调用在后台隐式地使用 Request.HTML 对象，而下面的代码则是显式地使用该对象：

```
var req2 = new Request.HTML({
    url:        'data.html',
    update:     $('content'),
    filter: 'p',
```

```
    evalScripts: false
});
req2.GET();
```

这段代码所做的工作与前面给出的.set()和.get()示例所做的工作基本一样：从 URL“data.html”载入内容，将内容分析成 HTML，从该 HTML 中过滤出<p>元素，然后使用过滤出来的元素替换 ID 为“content”的元素的内容。

在前面已经看到过新选项 filter，它将一个过滤器应用于经过分析的 HTML 中所包含的一系列元素。这里的新内容是 update 选项，它所做的事情基本上与在指定元素上调用.load()一样：元素的内容将被替换成从服务器响应正文中加载并经过分析的文档片段。

33.3.3　请求并使用 JavaScript 和 JSON 数据

虽然我们确实能够使用通过 Asset.javascript()函数注入到报头中的脚本标记从第三方域获取 JavaScript 和 JSON 资源，但是这种方式仅限于 HTTP GET，而且只能是使用查询字符串能够提供的参数。此外，要想检测某个资源是否已经成功地完成加载，需要使用服务器支持的回调技术，或者使用一个定期检查资源是否存在的函数。

如果不需要跨域资源的灵活性，那么可以使用 Request.JSON 类对本地 API 的 Web 请求进行更明确的控制。作为 Request 的子类，它提供了相同的所有选项和事件处理程序，并增加了 JSON 数据自动处理功能。

1. JSON 数据编码和解码

但在深入研究 Request.JSON 类之前，JSON 模块值得我们花费一点时间浏览一下。这个模块帮助将 JavaScript 对象编码成 JSON 文本格式以及从 JSON 文本格式解码成 JavaScript 对象。

例如，下面的代码使用 JSON 模块添加到原生数据类型 String 和 Number 中的.toJSON()方法：

```
$log( 'string with "quotes" and \\backslashes\\ and /slashes/'.toJSON() );
// "string with \"quotes\" and \\backslashes\\ and \/slashes\/"

$log( (105.3).toJSON() );
// 105.3
```

这段代码并不会让人觉得多么兴奋，因为 JSON 编码与原始 JavaScript 代码并无多大差异。但是，.toJSON()方法也可用于 Array 和 Hash 实例：

```
var some_array = [
    'foo', 'bar', 'baz'
];
$log( some_array.toJSON() );
// ["foo","bar","baz"]

var some_hash = $H({
    alpha: 'foo',
    beta: 'bar',
    gamma: 'baz'
});
```

```
$log( some_hash.toJSON() );
// {"alpha":"foo","beta":"bar","gamma":"baz"}
```

但当使用原生的通用对象时，并不能使用.toJSON()方法：

```
var some_obj = { some_key: 'some_value' };

try {
    some_obj.toJSON();
} catch(e) {
    $log( "Exception: " + e );
    // Exception: TypeError: some_obj.toJSON is not a function
}
```

相反，如果希望将一个原生 JavaScript 对象编码为 JSON，那么可以直接使用 JSON.encode()函数：

```
$log( JSON.encode(some_obj) );
// {"some_key":"some_value"}
```

广泛使用的.toJSON()方法实际上只是 JSON.encode()函数的一个特殊形式，JSON.encode()可以用于任何其他数据类型：

```
$log( JSON.encode(some_array) );
// ["foo","bar","baz"]

$log( JSON.encode(some_hash) );
// {"alpha":"foo","beta":"bar","gamma":"baz"}
```

反过来，要从给定的 JSON 数据字符串中获得 JavaScript 对象和数据类型，就要使用 JSON.decode()函数，如下所示：

```
var str     = some_hash.toJSON();
var new_obj = JSON.decode(str);

$log( some_hash.every( function(v,k) {
    return new_obj[k] == v
}) );
// true
```

上面的代码演示了.toJSON()与 JSON.decode()之间的反向关系：首先将一个散列编码，然后再将其解码。将原始对象的数据与新解码对象的数据进行比较，应该可以揭示原始对象中的所有数据都同样存在于经历编码和后续解码的新对象中。

但为了周密起见，下面的代码检查新对象中的所有数据是否能在原始对象中找到：

```
// new_obj is not yet a hash...
var new_hash = $H( new_obj );

$log( new_hash.every( function(v,k) {
    return some_hash[k] == v
}) );
// true
```

上面代码中的一个技巧是 JSON.decode()返回的对象实际上并不是一个类似原始对象的 Hash 对象：相反，它是一个通用 JavaScript 对象，需要将其传入便利函数$H()以将其转换成一个包含 MooTools 定制方法.every()的 Hash 对象。因此，这里有一个警告需要注意，JSON 并不理解 MooTools 定制类，因此在解码时不会生成这些类。

JSON 数据安全检查

MooTools 分析 JSON 的方式非常简单：使用 JavaScript 函数 eval()，除非给定经过预先加工的输入或进行监控，否则它将能够任意地执行任何代码：

```
var result1 = JSON.decode(
    "window.alert('sideeffect #1')"
);
// throws up an alert
```

当执行时，上面的代码确实不会生成 JSON 数据，而是会产生一条警告消息。

如果能够信任自己的代码生成的数据以及自己的服务返回的数据，那么使用 eval()并不失为一种不错的选择。它利用浏览器提供的 JavaScript 分析器，因此执行速度应该相当快，而且能够处理所有 JSON 构造。

但如果计划执行来自自己并不是那么信任的来源的 JSON，那么可以将第二个参数设为 true，这会强制把 JSON 字符串传给 JSON.decode()，首先让 JSON 中允许的字符通过：

```
var result2 = JSON.decode(
    "window.alert('sideeffect #2')", true
);
// no alert, null returned, tests 'JSON' with a regex
```

可以看到，这个函数调用未能通过测试，因此终止 eval()调用。这应该能够让 JSON 处理更加能够抵御违反该机制的攻击尝试。

2. 建立 JSON 数据请求

我们已经了解了在 MooTools 中如何处理 JSON 数据，接下来快速地了解 Request.JSON 对象。假设文件“data.json”的内容如下：

```
{
    "first":"hewey",
    "second":"dewey",
    "third":"louie",
    "fourth":[
        1, 2, 3, 4, 5
    ]
}
```

可以使用 Request.JSON 对象获取该数据，如下所示：

```
var req_json = new Request.JSON({
```

```
    url:    'data.json',
    secure: true, // true by default anyway

    onSuccess: function(json, txt) {
        $log("*** Loaded JSON data");

        $log_json( json );
        // {"first":"hewey","second":"dewey","third":"louie","fourth":[1,2,3,4,5]}

        $log( json == this.response.json );
        // true
    }

});
req_json.GET();
```

作为 Request 的子类，本章目前已经介绍的关于该类的其他内容也均适用于 Request.JSON，并增加了以下几项：

- 向请求中添加了 X-Request: JSON 报头。
- 同时还向请求中添加了 Accept: application/json 报头。
- 隐式地调用 JSON.decode()，该调用的结果将放入 Request.JSON 实例的 response.json 属性中，同时还将作为第一个参数传入正在调用的 onSuccess 处理程序。
- 选项 secure 是一个 Boolean 标志，将作为第二个参数传给 JSON.decode()调用(用来分析响应正文内容)。前面曾经描述过，这会强制接收到的 JSON 数据首先经过白名单(whitelist)测试以确定该 JSON 数据不会造成安全问题。注意，在默认情况下该选项为 true(如果忽略该选项)。

但除了这些调整之外，Request 对象的其他所有方面均适用于此处，包括 HTTP 支持的所有方法、额外的报头和响应正文处理以及 HTTP 方法仿真功能。此外，secure 标志为 JSON 数据的运行提供了一定的安全性检查。

Asset.javascript()调用不支持上述所有方面，因此在处理采用 JSON 并与自己的网页处于同一个域的 Web API 时，Request.JSON 很明显是最佳选择。而且，即使在不可避免地要进行跨域访问的场合中，我们通常也能够构建一个本地域代理，在有限的范围内中转自己页面上的某些调用。

33.4 本章小结

在这一章中，我们介绍了 MooTools 提供的一些 AJAX 和动态数据工具，包括使用 XmlHTTPRequest 对象包装 Web 请求和响应的 Request、Request.HTML 和 Request.JavaScript 系列类。我们还演示了 MooTools 提供的 cookie 操作和散列持久化机制，以及动态加载页面素材的函数。

在下一章也是最后一章中，我们将浏览 MooTools 提供的视觉效果工具，这是许多人判断这个框架是否值得使用的决定性功能之一。

第34章

构建用户界面以及使用动画

在本书有关 MooTools 的最后这一章中(但显然并非是最不重要的一章)，我们将介绍提供视觉效果的模块以及简洁但相当有用的用户界面窗口部件。实际上，这部分 MooTools 功能在该工具集本身面世之前就已经存在，它们最初是作为 Moo.Fx 发布的，这是一个轻量级的、全功能的 Prototype JS 框架伴随库。因此，这些效果和 UI 模块已经经历更长的时间而得以成熟，并将注意力放在了性能和开发人员易用性方面。

MooTools 提供的动画和用户界面模块构建在目前已经介绍的所有其他工具集便利工具的基础之上。在本章中，我们将了解如何使用元素的扩展属性和方法来方便地将平滑的视觉转换和其他细微的使用体验融入到 Web 应用程序中。

本章内容简介：

- 编排动画
- 使用用户界面窗口部件

34.1 编排动画

为了演示 MooTools 如何构建动画，或许需要提供几个快捷方式来构建元素动画，而不需要使用大量的冗余标记。

首先考虑如下 CSS 代码：

```
<style type="text/css">
    .mover {
        position: relative;
        margin: 0.25em;
        padding: 1em;
        width: 150px;
        float: left;
        border: 2px dotted #000;
        background-color: #ddf;
        text-align: center;
    }
</style>
```

这里建立了一个简单的样式，用于展示 MooTools 提供的各种效果。接下来，为了在页面上创建这样的元素，考虑下面的 JS 函数：

```
function createMoverDivs(parent_el, num, id_pre) {
    parent_el = $(parent_el);
    for (var i=1; i<=num; i++) {
        parent_el.adopt(
            document.newElement('div', {
                'class': 'mover',
                'id':    id_pre + i,
                'text': 'Click me'
            }),
            document.newElement('br', {
                'style': 'clear: both'
            })
        )
    }
}
```

在本章中到处都要使用上面代码中定义的函数来代替在许多地方使用的标记。它的作用很简单：给定一个父元素、要创建的元素数量以及一个 ID 前缀，该函数就会创建并插入多个带有唯一 ID 和 CSS 类“mover”的元素。这在演示 MooTools 技术的许多变体时非常方便。

图 34-1 给出了使用这个函数的一个页面示例。

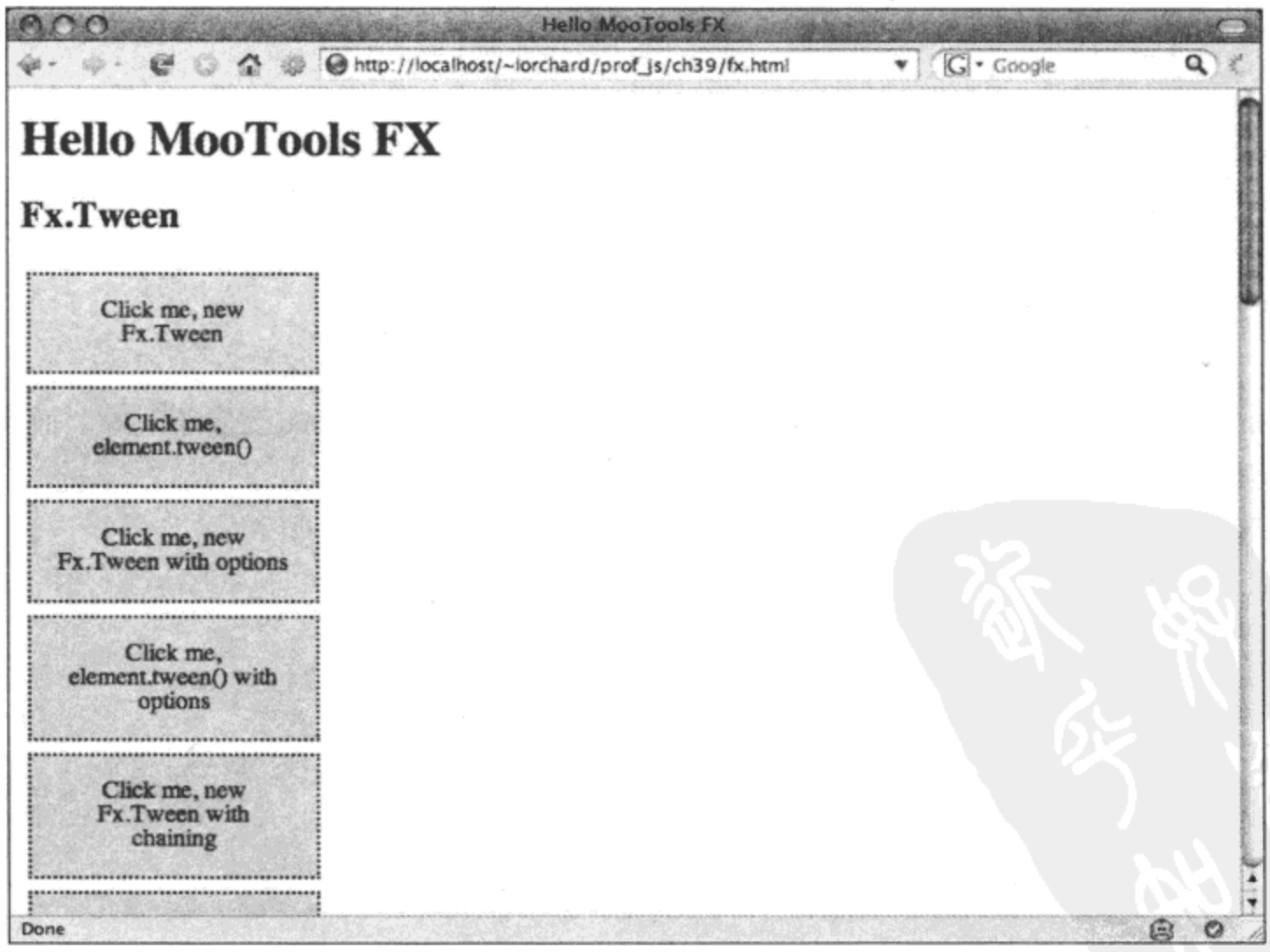


图 34-1

34.1.1　检查元素大小和位置

在深入研究动画之前，还有一项准备工作要做：由于许多动画都严重依赖于对位置、大小以及元素滚动的操作，因此值得我们花费一些时间来研究 MooTools 提供用来处理所有上述方面的 Element.Dimensions 模块。

下面的标记是一个演示平台：

```
<style type="text/css">
    #ex_element_dimensions .scrolling {
        width: 100px;
        height: 100px;
        overflow: auto;
    }
</style>
<div id="ex_element_dimensions">
    <h2>Element.Dimensions</h2>

    <div class="mover scrolling">
        I can has dimensions!
        Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Cras
        imperdiet velit at arcu. Nullam viverra lectus ac felis. Cum sociis
        natoque penatibus et magnis dis parturient montes, nascetur ridiculus
        mus. Nullam eu sapien dictum purus consequat cursus. Etiam non sapien
        sed elit dignissim tristique.
    </div>
    <br style="clear: both" />
</div>
```

