TURING 图灵程序设计丛书 Web开发系列

Apress®

Accelerated DOM Scripting with Ajax, APIs, and Libraries

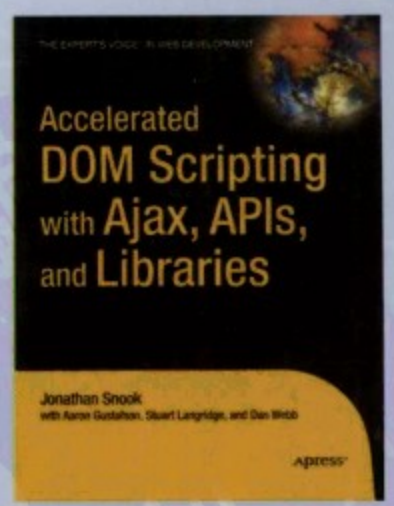
JavaScript 捷径教程

[加] Jonathan Snook
[美] Aaron Gustafson
[英] Stuart Langridge
[英] Dan Webb

著

郭晓刚 等译

- 四位顶尖专家合著
- 初级JavaScript程序员的绝佳进阶书
- 洞悉JavaScript内幕



人民邮电出版社
POSTS & TELECOM PRESS

TURING 图灵程序设计丛书 Web开发系列

Accelerated DOM Scripting with Ajax, APIs, and Libraries

JavaScript 捷径教程

[加] Jonathan Snook
[美] Aaron Gustafson
[英] Stuart Langridge
[英] Dan Webb
著

郭晓刚 等译

人民邮电出版社
北 京

图书在版编目（CIP）数据

JavaScript 捷径教程 / （加）斯努克（Snook, J.）等著；郭晓刚等译. —北京：人民邮电出版社，2009.1
（图灵程序设计丛书）
书名原文：Accelerated DOM Scripting with Ajax, APIs, and Libraries
ISBN 978-7-115-19259-2

Ⅰ. J… Ⅱ. ①斯…②郭… Ⅲ. JAVA 语言－程序设计－教材 Ⅳ. TP312

中国版本图书馆CIP数据核字（2008）第184147号

主 要 内 容

本书讲述了 JavaScript 以及 DOM 的应用。重点讲解了 JavaScript 库，并通过实际的示例说明了如何把这些库应用于你的项目。同时还解释了Ajax，教你如何充分计划并将其应用于项目。此外，你还将了解如何构建简单的动画对象来为页面中的元素增加动态效果。书中提到的各项技术均配有直观而简洁的示例，可帮助你快速掌握这些技术。

本书面向具有一定的 JavaScript 和 DOM 脚本开发经验的 Web 开发人员。

图灵程序设计丛书

JavaScript 捷径教程

◆ 著 [加]Jonathan Snook [美]Aaron Gustafson
[英]Stuart Langridge [英]Dan Webb
译 郭晓刚 等
责任编辑 傅志红
执行编辑 王慧敏

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
邮编 100061 电子函件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京隆昌伟业印刷有限公司印刷

◆ 开本：800×1000 1/16
印张：13
字数：307千字 2009年1月第1版
印数：1－3000册 2009年1月北京第1次印刷
著作权合同登记号 图字：01-2008-3858号
ISBN 978-7-115-19259-2/TP

定价：35.00元

读者服务热线：(010)88593802 印装质量热线：(010)67129223
反盗版热线：(010)67171154

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

Java 一览无余：**Java** 视频教程 | **Java SE** | **Java EE**

.Net 技术精品资料下载汇总：**ASP.NET** 篇

.Net 技术精品资料下载汇总：**C#**语言篇

.Net 技术精品资料下载汇总：**VB.NET** 篇

撼世出击：**C/C++**编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 **Ruby**、**Ruby on Rails** 精品电子书等学习资料下载

数据库精品学习资源汇总：**MySQL** 篇 | **SQL Server** 篇 | **Oracle** 篇

最强 **HTML/xHTML**、**CSS** 精品学习资料下载汇总

最新 **JavaScript**、**Ajax** 典藏级学习资料下载分类汇总

网络最强 **PHP** 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

经典 **LinuxCBT** 视频教程系列 **Linux** 快速学习视频教程一帖通

天罗地网：精品 **Linux** 学习资料大收集(电子书+视频教程) **Linux** 参考资源大系

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

FreeBSD/OpenBSD/NetBSD 精品学习资源索引 含书籍+视频

Solaris/OpenSolaris 电子书、视频等精华资料下载索引

前　言

本书会让你对JavaScript有更深刻的理解，并将之运用到Web开发的各个方面，比如Ajax、动画和各种DOM脚本编程任务。对JavaScript的透彻理解是提升自身代码质量、利用流行的JavaScript库加速开发过程的重要前提。DOM脚本编程是现今的热门，希望本书能锦上添花，将所有难点阐释得更加清楚。

本书读者

本书是为已经有过一些JavaScript经验的读者而准备的。读者应该已经清楚JavaScript的语法，已经编写过像弹出窗口这类简单的脚本。服务器端的编程经验不是必需具备的，但肯定没有坏处。

本书结构

本书先介绍JavaScript和DOM的有关知识，然后以之为基础展开论述DOM脚本编程的各种主题，比如DOM的操作、Ajax和视觉效果。

- 第1章介绍JavaScript在业界发展的最新状况。此外，还介绍了JavaScript在浏览器中执行的原理及其对代码编写的影响，以及如何调试脚本和测试中所用的各种工具。
- 第2章浏览若干HTML和CSS的重要技术之后，在坚实的根基之上引入JavaScript的主题。先重温JavaScript的基础知识，然后正式展开对DOM的介绍——如何在DOM中移动和操作。
- 第3章解释在JavaScript之上的面向对象编程。还探讨了JavaScript的闭包和函数式编程等特性，正是这些特性使JavaScript成为一门强大的语言，这在DOM脚本编程中尤其能表现出来。
- 第4章：JavaScript库已相当普及，本章剖析了一些流行的库，展示了如何轻松地将之运用到项目当中。最后还分析了挑选库的时候应当注意的事项。
- 第5章：Ajax已经无处不在。本章解释了什么是Ajax，揭示了Ajax请求内部的详情，还阐述了Ajax的陷阱及如何为之未雨绸缪。同时还将介绍数据交换的格式，学习如何根据使用的

场合选择最适合的格式。

- 视觉效果看似有些多余，第6章将带你重新认识如何在页面中有效利用动画加强用户体验。你将会看到如何构建自己的动画对象，也将学到怎样用JavaScript库来实现动画。
- 第7章表单验证是JavaScript最常担负的任务，DOM脚本编程用在这里正是再合适不过。本章将运用各种客户端脚本技巧攻克表单验证的难关，比如阻止表单提交、用JavaScript和DOM显示出错消息。
- 第8章通过一个案例来演示如何在页面上平滑、优雅地显示和隐藏元素。遵循“渐进增强”的原则，灵活运用CSS、HTML和DOM脚本编程打造一个现代的FAQ页面。
- 第9章是最后一章，介绍一个在线帮助系统，系统的目的是引导用户使用一个相对复杂的在线应用。它说明了普通的桌面应用也可以搬到网络上，同时仍然像原来一样触手可得。

先决条件

只需准备一个文本编辑器用来编写脚本代码，以及一个当前通用的浏览器用来测试代码。本书中代码示例皆以新近的浏览器为目标，包括IE 6、IE 7、Firefox 2、Safari 2和Opera 9。

与作者联系

可以通过本书作者的网站与我联系：http://snook.ca/。

致谢

我要特地感谢许多在本书创作过程中给予我帮助和启迪的人。感谢Apress整个团队，尤其要感谢Chris Mills和Richard Dal Porto的耐心。我也为有Dan Webb、Aaron Gustafson和Stuart Langridge这三位饱学睿智的合著者而自豪。非常感谢Cameron Adams进行了技术审校。下次大家见面的时候，我一定请大家喝一杯。

大力感谢JavaScript社区中不吝与所有人分享知识的各位，有Douglas Crockford、Andrew Dupont、Dustin Diaz、Dean Edwards、Christian Heilmann、Peter-Paul Koch（PPK）、Stuart Colville、Joe Hewitt、John Resig，还有我挂一漏万，没能一一记住的各位。

最后，没有我家人的帮助这本书也不可能成功面世。谢谢妈妈、Mel、Pat，还有Trish在我忙于写作的周末帮忙照看儿子Hayden。谢谢妻子Michelle督促并支持我完成这本书。

Jonathan Snook

目　录

第 1 章　JavaScript 的现状 …… 1

1.1　30 年河东，30 年河西 …… 1
1.2　JavaScript 遇上结合 DOM 的 HTML …… 2
1.3　Ajax 的崛起 …… 3
1.4　管理 JavaScript …… 4
1.4.1　代码装载 …… 4
1.4.2　代码解析 …… 4
1.4.3　正确地在 XHTML 页面中嵌入代码 …… 5
1.5　代码调试 …… 6
1.5.1　警告 …… 6
1.5.2　页面记录 …… 6
1.5.3　浏览器插件 …… 7
1.5.4　HTTP 调试 …… 10
1.6　小结 …… 12

第 2 章　HTML、CSS 和 JavaScript …… 13

2.1　基础知识 …… 13
2.2　HTML 最佳实践 …… 15
2.2.1　HTML 与 XHTML …… 16
2.2.2　两全其美 …… 16
2.3　CSS 基础 …… 17
2.3.1　将含义表达出来 …… 17
2.3.2　元素的标识 …… 17
2.3.3　应用 CSS …… 19
2.4　JavaScript 基础 …… 21
2.4.1　函数 …… 21
2.4.2　对象、属性和方法 …… 21
2.4.3　点号语法和方括号语法 …… 22
2.4.4　原型 …… 23
2.4.5　值传递和引用传递 …… 24
2.5　JavaScript 与 DOM …… 25
2.5.1　DOM 是什么 …… 25
2.5.2　DOM 树的结构 …… 26
2.5.3　document 对象 …… 27
2.5.4　通过类名获取元素 …… 29
2.6　在 DOM 中移动 …… 30
2.7　处理属性 …… 31
2.7.1　style 属性 …… 32
2.7.2　class 属性 …… 32
2.8　向 DOM 中插入内容 …… 33
2.9　浏览器嗅探与对象检测 …… 35
2.10　正则表达式 …… 35
2.11　格式化的惯例 …… 37
2.12　事件处理 …… 38
2.12.1　内联的事件处理 …… 38
2.12.2　this 关键字 …… 39
2.12.3　无侵入的 JavaScript …… 39
2.12.3　在页面加载前访问元素 …… 40
2.12.5　用 DOM 方法绑定事件 …… 42
2.12.6　事件捕捉与事件冒泡 …… 43
2.12.7　在 IE 中追加事件 …… 43
2.12.8　检查上下文 …… 44
2.12.9　取消行为 …… 46
2.12.10　综合练习 …… 47
2.13　事件委托 …… 48
2.13.1　搜寻冒泡中途经过的元素 …… 52
2.13.2　事件委托不适用的情形 …… 53
2.14　小结 …… 54

第 3 章　面向对象编程 …… 55

3.1　什么是面向对象编程 …… 55
3.2　函数 …… 56
3.2.1　添加方法和属性 …… 57
3.2.2　对象的实例化机制 …… 58
3.2.3　在构造函数中返回对象 …… 58
3.2.4　原型 …… 60
3.3　字面量对象 …… 61

3.4 for..in 循环 ····· 63
3.5 命名的参数 ····· 65
3.6 命名空间 ····· 65
3.7 闭包 ····· 66
3.8 封装 ····· 68
3.9 函数式编程 ····· 71
3.9.1 回调 ····· 71
3.9.2 函数的 call 和 apply ····· 73
3.9.3 在集合上应用函数 ····· 74
3.9.4 可串接方法 ····· 75
3.9.5 内部迭代器 ····· 75
3.10 小结 ····· 76
第 4 章 库 ····· 77
4.1 DOM 操作 ····· 77
4.2 应用上的便利措施 ····· 78
4.2.1 语言扩展和语言桥路 ····· 78
4.2.2 事件处理 ····· 78
4.2.3 Ajax ····· 79
4.2.4 字符串和模板处理 ····· 79
4.2.5 使用集合 ····· 80
4.2.6 处理 JSON 和 XML ····· 80
4.3 界面部件 ····· 81
4.4 流行的库 ····· 81
4.4.1 Dojo ····· 82
4.4.2 Prototype ····· 83
4.4.3 jQuery ····· 85
4.4.4 Yahoo! UI Library（YUI） ····· 86
4.4.5 Mootools ····· 87
4.4.6 Script.aculo.us ····· 89
4.4.7 ExtJS ····· 89
4.5 新出现的库 ····· 91
4.5.1 Base2.DOM ····· 92
4.5.2 DED|Chain ····· 92
4.6 怎样选择库 ····· 92
4.6.1 社区 ····· 93
4.6.2 文档 ····· 93
4.7 小结 ····· 93
第 5 章 Ajax 和数据交换 ····· 94
5.1 分析 Ajax 应用 ····· 94
5.2 解构 Ajax 过程 ····· 95
5.2.1 Ajax 的请求/响应过程 ····· 97
5.2.2 失败 ····· 98
5.2.3 绘制故事板 ····· 98
5.3 Ajax 的数据格式 ····· 100
5.3.1 XML ····· 100
5.3.2 XML 之外的选择 ····· 107
5.4 构造可重用的 Ajax 对象 ····· 110
5.5 为失败做准备 ····· 112
5.5.1 超时处理 ····· 113
5.5.2 HTTP 状态代码 ····· 115
5.5.3 多重请求 ····· 115
5.5.4 意外的数据 ····· 116
5.6 用库处理 Ajax 调用 ····· 117
5.6.1 Prototype ····· 117
5.6.2 YUI ····· 118
5.6.3 jQuery ····· 119
5.7 小结 ····· 120
第 6 章 视觉效果 ····· 121
6.1 为什么要使用视觉效果 ····· 121
6.2 构建一个简单的动画对象 ····· 122
6.2.1 回调 ····· 127
6.2.2 动画队列 ····· 129
6.3 扩展动画类 ····· 130
6.4 用库来实现动画 ····· 133
6.4.1 Script.aculo.us ····· 134
6.4.2 jQuery ····· 135
6.4.3 Mootools ····· 136
6.5 小结 ····· 137
第 7 章 表单验证与 JavaScript ····· 138
7.1 在服务器上验证 ····· 138
7.2 客户端 ····· 140
7.2.1 用 JavaScript 添加显示错误消息的 Span 块 ····· 147
7.2.2 阻止表单提交 ····· 148
7.3 用 Ajax 实现表单验证 ····· 150
7.3.1 服务器端验证 ····· 151
7.3.2 客户端 ····· 153
7.4 小结 ····· 156

第 8 章　案例研究：改良 FAQ 页面……157
8.1　第 1 课：瞄准目标……157
8.2　第 2 课：JavaScript 之舞……162
8.3　小结……176

第 9 章　案例研究：动态帮助系统……177
9.1　任务……177
9.2　计划和准备……178
9.2.1　总体设计……178
9.2.2　项目准备……178
9.3　编写标记……179
9.3.1　用布局处理共同的标记……179
9.3.2　添加一个示例应用页面……181
9.4　用 CSS 添加样式……182
9.5　Prototype 和 Low Pro 出场……183
9.6　让帮助栏可用……184
9.6.1　建立帮助控制器……184
9.6.2　添加行为……185
9.6.3　实现加载提示……187
9.7　最后润色……189
9.7.1　用 Moo.fx 添加动画……189
9.7.2　实现边栏内锚点……191
9.8　回顾……192
9.8.1　用符合语义的 HTML 奠定坚实的基础……192
9.8.2　恰当使用 HTML、CSS、JavaScript……192
9.8.3　用 CSS 选择符充当应用的胶水……193
9.8.4　对 Ajax 来说，简单是最好的……193
9.9　小结……193
9.10　源代码……194

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

Java 一览无余：**Java** 视频教程 | **Java SE** | **Java EE**

.Net 技术精品资料下载汇总：**ASP.NET** 篇

.Net 技术精品资料下载汇总：**C#**语言篇

.Net 技术精品资料下载汇总：**VB.NET** 篇

撼世出击：**C/C++**编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 **Ruby**、**Ruby on Rails** 精品电子书等学习资料下载

数据库精品学习资源汇总：**MySQL** 篇 | **SQL Server** 篇 | **Oracle** 篇

最强 **HTML/xHTML**、**CSS** 精品学习资料下载汇总

最新 **JavaScript**、**Ajax** 典藏级学习资料下载分类汇总

网络最强 **PHP** 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

经典 **LinuxCBT** 视频教程系列 **Linux** 快速学习视频教程一帖通

天罗地网：精品 **Linux** 学习资料大收集(电子书+视频教程) **Linux** 参考资源大系

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

FreeBSD/OpenBSD/NetBSD 精品学习资源索引 含书籍+视频

Solaris/OpenSolaris 电子书、视频等精华资料下载索引

第1章 JavaScript的现状

本章简要地回顾了Ajax的崛起以及它对JavaScript流行程度的影响，以便你能够对过去十年中业界的变化有个宏观的认识。然后解释了JavaScript在浏览器中如何执行，你又该如何应对。你将学到调试程序的各种方法以及各种工具。要想赶走那些阴魂不散的bug，理解代码的执行是非常重要的。

1.1 30年河东，30年河西

自1995年诞生以来，JavaScript已经走过了一段不短的路。它的用途从最初简单的图片和表单交互，到现在已经扩展到了各种各样的用户界面控制。网站已不再是静态的了。从表单验证到动画效果，再到灵活性和响应能力能够与桌面程序相匹敌的网站，JavaScript已经赢得了自身的地位。传统（且昂贵）的桌面程序（如文字处理、日历和邮件程序）都正在被便宜（且易用）的Web应用所取代，例如Writely、30 Boxes和Google Mail。

在10年的历程中，JavaScript有过兴衰起伏；可喜的是，现在它正胜利归来。为什么是这个时候呢？答案很简单：它无所不在。大多数开发者的目标是让他们的劳动成果能被所有人使用。HTML很早就实现了这个目标，其格式的主要部分在因特网于20世纪90年代后期真正起飞之前就已经成熟。为一种浏览器所编写的HTML在其他浏览器中的显示也几乎是一样的，不管平台是Mac、PC还是Linux。

不过，JavaScript以前并不成熟。它与HTML文档交互的能力因浏览器而异。它的两大主力——Netscape和IE的实现方式很不一样，也就是说，为完成相同的任务，必须准备两种完全不同的实现。人们常常需要编写辅助脚本，甚至是成套的JavaScript库来弥合两者之间的缝隙。要知道，JavaScript库当时并不那么流行，大多数人都认为JavaScript库对于他们的需求来说太臃肿了。使用库当然可以简化开发，但相比人们想要用JavaScript来解决的问题，库实在太庞大了。要知道当时的带宽可不是像现在的样子。出于带宽、安全的种种顾虑，而且有些公司完全禁用了JavaScript，这些都让JavaScript沦落到了玩具语言的境地。看起来就像是Web也可以完全不需要JavaScript。

随着IE成为“浏览器大战”当之无愧的胜利者，Netscape没落了。能够推测，开发者可以只针对IE进行开发，因为IE已经占据了超过90%的市场份额。很多开发者（包括我）也是这么想的。不过这时“无处不在”还没成为现实。一些企业环境以及家庭用户仍然继续使用Netscape作为默认浏览器。即便21世纪已经过去了好几年，我的客户仍然要求符合Netscape 4的标准。除了表单验证之类的简单功能之外，要想实现任何跨浏览器的功能仍然是一场苦战。

W3C（万维网联盟）的成员包括很多浏览器的开发者，该协会继续在完善和落实我们今天所用的很多技术，包括HTML/XHTML、CSS（层叠样式表）和DOM（文档对象模型）。

随着标准的确定和逐渐成熟，浏览器开发者有了一个坚实的基准，可以在开发中将其作为参照。情况出现转机。终于，Mozilla Firefox于2004年面世了，总算有了一个能跨操作系统并且出色地支持最新的HTML/XHTML、CSS和DOM标准的浏览器了。它甚至还支持一些非标准的技术，比如它自己的原生`XMLHttpRequest`对象（这是Ajax的关键要素，将在第5章讲述）。Firefox大受欢迎，尤其是在开发者中。比如，W3Schools网站上就显示出最近Firefox的使用率接近34%（参见http://w3schools.com，2007年5月）。

注解　浏览器统计数字不可全信。统计数字总是有水分，讨厌的水分。每个网站都有其特点，会吸引特定的人群，因此你的统计数字可能和别人的不一样。例如，我的网站主要是面向开发者的，访问者中有60%使用Firefox。所以说，建造出能适应所有浏览器的网站是很重要的，因为你没法知道你的用户会用什么浏览器，也没法预料市场会怎么变化。

Apple公司为Mac发布了Safari，以填补微软公司停止为Mac平台开发浏览器所留下的空白。Safari、Firefox以及Camino（与Firefox基于相同的Gecko引擎）坚定地支持HTML和CSS标准。早期的Safari版本对DOM支持有限，不过最新的版本已经好多了，而且它也包含了对`XMLHttpRequest`的支持。最重要的是，它们都支持同样的一套标准。

目前市面上的各种浏览器版本之间的差异已经很小，我们所寻求的“无所不在”就在眼前。浏览器间差异的缩小，意味着可以开发出比较小型的代码库来降低跨浏览器开发的复杂度。聪明的程序员们也已经在以前鲜有涉及的方面开始利用JavaScript的优点了。JavaScript开始复苏了！

Google示范性地说明了用JavaScript驱动的应用程序即将成为主流。以Google Maps（http://maps.google.com/）和Google Suggest（www.google.com/webhp?complete=1）为代表的许多应用程序都已展示出了JavaScript的威力、速度及交互能力。

1.2　JavaScript遇上结合DOM的HTML

虽然我们讨论的是JavaScript及其演进，但占据浏览器舞台中心的其实是DOM（与最初相比

它已经完全变了样）。Netscape的第2版，也就是发明JavaScript的时候，就可以访问表单和图片元素了。当IE第3版发布的时候，出于竞争的目的，它模仿了Netscape的做法，同时也保证了页面能够正确地显示。

到了两个浏览器都发布第4版的时候，它们都试图通过增加与更多页面交互的方式来扩展其能力，特别是提供了在页面上定位和移动各种元素的能力。它们实现目标的方法各不相同，而且所用的技术又都是专有的，因此导致了很多让人头痛的问题。

W3C开发出了第一个DOM推荐标准，用来标准化各种浏览器都采取的途径，这让开发者能够较容易地创造出跨浏览器的功能——就像HTML推荐标准一样。W3C DOM提供了通过JavaScript添加和删除元素的能力，从而有希望与完整的HTML（和XML）文档交互。无论是Mozilla还是IE 5+，都能出色地支持DOM Level 1推荐标准。

W3C随后推出了DOM推荐标准的第2版和第3版，两者继续在Level 1定义的功能上构建而成。（两个DOM版本之间的差异会在第2章中谈到。）

1.3 Ajax 的崛起

Ajax这个术语最初是“异步JavaScript和XML”（Asynchronous JavaScript and XML）的简写，由 Adaptive Path 的 Jesse James Garrett 所发明（www.adaptivepath.com/publications/essays/archives/000385.php）。它的本意是用一个术语囊括所用的一组技术。这种技术的核心是使用XMLHttpRequest对象，再加上DOM脚本编程、CSS和XML。

XMLHttpRequest是微软公司在1998年为Outlook Web Access开发的一种专有技术。它是一个ActiveX对象，让JavaScript能够与服务器通信而无需刷新页面。不过，直到Mozilla Firefox出现并包括了原生版本的XMLHttpRequest，这个对象才被广泛使用。随着Google Mail等应用的成功，其他浏览器的开发者也很快在浏览器中加入了这种技术。现在IE、Firefox、Opera和Safari全都支持原生的XMLHttpRequest对象。这一技术已无所不在，其流行就顺理成章了。W3C现在正试图为Ajax建立一个标准（参见www.w3.org/TR/XMLHttpRequest）。

注解 ActiveX是微软的一项技术，可以让组件在操作系统中彼此通信。结合使用JavaScript和ActiveX，实际上可以和用户机器上安装的许多程序进行交互。例如，在宽松的安全设置条件下，可以打开Microsoft Office程序与之交互，甚至从中把数据复制出来，这些所有操作都可以在一个网页中执行。实际上，任何提供了COM（组件对象模型）接口的程序，都可以用这种方法来操纵。

之前提过XML是Ajax中的一个关键概念，你可能会纳闷XML是如何参与进来的。按照Jesse

James Garrett最初的说法，Ajax将XML作为一种数据交换格式，将XSLT作为一种操纵格式，而XHTML则作为一种呈现格式。XML最初被定义为Ajax的一个主要组成部分，不过这个严格的定义已经被放宽，现在表示经由JavaScript，使用XMLHttpRequest对象和许多在实现网站或Ajax应用中所涉及的技术（如HTML和JSON）来与服务器通信的过程。

Ajax能够与服务器通信而不必刷新页面。但这意味着什么呢？这表明你可以执行异步操作（也就是Ajax的首字母A）。你可以在提交表单之前就验证表单。举个例子，你有没有在注册服务的时候却发现所选的用户ID已经被别人用了？所以只好按下后退按钮，输入另一个用户名（还要重新输入密码，因为密码输入框不会保留输入），然后再次按下提交按钮。只要输入的用户名已经被使用，就得一再重复这个烦人的过程。有了Ajax，你就可以在用户填写表单其余部分的同时检查用户ID。如果发现ID已经被人占了，用户会得到一条错误消息，以便在提交表单之前就改正。

拥有了这种崭新的能力，开发者早已开始马不停蹄地创造形形色色的应用了。但是，很多都是虚有其表但无实际用处，看起来很丰富但威力不足。当你打算添加最新的技巧时，务必要考虑其可用性和可访问性。这个主题的讨论将贯穿全书。

1.4　管理 JavaScript

如今，JavaScript应用可能变得庞大而臃肿。在让你入手JavaScript之前，我想先谈谈要把代码放在HTML页面的什么地方，以及保持长期可维护性的最佳方式。在测试和评估你自己的代码时，有些微妙之处必须记住。

1.4.1　代码装载

首先需要理解的是装载过程。当一个HTML页面被装载时，它会装载并解析过程中遇到的任何JavaScript。Script标签可以出现在文档的<head>中，也可出现在<body>中。如果有指向外部JavaScript文件的链接，它会先装载该链接，再继续解析页面。嵌入第三方的脚本时，如果远程服务器因负担过重而无法及时返回文件，就有可能导致页面的装载时间显著变长。因此最好尽量在接近HTML页面底部的地方装载第三方脚本。

```
<head>
<title>My Page</title>
<script type="text/javascript" src="myscript.js"></script>
</head>
```

而你自己编写的脚本应该放在文档的首部，而且应该尽快装载它们，因为它们可能包含页面的其余部分需要依赖的功能。

1.4.2　代码解析

代码解析是浏览器取得你所编写的代码，并将之转化成可执行代码的过程。这个过程的第一

件事是检查代码的语法是否正确。如果不正确，过程会立即失败。如果你试图运行一个包含语法错误的函数（比如中间少了半边括号），你很可能会得到一条错误消息，告诉你函数还没定义。

当浏览器确认代码合法之后，它会解析script块中所有的变量和函数。如果你要调用的函数来自其他script块或者其他文件，请确保它在当前script元素之前装载。在下面的例子中，虽然loadGallery函数的声明出现在调用之后，但它还是可以执行：

```
<script type="text/javascript">
loadGallery();
function loadGallery()
{
    /* gallery代码 */
}
</script>
```

而在下面的例子中，你将得到一条错误消息，因为第一个script元素将在第二个script元素之前解析并执行：

```
<script type="text/javascript">
loadGallery();
</script>

<script type="text/javascript">
function loadGallery()
{
    /* gallery 代码 */
}
</script>
```

我的做法一般是尽量把代码放到函数里面，然后从外部文件中加载它们，接着再执行一些代码来启动整个脚本。

1.4.3 正确地在 XHTML 页面中嵌入代码

在HTML页面中嵌入JavaScript非常简单，你从前面的例子也可以看出来。网上的很多例子常常还包括HTML注释标签，用来向不支持JavaScript的浏览器隐藏JavaScript代码。

```
<script type="text/javascript">
<!--
/* 运行我的代码 */
loadGallery();
//-->
</script>
```

不过，如今已经没人使用不支持JavaScript的浏览器了，因此HTML注释标签不再是必需的了。

在XHTML中事情又有些不一样。因为XHTML遵循XML的规则，所以脚本必须用CDATA块围

起来。CDATA块以<![CDATA[开头，以]]>结尾。

```
<script type="text/javascript">
<![CDATA[
/* 运行我的代码 */
loadGallery();
]]>
</script>
```

注解　在整本书里，我差不多总是使用HTML；如果你比较喜欢用XHTML，请注意区别。

1.5　代码调试

不管代码是多么简单，都不免会出现错误。因此，需要一种方法去了解错在哪里，为什么出错，以及如何修复错误。

1.5.1　警告

最常见的JavaScript调试技术可能要数alert()了。你完全不需要安装任何软件，也不需要写什么复杂的配置代码。要做的只是在代码中插进一行，把想要了解的信息放进警告语句，就可以等着看结果了。

```
alert(varname);
```

不过警告语句的效率不高，尤其是在跟踪一些时效性强的代码或者循环中的数值的时候。在时效性强的情况下（例如动画），警告语句会破坏现场，因为它要停下来等待输入才能继续。如果遇到循环的情况，你会被迫点击无数次OK按钮。万一不小心造成了无限循环，你就只能强行关闭浏览器才能重新得回控制权，而所有打开的页面都随着浏览器关闭而丢失——这可一点都不好玩！

警告的低效还体现在它只能显示字符串。如果你想了解一个数组的内容，只能先把数组拼接成一个字符串再传给警告语句。

1.5.2　页面记录

页面记录是一个方便易用的技巧，而且优于警告语句。在页面上建一个空的 ，把它设成绝对定位并且在溢出时增加滚动条。需要查看什么信息的时候，把数据附加到那个 后面（或放在前面）就可以了。

代码如下：

```
function logger(str){
```

```
    var el = document.getElementById('logger');
    // 如果未找到logger容器就创建一个
    if(!el) {
        el = document.createElement('div');
        el.id = 'logger';
        var doc = document.getElementsByTagName('body')[0];
        doc.appendChild(el);
    }
    el.innerHTML += str + '<br>';
}
var value = 5;
logger('value = ' + value);
```

为这个元素增添些许风格并确保它不会干扰布局的CSS代码如下所示：

```
#logger {
    width:300px;
    height:300px;
    overflow:scroll;
    position:absolute;
    left:5px; top:5px;
}
```

有人根据同样的原理制作出了一些很有用的精巧的记录器。在线杂志*A List Apart*上有篇文章介绍了一个fvlogger（http://alistapart.com/articles/jslogging）。log4javascript项目也值得一看（www.timdown.co.uk/log4javascript）。log4javascript用了一个独立的窗口来记录信息，不会干扰当前的文档，所以可能更好用一些。

1.5.3 浏览器插件

浏览器插件通常是一些精雕细琢的应用，不但为你提供关于JavaScript的各种细节，连页面上的HTML和CSS也都一网打尽。它们可以在你探索页面中所发生的一切时充当救生员的角色。但不好的一面是它们几乎总是特定于某个浏览器的。也就是说在某些浏览器上的测试会更困难，尤其是当问题只发生在缺乏插件的浏览器上的时候。

1. DOM Inspector

谈到JavaScript开发，Firefox是最适合开发用的浏览器之一。它的DOM支持是最优秀的，而且还有一些最优秀的调试工具。Firefox自己就内建了DOM Inspector，如图1-1所示。

在DOM Inspector的帮助下，你可以在文档树里穿梭，查看每个结点的各种属性。在上面的截图中可以看到，它列出了可通过JavaScript访问的各种属性。它还有各种视图，让你查看设置了哪些样式、属性值的计算结果，非常便于调查布局错乱的原因。

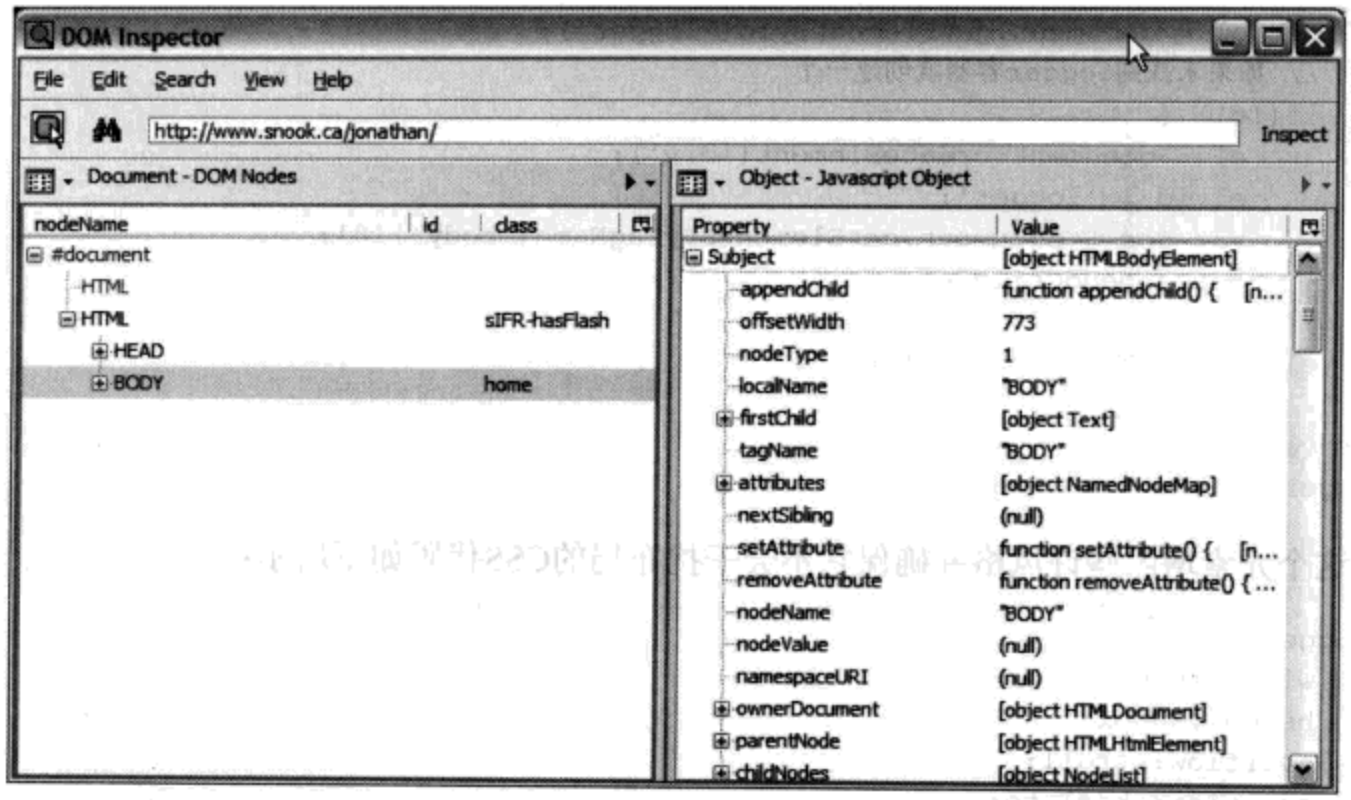


图1-1　Firefox DOM Inspector

2. Firebug

Firebug（www.getfirebug.com）目前在JavaScript和CSS调试工具中位居首位。到目前为止，它是你使用的工具库中最强大最灵活的工具。

Firebug让DOM Inspector更上一层楼。安装之后可以从状态栏打开Firebug，状态栏上的图标（如图1-2所示）能指示出当前页面上是否存在错误。

图1-2　Firebug的检查标记图标

点击图标即可展开Firebug的界面。小小的界面囊括了很多功能，我虽然不能一一详述，但我想着重指出一些对调试工作特别有帮助的关键功能。

在图1-3中见到的是控制台标签页。JavaScript错误信息、Ajax调用、性能分析结果、命令行执行结果都会显示在控制台的界面上。你可以展开对象观察它的属性，单击错误信息会把你带到源码中出错的那一行，展开Ajax调用可看到请求及响应的信息，还可以分析性能分析的结果，找出错误的来源。

HTML、CSS和Script标签页可让你观察到每个元素的状态。你还可以进行修改，而修改结果会在Firefox窗口中实时地显示出来。请记住在这里进行的修改都是暂时的，刷新页面或关闭窗口之后就会丢失，绝对不会修改原始的文件。

DOM标签页显示DOM树和所有的属性。Net标签页（见图1-4）显示所有文件请求以及加载

每个文件的时间。你可以根据这些信息判断是否存在瓶颈。

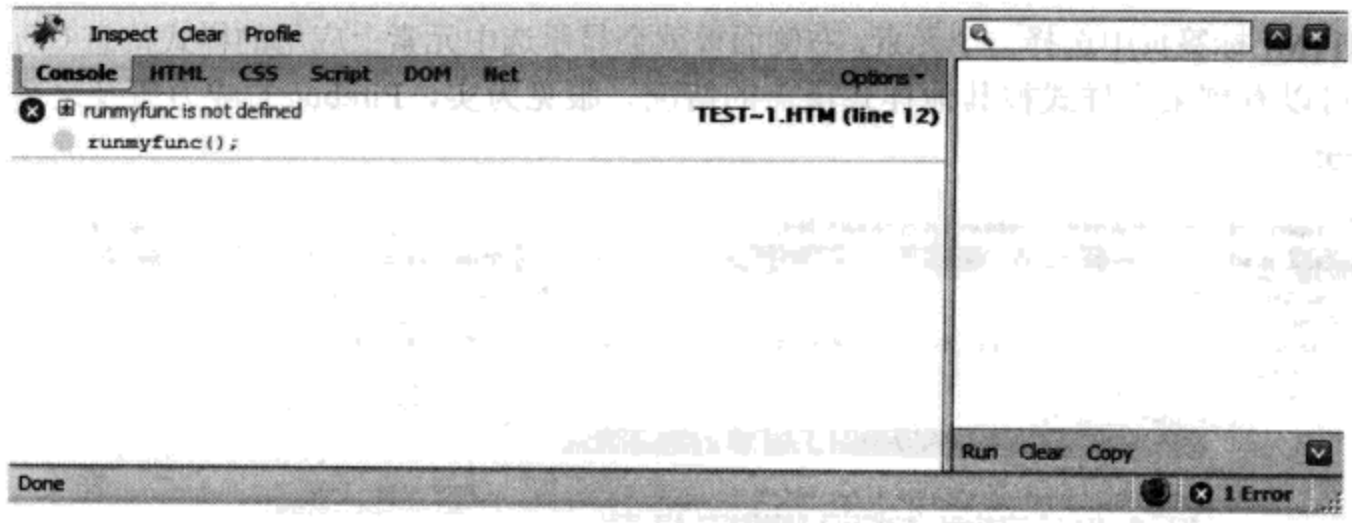


图1-3 Firebug控制台

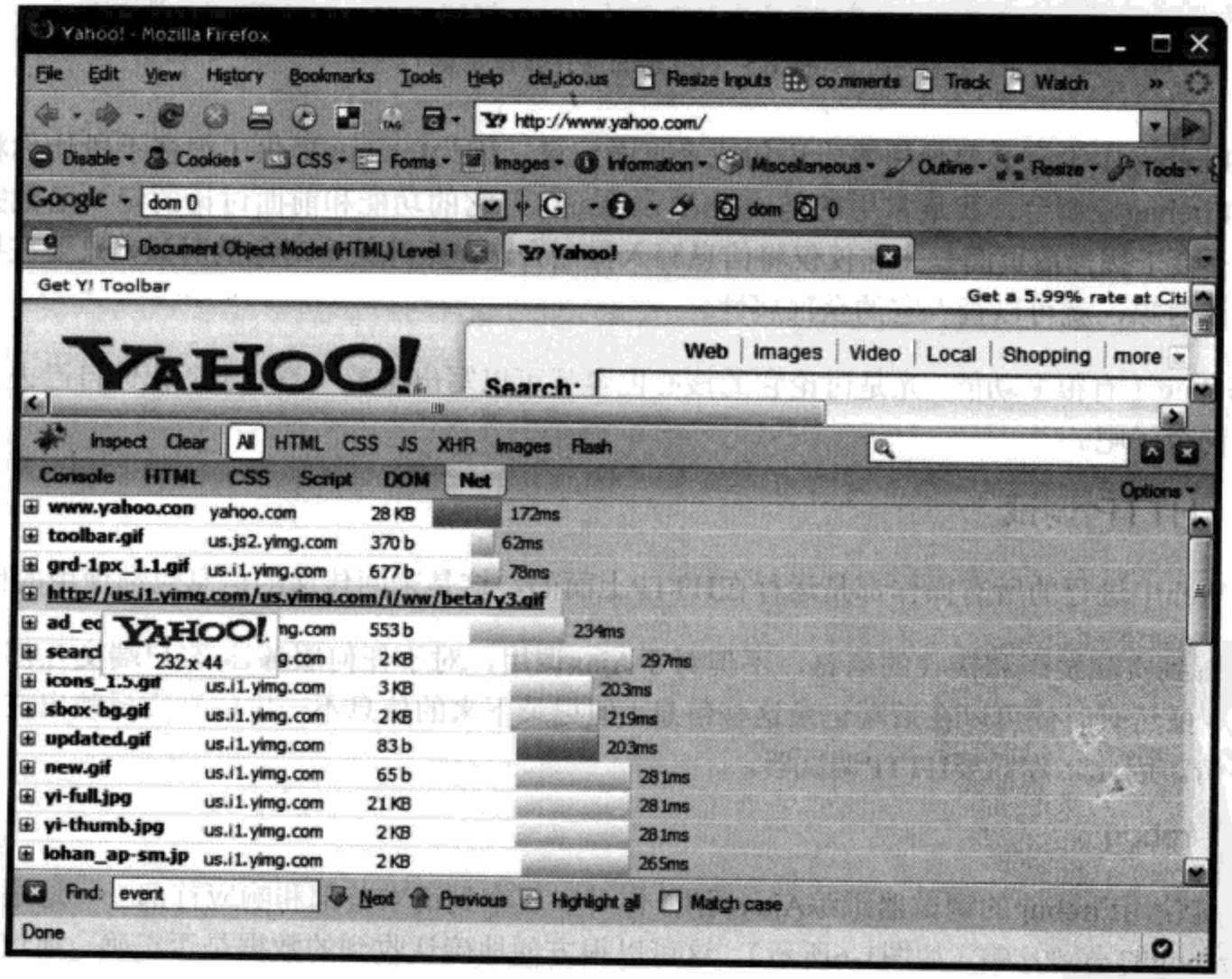


图1-4 Firebug中的Net标签页

主工具栏上有个Inspect按钮，你会经常用到它（至少我是!）。按下这个按钮之后，把鼠标移到HTML页面上，Firebug会随着鼠标移动突出显示相应的元素，HTML标签页下列出的元素也会

跟随着突出显示。

在HTML标签页中选择一个元素，右侧面板就会显示选中元素上应用的样式信息（见图1-5）。你甚至可以看到某个样式被其他样式覆盖的情况。眼见为实，Firebug的威力所及之处远不止JavaScript。

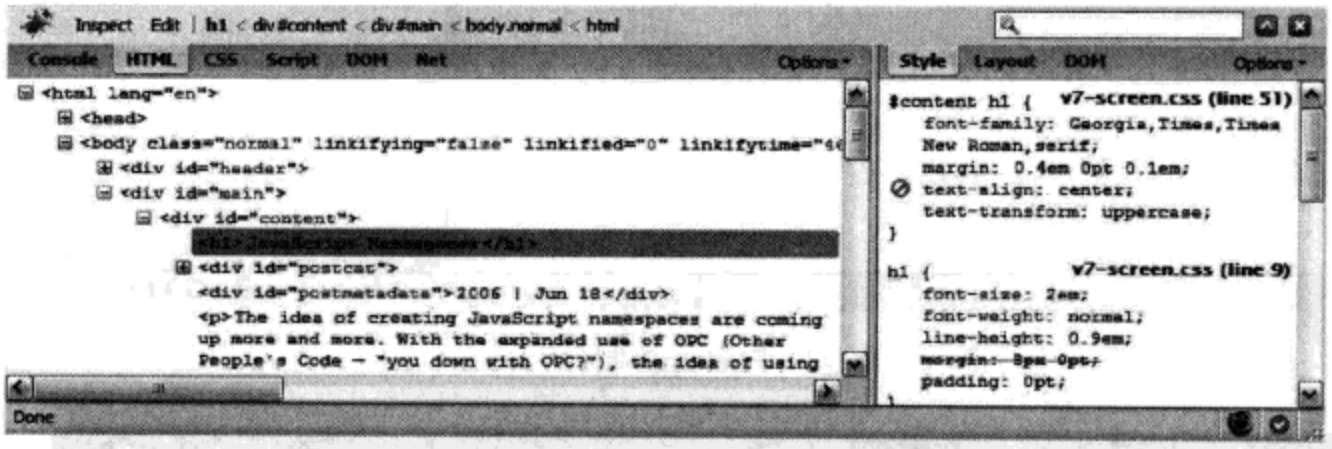


图1-5　在Firebug中选择元素

在Firebug中没有必要使用警告语句或者页面记录，因为Firebug提供了很多手段让你将信息记录到Firebug控制台。我最常用的是`console.log()`，它的功能和前面讨论的记录函数完全一样，但不会干扰当前页面——它仅仅将信息写入控制台。在跟踪某个对象的时候，只要在控制台中单击该对象，就可以查看它的全部属性。

Firebug还有很多功能，光是讨论它的核心内容就可以写出整整一章。还是让你自己去发掘那些闪光的金子吧。

1.5.4　HTTP 调试

在Web中进行的所有操作都是运行在HTTP上面的，这是来回传递的信息包都使用的协议。

能看到实际发送和接收的信息，不但对于Ajax调用，对于任何服务器/客户端交互都是很有帮助的。虽然有时候可以在后端记录这些信息，但记录下来的信息不一定能真实反映出在前端发生的事情。因此，你需要HTTP调试器。

1. Firebug

通过使用Firebug的调试器跟踪Ajax调用，你可以观察到请求首部和响应首部——这进一步说明了Firebug的不同凡响（如图1-6所示）。这可以很方便地确认收到的数据是否正确。通过对调用的检查还能看到向服务器发送了什么数据，以及从服务器收到了什么数据。

2. Live HTTP Headers

如果要进行更细粒度的HTTP请求分析，我建议使用Live HTTP Headers（http://livehttpheaders.

mozdev.org)。这个Firefox扩展能显示所有HTTP请求的请求/响应信息，不但便于进行Ajax调用(见图1-7)，而且便于监视页面请求（包括表单数据）、重定向甚至Flash中发出的服务器调用。它还能重发指定的请求，甚至在重发之前，还允许你修改请求首部，大大方便了对各种情形的测试。

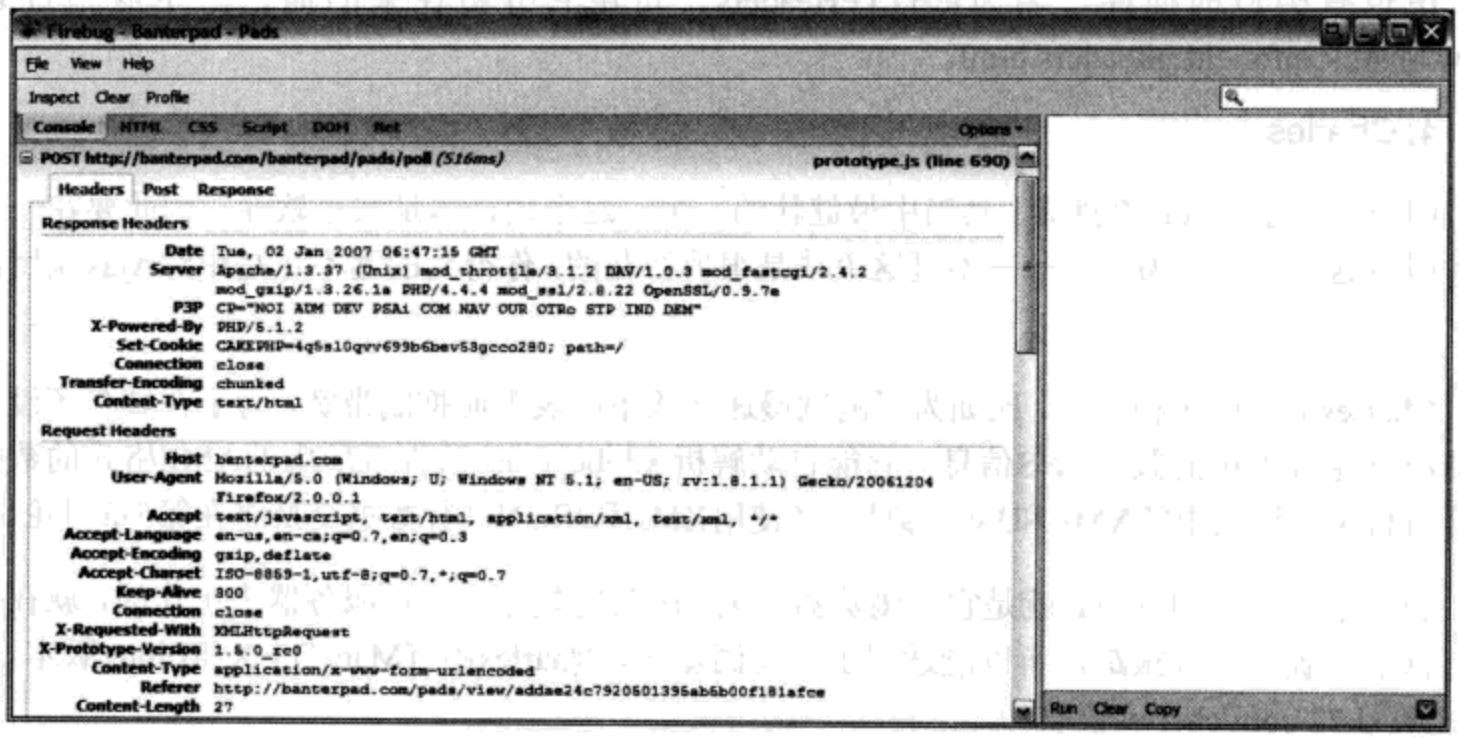


图1-6 Firebug中的Ajax调用检查

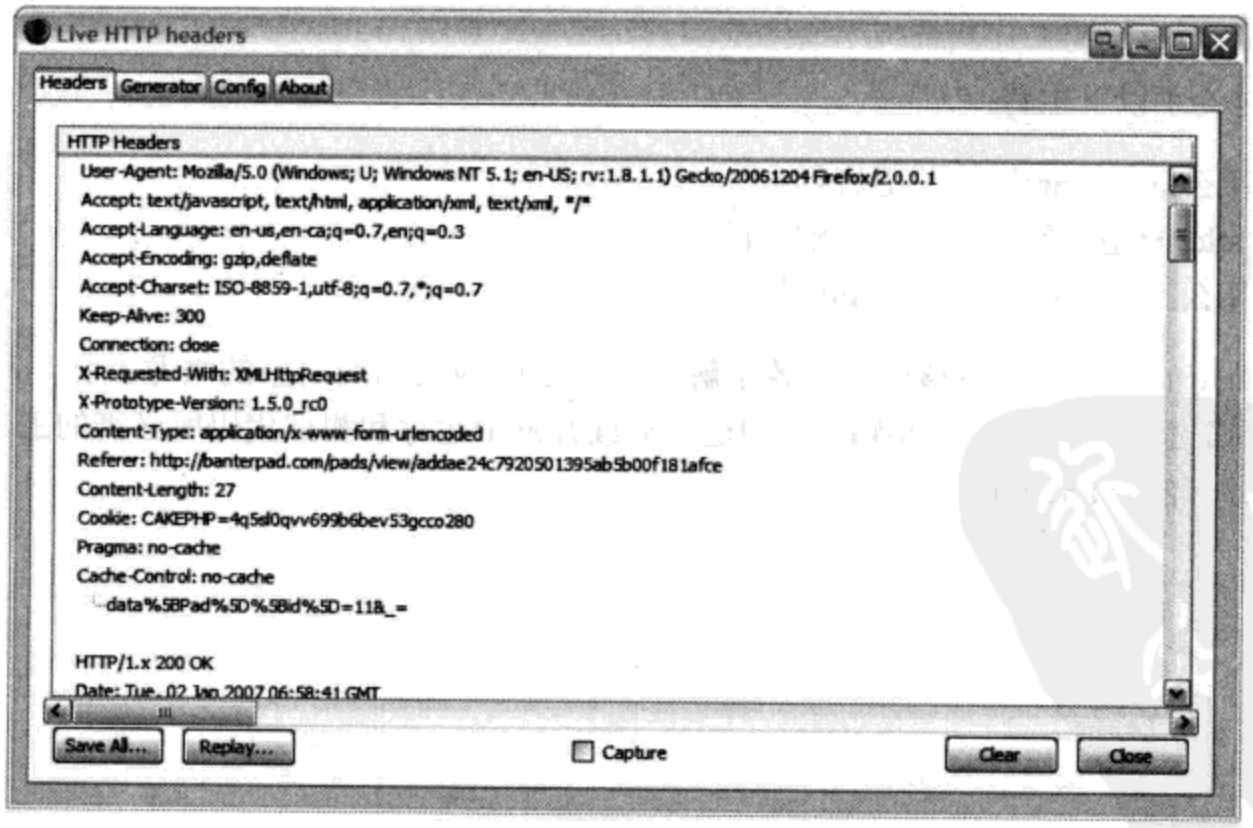


图1-7 在Live HTTP Headers中检查Ajax调用

Firebug在响应信息的分析方面更出色，因此将它和Live HTTP Headers结合使用能让你得到

更全面的分析。

3. ieHTTPHeaders

IE也有类似的插件，名为ieHTTPHeaders，可用它分析往来的通信。下载地址是：www.blunck.info/iehttpheaders.html。

4. Charles

Charles可能是HTTP调试工具当中最健壮的一个。这个调试器是共享软件，因此要花一点小钱才能将这件工具收为己有——不过这笔钱是很值得花的，你得到的远不止是跟踪Ajax调用的能力。

Charles能完成多种任务，比如为了测试慢速连接下的表现而抑制带宽，为了在域名还没上线的情况下完成测试而假冒DNS信息。它能自动解析Adobe Flash远程调用中的AMF格式的数据，也能解析Ajax调用中的XML和JSON数据。（使用XML和JSON进行数据交换将在第5章讨论。）

Charles还有一项优点，就是它与浏览器无关。因为它是作为代理服务器来工作的，就在代理服务器的位置上进行跟踪，所以能适用于任何浏览器。Charles还有Mac OS X和Linux版本。（可从www.xk72.com/charles获取。）

1.6　小结

本章讨论了以下主题：

- 为什么JavaScript比以前更流行。
- JavaScript是如何在浏览器中解析的。
- 用什么工具来调试JavaScript代码。

经过简短的介绍，你应该已经大致了解了JavaScript成为超级明星的来龙去脉，基本理解了把代码放到页面中时要注意的事情，也知道了后续开发中运行和测试代码所必需的工具。你已经被武装成为一名JavaScript武士了！

第2章 HTML、CSS和JavaScript

本章涵盖HTML、CSS（层叠样式表）以及如何通过DOM（文档对象模型）来访问页面元素和属性。讨论的内容包括事件处理、创建新元素和调整文档内容的样式。你将学到如何利用HTML、CSS和DOM脚本加快开发速度和降低维护的难度。

虽然你应该已经掌握了HTML和CSS，但我还是要先讲述一些基础的内容。同时还会给出一些DOM脚本编写和Ajax开发的技巧。

2.1 基础知识

谈到Web应用程序开发，没什么比HTML更基本的了。HTML是其他一切的基础，所以一定要掌握得更牢固一些。能拿起这本书，你应该已经对HTML有相当的了解，所以什么是元素这么基本的知识，这里就不再赘述。本章要回顾的是一些对下一步学习有重要意义的基础知识。

