TURING 图灵程序设计丛书 Web开发系列

Apress®

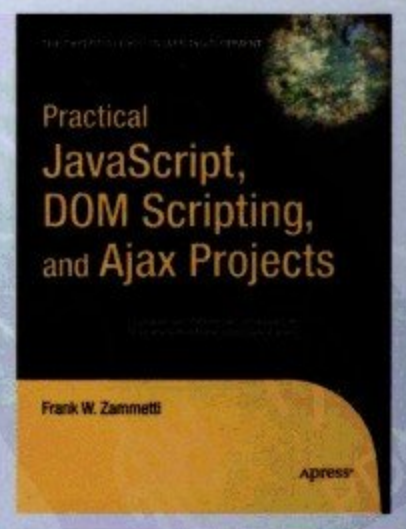
Practical JavaScript, DOM Scripting, and Ajax Projects

JavaScript实战

[美] Frank W. Zammetti 著

张韡珏 译

- 10个具体项目生动精彩
- 活学活用流行的JavaScript库
- 超级Web应用，构建并不困难



人民邮电出版社
POSTS & TELECOM PRESS

TURING 图灵程序设计丛书 Web开发系列

Practical JavaScript, DOM Scripting, and Ajax Projects

JavaScript实战

[美] Frank W. Zammetti 著

张晶珏 译

人民邮电出版社

北 京

图书在版编目（CIP）数据

JavaScript 实战 /（美）扎米特（Zammetti, F. W.）著；张晶珏译 . —北京：人民邮电出版社，2009.8
（图灵程序设计丛书）
书名原文：Practical JavaScript, DOM Scripting, and Ajax Projects
ISBN 978-7-115-18915-8

Ⅰ. J… Ⅱ. ①扎…②张… Ⅲ. JAVA 语言－程序设计
Ⅳ. TP312

中国版本图书馆CIP数据核字（2008）第150598号

内 容 提 要

本书是一部讲述 JavaScript 实战项目开发的精彩著作，由两部分组成。第一部分讨论一般性的 JavaScript 主题，包括 JavaScript 的简史、好的编码习惯、调试技巧和工具等；第二部分是 10 个具体项目，每一章都会提出一个不同的应用，分析其内在的工作原理，然后提供能够提高读者技巧的练习。这些项目的范围从通用的小工具（可执行的计算器）到当代的各种创意（混搭），再到单纯的趣味性（JavaScript 游戏）。

本书非常适合 Web 开发人员阅读和参考。

图灵程序设计丛书
JavaScript实战

◆ 著 [美] Frank W. Zammetti
译 张晶珏
责任编辑 傅志红
执行编辑 杨 爽

◆ 人民邮电出版社出版发行
邮编 100061 电子函件
网址 http://www.ptpress.com.cn
北京顺义振华印刷厂印刷

◆ 开本：800×1000 1/16
印张：28.25
字数：756千字 2009年8月第1版
印数：1－3 000册 2009年8月北京第1次印刷

著作权合同登记号 图字：01-2007-4258号
ISBN 978-7-115-18915-8/TP

定价：59.00元

读者服务热线：(010)51095186 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

版 权 声 明

译 者 序

“JavaScript，就是那种小时候长得很丑，长大了却谁都想要的孩子。”

诞生初期，由于很多所谓的“资深”程序员的滥用，让这个孩子饱受质疑，直到前几年DOM技术开始崛起，JavaScript才逐渐恢复了曾经的兴盛。并且，这时的JavaScript更多了一份成熟，少了一缕稚气。

JavaScript虽然已经被当作玩具使用了很多年，但是藏在它那让人迷惑的简单外表下面的，却是无法忽视的强大语言特性。

本书的作者从面向对象的JavaScript编程技巧开始，用诙谐的语言，深入浅出地讲解，示范了JavaScript和DOM技术的经典概念，并讨论了一些较新的不唐突的JavaScript，与之绑定的是柔性衰减的概念以及这些概念为什么在当今仍然被广泛使用。

本书的重头戏就是第二部分大量的项目实践，在修改和研习前人经典代码的基础上开发属于自己的JavaScript应用，才是程序员快速提高的捷径。

作者在所使用的实例应用中，包罗了时下流行的各大JavaScript代码库（有些或者应该叫做代码框架、窗口小部件集合等）。每一个项目都使用了不同的代码库，其中包括如下几个。

- 基于JavaScript语言本身进行扩展的Prototype。
- 成长最迅速、最受欢迎、功能强大的Dojo（作者形容Dojo有着很高的曝光率，并且都是正面报道）。
- 独特的、可以用来生成JavaScript的标签库JSTags。
- 特效出众的、基于Prototype的script.aculo.us。
- 提供大量干净简单的UI窗口小部件且具有完美文档的YUI（Yahoo! User Interface）。
- 展示了很多小而有趣的功能的Mochikit。
- 专注于Ajax、拖放和行为效果的、用于RIA的Rico。
- 非常轻量化、模块化的Mootools。

总之，这本书并不像一本教材，更确切一点，畅读之后，你会觉得它就是一位讲课生动的老师，带领你遨游JavaScript海洋，把你推向技术风浪的前沿，同时又给了你一个功能完备的冲浪板，接下来你就可以在风口浪尖上随意遨游了。

最后，我要特别感谢在翻译本书时，自始至终给予很大帮助的老师Laser（何伟平），他渊博的知识使我少走了许多弯路。还要感谢老爸老妈的照顾和鼓励，以及小狗豆豆在我翻译时停止吼叫的“好”习惯。

当然最重要的是感谢各位读者购买此书！希望阅读的过程可以为你们带来更多的享受。

前　言

JavaScript迅速成为Web开发的最重要的话题之一，它是任何一个真正的Web开发人员都不可缺少的东西。随着Ajax（会在本书中涉及）的兴起，JavaScript迅速地从用来改进网站的小技术进化到开发真正的、专业质量的应用程序的支柱。它不再是一个边缘人，已经成为关注焦点了。

在网上有很多介绍JavaScript的书和大量关于如何使用它的文章，这些都能很好地帮助你。比较难找到的是真实有效的例子。当然了，你可以找到大量简单的、虚构性的例子，但是想找到全面完整的实战应用就比较难了。很多开发人员都可以在分解、修改和改善一个实际应用的代码的过程中得到提高。这就是我撰写本书的目的：填补这个空白。

在本书里，你会发现有两章是关于一般性的JavaScript主题的，包括JavaScript的简史、好的编码习惯、调试技巧和工具等。然后，就是10章的具体项目。每一章都会提出一个不同的应用，分析其内在的工作原理，然后提供一些能够提高读者技巧且使读者深入理解所读知识的练习。这些项目的范围从通用的小工具（可扩展的计算器）到最时髦的各种创意（混搭），再到单纯的趣味程序（一个JavaScript游戏）。

在这个过程中，你会学习很多主题，包括调试技术、各种JavaScript库和一些独一无二的有用的编码技巧。我相信你也会发现这是一本很有趣的书，实际上，我一开始就建议找出书里面散布在四处的流行文化的影子（大多都有脚注，但不是全部）。[①]从这个意义上来说，我试图把本书做得像*Gilmore Girls*[②]的剧集一样。

好了，闲话少说。演出开始！

本书概述

本书分成两个主要部分。第一部分是“向我的小朋友JavaScript问好”，包含两章内容。

- 第1章介绍JavaScript的简史，从混沌初开到当前的广为接受。
- 第2章谈的是现代“专业”的JavaScript开发人员使用的技巧和方法。

第二部分是“项目”，包含10章内容。

- 第3章开始第一个项目：一个可扩展的、收集了若干个工具函数的包。
- 第4章开发了一个可扩展的JavaScript计算器，并且介绍了第一个JavaScript库——Rico。
- 第5章介绍了混搭（mashup）的概念，如今最热门的话题，使用非常火爆的script.aculo.us库开发了一个现实的例子。

---

① 很可惜，美国文化和中国文化有些区别，但是本书中大量电影、电视、书籍的引用的确是相当有趣的东西。——译者注

② 2000年到2006年播出的热门美剧，讲一个漂亮单身妈妈和她的漂亮女儿的生活。——编者注

- 第6章使用Dojo库解决了一个经常在JavaScript开发中面临的问题：客户端持久存储。
- 第7章讨论了一个Java Web Parts项目里非常有用的JSDigester组件，它允许你分析XML并且从中创建JavaScript对象，而不用自己编写枯燥的代码。
- 第8章开发了一个可扩展的验证框架，用来在客户端以纯声明的方式来实现客户端验证。
- 第9章介绍了YUI（Yahoo!用户界面）库，并且用它创建了一个很便利的通讯录应用。
- 第10章使用MochiKit库为电子商务应用创建了一个可拖放的购物车。
- 第11章是我们做的好玩的东西：一个JavaScript游戏。不是简单的俄罗斯方块或者是类似的小游戏，而是复杂得多的游戏。
- 第12章是我们深入了解Ajax的一章，Ajax可能是近年来JavaScript获得全新的重要地位的最主要原因，使用的是相对新的Mootools库。

获取本书的源代码

本书的所有代码都可以从Apress网站的Source Code中获取。[①] 实际上，因为本书比较特殊，所以在进入第3章之前，绝对需要下载源代码。访问http://www.apress.com，点击Source Code链接，然后找列表中的*Practical JavaScript, DOM Scripting, and Ajax Projects*。从这本书的主页上，你就可以下载源代码的压缩文件。源代码是按照章组织的。

获取本书的更新

写书是个繁重的劳动，比很多人想象的要重得多！与我私下里和朋友吹的正相反，我其实并不完美，我和所有人一样会犯错误。当然，这本书里我可没有犯错，绝对没有。

咳，又吹上了。……

要是在本书中发现任何错误，我先在这儿道歉了。先请放心我们每个人都尽量让它没有错误，但是我们还是要面对现实。我们以前都读过技术书，而且都知道现实并不完美！抱歉！抱歉！抱歉！

在Apress的网站（http://www.apress.com）上有一个本书的勘误表，还有关于如何提交你发现的错误的方法。做这些事通常需要一些心灵感应，不过我知道Windows下一个版本内置了这个功能，所以兄弟们歇着，放松些。

联系作者

我很乐意听到你对本书的内容和源代码样例的问题和评论。请随意给我直接发邮件fzammetti@omnytex.com（垃圾邮件会被防范软件击落并处理掉）。我会尽快回复你的询问，但也请记住：我有自己的生活（噢不，还是说没有吧），所以可能不会那么快回复。

最后，也是最重要的，感谢你购买此书！我携妻子与孩子们感谢你，相信我孩子的牙医、我的狗的兽医和我的房顶材料承包商都会感谢你！

致谢

很多人为此书的问世提供了各种各样的帮助，而且有些人甚至都不知道自己提供了帮助。我在此

① 也可以从图灵网站 www.turingbook.com 本书网页免费注册下载。——编者注

试图感谢所有提供过帮助的人，不过也许我记得不全，那就先致歉了！

首先，感谢Apress的所有同事，感谢你们为本书的出版所做出的贡献。这是我第二次与诸君合作，就像第一次合作一样愉快。Chris、Matt、Tracy、Marilyn、Laura、Tina和所有其他的人，感谢你们！

感谢Herman van Rosmalen，我的Java Web Parts项目（http://javawebparts.sourceforge.net）的主要合作者以及本书的技术审稿人。我知道你为了保持本书的正确性花了无数的时间，实在是无以为谢！现在，让我们回到JWP的工作中去吧！

还得好好谢谢Anthony Volpe，他为此书做了插图。我和他已经是10年左右的朋友了，合作过很多项目，包括3个PocketPC游戏（到http://www.omnytex.com看看）以及一些Flash游戏（http://www.planet-volpe.com/crackhead）和一些Web卡通（http://www.planetvolpe.com/du）。他是个优秀的艺术家，我相信看过本书你就知道，他是一个极富创造力的人，一个让我自豪的朋友。

还得感谢那些创建了本书用到的一些库的作者们，包括开发Dojo的弟兄们，Mootools小组的Sam Stephenson（Prototype）、Aaron Newton、Christophe Beyls和Valerio Proietti，MochiKit的Bob Ippolito，所有YUI开发人员以及所有script.aculo.us和Rico的开发人员。

最后，还得感谢每个购买此书的人！我真心希望你阅读此书时获得的乐趣和我撰写此书时获得的乐趣一样多！我也希望本书值得花费你辛苦挣来的钱，并且是有价值的、有趣的。

我几乎可以肯定自己漏掉了一些人，干脆让我感谢全世界吧。

说完这些，让我们看一些代码吧。

目　录

Part 1 第一部分

向我的小朋友 JavaScript 问好

最近“吃了”什么好书没？

——在《星际迷航：下一代》“Deja-Q”一集中，Q（对沃夫说）

因特网？那玩意儿还没完蛋？

——动画片《Simpson一家》中Homer Simpson

如今的编程，是下面两方面的竞争：一边是软件工程师奋力构建更大、更好的傻瓜型软件，另外一边是宇宙在努力制造更大更傻的傻瓜。就目前而言，宇宙这边领先。

——Rich Cook（美国科幻小说家）

开头的90%的代码只占10%的开发时间。剩下10%的代码占了其余90%的开发时间。

——Tom Cargill（著名C++专家）

只有两种编程语言：一种是人们天天报怨的，另外一种是没人使用的。

——Bjarne Stroustrup（C++之父）

只有两种行业把客户（customer）称作“用户”（user）。

——Edward Tufte（信息设计大师）

第1章 JavaScript简史

我希望史蒂芬·霍金先生不介意我借用他的著作的标题作为本章的标题[①]！就像他的书《时间简史》中说的那样，我们也将开始在另外一个宇宙里旅行，从它刚出现到现在。

在本章中，我们会探索JavaScript的起源。但这不仅仅是一节历史课，像霍金先生一样，我们将深入探索，给出隐藏在表象后的东西。在这个过程中，你会了解JavaScript早期开发中的固有问题以及这些问题中的大部分是如何得到解决的。在本章的结尾，你会充分了解需要避开的那些陷阱以及如何克服它们（相关的知识将在随后几章中介绍）。好，让我们准备开始历险，并让霍金先生为我们骄傲！

1.1 JavaScript的问世

1995年，Web尚处于婴儿期，可以说，绝大多数计算机用户还不能告诉你网站到底为何物，而且大部分开发人员在查阅和学习相关资料之前，也很难构建出一个网站来。微软当时也刚刚意识到，因特网将变得越来越重要。Google在当时也只是*Little Rascals*[②]中一个杜撰的词。

那时候，Netscape是老大，因为大多数人使用其Navigator浏览器作为登录因特网的主要工具。Java applet作为当时的一个新亮点，受到人们的青睐和重视。然而，Java对大多数开发者来说并不像某些公司（特别是Sun公司，Java的创造者）所期望的那么可用。Netscape需要更好些的东西。

Brendan Eich登场[③]，他原是MicroUnity Systems Engineering的雇员，当时刚受聘于Netscape。Brendan受命领导开发一个给非Java开发人员使用的新的、简单的、轻量级程序语言。在日益庞大的网站开发者军团中，有很多人并没有什么编程背景，所以面对Java的面向对象本性、编译需求、包和部署需求等特性时，会觉得很难对付。Brendan很快意识到，给这样的开发人员创建一个“顺手”的编程语言，他需要做出一些决定。比如，这个新的语言应该是弱类型的，并且是解释执行，因而具备极

---

① 《时间简史》是有史以来最著名的物理学和宇宙学的科普图书，本章的标题显然是受到该书的启发。该书的作者，剑桥大学的史蒂芬·霍金教授被公认为是世界上最著名的理论物理学家之一。他的书把许多当前有关宇宙的理论普及给众人，以及我们这些在讨论超弦、超对称以及量子奇点等类似事情时（当然不是在看《星际迷航》的时候）假装知道的人。更多信息，参阅http://en.wikipedia.org/wiki/Stephen_Hawking。

② 1927年，Google这个词在无声影片*Little Rascals*的*Dog Heaven*一集中被第一次使用，意思是喝口水。可以参见http://experts.about.com/e/g/go/google.htm。虽然这个参考并没有明确地说那是这个词第一次出现，但网络上许多的其他信息都暗示了这件事情。即使我能进入Jeopardy智力竞赛的决赛，我也不会押上我全部家当，这个话题是一个很礼貌的聚会讨论的话题。

③ JavaScript最初的作者名为Brendan Eich。——译者注

高的动态性。

他创建的这个语言最初被称为LiveWire，但由于它出众的动态本质，很快就被改名为LiveScript。然后，正如常见的那样，某些市场部的混混介入，决定把它的名字改为JavaScript，以便利用Java的名头。这个改变在Navigator 2.0 beta版阶段就已经实现了[①]。所以，不管怎么说，JavaScript从一开始就被叫做JavaScript。至少市场营销人员还算聪明地将Sun也拉了进来。在1995年12月4日，Netscape和Sun联合宣布了JavaScript，把它描述成HTML和Java语言的“补充”（当时创立它的初衷之一就是帮助Web开发人员更容易操作Java applet，所以也还算有些合理）。最让人觉得羞愧的是，在后来的多年里，人们持续不断地在邮件代码清单、公告板，甚至在开发人员和上网的公众之中混淆JavaScript和Java这两个（风马牛不相及的）概念。

没过多久，JavaScript就成了热门技术，不过这完全是靠它自己的本事挣的，和控制小应用程序（applet）根本没啥关系。Web设计者才刚刚开始掌握早期的静态网页，并想把它变得更加动态、与用户交互更强以及更加多媒体化。人们开始尝试创建交互式的、（相对）复杂的用户界面，JavaScript被看作完成这项工作的一个方法。一些看上去简单的操作（如根据鼠标事件来切换图片）开始流行起来，在JavaScript普及之前，这种简单的动态都需要大量浏览器插件。事实上，根据用户鼠标事件来切换图片这个JavaScript应用——在很长一段时间内可能都是JavaScript最流行的应用。操作表单及（通常情况下）验证它们，算是早期JavaScript第二流行的应用了。操作DOM（Document Object Model，文档对象模型）的功能，则花了更长的时间才进入舞台，其主要原因是早期的DOM 0级相对太简单了，主要的目的只不过是对表单、链接和锚（文字链）进行操作。

在1996年的早期，JavaScript创建不久，它被提交给ECMA（European Computer Manufacturers Association，欧洲计算机制造商协会）[②]进行标准化。ECMA (http://www. ecma-international.org) 提出了名为ECMAScript的规范，它涵盖了JavaScript的核心语法以及DOM 0级的一个子集。ECMAScript至今仍然存在，并且大多数浏览器都以某种形式实现这个规范。然而，很少听到有人谈论ECMAScript而非JavaScript。主要因为ECMAScript的名字是缩写语，太长，不好记。当然，这本书本身是讲JavaScript的，而不是ECMAScript。但是请记住：它们是一回事！[③]

是什么使JavaScript在如此短的时间内变得如此流行？最主要的原因大概是它的门槛极低。你需要做的所有事情就是打开任何一种文本编辑器，敲击一些代码，保存，然后在浏览器中载入文件。你用不着编译或者打包和部署——没那么多复杂的“编程”问题。JavaScript也没有复杂的IDE（Integrated Development Environment，集成开发环境）。它就像为你自己保存一点笔记一样简单。

JavaScript早期成功的另一个重要原因是它看起来很简单。你不需要担心数据类型，因为它曾经是（现在也还是）一种弱类型语言。它并不是面向对象的，所以你并不需要去考虑类的层级关系等。事实上，如果你不想，甚至可以不用处理函数（你想要脚本在页面载入的时候被立即执行）。用户不需要考虑什么多线程的事情，或者是惦记着学习所谓的泛型集合类。事实上，固有的JavaScript对象很少，因此，任何人，只要对编程有个模糊概念，就可以很快上手。正因为看上去简单，才导致早期的大量

① 作为相关历史的补充，你可能对Netscape Navigator 2.0引入的两个值得注目的特性都很感兴趣。除了JavaScript，还引入了frame。当然，这两者之中，有一个十分流行，而另一个却被大部分Web开发人员尽量避免使用，不过那又是另外一本书的故事了。

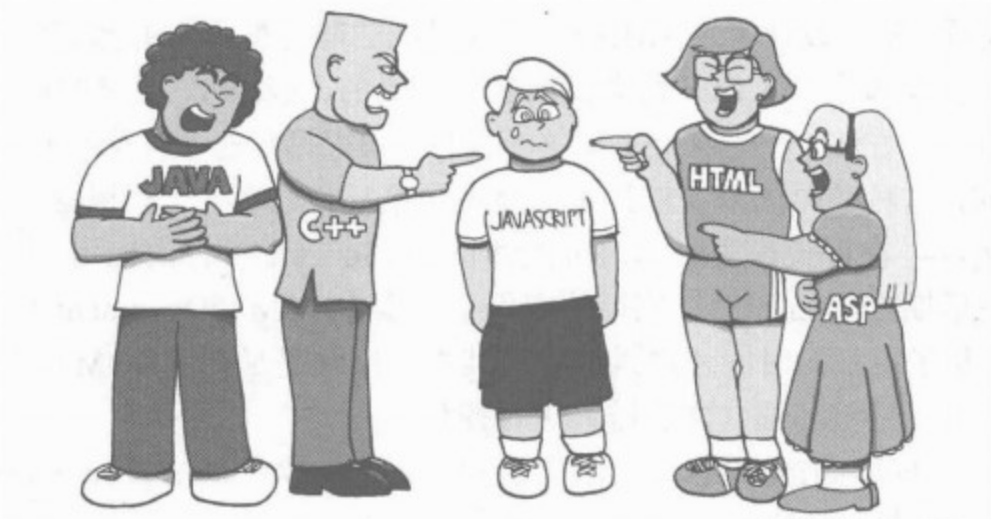
② ECMA目前已经是全球性的标准化组织，因此正式名称就是Ecma，原来含欧洲的全称形式已不再使用。——编者注

③ 严格地说ECMAScript只是一种规范，JavaScript（以及ActionScript等）均为具体实现，并不等同。——编者注

问题的出现。

不幸的是，JavaScript的幼年时期也并不都是鲜花和掌声。大量的高危安全缺陷严重损害了它的早期声望。大量面向非编程人员的书籍也导致大量不该写程序的人写了大量代码（何况还是为网站这么“公众化”的媒介写代码）。

然而，最大的问题也许是来自许多“真正的”程序员的精英们的态度。他们认为JavaScript缺乏开发工具（IDE、调试器等），它离开了浏览器（在一些测试环境中）就不能开发，以及太过简单，完全是一个“脚本玩具”，是那些可能只有业余爱好者、初学者或者菜鸟黑客才会使用的东西。在很长一段时期里，JavaScript在编程世界里很像一只“丑小鸭”。像Christina Crawford[①]一样永远被她“那个母亲”[②]（那个“真正”程序员）训斥。



可怜的JavaScript——其他语言就是这么无情

这种态度让程序员忽视了JavaScript表象下的惊人潜质，而这些潜质将随着JavaScript和那些使用它的人的技术的成熟而显而易见。这种态度同样也让很多优秀的开发人员远离JavaScript，而他们本应该帮助JavaScript加速成熟，而不是阻碍它的发展。但JavaScript天生就是做大事的，不管别人怎么说它！

1.2　JavaScript的发展：出牙期的疼痛

虽然JavaScript在早期并没有得到程序员的公平对待，但是毋庸置疑，他们的有些批评还是正确的。JavaScript的第一个版本还远远称不上完美——我认为Netscape或Brendan Eich都不会反驳这个事实！要知道，这些批评有些只是因为新技术本身需要几个版本来变得完善（就像人们频繁地批评微软公司总是需要若干个版本才能稳定那样），而另外一些，就是别的原因了。

那么，早期的JavaScript是被什么所折磨呢？有几个特别突出的：浏览器不兼容、内存占用和性

① Christina Crawford是电影明星Jane Crawford的养女，电影*Mommy Dearest*（http://www.imdb.com/title/tt0082766）讲述了她的故事。即使你不记得那部电影，你也一定记得电影中最让人难以忘记的一幕，Jane对Christina说的那句“再也不许用铁丝衣架！”

② 剧中妈妈如何在某夜拍完戏后从片场回家，发现养女的衣橱里没有按照她的吩咐使用上好绸缎包裹的衣架挂名贵衣服，却使用了比比皆是的铁丝做的衣架时，不禁勃然大怒，随手抄起衣架，没头没脑地将养女从睡梦中拉起来，毒打全身，还边打边骂：再也不许用铁丝衣架！——译者注

能问题。并且，还有一个JavaScript一开始并没有被大多数人接受的真正原因：开发人员自己！让我们一起来仔细探究一下这些领域，因为了解不久前我们的处境可以帮助我们认清现在我们的位置。

1.2.1 但它是相同的代码：浏览器的不兼容

为了更好地理解下面的讨论，并且顺便照顾一下那些喜欢图表胜过文字的人，我们来看两个时间线。图1-1表示了Netscape的Navigator浏览器的发布简史和与之同步的JavaScript版本史。图1-2表示了微软IE浏览器和它的基于JScript的JavaScript实现的相同信息。这些数据点都是准确的，不过我很有可能在哪里还是遗漏了什么发布点。而且，我没有把这些时间线延伸到现在，因为时间线上结束的那点之后，我们就处在ECMAScript的时期了，并且很大程度上可以在浏览器间兼容。

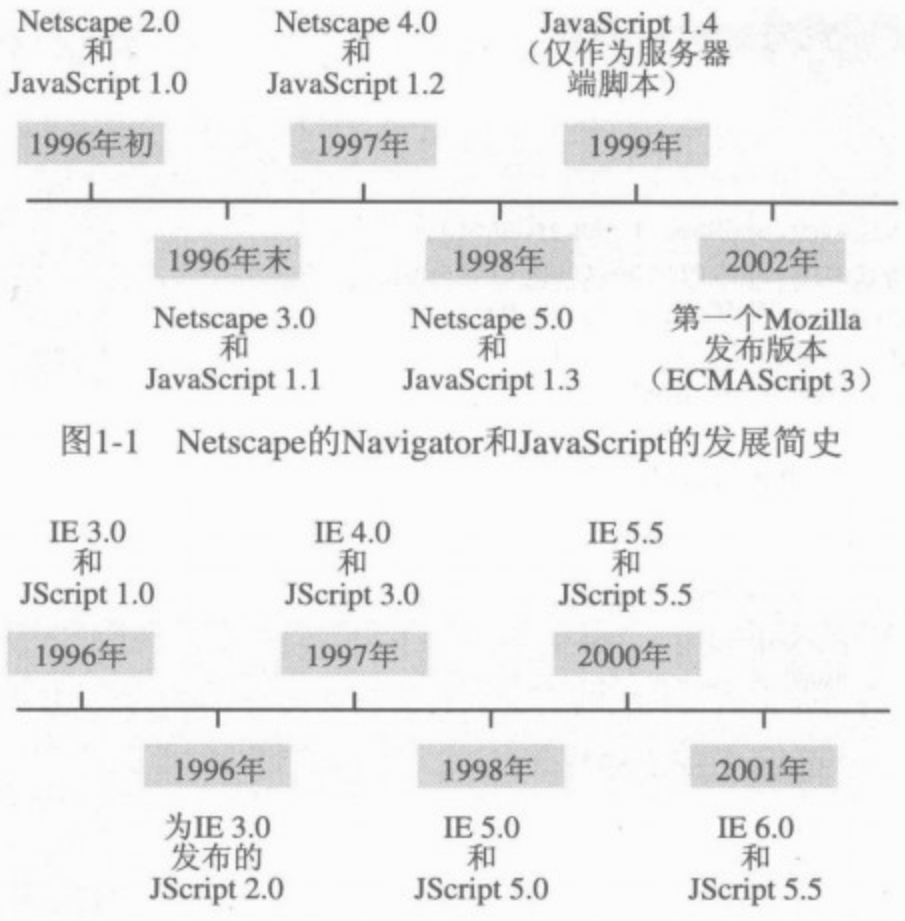


图1-1 Netscape的Navigator和JavaScript的发展简史

图1-2 IE和JScript的发展简史

当JavaScript出现时，微软的开发者意识到他们面临一个棘手的问题。不管早期的JavaScript存在过什么样的问题，很明显的是，它是Web开发者希望用的东西。要不还能怎样？开天辟地第一次，静态的页面可以动起来。[①]

微软对此找了个它自己的答案。事实上，是两个！首先，他们创建了VBScript，语法上是以Visual Basic产品为模型建立的。然后，微软还创建了JScript，这也是本节最重要的讨论，它是一个（基本）兼容JavaScript的东西。就是这个“基本”产生了问题。

长期以来，JavaScript被诟病的最大的问题之一就是它在不同的浏览器版本中不兼容（事实上，这

① 嗯，严格说不是第一次（获得动态网页能力），但的确是第一次不用某些笨拙的方法实现，与之对比的是那些毛病多多并且是需要额外下载时间的插件。要记得这些都是在宽带普及之前，是大家还在使用56kbit/s的调制解调器（而且那些调制解调器也很少能达到最高速度）的时代。

个情况直到近两三年才基本解决）。这个问题基本上是从微软自己的JavaScript实现出现后产生的。逻辑上，如果Netscape继续统治了浏览器市场的话，我们就不怎么需要讨论什么兼容性的问题了。最开始，[①]当微软发布了JScript 1.0的时候，它还是和JavaScript 1.0相当兼容的——兼容到足够进行跨浏览器开发。直到Netscape发布了JavaScript 1.1，兼容性问题才算真正开始。所以，如果你是一个微软的拥护者，那你可以随意批评Netscape。如果憎恨微软，那很明显是它的错！

从Netscape在Navigator 3.0中发布了JavaScript 1.1之后，微软的JScript实现版本就一直落后Netscape至少一个版本号，这样的状态持续了很长一段时间。所以，像图像转换这样的技术在Netscape浏览器中普及时，它在IE中却还没有出现（大约在IE 3.0时期）。为了处理差异，使用“浏览器嗅探”代码来启用或禁用某些功能变得流行起来。这种代码类似于代码清单1-1中展现的那样。

代码清单1-1　一段老式的浏览器嗅探例程

```
function Redirect()  {
  var WhatBrowser;
  var WhatVersion;
  WhatBrowser = navigator.appName.toUpperCase();
  WhatVersion = navigator.appVersion.toUpperCase();
  if (WhatBrowser.indexOf("MICROSOFT") >= 0) {
    if (WhatVersion.indexOf("3") >= 0) {
      top.location = "MainPage.html";
    } else {
      top.location = "BadVersion.html";
    }
  }
  if (WhatBrowser.indexOf("NETSCAPE") >= 0) {
    if (WhatVersion.indexOf("2") >= 0) {
      top.location = "MainPage.html";
    } else {
      top.location = "BadVersion.html";
    }
  }
}
```

在这段代码中，如果探测到的浏览器版本不是IE 3.*x*或更高版本，也不是Netscape 2.*x*，用户就被重定向到BadVersion.html，这个页面大概会提示浏览器不兼容。如果浏览器版本匹配最低需求，就结束于MainPage.html。不过，有很多理由可以说这段代码是有明显缺陷的，我想把找出缺陷当作你的练习作业。

① “On the gripping hand”（最开始）是一个在科幻小说《上帝眼中的尘埃》（*The Mote in God's Eye*）（1974年出版的一本著名科幻小说，具有星球大战式的宏伟场景和故事情节，并更加严谨——译者注）中的一个词，该书作者是拉里•尼文（Larry Niven）和杰里•奥耐尔（Jerry Ournelle），这个词也出现在该书的续集*The Gripping Hand*（握紧手）中。通常用于描述可用的第三个选择，比如，你说，“我们可以用A方法……；另一方面，我们可以用B方法，”你还可以说“我们还可以用C方法……”该词源自小说中描述的外星种族，Moties（魔特）族，它们的身体是不对称的，身体的一边有两只手，另外一边有一只手，而通常成单的这只胳膊是最有力的。这些都是上佳的科幻小说，如果你是科幻迷，并且还没看过它们，那我强烈推荐它们！大多数科幻迷都将它们奉为经典。（要是你没看过这两本书，还敢叫自己科幻迷？）

很重要的一点是，这种“嗅探”浏览器版本的代码流行了很长一段时间。实际上，对于同一个页面，我们通常要做两个不同版本：一个是为IE设计的，另一个是为Netscape设计的。这显然不是最理想的状态。但是在很长一段时期内，它确实是唯一的方法，因为总有那么一点儿代码在两个浏览器里表现得不一样。通常这是因为一种浏览器支持一种特性而另一种浏览器不支持——有时是因为私有扩展，有时是因为一种浏览器实现的是一个较早版本的JavaScript。其他的就属于两种浏览器的工作方式完全不同的原因。

但是，只检测浏览器的类型和版本是不够的，因为微软把浏览器和JScript语言设计成独立实体。它们可以各自独自更新而不触及另一个，因为JScript只是一个DLL（dynamic link library，动态链接库，程序在运行时链接的代码库）而已。当IE 3.0发布时，它仍使用JScript DLL的第一个版本。不久之后，当时大多数浏览器的版本仍然是IE 3.0时，微软将JScript更新至2.0版本。微软提供了两个函数：ScriptEngineMajorVersion()和ScriptEngineMinorVersion()[①]，这两个函数不仅仅非IE浏览器不支持，连JScript 1.0也不支持！所以，它们带来的麻烦比它们提供的功能更多。尽管如此，它们仍然不得不用，因为有时候你需要版本信息来相应地给自己的代码做分支。

回头看看那时候我们不得不对付的这种不兼容的例子，String类中的split()方法有一个可选的参数limitInteger，它可以限制字符串被转换成数组后数组元素的个数。然而，这个参数只被Navigator 4识别。另外一个例子是，在Navigator 3之后，Netscape才开始支持typeof操作符，而微软在JScript 1.0中就引入它了（这是一个被证明十分有用并被添加到ECMAScript 1.0规范中的专有扩展）。再举一个例子，看看下面的代码片段：

```
var d = new Date();
alert(d);
```

在早期，这样的简单代码可能会有问题，因为Date对象的toString()方法直到JScript 2.0的时候才在JScript中出现，而它在Netscape的实现中是与生俱来的。

诸如这些，各种各样的问题经常会在最不合适的时间出现。IE中的substring()和Navigator中的substring()对付负数值的做法完全不同[②]，而（开发的）截止日期又逼得很紧，这个时候（这种不兼容）简直就是灾难了。这就是浏览器嗅探程序在很长一段时间内都非常流行的原因，即便我们知道它并非好办法。

如果这就是JavaScript唯一的毛病，那我猜想开发人员可能会一边哭爹喊娘地抱怨，一边直接面对（不兼容）并习惯它。不过，糟糕的是，不兼容并非是对JavaScript的唯一的批评。

1.2.2 蜗牛和大象：JavaScript性能和内存问题

JavaScript可以很慢。没错，我说的！即使在今天，你也可以很容易地写出性能很差的代码。代码清单1-2显示了一个简单的例子。

① 这两个函数可以输出IE JScript的主从版本号。——译者注

② 按我的记忆是这样，不过老实说我没法从Google里搜索到证据来说明这些事情。所以我这里说的这个可以当作纯粹的谈资，只是希望我的记忆还不至于在我如此年轻的时候就如此不济。

代码清单1-2　很差的JavaScript性能举例（以及如何解决它）

```
<html>
  <head>
    <title>Listing 1-2</title>
    <script>

      function badTest() {
        var startTime = new Date().valueOf();
        var s = "";
        for (var i = 0; i < 10000; i++) {
          s += "This is a test string";
        }
        return new Date().valueOf() - startTime;
      }

      function goodTest() {
        var startTime = new Date().valueOf();
        var stringBuffer = new Array();
        for (var i = 0; i < 10000; i++) {
          stringBuffer.push("This is a test string");
        }
        var s = stringBuffer.join("");
        return new Date().valueOf() - startTime;
      }

      function betterTest() {
        var startTime = new Date().valueOf();
        var stringBuffer = new Array();
        for (var i = 0; i < 10000; i++) {
          stringBuffer[stringBuffer.length] = "This is a test string";
        }
        var s = stringBuffer.join("");
        return new Date().valueOf() - startTime;
      }

      function doTests() {
        var htm = "";
        htm += "Time badTest took: " + badTest() + "<br>";
        htm += "Time goodTest took: " + goodTest() + "<br>";
        htm += "Time betterTest took: " + betterTest();
        document.getElementById("result").innerHTML = htm;
      }

    </script>

  </head>

  <body>
    <a href="javascript:void(0);" onClick="doTests();">Click here to test</a>
```

```
    <br><br>
    <div id="result"> </div>
  </body>

</html>
```

正如代码清单1-2的标题所示，这个例子也提供了免费的奖品：一个你绝对可以在现实程序中使用的优化方法。这个例子使用3种方法（很显然是人为地）做了同一件事。

- 构造一个字符串，它由10 000个“This is a test string”组成（就是“This is a test stringThis is a test stringThis is a test string...”持续10 000次）。它是使用+操作符拼接的简单字符串。
- 创建一个数组，并使用push()方法向数组中添加“This is a string”10 000次。最后使用Array类的join()方法和一个空白字符来返回一个字符串，这个字符串由数组中所有的元素连接而成，中间没什么分隔符。
- 使用相同的数组方法，但是不使用push()，它明确地指定数组中的每一个元素，在实际操作时，你可以给那些下标与长度相同的数组设置元素值，数组就会逐一增长。

图1-3展示了在Firefox中，每一个方法使用的时间。你可以看出并没有哪个使用了特别长的时间。Mozilla的开发者在优化他们的JavaScript引擎方面做得相当出色，这在使用简单的+来拼接的例子中表现得尤为明显，它用了最短的时间。即使是不久以前，情况也还不是这样的。

现在，我们来看看在IE中进行的相同的速度测试，如图1-4所示。数组测试部分确实比在Firefox中快了一些，虽然快得并不多。但是很明显在IE中字符串拼接（使用+操作符）是个大忌。它比Firefox慢了95倍！

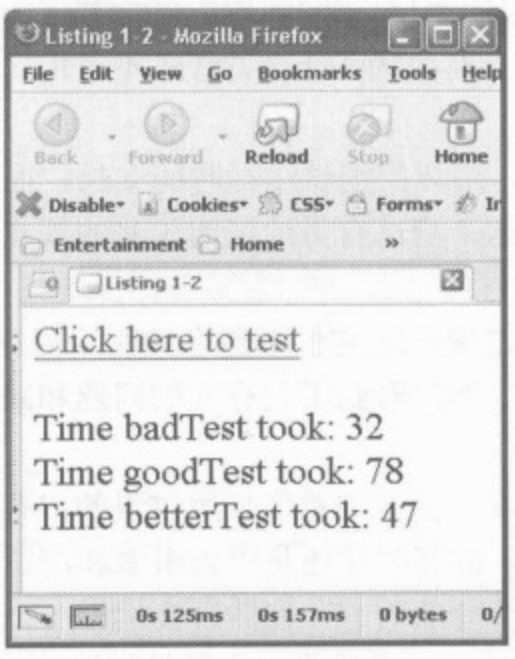


图1-3 在Firefox（1.5.0.6，写作时最新的版本）中的速度测试结果

图1-4 在IE（6.0.2900.2180，写作时最新的版本）中的速度测试结果

为了避免有人认为这里还有其他的干扰因素，我们这些速度测试都是在同一个PC中进行，并没有虚拟机和其他类似的程序。所以，这些区别基本上可以完全归于浏览器的差异。操作系统本身运行时的情况不同可能会有影响，但是我实际上在执行每一个测试之前都重启了电脑并且没有载入任何其他东西，所以这里的运行环境已经是尽量相同了。

说明　我在1.5.6版build 4.2的Maxthon（写本书时的最新版）中运行了相同的速度测试，Maxthon几乎是我用于日常浏览的首选浏览器。它是一个IE的封装，用不少特性扩展了IE，并且修补了不少漏洞。可以把它放在能和Firefox以及一些其他浏览器平起平坐的地位（起码我是这么看的），它用的是IE的呈现引擎（有人会认为这个是很差的东西，但是大多数站点在IE中都能看，而在Firefox中会有问题）。测试的结果非常出人意料：最坏的测试需要19141，好一点的测试需要141，更好的测试需要93。我还没有办法解释它为什么会慢这么多，尤其是通过字符串拼接的方法。我并不是说这就是对Maxthon的批评，这是举例说明在不同浏览器中性能的差异，即使看起来逻辑上并没有什么可预知的不同，在你的代码工作的时候还是有一些其他需要注意的。

这些并没有想游说读者哪种浏览器就是比其他好的意思。实际上，大部分Web开发者会告诉你Firefox是最棒的，虽然我们可以看到做同样事情的那3个方法中有两个都比IE稍慢。我想说明的要点如下。

- 同样的JavaScript代码片段在不同浏览器中执行的性能不完全相同，而且有时会差异很大。
- 从某种意义上来说，现代JavaScript引擎的性能仍然有很多有待完善的地方。

这就是现在的情况。过去更糟糕。图1-5表示了同样的例子运行在Windows 98的IE 4.0中的情况。

图1-5　在IE 4.0中的速度测试结果

喔，IE开发组很明显是够忙的了！那个最坏的测试，简单地使用+操作符，在现在的IE版本中竟然比IE 4.0中快了13倍！更好的测试结果也比IE 4.0中快了2倍。注意，这里好一点的测试不能运行，因为在这个版本的JScript中Array对象的push()方法还不可用。我想我们有理由猜测再往前的版本还会更明显地变慢。

在Netscape 3.01中，相同的测试的结果更糟糕。事实上，最坏的测试运行时间太长，而且消耗了太多的系统资源，以至于我不得不杀了那个进程。可以说，这个测试有力地证明了我的观点，这些年以来性能有了明显的提升。

Netscape 3.0还显示了早期JavaScript实现中的另一个普遍的弱点：它们不能有效利用内存。这个缺陷可以归因为大多数软件都在进化过程中。你写了一些东西，然后看到了它存在的问题和缺点，并在下一个版本中予以纠正。JavaScript引擎亦如此。

即使只是几年前，人们也很容易发现一些简单的代码片段就会让浏览器使用过量的内存。内存泄漏很普遍。尽管大多数时候都是因为开发人员不恰当的编码，但有时候还是因为引擎和浏览器自身造成了这种泄漏。要知道，JavaScript和Java一样，都是有内存管理的语言，在后台会运行一个垃圾收集器。如果JavaScript解释器自己有缺陷，那么垃圾收集器也有缺陷就不算是太离谱的想法了。

速度和内存的因素让众人对JavaScript有了速度慢且耗内存的印象。它还处于开发的早期，和所有（相对）复杂的软件一样，一出现的时候并不完美。这不意味着现在我们就没有这些问题，它们还存在（看看第一个例子就知道了）。但是这些问题已经不那么频繁地发生了。我甚至敢说它们很少发生了，除非是开发人员自己导致的。这些问题也不像以前反差那么大了。比如，除非你自己干了非常愚蠢的事情，否则很难杀死浏览器，像我对Netscape 3.01进行测试时所做的那样。

说说开发人员做的蠢事……

1.2.3 所有罪恶的根源：开发者!