上面示例中定义的标记构造了一个带有足够多内容(从而强制出现 CSS 中声明的溢出和滚动栏)的元素。这段标记应该提供了足够的结构来试验 Element.Dimensions 提供的各种实用工具。首先查看这些实用工具方法调用：

```
var el = $$('#ex_element_dimensions .mover')[0];

$log('*** el.getScrollSize()');
$log( el.getScrollSize() );
// Object x=117 y=512

$log('*** el.scrollTo()');
el.scrollTo( 0, el.getScrollSize().y / 3 );
//

$log('*** el.getScrolls()');
$log( el.getScrolls() );
// Object x=0 y=170
```

可以看到，这个模块提供了多个元素方法来实现如下操作：询问元素提供的滚动限制(.getScrollSize)、通过编程方式将元素滚动到指定的位置(.scrollTo)以及获取当前滚动位置

(.getScrolls)。可以在上面给出的元素上使用这些函数，甚至可以用于 document 或 window 对象以控制整个浏览器。

接下来，查看处理元素位置和大小的代码：

```
$log('*** el.getCoordinates()');
$log( el.getCoordinates() );
// Object left=12 top=65 width=136 height=136 right=148

$log('*** el.getSize()');
$log( el.getSize() );
// Object x=136 y=136

$log('*** el.getPosition()');
$log( el.getPosition() );
// Object x=12 y=65

$log('*** el.position()');
el.position({ x: 150, y: 10 });
```

第一个元素方法.getCoordinates()返回一个对象，其中列出了有关该元素的大小以及其在页面中位置的详细信息。由于这里的元素并没有采用绝对定位，因此这些数字是相对于它的父元素以及它的滚动位置。方法.getSize()和.getPosition()分别以单个对象的形式返回元素的大小和位置。

为了实际地感受这些方法，应该使用相对定位和绝对定位的不同组合(以及在父元素上的外边距和内边距组合)来实验这些方法。可以在图 34-2 中看到上面的代码的运行情况。

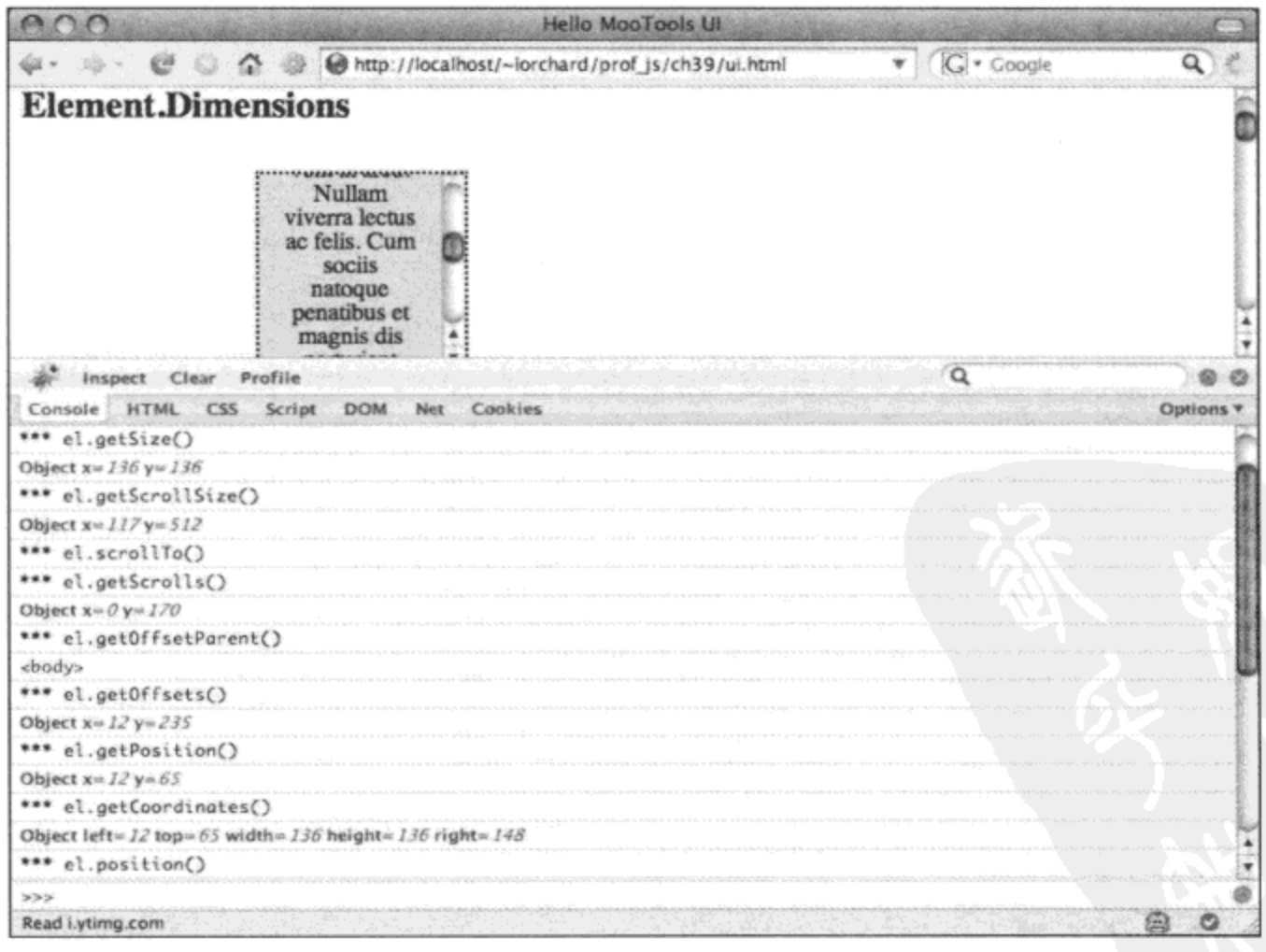


图　34-2

34.1.2　使用 MooTools Fx 编排动画

现在实际地开始研究 MooTools 动画：Fx.Tween 模块(以及它所基于的 Fx 和 Fx.CSS 模块)为通过随时间流逝改变 CSS 属性值的方式来产生动画提供了基础。由于这些属性包括位置、大小以及颜色，因此这就为生成运动以及其他转换提供了非常强大的基础。

首先，查看如下标记：

```
<div id="ex_tween">
    <h2>Fx.Tween</h2>
</div>
```

然后，回顾 createMoverDivs()函数，检查如下调用：

```
// Create a set of divs for this demo
createMoverDivs('ex_tween', 11, 'tween_mover');
```

上面的代码向标记中声明的父元素添加了 11 个编号的 元素，所有这些元素都将会在下面的示例中用到。回头查看图 34-1，了解这段代码的运行情况。

1. 使用 Fx.Tween 实例

查看如下使用 Fx.Tween 模块的示例：

```
$('tween_mover1').set({
    text: 'Click me, new Fx.Tween',
    events: {
        click: function(ev) {
            var tweener = new Fx.Tween(ev.target);
            tweener.start('left', '0px', '300px');
        }
    }
});
```

在上面的示例中，大部分代码的功能是附加一个事件监听程序来响应前面创建的第一个 元素上的单击事件，同时将其文本内容修改成稍微更有描述性的信息。可以认为这是第 32 章中介绍的.set()方法的又一次演示，本章中还有一些有关它的示例。

但在单击事件处理程序内部，我们创建了一个 Fx.Tween 实例。这个类的构造函数的第一个参数是一个元素，在这里就是传入的单击事件的目标元素。

一旦构造完毕，就会调用该对象的.start()方法。这个方法附带 3 个参数：

- CSS 属性的名称
- 该属性的起始值
- 该属性的结束值

一旦调用该方法，MooTools 就开始将指定 CSS 属性从起始值转换直到其等于结束值。在这个特定示例中，元素的 left 位置属性经过默认的 500 毫秒(半秒)持续时间从 0 像素横跨页面移动到 300 像素处。因此，通过改变位置属性就实现了动画的运动效果。

尽管用起来非常简单，但这里的解释掩盖了 MooTools 深层的一些细节。最重要的是，它在 Fx.CSS.Parsers 名称空间中实现了一组处理程序，它们知道如何解释带有单位的数字值(如“300px”)

甚至颜色值(如“#338833”)，而且一旦经过分析，这些处理程序就知道如何在指定的动画持续时间之内将属性值在两个端点值之间转换。

这意味着不仅可以产生带有运动的动画，而且可以在不同颜色、大小、滚动位置和任何其他数值型属性之间进行转换。

一般来说，我们绝不需要担心 CSS 分析器本身，这是因为绝大多数值得制作动画的属性都已经由 MooTools 提供的分析器涵盖。当然，确实有一些属性，没有任何一种形式的动画转换真正对其有意义(例如 font-face 和 list-style-type)；但 MooTools 仍然试图帮助我们提供这样的动画，至少确保该元素从起始值开始补间动画并在动画持续时间结束时以结束值终止。

2. 使用元素方法 tween

从本书有关 MooTools 的部分中目前已经了解的内容来看，只要有一个类可用于处理元素，就会有另一种直接集成到该元素自身中的方法来完成同样的工作。同样，在 Fx.Tween 模块中也存在这种模式：

```
$('tween_mover2').set({
    text: 'Click me, element.tween()',
    events: {
        click: function(ev) {
            ev.target.tween('left', '0px', '300px');
        }
    }
});
```

与前面一样，这里的大部分代码是为演示中涉及的 元素设置适当的描述信息，同时设置一个单击事件处理程序。在这个处理程序函数中，简单地调用了元素的.tween()方法，并向这个方法传入与提供给 Fx.Tween 对象的.start()方法相同的 3 个参数。

这里并没有什么巧合之处，实际上.tween()方法在后台创建了一个 Fx.Tween 实例，并使用 CSS 属性名称以及它的起始值和结束值作为参数来调用该实例的.start()方法。与前面的示例类似，这里的代码会导致页面上的第二个 元素在单击时从 0 像素处横跨屏幕移动到 300 像素处。

3. 指定 Fx.Tween 选项

除了指定将对其制作动画的 CSS 属性的起始值和结束值之外，Fx.Tween 还有其他几个选项可通过其构造函数的可选的第二个参数来设置，在下面这个示例中给出了这些选项：

```
$('tween_mover3').set({
    text: 'Click me, new Fx.Tween with options',
    events: {
        click: function(ev) {
            var tweener = new Fx.Tween(ev.target, {
                duration:   1000,
                transition: Fx.Transitions.Elastic.easeOut
            });
            tweener.start('left', '0px', '300px');
        }
```

```
    }
});
```

与第一个示例的用法一样，可以看到在这个单击事件处理程序中传入 Fx.Tween 构造函数的第一个参数是一个指向所单击 元素的引用。但第二个参数是一个对象，它提供了在初始化动画时应用的更多选项。

返回到启动 Fx.Tween 动画的元素方法，可以使用扩展元素属性 tween 设置在调用.tween()方法时要用到的选项，如下所示：

```
$('tween_mover4').set({
    text: 'Click me, element.tween() with options',
    tween: {
        duration:  1000,
        transition: 'elastic:out'
    },
    events: {
        click: function(ev) {
            ev.target.tween('left', '0px', '300px')
        }
    }
});
```

通过.set()方法和 tween 属性，使用上下文元素和给定选项创建一个 Fx.Tween 实例，并将其保存到元素存储中以供调用.tween()方法时使用。这样就能够在与操作其他元素属性和处理程序相同的上下文中创建补间动画。

下面列出了大多数从 Fx 类派生的动画(包括 Fx.Tween)中常用的选项：

- fps　每秒的帧数，默认为 50。
- unit　属性动画中使用的单位，也就是“px”、“em”或“%”。默认为 false，也就是使用传给.start()值时所指定的单位。
- link　接受如下 3 个字符串值之一：
  - cancel　多次.start()调用会中断正在运行的动画(默认值)。
  - chain　多次.start()调用均会附加到调用链中。
  - ignore　多次.start()调用会被忽略。
- duration　动画应该持续的时间，单位为毫秒。
- transition　指向用来定制动画线性进度的函数的引用，默认为平稳地逐渐加速，然后停止。

其中，duration、link 和 transition 可能是最常用的几个选项。duration 选项最容易理解，而 link 和 transition 选项将会在本章后面进行更详细的讲解。

4. 链接补间动画

一旦建立一个基本动画并运行起来，那么下一步要做的工作就是把它与一系列动画中的后续动画链接起来以构建更复杂的运动和转换。在 Fx 基类中通过使用 Chain 混入类(在第 31 章中介绍过该类)来实现该功能。可以利用 Fx.Tween 来使用这项功能，如下所示：

```
$('tween_mover5').set({
    text: 'Click me, new Fx.Tween with chaining',
    events: {
        click: function(ev) {
            var tweener = new Fx.Tween(ev.target, {
                duration:  1000,
                transition: 'elastic:in:out'
            });

            tweener
                .start('left', '0px', '300px')
                .chain(function(){
                    tweener.start('left', '300px', '0px')
                });
        }
    }
});
```

上面的代码构造了一对链接起来的动画，当单击该元素时，它从左边滑出 300 像素，然后立即返回到页面的左边。

由于 Fx 实现了 Chain 混入类，而 Fx.Tween 继承自 Fx 类，因此可以在 Fx.Tween 实例上使用.chain()方法，如同上面的代码所示。

.chain()方法调用的参数是一个函数引用，在后续调用.callChain()方法时会调用该函数。由于 Fx 类的实例在每次完成动画时都会调用自己的.callChain()方法，因此可以方便地利用这项功能将一个动画链接在另一个动画的后面，还可以在动画结束时触发其他代码。

我们可以在扩展元素属性和方法的上下文中使用这种链接功能，而且通过将 link 选项设置为“chain”可以使其更加简化，其代码如下所示：

```
$('tween_mover6').set({
    text: 'Click me, element.tween() with chaining',
    tween: {
        link: 'chain',
        duration:  1000,
        transition: Fx.Transitions.Bounce.easeOut
    },
    events: {
        click: function(ev) {
            ev.target
                .tween('left', '0px', '300px')
                .tween('left', '300px', '0px');
        }
    }
});
```

元素的.tween()方法返回的是该元素自身，而不是在后台创建的 Fx.Tween 对象的引用。但我们可以通过元素存储访问这个对象，因此调用.get('tween')就会返回所需的 Fx.Tween 实例，而且可以在该实例上调用.chain()方法来附加后续的动画，就像前面的示例一样。

该操作稍微有点繁琐，因此在编排复杂的动画链时，我们可能希望坚持直接创建 Fx.Tween

实例，并在引发快速而简单的转换时使用元素方法和属性。

34.1.3　研究预制动画和效果

虽然通过使用 Fx.Tween 类可以自由地编排任何类型的单一属性动画，但还有几个特殊类型的动画可以方便地用于 Web 应用程序。因此，MooTools 提供了几个非常易于引发的预制动画，而不需要执行在构造定制程度更高的动画时所需的先期工作。

1. 使用淡入淡出动画

下面的代码改变给定元素的不透明度，使其先淡出，而后淡入并恢复原状：

```
$('tween_mover7').set({
    text: 'Click me, element.fade()',
    events: {
        click: function(ev) {
            ev.target.fade();
            ev.target.fade.delay(1000, ev.target);
        }
    }
});
```