Web 标准

很多讨论Web标准和CSS的书籍都会强调内容和样式的分离，除此之外，正确地使用Web标准能大大简化Web开发。在老式的Web开发里（我是指你在1990年代学到的那些技术，比如表格和<font>标签）HTML被当作一种展示语言。只要能让设计不走样，人们会毫无顾忌地在代码里加入各种东西。这样做问题很大，因为网站会变得杂乱无章，更新起来极为困难——尤其是当开发者空降到一个从未接触过的项目的时候。

Web Standards Project（WaSP，见www.webstandards.org）在Jeffrey Zeldman和Molly Holzschlag等人的引领之下，着手寻求一种Web开发的新方式，以图让人们更便捷地开发出他们心爱的网站。正确使用Web标准有三个方面：

- 用CSS处理外观表现；
- 编写合乎规范的HTML；
- 编写有语意的HTML，给内容增添意义。

你可能在想JavaScript到哪里去了？你大概听过Web开发的“三层论”：HTML负责结构，CSS负责样式，JavaScript负责行为（即动态机能）。这绝对是一个需要牢记的重要观念，无论是在阅读本书的过程中，还是在你日常的Web开发工作里。

人们谈论Web标准的时候一般都会提到分离内容（HTML）和外观（CSS）。分离行为（JavaScript）也同样重要。分离行为使你能够一小块、一小块地给程序增加功能，从而简化程序的升级工作，另外还可以减少总体的带宽消耗。这种分离的JavaScript称为不唐突的（unobtrusive）JavaScript。图2-1中的文氏图描绘了这种分离关系，三者的交集表示实践中能达到的最佳效果。

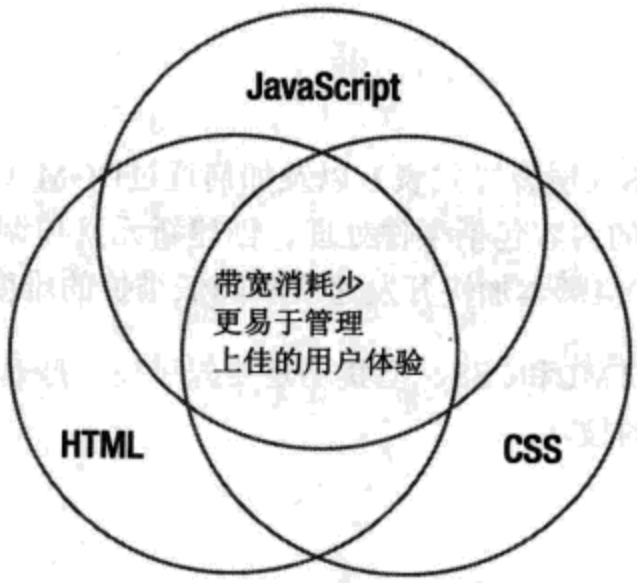


图2-1　现代Web开发的三大元素

1. 用CSS处理外观

从CSS Zen Garden（http://csszengarden.com）等网站的榜样可以看出，CSS完全有能力处理最复杂的设计。本章后面乃至全书，都会介绍高效使用CSS的一些方法。

2. 合乎规范的HTML

浏览器的容错能力使很多人（包括我在内）依赖于浏览器对特定代码的呈现方式，而不去确定HTML本身是否正确。随着时间发展，会出现新的浏览器，而浏览器在呈现不规范代码上的差异使开发网站就像碰运气。编写合乎规范的HTML，能保证不管现在还是将来，页面的呈现方式都与设计意图保持一致。合乎规范的HTML意味着要按照W3C（http://w3.org）制定的HTML或XHTML规范来编写代码。

QUIRKS模式和严格模式

编写合乎规范的HTML还有另一个出发点。有些浏览器当遇到不规范的HTML时，就会切换到“Quirks模式”，在这种模式下浏览器会用特殊的方式去呈现页面，目的是与旧的浏览器保持兼容。但是此时CSS不再按照规范处理，使得调试和查错更加困难。编写合乎规范的

HTML，浏览器就可以用“严格模式”处理页面。这种模式是以符合W3C规范为目标的，因此更能保证页面在不同浏览器上呈现一样的效果。

如果想检查自己编写的HTML是否合乎规范，很多开发工具都有内建的检查功能，比如Adobe Dreamweaver。也可以用W3C HTML Validation Service（http://validator.w3.org）来检查。

3. 有语意的HTML

编写有语义的HTML有着决定性的意义，但经常由于偏重于编写合乎规范的HTML而被忽略。使用语义正确的HTML意味着应该根据包含的内容选择适当的元素。

这样做的理由有几点。其一，使用屏幕阅读程序等辅助技术的人浏览页面的时候会少一些障碍，页面内容读起来更自然。否则缺少条理分明的元素安排，整个页面就是一大块含混不清的文字。

其次，有语意的HTML还意味着给予用户更多的页面控制能力。很多设计者一想到用户会搞乱他们精雕细琢的大作就害怕不已，但请相信我，给用户自由是一件好事。设计者和开发者总以为自己知道事物该如何运作，以为自己清楚这里应该是一个按钮，那个文本框的大小足够用了。但用户可能会用自定义的样式表和工具（比如Greasemonkey插件就可以让用户在任何页面上运行自定义的脚本）来提高网站或应用的阅读和使用体验，以满足他们各自的需要。

只要是开发Web应用，就应该用有语义的HTML标签提高代码可读性，把CSS编写得更容易操控，方便插入额外的JavaScript脚本来摆弄它们。

有语义的HTML是什么样子的呢？例如，文档的主标题应该用<h1>标签来标记，子标题则用<h2>到<h6>的标签标记出来。段落用<p>标签，强调的文字应该用标签而非<i>。如果不明白为什么应该用取代<i>，请思考一下斜体的意义：它是描述呈现方式的，本身并没有具体的含义。而具有强调的语义。和<b>的区别也在于此。单单把文字加粗并没有增加文字的含义；如果要表示着重的含义，请用。

坚持写有语义的标记这一原则也可以应用到为元素选择类名上。 比 有意义得多，尤其在你决定改变错误消息的外观的时候。有意义的类名对于编写DOM脚本也有帮助。搜索名为error的元素比搜索名为boldRed的元素更容易理解。

2.2 HTML最佳实践

下面所说的最佳实践虽然不是什么真理，但都是我的经验积累，同时也会解释其背后的理由。理解决策背后的理由总是重要的，即使这些决策看似与一般的潮流背道而驰。最要紧的是选择最适合你自己的做法。如果Web开发有什么放之四海而皆准的真理，那就是条条道路通罗马。

2.2.1　HTML与XHTML

HTML实际上不止一种：HTML和XHTML。XHTML 1.0（大致上）是XML版的HTML 4.01，也就是最“新”一版的HTML标准。很多人提倡使用XHTML，因为这个W3C标准比HTML 4.01更新。

XHTML希望成为Web开发的理想选择，当然具备了很多优点。作为一种XML格式，它对代码本身的写法有更严格的要求。比如所有元素都要有相应的终结标签。要终结一个没有标签体的元素，比如图片元素，应该在右括号前加一个斜杠。

```
<img src="image.gif" alt="My image" />
```

XHTML还要求标签的名字用小写字母，而且所有的属性值都要括起来。能识别XHTML的浏览器会严格地提醒你不合规范的代码。严格的要求能助你写出合乎规范的代码。

在XHTML里，CSS和JavaScript也有一些变化。CSS中的元素选择符会区分大小写的。JavaScript返回的元素名称也会变成小写，而不是像HTML那样的大写名称。

虽然XHTML的优点不可忽视，其缺点也需要认真对待。发送给浏览器的XTML必须以MIME类型`application/xhtml+xml`加以标识，但不幸IE不支持这种MIME类型，而会试图下载文件会转交给其他程序去处理。XHTML 1.0可以用MIME类型`text/html`发送，但此时浏览器会把它当作普通的HTML来呈现。

当文档被当做XML来处理的时候，有些比较旧的DOM对象和方法（如`innerHTML`、`document.images`和`document.forms`）在某些浏览器中会不可用。

由于用XHTML开发会带来太多复杂情况，我建议使用HTML 4.01 Strict（见www.w3.org/TR/REC-html40）。本书中所有例子都是根据这个标准编写的。

2.2.2　两全其美

选择了HTML并不意味着就不能坚持XHTML的一些优秀原则。有几条规则特别值得遵守：保证用引号括起属性值；标签名称保持小写；保证正确地终结标签。XHTML里简单地加上斜杠来终结的标签，在HTML里不能采取同样的写法。比如图像（`<img>`）和断行（`<br>`）标签都不需要写上终结斜杠，但列表项（`<li></li>`）和段落（`<p></p>`）等元素都应该写上终结标签。

坚持XHTML风格的规则使代码易于阅读，易于查错，而且当未来的浏览器改善了对XHTML的支持时，也易于过渡到XHTML。

2.3 CSS基础

和前面谈HTML一样，我假设你懂得一般的CSS语法，也同样会回顾一下若干基础知识。我还将介绍一些能改善CSS管理的策略，尤其是与JavaScript交互方面的策略。

2.3.1 将含义表达出来

再次回到语义问题，前面提到过根据内容选择适当元素的好处，现在来说说背后的理由。下面是一个没有明确含义的例子：

```
<div>This is a header</div>
<div>This is some text content.</div>
<div>Here is some additional content with <span> emphasis</span> and➥
<span>strong emphasis</span>.</div>
```

站在CSS的角度，完全没办法为任何一个元素设立单独的样式。显然这种写法行不通，那么，我们来给它加上一点含义：

```
<div class="header">This is a header</div>
<div class="text">This is some text content.</div>
<div class="text">Here is some additional content with <span class="emphasis">➥
emphasis</span> and <span class="strong">strong emphasis</span>.</div>
```

看，现在代码有意义了。但还不够，你还远远没有发挥出HTML的天生威力。我们再试一次：

```
<h2>This is a header</h2>
<p>This is some text content.</p>
<p>Here is some additional content with <em>emphasis</em> and <strong>strong➥
emphasis</strong>.</p>
```

使用有语义的HTML实现了两项目标：

- 给文档添加了浏览器能理解的含义。从消除阅读障碍的角度来说，使用屏幕阅读程序或关闭了样式的用户现在能更好地理解文档的内容。
- 减少了HTML标记，简洁总是好的。在服务器和客户端之间往返的字节更少了，给你奖一朵小红花。Ajax流行的部分原因就是它能减少传输的信息量（见第5章）。CSS之所以流行（以及吸引我去学习CSS）也是出于同样的原因。不再需要到处加<font>标签，用CSS就能调整整个页面的样式。不仅如此，CSS文件还可以被缓存，加快后续页面请求的速度。

当你进一步深入使用CSS，还会发现特异性（specificity）规则的好处：如果到处使用同样的元素，有些工作会很难实现（下文会简短讨论特异性）。

2.3.2 元素的标识

HTML有两种属性可资识别一个元素：id和class。

id属性给元素赋予一个全页面唯一的名称，而且它有严格的命名规则。W3C规范陈述如下：

> ID和NAME标记必须以字母开头（[A-Za-z]），后跟任意数量的字母、数字（[0-9]）、连字号（‘-’）、下划线（‘_’）、冒号（‘:’）和句号（‘.’）。

id属性有几种用途：

- 作为样式表选择符（比其他选择符的特异性更高）；
- 用作超链接的目标定位点；
- 在DOM脚本编程里用作引用单一元素的手段。

class属性给元素赋予一个或多个类名（用空格分隔）。相同的类名可用于页面上的多个元素。类名的命名规则比ID宽松得多。字母、数字、连字号以及大多数Unicode字符都可任你处置。不过我建议仅使用字母、数字和连字号——太多符号只会徒增困扰。

提示　类名应有助于从语义角度描述内容。像“bluetext”这样的类名，一旦你决定把设计改成绿色就会变得很碍眼。用“callout”或者“caption”之类的名字可描述得更准确。

在开发应用的时候应该知道什么时候该用哪个属性。最简单的规则是这样的：如果某种类型的元素只会有一个，那就用id；否则应该用class。

来看一个简单的例子：

```
<div id="todolists">
   <div class="section">
      <h3>General</h3>
      <ul class="general">
         <li>Groceries</li>
         <li>Dry cleaning</li>
         <li>Buy books</li>
      </ul>
   </div>
   <div class="section">
      <h3>Programming</h3>
   <ul>
      <li>Finish project</li>
      <li>Make cool examples</li>
      <li>Write article for site</li>
   </ul>
   </div>
</div>
```

可见，每一组标题和列表分别放在一个类名为section的 里，然后用一个设置了ID的 来囊括全局。

2.3.3 应用CSS

CSS可在不同层次上定义，主要有以下3种位置：

- 外部样式表（在文档头部使用<link>元素关联到HTML）；
- 在HTML文档内部通过<style>元素定义；
- 在元素层次通过style属性定义。

在以上三种层次中，后面的层次会取代前面的。也就是说在style属性里声明的样式会覆盖掉前面定义的样式，而在<style>元素中声明的样式会取代外部样式表中的定义。在一般情况下，我建议把所有CSS声明放置在一个或多个外部样式表里。这样会比较容易组织和重用它们，而且在做DOM脚本编程的时候也更易于管理。

1. 继承

继承意味着子元素会从父元素自动获得特定的CSS属性，有利于精简代码，提高效率。

以先前的HTML片段为例，给段落指定颜色同时意味着段落内部的强调文本也会继承得到同样的颜色：

```
<h2>This is a header</h2>
<p>This is some text content.</p>
<p>Here is some additional content with <em> emphasis</em> and <strong>strong➥
emphasis</strong>.</p>
```

再举一例，如果在<body>元素上声明了font-family样式，那么页面中的所有元素都会继承。同一种元素处在不同的位置，可以有不同的样式。请看下面的HTML：

```
<div id="main"><h2>This is a header</h2></div>
<div id="sidebar"><h2>This is a header</h2></div>
```

在样式表里可以声明如下样式：

```
#main h2 { color:red; }
#sidebar h2 { color:blue; }
```

在main区块中的标题会是红色，而sidebar区块中的标题则是蓝色——尽管两次选择的是同一种类型的元素。

2. 特异性

由于可以用几种类型的选择符来声明样式，所以需要定义一套规则来确定声明之间的优先级。特异性（Specificity）是一个需要重点掌握的概念。网站和应用越复杂，正确定义元素样式所需的选择符也越复杂。特异性分成4个级别来计算：

A. 若选择符为style属性，得分为1。用style属性定义的规则优先级最高。

B. 计算选择符中id属性的数量。
C. 计算选择符中其他属性的数量［包括类和仿类（pseudoclass）的数量］。
D. 计算选择符中元素名的数量［包括仿元素（pseudo-element）名］。

我们来看下面的举例，每个例子都比前一例的特异性更高（见表2-1）：

表 2-1　特异性示例，举例说明选择符的权重

声　明	A	B	C	D
.list {}	0	0	1	0
#todolist {}	0	1	0	0
#todolist .list {}	0	1	1	0
#todolist ul.list {}	0	1	1	1
body div#todolist ul.list {}	0	1	1	3
#pagetodo #todolist {}	0	2	0	0

判定特异性的高低考虑两项要素：

- 在同一个级别里，数字越大特异性越高。比如你在一条规则里用了3个类选择符（C级），在另一条规则里用了2个类选择符（同样是C级），那么前一条规则的特异性比后一条高。
- 高级别的选择符特异性总是高于低级别的选择符，无论数量如何。比如一条规则里有一个id选择符（B级），另一条规则里有3个类选择符（C级）和一个元素选择符（D级），那么前一条规则的特异性比后一条高。

如果发现给元素指定的样式没起作用，你可能会想直接用!important关键字强行应用样式。我建议最好避免这种做法，因为它会限制后续代码的灵活性（更高特异性的规则也必须加上!important关键字才能起作用）。

还有一个保持简单的小建议，那就是极可能用最少的选择符来定义元素样式。以后遇到特异性问题的时候再考虑按需要增加选择符。

我们来看一个简单的例子，首先是一段HTML：

```
<div id="main">
   <p class="intro">It's a fine morning today.</p>
   <p>Yes. It is a fine morning.</p>
</div>
```

可以用这样一段CSS来调整它的样式：

```
p { color:red; }
p.intro { color:blue; }
#main p { color:green; }
```

你可能会惊讶地发现intro段是绿色的，而不是预料之中的蓝色。绿色是因为第三行用了ID选择符，优先级比第一行的元素选择符、第二行的元素加上类选择符都要高。因此，要想让intro段按照设计意图呈现蓝色，至少要加上一个ID选择符。

```
#main p.intro { color:blue; }
```

提高特异性的基本法则是先理清当前的特异性到底由哪一个级别决定（A、B、C还是D）。然后在当前级别上增加特异性，或者改用更高级别的选择符。如果一条规则用了两个类选择符，要想高过它，至少需要用3个类选择符或者一个ID选择符。如果规则里用了一个ID选择符和一个元素选择符，那么高过它就需要至少一个ID选择符和一个类选择符，或者用一个ID选择符和两个元素选择符，当然两个ID选择符也行。

2.4 JavaScript 基础

如果曾经编写过JavaScript程序，那么下面讲述的内容你大概已经有所了解。不过我们还是先来复习一下基本的术语和JavaScript概念，因为它们对理解本书的内容非常重要。

2.4.1 函数

函数把一系列的命令打包成一次调用。函数让你把代码封装成一个一个任务，其后可以通过各种方式重用它们（第3章谈到JavaScript面向对象编程的时候会重点讲述）。例如：

```
function foo(){ } // 这是一个函数
```

函数可以是匿名的，也就是没有名字。匿名函数就像风中的精灵。下面是匿名函数的例子：

```
function (){ } // 这是一个匿名函数
```

匿名函数在JavaScript面向对象编程中很常用，因为它既可以避免命名冲突，又能把只与对象本身有关的代码隐藏在对象内部。

JavaScript中的函数比很多主流语言中的同类要趾高气扬一点：它们是一等公民。也就是说可以把函数赋值给变量，可以作为参数传递给其他函数，可以作为函数的返回值，可以存储在数组里，还可以作为对象的属性。

2.4.2 对象、属性和方法

对象里包含着变量（称为属性）和函数（称为方法）。JavaScript在这方面非常强大，可以在任何时刻给对象添加新的属性和方法。函数可以为对象搭建出结构。我们来看一个例子：

```
function foo(){ }
var bar = new foo();
```

扩展这个例子，给两个对象都增加新的属性：

```
function foo(){ }
var bar = new foo();
foo.value = 5;
alert(foo.value); // 显示值属性"5"
bar.value = 6;
alert(bar.value); // 显示值属性"6"
```

添加新方法也是类似的：

```
function foo(){ }
var bar = new foo();
foo.value = 5;
alert(foo.value); // 显示值属性 "5"
bar.value = 6;
alert(bar.value); // 显示值属性 "6"
function myfunc(){ }
bar.mymethod = myfunc; // 为函数赋值
bar.mymethod(); // 调用方法
```

还可以换一种写法，用匿名函数：

```
function foo(){ }
var bar = new foo();
foo.value = 5;
alert(foo.value); // 显示值属性"5"
bar.value = 6;
alert(bar.value); // 显示值属性"6"
bar.mymethod = function (){ }; // 为函数赋值
bar.mymethod(); // 调用方法
```

用匿名函数意味着不必担心myfunc()函数与页面中的其他对象或者变量发生冲突，同时代码也更清晰。

注解　在JavaScript里，函数就是对象。在上面的例子里，我给foo函数添加了一个value属性。

2.4.3　点号语法和方括号语法

JavaScript提供了两种方式来访问对象的属性。前面的例子用了点号语法。如果用过Java或者C++语言编程，应该对点号语法非常熟悉。你还可以用点号语法将命令串接起来（我在操作字符串是时候经常用到这种手法）。

比如你得到一个用户输入的字符串，在将它传给搜索引擎之前要先做一些清理工作：

```
// 下面语句的结果是"what up dog"
"What up, dog!".toLowerCase().replace(/[^a-z0-9 ]/g/");
```

方括号语法也差不多，只不过像数组一样用方括号来引用对象的属性。上一节的例子如果改成方括号语法会是这个样子：

```
alert(foo["value"]); // 应该看到的是"5"
```

甚至方法调用也可以加上方括号（参数跟在后面）：

```
foo["mymethod"]();
```

方括号语法也一样可以把命令串接起来：

```
// 下面语句的结果同样是"what up dog"
"What up, dog!"["toLowerCase"]()["replace"](/[^a-z0-9 ]/g/"");
```

这种写法比较难读一点，所以多数人会坚持用点号语法。不过方括号语法的好处是可以通过变量来执行函数：

```
function manipulateString(str, func)
{
    return str[func]();
}
newstring = manipulateString("WHAT UP", "toLowerCase"); // newstring = "what up"
```

后面两章会给出一些实际的应用。

2.4.4 原型

JavaScript是基于原型的语言，因此本质上是通过克隆已有的对象来创建新对象。这还意味着如果给原型增加了新的属性和方法，那么源自同一个原型的所有对象都会获得新的属性和方法，即使是增加前已经克隆的对象。这是通过对象的prototype属性实现的。

让我们继续改写前面的例子：

```
var foo = function(){ }
var bar = new foo();
foo.value = 5;
alert(foo.value); // 显示值属性"5"
bar.value = 6;
alert(bar.value); // 显示值属性"6"
bar.mymethod = function (){ }; // 为函数赋值
bar.mymethod(); // 调用方法
foo.prototype.othervalue = 6;
alert(bar.othervalue); // 显示"6"
```

如上所示，我给原始对象foo的原型添加了新的属性，而在已有对象bar里一样可以使用新增的属性。（第3章会详细讨论这个主题。）

2.4.5　值传递和引用传递

给函数传递参数有两种方式：值传递和引用传递。传递变量的时候，变量值会被复制一份然后在函数中使用。对变量（副本）的修改只反映到函数内部。在函数外部的原始变量不受影响。这就是值传递：

```
var foo = 5;
function bar(val)
{
   val = 6; // 将其改为6!
}
bar(foo);
alert(foo); //仍为5
```

对象作为参数传递的时候是引用传递。也就是说可以访问到对象的全部方法和属性，而且对对象的任何修改都会反映到函数外部。

```
var foo = function(){};
foo.prototype.value = 5;
function bar(obj)
{
   obj.value = 6; // 将其改为6!
}
bar(foo);
alert(foo.value); // 现在值为6!
```

那么，传递一个函数又是什么情况呢？

```
var foo = function(){};
foo.prototype.value = 5;
foo.prototype.addValue = function(){ foo.value = 6; }
function bar(func)
{
   func(); // 正在运行函数
}
bar(foo.addValue); // 传入函数
alert(foo.value); // 现在值为6!
```

例子里有几点需要注意。首先，传递函数的时候不要带上圆括号，因为可以传递一个函数而不一定要执行函数中的代码。虽然例中看得不清楚，但实际上是先将函数复制了一份（就像传递变量一样）再传递进去。稍后我会详细介绍对象引用和具体情况下的一些注意事项。

如果带了圆括号，函数会立即执行，然后将函数的返回值传递进去。下面是一个简单的示例：

```
function foo()
{
   return 6; // 返回一个值
}
```

```
function bar(val)
{
   alert(val);
}
bar(foo()); // 显示 6
```

2.5 JavaScript 与 DOM

JavaScript是赋予HTML和CSS生命的魔术师！JavaScript语言比较简明易懂，因此我假定你已经对JavaScript有所了解，或者至少了解JavaScript的基本语法。

JavaScript的威力在于它能够运用多种技术。浏览器提供了很多接口，比如window对象、XMLHttpRequest对象，还有document对象。window对象和XMLHttpRequest对象我们稍后再详述。现在要看的是DOM，JavaScript用来理解HTML文档并与之交互的接口。

2.5.1 DOM 是什么

DOM是一个应用程序编程接口（API），它定义了一组对象以及其中的属性和方法。DOM对于XML和HTML是通用的。

DOM实际上由多个标准组成，虽然我把它们当作一个大标准来说，但其实是3个构成要素各不相同的标准。

- DOM Level 0，实际上并没有这么一个W3C推荐标准，只是用它来指代IE第3版及Netscape第3版所具备的特性。
- DOM Level 1包括核心标准——也就是XML和HTML，以及一项针对HTML的扩展。该扩展还涵盖了向后兼容DOM Level 0特性的需要。DOM Level 1涵盖的许多HTML特性都得到了广泛而一致的浏览器支持。
- DOM Level 2加入了更多的XML和HTML扩展，并且支持操作样式信息、事件和选择范围（Range，该特性方便了在浏览器中做WYSIWYG编辑）。各浏览器从这一层开始，对如何实现标准出现了分歧。
- Mozilla开发的浏览器如Firefox一直紧跟W3C规范，IE则走上了另一条路。比如IE的事件处理机制，其实现可以追溯到1999年DOM Level 2成为推荐标准之前。很不幸，直到现在IE仍然沿袭最初的实现，一直都没有更新过。
- DOM Level 3最核心标准以及事件处理作了扩展，不过其中大部分规范都还没达到推荐标准的程度，而且几乎没有浏览器实现其中的任何规范。

就目前来说，遵守Level 1，再加上一点点Level 2就足够了。

注解　虽然以上标准常被称为Web标准，但正式的叫法是推荐。W3C委员会的成员协力开发出一套推荐标准，供全体成员遵循。

2.5.2　DOM树的结构

DOM被表示成一个树结构。在HTML里，如果一个标签处在另一个标签的内部，那么它在DOM里就被视为一个子元素。

```
<body>
   <div class="intro">Here is some text
      <p>More text</p>
      <p>More text</p>
   </div>
</body>
```

图2-2用图形表示出了以上HTML片段在DOM中的结构。

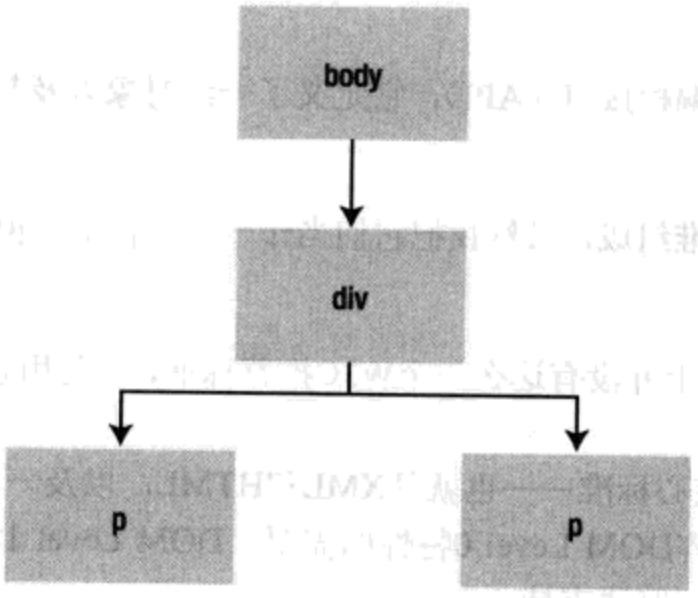


图2-2　简单的DOM树示意图

DOM还有一些不好掌握的细微之处。例如，DOM把元素看作一种结点类型，只不过DOM有12种结点类型，元素只是其中一种。好在大多数只有从事XML的人才需要关心，对付HTML懂得三种最常用的结点类型就够了：元素、属性和文本。

表2-2列出了相关的结点类型。

表2-2　结点类型和对应的结点类型ID

描　述	结点类型
Element（元素）	1
Attribute（属性）	2

（续）

描　述	结点类型
Text（文本）	3
Comment（注释）	8
Document（文档）	9

也就是说上一幅图还不够精确，因为没画出属性和文本结点。图2-3才是完整的DOM树示意图。

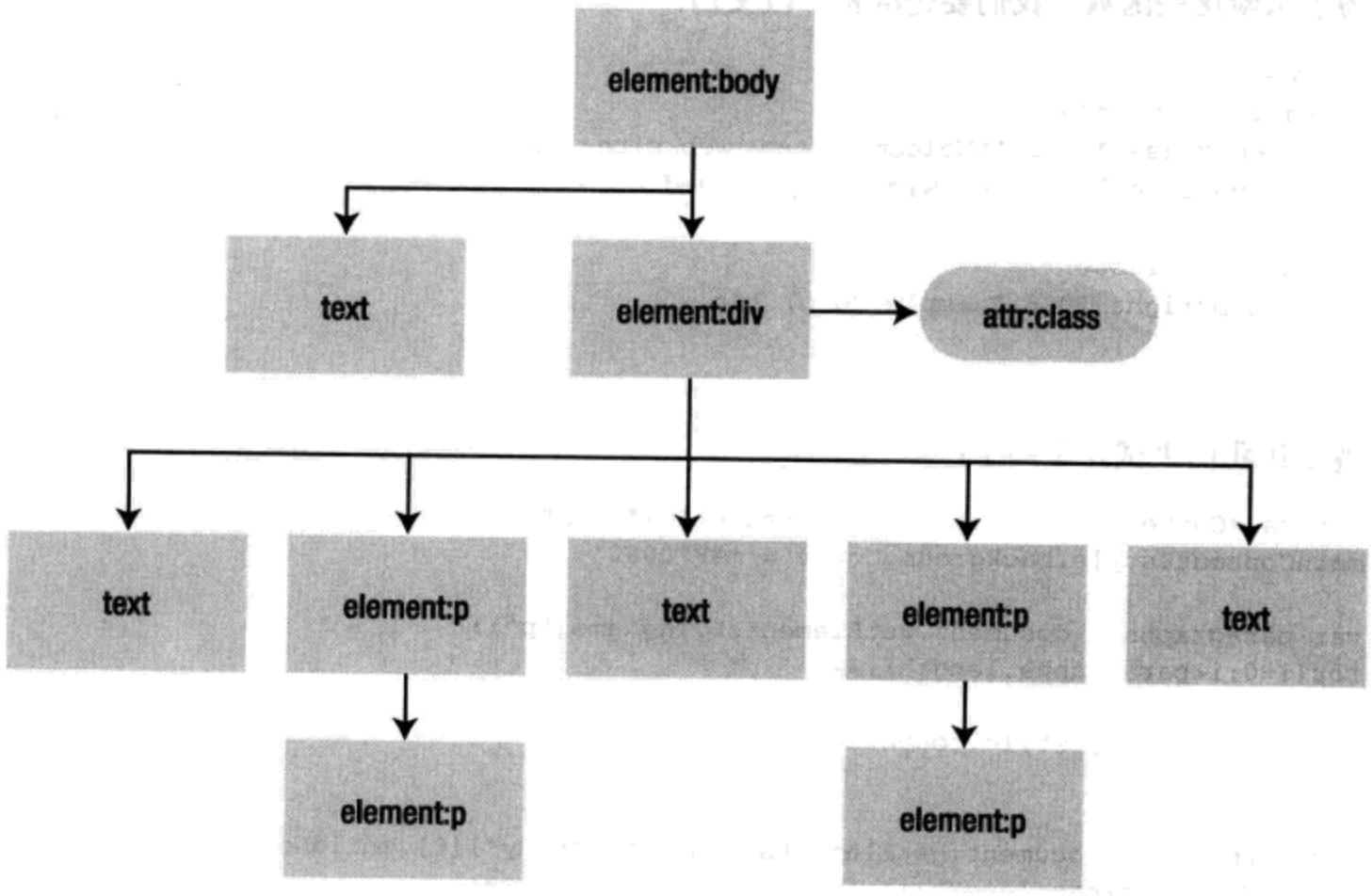


图2-3　DOM树示意图，增加了属性及文本结点

如图所示，整个树结构又多出了一些枝杈。请记住（很重要的一点），即便只是标签之间的空白也算作文本结点。

IE照例特立独行，不把空白算作结点。当你遍历DOM树的时候，请记住这点浏览器差异（本章后面会再详述）。

2.5.3 document 对象

现在知道了什么是DOM，让我们接着学习怎么用它（当然是通过document对象）。通过document对象可以引用到页面中的任意元素，增加新元素，以及删除已有的元素。

处理文档的时候，有几个函数可以用来获取一个或多个元素，最常用的是以下3个函数：

- getElementById()，从页面中获取单个元素。
- getElementsByTagName()，获取具有指定标签名的所有元素。按照W3C规范，HTML处理器会假定标签名是大写的。但对于现在的浏览器，大小写都没有问题。而在XHTML里，标签名必须为小写。因此我建议你使用小写。
- childNodes，结点的一个属性，可取得元素的所有直接子结点。

为了试验这些函数，我们要先准备一段文档：

```
<body>
   <div id="main">
      <p class="intro">Welcome to my web site</p>
      <p>We sell all the widgets you need.</p>
   </div>
   <div id="footer">
      Copyright 2006 Example Corp, Inc.
   </div>
</body>
```

现在让我们来试几个例子：

```
var mainContent = document.getElementById("main");
mainContent.style.backgroundColor = '#FF0000';

var paragraphs = document.getElementsByTagName("p");
for(i=0;i<paragraphs.length;i++)
{
   paragraphs[i].style.fontSize = '2em';
}

var elements = document.getElementsByTagName("body")[0].childNodes;
for(i=0;i<elements.length;i++)
{
   if(elements[i].nodeType == 1 && elements[i].id) alert(elements[i].id);
}
```

首先，我们取得ID为main的 ，把它的背景颜色改成红色。然后取得所有段落元素，逐一遍历，把每一个段落的字体大小都改成2em。最后取得<body>标签，循环遍历它的全部子结点，用警告对话框把每一个元素的ID都依次显示出来。

最后的例子用了几个技巧。第一，我们获取的是所有标签名为body的元素，HTML文档只会有一个body元素，所以可以用从[0]集合里获取第一个（也是唯一的）元素。其次我们检查了结点类型是否等于1。每种结点类型都对应着一个数字。元素结点是1，文本结点是3。属性也是一种结点，但它不能通过这种方式获取，而只能通过getAttribute()指定想取得的属性，或者通过attributes访问全部属性。

2.5.4 通过类名获取元素

前面介绍过元素的标识可以通过ID（方便用getElementByld()获取），也可以通过类名。但遗憾的是，规范里并没有定义getElementsByClassName()函数。根据类名来获取元素是快速取得同类元素的高效手段，所以让我们自己来编写这个函数：

```
function getElementsByClassName(node, classname)
{
    var a = [];
    var re = new RegExp('(^| )'+classname+'( |$)');
    var els = node.getElementsByTagName("*");
    for(var i=0,j=els.length; i<j; i++)
        if(re.test(els[i].className))a.push(els[i]);
    return a;
}
```

函数有两个参数：查找的起始结点、要查找的类名。函数返回一个元素数组，遍历它就可以访问到每一个元素。

我们来详细分析一下这个函数。首先创建一个正则表达式对象：

```
var re = new RegExp('(^| )'+classname+'( |$)');
```

正则表达式是一种用于字符串匹配的语法。它的功能很强，但也不容易理解。我时不时会为了理顺一条正则表达式而绞尽脑汁。（本章后面会详细解说正则表达式）。即使元素有多个类名，这条正则表达式也能匹配。它首先查找类名的开头，类名要么从字符串的第一个字符开始，要么跟在一个空格之后。接着表达式查找是否存在函数参数中指定的类名。最后查找类名的结尾，也就是一个空格字符或者字符串的结尾。“^”匹配字符串开头，“$”匹配结尾。“|”与JavaScript运算符“||”类似，这部分检查字符串是否与“|”左右任一个字符匹配。

接下来取得起始结点下的所有元素：

```
var els = node.getElementsByTagName("*");
```

前面已经介绍过getElementsByTagName()方法，用星号（“*”）作为它的参数意味着返回全部元素。

遍历元素集合，检查每一个类名是否与正则表达式相匹配：

```
for(var i=0,j=els.length; i<j; i++)
        if(re.test(els[i].className))a.push(els[i]);
```

className是元素的一个属性，保存着HTML中class属性的取值。用正则表达式来检验类名，如果匹配就会返回true。如果为true，将当前元素放入数组。

最后，全部元素检查完毕，函数返回元素数组：

```
return a;
```

2.6　在DOM中移动

取得一个元素之后，常常需要以它为参照在DOM树中上下左右地移动。一共有4种移动方法：

- childNodes，前文已经介绍过，childNodes可获取当前元素之下的所有结点。
- parentNode，获取当前元素的直接父结点。
- nextSibling/previousSibling，分别获取前一个和后一个结点。
- firstChild/lastChild，获取当前元素的第一个或最后一个子结点。

避开文本结点的问题

不同浏览器在处理文本结点上的差异使得在DOM中移动有些困难。以下面的代码为例：

```
<div id="node">
  <p>Some text.</p>
  <p>Some more text.</p>
</div>
```

认为 有两个子结点似乎很合理；在IE中正是如此。可是在其他主要浏览器中，都把标签之间的空白算作结点，因此 共有五个子结点而非两个。因此在从一个元素移动到另一个元素的时候，必须把这一情况计算在内，检查所在结点是否文本结点。例如：

```
var el = document.getElementById('node');
//取得第一个元素
var firstElement = el.childNodes[0];
if(firstElement.nodeType != 1) firstElement = el.childNodes[1];
```

如果发现第一个子结点不属于元素类型，就转而获取下一个子结点。我在此假定了起始标签和第一个元素之间只有文本结点，如果存在注释结点情况会更加复杂。为此我们需要设计一个可重用的函数来简化处理：

```
function getElement(node)
{
  while(node && node.nodeType !=1)
  {
    node = node.nextSibling;
  }
  return node;
}
```

如果传入的结点是元素，那么就跳过整个while循环。否则继续循环直到找到一个元素结点，或者在找不到更多结点的时候退出循环（此时返回null）。

现在可以把先前的例子改写成这样：

```
var el = document.getElementById('node');
//取得第一个元素
var actualFirstElement = getElement(el.childNodes[0]);
```

这样就能在所有浏览器上都得到一致的返回结果。

注解 HTML中的空白字符包括空格、Tab、换行、换页和回车。虽然看不见，但这些字符都要在文件中占据空间。浏览器呈现页面的时候，任何只包含空白字符的文本结点都不应呈现出来。浏览器还会将连续多个空白字符合并，只显示成单个空格，但<pre>标签中的内容除外。起始标签与第一个非空白字符之间的空白字符应被浏览器忽略。虽然各浏览器在如何呈现空白上是一致的，但通过JavaScript获取文本结点时对空白的处理却不一致。如果希望在不同浏览器中都用统一的方法操作文本结点，就必须先把字符串修整成统一的形式——截去字符串头尾的空白字符，把空格以外的空白字符替换成空格，然后把多个空格缩减成一个空格。

2.7 处理属性

处理属性有很多途径，其中元素有两个访问和设置属性的方法：getAttribute()和setAttribute()。以下面的HTML为例：

```
<a href="link.html" id="mylink">My Link</a>
```

获取href属性的代码如下：

```
var a = document.getElementById("mylink");
var href = a.getAttribute("href");
```

通过setAttribute()方法可以改变属性，代码如下：

```
a.setAttribute("href", "newlink.html");
```

DOM Level 1的HTML扩展允许用以下方式便捷地访问元素的属性：

```
var href = a.href;
```

这种方式更简洁，所以我比较偏好（你会发现我经常把简洁作为选择标准）。

请注意在不同浏览器上，href属性的行为有差异。如果直接用DOM属性href，浏览器会返回完全展开的URL。例如a.href会返回“http://example.com/link.html”。而通过getAttribute()方法的时候，IE同样会返回完全展开的URL，但Mozilla Firefox会直接返回HTML属性值，不作展开。因此为了保证结果一致，我会坚持使用a.href。

注解　attribute属性和getAttribute方法的返回结果有几点差异，大多数要归咎于IE的实现方式。例如getAttribute("class")看似正确，但由于IE的实现是直接把getAttribute方法映射到attribute属性，而class又是保留的关键字，所以getAttribute("class")是不行的，必须改成className。Tobie Langel和Andrew Dupont对此问题挖掘得很深入，请参阅Tobie的网站：http://tobielangel.com/2007/1/11/attribute-nightmare-in-ie。

2.7.1　style属性

DOM中的每个元素都有一个style属性，可用于动态调整元素的样式。所有CSS样式属性都可以在style属性中调整。

```
element.style.height = '100px'; // 高度设为100个像素
element.style.display = 'none'; // 把元素隐藏起来，不让用户看到
```

JavaScript不允许在方法和属性名中使用连字号，所以去掉了CSS属性名中的连字号，并且将第二个单词的首字母大写——这种大小写风格叫做“驼峰式（camel case）”。

```
element.style.backgroundColor = '#FF0000'; // 背景为红色
element.style.borderWidth = '2px'; // 边框为2px
```

也可以用CSS属性简写法：

```
element.style.border = '1px solid blue';
element.style.background = 'red url(image.gif) no-repeat 0 0';
```

随时间变化逐渐改变元素的样式属性可以产生动画的效果（在第6章中详述）。

2.7.2　class属性

为了避免与JavaScript类混淆，HTML属性class是通过className来引用的。先前定义getElementsByClassName()函数的时候已经出现过：

```
element.className = 'myclass';
```

不要低估用类名代替直接修改样式属性的优点。本书中介绍的很多Ajax交互都需要在页面上动态创建新元素，样式比较多的时候，通过style属性来修改会很繁琐。所以为了节省时间，应该在样式表里定义类选择符，然后当向DOM中增加新元素的时候，只需要设置className属性，就能得到相应的样式。

让我们以错误处理为例，快速体验一下这种做法。下面的CSS会用红色和大字体来显示文字：

```
.error { color:red; font-size:3em; }
```

如果在表单里发现了错误，在页面上显示一则错误消息，要好过难看而粗鲁的警告对话框：

```
document.getElementById("frm").onsubmit = function(){
var passcode = document.getElementById("passcode");
  if(!passcode.regexp.test(passcode.value))
  {
    var el = document.createElement("div");
    el.className = 'error';
    el.innerHTML = 'Not a valid passcode';
    document.getElementsByTagName("body")[0].appendChild(el);
    return false;
  }
}
```

我强烈建议在开发中避免直接修改元素的样式，除非样式属性的取值必须在运行时计算得出（例如动画）。

这样的做法有助于分离JavaScript的行为与呈现结果，正如CSS分离了HTML的内容与呈现结果。

2.8 向 DOM 中插入内容

上一个例子用到了我们还没介绍的几项DOM特性。其一是createElement()方法，它创建一个新的HTML元素，不过在你将它插入到文档之前，新元素只会孤零零地呆在一边。有3个DOM方法可以把新内容插入到文档中：

- appendChild()，将新元素作为父元素的最后一个子结点插入。
- insertBefore()，将新元素插入到指定的元素之前。
- replaceChild()，用新元素替换掉DOM中已有的元素。也可以用已经在页面上的一个元素替换掉另一个。

除了这3个方法，还有第四种（非标准的）方式可向文档添加内容：innerHTML属性。虽然不在任何规范之中，但所有浏览器都实现了这个属性，甚至在XHTML里也可以用。虽然Firefox 1.0在XHTML模式下不支持innerHTML属性，但Firefox 1.1+支持。

写入到innerHTML属性的HTML字符串会被解析并插入到文档之中。这是一次插入多个元素、属性和文本内容的高效方式。

我们来作一下比较，首先是使用DOM方法：

```
var el = document.createElement("div");
var txt = document.createTextNode("What are you looking at?");
var img = document.createElement("img");
img.src = 'imagename.gif';
img.alt = 'I\'m wearing glasses.';
img.height = 200;
```

```
img.width = 600;
el.appendChild(txt);
el.appendChild(img);
```

如果改成使用innerHTML属性：

```
var el = document.createElement("div");
el.innerHTML = 'What are you looking at? <img src="imagename.gif" alt="I\'m wearing➥
glasses." height="200" width="200">';
```

后者不单代码较为简短，而且在浏览器中的执行性能也较高。但是，请不要把innerHTML当作万灵药，而应该根据不同场合评估哪种方式更为合适。如果需要插入一大段HTML，那么innerHTML是最好的；在Ajax开发中常会遇到这样的情形。而使用DOM方法来插入元素可以实现细粒度的控制。结合使用两种技术才能得到最佳的结果。现在暂且把innerHTML放到一边，本书后面还会频繁地用到它。

给已有的 DOM 元素增加属性及方法

从DOM中得到的对象，其行为与一般的JavaScript对象没什么不同。因此可以在元素上添加额外的属性来保存数据，而无须引入新的函数或者对象。

举例来说，如果有一个表单项要求验证，我们可以把验证参数保存在元素里。

假设HTML代码如下：

```
<form id="frm">
  <input id="passcode" type="text">
  <input type="submit">
</form>
```

我们可以用下面的JavaScript代码实现验证：

```
var passcode = document.getElementById("passcode");
passcode.regexp = /^[0-9]+$/;

document.getElementById("frm").onsubmit = function(){
var passcode = document.getElementById("passcode");
  if(!passcode.regexp.test(passcode.value))
  {
    alert('Not a valid passcode');
    return false;
  }
}
```

在这个假想的例子里，我们从DOM中找到passcode元素，把一条正则表达式保存在里面。然后再表单上绑定了一个提交事件处理器，以便在提交之前检查表单字段的内容。如果用户输入的密码里有不允许的字符，就通知用户，并且返回false中止表单处理。

2.9 浏览器嗅探与对象检测

JavaScript编程中最令人头痛的问题，就是浏览器对各种特性的实现和支持程度各不相同。因此一般提倡尽可能在让浏览器执行一项任务之前，先确定它有这样的能力，以免它弹出烦人的出错消息。有两种办法可以确定浏览器的能力：浏览器嗅探和对象检测。

浏览器嗅探是过去的办法。浏览器有一个特殊对象叫做navigator，它有一些属性描述了浏览器的各个方面。浏览器嗅探大多是通过分析其中的userAgent属性来实现的。下面是一个例子：

```
" Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; ➥
.NET CLR 1.1.4322; .NET CLR 2.0.50727)"
```

这是Windows XP上IE 7的user agent字符串。可是问题在于浏览器会撒谎。由于网站开发者有时候会通过UA字符串来确定浏览器是否有能力使用该网站，并且把用了“不正确的”浏览器的用户阻挡在外，于是浏览器开发者发布新版浏览器的时候，会加入一段能通过检查的UA字符串来避开问题。IE假装成Netscape，Opera假装成IE。甚至有些浏览器允许用户任意修改UA字符串。简直是一团糟。

在此情况下我们只好采用对象检测的办法，在使用浏览器的某项特性之前先检查浏览器是否支持。对象检测比匹配UA字符串更可靠。虽然不应该使用每个方法之前都作对象检测，但至少在使用某些特性之前检测是必要的。

例如getElementByld可以有效地确定浏览器是否支持DOM：

```
if(document.getElementById)
{
   var el = document.getElementById('myelement');
}
```

不带括号去引用一个方法就能知道它是否存在。要想详细了解每个浏览器各自支持哪些方法，请参阅www.webdevout.net/browser-support-dom。

2.10 正则表达式

本章前面已经介绍过，正则表达式是一种实现字符串匹配的方法（常简称为regex或regexp）。正则表达式非常流行，尤其常用于验证表单数据，但它也是一头难以驯服的野兽。这一节将讨论一些基础内容，但无法涵盖正则表达式的所有方面。

注解 深入了解正则表达式请参阅Apress出版的*Regular Expression Recipes*，作者Nathan A. Good，ISBN: 1-59059-441-X。

实例化一个正则表达式有两种方式。第一种是通过正则表达式类：

```
var re = new RegExp('regex','ig');
```

构造函数有两个参数，第一个是要匹配的正则表达式字符串；第二个参数是设置匹配方式的标志，共有以下3种：

- i，忽略大小写。
- g，全局匹配。
- m，匹配多行。

声明正则表达式的第二种方式是通过字面量的形式：

```
var re = /regex/ig;
```

字面量形式并不用引号括起来，而是在前后加上正斜杠。标志字符紧跟在表达式的后面。

正则表达式要用在字符串上。正则表达式对象有两个主要方法：exec和test。

exec在字符串中搜索，并把匹配结果用一个数组返回。test在找到匹配的时候返回true，否则返回false。下面的例子检查是否符合美国或加拿大电话号码的格式：

```
var phonenumber = '613-555-1212';
/^\d{3}-\d{3}-\d{4}$/.test(phonenumber);// 返回true
```

String对象内建的match、search和replace方法都以正则表达式作为参数：

- match与exec相似，返回存放了匹配结果的一个数组。
- search返回匹配项在字符串中的位置索引，如果没找到则返回-1。
- replace将匹配项替换成指定的字符串。比如可以用它来调整日期格式：

```
var dt = '12-01-2007';
dt.replace(/^(\d{2})-(\d{2})-(\d{4})$/, '$3$2$1'); // 返回20070112
```

正则表达式从字符串头部开始，一路向后尝试用给定模式去匹配，直到字符串结尾。

表达式按照任务不同可以分成若干类。第一类是匹配自身的字符。例如A、Q或者数字8。更常用的是指定查找某一类字符或者某个范围的字符。字符范围用方括号定义，用连字号指示范围：

```
/[a-zA-Z]/
```

要匹配除给定范围以外的任意字符，可在范围前添加一个插入号（^）字符。

```
/[^a-zA-Z]/
```

可以用\w匹配任何字母、数字和下划线（\w的意思是“word character”），等价[a-zA-Z0-9_]。\W正好相反，匹配不在前述范围之中的任意字符。类似的还有\d匹配数字。

配除新行（newline）字符以外的任意字符，用句号（.）。

在搜索中还可以匹配字符串开头、结尾或者两者同时。若要规定匹配必须从字符串的第一个字符开始，使用插字号（^）。若要匹配最后一个字符，用美元符号（$）。

下面的例子匹配任意以http开头并且以.html结尾的字符串：

```
/^http.*\.html$/
```

还有一类表达式检查给定字符的重复次数。前面我们已经用过几次。例如{2}匹配的是连续两个字符。还可以定义区间如{2,4}，意思是匹配最少连续2个字符，最多连续4个字符。如果省略第二个参数则表示匹配从最低字符数开始，一路向上（如{2,}）。+符号表示字符出现1次或以上。*符号表示字符出现0次或以上。在下面的例子里，第一行表示在一条只含有数字的字符串里，至少有1个数字；第二行匹配空白字符串，或者含有1个以上数字的字符串。

```
/^\d+$/
/^\d*$/
```

用圆括号（）可记住找到的匹配项。在先前的日期格式变换例子中，我让表达式分别记住三个匹配项。随后在替换的时候，我用$符号后接一个数字来分别引用每个圆括号对应的匹配项，很容易就重排了字符串的组合次序。

正则表达式需要学习的内容还很多，远不止本节所能介绍的一点皮毛，因此我强烈建议你找时间进一步学习正则表达式。

2.11 格式化的惯例

虽然总能找到不同方式去做同一件事，但对于编写代码来说，确立一种固定的编码方式很重要，于己于人都能更清楚地理解代码的意图。本章在命名变量和函数的时候一直都遵循着相同的规则。

JavaScript程序一般都按照“驼峰型”大小写的惯例来书写。几乎所有JavaScript API，例如DOM和XMLHttpRequest，都采用此惯例。

注解 “驼峰型”大小写是一种将单词连写的惯例，它去掉单词之间的空格，并且将单词的首字母大写。例如“load calendar data”写成驼峰型就变成“LoadCalendarData”。它有两种变体，一种是大驼峰，将所有单词的首字母都大写（如LoadCalendarData）；另一种是小驼峰，第一个单词的首字母不大写（如loadCalendarData）。

变量名、函数名和对象名使用小驼峰，而类的名字一般使用大驼峰。类是一种通过new关键

字得到实例的程序结构（第3章将讲授类和对象）。

```
var element = document.createElement('div');
var object = new XMLHttpRequest();
```

2.12　事件处理

JavaScript在事件发生时触发执行，事件可以发生在页面加载的时候、用户单击的时候、文档改变的时候……代码如果没有封装在函数或者对象之中，那么就会在浏览器解析过后立即执行。封装在函数或对象中的代码必须经由事件处理器调用。

2.12.1　内联的事件处理

类似于通过style属性应用CSS，事件处理函数也可以直接应用到HTML元素之上。我们来看一个链接的单击事件：

```
<a href="mylink.html" onclick="foo()">My Link</a>
```

单击这个链接，foo()函数就会执行。对于具有基本行为的元素，如链接或表单，其行为会在事件处理函数执行完毕之后运行。在前面的例子中，foo()函数结束之后，用户将被带到mylink.html。为了阻止元素的默认行为，需要在onclick属性的末尾返回false：

```
<a href="mylink.html" onclick="foo();return false;">My Link</a>
```

或者也可以让函数决定返回true还是false，再传回给onclick事件处理器：

```
<a href="mylink.html" onclick="return foo();">My Link</a>
```

这样的用法在表单处理函数中最为常见。如果表单验证发现了任何错误，一律返回false，阻止向服务器提交表单。如果没有错误就返回true，表单被提交给服务器。

在链接中还可以用“javascript:”伪协议：

```
<a href="javascript:foo()">My Link</a>
```

我绝对不建议使用这种手法，因为它既不够周全，又助长了“不可访问的”编码习惯。“不可访问”是什么意思？我是指对于搜索机器人不可访问（机器人还不能理解JavaScript），对于禁用了JavaScript的用户也不可访问。最好的做法是保留默认的行为，然后用事件处理器覆盖它。

下面是一个弹出窗口的例子：

```
<a href="mylink.html" onclick="window.open(this.href);return false;">My Link</a>
```

如果用户开启了JavaScript，单击链接会在新窗口中打开目标页面。如果用户禁用了JavaScript，仍然可以浏览目标页面。

2.12.2 this 关键字

上一节最后的例子用到了this关键字，它的用途是引用当前的对象。在例子中，<a>元素就是当前对象。当你深入更复杂的事件处理和面向对象编程技术的时候，会看到this关键字扮演着一个显眼的角色。

2.12.3 无侵入的 JavaScript

前面提过要让HTML、CSS和JavaScript三大支柱保持分离。内联的事件处理器对分离的破坏，简直和style属性一样糟。不过你可以把所有行为都从HTML中剥离出来，集中到外部文件里，然后在有需要的文档中包含它们。

具体的做法是通过JavaScript将事件处理器绑定到对象上。例如，如果想在页面加载完成后执行某些代码，可以这样做：

```
window.onload = function()
{
   foo();
   bar();
}
```

如果想在图片上做一个鼠标移入的效果，代码类似这样：

```
image.onmouseover = function()
{
   this.src = 'newimage.gif';
}
```

当然鼠标移出之后还要再换回去：

```
image.onmouseout = function()
{
   this.src = 'oldimage.gif';
}
```

还记得可以把数据保存在元素的属性里，稍后再取出来使用吗？让我们把鼠标移出的脚本修改得更通用一些吧。鼠标移入的脚本也要一并修改：

```
image.onmouseover = function()
{
   this.oldsrc = this.src; // 把当前路径复制到一个自定义的属性c
   this.src = 'newimage.gif';
}
image.onmouseout = function()
{
   this.src = this.oldsrc; // 使用我们指定的旧路径
}
```

2.12.3　在页面加载前访问元素

前面所有的例子都假定需要的对象已经存在。只要经过浏览器读取和呈现，页面上每一个元素都可以在JavaScript中访问。但由于JavaScript代码通常包含在文档的头部，所以在页面整体还没加载完的时候，文档主体中的内容是不可访问的。访问一个还没诞生的对象将产生错误消息。因此在与页面上的任何元素交互之前，必须等待页面加载完毕。

通过window.onload事件可以得知页面已加载完成：

```
var el = document.getElementById("myelement"); // 将产生一则错误消息
window.onload = function()
{
   var el = document.getElementById("myelement"); // 嘿!我找到了我的元素!
}
```

只不过，这里有一个陷阱（总少不了陷阱!）。整个页面以及上面的全部图片都下载完毕之后，onload事件才会触发，这就是问题所在。onload事件处理器还远没能开始执行，用户就已经在和页面交互了。避开这个问题有几种选择，但不幸的是没有一种是完美的。最简单也最传统的做法是在HTML页面的最后放置一些JavaScript。位于代码之前的任意HTML元素都应该可以在脚本中访问。但这种办法算不上无侵入。