像我在前面几节中谈论的，早期的JavaScript实现中确实存在问题。今天你还能发现一些问题，不过少多了，并且出现的频率也低得多。然而有一个问题永远存在，就是开发者。简单地说，JavaScript是一个非常强大的语言，但它同样也很容易把事情弄糟。不努力尝试的话，很容易写出非常慢、臃肿且易错的代码。

与语言本身需要进化一样，开发者也需要不断进化。他们需要知道什么好用什么不好用。并且开发人员需要自己努力找到比较好的方法。JavaScript灵活且动态，这就诱使很多开发人员做了许多在那些比较僵硬的语言中不能做的事情。比如上节中的示例。如果你正在用Java，那么很有可能确切地知道使用字符串拼接是坏事（Bad Thing），而字符串缓冲区是你的好朋友。但是，在JavaScript中没有字符串缓冲区，所以一些开发者简单地认为字符串拼接就是该用的方法。正如那个例子中展示的，在Firefox中，这可能不是什么太大的问题，但是在IE中，你却是在自找麻烦。

另一个例子是给一个函数传递参数，请看代码清单1-3。

代码清单1-3 不够高效的编码示例

```
<html>
  <head>
    <title>Listing 1-3</title>
    <script>

      function Person1(firstName, lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.toString = function() {
          return this.firstName + " " + this.lastName;
        }
      }

      function Person2(attrs) {
        this.firstName = attrs["firstName"]
        this.lastName = attrs["lastName"];
        this.toString = function() {
          return this.firstName + " " + this.lastName;
        }
      }

      function showPerson() {
        var p1 = new Person1("Frank", "Zammetti");
        var p2 = new Person2({"firstName":"Frank","lastName":"Zammetti"});
        document.getElementById("divPerson").innerHTML = p1 + "<br><br>" + p2;
      }
    </script>
</head>
<body onLoad="showPerson();">
  <div id="divPerson"> </div>
```

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

这里，我们有两个不同的类来表示人：Person1和Person2。Person1的构造函数接收两个参数，firstName和lastName。Person2接收一个单独的数组参数attrs，是属性的数组。showPerson()函数创建了两个同样的人，一个使用Person1，另一个使用Person2。当我们想添加其他属性来辅助描述一个人时会发生什么呢？对于Person1，我们需要修改构造函数使其接收更多的参数。对于Person2，只需要添加适当的字段设置代码。还需要修改两个类对构造函数的调用，这是必要的。但是，Person1的调用告诉了我们什么呢？

```
var p1 = new Person1("Frank", "Zammetti");
```

仅通过这个调用，你无法推断出参数的含义。我们怎么才能知道Zammetti实际上不是我的名字，或Frank不是我父亲的姓呢（万一呢）？显然，Person2的调用语法在代码可读性上更好一点。用这种方法写的代码的扩展性更好些。

这相对来说比较次要，但是它是在近几年才被JavaScript开发者意识到的一个风格元素。在早期，你几乎不会看到在Person2中使用的那种方法。那时，你可能会看到函数调用的时候都伴随着许多参数。可是如果你问那些C++开发者他们会如何编写函数调用的代码，几乎肯定会听到他们的回答中包含某些集合，可能会是传入的一个值对象，或者类似的东西。

另一个在很长一段时期内都十分普遍的问题是变量的作用域。（JavaScript中的）所有东西都是在全局作用域下的，这与几乎所有其他语言不同，其他语言中的变量一般都是在它们被请求的层级内。另一件长时间困扰大量程序员，即使现在也偶尔发生的事情是JavaScript中缺乏块作用域（block scope）。请看代码清单1-4。

代码清单1-4　JavaScript缺乏块级作用域的例子

```
<html>
  <head>
    <title>Listing 1-4</title>
    <script>
      function test() {
        var i = 1;
        if (1) {
          var i = 2;
          if (1) {
            var i = 3;
            alert(i);
          }
          alert(i);
        }
        alert(i);
      }
    </script>
  </head>
  <body onLoad="test();"></body>
</html>
```

在地球上的任何其他一种语言中，alert返回的都是按顺序3、2、1的。但在JavaScript中，你得到的结果是：3、3、3。变量i只在第一次出现的时候被分配了一次，后面在低级范围中每一次的声明都是重写了那个变量的值。

一个更大的改变是向更合理的面向对象方向的变化。多年来，JavaScript开发人员（那些貌似非常了解JavaScript的人）并没有真正意识到JavaScript是面向对象的。他们往往会写一堆函数，就好了（很长一段时间里，外部化的JavaScript甚至都不是一种常用手段，而外部化是开发人员进化的另外一个方法）。但是，如果你观察一下大多数的现代JavaScript库，比如Dojo和script.aculo.us，就会发现里面有很清晰的面向对象设计。

早期对JavaScript的另一个批评是使用JavaScript的开发人员相对来说比较业余，并且并不完全了解自己做的事情。不幸的是，和大多数不快的总结一样，事情是从一些小的实情开始的。正如之前讨论过的，JavaScript的入门要求非常低。你只需要把HTML页面丢在一起，在里面放一些脚本，并用浏览器运行。不需要编译，不需要安装成套的开发工具。只要有记事本和一个可访问的网站就可以了。由于这些，什么都开始编写起脚本来。出于偶然，你的表单在客户端进行了验证，这个挺酷的，但是这个验证没有在服务器端进行，因为JavaScript的代码作者不知道在服务器端验证是好事（Good Thing）。你的图像轮转功能没有预先载入图像，因此每次鼠标事件都导致怪异的网络流量，更不用说是那些看上去已经不响应的用户界面了。还有对所有网页浏览者的毒药：弹出式广告。

上面所有的一切（除了弹出式广告之外，它们都是恶意的市场人员强行干涉Web技术的恶果）确实只是无经验的开发人员会做的事情，因为他们还不知道如何做得更好。这并不是JavaScript的错，因为总有些类似事情会发生，然后导致所有这些毛病。就像我们早期毛茸茸的祖先那样，我们必须有个进化的过程。

1.2.4 DHTML——魔鬼的时髦词

“恶意的开发人员”故事的另一个元素和DHTML（Dynamic HTML，动态HTML）有关。尽管DHTML的标签时至今日仍然适用于其效果。该术语的含义让人们不愿意再用它。这个含义就是尽管有很多付出，收获却很少。

早期的JavaScript开发人员发现他们可以做很多技巧的东西——从背景色的淡出到各种各样的滚动文本，以及不同的页面转换效果，比如擦除等。尽管这些特效看上去很酷，但是它们除了能让人看着好玩之外没有其他的用途。哦，别错解我的意思——看着好玩是很棒的！我自己就非常喜欢下载新的屏保或者一些给Windows shell增加新特效的工具。这些都很好玩！但是我却发现自己稍后总是会删除这些东西，不仅因为它们破坏了系统性能，还因为它们很快就让人厌烦、注意力分散。

早期的JavaScript开发人员是这种垃圾的巨大的布道者，而且这些东西迅速老化。有人开始怀疑Web是否值得投入，因为到处都是游戏场而没有严肃的业务，假如有人持这种观点，我是一点都不认为奇怪的。一个用各种视觉垃圾令访问者生厌的网站，是不会有人再想用它的。如果你想靠那个站点生存，并且你的公司的收入依靠它，那么坏消息很快就会到来了！

显然这不是技术本身的问题。即使我们手里有原子弹，也不意味着要到处扔啊！也许把网页上的惹人烦的各种效果和核战做类比有些夸张，但是这里的含义是正确的：一项技术的存在并不意味着你

要去做那些没必要做的事情。[①]

这里有一个快速的测试：如果你使用微软的Windows，查看一下你的计算机的“性能”选项（右键点击“我的电脑”，选择“属性”，点击“高级”标签页，然后点击“性能”组里面的“设置”按钮）。你是否关闭了最大化和最小化时放大和缩小窗口的功能？是否关闭了鼠标下的阴影？是否禁用了在应用程序关闭时增大和收缩任务条？许多人都习惯把这些东西关闭，不只是因为这么做能让系统快一些（或者至少感觉上如此），而且还因为这些特效有些的确碍事。在最小化一个窗口的时候看着它飞到任务栏上实在是没什么意义。现在，你可能会辩驳：这依赖于具体实现，因为在Macintosh上的特效就好很多而且没那么惹人烦，从某种程度上来说我同意。但你还是得问问自己，这些特效是否有助于你的工作。它让你工作效率更高了？我敢说几乎对任何人来说，回答都会是“不”。所以，即便是在厌烦或者是张扬上有度的区别，有些东西通常仍然是惹人烦、张扬并且没意义。糟糕的是，这就是很多人认同的DHTML的含义，尽管我希望事实不是这样，但它仍然是一个不值得宣扬的含义。

所以，JavaScript开发人员的进化的一部分就是开始认识到那些超级酷、很炫的特效应该先放在一边。开发人员开始意识到他们做的东西其实是降低效率的，因为这些特效在很多时候都是让人分散注意力并且令人生厌的。相反，过去的几年里，一个蔓延开的潮流是负责。有些人会说这是JavaScript朝着被认可、接受的进化过程中一个最重要的部分。

我们今天仍然能看到很多炫酷的特效，可能比以前更多。但这些特效是真正能提高用户感受的。比如，使用黄色淡出效果（由37signals初创http://www.37signals.com），一个页面的变化会在页面刷新之后被简明地突出显示出来，然后迅速地淡出成平常的状态。在刷新页面之后寻找变化通常是挺困难的，而这个技术帮助用户把注意力放在那些变化上。这就帮助用户更有效地工作。这是时下流行的比较好的特效的一个例子，对任何人来说，它比以前的都好多了。

提示　体会黄色淡出特效的正面用法的示例：看看ClearLeft的联系人表单http://clearleft.com/contact/。不用在框里输入任何数据，只是点击提交（submit）按钮，看看会发生什么。在37signals的BaseCamp产品（http://www.basecamphq.com/）里到处都可以看到这种特效（需要注册一个免费账号来玩）。体验一下就会知道在哪里需要这些特效以及为什么这种特效吸引了大量眼球。其他37signals的产品也使用了这种技巧，所以随便看看吧——在页面的顶端总是有些可以学习的好东西！如果你想直接看看代码，那就看看Matthew Linderman的博客：http://www.37signals.com/svn/ archives/000558.php。

所以如果你听到术语DHTML，也不必条件反射式地感到恐惧（有些人会这样），因为它实际上还是从单纯的技术定义上，准确地描述了我们今天要做的事情。不过，你可能同时也会认识到这个词的确也是有着太多的负面含义，这些负面的东西都是早期的JavaScript开发人员[②]带来的恶魔。

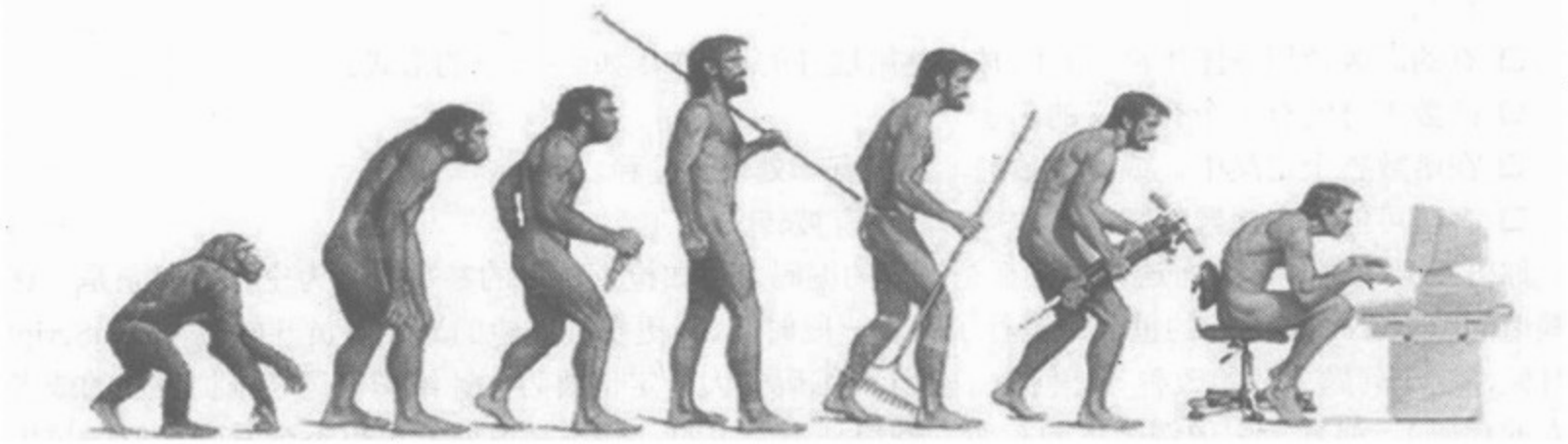
① 我记得一次电视商业秀，在那次活动中，一堆Web开发人员给老板显示他们新做的网站。老板说网站上需要更多会动的东西，比如一个会闪的图标。开发人员有些诡秘地看了看他，然后放了一个忽闪忽闪的图标在页面上。很明显所有在看这个秀的人都认为这个图标没起什么作用，而且实际上还起了反作用：因为它让这个站看上去很业余。视觉特效实在太容易被滥用了，甚至都不好玩了！

② 我不只是一个理发店的店长，也是一个顾客。在我的存档里还有些老的网站（万幸，它们都已经不在了），这些站点上实在是有一些很恶心的东西！我当然没法对DHTML的炫酷追求潮流病免疫。相信我，我自己也有忽闪的图标。我很高兴已经从我的错误中学到了东西（还有我的老板）。

1.3 进化还在继续：接近可用性

在第一批相对没有经验的开发人员使用JavaScript（并且经常使用得非常糟糕）之后，新一轮开始隐现。人们认识到一些普遍的错误并开始修正。

最重要的一条可能是，很多开始躲避JavaScirpt的、有经验的开发人员已经看到它的能力并开始用自己的聪明才智来改造JavaScript。这些带有真正计算机科学背景的人开始关注并指出错误所在，同时提供修正错误的方法。有了这些反馈，JavaScript就发生了类似文艺复兴那样的变化。新主意越来越多，改进也逐步加入其中。虽然这还不是最终的目的地，但却是在这条路上的一个重要里程碑。



JavaScript开发人员：走出树林，开始上网

1.3.1 建立一个更好的窗口小部件：代码结构

听起来可能没那么重要，但是使用干净、有效的方式来简单地组织代码结构，可以使代码更加可读、可理解，更容易在月复一月年复一年的时光中得以维护。你有多少次碰到下面这样的代码？

```
1:  function f(
2:  p1, p2)
3:  {
4:  p2 =
5:    p2.toUpperCase();
6:  s = ""
7:  for (i = 0; i < 10; i++) { s = s + p1;
8:      s += p2 + '-' + i
9:  }
10: if (p1 == "y") s += '<br>' + s
11:     if (p2 == 'n')
12: {
13:     s = s + "<br><br>"; }
14: }
```

拜托，别尝试指出这段代码想要做什么。它是没有意义的代码（我只是胡乱把一些东西写在一起）。不过虽然没有什么可理解的东西，但它在语法上是正确的并且可以执行。这个例子的重点是，代码的结构。它很让人愁，不是吗？让我们来看看它存在的问题吧（没有特别的顺序）。

- 有些行没有缩进（比如第2行），或者缩进不一致（第5行2个空格，第8行4个空格）。
- 参数名称不是描述性的。
- 引号使用不一致（单引号和双引号）。
- 有些行使用分号结尾，有些行没有。
- 有些代码段用花括号括起来了（如第7行到第9行的for循环），有些没有（第10行的if语句）。
- 在p2上调用toUpperCase()之前没有检查，如果只传入一个参数，或者第二个传入参数为空（null），那么就会抛出一个异常。
- 有时代码使用Sun标准，在一行的结尾设置一个左花括号开始一个代码段（第7行）。有时却又独立一行（第3行）。有时右花括号也独立一行（第14行），有时却在代码段的结尾（第13行）。
- 有的时候使用+=操作符，有的时候使用展开的+操作，如s = s +的形式。
- 函数本身没有一个有意义的名字。
- 在函数整个定义中，或在函数前，都没有一处单独注释。
- 有些可能引发问题的字符，即<和>，没有被转义。

你可能会争论说，上面这些大多都是懒惰的编码，比如检查传入的参数是否为空。可问题是，这种懒惰的编程在JavaScript的世界中流行了很长一段时间。当更多有经验的开发人员开始参与JavaScript的时候，这个问题才开始改善。任何为了生计而编码的人一定也要为生计维护代码（维护自己的或者别人的代码），而看上去像例子里的东西，恐怕是不能用的。并不是说你就再也不会看到类似的垃圾代码了，也不仅是在JavaScript中，但是现在它出现的频率确实越来越低了。

即使是函数的使用，像前面那段糟糕的代码示例中看到的，在JavaScript中也不是必需的。实际上，早期的网页里，你经常会看到整个网页并没有使用任何函数，或者用得非常少。你可以找到散布在整个页面中的<script>块，在页面被处理的时候执行。这样做仍然是可以的，而且有时是达到某些目的的最好方法，但是在像这样的整个网页中，通常就不是个好主意了！所以，开发人员开始认识到，使用函数可以很好地组织代码结构。使用onLoad页面事件来调用初始化函数，代替那些曾经在页面中的某个位置使用的匿名<script>块，变得流行起来。

另外一个相对重要的变化是外部化JavaScript的概念。这是不唐突的JavaScript的一个要素，下一章中会讨论这个概念。将脚本外部化可以让页面容易阅读，因为这样人们可以把精力集中在标记上，然后根据需要参照代码。这样做还有一个好处是容易重用，是另外一个早期JavaScript缺乏的东西。[①] 外部化的脚本会让人们更多地考虑重用。另外一个外部化脚本的好处是它能提升一些性能。浏览器会缓存一个.js文件，如果你碰巧在另外一个页面里用到了这个文件，那么浏览器就会少发出一个请求。另外一个不那么明显的好处是，其他人可以很容易地使用你的脚本，了解它如何工作。如果你曾经试图从一个200 kb的页面里挖掘几行JavaScript代码，看看原作者是如何实现一些好用的技巧的话，那就知道我说的是什么了。在一堆标记和其他无关的脚本［甚至还可能有样式表（.css），因为如果一个人

① 重用通常来说是挺难的！人们经常会（甚至是通常会）发现针对手边的需求写专用的代码是最容易的。人们需要花很多精力来思考通用性，才能把代码设计得既适用于其他类似的场合，同时又可以解决手边的问题。程序员通常都是懒虫（我当然知道，因为我就是其中一个），而且喜欢走容易的道路。就像愤怒、恐惧和贪婪是通向原力（电影《星球大战》中众武士拥有的超自然能力。——译者注）的阴暗面的道路一样，懒惰是通向无法很容易地重用的代码的道路。当然，不知道怎么做更好也是其中的一个因素。

没有外部化脚本，那十之八九也没有外部化样式表］里寻找一些东西实在是痛苦的事情。现代的浏览器工具让这件事情没那么痛苦了，但是这样的寻找仍然不是让人开心的体验，而在不久以前，这可是非常痛苦的事情。

1.3.2 重拾好习惯

虽然JavaScript早期的很多问题来自经验不够丰富的程序员的搀和，但是肯定不能说所有问题都是因为这样。因为有时候，似乎只要一晚上，就会有数以千计的经验丰富的好程序员会同时变蠢！

我前面说过，写JavaScript实在是太容易了——在文件里写一些代码，打开浏览器，然后就可以用了！在其他大多数语言里，你必须有个编译的环节，这个步骤可以过滤掉很多问题。然后会有一些静态的代码分析工具，可以找到更多需要修补的东西。甚至还有代码格式化工具，将代码格式化成恰当的标准格式。而在用JavaScript的时候，这些东西（通常）一个都没有。我在圆括弧里写了“通常”二字是因为现在已经有开发工具提供所有这些东西了（当然，不包括编译部分）。

可能“网络泡沫”也对此有“贡献”。我指的是公众刚开始上网，刚知道网络有多酷，然后每个人都认为可以很容易地从网络上挣到钱的时期。很多刚毕业的有MBA学位的学生，一周工作80小时、喝汽水可乐、吃多纳圈，穿着单调的衬衫，骑Segway[①]（噢，那时候还没有Segway，不过我这儿有！），总是宣称那些股票期权会多得你数都数不过来。这也可能让程序员变懒，他们只是把代码写了，然后拼凑起来，看上去能用就行了，以便能尽快实现所谓的商业计划，而实际上却导致了更多麻烦。

对的，你说得对，不过，嗯……可能也不对。

开发人员经过长时间学到的好习惯（如代码格式化、注释和逻辑代码结构）在JavaScript的环境里，必须重新学习。当然，那些从来没写过代码的人更是需要从头学起了。学习了这些东西之后，JavaScript就逐渐不再是那种令“专业”开发人员本能地摇头的东西了。现在，只要有足够的知识，JavaScript就可以成为一流的公民。

当然，最后一步还没到来。

1.4 终极进化：专业的JavaScript

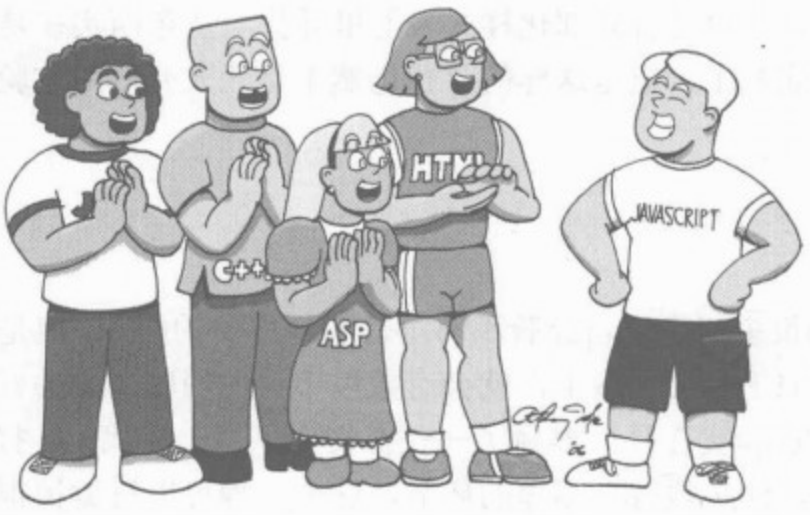
我们到达现在的情况，也就是这两三年。JavaScript已经长大了。

整个Ajax运动推动着JavaScript来到一个更重要的位置，但是即使是在这之前稍微早点的时候，好事情就开始出现了。那时，人们就开始期望制作更有趣、更富交互性、更加亲和用户、类似肥客户端样子的Web应用了，这导致了在客户端做更多事情的需求和期望。性能考虑自然也起了作用，不过我认为相对来说是比较不起眼的一个因素。

现实是JavaScript相当迅速地进入了一流公民（“专业”开发）的领域。可能最好的证明就是现在可以在大多数工作搜索站点上看到类似JavaScript工程师、JavaScript开发主管和高级JavaScript开发人员等需求的字眼了。而且在面试的时候，人们会直接面对面地这么说。

那么，除了Ajax之外，当前还有什么其他趋势可以归功到JavaScript名头下呢？让我们看看。

① Segway是一种电动双轮代步工具。——译者注



JavaScript：终获应有的尊重

1.4.1　浏览器

过去的几年中，主流的（甚至一些非主流的）浏览器都已经达到了相互之间的JavaScript实现基本兼容的程度。人们还能不时地发现一些不足的地方，但这些不足已经是很微不足道的了。现在，人们很少需要为不同的浏览器写分支代码，已经几乎没听说过需要嗅探浏览器来重定向到一个浏览器版本相关的页面。

现在，如果你编写ECMAScript兼容的代码，就会发现它可以在大多数主流的客户端浏览器里运行。当然这不等于说不需要针对浏览器做些性能优化。代码清单1-3中的例子就是个好例子。代码在浏览器间是兼容的，但是如果我们想做字符串拼接，那在IE里还需要做一些调整。

实际上，如今你会发现主要的问题和JavaScript的实现根本没关系，因为所有的主流厂商到目前为止，都已经遵循了好几个版本的ECMAScript。从兼容性来看仍然存在的问题实际上是DOM。

DOM是当前页面的内存表现形式。页面里的每一项都是一个树结构上的节点——这棵树是用页面上的元素之间的关系构造出来的。到目前为止，JavaScript已经以ECMAScript的形式标准化很久了，但是DOM在JavaScript发布之后很久都没有标准化。这就导致了主流浏览器厂商在做一些事情的时候使用了截然不同的方法。

举个例子，让我们看看在IE里和Firefox里如何处理按键事件。假设我们想在当前的文档上挂一个keyDown的事件钩子。我们可以这么做：

```
document.onkeydown=keyDown;
```

这个在IE里工作得挺好，但是在Firefox里，你必须这么做：

```
document.captureEvents(Event.KEYDOWN);
```

两个浏览器的基本的keyDown()函数的原型都是：

```
function keyDown(e) { }
```

在Firefox里，参数e是传递给函数的描述按键事件的event对象。但是在IE里，这个参数实际上根本没有传递进去，因为IE使用了一个叫事件冒泡的事件模型。要在IE里获取这个event对象，你需要引用window对象的event属性。这个不是JavaScript本身的区别，这是在不同浏览器的DOM里，不同的事件处理模型的区别。

看一个更长的例子，一旦你有了一个event对象，通常都会想看看是哪个键被按下去了。同样，我们还得克服DOM的区别。在IE里，event对象公开一个keyCode属性。在Firefox里，对应的属性叫charCode。所以，我们可能需要一些分支代码来获取键码。通常会像下面这样写：

```
document.onkeydown = keyDown;
if (document.layers) {
  document.captureEvents(Event.KEYDOWN);
}
function keyDown(e) {
  var ev = (e) ? e : (window.event) ? window.event : null;
  if (ev) {
    return (ev.charCode) ? ev.charCode:
      ((ev.keyCode) ? ev.keyCode : ((ev.which) ? ev.which : null));
  }
  return -1;
}
```

这段代码应该可以在任何浏览器里用。第2行的第一个if只在非IE的浏览器里为真，那些浏览器里的document对象会有layers属性。这个时候，调用captureEvents()。然后在keyDown()的内部，第1行会把ev设置为传入的参数e（如果有传入参数e）。如果document.layers属性不存在，则使用window.event。但是如果没有定义window.event，那么设置ev为空（有好几种情况可能发生这种事情，所以必须检查它）。然后，如果设置了ev，则找出键值：分别看看event对象里是charCode还是keyCode。如果ev是null，则返回 -1。

类似这样的代码变得越来越没必要了，因为浏览器也逐步开始尽量做到DOM标准兼容了。偶尔我们还需要这么做，但是比以前已经好多了！

浏览器厂商的另外一个进步是在性能和内存使用方面。虽然JavaScript还是解释性语言，但是在现在的浏览器里运行JavaScript已经比以前的好多了。优化变得越来越快，越来越猛。每个后继的JavaScript版本都在跳跃式地进步。可能导致进步最大的原因是更多的使用模式随着时间的推移不断出现。比如，DOM操作毫无疑问是JavaScript最常用的功能，所以在这方面就做了大量工作以尽量提高其效率。

类似地，垃圾收集算法也进步很大，这样在一段时间内用的内存更少。这些年JavaScript引擎自身也写得更好了，所以它们本身就会少用一些内存。代码也夯实了许多。现在，内存泄漏几乎都是因为开发人员的错误造成的，浏览器和JavaScript引擎实际上已经不再是主犯了。

最后，在现代的任何一个实现里，都很少有崩溃发生了。以前的情况是你会不时地看到一些崩溃的现象，比如说浏览器突然就“消失”了。有时候这是因为JavaScript引擎造成的（通常是因为开发人员做了些不那么聪明的事情，但引擎毕竟应该能对付才对）。这个事情从来不是大问题，但即使是因为这方面的改进，浏览器也获得了很好的口碑。

1.4.2 面向对象的JavaScript

大多数JavaScript开发人员的生活从他们发现原型（prototype）那天开始就改变了。一旦他们发现所有JavaScript对象都可以通过其原型来扩展，并且这个功能允许他们创建自定义类，世界就不可能再回到以前那样了。比如，看看下面的代码：

```
var answer = 0;
function addNumbers(num1, num2) {
  answer = num1 + num2;
}
function subtractNumbers(num1, num2) {
  answer = num1 - num2;
}
function multiplyNumbers(num1, num2) {
  answer = num1 * num2;
}
function divideNumbers(num1, num2) {
  if (num2 != 0) {
    answer = num1 / num2;
  } else {
    answer = 0;
  }
}
```

这段代码技术上没什么错误。它能运行。但它组织得足够好吗？我看不怎么样。在全局作用域里的answer变量代码味太浓，每个函数也都如此：都是孤立的函数，都在全局作用域里。下一章里会谈到的一个有助于编写更专业的JavaScript代码的风格就是：不要“污染”全局作用域。

在大多数其他语言里，使用全局变量会被认为是一个坏习惯，因为不在局部作用域里就意味着它们可以在程序的任何部分被修改，导致全局依赖和难以定位问题（通常都是转瞬即逝的，很难觉察）。JavaScript里也是一样。全局作用域里的函数相对没那么让人头疼，不过，缺乏结构意味着在函数之间没有内在的关系，并且没有逻辑上的分组来帮助人们从更高的级别理解它们。

与之对比，让我们看看用更加面向对象的方式重写代码会如何：

```
function NumberFunctions() {
  var answer = 0;
}
NumberFunctions.prototype.addNumbers = function(num1, num2) {
  this.answer = num1 + num2;
}
NumberFunctions.prototype.subtractNumbers = function(num1, num2) {
  this.answer = num1 - num2;
}
NumberFunctions.prototype.multiplyNumbers = function(num1, num2) {
  this.answer = num1 * num2;
}
NumberFunctions.prototype.divideNumbers = function(num1, num2) {
  if (num2 != 0) {
    this.answer = num1 / num2;
  } else {
    this.answer = 0;
  }
}
NumberFunctions.prototype.toString = function() {
  return this.answer;
}
```

要使用这段代码，我们需要做这样的事情：

```
var nf = new NumberFunctions();
nf.addNumbers(2, 1);
alert(nf);
nf.subtractNumbers(10, 3);
alert(nf);
nf.multiplyNumbers(4, 5);
alert(nf);
nf.divideNumbers(12, 6);
alert(nf);
```

这个版本的代码有如下几个优点。

- 没有对全局作用域的污染，因为有NumberFunctions函数。对于answer变量来说，这是最重要的。因为它是使用var关键字声明的，因此就不能从类的外部访问，只有NumberFunctions的函数可以修改它。
- 所有函数实际上都是NumberFunctions类的成员，因此构造了一个清晰的关系。
- 基本的面向对象：数据和操作数据的函数都封装得很好。

所有这些在其他现代语言里都很平常，但是的确花了不少时间才进入JavaScript。

不过，面向对象不是JavaScript向现代化进化的最后一站。还有其他几个概念需要提一下。

1.4.3 “负责的”JavaScript：迹象和前兆

“负责”这个词用在编程语言上看上去挺奇怪的。毕竟我们讨论的不是手枪也不是核武器。但它确实是时至今日JavaScript都仍然严重缺乏的一个非常重要的概念。

你可能已经听说过柔性衰减（graceful degradation）的概念。它是一个针对某个版本的浏览器设计的网页应该对旧一些的浏览器版本采取柔性衰减的策略：即使是不够好，也至少能用。同样的概念可以（而且也应该）适用在JavaScript中。[①]

一个更现代些的术语是不唐突的JavaScript。其实就是精减的柔性衰减，但是它也涵盖了其他的领域，比如使JavaScript做的页面让残障人士可用，创建可以经受时间考验的代码，以及把脚本与页面的标记和样式分离等。

另外一个经常提及的因素是让JavaScript提升网页浏览的体验，但是这个因素很少被人注意（这也是“不唐突”这个词用在JavaScript上的含义之一）。用户期待在现代的Web用户界面上有一定的交互和独立能力，JavaScript自然可以帮助做到这些了。但是一旦用户注意到这些，并且如果这些事情阻碍了他们使用，那你的代码可就侵犯了他们的体验了。有很多很好的条律可以界定帮助用户起飞的特性和用户讨厌的特性。

合适的错误处理也是负责的JavaScript的原则之一。以前的JavaScript的错误处理，大多都只是让浏览器显示一条错误信息。后来，人们发现可以钩上错误处理机制，然后显示一条错误信息，但仍然

① 柔性衰减最初出现在工业界，表示系统不会因为个别元器件的单点失效而导致整个系统的失效，还能保持一定的运转状况和功能。比较著名的例子有伽利略号木星探测器，后来在计算机里得到广泛应用，比如大型集群式搜索服务：使用大量服务器对数据进行分块分片提供服务，单点失效只会导致少量数据的缺失，但并不会导致服务的停止。——译者注

属于那种错误反馈类型，跟随便地说一句“不好意思，出错了，你完了”没什么区别。现代的JavaScript实现提供了更好的处理错误的办法，使用一种内置于语言的机制让你的代码可以继续运行，并且从异常中恢复出来。这么处理可以让代码更加健壮，让用户更愉快。

最后，尽管严格地说不是JavaScript自身的功能，但现在的开发工具比过去丰富很多。各种各样的浏览器插件和扩展，虽然不能说让编写JavaScript是一个愉快的过程，但至少也可以说没那么痛苦了。甚至有些强大的商业工具给JavaScript提供了完整的环境。大多数现代的IDE本身就支持JavaScript，把它当头等公民看待。

如果我的话看上去只是像在吹捧，那就对了！我保证下一章会更深入地讨论这些观点。我只是想在本章最后一节吊起你的胃口，告诉你后面跟着的是什么。坚持阅读下去……这绝对是一次有趣的体验！

1.5　小结

本章介绍了JavaScript的起源——它是如何从不那么光彩的早期进化到如今的专业水准的JavaScript。我们谈了一些早期JavaScript开发人员面临的问题，以及它们是如何逐步得以解决的。然后我们浏览了一下JavaScript走到今天的道路：它是如何变得更干净、更少侵略性和整体上更优雅的！下一章里，我们要了解一下，较之以前，现在的人们是如何更成熟地使用JavaScript的。

第2章 成功的JavaScript开发者的7个习惯

本章继续讨论第1章开始所讨论的内容，并详细讨论是什么使得JavaScript成为一流的语言。我们将关注于面向对象技术，还有一些让人头疼的东西，如不唐突的JavaScript、柔性衰减等，讨论如何使Web应用程序即使在用JavaScript的时候依然可用。（可不是简单的工作！）我们会研究错误处理和调试技术，因为有时脚本（噢，是经常的）运行得并不正确。我们还要看一些可以供你使用的工具，它们会使得使用JavaScript更顺手。最后，我们快速浏览现今最流行的一些JavaScript库，谈谈为什么你会发自内心地想要使用它们。这里会涵盖很多东西，那么我们开始吧！

JavaScript：让用户眼中看到的都是做得正确的

2.1 更多面向对象的 JavaScript

当很多JavaScript程序员开始使用JavaScript的时候，他们通常并没有意识到这个语言提供了一些面向对象的概念。当然，JavaScript完全不要求使用对象！[①]

① 其实它隐含地会使用对象，因为JavaScript在很多场合会使用内置对象，但是你的代码自身不必是面向对象的。

条条大路通罗马，同样，在JavaScript中，也有不止一种的方法来创建对象。

2.1.1　简单的对象创建

可能最简单的创建对象的方法是用一个新的Object开始，然后向其中添加内容。想要创建一个新Object，你只要这么做：

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

变量newObject现在指向一个Object的实例，Object是JavaScript中所有对象的基类。要给它增加一个元素，比如说，增加一个叫firstName的元素，你要做的只是：

```
newObject.firstName = "frank";
```

从代码中的这个位置开始，newObject.firstName就将含有一个值"frank"，除非它在后面被修改了。你也可以很容易地添加函数：

```
newObject.sayName = function() {
 alert(this.firstName);
}
```

现在newObject.sayName()调用的结果，是弹出一个“frank”的警告信息。与大多数成熟的面向对象语言不同的是，在JavaScript中，你不必为一个对象实例创建类或蓝图。你可以像这里所写的那样，在运行时创建它。在对象的整个生命周期中都可以这么做。在网页中，这就意味着你可以在任何时候给对象添加属性和方法。

事实上，JavaScript实现只是把所有对象当作关联数组。然后给数组加上了一个面具，使它的语法看起来更像Java或C++，使用点分隔表示法。为了强调这一点，你可以像这样检索newObject中firstName字段的值：

```
var theFirstName = newObject["firstName"];
```

同样，可以这样调用sayName()函数：

```
newObject["sayName"]();
```

这个简单的东西可以算是很多强大功能的基础。比如，假如你想基于某种逻辑调用某个对象的方法又如何呢？噢，你可以这么做：

```
var whatFunction;
if (whatVolume ==1) {
  whatFunction = "sayName";
}
if (whatVolume == 2) {
  whatFunction = "sayLoudly";
}
newObject[whatFunction]();
```

假设我们已经将函数sayLoudly()添加到newObject，这个函数在firstName字段中的alert()之前调用toUpperCase()。然后我们可以基于一个变量的值，来控制对象“大声”（全部大写）地或者“温柔”（如上所示，全部小写）地说出名字。

当向一个对象添加函数的时候，你可以使用已存在的函数。作为例子，让我们来继续添加那个

sayLoudly()函数：

```
function sayLoudly() {
  alert(this.firstName.toUpperCase());
}
newObject.sayLoudly = sayLoudly;
```

请注意这里this关键字的使用方法。可以说，它指向的对象将会在运行时动态计算。因此，在这个例子中，this指向的是函数sayLoudly()作为其成员的那个类——这里是newObject。有意思的是，如果sayLoudly()完全是另一个对象的一部分，关键字this就会引用另一个对象。这种运行时绑定也是JavaScript面向对象实现的一个非常强大的特性，因为它允许代码的共享，本质上来说，是一种继承的形式。

2.1.2 使用JSON创建对象

由于JSON（JavaScript Object Notation，JavaScript对象表示法）在Ajax请求中的使用而被越来越广泛的关注，所以很多人了解了它。然而，一些人还并不知道JSON实际上是JavaScript规范中的一个核心部分，而且它在Ajax登台之前就已经存在了。它最初的目的是为了快速简便地定义复杂的对象关系图，也就是那些嵌套于其他对象中的对象的实例。虽然它看上去是如此的简单，但它允许了创建对象的另外一种方式。

还记得我们曾说过，JavaScript里的对象只是隐藏在面具下的关联数组吗？这就是JSON可运转的因素。让我们来看看如何使用JSON来创建前面例子中的newObject：

```
function sayLoudly() {
  alert(this.firstName.toUpperCase());
}
var newObject = {
  firstName : "frank",
  sayName : function() { alert(this.firstName); },
  sayLoudly : sayLoudly
};
```

使用JSON和定义一个数组非常相似，除了你需要使用花括号而不是方括号。注意函数可以是内联的，也可以引用外部函数。（看到sayLoudly : sayLoudly可能有一些困惑，但是JavaScript理解为第一个sayLoudly是对象的一个成员，而第二个sayLoudly是对一个已存在的对象的引用。）

在JSON中，你可以随意地嵌套对象定义来创建对象的层级关系。例如，我们向newObject中添加一个名为LastName的对象：

```
function sayLoudly() {
  alert(this.firstName.toUpperCase());
}
var newObject = {
  firstName : "frank",
  sayName : function() { alert(this.firstName); },
  sayLoudly : sayLoudly,
  LastName : {
    lastName : "Zammetti",
    sayName : function() { alert(this.lastName); }
```

```
  }
};
```

然后，你可以通过下面的调用来显示姓的部分：

```
newObject.LastName.sayName();
```

2.1.3　类的定义

在JavaScript中，其实所有的东西都是一个对象。这是事实，只有一小部分例外，比如一些内置的原语。这部分讨论的最重要的一点是，函数本身就是对象。你已经了解到如何创建一个Object的实例并向其中添加属性和方法了，但是那意味着每次创建对象的一个新的实例时，你实际上都需要从零开始创建。这里肯定有更好的方法，不是吗？当然有：创建一个类。

在JavaScript中，类实际上就是一个函数。这个函数同样被当作类的构造函数来提供服务。那么，例如，让我们把newObject写作一个类，重命名为newClass：

```
function newClass() {
  alert("constructor");
  this.firstName = "frank";
  this.sayName = function() {
    alert(this.firstName);
  }
}
var nc = new newClass();
nc.sayName();
```

执行这段代码的时候，你会先后看到两个警告信息：首先，当执行到var nc = new newClass();这行的时候，弹出“constructor”，然后，当执行到nc.sayName();这行的时候弹出“frank”。你想要创建多少newClass的实例就可以创建多少，而且它们将会含有同样的属性和方法。一旦创建，它们还将弹出同样的警告信息，firstName含有同样的初始值。简而言之，你为创建newClass对象创建了一个蓝图，你定义了一个类。

但是，由此引出的一个问题是，newClass的每一个实例都含有firstName的一个副本和sayName()方法的一个副本，那么每个实例都占用了更多的内存。newClass的每个副本都有各自的firstName副本，这个可能是你想要的，但是如果所有的实例可以共享相同的sayName()副本的话，是不是更好了，能否节省些内存？很明显，在这个例子中，我们并不能有大量的内存节省，但是你可以设想一些更大的代码的情况，那些场合下，很可能差别就比较大了。幸运的是，有一种方法可以这样做。

2.1.4　原型

JavaScript中的每一个独立的对象都有一个与之关联的原型（prototype）属性。在我所知道的其他语言中，并没有完全与prototype相等的东西，但是它可以被看作一个简化的继承形式。基本上，它工作的方式是：当你构造一个对象的新实例时，定义在对象的原型中的所有属性和方法，在运行时都会附着在那个新的实例上。