该单击事件处理程序调用元素的.fade()方法，这会在后台使用 Fx.Tween 实例构造并启动一段动画。这个方法只有一个参数，它可接受如下选项之一：

- in　将不透明度从 0.0 经过制作动画变换成 1.0，将该元素淡入视图。
- out　将不透明度从 1.0 经过制作动画变换成 0.0，将该元素淡出视图。
- toggle　在后续调用中交替地对不透明度制作动画，将元素淡入和淡出视图。
- show　将元素的不透明度设置为 1.0，不带动画。
- hide　将元素的不透明度设置为 0.0，不带动画。
- 0.0-1.0　将元素的不透明度从它的当前值经过制作动画变换成指定值。

.fade()方法使用扩展属性“tween”处理 Fx.Tween，因此就像本章前面所示，可以使用该方法设置某些选项以重写默认值，例如 duration。

此外还要注意的是.fade()的第二次调用(通过.delay()函数方法)。尽管不如链接动画那么清晰或精确，但这是在完成一段动画之后调用另一段动画的简易方式。

2. 使用高亮显示动画

与淡入淡出动画一样，现代 Web 应用程序中流行的另一类动画是将页面上发生的变化通过颜色高亮显示形成快速的闪烁(flash)效果。可以使用 MooTools 实现效果，如下所示：

```
$('tween_mover8').set({
    text: 'Click me, element.highlight()',
    events: {
        click: function(ev) {
            ev.target.highlight();
            // ev.target.highlight('#8f8');
        }
```

```
    }
});
```

通过使用上面的代码，当单击该元素时，它将短暂地变成淡黄色，然后再淡入回原来的颜色。通过调用该元素的.highlight()方法来实现该操作，该方法也接受一个颜色作为其唯一可选参数，用来替换默认的黄色。

与.fade()类似，.highlight()方法也使用了隐含的 Fx.Tween 对象(在使用扩展属性“tween”时涉及该对象)。

34.1.4 使用 Fx.Slide 动画

淡入淡出和高亮显示可以改变静止元素的不透明度和颜色，而滑动动画则通过将元素扩大进入页面布局以及从页面布局中缩小退出来隐藏或显示该元素。

与其他预制动画不同的是，Fx.Slide 是作为 Fx 基类的一个独立子类实现的，而不仅仅是 Fx.Tween 的一个变体。可以像下面这样使用这个类的实例：

```
$('tween_mover9').set({
    text: 'Click me, new Fx.Slide()',
    events: {
        click: function(ev) {
            var slider = new Fx.Slide(ev.target, {
                duration:   1000,
                transition: Fx.Transitions.Bounce.easeOut,
                mode:       'vertical'
            });
            slider
                .slideOut()
                .chain(function() {
                    slider.slideIn();
                });
        }
    }
});
```

可以看到，这个类支持从 Fx 父类那里继承的所有功能，包括各种选项和动画链。但是需要注意，这里有一个新的选项“mode”，它的值可以是“vertical”(如果忽略该选项，则取该默认值)或“horizontal”。这个选项指定是否应分别沿着垂直轴和水平轴将该元素增大或缩小。

但除了一个新选项之外，Fx.Slide 提供的主要是如下的一组方法调用，每个调用都提供了一种稍微不同的效果：

- .slideIn() 将元素通过逐渐增大的方式滑进页面布局中。
- .slideOut() 将元素通过逐渐缩小的方式滑出页面布局。
- .toggle() 在后续的调用中交替地将元素滑进和滑出页面布局。
- .hide() 将元素从页面布局中隐藏，不带任何动画转换。
- .show() 将元素插入到页面布局中，不带任何动画转换。

注意，对于所有这些方法，该元素必须已经成为 DOM 的一部分；这个类提供隐藏和显示转换功能，但它并不实际地移除或插入元素。Fx.Slide 类只是影响元素在可见页面布局中的可见性

以及大小。

当然，除了直接使用 Fx.Slide 对象之外，还可以利用该模块提供的扩展元素属性和方法，如下所示：

```
$('tween_mover10').set({
    text: 'Click me, element.slide()',
    slide: {
        link:       'chain',
        duration:   1000,
        transition: Fx.Transitions.Elastic.easeOut,
        mode:       'horizontal'
    },
    events: {
        click: function(ev) {
            ev.target
                .slide('out')
                .slide('in');
        }
    }
});
```

与 Fx.Tween 动画类似，可以使用“slide”扩展属性提供选项，并调用.slide()方法启动动画。.slide()方法接受一个可选的字符串参数来指定 Fx.Slide 类提供的动画方法之一，也就是 in、out、toggle(默认值)、hide 或 show。

此外，与 Fx.Tween 动画类似，将 link 选项设为“chain”以便于后续调用.slide()来链接动画，并可以使用“slide”属性获取 Fx.Slide 类的隐式实例。

34.1.5　使用 Fx.Scroll 动画

MooTools 提供的另一类有用的动画涉及在元素内滚动显示内容，以及通过编程方式滚动浏览器窗口自身。通过另一个 Fx 子类 Fx.Scroll 来实现该功能，其用法如下所示：

```
$(' tween_mover10').set({
    text: 'Click me to scroll!',
    events: {
        click: function(ev) {

            var scroller = new Fx.Scroll(window, {
                duration: 1500,
                transition: Fx.Transitions.Bounce.easeOut
            });

            scroller
                .toTop()
                .chain(function(){
                    scroller.start(0, 200);
                })
                .chain(function(){
```

```
                scroller.toBottom();
            })
            .chain(function(){
                scroller.toTop();
            })
            .chain(function(){
                scroller.toElement(ev.target);
            });

        }
    }
});
```

当单击上面代码中涉及的元素时，并不会对元素自身产生任何影响。相反，浏览器窗口自身开始滚动。首先，它滚动到页面的顶部，然后向下滚动 200 像素，接下来沿着该方向一直滚动到页面底部，最终返回到所单击元素在浏览器窗口顶部所处的位置。

这段代码有助于演示 Fx.Scroll 提供的方法：

- .start(x, y) 启动一段动画滚动到预期的水平和垂直位置。
- .toElement(el) 启动一段动画滚动以使指定元素可见。
- .toTop() 滚动到垂直空间的顶部。
- .toBottom() 滚动到垂直空间的底部。
- .toLeft() 滚动到水平空间的左边。
- .toRight() 滚动到水平空间的右边。

但与目前已经给出的其他动画工具不同的是，Fx.Scroll 模块并没有提供相关的元素属性或方法。相反，我们需要直接使用这个类。

34.1.6 研究 MooTools Fx.Transitions

到目前为止，本章已经使用所有 Fx 子类均接受的“transition”选项，但尚未对其给出多少解释。您可能已经理解这个选项的作用，这是因为许多其他 JavaScript 框架也提供了类似的功能(例如，Dojo 和 YUI 将这些函数称为“缓动”)。

简而言之，在动画中进行线性运动毫无趣味可言。仅仅观察一个对象从 A 点缓慢地“爬行到”B 点是一件索然无味的事情，无法吸引观众的注意力。更有趣的做法是让屏幕上的对象以某种近似于带有惯性或动量的物理运动方式(甚至以某种不自然的方式，但这至少会在事情进展过程中带有变化)移动。

因此，在 MooTools 中，通用 Fx 选项“transition”可以设成一个函数引用，当使用一个介于 0.0 到 1.0 之间的值调用该函数时，它将以一种有趣的方式引入某种变换，从而让结果变得不同。

但是，与其试着乏味地讲解什么是“有趣的”方式，不如提供一个工具来交互地演示 MooTools 的 Fx.Transitions 名称空间下直接提供的所有转换函数。

下面的标记是这种工具的开始部分：

```
<div id="ex_transitions">
    <h2>Fx.Transitions</h2>

    <form>
        <select class="transitions"></select>
        <button class="animate">Animate</button>
    </form>

    <div class="mover">
        <div>Watch me!</div>
    </div>

    <br style="clear: both" />

</div>
```

上面的标记建立了一个简单的测试平台，提供了一个选择框窗口部件来挑选转换函数，并且给出了一个按钮来启动一段使用该函数的动画。这段动画将应用于该标记中唯一的 元素“mover”。

下面的代码找出 Fx.Transitions 下所有可用的转换函数，并构造一个由函数名称和相关函数组成的散列，在后面构造并运行该测试界面时将使用该散列：

```
// Collect candidates for MooTools transition functions.
var base_name   = 'Fx.Transitions.';
var transitions = $H();
Fx.Transitions.getKeys().each(function(name) {
    ['', 'easeIn', 'easeOut', 'easeInOut'].each(function(func) {
        var t_name = base_name + name;
        if (func == '') {
            var t_func = Fx.Transitions[name];
        } else {
            var t_func = Fx.Transitions[name][func];
            t_name += '.' + func;
        }
        if ($type(t_func) == 'function')
            transitions[t_name] = t_func;
    });
});
```

Fx.Transitions 名称空间实际上是一个 MooTools 散列对象，因此.getKeys()方法找出所有已定义转换的名称。在编写本书时，这包括如下的名称：

- Fx.Transitions.linear
- Fx.Transitions.Pow
- Fx.Transitions.Expo
- Fx.Transitions.Circ

- Fx.Transitions.Sine
- Fx.Transitions.Back
- Fx.Transitions.Bounce
- Fx.Transitions.Elastic
- Fx.Transitions.Quad
- Fx.Transitions.Cubic
- Fx.Transitions.Quart
- Fx.Transitions.Quint

在所有这些键下面都有一个函数，但作为属性附加到这些函数的通常又是一些名为“easeIn”、“easeOut”和“easeInOut”的函数。这些函数均作为基础转换函数的修饰符，可以不经修改地应用，或者反过来应用，又或者沿着镜像方向应用。

上述代码的最终结果是产生一个名称散列(用于下拉式选择框)，这些名称映射到动画中使用的实际函数。下面的代码使用这些键构建选择框的内容：

```
// Build selector options from the collected transitions.
$$('#ex_transitions .transitions').adopt(
    transitions.getKeys().map(function(name) {
        return document.newElement('option', {
            'text': name, 'value': name
        });
    })
);
```

对于从转换列表中查找到的每一个键，都会创建一个新的<option>元素并注入到下拉式选择框中。由于这里组合使用了元素.adopt()方法以及数组.map()方法，因此这段代码相当简洁。

最后，下面的代码将“animate”按钮连接到一个处理程序，该处理程序将引发一段使用当前选中的转换函数的动画：

```
// Wire up the button to animate using the selected transition.
$$('#ex_transitions .animate').addEvent('click', function(ev) {
    ev.stop();

    var t_name = $$('#ex_transitions .transitions').get('value')[0];
    var trans  = transitions.get(t_name);

    $$('#ex_transitions .mover').set('tween', {
        transition: trans,
        duration: 500
    }).tween('left', '0px', '500px');
});
```

除了获取选择框的当前值并将其映射到可用转换函数之外，上面的代码与在本章中目前已经看到的 Fx.Tween 示例并无多大差异。但一旦将这些代码片段组合起来，就应该能够试验多种不同的转换和缓动类型，以了解每种函数对动画显示的运动类型会产生什么影响。

图 34-3 给出了所有上述标记和代码的最终运行结果，其中列出了许多可用的转换函数。

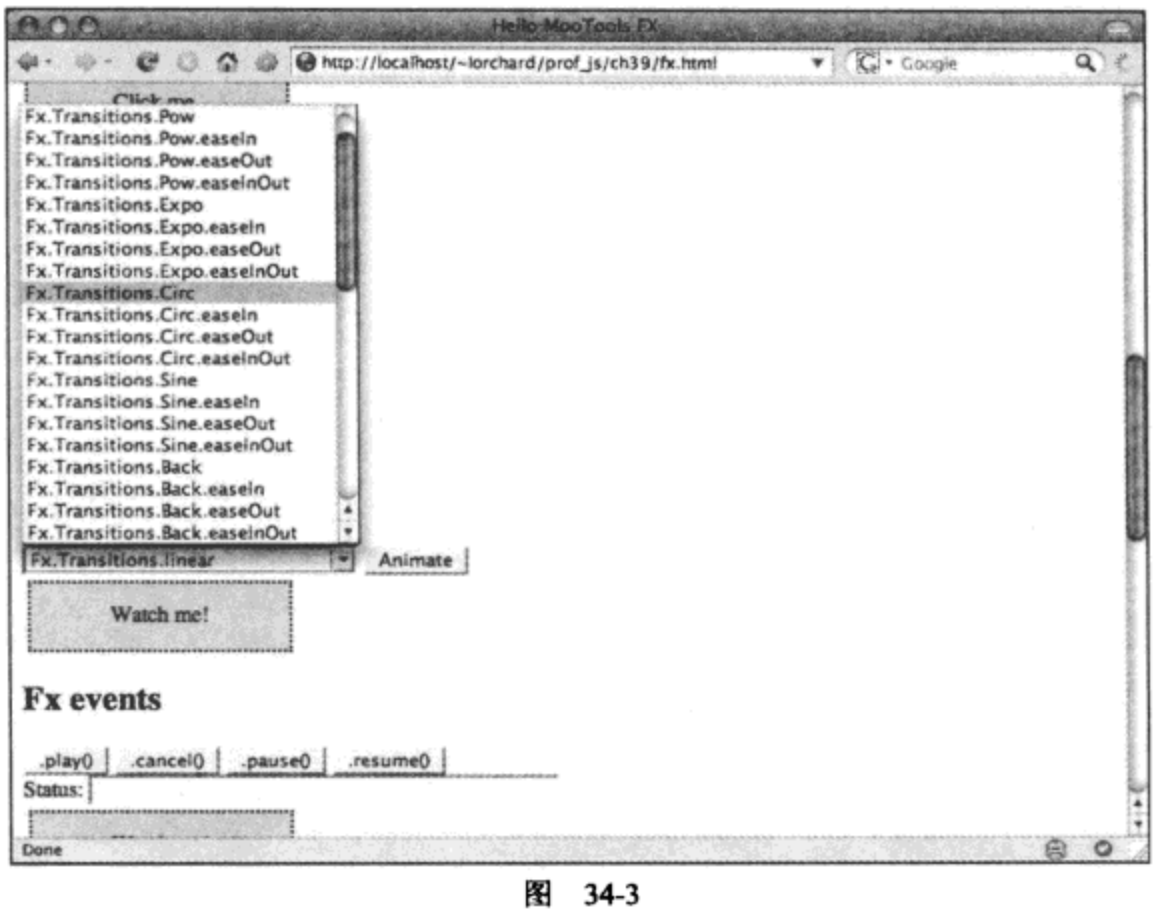


图　34-3

使用字符串表达转换选择

对于“transition”选项，还有一种似乎比较有趣且可能用起来比较方便的技巧，而且该技巧目前在本章中至少已经使用了一次：

```
var tweener = new Fx.Tween(ev.target, {
    duration:   1000,
    transition: 'elastic:inOut'
});
```

注意，这里并没有使用实际函数的引用，而是使用一个字符串标识转换。可以将这个字符串的格式描述成如下：

```
{transition name}[:in][:out]
```

因此，这里并没有直接使用 Fx.Transitions 键的引用，而是仅使用该名称，并且可以选择在其后面附加一个冒号和 in、out 或 inOut 中的一个值来使用某个缓动修饰符。

34.1.7　研究动画事件

虽然可以通过.chain()方法在动画结束时执行注册的代码，但是基于 Fx 类的动画还提供了几个事件可供通过.addEvent()方法使用，这一点得益于第 31 章中介绍的 Events 混入类。此外，Fx 动画还提供了几个方法可供调用以控制正在运行的动画。

下面的标记是一个交互式示例的基础。

```
<div id="ex_events">
    <h2>Fx events</h2>
```

```
    <form>
        <button class="start">.start()</button>
        <button class="cancel">.cancel()</button>
        <button class="pause">.pause()</button>
        <button class="resume">.resume()</button>

        <br style="clear: both" />

        <label for="status">Status:</label>
        <input class="status" type="text" size="40" />
    </form>

    <div class="mover">
        <div>Watch me!</div>
    </div>

    <br style="clear: both" />

</div>
```