第二种办法是用一个计时器，在使用任意元素之前先检查是否存在；window.onload只作为最后的补救手段。Stuart Colville（http://muffinresearch.co.uk）按照这种思路写了一段Element Ready脚本，可以检查元素是否存在。如果不存在，脚本会在若干毫秒之后再检查一次。它会继续检查，直到找到元素，或者等到window.onload事件触发。下面是经过我稍作修改的Element Ready脚本：

```
var ElementReady={
  polled:[], /* 存储轮询的元素 */
  timer:null, /* 存储计时器 */
  timerStarted: false,
  ceasePoll:function()
  {
    clearTimeout(this.timer);
    this.timerStarted = false;
  },
  startPoll:function()
  {
    if(!this.timerStarted) this.timer = ➥
setTimeout(function(){ElementReady.check(false)},100);
  },
  check:function(clean)
  {
    for(var i=0;i<this.polled.length;i++)
    {
      if(document.getElementById(this.polled[i]['element']))
```

```
    {
      this.polled[i]['callback']();
      this.polled.splice(i--,1);
    }else if(clean){
      this.polled.splice(i--,1);
    }
  }
  if(this.polled.length == 0) this.ceasePoll();
},
cleanUp:function()
{
  this.check(true);
  this.ceasePoll();
},
chkDomId:function(elId,callback) {
    var el = document.getElementById(elId);
    if (el)
    {
      callback();
    }else{
      this.polled[this.polled.length] = ➥
{'element':elId, 'callback':callback};
      this.startPoll();
    }
  }
};
```

```
ElementReady.chkDomId('message',doStuff);
ElementReady.chkDomId('message2',doStuff2);

window.onload = function() {
  ElementReady.cleanUp();
};
```

Element Ready有很多方法和属性，大多只用于脚本内部。有两个函数是关键所在：chkDomId()和cleanUp()。chkDomId()函数有两个参数，第一个是希望得到的对象的ID，第二个参数是元素就绪之后要调用的函数。cleanUp()函数在window.onload事件触发时运行，最后确认所有元素都已加载，并且运行相应的回调函数。

Dean Edwards（http://dean.edwards.name）经过研究（与其他几个人一起）找出了一种在大多数浏览器上都可行的办法，去检测文档是否已经加载完成。他的方案不好的方面是每种浏览器上的实现途径都不一样，而且有些浏览器只有比较新的版本才能用。

```
// 针对Mozilla和Opera 9+浏览器
if (document.addEventListener) {
    document.addEventListener("DOMContentLoaded", init, false);
}
```

```
// 针对Internet Explorer (使用条件注释)
/*@cc_on @*/
/*@if (@_win32)
document.write("<script id=__ie_onload defer src=javascript:void(0)><\/script>");
var script = document.getElementById("__ie_onload");
script.onreadystatechange = function() {
    if (this.readyState == "complete") {
        init(); // 调用onload处理器
    }
};
/*@end @*/

// 针对Safari
if (/WebKit/i.test(navigator.userAgent)) { // 嗅探
    var _timer = setInterval(function() {
        if (/loaded|complete/.test(document.readyState)) {
            clearInterval(_timer);
            init(); // 调用onload处理器
        }
    }, 10);
}

// 其他浏览器
window.onload = init;

function init() {
    // 如果此函数已经被调用过，直接退出
    if (arguments.callee.done) return;
    // 给此函数加上标记，以免执行两次
    arguments.callee.done = true;

    //完成其他动作
};
```

在Dean的代码中，`init()`是唯一被调用的函数。前面Element Ready检查指定的元素是否存在和就绪，但Dean的方案检查的是整个文档是否就绪。Dean方案基本上靠浏览器告诉我们它是否已完成加载，而前一种方案持续主动检查加载状态。

2.12.5　用DOM方法绑定事件

到目前为止我们已经介绍过内联和使用对象属性两种方法绑定事件处理器。内联的事件处理器很难保持井井有条的划分，而通过对象属性绑定意味着每个属性上同一时刻只能绑定一个处理器。DOM中的`addEventListener`方法可以克服以上问题，让多个事件处理器和谐共处。

你可能也注意到每个事件只能添加一个事件处理器。对于短小的脚本来说这点限制不在话下，如果所有事件处理都在你的掌控之下，大型的脚本也不成问题。但当你开始使用别人的代码（到第4章学习JavaScript库的时候就会遇到），就有可能产生冲突。

DOM为你准备了解决方案——事件监听器：

```
element.addEventListener(event, listener, false);
```

event参数表示事件的类型（诸如*click*、*focus*、*blur*等，与前面不同的是不带*on*前缀）。第二个参数是事件触发时应当执行的函数（不要带括号，否则函数会立即执行）。最后一个参数是一个布尔值，表示事件处理器是否启用事件捕捉。

2.12.6 事件捕捉与事件冒泡

如图2-4所示，事件触发的时候，首先从最高层的文档开始，向下传播到实际发生单击的元素（这就是捕捉阶段），然后反过来向上传播事件（冒泡阶段）。在这种W3C标准方式里，事件处理器可以放在任意一个阶段。如果在捕捉阶段停止了事件，下方的元素就不会接收到事件。类似地，在冒泡阶段也可以停止事件，不让它继续向上冒泡。

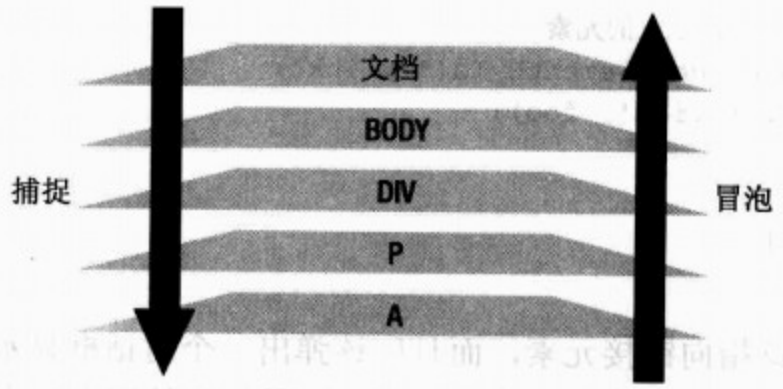


图2-4 事件捕捉与事件冒泡的流程

用stopPropagation方法停止事件在DOM树中继续向上或向下传播：

```
evt.stopPropagation();
```

更详细的信息请参阅W3C文档：www.w3.org/TR/DOM-Level-3-Events/events.html# Events-flow。IE不支持事件捕捉，所以事件捕捉功能很少被使用。

2.12.7 在 IE 中追加事件

事件处理中的最大问题，就是IE和其他浏览器的处理方式不一样。IE所用的方法叫做attachEvent()，它只有两个参数：事件名称（带*on*前缀）和要调用的函数。

```
element.attachEvent('onclick', functionname);
```

为了适应浏览器差异，只得分别用两段代码。我们把添加事件监听器的代码封装成一个可重用的函数：

```
function addListener(element, event, listener) {
  if (element.addEventListener){
    element.addEventListener(event, listener, false);
  } else if (element.attachEvent){
```

```
        element.attachEvent('on'+event, listener);
    }
}
```

现在可以给单个事件绑定多个事件处理器了：

```
addListener(window, 'load', foo);
addListener(window, 'load', bar);
```

事件监听器用起来很方便，不过处理事件的时候有一点要注意的。还记得this关键字吗？在IE中使用attachEvent()的时候，this关键字指向的并非事件绑定的目标对象，而是window对象。

下面的例子可以说明问题：

```
// 假设页面上有一个id为mylink的元素
var mylink = document.getElementById("mylink");
addListener(mylink, 'click', foo);
function foo()
{
   alert(this.href);
}
```

你可能以为this应该指向链接元素，而且应该弹出一个对话框显示href。但在IE里并非如此。让我们好好分析一下this关键字，看看JavaScript是如何处理上下文的。

2.12.8　检查上下文

当执行一个函数的时候，this关键字指向函数的拥有者。默认情况下，函数属于window对象：

```
function myfunction()
{
   alert(this); // this指向window对象
}
```

给对象添加函数就相当于把对象串接在一起。一个对象属于另一个对象，另一个对象又属于更高一层的对象：

```
var el = function ()
{
   alert(this); // this指向window对象
}
el.methodname = function()
{
   alert(this); // this指向我们的el对象
}
```

当把对象的方法当作参数传递给另一个函数的时候，很容易造成混淆：

```
var el = function ()
{
   alert(this); // this指向window对象
}
el.methodname = function()
{
   alert(this);
}

function myfunc(func)
{
   func();
}
myfunc(el.methodname);
```

执行路径的跳转让人有些眼花缭乱，我们还是一步步来看吧：

(1) 创建一个el对象，给它定义一个methodname()方法。如果现在执行el.methodname()，this将指向el对象。

(2) 创建一个myfunc()函数，它有一个参数。传进来的参数应该是一个函数，那样myfunc()才能执行它。

(3) 执行myfunc()函数，把el对象的方法作为参数传递进去。于是myfunc()函数执行传给它的方法。

不好理解之处就在这里。虽然执行的是el.methodname()，但传给myfunc()的仅仅是methodname函数的引用，而非整个el对象的引用。因此当methodname函数执行的时候。它显示出来的this是window对象，因为myfunc()属于window对象。

这样的设计既有好处也有坏处，取决于你的需要。好在JavaScript还提供了call()方法，允许方法在另一个对象的上下文中执行：

```
function myfunc(func)
{
   func.call(el);
}
```

例中func函数将被执行——而且是在el对象的上下文中执行。因此在func内部，this将指向el。

很多JavaScript库都有自己的一套方法去绑定对象（对象在第3章中讨论）。

再次回到我们的监听器代码，我们应该用call()改写addListener()函数，以便传递正确

的上下文。代码中用了一个匿名函数来封装引用：

```
function addListener(element, event, listener) {
  if (element.addEventListener){
    element.addEventListener(event, listener, false);
  } else if (element.attachEvent){
    element.attachEvent('on'+event, function(){listener.call(element)});
  }
}
```

2.12.9　取消行为

现在我们知道怎么调用事件了，有时候我们也需要取消事件。例如，在验证表单的时候，如果用户输入了无效数据，那么我们需要告诉浏览器阻止表单提交。

再来审视一下前面用过的密码验证例子：

```
var passcode = document.getElementById("passcode");
passcode.regexp = /^[0-9]+$/;

document.getElementById("frm").onsubmit = function(){
var passcode = document.getElementById("passcode");
  if(!passcode.regexp.test(passcode.value))
  {
    alert('Not a valid passcode');
    return false;
  }
}
```

在还没有介绍事件监听器的时候，我们直接把事件处理器绑定到元素上，因此可以通过返回false取消元素的默认行为。换句话说，如果在链接上用这种办法，就可以阻止浏览器跟随链接转到新地址；如果在表单上用，就可以阻止表单提交（正如上面的密码验证例子）。

但当我们用了事件监听器之后，就不能用同样的方式取消元素的默认行为了。好在DOM的事件对象给我们打开了另一扇门：preventDefault()。让我们用事件监听器重写这个例子：

```
var passcode = document.getElementById("passcode");
passcode.regexp = /^[0-9]+$/;

function isPasscodeValid(evt)
{
  var passcode = document.getElementById("passcode");
  if(!passcode.regexp.test(passcode.value))
  {
    alert('Not a valid passcode');
    evt.preventDefault();
  }
}
```

```
addListener(document.getElementById("frm"), 'submit', isPasscodeValid);
```

注意isPasscodeValid函数有一个evt参数，DOM会把事件对象传递给它。

事件对象里保存了很多与事件有关的属性和方法，比如事件的类型、按下的是哪个鼠标键（键盘事件则保存按下的键盘键），还有事件的目标元素。

IE需要特殊处理。因为IE 6之前的版本不会把事件对象作为参数传进去，而是在window层次设了一个事件对象。除此之外，IE的事件对象不认识preventDefault()；而要通过returnValue属性达到我们的目的。

动手调整函数让它能在IE下使用：

```
function isPasscodeValid(evt)
{
  evt = evt||window.event;
  var passcode = document.getElementById('passcode');
  if(!passcode.regexp.test(passcode.value))
  {
    alert('Not a valid passcode');
    if(evt.preventDefault)
    {
      evt.preventDefault();
    }else{
      evt.returnValue=false;
    }
  }
}
```

这是前面介绍的“对象检测”技术一个很好的应用实例。因为你不知道执行代码的是哪种浏览器，所以必须在使用前先检测对象或者方法是否存在。首先运用了OR（||）运算符，如果对evt的求值结果为false（evt为null、undefined、0或false都会得到此结果），就将window.event赋值给它。检查preventDefault()方法是否存在的时候用了另一种办法。不带括号地调用这个函数，如果函数存在我们就得到它的引用（使条件为真），如果不存在我们将得到undefined（使条件为假）。如果属性可能返回0、false或者null，就不能用这种办法检查属性是否存在。只好繁琐一点，用typeof propertyName == 'undefined'检查属性是否undefined。

2.12.10 综合练习

让我们把本章学到的知识全部融会成一个例子。在这个例子里，我们检查页面上所有的链接，只要是外部链接，就让它在新窗口中打开。检查是否外部链接的办法是比较当前域名和链接地址里的域名。如果两个域名不匹配，那么就认为是外部链接。

```
function addListener(element, event, listener) {
```

```
    if (element.addEventListener){
      element.addEventListener(event, listener, false);
    } else if (element.attachEvent){
      element.attachEvent('on'+event, function(){listener.call(element)});
    }
}
function changeLinksToNewWindow()
{
  // 获取当前页面的url，从中截取从"http://"到第一个"/"之间的字符串
  // 第一个（也是唯一的）匹配项就是我们要找的当前域名
  var currentDomain = window.location.href.match(/^http:\/\/[^\/]+/)[0];
  var elements = document.getElementsByTagName('a');
  for(var i=0;i<elements.length;i++)
  {
    // 若链接的href属性中包含了当前域名，返回值将>=0
    if(elements[i].href.lastIndexOf(currentDomain) >= 0)
    {
      addListener(elements[i], 'click', openWin);
    }
  }
}

function openWin(evt)
{
  evt = evt||window.event;
  window.open(this.href);
    if(evt.preventDefault)
    {
      evt.preventDefault();
    }else{
      evt.returnValue=false;
    }
}

addListener(window, 'load', changeLinksToNewWindow);
```

例子里共有3个函数外加一个事件处理器绑定。窗口加载完毕之后，changeLinksToNewWindow()函数找出页面上所有链接，逐一检查每个链接地址中是否包含当前域名。当前域名可从window.location对象中获得。如果不含当前域名，就给链接增加一个单击事件处理器。当有人单击的时候，链接会在新窗口中打开。

2.13　事件委托

有时候要绑定的元素实在太多，有时候需要频繁地在DOM中增加响应某些事件的新元素，一个个地添加事件处理器实在太繁琐。这时可以使用事件委托。

冒泡机制使DOM中远离事件发生地的上层元素也能接收到事件（见图2-5），这就是事件委

托的实现基础。我们可以在上层接收事件，然后通过事件对象的target属性（IE中是srcElement属性）判断事件到底来源于哪个元素。

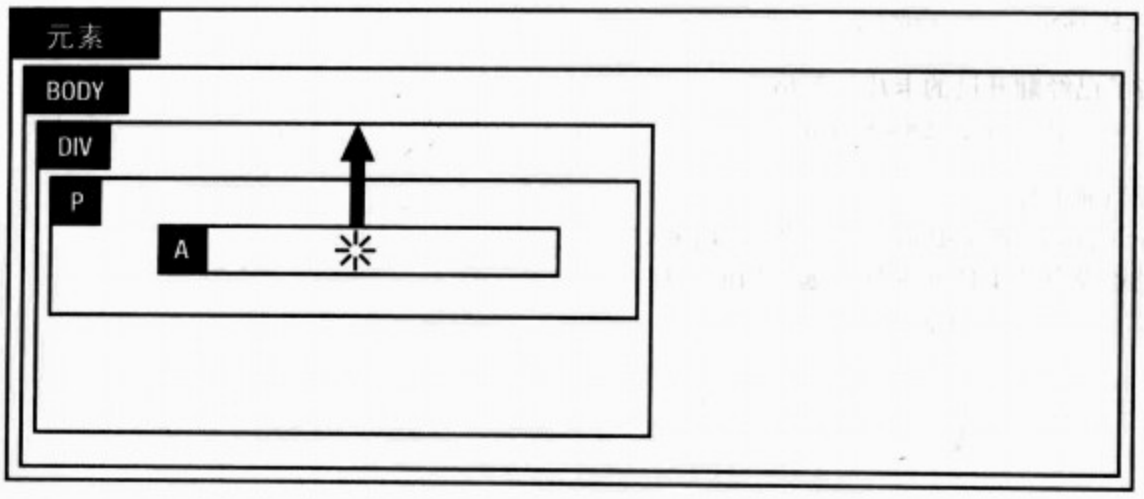


图2-5 单击事件从链接开始向上冒泡

```
//  找到事件的target，如果target不存在就找srcElement
var target = evt.target || evt.srcElement;
```

为了演示事件委托，我设计了一个非常简单的配对游戏：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" ➥
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<title>Simple Match Game</title>

<link rel="stylesheet" href="site.css" type="text/css">
<style type="text/css">
li {
  padding:80px 20px;
  width:200px;
  list-style:none;
  float:left;
  border:1px solid blue;
  text-align:center;
  text-indent:-9999px;
}
li.flipped {
  text-indent:0;
}
</style>
<script type="text/javascript">
var selectedPiece;
var totalMatches = 4;
var matchesFound = 0;
function checkPiece(evt)
{
```

```
var currentPiece;
evt = evt || window.event;
var target = evt.target || evt.srcElement;
currentPiece = target;

// 单击了已经翻开过的卡片，忽略
if(currentPiece.className == 'flipped') return;

// 翻开当前卡片
currentPiece.className = 'flipped';
// 如果还没有选中任何卡片，选中当前卡片
if(!selectedPiece)
{
  selectedPiece = currentPiece;
  return; // 完成一步
}

if(selectedPiece.innerHTML == currentPiece.innerHTML)
{
  matchesFound++;
  if(matchesFound == totalMatches)
  {
    alert('You found them all! Great game!');
  }else{
    alert('good match!');
  }
}else{
    alert('sorry, not a match!');
    // 重置样式和选中的卡片
    currentPiece.className = '';
    selectedPiece.className = '';
  }
  selectedPiece = null; // 重置选中的卡片
}

window.onload=function()
{
  var el = document.getElementById('pieces');
  el.onclick = checkPiece;
}
</script>
</head>
<body>

<ul id="pieces">
  <li>Shark</li>
  <li>Lion</li>
  <li>Lion</li>
  <li>Shark</li>
  <li>Dolphin</li>
```

```
    <li>Squirrel</li>
    <li>Dolphin</li>
    <li>Squirrel</li>
</ul>

</body>
</html>
```

在这个非常简单的例子里，无序列表中每一个列表项目代表一张卡片。仅在无序列表容器（ul）上添加了事件处理器，而不需要分别给每一个列表项目添加事件处理器。

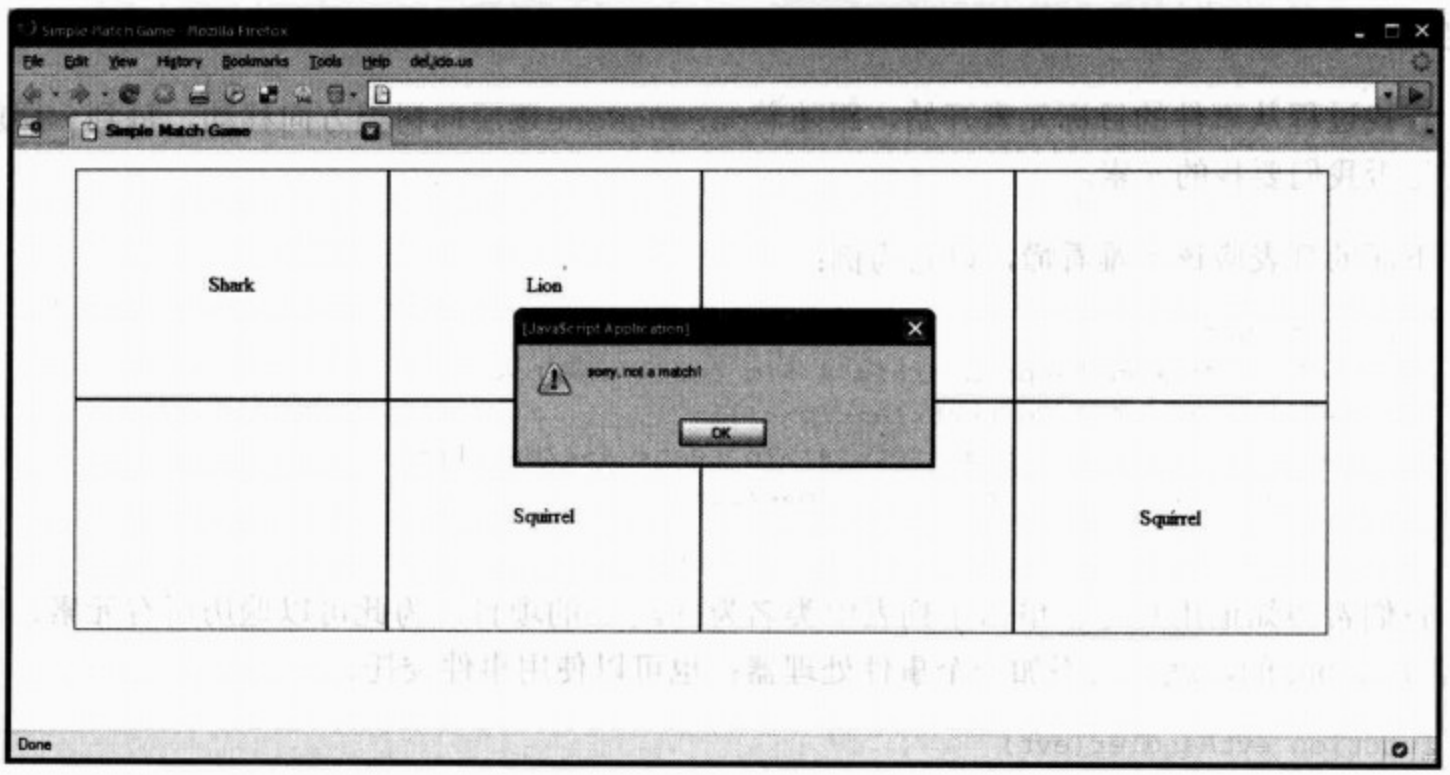


图2-6 配对游戏

```
window.onload=function()
{
  var el = document.getElementById('pieces');
  el.onclick = checkPiece;
}
```

只要无序列表内包含的任何元素发生了点击，事件就会被送到checkPiece函数。

```
function checkPiece(evt)
{
  evt = evt || window.event;
  var target = evt.target || evt.srcElement;
  currentPiece = target;
  ...
}
```

找出事件的源头，保存在currentPiece变量里。如果之前没有选中任何卡片，当前卡片就

被选中，成为等待配对的一半，暂存在selectedPiece。如果已经有选中的卡片，就比较它和当前卡片的innerHTML。如果相同就配对成功，通知用户并清除暂存的选中卡片。如果配对失败，两张卡片都被翻回去，用户重新开始下一轮配对。如图2-6所示游戏继续进行，直到用户找出的配对数量达到最大值（totalMatches）。

2.13.1　搜寻冒泡中途经过的元素

若感兴趣的既不是发出事件的元素，也不是处理事件的元素，情况就有些复杂。幸好还不是太难解决，只需要知道如何识别出想要的元素——可以通过ID，也可以通过类名，还可以是某种确定的HTML结构。

查找过程从事件的目标元素开始，跟随着parentNode逐层向树根方向移动，每到一层就检查它是否我们要找的元素。

下面的列表应该不难看懂，以它为例：

```
<ul id="test">
  <li class="theOne"><p><a href="#">To test</a></p></li>
  <li><p><a href="#">To test</a></p></li>
  <li class="theOne"><p><a href="#">To test</a></p></li>
  <li><p><a href="#">To test</a></p></li>
</ul>
```

我们希望知道用户是否单击了列表中类名为theOne的项目。为此可以遍历所有元素，凡是类名为theOne的，就给它添加一个事件处理器；也可以使用事件委托。

```
function evtHandler(evt)
{
  evt = evt || window.event;
  var currentElement = evt.target || evt.srcElement;
  var evtElement = this;
  while(currentElement && currentElement != evtElement)
  {
    if(currentElement.className == 'theOne')
    {
      alert('I have got one!');
      break; // 中断while语句
    }
    currentElement = currentElement.parentNode;
  }
}

var el = document.getElementById('test');
el.onclick = evtHandler;
```

采用事件委托，只需要给最外层的列表容器添加一个事件处理器，即可布下天罗地网，列表

范围内任何元素发生单击都会被捕捉到。currentElement变量一开始保存的是事件的目标元素，也就是搜寻的起点。我要检查它是不是一个有效的元素（因为在循环过程中有可能使它为null）。我还要检查当前元素是不是触发事件处理的元素evtElement，因为我只想在从事件源到触发事件处理的元素的范围内查找。理论上可以一直向上查找，直到文档的最高层，那时parentNode将等于null，所以为了以防万一，我们在循环条件中检查了当前元素是否存在。

在循环内部，通过匹配特定的条件来确定当前元素是否我们搜寻的目标。具体来说，也就是检查元素的类名是否等于theOne。如果是，执行必要的任务后中断循环（达到目的之后就没必要继续搜寻了）。如果不是我们要找的，就将父元素设为当前元素，然后回到while的开头，再循环一次。

有时候我们知道要找的元素总会固定出现在距离目标元素的某个位置，比如它总是直接父元素，或者总是某个兄弟元素。在这种情况下，可以省略掉循环，直接用DOM方法定位搜寻目标。

```
target.parentNode.nextSibling.innerHTML = 'I have been found!';
```

请翻到本章前面2.6节，查阅使用nextSibling时的注意事项。

2.13.2 事件委托不适用的情形

有时候事件委托并非最合适的解决之道。使用固定定位、相对偏移定位、绝对定位等方式，可使一个HTML元素叠在另一个不属于同一树结构的元素上方，这种情况下事件委托往往不适用。图2-7是一个相对定位使元素重叠的例子。

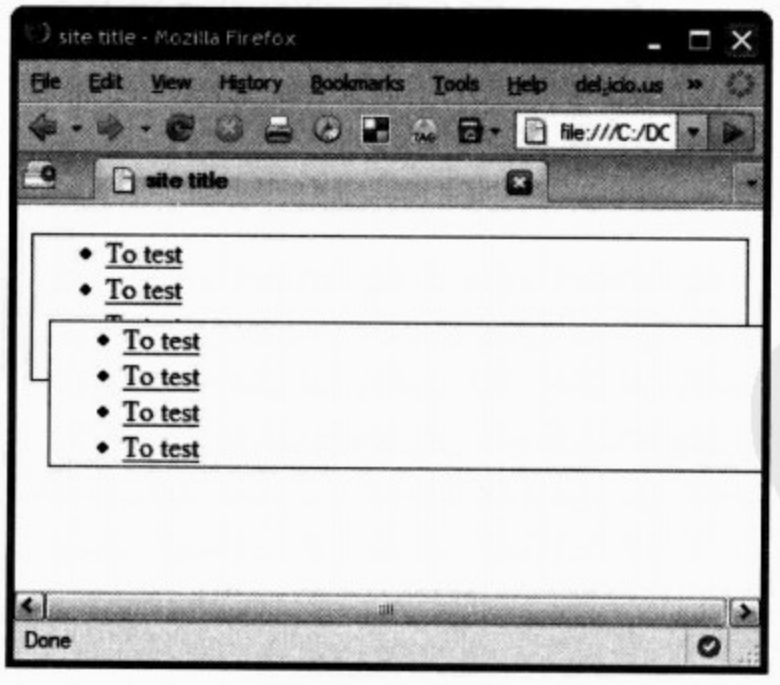


图2-7 两个元素重叠，叠在上方的元素发出事件

这样的情形看似不常见——确实很少遇到。一般不会有什么问题，然而，一遇到拖放操作它就会露出破绽。在屏幕上拖动元素的时候，我们通常希望下方经过的元素能有所提示，告诉我们

那里是否有效的释放区域。但由于被拖动的元素正挡在鼠标下方（使用绝对定位），使得下方经过的元素并不在事件传播的路径上，因此绝无可能简单地通过事件委托让下方的元素做出响应。

最终我们只能采取其他手段确定响应的时机，例如让被拖放的元素偏移到一旁，或者比较鼠标位置和释放区域的位置。

2.14　小结

本章涵盖了许多基础知识，主要包括以下内容：

- HTML以及语义的重要性。
- CSS，继承和特异性的重要概念，还有解决一些疑难问题的小建议。
- 部分JavaScript基础知识。
- DOM，如何访问元素和属性，如何向DOM加入新的内容，以及事件处理。

下一章将介绍面向对象编程的概念和使用JavaScript实施面向对象编程的技巧。下一章将以本章介绍的内容为基础，不但解释前面用过的一些技巧背后的原理，还会进一步深入到更高级的JavaScript编程。你将学习到一些由于JavaScript库的出现而变得十分流行的技术。

第3章 面向对象编程

第2章涵盖的各种知识将成为本章的基础。本章将解释什么是面向对象编程，如何以及为什么在JavaScript中进行面向对象编程。本章还会涉及一些高级的代码管理技巧，这些技巧能让你的代码更强大、更灵活。

3.1 什么是面向对象编程

在上一章中，你已经知道了JavaScript中的对象是什么样子的，还知道了怎样给对象添加方法和属性。OOP（面向对象编程）这个术语包含了很多概念。OOP是Java、C#等语言的核心，因此在那些语言中是不可能避开OOP的。而JavaScript不同，JavaScript传统上是通过简单的函数和变量来编程的。

如果你从来没用过OOP，可能会觉得不必自找麻烦。你已经有了变量和函数，如果需要重用什么东西，把它变成一个函数就可以了。如果两个函数需要访问同一个变量，你只需要保证该变量在两个函数之外声明。这些方法很适合短小的脚本，但脚本数量庞大之后就会变得相当混乱，难以阅读和维护。

事实上当你使用JavaScript时，其实就已经在使用对象了。比如window对象、一个HTML元素或者一个XMLHttpRequest对象，它们全都是对象。在JavaScript中，一切都是对象。学会使用面向对象的概念能帮助你较早编写出更加灵活的程序，不再像以前只能做一些简单的交互。而且其他人使用你的代码也会更容易（如果你打算通过互联网分享代码，或者身处一个大型的开发团队，那么这一点就很重要）。

OOP是一种编程范式，是一种编码时的思考方法，它包含以下特征：

- 类是对象的定义和蓝图。
- 对象是类实例化的结果，即类的实例。如果把类比作蓝图，那么对象就是按照蓝图建出来的房子。
- 属性就像变量一样定义了一个特定的状态或者储存了某个值，只不过属性是包含在类里

面的。属性一般是对对象的某种描述。

- 方法是依附于对象的函数，它可以接受一些参数，并且可以返回一些值。方法用来通过某种方式与对象交互，是对象行动和反应的途径。方法常被用来改变对象的状态（即属性）。虽然有些语言，比如Java，要求通过方法来修改属性（被称为getter和setter方法，因为它们获取和设置属性的值），但JavaScript允许你直接修改属性。这样就不需要写一些只改变属性值的过于简单的方法了。
- 封装隐藏实现的细节，并且根据“谁需要访问”限制对一些功能的访问。举个例子，假设有一个动画对象，它在一个元素列表里保存了准备在屏幕上移动的若干元素。如果你的代码要求该列表中的元素必须按特定的顺序摆放，那么你就有必要防止其他人直接访问该列表。在其他语言中，封装通常是通过将函数分为private、protected和public来完成的。
- 继承是将对象细分成子类。子类可以得到父对象的属性和方法，同时又能拥有自己的属性和方法。比如DOM（文档对象模型）就明显体现出了继承的行为。DOM中所有的元素都共有一套方法和属性，但其中的一些元素，如<select>，不但继承了公共的方法和属性，而且定义了自己的一些方法和属性。
- 多态使两个子类中的同名方法能够具有不同的行为。例如你可以有两个不同的Ajax子类，一个用来处理JSON调用，另一个用来处理XML调用。两个类都可以有一个模板方法，用于将Ajax调用的返回值转成HTML片段。两个类都从Ajax父类中继承了Ajax通信方面的函数，这部分是相同的，但模板方法必须不一样，因为它们要处理的数据格式不同。（如果不了解JSON和XML的差异，不必担心，第4章将为你介绍Ajax、JSON和XML）。

JavaScript是一种基于原型的语言，因此它的运作方式从设计上就与基于类的语言（如Java或C++）不一样。不过，上面提到的面向对象特性仍然可以在JavaScript的范型下重新创造出来。很多来自传统OOP语言的开发者在设计脚本的时候，都让脚本的行为方式接近自己熟悉的语言。了解上述OOP概念有助于加强你的编程技能。

现在让我们深入到有意思的部分吧。

3.2　函数

在JavaScript中，函数处在OOP的最核心部分，因为函数就是对象。正是有了函数的骨架，方法和属性才有所依附。

```
function CustomObject(){ }; //  或者写成
var CustomObject = function(){ };
```

有了上面的函数作为模板，就可以通过new关键字创建新对象：

```
var newObject = new CustomObject();
```

有类的语言就会有构造函数。构造函数是一个特殊的方法，它在新对象建立的时候执行，负责完成一些启动任务，比如定义默认的属性或其他动作之类的事情。

下面是Java中的类和构造函数的例子（关键部分已加粗）：

```
public class Hello
{
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
Hello notifyTheWorld = new Hello();
```

首先看class Hello，它声明了类的名字，并用一个花括号包住所有的方法。每当从这个类创建新对象的时候就会调用Hello函数。让我们看看换成用JavaScript来写会是什么样子：

```
function Hello()
{
  alert("Hello, World!");
}

var notifyTheWorld = new Hello();
```

粗看起来两者颇有相似之处，但其实差异很大。在Java中，实例化新对象的过程中会执行main函数。而在JavaScript中，运行该函数就是实例化新对象，函数本身就是类定义。每当新对象实例化的时候，就会执行函数中的代码。

这一差别决定了你在实例化对象和扩展对象的时候能做什么事情，不能做什么事情，因此了解这一差别是很有必要的。

3.2.1 添加方法和属性

有了对象之后，还需要给它提供方法和属性。这个主题上一章已经讨论过，但让我们再回顾一遍，同时为你演示几种不同的实现方式。

```
var CustomObject = function(){ };
CustomObject.value = 5;
CustomObject.methodName = function(){ alert(this.value) };
CustomObject.methodName(); // 是5!
```

上面的代码成功地为对象添加了方法和属性，但它存在很大的不足。因为新增的方法和属性仅在这个对象上才有效，当你通过该函数再次新建一个对象时，刚才添加的方法和属性都不会出现在新对象上（如下面的代码所示）：

```
var newObject = new CustomObject();
newObject.methodName(); // 未定义……不会起作用.
```

要想在类中添加属性，应该在函数内部使用this关键字（稍后会解释其中的原因）：

```
var CustomObject = function(){
   this.value = 5;
   this.methodName = function(){ alert(this.value) };
};
```

这样就给类添加了新的属性，从这个类实例化出来的新对象将同样拥有新的属性：

```
var newObject = new CustomObject();
newObject.methodName();// 结果是5!
```

通过这种方式定义的属性在所有新的对象实例中都是可用的。

3.2.2　对象的实例化机制

每当通过new关键字执行一个函数，一个新对象就会被创建出来，并被用作函数内部的上下文。在函数内部，this关键字指向新创建的对象，而且你可以给它加上新的属性和方法。函数执行完毕后，新对象被赋值给你指定的变量。

现在你可以在将信息赋给新对象之前对其进行一些处理：

```
var Adder = function(valueA, valueB){
   var newvalue = valueA + valueB;
   this.value = newvalue;
   this.result = function(){ alert(this.value) };
};
var added = new Adder(5, 6);
added.result(); // 结果是11!
```

3.2.3　在构造函数中返回对象

你也可以在实例化过程中无视this关键字，而明确地返回自己指定的对象（并为其添加属性和方法）：

```
var Adder = function(valueA, valueB){
   var newvalue = valueA + valueB;
   var object = new Object();
   object.value = newvalue;
   object.result = function(){ alert(this.value) };
   return object;
};
var added = new Adder(5, 6);
added.result();// 结果是11!
```

以这样的方式返回对象会给你很大的灵活性，尤其是在处理继承的时候。下面的例子用了一个字面量对象（稍后会讨论到）：

```
var coreMethods = {
  add:function(a, b){
    return a + b;
  },
  minus:function(a, b){
    return a - b;
  },
  multiply:function(a, b){
    return a * b;
  },
  divide:function(a, b)
  {
    return a / b;
  }
};

var SimpleMath = function()
{
  var methods = coreMethods;
  methods.power = function(a, b)
  {
    return Math.pow(a,b);
  }
  return methods;
}

var sm = new SimpleMath();
alert(sm.power(5,6));
```

上面的代码先是在SimpleMath对象的外部定义了若干核心方法，然后将它们应用到SimpleMath对象身上。这些核心方法可以单独使用，也可以依附在其他对象身上，

在下面的例子中，Pizza对象不但同样继承了那些核心方法，而且还进一步扩展，加入了自己的方法。Pizza对象实例化过程中要用到切片数量的参数。之后你就可以通过split方法计算每人能分到多少块饼，而split方法会自动调用核心方法中的divide方法来完成计算。

```
var Pizza = function(slices)
{
  var methods = coreMethods;
  methods.split = function(friends)
  {
    return methods.divide(slices,friends);
  }
  return methods;
}

var za = new Pizza(16);
alert(za.split(4)); // 对4进行警告
```

3.2.4　原型

JavaScript被称为基于原型（prototype）的语言（与基于类的语言相对），其原因在于继承是通过原型链来实现的。在前面的例子中，每个新对象实例化时都会把原对象的属性和方法复制一份。如果存在1000个对象，就会有1000份属性和方法——每一份都单独占据一块内存。

为了避免这种额外开销，JavaScript引入了一个特殊的原型属性，附在原型属性上的方法将被所有的对象共享：

```
var Adder = function(valueA, valueB){
  var newvalue = valueA + valueB;
  this.value = newvalue;
};
Adder.prototype.result = function(){ alert(this.value) };
var added = new Adder(5, 6);
added.result(); // 结果是11
```

只有result()方法是加在原型上的。因为value属性依赖于构造函数中传入的参数，所以不能加在原型上，而要在运行时通过this关键字加入。

原型属性还有一项便利之处，就是即使在对象实例化之后，仍然可以通过原型属性给基对象添加属性。已经实例化的对象只要属于同一个原型，都会获得新加入的属性。我们调整一下前面的例子来说明：

```
var Adder = function(valueA, valueB){
  var newvalue = valueA + valueB;
  this.value = newvalue;
};
Adder.prototype.result = function(){ alert(this.value) };

var added = new Adder(5, 6);
added.result(); // 结果是11

Adder.prototype.multiply = function(valueC){ alert(this.value * valueC) };
added.multiply(5); // 结果是55!
```

继承关系的处理如下：

```
var Dog = function(){ };
Dog.prototype.bark = function(){ alert('woof') };

var Chihuahua = function(){ };
Chihuahua.prototype = new Dog();

var sparky = new Chihuahua();
sparky.bark(); // 噢!
```

JavaScript中的继承分为深继承和浅继承。这里给出的例子属于浅继承。深继承的意思是一

个类继承自另一个类，而那个类又继承自另一个类，以此类推。浅继承的意思是某个类继承自另一个类，然后继承关系到此为止。JavaScript从来都不是为深继承而设计的，在JavaScript中也很少需要深继承。

要想进一步了解深继承，可以查阅以下资料：

- Douglas Crockford 所著 *Classical Interitance in JavaScript*（http://javascript.crockford.com/inheritance.html）
- Dean Edwards所著*Base*（http://dean.edwards.name/ weblog/2006/03/base/）

3.3 字面量对象

使用字面量对象是创建新对象的另一种主要手段，使用起来极为简单：

```
var customObject = {};
```

这样就创建好了一个对象。给对象添加属性和方法也同样简单：

```
var customObject = {
  customProperty: 5,
  customMethod: function(){ /* 这里用了一个匿名函数*/ }
};
```

每个属性都写成“键-冒号-值”的格式，属性声明之间用逗号隔开。键的名字在内部会被转换成字符串。虽然在大多数情况下都不需要关心这一点，但有时候可以巧妙地利用它完成一些事情。下面的代码是完全合法的：

```
var customObject = {
  "My custom property": 5,
  5:6,
  "5":7
};
```

请记住由于键会被转换成字符串，所以第二个“5”实际上会覆盖掉第一个5，于是customObject["5"]的最终取值为7。

如果你在键的名字中使用了空格等特殊字符，或者属性名以数字开头，那么访问这些属性就只能通过方括号语法或者用for..in循环遍历所有的属性。

任何时候都可以通过圆点或者方括号给对象添加新的属性：

```
customObject.value = 6;
customObject["otherValue"] = 7;
customObject.newMethod = function(){};
```

字面量对象的局限是不能把它当作一个类来实例化新的对象。定义一个字面量对象仅仅是定义了一个对象。只能拥有一个对象也有它的好处，因为有时候恰恰就希望只能从一个中心位置进行控制（也就是常说的单例设计模式）。

注解　设计模式是解决某种问题的惯常做法。通过深入理解解决某个问题的各种手段，你可以从中选择最适合的方案。如果想更深入了解设计模式，请阅读维基百科上的解释：http://en.wikipedia.org/wiki/Design_pattern(computer_science)。

上一章中定义的ElementReady对象是一个很好的例子，请看下面的代码片段：

```
var ElementReady={
  polled:[], /* 存储轮询的元素 */
  timer:null, /* 存储计时器 */
  timerStarted: false,
  ceasePoll:function()
  {
    clearTimeout(this.timer);
    this.timerStarted = false;
  },
  startPoll:function()
  {
    if(!this.timerStarted) this.timer = ~CCC
setTimeout(function(){ElementReady.check(false)},100);
  }
}
```

字面量对象很适合作为中心访问点，在这个例子中只需要使用一个计时器就能控制所有函数的执行。

请记住在JavaScript里没有什么是不可能的。如果你想从字面量对象创建出新的对象，可以采用下面的方式：

```
function object(o)
{
   function F(){}
   F.prototype = o;
   return new F();
}

var newObject = object(ElementReady);
```

object()函数接受一个对象作为参数。它先创建一个函数（对象），然后把传入的对象设为该函数的原型，实质上相当于把传入对象的所有属性和方法都复制给了新的对象。然后就可以实例化并返回新对象了。这个小函数要归功于Douglas Crockford。

3.4 for..in 循环

虽然前面已经提到过for..in循环，但它在使用中还有一些方面需要留意，所以在此进一步详细讨论。如果你对for..in循环不熟悉，其实它和普通的for循环很相似，只不过用它可以像遍历关联数组一样遍历对象的属性：

```
// 即将要循环遍历的对象
var coreMethods = {
  add:function(b){
    return a + b;
  },
  minus:function(b){
    return a - b;
  },
  multiply:function(b){
    return a * b;
  },
  divide:function(b){
    return a / b;
  }
};

for (var property in coreMethods) {
    alert(coreMethods[property]); // 对每一个函数进行警告
}
```

for..in循环会遍历对象及其原型的方法和属性，并将键名（方法及属性的名称）逐一赋给在in之前的变量（上例中的property变量）。如果用自定义方法扩展了Object，会出现一种特别棘手的情况。用自定义方法来扩展Object是很常见的，毕竟它用起来很便利。我们举一个例子，假设你为了实现搜索结果的分页在一个对象中存储了若干属性：

```
var queryComponents = {
  sortBy: 'name',
  page: 1,
  pages: 10,
  resultsPerPage: 20
}

function queryBuilder(obj)
{
  var querystring = '?';
  for(var property in obj)
  {
    // 在将&添加到各个值之前，确保对某些已进行了追加
    if(querystring.length > 1) querystring += '&';
    querystring += property + '=' + obj[property];
  }
```

```
    return querystring;
}
```

用queryComponents对象来定义查询字符串非常方便，如果想要每页30条结果而不是20条，只要改一个值就可以了。queryBuilder函数遍历对象的全部属性，构造出传回服务器的查询字符串。执行queryBuilder(queryComponents)得到如下结果：

```
?sortBy=name&page=1&pages=10&resultsPerPage=20
```

然而，如果有人扩展了Object.prototype会发生什么事呢？

```
Object.prototype.extend = function(obj) {
  for (var property in obj) {
    this[property] = obj[property];
  }
  return this;
}
```

上面的函数也是很好用的，因为它通过把一个对象的方法和属性复制给另一个对象，实现了一定程度的继承关系。但现在再运行queryBuilder函数将得到如下结果：

```
?sortBy=name&page=1&pages=10&resultsPerPage=20&extend=function (obj) {
  for (var property in obj) {
    this[property] = obj[property];
  }
  return this;
}
```

要想只检查属于目标对象自身的属性，应该使用hasOwnProperty方法。于是queryBuilder函数改写如下（注意加粗的部分是新增的）：

```
function queryBuilder(obj)
{
  var querystring = '?';
  for(var property in obj)
  {
    if(obj.hasOwnProperty(property))
    {
      // 在将&添加到各个值之前，确保对某些已进行了追加
      if(querystring.length > 1) querystring += '&';
      querystring += property + '=' + obj[property];
    }
  }

  return querystring;
}
```

在遍历全部属性的时候，检查一下属性是否直接属于目标对象，确保属性不是来自prototype。在使用for..in循环的时候，检查hasOwnProperty总是一个好习惯。

3.5 命名的参数

字面量对象可用来实现命名的参数以及可选的参数，还可以很方便地为对象定义默认选项。在定义函数的时候，通常会指定若干参数作为该函数的选项。如果忘了提供某个参数，它会被当作未定义的值传递进去：

```
function func(a, b, c)
{
   alert(a); // 未定义
}
func();
```

用上面的写法，即使你只关心第一和第三个参数，也要为第二个参数传些什么进去。为了用字面量对象来充当选项，可以让函数只接受一个参数：

```
function func( options )
{
  alert(options.a); // 对5进行警告
}
var myOptions = { a: 5, b: 6, c: 7 };
func( myOptions );
```

要想实现默认选项，可以在函数内部先定义默认的取值，然后用传递进来的参数覆盖它们：

```
function func( updates )
{
  var options = { a: 5, b: 6, c: 7 };

  for (var property in updates) {
    options[property] = updates[property];
  }

  alert(options.a); // 对8进行警告
}
func( {a:8} );
```

上面的for循环把updates对象的所有属性都复制给了内部的options对象。

3.6 命名空间

字面量对象在实践中可以用来模拟命名空间。命名空间是容纳一组相关项的容器。命名空间在其他语言如Java中是一种普通的，甚至必不可少的概念。虽然JavaScript并没有为命名空间提供

专门的语言构造，但可以用字面量对象和命名约定来达到同样的目的。

使用命名空间有以下优点：

- 使用命名空间有利于保持global对象（也称为window对象）的清洁。随着脚本越来越庞杂，再加上使用第三方脚本，如果把所有东西都放到全局层次，很容易出现命名冲突。比如Prototype JavaScript库和jQuery库都用$()来获取HTML元素，但两者的行为是不一样的，而且返回对象拥有的方法也不同。最终结果是最后包含的库会覆盖前面包含的库的定义，于是依赖于前一个库的代码就会出错。jQuery允许你重新映射$()来避免这样的问题。别以为这种情形不常发生，比如写作这本书的时候，Digg.com网站就正好处于从Prototype切换到jQuery的过程之中，他们切换的时候就要非常小心避免代码的冲突。
- 使用命名空间可以改善代码的可读性。通过把东西封装到一个单独的对象之中，相当于建立了一种代码的从属关系，让空间内的代码更加自立和完备。例如我把我的网站的代码都放进SNOOK命名空间，就清楚地表明它们是相关联的。Yahoo! User Interface库（第4章会介绍）也采取了同样的做法，它用了YAHOO命名空间。这样就避免了任何混淆的情况。而且可以把共享的变量保存在命名空间里面，一方面所有函数都可以访问这些共享变量，另一方面又不必担心变量与其他脚本冲突。

是不是必须使用命名空间呢？当然不是。过去十年的JavaScript开发没有命名空间也一样好好的。但从一开始保持整洁毕竟没有坏处。

以下取自作者本人的博客网站的代码可以说明命名空间的用法。我的博客上有个评论表单，让读者针对某个帖子留言。如果读者填写了姓名、E-mail地址和URL，下次访问的时候就会将其记住。代码中先声明了一个命名空间，然后每个方法都被附在命名空间对象上。

```
// 声明命名空间
var SNOOK = {};

SNOOK.prepareCommentForm = function(){ /* 初始化字段 */ }
SNOOK.prepareField = function(options) { /* 附加事件处理程序，等等 */ }
SNOOK.setCookie = function(name, value, expires){ /* 设置cookie */ }
SNOOK.getCookie = function(name){ /* 获取cookie的值 */ }
SNOOK.remember = function(fld){ /* 记住用户输入的数据*/ }
```

3.7　闭包

JavaScript中的闭包使一向被严重误解的特性。通过**闭包**，子函数得以访问父函数的上下文环境，即使父函数已经结束执行。在图3-1中，函数A和函数B可以访问window对象中声明的所有变量和函数。类似的，函数C和函数D是在函数B中声明的，它们可以访问window对象中声明的所有变量和函数，以及函数B中声明的所有变量和函数。函数C中声明的任何函数都可以沿着这

棵树一路上溯，访问上游所有的变量和属性。

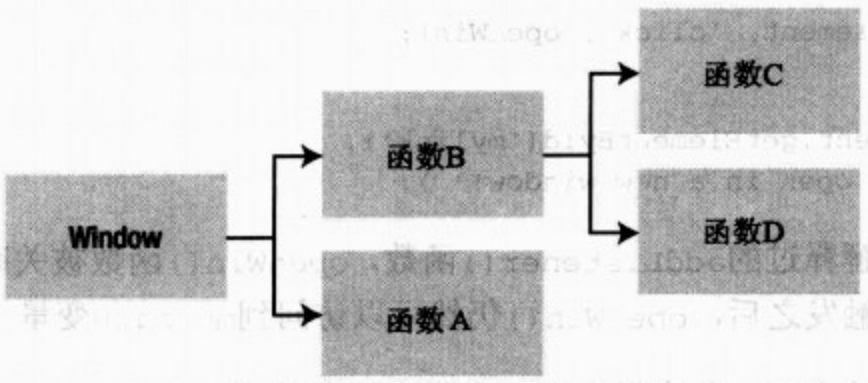


图3-1 由闭包产生的变量及函数的访问层次关系

闭包最常见的用途是声明事件处理器：

```
function attachBehavior(){
   var element = document.getElementById('main');
   element.onclick = function(){ element.innerHTML = 'Surprise!'; };
}
```

绑定到onclick事件上的函数建立了一个闭包，因此它能够访问到element变量。

不管如何被传来传去，函数总会保持对最初的作用域的访问能力。函数可以被当作返回值返回，也可以被当作另一个函数的参数，不然怎样，它始终能访问最初的作用域。但有一点必须记住，你想要保持访问能力的变量来自哪个函数，就必须在哪个函数内声明闭包。

```
function onclickHandler(){
   element.innerHTML = 'Surprise!';
}
function attachBehavior(){
   var element = document.getElementById('main');
   element.onclick = onclickHandler;
}
```

如果运行attachBehavior函数并单击main元素，将会出现元素未定义的错误，因为onclickHandler函数是在attachBehavior函数外部定义的。即使onclickHandler函数已经被赋值给attachBehavior函数内部的变量，也不会起作用。onclickHandler函数在哪里声明才是决定性的。

在分离事件处理代码的情况下，用闭包来引用信息很有效，这样就不必为保持对this的正确访问而担心：

```
function launcher(element, message)
{
  function openWin()
  {
```

```
    alert(message);
  }
  addListener(element, 'click', openWin);
}

launcher( document.getElementById('mylink'),
'This link will open in a new window!' );
```

通过使用上一章解释过的addListener()函数，openWin()函数被关联成事件处理器。由于闭包的存在，在事件触发之后，openWin()仍然可以访问到message变量。

再熟习闭包的用法之后，你会发现自己常常会避免使用this。

闭包过去有点名声不佳，那是由于当闭包与DOM对象一同使用时，IE对它们的处理有问题。IE过去在此情况下会遇到不能释放内存的麻烦，即使已经离开当前页面也没法释放相关内存，要重新启动浏览器才能收回泄露的内存。微软已经在IE 7中修复了这个错误，而且在2007年6月为IE 6提供了一个补丁来修复这个内存泄露问题。但不幸该补丁只适用于Windows XP上运行的IE 6，因此旧版的Windows或者没有安装该补丁的机器仍然要受到这个内存泄露问题的折磨。

对于小的应用，内存泄露可能不明显，而且相关代码要执行很多次才会对系统性能产生显著的影响。但对于比较大型的应用，用户会在同一个页面上停留很长时间，反复执行DOM和事件处理操作。因此出现问题的几率会比较高，从而影响到用户的体验——尤其现在很多人都不会经常关闭浏览器。（我有时候会好几天，甚至几个星期都不关闭浏览器。）

为了避免内存泄露，可以避免使用闭包：

```
function attachBehavior(){
   var element = document.getElementById('main');
   element.onclick = onclickHandler;
}

function onclickHandler(){
   // 该关键字涉及被单击的元素，而不是变量'element'
   this.innerHTML = 'Surprise!';
}
```

很多JavaScript库，尤其是本书涉及的库，都采取了有针对性的实现方式来尽量降低内存泄露的潜在可能性。

3.8　封装

封装帮助你向脚本的用户隐藏实现的细节。还记得本章开头的Java代码吗？

```
public class Hello
{
```

```
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

从加粗的部分可以看到Java是如何让开发者向代码的用户隐藏实现细节的。公有方法是与对象交互的访问点。如果把方法设成private，那么就只有类本身才能访问该函数。类似地，如果把类设成private，那么就只有同一个命名空间中的其他类才能访问该类。

如果开发代码是要提供给他人使用，比如在一个开发团队中，或者作为助手脚本对外发布，通常应该规定一个应用程序编程接口（API）。有些属性和方法是其他人可以使用的，而另外一些只是为了自己编程方便而写的。

看下面的ElementReady脚本：

```
var ElementReady={
  polled:[], /* 存储轮询元素 */
  timer:null, /* 存储计时器 */
  timerStarted: false,
  ceasePoll:function(){...},
  startPoll:function(){...},
  check:function(clean){...},
  cleanUp:function(){...},
  chkDomId:function(elId,callback) {...}
};
```

在这个对象中，大部分属性和方法都只供内部使用，只有chkDomId()、cleanUp()和check()方法提供给对象外部使用的。在此情况下，是不是应该把内部用的属性保护起来呢？运用前面学到的对象创建和闭包的知识，实际上可以重新设计这个类，让外部只能访问到那3个方法，而其他东西都只能通过那3个方法间接访问：

```
var ElementReady= new function(){
  var polled = []; /* 存储轮询元素 */
  var timer = null; /* 存储计时器 */
  var timerStarted = false;
  var ceasePoll = function(){...};
  var startPoll = function(){...};
  return {
    check:function(clean){...},
    cleanUp:function(){...},
    chkDomId:function(elId,callback) {...}
  }
};
```

我来解释一下新的实现。首先主对象ElementReady从字面量对象改成了由一个匿名函数实例化而成的对象。原先字面量对象的内部属性和方法变成了普通的变量，也就是ElementReady