我知道，第一次看到这个概念时，可能有一些难以理解，但是幸运的是，它很容易演示：

```
function newClass() {
  this.firstName = "frank";
}
newClass.prototype.sayName = function() {
  alert(this.firstName);
}
var nc = new newClass();
nc.sayName();
```

执行的时候，这个代码会弹出同样的警告信息说“frank”。与前面的例子不同的是，无论你创建了多少个newClass的实例，在内存中sayName()函数只会有一个单独的实例。这个方法实际上是附加在每个实例上，而且this关键字还是在运行时被计算的。所以this通常指向它所属的那个特定的newClass实例。例如，如果你有两个newClass的实例分别叫做nc1和nc2，然后一个对nc1.sayName()的调用将this指向nc1，一个对nc2.sayName()的调用将this指向nc2。

2.1.5 你应该使用哪种方法呢

前面所说的每个方法都有各自的优点和缺点，而且我怀疑是否存在孰好孰坏的共识。它们在功能上都是相同的，所以很大程度上它取决于你更喜欢代码看起来像什么样子。也就是说，我想有一些常用的准则来帮助你做决定。

最重要的一个可能是，如果你要创建一个类，这个类非常大，而且它可能会有复杂的实例，那么几乎可以肯定要使用原型的方法。这样可以带来最好的内存使用效率，这通常是一个重要的目标。

如果你要创建一个单独的类而且知道这个类将只有一个实例，我个人会倾向定义一个类。对我来说，这样的代码是逻辑化最强、最类似于更全面的面向对象语言的，因此可能更易于项目中的新开发人员理解。

如果(a)你的对象层级关系嵌套层次很多和/或(b)你需要在一个动态方式中定义一个对象（一段逻辑代码的输出结果），那么，JSON方法可能是一个好的选择。如果需要将对象序列化并且通过网络进行传输，JSON也几乎非常明显是首选。如果你需要重构一个从服务器传送来的对象时也是如此。我怀疑做这些事情的时候，没有比JSON更简单的方法，这不是小范围的怀疑，而是因为这些事情就是设计JSON的目的。

2.1.6 面向对象的好处

无论你选择哪种方法，将代码面向对象化都有很多好处。一个重要的好处就是，每一个对象本质上就是一个命名空间。可以用此来模拟Java和C#的包，就像在下一章中会看到的。

另一个好处是可以使用对象来隐藏数据。看看下面的代码：

```
function newClass() {
  this.firstName = "Frank";
  lastName = "Zammetti";
}
var nc = new newClass();
alert(nc.firstName);
alert(nc.lastName);
```

执行这段代码会有两个警告信息：第一个说“Frank”，第二个说“undefined”。这是因为lastName

字段在newClass的实例外是不可访问的。请注意字段定义的不同。任何使用this关键字定义的字段，比如firstName那样，都可以在类之外访问。而任何没有使用this定义的字段，都只能在类的内部访问。这个规则对于方法也同样适用。

同样，不要忘记JavaScript的内置对象可以通过使用它们的原型来扩展，事实上，JavaScript的名为Prototype的库就是在做这个事，这些你将会在2.6节中看到。但是，如果不小心的话，你真的可能把事情搞糟，所以扩展内置对象时要谨慎。

你可以从其他的对象中“借”函数，并把它们添加到自己的对象中。例如，假设你想要通过输出newClass本身来显示newClass的firstName字段。为实现这个目标，你实现了一个toString()函数。再假设你还想在它上面使用String对象中的toUpperCase()函数。很简单，我们可以这么做：

```
function newClass() {
  this.firstName = "frank";
  this.toUC = String.toUpperCase;
  this.toString = function() {
    return this.toUC(this.firstName);
  }
}
var nc = new newClass();
alert(nc);
```

执行这段代码，结果是一个警告提示说“FRANK”。注意调用了toString()函数，但并不是作为firstName String对象的一个方法。取而代之的，是通过指向它的一个引用调用的，这个引用是newClass的属性之一，名为toUC()。这是一个很好用的功能，尤其是当你创建自己的对象之后，又想创建一个新的，并使用原来的代码。那你不用复制、剪切以及粘贴，只需引用其他类里的已存在的方法就可以了。

2.2　柔性衰减和不唐突的JavaScript

不唐突的JavaScript是一个最近才流行起来的术语。简单地说，它是一种趋势，网页中的JavaScript以一种不影响页面的方式来写。

不唐突的JavaScript的基本原则非常简单，可以被归纳为以下几点。

- 保持JavaScript独立。
- 通常允许柔性衰减。
- 从不使用浏览器嗅探脚本来判断一个浏览器的能力。
- 任何情况下，都不要写浏览器不兼容的代码，具体说，就是别写浏览器相关的JavaScript代码。
- 合适的作用域变量。
- 对于可访问性，避免需要鼠标事件触发的触发器。

但是，不唐突的JavaScript术语对于不同的人也可以有不同的意思。一些人喜欢扩展规则并使其更加严格。另一些人喜欢修剪规则并使其更加灵活。关键是使用我们从以前的错误中学到的方法来实现JavaScript。

现在让我们来看看每一个基础规则的详细内容。

2.2.1　让JavaScript保持独立

这个观点是把JavaScript当作应用程序的一层来处理，并且使用定义明确的交互点，试着使它尽可

能地独立。例如，总是从外部文件导入JavaScript，并没有任何的JavaScript嵌在HTML中。

我认为，这是一条比较灵活的规则。例如，主页上的一些配置变量不会令人沮丧，但是很多人甚至会要求把这种变量也放外边去。但是基本观点是说：最大限度地保持脚本独立。

考虑一下CSS（Cascading Style Sheet，层叠样式表）。你已经习惯于外部的样式表了，对吗？那么用同样的方法来看待JavaScript。这样做逻辑上把页面分成了几个部分，我们就可以很容易地找到自己感兴趣、想要修改的地方。同时它还给你留下一条重用的途径。外部保存的脚本有更好的机会在其他页面、其他站点以及其他项目中重用。虽然不能保证肯定能被使用，但它往往是有帮助的。

一些人甚至提倡通过脚本来添加事件处理函数。通常，这么做的原因是把脚本和标记完全分离，避免在修改函数名之后要修改代码中很多的地方。我不完全同意这个建议，主要是因为，一个事件处理函数是对于一个指定元素特定的，所以为什么不直接把它附加在那个元素上呢？对于我来说，如果我需要改变一个事件处理函数，直接找到那个元素要比找出哪个外部.js文件包含那段代码更加容易。不过有一个例外，如果处理函数是共享的（被多于一个的元素使用），这种情况下，我就会将它外部化。

到底什么样才是最好的，留给你自己去归纳。然而，我非常鼓励你保持事件处理函数尽可能地小，无论把它们放在哪。它们应该只是调用一些大块的代码或者执行一两行语句。如果你想办法让事件处理函数与元素在同一行内，那会是一个很好的主意。

2.2.2 允许柔性衰减

离开了JavaScript，页面也还应该工作，即使是在以衰减的形式。一个很好的例子是表单验证。不要只使用调用一个提交表单的函数的按钮，因为如果离开了JavaScript，这个表单就不能提交了。比如，尝试下面代码清单2-1的代码，并观察一下如果禁用了JavaScript会发生什么。

代码清单2-1 不能衰减的表单提交

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

      function doSubmit(inForm) {
        if (inForm.firstName.value == "") {
          alert("You must enter a first name");
          return false;
        }
        if (inForm.lastName.value == "") {
          alert("You must enter a last name");
          return false;
        }
        inForm.submit();
        return true;
      }

    </script>

  </head>
  <body>
```

```
    <form name="test" action="#" method="post">
      First name: <input type="test" name="firstName">
      <br>
      Last name: <input type="test" name="lastName">
      <br>
      <input type="button" onClick="doSubmit(this.form);" value="Submit">
    </form>

  </body>
</html>
```

如果禁用JavaScript后运行代码清单2-1。你将看到什么都没有发生，因为表单的提交依赖JavaScript的执行。这显然是不好的。

取代它的方法是，在onSubmit事件的响应中进行验证。使用这种方法，如果JavaScript启用，用户可以从客户端验证中受益。但是如果JavaScript被禁用了，表单仍然可以被提交。代码清单2-2展示了它是如何工作的。

代码清单2-2　柔性衰减的表单提交

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

      function doSubmit(inForm) {
        if (inForm.firstName.value == "") {
          alert("You must enter a first name");
          return false;
        }
        if (inForm.lastName.value == "") {
          alert("You must enter a last name");
          return false;
        }
        inForm.submit();
        return true;
      }

    </script>

  </head>
  <body>

    <form name="test" action="#" method="post"
      onSubmit="return doSubmit(this);">
      First name: <input type="test" name="firstName">
      <br>
      Last name: <input type="test" name="lastName">
      <br>
      <input type="submit" value="Submit">
    </form>
```

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

顺便说一下，一个致命的错误是：使用单纯的客户端验证，并假设从客户端来的任何数据都是好的。事实上，你的系统通常应该被设计成认为从客户端来的数据都是坏的。在客户端做验证是完全可行的。不过，几乎没人愿意使用的系统把客户端验证作为唯一验证方法。

好，我们来谈一些观点。有人认为，我们应该在所有Web应用上附加一些不唐突的原则和一些合理的、柔性衰减的安全约束。我并不同意这种观点，甚至可以说，我认为这种观点完全是无原则的。

玩一下和第11章里那个项目一样的网络游戏，你能想象没有JavaScript的时候能写这种东西（游戏）吗？没JavaScript就几乎等于要求Electronic Arts公司用Logo语言开发下一个版本的疯狂足球，或者是要求Bungie用HTML写Halo（一种很酷的3D游戏），或者是要微软用一种既不支持逻辑分支、循环、变量，也不支持数据结构的语言写一个Windows版本。

问题在于游戏需要可执行的代码，如你在第11章看到的那样。（看去吧，满足自己的好奇心，浏览一下！我们等你回来的时候再继续本章。）你认为在这样的项目里柔性衰减是合理的目标吗？除了对页面的衰减只能说一句："抱歉，本游戏需要打开JavaScript支持。"我当然不能同意这么做。JavaScript在这种应用里是否应该只是"可选的"东西？我毫不怀疑在某个地方的某个人可能已经写了一个网络游戏，只需要HTML并且柔性衰减到不需要JavaScript。不过我也毫不怀疑如此不唐突的东西绝对是一个超级的例外。

我们所处的时代是"RIA（Rich Internet Application，富因特网应用）"时代，RIA开始统治因特网。比如说，Google，现在就处在把一整套office软件套件放在网上给所有用户使用的早期时代[①]。你认为所有开发人员都会力图遵守所有不唐突的原则吗（除了少数最低限度的之外）？并且这么做了之后还能用？几乎可以肯定地说这不可能，因为对于这种高端应用来说，这就是不可能的任务。让我说得更明白些：在RIA世界里，大多数这里描述的不唐突的要求仍然是可为的。只是其中的某些可能不能做到罢了。

这就是网站和Web应用之间的差距。从某种意义上来说，网站的目的是散播信息。它只有很有限的用户交互的需求，通常来说，简单的HTML表单就足够好用了。在一个网站里，所有这里描述的规则几乎都应该遵守，而且更重要的是，这些规则是可以遵守的。相对来说，Web应用则更复杂并且需要更高级的用户交互。它们就是为了替换肥客户端、应用程序等那些用户已经习惯了几十年的东西。这些用户需要功能强大同时又简洁的动态交互界面、铃声和报警声，以及一些不写代码就是不能实现的特性——并且有些代码必须在客户端里组织起来。在这种场合，我的观点是：遵守这些规则就是不合理的，会导致无数项目的失败。

不过，对于任何在Web上给公众使用的东西，你都毫无疑问地应该争取优秀的可用性、合理的柔性衰减，以及所有其他不唐突的JavaScript目标。对于无法实现这些目标的地方，我们应该有很清晰、很合理的原因，并且绝对可以确信：如果要实现这些目标，你的应用就无法实现。

所以，简而言之，我认为你应该审视一下你正在做什么，以及你的目标是什么，并决定这些指导哪些应该遵循。不要犯错误，我相信你会试着遵守全部这些规则。但是我估计那有时并不可能。再说一次，这是我个人的意见，请根据自己最好的判断形成自己的意见。

---

① Google的应用叫Google Docs & Spreadsheets。你可以在http://docs.google.com尝试一下。

2.2.3　不要使用浏览器嗅探例程

与其使用浏览器嗅探脚本去确定一个浏览器的能力，还不如检查对象的存在和对象的能力。这个问题的补充是，只是由于开发者的懒惰而引起的JavaScript错误，比如在访问一个给定对象之前没有检查它是否存在，是无理且不好的。

一个例子，看看下面的代码：

```
function setContent(inObj, inContent {
  inObject.innerHTML = inContent;
}
```

这里，如果对象inObj并不支持innerHTML，就会出现一个错误。可能由于innerHTML不是DOM中一个标准的部分（虽然，实际上我不知道有哪个浏览器没实现它）。重写这部分来避免这个错误很简单：

```
function setContent(inObj, inContent {
  if (inObj.innerHTML) {
    inObject.innerHTML = inContent;
  }
}
```

通常像这样检查是很好的习惯，它实际上是确定一个浏览器是否支持一个特定的能力。这样做的一个最好的例子是基本的Ajax编程，你需要一个XMLHttpRequest对象的实例。不幸的是，不同的浏览器使用不同的方法来支持这个对象的使用（第12章将详细讲述Ajax，所以如果这对你来说是新的，也不必担心其细节）。然而，你可以检查不同的对象，并基于它们的存在或不存在，来分支代码，像这样：

```
var xhr = null;
if (window.XMLHttpRequest) {
  xhr = new XMLHttpRequest();
} else if (window.ActiveXObject) {
  xhr = new ActiveXObject("Microsoft.XMLHTTP");
}
```

当然，如果能写根本不需要分支的代码会更好，但是那通常不太可能。浏览器嗅探是个坏主意，因为，很多开发者多年来的体会是，你总是需要保持嗅探代码的更新，以便让它能识别新的浏览器。对象存在性的检查就没有那么脆弱了，通常不需要重写代码来处理新的浏览器，所以这个技术比浏览器嗅探好得多。

2.2.4　不要写浏览器相关或者语言相关的JavaScript代码

在任何情况下，你都不应该创建浏览器不兼容的JavaScript，或者更准确地说是浏览器相关的代码，除非有非常好的理由。

很明显，这是一个对每个人编写JavaScript的都适用的规则，即使只是相对简单的代码。一个简单的事实是，在现代浏览器中的JavaScript非常接近100%兼容。虽然这里或那里还有异常，但是你会发现，实际上，绝大部分差别都是关于DOM的不同，以及在特定浏览器中使用DOM的方法。所以，虽然这个原则说的是跨浏览器的JavaScript的问题，但实际上，它和访问DOM关系更密切。

一个小例子是，你应该不再需要检查像document.layers或document.all这样的东西来判断如何在页面上访问一个元素。基本上所有的现代浏览器都支持document.getElementById()，这是标准指定的兼容方法，是你应该使用的。任何时候你发现自己正在编写一个特殊专用的JavaScript代码，或者是为一个特殊浏览器编写代码，那么问问自己，(a)是否有一个兼容标准的方式来编写这些代码？(b)它将会跨所有你想支持的浏览器吗？当你找到这些例外，并在注释中很清楚地写明你为什么这么做，这会减少你在修改的时候的烦恼，同时也会提醒你，代码的哪部分需要在以后检查一下，是否可以升级到依从标准的方法。

2.2.5 合适的变量作用域

变量应该是局部的，除非它确实具有全局的意义。尤其是在使用Ajax时请小心。因为在一个异步的世界里，全局变量的使用可能引起许多难以调试的问题。

举一个糟糕的变量作用域的例子，请看代码清单2-3。

代码清单2-3 糟糕的变量作用域

```
<html>
  <head>

    <script>

      var fauxConstant = "123";

      function badFunction() {
        fauxConstant = "456";
      }

      function goodFunction() {
       var fauxConstant = "456";
      }

      function testIt() {
        alert(fauxConstant);
        goodFunction();
        alert(fauxConstant);
        badFunction();
        alert(fauxConstant);
      }

    </script>

</head>

<body>
  Three alerts will follow... the first should and does say "123."
  The second should and does say "123" again. And the third should say
  "123" but instead says "456."
  <br><br>
```

```
    <input type="button" value="Click to test scoping" onClick="testIt();">

  </body>
</html>
```

在这个例子中，注意显示出来的最后一个值，是如何因变量作用域设置不同而出错的。原因是goodFunction()和badFunction()都将创建一个和全局变量有相同名字的变量，然后在局部使用它，但不改变全局版本的值。如你所猜到的，badFunction()并没有那么工作，它修改到了全局版本。如果badFunction()中的fauxConstant是在局部声明的，就像在goodFunction()中那样，那这个问题就可以避免。如果确实就是想有一个全局变量，那么goodFunction()就是错的，因为它声明了一个局部变量。然而，名字fauxConstant应该是一个暗示，它不希望在声明和初始化后，值会改变。由于在JavaScript中并没有真正的常量，我们只能假造一个——因此使用名字fauxConstant。总之，在任何可能的时候都要把变量限制在局部作用域。

另外一点是，记住任何声明在函数内部的、没有使用关键字var的JavaScript变量，都将继续在这个函数外部存在。这经常会导致一些难以调试的问题，所以给自己一个方便，永远使用关键字var，除非你非常确定地知道你有一个不使用（var）的原因。

2.2.6　别用鼠标事件来触发需要的事件

为了提高可访问性，你应该避免那些需要鼠标事件做驱动的触发器。虽然没有严格定义说明onChange是一个鼠标事件，但是经常被误用。例如，我们都见过一些网站，使用一个<select>，当它变化后，网站定向到一个新的页面。就像代码清单2-4的例子一样。这样通常是不好的，因为这样的页面离开了鼠标可能不能使用，这意味着那些有某种残疾的人会很难、甚至无法使用你的网站。

代码清单2-4　不可用的页面跳转

```
<html>
  <head>
  </head>
  <body>
    <select onChange="alert('Change to page ' + this.value);">
      <option value="page1.htm"></option>
      <option value="page1.htm">Page 1</option>
      <option value="page2.htm">Page 2</option>
    </select>
  </body>
</html>
```

相反，应该依靠可以用键盘激活的事件，如代码清单2-5所示。

代码清单2-5　更可用的页面跳转

```
<html>
  <head>
  </head>
  <body>
    <select id="theSelect">
```

```
        <option value="page1.htm"></option>
        <option value="page1.htm">Page 1</option>
        <option value="page2.htm">Page 2</option>
      </select>
      <br>
      <input type="button" value="Click to change pages"
        onClick=
        "alert('Change to page ' + document.getElementById('theSelect').value);">
    </body>
  </html>
```

这个例子在<select>旁边放置了一个按钮，这个按钮激活页面的改变。这个可以使用鼠标和键盘简单地激活，极大地提高了页面的可访问性。

2.3 并不只是为了秀：关注可访问性

在现代RIA中，尤其是那些使用Ajax技术的，对于残障人士的可访问性，是一个很大的难题。任何不这么说的人都很有可能想卖给你一个你并不想要的解决方案。事实是，可访问性问题日益严重，并没有缩小，这些应该归于“现代”Web应用的天性。

可访问性通常可以归纳成两个主要问题：帮助视觉障碍者和帮助那些行动失调的人。那些有听力问题的人，Web应用的问题会少一些，不过每天都会在网上出现更多的富多媒体的应用，听力障碍的用户面临的问题也会越来越多。那些行动失调的人会很关心像键盘快捷键这样的东西，因为他们趋向于使用比移动鼠标更方便的方法（并且通常比为实现特定的设备而进行的鼠标移动更容易）。

经常被忽视的一个问题是另外的一种视觉障碍：色盲。Web开发者通常做了很好的工作来帮助那些盲人，但他们通常并没有给那些色弱人士同样的关注。重要的是要理解色盲人群通常并不是只能看到黑和白的世界[①]。色盲，或者说是色弱，是与人的眼球的视觉细胞合作的三种颜色感光细胞中的一种有问题。三种颜色感光细胞中的每一种，还有视觉细胞，各自对红、绿、蓝三种波长中的其中一种光敏感。普通的眼球，即这些感光细胞和具有正常功能的视觉细胞，可以让人们看到不同分量的红、绿、蓝颜色混合的结果。那些有色盲症的人，有些时候无法区分这些不同颜色混合的区别，而有时候甚至完全不能识别某种颜色。比如，有些色盲症的人，一个由混有少量红点的蓝色点组成的区域，看起来会成为同一个颜色的点组成的区域。在这里http://colorvisiontesting.com/what%20colorblind%20people%20see.htm，你可以看到很多例子说明颜色缺陷对于有正常视觉的人的影响。

2.4 当生活赐予你葡萄，就酿成酒吧：错误处理

不久以前，在JavaScript中的错误/异常处理，加起来也不过如此：在发生了错误的时候，浏览器给用户显示一个不太令人沮丧的信息。大多数，“真正”的语言都有相当成熟的异常处理机制。但是JavaScript并不是它们中的一员。Java有try...catch块，C++也是，甚至比Java更早。即使是相当垃圾的Visual Basic也有On Error，被称作“非结构化”的异常处理（与之对应的是try...catch，它被称作“结构化的”）。但是即使是它，也比JavaScript需要提供的东西早了很长一段时间。

有时候，一个聪明的JavaScript程序员发现可以借助浏览器的异常处理机制。所以现在，代替如图

① 只能看见黑、灰和白的情况叫全色盲，实际上很罕见。全色盲其实比典型的色弱更容易对付。

2-1所示那样简单的老式的浏览器错误信息的是，一些让人更舒服一点的错误处理版本，如图2-2所示。

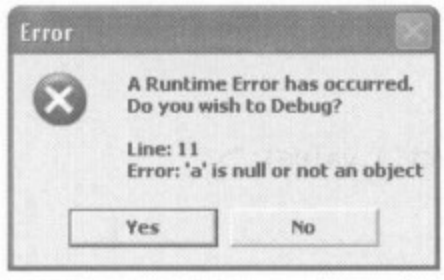


图2-1　一个来自IE的简单的JavaScript错误信息

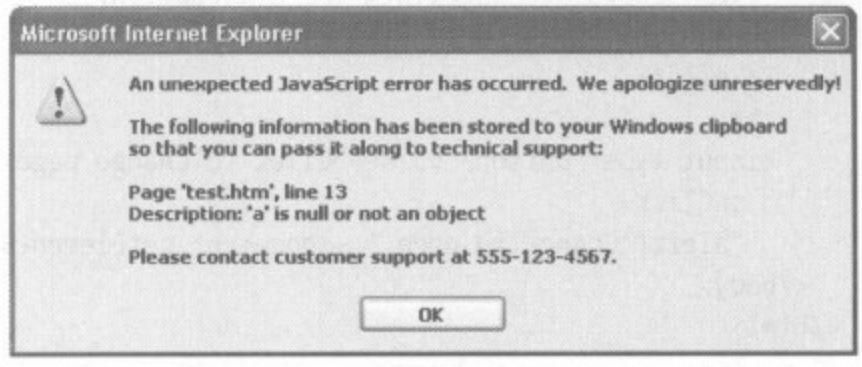


图2-2　一个自定义的JavaScript错误信息

自定义错误处理函数的代码是非常简单的，但仍然是我们的工具箱中非常有用的工具。代码清单2-6展示了生成图2-2中信息的页面代码，以及用于测试它的产生错误的代码。通常，错误经常可以被结构化的异常处理程序来处理，不过有时候，有个类似这样的最后一道错误处理函数也是很好的。实际上，说服自己总是保留这么一个处理函数并不太难，即使你尝试了所有的方法来避免它被调用……当然了，你绝对应该避免它被调用。

代码清单2-6　一个错误处理函数和测试案例的代码

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

      window.onerror = handleError;
      var s = null;
      s.toString();

      function handleError(desc, page, line) {
        s = "An unexpected JavaScript error has occurred.  ";
        s += "We apologize unreservedly!\n\n";
        s += "Page 'test.htm', line " + line + "\n";
        s += "Description: " + desc + "\n\n";
        s += "Please contact customer support at 555-123-4567.";
        alert(s);
      }

    </script>

  </head>

  <body>
  </body>

</html>
```

说明 错误和异常之间的区别经常被一些开发者忽略，但是在你设计代码时，这却是非常重要的区别。错误，是一个条件，你并不预期它发生，并且通常会（甚至是“应该”，这是一个有争议的命题）导致程序崩溃（或者，最好的情况是，生成一个错误提示信息说程序不能继续）。另一方面，异常，是一个状态，你预期会并且可能发生，而且程序应该能够以某种方法处理它并继续运行。你通常会发现一个错误实际上是一个伪装的异常，也就是说，如果你认真想了并且做了合理的计划，就可以像处理其他异常一样处理它。从开发人员的角度来说，它可能需要更多的工作，但那是一张通向编写更健壮代码之路的门票。

那么，在JavaScript环境里，什么是我提到的结构化的异常处理呢？像Java、C++以及很多其他语言，try...catch结构是其核心。代码清单2-7展示了一个try...catch的简单例子。

代码清单2-7 JavaScript的异常处理

```
<html>
  <head>

    <script>

      function test(inVal) {
        try {
          inVal = inVal.toLowerCase();
        } catch(error) {
          alert("An error has occurred.  Error was:\n\n" + error.message);
        }
      }

    </script>

  </head>

  <body>
    <input type="button" value="Test" onClick="test(null);">
  </body>

</html>
```

在这段代码中的异常，应该非常容易找出来：它试图在传入的一个空的字符串上调用toLowerCase()。这可能是一种在设计的时候不会被开发者捕获到的问题，因为调用函数test()的条件可能取决于很多因素（在这里，很明显开发者应该捕获它，因为null是特意传递进去的，但是你知道我是什么意思）。异常往往是一些只在运行时，由一些用户产生的条件触发的东西。不管怎么说，这个例子说明了try...catch如何工作。

总之，开发者应该把一些知道会抛出异常的情况包含在try{}中。后面跟着一个catch{}块，如果一个异常出现在try{}块中，会执行catch{}块。在代码清单2-7的例子中，异常处理代码只是弹出了一个警告信息，有时这样是全部你可以做的事。

2.5　当它并没有向正确的方向发展时：调试机制

朋友们，让我们面对现实吧：我们开发者像那些棒球运动员一样，如果能做到每10次尝试，就有3次正确，那已经非常好了！我的意思是，现代开发模型已经和以前的非常不同了。

我承认，自己并没有经历编程还是一种对耐力的训练的时代，但是我对那个年代有所耳闻。程序员会花费整天的时间在特殊的纸上写程序，尽他们所能地从头到尾考虑每一个最终细节。然后把那些纸送到另一个部门，在那里将程序输入到一个机器中生成打孔纸带。第二天，程序员（有时甚至是另外的整个部门）会把那些打孔卡片喂到计算机中（我还没有说搬着打孔卡片的箱子到处旅行，然后在老板发现之前疯狂地整理成百上千卡片的顺序所花的时间）。然后程序员坐在旁边，等待一些输出，从中确认他们的程序是正确的，或者确认它们是否需要再重新来过。

今天的情况明显好多了，我们通常可以立即得到关于程序的正确性的反馈。事实上，我们经常能获得一些早期的暗示，提示已经犯的错误。纠正它们是一个简单的工作，只需要敲入一些新的代码并点击按钮即可。没错，我们的日子要比早期的日子好过多了。

即便如此，你有几次能做到只不过写了数十行程序，然后第一次运行就运转得十分完美？这是很罕见的。这就是那句俗话“真正的程序员总是只诅咒死气沉沉的东西”诞生的原因！

在JavaScript的世界中，情况以飚车的速度好转。也就是说，在现代Web应用中调试JavaScript，通常不是最令人愉快的体验。它不像穿孔纸带时代那么糟糕，但是通常也没有工作在现代的第四代语言（4GL）那么美好。直到最近，IDE（集成开发工具）才开始逐步在调试能力和静态代码分析能力上开始完全支持JavaScript。所以，我们还需要开发我们自己的调试技术，并学着令它们更加好用。当然，一个称手的调试工具已经不像两三年前那么罕见了，并且，一些技术正开始给使用调试工具让步。

可能最早的调试技术，如果我们可以说它是调试的话，就是“警告调试”。这就导致在代码中，散布着大量alert()的调用来显示各种各样的信息。例如，假设需要调试代码清单2-8中的代码。

代码清单2-8　使用alert()调试

```
<html>
  <head>

    <script>

      function test() {
        var a = 0;
        alert("checkpoint 1");
        a = a + 1;
        alert("checkpoint 2");
        a = a - 1;
        alert("checkpoint 3");
        a = a.toLowerCase();
        alert("checkpoint 4");
      }

    </script>
```

```
  </head>

  <body>
    <input type="button" value="Test" onClick="test(null);">
  </body>

</html>
```

在这个例子中，你知道正在寻找函数中某个位置的错误。所以，在代码中散布大量alert()调用，显示一些检查点的信息。当看这个页面并点击Test按钮时，会得到一系列的弹出框，最终会发生错误。现在知道了，错误发生在显示了“checkpoint 3”和“checkpoint 4”的两个弹出框之间。实际上，你创建了一个类似“单步”调试的东西。这个能用，并且通常让自己这么做的原因是简单，因为这样可以非常快并且高效地完成。同时，它显然不是最好的答案。如果这段代码中包含一个重复了300次的循环会怎么样？真的想点击300次警告的弹出框吗？才不呢！

可供选择的是什么呢？好的，如果在Firefox中开发，你可以使用一个非常好的工具，叫做Firebug，我会在下一节中介绍更多的细节。作为一个预览，Firebug提供将日志记入一个控制台的功能。所以，代码清单2-8中的代码可以被改成代码清单2-9中那样。

代码清单2-9 Firebug控制台日志

```
<html>
  <head>

    <script>

      function test() {
        var a = 0;
        console.log("checkpoint 1");
        a = a + 1;
        console.log("checkpoint 2");
        a = a - 1;
        console.log("checkpoint 3");
        a = a.toLowerCase();
        console.log("checkpoint 4");
      }

    </script>

  </head>

  <body>
    <input type="button" value="Test" onClick="test(null);">
  </body>

</html>
```

现在，如果你看Firebug的控制台，就不用一堆弹出窗口，而是可以在Firebug里显示的信息。亲爱的！注意，试图在IE中运行这段代码会导致错误，因为IE并不知道那个console对象。如果你想在IE中

工作，可以创建下面的代码，并把它添加到页面中（很像一个导入的脚本）：

```
function Console() {
  this.log = function(inText) {
    alert(inText);
  }
}
console = new Console();
```

当然，这就回到了将日志写在一个alert()弹出框中的模式，所以你可能会想要替换成写在页面的一个 中。不过，基本的观点还是如何模仿那个console对象，这是一个小技巧，尽管还不完美。

但是，如果你不在Firefox中工作，或者没有安装Firebug会怎样呢？比如需要将代码发给客户端，但又不能确定那个客户有Firefox和Firebug，又会怎样？好，一个选择是，写一个你自己的简单信息日志记录程序。那实际上是第3章的项目中的一部分，所以我会保留到那时，但完全可以说这是一个相当简单的练习。

不过，一个日志记录程序实际上不就是一个不那么烦人的alert()调试实现吗？“有没有合适的调试器呢？”我听到你在问了。使用C/C++、Java、Visual Basic或者其他语言开发时，IDE中合适的调试器，对吗？如果JavaScript打算和软件巨头玩一把，那么它就必须装备好了来打。好，猜猜是什么？JavaScript调试器过去几乎是一个自相矛盾的事物，但不再是了！现在，有一大堆的选择——有些比其他的好用，有些是免费的，有些不是，但它们确实存在。让我们来聊聊所有的JavaScript编程人员都应该放在自己的工具箱中的调试器和一些其他工具。

2.6　让生活更加美好的浏览器扩展

Web浏览器再也不只是用来呈现标记！现代Web浏览器，实际上已经是运行其他软件的运行时平台了。浏览器在这方面的优缺点不一，但几乎都从某种程度上以扩展的形式提供了一些扩展性。在本节中，我们将要看一些我个人特别喜欢的，特别是对我的日常开发有帮助的扩展工具。当然，我只能介绍现有的扩展中的一小部分。不过我相信，应该能涵盖在每种浏览器中最有用的了（我的意思是对于一个开发者来说），但是请根据你自己的需要来探究和发现，因为还有很多呢！

2.6.1　Firefox扩展

不论你在日常浏览中使用IE、Firefox、Opera或是一些其他的浏览器，我都非常建议你在Firefox中进行主要开发。原因有两个。第一，Firefox比其他浏览器（尤其是比IE）更加遵守标准，所以你在Firefox中开发的代码和标记会更加遵循标准。第二，Firefox拥有一些如今能够找到的最好的客户端开发工具，并且它们几乎全部是免费的。

可以通过打开Firefox，点击“工具”菜单，选择“附加软件”来找到所有Firefox扩展（或插件，它是插件的另一种叫法）。这样会打开一个工具条，你可以看到一个看起来像旁边有黑色下箭头的小灰色齿轮的图标。点击那个齿轮，打开一个菜单，在靠近下方的位置列着Firefox 扩展。点击那个条目，你就会被带到Firefox插件的页面。或者，你可以简单地导航到https://addons.mozilla.org/ firefox/extensions。在插件页面，你可以浏览所有可用的插件。

现在要说一些我个人认为最有用的扩展，但是非常建议你花一些时间去浏览它们，因为那里还有

更多的东西。

1. Venkman

像Venkman这么功能强大的调试器竟然是100%免费的，确实很令人惊讶！在这里，可以得到所有你最喜欢的调试技巧，包括调用栈导航，监视指定变量值的能力，在代码中设置断点，以及实时修改变量值来观察代码的反应。如图2-3所示，Venkman看起来非常像那些可能在服务器端使用过的调试器。

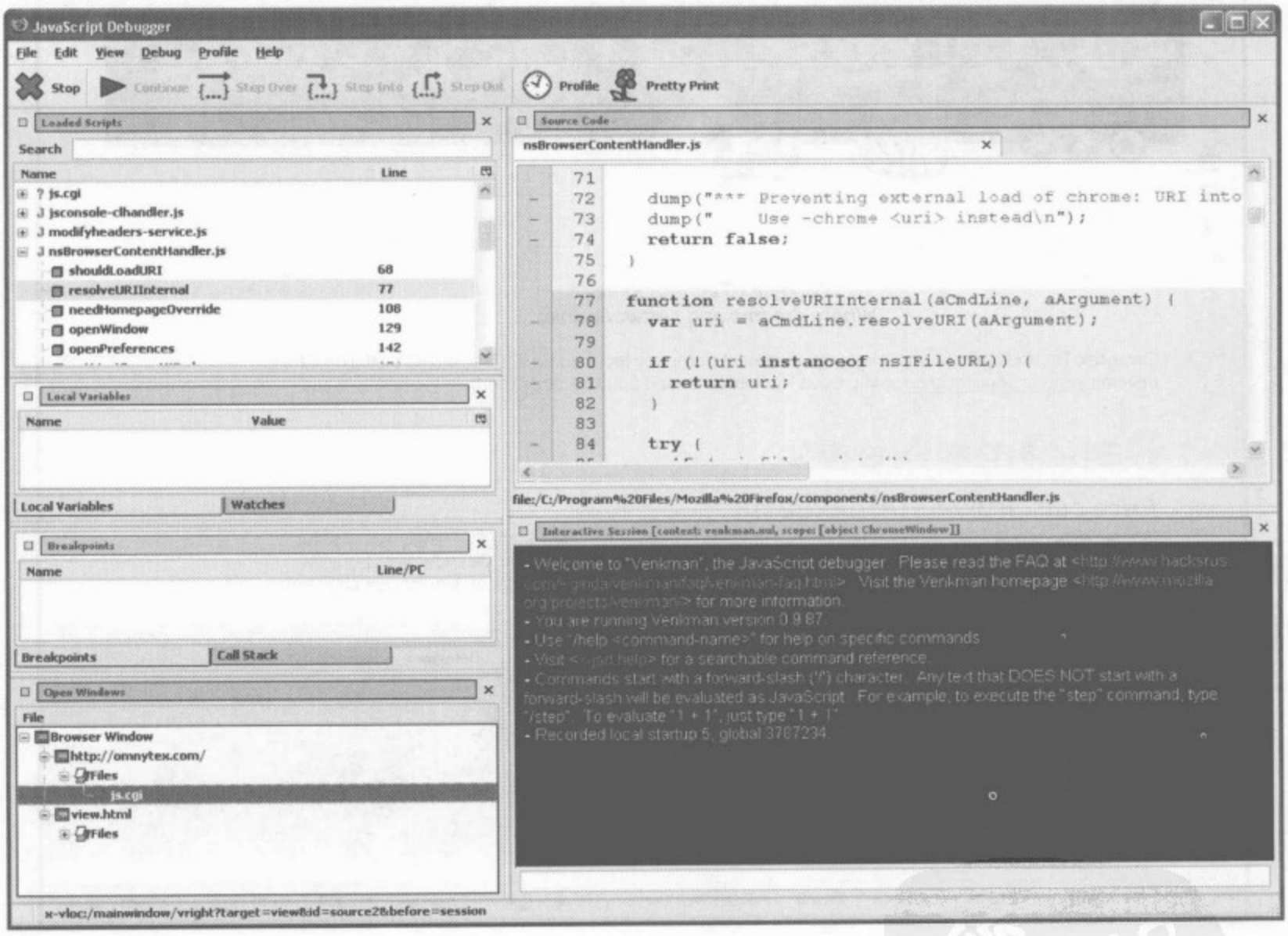


图2-3 Firefox的Venkman调试器

2. Firebug

Firebug非常迅速地获得了当今Firefox扩展中最流行的开发者插件之一的称号。事实上，我们中的很多人，几乎已经只使用Firebug来支持客户端开发了。几乎不需要其他什么！

Firebug提供了很多不同的功能，它们都被包在一个精巧的包里。比如，Console标签，你可以在图2-4中看到，有过滤功能，可以显示各种各样的错误和警告。它的一个非常好的功能是，你可以展开错误并查看整个栈轨迹。那个列表中的每一项都是可点击的，并且直接带你进入出错的行。同样，你可以只像下面这么做来使用这个控制台：