可以在图 34-4 中看到这段标记的呈现结果。根据按钮的名称可以猜到 Fx 动画提供了如下的方法：

- .start() 启动动画。
- .stop() 停止动画。
- .pause() 暂停动画。
- .resume() 恢复暂停的动画。

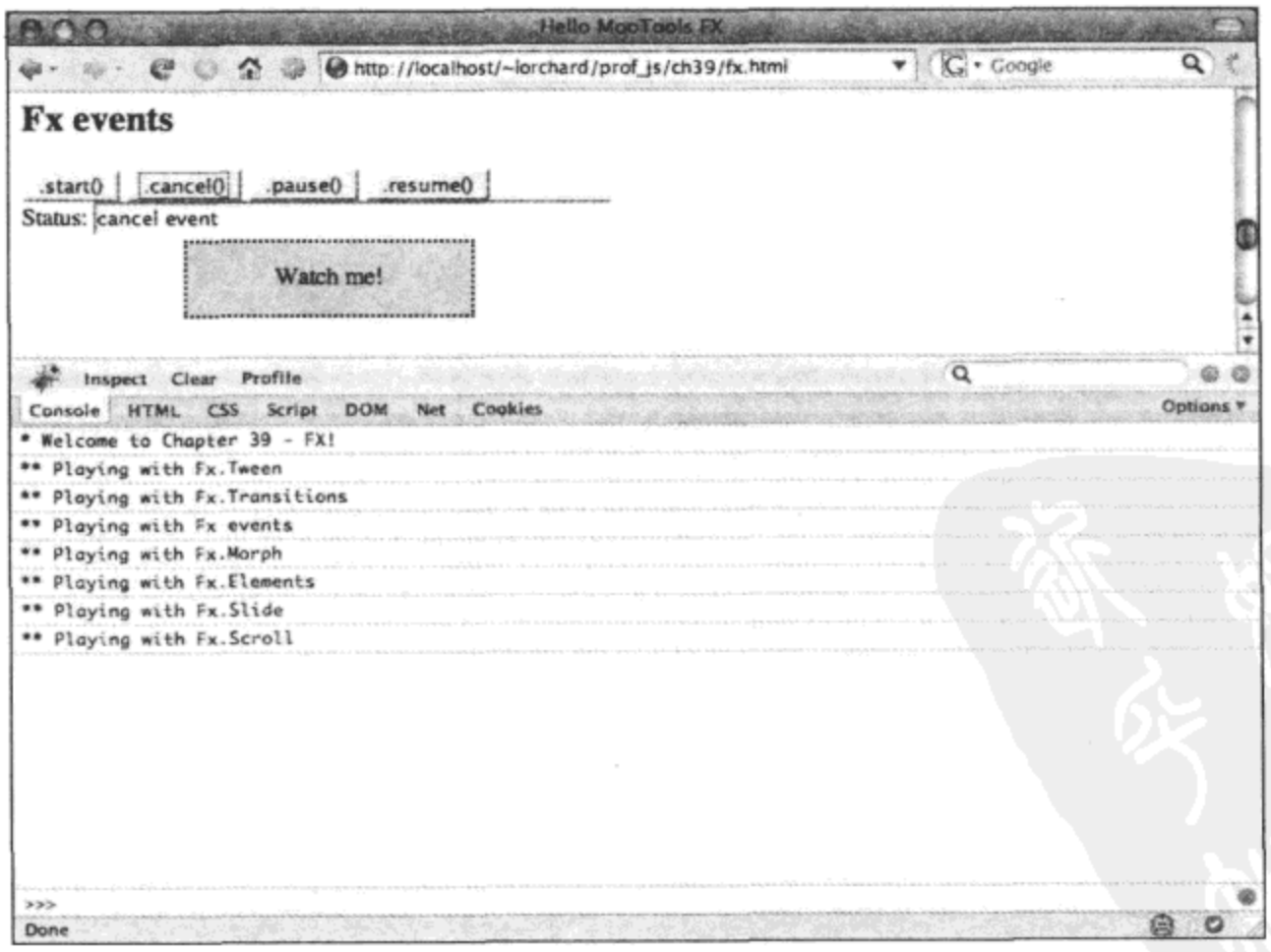


图 34-4

接下来，下面的代码将该动画设置成一个实验主体：

```
var anim = $$('#ex_events .mover')[0]
    .set('tween', {
        duration:  2000,
        transition: Fx.Transitions.linear
    })
    .get('tween');
```

在上面的代码中，我们为动画指定了一个比通常更长的持续时间，这样就有足够的时间来进行试验；而且将转换设置为极为简单的“linear”变换，让其简单地“爬过”屏幕即可。

在动画对象就绪之后，下面就要将标记中的按钮连接起来以向动画发出命令：

```
$$('#ex_events .start)
    .addEvent('click', function(ev) {
        ev.stop();
        anim.start('left', '0px', '500px');
    });

$$('#ex_events .cancel')
    .addEvent('click', function(ev) {
        ev.stop();
        anim.cancel();
    });

$$('#ex_events .pause')
    .addEvent('click', function(ev) {
        ev.stop();
        anim.pause();
    });

$$('#ex_events .resume')
    .addEvent('click', function(ev) {
        ev.stop();
        anim.resume();
    });
```

上面的代码相当简单：每个按钮都有一个单击事件处理程序，用来调用动画对象提供的不同方法，首先是启动该动画(从左向右移动)的.start()方法。

现在，将一些事件处理程序连接起来以监控该动画：

```
var status = $$('#ex_events .status')[0];
anim.addEvents({
    start: function() {
        status.set('value', 'start event');
    },
    complete: function() {
        status.set('value', 'complete event');
    },
```

```
    chainComplete: function() {
        status.set('value', 'chainComplete event');
    }
    cancel: function() {
        status.set('value', 'cancel event');
    }
});
```

正如上面的代码所示，在 MooTools 中，基于 Fx 的动画在执行过程中会引发一个或多个如下事件：

- **start** 在动画启动时引发。
- **complete** 在动画完全结束时引发。
- **chainComplete** 当动画链完全结束时引发。
- **cancel** 当动画提前终止时引发。

在上面的代码中，这里的每个事件都有一个对应的处理程序，它们均会使用一条关于最近引发事件的消息来更新具有 CSS 类“status”的文本输入字段的内容。您应该可以看到，当单击按钮以启动和停止动画时以及在动画运行的过程中，该消息会适当地发生变化。可以利用附加到这些事件上的处理程序来协调应用程序其他部分的执行，包括更多动画或者其他功能(例如引发 AJAX 请求)。

34.1.8 利用 Fx.Morph 对多个属性制作动画

Fx.Tween 能够对元素上的单个 CSS 属性制作动画，而 Fx.Morph 模块将该功能扩展，从而实现对多个属性制作动画。作为另一个 Fx.CSS 子类，Fx.Morph 的用法与 Fx.Tween 非常类似。

下面的标记为后续几个示例做好准备：

```
<div id="ex_morph">
    <h2>Fx.Morph</h2>
</div>
```

然后，下面这个函数调用为动画创建 元素：

```
createMoverDivs('ex_morph', 2, 'morph_mover');
```

现在尝试使用 Fx.Morph 类，如下所示：

```
$('morph_mover1').set({
    text: 'Click me, new Fx.Morph()',
    events: {
        click: function(ev) {
            var morph = new Fx.Morph(ev.target, {
                duration: 1000
            });

            morph
                .start({
```

```
                left:  [ '0px', '300px' ],
                width: [ '150px', '300px' ],
                backgroundColor: [ '#ddf', '#dfd' ]
            })
            .chain(function() {
                morph.start({
                    left:  [ '300px', '0px' ],
                    width: [ '300px', '150px' ],
                    backgroundColor: [ '#dfd', '#ddf' ]
                })
            });

        }
    }
});
```

在使用 Fx.Tween 类时，.start()方法有 3 个参数：CSS 属性的名称以及该属性的起始值和结束值。这个方法在 Fx.Morph 中得到扩展，它使用一个散列或对象作为其唯一的参数。这个散列的键指定了 CSS 属性，而它的值应该是一个由两个元素组成的数组，分别是预期的起始值和结束值。

在上面的代码示例中，注册的单击事件处理程序会让指定的元素同时完成以下几项工作：向左移动、慢慢增大并缓慢地变成绿色。然后，执行一段链接动画，使该元素恢复到原始的位置、大小和颜色。

与 Fx.Tween 类似，Fx.Morph 类也可以与扩展元素属性和方法结合使用，如下所示：

```
$('morph_mover2').set({
    text: 'Click me, element.morph()',
    morph: {
        duration: 1000,
        link:     'chain'
    },
    events: {
        click: function(ev) {
            ev.target
                .morph({
                    left:  [ '0px', '300px' ],
                    width: [ '150px', '300px' ],
                    backgroundColor: [ '#ddf', '#dfd' ]
                })
                .morph({
                    left:  [ '300px', '0px' ],
                    width: [ '300px', '150px' ],
                    backgroundColor: [ '#dfd', '#ddf' ]
                });
        }
    }
});
```

可以使用“morph”元素属性设置选项，这会隐式地创建一个存放在元素存储中进行管理的Fx.Morph 实例。然后，可以调用元素方法.morph()，将其实参设为一个由指定启动该动画时所使用的属性和值组成的散列或对象。与 Fx.Tween 使用的方式相同，可以使用通过“morph”属性获取的对象将后续的动画链接进来。

Fx.Morph 类一次性跟踪多个属性的变化，从而提供了 Fx.Tween 的有用升级，使得我们可以编排更详细的动画。图 34-5 给出了前面讲解的动画的运行情况。

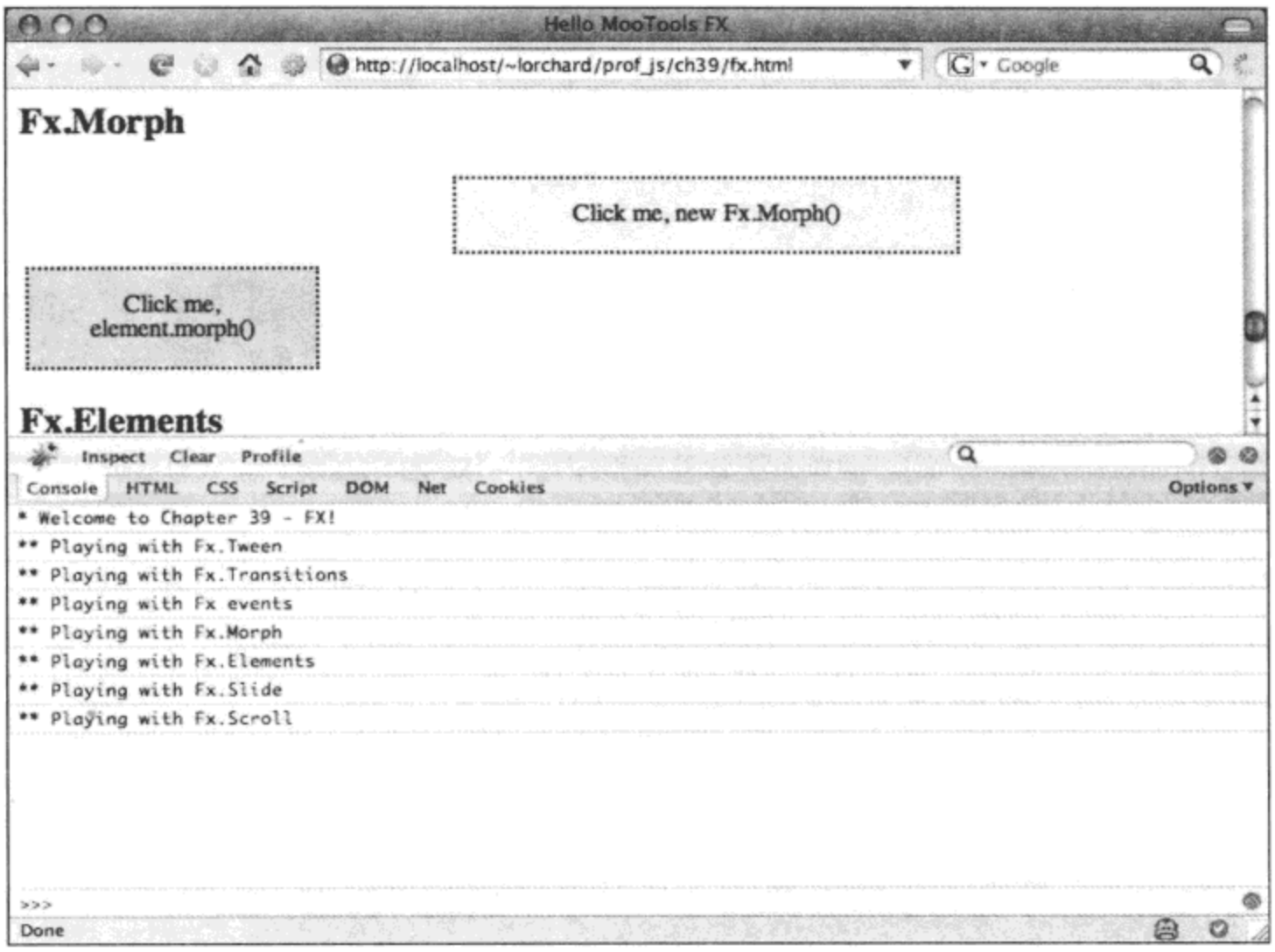


图 34-5

34.1.9 利用 Fx.Elements 对多个元素制作动画

Fx.Elements 类比 Fx.Morph 更进一步，它可用来编排会一次性影响多个元素上的多个 CSS 属性的动画。

下面的标记为后面的代码建立测试平台：

```
<div id="ex_elements">
    <h2>Fx.Elements</h2>
</div>
```

下面的函数调用创建了动画中要使用的多个 元素：

```
createMoverDivs('ex_elements', 4, 'elements_mover');
```

现在代码都已经准备就绪，可以尝试使用 Fx.Elements 类：

```
$$('#ex_elements .mover').addEvent('click', function(ev) {

    var fx_ele = new Fx.Elements('#ex_elements .mover', {
        duration:   750,
        transition: Fx.Transitions.Bounce.easeOut
    });

    fx_ele.start({
        '0': {
            left:  [ '0px', '300px' ],
        },
        '1': {
            width: [ '150px', '300px' ],
        },
        '2': {
            backgroundColor: [ '#ddf', '#dfd' ]
        },
        '3': {
            left:  [ '0px', '300px' ],
            width: [ '150px', '300px' ],
            backgroundColor: [ '#ddf', '#dfd' ]
        }
    });
});
```

上面的代码在所有已创建的 4 个 元素上设置单击事件处理程序，这样当单击它们中的任何一个时，所有这 4 个元素都会同时以不同的方式运行动画。其中，第一个元素向左移动，第二个元素不断增大，第三个元素变换颜色，而第四个元素则同时进行上面几种转换。

效果非常引人注目，但它的创建却相对简单。Fx.Elements 实例的创建方式有如下几种，既可以使用一个元素列表(例如$$()函数的返回值)，也可以运用一个 CSS 选择器字符串来生成元素列表。而且作为另一个 Fx.CSS 子类，该构造函数的第二个参数的选项与目前给出的其他 Fx 类一样。

预期动画的描述也非常简单：Fx.Elements 实例的.start()方法的参数是一个散列或对象，它的键根据构造时提供的元素列表中的数字索引推导而来。这些键又都指向用来描述一个或多个需要制作动画的 CSS 属性的散列或对象，就像 Fx.Morph 类一样。