对象的私有变量。最后，把要对外公开的3个函数放在一个字面量对象中返回。

在闭包的帮助下，公开函数仍然可以访问内部的变量。对内部变量的引用方式也发生了改变——去掉了内部变量前面的this关键字。

将来你可以编写新版本的ElementReady，只要那3个函数的接口不变，其余部分的实现你怎么改都没有关系。

最后的实现如下：

```
var ElementReady= new function(){
  var polled = []; /* 存储轮询元素 */
  var timer = null; /* 存储计时器 */
  var timerStarted = false;
  var ceasePoll = function()
  {
    clearTimeout(timer);
    timerStarted = false;
  };
  var startPoll = function()
  {
    if(!timerStarted) {
      timer = setTimeout( function(){ElementReady.check(false)}, 100);
    }
  };

  return {
    check:function(clean)
    {
      for(var i=0;i<polled.length;i++)
      {
        if(document.getElementById(polled[i]['element']))
        {
          polled[i]['callback']();
          polled.splice(i--,1);
        }else if(clean){
          polled.splice(i--,1);
        }
      }
      if(polled.length == 0) ceasePoll();
    },
    cleanUp:function()
    {
      check(true);
      ceasePoll();
    },
    chkDomId:function(elId,callback) {
      var el = document.getElementById(elId);
```

```
        if (el)
        {
          callback();
        }else{
          polled[polled.length] = {'element':elId, 'callback':callback};
          startPoll();
        }
      }
    }
};
```

同样的封装手法也可以用来创建类。类可以实例化成许许多多的对象，而每个对象都能访问内部隐藏的属性和方法（叫做私有成员）：

```
function CurrentAnswer(num)
{
  var current = num;

  var newObject = {
    getCurrent: function(){ return 'The current answer is: ' + current; }
  }
  return newObject;
}

var curr = new CurrentAnswer('5');
alert(curr.getCurrent()); // 对字符串'The current answer is: 5'进行警告
```

通过这样的方式，既为对象的交互定义出稳固的API，同时又隐藏了内部的工作原理。

3.9 函数式编程

函数式编程就像OOP一样，也是一种编程范型。它的概念是把函数当作输入参数，而且可以返回另一个函数作为结果（称为高阶函数）。这是一种非常强大的手段，JavaScript语言在这方面的能力令许多JavaScript库受益匪浅。

3.9.1 回调

回调是指把一个函数（或其名称）传递进另一个函数，以便第二个函数执行的时候，可以“把第一个函数叫回来”。回调很常见，尤其是在事件处理的场景中。实际上大多数自定义事件系统都是采用回调的方式来实现在事件发生时调用某个函数。

回调的用法一般是把一个函数作为参数传递给另一个对象：

```
function Animation(startFunction, endFunction)
{
  startFunction();
```

```
    /* 此处上演我们的动画魔术 */
    endFunction();
}

Animation( function(){ alert('Start!') }, function(){ alert('End!') } );
```

这里传进了两个匿名函数：第一个函数会在动画开始前被调用；第二个函数会在动画结束后被调用。

把对象的一个方法当作回调函数来传递可能会出现问题。在这种情况下，方法和对象切断了连接。因此当你在其他函数中执行该方法的时候，很可能出错：

```
function Pizza(includePepperoni)
{
    this.pepperoni = includePepperoni;
    this.hasPepperoni = function(){ return this.pepperoni; }
}
var newPizza = new Pizza(true);

function eatPizza( hasIngredient )
{
    alert('Has Ingredient? ' + hasIngredient() );
}

eatPizza( newPizza.hasPepperoni );
```

执行这段代码将出现对象未定义的错误。当hasPepperoni()方法被执行的时候，它已经脱离了newPizza对象，被放到了eatPizza()函数的父上下文——也就是window对象中执行，该对象并没有hasPepperoni属性。

在另一个对象的上下文中执行回调函数应该使用JavaScript的call()函数。在这个例子中，call在pizza对象的上下文中执行haslngredient函数，也就是在传递进来的newPizza对象上执行传递进来的hasPepperoni()函数。实际上等价于newPizza.hasPepperoni()。这样一来hasPepperoni函数中的this关键字就能够正确地告诉你比萨饼上有没有意大利香肠了。

```
function eatPizza( hasIngredient, pizza )
{
  alert('Has Ingredient? ' + hasIngredient.call(pizza) );
}

eatPizza(newPizza.hasPepperoni, newPizza );
```

这里还可以用另一种方式实现同样的目的。你可以沿用前一种eatPizza()函数的写法，但改用闭包来传递信息：

```
function eatPizza( hasIngredient )
{
  alert('Has Ingredient? ' + hasIngredient() );
```

```
}

eatPizza( function(){ return newPizza.hasPepperoni() } );
```

传递进eatPizza()的匿名函数会告诉你同样的答案。在JavaScript里，总是会有不同的方法能达到同样的目的。

3.9.2 函数的 call 和 apply

从前面的例子看到，用call可以把函数放到指定对象的上下文中执行。不但如此，它还可以给函数传递任意数量的参数：

```
hasIngredient.call(pizza, 'hot');
```

上面的代码等价于：

```
newPizza.hasPepperoni('hot');
```

apply函数的作用与此类似，只不过它不像call要分别指定每个参数，而是用一个数组来容纳所有的参数。参数的次序和它在数组中出现的次序相同。

```
hasIngredient.apply(pizza, ['hot','medium','mild']);
```

上面的代码等价于：

```
newPizza.hasPepperoni('hot','medium','mild');
```

Prototype JavaScript库可以说恰如其分地使用了apply函数。它扩展了Function原型，因此每个函数都可以通过非常简洁的语法绑定到一个对象上：

```
Function.prototype.bind = function() {
  var_method = this, args = $A(arguments), object = args.shift();
  return function() {
    return_method.apply(object, args.concat($A(arguments)));
  }
}
```

$A是Prototype JavaScript库中的一个函数，它会遍历传递给它的参数集合，把集合的元素都添加到一个数组。然后程序用数组方法shift敲掉数组的第一个元素，并将这个元素单独保存为object对象。这个元素就是要绑定的对象，也是传递给待绑定函数的第一个参数。接着，程序会返回一个关键的匿名函数。在那个闭包中，程序把待绑定函数_method和绑定对象object集合起来，并且将前面的参数数组添加到_method的参数列表当中。

下面的例子说明了各个部分的关系：

```
function ObjectA(){ /* 函数体 */ }
ObjectA.methodB = function ()
{
```

```
    // 参数现在包含6个元素:
    alert($A(arguments)); // 1、2、3、4、5、6
}

var bound = ObjectA.methodB.bind(ObjectA, '1',2',3',4');
bound('5','6');
```

通过使用call或者apply，尤其是像Prototype库这样的用法，可以使代码更加简洁，更具可读性。

3.9.3　在集合上应用函数

回调有一种特别灵巧的用法，就是把函数应用到一个数组或一个对象（两者功能十分类似）的一系列元素上。你可以对集合的任意或全部成员应用一个函数。

下面是取自jQuery （http://jquery.com）的一个非常不错的例子：

```
$("p").each(function(){
    this.innerHTML = this + " is the Element";
});
```

尽管这段代码很简短，但其间发生了很多事情。首先名为$()的函数接受一个字符串作为参数。在上例中，它会查找页面中所有的<p>元素，换言之，$("p")会返回文档中所有段落元素的集合。然后each()方法对集合中的元素逐一应用给定的函数（上例中的函数将每个段落标签的innerHTML属性都替换成了一个字符串）。

现在让我们试一下类似的例子：

```
function SpecialArray(arr)
{
  this.arr = arr;
}
SpecialArray.prototype.map = function(func)
{
  for(var i = 0; i < this.arr.length; i++)
  {
    this.arr[i] = func(this.arr[i]);
  }
  return this;
}

var obj = new SpecialArray( ['A','B','C'] );
obj.map( function(el){ return el.toLowerCase() } ); // 返回 ['a','b','c']
```

首先有一个名为的SpecialArray新对象，它有一个内部数组和一个名为map()的方法。map()方法把传递给它的一个函数逐一应用到内部数组的每一个元素上。例中的函数把数组元素从大写字母变成了小写字母。通过类似的方法，你可以随意操纵数组的元素，要完成什么功能，只要传

递相应的函数就可以了：

```
obj.map( function(el){ return el + '!' } ); // 返回 ['A!','B!','C!']
```

数组元素的类型还可以多种多样：

```
var obj = new SpecialArray( [1,2,3] );
obj.map( function(el){ return el * el } ); // 返回 [1,4,9]
```

3.9.4 可串接方法

像object.method()这样的写法我们已经看过很多。有时候为了操纵对象而频繁给变量赋值显得相当累赘。有一种办法可以让代码看起来更整洁，更简单，就是让方法可以串接在一起。这在字符串操作中很常见：

```
"I went to my store".toUpperCase().replace("MY", "YOUR");
// 返回"I WENT TO YOUR STORE"
```

由于每个方法的返回值都是字符串，所以只需在后面接上新的方法就可以接续下一步的处理。如果串接的方法很多，可以让每个方法单独占一行，这样看起来比较整洁：

```
"I went to my store"
  .toUpperCase()
  .replace("MY", "YOUR");
// 返回"I WENT TO YOUR STORE"
```

要想让方法具有串接的能力，只需要保证方法总是有返回值。只要有返回的数据，就可以继续操纵该数据。

前面例子中的SpecialArray就是一例。因为map方法总是返回它所处理的SpecialArray对象，于是我们可以一个接一个地调用map方法，一次又一次地操纵数组的元素：

```
var obj = new SpecialArray( ['A','B','C'] );
var arr = obj
  .map( function(el){ return el.toLowerCase() } ) // 返回 ['a','b','c']
  .map( function(el){ return el += '!' } ); // 返回 ['a!','b!','c!']
```

3.9.5 内部迭代器

集合在DOM的脚本处理中非常常见,从简单的数组到getElementsByTagName()调用返回的结点列表，不一而足。创建一个类（或者扩展JavaScript内建的类）可以让你在处理集合时得到更大的灵活性。内部迭代器这种机制让你通过几个暴露出来的接口浏览遍历集合的内容。

下面是一个可供浏览遍历的集合对象的例子：

```
function Collection (arr)
{
```

```
    this.current = 0;
    this.items = arr;
}

Collection.prototype.getCurrent = function()
{
    return this.items[this.current];
}
Collection.prototype.getNext = function()
{
    return this.items[++this.current];
}
Collection.prototype.getPrevious = function()
{
    return this.items[--this.current];
}

var coll = new Collection( [1,2,3,4] );
alert( coll.getCurrent() ); // 1
alert( coll.getNext() ); // 到2上
alert( coll.getPrevious() ); // 返回到1
```

主类Collection保存了指向当前元素的一个指针，并在items属性中保存了一个数组。通过getCurrent、getNext()和getPrevious()这3个方法可以在数组中来回移动，移动的同时current指针会随之更新。还可以给这个类加上前面见到的map()方法，以及对指针到达集合开头或末尾的检测。

很多JavaScript库都实现了迭代器，也为操作数据集合提供了大量便利的方法。

3.10　小结

读完本章，你已经了解了如何使用JavaScript进行面向对象编程。在继续阅读之前，你应该掌握以下内容：

- 创建对象的各种方法，以及何时选择哪种方法较为有利。
- 如何扩展对象，为对象添加方法和属性，以及何时应当使用原型。
- 如何发挥闭包的优势。
- 如何使用回调。

第4章我们将进入JavaScript库的奇妙世界，看看库提供了什么，为何要使用库以及如何使用库。

第4章 库

差不多从出现JavaScript开始，JavaScript库就以各种形式存在了。随着你经历一个又一个项目，不可避免地会重用其中的一些函数。慢慢地重用得多的函数就形成了一个核心部分，每次动手做新项目之前，你都会先把它们复制过来。好的库代码重用能带来可靠性，使用同样一段代码的项目越多，意味着代码经过了更多人的检验，错误和跨浏览器问题就更容易解决。

别人的库当然也是可以拿来用的。使用Prototype和jQuery这类现成的库能给你带来更高水平的可靠性，你自己维护的代码是很难达到同样高度的。

使用库的代价是增加了文件的大小。有些库如果不加裁减，大小会超过300KB。不过库开发者们已经意识到了这个问题，因此采用了非常模块化的方式来编写代码，让你得以单独挑出需要的功能。通过这种办法代码膨胀问题被压缩到了最低程度。像Mootools.net这类网站，不但能让你选择模块，还能让你选择是否对文件进行压缩。

按照库的目的将它们总结成3类：

- 文档对象模型（DOM）的访问、遍历及操纵；
- 应用上的便利措施，包括语言扩展；
- 界面部件。

4.1 DOM操作

由于你每天都在与HTML和层叠样式表（CSS）打交道，因此对于一个可靠的库来说，DOM可能算是最重要的接口了。这类库不但能帮你更高效地从DOM中获取元素，还能消除DOM操纵及遍历上的障碍，因为这些操作在各种浏览器上是不一致的。

现在流行的库大多具备一些处理DOM的方法，包括通过CSS选择器选择结点，通过`nextSibling()`、`previousSibling()`等函数遍历结点的集合。库通常还会提供一些在DOM中插入新元素的便利措施，因为这项操作实在是太累人了。

动画

处理动画是DOM操作中的一部分工作。能处理动画意味着能读取及修改一些DOM属性，如style对象和元素位移。动画实质上只是在相应的时间点操纵元素的属性。如果使用得当，动画可以很好地增加页面的趣味性，还能提高站点及应用的用户体验。

4.2　应用上的便利措施

Google Docs and Spreadsheets、Google Mail等应用说明了桌面正在慢慢地向Web靠拢。在这样的应用中，你要处理的不仅仅是DOM，还有庞大的数据集合。JavaScript具备一些基本的数据处理机制，比如数组和简单的迭代。不过，庞大的数据集合通常要求过滤功能，以及将数据快速加载到DOM中的方式。

库解决该问题的办法是将大部分单调沉闷的操作自动化，同时为各种JavaScript及DOM特性提供统一的应用程序编程接口（API）：

- 语言扩展和语言桥路；
- 事件处理；
- Ajax；
- 字符串和模板处理；
- 使用集合；
- 处理JSON和XML。

4.2.1　语言扩展和语言桥路

JavaScript和DOM都非常有用，但是（从上一章可以看出）它们并非为某些情况而设计的（例如深度继承，一个对象继承自另一个对象，另一个对象又继承自另一个对象……）。同样，有些浏览器实现了新的语言特性，而其他浏览器还没跟上。此时可以用语言桥路来暂时代替这些新特性，语言桥路是一种让此浏览器的特性出现在彼浏览器上的代码。Array.push()方法是个很好的例子，比较旧的浏览器如IE 5不支持这个函数。用下面这段简单的函数就能起到桥梁的作用，跨越IE 5与更先进的浏览器之间的空隙：

```
// 如果方法不存在则添加该方法
if (!Array.prototype.push) {
  Array.prototype.push = function(obj) {
    this[this.length] = obj;
  }
}
```

4.2.2　事件处理

事件处理本应归入“语言扩展和语言桥路”的类别，我把它单独分为一类是因为它太重要了。

事件处理是使用JavaScript进行Web开发时面对的最重要的问题，远远超过其他方面。库为解决这个问题提供了统一的界面，包括事件绑定、维护对象的作用域以及事件的停止。我们来看一个Prototype的例子：

```
Event.observe(element, 'click',
  (function(){ alert(this.href) }).bindAsEventListener(element)
  );
```

在Prototype中，Event对象有一个observe()方法，让你可以在指定的对象上观察事件。在本例中，你的目的是跟踪一个元素（例中是一个链接）上的单击事件。第三个参数让你传入一个函数，一旦事件发生它就会被调用。因为例子很简单，所以我只用了一个匿名函数，但请注意bindAsEventListener()方法。该方法只接受一个参数：作用域包括该函数的元素。当该函数被调用时，this即指向element。bindAsEventListener()方法利用apply()方法（参见第3章）来维护作用域。

4.2.3 Ajax

Ajax最初是异步JavaScript和XML（Asynchronous JavaScript and XML）的缩写，但后来它成了总括许多技术的一个词。不过在Ajax的核心部分，基本的概念还是围绕着通过JavaScript与服务器通信，收发数据而无需刷新页面。这些都是通过XMLHttpRequest（常缩写成XHR）对象实现的。

XHR对象最早是微软在2000年创造出来的一个ActiveX对象。后来Mozilla在2002年创造了一个XHR的原生实现，此后Safari和Opera都加入了对XHR的支持。

Ajax本身是很直截了当的，不过要加上对各种意外情况的处理就不是那么简单明了了。JavaScript库提供的框架不但能处理成功的调用，也能处理有问题的调用（例如超时）。第5章将详细讨论Ajax。

4.2.4 字符串和模板处理

在处理基于Ajax的Web程序的时候，常常要把从服务器取回的数据通过某种方式放到页面上。最快的方式是从服务器取回一整段HTML，然后塞到页面上。不过这种方式有些不实际，因为花费了许多带宽却只传递了一点点数据。模板处理可以解决这个问题，它先把从服务器取回的数据与模板合并，再嵌入到页面中。

此外，Web编程会一再用到字符串，如果能过滤和调整字符串的大小写就再方便不过了。

下面的例子使用Prototype将一个数据集与模板相结合，创建出一个链接列表：

```
<ul id="myul"></ul>
```

```
<script type="text/javascript">
var ul = $('myul');
//the dataset
var linkdata = [{name: 'About', url: '/about/'},
   {name: 'Contact', url: '/contact/'},
   {name: 'Help', url: '/help/'}];

//模板
var templ = new Template('<a href="#{url}">#{name}</a>');

//逐个添加到文档。
linkdata.each(function(conv){
   li = document.createElement('li');
   li.innerHTML = templ.evaluate(conv);
   ul.appendChild(li);
});
</script>
```

例子首先通过Prototype的$()函数取得一个无序的空列表，然后在一个普通的对象数组中声明了若干链接信息。由于Prototype会自动给所有的数组加上each()方法，所以我们可以通过each()逐一处理链接数据。链接信息数组中的每个元素都被放进模板中求值，最后输出成一个新的列表元素，然后追加到列表的后面。例中的链接数据是直接写在脚本里的，而在实际中一般会通过Ajax调用取得。

4.2.5 使用集合

集合是一个对象数组，而JavaScript中的数组功能有限。Prototype为集合操作提供了一个非常健壮的Enumerable类。类中的方法会自动扫描数组，帮你完成移除元素、添加元素、返回元素的子集等操作。

前面的例子已经提到过数组中的each()方法，可用它在数组元素中循环。迭代比起每次都写一段for循环要方便多了。

4.2.6 处理 JSON 和 XML

Ajax是少不了处理数据集的。数据集的传递格式要让人能够迅速地理解数据的结构。最流行的两种格式是JSON和XML。

JSON（http://json.org）是JavaScript对象表示法（JavaScript Object Notation）的缩写，它通过JavaScript的一个子集安全地定义及传输数据。JSON解析器在几十种服务器端语言中都能找到，因此在项目中加入JSON是轻而易举的事情。在从服务器端向客户端传送数据这个用途上，JSON正慢慢地取代XML的位置，这有以下两个原因：

- JSON格式的数据几乎总是比较小，因为定义数据所需的标签更少。
- 在客户端，JSON无论解析还是使用速度都更快，因为它本身就是JavaScript。

返回的XHR对象通过responseXML属性内建了XML支持，因此你可以通过熟悉的DOM方法来遍历XML。

JavaScript库简化了JSON和XML的处理，它们会自动检测XHR调用所返回的数据类型，然后采取相应的解析方式。有了JSON，就可以通过对数据的解析来防范服务器返回的无效和危险的信息。

4.3 界面部件

界面部件是一些预制的组件（比如文件浏览器、多标签界面以及定制的对话框），可以把它们插入到应用程序中，以完成一些独立的任务，如图4-1所示。界面部件实际上将头两个分类（DOM操纵和应用上的便利措施）结合成一台运行顺畅的机器。预制的界面部件最适合用来解决常见的设计问题。构建复杂界面时绝对避不开各种复杂的极端情况，采用预制的界面部件可以避免这种麻烦。

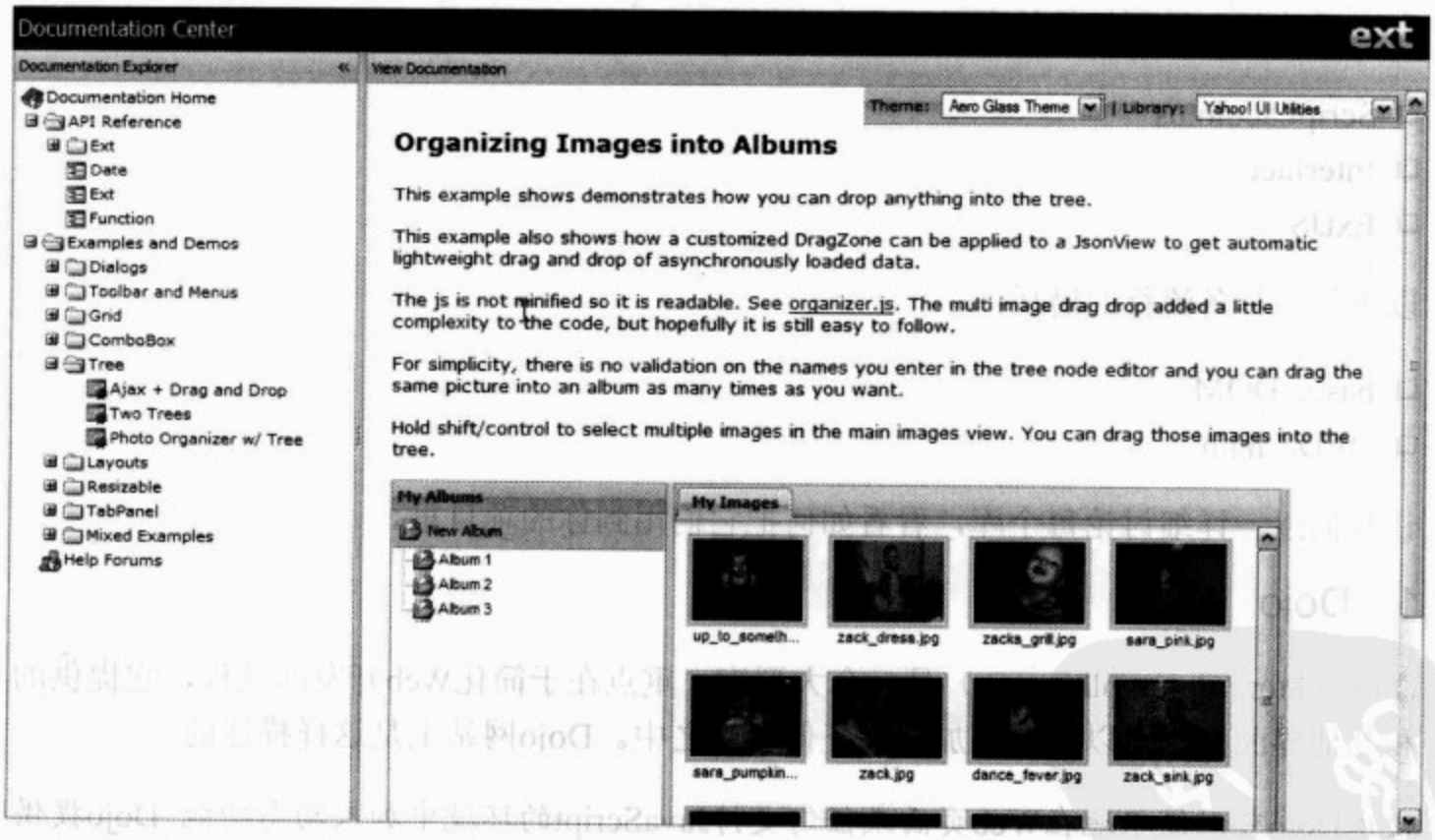


图4-1 ExtJS库中的一个样例页面，展示了树、布局元素、拖放等界面部件

4.4 流行的库

前面已经提到过一些库，不过你要知道：世上有几百个库都在做几乎一样的事情，而且每天都有新的库出现。每个库实现某项特性的方式可能稍有不同，这要看开发者觉得什么特性更

重要。

幸好其中有几个库是得到广泛支持的（包括来自大公司的支持），而每个库都有各自的优缺点。不过，通过插件或者来自竞争对手的技术，缺点常常可很快得到修补。

文档是许多开源活动的最大缺陷。你的确会发现不少库的文档既稀少又含糊，但有些库在这方面做得比较好。

目前以下几个库处于领先地位：

- Dojo
- Prototype
- jQuery
- Yahoo! UI Library (YUI)
- Mootools

专注于动画和界面部件的有以下几个库插件：

- Script.aculo.us
- Interface
- ExtJS

以下新库具备流行的潜质：

- base2.DOM
- DED|Chain

让我们逐一详细讨论每个库，看看如何把它们用到你的项目里。

4.4.1　Dojo

Dojo（http://dojotoolkit.com）是一个大型库，重点在于简化Web开发的过程，它提供的界面部件和其他界面元素可以简单地加入到任何项目之中。Dojo网站上是这样描述的：

> 使用Dojo可以简单地在Web页面或任何支持JavaScript的环境中加入动态功能。Dojo提供的组件能让你的网站可用性更高、响应更快、功能更强。用Dojo你能更简便地构建出可降级的用户界面，更快实现与prototype交互的界面部件，还能实现动画过渡。

Dojo库的着眼点在于建立一个平台，让人们在上面构建类似于桌面程序的Web应用，比如图4-2所示的E-mail应用。如果你只是想给博客加一点动画效果，用Dojo就属于牛刀杀鸡了。

Dojo涵盖了库需要解决的三个关键问题域：它使DOM处理更加顺畅，它包含了许多应用上的便利措施，而且还包含了许多预制的界面部件。

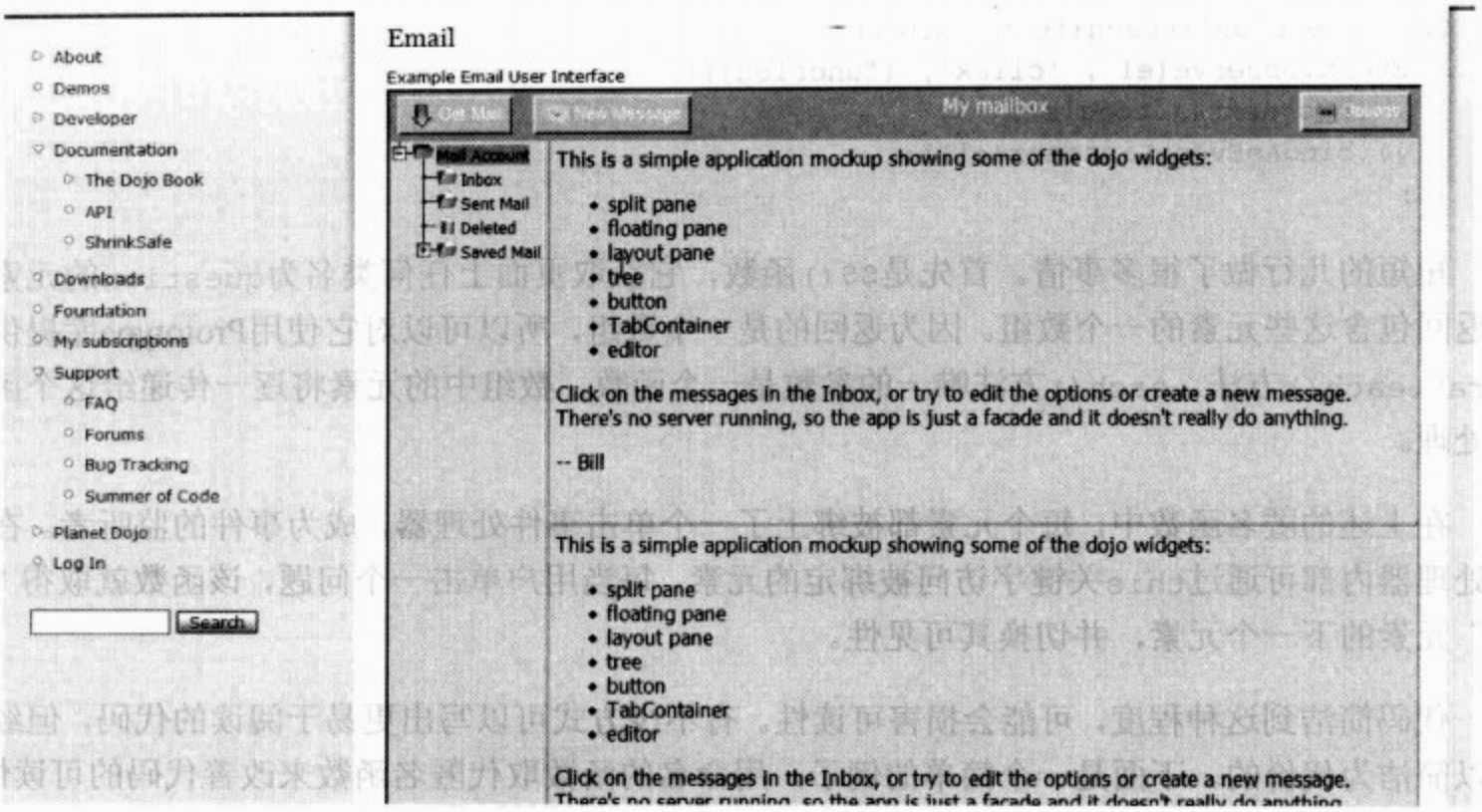


图4-2 Dojo示例，其中用到了Dojo工具包提供的许多界面部件

4.4.2 Prototype

Prototype（http://prototypejs.org）是最早得到广泛流行的库之一，而且现在使用的许多JavaScript技术都是由它推广而来的。Prototype最初由Sam Stephenson（http://conio.net）设计，后来集成到了Ruby on Rails（http://www.rubyonrails.org）中。Prototype解决问题的许多方式都能看出Ruby的风格。

Prototype流行的其中一个原因是它让许多事情都大大简化了，包括名声不好的美元符号函数（$）。通过使用$函数，返回的DOM元素上会被自动地加上许多方法，因此对元素的操纵能力将大大增加。该库最近的一些版本增强了方法链（method chaining），使它成为了一项功能强大的工具。

```
$('elementId').show(); // 使用display:none显示隐藏的某个元素
```

Prototype与其他库不同，它非常关注两个关键方面：操作DOM以及应用上的便利措施，包括许多字符串函数和一个定制的枚举对象，用于扩展一个定制的散列对象以及内建的Array对象。只要提到构建Web应用，尤其是Ajax驱动的Web应用，Prototype都是一个可靠的解决方案。

Prototype最新版本还推出了双美元符号函数（$$），用于通过CSS选择器获取元素数组。下面的例子是一个可以展开和收起答案部分的FAQ页面，你可以看出用$$函数来实现有多么简单：

```
<div class="question">What is an apple?</div>
<div class="answer">It's a fruit!</div>
```

```
$$('.question').each(function(el){
  Event.observe(el , 'click', (function(){
    this.next().toggle()
  }).bindAsEventListener(el));
});
```

短短的几行做了很多事情。首先是$$()函数，它获取页面上任何类名为question的元素，并返回包含这些元素的一个数组。因为返回的是一个数组，所以可以对它使用Prototype库提供的Array.each()方法。each()方法唯一的参数是一个函数，数组中的元素将逐一传递给这个函数去处理。

在上述的匿名函数中，每个元素都被绑上了一个单击事件处理器，成为事件的监听者。在事件处理器内部可通过this关键字访问被绑定的元素。每当用户单击一个问题，该函数就取得“问题”元素的下一个元素，并切换其可见性。

代码简洁到这种程度，可能会损害可读性。有不少方式可以写出更易于阅读的代码，但经常是以简洁为代价的。下面是一个简单的例子，用命名的函数取代匿名函数来改善代码的可读性：

```
function onEach(el)
{
  function toggle()
  {
    this.next().toggle();
  }
  Event.observe(el, 'click', toggle.bindAsEventListener(el));
}
$$('.question').each( onEach );
```

Prototype实现面向对象设计的途径完全是字面量对象风格的。可通过指定一个初始化函数来创建构造器：

```
<input type="text" id="searchfield" value="Search">

<script type="text/javascript">
var FormField = Class.create();
FormField.prototype = {
  initialize: function(id) {
    var el = $(id);
    Event.observe(el,'focus',(function()
    {
      if(this.value == this.defaultValue) this.value = '';
    }).bindAsEventListener(el));

    Event.observe(el,'blur',(function()
    {
      if(this.value == '' ) this.value = this.defaultValue;
    }).bindAsEventListener(el));
  }
```

```
};

new FormField('searchfield');
</script>
```

例子中用了Prototype创建类的方法，然后在字面量中定义initialize()函数，并把它绑定到类的原型。当调用构造器来创建一个新对象的时候，initialize()函数会自动运行。例中监听了焦点和失焦点两个事件。获得焦点的时候，如果输入框内含的值是默认值，事件处理器将清空输入框的内容，让用户重新输入。失去焦点的时候，如果输入框是空白的，事件处理器将把它设为默认值。

4.4.3 jQuery

jQuery（http://jquery.com）既快又灵活，而且在众多库中，它第一个真正突出了方法链的威力。jQuery封装得很好，它的jQuery命名空间保证了与其他库的和谐共处。它也提供了一个$函数，该函数被映射到一个内部方法。如果需要配合Prototype等库一起使用，可以在jQuery中关闭$函数，避免冲突。

jQuery虽然很简练，但仍然是极其强大的。不过说到在Web应用中实现一些类桌面的处理任务时，jQuery还是有不足的。jQuery没有模板功能，库本身也没有处理数据集的能力。如果你打算为站点增加一些交互性，jQuery是一个优秀的解决方案。

从下面的简单例子可以看出jQuery真正的亮点：

```
$("p.surprise").addClass("ohmy").show("slow");
```

这个例子可能看起来和Prototype很相似。首先请注意调用CSS选择器是通过$函数；在Prototype中，同样的功能必须使用$$函数来完成。如果在不同项目中轮番使用这两个库，要小心别在这个地方栽跟头。

通过CSS选择器取得元素之后，例中给每个元素都增加了类名ohmy。随后调用show()方法显示元素的动画效果。这段脚本会让每个类名为surprise的段落都慢慢地过滤出来。

事件处理也是使用类似的方法链完成的。下面的例子首先取得页面中所有的段落元素，然后给每个段落元素都附上一个单击事件。事件触发时，脚本会从元素中取出文本。text()是jQuery对象的一个方法。

```
$("p").bind("click", function(){
  alert( $(this).text() );
});
```

方法链运作得那么顺畅是因为jQuery每次都返回一个jQuery对象。实际上$()仅仅是jQuery函数的一个快捷方式。这个函数把自身当作是一个类，每次运行都从它自己实例化出一个新对象。

通过这种方式，jQuery对象可以被当成一个单例对象来访问（如下例所示），或者被当作一个对象生成器（如上例所示）。下例实例化一个Ajax请求，然后从中取出responseText属性，并赋值给一个变量（Ajax的运作方式详见第5章）。

```
var html = $.ajax({
  url: "/servercall/",
  async: false
}).responseText;
```

查看jQuery的文档请查阅http://docs.jquery.com。

4.4.4　Yahoo! UI Library（YUI）

YUI（http://developer.yahoo.com/yui）是由Yahoo!的员工开发和支持的。Yahoo!的很多产品都用于YUI，因此它是设计完善并且极其健壮的。YUI库的设计方式相对传统——每个方法都只是带有若干参数的函数调用。它没有jQuery的方法链接，也没有Prototype的很多方便的函数（但请参阅本章后面提到的DED|chain，DED|chain为YUI增加了方法链）。YUI是一个经过深思熟虑才产生出来的库，它具有超越本章所提到的许多库的成熟的内建功能。

YUI使用了大量的命名空间。首先是一个主要的YAHOO对象，所有其他东西都是从这个点延伸出来的。例如要想通过标识符获取元素，请使用下面的语句：

```
YAHOO.util.Dom.get('elementID');
```

在库要解决的三类问题中，YUI主要专注于DOM工具，表现出来就是Dom命名空间和Anim命名空间（用于动画）中的那些功能。它还提供了很多界面部件，在图4-3中可以看到其中的一些：

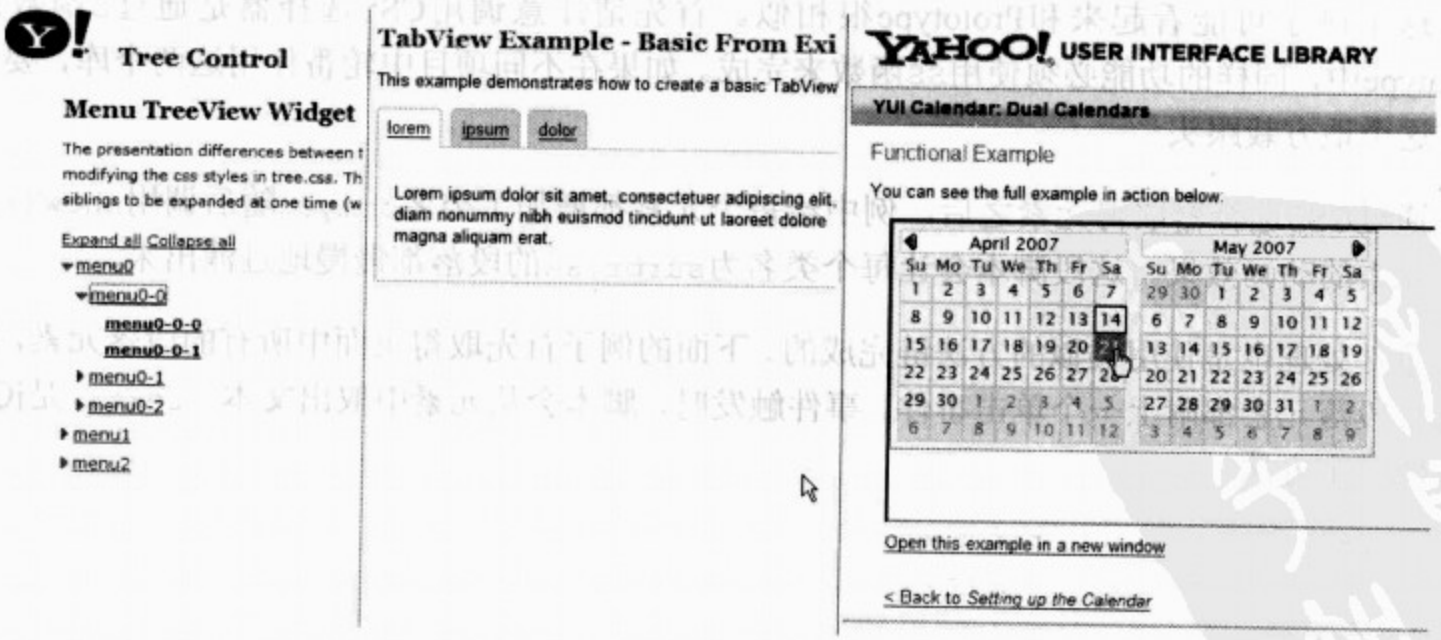


图4-3　YUI库提供的TreeView、TabView和Calendar控件

这些界面部件能帮助你快速地在应用程序中添加复杂的功能。日历部件是其中非常常用的一

个，可以用它来选择日期。

```
function selEvent(type, args)
{
   // type = event type ='select'

   // 所选数据是数组中的第一个元素
   var dates = args[o];
   // 单击的日期是数组的第一个元素
   var date = dates[o];
   // 日期以[YYYYY, MM, DD]的格式存储在一个数组中
   var year = date[0], month = date[1], day = date[2];
}
widget = new YAHOO.widget.Calendar("cal1","calwidget",{close:true,iframe:true});
widget.selectEvent.subscribe(selEvent, this, true);
widget.render();
```

日历部件有3个参数：第一个参数是日历部件本身的唯一标识符，第2个是一个HTML元素的ID，该元素充当日历部件的占位符，第3个是存放设置选项的字面量对象。选项的意思是显示一个关闭按钮，并且把一个<iframe>放在日历部件的后方。这个<iframe>的作用是覆盖在<select>框的上面，因为<select>不允许其他HTML元素在它上面显示。这主要是为了解决IE 6及以下版本存在的问题，IE 7已经修复该问题。

过长的命名空间看起来很啰嗦，不过其实定义快捷方式是很容易的。下面的代码就模仿Prototype定义了一个$函数，并指向Dom对象的get()方法：

```
var $ = YAHOO.util.Dom.get;
var el = $('elementID');
```

利用第3章中描述的封装技术，你可以在函数调用或者类的内部创建快捷方式，从而使全局命名空间尽量保持干净。

4.4.5 Mootools

Mootools（http://mootools.net）相对而言算是库界的新丁。它最早叫做Moo.fx，是在Prototype库基础上建立的一个效果库。它的开发者们是找到一个机会创造出了这个精简而且模块化的库。Mootools一大优点是它的下载配置程序（见图4-4），你可以通过它选择自己需要库中的哪些模块，还可以选择需要的压缩级别。模块间的依赖关系由程序自动处理。你还可以决定使用压缩版本的库还是带有完整文档的版本。使用最精简的代码可以减少文件的大小，从而减少带宽的占用。很多其他库，如jQuery和YUI，都提供了精简的版本。

注解 在项目的开发阶段，最好使用库的非压缩版本。这样调试起来更容易，因为非精简形式的代码更容易跟踪。在Mootools目前的1.05版中，如果使用精简形式的代码，Firebug会报告存在不正确的函数名。总之在正式发布的时候切换到压缩版本就可以了。

图4-4　Mootools的下载配置程序

Mootools的诸多特性中我最喜欢的是，它不但能给元素设置动画，而且还能一次设置多个元素：

```
var myElementsEffects = new Fx.Elements($$('a'));
myElementsEffects.start({
   '0': { //改变第一个元素的不透明度和宽度
     'opacity': [0,1],
     'width': [100,200]
   },
   '4': { //以及第5个元素的不透明度
     'opacity': [0.2, 0.5]
   }
});
```

例中用到了Fx.Elements类，它也像Prototype一样使用$$()函数；在这里$$()函数被用来传入一个链接列表。start()方法启动动画，它的唯一参数是一个设置各选项的对象。该字面量对象通过键来指出哪个元素需要动画效果，例中是第1个和第5个（从0数起）。第1个元素的不透明度从0%增加到100%（即从完全不可见变到完全可见），并且宽度从100px增加到200px。第5个元素的不透明度从20%增加到50%。

如果你有许多相互依赖的动画，比如在一个区域扩大的同时，其他一些区域要相应缩小，这时能一次设置多个元素就非常方便了。

4.4.6 Script.aculo.us

Script.aculo.us（http://script.aculo.us）是一个动画及界面部件库，它也是在Prototype的基础上构建的。联合使用Prototype和Script.aculo.us相当之流行，不少服务器端的框架都把它们当作默认的配置。

只需简单几行就能快捷地应用效果：

```
new Effect.Opacity('myElement',
    { duration: 2.0,
      transition: Effect.Transitions.linear,
      from: 1.0, to: 0.5 });
```

该类的第一个参数是元素的ID（或元素本身），第二个参数是设置各选项的一个字面量对象。上面的例子在2秒的时间段内将元素的不透明度从100%降低到50%。通过transition属性可以用数学方式来精确地设定动画的过渡，让效果看起来更自然。过渡可以先慢后快，也可以先快后慢，甚至可以先来回变化几个回合再停留在最终的效果上。

Script.aculo.us的控件是它最出彩的部分，在任何项目中加入Script.aculo.us控件都超级简单。下面是一个可排序列表的例子：

```
Sortable.create("firstlist",
    {dropOnEmpty:true,
     containment:["firstlist","secondlist"],
     constraint:false});
```

Sortable控件把一个列表中的元素都变成可拖动的，因此每个元素都可以被拖到表中的其他位置，达到排序的效果。甚至可以在两个列表之间拖放列表元素，比如上例中元素就可以在firstlist和secondlist之间拖动。

4.4.7 ExtJS

ExtJS（http://extjs.com）是一个界面部件库，但它的优雅和灵活超越了这里提到的所有库。过去它曾经叫做YUI.Ext，因为当时它是专门用于YUI库的一个增强包（正如Script.aculo.us与Prototype的关系）。但是它在接近1.0版发布的时候经过了一次改写，从此ExtJS可以搭配YUI、jQuery及Prototype使用。而到了现在的1.1.版，ExtJS又增加了一个独立版本，不再依赖其他库。

ExtJS库的文档也使用了它本身的组件，包括树和布局控件，如图4-5所示。

ExtJS特别适合用来建立应用程序原型，因为在它的帮助下，很多特性实现起来都非常简单。图4-5左侧类似文件浏览器风格的导航，只需要几行代码就可以实现：

```
Ext.onReady(function(){
    // 速记
    var Tree = Ext.tree;
```

```
    var tree = new Tree.TreePanel('tree-div', {
        animate:true,
        loader: new Tree.TreeLoader({
            dataUrl:'get-nodes.php'
        }),
        enableDD:true,
        containerScroll: true
    });

    //  设置根结点
    var root = new Tree.AsyncTreeNode({
        text: 'Ext JS',
        draggable:false,
        id:'source'
    });
    tree.setRootNode(root);

    // 呈现树
    tree.render();
    root.expand();
});
```

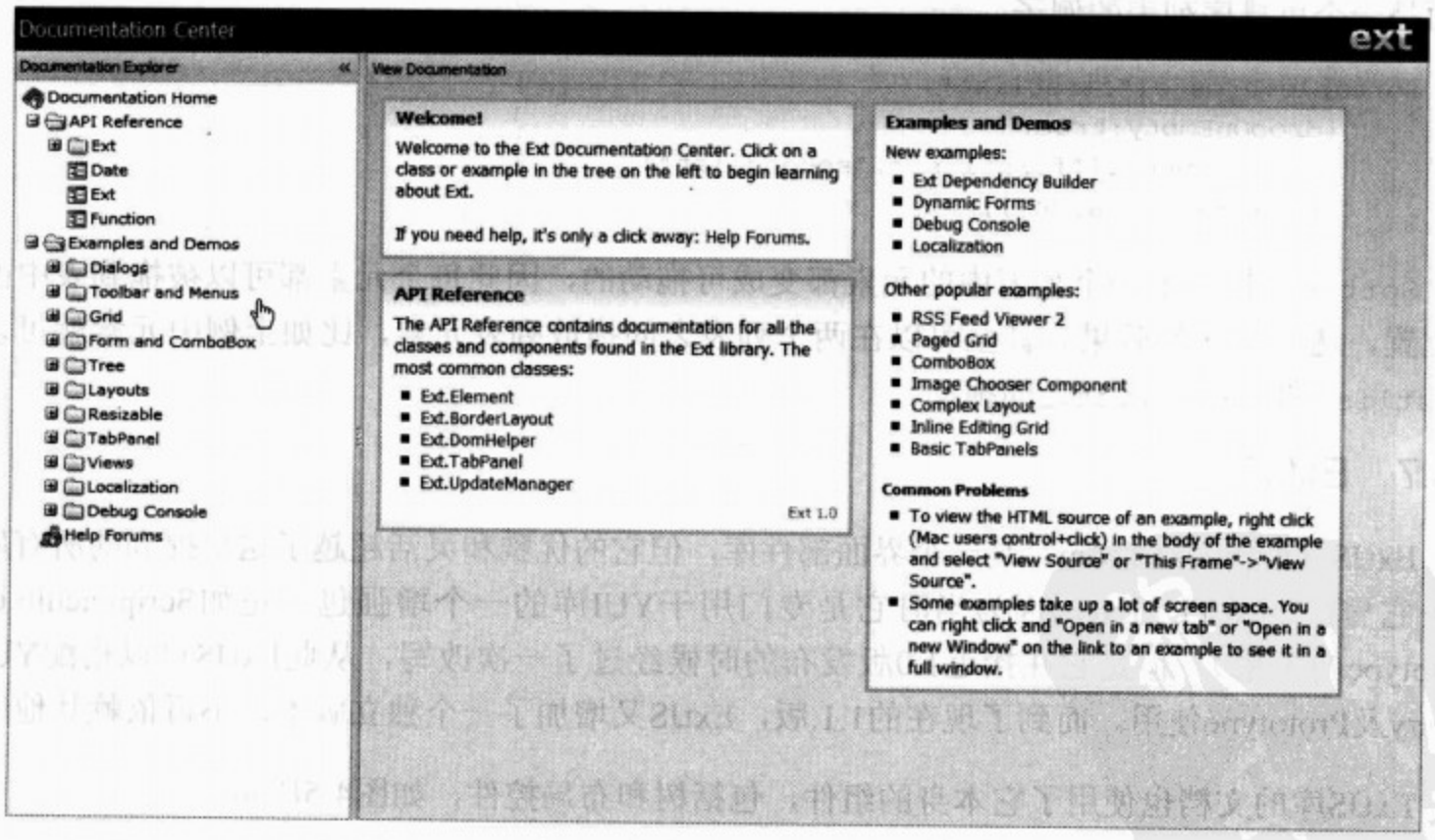


图4-5　ExtJS的文档，其中使用了ExtJS的界面部件

Ext.js有个`onReady()`函数，DOM准备好之后就会执行它，一般都正好发生在`window.onload`事件触发之前。jQuery也有个类似的函数，Prototype也可通过Dan Webb的LowPro插件（http://www.danwebb.net/lowpro）得到同样的功能。

树结构是由Ext.tree命名空间里的若干对象处理的：TreePanel、TreeLoader和AsyncTreeNode。TreePanel有两个参数：第1个是面板父元素的ID，第2个参数是用来设置选项的字面量对象。选项当中有一个loader，它通过TreeLoader对象从服务器加载数据。在上例中，数据是从一段PHP脚本获得的JSON对象。

然后就是创建一个新结点，并把它设为树面板的根结点。根结点渲染之后执行了展开操作。结点展开的时候会使用TreeLoader对象加载子结点。一旦数据被加载，就会被缓存在客户端，因此后续的收回/展开操作就不用再调用服务器了。

树结点的扩展性非常强，可以在树结点上增加其他属性。结点的外观也是完全可定制的，从图4-6就可以看出来。

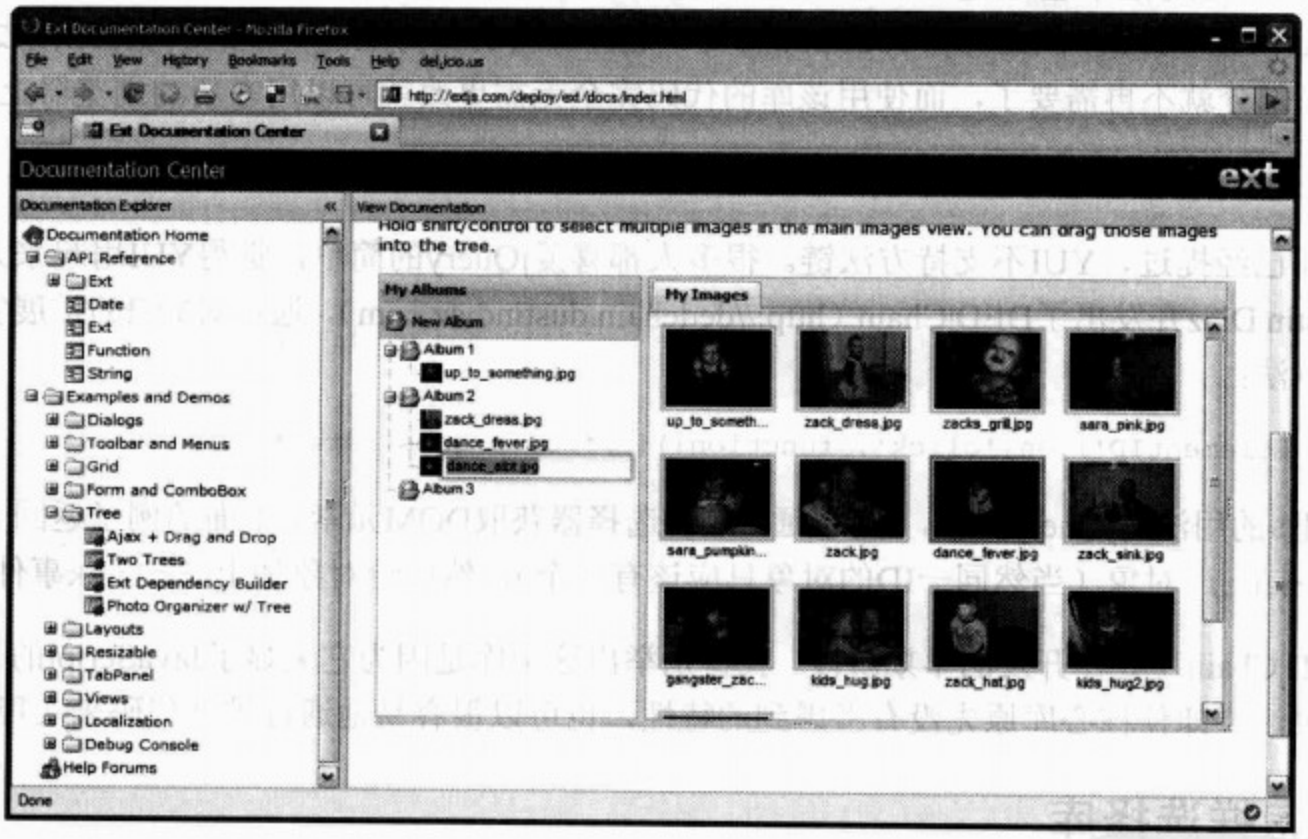


图4-6 使用了ExtJS的TreePanel部件的相册

ExtJS库的界面部件包括定制的对话框（而不必使用警告窗口和弹出窗口）、多标签界面、数据网格、布局，还有很多。

ExtJS并不仅仅是一个界面部件库，因为界面部件背后的很多组件都拿出来单独使用，成为应用上的便利措施，比如内建的DOM工具、事件处理、各个状态管理类，还有用于XML和JSON处理的数据格式类。

4.5 新出现的库

显然谁都不会满足于“还过得去”——而且还可能出于很好的理由。各个库都在为满足越来

越多用户的需求而持续不断地调整和加入新的功能。为了解决现有库无法满足的需求，总免不了有些人会尝试打造自己的库。

4.5.1 Base2.DOM

Dean Edwards为JavaScript的普及贡献良多（包括他为修复IE 6和IE 7的CSS问题而写的IE7Scripts），他总结出了自己的一套DOM工具（http://dean.edwards.name/weblog/2007/03/yet-another）。

Dean在库的开发上选择了另一条路径：按照W3C（World Wide Web Consortium）和JavaScript 1.5（http://developer.mozilla.org/en/docs/Core_DavaScript_1.5_Reference）规范来设计API，他的想法是提供一些规范中已经定义，但某些浏览器还不支持的特性（希望将来会支持），比如`addEventListener()`、`Array.forEach()`等。这样当新的支持上述特性的浏览器出来之后，库中的相应部分就不再需要了，而使用该库的代码完全无需更改就可以用在新的浏览器上。

4.5.2 DED|Chain

前面已经提过，YUI不支持方法链。很多人都喜爱jQuery的简单，觉得YUI用起来太麻烦。于是Dustin Diaz开发出了DED|Chain（http://dedchain.dustindiaz.com），通过对YUI的扩展实现了可链接的方法：

```
_$('#elementID').on('click', function(){ /* code goes here */ });
```

这里$的用法与jQuery类似，$可以通过CSS选择器获取DOM元素。上面的例子返回了所有ID为的`elementID`对象（当然同一ID的对象只应该有一个），然后给对象加上`onclick`事件处理器。

DED|Chain还处在开发的早期阶段，我特别举出这个库是因为它突显了JavaScript的一项关键特性：灵活。即使核心库原先没有考虑到的特性，也可以很容易地通过扩展代码来实现。

4.6 怎样选择库

面对那么多库，还不断有新的库加进来，怎样才能缩小选择的范围呢？当然你的选择完全取决于要满足的需求。本章开头就说过，库一般可被划分为3大类：DOM工具、应用程序的辅助和界面部件。首先应该从这3个方面来考察你打算构建的程序，缩小选择的范围。