```
console.log("message");
```

这太好用了！

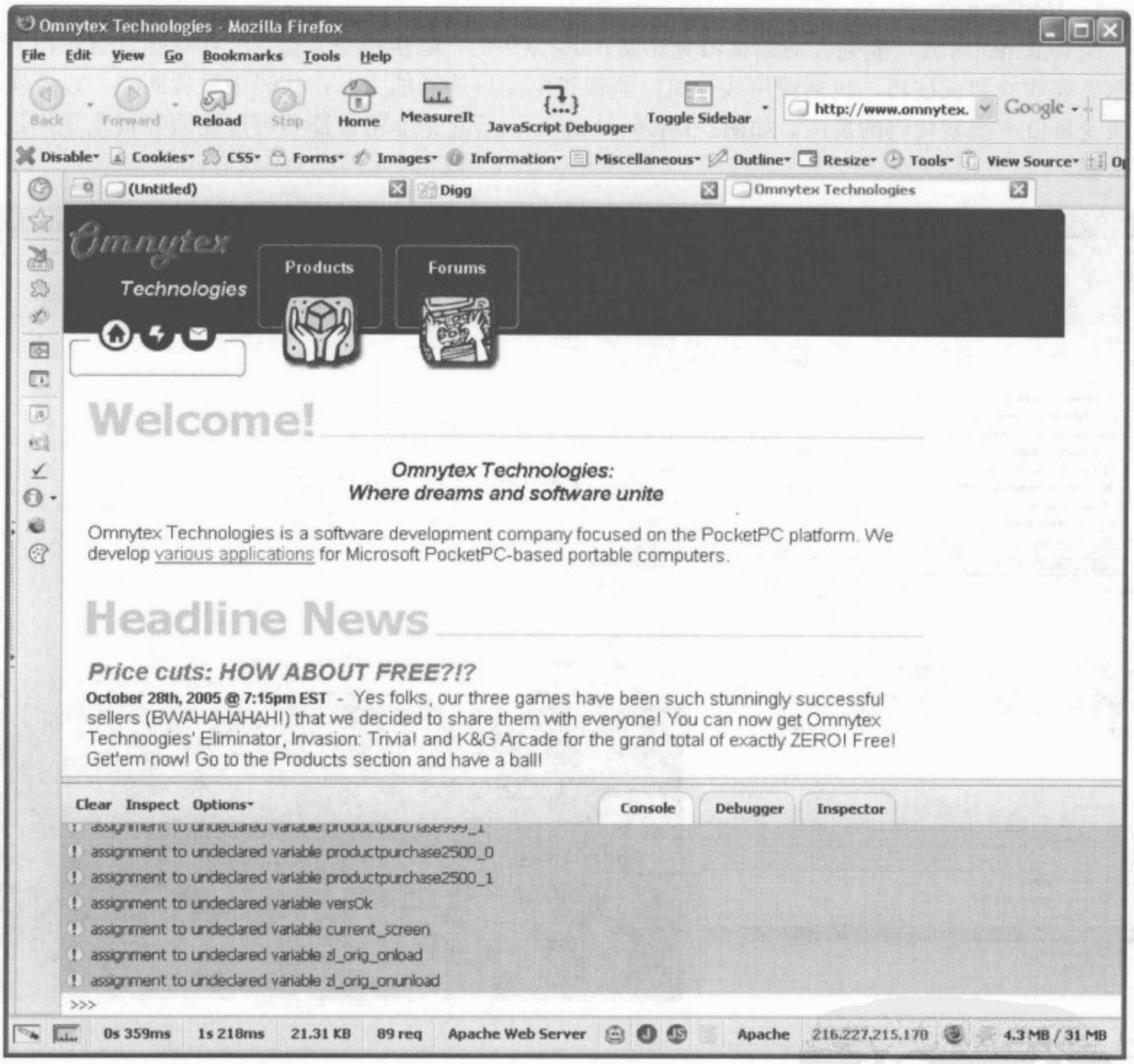


图2-4　Firebug：对Firefox来说可能是到目前为止最重要的扩展

Firebug注册Ajax请求，这是我所见过的其他扩展很少有的。你可以展开请求，并观察传递的参数、POST的主体部分、应答，等等。如果你正在做Ajax的开发工作，这绝对是无价之宝。

Firebug还提供一个调试器，它显示整个栈轨迹，也具有实时修改值和设置断点的作用。调试器如图2-5所示。

Firebug的Inspector工具是另一个非常好的功能。它允许你用鼠标盘旋于页面的项之上，查看它们的定义，包括它们的样式、从哪里继承等。你可以在任意点查看DOM树，想要挖多深就挖多深。

请记住，我只是简单地描述了一些Firebug表面的功能。总之，它带来了很多其他扩展中流行的功能，并把它们包成一个很小的、但功能强大的包。顺便说一下，你可以把Firebug放在浏览器的侧边，

而不是放在底端，我只是选择放在底端，像图2-4和图2-5中那样。

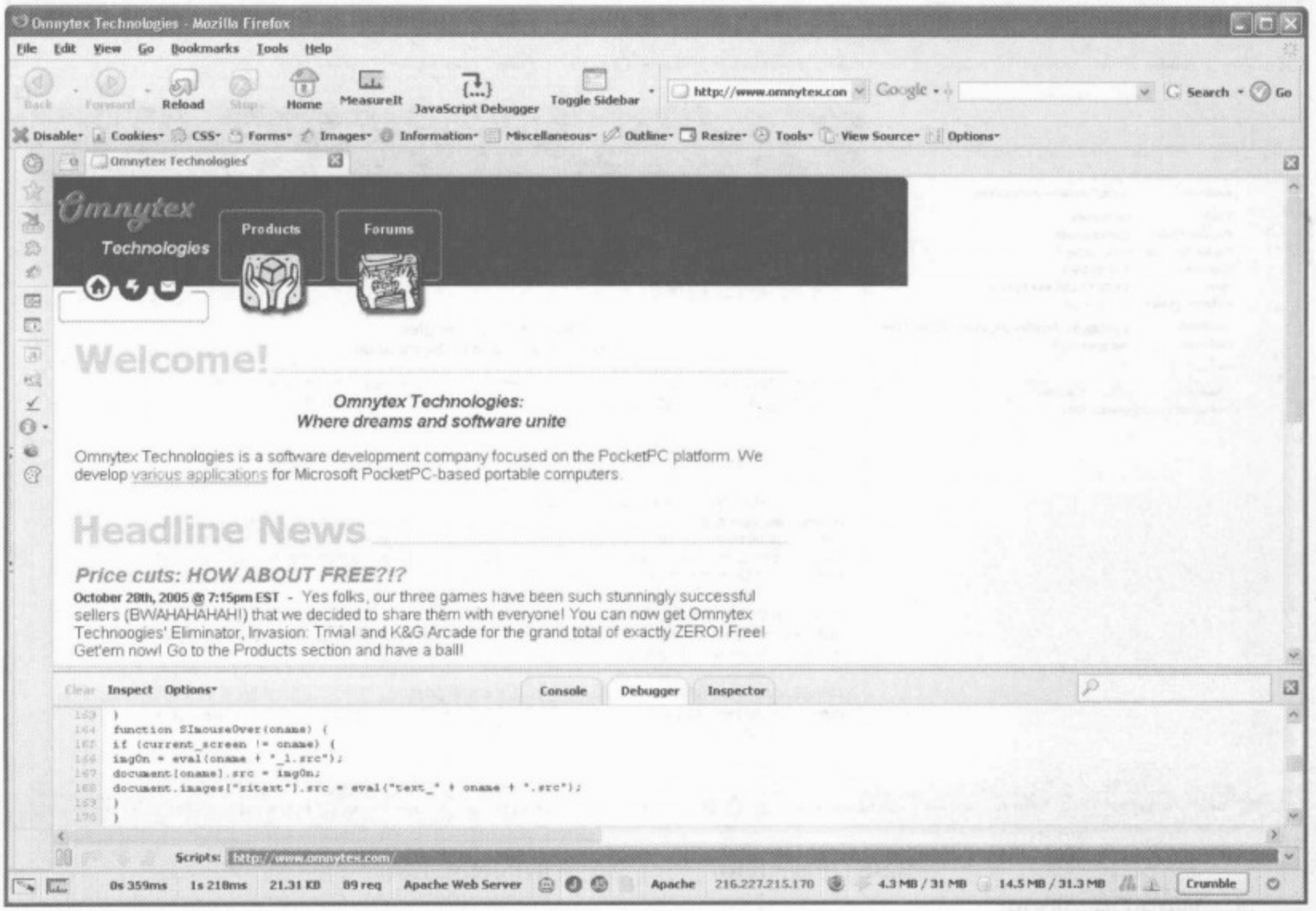


图2-5 Firebug的调试器，虽然简单至极，但同时也是非常强大的

如果你像我推荐的那样主要是在Firefox中进行开发，并且没有装其他的扩展，那么安装Firebug。

3. Page Info

Page Info是一个非常有用同时又非常简单的Firefox工具，如图2-6所示。Page Info事实上是绑定在Firefox中的，并且可以从浏览器的工具菜单中访问。

顾名思义，Page Info显示关于当前页面的信息，比如下面的这些内容。

- 页面使用的呈现模式。
- 页面所有表单的列表，以及它们的所有相关信息。
- 页面中所有链接的列表。
- 页面中所有图像和其他媒体资源的链接（还有图像的维数和各项的其他信息）。
- 页面使用的cookie列表（以及移除其中一个或全部一起移除的功能）。
- 页面中使用的脚本和样式表的列表（以及单独打开每一个的功能）。
- 所有页面的依赖关系的树状显示（图像、脚本、applet等）。
- 对于页面所有的请求和应答的HTTP头，以及页面与安全有关的信息。

如果这些对你来说是无价珍宝般的信息，那就对了！

图2-6　Page Info，另一个无价的Firefox工具

4．Web Developer

Web Developer工具条是另外一个你不想错过的扩展工具。这个插件，以工具条的形式提供了如此丰富的功能，以至于我连它很小一部分功能都介绍不完。所以，我将只是不分先后顺序地列出一些你可以做的事情。但是，请记住，我们怎么强调它的功能强大都不为过，下面只是它所提供的功能中非常少的一部分。

- 禁用各种元素，如JavaScript、meta重定向、页面颜色等。
- 查看cookie信息，还可以清空单个或一个域下的全部cookie等。
- 编辑页面上使用的CSS，完全禁用CSS，或只是查看导入的样式表。
- 修改表单的提交方法，使禁用的表单区域可写，将<select>元素改成文本区域等。
- 通过很多方法来操作图像，还可以获得关于它们的各种信息。
- 获得你可以想象的，关于页面的所有信息片段。
- 使锁定的框架可以重新设定大小。
- 突出显示任何你想要的页面元素类型，因此，你可以很清楚地看到所有你想要的表格、 元素、表单等。
- 重新设置窗口大小，也可以将它自动重设到任何最通用的窗口大小（比如，方便地查看页面在640×480的显示器中是什么样子的）。

- 快速获得大量在线验证器的入口，以及JavaScript和Java控制台。
- 查看生成的源代码，这绝对是一个非常有价值的帮助。

现在你可能已经意识到，这个工具条（Web Developer）和Page Info有大量的（功能）重叠。然而，Page Info 的信息组织更有层次感，并且更容易访问。不过，两个工具都能提供相同的功能。

使用这个工具条五分钟就足以使你信服它的价值了，所以我建议现在就花五分钟来看看它。去吧，我会等你。

已经回来了？好的，那我们转过头来看一些IE的扩展插件。

2.6.2 IE扩展

如果我们对比Firefox扩展和IE扩展的丰富程度的话，那么至少在开发工具的领域，你很快会发现Firefox更有优势。就是啊，只是Firebug就足以把所有IE的可用插件给比下去。当然，这不等于IE就没有一些比较出色的工具可用。

1. HttpWatch

HttpWatch（http://www.httpwatch.com），如图2-7所示，提供了捕获浏览器和远程服务器之前的请求和响应的功能。我这儿说的“捕获”，意思就是“捕获”！每个细节都会记录下来用于分析，而且还有个优点，它的信息组织得非常好，我们可以很容易地获取需要的信息。其中有一个最好的特性就是它可以显示发送或检索的源HTTP数据流。在调试很多古怪的问题的时候可以给我们很大帮助。

不过很不幸的是，HttpWatch不是免费的，而且对于个人来说，特别不便宜。也就是说，它是很有用的工具而且值这个价码。下载一个试用版看看吧！

2. Web Accessibility工具条

Web Accessibility工具条（http://www.visionaustralia.org.au/ais/toolbar）在检查站点的可访问性的时候绝对是一个极好的助手，而且它还有其他有用的功能。它类似Firefox的Web Developer，但是功能不是那么丰富（这是很正常的，因为这个工具条的目标功能区相对窄一些）。下面是一些它提供的功能。

- 窗口大小调节的功能。
- 迅速且容易地访问大量在线的验证器。
- 操作页面上的任何有关图像的信息。
- 颜色对比度的分析。
- 所有的页面结构分析（帮助你保证页面是所有屏幕阅读器合理可读的）。
- 模拟各种有障碍人士的功能，包括色盲/色弱和白内障。
- 显示大量的页面信息。

可访问性实现起来比较困难，有时候在现代的RIA中几乎是不可能的事情。这个工具条会在这件高难度的事情上给你不小的帮助，甚至在某些场合下，你不考虑可访问性的时候，这个工具条仍然是可以帮助你的好工具。它是免费的，因此我很难想出不安装它的理由。

3. IEDocMon

IEDocMon（http://www.cheztabor.com/IEDocMon/index.htm）是另一个让你觉得竟然会有人免费发布这样的东西的工具！这个IE扩展允许你查看页面的DOM树，展开到你需要的点，就像图2-8显示的

那样。它可以从树中突出显示页面上的元素，这样就可以确保看到正确的东西。它可以显示表示当前元素的HTML片段，这样你可以精确地找到自己要的东西。（在你查找一些聪明的开发人员是如何利用了特定的技巧的时候特别有用！）并且，它也可以一样处理脚本，这样你就可以把注意力集中在执行指定任务的那一部分脚本。

图2-7　在IE的世界里，HttpWatch是无法匹敌的

IEDocMon的最好的特性之一是它能够监视选定的元素的事件。比如，你有一个 ，并且当鼠标放在它上面的时候，它会改变颜色，但是貌似这个功能不能工作。使用IEDocMon，你可以从DOM树中选取这个 ，然后可以看到在那个元素上发生的每个事件。特别是在一个复杂的RIA里面，这个功能像金子一样有用，而且坦率地说，我还没发现任何其他浏览器的其他插件能实现这个功能。

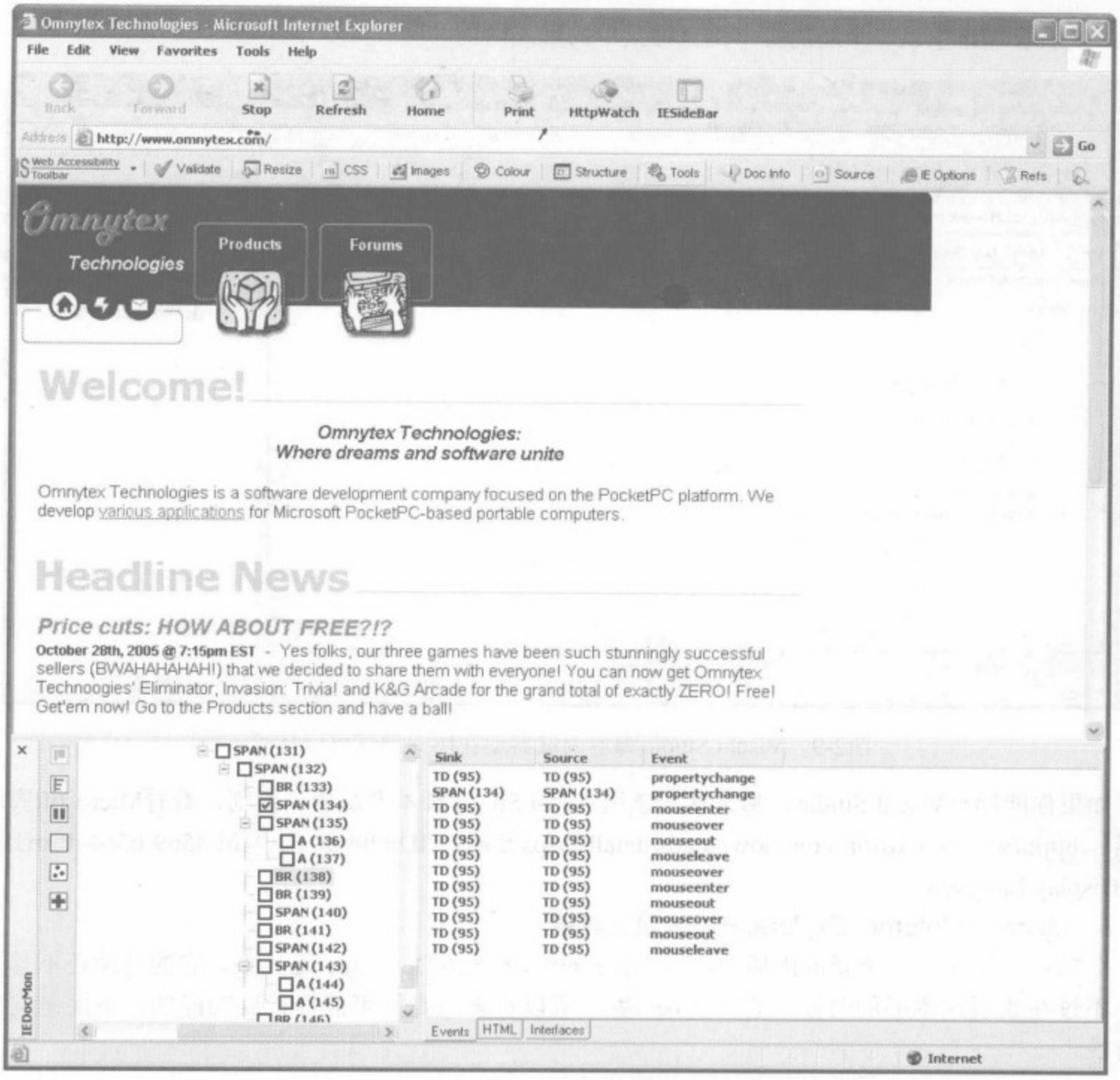


图2-8 IEDocMon：任何说免费没有好东西的人，一定没有看过这个

4. Visual Studio脚本调试器

微软的Visual Studio脚本调试器，是Visual Studio的一部分，是一个全功能的及时调试器，如图2-9所示。它可以截取IE页面上的错误，然后弹出窗口显示出错的行并且允许你实时操作代码。

作为Visual Studio的一部分，这个调试器不仅是商品软件，而且相当沉重。如果你很幸运已经注册了MSDN或者已经有了Visual Studio，那么这个调试器可以为你在IE里的工作提供巨大的帮助。

不幸的是，在IE里的调试器很少，因此你的选择从某种意义上来说是受限制的。如果你必须使用IE，那么这个调试器能让工作相当舒服，除非你想找那种轻量、活泼的调试器。

5. Microsoft脚本调试器

不要和Visual Studio脚本调试器混淆了，这是一个独立的Miscrosoft脚本调试器。这个调试器和Visual Studio脚本调试器不同，它不是全功能的。尽管它轻量很多，但它还是相当有用的。一旦安装

了它，它就会在IE中以一个新的脚本调试器菜单的形式出现。

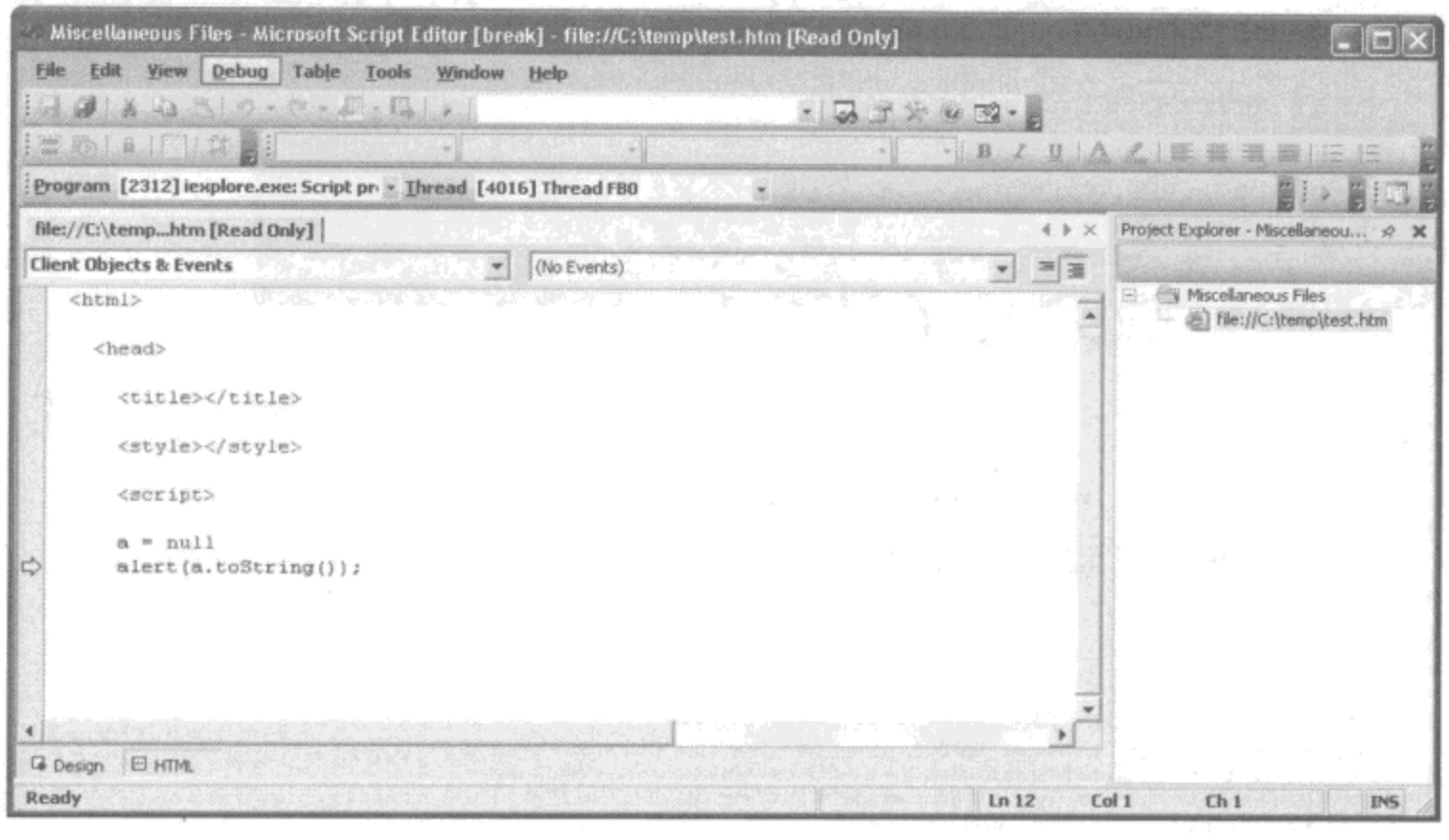


图2-9　Visual Studio脚本调试器使在IE中调试可以接受

如果你能拥有Visual Studio，那么你应该用Visual Studio脚本调试器。否则，看看Microsoft脚本调试器：http://www.microsoft.com/downloads/details.aspx?FamilyID=2f465be0-94fd-4569-b3c4-dffdf19ccd99&Display Lang=en。

6. Microsoft Internet Explorer开发人员工具条

微软最近发布了一个新的IE插件，很类似Firefox的Web Developer工具条。它的功能还不算太丰富，不过在我写这本书的时候，它还是beta版，所以将来还是很可能会扩展功能的。下面是它的几个特性。

- 列出表、图像、选中的标签和其他项的功能。
- 缩放浏览器窗口尺寸到指定尺寸的功能。
- 全功能的对齐尺（帮助对齐和测量页面的项）。
- 以树状视图的方式来浏览DOM的功能。

如果你想试试这个，那可以在这里下载（相当冗长且不方便）：http://www.microsoft.com/ downloads/details.aspx?familyid=e59c3964-672d-4511-bb3e-2d5e1db91038&displaylang=en。

2.6.3　Maxthon扩展：DevArt

Maxthon是我的常用浏览器。它是一个IE的封装，提供了很多IE缺乏的高级特性，但是并没有改变下层的IE的呈现引擎。这就意味着我不用担心有站点看不了。它还处理了许多IE的安全漏洞，因此它比“赤裸裸的”IE要安全多了。不过我们在这儿不谈浏览器的优劣和安全性，谈的是开发工具。

和Firefox类似，Maxthon（http://www.maxthon.com）有比IE更健壮的扩展架构。这些扩展中的一

个是DevArt（http://forum.maxthon.com/index.php? showtopic=14885），它提供了很多有效的开发特性，如图2-10所示。

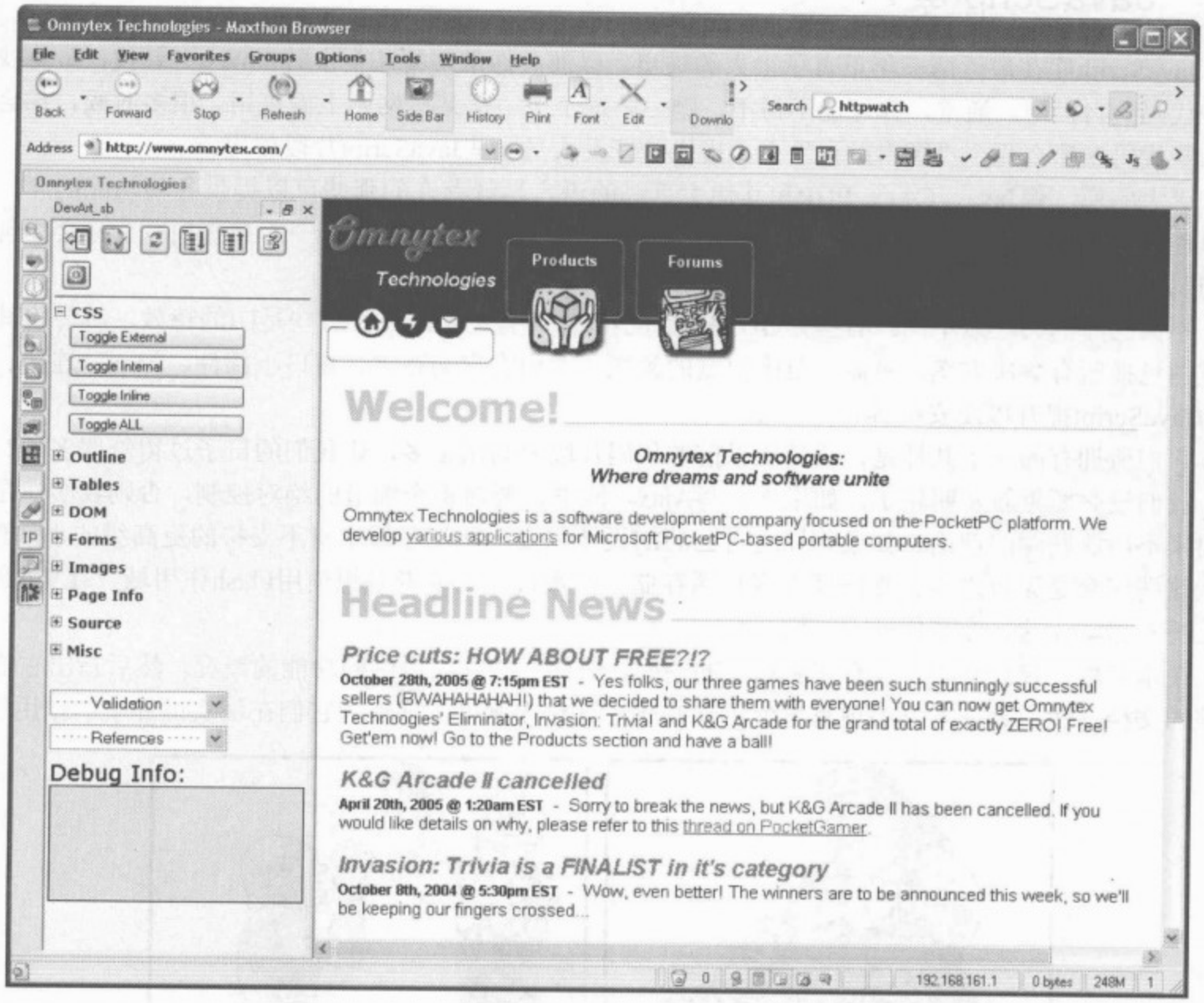


图2-10 DevArt，一个出色的Maxthon扩展

DevArt提供下面的功能。

- 开关样式表。
- 打开表、图片和 元素的轮廓。
- 删除表。
- 显示DOM树。
- 显示表单里隐藏的输入域和显示输入域的值。
- 显示页面中所有的图片的源以及它们的尺寸。
- 显示当前页的响应头。
- 查阅页面生成的源代码（也就是说，浏览器实际用来呈现页面的代码）。
- 用多种方式检验当前页。

如果你使用Maxthon，所有这些都让DevArt成为开发人员必须有的工具。DevArt有两种形式来使

用：工具条和边界条。

2.7　JavaScript库

JavaScript库在最近两三年里有了很大的进步。以前，你常常花几个小时在Web上寻找，最终找到一些代码给自己用。通常，你可能不得不，嗯……这个……从某些网站上偷来用。很多时候，你会发现这些代码在那么几个“脚本站点”上，提供给开发人员一些JavaScript片段使用。

更大的库，像Java、C++、PHP和其他大哥级的语言早就存在的那些可以提供各种功能的库，只是最近几年才在JavaScript世界里出现。从很多方面来说，我们现在都处于一个黄金时代，你会发现可选的比想要的还多。

不同的库的关注点不同：有些是GUI的窗口小部件，有些是Ajax，有些是UI的特效，等等。其他库试图包揽所有解决方案，覆盖了范围很宽的领域，比如客户端存储、窗口小部件、Ajax、连接、基础的JavaScript提升以及安全等。

它们所拥有的一个共性是，质量比以前的代码片段要高出很多，让我们的日子过得舒服多了！通常，我们没必要重新发明轮子。如果你在写Ajax，除非需要对每个细节的绝对控制，否则我实在是想不出来不用这些库的理由。如果你知道自己的UI设计需要一些浏览器本身不支持的更高级的小部件，那么这些库会是无价之宝。你需要在客户端存储一些数据吗？或者是想使用Flash作用域（在第6章介绍）吗？那就让这些库帮你处理它们吧！

在本节里，我会介绍一些有代表意义的的库，给你一些它们提供的功能的概况，然后指出它们更完整的文档。我们会在本书的项目里使用这些库，这样你就可以感受到它们在现实世界中的作用了。

没理由重新发明轮子，所以使用库，要不然你会像Tor那样

现如今已经有非常多的库，而我们只能介绍其中的少数几种，而且即便如此，也不能说得太详细。所以，请不要把我们这里的内容看作是这些库的权威性的参考。我在这里写一些简单的介绍，吊起你的胃口。我相信一旦你阅读了本节，并且看了这些库在项目里面的用途，你就会想把它们下载下来然后学习更多细节了。除了自己认真琢磨之外，我相信你没法知道它们都能提供哪些好处。

2.7.1 Prototype

有些库的注意力集中在Ajax上，其他比较关注GUI窗口小部件，还有一些提供各种各样特效，你可以很容易添加到自己的页面中去。Prototype（原型）是一个存在于低于所有这些功能的层次里的东西，证据就是许多库实际上都是在原型（Prototype）的基础上构建的。

Prototype（http://prototype.conio.net）从某种意义上可以看作是一个对JavaScript本身的扩展。在更技术的层面上，它的实现方式是通过扩展一些JavaScript内置的对象来获得相对直接的扩展。

和其他大多数这类库一样，熟悉Prototype的唯一办法就是研究和使用它。这也是本书的项目的另一个要点（这些项目从下一节开始介绍）。不过，我想重点描述几个Prototype提供的比较著名的东西。

- Prototype包括一堆缩写工具函数，包括$()，$()是一个编写document.getElementById()的缩写。类似的还有，$F()返回任何表单域的值。还有其他一些，但这两个是我觉得最有用的。
- Prototype通过Ajax对象提供相对基本的Ajax支持。它的一个最有用的特性是Ajax.Updater，提供了一个简单、快速的办法给一个现有的页面元素填充来自服务器响应的数据（假设是HTML）。这应该是Ajax世界最常见的工作了。
- PeriodicalExecuter对象提供了一个简单的方法用来在固定的周期里反复执行一段代码。这样就免除了我们设置超时等工作的环节。
- Prototype扩展了内置的Array对象，提供了一些其他的附加价值，比如Enumerable（可枚举的）界面，这样你可以非常利落地遍历一个数组，而不用使用数组的长度设置一个for循环来实现。

有些人反感Prototype，因为它扩展了内置的对象，这样可能会导致一些细微的问题。有些人因此从不使用Prototype或者任何使用它的库。我的看法是，从我自己的经验以及大量有关这方面的读物了解到的是，虽然这些问题是不容忽视的，但是还不足以说服我不用Prototype。当然，你还是应该注意潜在的问题，以防它们的出现。

2.7.2 Dojo

Dojo（http://dojotoolkit.org）可能是最近成长最为迅速的、最受欢迎的库了。它有广泛的范围，你可以在这个项目里寻找自己需要的任何东西，比如那些做现代的客户端开发的东西。

在我前一本有关Ajax的书里（*Practical Ajax Projects with Java Technology*），曾说过Dojo有个问题：缺乏文档和好的例子。我说使用Dojo就像和自己修栅栏那样，经常需要查阅代码以弄明白如何做事。虽然还不能说现在完全好了，但是我的确可以退一步了。在这些方面，Dojo已经改进很多了。现在我们可以看到很不错的文档，并且可以找到更多例子了。这个现象很自然，所有那些知道自己在干什么的人使用的库，逻辑上都会表现出这种现象，Dojo也一样。

同时，IBM、Sun和一些其他厂商也承诺支持Dojo，并且它们谈论的帮助的首要领域之一就是文档。因此我们有理由相信这样的进步会继续下去，甚至会更快。但是在那之前，如果你想用这个库，那么我仍然强烈建议你注册Dojo的邮件列表。大量乐于助人的人在那里尽力回答问题。同时，还是要有足够的耐心，因为通常获得有用的回答需要一两天时间。不过通常会有人回答，而那回答是很有价值的！

所以，Dojo给我们提供了什么？成吨的东西！就像我前面说的，Dojo有非常宽的应用范畴，下面

我列出来的几个是我觉得让人无比兴奋的东西。

- 窗口小部件。平心而论，Dojo最有名的就是它的众多窗口小部件。有些绝对比其他的好，不过这是预料中的。它们都是从相同的基本的窗口小部件框架里扩展出来的，因此它们输出类似的基础功能的集合。这些有利条件令我们在大多数用它工作的时候都非常容易。一些更值得一提的窗口小部件如下。
- Fisheye，模拟Apple的启动条，当鼠标放在图标上面的时候它会放大图标。
- 树视图窗口小部件，给你展开/缩起树界面的功能，类似Windows的资源管理器文件夹列表。
- 一个非常棒的幻灯放映窗口小部件。
- 一个在页面上插入Google地图的窗口小部件。
- 模拟大多数基本的HTML表单元素，以及扩展的表单元素，比如用于打开一个下拉列表的带箭头的按钮、一个富文本编辑器、一个日期选择器、一个颜色选择器和一个组合框等。
- Dojo对Ajax的支持是非常好的，我不敢说它是最好的，但是你用它是绝对没错的。
- Dojo提供了大多数常用的效果，比如擦除和淡出，并且很容易实现。
- Dojo提供拖放的支持，非常容易使用。
- Dojo提供的存储支持据我所知是独一无二的。它提供了使用客户端可持续的数据存储支持，比如Flash存储（这实际上是黄油上的小甜饼，是Adobe Flash插件提供的支持）。
- Dojo包含一堆集合实现和其他常用数据结构的实现，可以让JavaScript代码更健壮并且更像在服务器端写的东西。
- 很多“核心”的库都是Dojo的一部分。它们提供一些东西，比如高级的字符串操作，简化的DOM操作，令JavaScript本身更容易和更强大的函数以及用于操作HTML的函数。
- Dojo包含一些基本的加密功能和一些更强大的数学相关的功能。

尽管在我写关于Ajax的书籍的时候，Dojo取得了长足的进步，但我仍然觉得应该树一个小小的警告牌：Dojo会时不时地给你些意外。我目前正在一个非常复杂的项目里使用它，在它让我们的工作更容易的同时，我仍然必须和它战斗以修复一些问题。现在来说，这些问题中有一些（甚至可能是大多数）是因为我对它还不够熟悉。我在学习所有Dojo提供的东西，以及它们是如何运作的，和其他大多数人一样！不过，我的确是发现了这里那里的一些bug，还有一些可以做得更好的地方。这些情况就意味着你在使用Dojo的时候首先要有准备。而不要只是期望它会即插即用，即使你发现它确实越来越能即插即用了。你会碰到问题，而且有些问题是现有的任何文档都不能回答的。有时候Google或者Yahoo能帮助我们，但是你会经常需要问一些问题，这个时候邮件列表就有用了。

最后，我认为，Dojo提供了如此之多的特性，以至于问它是否值得用的人一定是没脑子的了。你遇到的或是克服的任何问题，都不足以抵消它的强大以及它给你节约的劳动。Dojo有非常光明的未来，即使在我使用它的第一年，都能很明显地感觉到它的进步。尝试一下吧……它有着很高的曝光率，并且都是正面报道！

2.7.3　Java Web Parts

Java Web Parts（http://javawebparts.sourceforge.net）是一个面向Java开发人员的项目，但是也提供了一些JavaScript功能。它提供这方面功能的方式比较独特：它是一个叫做JSTags的部件的一部分，JSTags是一个标签库，生成JavaScript。我们可以在这里找到很多有用的函数，包括下面的这些。

- JSDigester，一个流行的Jakarta Commons Digester组件的客户端实现。
- 一个把表单转换成XML的函数。
- 操作cookie的函数。
- 验证字符串输入的函数。
- 一个可以禁用右键点击功能的函数。

如果你不是Java开发人员，那你可以把标签生成的JavaScript偷出来然后独立使用它。如果你是Java开发人员，那么这个标签库能让你工作更容易。

还要注意的是Java Web Parts里的另一个标签库：UI窗口小部件，它提供了一些很好的窗口小部件，包括弹出的日程表和交换器。

最后，APT（AjaxParts Taglib）是我认为目前可获得的最好的Ajax库。除非你是Java开发人员，否则不会使用这个东西，但是如果你是Java开发人员，那就准备轻松享用Ajax吧！APT允许你通过在页面里添加标签以及配置一些XML就完成Ajax的添加。页面上的每个Ajax函数都是在一个XML的配置文件里定义的——你无需写任何一行JavaScript代码。所有常用的Ajax函数都是内置的，可能可以满足95%的需要。对于另外5%，APT可以以简单合理的方式来扩展。所以，如果你需要做更高级的事情，那可以做，仅需要写一些满足特定场合需要的JavaScript代码，而且还是不用写底层的Ajax代码。所有这些都令APT成为Java Web开发的绝佳选择。

2.7.4 script.aculo.us

script.aculo.us（http://script.aculo.us）是基于Prototye的库之一。script.aculo.us提供了好多方面的功能，但坦率说，它最出色的地方是：特效。如果你想使用淡出、擦除、动画和类似的效果，script.aculo.us是你应该考虑的第一个库。下面是它提供的一些项。

- 5个核心效果：透明度、延展、移动、突出显示和平行。平行是一个允许你组合特效的特效，它导致下一项的产生。
- 组合特效，可以看作是更高级的特效，通过组合几个核心特效来创建。这个的例子是抖动、搏动、阴影和摩擦。
- 几个控制，比如自动补齐输入框和现成编辑内容。
- Builder对象，让我们可以更容易地用JavaScript构建DOM片段。

script.aculo.us还提供单元测试和功能测试的能力，这个是在所有库里面比较独特的。如果你曾经用过JUnit，那么这方面的支持看上去很类似！

我强烈建议浏览一下script.aculo.us的网页，并且花一些时间看看那里的各种演示，尤其是前面提到的特效。观察那些特效是欣赏这个库已经提供的功能的最好办法。不过也别忽略其他东西。你会发现很多好特性和特效没什么关系！不过如果你就是要找特效，那么script.aculo.us就是你的救世主。（哦……或者是类似那样的角色！）

2.7.5 YUI库

YUI库（Yahoo! User Interface Library，Yahoo! 用户界面库）（http://developer.yahoo.com/yui），或者常说的YUI库，自介绍给公众之后已经获得了大量的注意。它提供了一系列的UI窗口小部件和常用的JavaScript函数，所有的东西都在一个很干净并且有着很好的文档的包中。尽管YUI库不像这里的其

他几个库那么吸引眼球，但它提供的窗口小部件简单、易用，并且相当轻量，整个库也如此。下面是它提供的一些项。

- 简单的跨浏览器日志。
- 一个Event（事件）组件，允许你给一些元素附加事件，当DOM元素被检测到的时候执行代码，并且从代码中把浏览器事件模型完整地抽象出来，以及其他的一些事件相关的事情。
- ConnectionManager对象，以一种干净简单的方式提供Ajax功能。
- 一些操作DOM的工具函数，比如一些让你可以很容易获取一个视区的宽度和高度的功能，并且还是跨浏览器的。（这种事情有时候非常麻烦！）
- 基本的动画支持，包括沿着一个曲线移动和滚动。
- 一个相当丰富的拖放功能（实际上是我用过的最健壮的拖放实现，提供了大量可以钩进去的事件）。
- UI窗口小部件，包括一个日历、一个下拉菜单、一个滑出器、一个树视图和大量组织UI的容器。

在第一次看到YUI库的时候会让人觉得有一些奇怪，可能会让你忽视其真正的能力。花点时间，看看文档（相当棒！）和例子。过几分钟后，我相信你就会觉得它有多好、多有用了。

2.7.6　MochiKit

“MochiKit令JavaScript没那么烂。”这个是该站点的标签行，我能跟谁争论去？实际上，MochiKit提供的一些特性（http://www.mochikit.com）的确是做到了标签行说的那样。

你试过在自己的表格上用圆角吗？你是否真正尝试过并且知道实现这个圆角需要多少复杂的技巧和技术？非常非常多，而且要想同时实现跨浏览器的支持是极为烦琐的事情。MochiKit可以帮你做这个事情。看上去好像提供的功能不多，但是用好了的话，圆角确实可以让页面看上去更舒服。

MochiKit还有一个好用的特性，就是用一个很好的格式显示当前页的源代码。尽管我知道这个功能对终端用户没什么用，但是它的确对开发人员很有用！

MochiKit还提供了一个客户端的可排序的窗口小部件和一个跨浏览器的键盘事件处理机制。在MochiKit站点的演示页面上，你还能看到一些很好的例子，比如一个现场的正则表达式计算器和一个最小的JavaScript解释器。所有这些都表明MochiKit有些很有趣的功能。

2.7.7　Rico

下一个要介绍的库是Rico（http://openrico.org），其名称已经非常明显了“用于RIA（富因特网应用）的JavaScript”。Rico在4个主要方面提供了功能：Ajax、拖放、剧院效果和行为。它的关注点相当少，因此你可以预期它在这些领域做得特别好，事实也是如此。下面是它提供的功能的一个概述。

- 在Ajax部分，Rico提供了一个自动从Ajax请求填充表单元素的特性，非常好用。它还提供了从一个Ajax请求的对innerHTML的原型化的修改，使用还很简单。
- 在拖放部分，基础功能实现得很好。它还提供一些高级特性，比如定制拖放区域和自定义元素的可拖放功能。
- 在剧院部分里，Rico提供了很容易移动和改变元素尺寸的功能，以及很容易圆化一个段的角的功能。

- 在行为部分，你可以看到像手风琴这样的东西，它可以让你把一堆 元素转换成一个漂亮的手风琴。同时还有实时的网格，它可以把一个普通的HTML表和一个Ajax数据源连接起来，让我们可以实时地加载和排序数据，以及做一些其他处理。

我自己是被手风琴行为和圆角功能震呆了。这两个功能的实现都是顶级的，并且都是我经常想在自己的项目中使用的。我不是说Rico其他的东西就不好，但是这两个功能的确是掩盖了其他功能的光芒。

2.7.8 Mootools

随着这些库的列举，我再说最后一个（但绝对不是仅有的最后一个），Mootools（http://mootools.net/）。Mootools是我在写本书的时候最后发现的一个库，但是我认为绝对应该把它包括进来。Mootools是一个非常轻量化、模块化的库，分成好几个部分，你可以根据自己的喜好选取是否包含它。它是完全面向对象的库，设计成开发人员可扩展的方式。它的各种模块覆盖了非常广泛的需求，包括下面这些。

- 一个JavaScript的职责链（Chain of Responsibility，CoR）模式的实现[①]。
- 一堆特效、变化和特效相关的函数。
- Ajax功能（这一点和其他大多数库一样，只是实现得更精致、更简单、更干净、用起来更爽）。
- 处理函数的功能，创建和使用JSON，与浏览器窗口交互，以及一些很有用的字符串工具。

Mootools的一个特别酷的地方实际上是在库之外，那就是它的下载页。和常见的“下载这个或者那个”的页面不一样的是，它提供了一堆可选的模块，你可以选择自己需要的。然后下载会根据你的选择在运行时生成下载项，甚至还包括压缩。使用这个功能，你可以创建自定义的Mootools库。

2.8 小结

哇！本章的主题实在是太多了！我们从面向对象的JavaScript编程技巧开始，稍微超越了一点点第1章的内容。然后我们看了一些相对新的术语，不唐突的JavaScript并且讨论了它的含义和它应该在何时处理以及为什么需要不唐突。与之绑定的是柔性衰减的概念和这些概念为什么在如今仍然使用。然后讨论了可访问性和一些让我们的站点在残障人士那里仍然保持可访问性的方法。然后看了看JavaScript里面的更强壮的错误处理。之后，我们又开始了一个小漫游，看了看调试的技巧和现如今可用的工具。最后，我们了解了一些节约时间的浏览器插件和一些流行的JavaScript库。

用这些知识武装完毕之后，下一章将开始构成本书核心的进攻序幕。让我们好好琢磨一下吧，如何？

① 你可以争论说它从技术上实际并不是CoR实现，并且要澄清的是，这是我的描述，而不是Mootools开发小组的描述。不过这基本上是它做的事情，并且在我所知的范畴里是独一无二的。有关CoR模式更多一般性的信息，参阅http://en.wikipedia.org/wiki/Chain-of-responsibility_pattern。

Part 2 第二部分

项　目

重要的是不要停止质疑。

——阿尔伯特 • 爱因斯坦

我认识的最愚蠢的人是那些什么都知道的人。

——马尔科姆 • 福布斯（Malcolm Forbes）

人类几乎是拥有从其他人的经验中学习的能力的唯一物种，但也很明显地是不情愿这么做的物种。

——道格拉斯 • 亚当斯（Douglas Adams）

计算机可以比人类历史上的任何发明都能让你更快地制造更多错误，也许仅次于手枪和龙舌兰酒。

——米奇 • 拉特利夫（Mitch Ratliffe）

创造力就是蠢事突然停止。

——埃德温 • 兰德（Edwin Land）

决不要相信那些打不开窗口的计算机。

——史蒂夫 • 沃兹尼克（Steve Wozniak），苹果电脑的设计者，计算机先驱

第3章

Hodgepodge：构建可扩展的JavaScript库

不，我们并不是在这里讨论那只兔子[①]。凡是写了一段时间的代码的程序员，都会构建自己的小私有库，来保存一些方便的、在项目间可重用的代码。在这点上，JavaScript当然也不例外。本章中，你将为自己集成这么一个库——在后面的项目中你会发现它有大用处。

本章将在一个假想的初级开发者和他的行为有点古怪的资深导师间，以一问一答的形式进行阐述。我们不仅会将一些有用函数集成在一起，还会讲述如何在一个伪包结构中组织它们，以避免命名空间冲突，并使你更容易找到想要的东西，同时也使它变得更像“真正的”语言一点，比如Java。哦，对了，本章的小讽刺幽默也挺有趣的！现在让我们开始重现反斗智多星[②]的场景吧！

3.1 Bill，菜鸟的一天

“天啊，上午11:30了，他们希望我马上开始写代码吗？我讨厌这个地方。”

Gilbert已经在Initech[③]工作了整整五年（他对做一个开发者异常衷心！），但还是没法习惯“早起”。大概上午10:45左右，Gilbert从床上爬了起来，冲个澡（通常地），然后跳进车里，开车15分钟去上班。还好，他是一个出色的开发人员，而且每个人都知道，所以可以略微偷点懒。

但是今天，他面对的是他从未面临过的挑战——一个连自己都不知道是否能处理的深度的恐惧，何况是在这么乱糟糟的早上。

今天是Bill工作的第三天。Bill是一个纯粹的新手[④]。

---

① Hodge-Podge是Berke Breathed的喜剧卡通*Bloom County*里的兔子的名字。

② 反斗智多星（*Wayne's World*）是一部1992年的电影，由Mike Myers、Dana Carvey和Tia Carrere主演，故事情节大概是两个懒鬼朋友Myers和Carvey，试图推销他们的一部叫“*Wayne 's World*”的有线电视节目。可以想象，后面很快就是在简单情节上夸张和疯狂地搞笑。至少有一个镜头是由一个角色进行倒叙的，配合着晃动的屏幕，场景慢慢地消失回到了之前的时候（而Myers和Carvey在滑稽的挥手和各种音效中出场）。还没想起来？那就去YouTube（http://www.youtube.com）搜一下吧，肯定能很快找到一个片段！（反斗智多星和王牌大间碟是同一个导演的作品，其中充满搞笑镜头。——译者注）

③ Initech 是电影*Office Space*中的角色所服务的虚构的公司的名字。

④ n00b是Leetspeak（1337）里的newbie（新手、菜鸟）的意思。菜鸟是指刚开始做某件事情的人。Leetspeak是一种拼写方式，更准确地说是一种书写字母的方式，通常和“地下”计算机组织有关（软件盗版、黑客、骇客、色情游戏等）。

因此，Gilbert很不热心，他自己都感觉得到。

“嗯，Gilbert，先生?” Bill怯怯地问。

“你今天给我带来了什么礼物？”Gilbert回答。

“两听可乐、一些胶皮糖和一罐DD咖啡。”

Gilbert对他的年轻学徒表示满意。“你会进步并成为熟手的。”

“嗯……Jack，那个高级构架师……”

“不要在我跟前提那个讨厌鬼的名字!”Gilbert喊道。

Bill被吓了一跳，虽然Bill已经有经验了，他知道Gilbert看不起那些用他的话来说“整天坐在那里只是写写‘论文’，而从不实践的人”。不过Bill的确是有备而来。

“我知道那个讨厌鬼不是你这个级别，老师，但是他是我的上级，因此我必须默许他的愿望。”

Gilbert考虑了一会他年轻学徒的话。“那倒是，他是你的上级，很少有人不是他的下属。你可以继续下去。”

“好的，”Bill继续说，“他让我往在线账务系统中添加一些函数，我有一些问题，希望你能帮我。因为只有你可以了。”Bill知道，最后那一句话能让他赢得一些分数。果然，Gilbert看起来很高兴。

“你可以提你的问题，菜鸟，这样你可以得到我的知识。”

然后Bill就开始说了。

暂停，回到现实中一分钟。是的，这是我，作者的声音！我只是想确认你还在这里。我们现在要做的事情是，创建一个JavaScript函数库，还要学习如何把它放入一些类似于Java或C#包的伪包结构中。有些函数（但可能不是全部）会在后面的项目中用到，而且会给你自己的那个小库一个很好的开端。你绝对应该添加自己的代码来创建这个库，并且你会发现，在以后的工作中，它会成为一个非常方便的工具!

3.2 全面的代码组织

在本节中，我们将看看，如何以一种干净的、合理的方式来组织JavaScript代码，同时也尽量避免命名冲突，让代码更加安全一点。

问题：怎样组织JavaScript代码来避免命名冲突，并将相关函数干净地分组在一起？

我想保证自己的代码组织得很好，并且以这种方法添加的任何代码，都没有与系统中已存在的代码名称发生冲突的风险。

回答：在JavaScript中，你可以在某种程度上效仿在Java、C#或其他现代语言中使用的包系统。所有你需要做的就是创建一个新类，然后将所有的工具函数作为那个类的成员。

例如，假设你想要创建一个包，它包含了一串函数，用来显示各种不同的预定义的警告信息。你可以通过下面的代码来实现：