还可以使用 Fx.Elements 链接动画，订阅动画生命周期事件，以及调用前面介绍的方法来暂停或停止动画。借助 MooTools 提供的继承系统，非常容易在一组现有约定之上构建另一组约定，从而实现与 Fx.Elements 类提供的动画同样强大和复杂的功能。

图 34-6 给出了上面示例中的动画接近完成时的情况。

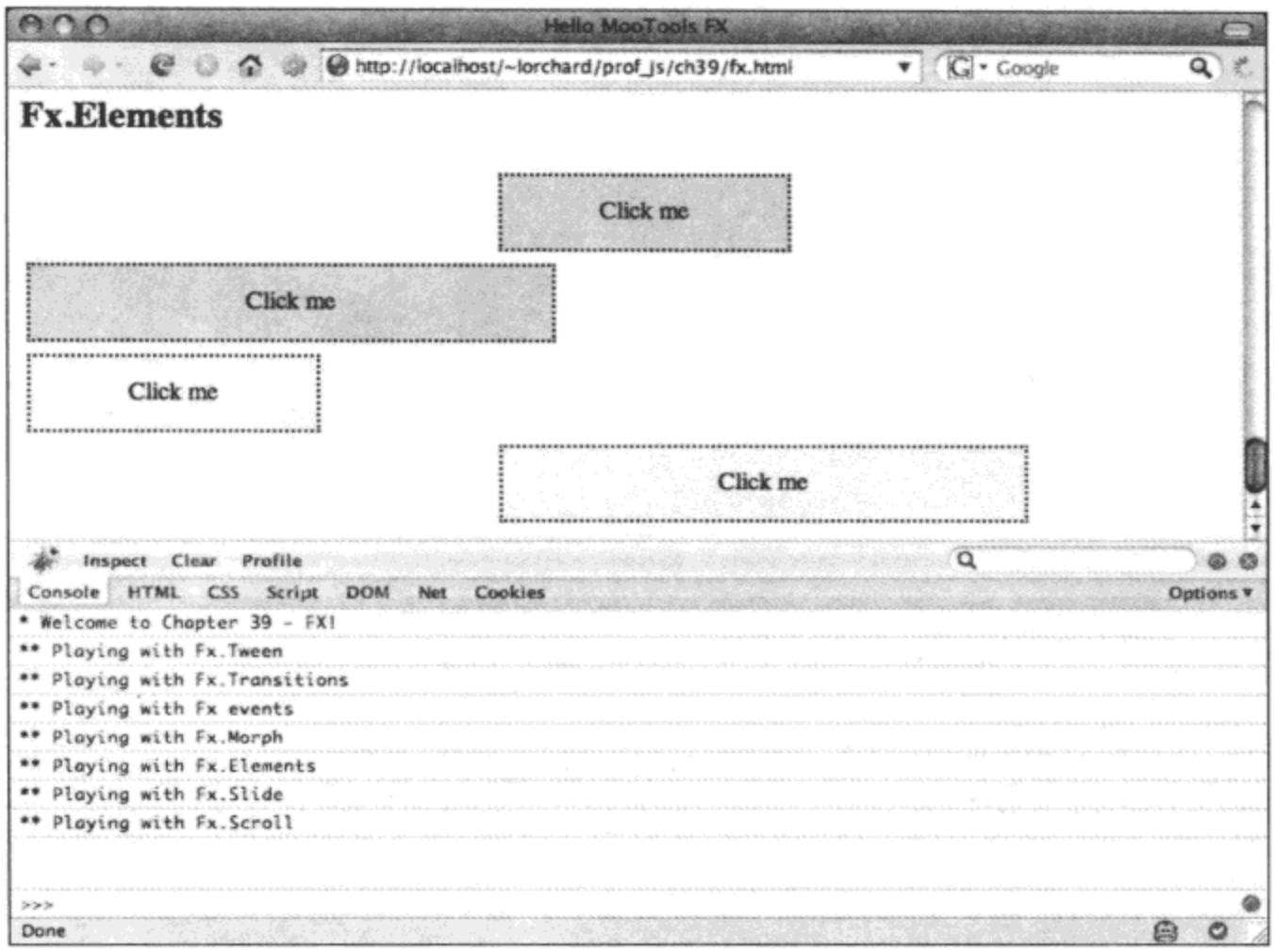


图 34-6

34.2 使用用户界面窗口部件

有些框架(例如 Ext JS 和 Dojo)提供了丰富且极其灵活的用户界面组件或窗口部件库。这些窗口部件中有一些已经相当成熟和强大，足以替换原生浏览器控件，并方便用来构造一些非常接近于完整桌面应用程序的界面。

尽管 MooTools 并没有像这些框架一样走得这么远，但它确实提供了少量非常方便的一组用户界面窗口部件来封装与模式和控件的便利交互。如果您并不完全打算构建一个桌面应用程序，那么 MooTools 提供的功能或许正好能够满足您的需求。

34.2.1 构建折叠布局

MooTools 的折叠布局窗口部件为我们在一个有限的垂直空间内显示大量信息提供了一种解决方案，允许用户通过单击头部展开相关部分并关闭所有其他部分来导航内容。这个窗口部件一次只允许显示一块内容，而剩余内容只显示头部，因此将呈现窗口部件所需的空间降到最低。

下面的 CSS 和 HTML 标记构造了几个可视的内容块，为后面的演示做好准备：

```
<style type="text/css">
    #accordion { width: 33%; }

    p { margin: 0; padding: 0; }
```

```
    dl.has_items dt {
        font-weight: bold;
        background: #ccc; margin: 0.25em; padding: 0.5em;
    }

    dl.has_items dd {
        background: #eee; margin: 0.25em; padding: 0.5em;
    }
</style>

<div id="ex_accordion">

    <h2>Accordion</h2>

    <dl id="accordion" class="has_items">

        <dt id="acc_item1">First Item</dt>

        <dd id="acc_content1"><p>
            Lorem ipsum dolor sit amet, consectetuer adipiscing
            elit. Cras imperdiet velit at arcu. Nullam viverra lectus ac
            felis.
        </p></dd>

        <dt id="acc_item2">Second Item</dt>

        <dd id="acc_content2"><p>
            Cum sociis natoque penatibus et magnis dis parturient
            montes, nascetur ridiculus mus.
        </p></dd>

        <dt id="acc_item3">Third Item</dt>

        <dd id="acc_content3"><p>
            Nullam eu sapien dictum purus consequat cursus. Etiam non
            sapien sed elit dignissim tristique.
        </p></dd>

    </dl>

</div>
```

上面的标记构建了一个又细又高的内容块，它有 3 个部分，每个部分包含一个头部标题。在图 34-7 中可以看出，该内容占据了浏览器窗口高度的绝大部分。如果这里有更多的内容，那么就需要向下滚动页面才能看到全部内容。

图　34-7

虽然这并不能解决所有问题，但是将所有标题头部放在同一个页面上至少让用户知道有哪些内容可以进一步精读，这或许更为有趣。这就是折叠布局派上用场的地方。

在做了大量的铺垫之后，接下来查看 Accordion 类的实际用法：

```
var accordion = new Accordion(
    $$('#ex_accordion dt'), $$('#ex_accordion dd'), {
        duration: 500,
        transition: Fx.Transitions.Bounce.easeOut,
        display: 1
    }
);
```

Accordion 类的构造函数接受 3 个参数：

- 用作头部的元素列表。
- 用作与头部关联的内容的元素列表。
- Fx.Elements 类支持的动画选项，同时还有其他几个选项。

Accordion 支持的额外选项包括如下：

- display　首先要显示的元素的索引(默认为 0)，带有视觉转换。
- show　首先要显示的元素的索引(默认为 0)，没有视觉转换。

- height　如果设为 true，那么对显示或隐藏部分的高度制作动画。默认为 true。
- width　如果设为 true，那么对显示或隐藏部分的宽度制作动画。默认为 false。
- opacity　如果设为 true，那么不透明度属性也将参与到隐藏和显示部分的过程中。默认为 true。
- fixedHeight　窗口部件的整体高度将保持不变。默认为 false。
- fixedWidth　窗口部件的整体宽度将保持不变。默认为 false。
- alwaysHide　如果设为 true，那么这个唯一显示的部分也可以关闭。默认为 false。

利用这些选项的不同组合以及我们自己的 CSS 样式，可以定制这个窗口部件的初始状态、它运行动画的方向，并调整在执行动画过程中使用的效果。与其他动画相似，Accordion 窗口部件也支持几个事件，我们可以附加自己的监听程序来响应用户交互：

```
accordion.addEvents({
    start: function() {
        $log('starting animation');
        $log(arguments);
    },
    complete: function() {
        $log('completed animation');
        $log(arguments);
    },
    cancel: function() {
        $log('canceled animation');
        $log(arguments);
    },
    active: function(header, content) {
        $log("showing " + header.get('text'));
    },
    background: function(header, content) {
        $log("hiding " + header.get('text'));
    }
});
```

除了 Fx.Elements 提供的事件集之外，下面这对事件是 Accordion 类特有的事件：

- active　当单击头部元素且对应的元素内容可见时引发该事件。
- background　对于每个未单击的头部元素以及隐藏的内容元素，都会引发这个事件。

当动画转换开始之前就会调用这些事件，因此我们可以完成一些有趣的操作，例如在即将显示该部分之前动态更新它的内容。

图 34-8 给出了 Accordion 对内容元素大小的影响，同时还给出了在 Firebug 中的运行示例，日志文本消息显示了在该窗口部件上单击几次之后的结果。

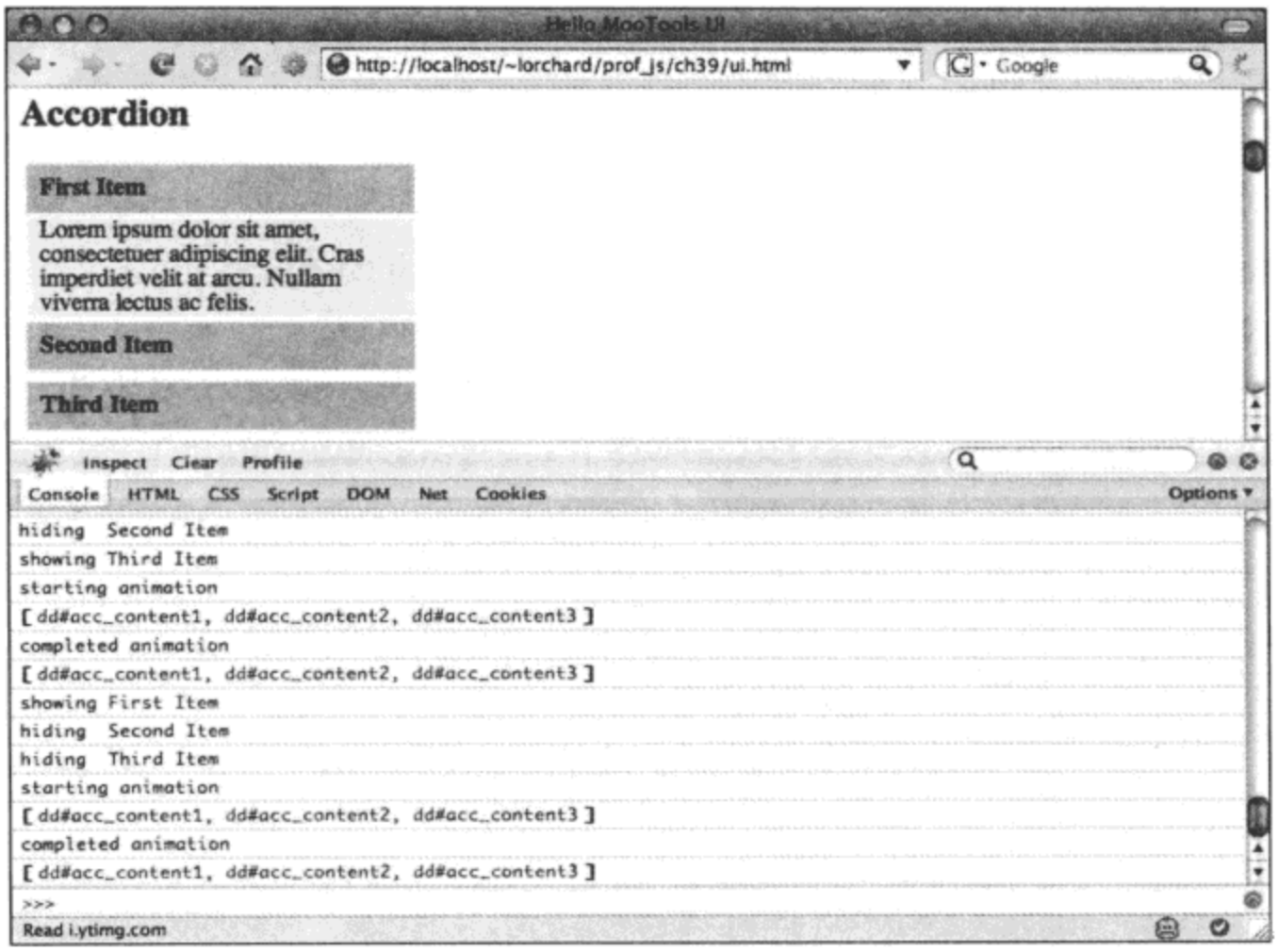


图 34-8

34.2.2 向页面导航中添加平滑滚动

下一个窗口部件并不是类似于过渡性装饰物那样的布局实用工具。下面的标记定义了带有内部锚点链接的内容：

```
<div id="ex_smoothscroll">
    <h2>SmoothScroll</h2>
    <ul>
        <li><a href="#scroll1">Scroll to #1</a></li>
        <li><a href="#scroll2">Scroll to #2</a></li>
        <li><a href="#scroll3">Scroll to #3</a></li>
    </ul>
</div>

<div id="scroll1">
    <h3>Scroll link destination #1</h3>
    <p>
        Cum sociis natoque penatibus et magnis dis parturient
        montes, nascetur ridiculus mus.
    </p>
</div>

<div id="scroll2">
    <h3>Scroll link destination #2</h3>
    <p>
```

```
        Lorem ipsum dolor sit amet, consectetuer adipiscing
        elit. Cras imperdiet velit at arcu. Nullam viverra lectus ac
        felis.
    </p>
</div>

<div id="scroll3">
    <h3>Scroll link destination #3</h3>
    <p>
        Nullam eu sapien dictum purus consequat cursus. Etiam non
        sapien sed elit dignissim tristique.
    </p>
</div>
```

实际上，上面的内容最有趣的地方在于，如果将链接的 元素散布在其他内容中，就会为垂直滚动获得一些空间。这样做会让如下窗口部件的使用更加引人注目，下面的代码实例化这个窗口部件：

```
var smooth = new SmoothScroll({
    links:      $$('#ex_smoothscroll a'),
    duration:   1000,
    transition: Fx.Transitions.Bounce.easeOut
});

['start', 'cancel', 'complete'].each(function(ev_name) {
    smooth.addEvent(ev_name, function(ev) {
        $log('SmoothScroll fired event ' + ev_name);
    })
});
```

SmoothScroll 窗口部件在其构造函数中接受 Fx 选项，还有一个额外的名为 links 的选项。这个额外选项是一个由可单击链接组成的列表，每个链接都会有一个特殊的单击事件处理程序。

当单击其中一个链接时(而不是只靠近链接内容)，就会启动一段动画来滚动窗口，直到所需的元素变得可见。由于支持转换函数，因此上面的代码使得这种滚动更加有趣：让窗口在用户导航的每部分内容上弹跳。