如果只是想给博客增加一些交互性，比如简单的滑动效果，你所选择的库应该专注于核心DOM特性，以及一些基本的效果。例如Mootools或者jQuery都很合适。

如果你打算构建Web应用，并且需要操纵数据集和建立复杂的用户界面原型，那么结合使用Prototype和ExtJS是比较好的方案。

考察一个库的时候，务必动手试验，还应该看一下源代码。你必须对库的结构有很好的理解，

才能发挥出它的威力。而且熟知库的结构之后，同类之间的比较才更有效果。

4.6.1 社区

在搜索引擎上进行一些调查，看看有哪些人在用某个库，他们都遇到些什么问题。社区的人气能说明一个库的可靠程度。你还可以顺便找到一些资源，比如博客和论坛，万一遇到问题的时候也好有个去处。

4.6.2 文档

由于很多库都是由很小的团队在业余时间开发和维护的，文档显然会落到To-do列表的底部。比如Prototype也是最近才解决了文档的问题。之前除了一些第三方资源，这个库完全没有文档。后来一群人团结起来才保证了库本身的开发能继续下去，并且建立了与其名声相称的网站和文档。

提示 选择库的时候，务必将文档纳入考察。文档是否最新？有没有适当的例子？

幸好，随着越来越多的人使用这些库，互联网上会出现越来越多的例子，文档也会越来越全。JavaScript库几乎总是开源的，欢迎你也为库的开发做出贡献。

4.7 小结

这个领域已经成熟，你可以从中选取适合下一个项目的库。每次都重新进行发明是不必要的。本章列举的库都很流行，也就是说有成千上万（乃至上百万）的人在使用它们，你自己从头开始写出来的代码是很难胜过这种质量保证的。使用这些库可以节省很多时间，无论是跨浏览器开发的一致性，还是测试、维护都能体现出使用库的优越性。每个库都有各自的优点和缺点，选择合适的工具能让你事半功倍。

接下来，我们将讨论Ajax和视觉效果。在接下来的两章以及案例研究里，你会看到JavaScript库这个主题再次出现，并且会看到它们是如何应用到项目中的。

第5章 Ajax和数据交换

如果你认为Ajax这个词使用起来很随意，你可能会以为它是对JavaScript的新的代称。Ajax这个词是Adaptive Path公司的Jesse James Garrett（可能还有）在不久之前（2005年）造出来的。Garrett（可能还有他们公司的其他人）造这个词是为了方便描述一种特殊的交互：异步JavaScript和XML（Asynchronous JavaScript and XML）。

通过Ajax技术，JavaScript可以和服务器通信，而且服务器用XML格式返回结果。然后可以用服务器返回的新数据更新用户在屏幕上看到的内容，在整个过程中不需要刷新页面。这样做的好处在于整个过程是异步的——通信工作都在幕后执行，与此同时用户可以继续操作当前的页面。

本章涵盖以下内容：

- 解构Ajax的过程；
- 理解数据格式；
- 构建一个可重用的Ajax对象；
- 使用JavaScript库来处理Ajax调用。

首先我们来看一个Ajax的例子，看它能达到怎样的效果。

5.1 分析 Ajax 应用

Google Docs & Spreadsheets以及很多Google在线应用出色地示范了Ajax的威力。在Google Docs & Spreadsheets应用中，对文档的修改会持续不断地发送回服务器保存起来。如图5-1所示。这种交互过程和桌面应用如Microsoft Word很相似。除了用来使网络应用的体验接近于桌面应用程序，Ajax技术用在一些较小的方面也是很有效的，它可以大大提高网站的响应能力，也能提高用户的乐趣。

请看图5-2的例子，在上面可以给感兴趣的项目加上星号。如果用传统的方式来实现，单击星号会使得整个页面重新载入（整个16KB的页面内容，不包括JavaScript、CSS和图像在内）。而

用Ajax来实现的话，用户的注意力不会被页面刷新打断，而且来回传输的数据只有几个字节。

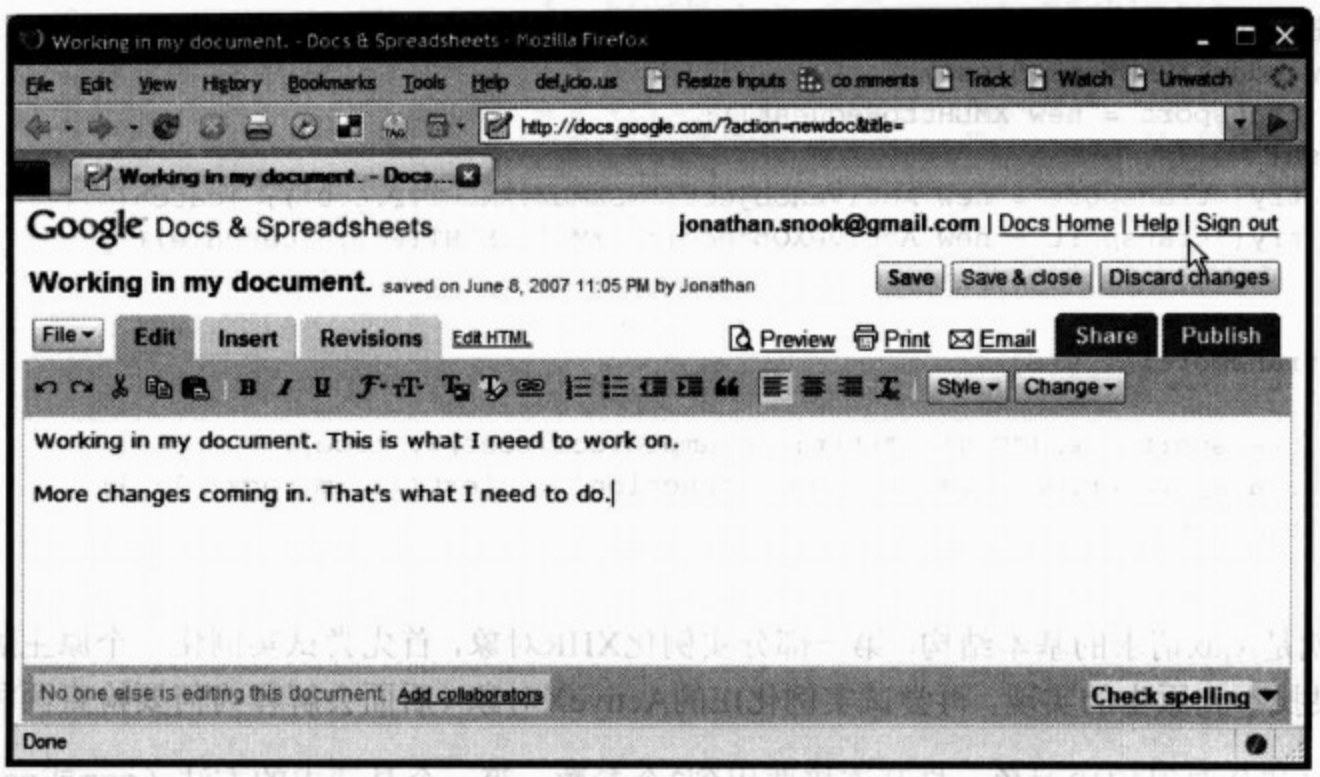


图5-1 Google Docs每隔几分钟自动保存用户的更改，防止丢失数据

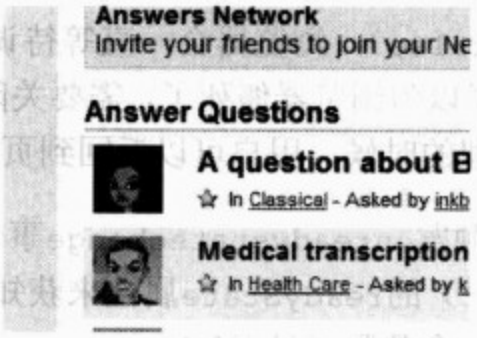


图5-2 对感兴趣的项目设置星号标志，无需刷新页面

5.2 解构 Ajax 过程

Ajax的核心是XMLHttpRequest对象（常被简称为XHR）。最早可以追溯到2000年，微软为了在Outlook Web Access中使用而开发了这么一个ActiveX对象。但在缺乏跨浏览器支持的情况下，这个功能几乎已退出人们的视线。到了Mozilla实现了它的原生版本的XHR对象之后，事情出现了转机。最后随着Safari和Opera的相继实现，XHR开始发展了。现在这个可爱的词已经发展成为一个行业了。

虽然Ajax这个词字面上所指的是非常明确的一种交互类型，不过实际中的含义已经延伸到了较大的范畴：通过JavaScript使用XMLHttpRequest对象进行的所有服务器通信。返回的数据可以是XML，也可以是HTML、JSON、CSV，甚至是任何你觉得合适的文本格式。

发出Ajax请求的过程相当直截了当。下面是一个简单的例子：

```
//使用为所有人准备的自有版本，而不是IE 6以下的版本
if(window.XMLHttpRequest) {
    transport = new XMLHttpRequest();
}else{
    try{ transport = new ActiveXObject("MSXML2.XMLHTTP.6.0"); }catch(e){}
    try{ transport = new ActiveXObject("MSXML2.XMLHTTP"); }catch(e){}
}

if(transport)
{
    transport.open("GET", "http://example.com/test/", true);
    transport.onreadystatechange = function(){ alert('I am back!'); };
    transport.send();
}
```

这就是Ajax请求的基本结构。第一部分实例化XHR对象，首先尝试实例化一个原生版本的对象。如果找不到原生的实现，再尝试实例化IE的ActiveX对象（后面会解释为何要检查两种实现）。

第二部分取得XHR对象，打开连接要用到3个参数：第一个是请求的方法（GET或POST），第二个是要打开的URL，第三个参数决定调用是同步的还是异步的。

如果第三个参数设为同步（false），浏览器会一直等待调用返回，期间用户不能做任何事情。这种方法的坏处是用户可能误以为浏览器僵死了，需要关闭。如果把参数设为true，则调用是异步的，当调用正在后台被处理的时候，用户可以返回到页面上继续操作。

XHR请求的状态变化会多次触发onreadystatechange事件处理器。在该事件处理器内部，可通过检查XHR对象（transport）的readyState属性来获知调用的状态。readyState属性在某一时刻的取值将是0到4之间的一个整数（见表5-1）。

表5-1　readyState的取值

取　值	状　态	说　明
0	未初始化	还没有调用XHR对象的open方法
1	加载中	还没有调用send方法
2	已加载	已调用send方法；响应首部（response header）已就位
3	交互	响应数据正在下载之中，可通过XHR对象的responseText属性取得
4	完成	所有操作都已完成，完整的结果可从XHR对象的responseText属性（和responseXML属性）取得

一般来说标准的做法是检查的readyState取值是否为4（对XHR来说相当于window.onload）：

```
transport.onreadystatechange = function(){
  if(transport.readyState == 4)
```

```
    {
      alert('I am done!');
    }
};
```

差不多就这样了。当然还有很多细节需要考虑，我们且从请求/响应过程开始说起。

5.2.1 Ajax 的请求/响应过程

对于传统的请求过程来说，用户发出对数据的请求，然后等待服务器发回响应，然后等待浏览器渲染页面。而在Ajax的环境下，可以大大减少来回的数据传输量。对数据的请求也可以和用户在页面上的其他操作同时进行，整个过程不需要用户发起请求，用户也不必等待响应序列和页面刷新。响应只需要改变当前文档对象，不需要影响整个页面（以及图片和CSS等未缓存的资源）。接下来就可以通过JavaScript更新DOM的内容而无需刷新页面。

图5-3展示了Ajax和非Ajax两种过程之间的区别。总而言之，Ajax意味着响应程度更高的界面，完成同样任务所需的时间也较短。

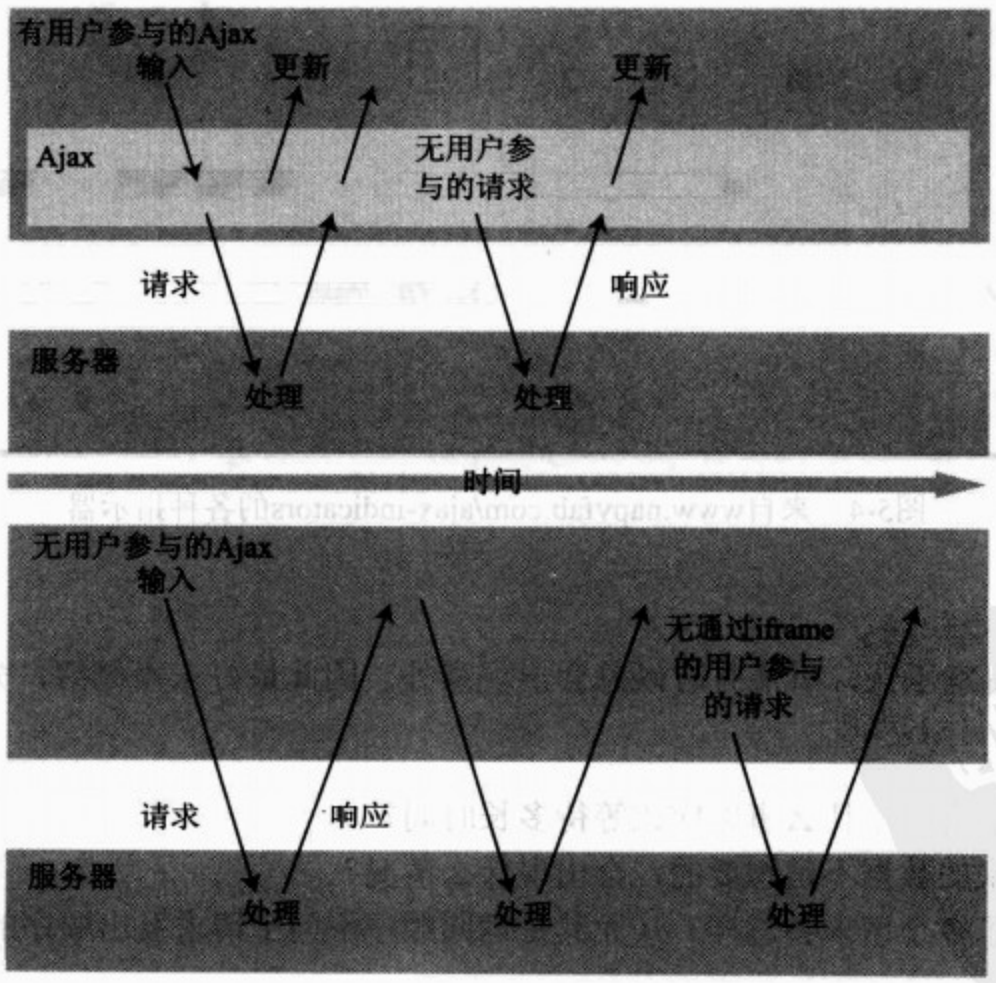


图5-3 无论有没有用户参与，Ajax都可以更新客户端（与传统非Ajax交互对比）

设计任何Ajax方案的时候，都应该考虑到用户。与浏览器默认行为不符的设计会使用户处处受挫。如果某段程序死机或者处理的时间过长，用户并不能了解真实的情况，而只会认为网站坏

掉了。

如果请求是用户发起的，就应该告知用户事情正在处理当中。一般可以在靠近交互发起的地方放一个动画指示器（见图5-4），让用户知道现在要等一等，等待期间用户可以继续浏览页面的其他部分。

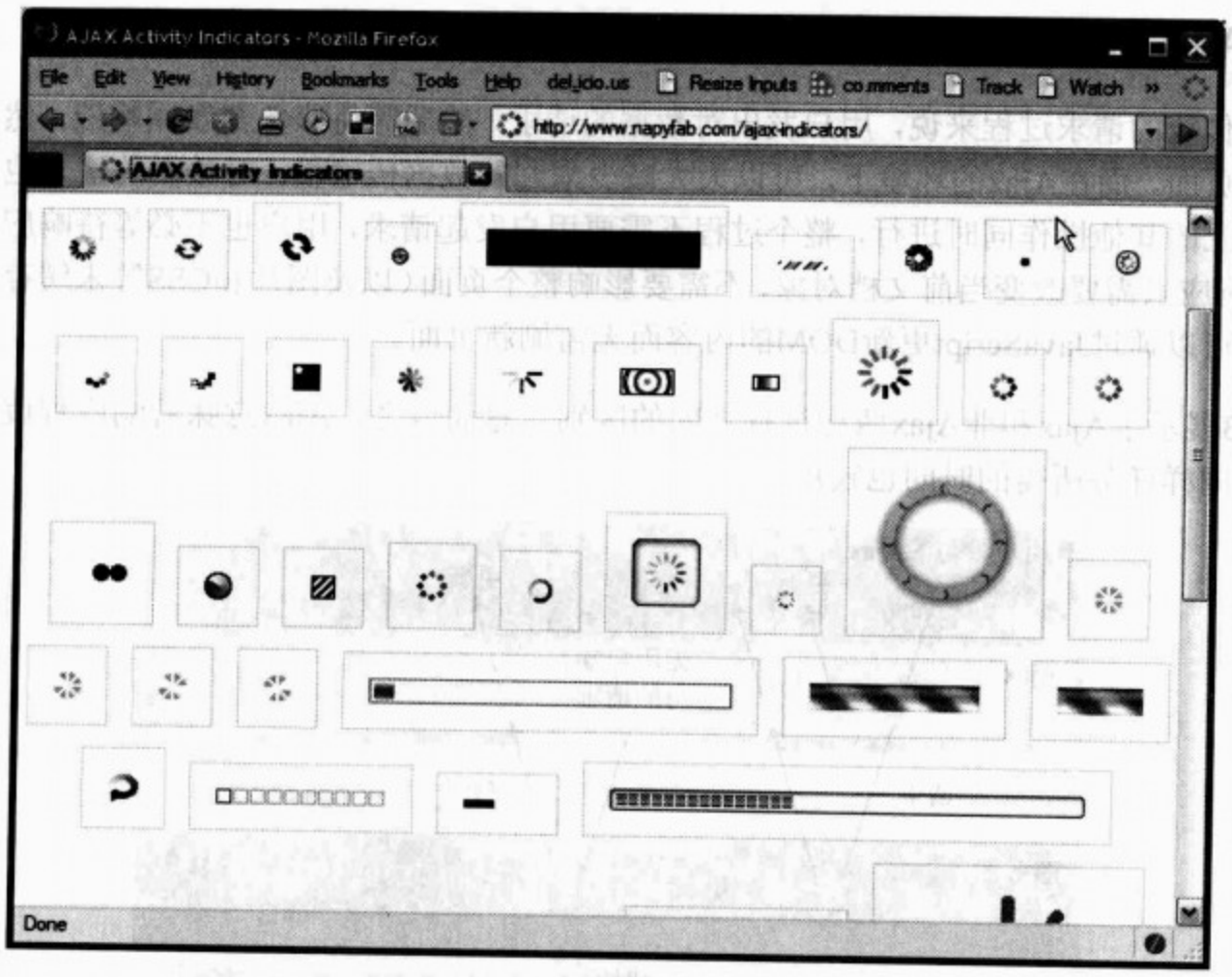


图5-4　来自www.napyfab.com/ajax-indicators的各种指示器

5.2.2　失败

不管你是否愿意承认，事情有时候总会出现意外。因此最好未雨绸缪，先问自己以下几个问题：

- 请求超时了会发生什么事？应该等待多长时间？
- 要是取回来的数据不是想要的，会出现什么情况？
- 如果发出了多个请求会怎样？（尤其是返回顺序不同于请求发出顺序的情况！）

下面将逐一研究这些问题，并把问题的解决方法整合成一个可重用的库。

5.2.3　绘制故事板

计划一个Ajax应用的过程中要考虑很多交互点，Yahoo!称之为interesting moments（感兴趣的

时刻)。图5-5是Yahoo!为拖放功能绘制的故事板矩阵的一角。我建议你在计划应用的交互点的时候也采取类似的做法。

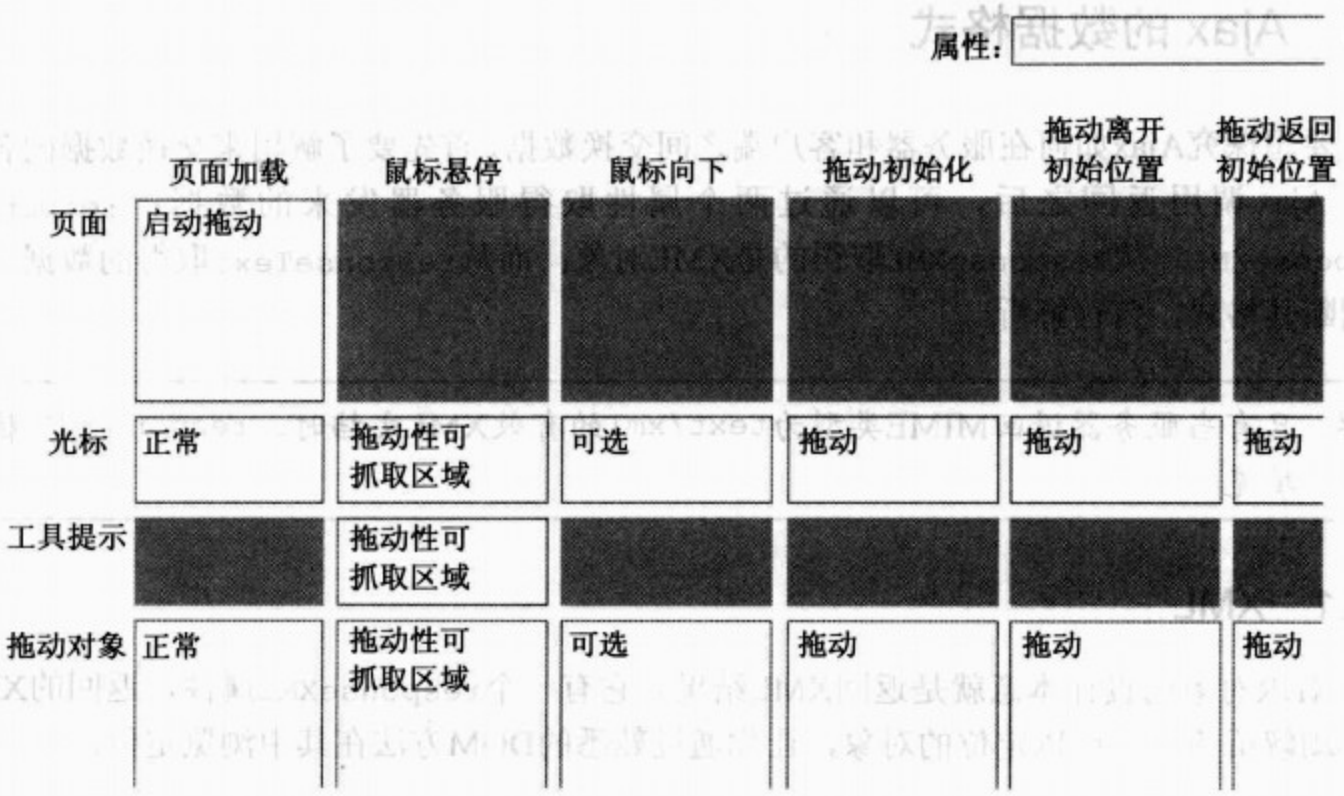


图5-5 Yahoo!提供的故事板矩阵，参见http://developer.yahoo.com/yui/dragdrop/#storyboard

Yahoo!的故事板矩阵是一张表格，对象列在左侧，事件列在上方。我还喜欢画一张流程图来说明可能发生的交互，以防止遗漏掉某些情况。流程图还可以帮助我完整地考虑整个过程，而不是一下子就跳到结果。

图5-6是一张流程图，上面描述了更新页面上一个HTML表格中的数据时会遇到的各种情况。图中甚至考虑到了用户会话过期的情况——在需要身份验证的应用中经常会遇到。

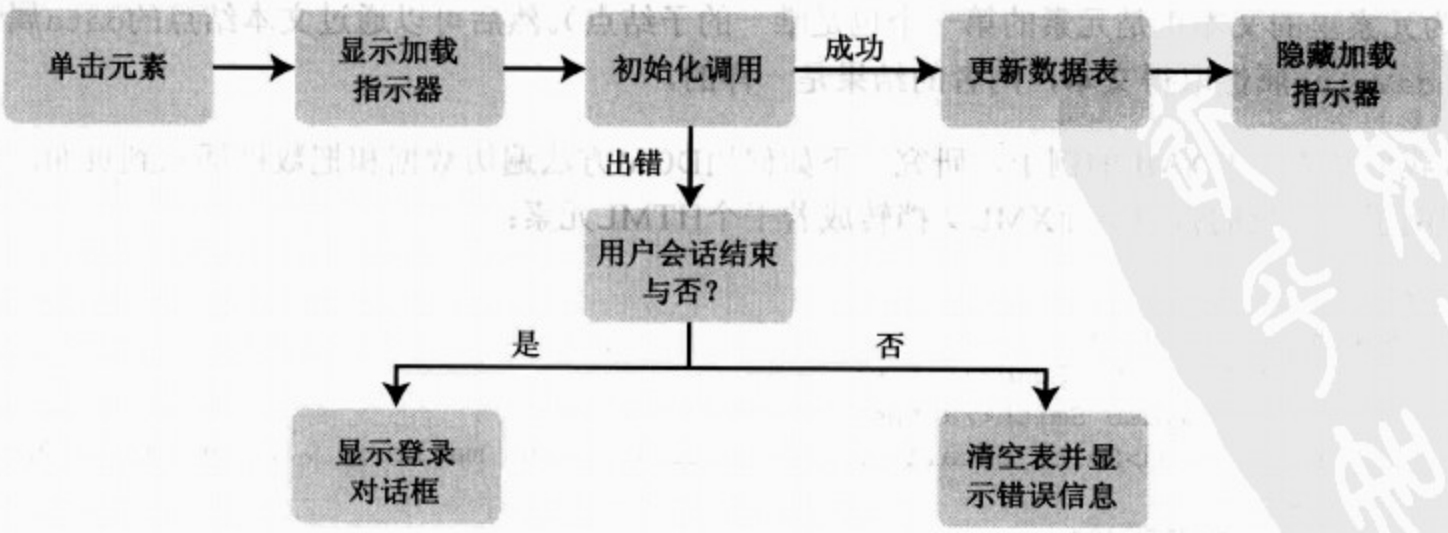


图5-6 流程图的例子，解释了更新数据表时可能发生的交互过程

看完如何规划一个Ajax界面的实现，接下来我们将解释交换数据的各种方法。再下一步就动手打造一个可重用的Ajax对象，并且防范好各种意外情况。

5.3 Ajax的数据格式

要想探究Ajax如何在服务器和客户端之间交换数据，首先要了解用来交换数据的各种数据格式。Ajax调用返回之后，可以通过两个属性取得服务器发来的数据：responseText和responseXML。从responseXML取得的是XML对象；而从responseText取得的数据，需要你自己判断其格式，自行解析。

注解　只有当服务器返回MIME类型为text/xml的有效XML文档时，responseXML属性才会被填充。

5.3.1 XML

XHR对象的设计本意就是返回XML结果。它有一个responseXML属性，返回的XML数据会被自动解析成一个可以定位的对象，让你通过熟悉的DOM方法在其中浏览定位。

```
var doc = transport.responseXML.documentElement; // 抓取根结点
var songs = doc.getElementsByTagName('song'); // 获得所有的song结点
for(var i=0;i<songs.length;i++)
{
    // 假设每个结点仅包含文本内容，抓取文本结点并显示其内容。
    alert('I love ' + songs[i].firstChild.data);
}
```

XML的DOM和HTML用法有些不同，因为少了一些HTML中的便利，所以取得数据的方法稍有差异。从前面的例子可以看出，在XML里，文本内容也是结点，必须用firstChild来取得（因为元素里的文本正是元素的第一个也是唯一的子结点）。然后可以通过文本结点的data属性或者nodeValue属性取得文本，两者的结果是一样的。

我们来看一些XML的例子，研究一下如何用DOM方法遍历数据和把数据插入到页面。在这个的例子里，我们将把下面XML文档转成若干个HTML元素：

```
<root>
  <book id="id15669">
    <title>The Long Road</title>
    <author>Hayden Smith</author>
    <description>Smith details his battles from the mailroom ...</description>
  </book>
  <book id="id15670">
    <title>Time: fact or fiction</title>
```

```
    <author>Dr. Michelle Doe</author>
    <description>Is time just a figment of our imagination?...</description>
  </book>
</root>
```

下面是HTML文档的基本结构，其中还包括给将要插入的数据准备的样式。书籍列表会插入到ID为books的元素里面：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
  "http://www.w3.org/TR/html4/strict.dtd">
<html lang="en">
<head>
<title>Books</title>

<style type="text/css">
body {
  font-family:Arial,Helvetica,sans-serif;
  }
.book {
  border-top:1px solid #CCC;
  padding:10px 5px
  }
.book h2 {
  margin:0;
  font-size:1em;
  }
.book .author {
  margin:0;
  font-weight:bold;
  font-size:.9em;
  }
.book p {
  margin:0
  }
</style>
</head>
<body>
  <div id="books"></div>
</body>
</html>
```

我们的目标是遍历XML中的每一项，并把它转成下面的HTML结构：

```
<div class="book" id="id15669">
  <h2>The Long Road</h2>
  <p class="author">Jonah Smith</p>
  <p>Smith details his battles from the mailroom to the CEO of Megacorp</p>
</div>
```

收到Ajax调用的响应之后，用getElementsByTagName()取得所有书籍元素，通过循环逐一为每本书创建相应的HTML元素和文本结点，把每本书组织成一个小枝丫，然后添加到books元素，成为页面的一部分：

```
var doc = transport.responseXML.documentElement; // 抓取根结点

var books = doc.getElementsByTagName('book'); // 得到所有的song结点

var container = document.getElementById('books');
var book, title, author, description, text;
for(var i=0;i<books.length;i++)
{
  // 创建book容器
  book = document.createElement('div');
  book.className = 'book';
  book.id = books[i].getAttribute('id');

  // 创建书名
  title = document.createElement('h2');
  text = document.createTextNode(books[i].childNodes[1].firstChild.data);
  title.appendChild(text);
  book.appendChild(title);

  // 创建作者信息
  author = document.createElement('p');
  author.className = 'author';
  text = document.createTextNode(books[i].childNodes[3].firstChild.data);
  author.appendChild(text);
  book.appendChild(author);

  // 创建描述信息
  description = document.createElement('p');
  text = document.createTextNode(books[i].childNodes[5].firstChild.data);
  description.appendChild(text);
  book.appendChild(description);

  // 将整个book结点添加到文档中
  container.appendChild(book);
}
```

上面的代码取得第1、3、5个元素。请记住空白文本结点也被认为是元素，因此要跳过它们。幸好这一点在所有浏览器（包括IE）上都是一致的。元素的firstChild是文本结点，data属性取得的正是结点里的文本内容。最后产生的HTML如图5-7所示。

按这样的风格来使用DOM方法非常繁琐。另一种办法是让Ajax调用直接把完整的HTML结构嵌在一个XML结点里返回给客户端。学习这种方法首先必须清楚理解XML的语法和行为——尤其是编码和文档的合法性。

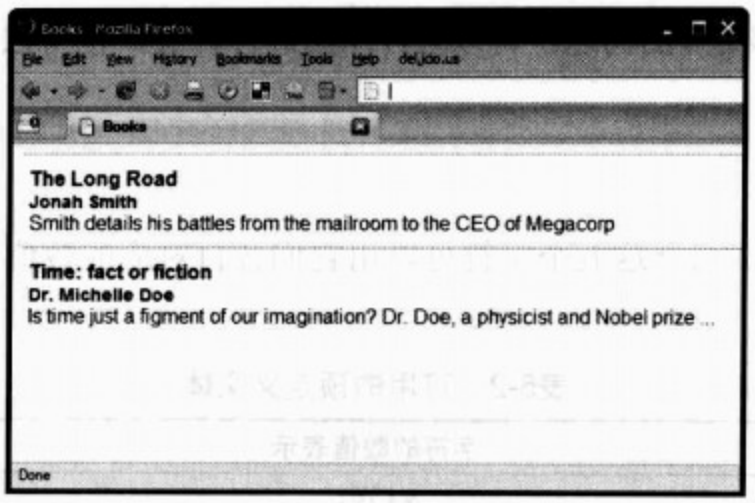


图5-7 XML文档转换成HTML的最终输出

1. XML合法性和编码

XML是非常严格的语言，它不允许任何错误（俗称“Draconian error handling”，即苛刻的错误处理）。在大多数情况下严格是好的，但如果对使用场景考虑不周，可能会误中埋伏。观察下面的两个例子，看你能不能看出其中的问题所在：

```
<myhtml>This is some content I want embedded on the page</myhtml>
<myhtml>This is some <strong>content</strong> I want embedded on the page</myhtml>
```

按照先前的做法把以上元素嵌入到页面：

```
var doc = transport.responseXML.documentElement; // 抓取根结点
// 得到所有的myhtml结点，我有其中的一个结点
var embedhtml = doc.getElementsByTagName('myhtml')[1];
// 抓取将要嵌入html的页面上的元素
var el = document.getElementById('placeholder');
el.innerHTML = embedhtml.firstChild.data;
```

第一个<myhtml>元素一切正常，整个字符串完好地嵌进了页面。但第二个<myhtml>就出问题了，只有“This is some”出现在页面上。这是因为标签被看作是一个单独的结点，如图5-8所示。

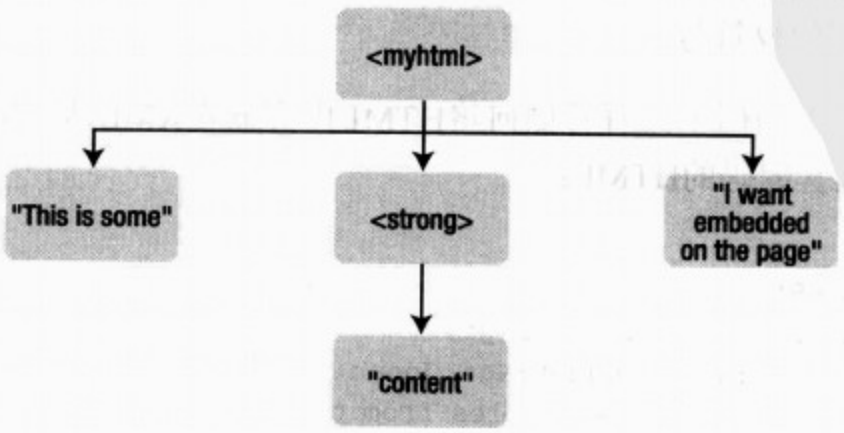


图5-8 实际的XML结构

有两种办法绕过嵌套结点的问题：可以对HTML进行转义，也可以将HTML内容嵌套进一个CDATA结点。

2. HTML编码

在XML里，&、<、>、'、"这五个字符可以用它们各自对应的数值表示形式或者预定义实体来替换，如表5-2所示。

表5-2　可用的预定义实体

字　符	字符的数值表示	预定义实体
&	&#38;	&amp;
<	&#60;	&lt;
>	&#62;	&gt;
'	&#39;	&apos;
"	&#34;	&quot;

了解了上述内容，你就可以重新编码前面的例子了：

```
<myhtml>This is some &lt;strong&gt;content&lt;/strong&gt;
   I want embedded on the page</myhtml>
```

如果你熟悉HTML的编码字符，很可能反而掉进另一个陷阱：在XML里使用HTML实体。XML遇到不认识的HTML编码字符一样会出错而导致失败。

```
<myhtml>This page copyright by Smith & Smith</myhtml>
<myhtml>This page copyright by Smith &amp; Smith</myhtml>
<myhtml>&copy; Smith &amp; Smith</myhtml>
```

第一行的错误很明显，&符号没有被编码。第二行没问题，正确地对&进行了编码。第三行试图使用代表版权符号的HTML实体（©），会导致XML的处理过程出错。要改正很简单，只要对HTML实体里的&字符再进行一次编码就行了：

```
<myhtml>&amp;copy; Smith &amp; Smith</myhtml>
```

从XML中获取该结点中的文本的时候，&会被转回&字符，然后在插入到HTML DOM的时候，©会被转回版权符号。

再回到前面书籍列表的例子，看看如何将HTML内容嵌在XML结点里返回给客户端。XML将需要在每个结点中包含编码的HTML：

```
<root>
  <book id="id15669">
    &lt;h2&gt;The Long Road&lt;/h2&gt;
    &lt;p class=&quot;author&quot;&gt;Jonah Smith&lt;/p&gt;
    &lt;p&gt;Smith details his battles from the mailroom to the CEO of Megacorp&lt;➥
/p&gt;
  </book>
```

```
  <book id="id15670">
    &lt;h2&gt;Time: fact or fiction&lt;/h2&gt;
    &lt;p class=&quot;author&quot;&gt;Dr. Michelle Doe&lt;/b&gt;&lt;/p&gt;
    &lt;p&gt;Is time just a figment of our imagination? Dr. Doe, a physicist and➥
Nobel prize ...&lt;/p&gt;
  </book>
</root>
```

这样一来，将数据添加到页面就简单多了：

```
for(var i=0;i<books.length;i++)
{
  // 创建book容器
  book = document.createElement('div');
  book.className = 'book';
  book.id = books[i].getAttribute('id');
  book.innerHTML = books[i].firstChild.data;

  // 将整个book结点添加到文档中
  container.appendChild(book);
}
```

注意加粗的部分，这一行将每个书籍结点的内容放进一个充当容器的div元素。甚至还可以让服务器返回整个要插入的HTML片段，以进一步简化客户端的代码，当然HTML片段同样要经过转义：

```
<root>
  &lt;div class=&quot;book&quot; id=&quot;id15669&quot;&gt;
    &lt;h2&gt;The Long Road&lt;/h2&gt;
    &lt;p class=&quot;author&quot;&gt;Jonah Smith&lt;/p&gt;
    &lt;p&gt;Smith details his battles from the mailroom to the CEO of Megacorp&lt;➥
/p&gt;
  &lt;/div&gt;
  &lt;div class=&quot;book&quot; id=&quot;id15670&quot;&gt;
    &lt;h2&gt;Time: fact or fiction&lt;/h2&gt;
    &lt;p class=&quot;author&quot;&gt;Dr. Michelle Doe&lt;/b&gt;&lt;/p&gt;
    &lt;p&gt;Is time just a figment of our imagination? Dr. Doe, a physicist and➥
Nobel prize ...&lt;/p&gt;
  &lt;/div&gt;
</root>
```

这样就不需要再遍历任何元素了，直接把内容添加到页面即可：

```
var doc = transport.responseXML.documentElement; // 抓取根结点
var books = doc.getElementsByTagName('book'); // 得到所有的song结点
var container = document.getElementById('books');
container.innerHTML = doc.firstChild.data;
```

但是，大量经过转义的HTML很难阅读，也很难看出返回的HTML是否正确。

3. CDATA结点

可以用CDATA节来替代文本结点。W3C规范（http://www.w3.org/TR/2004/REC-xml-20040204/#sec-cdata-sect）对CDATA节定义得很清楚：

> CDATA段可以出现在字符数据能够出现的任何地方，它们用于转义包含会被识别为标记的字符的文本块。CDATA节以字符串"<![CDATA["开始，以字符串"]]>"结束。①

CDATA节比前一种做法灵活得多，可以大大减少需要操心的编码问题，处理起HTML内容就容易得多了：

```
<myhtml><![CDATA[&copy; <strong>Smith &amp; Smith</strong>]]></myhtml>
```

获取CDATA节的内容和获取文本结点的内容做法是一样的。前面书籍列表的例子改用CDATA节之后变成下面的样子：

```
<root><![CDATA[
  <div class="book" id="id15669">
    <h2>The Long Road</h2>
    <p class="author">Jonah Smith</p>
    <p>Smith details his battles from the mailroom to the CEO of Megacorp</p>
  </div>
  <div class="book" id="id15670">
    <h2>Time: fact or fiction</h2>
    <p class="author">Dr. Michelle Doe</b></p>
    <p>Is time just a figment of our imagination? Dr. Doe, a physicist and➥
Nobel prize ...</p>
  </div>
]]></root>
```

现在查看HTML内容一点都不困难。嵌套在XML结点里的整个HTML片段还是照原来一样处理：

```
var doc = transport.responseXML.documentElement; // 抓取根结点
var books = doc.getElementsByTagName('book'); // 得到所有的song结点
var container = document.getElementById('books');
container.innerHTML = doc.firstChild.data;
```

4. XSLT

采用XML作为数据格式的理由之一是可以利用XSLT。XSLT是一种转换语言，可将XML文档转换成其他格式（通常是另一种XML格式，如XHTML）。但不幸的是，其跨浏览器支持要么

---

① 这段译文引自裘强翻译的XML规范的简体中文译本（http://lightning.prohosting.com/~qqiu/REC-xml-20001006-cn.html#sec-cdata-sect）。请注意XML规范应以英文版为准（http://www.w3.org/TR/2004/REC-xml-20040204/）。

——译者注

很慢，要么就错误频频，因此有一些专为解决此问题而开发的库：

- **Google AJAXSLT**（http://goog-ajaxslt.sourceforge.net/），Google开源了他们的跨浏览器XSLT库，支持的浏览器包括Safari 1.3+、Opera 7.5+、IE 6+和Firefox 1+。
- **Sarissa**（http://dev.abiss.gr/sarissa/），Sarissa是对一些XML API的跨浏览器包装，这些API包括XMLDocument、XMLElement、XMLHttpRequest、XMLSerializer和XSLTProcessor。

由于在客户端使用XSLT的目的几乎总是将XML文档转换成HTML，所以建议你直接在服务器上创建HTML片段，避开难缠的跨浏览器问题。

5.3.2 XML之外的选择

XML很好用，但缺点也很明显。在文档中浏览可能非常复杂和繁琐，处理各种跨浏览器的问题也毫无乐趣可言。也许我们可以试试responseText属性，探索一下别的可能。用字符串发送响应有很强的功能，因为你可以把字符串转换成更有用的东西。

1. HTML

最简单轻巧的办法就是发回一串HTML，客户端只要把收到的HTML片段插进文档就行了：

```
var htmlSnippet = transport.responseText; // 抓取文本响应
var el = document.getElementById('placeholder');
el.innerHTML = htmlSnippet;
```

注解 如果返回的HTML片段含有<script>标签，那么标签中的脚本代码是不会被执行的。

2. JavaScript

如果有一些在特定情况下需要执行的JavaScript，但又不想对其进行缓存，可以通过responseText返回代码，然后用eval()执行：

```
eval(transport.responseText); // 抓取文本响应
```

这种技巧不常用。把JavaScript放在<head>里的<script>元素中才是最实际的做法。不过该技巧有一种变体非常流行，已经成为一种优秀的数据传输方法，甚至超过了XML。那就是JSON（JavaScript Object Notation，JavaScript对象表示法）。

由于JSON拥有与XML相似的结构，且能很好地与JavaScript应用融为一体（因为它就是JavaScript），所以日益流行。

JSON的语法是对象表示法的一个子集，它的设计本意是为了更好地与其他编程语言交换数据。JSON对象是一个字面量对象，但只允许包含以下几种类型：字符串、数值、数组和其他字面量对象。字符串必须用双引号括起来（JSON还要求把键用双引号括起来）。

下面的JSON对象保存了某个购物车里的物品和数量。它包含两个字面量对象：fruits和vegetables，两者分别包含若干物品：

```
var shoppingCart = {
   "fruits": {
      "apples":5,
      "apricots":4,
      "oranges":6,
      "mangos":5
    },
    "vegetables":{
       "celery":2,
       "lettuce":1,
       "green peppers":5
    }
};
```

要引用里面的celery元素，可以这样写：

```
shoppingCart.vegetables.celery; // 或者...
shoppingCart["vegetables"]["celery"];
```

你可能注意到"green peppers"中间有个空格。因为它是一个字符串，所以空格是完全允许的。但要注意，如果在成员名字里用了空格，那就不能用点语法来引用它了。所以只能用方括号语法来引用"green peppers"属性：

```
shoppingCart.vegetables["green peppers"]; // 或者...
shoppingCart["vegetables"]["green peppers"];
```

.NET、PHP、Java等语言都有JSON的解析器，能把JSON对象转成对应语言的原生对象。在JSON网站上可获得更多的信息：http://www.json.org。

注解　字符串应该用双引号（"）括起来，而不应该用单引号（'）。虽然在用JavaScript来执行代码的时候，单引号也不会出错，但如果用JSON解析器（服务器端和客户端）来解析JSON对象，就很可能出问题。

JSON数据从服务器返回之后，也像其他格式的数据一样需要解析。因为存在代码注入攻击的危险（绝对不要假设收到的数据是安全的），所以强烈建议采用客户端解析器，JSON网站上就提供了这样一个解析器。

```
var obj = transport.responseText.parseJSON();
if(obj) performMagic(obj);
```

接下来可以进一步检查对象中是否包含你所需的数据，然后进行正常的处理。

3. 符号分隔的字符串

符号分隔的字符串用指定的字符将几个值分隔开。例如URL里的查询字符串是用&符号分隔的，而键/值对是用等号分隔的。

```
search=my+search+phrase&sortBy=title&page=2
```

将查询字符串转成JavaScript对象要分为两步。第一步在&符号处将字符串断开，第二步再分开键/值对：

```
var qs = "search=my+search+phrase&sortBy=title&page=2";
var data = qs.split('&');
for(var i=0;i<data.length;i++)
{
    data[i] = data[i].split('=');
}
alert(data[1][1]); //对"title"报警
```

最后会得到一个多维数组。要想访问其中的某个键/值对，要么必须知道它在数组中的位置，要么就只能遍历数组第一维的所有项了。如果把键/值对变成对象，访问起来会容易一些：

```
var qsObject = {}; // 对象存储
var qs = "search=my+search+phrase&sortBy=title&page=2";
var data = qs.split('&');
var tmp;
for(var i=0;i<data.length;i++)
{
    tmp = data[i].split('=');
    qsObject[tmp[0]] = tmp[1];
}
alert(qsObject.sortBy); // 对"title"报警
```

请记住，如果查询字符串中有重复的键，出现在后的会取代出现在前的。

也可以用CSV（comma-separated value，逗号分隔的值）格式将字符串分成若干部分。每条记录另起一行，每个字段用逗号分开。解析一行CSV数据并不只是将字符串从逗号处隔开那么简单。请看下面的例子：

```
"Mr. Smith, Esq.", 2006, January, 26,Need to get in touch
```

注意第一个字段里的逗号。第一个字段用了一对双引号括起来，表示那才是完整的字段。那要是字段里面又有引号该怎么办呢？情况就更复杂了。要是有空白的记录又如何？那就更困难了。只有在数据简单、分隔容易的情况下，这种字符串解析方法才有实用价值。

用JSON之外的其他格式返回数据，通常意味着必须在JavaScript里解析数据之后才能使用，于是JSON就成为了普遍使用的格式。

注意　无论你多么相信返回的数据是可靠的，事实都可能相反。请保持警惕并预做准备。

5.4　构造可重用的Ajax对象

现在你已经初步了解了Ajax和它如何来回传数据，然后我们探讨一下怎样建立一个对象，供你在项目中使用。

首先要创建一个可以实例化的对象。因为每次发起Ajax请求都要实例化这个对象，所以应该把它定义成一个可重用的类：

```
function Ajax()
{
  var transport;
  if(window.XMLHttpRequest) {
    transport = new XMLHttpRequest();
  }else{
    try{ transport = new ActiveXObject("MSXML2.XMLHTTP.6.0"); }catch(e){}
    try{ transport = new ActiveXObject("MSXML2.XMLHTTP"); }catch(e){}
  }
  if(!transport) return;
  this.transport = transport;
}

Ajax.prototype.send = function(url, options)
{
  if(!this.transport) return;
  var transport = this.transport;
  var _options = {
      method:"GET",
      callback:function(){}
      };
  // 覆盖各个选项
  for(var key in options)
  {
    _options[key] = options[key];
  }

  transport.open(_options.method, url, true);
  transport.onreadystatechange = function(){ _options.callback(transport) };
  transport.send();
}
```

在构造函数里首先确定应该用哪个XHR对象。第一步检查XMLHttpRequest对象，因为所有的现代浏览器都支持它，包括Firefox 1+、Safari 1.2+、Opera 7.6+和IE 7+。对于IE 5和IE 6用户（或

者关闭了原生XHR对象的IE 7用户），尝试实例化ActiveX版的XHR对象。

实例化ActiveX版XHR对象的语句要用try/catch块括起来，因为如果IE浏览器禁用了ActiveX对象，会产生一个警告对话框。先检查新版本的，没有再退回到旧版的。（更多内容请参见5.4.1节。）

完成构造函数之后，还要定义一个send()方法来向服务器发送请求。它定义了两个参数：要请求的URL和一个保存选项的对象。

注解 记住，用字面量对象来传递可选的参数是保持代码清晰、简洁的好办法。

选项有两个：请求的方法（GET或POST）和回调。每当Ajax对象的readyState属性发生变化，这里指定的回调就会被调用。接下来把选项保存到内部的_options对象，打开URL，挂上事件处理器，然后就可以发送请求了。

注解 如果打算重用这个Ajax对象，让它发送多个请求，那么必须在open调用之后声明onreadystatechange事件，否则在IE 5和IE 6上只有第一次调用能成功执行，其后都会失败。

现在我们看一下如何使用这个崭新的Ajax对象：

```
function processRequest(transport)
{
  if(transport.readyState == 4)
  {
    var obj = transport.responseText.parseJSON();
  }
}
```

这就是处理响应的回调函数。它的唯一参数就是我们的XHR对象transport。在回调函数内部，检查XHR对象的状态，如果状态值等于4，就说明已经成功返回了结果。然后用JSON库解析响应结果，将之转成一个JavaScript对象。

一切准备就绪，可以发送请求了：

```
var ajax = new Ajax();
ajax.send('/path/to/script', {callback:processRequest});
```

指定要调用的URL和处理响应的回调函数——processRequest()。我没有指定请求的方法，所以程序将采用默认值GET。

不同版本的ActiveX对象

如果你曾经对各种Ajax方案有过走马观花的认识，可能会注意到它们用了很多种XMLHttpRequest对象。微软的XML实现（称为MSXML）有很多种版本。

列举如下：

- MicrosoftXMLHTTP；
- Msxml2.XMLHTTP；
- Msxml2.XMLHTTP.3.0；
- Msxml2.XMLHTTP.4.0；
- Msxml2.XMLHTTP.5.0；
- Msxml2.XMLHTTP.6.0。

微软的XML团队曾写过一篇文章详细解释各版本之间的区别和各自的适用范围（但仍然没能完全澄清）：http://blogs.msdn.com/xmlteam/archive/2006/10/23/using-the-right-version-of-msxml-in-internet-explorer.aspx。

MSXML 1.0和2.0已经不被微软所支持。4.0版从未随操作系统发布，5.0版是给Microsoft Office用的特殊版本。Microsoft.XMLHTTP和Msxml2.XMLHTTP这两个没有版本号的ID，目前直接映射到3.0版（即使机器上安装了6.0版）。因此只剩下两个“program ID”（或按微软的习惯称为progID）是我们需要考虑的：

- Msxml2.XMLHTTP；
- Msxml2.XMLHTTP.6.0。

6.0版随IE 7引入，其中包含了对若干缺陷的修补（例如前面提过，在IE 5和IE 6中，onreadystatechange必须在open调用之前声明的缺陷）。虽然IE 7自带了原生的XHR对象，但该对象有可能被用户禁用，所以最好还是检查一下Msxml2.XMLHTTP.6.0是否存在，如果不存在就回退到Msxml2.XMLHTTP。

5.5 为失败做准备

现在基本的Ajax对象已经就位，让我们检查一下还有哪些没考虑到的问题：

- 如果请求超时会发生什么事？应该等待多长时间？
- 如果取回的数据与预期不符该怎么办？
- 如果发出了多个请求会发生什么情况？（尤其是在响应返回的顺序和请求发出的顺序不一致的情况下！）

5.5.1 超时处理

Ajax调用一般会一直保持打开状态，直到服务器关闭连接。如果遇到服务器响应不畅的情况，用户可能会觉得等待时间太长了。更好的办法是等待一段时间后让调用超时，并处理超时错误。为此我对Ajax对象作了相应的修改，请见加粗部分：

```
function Ajax()
{
  var transport;
  if(window.XMLHttpRequest) {
    transport = new XMLHttpRequest();
  }else{
    try{ transport = new ActiveXObject("MSXML2.XMLHTTP.6.0"); }catch(e){}
    try{ transport = new ActiveXObject("MSXML2.XMLHTTP"); }catch(e){}
  }
  if(!transport) return;
  this.transport = transport;
}

Ajax.prototype.send = function(url, options)
{
  if(!this.transport) return;
  var transport = this.transport;
  var aborted = false;
  var _options = {
    method:"GET",
    timeout:5,
    onerror:function(){},
    onsuccess:function(){}
    };

// 覆盖各个选项
for(var key in options)
{
_options[key] = options[key];
}

function checkForTimeout()
{
  if(transport.readyState != 4)
  {
    aborted = true;
    transport.abort();
  }
}
setTimeout(checkForTimeout, _options.timeout * 1000);
function onreadystateCallback()
{
```

```
      if(transport.readyState == 4)
      {
        if( !aborted && transport.status >= 200 && transport.status < 300 )
        {
          _options.onsuccess(transport);
        }else{
          _options.onerror(transport);
        }
      }
    }

    transport.open(_options.method, url, true);
    transport.onreadystatechange = onreadystateCallback;
    transport.send('');
}
```

代码中增加了不少东西，让我们一段一段地解释吧：

```
var aborted = false;
var _options = {
    method:"GET",
    timeout:5,
    onerror:function(){},
    onsuccess:function(){}
    };
```

aborted变量是稍后用来决定是否要手动终止调用的标志。_options对象中增加了一个timeout变量，定义的是放弃请求之前应该等待的秒数。_options对象中还去掉了callback属性，取而代之的是onerror()和onsuccess()两个函数。

```
function checkForTimeout()
{
  if(transport.readyState != 4)
  {
    aborted = true;
    transport.abort();
  }
}
setTimeout(checkForTimeout, _options.timeout * 1000);
```

这个函数在超时之后调用，检查是否成功返回了响应结果。如果没有，就把aborted变量设为true，表示必须手动结束这次调用——通过XHR对象的abort()方法。这时将自动触发onreadystatechange事件处理函数。

```
function onreadystateCallback()
{
 if(transport.readyState == 4)
 {
```

```
    if( !aborted && transport.status >= 200 && transport.status < 300 )
    {
     _options.onsuccess(transport);
    }else{
     _options.onerror(transport);
    }
   }
}

transport.open(_options.method, url, true);
transport.onreadystatechange = onreadystateCallback;
transport.send();
```

onreadystateCallback()函数负责处理onreadystatechange事件。该函数会检查XHR对象的状态，并据此决定要分发到onsuccess还是onerror事件处理器。我先检查了调用是否手动终止的，然后检查HTTP状态代码是否在200到300的区间——这个区间的状态代码表示调用是成功的。HTTP状态代码通过XHR对象的status属性取得。

最后，onreadystatechange事件处理器原来是通过选项对象传进来的，现在换成了内部的事件处理函数。

5.5.2 HTTP 状态代码

浏览器发出一次调用，服务器就返回一个响应。在响应里面有一个状态代码，能告知浏览器一些重要的信息。要想详细了解所有状态代码，请查阅W3C的HTTP/1.1规范（www.w3.org/Protocols/rfc2616/rfc2616-sec10.html）。

最想见到的状态代码当然是200——表示成功的响应。在200区间的代码都表示成功。300区间的响应表示重定向。浏览器会自动处理重定向，转到新的页面文档——这次应该得到200响应状态。400区间是客户端错误。请求可能没有正确发送，也可能请求的页面不存在——也就是可怕的404错误！最后，500区间表示某种服务器错误。对于Ajax请求来说，我们只希望得到200区间的响应。前面的例子已经对此作了防范，它会检查状态代码是否大于等于200且小于300。