```
jscript = function() { }
jscript.ui = function() { }
jscript.ui.alerts = new function() { }
jscript.ui.alerts.showErrorAlert = function() { }
```

执行这段代码的时候会发生什么？简单地说，你将有一个名叫jscript的对象，它是一个函数的引用。记住，在JavaScript中，函数就是对象，并且你可以使一个变量引用那个对象（类似于在C中用一

个指针指向一个函数）。在这个对象（jscript）中，是一个叫做ui的成员，它本身就是一个函数。然后那个对象（ui）内部的是另外一个叫做alerts的对象，它也是一个函数。最后，在那个对象（alerts）内部的是另一个对象，叫做showErrorAlert，它又是一个函数。

我们其实正在创建一个对象的层级关系，嵌套在父对象jscript内部。这是包的根，后面每一行都向那个对象添加了一个成员，代表一个新的子包。

然后，我们可以引用jscript的任何成员（子包）或沿着子对象的层级关系来引用任何对象（换句话说，函数或字段被定义在某个给定的子包里）。如果你用过Java或C#，你会发现，这些动作实际上给了我们包的外观。我用“外观”这个词是因为，事实上，你有一系列可以单独地实例化的对象。例如，你可以这样做：

```
var v = new jscript.ui.alerts();
```

显然，你在Java包里不能这么做，但是在JavaScript里可以这么做。而且我们还不能禁止这么做，因为事实上每个函数都是一个类的构造函数，而不是创建实例的机制。换句话说，下面的代码很诱人尝试：

```
jscript = new function() {
  return null;
}
var v = new jscript();
```

看起来有理由怀疑v的值可能为null。但是，事实上，那并不是它的值。因为null是jscript所指向的那个函数返回的，而不是那个函数被执行的结果。

在这种情况下，实例化的函数可以执行（记住，它本质上是一个构造函数），但像在Java中一样，并没有返回类型，也就是为什么返回null并没有做我们期望的事情。不过，这允许我们做下面的事情：

```
jscript = new function() {
  alert("Do not instantiate me!");
}
```

很明显，这比上面的那个让实例化却什么都不做的好不了多少，但是总比什么都没有好。

现在我们继续，如果你想有一个方法能在发生错误的时候显示一个警告，并且希望它成为这个包的一部分，该怎样呢？这很简单，你这么写就可以：

```
jscript = function() { }
jscript.ui = function() { }
jscript.ui.alerts = new function() { }
jscript.ui.alerts.showErrorAlert = function() {
  alert("An error occurred");
}
```

现在，你可以调用这个：

```
jscript.ui.alerts.showErrorAlert();
```

这会弹出一个你希望的警告“An error occurred”。

这时Bill打断Gilbert的解释，并问道，“但是如果我想在那个警告包中有一个类该怎样呢？就像在Java中可以做的那样，比如展示一个特定的信息？”Gilbert微笑着对他的学生的专注表示满意，并回答，“不是个问题……”。

```
jscript.ui.alerts.MessageDisplayer = function(inMsg) {
  this.msg = inMsg;
  this.toString = function() {
    return "msg=" + this.msg;
  }
}
var v = new jscript.ui.alerts.MessageDisplayer("Hello!");
alert(v);
```

这样实际上在jscript.ui.alerts包中创建了一个名叫MessageDisplayer的类。当执行到最后两行的时候，会创建一个新的MessageDisplayer的实例，字符串“Hello!”被传给构造函数。然后当调用alert()，并传递指向那个实例的变量时，就会调用toString()函数，我们就得到了预期的弹出信息说“msg=Hello!”。

“很帅!”Bill喊道，对于他的新知识非常兴奋。“让我看看能不能把它们放在一起。”Bill在键盘旁敲了一会，最终完成了代码清单3-1中所示的代码。

代码清单3-1 在JavaScript里的伪包的一个完整的例子

```
<html>
  <head>
    <title>JavaScript Packaging Example</title>
    <script>
      jscript = function() { }
      jscript.ui = function() { }
      jscript.ui.alerts = new function() { }
      jscript.ui.alerts.showErrorAlert = function() {
        alert("An error occurred");
      }
      jscript.ui.alerts.MessageDisplayer = function(inMsg) {
        this.msg = inMsg;
        this.toString = function() {
          return "msg=" + this.msg;
        }
      }
      function test() {
        jscript.ui.alerts.showErrorAlert()
        var v = new jscript.ui.alerts.MessageDisplayer ("Hello!!");
        alert(v);
      }
    </script>
  </head>
  <body>
    <input type="button" value="Test Alert"
      onclick="test();">
  </body>
</html>
```

Gilbert检查了Bill的代码，尝试运行，并看到预期的运行结果，他很高兴。

“只有一件事情可以使它更好，”Gilbert说，“你知道那有可能是什么吗？”Bill想了一会，然后突

然意识到Gilbert指的是什么。“对，一些和import（导入）相关的事情！”Bill大声说。“没错！”Gilbert回答，“你知道吗？并没有那么难。”

假设我们想有个叫jscript.string的包，然后希望能够把这个包独立于任何其他可能存在于jscript中的包输入到代码中。那么可以创建这样的一个文件：

```
if (typeof jscript == 'undefined') {
  jscript = function() { }
}

jscript.string = function() { }

jscript.string.sampleFunction = function(inMsg) {
  alert(inMsg);
}
```

现在，为了把这些“导入”一个页面，我们简单地这么做：

```
<script src="jscript.string.js"></script>
```

如果这是唯一的导入，那么我们就得到一个jscript对象，它包含一个字符串函数，这个字符串函数最后包含sampleFunction函数，所有这些都是在一个层级关系中，最后组成了伪包。更好一点的做法是，如果在jscript下面有另外的包，并且我们要导入它们，那么这儿用的if检查语句可以确保每一个包对象总是只有一个副本。一个最终的好处：如果后面想要扩展包，比如，添加一个jscript.string.format包，我们需要做的就是添加一个新的jscript.string.format.js文件，使用同样的检查方法，再添加一个来检查jscript.string是否已被定义，如果没有定义就实例化。

如果我们确实想要更疯狂些，可以将每个单独的包中的每个函数或每个对象放在它自己的.js文件中。这么做就像在Java或C#中导入你想要的指定类（这里，我们其实是把独立函数等同于类）。目前来看，你基本上可以只使用通配符导入的方法就足够了。但是我希望指出的是，如果你愿意，也可以拥有特定类导入的功能。

不过要记住，在Java中，如果导入不使用的东西并不会影响最终类的大小，与之不同的是，在JavaScript中，导入的东西都会影响用户下载的页面的大小，不论是否使用了它。所以，只导入那些你确实需要的东西很重要。还要记住的是，那并不仅仅是大小的问题。每个被导入的.js文件都需要另一个从服务器取回源代码的过程。（这里忽略了浏览器缓存，缓存可以在某些情况下减少请求数，但是第一次请求的时候肯定不能忽略！）

“你还有其他问题吗，我年轻的学徒？”Bill以为Gilbert要拔出一把光剑来。他今天表现得比往常更加怪异。不过Bill确实还有一些问题，所以他继续。

3.3　创建包

既然我们知道了如何组织包，那么就准备开始创建JavaScript库吧。库是一个包含各种不同有用的函数集合。你会在整本书中使用其中的一些，那，开始吧！

3.3.1　构建jscript.array包

在本节中，会写一些代码来帮助我们使用数组，并开始创建第一个包jscript.array。

问题：如何将一个数组的内容复制到另一个数组？

嗯，这里有另外一个Jack提出的情况：当用户第一次访问应用程序时，它建立了一个分类的数组，但用户可以以后向数组中添加内容。我希望创建用户输入的实体，生成数组，并将它们追加到已存在的数组上。

回答：你在开玩笑吗？就这些？看一眼代码清单3-2。

代码清单3-2 copyArray()函数

```
jscript.array.copyArray = function(inSrcArray, inDestArray) {

  var i;
  for (i = 0; i < inSrcArray.length; i++) {
    inDestArray.push(inSrcArray[i]);
  }
  return inDestArray;

} // End copyArray().
```

字面意思上，只是在inSrcArray中循环，并把每一个元素都放到inDestArray中。结果是，inDestArray会被扩展X个元素，并且将包括inSrcArray内容，这里X是isSrcArray的长度。

问题：怎样在一个数组中查找指定的元素呢？

假设用户在页面中输入一系列值。看起来合理的是应该把它们放到一个数组中。如果后面需要查找一个特定值是否在数组中应该怎么做呢？JavaScript本身就可以为我们做这个吗？

回答：不行。我们需要把一些代码放在一起来实现它。代码清单3-3展示了那些代码。

代码清单3-3 findInArray()函数

```
jscript.array.findInArray = function(inArray, inValue) {

  var i;
  for (i = 0; i < inArray.length; i++) {
    if (inArray[i] == inValue) {
      return i;
    }
  }
  return -1;

} // End findInArray().
```

只是在inArray中迭代，并检查每个元素是否匹配我们想找的inValue。如果找到了，就返回所找到值所在的下标，实际上，你找到的时候的确会做些什么操作。如果没有找到，就返回–1，这个几乎是任何类型的查找中的“不，没有找到”的常见返回值。

问题：假设我有一个数值的数组，如何来计算数组中所有元素的平均值呢？

Jack让我添加一些计算所有用户输入的开销项的平均值的功能。自然，他已经指定这些在客户端发生。怎么弄？

回答：你知道如何计算一个平均值，对吧？很好，在一个数值数组中也是同样的方法，如代码清

单3-4所示。

代码清单3-4　arrayAverage()函数

```
jscript.array.arrayAverage = function(inArray) {

  var accumulator = 0;
  var i;
  for (i = 0; i < inArray.length; i++) {
    accumulator += inArray[i];
  }
  return accumulator / inArray.length;

} // End arrayAverage().
```

从inArray开始迭代，把你遇到的所有值叠加在一起，然后将得到的总和除以inArray的长度。你就得到了想要的平均值了。

3.3.2　构建jscript.browser包

本节将开始一个代码包，用来在Web浏览器里使用，它是一个独立的、完整的、与任何特定页面无关的包。

问题：如何获取正在使用应用程序的浏览器的标识信息？

通常，记账类应用程序只支持IE，Jack希望我来纠正这个，我认为第一步就是显示一些关于用户登录网站使用的浏览器的标识信息。那我要怎么做呢？

回答：使用代码清单3-5。

代码清单3-5　getBrowserIdentity()函数

```
jscript.browser.getBrowserIdentity = function() {

  return navigator.appName + " " + navigator.appVersion;

} // End getBrowserIdentity().
```

这个代码将会返回一个字符串，它由浏览器的名称和版本组成，比如：

```
Microsoft Internet Explorer 4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; Maxthon;
WebCloner ; .NET CLR 1.1.4322; .NET CLR 2.0.50727; InfoPath.1)
```

Microsoft Internet Explorer部分是navigator.appName()的结果，剩下的是navigator.appVersion()的结果。

3.3.3　构建jscript.datetime包

本节讲解跟日期和时间有关的代码，并把它们放到我们那个不断扩充的包结构中的一个新包里。

问题：怎样才能很容易地知道一个给定月有多少天，而不必去记那个愚蠢的口诀？

我需要检验用户输入的日期，在某个给定的月份中是不是合法的。比如，如果用户输入31，我就需要确定被输入的月份有31天。

回答：好的，大多数孩子学习一个口诀[①]来记住，那就是：

4月、6月、9月和11月只有30天。

教计算机来理解口诀可能是一个不可能完成的任务，但是幸运的是，匹配这个口诀的“算法”，是相当简单的，如代码清单3-6所示。

代码清单3-6 getNumberDaysInMonth()函数

```
jscript.datetime.getNumberDaysInMonth = function(inMonth, inYear) {

  inMonth = inMonth - 1;
  var leap_year = this.isLeapYear(inYear);
  if (leap_year) {
    leap_year = 1;
  } else {
    leap_year = 0;
  }
  if (inMonth == 3 || inMonth == 5 || inMonth == 8 || inMonth == 10) {
    return 30;
  } else if (inMonth == 1) {
    return 28 + leap_year;
  } else {
    return 31;
  }

} // End getNumberDaysInMonth().
```

首先，需要判断输入的inYear是否为一个闰年。要实现这个功能，我们再写一个名为isLeapYear()的函数用于检查。因为在闰年中，2月有29天，而在非闰年中，只有28天。如果有这个函数，就可以检查inMonth是不是4月(3)、6月(5)、9月(8)或11月(10)，如果是则返回30。

注意第一步是从传入的月份上减去1。调用者按一般逻辑，使用1的值来表示1月，12来表示12月。但是为了使闰年容易计算，我们在函数的内部减去1，那么1月就变成了0，而12月变成了11。这就是为什么4月、6月、9月和11月看起来都少了1。如果inMonth是2月(1)而且是闰年，那么返回28加1，也就是29；如果不是闰年，就返回28加0，也就是28；如果是其他任何月份，返回31。

问题：怎样判断给定的年份是否为闰年？

很有意思你提到了闰年，Gilbert，因为Jack发现了系统在使用闰年的2月的时候有一个漏洞。所以，我需要判断一个特定年份是否为闰年来修补那个漏洞。

回答：这里，我们还需要使用一些数学知识，但也是相当简单的，如代码清单3-7所示。

代码清单3-7 isLeapYear()函数

```
jscript.datetime.isLeapYear = function(inYear) {

  if ((inYear % 4 == 0 && !(inYear % 100 == 0)) || inYear % 400 == 0) {
    return true;
```

① 说实话我自己都记不住，不过下面这个链接里的好像就是大家小学的时候学到的东西：http://www.kidport.com/Grade1/TAL/G1-TAL-Rhymes.htm。

```
  } else {
    return false;
  }

} // End isLeapYear().
```

这个算法基本是这样的：如果年份可以被4整除且不能被100整除，或它可以被400整除的话，那么它就是闰年。

3.3.4　构建jscript.debug包

在本节里，jscript包将继续增长，因为我们会介绍一个新包，它包含有助于调试JavaScript的代码。

问题：如何显示任意的一个对象的所有属性以及它们的值呢？

你知道，Gilbert，我发现在调试JavaScript的时候，使用一些对象，用于查看它的当前状态——它所有的属性和值，这些对象对调试很有帮助。我知道可以使用一个调试器来完成，但是通常一个简单的alert就足够了。有什么办法来那么做吗？

回答：当然有，Bill！代码清单3-8展示了怎么做。

代码清单3-8　enumProps()函数

```
jscript.debug.enumProps = function(inObj) {

  var props = "";
  var i;
  for (i in inObj) {
    props += i + " = " + inObj[i] + "\n";
  }
  alert(props);

} // End enumProps().
```

我们使用for…in循环风格来遍历inObj的属性。对于每一个属性，我们把它的名字（i的值）和值（使用数组表示法来获取成员）添加到一个字符串中。最后，我们将字符串传给alert()，就达到目标了！

问题：如何实现一个健壮的自动记录日志机制呢，类似于Jakarta Commons Logging？

我经常发现想在代码中放一些日志信息，但是不知道在JavaScript中该如何做。比如，在带有Jakarta Commons Logging包的Java中，可以创建一些对象，例如一个日志记录程序，并给它传信息来写日志，而不需要知道底层的日志实现细节。有什么想法吗，Gilbert？

回答：好的，“一个健壮的日志系统”，如果你说的是一个可以基于严重级别来记录（或者不记录）日志的日志系统，而并不是其他的什么，我们当然可以做到。看看代码清单3-9。

代码清单3-9　DivLogger类函数

```
jscript.debug.DivLogger = function() {


  /**
```

```
   * The following are faux constants that define the various levels a log
   * instance can be set to output.
   */
this.LEVEL_TRACE = 1;
this.LEVEL_DEBUG = 2;
this.LEVEL_INFO  = 3;
this.LEVEL_WARN  = 4;
this.LEVEL_ERROR = 5;
this.LEVEL_FATAL = 6;


/**
 * These are the font colors for each logging level.
 */
this.LEVEL_TRACE_COLOR = "a0a000";
this.LEVEL_DEBUG_COLOR = "64c864";
this.LEVEL_INFO_COLOR  = "000000";
this.LEVEL_WARN_COLOR  = "0000ff";
this.LEVEL_ERROR_COLOR = "ff8c00";
this.LEVEL_FATAL_COLOR = "ff0000";


/**
 * logLevel determines the minimum message level the instance will show.
 */
this.logLevel = 3;


/**
 * targetDIV is the DIV object to output to.
 */
this.targetDiv = null;


/**
 * This function is used to set the minimum level a log instance will show.
 *
 * @param inLevel One of the level constants.  Any message at this level
 *                or a higher level will be displayed, others will not.
 */
this.setLevel = function(inLevel) {

  this.logLevel = inLevel;

} // End setLevel().
/**
 * This function is used to set the target DIV that all messages are
 * written to.  Note that when you call this, the DIV's existing contents
 * are cleared out.
 *
```

```
 * @param inTargetDiv The DIV object that all messages are written to.
 */
this.setTargetDiv = function(inTargetDiv) {

  this.targetDiv = inTargetDiv;
  this.targetDiv.innerHTML = "";

} // End setTargetDiv().


/**
 * This function is called to determine if a particular message meets or
 * exceeds the current level of the log instance and should therefore be
 * logged.
 *
 * @param inLevel The level of the message being checked.
 */
this.shouldBeLogged = function(inLevel) {

  if (inLevel >= this.logLevel) {
    return true;
  } else {
    return false;
  }
} // End shouldBeLogged().


/**
 * This function logs messages at TRACE level.
 *
 * @param inMessage The message to log.
 */
this.trace = function(inMessage) {

  if (this.shouldBeLogged(this.LEVEL_TRACE) && this.targetDiv) {
    this.targetDiv.innerHTML +=
    "<div style='color:#" + this.LEVEL_TRACE_COLOR + ";'>" +
    "[TRACE] " + inMessage + "</div>";
  }
  } // End trace().


} // End DivLogger().
```

注意在trace()函数之后，实际上还应该有更多的一些函数：debug()、info()、warn()、error()和fatal()——分别对应每一个日志级别。我省略它们只是为了节约些空间，但是它们本质上就是trace()方法，除了用debug、info、error或者fatal替换trace这个名字。

为了达到这个效果，你需要实例化一个DivLogger，如下：

```
var log = new jscript.debug.DivLogger();
```

这个日志记录程序还有另外一个事情需要做，就是调用setTargetDiv()，给它传递一个 元素的引用，所有日志输出信息都应该写到这里。然后，只要调用log.xxxx(yyyy)，xxxx是指严重级别(trace、debug、info、warn、error或fatal)，yyyy是指日志信息。也可以调用log.setLevel()来设置需要记录的信息的级别。例如，你这么做：

```
log.setLevel(log.LEVEL_ERROR);
```

那么从那点开始，只有LEVEL_ERROR或LEVEL_FATAL级别的信息才会被记入日志。

这是一个非常简易的日志记录程序，它只是在 标签上添加一些信息，并且它还把日志信息的内容添加了颜色，这样当浏览那些输出日志时，可以更容易地识别某种特定类型的信息。你可以很容易地写出另外一个实现，使用Ajax调用来向服务器上的一个数据库写信息，或者写到任何你想要的地方。不论你是否需要targetDiv和代码颜色，它们的基础结构都是一样的。

3.3.5 构建jscript.dom包

在本节中，我们添加一个新的函数包，它将辅助我们来操作DOM。

问题：如何将任意的一个DOM元素居中？

Gibert，现在，当一个表单被提交时，应用程序弹出一个“请等待”的信息。它只是一个z-index设置为大的数值的 ，所以它可以被显示在其他所有东西的上面。不幸的是，那个写代码的临聘人员并不知道如何将 居中，所以它通常是在左上角的位置，Jack为此很不高兴。我怎么将它居中呢？

回答：我有一些修饰代码来做这个。有趣的是，它是在很久以前写的，事实证明，它使用了层(layer)的概念，那个老Netscape术语。那并不是问题，因为它还是可以工作的。代码清单3-10展示了如何将一个元素水平居中。

代码清单3-10 layerCenterH()函数

```
jscript.dom.layerCenterH = function(inObj) {

  var lca;
  var lcb;
  var lcx;
  var iebody;
  var dsocleft;
  if (window.innerWidth) {
    lca = window.innerWidth;
  } else {
    lca = document.body.clientWidth;
  }
  lcb = inObj.offsetWidth;
  lcx = (Math.round(lca / 2)) - (Math.round(lcb / 2));
  iebody = (document.compatMode &&
    document.compatMode != "BackCompat") ?
    document.documentElement : document.body;
  dsocleft = document.all ? iebody.scrollLeft : window.pageXOffset;
```

```
    inObj.style.left = lcx + dsocleft + "px";

  } // End layerCenterH().
```

这可以将任何公开了left样式属性的元素居中，这就意味着它可以处理绝大多数元素，而不只是 元素。它的运作方式是，首先获得浏览器窗口大小——内容填充的实际大小。不同的浏览器具体的处理方法不同：要么是取window的innerWidth属性，要么是取body的clientWidth属性。然后，我们通过对象的offsetWidth属性，来获得它的宽度。最后，就是简单的数据计算，算出对象居中时，X坐标在哪（基本上就是从窗口宽度的一半减去元素宽度的一半）。

还有一些计算要做，就是把页面可能的水平滚动距离计算到总宽度中。我们又要碰到不同的浏览器用不同的方法来获得这个值的情况，更糟糕的是，在IE中，因呈现模式不同还有不同的方法！当呈现一个向前兼容但不向后兼容的形式时，我们需要使用document.documentElement元素；否则，需要使用document.body元素。然后，我们看看document.all元素是否在DOM中出现。如果出现了，则表示我们正在使用IE，然后就访问前面我们已经知道所需要的元素的scrollLeft属性。如果document.all没有出现，那么我们使用的不是IE，所以我们需要窗口的pageXOffset属性。无论怎样，我们现在都有了水平滚动页面的值，把它添加到用来计算居中元素的值里，乌啦！我们已经把它居中在当前展现的页面中了！

你可能会想，这仅是水平居中。那竖直居中怎么做呢？好的，代码几乎是相同的。只是做一些替换：width换成height，style.left换成style.top，scrollLeft换成scrollTop以及pageXOffset换成pageYOffset。代码清单3-11展示了修改后的代码的样子，基本是一样的。

代码清单3-11　layerCenterV()函数

```
jscript.dom.layerCenterV = function(inObj) {

  var lca;
  var lcb;
  var lcy;
  var iebody;
  var dsoctop;
  if (window.innerHeight) {
    lca = window.innerHeight;
  } else {
    lca = document.body.clientHeight;
  }
  lcb = inObj.offsetHeight;
  lcy = (Math.round(lca / 2)) - (Math.round(lcb / 2));
  iebody = (document.compatMode &&
    document.compatMode != "BackCompat") ?
    document.documentElement : document.body;
  dsoctop = document.all ? iebody.scrollTop : window.pageYOffset;
  inObj.style.top = lcy + dsoctop + "px";

} // End layerCenterV().
```

我把它分成了两个独立的函数，因为你可能有时只希望在一个方向上居中，而另外一个方向不居

中。现在这样就可以做到。

说明 在代码清单3-10和代码清单3-11中，你可能已经注意到，我打破了自己在第1章中的一个规则：自描述的变量名称。lca、lcb、lcx、lcy——它们到底是什么意思呢？好，下次你看到我就可以用戒尺打我的手板了！不过，好的规则都是可以偶尔违反一下的。这里的原因可能是，我们正在讨论非常短期存在的变量，它们的上下文可以很容易地在同一屏看到。而且，这些是被作为计算的中间变量来使用，所以它们不含有任何可延续的含义。想想那些执行循环时的计数变量，人们频繁地只使用单独的字母，它们也通常不会困扰任何人。所以，虽然我的确坚持原始的规则，但是偶尔打破也是可以的。只要用自己的头脑判断一下即可！

问题：当使用一个Ajax请求时，我得到一堆返回的内容。如果那堆内容中包含一些<script>段，如何执行它们？

考虑到这个应用的历史，对此我确实有点惊讶，但它确实是使用Ajax来更新一些屏幕的某些部分。遗憾的是，Jack发现它不能完全工作（顾问又在看这个问题），因为从服务器返回的信息中包含一些<script>块，它们没有被执行。有什么方法执行它们吗？

回答：好，当然有，Bill，并且这是一个在使用Ajax时经常遇到的问题。看一看代码清单3-12中是怎样做的。

代码清单3-12 execScripts()函数

```
jscript.dom.execScripts = function (inText) {
  var si = 0;
  while (true) {
    // Finding opening script tag.
    var ss = inText.indexOf("<" + "script" + ">", si);
    if (ss == -1) {
      return;
    }
    // Find closing script tag.
    var se = inText.indexOf("<" + "/" + "script" + ">", ss);
    if (se == -1) {
      return;
    }
    // Jump ahead 9 characters, after the closing script tag.
    si = se + 9;
    // Get the content in between and execute it.
    var sc = inText.substring(ss + 8, se);
    eval(sc);
  }

} // End execScripts().
```

我确信还有其他的方法来写这段代码，比如一些有趣的正则表达式，但是有时我喜欢让生活中简单些，所以使用直接了当的方法。我们传入一个字符串inText，然后开始扫描它，查找<script>块。

（注意，它文字上必须是<script>，而像<script type="text/javascript">就不会被找到。有没有觉得有提高的余地？）如果我们找到了一个，那么就查找它相应的结束标签</script>。找到后，我们就取得那两个标签之间的子串并eval()（执行）它。然后小心地把位置置于我们找到的结束标签</script>后，然后再做同样事情。一旦再也找不到<script>标签了，我们的工作就完成了。

问题：如何引用任意数量的DOM元素？

我知道可以使用document.getElementById()来获得一个DOM元素的引用——这没问题。但是如果我想获得一批元素的引用，写出所有的那些调用就显得有点繁重了，有没有一个简单一点的方法？

回答：当然，一个相对简单的封装函数可以为你节省时间和精力。看看代码清单3-13。

代码清单3-13　getDOMElements()函数

```
jscript.dom.getDOMElements = function() {

  if (arguments.length == 0) {
    return null;
  }
  if (arguments.length == 1) {
    return document.getElementById(arguments[0]);
  }
  var elems = new Array();
  for (var i = 0; i < arguments.length; i++) {
    elems.push(document.getElementById(arguments[i]));
  }
  return elems;

} // End getDOMElements().
```

这个函数会接受可变数量的参数，认为它们是DOM元素ID。记住每一个JavaScript函数天生地指向一个arguments数组，也就是传入这个函数的所有参数的数组。所以，如果没有传入任何参数，这个函数就会返回空，它本身是一个非法调用。如果只传入一个参数，我们就会执行典型的document.getElementById()并返回它，就是这样。但是当传入的参数的不止一个时，那就是它变得有意思的时候了！我们循环arguments数组，对每一个参数执行document.getElementById()。把调用产生的结果压入到新创建的一个数组中，并且当完成时返回那个数组。数组中的每一个元素，现在都是指向一个传入的DOM ID的引用。然后你可以对那个数组做任何你想做的事情，并且不需要自己写出所有的document.getElementById()调用。

3.3.6　构建jscript.form包

在即将要创建的jscript.form包中，介绍了一些协助我们使用HTML表单和表单元素的代码。

问题：如何从一个HTML表单生成XML？

Jack让我为一个应用程序创建一个简单的Web服务接口。为了测试它，我想在应用程序的一个页面上放置一个HTML表单，把它转换成XML，并提交到一个指定的URL。唯一让我头疼的是转换成XML的部分。

回答：这是那些听起来很难的任务之一，但是它实际上非常容易。我已经把代码一起放到代码清

单3-14展示给你了。

代码清单3-14 formToXML()函数

```
jscript.form.formToXML = function(inForm, inRootElement) {

  if (inForm == null) {
    return null;
  }
  if (inRootElement == null) {
    return null;
  }
  var outXML = "<" + inRootElement + ">";
  var i;
  for (i = 0; i < inForm.length; i++) {
    var ofe = inForm[i];
    var ofeType = ofe.type.toUpperCase();
    var ofeName = ofe.name;
    var ofeValue = ofe.value;
    if (ofeType == "TEXT" || ofeType == "HIDDEN" ||
      ofeType == "PASSWORD" || ofeType == "SELECT-ONE" ||
      ofeType == "TEXTAREA") {
      outXML += "<" + ofeName + ">" + ofeValue + "</" + ofeName + ">"
    }
    if (ofeType == "RADIO" && ofe.checked == true) {
      outXML += "<" + ofeName + ">" + ofeValue + "</" + ofeName + ">"
    }
    if (ofeType == "CHECKBOX") {
      if (ofe.checked == true) {
        cbval = "true";
      } else {
        cbval = "false";
      }
      outXML = outXML + "<" + ofeName + ">" + cbval + "</" + ofeName + ">"

    }
    outXML += "";
  }
  outXML += "</" + inRootElement + ">";
  return outXML;

} // End formToXML().
```

让我们假设有如下的HTML表单：

```
<form>
  <input type="text" name="firstName"><br>
  <input type="hidden" name="lastName"><br>
  <input type="password" name="password"><br>
  <textarea name="notes"></textarea><br>
  <select name="gender">
```

```
    <option value="male">Male</option>
    <option value="female">Female</option>
  </select>
  <input type="radio" name="married" value="yes">Yes<br>
  <input type="radio" name="married" value="no">No<br>
  <input type="checkbox" name="haveKids">Check if you have kids</input><br>
</form>
```

这个函数会接收一个表单inForm的引用以及一个字符串inRootElement，它是要创建的XML文档的根元素的名字。在检查了输入值是好的之后，我们开始创建字符串outXML，首先向它添加根元素。

然后开始遍历表单的子节点。对于每一个节点，我们取得它的类型、名字和值。然后，我们查看它的类型是什么。如果是一个text、hidden、password、select-one或一个textarea域，我们向XML字符串中添加一个元素，将元素的名称作为其标签，以及起止标签中间的值作为该标签的值。顺便说一下，select-one在不是多选的时候，是你在类型属性为<select>域中看到的值（上面的代码不能处理多项选择的域……那些我想你可以扩展的!）。对于radio域，我们实际上检查它们每一个，即使它们全都在同一个组里。当然，因为一次只能有一个被选择，所以这并不是问题。XML字符串的生成类似其他元素类型。最后，对于checkbox域，我们将根据对应项是否被选中而发送值"true"或"false"。

那么，假设我们把Person作为根元素传入，并假设表单域中的实体是Frank、Zammetti、myPassword、Hello、Male以及Yes，并且haveKids被选取，就会产生下面的XML：

```
<Person>
  <firstName>Frank</firstName>
  <lastName>Zammetti</lastName>
  <password>myPassword</password>
  <notes>Hello</notes>
  <gender>male</gender>
  <married>yes</married>
  <haveKids>true</haveKids>
</Person>
```

这个XML是以调用者所期望的方式，作为一个字符串返回的，类似通过POST体给一个Web服务提交的东西那样，正如你建议的，Bill。

问题：如何在<select>域里查找并随意地选择一个指定的选项？

当财务应用程序的某个页面第一次显示的时候，有一个有些选项的<select>域，开始的时候，根据不同的情况，其中的某个选项会被选择。如何选择一个指定的选项呢？还有，如果我只是想找到它，并不选择它又该如何做呢？

答案：Bill，你可以简单地使用蛮力方法。同时，即使你还没提到，我想在查找时判断是否大小写相关也是很有用的。代码清单3-15完成了这个功能。

代码清单3-15　selectLocateOption()函数

```
jscript.form.selectLocateOption = function(inSelect, inValue, inJustFind,
  inCaseInsensitive) {

  if (inSelect == null ||
```

```
    inValue == null || invalue == "" ||
    inCaseInsensitive == null ||
    inJustFind == null) {
    return;
  }
  if (inCaseInsensitive) {
    inValue = inValue.toLowerCase();
  }
  var found = false;
  var i;
  for (i = 0; (i < inSelect.length) && !found; i++) {
    var nextVal = inSelect.options[i].value;
    if (inCaseInsensitive) {
      nextVal = nextVal.toLowerCase();
    }
    if (nextVal == inValue) {
      found = true;
      if (!inJustFind) {
        inSelect.options[i].selected = true;
      }
    }
  }
  return found;
} // End selectLocateOption().
```

在通常的烦琐的检查后，我们开始遍历那个inSelect传入的指定<select>元素的选项。检查每个选项的值。如果与我们要查找的那个值inValue匹配，那么就把found标志位设为true，它将会是函数的返回值。还有，我们检查inCaseInsensitive的值。如果它是true，那么匹配就将忽略大小写。如果是false，那么大小写就必须完全匹配。一旦我们找到那个选项，或者查看了所有选项，我们就通过参数inJustFind的值来查看调用者是否要求该选项处于被选择状态。如果值是true，那么就没有什么其他要做的了，但是如果是false，就还需要选定这个选项。最后，返回found的值，如果找到了，它就是true（并不取决于它是否被选择），如果没找到就是false。

问题：如何在一个<select>中提供全选功能？

Gilbert，Jack让我给用户提供在页面中同时选择某个<select>的全部选项的功能。有什么简单的方法来做这个事情吗？

回答：当然有。直接看看代码清单3-16。

代码清单3-16 selectSelectAll()函数

```
jscript.form.selectSelectAll = function(inSelect) {

  if (inSelect == null || !inSelect.options || inSelect.options.length == 0) {
    return;
  }
  var i;
  for (i = 0; i < inSelect.options.length; i++) {
```

```
      inSelect.options[i].selected = true;
    }

  } // End selectSelectAll().
```

inSelect是你想要处理的那个<select>的引用。然后你遍历它包含的选项集，并把每一个的selected设为true就可以了！

我肯定你还想要一个selectUnselcetAll()函数。好的，修改这行就可以了：

```
inSelect.options[i].selected = true;
```

改成：

```
inSelect.options[i].selected = false;
```

你就完成了！

你可能想把它组合在一个函数中，通过传入一个布尔值参数来选择。但是现在你可以让它们保持独立——没有什么坏处。

3.3.7　构建jscript.lang包

在本节中，我们会建立jscript.lang包，它将包含一些在基础的语言水平上帮助我们使用JavaScript的代码。

问题：如何获得一个对象的属性并把它们复制给另一个对象？

在一些情况下，我有一个想和另外一个对象合并的JavaScript对象。换句话说，我想复制一个对象的全部属性给另一个对象。你能告诉我怎么做吗，Gilbert？

回答：有问必答。代码清单3-17是对你的恩赐，Bill。[①]

代码清单3-17　copyProperties()函数

```
jscript.lang.copyProperties = function(inSrcObj, inDestObj, inOverride){

  var prop;
  for (prop in inSrcObj) {
    if (inOverride || !inDestObj[prop]) {
      inDestObj[prop] = inSrcObj[prop];
    }
  }
  return inDestObj;

} // End copyProperties().
```

使用for...in循环遍历inSrcObj的属性。对于每一个属性，查看是否已经在inDestObj中存在，如果存在，通过传入true作为inOverride参数的值，来看看调用者是否让我们覆盖已存在的属性。如果它存在（是真值）我们就重写，如果它不存在，我们使用数组符号设置inDestObj的属性的值。这样做具有添加还不存在属性，或修改已经存在的属性为inSrcObj中找到的值的效果。然后，返回inDestObj，到此就完成了我们的工作了。

① 此处作者引用了《圣经》的一句话。——译者注

3.3.8 构建jscript.math包

现在我们要创建一些代码来实现一些数学函数的功能（噢，好吧，实际上只有一个函数），我们会把它放到一个新包里，适当地命名为jscript.math。

问题：如何在一个指定范围内生成随机数呢？

好的，Gilbert，我承认这个不是Jack想让我做的，因为毕竟在一个财务系统中使用随机数不会是个好事！但是我另外还在做一些JavaScript的游戏，我想知道如何在一个给定范围内生成一个随机数。

回答：哦，好的，Bill，别让我助长你的懒惰！正如你可能已经发现的，在JavaScript中并没有提供这个免费的随机数生成功能，所以我会告诉你如何去做，现在请先看看代码清单3-18。

代码清单3-18 genRandomNumber()函数