这个窗口部件为大型内容页面增添了一点视觉效果，与此同时还能让用户知道发送的内部链接在页面中的位置。

34.2.3　启用可拖动元素

在现代 JavaScript 框架用户界面中，一项预期的基本功能是以某种方式启用可拖动元素。下面的标记定义了这样的一个元素：

```
<style text="type/css">
    .mover .handle {
        background: #eee;
        padding: 0.5em;
        border: 1px dotted #333;
```

```
    }
</style>
<div id="ex_drag">

    <h2>Drag</h2>

    <div class="mover dragger">
        <div class="handle drag">Drag this handle!</div>
        <p>This is inert content.</p>
        <div class="handle toggle">Enabled</div>
    </div>

    <br style="clear: both" />

</div>
```

这段标记(图 34-9 给出了它的呈现结果)构造了一个带有手柄的可拖动方框，其中有一些文本内容以及一个用来启用和禁用拖动的方框。启用拖动是一件非常容易的事情：

```
var dragger = new Drag( $$('#ex_drag .dragger')[0], {
    handle: $$('#ex_drag .mover .drag')[0],
    snap: 25,
    grid: 25
});
```

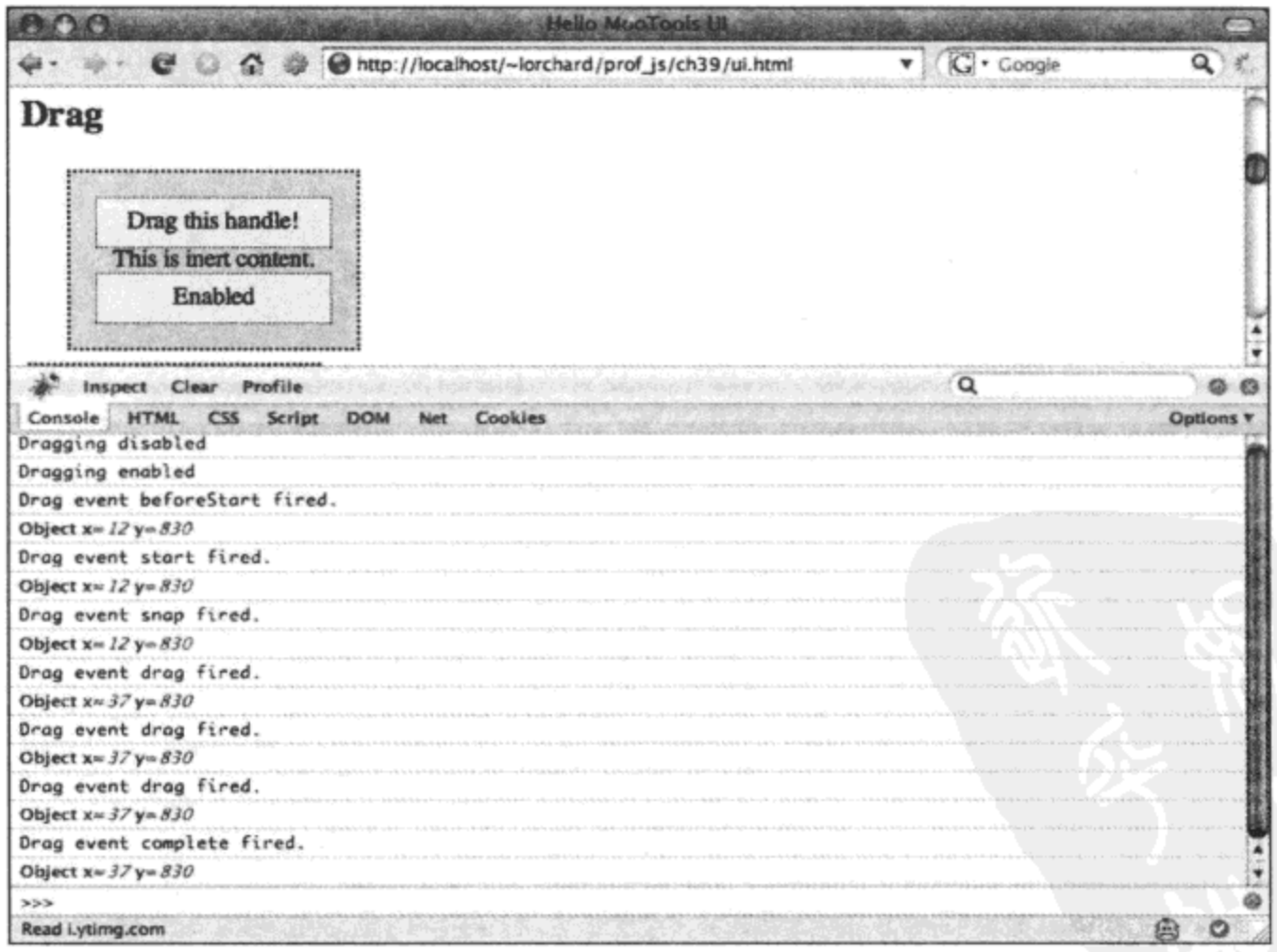


图 34-9

Drag 对象构造函数的第一个参数是一个指向拖动操作移动的元素的引用，而第二个参数是一个由下列选项组成的对象或散列：

- handle　指向将要接受拖动手势的元素的引用。默认为传给构造函数的第一个元素。
- grid　利用指定的间距(单位为像素)形成规则的位置网格，可以将元素拖动到这些位置上。默认为 false。
- snap　在元素开始响应拖动手势之前必须将其拖动指定的距离(单位为像素)。默认为 6。
- limit　一个用来定义拖动限制的 x 和 y 值的对象或散列。
- modifiers　一个用来指定由拖动操作的 x 和 y 坐标修改的 CSS 属性的对象。默认为{x: 'left', y: 'top'}，这直接控制该元素的位置。
- style　如果该选项为 true(默认值)，那么使用指定的修饰符更新元素的样式属性。如果为 false，就会更新元素属性。
- invert　如果该选项为 true，那么倒置拖动操作报告的 x 和 y 值。默认为 false。
- unit　附加到报告值末尾的 CSS 单位。默认为 px。

如果只是使用 Drag 对象的默认值，那么可以将给定元素自由地拖动到页面的任何位置。但通过调整这些选项，可以以各种有趣的方式来改变这种行为。

如同前一个示例所演示的那样，可以只使用一个手柄元素(例如一个标题栏或其他子元素)让一个元素可拖动。可以将拖动限定到指定的区域并锁定到指定的位置网格中。利用 modifiers 和 style 选项可以彻底修改拖动时表现的行为。

Drag 对象还提供了一对方法.detach()和.attach()来分别禁用和启用该对象的鼠标事件监听程序：

```
$$('#ex_drag .mover .toggle').addEvent('click', function(ev) {
    if (/Enabled/.test(ev.target.get('text'))) {
        dragger.detach();
        $log('Dragging disabled');
        ev.target.set('text', 'Disabled');
    } else {
        dragger.attach();
        $log('Dragging enabled');
        ev.target.set('text', 'Enabled');
    }
});
```

上面的代码将前面标记中提供的 元素连接到一个事件处理程序，以交替地禁用和启用拖动功能。可以使用这两个方法有条件地(根据所需的任何条件)允许或禁止拖动，而完全不需要创建或销毁 Drag 对象。

最后，在建立一个满足需要的 Drag 窗口部件之后，可以附加事件监听程序来响应与该窗口部件交互过程中所处的不同阶段：

```
[ 'beforeStart', 'start', 'snap', 'drag', 'complete' ]
    .each(function(ev_name) {
```

```
        dragger.addEvent(ev_name, function(drag_el) {
            $log("Drag event " + ev_name + " fired. ");
            $log( drag_el.getPosition() );
        })
    });
```

正如上面的代码所演示的那样，Drag 窗口部件支持如下的事件：

- **beforeStart** 当用户在该元素上方按下鼠标按键时就会引发该事件(但是拖动手势启动之前引发)。
- **start** 一旦用户启动拖动手势，就引发该事件。
- **snap** 当用户已经拖动足够远的距离以满足 snap 选项的距离要求时，就会引发该事件。
- **drag** 每个拖动该元素的鼠标事件都会引发该事件。
- **complete** 一旦用户在启动拖动手势之后释放鼠标按键，就会引发该事件。

对于所有这些事件的处理程序，都会为其提供作为 Drag 对象构造函数第一个参数传入的元素。通过响应这些事件组合，可以监听默认的元素拖动行为，也可以完全实现自己的拖动行为。

让元素可调整大小

为了定制拖动行为，MooTools 提供了一个使用 Drag 对象的元素方法。通过调用.makeResizable()方法，可以让一个元素在拖动时增大或缩小，而不是在页面上四处移动。

为了进行演示，下面的标记声明了一个通过拖动调整大小的元素：

```
<div class="mover resizer">
    <div class="handle drag">Drag to resize me</div>
</div>
<br style="clear: both" />
```

现在，查看如何让这个元素可调整大小：

```
$$('#ex_drag .resizer').makeResizable({
    handle: $$('#ex_drag .resizer .handle')[0],
    snap: 25,
    grid: 25,

    onComplete: function(el) {
        $log("Element resized!");
        $log( el.getSize() );
    }
});
```

元素方法.makeResizable()接受的唯一参数是一个由选项组成的散列或对象，与 Drag 对象接受的选项一致，包括 handle、snap、grid 和 onComplete。前面定义的 onComplete 事件处理程序报告所有重新调整大小拖动手势的完成情况，并在日志文本消息中显示最终的元素大小。

在图 34-10 中可以看到这段代码运行时的快照。

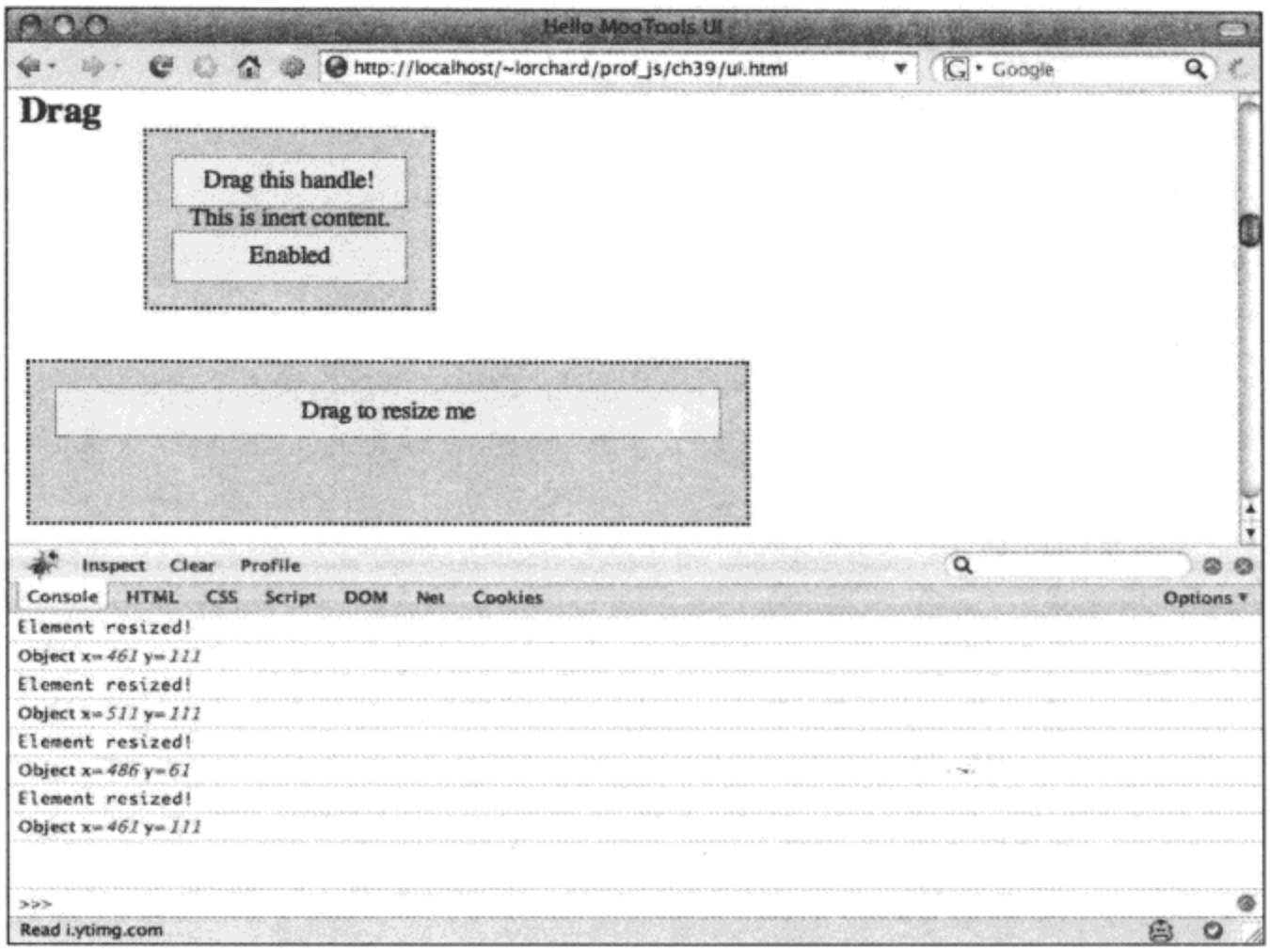


图　34-10

34.2.4　自动滚动窗口和元素

在前面的示例中您或许已经注意到，如果将元素拖向窗口的边缘，而且希望将它移出页面当前可见部分的边界之外，那么就需要停放该元素并进行滚动。许多应用程序通过在鼠标手势到达距离窗口边缘足够近的距离时自动滚动视图来避免这种情况。

在 MooTools 中，可以结合使用 Scroller 类和 Drag 类来实现该功能。下面构造了一个可拖动元素：

```
<div id="ex_scroller">

    <h2>Scroller</h2>

    <div class="mover dragger">
        <div class="handle drag">Drag this handle!</div>
        <p>This is inert content.</p>
    </div>

    <br style="clear: both" />

</div>
```

现在，在创建一个 Drag 实例以使这个元素可移动之前，查看下面的代码：

```
var my_scroller = new Scroller(window, {
```

```
        area:    Math.round(window.getWidth() / 10),
        velocity: 1,
        /*
        onChange: function(x, y){
            this.element.scrollTo(x, y);
        }
        */
    });
```

Scroller 构造函数的第一个参数是一个 DOM 元素，在这里是窗口本身的引用。这里还可以使用指向其他可滚动元素的引用，例如一个将样式属性 overflow 设为“auto”的包含大量内容的 元素。

Scroller 构造函数的第二个参数应该是一个由多个选项组成的散列或对象，可包括如下选项：

- area　当离元素边界的距离小于该值时就会开始自动滚动。在前面的示例中，这个值设为当前窗口大小的宽度的 1/10。
- velocity　滚动位置移动速度的倍数。
- onChange　将用来影响托管元素上的滚动的函数。默认(在这个示例代码中注释的部分)是调用该元素的.scrollTo()方法，但可以通过定义该选项来改变这种行为。

Scroll 类还提供了两个方法.start()和.stop()来控制是否激活自动滚动功能。下面的代码将一个 Drag 对象连接到几个用来调用这些方法的事件处理程序：

```
var dragger = new Drag(
    $$('#ex_scroller .dragger')[0],
    {
        handle: $$('#ex_scroller .mover .drag')[0],
        onBeforeStart: function(drag_el) {
            my_scroller.start();
        },
        onComplete: function(drag_el) {
            my_scroller.stop();
        }
    }
);
```