```
if( !aborted && transport.status >= 200 && transport.status < 300 )
```

5.5.3 多重请求

在Ajax网站和应用中，多重请求是司空见惯的。有两种场景必须为之做好准备：

- 第一次请求已经发出，而后续的请求应该覆盖掉第一次请求。例如，用户在搜索框里填写内容然后按下回车，但在调用返回之前，用户意识到刚才的输入有误，于是作了修改并再次按下回车。在这种情况下，用户并不想要第一次的搜索结果，只想要第二次的。所以你应该检测到用户发出了第二次请求，并决定是否应该覆盖掉第一次的请求。
- 第二种场景是连续发出了多次请求，但调用返回的顺序是不定的。例如一个聊天程序要

不断轮询服务器以获得新的消息，消息的返回次序应该和调用发出的次序一致。

要想保证调用的返回次序，大致上等于用一个异步系统来模仿一个同步系统的行为。这时可以用一个令牌来跟踪每次调用。令牌可以用一个整数值来充当，每次调用的时候递增这个整数就可以了。

然后只有当令牌能对上号的时候才交给回调函数处理，如果中间缺了号，那就一直等到缺号的响应返回之后再继续下去，或者等到超时之后放弃缺号的响应。

5.5.4　意外的数据

不应该假定服务器返回的数据是正确的。前面我们已经有了一个onerror事件处理函数防备着服务器丢回什么意想不到的东西。接下来，我们继续改善错误检查功能，让它更上一层楼。

如果打算以特定的格式返回数据，如XML或JSON，应该在服务器端布置一种应急方案，让它在结果里返回某种错误代码。然后让客户端在处理服务器发回的结果之前，先检查错误代码。如果服务器返回的不是想要的内容（例如无效的JSON对象，或者未处理的服务器端错误），客户端也要相应作出处理。

下面是一个JSON格式的错误消息的例子：

```
{"error":{"id":1,"message":"Your session has expired"}}
```

onsuccess事件处理函数的代码如下：

```
var UNKNOWN = 0;
function processRequestSuccess(transport)
{
  var obj = transport.responseText.parseJSON();
  // 如果JSON解析没有执行，则不会有对象存在，不管怎样，这都意味着服务器失败
  if(!obj)
  {
    processError(UNKNOWN);
    return;
  }
  // 如果我的对象里有一个error属性，服务器会返回一条错误信息表示操作失败
  if(obj.error)
  {
    processError(obj.error.id, obj.error.message);
  }
  // 按照正常情况继续处理请求
  // ...
}
```

错误代码和消息作为参数传给processError()函数处理。它可以显示一个警告对话框，也可以将错误消息写到页面上。

5.6 用库处理 Ajax 调用

一路过来，我们已经看到每次Ajax调用都牵扯到许多方面，要考虑的问题甚多。上一章提到的JavaScript库大都包含有Ajax组件。也就是说，JavaScript库已经把最累人的工作给完成了——这也正是JavaScript库现在这么流行的原因。有了广泛的用户基础，不但缺陷会更快被发现，而且许多规划问题已经预先为你考虑周全。我们就来看看许许多多JavaScript库中的几个代表。

5.6.1 Prototype

Prototype库内建了一些很方便的Ajax功能：

```
new Ajax.Request(url, {
  method: 'get',
  onSuccess: function(transport) { }
});
```

Prototype库所选择的接口形式和我们先前设计的Ajax对象十分相似。只不过Prototype库更进一步，把许多功能都自动化了。例如它除了成功和失败两种事件之外，还提供了更多的事件，以下是其中的一些：

- onCreate，发生在对象实例化之后，但在对象的任何方法被使用之前。
- onComplete，由请求完成触发，且发生在其他事件都已触发之后。很适合用来停止动画和装载通知用户当前进度的指示器。
- onException，在请求无法处理时触发。例如响应返回的JSON对象格式不正确，就会触发此事件。
- onFailure，如果调用结束，但没有返回在200到300之间的HTTP状态代码，就触发此事件（类似于我们前面设计的Ajax对象）。
- onSuccess，和我们前面的设计一样，当调用成功完成时触发。
- onXXX，X表示HTTP状态代码。一般不常用，因为通常返回的都是200。

Prototype有很多好功能，比如它能自动处理JSON。如果服务器返回的内容类型为text/javascript或者application/javascript（以及几种变体），它会自动用JSON过滤器处理Ajax对象的responseText属性。你也可以用X-JSON首部来传递数据，但用这种方式传递的数据量有限制（不适合传递大量数据，但倒是适合用来传递JSON格式的错误消息）。

1. Ajax.Updater

Ajax.Updater是Ajax.Request对象的特例。它会自动把responseText插入到指定的HTML元素：

```
new Ajax.Updater(container, url, options)
```

options参数和我们之前的设计是一样的，只不过它支持的选项更多一些。其中最值得注意的是insertion属性。在默认情况下，Updater会替换掉HTML元素中的内容，但如果指定了insertion参数，它会把“替换”变成“增补”。insertion参数接受一个Insertion对象（这也是Prototype的一个类），通过Insertion对象可以指定把新内容增补到元素之前（Insertion.Before）、元素的顶部（Insertion.Top）、底部（Insertion.Bottom），或者元素之后（Insertion.After）。

以一个简单的聊天程序为例，它应该有一个聊天窗口、一个文本输入框，还有一个发送按钮：

```
<div id="chat"></div>
<input type="text" id="msg" value="test">
<input type="button" id="send" value="Send">
```

我们要给发送按钮挂上一个事件监听器，把msg输入框的内容发送到服务器。Ajax调用返回之后，程序会自动把响应内容追加到chat元素的尾部：

```
function sendMessage()
{
  // 用响应内容来更新chat元素
  new Ajax.Updater($('chat'), '/path/to/script', {
    parameters: { text: $('msg') },
    insertion: Insertion.Bottom
  });
}
// 单击send按钮时要运行sendMessage
Event.observe($('send'), 'click', sendMessage);
```

2. Ajax.PeriodicalUpdater

如果想给聊天程序加上轮询服务器的功能，可以用PeriodicalUpdater。从名字就可以推想到它会每隔*X*秒向服务器发出一次调用，并且用响应的内容更新chat元素：

```
new Ajax.PeriodicalUpdater($('chat'), '/path/to/script', {
  frequency: 2, /* 2秒*/
  insertion: Insertion.Bottom
});
```

5.6.2　YUI

在YUI库里，一切都是通过连接管理器处理的。下面是一个例子：

```
var transaction = YAHOO.util.Connect.asyncRequest('GET', sUrl, callback);
```

第一个参数告诉YUI应该用GET还是POST方法进行请求。第二个参数是要请求的URL。第三个参数是一个回调对象。回调对象里包含了成功和失败时的回调函数，还可以传禁区一些调用返回时需要用到的参数。

```
var callback = {
  success: myObject.processRequestSuccess,
  failure: myObject.processRequestFailure,
  argument: [argument1, argument2, argument3],
  scope: myObject
}
```

在这个对象里还可以指定回调函数的作用域。这样当成功和失败事件的回调函数属于一个较大的对象的时候，有利于控制this变量的作用域。

如果请求方式是POST，还可以加上第四个参数，用一个查询字符串来传递数据：

```
var transaction = YAHOO.util.Connect.asyncRequest('POST', sUrl, callback,➥
 'key1=encoded+data&key2=even+more+data');
```

YUI库甚至有一个自动获取表单字段的函数，它会自动把字段添加进请求中（因此无需使用第四个参数）：

```
YAHOO.util.Connect.setForm(formObject);
var conn = YAHOO.util.Connect.asyncRequest('POST', 'http://example.com/', callback);
```

5.6.3 jQuery

jQuery在很大程度上是围绕DOM操作来设计的。它处理Ajax的方式比Prototype的Ajax.Updater更进了一步：

```
$('#myelement').load('/updatestatus');
```

上一章已经讲过，$函数用来获取页面中的元素（上面的代码获取的是一个ID为myelement的元素）。然后它向指定的URL updatestatus发出请求，并用响应的结果替换掉myelement元素中的内容。

如果想用常规的方法进行Ajax调用，可以通过jQuery对象中的ajax()函数：

```
var options = {
    url: 'document.xml',
    type: 'GET',
    dataType: 'xml',
    timeout: 1000,
    error: function(){
        alert('Error loading XML document');
    },
    success: function(xml){
        // 对xml进行操作
    }
}
$.ajax(options);
```

所有选项，包括URL都是通过一个字面量对象参数来设置的。

5.7　小结

本章我们介绍了什么是Ajax，并且比较了它和传统页面调用的差异。介绍了各种数据交换的格式，以及它们各自适合的场景。

本章还逐步讲解了如何自己设计一个Ajax对象，并且演示了如何为各种意外情况规划和扩展Ajax对象。最后，我们概要地介绍了如何利用流行的JavaScript库来替我们完成繁琐的工作。

第6章将探讨视觉效果以及如何将之结合到网站设计中。

第6章 视觉效果

动画、滑动、淡入淡出等视觉效果会使页面更加生动，更有吸引力。虽然视觉效果经常有滥用的危险，但我们还是先了解一下为什么要在页面中加入视觉效果，以及它们能解决什么问题。理解这两点之后，你将建立自己的动画对象来实践目前为止所学到的理论。最后，你将学习使用第4章中介绍的JavaScript库来完成动画效果。

6.1 为什么要使用视觉效果

视觉效果有时候会被认为过于浮夸，华而不实。过度的视觉效果确实如此，但恰如其分的视觉效果是很有帮助的。视觉效果可以提醒用户页面上发生的情况。

从传统来说，在网页上执行的动作都是有着固定的行为反馈的。单击一个连接或者一个表单提交按钮，浏览器的图标就会开始旋转，直到页面刷新完毕。而在Ajax应用中，可能完全不存在页面刷新，这时候就需要用一种方式告诉用户执行了什么动作，或者正在执行什么动作。

例如，Google的邮件、日历等应用会在页面的右上角显示一个装载提示，告诉用户程序正在获取数据，如图6-1所示。

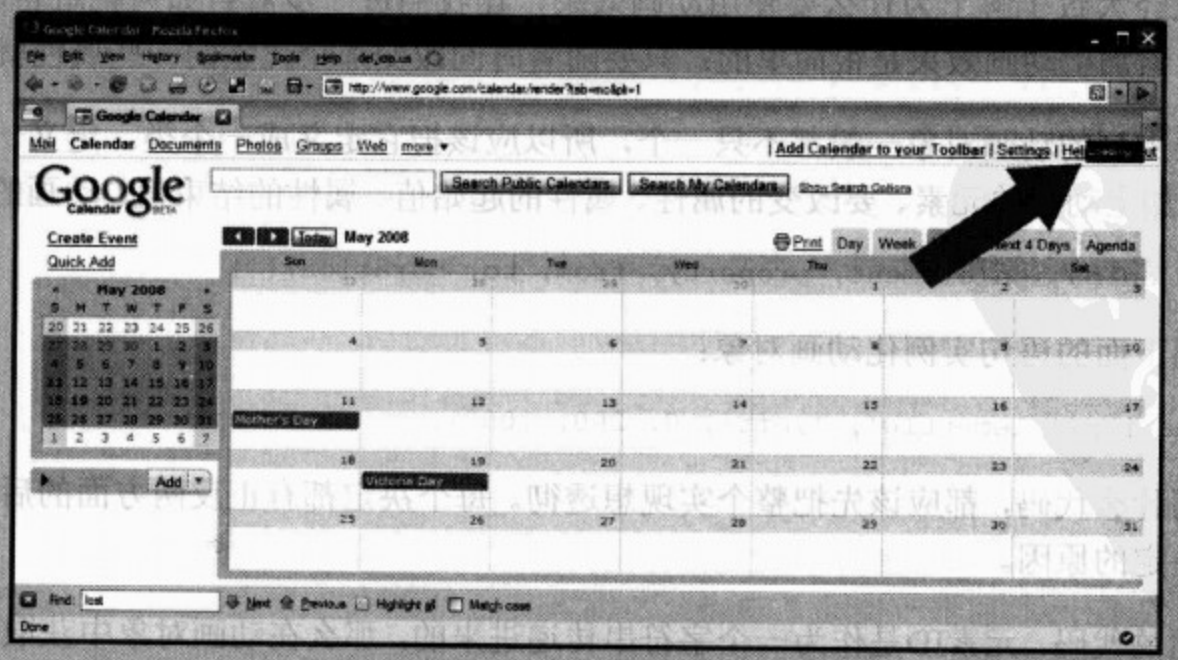


图6-1 Google日历的装载提示

提示和动画告诉用户，程序还在听话地运行着，没有出现什么莫名其妙的错误。

动画还适合用来揭露或者隐藏信息。如果只是简单地改变一些页面元素的可见性，那么用户在不经意间很可能会忽略页面上发生的情况。要是给元素加上动画效果，用户就能马上注意到它，并且把自己的操作和动画效果之间的因果关系联系起来。

动画还可以改善其他方面的体验：

- 加上淡入淡出或者滑入滑出效果的下拉菜单不那么突兀，用户体验要优于突然出现/消失的菜单，特别是当用户在多个导航项目上滑动鼠标的时候。
- 如果指向页内锚点的链接（以#符号开头的链接）能平滑地卷动到目标区域，而不是一下子跳到目标区域，用户体验会比较好。这样用户会清楚地知道自己还在同一个页面上。
- 对于拖放操作，如果能在用户中途放开鼠标的时候，用动画显示被拖动的项目返回原来的位置，用户就能清楚地知道拖放操作没有完成。

必须小心谨慎，不要滥用动画。尤其应该注意保持动画效果简短直接。动画效果的时间过长意味着用户不得不停下来等待动画结束，而不能执行他们真正想要完成的工作。大多数动画都不应该超过半秒钟，务必实际试验一下你的作品，确保不会出现拖沓和繁琐的感觉。

Yahoo!有个设计模式图书馆，里面介绍了Yahoo!大量使用的各种设计模式。其中有很多模式都与动画和过渡有关，不但介绍了模式所解决的问题，还点明了应该避免的误用情形。这个图书馆的地址是：http://developer.yahoo.com/ypatterns/index.php。

6.2　构建一个简单的动画对象

现在你已经大致了解了为什么要使用动画效果，让我们进一步看看如何构造自己的动画对象。给一个元素加上动画效果是很简单的：只要随着时间改变那个元素的一些属性就可以了。

页面里要用到的动画对象一般都不只一个，所以应该把它定义成一个类。定义一个有5个参数的函数：要加上动画的元素、要改变的属性、属性的起始值、属性的结束值、动画的持续时间。

```
function Animation(element, property, from, to, duration){ }
```

接下来用下面的语句实例化动画对象：

```
new Animation('elementID', 'left', 0, 200, 1000);
```

不管编写什么代码，都应该先把整个实现想透彻。每个决定都有正反两方面的后果，你应该想清楚每个决定的原因。

看看前面的代码，元素ID是作为一个字符串传递进来的，那么在动画对象中获取该元素就有两种办法：一是通过DOM方法document.getElementById()；二是采用JavaScript库的调用，如

$()。如果选择某个库的方法，那么这个动画对象就和那个库紧紧绑在一起了。另一种选择是DOM方法，但它写起来有点罗嗦。既然我们想让这个动画对象能适合各种情况，那还是用DOM方法比较好，保证它不依赖于任何库。我们把获取的元素保存在一个变量里，方便后续的代码使用。

```
function Animation(element, property, from, to, duration)
{
  var el = document.getElementById(element);
  if(!el) return false;
}
```

用DOM方法获取元素之后马上执行一次检查，看看元素是否存在，这样可以防止万一元素未定义的情况下用户的浏览器里弹出难看的错误窗口。如何得体地处理错误就留给开发者再去决定了。

要是你打算执行动画的元素没有ID怎么办呢？为了让这个类更灵活，让我们把它改成既可以传入元素引用，又可以传入ID字符串。如果传入的是ID字符串而不是对象引用，程序就会通过DOM方法去获取元素。两全其美！

```
function Animation(element, property, from, to, duration)
{
  var el = element;
  if(typeof el == 'string') el = document.getElementById(element);
  if(!el) return false;
}
new Animation(document.getElementById('elementID'), 'left', 0, 200, 1000);
```

实例化一个对象要用到5个参数，但一眼看上去很难看出参数里的那些数字都代表什么含义。必须翻查类的定义才能了解参数的含义。为此你可以把参数改成字面量对象，键的名称看起来更直观一些。而且如此改动能使API更加灵活，因为即使今后增加了更多的参数，实例化对象的代码也不会变得更复杂。那我们现在就把动画类改成只接受一个options参数吧，元素ID可以从options对象中取得：

```
function Animation(options)
{
  var el = options.element;
  if(typeof el == 'string') el = document.getElementById(options.element);
  if(!el) return false;
}
```

现在实例化动画对象的时候，给它传递一个options对象就行了：

```
var options = {
      element:document.getElementById('elementID'),
      property: 'left',
      from: 0,
      to: 200,
```

```
        duration: 1000
    };
new Animation(options);
```

下一步该执行动画了。但不幸我们没法直接告诉文档哪个元素要“动起来”。实际的做法就像传统的动画片拍摄一样——过一小段时间把元素的位置移动一点，在短时间内把它移动好几次就可以创造出元素在运动的假象，如图6-2所示：

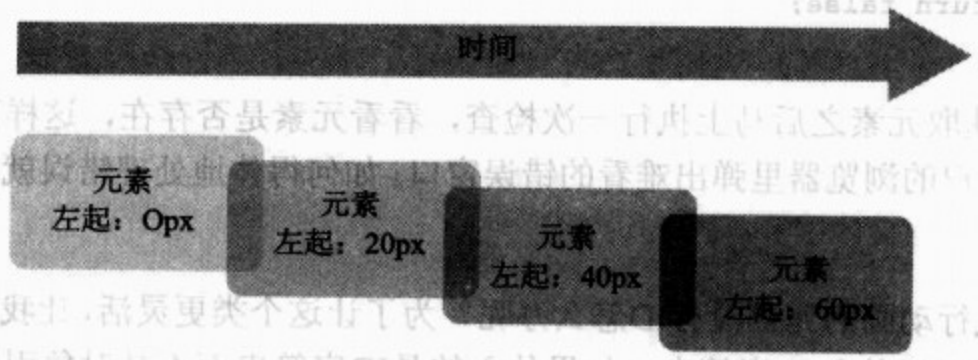


图6-2　随着时间改变属性值

为此要用到setInterval()或者setTimeout()。两者有什么不同呢？它们都有两个参数：第一个参数是要执行的代码，第二个参数是代码执行之前等待的时间，以毫秒为单位。两者都返回一个ID，可以用来随时取消操作：

```
var intervalID = setInterval(performAnimation, 1000); // 每1000毫秒调用一次函数
var timeoutID = setTimeout(performAnimation, 1000); // 1000毫秒之后调用函数
```

你也可以传入一个待求值的字符串，用来传递参数很方便（不过我还是宁可用前一种写法）：

```
var intervalID = setInterval("performAnimation("+id+")", 1000);
var timeoutID = setTimeout("performAnimation("+id+")", 1000);
```

setInterval()和setTimeout()的差别在于，setTimeout()只会执行一次代码，而setInterval()会每隔一段时间（你设定的执行间隔）执行一次代码，直到调用被取消。

要想让setInterval()停下来，只需要调用clearInterval()函数，参数是先前调用setInterval()时返回的ID：

```
clearInterval(intervalID);
```

类似的，要想让setTimeout()停下来，应该调用clearTimeout()，参数是先前调用setTimeout()时返回的ID。取消一个已经执行过的setTimeout()操作，或者ID参数不正确，不会发生任何事情，因此不必担心出现什么JavaScript错误。

```
clearTimeout(timeoutID);
```

现在有两种办法实现定时序列，可以考虑怎样完成动画了。一种是像电影一样每秒钟固定执行那么多次。一般能消除动画停滞感的帧频［以fps（帧每秒）为单位］是24fps，取整到30fps也

很常见。给setInterval()设定帧频轻而易举：

```
var intervalID = setInterval(performAnimation, 33);
```

第一个参数是要调用的函数，第二个参数是33毫秒（1000毫秒除以30fps的结果取整）。

另一种计算帧频的方式是看属性的变化幅度，确定在给定的时间内它要多少步［即多少个px（像素）］才能移动到目的地。例如，属性值原来是50px，现在要在3秒内向左移动到200px，那么200减去50再除以3秒，结果是每秒50个像素。1000毫秒除以50步，结果是每20毫秒移动一个像素：

```
var intervalID = setInterval(performAnimation, 20);
```

对于比较小的步数这个算法是合适的，但如果要在1秒内把对象从0移到1000，那岂不是有1000步？实际上只需要30就足够得到平滑的动画了。因此有必要把第一种计算方式也结合进来：

```
function Animation(options)
{
  var el = options.element;
  if(typeof el == 'string') el = document.getElementById(options.element);
  if(!el) return false;
  var fps = 30;
  function animate()
  {
  }
  var intervalID = setInterval(animate, 1000 / fps);
}
```

现在有了一个定时器，而且会以30fps的频率执行。下一步是确定在给定的帧频和持续时间下，对象的动画到底要多少步才能完成。算出来之后，只要每次执行animate()函数的时候递增步数就可以了。一旦达到规定的步数，就把定时操作清除掉——动画就结束了。这样一个基本的动画类就完成了：

```
function Animation(options)
{
  var el = options.element;
  if(typeof el == 'string') el = document.getElementById(options.element);
  if(!el) return false;
  var fps = 30;
  // 存储我们现在执行的步
  var step = 0;
  // 确定总的步数
  var numsteps = options.duration / 1000 * fps;
  // 确定步之间的间隔
  var interval = (options.from - options.to) / numsteps;

  function animate()
```

```
    {
      // 新的位置将会是什么
      var newval = options.from - (step * interval);
      // 比较之后步将递增
      if(step++ < numsteps) {
        // 使用Math.ceil成为整数和类型
        el.style[options.property] = Math.ceil(newval) + 'px';
      }else{
        // 将元素设置为它最终的点
        el.style[options.property] = options.to + 'px';
        // 清除interval。intervalID可通过闭包访问
        clearInterval(intervalID);
      }
    }
    var intervalID = setInterval(animate, 1000 / fps);
}
```

现在动画对象有了修改DOM的能力。但如果你想对动画的过程拥有更强的控制能力（例如决定何时启动、停止、重置动画）又如何呢？让我们进一步扩展动画类，给它增加一些新的方法。比如完成启动和停止功能的start()和stop()方法，还有后退到动画开头的gotoStart()方法，以及快进到动画结尾的gotoEnd()方法：

有两项修改是最关键的。首先是不让setInterval()在动画对象实例化的时候自动执行，而是只有当你执行start()方法的时候才开始。其次是提供一个公共的API，让实例化的结果作为一个对象返回（这是第3章所讨论的封装概念的绝佳例子）：

```
function Animation(options)
{
  var el = options.element;
  if(typeof el == 'string') el = document.getElementById(options.element);
  if(!el) return false;
  var fps = 30;
  // 存储我们正在执行的步
  var step = 0;
  // 确定总的步数
  var numsteps = options.duration / 1000 * fps;
  // 确定步之间的间隔
  var interval = (options.from - options.to) / numsteps;
  var intervalID;

  function animate()
  {
    // 新的位置将是什么
    var newval = options.from - (step * interval);
    // 比较之后步会递增
```

```
    if(step++ < numsteps) {
      // 使用Math.ceil成为整数和类型
      el.style[options.property] = Math.ceil(newval) + 'px';
    }else{
      el.style[options.property] = options.to + 'px';
      publicMethods.stop();
    }
  }

  var publicMethods = {
    start:function(){
      intervalID = setInterval(animate, 1000 / fps);
    },
    stop:function(){
      clearInterval(intervalID);
    },
    gotoStart:function(){
      step = 0;
      el.style[options.property] = options.from + 'px';
    },
    gotoEnd:function(){
      step = numsteps;
      el.style[options.property] = options.to + 'px';
    }
    return publicMethods;
  }
```

intervalID变量被移到了最外层，和其他变量呆在一起。不把它放在start()函数内部是因为stop()函数也要访问它，放在start()函数内部就要用到闭包才行了。另外要注意animate()函数现在不再直接清除定时操作，而是调用stop()方法来间接完成。这样所有与“停止”有关的逻辑都在同一个地方，方便以后扩展API。

那么怎样扩展API呢？接着读下面的内容。

6.2.1 回调

现在执行动画的时候可以不用干预它的细节了。我们来看看怎样在特定的时刻触发自定义的事件，让其他代码可以绑定到动画上，以执行一些相关的任务。对于所有的动画，我们一般只关心3种时刻：

- **动画开始**。与动画开始有关系的任务自然应该绑定到动画开始的时刻。比如你可能想在动画开始的时候改变一个字符标签显示的内容。
- **动画的每一步**。一般不太会对动画的每一步感兴趣，因为它实在发生得太频繁了。不过

如果你正在制作一个游戏，那么知道页面上的两个元素发生了交错还是很有意义的。（是的，用JavaScript也可以写游戏。）

- **动画结束**。这个时刻很适合进行移除页面上的元素、添加新元素、执行Ajax调用之类的操作。

从该API可以看出，animate()函数知道动画什么时候开始、什么时候结束，什么时候执行动画的每一步，显然应该从这里添加代码。为什么不把开始和停止的回调放在start()和stop()方法里面呢？因为动画可能在中途停止和重新开始，把代码放在这两个方法里不能正确判断那样的情况。给API中的操作加上回调是没必要的，因为它们都是你主动调用的，要同时执行什么任务，直接执行就好了。你没办法知道的只是动画什么时候开始、什么时候结束以及什么时候进入下一步——回调也不知道。

让我们动手吧。下面只列出了animate()函数（因为我们只需要修改这个函数）：

```
function animate()
{
  // 新的位置将会是什么
  var newval = options.from - (step * interval);
  // 比较之后步会递增
  // 检查属性是否存在，步是否为0（即第1步）
  if(options.onStart && step == 0) options.onStart();
  if(options.onStep) options.onStep();
    if(step++ <= numsteps) {
      // 使用Math.ceil成为整数和类型
      el.style[options.property] = Math.ceil(newval) + 'px';
    }else{
      el.style[options.property] = options.to + 'px';
      if(options.onEnd) options.onEnd();
      publicMethods.stop();
    }
}
```

options对象现在多了一些额外的属性：

```
var options = {
    element:document.getElementById('elementID'),
    property:'height',
    from: 0,
    to: 200,
    duration: 1000,
    onStart: function(){ console.log('started') },
    onStep: function(){ console.log('stepped') },
    onEnd: function(){ console.log('ended') }
};
```

请记住这些函数中的console.log()只是用来跟踪什么时候发生了调用。第1章已经说明，console.log()这种调试技术在IE和Opera中都不可用。

6.2.2 动画队列

通过动画队列可以建立一个事件的序列。利用前面介绍的动画对象的回调功能，可以编写出触发多个动画的脚本。

假设有3个并排的元素，你打算让它们依次出现。为此只要把第一个对象的onEnd回调设置成启动第二个对象的动画，然后把第二个对象的onEnd回调设置成启动第三个对象的动画。这样一来，3个动画就会一个接一个地执行了：

```
var options1 = {
    element:document.getElementById('element1'),
    property:'height',
    from: 0,
    to: 200,
    duration: 1000,
    onEnd: function(){ a2.start(); }
  };
var a1 = new Animation( options1 );

var options2 = {
    element: document.getElementById('element2'),
    property:'height',
    from: 0,
    to: 200,
    duration: 1000,
    onEnd: function(){ a3.start(); }
  };
var a2 = new Animation( options2 );

var options3 = {
    element: document.getElementById('element3'),
    property:'height',
    from: 0,
    to: 200,
    duration: 1000
  };
var a3 = new Animation( options3 );

// 启动所有动画
a1.start();
```

这段代码会创造出图6-3所示的动画序列。

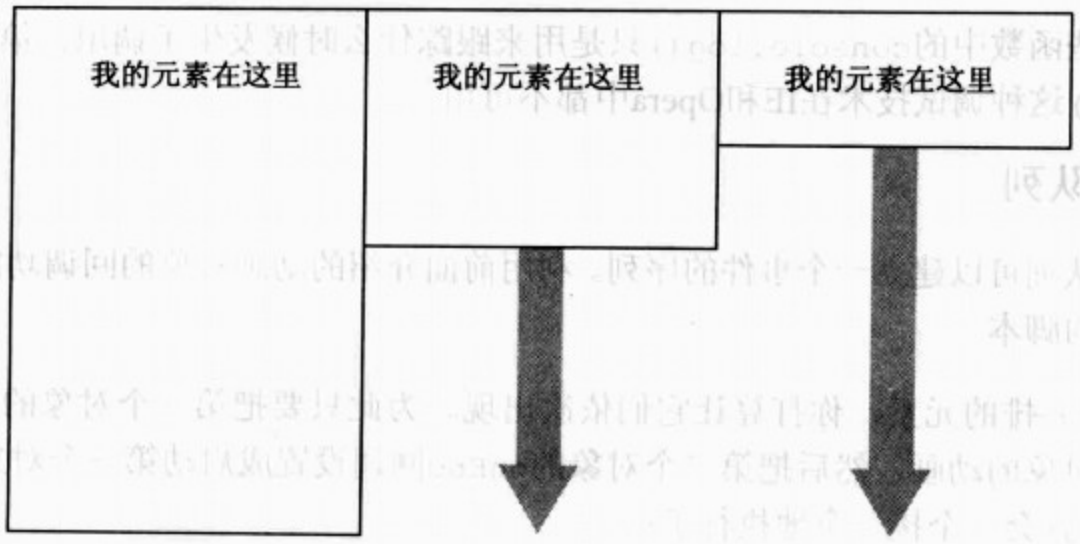


图6-3　页面元素的动画过程

6.3　扩展动画类

有了基本的动画对象作为基础，你可以针对特殊的需要打造出特别的类。假设你有一个FAQ（frequently asked question，常见问答）页面，页面上的内容按照问题→答案→问题→答案的顺序依次排列下来。

前面建立的动画对象不能完全满足需要。首先，你需要记住对象原先的高度，以便把答案隐藏起来之后能恢复到原来的大小。另外，对象有两种状态（展开或关闭）你要记住当前的状态，以便实现状态切换。

把新的类命名为Toggler，它只有一个参数，也就是控制开、闭的元素：

```
function Toggler(element){ }
```

页面上的HTML大致如下。每个问题元素的都有个类名question。例中的答案元素也有类名answer，不过在这个例子里不会用到：

```
<div class="question">Question 1</div>
<div class="answer">Lengthy description ... </div>
<div class="question">Question 2</div>
<div class="answer">Lengthy description ... </div>
<div class="question">Question 3</div>
<div class="answer">Lengthy description ... </div>
<div class="question">Question 4</div>
<div class="answer">Lengthy description ... </div>
```

当窗口加载之后，要先获取所有的question元素，分别为它们创建新的Toggler对象。这里用到了第2章介绍的getElementsByClassName()函数：

```
var els = getElementsByClassName(document, 'question');
```

```
for(var i=0;i<els.length;i++)
{
  new Toggler(els[i]);
}
```

现在有了一堆Toggler对象，我们还需要丰富一下Toggler类——要根据选中的问题找到相应的答案。在这个例子里，答案总是紧跟在问题后面的，因此用nextSibling这个DOM属性就可以得到正确的答案。你可能还记得，IE不计算两个结点之间的空白字符结点。因此为了确保所有的浏览器都能找到正确的答案，应该检查nextSibling返回的是不是一个元素，如果不是，再取下一个元素。这里还要把答案元素的初始高度保存起来。

```
function Toggler(element){
  var answer = element.nextSibling;
  if(answer.nodeType !=1) answer = answer.nextSibling;
  var startHeight = answer.offsetHeight;
  var hidden = false;
}
```

下一步，加上实际执行切换动作的代码。它会初始化一个新的动画对象，并且每次执行的时候交换传给动画对象的to和from选项，以达到控制动画移动方向的目的：

```
function Toggler(element){
  var answer = element.nextSibling;
  if(answer.nodeType !=1) answer = answer.nextSibling;
  var startHeight = answer.offsetHeight;
  var hidden = false;

function toggle()
{
  var start, stop;
  if(hidden)
  {
    start = 0;
    stop = startHeight;
  }else{
    start = startHeight;
    stop = 0;
  }

  var options = {
    element: answer,
    from:start,
    to:stop,
    duration:250,
    property:'height'
  };

  // 初始化并启动动画
```

```
      (new Animation(options)).start()
      // 切换隐藏的属性
      hidden = hidden ? false : true;
    }
  }
```

定义好toggle函数之后，还有最后的两块拼图（附加到事件处理程序并把答案的初始状态设为隐藏）：

```
function Toggler(element){
  var answer = element.nextSibling;
  if(answer.nodeType !=1) answer = answer.nextSibling;
  var startHeight = answer.offsetHeight;
  var hidden = false;

  function toggle()
  {
    var start, stop;
    if(hidden)
    {
      start = 0;
      stop = startHeight;
    }else{
      start = startHeight;
      stop = 0;
    }

    var options = {
      element: answer,
      from:start,
      to:stop,
      duration:250,
      property:'height'
    };
    // 初始化并启动动画
     (new Animation(options)).start()
    // 切换隐藏的属性
    hidden = hidden ? false : true;
  }

  element.onclick = toggle;
  toggle();
}
```

加上一小段CSS：

```
.question {
  font-weight:bold;
  margin-top:10px;
```

```
  cursor:pointer; /* 使用相同的指针作为链接 */
}

.answer {
  /* 必须由隐藏的overflow来做动画 */
  overflow:hidden;
}
```

就这样，处理FAQ问题的方法就有了，如图6-4所示。

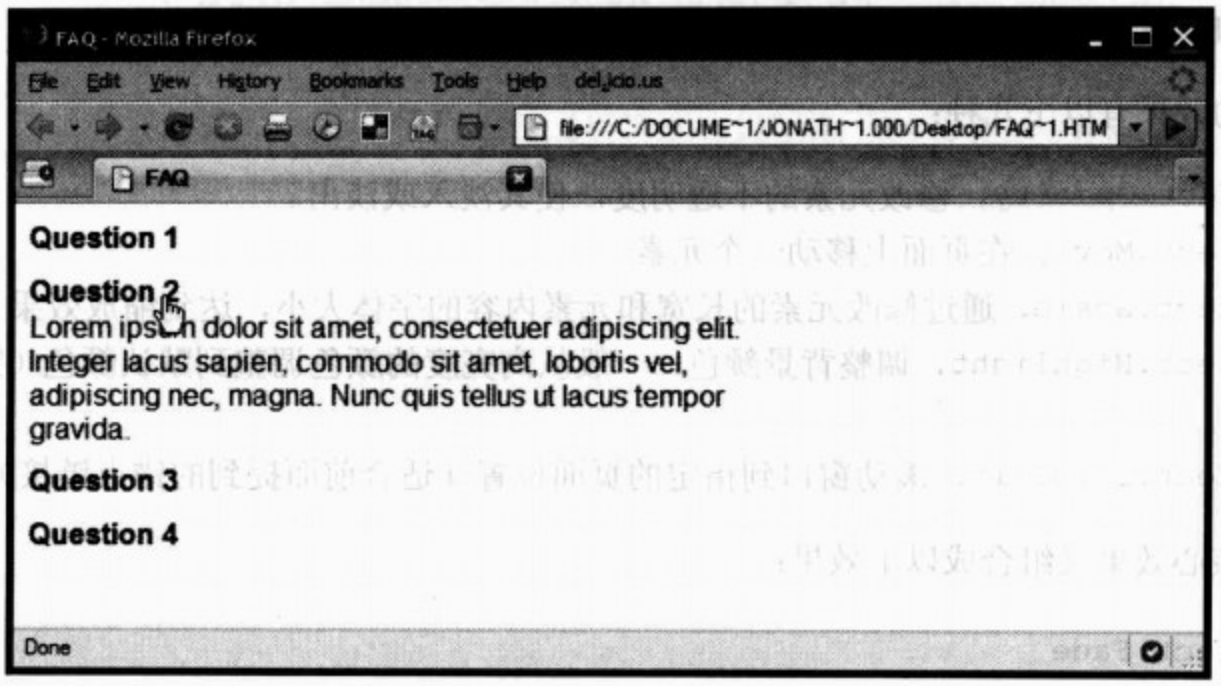


图6-4 展开了一项的FAQ

注解 第8章有一个FAQ的案例研究，会介绍更多FAQ的原理。

6.4 用库来实现动画

虽然我们已经自行构造了一个相当不错的动画类，但还有不少方面没有考虑到，这也就是库的存在意义：它们已经解决了很多我们没想到的问题。

可以举对象的不透明度为例。不透明度不像左边距、上边距之类的属性以像素为单位，它的取值范围从0到1。因此要特殊对待，根据属性值的情况分别处理。

现在我们介绍以下库中所提供的各种动画功能：

- Script.aculo.us
- jQuery
- Mootools

6.4.1 Script.aculo.us

Script.aculo.us扩展了Prototype，通过它的核心效果组件和效果组合组件提供了多种多样的动画功能。

就像我们先前开发的动画类一样，Script.aculo.us也有一个基本的效果类，叫做Effect.Base。其他效果类都是从它派生出来的。和我们的实现不同，Effect.Base类并不执行实际的属性修改，而是交给子类去完成。基类负责时间控制，让子类去做各自最擅长的事情，以及处理各种特殊的情况（比如不透明度的特殊取值范围）。

核心的效果有以下几种：

- Effect.Opacity，修改元素的不透明度，使其淡入或淡出。
- Effect.Move，在页面上移动一个元素。
- Effect.Scale，通过修改元素的长宽和元素内容的字体大小，达到缩放效果。
- Effect.Highlight，调整背景颜色，一般从高亮度的颜色调整到默认颜色（突显状态改变）。
- Effect.ScrollTo，滚动窗口到指定的页面位置（适合前面提到的锚点链接）。

上述核心效果又组合成以下效果：

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

实例化一个效果的时候要向构造函数传递一个元素或者元素ID作为参数：

```
new Effect.Puff('elementID');
```

所有的效果类第二个参数都是一个options对象，不过选项的内容各有分别。

Script.aculo.us网站上有所有效果的演示（http://wiki.script.aculo.us/scriptaculous/show/CombinationEffectsDemo），你可以看到实际的例子（见图6-5）。

图6-5 抽烟（Puff）效果，它先变淡然后“喷出来”就像抽烟一样

如果你已经在项目中用了Prototype，并且Prototype和Script.aculo.us库都已经默认包括在内（比如Ruby on Rails项目），添加Script.aculo.us效果是极其简单的。

6.4.2 jQuery

这个极其精简的库（只有20KB）无需额外费力就可以完成许多简单的动画。由于其结构，jQuery非常适合用来添加一些简单的动画。随便把一个元素交给任何动画对象就可以了。标准的jQuery动画有以下几种：

- fadeIn()、fadeout()和fadeTo()，这几个动画效果可以让对象淡入、淡出或者从当前值渐变到指定的值。

- slideDown()、slideUp()和slideToggle()，这几个动画和我们前面完成的常见问答页面的效果一样，即展开和隐藏页面的某一部分。slideToggle()可让页面元素在slideDown()和slideUp()之间进行切换。
- show()、hide()和toggle()，show()和hide()将元素淡入淡出并改变其大小，toggle()在两者之间切换。

这些效果都很容易使用，而且可以在'slow'、'normal'、'fast'3种速度之间调整，也可以指定动画持续的毫秒数。你还可以加上第二个参数，也就是在动画完成时将要执行的回调。

```
$('#elementID').fadeOut('fast', function(){
  alert("I'm done the animation")
});
```

jQuery还提供了animate()方法让你同时改变多个属性。要改变的属性被放在一个字面量对象中，作为第一个参数传给animate()方法。第二个参数是速度，第三个是缓冲效果（easing），最后一个是回调函数。只有第一个参数是必需的，其他都是可选的。jQuery还（成功地）尝试了让可选参数可以以任何顺序出现。

```
$("#elementID").animate(
  { height: 'toggle', opacity: 'toggle' },
  "fast",
  "easein",
  function(){alert('done!');}
);
```

注解　缓冲效果是通过数学计算来让动画的速度随着时间变化。动画可以一开始很慢，然后越来越快（或者相反），比起速度一成不变的动画感觉起来更加自然。

缓冲效果是通过一个jQuery插件处理的，这个额外功能也被放在jQuery命名空间里。请到http://gsgd.co.uk/sandbox/jquery.easing.php下载jQuery Easing Plugin。常用的缓冲效果有easein和easeout，更花哨的效果还有bouncein/bounceout和backin/backout。

关于jQuery插件的更多信息请参见http://docs.jquery.com/Plugins。

6.4.3　Mootools

Mootools是由Moo.fx的作者制作的，它流行的重要原因是它非常小（3KB），随带了一个精简版的Prototype，叫做Prototype Lite（5KB）。它的功能严格限制在动画方面。在写作本书的时候，开发人员去掉了Prototype的要求，创建了各种有用的组件。效果被分成几个组件（与Script.aculo.us相似）。

Fx.Base类似于Script.aculo.us，其他类都从它扩展而来。在众多Fx类中，只有3个是专门提

供动画功能的：

- Fx.Style，随着时间变化修改元素的样式属性。
- Fx.Scroll，滚动窗口或者设置了overflow:scroll属性的元素。
- Fx.Slide，通过滑动的动画效果显示或隐藏页面的内容。

其他Fx类都是处理各种动画任务的工具类，比如过渡（Fx.Transitions），或同时给多个元素设置多个样式效果（Fx.Elements）。按照在Style对象的设计，只有调用start()方法之后动画才会启动。

```
var anim = new Fx.Style('elementID', left',{duration:500});
anim.start(0, 100);
```

6.5 小结

视觉效果在你的作品中会占有一席之地，因为它能解决网页与生俱来的可用性问题。本章讲解了如何构建你自己的动画对象，并且在讲解过程中实践了前几章讨论的各种概念。最后，还讲解了一些流行的JavaScript库是如何处理动画的，以及它们所提供的额外功能。

在第7章中，通过Stuart Langridge你可以了解DOM脚本处理的一个最主要用途：表单验证。

第7章 表单验证与JavaScript

本章作者：Stuart Langridge

大多数Web应用都免不了用到<form>标签。如果通过<form>来收集信息，你可能会希望验证一下用户提交的信息，以确保内容是合理的。字段旁边加上“*”表示必填项目的管理也是由此而来。还有一点必须重视的是，如果你希望得到的数据以某种格式出现，那么就应该在程序里迫使用户按规定的格式填写。这就是表单验证的意义所在：让程序检查用户的输入，确保它们是你想要的信息。如果你要问用户家里有几只鸡，那么就应该检查确认用户输入的是一个数字，而不是“我家没有鸡”之类的回答。

验证可以提高数据的质量，它还意味着后续的程序可以假设得到的数据是有效的。例如，如果你把一只鸡卖给用户，那么你就可以写成“你想不想要（$chickens + ）只鸡？”，而不必担心把用户的输入内容加1会有什么奇怪的结果。验证是很重要的。

这是一本关于DOM脚本处理的书，因此你可能想直奔JavaScript的主题，但事情并非如此。对于Web验证有一点非常重要，那就是你不能只依靠JavaScript来做这件事情。用户可能关闭JavaScript，也可能正在用手机上网，他们可能正在火车上通过黑莓手机向邻座展示你那漂亮的chickenbuyer.example.com网站。在服务器端和客户端都要进行验证。因此我们先简要看一下服务器端的验证，再看看如何以及为什么把它扩展到JavaScript。

7.1 在服务器上验证

检查用户提交的表单要用到正则表达式。代码清单7-1的例子是用PHP编写的，不过现在所有语言都支持正则表达式，因此你可以随意选择喜欢的服务器端技术。如果你不熟悉正则表达式，在网上可以找到丰富的资料。http://www.regular-expressions.info/是很好的入门教程，详细适中。正式的书籍有Jeffrey Friedl所著的*Mastering Regular Expressions*，在http://regex.info/网站上有介绍。

代码清单7-1 simple-form.php，简单的PHP表单验证

```
<?php
```

```
$VALIDATIONS = Array(
  "firstname" => Array("regexp" => '.+', "error" => "Enter a name"),
  "lastname" => Array("regexp" => '.+', "error" => "Enter a name"),
  "heads" => Array("regexp" => '^\d+$', "error" => "Number of heads ➥
 should be a whole number"),
  "dob" => Array("regexp" => '^\d\d[\/.-]\d\d[\/.-]\d\d\d\d$', "error" => ➥
"Enter dates in format DD/MM/YYYY"),
  "email" => Array("regexp" => '^.+@.+\..+$', "error" => ➥
"This address is not valid")
);

$ERRORS = Array();

if (isset($_GET["submit"])) {
  # form was submitted
  foreach ($VALIDATIONS as $field => $data) {
    if (!isset($_GET[$field])) continue; # skip any that aren't sent

    $regexpstr = $data["regexp"];

    if (preg_match("/$regexpstr/", $_GET[$field]) == 0) {
      $ERRORS[$field] = $data["error"];
    }
  }

  if (count($ERRORS) == 0) echo "Data OK; now redirect!";
}
?>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>

<title>A simple PHP form using regular expressions for validation</title>
<link rel="stylesheet" href="styles.css">
</head>

<body>
<h1>A simple PHP form using regular expressions for validation</h1>
<form>
 <p><label for="firstname">First name</label>
    <input type="text" id="firstname" name="firstname">
    <span class="error">
    <?php if (array_key_exists("firstname",$ERRORS)) echo $ERRORS["firstname"]; ?>
  </span>
 </p>

 <p><label for="lastname">Last name</label>
    <input type="text" id="lastname" name="lastname">
    <span class="error">
      <?php if (array_key_exists("lastname",$ERRORS)) echo $ERRORS["lastname"]; ?>
```

```
    </span>
  </p>

  <p><label for="heads">Number of heads</label>
    <input type="text" id="heads" name="heads">
    <span class="error">
      <?php if (array_key_exists("heads",$ERRORS)) echo $ERRORS["heads"]; ?>
    </span>
  </p>

  <p><label for="dob">Date of birth (DD/MM/YYYY)</label>
    <input type="text" id="dob" name="dob">
    <span class="error">
      <?php if (array_key_exists("dob",$ERRORS)) echo $ERRORS["dob"]; ?>
    </span>
  </p>

  <p><label for="email">Email address of someone you don't like for
    spamming purposes</label>
    <input type="text" id="email" name="email">
    <span class="error">
      <?php if (array_key_exists("email",$ERRORS)) echo $ERRORS["email"]; ?>
    </span>
  </p>

  <p><input type="submit" name="submit" value="Send answers"></p>
 </form>
 </body>
 </html>
```

simple-form.php是个简单的例子，它演示了如何用PHP在服务器端执行正则表达式验证。每个字段都指定了一条正则表达式，如果用户的输入不合要求，就会显示一条错误消息。例如“number of heads”字段必须是数字，因此它的正则表达式是^\d+$。（注意+的意思是“一个或更多”，因此这个字段是必须填写的。）如果是可选字段（其内容可以是空白），那么相应的正则表达式就应该是^\d*$，因为*的意思是“零个或更多”。

注意，例子中有些正则表达式是过于简化的。正则表达式是优秀的工具，但并不完美。例如，检查E-mail的表达式^.+@.+\..+$允许像stuart@somewhere@somewhere@somewhere.com这样的嵌套。很多时候你都不能完全依赖正则表达式去保证正确性。http://www.ex-parrot.com/~pdw/Mail-RFC822-Address.html上有一个“正确”检查E-mail地址的正则表达式，长达6251个字符。在实践中，你一般只需要通过验证来剔除一些明显的错误，而没必要考虑所有的可能。

7.2 客户端

回到JavaScript。JavaScript原生地支持正则表达式。字符串有个search()方法可以接受正则

表达式作为参数，如果找到匹配的字符，它就会返回第一个匹配的字符的位置，如果没找到就返回-1。于是我们可以用JavaScript完成以下工作：

(1) 定义一个正则表达式及其对应的字段的列表。

(2) 页面加载时，给每个具备相应正则表达式的字段绑定onBlur事件处理函数。

(3) 当输入焦点离开字段时，onBlur处理函数得到执行，它将用相应的正则表达式检查用户刚才的输入。

(4) 如果正则表达式与用户输入不匹配，则显示一条错误消息，让用户知道自己输入错了。

注意JavaScript验证代码是在页面装载的时候绑定到字段上的，这符合DOM脚本编写中不唐突和渐进增强的原则。如果用户的浏览器不支持JavaScript，那么绑定根本就不会发生。由于你已经在服务器端执行了验证，所以数据的有效性仍然受到保障，页面也不会被破坏。而且，并没有太多的JavaScript代码和HTML标记混杂在一起，脚本都包含在页面首部的<script>标签里面。

PHP代码中已经有字段和正则表达式的列表了，那么最好是让PHP动态地把列表输出为JavaScript能理解的形式（免得要输入两次）。JavaScript本身支持正则表达式，在JavaScript中，定义正则表达式要在两端加上斜杠（如/^hello$/）。因此输出的JavaScript结构应该像下面的样子：

```
VALIDATIONS = {
  "firstname": { 'regexp': /.+/, 'error': 'Enter a name' },
  "lastname": { 'regexp': /.+/, 'error': 'Enter a name' },
  "heads": { 'regexp': /^\d+$/,
      'error': 'Number of heads should be a whole number' }
};
```

可以通过服务器端的PHP代码直接把上述结构写到<script>标签之中，省去定义两次列表的麻烦。代码清单7-2正是如此，PHP代码遍历$VALIDATIONSPHP数组，生成等价的JavaScript关联数组。

下一步是遍历验证项目的列表，找到每个关联的字段，给它们逐一加上onBlur事件处理函数：

```
for (fieldname in VALIDATIONS) {
  fld = document.getElementById(fieldname);
  if (!fld) continue; // 如果该字段在页面中不存在，则忽略
  addEvent(fld, "blur", checkField);
}
```

VALIDATIONS是一个关联数组（有时候也称为散列表或者字典），也就是说你可以用for (key in VALIDATIONS)遍历所有的键。数组中的键就是我们所关心的字段的名称，那么我们就就用这个键作为元素的ID在页面中查找元素（如果找不到就直接退出），然后针对找到的元素，把checkField()函数设置为它的blur事件的处理函数。

checkField()函数实现需求中的第3和第4步——用相应的正则表达式检查用户在字段中的输入，如果不匹配就显示错误消息。checkField()函数代码如下：

```
function checkField(e) {

  fld = window.event ? window.event.srcElement : e.target;

  fieldname = fld.id;

  if (VALIDATIONS[fieldname]) {
    re = VALIDATIONS[fieldname]["regexp"];

    if (fld.value.search(re) == -1) {
      // 正则表达式不匹配
      // 找到该字段的span.error元素，将错误消息放在里边

      span = fld.parentNode.getElementsByTagName('span')[0];
      span.innerHTML = VALIDATIONS[fieldname]["error"];
    } else {
      // 正则表达式*完全*匹配
      // 删除错误消息!

      span = fld.parentNode.getElementsByTagName('span')[0];
      span.innerHTML = "";
    }
  }
```

首先，因为它是一个事件处理函数，所以你先要取得触发事件的元素——即文本输入框本身。此处用到一个小小的跨浏览器编码技巧：在提供了window.event对象的浏览器（如IE）中，通过window.event.srcElement获得该元素的引用，其他浏览器则通过W3C定义的event对象的target属性获得。

接下来查找VALIDATIONS对象，看看有没有与该元素相对应的正则表达式，方法是检查VALIDATIONS[fieldname]是否存在。如果存在，那么从VALIDATIONS数组取出我们所要的那条正则表达式，准备下一步验证。

文本字段的当前值是一个字符串，保存在fieldobject.value属性当中。前面已经提到过，字符串有一个search()方法可以检查字符串与指定的正则表达式是否匹配，不匹配则返回-1。那么接下来的代码意思就是说，“如果字段中的当前值与正则表达式不匹配”，就显示错误消息。

```
if (fld.value.search(re) == -1)
```

页面已构建，包含该字段的部分是这样的：

```
<p><label for="dob">Date of birth (DD/MM/YYYY)</label>
```

```
    <input type="text" id="dob" name="dob">
    <span class="error"></span>
  </p>
```

因此每个字段都有一个关联的span标签用于显示错误消息。在这段HTML对应的DOM树结构里，<label>、<input>和都是<p>元素的子元素。因此获取元素的最佳方式是：

```
fld.parentNode.getElementsByTagName('span')[0]
```

取得元素之后，把相应的错误消息写到元素的innerHTML属性就可以了。反过来，如果if语句执行的正则表达式检查结果不是-1，也就是说字段值是正确的，那么就应该把元素innerHTML属性设成空白。

接下来再添加一个跨浏览器的addEvent()函数并把所有JavaScript代码块都打包进一个简单的对象（以防止函数、变量的名称和你加载的其他脚本出现意外冲突）。PHP代码也可以稍作整理（输出一个带有函数的字段，而不是直接将字段内联在页面中），最后得到代码清单7-2。

代码清单7-2 simple-form-tidier-js.php，在simple-form.php的基础上添加了JavaScript正则表达式验证

simple-form.php

```
<?php
$VALIDATIONS = Array(
  "firstname" => Array("regexp" => '.+', "error" => "Enter a name"),
  "lastname" => Array("regexp" => '.+', "error" => "Enter a name"),
  "heads" => Array("regexp" => '^\d+$',
                   "error" => "Number of heads should be a whole number"),
    "dob" => Array("regexp" => '^\d\d[\/.-]\d\d[\/.-]\d\d\d\d$',
                   "error" => "Enter dates in format DD/MM/YYYY"),
    "email" => Array("regexp" => '^.+@.+\..+$',
                     "error" => "This address is not valid")
);
  $ERRORS = Array();

  if (isset($_GET["submit"])) {
    # form was submitted

    foreach ($VALIDATIONS as $field => $data) {
      if (!isset($_GET[$field])) continue; # skip any that aren't sent

      $regexpstr = $data["regexp"];

      if (preg_match("/$regexpstr/", $_GET[$field]) == 0) {
```

```
            $ERRORS[$field] = $data["error"];
          }

      }

      if (count($ERRORS) == 0) echo "Data OK; now redirect!";
  }

  ?>

  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
  "http://www.w3.org/TR/html4/strict.dtd">

  <html>
  <head>

  <title>A simple PHP form using regular expressions for validation</title>

  <link rel="stylesheet" href="styles.css">

  <script type="text/javascript">
  validator = {

    VALIDATIONS: {
  <?php

      foreach ($VALIDATIONS as $field => $data) {

        $regexpstr = $data["regexp"];

        $errorstr = $data["error"];
        echo "\"$field\": { 'regexp': /$regexpstr/, 'error': '$errorstr' },\n";
      }
  ?>
    },

    init: function() {
      // 检查该浏览器是否有执行我们需要的DOM脚本化的戳记

      if (!document.getElementById) return;

      // 遍历VALIDATIONS列表，找到所对应的各个字段
      // 它应用并附加了一个onBlur处理函数，所以当用户不再使用字段，它会根据regexp检查字段的内容
      for (fieldname in validator.VALIDATIONS) {
        fld = document.getElementById(fieldname);
```

```
    if (!fld) continue; // 如果该字段在页面中不存在，则将其忽略

    validator.addEvent(fld, "blur", validator.checkField);
  }
},

checkField: function(e) {
  fld = window.event ? window.event.srcElement : e.target;

  fieldname = fld.id;

  if (validator.VALIDATIONS[fieldname]) {
    re = validator.VALIDATIONS[fieldname]["regexp"];

    if (fld.value.search(re) == -1) {
      // 正则表达式不匹配
      // 找到该字段的span.error元素，将错误消息放入其中

      span = fld.parentNode.getElementsByTagName('span')[0];
      span.innerHTML = validator.VALIDATIONS[fieldname]["error"];
    } else {
      // 正则表达式*完全*匹配
      // 删除错误消息!

      span = fld.parentNode.getElementsByTagName('span')[0];
      span.innerHTML = "";
    }
  }
},
  addEvent: function( obj, type, fn ) {
    if (obj.addEventListener) {
      obj.addEventListener( type, fn, false );
    } else if (obj.attachEvent) {
      obj["e"+type+fn] = fn;
      obj[type+fn] = function() { obj["e"+type+fn]( window.event ); }
      obj.attachEvent( "on"+type, obj[type+fn] );
    }
  }
}

validator.addEvent(window, "load", validator.init);