```
jscript.math.genRandomNumber = function(inMin, inMax) {

  if (inMin > inMax) {
    return 0;
  }
  return inMin + (inMax - inMin) * Math.random();

} // End genRandomNumber().
```

首先，我们做一个快速的小检查：如果inMin的值（范围的开始值）大于inMax（范围的结束值），那么只要返回0就可以了，这表示调用者做了一些愚蠢的事情，并且我们不想因此坏掉！一旦搬开这个绊脚石，我们就使用在return语句中看到的基本的公式，它总是得到一个在指定范围内的数。

3.3.9 构建jscript.page包

jscript.page包将包含把当前页面当成一个整体来处理的代码。让我们开始创建它吧！

问题：如何用程序启动对当前页面的打印？

Jack让我在最终的报告页面上添加一个打印按钮。我知道用户只要点击浏览器的打印按钮就可以了，但是他坚持让我这样做！

回答：好的，你可以随时调用window.print()，但是只能在最近的浏览器中使用。并且注意，即使是使用我将要给你看的这个函数，用户依然会得到那个通常的打印对话框。没有不需要用户干预的、简单的初始化打印的方法。（这可能是件好事……想象一下正确实现出来你能砍掉的所有树！）[①] 所以，一个小的封装函数就可以了。看看代码清单3-19。

代码清单3-19 printPage()函数

```
jscript.page.printPage = function() {

  if (parseInt(navigator.appVersion) >= 4) {
    window.print()
  }

} // End printPage().
```

① 华盛顿砍树的典故。——译者注

我们只是做了一个快速的版本检查，来确定浏览器是否支持window.print()调用，就是这些。关于这个确实没有太多可说的，Bill。

问题：如何使用一个传入页面的参数呢？

你知道，Gilbert，有些时候我试图使用简单的HTML页面在本地做一些原型。如果我想提交一个表单，并且要提交给另一个HTML页面，有什么方法让我访问这些参数吗？

回答：当然了，Bill，当然有办法。代码清单3-20显示了这个方法。

代码清单3-20　getParameter()函数

```
jscript.page.getParameter = function(inParamName) {

  var retVal = null;
  var varvals = unescape(location.search.substring(1));
  if (varvals) {
    var search_array = varvals.split("&");
    var temp_array = new Array();
    var j = 0;
    var i = 0;
    for (i = 0; i < search_array.length; i++) {
      temp_array = search_array[i].split("=");
      var pName = temp_array[0];
      var pVal = temp_array[1];
      if (inParamName == null) {
        if (retVal == null) {
          retVal = new Array();
        }
        retVal[j] = pName;
        retVal[j + 1] = pVal;
        j = j + 2;
      } else {
        if (pName == inParamName) {
          retVal = pVal;
          break;
        }
      }
    }
  }
  return retVal;

} // End getParameters().
```

这个函数实际上允许我们用名字获取指定参数，或者是在一个数组里获取全部参数。如果传递了inParamName进来，那么就会返回这个名字的参数（如果没有找到则返回空）。如果传了一个空值作为inParamName的值，那么就会返回所有参数的数组。

location.search.substring(1)是获取查询字符串的引用的方法。通过从URL的第2个字符开始（就是(1)参数做的事情），我们可以删除开头的问号，只留下参数。然后，只要对字符串调用split()函数，用与号（&，就是分隔参数的符号）把每个参数分开，就得到一个数组。然后遍历数组，每个元素进

一步用等号分割，因为每个参数都是一个name=value对。

最后，就是看看指定的参数是否要求返回，或者所有参数都被返回，这个很简单。对于后者，我们一开始会初始化一个新的数组，向里面添加参数以及其值，因此这个新数组的样子就是：name,value,name,value等。在处理完参数之后，返回这个数组。如果是要求返回指定参数，那么一找到该参数，就马上返回其值。

问题：如何使用JavaScript打破一个框架？

Jack并没有让我做这个，我自己注意到这个问题。我们那个主页，是所有用户开始的地方，它有到所有应用程序的链接，包括这个财务应用程序。不幸的是，首页使用框架建立的，而且当你去到任何一个应用程序的时候，依然在那个框架里面。大多数情况下这是不好的，所以我想要给应用程序提供一种方式来打破那个框架。

回答：你知道现在有多少人十分讨厌框架吗？有自尊的Web开发者很少再使用框架了。不过，我不受这种笨蛋的困扰，所以并不介意使用得当的框架。不过我离题了。我用了代码清单3-21来回答你的问题。

代码清单3-21 breakOutOfFrames()函数

```
jscript.page.breakOutOfFrames = function() {

  if (self != top) {
    top.location = self.location;
  }

} // End breakOutOfFrames().
```

打破框架是一个简单的事情，只要确保浏览器中的文档也是在顶部的。也就是说，如果它是一个框架，那就是父框架文档。如果这个文档不是顶部文档，那么我们就把当前文档设置为顶部文档，这通常引发任何的框架被新文档（服务器返回的新结果）重写。

3.3.10 构建jscript.storage包

客户端存储并不真的都那么复杂，但是我们可以用一些实用的函数来使它更简单，就这是我们正在jscript.storage包中整合的东西。

问题：如何创建一个cookie并把它保存在客户端呢？

Jack指出这是应用程序的一部分，我们在服务器上存储用户的个人设置。他认为（我赞同）把它存储在客户端会更高效。我知道存储在客户端有一些JavaScript的局限，但是看起来在这里用cookie很合适。该如何创建一个呢？

回答：你是对的，cookie对这样的事情来说非常完美：每个网站在客户端只存储一小部分数据。让我们讨一下cookie怪兽的欢心，做一块“饼干”吧，如代码清单3-22所示。

代码清单3-22 setCookie()函数

```
jscript.storage.setCookie = function(inName, inValue, inExpiry) {

  if (typeof inExpiry == "Date") {
```

```
    inExpiry = inExpiry.toGMTString();
  }
  document.cookie = inName + "=" + escape(inValue) + "; expires=" + inExpiry;

} // End setCookie().
```

每个cookie有一个名和一个值，明显地，还有一个过期时间。因此，这个函数接受这3个参数：inName、inValue和inExpiry。时间需要GMT日期字符串的形式，所以我们要允许inExpiry可以为一个实际的Date对象，或者可能是一个GMT日期格式的字符串。如果是一个Date对象，调用它的toGMTString()方法获得适当的格式。然后，设置document.cookie等于一个字符串，它的形式是xxxx=yyyy;expires=zzzz，其中，xxxx是cookie的名字，yyyy是值，zzzz是日期字符串。把一个文档的属性设置到一个cookie中，对你来说可能看起来有点奇怪，还有，这是否意味着如果我们试着设置另外一个cookie，第一个cookie会被重写呢？不用担心，因为浏览器完全可以处理它。那只不过是一点语法上的古怪，可能是从Netscape时期留下的。

问题：如何获得一个指定cookie的值呢？

设置cookie很容易，那然后怎么取它的值呢？

回答：这也不难，Bill。仔细看一下代码清单3-23。

代码清单3-23　getCookie()函数

```
jscript.storage.getCookie = function(inName) {
  var docCookies = document.cookie;
  var cIndex = docCookies.indexOf(inName + "=");
  if (cIndex == -1) {
    return null;
  }
  cIndex = docCookies.indexOf("=", cIndex) + 1;
  var endStr = docCookies.indexOf(";", cIndex);
  if (endStr == -1) {
    endStr = docCookies.length;
  }
  return unescape(docCookies.substring(cIndex, endStr));

} // End getCookie().
```

当找回document.cookie的值时，你得到的是一个巨大的字符串，它由那个页面可得到的所有cookie组成。所以，想找到你感兴趣的那个cookie最简单的方法就是，查找字符串xxxx=，xxxx是你想要的那个cookie的名字。

如果调用indexOf()得到的返回值是-1，那么就表示没有那个cookie，所以只要返回空。如果找到了，那么需要找它的结束点。因为所有的cookie都有;expires=zzzz字符串跟在后面，所以我们可以查找分号。一旦有了我们想要的那个cookie的开始和结束位置，就只返回子串，确定对其使用unescape()，因为它是用URL编码的字符串存储，这样调用者就很开心啦。

问题：如何删除cookie？

好，Gilbert，那么我可以创建和找回cookie了。就还剩下一件事情：怎么删除它们？

回答：好的，严格地讲，你实际上不能完全删除一个cookie。不过，你可以找回那个cookie，把它的过期时间改变为一些已过期的时间，并重新设置，那样会重写已存在的cookie，并且浏览器会立即看到它已经过期了，然后就直接删除它。详细情况请看代码清单3-24。

代码清单3-24 deleteCookie()函数

```
jscript.storage.deleteCookie = function(inName) {

  if (this.getCookie(inName)) {
    this.setCookie(inName, null, "Thu, 01-Jan-1970 00:00:01 GMT");
  }

} // End deleteCookie().
```

我们使用之前创建的getCookie()函数，然后把返回值传递给前面说过的setCookie()。我们还要传进一个带过期时间1970年1月1日的字符串。除非系统时钟非常混乱，cookie都会被设置，覆盖那个已经存在的cookie。然后立即就会过期了，并被浏览器删掉。我想这么做可能有一些绕，但是它绝对可以用！

3.3.11 构建jscript.string包

现在到了最后一个要为我们的库创建的包——jscript.string包。我相信你可以猜到它的目的：帮助我们使用字符串！

问题：如何计算一个子串在字符串中出现的次数？

Jack要求的一个功能是，检查一些用户可以输入的自由格式文本并看看指定的关键词出现了多少次，这样我们可以判断输入是否可疑。如何计算给定字符串中指定的子串出现了多少次呢？

回答：其实并不是大问题。看看代码清单3-25。

代码清单3-25 The substrCount()函数

```
jscript.string.substrCount = function(inStr, inSearchStr) {

  if (inStr == null || inStr == "" ||
    inSearchStr == null || inSearchStr == "") {
    return 0;
  }
  var splitChars = inStr.split(inSearchStr);
  return splitChars.length - 1;

} // End substrCount().
```

我们可以调用这个函数，并传给它一个待查询的字符(inStr)和一个被查找的字符串(inSearchStr)。首先，函数做了一些简单的排查来确定输入的变量是合法的。如果其中有非法的，它就会返回0。然后，它使用了一个JavaScript String对象的方法split()。这个方法本质上类似于Java的StringTokenizer。它把字符串切割为片段，在指定的子串分隔它。所以，也就是说，如果我们有字符串Sally sells seashells by the seashore on this dreary day，并且想把它在ea子串的位置进行切分，它会被分成这样：Sally sells seashells by the seashore on this dreary day。结果是4段：Sally sells s, shells

by the s,shore on this dr,and ry day。

实际上调用split()返回的是一个数组，这个例子中，就是像说明的那样包含了4个元素。所以所有需要做的就是返回数组的长度减1，这就是答案。“为什么减1？”你可能会问。想想字符串XYZ在Y处截断。那么数组的长度将会是2，X和Z是组成元素。但是我们想知道Y出现了多少次，那总是split()返回的结果数组的长度减1。如果我们查找的元素没有找到，那么被split()返回的数组还有一个单独的元素：那个我们试图切分的字符串。所以从数组的长度上减去1仍然返回了正确的结果0。

“那有意义吧？”Gilbert问。“哦，是的，绝对是！”Bill回答。“但是我还有问题，”他补充说。

问题：如何从一个字符串中删除指定的字母，或者换一种说法，从字符串中删除除了指定字母之外的其他任何字母？

Jack还想让我修改页面的代码，让用户输入费用分类。现在看起来，用户可以输入任何东西，即使只有数字才是合法的。我想我应该写个函数来删除那些不再被允许的字母列表中的所有字母，还要只删除出现在禁止列表中的字母，只是要确保我将来的基础都照顾到了。

回答：这些目标没有非常困难的。基本上都是扫描源字符串和检查每一个字母。如果它匹配了任何其他字符串的字母，那么要么是复制，要么是不复制到某个新字符串里。你可以写两个独立的函数来做这件事，但是单独的一个应该就可以了。代码清单3-26展示了做了这两种类型的删除的函数。

代码清单3-26　stripChars()函数

```
jscript.string.stripChars = function(inStr, inStripOrAllow, inCharList) {
if (inStr == null || inStr == "" ||
  inCharList == null || inCharList == "" ||
  inStripOrAllow == null || inStripOrAllow == "") {
   return "";
}
inStripOrAllow = inStripOrAllow.toLowerCase();
var outStr = "";
var i;
var j;
var nextChar;
var keepChar;
for (i = 0; i < inStr.length; i++) {
  nextChar = inStr.substr(i, 1);
  if (inStripOrAllow == "allow") {
    keepChar = false;
  } else {
    keepChar = true;
  }
  for (j = 0; j < inCharList.length; j++) {
    checkChar = inCharList.substr(j, 1);
    if (inStripOrAllow == "allow" && nextChar == checkChar) {
      keepChar = true;
    }
    if (inStripOrAllow == "strip" && nextChar == checkChar) {
      keepChar = false;
```

```
        }
      }
      if (keepChar == true) {
        outStr = outStr + nextChar;
      }
    }
    return outStr;

  } // End stripChars().
```

在一个快速的简单排查，只是为了确认输入值合法之后，我们开始扫描源字符串inStr。对于每一个字符，我们扫描被允许（或不被允许的）值的列表，它是通过inCharList传入的。对于每一个字符，我们都检查inStripOrAllow参数的值，它表示允许或删除。如果是允许的，并且我们正在检查的inStr中的字符在inCharList中出现，那么就保留住这个字符。如果inStripOrAllow是删除，并且当前字符也出现在inCharList中，那么这种情况下，我们就删除那个字符。为了理解这个动作，注意在内部循环开始之前设置keepChar的方式。

当我们在检查被允许的字符时，假设字符没有在列表中找到，那么它就不被允许。相反，当我们删除时，假设字符会被允许除非它在列表中被找到。最终，不管在哪种情况下被保留下来的字符都被添加到outStr中，然后返回。所以，那个返回的字符串会删除任何指定的字符。

问题：如果我不想真正地改变那个字符串，只是测试一下是否只包含合法字符或者只包含非法字符，该怎么做？

stripChars()确实很方便，但是我想有些地方只是想测试一下一个给定的字符串是否只包含合法字符，或者只是想看看是否包含非法字符。

回答：这也不是什么大问题，它和stripChars()非常类似，但是我想我们可以更高效来完成它，如代码清单3-27所示。

代码清单3-27 strContentValid()函数

```
jscript.string.strContentValid = function(inString, inCharList, inFromExcept) {

  if (inString == null || inCharList == null || inFromExcept == null ||
    inString == "" || inCharList == "") {
    return false;
  }
  inFromExcept = inFromExcept.toLowerCase();
  var i;
  if (inFromExcept == "from_list") {
    for (i = 0; i < inString.length; i++) {
      if (inCharList.indexOf(inString.charAt(i)) == -1) {
        return false;
      }
    }
    return true;
  }
  if (inFromExcept == "not_from_list") {
    for (i = 0; i < inString.length; i++) {
```

```
        if (inCharList.indexOf(inString.charAt(i)) != -1) {
          return false;
        }
      }
      return true;
    }

  } // End strContentValid().
```

还是那样，从一个简单的排查开始，不过这通常是个好主意！之后，我们又扫描了输入字符串inString。不过，这次我们的工作简单多了，因为并不需要扫描整个字符串。所有需要做的就是判断当inFromExcept是from_list时当前的字符是否没有在inCharList中出现。如果没有出现，就返回false。在inFromExcept是not_from_list的情况下，我们检查确定当前的字符并没有出现在inCharList中，如果出现，也返回false。如果扫描完整个inString也没找到，就返回true。

问题：如何在一个字符串中替换出现的所有某个子串？

我知道在JavaScript中String对象有replace()方法，可以把字符串中的一个子串替换成另一个子串。不过，如果我想把一个字符串中所有的某个子串全部替换该怎么做呢？

回答：你已经深切注意到内置函数replace()的这个缺点了，Bill。幸运的是，处理所有的出现位置也并不是大问题，即使它需要你做一些工作。代码清单3-28展示了如何完成它。

代码清单3-28　replace()函数

```
jscript.string.replace = function(inSrc, inOld, inNew) {

  if (inSrc == null || inSrc == "" || inOld == null || inOld == "" ||
    inNew == null || inNew == "") {
    return "";
  }
  while (inSrc.indexOf(inOld) > -1) {
    inSrc = inSrc.replace(inOld, inNew);
  }
  return inSrc;

} // End replace().
```

是的，确实是这样！就是简单循环的事情，在inSrc中查找inOld，每次找到后，都把它替换成inNew。在循环中持续这么做，直到inSrc中不再出现inOld，就完成了。不能再容易了！

问题：我如何从字符串的开头删掉空格？

我知道在大多数其他语言中，字符串类都有方法可以从一个字符串的开头删除空格，通常是leftTrim()或其他类似的。JavaScript却不能，我怎么做呢？

回答：哦，拜托，Bill，你能给我一些更有挑战的吗？代码清单3-29就是你的答案。

代码清单3-29　leftTrim()函数

```
jscript.string.leftTrim = function(inStr) {

  if (inStr == null || inStr == "") {
    return null;
```

```
    }
    var j;
    for (j = 0; inStr.charAt(j) == " "; j++) { }
    return inStr.substring(j, inStr.length);

  } // End leftTrim().
```

实际上只需要找到字符串里第一个非空白的字符，我们的做法是遍历inStr，直到碰到一个非空格为止。然后我们使用内置的substring()函数并返回从循环变量指示的位置（就是第一个非空格的位置）开始到结尾的子串。

噢，对了，在你问之前，很容易写一个rightTrim()函数，如代码清单3-30所示。

代码清单3-30 rightTrim()函数

```
jscript.string.rightTrim = function(inStr) {

  if (inStr == null || inStr == "") {
    return null;
  }
  var j;
  for (j = inStr.length - 1; inStr.charAt(j) == " "; j--) { }
  return inStr.substring(0, j + 1);

} // End rightTrim().
```

逻辑上是一样的，除了我们这次是从字符串后面向前面遍历之外。因为我们需要找最后一个非空白字符的位置。然后我们返回从字符串开始到最后一个非空白字符位置的子串。

我猜你接着想问的就是如何能同时裁两边？当然，只要轮流调用两个函数就行，但可以做得更简单些。代码清单3-31显示了fullTrim()方法，用起来比较方便。

代码清单3-31 fullTrim()函数

```
jscript.string.fullTrim = function(inStr) {

  if (inStr == null || inStr == "") {
    return "";
  }
  inStr = this.leftTrim(inStr);
  inStr = this.rightTrim(inStr);
  return inStr;

} // End fullTrim().
```

可以直接用我们刚写的代码，对不？所以我们先后调用leftTrim()和rightTrim()就可以了。

问题：我怎样才能把一个字符串分割成几个指定长度的片段？

Gilbert，最后一件跟字符串有关的事情。我们有个任意输入的文本输入框，用于对开销的脚注，Jack希望我把它存储在若干个数据库字段里，每个是100字符长。我怎么才能把用户输入的东西分割成100字符长的小段呢？

回答：虽然把字符串分成几段并不难，但是要注意的是我们不能在一个词的中间把字符串给切断

了。Jack当然不会喜欢这个，对吧？所以，我们必须考虑这个问题。代码清单3-32显示了很多种解决这个问题的方法中的一种：

代码清单3-32　breakLine()函数

```
jscript.string.breakLine = function(inText, inSize) {

  if (inText == null || inText == "" || inSize <= 0) {
    return inText;
  }
  if (inText.length <= inSize) {
    return inText;
  }
  var outArray = new Array();
  var str = inText;
  while (str.length > inSize) {
    var x = str.substring(0, inSize);
    var y = x.lastIndexOf(" ");
    var z = x.lastIndexOf("\n");
    if (z != -1) {
      y = z;
    }
    if (y == -1) {
      y = inSize;
    }
    outArray.push(str.substring(0, y));
    str = str.substring(y);
  }
  outArray.push(str);
  return outArray;

} // End breakLine().
```

首先最重要的事情：确定我们有一个需要切断的字符串，并且还要确认指定的大小大于等于1，因为任何其他的都没有意义。同时，检查inText的大小是不是小于或等于指定的大小。如果是的话，就返回inText，我们就做完了！

在完成那些检查之后，把inText复制到一个名为str的变量中，并且开始一个循环，它将循环到str比指定的大小更长。看，每一次循环递归，我们都会减少str，到最后它会比inSize更短，此时循环就会结束。在循环中，我们获得了一个子串，它的长度等于inSize。然后我们找到字符串中最后的空白或者换行的位置。如果找到，我们就设置变量y为它的位置。如果都没有找到，y被设置为字符串的大小。最后我们把子串压到outArray中，并且把刚刚移走的片段切下来，然后循环再次开始。最后，我们返回outArray，它包括inText，切断成适当大小的段（或小一点儿，取决于在哪里切断）。

“好的，”Bill说 ，“这堂课够长的！我学了很多东西。谢谢你，Gilbert!”

“嗯，这都是一天的工作，” Gilbert显得有些得意，“我会让你成为一个好的程序员的。”

“好，有你教我的这些，我应该可以完成Jack所有的需求了，”Bill说，“我应该可以给Jack很好的印象，估计他很快就会给我你期望的提升！哈哈，现在谁才是酷哥？谁才有疯子样的技能？我赢了，

你输了，瘸子！”[①]（需要理解这些密码，请访问http://www.learnleetspeak.com。）

Gilbert 坐着，惊呆了，看着张牙猛笑的Bill，慢慢地才反应过来刚发生了什么。他张开嘴，刚想反唇相讥，Bill却在他说出第一个音符之前就转身离开了这个房间，脚步踏得震天响。

Gilbert看着自己的呆伯特杯子，自己的一堆IT证书，自己的1:100的联邦旗舰企业号D和E（科幻电影《星际迷航》里面的飞船）的模型，自己上个万圣节用过的达斯•摩尔[②]的面具，发现自己很可能很快就要把这些东西都装包带走了。

Gilbert在Initech工作了五年之后，第一次意识到自己能呆的日子不多了，也不能再自大自负了。

3.4 测试所有代码片段

噢，回到现实，这次要听好！

这本书中的源代码都可以从Apress网站（http://www.apress.com）的Source Code/Download区下载到。一个可以下载，但是本章里却没有出现的部分，是测试HTML文档，用它可以练习我们在这里创建的所有函数。它同样也是一个表单格式的文档，因为它演示了所有函数的基本用法。我强烈建议抓取出那些源代码，并看一下所有这些是如何在一起工作的。去吧，Gilbert不会介意的！事实上，在后面的几章里，你确实需要那些源代码。如果你现在就抓取了所有的东西，那么就能顺利地通过后面的几章。

一个快速的演示，只是为了说明当运行应用程序时你会看到什么。图3-1展示了jscript.array包中可用的测试，以及弹出窗口显示的findInArray()函数的结果。

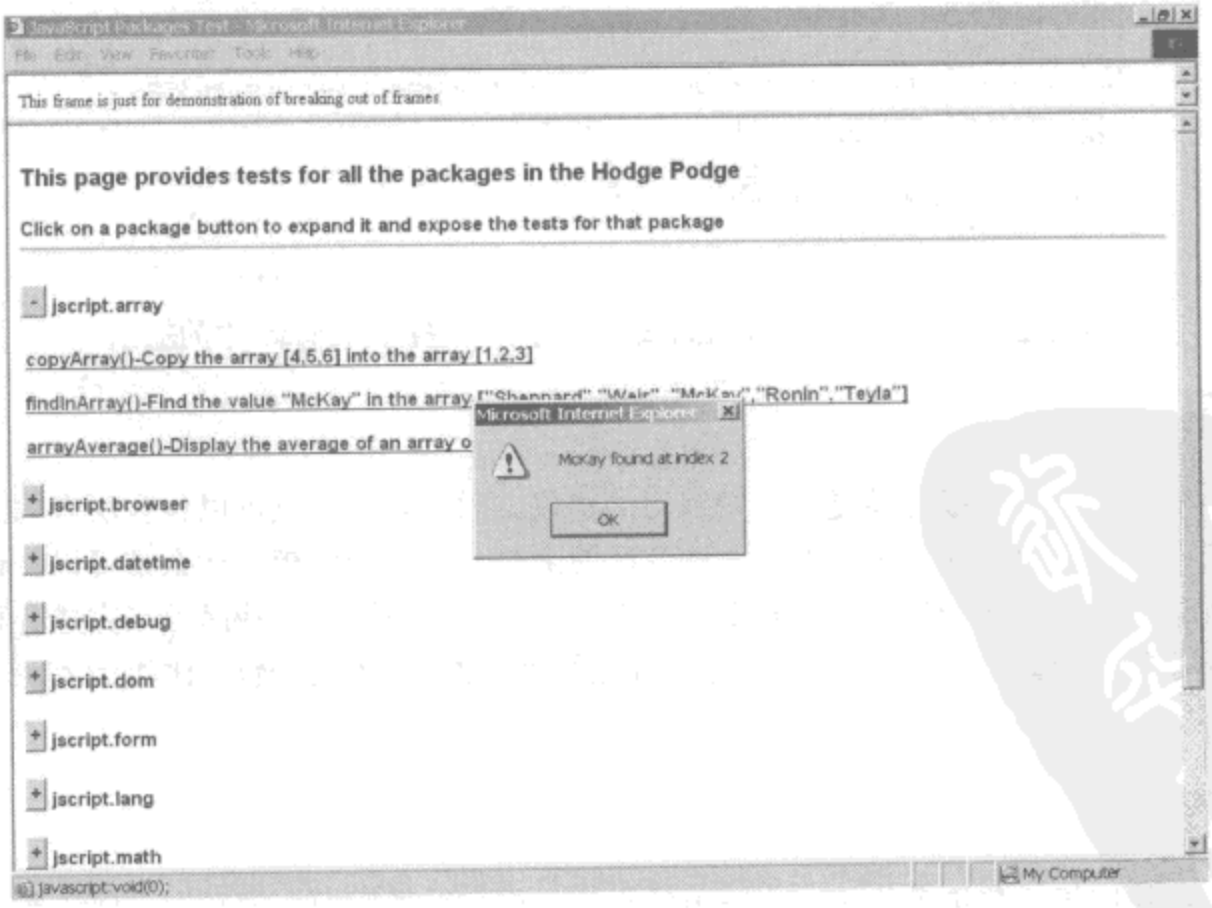


图3-1 对jscript.array包的一些测试，弹出框展示了findInArray()函数测试的结果

① 原文为：who's the C00l D00D with the M4d 5KiLL2 now, huh? I R0><0R, j00’ r3 0WN3D, ll4M4！"。——译者注

② 星站前传中使用双向光剑的西斯武士。——译者注

另外一个使你胃口大开的例子是图3-2中展示的jscript.debug包测试组，尤其是这里会起作用的DivLogger。（糟糕的是，你不能识别这里屏幕截图的代码颜色，但是请相信我，每一个信息都是有颜色的代码的!）

这个测试页面说明的另一个重要的方面是，如何把这些作为一个库来使用。事实上，找到所有自己在页面中想要的各种包的源文件，然后用合适的<script>标签“导入”那些包，就是所有我们需要做的。除了这些就不需要其他什么包了——不需要生成一个DLL或其他什么东西。不过，很有必要给这些源代码文件单独建立一个目录。这样当你想在其他项目中引用这个库的时候，只需要复制一下单独的目录就可以了。

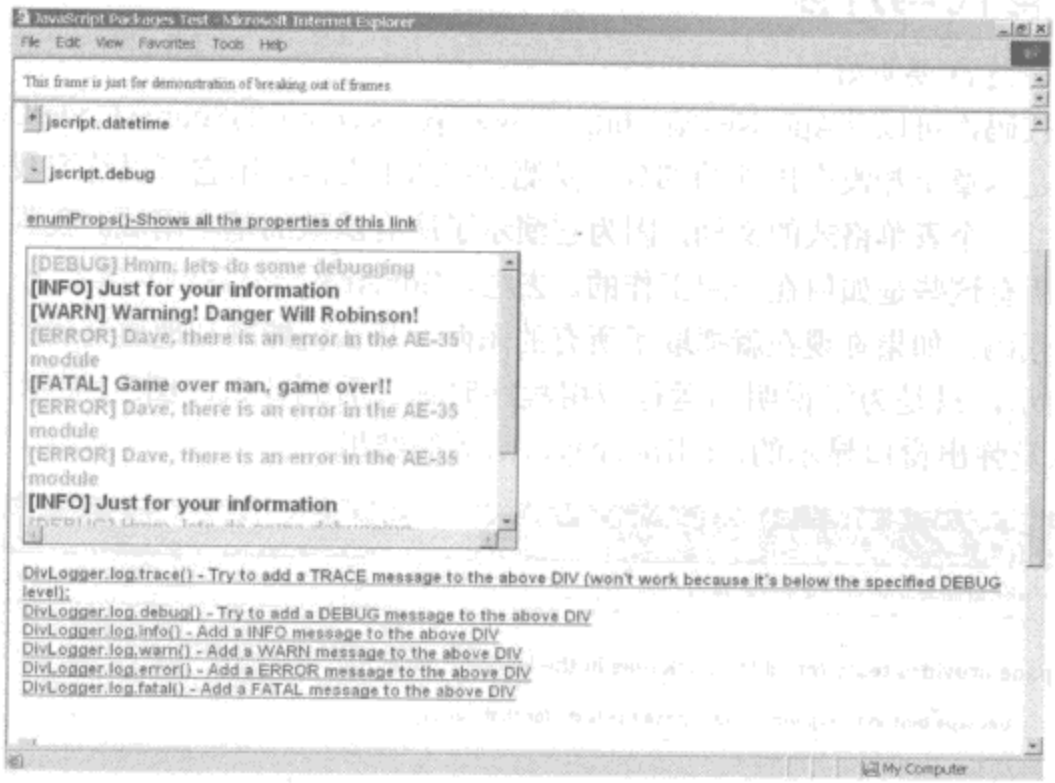


图3-2　jscript.debug包的测试，使用了一些在DivLogger中的输出

3.5　练习

面对这样的一章，我们很容易提出一些练习的建议，因为有个建议可以包含所有内容：添加一些东西！我建议向那些已存在的包中，添加各种各样的函数——无论你想要什么都可以，让它们变得唾手可得。我还建议添加一两个包，只是看看工作机制。如果你想添加一些函数来完成各种功能，可能要在jscript下添加一个新的包，也可能给一个已存在的包添加一个新的子包，比如jscript.dom.effects。可能性是无限的。

3.6　小结

在本章中，我们将函数集合成了一个精美的小库，你会发现后面会很有用。你还看到了如何创建一个基本的包结构，帮助我们避免命名空间的冲突、避免过多使用全局变量以及避免到处都是函数。

很可能并非所有的函数都会在这本书的项目中被使用。（我在写项目代码之前写本章，所以有些猜测可能是错的。）但那并不碍事，因为它们是有用的并会帮助你继续前行。

第4章

CalcTron 3000：JavaScript计算器

自从巴比伦人首先在沙子上把石头摆成一些行进行计数以来（或者如一些权威说的可能是中国人——我并不是历史学家，所以我把那个争论留给更有资格议论的人们），无论达斯廷·霍夫曼[①]还是拉塞尔·克罗[②]，计算器或者运算者都扮演了一个重要的角色，在人类每天的生活中，为什么不把这个东西带到现代，并使用JavaScript为自己创建一个呢？

除了简单的加、减、乘、除功能，计算器（dubbed CalcTron）还将包括一些其他的常用功能，比如百分比、平方根，并且因为我们是程序员，所以还包括进制转换。当然，这些并不足以讨一个奇客的欢心，所以我们将把这个做成一个完全的可扩展计算器，可以根据需求在上面添加功能。我们还会通过使用一些样式和比较酷的特效，竭尽全力让界面有点新意。然后看看是不是增加更多的特性可以让之获得感觉并最后征服世界，不过事情还得一件一件地办！

4.1 计算器项目的需求和目标

一个计算器根本上并不是一个复杂的项目，只要你别试图包含所有可能的功能点。同时，它应该是一个很好的项目，可以揭示一些JavaScript的概念，并让你思考一些东西。让我们抛出一些需求，它们会有助于完成那个目标。

- CalcTron应该展现一个相对灵活的界面，它可以在我们添加新功能点的时候改变。特别地，我们将允许CalcTron在几个模式中切换，每一个都有它自己定义的布局（包括在一些预定义的约束）。让我们允许用JSON定义这些布局。
- 一个计算器基本上不是视觉上最刺激的项目，所以为了减少厌倦，我们将在可能的地方放置一些特效和视觉变化。我们准备使用库来实现这个目标，以便尽量节省劳动。
- CalcTron应该是可扩展的，允许我们添加所需的新功能。

我承认，这是个相当短的列表。然而，一旦我们开始编码之类的事情，你将会看到一个表面上看很小的项目，实际上可不那么简单。

4.2 CalcTron预览

我们先看看CalcTron的样子，就是图4-1的样子。

① 在电影《雨人》中扮演一个有计算天赋，其他方面却属于弱智儿童的计算天才。——译者注

② 在电影《美丽心灵》中扮演普林斯顿数学天才纳什。——译者注

图4-1　标准模式的CalcTron

CalcTron默认提供了两种操作模式：标准模式，如图4-1所示；BaseCalc[①]模式，如图4-2所示。

图4-2　BaseCalc模式下的CalcTron

① 基础运算。——译者注

当点击Mode按钮时，程序会给你展示一个弹出对话框，它飞到屏幕上方并允许你为计算器选择一个新的操作模式，如图4-3所示。这个弹出框非常简单，但它是使CalcTron可扩展的不可分割的一部分。

你在截屏中看不到的一个东西是，这个弹出框会随机的从4个角中的任意一个飞入到浏览器的中央。这个飞行特效是借助于一个名叫Rico的库来完成的。除了飞行弹出框之外，Rico做了计算器本身的圆角。让我们迅速看一下Rico都提供了些什么功能。

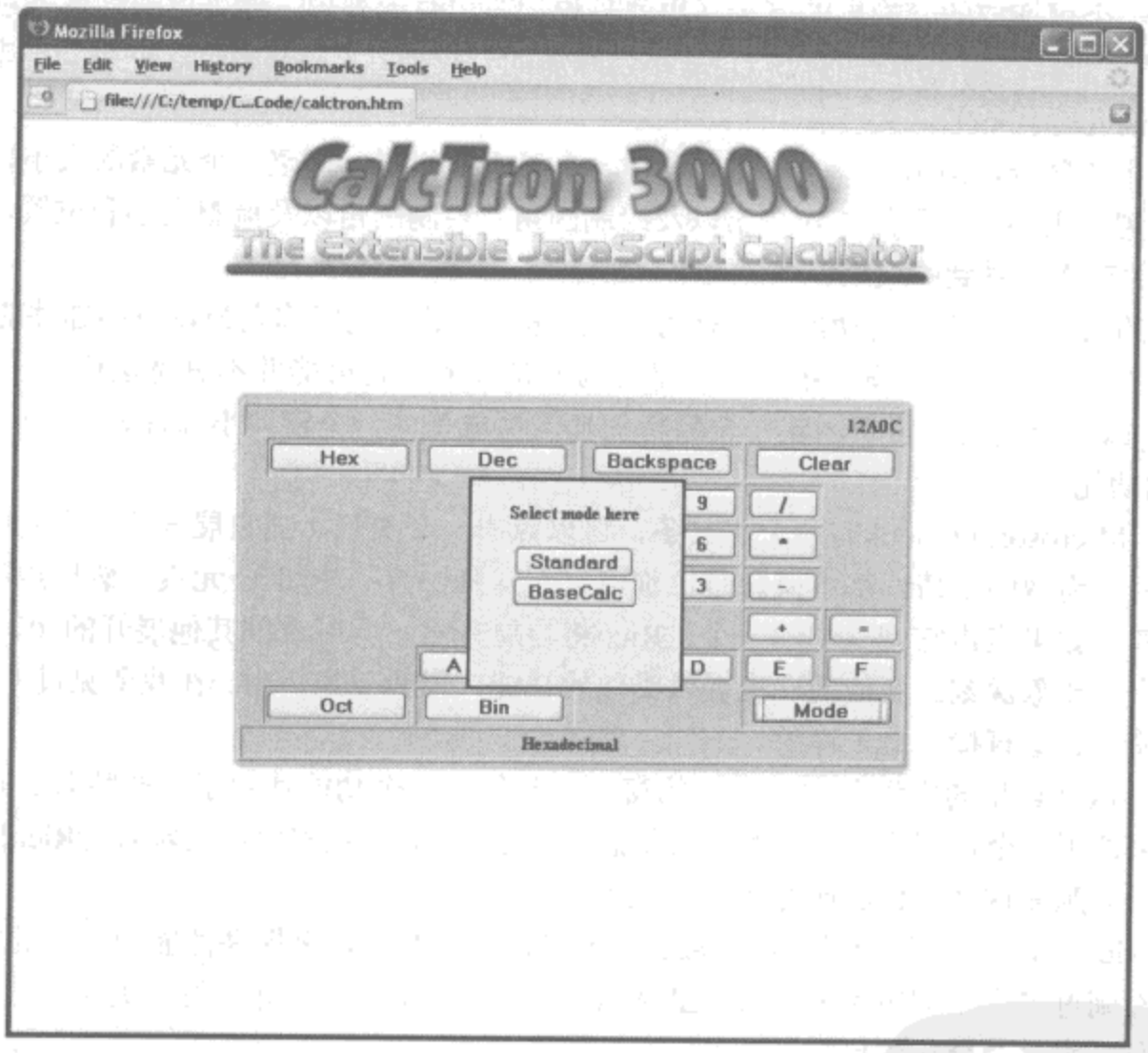


图4-3 修改模式的弹出对话框

4.3 Rico特性

我在第2章中和一些其他的具有代表性的JavaScript库一起介绍了Rico。Rico（http://openrico.org）是一个短小的库（比如和Dojo比起来），它只涵盖了相当少的方面，但是做的非常好。Rico提供4个主要方面的功能：Ajax、拖放控制、剧院特效和行为。

Rico是一些在Prototype库基础上建立起来的库之一。Rico本身放在一个相当小的JavaScript（88 kb）文件中。加上46 kb的Prototype（因版本而异），你可以看到它并非很大。如它们自称的，它里面东西不少。

Rico提供两个Ajax函数：一个被用于更新标签元素的innerHTML，另一个基于XML响应来更新多个元素。这两个函数都使用一个有趣的模型——Rico提供的Ajax引擎，注册一个给定的Ajax请求，并给它一个ID。然后可以在不同的环境和不同时间里，通过引用这个ID来重用这个请求。后者期望从服务

器得到一个XML响应，然后用它填充屏幕上的元素。如果进入Rico的网站并导出样本程序（可以通过点击Demos链接得到），你就会看到一些关于这个的很好的例子（以及这里描述的其他所有东西）。

Rico还提供一些漂亮的拖放支持。除了可以使任意元素（通常是 元素）可拖放之外，它还允许你定义放置的区域。这就意味着，你可以让一个 可拖动，并将另一个 定义为一个放置区域。当拖动第一个 到第二个的时候，它就成为了放置区域的一个子节点。Rico实例页面展示了这个例子，还有一个可拖放的交换框的例子（也就是说，左边有一列项，你可以拖动这些项到右边的列表框中）。如果完全自己编写所有的代码会是相当麻烦的一场战斗。Rico则把这件事变得非常容易（大多数情况下只需几行代码）。

在剧场效果的部分，Rico提供的功能如调整一个元素的位置、调整一个元素的大小以及同时调整一个元素的位置和大小、元素的淡入淡出以及绘制圆角。绘制圆角以及调整大小和位置，是两个你将会在CalcTron的动作中看到的效果。

Rico提供的另外一个方面的功能叫做行为。行为是大多数其他库称之为窗口小部件的东西；至少，在Rico中，现在来说，这点（窗口小部件就是行为）是真的（它可能并不总是这样）。行为通常与剧场效果和/或Ajax一起使用，用于创建一个独特的功能性成员（一个窗口小部件）。下面是一些Rico目前可以提供的功能。

- 折叠：Microsoft Outlook有一个工具条，可以点击一个类目，把它展开成一个视图。这大致就是折叠行为做的事情。基本概念是，你可以在屏幕中有一堆 元素。然后声明它们组成一个折叠。如果点击它们其中的一个，Rico将会展开它，同时收缩其他展开的 。这个效果确实令人印象深刻。其他的库也提供类似的功能，但我要说，Rico在我所见过的实现方法中，是最简单、最直接、最干净的。
- 天气：这是典型的告诉我本地天气的窗口小部件。这个功能使用了一些特效、折叠行为以及Ajax来调用一个远程服务器获得天气信息。再说一遍，我强烈建议大家看看Rico网站的实例页面，因为观看这些动作的确令人印象深刻。
- LiveGrid：这是一个使用了Ajax连接、缓冲以及压缩等策略来提高性能的典型的数据表格。你可能看到过多个不同版本的实现，虽然Rico实现的也挺好，它并不特别突出。它确实有用，但是在我看来只是合格而已。

如你所看到的，Rico并不能涵盖所有JavaScript和RIA开发者会有的需求。它只是针对几个目标区域，但它在这几个方面做得都很不错。尤其是拖放支持，是我所见过的最好的实现之一。在第10章的项目中，我们将使用MochiKit来支持拖放。虽然那个库也提供了非常好的拖放功能，但是如果必须要再重做一遍的话，坦率地说，我会选择Rico做这个事。那并不意味着就一巴掌彻底打死MochiKit，你将看到它也是一个很好的库。我只是强调在这方面，Rico的功能集是多么的好。

我还觉得折叠行为真的非常棒。我曾考虑过把它硬塞进CalcTron中，不过后来决定还是不强加了。

提示　我说过很多遍了，但是现在还是值得重复一下：花一些时间在Rico的示例页面，看看所有的这些东西实际的效果。我想你一定会喜欢在那里看到的。

现在你已经看到了CalcTron和Rico了，并且理解了一些我们的目标和对这个项目的期望（我还希

望你已经玩了一会儿CalcTron了），那我们就开始探究它们是怎么实现的吧。

4.4 剖析CalcTron的解决方案

为了弄清楚CalcTron是如何组织在一起的，让我们仔细看看应用程序的目录结构，如图4-4所示。

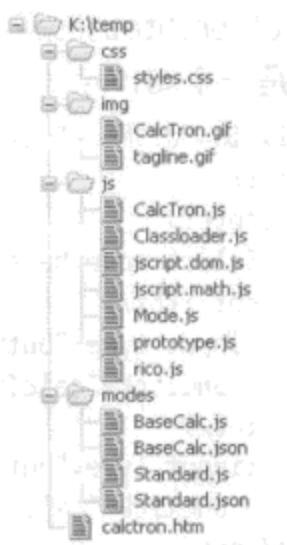


图4-4 CalcTron项目的目录结构

解决方案的组成部分如下。

- calctron.htm：在根目录中，是我们的起点。calctron.htm文件定义了应用程序的基本布局，包括所有需要的JavaScript，并开始执行应用程序。
- css：这个目录是用来放置styles.css文件的，它是CalcTron用来定义展现的样式信息。
- img：这个目录是图片存储的地方。在CalcTron中只用到了两个图片：标题图片和在它下面的标签行。
- js：这个目录包含一些JavaScript文件。正是这些文件组成了CalcTron。在这里还可以找到这个应用程序所使用的一些支持库。
- modes：这个目录为计算器的每一个模式存储一个JavaScript文件和一个JSON文件。

其实东西不对吧，不是吗？有了这样的概览，我们就可以直接进入细节，看看代码了。

4.4.1 编写calctron.htm

CalcTron开始先导入它的样式表，我们一会儿看样式表以及需要的所有JavaScript源文件。