在上面的代码中，当拖动手势启动时就调用.start()方法来激活自动窗口滚动，然后当用户释放鼠标按键时再次调用.stop()方法以使该功能失效。这可以确保只有当执行拖动手势期间才会出现窗口滚动，在其他情况下则不会出现。

34.2.5 启用拖放目标

在讨论图形用户界面时，“拖动”后面最常用的单词是“停放”，它描述的通常是如下操作：首先让用户界面中的一个对象变成可拖动，使用户能够将其从 A 点拖动并将其停放在 B 点或 C 点，然后根据用户的选择执行一些有趣的操作。

在 MooTools 中，可以利用 Drag.Move 类实现拖放操作。下面的标记提供了小型拖放界面所需的元素。

```
<style type="text/css">
    #ex_drag_move .container {
        width: 400px;
        padding: 1em;
        margin: 1em;
        border: 2px dotted #333;
    }
</style>
<div id="ex_drag_move">

    <h2>Drag.Move</h2>

    <div class="container">

        <div class="mover drop" id="dest1">Bucket #1</div>
        <div class="mover drop" id="dest2">Bucket #2</div>
        <div class="mover drop" id="dest3">Bucket #3</div>
        <div class="mover drop" id="dest4">Bucket #4</div>
        <br style="clear: both" />

        <div class="mover dragger">Drag me!</div>
        <br style="clear: both" />

    </div>

</div>
```

在上面的标记中，定义了 4 个方框，它们将连接起来作为第五个可拖动方框的停放目标。下面的代码实例化一个 Drag.Move 对象，从而实现了期望的功能：

```
var dragger = new Drag.Move(

    $$('#ex_drag_move .dragger')[0], {

        droppables: $$('#ex_drag_move .drop')

    }

);
```

Drag.Move 构造函数的第一个参数是将要成为可拖动对象的元素，第二个参数是一个由选项组成的对象或散列。作为 Drag 的子类，Drag.Move 类接受 Drag 构造函数的所有选项，包括 snap、grid、limit 和 handle。其他选项可能并没有多大意义，因为 Drag.Move 的功能依赖于它自己的默认值。

Drag.Move 引入的新选项名为 droppables，它是由一系列监控托管元素拖放事件的元素组成的列表。这种对几个新事件的监控非常有用，其用法如下所示：

```
[ 'enter', 'leave', 'drop' ]
    .each(function(ev_name) {
        dragger.addEvent(ev_name, function(drag_el, drop_el) {
```

```
            $log(
                "Drag fired " + ev_name + " event" +
                ( (drop_el) ? ", element " + drop_el.id : '' )
            );
        });
    });
```

虽然 Drag.Move 类支持 Drag 的所有事件，但这里最有趣的事件如下：

- enter 当拖动的元素经过一个 droppables 元素上方时引发该事件。
- leave 当拖动的元素从一个 droppables 元素上方移开时引发该事件。
- drop 当拖动的元素在一个 droppables 元素内释放时引发该事件。

在这 3 种事件之间，Drag.Move 类为我们提供了构建非常复杂而高效的拖放界面所需的一切内容。图 34-11 预览了这段代码在浏览器窗口中运行时的情况以及日志文本消息。

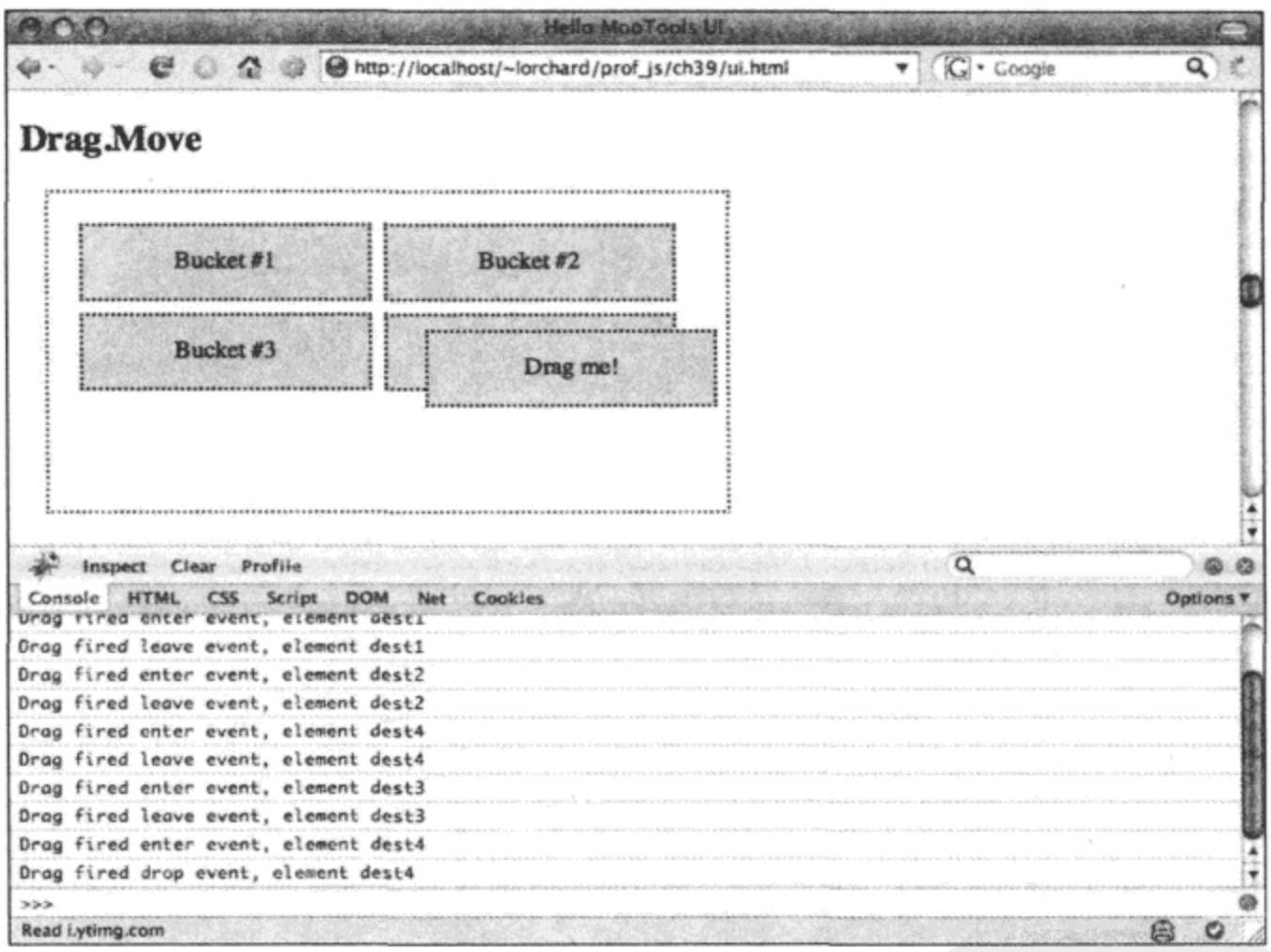


图 34-11

34.2.6 构建可排序列表

可排序列表是拖放的另一种有用实现，允许用户交互地排列列表项。为了帮助构建这些列表，MooTools 提供了 Sortables 类。

考虑下面的标记，它构造了一组内容块：

```
<div id="ex_sortables">

    <h2>Sortables</h2>
```

```
<div id="sortables">

    <dl class="has_items" id="sortable_item1">
        <dt>First Item</dt>
        <dd><p>
            Lorem ipsum dolor sit amet, consectetuer adipiscing
            elit. Cras imperdiet velit at arcu. Nullam viverra lectus ac
            felis.
        </p></dd>
    </dl>

    <dl class="has_items" id="sortable_item2">
        <dt>Second Item</dt>
        <dd><p>
            Cum sociis natoque penatibus et magnis dis parturient
            montes, nascetur ridiculus mus.
        </p></dd>
    </dl>

    <dl class="has_items" id="sortable_item3">
        <dt>Third Item</dt>
        <dd><p>
            Nullam eu sapien dictum purus consequat cursus. Etiam non
            sapien sed elit dignissim tristique.
        </p></dd>
    </dl>

</div>

</div>
```

在上面的标记中，使用字典列表定义内容块。<dt>元素用作标题栏，而<dd>元素存放块内容。Sortables 类并不限定使用这种特定模式的标记，但使用与此类似的模式使得我们很容易利用 CSS 选择器标识不同部分。

现在查看下面的代码示例，它创建了 Sortables 类的一个实例：

```
var sortable = new Sortables( $('sortables'), {

    handle:     'dt',
    clone:      true,
    revert:     true,
    opacity:    0.25,
    constrain:  true

});
```

Sortables 类的构造函数的第一个参数是一个或多个元素，或者是一个能够标识将要作为可排序列表连接的元素的 CSS 选择器字符串。这意味着，可以一次性使用同一个 Sortables 实例来实际

地连接多个列表。在上面的代码中，只有一个元素提供为可排序列表，也就是标记中带有 ID “sortables”的容器 元素。

这个构造函数的第二个参数应该是一个包含选项的散列或对象。这些选项包括如下。

- handle 这个选项可与 CSS 选择器结合使用，以指定每个列表项内的某个子元素作为拖动手柄。
- snap 与 Drag 类相似，这是在开始排序之前必须拖动该项的距离。
- opacity 用来预览当停放时元素将停靠到何处的不透明度值。
- clone 如果该选项为 true，那么拖动元素将被复制并跟随光标直到停放。如果设为 false，那么拖动(重新排列元素)将交换元素而不带任何转换。
- revert 如果 clone 选项设为 true，那么这个选项将确定是否启动一段简短的动画来描绘释放鼠标按键时复制元素移入停靠位置的过程。
- constrain 如果设为 true，那么列表项的拖动运动将被限定到它们的父列表，而不允许在列表之间拖动列表项。

一旦可排序列表准备就绪，那么当用户完成列表项移动之后就要从该列表中得到一条更新消息。为此，Sortables 类提供了一组事件可供监听程序使用，并且提供了一个便利的方法来获取一个反映新排序顺序的元素 ID 列表。

下面的代码将重要事件连接起来并在每个阶段报告这些事件：

```
[ 'start', 'sort', 'complete' ]
.each(function(ev_name) {
    sortable.addEvent(ev_name, function(el_dragged) {
        $log('Sortable fired event ' + ev_name);
        $log( el_dragged );
        $log( sortable.serialize() );
    });
});
```

上面代码中连接的事件包括如下几个：

- start 当用户首先开始拖动元素时引发该事件。
- sort 当拖动元素能够与另一个元素交换位置时引发该事件。
- complete 当用户释放拖动元素时引发该事件。

注意，在这个事件处理程序中调用了 Sortables 对象方法.serialize()，这将返回一个元素 ID 列表(按照它们的当前排列顺序)。当在 start 或 sort 事件处理程序中调用该方法时，其中的一个 ID 可以是 null，以反映出可选的复制元素。

但是，一旦引发 complete 事件，.serialize()方法就应该返回完整的干净列表，这是更新其他数据结构、引发 AJAX 请求以保存修改过的顺序以及执行任何想做的工作的最佳地方。

图 34-12 给出了正在进行的手动排序操作的快照，并且给出了一个日志文本消息样本以显示几条有关引发事件的消息。

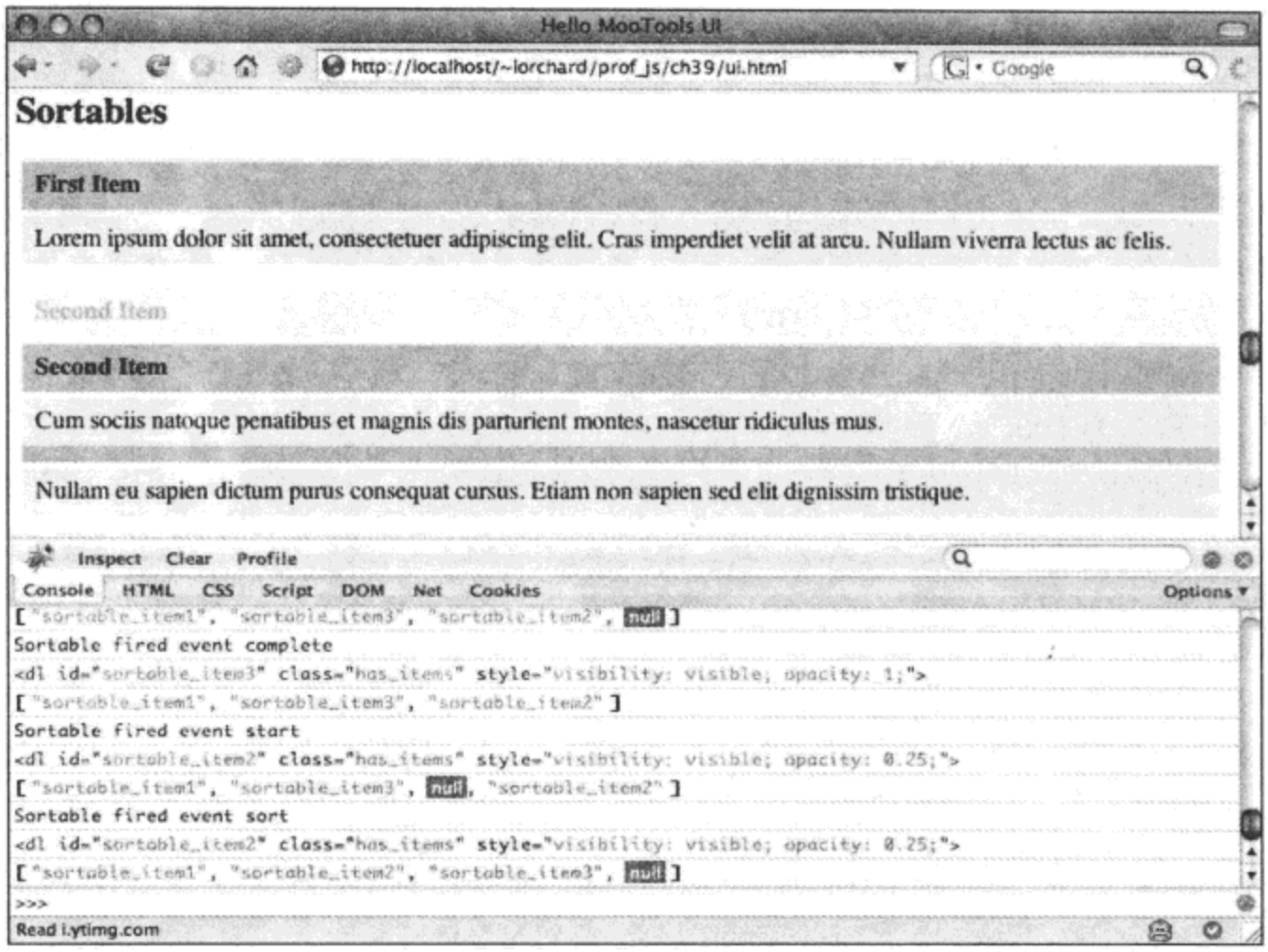


图　34-12

34.2.7　使用工具提示

当用户对用户界面的某些方面感到困惑时，一个自然的举动是在让人感到困惑的元素周围悬停。在这种情况中，工具提示提供了一种在光标附近浮动的临时窗格中显示帮助文本的友好方式。MooTools 提供了 Tips 类来帮助我们不需要编写多少代码就可以构建工具提示。