</script>
</head>
<body>
```

```
<h1>A simple PHP form using regular expressions for validation</h1>

<form>

<?php
function field($name, $text) {

  global $ERRORS;

  echo "<p><label for=\"$name\"";

  if (array_key_exists($name,$ERRORS)) {
    echo " class=\"error\"";
  }

  echo ">$text</label>\n";
  echo "<input type=\"text\" id=\"$name\" name=\"$name\">\n";

  if (array_key_exists($name,$ERRORS)) {
    $err = $ERRORS[$name];

    echo "<span class=\"error\">$err</span>";
    } else {
    echo "<span class=\"error\"></span>";
    }

    echo "</p>\n";
}
field("firstname", "First name");
field("lastname", "Last name");
field("heads", "Number of heads");
field("dob", "Date of birth (DD/MM/YYYY)");
field("email", "Email address of someone you don't like for spamming purposes");
?>

  <p><input type="submit" name="submit" value="Send answers"></p>

</form>
</body>
</html>
```

这样我们就一步步运用DOM脚本技术完成了带正则表达式的表单验证。

有了这么一个表单验证的基本框架，你可以在上面增加各种各样的改进。在此我们将介绍其中两种：一是让页面自己增加显示错误信息的span块（而不是像现在这样预先写在HTML里面）；二是当表单中存在错误时阻止表单的提交。

7.2.1 用 JavaScript 添加显示错误消息的 Span 块

在代码清单7-2里，服务器端的PHP代码不管字段的内容是否正确，为每个字段都输出一个<span class="error"x/span>作为错误消息的占位符。当这些span标签内容为空白的时候对页面毫无贡献，而且散落在HTML代码里不是很好看。如果能通过页面中的JavaScript脚本自动添加和删除这些span标签，则可以使页面代码更加整洁。注意客户端的JavaScript检测到表单错误并不意味着一定要插入新的span标签（span标签可能已经存在）因为如果服务器端检测到表单错误时同样会输出错误信息。所以当JavaScript检测到表单错误，首先要检查是否已经存在错误消息span。如果存在就只修改其内容，反之则插入新的标签。

代码清单7-3对代码清单7-2的代码做了小小的改动，修改仅限于checkField()这个JavaScript函数，其他都原封不动。

代码清单7-3 simple-form-tidier-js-create-spans.php中的checkField()函数

```
checkField: function(e) {
  fld = window.event ? window.event.srcElement : e.target;
  fieldname = fld.id;
  if (validator.VALIDATIONS[fieldname]) {
    re = validator.VALIDATIONS[fieldname]["regexp"];

    if (fld.value.search(re) == -1) {
      // 正则表达式不匹配
      // 找到该字段的span.error元素，并将错误消息放在其中
      spans = fld.parentNode.getElementsByTagName('span');
      if (spans.length == 0) {
        // 没有错误消息span则需创建一个
        span = document.createElement('span');
        span.className = 'error';
        fld.parentNode.appendChild(span);
      } else {
        span = fld.parentNode.getElementsByTagName('span')[0];
      }
      span.innerHTML = validator.VALIDATIONS[fieldname]["error"];
    } else {
      // 正则表达式*完全*匹配
      // 已经有一个span.error了？
      spans = fld.parentNode.getElementsByTagName('span');
      if (spans.length == 0) {
        // 没有错误消息span，所以什么都不做
      } else {
        // 删除错误消息span
        span = fld.parentNode.getElementsByTagName('span')[0];
        span.parentNode.removeChild(span);
```

```
            // 如果字段的标签上有class="error"，则将其删除
            lbl = fld.parentNode.getElementsByTagName('label')[0];
            if (lbl.className == 'error') {
              lbl.className = lbl.className.replace(/\berror\b/,'');
            }
          }
        }
      }
    }
```

加粗的部分就是修改了的地方。显示错误消息的时候，不再假设span存在，而是先检查看是否存在相应的span标签（查找文本字段的兄弟结点）。如果没找到，用createElement()创建一个新的span并插入到文档中。移除错误消息的代码也一样，先检查span是否存在，如果存在才执行移除动作。

服务器端会在出错的字段的文字部分插入class="error"以起到突出显示的效果，因此移除错误消息的时候把它也一并去掉。

7.2.2　阻止表单提交

再介绍另一项有意义的改进：如果表单上存在错误，我们应该阻止表单被提交。一个存在错误的表单即使勉强提交到服务器，结果还是会返回来给用户修改的。明知是没有用的事情，就不要让用户浪费时间了。不过做这项改进的时候要十分小心：千万不可由于JavaScript出错或者JavaScript被禁用而导致表单无法提交。我们首先需要一个函数来启用或禁用表单的提交按钮——如果表单有错，就禁用提交按钮；如果没有错误，就启用提交按钮。这个简单的函数不依赖于提交按钮的当前状态；如果按钮已经是启用的，这个函数再启用它一次也不会有什么不好的后果。

```
checkForErrors: function() {
  // 在页面中寻找span.error
  var spans = document.getElementsByTagName('span');
  for (var i=0; i<spans.length; i++) {
    // 该span有class=error吗?
    if (spans[i].className.match(/\berror\b/)) {
      // 禁用提交按钮结束
      document.getElementById("submitButton").disabled = true;
      return;
    }
  }
  // 没有span.error元素，所以启用提交按钮
  document.getElementById("submitButton").disabled = false;
}
```

为了方便编写这个函数，我们给表单的提交按钮增加一个ID属性。

有几个地方需要调用checkForErrors()函数。每当一个字段（在checkField()函数中）验证完毕，就会调用checkForErrors()。实际上可以仅在字段的错误状态发生改变的时候调用checkForErrors()，这么做似乎更明智一些。但我们这里为了让代码更容易理解，就不管三七二十一了。

init()函数执行初始化的时候也会调用checkForErrors()函数。这样当服务器端的PHP代码认为某个字段存在错误的时候，提交按钮也会在页面加载之后立即被禁用，直到用户更正。有一点很重要，那就是提交按钮是由JavaScript禁用的。如果我们在PHP代码里禁用了提交按钮，而用户的浏览器又没有打开JavaScript，那就没有办法再启用提交按钮了。服务器端和客户端代码的职责必需划分清楚，如果需要客户端代码去纠正或者更改的事情，万不可交给服务器端去做，因为在不支持客户端脚本的环境中是行不通的。

最后得到代码清单7-4，加粗的是修改了的部分。

代码清单7-4 simple-form-tidier-js-prevent-submission.php

```
...
<html>
<head>
<title>A simple PHP form using regular expressions for validation</title>
<link rel="stylesheet" href="styles.css">
<script type="text/javascript">
...

  init: function() {
    // 检查浏览器是否有戳记来执行所需的DOM脚本化
    if (!document.getElementById) return;
    // 遍历VALIDATIONS列表并找到对应的各个字段
    // 它应用并附加了一个onBlur处理程序，所以当用户不使用字段时，它根据regexp检查字段的内容
    for (fieldname in validator.VALIDATIONS) {
      fld = document.getElementById(fieldname);
      if (!fld) continue; // 如果该字段在页面中不存在，则将其忽略
      validator.addEvent(fld, "blur", validator.checkField);
    }
    validator.checkForErrors();
  },

  checkField: function(e) {
    ...
    // 最后，适时禁用或启用提交按钮
    validator.checkForErrors();
  },
```

```
    checkForErrors: function() {
      // 在页面中寻找span.error
      var spans = document.getElementsByTagName('span');
      for (var i=0; i<spans.length; i++) {
        // 该span有class=error吗?
        if (spans[i].className.match(/\berror\b/)) {
          // 禁用提交按钮结束
          document.getElementById("submitButton").disabled = true;
          return;
        }
      }
      // 没有span.error元素，则启用提交按钮
      document.getElementById("submitButton").disabled = false;
    },

    addEvent: function( obj, type, fn ) {
      ...
    }
}
</script>
</head>
<body>
<h1>A simple PHP form using regular expressions for validation</h1>
<form>

  ...
  <p><input type="submit" name="submit" id="submitButton" value="Send answers"></p>
</form>
</body>
</html>
```

7.3　用 Ajax 实现表单验证

正则表达式是一个功能强大的工具，但它也有自身的局限。在前面的例子中，检查日期的正则表达式^\d\d[\/.-]\d\d[\/.-]\d\d\d\d$允许的输入是两位数字、一个分隔符、两位数字、一个分隔符和四位数字。这条表达式可以有效地阻止一些明显错误的输入，比如“我不告诉你我的生日”，但对一些无效的输入，比如“99/99/9999”、“32/01/1995”、“29/02/2007”就无能为力了。正则表达式不可能完全捕获各种畸形的状况。还有很多情况是正则表达式完全起不到作用的：比如没有规定格式的日期、数值范围，还有URL（可以检查URL形式是否正确，但检查不了URL是否真的有效）。检查这些用户输入，要真正的程序才行。

7.3.1 服务器端验证

要在服务器端给每个字段指定一个检查用户输入函数并不困难。代码清单7-5是一个PHP的例子。

代码清单7-5 noajax-form.php，演示PHP服务器端验证

```
<?php

require_once "validation.php";

$ERRORS = Array();

if (isset($_GET["submit"])) {

  # form was submitted

  foreach ($_GET as $field => $data) {
    $check = validate($field, $data);

    if ($check != "") {
      $ERRORS[$field] = $check;
    }
  }

  if (count($ERRORS) == 0) echo "Data OK; now redirect!";
}
?>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">

<html>
<head>
<title>A simple PHP form using separate code for validation</title>
<link rel="stylesheet" href="styles.css">
</head>

<body>
<h1>A simple PHP form using separate code for validation</h1>

<form>

 <p><label for="dayofyear">Favourite day of the year (1-365)</label>
    <input type="text" id="dayofyear" name="dayofyear">
```

```
    <span class="error">
    <?php if (array_key_exists("dayofyear",$ERRORS)) echo $ERRORS["dayofyear"]; ?>
    </span>
  </p>

  <p><label for="date">Favourite date of all time</label>
    <input type="text" id="date" name="date">
    <span class="error">
    <?php if (array_key_exists("date",$ERRORS)) echo $ERRORS["date"]; ?>
    </span>
  </p>

  <p><label for="word">Favourite word</label>
    <input type="text" id="word" name="word">
    <span class="error">
    <?php if (array_key_exists("word",$ERRORS)) echo $ERRORS["word"]; ?>
    </span>
  </p>

  <p><input type="submit" name="submit" value="Send answers"></p>
</form>

</body>
</html>
```

这段代码使用了$ERRORS方式，表单的结构和前面用正则表达式来检验的表单也差不多，只不过现在对每个提交的表单值都调用validate()函数来验证。validate()函数的定义放在一个单独的文件中（原因稍后会解释），都是些浅显的PHP代码，如代码清单7-6所示。

代码清单7-6　validation.php包含每个字段的验证函数

```
<?php

function validate($field, $value) {

  switch ($field) {

    case "dayofyear":
      if (is_numeric($value) && intval($value) > 0 && intval($value) <= 365) {
        return "";
      } else {
        return "Day of year must be between 1 and 365";
      }
      break;

    case "date":
```

```
        if (strtotime($value) === false) {
          return "Invalid date (try 10 September 2000, +1 week, or next Thursday)";
        } else {
          return "";
        }
        break;

      case "word":
        if ($value == "" || is_numeric($value)) {
          return "You must supply a favorite word";
        } else {
          return "";
        }
        break;

      default:
        return "";
    }
}
```

把字段的名称和取值传给validate()函数，如果取值是有效的，函数返回一个空白字符串；如果无效则返回错误消息。显然如果有需要，可以在validate()函数内部任意调用其他不管多复杂的函数。

7.3.2 客户端

把执行表单验证的例程从页面中分离出去带来一个很大的好处，那就是我们现在用JavaScript检查每个字段的有效性的时候，可以改为向服务器发出Ajax调用，让服务器端代码来完成实际的验证工作。步骤如下：

(1) 页面加载的时候，给每个字段的blur事件附加一个处理函数。

(2) 当用户的输入焦点离开某个字段时，onBlur事件处理函数开始执行，从字段中取得当前值，并向服务器发出一个XMLHttpRequest调用，向服务器传递字段的名称和取值。

(3) 接下来服务器执行表单验证代码（表单在服务器端所使用的同一段验证代码，而非其任何副本），然后返回validate()函数的执行结果。

(4) JavaScript接收到Ajax调用的结果，如果是错误消息，就更新页面，指出错误。

再强调一次，我们现在只有一个validate()函数。服务器端代码在验证表单的时候调用的就是它，改进后的表单通过Ajax调用的也是它。

实现这个新表单要做的事情不少。在原先用正则表达式完成验证的例子中，并没有用到太复杂的JavaScript。而在基于Ajax的实现中，要通过JavaScript绑定事件、发出Ajax调用、解析Ajax调用的结果、修改页面的DOM。这个时候很应该考虑引入JavaScript库来帮助你完成以上繁重的

工作。JavaScript库可以大大简化DOM操作和Ajax调用，尤其是XMLHttpRequest的处理十分复杂，交给库去做要容易得多。

以jQuery为例，要引入这个库十分简单——如果你正在开发的应用能连到因特网，那么只要在页面的<head>部分加入以下内容：

```
<script type="text/javascript"
        src="http://code.jquery.com/jquery-latest.pack.js"></script>
```

如果你正在开发的是内部应用，用户将没办法访问因特网，那么可以从http://jquery.com下载jQuery。

通过JavaScript向服务器发送少量数据用JSON很合适。因此还会用到一个名为ServicesJSON的库（http://mike.teczno.com/json.html）。通过这个库可以方便地从服务器端返回JSON格式的数据而无需关心数据格式方面的细节。由于需要传递给向服务器的信息只是一个字段的名称和取值，那么用查询字符串（query string）来传递是最简单的了。服务器从URL里分离出这两项信息，调用validate()函数完成验证，最后用JSON格式返回结果。服务器端的PHP代码如代码清单7-7所示。

代码清单7-7　ajax-validate.php调用validate()函数并以JSON格式返回结果

```
<?php
require_once "validation.php";
require_once "JSON.php";

$field = $_GET["field"];
$value = $_GET["value"];
if (!isset($field) || !isset($value)) {
  return_json("");
  die();
}

$check = validate($field, $value);

return_json($check);
die();

function return_json($data) {
  $json = new SERVICES_JSON();
  echo $json->encode($data);
}
?>
```

直接在浏览器输入以下地址来测试Services JSON PHP库的效果，请根据实际的情况替换尖括

号部分：

http://<你的服务器地址>/ajax-validate.php?field=dayofyear&value=<不正确的取值>。

应该得到以下结果：

```
"Day of year must be between 1 and 365", which is the correct error message.
```

jQuery内建支持通过Ajax请求JSON数据，即$.getJSON()函数。用以下代码向指定的URL发出请求并取得返回的JSON数据：

```
$.getJSON("ajax-validate.php",{
  "field": "dayofyear",
  "value": "invalid-value"
}, function(data) {
  alert("this function is called with the JSON data: " + data);
});
```

如上所示，传入URL，获得组成查询字符串的参数的JavaScript关联数组，以及处理返回数据的内联回调函数。

Ajax表单验证中JavaScript的那一半就介绍完了，总结起来非常简短：

```
$(document).ready(function(){
  $('input[@type=text]').blur(function(){
    var thisfield = this;

  $.getJSON("ajax-validate.php",{
    "field": this.name,
    "value": this.value
  }, function(data) {
    $(thisfield).siblings("span.error").empty();
    if (!(data == "")) {
        $(thisfield).siblings("span.error").append(data);
      }
    });
  });
});
```

初看上去可能有些复杂，但其实只是把前面介绍的几点结合在一起。在这个最终版本里有几个值得注意的小技巧。

第一是在jQuery事件处理函数内部，this指的是实际发生事件的元素。代码中将它保存到了一个变量thisfield，为的是在回调函数中能够继续访问它。

第二是jQuery允许通过$(variable)的写法从现有的变量创建jQuery查询对象，上述示例采用了这种方式从存储的thisfield变量创建查询对象。

例中还用到了另一个jQuery方法：siblings()。它的作用等价于先前正则表达式版代码中的fid.parentNode.getElementsByTagName('span')[0]，但显然新的写法可读性提高了很多。siblings()方法本身会返回同一个父结点下的所有子结点，但可以用一个CSS选择符作为参数，筛选出你所关心的结点。

最后，例中用了empty()方法来清除错误消息span的内容，用了append()方法来向错误消息span插入服务器传回的错误消息。

用Ajax的方式处理表单验证可以有效提高表单的使用体验，避免了页面刷新，也避免了等待重新下载整个页面，同时又能充分利用服务器端在编写复杂代码上的优势，使验证例程更加全面，更有效地防止用户输入不正确的内容。

7.4　小结

表单验证很重要。如果你重视数据的质量，就应该尽力去确保其质量。（要是不重视数据质量，那还收集数据干什么呢？）虽然数据验证需要在服务器端进行，但从用户的感受和程序的可用性考虑，JavaScript增强的验证仍然是有必要的。

在一两个库的帮助下，可以让服务器端的验证同时在用户的浏览器中发挥作用，而无需另写一份JavaScript的验证代码。

愿你从此收集到的数据都是正确的！

第8章

案例研究：改良FAQ页面

本章作者：Aaron Gustafson

在因特网的历史上，少有像FAQ（常见问题）页面一样历久不衰的东西。几乎每个站点上都能见到它的身影，尽管可能改名换姓。FAQ页面的样子更是从出现以来就没怎么变过。

大多数FAQ页面都在开头用列表的形式列出全部问题，列表中的每一条都是一个链接，（通过id或name引用）分别指向页面下部成对的问题和回答。也就是图8-1的样子。维护这样的FAQ页面很快就会不胜其烦，因为不仅要创建页面下部的问题/答案，还要更新页面上部的列表（可能还有其他页面要一并处理），使之保持一致。DRY原则真要命。

注解 DRY，即Don't Repeat Yourself的缩写。

多年以来，人们一直在寻找更好的方式去管理FAQ，以图使工作轻松一些。本案例研究将探讨一种改良FAQ的途径，循着渐进增强的原则，先用单纯的HTML标记满足基本需要；再纯以CSS增加一层交互，提供更丰富的体验；最后加上一层JavaScript交互，让效果达到极致，至少也要赶上一点Web 2.0的潮流。

注解 渐进增强的概念已在第7章介绍过。

我们用Firebug（http://getfirebug.com）的FAQ页面作为实验材料，Firebug是一个流行的Firefox调试插件。本案例所用的文件可在/starting files/目录下找到。我们要改良的HTML文件是faq.html，不过大多数时间将耗费在调整faq.js和firebug.css上面。

8.1 第1课：瞄准目标

几年前我在试验用定义列表（dl）标签编写FAQ中一对对的问题/答案的时候，意识到页面顶部的问题列表是多余的。巧妙地运用CSS，可以将答案隐藏起来，效果看上去跟问题列表是一

样的。只有当单击了一条问题链接，才把相应的答案显示出来。这样做要用到一个CSS3仿类选择符（:target），不过这种做法完全符合渐进增强的精神。

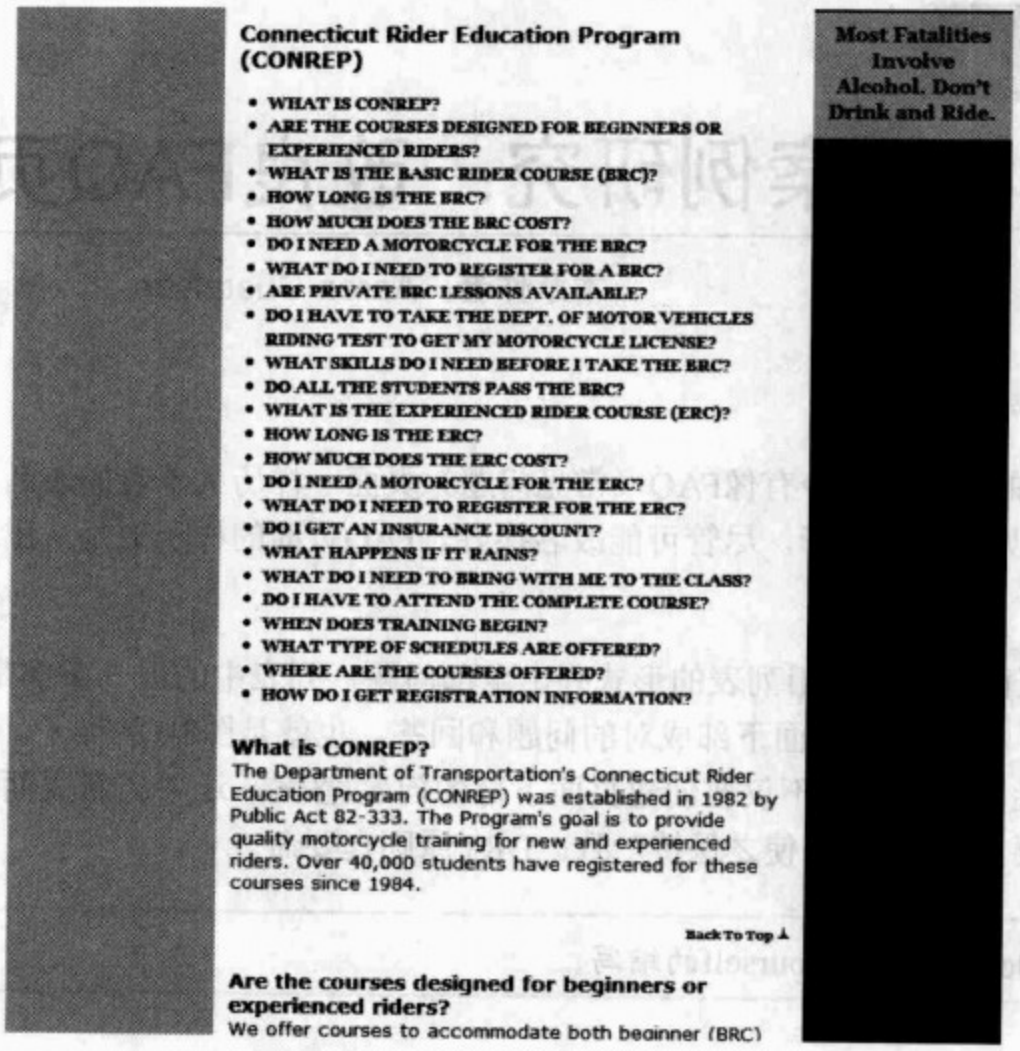


图8-1　典型的FAQ页面，截自http://ride4ever.org

用dl标签来组织FAQ，结果类似这样：

```
<dl>
  <dt><!-- QUESTION --></dt>
  <dd><!-- ANSWER --></dd>
  <dt><!-- QUESTION --></dt>
  <dd><!-- ANSWER --></dd>
  ...省略...
</dl>
```

对以上标签稍作调整，即可利用:target让答案默认隐藏起来，仅仅在单击了相应链接之后才露面。调整后faq.html的结构是这样的：

```
<dl class="faq">
  <dt><a href="#faq_1"><!-- QUESTION --></a></dt>
  <dd id="faq_1"><!-- ANSWER --></dd>
  <dt><a href="#faq_2"><!-- QUESTION --></a></dt>
```

```
  <dd id="faq_2"><!-- ANSWER --></dd>
  ... 省略...
</dl>
```

我已经预先按此思路编写好了faq.html文件。在浏览器中打开它，结果应该类似于图8-2。

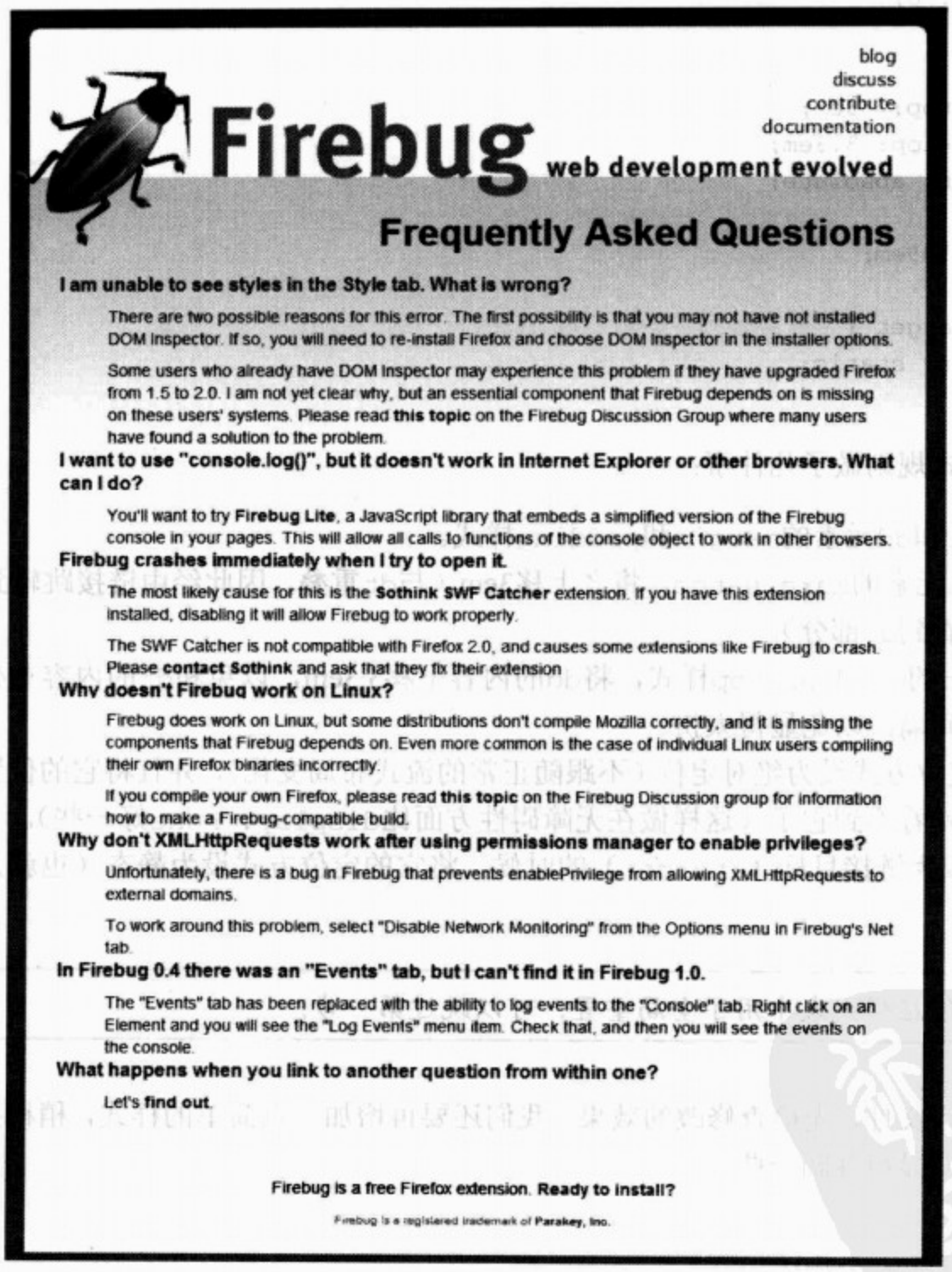


图8-2 FAQ页面的默认布局

这个页面在功能上全无问题。每则问题都有对应的答案，而且，每则问题都有id属性，因此可以收藏到浏览器的书签（或收藏夹）里。单击收藏的链接，浏览器就会直接跳到答案所在的位置，如你所愿。

现在打开firebug.css，在文件末尾加上下列样式：

```
.faq dt,
.faq dd {
  margin: 0;
  padding: 0;
}
.faq dd {
  margin-top: -3em;
  padding-top: 3.3em;
  position: absolute;
  top: 0;
  left: -999em;
}
.faq dd:target {
  position: static;
}
```

新增的样式规则做了几件事：

(1) 重置dt和dd元素的margin和padding样式。

(2) 调整dd元素的margin-top，将之上移3em（与dt重叠，因此经由链接跳转到dd部分的时候，可以同时看到dt部分）。

(3) 调整dd的padding-top样式，将dd的内容下移3.3em，以免和dt的内容重叠，同时留下一点空白作为间隔，以免显得太挤。

(4) dd的定位方式设为绝对定位（不跟随正常的流式布局变化），并且将它的位置向左移到页面之外，这样就看不到它了（这样做在无障碍性方面比display: none好一些）。

(5) 当dd成为链接目标（:target）的时候，将它的定位方式设为静态（也就是恢复成正常的流式布局）。

提示　如果已经在样式表中用了全局重置，可以跳过第一步。

别急着保存文件，先检查修改的效果。我们还要再增加一点简单的样式，稍稍拉开元素之间的间隔，让页面显得开阔一些。

```
.faq dt {
  font-weight: bold;
  margin: 1em 0 0;
}
.faq dd > :last-child {
  margin-bottom: 0;
  padding-bottom: 0;
}
```

以上样式给dt的上方增加了一点空间，并且防止dd的最后一个子元素插入多余的空白。现在保存文件，刷新浏览器欣赏一下自己的作品吧。页面刚打开的时候应该只见到问题列表，单击其中一则问题，答案就会露出来，如图8-3所示。IE 6用户暂时还看不到同样的效果，需要先按照下面的补充信息作一些修补工作。

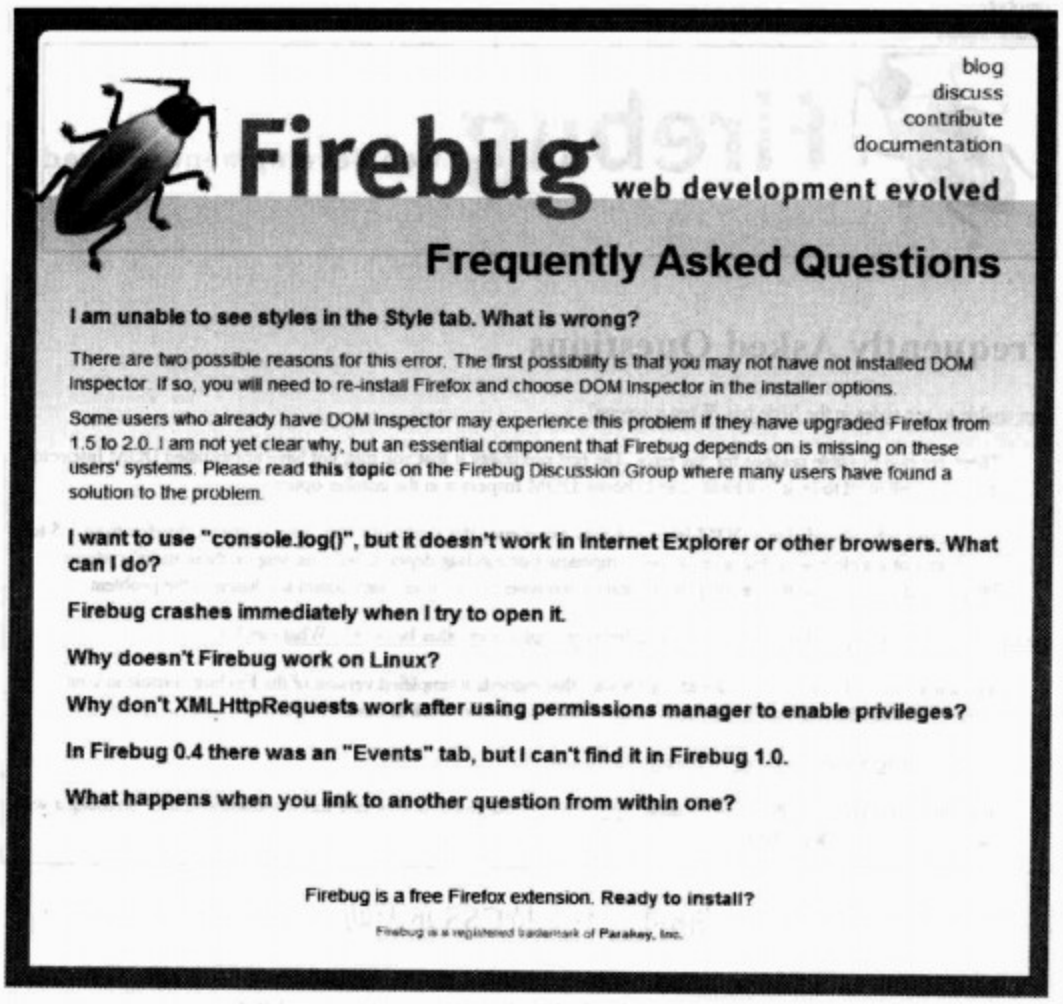


图8-3 利用:target，当用户单击问题的时候显示答案

修补IE 6

Suckerfish Shoal（http://tmldog.com/articles/suckerfish/shoal/，作者Patrick Griffiths和Dan Webb）以及Dean Edwards开发的IE 7脚本（http://dean.edwards.name/IE7/）都可以让IE 6支持target样式。如果你需要克服很多IE 6的CSS和HTML缺陷，我建议使用Dean Edwards的IE 7脚本，因为它不需要修改已有的样式。

建议使用以上修复方案的时候采用条件注释，给符合标准的浏览器减轻一些下载负担：

```
<!--[if IE lte 6]>
<script type="text/javascript" src="/js/ie7/ie7-core.js"></script>
<script type="text/javascript" src="/js/ie7/ie7-css3-selectors.js"></script>
<![endif]-->
```

在条件注释的控制下，只有IE 6及更旧版本的用户才会加载ie7-core.js和ie7-css3-selectors.js。

即使用不支持CSS的浏览器来查看这个页面，仍然会得到井井有条的问题和答案，并且完全保留了符合语义的标记之优点。见图8-4。

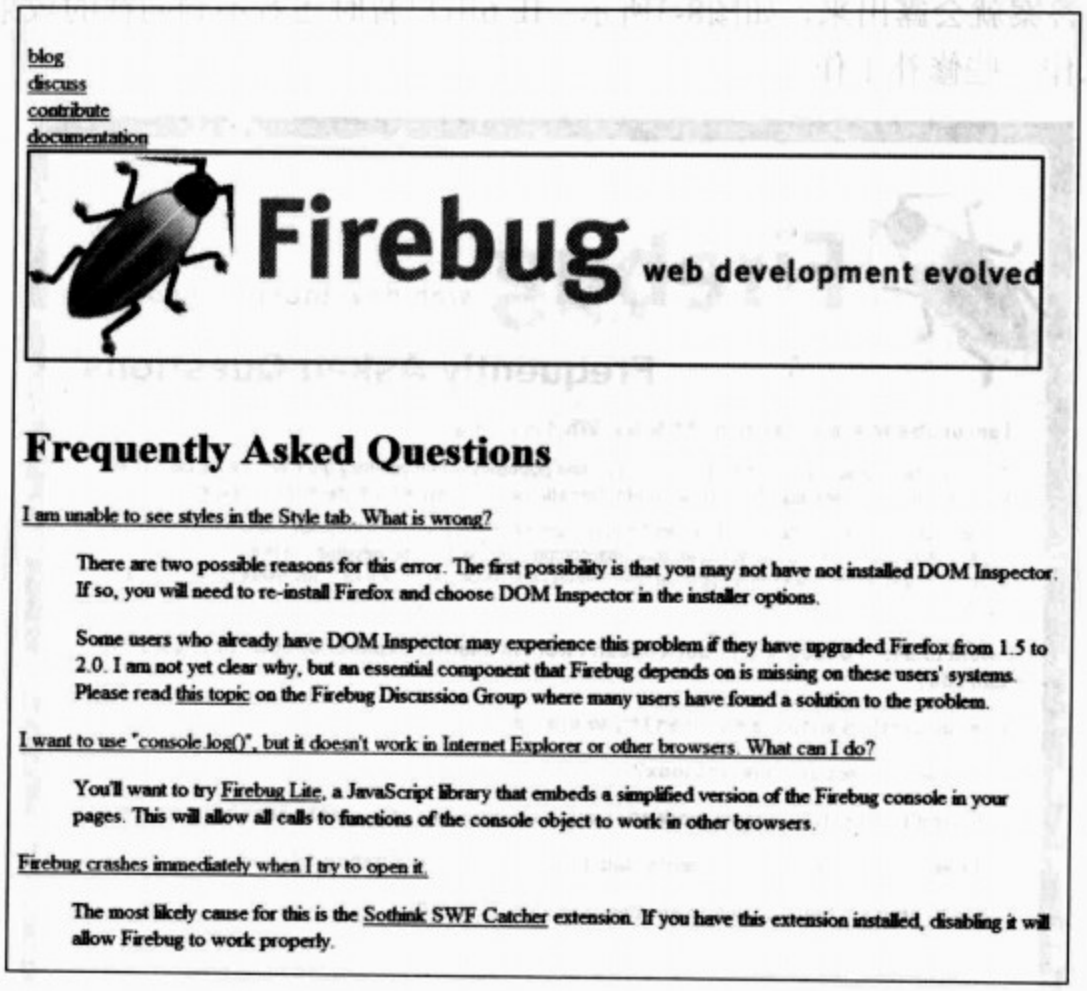


图8-4　不支持CSS亦无妨

用CSS实现的交互完成之后，就到了令人振奋的JavaScript部分。

8.2　第 2 课：JavaScript 之舞

前面的CSS交互已经不错了，不过还可以更惊艳。很多Web 2.0网站上都能看到的那种平滑展开的效果我一直都很喜欢。不过光是那种效果还不能满足需要，我们还要做一些调整。在这一节里，我们要通过JavaScript在FAQ页面上完成以下交互：

- 当单击一则问题时，触发相应的答案滑动展开。
- 让一则答案可以引用另一则答案（展开被引用的问题，但不合上原来的问题，以便看清引用关系）
- 使每一个问题/答案对都可以作为书签收藏。
- 滚动窗口，使焦点定位到最后展开的答案。

我们将用到若干库和辅助函数：

- Prototype和Moo.fx（http://moofx.mad4milk.net/）将帮助我们完成总体构造和动画。
- 为了使页面在加载过程中即开始执行脚本，需要用到Jesse Skinner编写的addDOMLoadEvent()函数（www.thefutureoftheweb.com/blog/adddomloadevent）。
- 为了方便调试，我把jsTrace（http://code.google.com/p/easy-designs/wiki/jsTrace）也带上了。

所有代码都已经包含在项目文件里。打开faq.html，可以看到已经包含了两个库以及jsTrace的脚本文件（dom-drag.js和jsTrace.js），另外还有main.js（内含addDOMLoadEvent()的代码以及jsTrace需要的trace()函数）和faq.js（这是我们构建FAQ对象的地方）：

```
<script type="text/javascript" src="prototype.js"></script>
<script type="text/javascript" src="moo.fx.js"></script>
<script type="text/javascript" src="dom-drag.js"></script>
<script type="text/javascript" src="jsTrace.js"></script>
<script type="text/javascript" src="main.js"></script>
<script type="text/javascript" src="faq.js"></script>
```

关闭faq.html，打开faq.js。这就是FAQ对象的基本结构：

```
var FAQ = {
  // 打开项
  open_items: [],
  // 运行进程
  processes: [],
  // 计时器包装
  timer:          new Object(),
  // 正在打开的是什么
  to_open:        null,
  // 要滚动到的位置
  scrolling_to:   null,
  /* 缓存当前的滚动位置，用于避免超出头尾范围的滚动 */
  scroll_cache:   null,

  // ----- 初始化
  initialize:     function(){
    // 启动事项
  },

  // ----- 打开/关闭/完成
  open:           function(){
    // 打开函数
  },
  closeAndGo:     function(){
    // 关闭函数&滚动触发器
  },
  complete:       function( dd ){
    // 内务
  },
```

```
    // ----- 滚动事项
    goTo:           function(){
      // 滚动管理器
    },
    /* 以Travis Beckham (squidfingers.com)的平滑滚动为基础,
     参考Shaun Inman (shauninman.com)的意见做了一些调整 */
    getScrollLeft: function(){
      if( document.all ){
        return ( document.documentElement.scrollLeft ) ?
                 document.documentElement.scrollLeft :
                 document.body.scrollLeft;
      } else {
        return window.pageXOffset;
      }
    },
    getScrollTop:   function(){
      if( document.all ){
        return ( document.documentElement.scrollTop ) ?
                 document.documentElement.scrollTop :
                 document.body.scrollTop;
      } else {
        return window.pageYOffset;
    }
  },
  scroll:           function(){
    // 平滑滚动的逻辑
  },
    // ----- 元素查找函数
    getDT:          function(){
      // 查找与DD配对的DT
    },

    // ----- 进程管理
    processing:     function(){
      // 看看是否在处理一些事情
    },
    wait:           function( method ){
      // 令脚本等待开始执行
    }
  };
```

现在先让你对总体的结构有所了解，接下来将逐一介绍每个属性和方法。如你在注释中所见，Travis Beckham（http://squidfingers.com）和Shaun Inman（http://shauninman.com）为我们将要建造的滚动管理贡献了不少代码和想法。有了初步印象之后，我们就从FAQ.initialize()开始动手吧。

1. 开动引擎

程序首先要扫描整个文档，找到所有类别为faq的dl元素，以进行下一步操作。

(1) 通过Prototype的$$()函数很容易实现。我们还插入了几处trace()调用，方便观察脚本的运行情况。trace()函数的具体用法请看下面的补充材料“使用jsTrace”。

```
initialize:       function(){
  trace( 'initialize()' );
  // 收集DL&循环
  $$( 'dl.faq' ).each( function( dl ){
    trace( 'DL loop' );
    // 此处发生神奇的一幕
  }.bind( this ) ); // 结束DL循环
},
```

注解 如果觉得$$()用在这里速度太慢，完全可以改成用一般的DOM方法。请注意保证返回结果是可以遍历的集合，并且注意跳过类别不是faq的dl元素。

使用jsTrace

jsTrace是受到JavaScript的亲戚ActionScript的启发，模仿ActionScript中的trace()方法开发出来的。它在页面上叠加显示一系列输出信息，帮助开发者了解程序中发生的情况。jsTrace的用法很简单，只需要定义一个trace()函数，让它向jsTrace窗口发送消息。

```
var trace;
if( typeof( jsTrace ) != 'undefined' ){
  trace = function( msg ){
    jsTrace.send( msg );
  };
} else {
  trace = function(){ };
}
```

如果jsTrace未定义，则将trace()设为空函数，这样当我们把jsTrace文件的包含语句删掉或者注释掉的时候，脚本中的trace()调用不会导致JavaScript错误。当然脚本优化的时候应该删除掉所有trace()调用，但在开发过程中能轻松地开启和关闭jsTrace会很方便。

可以拖动jsTrace调试窗口改变它的位置，也可以用右下角的三角形改变调试窗口大小。按右上角的X可以完全关闭jsTrace。如果允许cookies，那么jsTrace会记住调试窗口的位置和大小，这样你就不必每次刷新页面都调整。

(2) 别忘了还要在页面加载的时候执行FAQ.initialize()。在faq.js的尾部，FAQ对象的结束括号之后，加入以下内容：

```
if( typeof( Prototype ) != 'undefined' &&
    typeof( fx ) != 'undefined' &&
    document.getElementsByTagName( 'dl' ) ){
  addDOMLoadEvent( function(){ FAQ.initialize(); } );
}
```

在Prototype和Moo.fx都存在，并且文档中至少有一个dl元素的情况下，DOM加载完成就会执行FAQ.initialize()。加上这样的防备条件，是因为如果缺少库会产生很多JavaScript错误，而页面上要是没有dl元素，脚本就没有意义了。

(3) 保存文件，刷新浏览器，应该会看到图8-5所示的jsTrace窗口。

注解　我们在脚本里各个地方都插入了跟踪语句，因此你可以随时停下来刷新一下浏览器，看看执行结果和你预想的是否相同。

图8-5　jsTrace显示程序的执行情况

(4) 目前所有答案都是闭合起来的（其实是因为被丢到了页面左边界之外，所以看不到）。显然我们不希望CSS动作与JavaScript动作发生冲突，因此需要想办法通知CSS展开相应的答案。只需要给dl增加一个class作为展开信号：

```
initialize:       function(){
  trace( 'initialize()' );
  // 收集DL&循环
  $$( 'dl.faq' ).each( function( dl ){
    trace( 'DL loop' );
    // 打开FAQ
    dl.addClassName( 'on' );
  }.bind( this ) ); // 结束DL循环
},
```

(5) 打开infirebug.css文件，在已有的.faq dd:target样式定义前面加上相应的选择符(.faq.on dd)：

```
.faq.on dd,
.faq dd:target {
  position: static;
}
```

注解　请记住IE 6不支持多重类选择符，它会默认使用选择符中最后出现的类（在这里是on）。如果要兼容IE 6，并且觉得on太容易出现冲突，可以改成faq-on。

(6) 顺便在样式表里加上以下规则：

```
.faq.on dd {
  margin-top: 0;
  padding-top: 0;
}
```

目的是重置dd元素的margin-top和padding-top，因为当答案展开的时候不需要它们。

(7) 保存CSS文件，刷新浏览器，现在答案是完全展开的。

(8) 回到FAQ对象，我们要做两件事：实现dd元素的展开和闭合；在锚点上添加事件处理器。dd元素的部分比较简单：

```
initialize:     function(){
  trace( 'initialize()' );
  // 收集DL&循环
  $$( 'dl.faq' ).each( function( dl ){
    trace( 'DL loop' );
    // 打开FAQ
    dl.addClassName( 'on' );
    // 循环遍历DD元素
    $A( dl.getElementsByTagName( 'dd' ) ).each( function( dd ){
      // 设置高度效果(使用moo.fx)
      dd.heightFX = new fx.Style(
        dd, 'height',
        { duration: 500,
          onComplete: function(){
                        this.complete( dd );
                      }.bind( this )
        }
      );
      // 保存原始高度，留待以后使用
      dd.openHeight = dd.getHeight();
      // 闭合DD
      dd.heightFX.set( 0 );
    }.bind( this ) ); // 结束DD循环
  }.bind( this ); // 结束DL循环
},
```

程序收集FAQ dl列表中的<dd>元素，（通过Prototype的$A()函数）构成一个可以遍历的集合。然后通过Prototype的Enumerable.each()方法，将每一个dd元素的heightFX属性都设置成一个fx.Style实例，这是Moo.fx完成CSS属性过渡的一般方式。在这个例子里，我们在半秒（500ms）内完成dd的height属性过渡，并且在动画效果结束后触发FAQ.complete()方法（这个方法目前还是空的，将来会用它完成清理工作）。最后，用Prototype的Element.getHeight()方法把dd的原始高度保存在openHeight属性里，然后通过fx.Style对象的set()方法将dd的高度设为0，也就是将所有答案都闭合起来。

注解　bind(this)在循环和特效的执行过程中维护适当的this作用域。上述代码中的this总是指向FAQ对象。

(9) 如果现在刷新浏览器，你会发现显示结果简直一团糟。因为我们压缩了dd的高度，但又没有告诉它隐藏内容，于是内容全都溢出来了。为此我们要稍稍修饰一下样式表：

```
.faq.on dd {
  margin-top: 0;
  padding-top: 0;
  overflow: hidden;
}
```

(10) 再刷新一次，这回应该没问题了。dd元素全都是闭合的，不会再有内容突出来。接下来该解决锚点的问题。

(11) 锚点的处理还不算太过复杂。我们要遍历dl，查找所有的链接，判断是否页面内链接，以及链接地址中指定的id是否存在，以此来决定是否给链接元素添加onclick事件处理器。设置事件处理器用到Prototype的Event.observe()方法。在事件处理器内部还要判断，到底链接本身就是一则问题（此时应闭合其他所有已展开的答案，滚动到当前选择的问题，展开答案），还是只是指向另一则问题的引用（应展开新的答案，并滚动到新答案的位置）。这项判断实现起来很简单，只需要检查链接的parentNode是否dt，然后相应调用FAQ对象的方法。

```
initialize:      function(){
  trace( 'initialize()' );
  // 收集DL&循环
  $$( 'dl.faq' ).each( function( dl ){
    ... 省略...
    // 循环遍历锚点
    $A( dl.getElementsByTagName( 'a' ) ).each( function( a ){
      var href = a.getAttribute( 'href' );
      /* 如果不是链接不是指向页面内的锚点，或者页面内找不到链接的目标，那么跳过该链接不作处理 */
      if( !href.match( /#/ )||
          !$( href.replace( /.*?#(.*)/, '' ) ) ) return;
      // 设置事件处理器
      Event.observe( a, 'click', function( e ){
        var el = Event.element( e );
        var id = el.getAttribute( 'href' ).replace( /.*?#/, '' );
        trace( 'looking for ' + id );
      // 检查链接是否已经展开
      if( this.open_items.indexOf( id ) == -1 ){
        this.to_open = id;
        // 检查链接是否位于DT元素内部
        if( el.parentNode.nodeName.toUpperCase() == 'DT' ){
          /* 如果是，闭合当前所有已展开的问题，跳转到所选问题 */
          this.closeAndGo();
```

```
            } else {
              // 否则单纯跳转到所选问题
              this.goTo();
            }
          }
          return false;
        }.bind( this ), false );
      }.bind( this ) ); // 结束锚点循环
    } // 结束DL循环
  },
```

在此要说明一下程序的组织和各部分的联系。上面的代码调用了两个方法：FAQ.closeAndGo()和FAQ.goTo()，它们分别对应“闭合-滚动-展开”（单击问题的时候）和“滚动-展开”（单击引用的时候）两种交互情形。稍后我们会探讨这两个方法内部的逻辑。请注意由于这两个方法还没实现，现在单击链接是看不到什么效果的。

我们也用到了FAQ对象的一下属性，首先接触的是FAQ.open_items。在事件处理器内部，执行任何动作之前，都要先搜索这个数组，（用Array.indexOf()方法）检查链接目标的id是否已经存在于数组之中。稍后我们编写FAQ.open()方法的时候，将会把新展开的答案的id添加进FAQ.open_items数组。

我们还用到了FAQ.to_open属性。这个属性的用途是跟踪记录正在展开的元素的id，省得再三地通过参数把它传递给各个方法。

现在刷新页面，试着单击几则问题，你会在jsTrace调试窗口中看到事件的跟踪信息。当然我们现在还没实现随着单击展开答案的功能，不过可以看到程序已经能够在单击操作的触发下找到正确的id，如图8-6所示。

图8-6 单击几则问题之后的跟踪结果

2. 开合动画

有了FAQ.initialize()搭建的基础，可以开始着手实现展开和闭合dd元素的动画效果。单击事件根据情况触发FAQ.closeAndGo()和FAQ.goTo()两个方法中的一个。

(1) 先来完成FAQ.open()：

```
open:                function(){
  trace( 'open()' );
  var dd = $( this.to_open );
  dd.heightFX.custom( 0, dd.openHeight );
},
```

这个方法很直观，它找到你想展开的dd元素（$(this.to_open)），然后在元素上实现一段自定义动画，使dd的高度从0逐渐增加到dd.openHeight属性中保存的dd元素的原始高度。

(2) 接着让FAQ.goTo()方法展开dd（稍后还会增加一些与滚动相关的代码，现在先尽量保持简单）：

```
goTo:                function(){
  trace( 'goTo()' );
  this.open();
},
```

(3) 完成FAQ.closeAndGo()方法的逻辑：

```
closeAndGo:          function(){
  trace( 'need to close '+this.open_items.length+' dds' );
  if( this.open_items.length > 0 ){
    $A( this.open_items ).each( function( id ){
      var dd = $( id );
      dd.heightFX.custom( dd.openHeight, 0 );
    }.bind( this ) );
  }
  this.goTo();
},
```

FAQ.closeAndGo()所做的事情，就是闭合任何已展开的dd元素（它们的id都保存在FAQ.open_items里面），然后调用FAQ.goTo()。目前还缺少把它们加进FAQ.open_items数组的部分，这就轮到FAQ.complete()方法出场了。

(4) 还记得在效果完成时候要调用FAQ.complete()吗？这正是向FAQ.open_items数组增删元素的绝好时机。因为在展开和闭合两种情况下都要调用这个方法，所以要把执行效果的dd元素的引用传递进去，以区分两种情况。如果dd元素的id和FAQ.to_open相同，我们就知道它是刚刚完成了展开效果，于是用Array.push()将它加进FAQ.open_items数组。如果不相同，那么就是闭合的情况，于是我们用Prototype的Array.without()方法将它从数组中去掉：

```
complete:            function( dd ){
  trace( 'transition complete' );
  var id = dd.getAttribute( 'id' );
  if( this.to_open == id ){
    this.open_items.push( id );
  } else {
```

```
        this.open_items = this.open_items.without( id );
      }
    },
```

(5) 保存刚才所做的工作，刷新浏览器。单击第一则问题，应该能看到答案缓缓展开。单击第二则问题，刚才展开的第一则会缓缓收回，同时第二则的答案展开。再试一下单击最后一则问题，测试一下引用链接事件。答案展开之后，单击答案里面的链接；这时会看到第一则展开，而最后一则的展开状态仍然不变，一如我们的设计。随便单击另一则问题会使两者一齐收回，同时展开单击的问题。

提示 如果动画看起来不连贯，试一下把faq.html中用来包含jsTrace的<script>元素注释掉。在jsTrace未定义的情况下，trace()调用将被忽略，程序应该会运行得较为顺畅。

一切都进展得不错。但先别忙着处理滚动，我们还要谈一谈冲突的问题。

3. 减少冲突

如果在一个页面上挤了太多的动画、滚动之类的效果，有可能使用户分心，感到应接不暇。另外，如果FAQ里的答案非常长，那么在答案伸缩的过程中会出现怪异的忽前忽后的滚动。

如果能有条理地触发事件，就能避免效果之间的冲突了。应该留一些时间让已展开的问题完成闭合，再开始滚动页面；新问题展开之前也应该暂停滚动。

设立过程队列是其中一种解决方案，让各个方法在队列中轮候。实现过程中我们将遇到两个需要实现的辅助方法和FAQ对象中已经存在的一些属性。

(1) 首先实现FAQ.processing()方法。这是个简单的方法，它的功能是如果FAQ.processes队列不为空就返回true，如果为空就返回false。它起到了指示器的作用，告知待执行的方法此刻执行是否安全。

```
processing:      function(){
  trace( 'current processes: ' + this.processes.toString() );
  return ( this.processes.length > 0 ) ? true : false;
},
```

(2) FAQ.wait()的参数是需要加入轮候的方法名称，它会设置一个定时器，10ms之后再尝试执行该方法：

```
wait:            function( method ){
  trace( 'waiting to run this.' + method + '()' );
  this.timer[ method ] = setTimeout( 'FAQ.' + method + '()', 10 );
  return false;
},
```

(3) 执行实际工作的方法也要相应修改，改动之处并不难理解。首先从FAQ.open()开始：

```
open:           function(){
  if( this.processing() ) return this.wait( 'open' );
  clearTimeout( this.timer['open'] );
  trace( 'open()' );
  var dd = $( this.to_open );
  dd.heightFX.custom( 0, dd.openHeight );
},
```

调用FAQ.open()的时候，它检查是否有其他过程正在进行当中。如果有，就等待10ms之后再试。一旦畅通无阻的时候，它就把定时器（保存在FAQ.timer对象里头）清除掉，剩下的部分像原先一样执行下去就行了。

(4) FAQ.closeAndGo()和FAQ.goTo()也照此办理：

```
closeAndGo:     function(){
  if( this.processing() ) return this.wait( 'closeAndGo' );
  clearInterval( this.timer['closeAndGo'] );
  trace( 'need to close '+this.open_items.length+' dds' );
  ...省略...
},
...省略...
goTo:           function(){
  if( this.processing() ) return this.wait( 'goTo' );
  clearInterval( this.timer['goTo'] );
  trace( 'goTo()' );
  ...省略...
},
```

(5) 还要安排一下在轮候队列中添加和移除dd闭合过程（滚动部分马上就会谈到）：