```
<link rel="StyleSheet" href="css/styles.css" type="text/css">

<script src="js/prototype.js" type="text/javascript"></script>
<script src="js/rico.js" type="text/javascript"></script>
<script src="js/jscript.math.js" type="text/javascript"></script>
<script src="js/Mode.js" type="text/javascript"></script>
<script src="js/Classloader.js" type="text/javascript"></script>
<script src="js/CalcTron.js" type="text/javascript"></script>
```

首先引入Prototype，然后是Rico，Rico需要Prototype（我们还将直接使用Prototype中的一个函数）。下一步是一个在第3章中创建的包——jscript.math包。需要math包是因为代码需要这个包做一个随机的判断，决定模式转换的弹出窗口要从哪个边角飞入，因为math包包含了随机数生成函数。在那之后引入了Mode类、Classloader类以及CalcTron类。我们稍后看看它们。

在这些引入之后，跟着单独的一行JavaScript，事实上它是让所有代码运转起来的关键：

```
<script>
  var calcTron = new CalcTron();
</script>
```

这一行代码创建了一个CalcTron类的实例，并把它赋给一个变量calcTron。CalcTron类是应用程序的核心，但是我们先不用想这么远，calctron.htm里还有不少东西。

由于calctron.htm是用户载入的文件，在载入时发生了什么是很重要的：

```
<body onLoad="calcTron.init();">
```

CalcTron类的init()方法负责所有应用级别初始化，因此调用它来响应页面的onLoad事件。

calctron.htm的body部分是页面的基本架构，如前面图4-1、图4-2和图4-3所示。我们遇到的第一个元素是一个 ：

```
<div id="divMode" class="cssDivMode">
  <br/>
  <center>
    Select mode here
    <br/><br/>
    <input type="button" value="Standard"
      onClick="calcTron.setMode('Standard');">
    <br/>
    <input type="button" value="BaseCalc"
      onClick="calcTron.setMode('BaseCalc');">
  </center>
</div>
```

这个 是模式转换的弹出窗口。它非常直白。事实上，它唯一感兴趣的东西是onClick事件。正如前面章节讨论过的，你通常想避免这样的内联JavaScript。不过，如果只是一个函数调用的时候，我并不认为内联会有多大的破坏，我这儿就这么用。如名字所暗示的，calcTron对象的setMode()方法把模式设置为指定的模式。我们稍后会看看那些细节（在4.4.3节），所以等轮到它的时候再讲吧。

在这个 之后是另一个 ，有个ID是mainContainer，这里是计算器结构实际上被放置的地方。它里面是一个表格，每一个单元格都是计算器的一个按钮，前面是结果，后面是底部的信息区域。说实话，这部分是相当大的一段简单的HTML代码，所以这里不会全部罗列了。不过，让我们简单扼要的看看它。首先是结果部分：

```
<tr>
  <td nowrap colspan="10" align="right" valign="middle">
    <div style="height:16px;" id="divResults"></div>
  </td>
</tr>
```

这里并没有什么特别的。也就是说，注意在单元格里面的 ，ID是divResults，它是展现结果（或者用户当前输入的数字）的地方，方法是修改它的innerHTML属性。

接着是在最上面操作按钮（计算器有5个命令按钮在上部，5个在下面，中间有输入按钮）：

```
<tr>
  <td nowrap colspan="2" align="center" valign="middle"> ➥
    <input type="button" class="cssInputCommandButton" id="commandButton0" ➥
      onClick="calcTron.currentMode.commandButton0();"> ➥
  </td>
  <td nowrap colspan="2" align="center" valign="middle"> ➥
    <input type="button" class="cssInputCommandButton" id="commandButton1" ➥
    onClick="calcTron.currentMode.commandButton1();"> ➥
  </td>
```

```
    <td nowrap colspan="2" align="center" valign="middle"> ➥
      <input type="button" class="cssInputCommandButton" id="commandButton2" ➥
      onClick="calcTron.currentMode.commandButton2();"> ➥
    </td>
    <td nowrap colspan="2" align="center" valign="middle"> ➥
      <input type="button" class="cssInputCommandButton" id="commandButton3" ➥
      onClick="calcTron.currentMode.commandButton3();"> ➥
    </td>
    <td nowrap colspan="2" align="center" valign="middle"> ➥
      <input type="button" class="cssInputCommandButton" id="commandButton4" ➥
      onClick="calcTron.currentMode.commandButton4();"> ➥
    </td>
  </tr>
```

我在这里加了些换行，让它们能放在一个页面中。在你执行的真实代码中，每个表格单元都是单独一行的代码。糟糕的是，IE并不能像我们预期地那样完全忽略空格，所以如果代码真的像这里的格式一样，那在IE里显示得可能就不正确了。

重申一下，你在onClick事件处理函数中看到了单独的函数调用。注意，没有按钮有值（一个标签），因为当选择一个计算器模式的时候会动态添加按钮的标签，一会儿你就会看到。还要注意命令按钮的最下面一行看起来和这个基本一样，除了最后一个按钮，它总是模式转换按钮。看起来像这样：

```
<td nowrap colspan="2" align="center" valign="middle">
  <input type="button" class="cssInputCommandButton" style="display:block;"
    value="Mode" onClick="calcTron.changeModePopup();">
</td>
```

如你所见，这里赋了一个值，并且onClick处理函数是不同的。其他的就没什么特别了。在顶部和底部的命令按钮行之间，是5行输入按钮。让我们看看其中一行：

```
<tr>
  <td nowrap align="center" valign="middle">
    <input type="button" class="cssInputButton" id="button0_0"
      onClick="calcTron.currentMode.button0_0();">
  </td>
  <td nowrap align="center" valign="middle">
    <input type="button" class="cssInputButton" id="button0_1"
      onClick="calcTron.currentMode.button0_1();">
  </td>
  <td nowrap align="center" valign="middle">
    <input type="button" class="cssInputButton" id="button0_2"
      onClick="calcTron.currentMode.button0_2();">
  </td>
  <td nowrap align="center" valign="middle">
    <input type="button" class="cssInputButton" id="button0_3"
      onClick="calcTron.currentMode.button0_3();">
  </td>
  <td nowrap align="center" valign="middle">
    <input type="button" class="cssInputButton" id="button0_4"
      onClick="calcTron.currentMode.button0_4();">
  </td>
```

```
<td nowrap align="center" valign="middle">
  <input type="button" class="cssInputButton" id="button0_5"
    onClick="calcTron.currentMode.button0_5();">
</td>
<td nowrap align="center" valign="middle">
  <input type="button" class="cssInputButton" id="button0_6"
    onClick="calcTron.currentMode.button0_6();">
</td>
<td nowrap align="center" valign="middle">
  <input type="button" class="cssInputButton" id="button0_7"
    onClick="calcTron.currentMode.button0_7();">
</td>
<td nowrap align="center" valign="middle">
  <input type="button" class="cssInputButton" id="button0_8"
    onClick="calcTron.currentMode.button0_8();">
</td>
<td nowrap align="center" valign="middle">
  <input type="button" class="cssInputButton" id="button0_9"
    onClick="calcTron.currentMode.button0_9();">
</td>
</tr>
```

这些和命令按钮非常相似，只是onClick处理函数调用的那个函数不同，还有就是赋给它们的样式类也不一样。其余4行是一样的，除了每个按钮的id不同之外。

跟在按钮之后的是信息区域：

```
<td nowrap colspan="10" align="center" valign="middle">
  <div style="height:16px;" id="divInfo"></div>
</td>
```

这与结果区域非常相似，只是样式不同。这样，我们就基本上看完calctron.htm了。我说过，不是造火箭，挺简单的。

4.4.2　编写styles.css

很自然，styles.css是CalcTron使用的主要样式表。你已经在看calctron.htm的时候看到一些它包含的样式，所以让我们看看有没有其他有意思的事情。整个样式表如代码清单4-1所示。

代码清单4-1　CalcTron的styles.css文件

```
/* Style applied to all elements */
* {
  font-family      : arial;
  font-size        : 10pt;
  font-weight      : bold;
}
/* Style applied to the outer calculator container */
.cssCalculatorOuter {
  position         : absolute;
  background-color : #c6c3de;
}
```

```
/* Style applied to the table cell where command buttons are placed */
.cssSpanCB {
  width            : 110px;
}

/* Style applied to the table cell where input buttons are placed */
.cssSpanB {
  width            : 60px;
}

/* Style applied to input buttons */
.cssInputButton {
  width            : 50px;
  display          : none;
}

/* Style applied to command buttons */
.cssInputCommandButton {
  width            : 100px;
  display          : none;
}

/* Style applied to the mode switch popup DIV */
.cssDivMode {
  display          : none;
  z-index          : 100;
  position         : absolute;
  border           : 2px solid #000000;
  background-color : #efefef;
}
```

让我们看看每个选择器。

- 第一个选择器很有意思。它用于页面中的所有元素，是一个通吃的样式。它的一个好处是，它包含表格、单元格以及其他的元素，而通常这些是不包含的，所以它真的包含了所有东西。这个样式把字体设置为Arial、10pt，并使全部字体用粗体。
- 下一个样式是用于外层 的。我们把它设置成紫蓝色，并绝对定位，这是必要的，因为我们需要将它居中。
- 下一个，是分别用于包含命令按钮和输入按钮的单元格的样式。它们利用一些空白填充，确保每个单元格的大小完美地匹配一个按钮。
- 然后是用于命令和输入按钮的两个样式。这就确保所有的按钮有一个恒定的大小。它们还确保所有的按钮开始时是隐藏的，这样，在应用程序载入和初始化模式的时候，就避免了不必要和愚笨的闪烁。
- 最后是一个用于模式转换弹出框的样式。我们通过设置它的z-index来确保它浮动在计算器的上面，并给它一个实心的颜色和一个边框。和外部 样式一样，它是绝对定位的，这是我们可以使它从一个角飞入并居中的唯一方法。

4.4.3　编写CalcTron.js

就像前面提到的，CalcTron是驱动这个应用程序的主类——应用程序的核心。它包括描述计算器全部状态的字段、初始化代码、处理模式转换的代码。它所做的事，即使是像它这样的应用程序的核心，真的一点都不复杂，如图4-5所示的UML图。

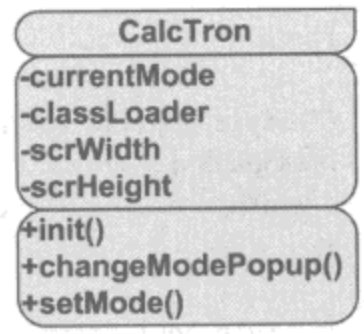


图4-5　CalcTron类的UML图

它拥有的全部家当，就只是4个字段和3个方法。让我们从那4个字段开始看起。

- currentMode：这个字段存储CalcTron的当前模式的id，实际上是当前模式的名字。CalcTron有两个模式：标准模式和BaseCalc模式，并且，它们是在这个字段中可以找到的仅有的两个值（没有算入它的初始空值）。
- classLoader：这个字段是指向Classloader类的一个实例的引用，我们会在下一节中描述这个类。简单地说，这个类就是用来负责载入特定模式所需要的函数。它还检验了类是否符合一个特定的接口需求，通常这意味着该类是否是合法的CalcTron Mode类。
- scrWidth和scrHeight：这些字段存储启动时浏览器窗口的宽和长。这个信息用于将计算器本身居中，并且还用于模式改变弹出窗口所需的各种计算。

继续向下看，是CalcTron类的方法，我们首先遇到了init()。这个方法被calctron.htm文件的onLoad调用，它的任务就是初始化许多东西，让CalcTron做好与用户交互的准备。init()函数如下：

```
this.init = function() {

  // Figure out how wide the browser content area is.
  if (window.innerWidth) {
    this.scrWidth = window.innerWidth;
  } else {
    this.scrWidth = document.body.clientWidth;
  }

  // Figure out how high the browser content area is.
  if (window.innerHeight) {
    this.scrHeight = window.innerHeight;
  } else {
    this.scrHeight = document.body.clientHeight;
  }

  // Round the corners of the main content div.
  new Rico.Effect.Round(null, "cssCalculatorOuter");

  // Set initial mode to standard.
  this.setMode("Standard");

} // End init().
```

首先，函数判断浏览器窗口的内容区域的宽度。一些浏览器通过window对象的innerWidth属性来

展现这个信息，而其他的浏览器是通过document.body对象的clientWidth属性，所以需要一些分支代码。后面跟着实际上一样的决定浏览器内容区域高度的代码。

在那之后，是我们在这个应用程序中Rico的第一次曝光。如前面看到Rico时所说的，它的很好的功能之一就是，提供了任何元素的圆角化能力，所以我们可以在计算器周围有一些漂亮的、柔软的圆角来代替普通的坚硬的方形拐角。代码实例化一个新的Rico.Effect.Round对象，并传给它样式表的名字。Round()的第一个参数实际是一个有待圆角化的元素，第二个参数是一个类的名字。我决定使用类名是因为一开始我并不确定是否会有其他需要圆角化的形状，而且使用类的意思是我可以圆角化任何使用相同的样式类的对象。

提示 你可以传给Round()第3个变量，进一步定义圆角化。比如，如果第3个参数传递了{ corners : 'tl br'}，那么就是在说，只有左上角和右下角要被圆角化。参照Rico文档和实例，了解这些选项更多更深入的解释。

圆角化完成之后，还有一个重要的事情就是设置CalcTron的初始模式。这是通过下面的语句来完成的。

```
this.setMode("Standard");
```

标准模式是我们的开始模式，就如同Windows的内置计算器（也是CalcTron的原模型，至少标准模式是仿照它的）一样。当完成这个之后，CalcTron就准备好给用户提供服务了。

CalcTron的下一个方法是changeModePopup()方法。当点击Mode按钮时调用它。虽然它比init()稍微长一点，但是它仍然是非常简单的家伙：

```
this.changeModePopup = function() {

  // This is the width and height of the div as it should ultimately appear.
  var divWidth = 150;
  var divHeight = 150;

  // Get reference to mode change div and reset it to begin animation. It's
  // going to randomly come flying from one of the corners of the screen,
  // so first choose which corner, then set the top and left attributes
  // accordingly.
  var modeDiv = $("divMode");
  modeDiv.style.width = "0px";
  modeDiv.style.height = "0px";

  // What corner does it fly from?
  var whatCorner = jscript.math.genRandomNumber(1, 4)

  // Set the starting coordinates accordingly.
  switch (whatCorner) {
    case 1:
      modeDiv.style.left = "0px";
      modeDiv.style.top = "0px";
    break;
```

```
    case 2:
      modeDiv.style.left = this.scrWidth - divWidth;
      modeDiv.style.top = "0px";
    break;
    case 3:
      modeDiv.style.left = "0px";
      modeDiv.style.top = this.scrHeight - divHeight;
    break;
    case 4:
      modeDiv.style.left = this.scrWidth - divWidth;
      modeDiv.style.top = this.scrHeight - divHeight;
    break;
  }
  // Calculate the final left and top position for the div so it's centered
  // in the browser content area.
  var left = (this.scrWidth - divWidth) / 2;
  var top = (this.scrHeight - divHeight) / 2;

  // Show the div so the animation can begin.  Since its width and height are
  // zero, it won't actually be visible just yet.
  $("divMode").style.display = "block";

  // Ask Rico to do the animation for us.
  new Rico.Effect.SizeAndPosition("divMode", left, top, divWidth, divHeight,
    400, 25, null
    );

  } // End changeMode().
```

首先，我们有两个变量，divWidth和divHeight，它们定义了模式转换弹出窗口的宽度和高度。这些是后面的计算所需要的变量。

我们通过获取一个指向divMode的引用来开始实际的工作。这是使用$()函数来完成的，它实际上是Prototype的一部分，并不是Rico的。$()是简化的术语，是到处都是的document.getElementById()的速写版，不过它还添加了同时获得指向多个对象的引用的能力。我们拿到指向 的引用之后，就把它的宽和高设置为零。那个模式转换弹出框不仅仅是从浏览器4个角之一的地方飞入，它还要在飞入的同时扩展成合适的尺寸。所以，开始的时候它的尺寸应该尽可能小，这样才能产生正确的效果。

然后，我们看调用了第3章里介绍过的jscript.math.genRandomNumber()函数。它生成一个1到4之间的随机数，包括1和4。这个随机数决定弹出框从哪个边角飞入。

在那之后，我们看看这个值上面的一个switch。根据选取的边角，设置divMode的left和top属性，使其从适当的角开始。

跟着的两行代码用于计算弹出框的最终位置。这些语句使用了在init()中计算的scrWidth和scrHeight，以及在这个函数开头部分设置的divWidth和divHeight。分别对于宽和高，得出相应两个值的差值，然后除以2，这样得出将弹出框居中在内容区域所需要的适当的坐标值。

最后，看看Rico的第二个用法——Rico.Effect.SizeAndPosition对象。我们给这个对象传入待操作的 的名字（divMode）、它应止于的最终X和Y（分别是left和top）坐标、结束时的宽和高、整

个过程会持续多少毫秒（400）以及会有多少步（25）。这意味着，Rico将把divMode从当前位置移动到指定的left和top变量的位置上，同时，将把它从当前的大小延展到指定的divWidth和divHeight的大小。这将在400 ms经过25步完成（也就是说，每一步将花费16 ms）。我们可以通过一个函数调用就得到所有的动作，这不是很酷吗？

CalcTron的最后一个方法是setMode()，当点击模式转换弹出框上的按钮的时候调用它。这真的是一个有趣的小函数，因为在模式转换的时候，它并不是只被调用一次，而是被调用两次，并且它每次会做一些不同的事。让我们先看看这个方法：

```
this.setMode = function(inVal) {

  // First time through: should have been passed a string naming the mode
  // to switch to. We simply pass it to the classloader to load it for us.
  if (typeof inVal == "string") {

    $("divMode").style.display = "none";
    this.classloader.load(inVal);

  } else {

    // Second time through, inVal is an instance of a class descending from
    // Mode. In that case, we ask the classloader to verify it for us,
    // and assumimg it's valid, we store a reference to it and ask it to
    // initialize itself.
    if (this.classloader.verify(inVal, new Mode())) {
      this.currentMode = inVal;
      this.currentMode.init();
    } else {
      alert("Not a valid mode class");
    }

  }

} // End setMode().
```

好，那么它是相当小的，但还是很有趣。当点击模式选择按钮时，会第一次调用它，传入了将要切换到的模式的名字。所以，第一个检查是看看传入的参数是否是一个字符串。如果是，那么隐藏模式转换弹出框，并且用传给setMode()的值调用Classloader实例。

Classloader载入类，当它完成时，就再一次调用setMode()（一会儿你就会看到具体是怎么调用的了）。然而，在这时，传入的不是字符串，而是继承于Mode的类。这里就是else子句执行的地方。这里，我们让Classloader来验证此类，如果没有通过，就只弹出一个警告，因为我们没什么可以做的。如果验证通过，我们设置CalcTron的currentMode字段，指向这个传入的类，然后在其上调用init()方法。

那么，这些就是CalcTron.js。有些地方可能对你来说还不是非常清楚，所以让我们现在开始把那些东西清理一下。我们从我已经提到过很多次的Classloader类开始。

4.4.4 编写Classloader.htm

我曾说过，CalcTron是设计成可扩展的，也就是说，你可以几乎毫不费力地向其中添加模式。每

一个模式都以扩展的Mode类的方式实现。我们在后面会看看实现这些类的方法，在这之前，让我们先说说如何载入那些类。

Java有一个相当复杂的机制，叫做classloader。它的功能，我确信你肯定想知道，是用来……卖个关子……是载入类。不过，它做的比这个多多了。它还负责验证类、检查它们是否违反安全、确认它们不冲突，等等。

JavaScript并不提供任何很奇特的东西，但是它从未阻止我们自己做这些。一个JavaScript classloader应该提供什么功能呢？好的，安全性并不是问题，因为JavaScript一开始就是在一个相对安全的沙盒中执行的。所以我们可以跳过它。我们需要检查冲突的类吗？可能，但是除了确保我们从类所在的服务器上装载后，得到的不是一个错误之外，也没什么好做的。

一个我们可以执行的操作是，验证那个类是否符合一个特定的公共接口。每一个模式实现类都必须实现一些函数，这样CalcTron才能使用它。我们可以验证一个已载入的类的确是按要求做的。

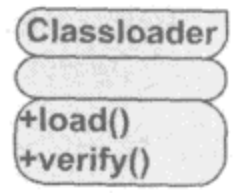


图4-6　Classloader类的UML图

但是我们可能首先应该讨论如何确切地载入一个JavaScript类，因为对于JavaScript来说，即使这个简单的概念也不是原生的。不过，在做那些之前，让我们先看看Classloader类的UML图，如图4-6所示。

嘿嘿，我们可能还要看一下代码。没什么复杂的东西。完整的Classloader代码在代码清单4-2中列出。

代码清单4-2　Classloader.js文件

```
function Classloader() {


 /**
  * Load a named class.
  *
  * @param inClassName The name of the class to load.  We assume it's a
  *                    calculator mode class, so it's always in the modes
  *                    subdirectory.
  */
 this.load = function(inClassName) {

   // Dynamically create a new script tag, point it at the mode source file,
   // and append it to the document's head section, thereby loading and
   // parsing it automatically.
   var scriptTag = document.createElement("script");
   scriptTag.src = "modes/" + inClassName + ".js";
   var headTag = document.getElementsByTagName("head").item(0);
   headTag.appendChild(scriptTag);
 } // End load().


 /**
  * This function verifies that a given class matches another. In other
  * words, it ensures that all the functions of inBaseClass are found in
```

```
     * inClass, which means they have the same public interface. It also
     * checks that the id field is present, which is required by code outside
     * a mode class (note that all other non-function fields are ignored, since
     * they do not contribute to the public interface).
     *
     * @param  inClass     The class to verify.
     * @param  inBaseClass The class to verify against.
     * @return             True if inClass is "valid", false if not.
     */
    this.verify = function(inClass, inBaseClass) {

      var isValid = true;
      for (i in inBaseClass) {
        if (i != "resultsCurrent" && i != "resultsPrevious" &&
          i != "resultsCurrentNegated" && i != "resultsPreviousNegated" &&
          !inClass[i]) {
          isValid = false;
        }
      }
      return isValid;

    } // End verify().


} // End Classloader class.
```

说完上面的前言之后，让我们看看它都是怎么工作的。

第一步是一个对load()方法的调用。这个方法接收的参数是待载入的类的名字。这个名字必须与包含它的JavaScript源文件的名字完全匹配，并且是区分大小写的。因为这个Classloader是特定的载入Mode类的，它假设源文件是可以在mode子目录下找到的。这就是load()方法：

```
this.load = function(inClassName) {

  // Dynamically create a new script tag, point it at the mode source file,
  // and append it to the document's head section, thereby loading and
  // parsing it automatically.
  var scriptTag = document.createElement("script");
  scriptTag.src = "modes/" + inClassName + ".js";
  var headTag = document.getElementsByTagName("head").item(0);
  headTag.appendChild(scriptTag);

} // End load().
```

通过动态脚本装载远端的内容

一级警报！大多数Ajax技术都有一个问题，就是它们不能跨域工作。目前，所有的浏览器都有一个安全限制，它让你只能向原始页面所在的域提交Ajax请求。这就使得很多事情比必要操作更加麻烦。有一些解决方案可以绕开这个限制。其中最有用的一个就是动态<script>标签的小技巧。

原则上，如果你创建一个新的<script>标签，并把它插入到DOM中，浏览器就会延展开、载入源文件并运行它，就如同在载入时页面本身的<script>标签一样。因为一个<script>标签是没有域的限制的，那么你就可以做跨域调用了。现在，如果<script>标签的源内容遵循一定的特殊规则，那实际上你就可以用这个方法来做Ajax了。什么是特殊规则呢？简单地说就是请求必须包含一个对某个回调函数的JavaScript调用。

这样，当那个<script>标签被插入时，浏览器载入它指定的源文件。这个源文件包含了一个对页面中已存在的某个JavaScript函数的调用。浏览器执行了这个源文件，这个源文件执行了对那个函数的调用。这类似于Ajax中的回调函数，只是在这种情况下它只被调用一次，而对应在Ajax调用中是多次的。换句话说，<script>标签的源文件并不必是脚本文件。好的，不是普通意义上的脚本，它还是JavaScript。

我说的这些东西，可以由一个简单的例子非常形象地说明。假设我们想要从远程服务器取回一个URL的列表，并在用户点击一个按钮的时候，把这些展现在一个alert()框中。我不知道我们为什么要这么做，但这是他们为何发明这个例子的原因。总之，假设我们想做这个，并假设在Omnytex的Web服务器上有个文件js_book_ch4_url_list.txt（这个文件的确存在，如果你尝试这里的例子，那么都会运转的，但是如果不知什么原因失败了，那只要创建一个同名、包含所示内容的文件，把它放在某地的某个Web服务器上，并相应的改变目标URL）。那么，当我们访问http://www.omnytex.com/js_book_ch4_url_list.txt时，就得到如下的回答：

```
showURLs("www.microsoft.com", "www.omnytex. com", "www.apress.com");
```

下面是我们想要做的事情的代码：

```
<html>
<head>
<script>
  function testIt() {
    var scriptTag = document.createElement("script");
    scriptTag.src = "http://www.omnytex.com/js_book_ch4_url_list.txt";
    var headTag = document.getElementsByTagName("head").item(0);
    headTag.appendChild(scriptTag);
  }
  function showURLs() {
    var s = "";
    for (var i = 0; i < showURLs.arguments.length; i++) {
      s += showURLs.arguments[i] + "\n";
    }
    alert(s);
  }
</script>
</head>
<body>
  <input type="button" value="Click here to display URLs" onClick="testIt();">
</body>
</html>
```

让我们看看这段代码。用户点击按钮会调用testIt()。在testIt()中，我们创建了一个新的

<script>标签，把它的src属性设置为指向远程的URL文件。新的<script>标签附着在文档的<head>标签上。然后浏览器装载并执行这个远程的文件。执行导致对包含在远程的文件里的showURLs()的调用。showURLs()执行，它遍历传递给它的所有参数，可以是从无到任意多个。对每个参数，它都将其附着在一个字符串上，并且在末尾换行。最后它显示一个alert()，显示所有字符串。

让我们看看Classloader.js文件中的load()。是不是看上去很像？它们实际上是完全一样的代码，只是src属性上的URL不同。

用load()方法装载了类之后，就会调用verify()方法。这个方法接受装载的被验证类和另外一个用于验证的参照类。verify()遍历参照类的所有属性，对其中的每个属性都检查被验证类里面是否存在对应的属性。如果属性不存在，verify()就把isValid标志设置为false。这个标志是该方法的返回值。如果找到所有的属性，就会返回true。请注意这个方法只检查函数，除了id之外，它忽略所有数据字段，保留id是因为它是Mode子类的公共接口的要求。所有其他的数据字段都与公共接口无关，所以被忽略。

俗话说，一图抵千言，出于这个考虑，图4-7显示了装载CalTron里面的类的所有的步骤。

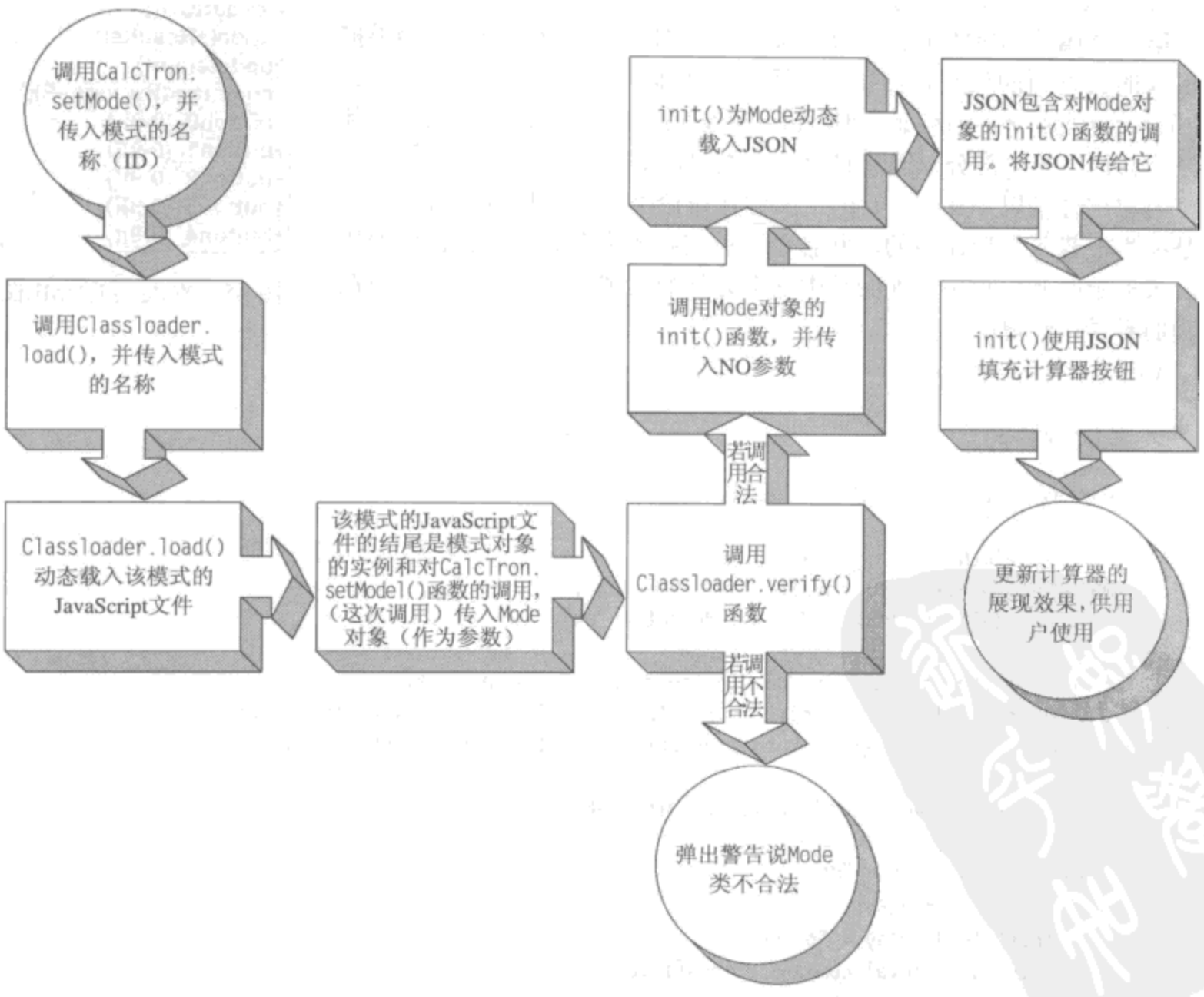


图4-7 与装载一个类相关的流程图

这个流程在BaseCalc模式下有一些区别。在那个模式里，在步骤“JSON包含对Mode对象的init()的调用，给它传递JSON”之前都是一样的。在BaseCalc模式下，这一步仍然会执行，区别是在BaseCalc模式下，init()方法被覆盖了，因此它会先做一些处理，然后调用超类（你可能不熟悉这个术语，其实就是父类），也就是Mode类的init()。之后的流程图又是一样的了。要是这些看上去不那么容易理解，别着急。我们还没看Mode类和两个计算器模式的类呢，只是先说了几句而已。那现在就让我们进入这个主题，把整个环境设置到正确的环境里。

4.4.5　编写Mode.js

CalcTron支持的每一个模式，包括内置的标准模式和BaseCalc模式，都是用一个扩展了Mode类的另外一个类来实现的。Mode类本身包括一些所有计算器模式都需要的公用字段，并且Mode类还包含一些预期任何计算器模式都要实现的存根方法。这些方法一起组成了一个Mode实现类需要公开出来，实现与CalcTron类交互的公共接口。在图4-8的UML图中你可以看到Mode类的整体结构。

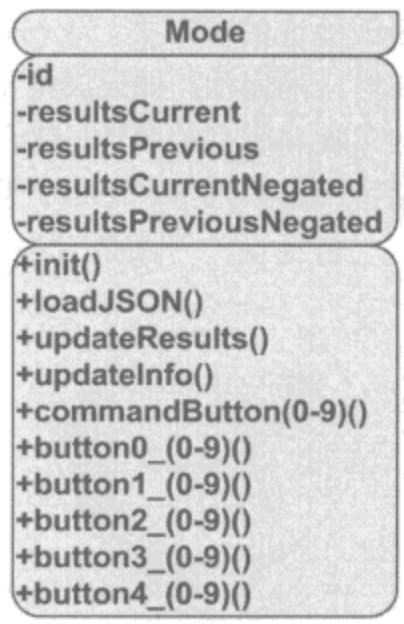


图4-8　Mode类的UML图

第一个属性id事实上也是公共接口的一部分，因为那些自Mode类扩展出来的类里面的代码也需要它。不过，其他的——resultsCurrent、resultsPrevious、resultsCurrentNegated以及resultsPreviousNegated——都不是公共接口的一部分，因为只有这个类自己需要它们。

方法部分是以init()开始的，就像在讨论类载入过程时你看到的，它是在类被载入之后调用的，并且验证被装载的类并且初始化计算器模式。大多数时候，使用在Mode类中这个方法的实现就足够了，那么现在让我们看看代码吧：

```
this.init = function(inVal) {

  if (inVal) {

    var mainDiv = $("mainContainer");

    // Size width and height as specified.
    mainDiv.style.width = inVal.mainWidth + "px";
    mainDiv.style.height = inVal.mainHeight + "px";

    // Center the main content div in the browser content area.
    mainDiv.style.left = (calcTron.scrWidth - parseInt(inVal.mainWidth)) / 2;
    mainDiv.style.top = (calcTron.scrHeight - parseInt(inVal.mainHeight)) / 2;

    // Command buttons (10 of them, numbered 0-8).
    for (var i = 0; i < 9; i++) {
      var btn = $("commandButton" + i);
      if (inVal.commandButtons[i].enabled == "true") {
        btn.style.display = "block";
        btn.value = inVal.commandButtons[i].caption;
      } else {
        btn.style.display = "none";
```

```
    }
  }

  // Buttons (50 of them, 10 in a row numbered 0-9 in 5 rows numbered 0-4).
  for (var y = 0; y < 5; y++) {
    for (var x = 0; x < 10; x++) {
      btn = $("button" + y + "_" + x);
      if (inVal.buttons[y][x].enabled == "true") {
        btn.style.display = "block";
        btn.value = inVal.buttons[y][x].caption;
      } else {
        btn.style.display = "none";
      }
    }
  }

} else {

  this.loadJSON(this.id);

}

// Show current mode in info box and clear results div.
this.updateResults("");
this.updateInfo(this.id + " Mode");

} // End init().
```

你可能会回忆起来，在模式更换的时候，这个方法实际上会被调用两次。第一次调用，什么都不传给它。当再次调用的时候，就直接转到else分支，结果就是调用loadJSON()，并把之前填充的id字段的值传给它。loadJSON()从概念上和内容上都非常类似Classloader类的load()方法，如你在这里所见：

```
this.loadJSON = function(inID) {

  var scriptTag = document.createElement("script");
  scriptTag.src = "modes/" + this.id + ".json";
  var headTag = document.getElementsByTagName("head").item(0);
  headTag.appendChild(scriptTag);

} // End loadJSON().
```

它创建了一个动态的<script>标签，src属性指向在modes子目录下的JSON文件，这个文件的名称与那个模式的名称（id字段的值）匹配，这会导致载入和执行JSON。执行完毕之后，会再次调用init()，这次还带了执行过的JSON本身作为一个参数。当这些发生时，就执行if语句段。

首先，if段代码把计算器外部 的宽度和高度，分别设置为JSON中的属性mainWidth和mainHeight指定的值。这么做是因为动态计算这些值比想象得要麻烦多了。取而代之的是，我决定把责任交给模式的开发者。对他来说只是多一两次试验和出错的机会，但算不上创建一个新的模式那么浩大的战争（我应该知道——因为我做了两次）。

在那之后，计算器使用与你在前面看到的那个让弹出框居中相同的逻辑置中。

然后，我们开始一些稍微有意思一点的东西：生成命令按钮。

```
// Command buttons (10 of them, numbered 0-8).
for (var i = 0; i < 9; i++) {
  var btn = $("commandButton" + i);
  if (inVal.commandButtons[i].enabled == "true") {
    btn.style.display = "block";
    btn.value = inVal.commandButtons[i].caption;
  } else {
    btn.style.display = "none";
  }
}
```

在一个模式的布局中，有10个可用的命令按钮：5个在上部、5个在下部。底部的最后一个按钮总是模式转换按钮，所以开发者有9个可用的按钮，因此，for循环的次数是9。对于每一次遍历，首先取得按钮的引用，如果你还记得的话，它实际已经存在了。然后检查在JSON中对于适当的按钮的enabled值。如果它是false，那么这个按钮不可用，于是我们把display样式的属性设置为none来隐藏它。如果它是启用的（true），那么我们把display设置为block来显示它，并且还要修改标签为JSON中指定的内容。事件处理函数已经连接上，正如在calctron.htm中看到的，所以这就是我们这里需要做的所有事情。

对于输入按钮也用完全相同的方式处理，不过，开发者总共有50个可用的不同的按钮，每行10个，总共5行。所以，我们需要2个for循环：一个是对于行（y）的，一个是对于在一行中每个按钮的（x）。

loadJSON()之后的是updateResults()方法。它只是在计算器顶部的结果框中显示resultsCurrent字段的值，如果resultsCurrentNegated字段的值为true的话，它还负责在resultsCurrent字段的值之前添加一个负号。接着是updateInfo()，它只是在计算器底部的信息栏中展示它的参数。

在那两个函数之后，是一个非常大的空函数集——它们分别对应计算器的每一个命令按钮和输入按钮。这么做是因为，即使模式的实现类不需要某个按钮，它还是会公开一个它需要的事件处理函数，只是那个函数什么都不做罢了。这只是为了保持公共接口是一个已知的常量，这样一个按钮的onClick处理函数就不会有机会调用一个不存在的方法。（没错，一个开发者可以在实现类中把某个方法设置为空，但是他还是需要明确这么做，而且我们无法防止一个开发者有意的破坏。）

正如我提到过的，Mode类是所有的计算器模式实现类的超类，所以或许你可以猜到，现在是时候来看看那些实现类了。我们还会看看定义了每一个模式的JSON，因为它们是手拉着手一起工作的。

4.4.6　编写Standard.json和Standard.js

既然我们已经看过Mode基类，那就让我们看看从它扩展而来的类吧：定义CalcTron模式的实现类。让我们从标准模式开始。

两个元素定义一个CalcTron模式：从Mode类扩展出来的实现类和一个JSON文件，后者描述模式。比如，标准模式的JSON在代码清单4-3中列出。

代码清单4-3 描述标准CalcTron模式的JSON

```
calcTron.currentMode.init(

  {

    "mainWidth" : "340", "mainHeight" : "248",

    "commandButtons" : [
      { "enabled"   : "false", "caption" : ""          },
      { "enabled"   : "false", "caption" : ""          },
      { "enabled"   : "false", "caption" : ""          },
      { "enabled"   : "true",  "caption" : "Backspace" },
      { "enabled"   : "true",  "caption" : "Clear"     },
      { "enabled"   : "false", "caption" : ""          },
      { "enabled"   : "false", "caption" : ""          },
      { "enabled"   : "false", "caption" : ""          },
      { "enabled"   : "false", "caption" : ""          },
    ],

    "buttons" : [
      [
        { "enabled" : "false", "caption" : ""          },
        { "enabled" : "false", "caption" : ""          },
        { "enabled" : "false", "caption" : ""          },
        { "enabled" : "false", "caption" : ""          },
        { "enabled" : "false", "caption" : ""          },
        { "enabled" : "true",  "caption" : "7"         },
        { "enabled" : "true",  "caption" : "8"         },
        { "enabled" : "true",  "caption" : "9"         },
        { "enabled" : "true",  "caption" : "/"         },
        { "enabled" : "true",  "caption" : "sqrt"      }
      ],
      [
        { "enabled" : "false",  "caption" : ""         },
        { "enabled" : "false",  "caption" : ""         },
        { "enabled" : "false",  "caption" : ""         },
        { "enabled" : "false",  "caption" : ""         },
        { "enabled" : "false",  "caption" : ""         },
        { "enabled" : "true",   "caption" : "4"        },
        { "enabled" : "true",   "caption" : "5"        },
        { "enabled" : "true",   "caption" : "6"        },
        { "enabled" : "true",   "caption" : "*"        },
        { "enabled" : "true",   "caption" : "%"        }
      ],
      [
        { "enabled" : "false", "caption" : ""          },
        { "enabled" : "false", "caption" : ""          },
        { "enabled" : "false", "caption" : ""          },
        { "enabled" : "false", "caption" : ""          },
```