下面的标记表示一组链接：

```
<div id="ex_tips">

    <h2>Tips</h2>

    <a class="mover" title="Tip #1" href="#" rel="some text">Hover on me!</a>
    <a class="mover" title="Tip #2" href="#" rel="some text">Hover on me!</a>
    <a class="mover" title="Tip #3" href="#" rel="some text">Hover on me!</a>
    <a class="mover" title="Tip #4" href="http://decafbad.com">Hover on me!</a>

    <br style="clear: both" />

</div>
```

现在查看下面这段创建 Tips 类实例的代码：

```
var tips = new Tips( $$('#ex_tips .mover'), {
    className:  'mytool',
    showDelay:  250,
```

```
        hideDelay:  250,
        offsets:    { x: 20, y: 20 },
        fixed:      false
    });
```

Tips 构造函数的第一个参数应该是需要显示工具提示的元素列表。在上面的代码中，这包括标记中声明的所有链接。这个构造函数的第二个参数是一个定义选项的散列或对象，包括如下选项：

- **showDelay** 当用户在元素上悬停该选项指定的时间(单位为毫秒)之后才会显示工具提示。默认为 100。
- **hideDelay** 当光标离开元素之后工具提示仍然在该元素上停留该选项指定的时间(单位为毫秒)。默认为 100。
- **offsets** 一个用来指定工具提示在显示时距离光标的水平距离和垂直距离的对象。Tips 类管理工具提示窗格的显示方向(根据光标在窗口中的位置)。默认为{ x: 20, y: 20 }。
- **fixed** 如果该选项设为 true，那么工具提示将显示在距离光标悬停元素左上角的固定偏移处。否则，工具提示将跟随光标位置。
- **className** 为了实现工具提示窗格，MooTools 创建了一个新的 DOM 元素。这个选项的值将用作该工具提示的容器上的 CSS 类，从而允许对提示及其内容进行样式化。

针对最后一个选项 className，下面的 CSS 代码提供了一个示例来说明如何样式化这个自动注入的工具提示窗格：

```
<style type="text/css">
    #ex_tips a {
        cursor: help;
    }
    .mytool .tip {
        opacity: 0.75;
        background: #333;
        color: #fff;
        z-index: 1000;
        padding: 0.25em;
    }
    .mytool .tip-title {
        font-weight: bold;
        font-size: 13px;
    }
    .mytool .tip-text {
        font-size: 13px;
    }
</style>
```

上面示例中的 CSS 代码为链接自身、整个工具提示窗口、工具提示中的标题以及工具提示中的其他文本提供了样式。

在默认情况下，Tips 类从元素的 title 属性中获得提示的标题，从元素的 rel 或 href 属性中获得提示的文本信息。但是，可以通过编程方式使用元素存储机制为提示指定自己的标题和文本信息，如下所示。

```
$$('#ex_tips .mover')[1]
    .store('tip:title', 'Custom title');

$$('#ex_tips .mover')[2]
    .store('tip:text', 'Custom text here');
```

上面的代码为第二个链接设置提示标题，为第三个链接的工具提示设置文本消息。这可以让我们选择是在标记中定义工具提示内容，还是在代码中设置该内容。

最后，Tips 类提供了几个可以连接处理程序的事件，如下面的代码所示：

```
tips.addEvents({
    show: function(el_tip) {
        var tip_title =
            el_tip.getElement(".tip-title").get("text");
        $log("Tip shown: " + tip_title);
    },
    hide: function(el_tip) {
        var tip_title =
            el_tip.getElement(".tip-title").get("text");
        $log("Tip hidden: " + tip_title);
    }
});
```

尽管上面的代码在创建 Tips 对象之后附加事件监听程序，但完全可以重写这种行为，如下所示：

```
var tips = new Tips( $$('#ex_tips .mover'), {
    className:  'mytool',
    showDelay:  250,
    hideDelay:  250,
    offsets:    { x: 20, y: 20 },
    fixed:      false,

    onShow: function(tip){
        tip.setStyle('visibility', 'visible');
    },

    onHide: function(tip){
        tip.setStyle('visibility', 'hidden');
    }

});
```

如果在构造函数的 show 和 hide 选项中指定事件处理程序，那么这些处理程序将会把默认行为替换，如上面的代码所示。因此，我们可以完全重写默认行为，也可以在后期附加新的监听程序来响应工具提示事件。

图 34-13 给出了工具提示的实际运行情况以及前面定义的事件处理程序所生成的日志文本消息。

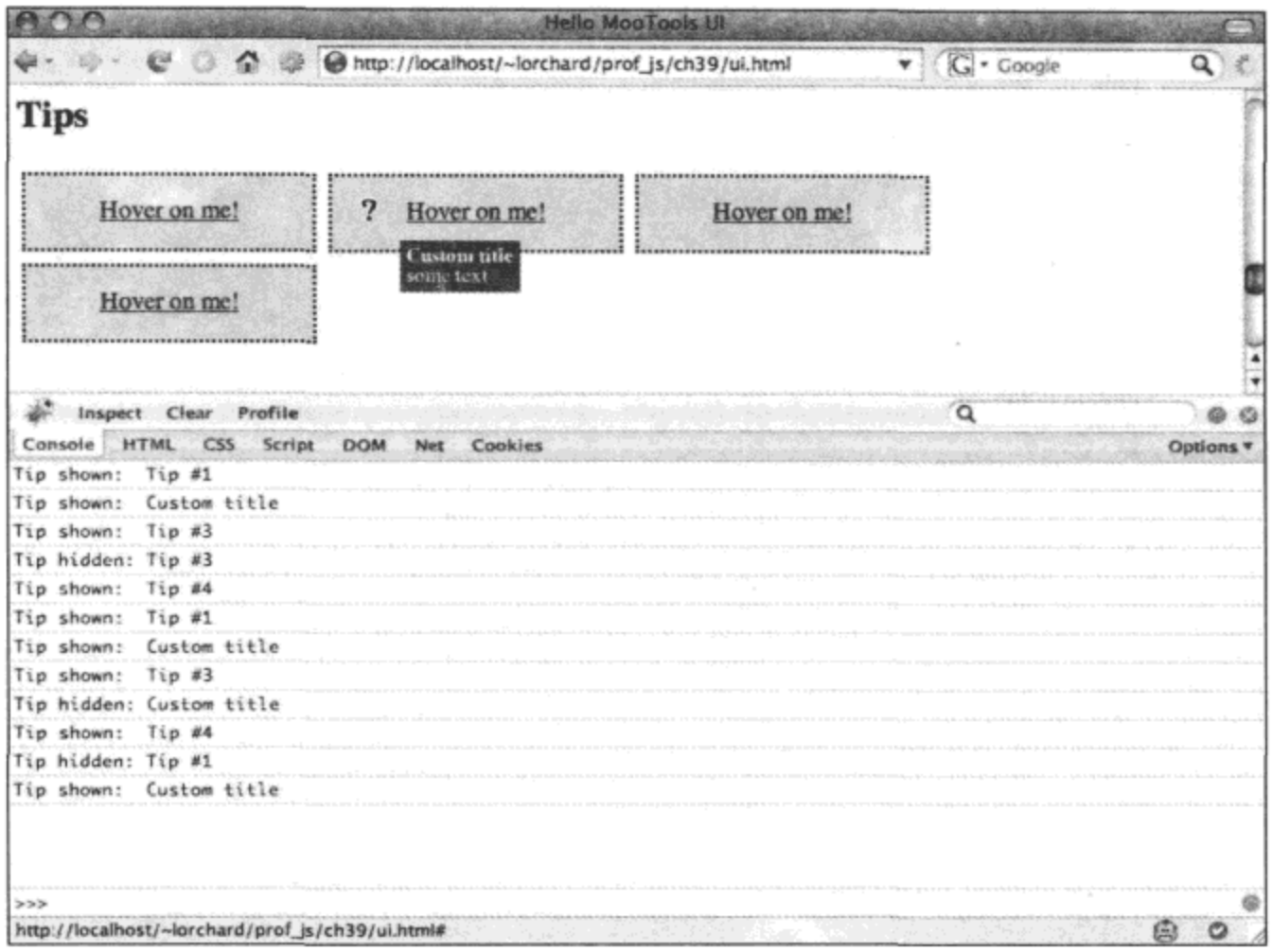


图　34-13

34.2.8　构建滑块控件

尽管 MooTools 并没有提供大量的用户界面或表单控件，但它确实提供了少数几个有用的轻量级窗口部件。其中一个就是 Slider 类，它实现了一种简单的一维值滑块控件，该控件有一个可拖动的、受到轨迹约束的调节器元素。

下面的标记构建了一个滑块：

```
<style type="text/css">
    .slider {
        width: 300px;
        height: 25px;
        background: #ccc;
    }
    .knob {
        width: 25px;
        height: 25px;
        text-align: center;
        background: #333;
        color: #fff;
    }
    .status {
        margin: 1em 0 0 0;
    }
    .status code {
```

```
        font-weight: bold
    }
</style>

<div id="ex_slider">
    <h2>Slider</h2>

    <div class="slider">
        <div class="knob">*</div>
    </div>

    <div class="status">
        Value = <code>--</code>
    </div>

</div>
```

上面示例中的标记定义了一个水平滑块轨迹 元素，在该元素内部还有一个调节器元素。CSS 为这个控件提供了一些简单的表现方式，但可以根据自己的设计来应用任何样式；Slider 类并没有在这个控件上强制实施任何主题或样式。

当标记就绪之后，下面创建 Slider 类的一个实例：

```
var slide = new Slider(
    $$('#ex_slider .slider')[0],
    $$('#ex_slider .knob')[0],
    {
        snap:   true,
        range:  [10, 200],
        wheel:  true,
        steps:  10,
        mode: 'horizontal',

        onChange: function(pos) {
            $log("Slider set to " + pos);
            $$('#ex_slider .status code').set('text', pos);
        }

    }
).set(50);
```

Slider 构造函数的第一个参数是一个指向滑块容器元素的引用，第二个参数应该指向该滑块内的调节器元素。第三个和第四个参数则是一个由选项构成的对象或散列，包括如下选项：

- **range**　值为 false，或者是一个由该滑块将要使用的最小值和最大值组成的数组。默认为 false。
- **steps**　这个值与 range 选项一起用来将该范围切分成指定数量的块。默认为 100。
- **snap**　如果该选项设为 true，那么滑块在拖动时将在滑块范围内跳跃前进。默认为 false。
- **offset**　在默认情况下，这个选项使用调节器元素的 offsetHeight 或 offsetWidth 来判断它在拖动期间的位置。但是，可以利用这个选项来提供自己的偏移位置。

- **wheel** 如果该选项为 true，就可以使用鼠标滚轮来移动该滑块。默认为 false。
- **mode** 可以是 horizontal 或 vertical。默认为 horizontal。

Slider 类也支持包括如下的事件：

- **change** 滑块中的每次变化都会引发该事件。
- **complete** 当滑块拖动结束时引发该事件。

滑块的当前数字值会作为参数传给这些事件的处理程序。

图 34-14 给出的是本节中的代码所构建的滑块预览，同时还给出了一段显示拖动滑块调节器时产生的值的日志文本消息。

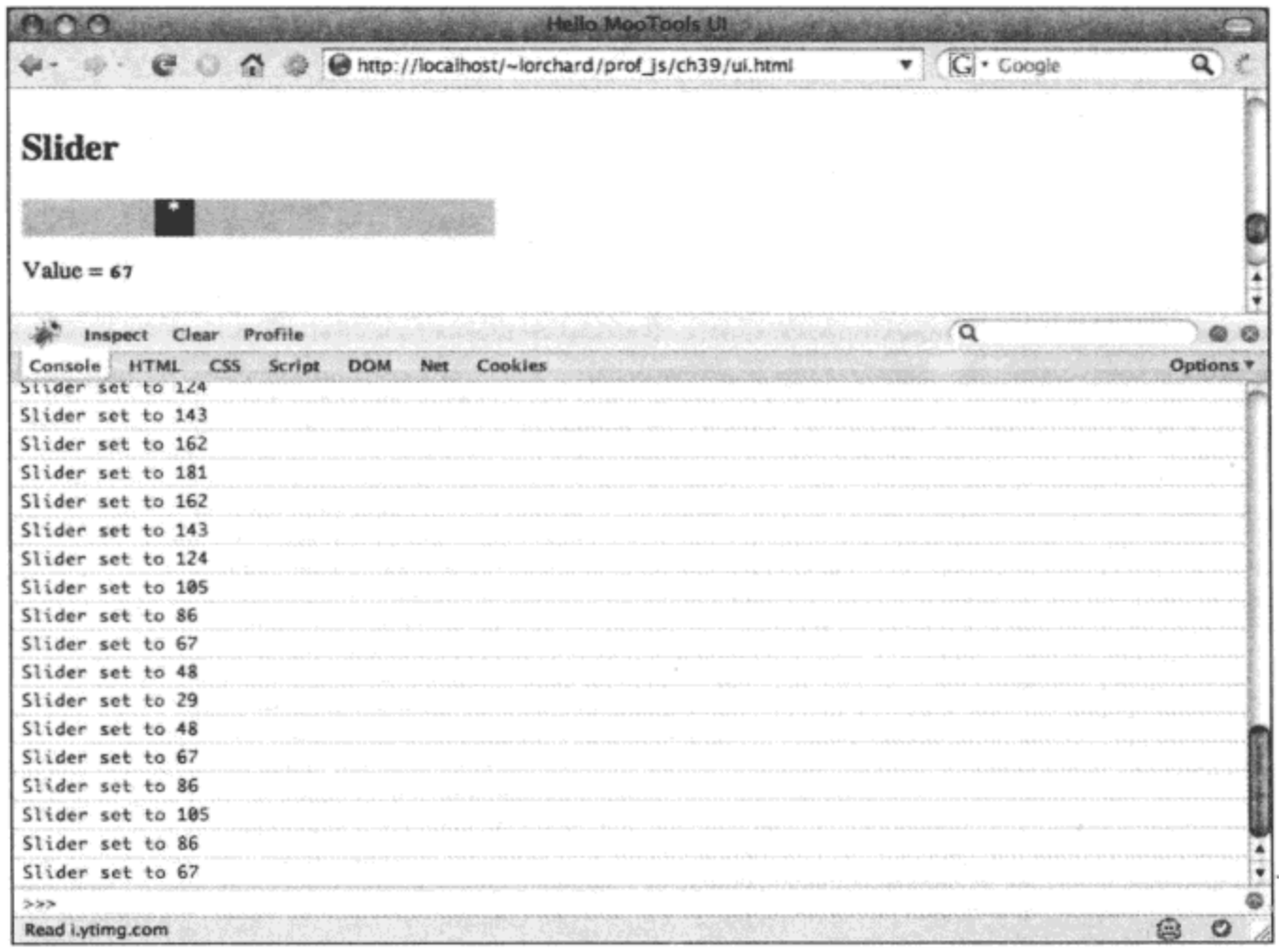


图 34-14

34.3 本章小结

在本章中，我们浏览了 MooTools 提供的动画工具和用户界面窗口部件。对于这个框架的许多开发人员和爱好者，本章讲解的功能是构建在本书这一部分的前几章基础之上的产物。

通过使用 Fx 系列类，我们可以编排一些非常复杂的、效果极其丰富的视觉转换，它们均是随着时间进展操作 CSS 属性。但是，如果在页面上只需要少数几个华丽的修饰物，那么 MooTools 为常用转换提供了一组快速的快捷方式，这样就不需要从头开始构造完整的动画。除了动画之外，本章还介绍了 MooTools 为管理内容和交互式用户输入而提供的一组用户界面布局和控件。