```
closeAndGo:     function(){
  ...省略...
  if( this.open_items.length > 0 ){
    $A( this.open_items ).each( function( id ){
      trace( 'closing ' + id );
      this.processes.push( id );
      var dd = $( id );
      dd.heightFX.custom( dd.openHeight, 0 );
    }.bind( this ) );
  }
  this.goTo();
},
complete:       function( dd ){
  ...省略...
  this.processes = this.processes.without( id );
},
```

注解 可以不用把展开过程添加到队列中，这是安全的，因为在这个例子里，FAQ.open()总是最后一个被调用的方法。

(6) 保存代码，刷新浏览器，试着单击几个链接，你会看到以展开的答案闭合之后，新的答案才会展开。真不错。

现在该轮到滚动了。

4. 滚动

滚动效果实际上挺简单的，只牵扯到几个方法。FAQ.getScrollLeft()和FAQ.getScrollTop()获取当前的滚动位置，我们在前面已经看过了，那就直奔FAQ.scroll()吧。

(1) FAQ.scroll()全程处理页面的滚动。因为有FAQ.processes告诉我们当前的执行情况，所以FAQ.scroll()可以轻易地知道滚动正待开始还是已在进行当中。通过比较当前滚动位置(从FAQ.getScrollLeft()和FAQ.getScrollTop()得知）和打算滚动到的位置，FAQ.scroll()能确定何时停止滚动。当前缓存位置还会缓存到FAQ.scrollcache，因此也很容易确定窗口是否已经不能再滚动下去（也就是已经滚动到了窗口的最顶或者最底）：

```
scroll:          function(){
  if( this.processes.indexOf( 'scroll' ) != -1 ){
    // 滚动
    var left = this.getScrollLeft();
    var top = this.getScrollTop();
    if( // 几乎到位了
        ( Math.abs( left - this.scrolling_to[0] ) <= 1 &&
          Math.abs( top - this.scrolling_to[1] ) <= 1 ) ||
        // 已经到头了，没法再滚动
        ( this.scroll_cache &&
          ( this.scroll_cache[0] == left &&
            this.scroll_cache[1] == top ) ) ){
        trace( 'wrapping the scroll()' );
        window.scrollTo( this.scrolling_to[0], this.scrolling_to[1] );
        clearInterval( this.timer.scroll );
        this.scroll_cache = null;
        this.processes = this.processes.without( 'scroll' );
    } else {
        trace( 'scrolling()' );
        window.scrollTo( left + ( this.scrolling_to[0] - left )/2,
                         top + ( this.scrolling_to[1] - top )/2 );
        this.scroll_cache = [ left, top ];
    }
  } else {
    trace( 'starting the scroll()' );
    this.processes.push( 'scroll' );
```

```
      this.timer.scroll = setInterval( 'FAQ.scroll()', 100 );
    }
  },
```

我们再次定义了一个定时器（FAQ.timer.scroll）以100ms的间隔触发FAQ.scroll()，以便能平滑地将用户带到FAQ.scrolling_to中设定的滚动目标。

(2) 滚动目标在FAQ.goTo()中设定，其中在调用FAQ.scroll()之前借助了一个辅助方法FAQ.getDT()。FAQ.getDT()凭借Prototype的Position.cumulativeOffset()方法和this引用得到dt元素的位置：

```
goTo:               function(){
  ...省略...
  /* 我们打算滚动到DT，所以先要知道它的位置 */
  this.scrolling_to = Position.cumulativeOffset( this.getDT() );
  trace( 'DT position: '+ this.scrolling_to[0] + ',' + this.scrolling_to[1] );
  this.scroll();
  this.open();
},
...省略...
getDT:              function(){
  trace( 'looking for the DT associated with ' + this.to_open );
  var el = $( this.to_open ).previousSibling;
  while( el.nodeName.toLowerCase() != 'dt' ){
    el = el.previousSibling;
  }
  return el;
},
```

因为我们有过程队列处理作保障，所以可以放心地在FAQ.goTo()中调用FAQ.open()，不怕与刚刚执行的FAQ.scroll()发生冲突。就剩下最后的冲刺了，还有一点整理工作需要完成。

5. 最后整理

因为滚动发生在展开目标答案之前，有时候在答案完全展开之后会产生一些额外的滚动空间。为了补偿这些空间，可以让FAQ.complete()方法再试着滚动一次，以防万一：

```
complete:           function( dd ){
  ...省略...
  if( this.to_open == id ){
    this.open_items.push( id );
    // 再执行一次滚动（以防万一页面发生了变化）
    this.scrolling_to = Position.cumulativeOffset( this.getDT() );
    this.scroll();
  } else {
    this.open_items = this.open_items.without( id );
  }
```

```
    this.processes = this.processes.without( id );
  },
```

最后一点小调整是关于书签/收藏的。如果要让书签直接定位到某一则问题，那么应该满足以下需求：

- 页面加载的时候自动展开书签所指的问题。
- 开启JavaScript状态下所设的链接，在禁用JavaScript的情况下应该也能正常使用。
- 页面加载的时候不会跳转到书签所指的锚点（因为我们希望尽可能由我们的代码去控制滚动）。

满足以上全部需求需要做一点动态的id重写，让脚本把URI字符串中找到的片段标识符按照新的方案转换成id，然后触发相应的dd展开。这样做是为了让页面在加载的时候找不到书签中的id，从而使滚动全部交由JavaScript控制。所有逻辑都在FAQ.initialize()方法里：

```
initialize:     function(){
  ...省略...
  $$( 'dl.faq' ).each( function( dl ){
    ...省略...
    $A( dl.getElementsByTagName( 'dd' ) ).each( function( dd ){
      ...省略...
      // 重置ID（让书签功能生效）
      var new_id = 'FAQ_' + dd.getAttribute( 'id' );
      dd.setAttribute( 'id', new_id );
      // 闭合这个DD
      dd.heightFX.set( 0 );
    }.bind( this ) ); // 结束DD循环
    // 遍历锚点
    $A( dl.getElementsByTagName( 'a' ) ).each( function( a ){
      var href = a.getAttribute( 'href' );
      /* 如果链接不是页面内锚点，或者找不到TARGET，那么就跳过该链接不作任何处理。 */
      if( !href.match( /#/ )||
          !$( href.replace( /.*?#(.*)/, "FAQ_$1" ) ) ) return;
      // 设置事件处理器
      Event.observe( a, 'click', function( e ){
        var el = Event.element( e );
        var id = 'FAQ_' + el.getAttribute( 'href' ).replace( /.*?#/, '' );
        ...省略...
      }.bind( this ), false );
    }.bind( this ) ); // 结束锚点循环
  }.bind( this ) ); // 结束DL循环
  // 检查是否遇到了书签情形
  if( window.location.toString().indexOf( '#' ) != -1 ){
    var id = 'FAQ_' + window.location.hash.toString().replace( /#/, '' );
    trace( 'loading with bookmark: ' + id );
    if( !$( id ) ){
      trace( "can't find " + id );
```

```
        } else {
          this.to_open = id;
          this.open();
        }
      }
    },
```

第一步修改是在遍历dd的循环里，重写dd元素的id。接着在锚点循环里，把链接目标地址里的id部分改写成重写过的id（'FAQ_'+原来的id）。这样程序的其余部分就不会受到以上改动的影响。

最后一步是增加一段程序，处理URI中包含id引用的情况（也就是直接指向某一则问题的书签或者链接）。我们把URI中的片段标识符部分按照新方案改写成新id，并且检查具有这个id的元素是否存在。如果存在就打开它，小菜一碟。

保存修改，刷新浏览器，好好看看吧。作品就在我们的眼前：一个美观的、渐进增强的FAQ页面。

8.3　小结

在这个案例研究里，我们了解了渐进增强的FAQ界面的创建过程。最基础的版本是用符合语义的标记（定义列表dl）列出的一串问题和答案。第二层体验用一些高级CSS稍加点缀，借助:target仿类选择符以无障碍的方式显示和隐藏内容。最后一层体验由JavaScript来承担，它以漂亮的滑动效果动态地展开、闭合答案，并且将页面滚动到适当的位置，为读者提供一点便利。

除了学习完成任务所需的技术，我们实际尝试了Prototype和Moo.fx库的若干功能，还学到了如何通过监控执行过程来处置潜在的脚本冲突。

第9章 案例研究：动态帮助系统

本章作者：Dan Webb

现代Web应用的一大特征就是比前代应用的界面更丰富、更动态，更接近桌面应用的体验。这主要应该归功于一些重要的JavaScript库（当然还有它们的开发者），让我们在开发界面的时候有了更坚实的立足点。整本书中所介绍的JavaScript库，替我们照料了大量的跨浏览器兼容和JavaScript离奇用法的细节。然而，我们在开发Web应用的过程中，需要克服的远不止这些困难。用户界面不但必须与服务器端的代码合作无间，还要尽可能地稳健可靠（即使在不支持JavaScript的平台上），易于维护。

以此为目标，我们用一个帮助系统作为例子，向你演示如何实现典型的动态UI功能。全面讲解从计划到设计各个方面，乃至与服务器端元素的接口，以使你清楚地认识到如何在真实的应用程序中编写出坚实可靠的JavaScript代码。

本章的示例代码基于Ruby On Rails，它是现在正当红的一个Web开发框架。我选择Ruby On Rails是因为它与我们所选的JavaScript库——Prototype和Low Pro配合得很好，也因为它在发挥自身作用的同时，并不会干扰到我们在此项目中真正关注的部分——也就是HTML、CSS、JavaScript和渐进增强的理念。即使你不熟悉Ruby On Rails也不必担心，例子中展示的概念可以很容易地套用到PHP、Java、Django等你喜欢的任何平台中。

9.1 任务

我们具体要做些什么？还是先把场景说明一下吧。任何应用，不管是不是Web应用，只要功能不算太简单，总免不了要有一个帮助系统引领用户和提供协助信息。很多Web应用的典型做法是提供一个帮助链接，单击之后在一个新窗口显示帮助内容。用户要东翻西找才能见到自己关心的部分，然后还要在帮助窗口和程序窗口之间来回倒腾，才能把帮助信息和程序对上号。很不理想。

某些桌面程序可能已经找到了更好的解决方法——Office大眼夹助手。哦，也许不是，开个玩笑!我真正想说的是在Microsoft Word、Excel等程序中的边栏中出现的上下文敏感的帮助。当用

户在使用程序的某一部分时遇到不解之处，只要按下某个组合键，或者单击界面上的帮助按钮或图标，相应的帮助就出现在当前程序的旁边——无需在数不清的帮助内容里大海捞针，也不用在帮助窗口和应用窗口之间倒来倒去。在你需要的时刻，把你需要的帮助送到你的眼前。

凭着一点Ajax和服务器端的小法术，这个项目准备把以上功能从桌面搬到Web应用上去。不多啰唆了，我们动手吧。

9.2　计划和准备

除了一般的准备工作，比如新建文件、下载库（下文将详细说明）之类，还有必要事前做一点规划。我说的不是什么设计图和书面说明，而是请你在开始敲打键盘之前先暂停一下，想想整个项目该怎么组织到一起。

为了使最终的实现能应付各种情况，我们将从最普通的帮助系统开始，渐进增强我们的帮助栏。也就是说先要用纯HTML和CSS实现静态版本，保证它不出问题，然后再用DOM脚本编程来改进帮助栏的UI。这样即使用户的浏览器不支持JavaScript，或者防火墙阻止了JavaScript，用户也能顺利地使用帮助，虽然样子简陋一些。总而言之，要让帮助内容尽可能在所有情况下都能看得到，只让具有高级JavaScript支持的浏览器访问帮助是不可接受的。

要想顺利实现渐进增强，一点未雨绸缪是必不可少的。我从Jeremy Keith的书里摘一段话：从一开头就为渐进增强作计划，到最后才去实现。虽然一开始做的仅仅是基本功能，帮助栏要到后面才作为增强加诸其上，但从一开始就为此规划是值得的。等你实现过几次渐进增强，事前规划就会变成习惯，现在先按部就班地计划一下功能该如何实现吧。

9.2.1　总体设计

规划一项渐进增强的特性，关键在于先理顺基本版的流程，然后找出增强版从何处开始另辟蹊径（见图9-1）。

从图中可以看出，就程序的流程而言，同一功能的基本版及增强版差异很小。好极了，正好说明我们的设计没错。实现渐进增强的时候应该注意这种信号。如果你发现有和没有JavaScript的情况下，程序的设计截然不同，那就说明应该还有更好的方法，你该重新换一个思路了。

9.2.2　项目准备

我假设你已经在机器上装好了Ruby和Ruby On Rails。如果还没有，请访问Ruby On Rails的网站，上面有完整的安装指导（http://rubyonrails.org/download）。装完Rails之后，请到www.apress.com下载示例代码，解压到你的工作目录（本章后面也附了代码，方便大家参考）。里面已经准备好了一个Rails应用，也已经为你完成了一些力气活，所以你可以专注在UI上。

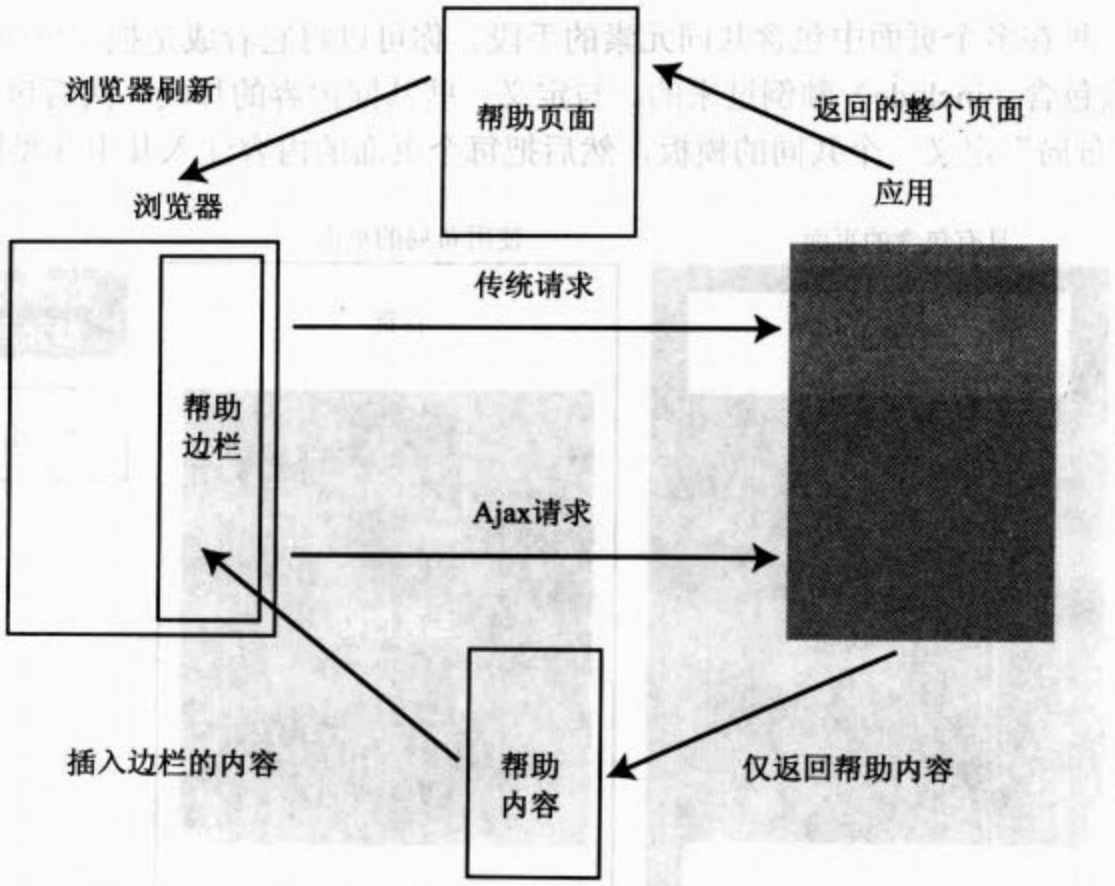


图9-1 帮助功能的基本版及增强版的流程图

例子里需要修改到的文件都放在public目录（样式表及脚本）和app/views目录（HTML模板，文件扩展名.rhtml）。请注意用到的库都放在public/javascripts。在这个项目里，我们用了Prototype（prototype.js）、Low Pro（lowpro.js）和处理视觉效果的Moo.fx for Prototype（mootx.js）。这些我们留到以后再说——现在要动手写一点HTML了。

9.3 编写标记

虽然这本书的主题不是HTML，还是应该强调把HTML部分作为开发流程中一个需要深思熟虑的部分，而不是可以过后再补上的东西。HTML是任何Web应用的UI的基础，所以使用HTML要小心谨慎，注意语义，合理的ID和类名对DOM脚本编程会有很大帮助。

仔细斟酌所用的元素是否适合当前内容，不要落入“恋div癖”的陷阱（什么东西都用 元素包起来，把HTML变得很臃肿）。干净整洁、意义明确的标记意味着浏览器在很多情况下都能为你代劳。用<a>标签链接内容，用<ul>和<ol>标记列表和表单，用按钮触发服务器端的动作。顺着浏览器的脾性去做，别误用元素然后再在误用的基础上用JavaScript编码。提醒的话说完了，我们开始编写代码吧。

9.3.1 用布局处理共同的标记

首先要编写一些共同的HTML，它们将组成应用的布局。用Ruby On Rails的话来说，布局

（layout）是一种在多个页面中包含共同元素的手段。你可以将它看成是把（PHP、ASP等平台上的）服务器端包含（include）颠倒过来的。与定义一些共同内容的片段，然后包含进每个文档里不同的是，“布局”定义一个共同的模板，然后把每个页面的内容注入其中（见图9-2）。

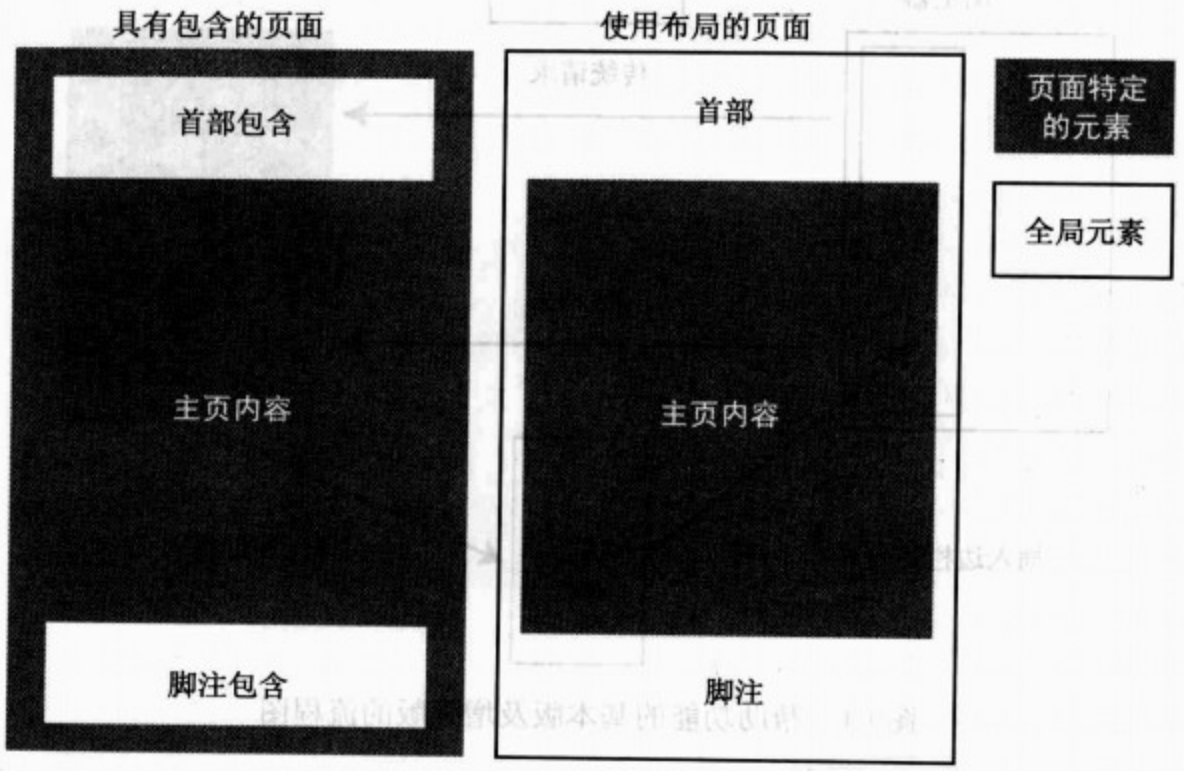


图9-2　包含与布局的区别

Ruby On Rails的布局特性，在其他框架，如CakePHP、CodeIgniter和Django里都能找到等价的特性。即使你所用的框架没有此功能，自己实现也很简单。我们在后面将会看到，布局所起的作用不同一般。

下面是预先准备好的布局：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html>
  <head>
    <meta http-equiv="Content-type" content="text/html; charset=utf-8">
    <title>Pro DOM Scripting with Ajax, APIs and Libraries Chapter 9</title>
  </head>

  <body>
    <div id="content">
      <div id="header">
        <h1>Examplr Beta</h1>
      </div>

      <%= yield %>
```

```
    </div>
  </body>
</html>
```

在Rails里，<%= yield %>语句标明页面内容插入的位置。以上简单的布局包括一个内容区域（ID为content的 ），以及在全应用之内都一致的页首。进行到9.4节的时候，还会再增加一个帮助边栏所在的区域。

这个文件在示例代码里的app/views/layouts/application.rhtml。

9.3.2 添加一个示例应用页面

布局就位之后，可以把注意力转到应用页面。出于演示的目的，我们只建一个页面，不过这页面是一个相当含糊的“添加链轮”表单，用户肯定需要帮助才能应付。这部分的HTML标记也是很简单的：

```
<h2>Add A Sprocket <a href="/help/sprocket" rel="help">?</a></h2>

<form action="/nowhere" method="post">
  <fieldset>
    <p><label for="name">Name</label> <input name="name" id="name" /></p>
    <p><label for="spid">Sprocket ID <a href="/help/sprocket#sprocketid"
rel="help">?</a></label> <input name="spid" id="spid" /></p>
    <p><label for="desc">Description</label> <textarea name="desc"
id="desc"></textarea></p>
    <p><label for="tr">Tacion Rating <a href="/help/tacion"
rel="help">?</a></label> <input name="tr" id="tr" /></p>
    <p class="check"><label for="xx75">XX-75 Approved <a href="/help/xx75"
rel="help">?</a></label> <input name="xx75" id="xx75" type="checkbox" /></p>
    <p class="submit"><input type="submit" name="submit"
       value="Add Sprocket" /></p>
  </fieldset>
</form>
```

文件的位置在示例代码里的app/views/main/index.rhtml。请注意表单里所有上下文相关帮助的链接都加上了取值为"help"的rel属性。rel属性描述链接与所链资源之间的关系，也是挂接许多类型脚本的好地方。后面将用它来判断单击链接时是否应该显示帮助栏里的回答。你也可以用类名区分帮助链接与一般的链接，但从语义上看，用rel来描述关系是最合适的。

这些帮助链接指向help目录下的各个页面。例子里已经替你做了这部分工作，但不妨看一下控制器的代码：

```
class HelpController < ApplicationController

  def show
    render :template => '/help/' + params[:path].join('/'), :layout => 'help'
```

```
    end

  end
```

如果你不熟悉Ruby和Rails，上面的操作是用名为help的布局去渲染app/views/help目录下指定的模板。例如，如果指定的URL是/help/sprocket，就会渲染位于app/views/help/sprocket.rhtml的模板。（我已经在目录下放了几个帮助页面。）帮助布局和主布局的外观区别不大。请试着单击几个帮助链接，你将看到结果：帮助内容在独立的页面中显示。

第一步的目标已经达成：有了表单，也有了能打开帮助内容的链接。现在是时候开始渐进增强，让帮助栏更上一层楼。

9.4　用 CSS 添加样式

这本书的主题是JavaScript，不是CSS，所以我不打算太过深入表单样式的细节。例子里面包含了一个CSS文件（public/stylesheets/main.ess）。要在Rails项目里使用这个文件，请打开布局文件（app/views/layouts/application.rhtml），在<head>部分插入下面这行代码：

```
<%= stylesheet_link_tag "main" %>
```

现在表单页面应该有了样式。这时应该开始考虑帮助面板和它的样式。首先要在HTML里加入帮助面板，也就是在布局文件中插入 。可以在里面先写两句虚假的帮助内容，方便测试。现在布局文件成了这个样子：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html>
  <head>
    <meta http-equiv="Content-type" content="text/html; charset=utf-8">
    <title>Pro DOM Scripting with Ajax, APIs and Libraries Chapter 9</title>
    <%= stylesheet_link_tag "main" %>
  </head>

  <body>
    <div id="content">
      <div id="header">
        <h1>Examplr Beta</h1>
      </div>

    <%= yield %>

    </div>
    <div id="help">
      Some example help content.
    </div>
```

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

现在帮助面板已经在页面里了，只不过吊在表单的后面，显得很难看。那就用CSS来纠正它。帮助面板有两种状态：关闭（默认状态）和打开。关闭的时候，我们要把整个面板隐藏起来，因此在CSS文件里面添加以下样式：

```
#help {
  display: none;
}
```

帮助面板打开的时候，主面板需要调整大小，在右侧给帮助面板腾出位置，然后把帮助面板在那里显示出来。虽然可以通过JavaScript操作元素的style属性达到目的，但最好是用更合适的工具来完成这项工作。关系到呈现的应该找CSS，所以应该在body元素上加一个class作为标识，区分帮助打开和关闭两种页面状态。如果body的类名是with-help，就应用相应的样式，显示出帮助面板。请在CSS文件中加入以下样式：

```
body.with-help {
  margin-right: 350px;
}
body.with-help #help {
  background: #F4EEBC;
  border: 1px solid #000;
  border-color: #CCC #333 #333 #CCC;
  width: 320px;
  position: absolute;
  top: 0;
  right: 0;
  margin: 8px 30px;
  padding: 10px;
  overflow: hidden;
}
```

如果浏览器里安装了Firebug，可以在控制台中输入以下代码，测试帮助面板的打开和关闭：

```
document.body.className = "with-help";
```

9.5 Prototype 和 Low Pro 出场

Prototype是当代JavaScript库的先驱，现在正支撑着网上许多最新的应用。Prototype（下载和详细文档请访问http://prototypes.org）已经包含在Ruby On Rails之中，它也可以并且经常独立使用。Scriptaculous常与Prototype搭配使用，提供视觉效果和组件。不过由于例子里只会用到一些轻量级的效果，所以我们选择了更轻更小的Moo.fx库。

最后，为了保持不唐突的编码风格，我们需要来自Low Pro（http://danwebb.net/lowpro）库的

一些帮助，它也是Prototype库的一个扩展。Low Pro包含了大量对Prototype事件处理、DOM操作及创建的扩展，还有我们在例子里用得最多的：行为。

在项目中使用库

Prototype、Low Pro和Moo.fx都已经包含在例子文件里，要把它们加进项目，请打开application.rhtml布局文件，在头部插入以下代码：

```
<%= javascript_include_tag 'prototype', 'lowpro', 'moofx' %>
```

当然也可以直接写HTML：

```
<script type="text/javascript" src="/javascripts/prototype.js"></script>
<script type="text/javascript" src="/javascripts/lowpro.js"></script>
<script type="text/javascript" src="/javascripts/moofx.js"></script>
```

这样就准备好让帮助栏成为动态的了。

9.6　让帮助栏可用

神奇的一刻到了。我们即将在基本的帮助系统之上叠加JavaScript行为的层次，把它变成一个动态的、上下文敏感的帮助边栏。实际上只需要加入很少一点JavaScript，就能完成飞跃，这也正是设计稳固的标志。如果应用的每个部分都能做好各自的分内事，那么JavaScript就只需要充当胶水的角色，将页面上的事件与动作连接起来，完成我们需要的行为。

9.6.1　建立帮助控制器

为了方便管理代码，我们要把帮助栏的实现包装成一个对象。因为每个页面只会有一个帮助栏，所以可以只用单独一个Help对象来表示。打开public/javascripts/application.js，动手编写帮助控制器。首先是基本的打开和关闭功能：

```
var Help = {
  open : function() {
    $(document.body).addClassName('with-help');
  },
  close : function() {
    $(document.body).removeClassName('with-help');
  }
};
```

前面已经提过，要打开和关闭边栏，只需要在body元素上添加或者删除with-help类。接着要建一个方法，通过Ajax请求取得帮助页面的内容，并更新help元素。Prototype的Ajax.Updater方法（详见第5章）正好合适：

```
Help = {
```

```
    open : function() {
      $(document.body).addClassName('with-help');
    },
    close : function() {
      $(document.body).removeClassName('with-help');
  },
    request : function(url, callback) {
      new Ajax.Updater('help', url, {
        method: 'get',
        onComplete: callback.bind(this)
      });
    }

};
```

刚刚建的request()方法，参数是一个URL和一个回调函数，当Ajax请求成功完成的时候就会调用给定的回调函数。方法里有一个Ajax.Updater调用，它会更新ID为help的元素的内容，也就是更新我们的帮助栏。Ajax.Updater调用里还指明传入的回调函数将在调用完成时执行。注意这里用了Prototype的bind()方法，保证回调函数内部的this关键字指向Help控制器对象。

当单击一个帮助键接，我们希望触发请求去获取指定的URL，而当请求取得帮助内容之后，我们希望显示边栏。现在就把这两步动作变成控制器里的一个新方法：

```
Help = {
    openWith : function(url) {
      this.request(url, function() {
        if ($(document.body).hasClassName('with-help') == false) this.open();
      });
    },
    open : function() {
      $(document.body).addClassName('with-help');
    },
    close : function() {
      $(document.body).removeClassName('with-help');
    },
    request : function(url, callback) {
      new Ajax.Updater('help', url, {
        method: 'get',
        onComplete: callback.bind(this)
      });
    }
};
```

新增的openWith()方法正是我们需要的。这样一来控制器对象的架子就基本搭好了。下一步要把它和页面上的帮助连接联系起来，Low Pro恰能助我们一臂之力。

9.6.2 添加行为

可以把LowPro的Event.addBehavior()方法当作CSS的等价物，只不过它所处理的不是样

式而是行为。实际上Event.addBehavior()的用法也很像CSS，它用了一种CSS选择符的扩展形式去选取元素及事件，将行为应用到元素和事件上。下面是一些典型的用法：

```
Event.addBehavior({
  'a.product:click' : function() {
    // 当类名为'product'的a元素被单击，函数内的代码就会执行
  },
  'div.description:mouseover' : function() {
    // 当鼠标移到类名为'description'的div上面时，此函数就会执行
  }
});
```

可以任意调用Event.addBehavior()，次数不限，它会自动将众多行为妥善应用到相应元素上。而且，它会默认在每次Ajax请求返回之后重新应用各行为，确保新元素也有适当的行为。将以下代码放入application.js，连接帮助控制器与页面上的帮助链接：

```
Event.addBehavior({
  'a[rel=help]:click' : function() {
    Help.openWith(this.href);
    return false;
  }
});
```

这里用了一个属性选择符去选取所有rel属性值为help的<a>元素，然后在单击<a>元素时触发Help.openWith()，把链接的href属性传递给Help.openWith()。与普通的事件处理器相同，返回false将阻止链接的默认行为，因此不会跳转到帮助页面

是时候测试一下了，让我们更新一下应用布局，把application.js加入页面：

```
<%= javascript_include_tag 'prototype', 'lowpro', 'moofx', 'application' %>
```

单击任意一个帮助链接，帮助的内容就会显示在边栏里。不过我们离目标还有一段距离。目前返回的帮助内容里包含着整个HTML页面，有一些多余的部分，我们希望只把中间的内容部分注入到帮助栏。因此要回到Rails的帮助控制器（app/controllers/helpcontroller.rb）做一些调整，如果页面是通过Ajax请求的，就让它用另一个布局渲染帮助内容：

```
class HelpController < ApplicationController

  def show
    template = '/help/' + params[:path].join('/')
    if request.xhr?
      render :template => template, :layout => 'help_sidebar'
    else
      render :template => template, :layout => 'help'
    end
  end
```

```
end
```

代码中用了一行条件语句去判断请求是否来自XMLHttpRequest（也就是Ajax），随后返回同样的模板，只不过布局换成了helpsidebar。这是Rails的例子，所以用了request.xhr?判断请求是否来自Ajax；如果你用的是其他平台，实现同样的功能并不困难。在request.xhr?方法内部，它仅仅检查HTTP首部X-Requested-With是否为'XMLHttpRequest'。Prototype会默认在所有Ajax请求中添加这个HTTP首部。

helpsidebar布局非常简单，它并不是完整的HTML文档，只是一个小片段。这就是局部更新页面的一般方法。还要在布局中添加一个关闭链接，以便用户关闭帮助栏。

```
<p id="close_help"><a href="">X</a></p>

<%= yield %>
```

再次测试，效果应该好多了，打开边栏也不会弄乱页面布局。关闭链接还要下一点功夫才能起作用，在Event.addBehavior()块里添加一条规则：

```
Event.addBehavior({
  'a[rel=help]:click' : function() {
    Help.openWith(this.href);
    return false;
  },
  '#close_help a:click' : function() {
    Help.close();
    return false;
  }
});
```

9.6.3 实现加载提示

虽然功能已经完成，但如果能在帮助内容加载时间比较长的时候给用户一点提示，那就更好了。有一种简单漂亮的实现方式，就是建立一个全局的加载提示，自动响应所有的Ajax请求。在Prototype的协助之下，实现加载提示非常简单。

由于页面上只会有一个全局的加载提示，所以可以用一个单件（singleton）对象来表示它，名称自然就是Loader了。Loader需要封装三个基本的函数——初始化（包括创建加载提示本身的元素），显示加载提示，隐藏加载提示：

```
Loader = {
  initialize: function(parent) {
    this.loader =
      $img({ src: 'images/loader.gif', alt: 'Loading...', id: 'loader' });
    parent.appendChild(this.loader);
    this.hide();
```

```
    Ajax.Responders.register({
      onCreate: function() {
        Loader.show();
      },
      onComplete: function() {
        Loader.hide();
      }
    });
  },
  show: function() {
    this.loader.show();
  },
  hide: function() {
    this.loader.hide();
  }
};
```

大部分工作落在initialize()方法的肩膀上。首先，用Low Pro的DOM生成器创建提示器本身的元素。Low Pro的DOM生成器不但提供了建立DOM结点结构的简便方法，它还会处理一些跨浏览器的问题。每一种HTML标签都有一个对应的$xxx()函数负责创建结点。如果把一个字面量对象作为第一个参数传给$xxx()函数，字面量里的键/值对就会变成HTML元素的属性和属性值。如果传递的参数不止一个，那么多出来的参数将成为新结点下面的子结点。上面的例子仅仅创建了单个<img>标签，比较简单。从下面的例子可以看出一个比较复杂的结点结构是怎样建立的：

```
var product = $div({ 'class' : 'product' },
  $h2('Sprocket 47'),
  $p({ 'class' : 'description' }, 'The worlds best sprocket'),
  $a({ href : '/sprockets/74'}, 'Read more')
);
```

再回到Loader，创建完加载提示的图片元素，就把它插入成为传入的父元素的子结点。接下来initialize()方法利用Prototype的Ajax.Responders在需要时显示和隐藏提示图片。Ajax.Responders.register()的作用是注册全局的事件处理器，每当有Ajax请求发起或者完成的时候，相应的事件处理器就会执行。这种机制使实现全局的加载提示器变得非常简单，也非常精致。

编写完加载提示后就要把它绑定到页面上。再一次用到Event.addBehavior()：

```
Event.addBehavior({
  '#header' : function() {
    Loader.initialize(this);
  }
});
```

这里Event.addBehavior()的用法稍有不同。若CSS选择符中不包含事件类型，那么给定

的函数就会在DOM加载后立即执行。我们恰好利用这一点去初始化Loader对象，CSS选择符选取的元素成为加载提示器的父结点。我在main.css中添加了一点CSS，让加载提示总是显示在页面头部的右上角。这样加载提示就完成了。

使用Low Pro的时候，Event.addBehavior()成为粘合HTML文档与核心JavaScript代码的胶水，将JavaScript逻辑从文档中解耦出来。这样对维护有莫大的好处。尤其是在HTML发生变化的时候，与其在代码中翻查哪段代码影响哪个元素，用Event.addBehavior()的方式就只需要修改一下CSS选择符反映出文档结构的变化。

9.7 最后润色

帮助栏已经很接近完工，不过总是可以再做些工作令其更完美一些。也就是说我们打算添加一些动画和一点额外的功能。

9.7.1 用 Moo.fx 添加动画

本章开头已经粗略提过Moo.fx这个精简的效果库，现在可以试一试它的本领了。让我们把帮助栏改成平顺地滑进滑出的显示方式，别再突然地出现和消失。

从最本质上说，JavaScript中的动画无非是随着时间变化更动元素的若干属性。Moo.fx通过fx.Style构造器给了我们一个基本但丰富的接口。不过我们首先要从CSS着手，检查一下实现滑动效果需要变动哪些样式属性。牵涉到的样式规则如下：

```
body.with-help {
  margin-right: 350px;
}

body.with-help #help {
  background: #F4EEBC;
  border: 1px solid #000;
  border-color: #CCC #333 #333 #CCC;
  width: 320px;
  position: absolute;
  top: 0;
  right: 0;
  margin: 8px 30px;
  padding: 10px;
  overflow: hidden;
}
```

不难看出margin-right属性需要从初始值增大到350px才能给帮助栏留下足够的空间。同时要把帮助栏的width属性从0增大到320px，达到一种慢慢打开的效果。当然关闭帮助栏的时候还要把上述过程反过来执行一遍。别忘了还要照旧添加和删除with-help类名。

打开application.js，为执行动画建立一个效果对象。所有效果都可以放在Help对象的fx属性里面，而且所有效果都只需要创建一次，创建时机是DOM准备好的时刻，因此要用到Low Pro的Event.onReady()方法：

```
Event.onReady(function() {
  Help.fx = {
    openHelp: new fx.Style('help', 'width', {
      onStart : function() {
        $(document.body).addClassName('with-help');
      }
    }),
    closeHelp: new fx.Style('help', 'width', {
      onComplete : function() {
        $(document.body).removeClassName('with-help');
      }
    }),
    slideBody: new fx.Style(document.body, 'margin-right')
  };
});
```

上面的代码定义了三个效果。第一个是openHelp，操作help元素，也就是帮助栏的width属性。我们用了它的onStart回调函数给页面的body元素添加with-help类名。第二个closeHelp与第一个效果类似，差别在当效果结束时用了onComplete回调函数删除with-help类名。最后是slideBody，它操作document.body的margin-right属性。接下来修改Help对象的打开和关闭方法，用上这几个效果：

```
Help = {
  SIDEBAR_WIDTH: 350,
  SIDEBAR_MARGIN: 30,
  openWith : function(url) {
    this.request(url, function() {
      if ($(document.body).hasClassName('with-help') == false) this.open();
    });
  },
  open : function() {
    Help.fx.openHelp.custom(0, this.SIDEBAR_WIDTH - this.SIDEBAR_MARGIN);
    Help.fx.slideBody.custom(this.SIDEBAR_MARGIN , this.SIDEBAR_WIDTH);
  },
  close : function() {
    Help.fx.closeHelp.custom(this.SIDEBAR_WIDTH - this.SIDEBAR_MARGIN , 0);
    Help.fx.slideBody.custom(this.SIDEBAR_WIDTH , this.SIDEBAR_MARGIN );
  },
  request : function(url, callback) {
    new Ajax.Updater('help', url, {
      method: 'get',
      onComplete: callback.bind(this)
```

```
    });
  }
  };
```

现在打开和关闭的时候，通过效果对象的custom方法传递动画开始和结束时的属性值，执行动画。

9.7.2 实现边栏内锚点

你可能注意到在纯HTML版的帮助系统里，单击表单里“Sprocket ID”项目的帮助，会直接定位到帮助页面中的相关段落。这是由于我们用了标准的HTML页面锚点来确保浏览器滚动到相应段落：

```
<a href="/help/sprocket#sprocketid" rel="help">?</a>
```

而当我们实现增强版的时候，拦截了浏览器的标准行为，也就失去了这种页面内的定位效果。如果能修改一下脚本，把用户直接带到帮助栏里的相关部分就更好了。还可以更进一步把相关部分突出显示，让用户更容易找到。

解决DOM脚本编程问题有一种常见思路，就是尽可能利用HTML中包含的信息。前面我们利用了帮助链接的href属性，告知系统装载哪块内容。现在我们要更细致地检查href属性包含的信息，从中取出锚点的部分，那也就是我们要滚动到的段落的ID。修改一下Help对象的openWith()方法：

```
  Help = {
    openWith : function(url) {
      var urlParts = url.split('#');
      var path = urlParts[0], anchor = urlParts[1];

      this.request(url, function() {
        if ($(document.body).hasClassName('with-help') == false) this.open();

        if (anchor && anchorEl = $(anchor)) {
          anchorEl.scrollTo();
          anchorEl.addClassName('highlighted');
        }
      });
  },
  ...
```

首先用split()把URL的路径和锚点部分划分开。请求页面的动作和原来一样，但这次的回调函数有些不同。如果URL里有锚点部分，并且锚点指向的元素存在，那么就用Prototype的scrollTo()方法把浏览器窗口滚动到该元素的位置。这基本上是模仿了浏览器处理锚点时的默认行为。最后在元素上添加highlighted类名，给锚点指向的元素增加一点样式。

9.8　回顾

本章在Prototype、Low Pro，还有Rails的助力下，以相当简单的办法取得了很大的成果。让我们再回顾一下在帮助栏实现过程中所作的决策，仔细评估每一项决策的利益。

9.8.1　用符合语义的 HTML 奠定坚实的基础

JavaScript给了开发者几乎无限制的力量去操纵HTML元素的外观和行为，常常导致JavaScript开发者一头扎进脚本编程，仅仅将HTML当作外围的辅助。然而，只有把如何用静态HTML最好地表现页面中的信息作为思路的起点，才能充分地利用浏览器的内建行为，不但让你写更少的代码，而且保证了即使没有JavaScript可用的用户，也能最大限度地正常使用我们的应用。

实现功能时，先用静态HTML完成一个能正常工作的版本，这是值得提倡的一般原则。且勿担心可用性和外部的观感，功能正常就好。还要注意尽量提高HTML的语义，因为内容的语义越丰富，脚本的立足之处就越多。

这一章在编写任何一行JavaScript代码以前，就先确保了静态HTML版本运行无误。这就给后续建设打下了坚实的基础，保证帮助内容在各种情况下都能访问，能被搜索引擎索引、方便打印、方便收藏。有了这一切，下一步编写JavaScript就只是在这个顺利运转的系统上锦上添花。我们一直说的渐进增强就是一个优秀的例子。

9.8.2　恰当使用 HTML、CSS、JavaScript

HTML用于内容和结构，CSS用于呈现，JavaScript用于（与浏览器）交互。它们都有各自的专门用途，也最擅长完成本行工作。然而我们时常见到一些开发者的坏习惯，比如让HTML承担部分呈现职责（为了而用`<strong>`，内联的样式属性，等等），甚至让CSS承担部分交互（纯CSS的下拉菜单就是一例），而最大的诱惑藏在JavaScript之中。

JavaScript是浏览器里的一个特殊角色，摆布呈现与内容对它来说只是举手之劳。即便如此，一般不提倡直接操作样式属性以及向页面中添加内容。如果要通过JavaScript改变某元素的呈现状态，应该通过添加样式类名的方式实行（就像前面例子里添加`with-help`类名一样）。类似地，与其通过脚本生成大量的HTML，最好是一开始就把它们放在文档里。只有动画是一般接受的例外，因为它必须要随时间变化更动样式属性。

遵循这种工作方式的主要好处是，改变应用的外部观感完全不需要触动JavaScript。如果想改一下选中段落突出显示的样子，只要在设计软件里打开CSS文件，修改突出显示的样式规则。类似地，如果所有内容都包含在HTML里，那么一名不懂技术的团队成员也能安全地修改内容，因为他根本不会触碰任何JavaScript代码。最后，如果所有内容都在HTML里，那么你可以担保用户能看到全部内容，即使JavaScript出了什么问题。

9.8.3 用 CSS 选择符充当应用的胶水

选择元素、应用样式属性的工作CSS选择符完成得非常出色，而且现在主流JavaScript库中的实现十分可靠，速度越来越快，除了浏览器内建的CSS功能，对CSS标准的支持也越来越完善。这意味着CSS选择符现在不但被用来粘合样式与文档，还被用来粘合JavaScript行为。Low Pro是其中一个自动化了行为粘合过程的JavaScript框架，给了我们一个本质上是样式表的“行为表”。

用这样的方式装配JavaScript，主要好处是得以解耦应用代码与文档本身。在本章的项目里，我们用LowPro把程序里的`Help.openWith()`调用连接到的`rel="help"`链接。如果你打算做些修改（比方把目标改成`class="help"`的链接），一点都不麻烦。简直连团队里不懂编程的的设计师都能完成这项任务。这个帮助系统比较简单，如果是更大型的项目，可维护性方面的改善会更加显而易见。可以把一个非常复杂的JavaScript拆成若干行为，然后通过`Event.addBehavior()`粘接到文档上。如果将来HTML文档的结构变了，调整脚本的工作量就微不足道了。

9.8.4 对 Ajax 来说，简单是最好的

JavaScript与服务器的通信方式有很多，首先是Ajax里的“X”：XML。然后出现了JSON、RJS和简单HTML等等格式，其他非主流的技术更是不可胜数。按照经验法则，应该尽量选择能完成任务的最简单的方案。很多时候连JSON都算大材小用，只需要简单地发出请求，返回一段更新过的HTML，替换掉文档中相应的部分就可以了。Prototype的`Ajax.Updater`使这类操作极其简单。

在这个项目里，`Ajax.Updater`起了很大的作用。没有必要一面把Ajax响应打包成JSON，一面又写客户端的代码去解包。因此我们直接把返回的HTML放进边栏的`<div>`。

保持Ajax通信简单有几大好处。首先，因为每台机器上，每一个用户都需要下载和运行JavaScript代码，所以代码少一些总是好的。如果传递一些HTML片段就够了，为什么要费劲写一堆代码解包XML和JSON呢？第二，浏览器运行JavaScript从本质上说就是十分缓慢而且靠不住的，代码的工作越少，正常运行的机率就越大，应用的响应能力也就越强。当然也有很多场景JSON和XML不可或缺，但总体来说能找到更简单的办法完成任务——请力争找到更简单的方案。

9.9 小结

通过本章中实现帮助栏功能的整个过程，希望让读者看到，渐进增强的实现手法一点也不比旧的、侵入式的脚本编程难。破坏浏览器功能、降低可访问性、在手机上不能访问、在防火墙后不能访问等问题在Low Pro等工具的帮助下，得到功能强劲的应用不再需要付出如此代价。常有人误以为渐进增强比无侵入的实现手法更困难，甚至以为大多数应用都不可能实现渐进增强，我敢肯定你看完这章之后会有不同的想法。渐进增强应该成为你进行DOM脚本编程的默认方式。

9.10　源代码

Rails会为应用生成大量的模板代码，不过对于这个例子来说，只需要关心其中几个文件。下面就是需要注意的几个文件，列在这里方便你参考。

代码清单9-1　应用的布局（app/vievis/layouts/application.rhtml）

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html>
  <head>
    <meta http-equiv="Content-type" content="text/html; charset=utf-8">
    <title>Pro DOM Scripting with Ajax, APIs and Libraries Chapter 9</title>
    <%= stylesheet_link_tag 'main' %>
    <%= javascript_include_tag 'prototype', 'lowpro', 'moofx', 'application' %>
  </head>

  <body>
    <div id="content">
      <div id="header">
        <h1>Examplr Beta</h1>
      </div>

    <%= yield %>
    </div>

    <div id="help"></div>
  </body>
</html>
```

代码清单9-2　完整页面的布局（app/views/layouts/help.rhtml）

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html>
  <head>
    <meta http-equiv="Content-type" content="text/html; charset=utf-8">
    <title>Pro DOM Scripting with Ajax, APIs and Libraries Chapter 9</title>
    <%= stylesheet_link_tag 'main' %>
  </head>

  <body>
    <div id="content">
      <div id="header">
        <h1>Examplr Help</h1>
      </div>

      <%= yield %>
```

```
      </div>
    </body>
</html>
```

代码清单9-3 帮助边栏的布局（app/views/layouts/help_sidebar.rhtml）

```
<p id="close_help"><a href="">X</a></p>

<%= yield %>
```

代码清单9-4 表单页面（app/views/main/index.rhtml）

```
<h2>Add A Sprocket <a href="/help/sprocket" rel="help">?</a></h2>

<form action="/nowhere" method="post">
  <fieldset>
    <p><label for="name">Name</label> <input name="name" id="name" /></p>
    <p><label for="spid">Sprocket ID <a href="/help/sprocket#sprocketid"
rel="help">?</a></label> <input name="spid" id="spid" /></p>
    <p><label for="desc">Description</label> <textarea name="desc"
id="desc"></textarea></p>
    <p><label for="tr">Tacion Rating <a href="/help/tacion"
rel="help">?</a></label> <input name="tr" id="tr" /></p>
    <p class="check"><label for="xx75">XX-75 Approved <a href="/help/xx75"
rel="help">?</a></label> <input name="xx75" id="xx75" type="checkbox" /></p>
    <p class="submit"><input type="submit" name="submit" value="Add Sprocket"
/></p>
  </fieldset>
</form>
```

代码清单9-5 应用的CSS文件（public/stylesheets/main.css）

```
body {
  background: #999;
  padding: 0 30px;
  font-family: helvetica, arial, sans-serif;
}

#content {
  background: #FFF;
  border: 1px solid #FFF;
  border-color: #CCC #333 #333 #CCC;
}

#header {
  background: #5D8ED3;
  padding: 1em;
  color: #FFF;
  font-family: georgia, serif;
  position: relative;
```

```
}

#content h2, #content form, #content p, #content h3 {
  margin: 1em 1em;
}

fieldset {
  border: 0;
  width: 50%;
}

#content h2 {
  font-family: georgia, serif;
  border-bottom: 1px solid #5D8ED3;
  padding-bottom: 0.5em;
}

label {
  display: block;
}

input, textarea {
  width: 99%;
}

textarea {
  height: 7em;
}

p.check label {
  display: inline;
}

p.check input, p.submit input {
  width: auto;
}

fieldset p {
  padding: 0.7em 0;
  margin: 1px;
}

p.submit input {
  font-size: 1.3em;
}

#help {
  display: none;
}
```

```
body.with-help {
  margin-right: 350px;
}

body.with-help #help {
  display: block;
  background: #F4EEBC;
  border: 1px solid #000;
  border-color: #CCC #333 #333 #CCC;
  width: 320px;
  position: absolute;
  top: 0;
  right: 0;
  margin: 8px 30px;
  padding: 10px;
  overflow: hidden;
}

#help p, #help h2 {
  width: 300px;
}

#help p#close_help {
  position: absolute;
  top: 0;
  right: 15px;
  width: auto;
}

#close_help a {
  color: black;
  text-decoration: none;
  font-weight: bold;
}

#loader {
  position: absolute;
  top: 10px;
  right: 10px;
}
```

代码清单9-6 应用的JavaScript文件（public/javascripts/application.js）

```
Event.addBehavior({
  'a[rel=help]:click' : function() {
    Help.openWith(this.href);
    return false;
},
```

```
    '#close_help a:click' : function() {
      Help.close();
      return false;
    },
    '#header' : function() {
      Loader.initialize(this);
    }
  });
  Help = {
    openWith : function(url) {
      var urlParts = url.split('#');
      var path = urlParts[0], anchor = urlParts[1];

      this.request(url, function() {
        if ($(document.body).hasClassName('with-help') == false) this.open();

        if (anchor && (anchorEl = $(anchor))) {
          anchorEl.scrollTo();
          anchorEl.addClassName('highlighted');
        }
      });
    },
    open : function() {
      Help.fx.openHelp.custom(0, 320);
      Help.fx.slideBody.custom(30, 350);
    },
    close : function() {
      Help.fx.closeHelp.custom(320, 0);
      Help.fx.slideBody.custom(350, 30);
    },
    request : function(url, callback) {
      new Ajax.Updater('help', url, {
        method: 'get',
        onComplete: callback.bind(this)
      });
    }
  };

  Event.onReady(function() {
    Help.fx = {
      openHelp: new fx.Style('help', 'width', {
        onStart : function() {
          $(document.body).addClassName('with-help');
        }
      }),
      closeHelp: new fx.Style('help', 'width', {
        onComplete : function() {
          $(document.body).removeClassName('with-help');
        }
```

```
    }),
    slideBody: new fx.Style(document.body, 'margin-right')
  };
});
Loader = {
  initialize: function(parent) {
    this.loader = $img({ src: 'images/loader.gif', alt: 'Loading...', id:
  'loader' });
    parent.appendChild(this.loader);
    this.hide();
    Ajax.Responders.register({
      onCreate: function() {
        Loader.show();
      },
      onComplete: function() {
        Loader.hide();
      }
    });
  },
  show: function() {
    this.loader.show();
  },
  hide: function() {
    this.loader.hide();
  }
};
```

版 权 声 明

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

Java 一览无余：**Java** 视频教程 | **Java SE** | **Java EE**

.Net 技术精品资料下载汇总：**ASP.NET** 篇

.Net 技术精品资料下载汇总：**C#**语言篇

.Net 技术精品资料下载汇总：**VB.NET** 篇

撼世出击：**C/C++**编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 **Ruby**、**Ruby on Rails** 精品电子书等学习资料下载

数据库精品学习资源汇总：**MySQL** 篇 | **SQL Server** 篇 | **Oracle** 篇

最强 **HTML/xHTML**、**CSS** 精品学习资料下载汇总

最新 **JavaScript**、**Ajax** 典藏级学习资料下载分类汇总

网络最强 **PHP** 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

经典 **LinuxCBT** 视频教程系列 **Linux** 快速学习视频教程一帖通

天罗地网：精品 **Linux** 学习资料大收集(电子书+视频教程) **Linux** 参考资源大系

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

FreeBSD/OpenBSD/NetBSD 精品学习资源索引 含书籍+视频

Solaris/OpenSolaris 电子书、视频等精华资料下载索引

“帮助你深入理解JavaScript的绝佳资源。”

——Nate Klaiber，资深Web技术专家

Accelerated DOM Scripting with Ajax, APIs, and Libraries

JavaScript捷径教程

你是否对Web开发和设计已经略知一二，但是JavaScript的灵活多变仍然时常让你感到困惑？面对众多的JavaScript库，你是否无所适从？

没关系，这部由几位世界顶尖专家合著的好书将帮助你度过难关。书中在回顾了必要的HTML、CSS和JavaScript的基础知识之后，直入主题，深入剖析了学习和使用JavaScript的过程中最难掌握的地方——面向对象概念、闭包和事件处理等，并强调了不唐突、注重可用性和渐进增强（progressive enhancement）等现代Web开发理念。在此基础上，本书还讲解了jQuery、Prototype、Mootools等最流行的JavaScript库的背后机制，探讨了如何在不同场合中实际选择和使用这些库。最后，实现了一些很酷的视觉特效、表单验证和两个完整的案例。

Jonathan Snook 著名Web技术专家。W3C CSS工作组顾问，Digital Web Magazine等著名媒体的专栏作家。他曾荣获英国Web设计大奖，服务的客户包括苹果、红牛和FedEx等国际大公司。

Aaron Gustafson 著名Web技术专家。Web Standards Project成员，著名Web开发媒体A List Apart编辑。

Stuart Langridge 著名Web技术专家。Web Standards Project成员。*DHTML Utopia*一书的作者。

Dan Webb 著名Web技术专家。Prototype核心开发成员。

图灵Web开发图书阅读路线图

Apress®

本书相关信息请访问：图灵网站 http://www.turingbook.com

读者/作者热线：(010)88593802

反馈/投稿/推荐信箱：contact@turingbook.com

上架建议 计算机/网络开发/程序设计

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-19259-2

ISBN 978-7-115-19259-2/TP

定价：35.00 元