```
    { "enabled" : "false", "caption" : ""          },
    { "enabled" : "true",  "caption" : "1"         },
    { "enabled" : "true",  "caption" : "2"         },
    { "enabled" : "true",  "caption" : "3"         },
    { "enabled" : "true",  "caption" : "-"         },
    { "enabled" : "true",  "caption" : "1/x"       }
  ],
  [
    { "enabled" : "false", "caption" : ""          },
    { "enabled" : "false", "caption" : ""          },
    { "enabled" : "false", "caption" : ""          },
    { "enabled" : "false", "caption" : ""          },
    { "enabled" : "false", "caption" : ""          },
    { "enabled" : "true",  "caption" : "0"         },
    { "enabled" : "true",  "caption" : "+/-"       },
    { "enabled" : "true",  "caption" : "."         },
    { "enabled" : "true",  "caption" : "+"         },
    { "enabled" : "true",  "caption" : "="         }
  ],
  [
    { "enabled" : "false", "caption" : ""          },
    { "enabled" : "false", "caption" : ""          },
    { "enabled" : "false", "caption" : ""          },
    { "enabled" : "false", "caption" : ""          },
    { "enabled" : "false", "caption" : ""          },
    { "enabled" : "false", "caption" : ""          },
    { "enabled" : "false", "caption" : ""          },
    { "enabled" : "false", "caption" : ""          },
    { "enabled" : "false", "caption" : ""          },
    { "enabled" : "false", "caption" : ""          }
  ]
 ]

}

);
```

如果你之前曾经见过JSON，你可能会认为，“那看起来不是很对啊。”没错，事实上，这是封装在一个函数调用中的JSON。JSON本身是函数的一个参数，或者更准确地说，是一个从它（JSON）构造而来的对象。回忆一下，当Mode类的init()方法第一次被Classloader类调用的时候，什么都没传递给它，但是第二次的时候，给它传了这个JSON。所以到底是什么东西在第二次调用了init()呢？答案就是JSON。好的，基本上是——JSON是使用之前讨论过的动态<script>标签技术动态载入的。

和页面中的任何<script>标签一样，浏览器计算从服务器返回的内容。在这么做的时候，它执行了所有不在函数中的JavaScript，calcTron.currentMode.init()也是用同样方法调用的。在模式载入的过程中，calcTron实例的currentMode属性指向在JSON之前被载入的Standard类。所以当载入JSON并计算它的时候，就发生了对init()的调用，并且把JSON传给了它。然后如前面所说的init()就做它自己的事情去了。

提示 动态<script>标签技术越来越普遍了。从某种意义上来说，Yahoo是人们开始意识到的第一个。它基本上就是一种定义一个回调函数的方法，当数据返回时它会被执行。数据并不一定是JSON，虽然它是普遍流行的返回类型。它可以是一个实际的对象（想象一下定义一个类并创建一个实例来替代JSON，然后字段都填充好，然后调用init()，并给它传递一个那个对象的引用）。这逐渐变成了一种流行的方法，因为它允许了跨域的Ajax调用。只要客户端知道将要调用的回调函数，并且服务器支持这个协议，那就没有什么同域名的限制了，这是ajax的普遍问题。只要想象一下，你和你的朋友都可以创建CalcTron模式。把它们放在你自己的服务器上，并且你的CalcTron的本地副本可以从世界上任何地方访问它们（只要你通过更新那个模式转换弹出框来把这些新模式通知给CalcTron），是多么酷啊。

JSON本身的意思是相对简单的。前面提到过的mainWidth和mainHeight定义了计算器 的宽度和高度。然后出现了一组元素，叫做commandButtons。这组中的每一个元素都定义了一个独立的命令按钮。每一个按钮，不管是命令按钮还是输入按钮，都有两个属性：enabled和caption。enabled属性定义了它是可见的（true）还是不可见的（false）。caption属性是出现在按钮上的文字。

在输入按钮之后，是一组已命名的按钮。在这组中，还有5个小组，每个小组对应着一行按钮。每一个小组都包含有10个元素，分别对应一个按钮。

事件处理函数

在calctron.htm中，每一个按钮都定义了一个onClick事件处理器。Mode类包含存根函数，它们与按钮对应。这些函数和你看到的JSON之间是一种隐含的合作。比如，我们看看命令按钮组：

```
"commandButtons" : [
  { "enabled"   : "false", "caption" : ""          },
  { "enabled"   : "false", "caption" : ""          },
  { "enabled"   : "false", "caption" : ""          },
  { "enabled"   : "true",  "caption" : "Backspace" },
  { "enabled"   : "true",  "caption" : "Clear"     },
  { "enabled"   : "false", "caption" : ""          },
  { "enabled"   : "false", "caption" : ""          },
  { "enabled"   : "false", "caption" : ""          },
  { "enabled"   : "false", "caption" : ""          },
]
```

我们如何知道当点击Backspace按钮时，应该调用那个函数呢？它并没有在这里命名，那么我们怎么知道呢？简单地说，用的是位置。定义在JSON中的第一个按钮在被点击的时候将调用commandButton0()然后第二个是commandButton1()，第三个是commandButton2()，到了最后Backspace会调用commandButton3()。对于输入按钮，以第一行按钮为例：

```
[
  { "enabled" : "false", "caption" : ""          },
  { "enabled" : "false", "caption" : ""          },
  { "enabled" : "false", "caption" : ""          },
  { "enabled" : "false", "caption" : ""          },
```

```
    { "enabled" : "false", "caption" : ""          },
    { "enabled" : "true",  "caption" : "7"         },
    { "enabled" : "true",  "caption" : "8"         },
    { "enabled" : "true",  "caption" : "9"         },
    { "enabled" : "true",  "caption" : "/"         },
    { "enabled" : "true",  "caption" : "sqrt"      }
  ]
```

这里的工作方式完全相同，只是我们需要知道这里按钮的行数和列数。对于输入按钮的第一行，行数是0。所以，这里第一个按钮会调用函数button0_0()。跳到屏幕中的数字7的按钮，它会调用函数button0_5()。

看完JSON文件之后，我们可以转移到Standard.js文件，标准模式实现类本身。Standard类的UML图在图4-9中展示。

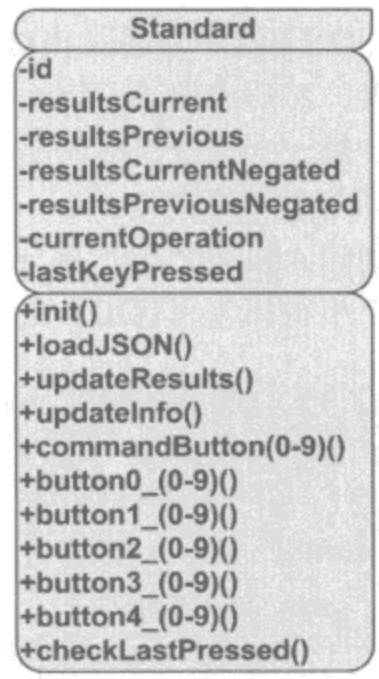


图4-9　Standard类的UML图表

在这个图中，我用缩写来写commandButton和Button的方法，因为它们太多了。缩写的意思就是，有commandButton0()、commandButton1()、commandButton2()等，直到commandButton9()。对于按钮的方法，有button0_0()、button0_1()、button0_2()，等等，直到button0_9()，然后重复button1_x()、button2_x()、button3_x()以及button4_x()，这里x是0～9。

就像在源代码（你应该已经从Apress网站下载了）中看到的，你看到的第一件事情就是，把id字段设置为Standard。对你写的任何实现类来说都必须设置id字段，并且这个值必须完全匹配模式的.js和.json文件名，并且是区分大小写的。然后是这个模式的两个专有字段，currentOperation和lastKeyPressed。估计你自己写的任何模式很可能也需要这两个字段，但是我把它们拿出了Mode类，因为可能有一些情况下并不需要它们。它们的名字应该明确地表明它们的用途：存储当前正在执行的操作符（最后一个点击的操作符按钮，如+、–、×或/），以及存储被点击的最后一个按钮（是合适的操作符需要的，如你将要看到的）。

我想稍微跳跃一点点，用一种比代码中的顺序更合理一些的顺序来讲解这些方法。首先，让我们看看与Backspace按钮相关的commandButton3()方法：

```
this.commandButton3 = function() {

  if (this.resultsCurrent != "") {
    this.resultsCurrent =
      this.resultsCurrent.substr(0, this.resultsCurrent.length - 1);
    this.updateResults();
  }

} // End commandButton3().
```

不可否认，它并不是一个复杂的函数。在确认了有一个当前值后（因为如果没有的话，就没有什么东西可删除，因此这种尝试就会得到一个错误信息），我们就从当前结果字符串中删掉最后一个字

符，并调用updateResults()来显示它。

现在可能是回答一个可能已经出现在你眼前的问题的好时候：为什么resultsCurrent和resultsPrevious字段的值通常都是字符串呢？答案是，这样做可以让退格更加容易。它还让显示负数更加方便，并可以令BaseCalc模式下的进制转换更容易。在真实的计算发生之前把这些值保留为字符串可以让代码更干净、更整洁一些。

Clear按钮同样也是一段非常简单的代码：

```
this.commandButton4 = function() {

  this.resultsCurrent = "";
  this.updateResults();

} // End commandButton4().
```

我敢打赌你不需要我解释这些，所以让我们继续前进吧。

每一个数字按钮处理函数都非常相似，那么让我们就看看其中的一个：

```
this.button0_5 = function() {

  this.checkLastPressed();
  this.lastKeyPressed = "7";
  this.resultsCurrent += "7";
  this.updateResults();

} // End button0_5().
```

如前所论述，通过纯粹的位置计算，数字7按钮最后映射到button0_5()方法。在它里面，首先调用checkLastPressed()方法，我们马上就会看到，现在先跳过它。我们把7作为点击的最有一个按钮，然后把数字7添加到当前结果中。然后重新显示当前结果。其结果是，如果结果框中正在显示的是31，那么现在就会显示317，那么我们就会知道点击的最后一个按钮是7了。

现在我们看看checkLastPressed()方法，了解一下为什么知道最后一个被点击的按钮是非常重要的：

```
this.checkLastPressed = function() {

  if (this.lastKeyPressed == "+" || this.lastKeyPressed == "-" ||
    this.lastKeyPressed == "*" || this.lastKeyPressed == "/") {
    // Time to start entering a new number, but save the current one first.
    this.resultsPrevious = this.resultsCurrent;
    this.resultsPreviousNegated = this.resultsCurrentNegated;
    this.resultsCurrent = "";
    this.resultsCurrentNegated = false;
  }
  // When equals is pressed, it's also time to start a new number, but in
  // that case we clear the previous number too.
  if (this.lastKeyPressed == "=") {
    this.lastKeyPressed = "";
    this.resultsCurrent = "";
    this.resultsCurrentNegated = false;
```

```
    this.resultsPrevious = "";
    this.resultsPreviousNegated = false;
    this.currentOperation = "";
  }

} // End checkLastPressed().
```

这个函数的价值是，只有在点击一个新的按钮的时候，才会发生特定的事情。比如，当用户点击+、–、×或/时，那后面就应该是开始一个新的数字的时候了。打开标准的Windows计算器，并输入一个数字，点击某个运算符按钮，然后再点击一个数字按钮。注意开始了一个新的数字，老的数字被清除了（实际上它是被存储起来了）。如果我们想模仿那个功能，但不想到处都是大量的重复代码，那就只有一种方式来完成，就是使用这个函数。所以，我们检查那4个操作符中是否有一个被点击。如果是，就复制当前数字给前面的字段（resultsPrevious = resultsCurrent），我们还要复制这个值是否为负数（resultsPreviousNegated = resultsCurrentNegated）。然后清除当前数字并把它置为正。其结果就是像标准Windows计算器那样工作了。

用户点击等号时也是一个需要在这里处理的情况。点击等符号很明显是要执行一些计算，而那个动作是在该按钮的事件处理函数中被完成的。但是如果又点一个按钮时会发生什么呢？我们需要开始输入一个新的数字，但是在那种情况下，我们并不能把当前值保存为前一个值。而是只要开始一个新的数字就行了。等号按钮通常和重置按钮类似，所以在点击它时，我们需要清理一些额外的字段，分别是lastKeyPressed和currentOperation。

把上面的描述当作中转（不是能骑的那种）[①]，我们现在可以跳到处理等号按钮的方法上了：

```
this.button3_9 = function() {

  if (this.currentOperation) {
    var answer = 0;
    // Negate the current value if the flag says to.
    var resCurrent = parseFloat(this.resultsCurrent);
    if (isNaN(resPrevious)) {
      resPrevious = resCurrent;
    }
    if (this.resultsCurrentNegated) {
      resCurrent = resCurent * -1;
    }
    // Negate the previous value if the flag says to.
    var resPrevious = parseFloat(this.resultsPrevious);
    if (this.resultsPreviousNegated) {
      resPrevious = resPrevious * -1;
    }
    // Now perform the current operation.
    switch(this.currentOperation) {
      case "+":
        answer = resPrevious + resCurrent;
```

① Segway（思维车）：一种硅谷周围的小孩儿都愿意拥有的酷酷的滑板车，是超级天才Dean Karnen设计的。没错，我知道这个俏皮话不好理解，不过我在用Segway（思维车）！

```
        break;
        case "-":
          answer = resPrevious - resCurrent;
        break;
        case "*":
          answer = resPrevious * resCurrent;
        break;
        case "/":
          answer = resPrevious / resCurrent;
        break;
      }
      // Reset some variables so we're ready for the next operation or input
      // key, and finally, update the results to show the answer.
      this.resultsCurrent = "" + answer;
      this.resultsPrevious = "";
      this.resultsPreviousNegated = false;
      this.currentOperation = null;
      this.lastKeyPressed = "=";
      this.updateResults();
    }

  } // End button3_9().
```

首先，一个简单的排查：当前是否有操作在执行？意思是，如果用户输入一个数字就点击了等号，就什么事情都不会发生。如果当前有操作，我们就从获得当前值的数字形式开始。如果resultsCurrentNegated标志位表示了它是一个负数，就把它乘以-1使其变成负的。然后，我们对前面的数字也做同样的操作。这里，我们还要做一个检查：如果没有先前的结果，也就是说，parseFloat()的结果值并不是一个数字（我们是使用内置函数isNaN()来判断），那我们就把前一个值作为当前值。这允许我们去模仿标准的Windows计算器，在它里面，你可以使用9+=来获得18，然后再依次+=得到36等。

之后，代码就转换到当前操作上，并且执行对应的操作。然后，把resultsCurrent设置为结果，并把它接在一个空字符串上将其转换为一个字符串。清除前一个数值，把负号标志位重置为false，清空当前符号，并把等号记录为最后一个点击的按钮。最后，调用updateResults()，显示结果（记住，updateResults()显示resultsCurrent的值，我们则刚刚把答案存储在它里面）。这些真的非常简单。

上面的逻辑可以处理包括两个数字的操作。那么那些只包含一个数字的又怎么办呢——比如平方根和倒数？好的，让我们先看看平方根：

```
this.button0_9 = function() {

  if (this.resultsCurrent != "") {
    this.resultsCurrent = Math.sqrt(parseFloat(this.resultsCurrent)) + "";
    this.updateResults();
  }

} // End button0_9().
```

只要有一个当前值，我们就可以使用Math包中的sqrt()函数来做这个工作，然后把结果附加到空

字符串上，这样我们赋给resultsCurrent的值就依然是一个字符串。然后调用updateResults()，显示新的数字，就完了。

倒数也同样简单：

```
this.button2_9 = function() {

  if (this.resultsCurrent != "") {
    this.resultsCurrent = (1 / parseFloat(this.resultsCurrent)) + "";
    this.updateResults();
  }

} // End button2_9().
```

就只剩下一个函数了，那就是百分比。百分比有点像混合型的计算，它和加法、减法、乘法和除法一样需要两个数字。但是当点击%按钮的时候它就立即执行操作，所以计算是在一个事件处理函数方法中执行的，就如同对平方根和倒数一样。下面是那个方法的代码：

```
this.button1_9 = function() {

  if (this.resultsCurrent != "" && this.resultsPrevious != "") {
    var a = parseFloat(this.resultsPrevious) / 100;
    var b = a * parseFloat(this.resultsCurrent);
    this.resultsCurrent = b + "";
    this.updateResults();
  }

} // End button1_9().
```

现在，确认我们有一个当前值的检测中还包括一个确认有前一个值的检测。如果两个都有，那我们就获得它们的数值版本，并把前一个值除以100。然后把结果乘以当前值，得出一个百分比的值。然后就是再一次把resultsCurrent设置为此结果并使用updateResults()显示它。

剩下的最后一件要讨论的事情就是这个类究竟是如何从Mode类扩展的呢？答案是在Standara.js文件中代码的最后两行（不包括注释）：

```
// Standard inherits from Mode.
Standard.prototype = new Mode();
// Continue the sequence of events after this class is loaded.
calcTron.setMode(new Standard());
```

原型的概念在第2章中已经讨论过，所以如果第一行并不完全明白的话，我更希望你去那章看看。那一行实现了继承，它就是如此简单。

不过，第2行是新的，它继续进行前面讨论过的类装载流程。

这些就是标准CalcTron模式。接下来是BaseCalc模式。

4.4.7　编写BaseCalc.json和BaseCalc.js

出于节约空间的目的，我不打算展示这个模式的JSON，因为老实说，只要你看过一个模式的JSON，就几乎相当于看过全部了。你可以自己看看，但是如果你已经看过标准模式的JSON，那在它上面不要花费多于一分钟——它们完全相同。

BaseCalc模式的实现类也是非常类似于Standard模式类，但是还有一些区别。那我们从研究它的UML图开始吧，如图4-10所示。

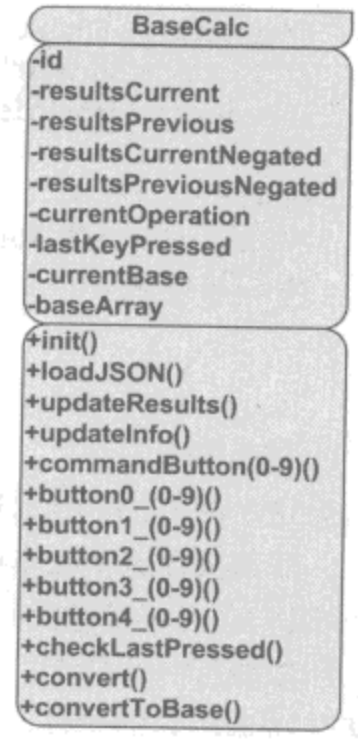


图4-10 BaseCalc类的UML图

正如你预期，它包含所有与Standard类相同的字段和方法，这是因为它们都是扩展自Mode类。不过它还有一些附加的条目，这在扩展一个类的时候也是允许的。currentBase字段记录了当前值的进制：十进制、十六进制、二进制或八进制。baseArray是一个字母数字值的数组，它是在进制转换过程中使用，后面会看到。除了这些额外字段之外，你还看到一些附加的方法：conver()和convertToBase().()。我们一会就会接触它们，之前先看看一些校准的东西。

当你查看这些代码的时候，你会发现它和Standard类非常相似，但是一个区别应该跳到你眼前：BaseCalc实现了init()方法。事实上，它覆盖了在超类（Mode）中的init()。让我们看看那里到底发生了什么稀奇古怪的事情：

```
this.init = function(inVal) {

  if (inVal) {

    // Initialize array for base conversions.
    this.baseArray[1] = "0";
    this.baseArray[2] = "1";
    this.baseArray[3] = "2";
    this.baseArray[4] = "3";
    this.baseArray[5] = "4";
    this.baseArray[6] = "5";
    this.baseArray[7] = "6";
    this.baseArray[8] = "7";
    this.baseArray[9] = "8";
    this.baseArray[10] = "9";
    this.baseArray[11] = "A";
    this.baseArray[12] = "B";
    this.baseArray[13] = "C";
    this.baseArray[14] = "D";
    this.baseArray[15] = "E";
    this.baseArray[16] = "F";

    // Call superclass constructor. Note that this only works if the
    // method of the superclass does not reference anything specific to the
    // subclass... see the notes about the id field on the next statement!
    BaseCalc.prototype.init(inVal);

    // Note that the call to init() of the superclass will result in the
    // information bar saying "null Mode" because the this reference points
    // to the instance of Mode that is the prototype for this BaseCalc
```

```
        // instance. So, we need to set it here using the id field of this
        // instance so what's in the info bar is correct.
        this.updateInfo(this.id + " Mode");

      } else {
        // Load the JSON for this tab.
        this.loadJSON(this.id);

      }

    } // End init().
```

因为这个版本的init()覆盖了Mode里面的同名方法，所以，任何时候调用calcTron.currentMode.init()，都会执行这个版本的init()（因为当我们转到那个模式的时候，currnetMode将指向BaseCalc的一个实例）。而不是像Standard类里面那样，用在Mode中的那个。第一次调用它的时候，没有传入任何值，所以执行else分支，并且载入了这个模式的JSON。和在Mode类版本的init()中一样。

第二次调用它的时候，会被传入那个已载入的JSON，所以我们进入到if语句段中。那里，你看到的第一件事情就是初始化baseArray。那个数组的目的我们稍后再讲，现在只要知道这里是给它填充数值就足够了。

然后，你就会看到一个有意思的东西，这可能是你在JavaScript中第一次见到的：调用超类版本的函数。为了这么做，我们引用BaseCalc类的原型，并调用它的init()。如果你很熟悉Java的话，你可以认为BaseCalc.prototype等效于Java中的super()。不同的就是，在JavaScript中，必须指定你想要调用的那个方法的名字，因此，就添加了init()。这个调用同时还带了JSON输入，所以现在执行了你在Mode基类中看到的那个普通的init()的功能。调用返回后，我们就更新信息条来描述新的模式。机敏的读者可能想知道为什么必须这么做，是因为它是在Mode类的init()方法中完成的。下面是Mode中实现这个目的的代码：

```
this.updateInfo(this.id + " Mode");
```

这里的问题是，这个引用指向的并不是BaseCalc的实例，而是指向Mode本身，所以在信息条中显示了一条错误的信息。因此我们实际上是在控制回到BaseCalc的init()之后，覆盖了Mode的init()做的事情，这样问题就解决了。

BaseCalc的lastKeyPressed()方法实际上和Standard的版本相同，只有在第一个if段中有多出来的一行代码。这行代码是把当前值转换为十进制的：

```
this.convert("dec");
```

每次做计算的时候，我们都使用十进制，然后只要把结果转换成任何当前进制就可以了。注意有一个字段——currentBase，它告诉我们当前值的进制信息，但是并没有表示前一个值的进制信息的字段。原因我们很清楚，因为有了这行代码，前一个值一定是十进制的。如我说过的，那个方法的剩余部分都与你已经看到过的相同。

Backspace和Clear命令按钮也与它们在Standard中相同，所以就没有必要在这里介绍它们了。

下面，我们来看看让我们可以转换模式的4个命令按钮。它们都是相似的，所以我们在这里就只看看十六进制的那个：

```
this.commandButton1 = function() {

  this.updateInfo("Hexadecimal");
  if (this.resultsCurrent != "") {
    this.convert("hex");
    this.updateResults();
  }
  this.currentBase = "hex";

} // End commandButton1().
```

这里，我们只是展示底部信息条的新模式。如果有当前值，就调用convert()，把目标进制传递给它，把它给转换成对应的进制——本例中是hex（十六进制）[其他可用的参数值是dec（十进制）、oct（八进制）以及bin（二进制）]。我们也更新结果显示，它让我们可以把当前值转换成任何进制。最后，记录新的数进制，就完成了。

像命令按钮一样，输入按钮和Standard类中的几乎完全相似。不过，在有些按钮中还是有点不同，让我们以数字7的按钮为例：

```
this.button0_5 = function() {

  if (this.currentBase == "bin") {
    return;
  }
  this.checkLastPressed();
  this.lastKeyPressed = "7";
  this.resultsCurrent += "7";
  this.updateResults();

} // End button0_5().
```

在这里，我们从一个检查开始，确认当前进制不是二进制，如果是二进制的，就什么都不做，立刻返回。这样做的效果是，在数字进制使用二进制时，只有按钮0和1才有反应，所有其他都将没反应（0和1之后的任何按钮都经过这个检查）。类似的，在按钮8和9里，你还会看到如果进制是二进制或八进制的，那么就会立即返回一个检查。以及如果数值进制不是十六进制的话，A-F按钮也会立即返回的检查。其结果是，只有对于当前所选择的数值进制合法的按钮才能起作用，这也是我们期待的。

+、−、×和/按钮都与Standard中的相同，所以这里就跳过它们了。在这个模式中，没有负数、百分比、平方根或者倒数，所以也没什么可看的。

现在我们来看等号的方法。它还是很像是在Standard中的一个副本，所以我们就只看看区别。首先，前面说过，所有的计算都是使用十进制来完成的。所以，你可能已经猜到，我们看到的第一件事就是把当前值转换为十进制数（前一个数值已经是十进制的了，所以只要关心当前的数值就可以了）。然后是在当前操作上的和在Standard中相同的switch。在那之后就是一些区别了：

```
// Next, convert the new current value to the current base.  Before we
// can do that though, we need to set the current base to decimal to
// match the answer we have.
var storedCurrentBase = this.currentBase;
this.currentBase = "dec";
```

```
this.convert(storedCurrentBase);
this.currentBase = storedCurrentBase;
```

我们需要把结果转换到当前的数值进制。不过，当看convert()方法时，你会注意到currentBase字段已经被convert()修改成为进制转换指定的进制了。这是因为conver()为4个数值进制命令按钮用来转换当前值并记录新进制的。所以我们这里首先保留当前数值进制，然后调用convert()，结果是currentBase被设定为dec。然后把currentBase设置为调用convert()之前存储的进制，它可能是也可能不是dec，结果是当前的进制保留下来，而实际上是没有触动convert()做的修改。

下面就是convert()方法本身：

```
this.convert = function(inNewBase) {

  var currentValue = null;
  switch (this.currentBase) {
    case "dec":
      currentValue = parseInt(this.resultsCurrent, 10);
    break;
    case "hex":
      currentValue = parseInt(this.resultsCurrent, 16);
    break;
    case "oct":
      currentValue = parseInt(this.resultsCurrent, 8);
    break;
    case "bin":
      currentValue = parseInt(this.resultsCurrent, 2);
    break;
  }
  switch (inNewBase) {
    case "dec":
      currentValue = this.convertToBase(currentValue, 10);
    break;
    case "hex":
      currentValue = this.convertToBase(currentValue, 16);
    break;
    case "oct":
      currentValue = this.convertToBase(currentValue, 8);
    break;
    case "bin":
      currentValue = this.convertToBase(currentValue, 2);
    break;
  }
  this.resultsCurrent = "" + currentValue;

} // End convert().
```

它是以获得当前值并将其转换为一个数字值开始。我们使用parseInt()函数实现这个目的。这个函数接受数值进制作为它的第二个参数（它的第一个参数是待转换的数字）。然后我们在currentBase分支上来判断做什么转换。一旦有了当前值的数值版本，我们就在inNewBase上switch并调用convertToBase()，给它传入当前值和目标进制。然后当前值就被转换为目标进制的数值，并转换成一

个字符串。

你可能已经注意到，在convert()中，实际上没有发生真正的转换。转换函数是通过convertToBase()方法来实现的：

```
this.convertToBase = function (inNumber, inNewBase) {

  var str = "";
  var calc = inNumber;
  while (calc >= inNewBase) {
    var divVal = calc % inNewBase;
    calc = Math.floor(calc / inNewBase);
    str += this.baseArray[divVal + 1];
  }
  str += this.baseArray[calc + 1];
  var len = str.length;
  var fnl = "";
  for (var j = 0; j < len; j++) {
    var a = (len - j) - 1;
    var b = len - j;
    fnl += str.substring(a, b);
  }
  return fnl;

} // End convertToBase().
```

如你所看到的，这里就是baseArray字段登台的地方了。简单地说，这个函数把引入的数字不断地除以进制，每除一次都从baseArray中拿到适当的元素添加到输出字符串。这个迭代的过程在余数小于进制时结束。这时，再在baseArray中查找余数对应的字符。最后要把输出字符串翻转过来，因为得到它的时候，结果是反的。翻转完了之后，就返回字符串，得到的就是输入数值转换到请求的进制的结果了。

这就结束了BaseCalc的代码评审，实际上也结束了CalcTron。

4.5 练习

我承认，在成长的过程中，我从来不是一个疯狂的数学爱好者，但是作为一个成年人，我越来越感激它了——不管是从实践的角度还是单纯的对智力的追求的角度。我希望，CalcTron至少可以开始引导你走相同的路。结束之前，我提供一些建议，你可以从这个项目中学到更多的东西——不仅仅是关于JavaScript的，还有一些一般的数学概念。

- 添加记忆功能：这个比较简单的，你可以从这里开始。大多数计算器都有记忆功能，但是CalcTron没有。我故意把它留下来给你一个简单的开始，那么去做它吧。
- 扩展Classloader：把Classloader类修改得更加一般化，让它可以载入任何类。也许是传给load()方法一个包的全称，如com.omnytex.javascript.SomeClass，然后把com.omnytex. javascript部分当作一个子目录使用（那么它就变成com/omnytex/javascript）。还有，改变load()方法，让它以参数形式接受用于验证的参照类，并当load()完成时自动调用verify()（你也可以把确认步骤设置为可选的）。另外，研究一下，看看是否有方法来确定一个<script>标签是否成功

载入，并把它添加为一个验证步骤（我在写这些的时候也并不确定这是否可行，所以对于我们来说都是很有趣的练习）。

- 添加转换功能：开始的时候，本章后面跟着一个关于创建ConvertTron的章节。这个概念是扩展CalcTron来包含转换能力。不过在项目进展过程中，出版的事也发生了变化，所以那章就没能加进来。不过，基本的概念还是很好的。转换不难添加，那么去试试吧。
- 添加简单的算术功能：一个简单的代数方程解析器怎样？我个人不会尝试创建一些代码来解联立方程那样的东西，但是9=*x*+5应该并不难解决。写一个运算法则来处理这种简单的方程式，可能并不太困难，但是却应该有足够的挑战，是一个很好的练习。

4.6 小结

本章的项目看起来是一个非常简单的想法：基于JavaScript的计算器。它对我来说可远称不上简单。我们有基本的计算器概念并把它转化成一个可扩展的框架，这个框架可以随着数学需求的增长而增长。你看到了在JavaScript中，如何实现一些通用的面向对象技术，以及可以用哪些技术如何增强基本应用程序。你甚至还看到了一个编写一种比较原始的类装载器的方法，它可以为你自己的应用验证载入的类是否有效（当然是某种程度上的）。

你还知道了如何动态载入JavaScript和JSON，而不重新载入页面也不使用意味着Ajax技术的XMLHttpRequest对象。还有，你看到了一些Rico库提供的功能。我希望你会同意本章坚持的“海水不可斗量”的说法。即使是一个相对简单的项目，也可以展示一些非常有意思的技术。

第5章

怪物混合：混搭

5

很久以前，一些像上帝一样的开发人员，提出了API的概念。简单地说，API就是其他（需要使用它的）程序或系统熟知的接口。开发人员提出这个概念，然后所有人都看到这个概念，并觉得那是个好东西。但是，用Leonard H. McCoy博士的不朽名言来说“……工程师，他们喜欢改变事物。”我们不能只粘在API这个术语。不，我们只需要提出些新的东西，那个东西就是术语混搭（mashup）。因为，它是一个在这个时代风行的名词，并且还是一个在很大程度上涉及JavaScript的东西，所以它是这本书最合适的主题。那么，在本章中，我将要介绍混搭的概念，然后我们会把概念付诸实际，用在一个有用的小应用程序中。

5.1 什么是混搭

混搭这个概念在逐渐地被人们了解，其基本原理是网站或者应用程序的内容来自多个源，最常见的是通过某种公共的编程接口，把这些信息整合成为一个新的体验——那就是，一个新的应用程序。如果这个听起来有点像给你提供Web服务，没错，你理解的差不多。实际上，Web服务有些时候是和混搭相关的，不过Web 服务通常包含服务器架构和一些服务器端代码等，不一而足。通常你不会从一个JavaScript客户端调用一个真的Web服务（SOAP、UDDI、WSDL等所有Web服务表示的术语），并且几乎肯定不会从浏览器调用（通常指的是不带插件或者类似技术的浏览器）。

不，在最近，混搭这个词的意思已经逐渐变成基于浏览器的JavaScript客户端把来自不同公司和供应商的公共API的内容整合成新的应用程序。这些API通常都是当作Web服务来引用，并且即使它们在完整的技术栈的环境里可能并不真的是Web服务，但它们实现的功能却和那些Web服务基本相同。它们通过网络提供服务和函数——特别是Web，所以把它们称作的Web服务并不是那么风马牛不相及！

一些公司正开始涉及API业务，包括那些你肯定已经听说过的公司：Google、Yahoo、Amazon和eBay，这只是其中几个。Google和Yahoo事实上已经是领导者了，尤其是Yahoo，发明了一种非常棒的技巧，这将会是本章中我们要做的应用程序的中心部分。

混搭也是人们使用Web 2.0这个词时所要表达的意思的一部分。Web 2.0对不同的人的含义是不一样的，但是共享资源通常是用户认识的其中一个含义，所以混搭刚好落在这个范畴。

Web 2.0经常提到的另外一件事情是特效。比如，看看像Digg那样的网站。我本打算在这里放一个截屏，不过说实话，截屏无法概括要点，因为必须看现场的效果。所以请访问http://www.digg.com，如果你还不是一个频繁的访问者（顺便说一下，你应该是），就多看下。你看的时候，注意一下各种各样的效果。比如，假设你没有登录，尝试点击Digg It，它是在一个圆圈旁边的按钮。注意到文字是

如何淡出的以及一个文字上的一个小的弹出框，它告诉你关于登录的事情了吗？现在如果你创建一个账号、登录，并试着再次点击那个按钮，你就会看到Digg计数淡出，然后淡入一个新的数值。这些都是各种UI效果的例子，而UI效果是Web 2.0中最为看重的一部分。

5.2　怪物混搭的需求和目标

现在，带着所有关于混搭想法，让我们讨论一下本章的应用程序将要做什么，以及它要说明的事情。

- 应用程序的基本功能是用户可以输入一个ZIP代码并从中获取一个酒店列表。
- 对于每一个酒店，我们应该可以点击它的名字来看到一些扩展的信息。
- 除了附加信息之外，我们还要看到一个那个地方的地图。
- 我们应该可以放大缩小那个地图。
- UI将会使用著名的script.aculo.us库提供的多种效果。
- 我们将使用一些Google和Yahoo的API来获得酒店信息以及地图。
- 这个应用程序应该单纯地基于浏览器，而不需要任何服务器组件就可以运行。

实现所有这些就能让我们做一个完全符合行业标准的并且非常符合Web 2.0普遍定义的应用。让我们先从看看Yahoo和Google的API开始吧。

5.3　Yahoo的API

Yahoo不久前做了一些非常酷的事情，正是这个酷玩意才让本章的应用可能实现。不过在讨论它之前，我们不得不先说说这个酷玩意出现以前是什么样子的。

一段时间以来，一些公司已经开放了许多公共的API让人们使用，Yahoo也是其中之一。例如，你可以远程执行一个Yahoo搜索，或者从你自己的应用程序中得到一个Yahoo地图，等等。这些API——如果你愿意的话，也可以叫它们“Web服务”，通常使用XML作为它们的数据交换机制。你可以给指定的URL张贴一些XML，然后就会获取一个XML返回。它是如此的简单。这些服务的种类不需要所有像SOAP、UDDI、WSDL等那样的Web服务。

你可能已经听说过Ajax这个名词，它的意思是Asynchronous JavaScript And XML（异步javaScript和xml）。实际上，如果你看过第4章，你已经在实践中了解一些Ajax了（不过，具有讽刺意味的是，和本章要做的一样，它并不使用原型的XMLHttpRequest对象）。我们会在后面的章节中，更详细地了解Ajax，但是现在，你只要知道Ajax是一种可以用来“给服务器发送请求，并以某种方法使服务器返回结果而不重新载入整个页面”的技术就行了。最通常的操作只是把返回的结果插入到页面的某个地方，实际上就是在“带外”[①]执行一个页面的的局部更新。Ajax通常（一些人可能说是必须的，但是我不同意）暗含有在浏览器中使用XMLHttpRequest对象来发送请求的意思。

XMLHttpRequest是一个组件，它帮你将请求发送给服务器，然后调用一个指定的JavaScript回调函数来处理结果。为了这里的这个讨论，你需要意识到这个对象有一个一贯的限制，那就是所谓的同域的限制。意思是XMLHttpRequest对象不允许对其文档本身所处的服务器的域之外的其他的域发起请求。比如，

① “带外”指的是一次页面的刷新。——译者注

如果有一个名为page1.htm的页面，位于http://www.omnytex.com，你可以发请求给任何 www.omnytex.com下的URL。然而，如果你企图发请求给在www.yahoo.com下的一些URL，XMLHttpRequest对象就不允许你这么做了。这意味着，如果你试图直接从浏览器访问Yahoo发布的API是不可能的。

有一些方法可以绕开同域的限制。最普遍的可能是在你自己的服务器上写一个服务器端的组件来扮演一个代理服务器。所以可以通过XMLHttpRequest发送请求给一些类似www.omnytex.com/proxy的URL，它会生成一个请求到www.yahoo.com，并返回结果。这是非常酷的。

不过，如果可以直接从浏览器中向Yahoo发请求，而不需要一个服务器端组件的话，不是会好用得多吗？是的，它真的会好用得多！你可能已经猜到了，有一种方法可以这么做。看看下面的一些JavaScript代码：

```
var scriptTag = document.createElement("script");
scriptTag.setAttribute("src", "www.yahoo.com/someAPI");
scriptTag.setAttribute("type", "text/javascript");
var headTag = document.getElementsByTagName("head").item(0);
headTag.appendChild(scriptTag);
```

那么，这里用的是一个新创建的<script>标签。我们把它的src属性设置为指向Yahoo中的某些API，最后把那个新标签添加到文档的<head>上。浏览器将会去那个URL并取回它的资源，然后解析它，如同其他引入的JavaScript文件一样。

现在，如果就是这些的话，其实并没有什么用。如我所说的，Yahoo API返回XML，并且浏览器执行解析XML（一些浏览器可能从它生成一个DOM对象，但是即便如此，就只有它自己的话也是没什么用的），没什么意义。不像使用XMLHttpRequest对象那样，你用不着处理任何事件，回调函数可以在返回的内容上进行操作，等等。

现在我们看看之前提到的很酷的东西！

假设我们有一些从Yahoo的服务中返回的XML，如下：

```
<name>Frank</name>
```

它可能没有那么有趣，但是它完全是个合法的XML。所以与之等价的JSON是什么样子呢？它就只是下面这样而已：

```
{ "name" : "Frank" }
```

好的，现在假设我们把那个JSON传给一个JavaScript函数，如下：

```
someFunction( { "name" : "Frank" } );
```

传给someFunction()是什么参数？如它显示的，它是一个由JSON组成而来的对象。意思是说如果someFunction()是这样的：

```
function someFunction(obj) {
  alert(obj.name);
}
```

结果就会是一个弹出框说“Frank”。

你是不是正要开始看看Yahoo可能做了什么？如果你说返回像下面这样的东西：

```
someFunction( { "name" : "Frank" } );
```

那你是绝对正确的！

Yahoo带来的是返回JSON取代XML的想法，并且把JSON封装在一个函数调用中。当调用API函数时，你告诉它回调函数是什么。那么，假设我们想和一个返回人名的Yahoo API交互，就如同例子中已经说过的。页面看起来应该像这样：

```
<html>
  <head>
    <title>Dummy Yahoo API Test</title>
    <script>
      function makeRequest() {
        var scriptTag = document.createElement("script");
        scriptTag.setAttribute("src", "www.yahoo.com/someAPI/callback= ➥
myCallback&output=json");
        scriptTag.setAttribute("type", "text/javascript");
        var headTag = document.getElementsByTagName("head").item(0);
        headTag.appendChild(scriptTag);
      }
      function myCallback(inJSON) {
        alert(inJSON.name);
      }
    </script>
  </head>
  <body>
    <input type="button" value="Test" onClick="makeRequest();">
  </body>
</html>
```

当点击按钮时，调用了makeRequest()，并且它使用动态<script>标签的技巧来调用Yahoo API函数。注意URL，它指定了回调函数的名字，我们想要返回的JSON形式，而不是通常的XML。应答返回时，浏览器通过<script>标签计算要插入文档的内容，如下：

```
myCallback( { "name" : "Frank" } );
```

myCallback()在这点运行，并且带着传入给它的JSON中分析得到的那个对象。你可以从任何域名载入这个页面，它都会工作。所以，我们完成了XMLHttpRequest对象所做的工作（当然是基本概念了），并且已经可以绕开同域限制了。太好了！

Yahoo是第一个使用这种技巧的（我所认为的），但你也将看到，其他公司也已经开始跟随这种方法了，因为这样就允许单纯的客户端混搭和API的使用。你就不再需要一个服务器端的代理了现在可以直接跨域发送请求了。

注意　虽然这个技术非常有用，因为它允许你直接发送请求给任何想要的服务器，但它也有引出恶意代码的潜在可能。要记住，返回的东西是脚本，它和页面中其他脚本有相同的执行权限。这就意味着，可能存在潜在的恶意脚本进行偷取cookie、嗅探、钓鱼等动作。因此，你需要关心所选的服务和组织机构。比如，访问Yahoo或Google的API一般不会出现任何安全性的问题，但是不那么知名的公司可能就没有这么安全了。

现在你知道了，我们如何与Yahoo API以及Google API进行基本的交互，下面让我们看看将要在这

个应用程序中使用的Yahoo的函数。

5.3.1 Yahoo Maps地图服务

Yahoo将为我们提供可以在应用程序右边看到的那个地图（你可以看看图5-2的预览）。Yahoo Maps是个已经存在了一段时间的服务，甚至在提供公共接口之前就已经存在了。它允许你获取指定地址的地图，以及其他的特征，比如交通和当地的名胜古迹。Yahoo提供的API有许多不同的服务，但是对于我们的目标，我们就关注地图服务。

Yahoo Maps地图API可以让你获得一个根据你的请求中的参数生成的地图的图像的引用。你可以在请求中指定纬度和经度或者地址（我们将指定地址）。

这个服务是通过一个简单的HTTP请求被引用的，如下：

```
http://api.local.yahoo.com/MapsService/V1/mapImage?appid=YahooDemo&location=11719
```

location参数指定的就是一个美国邮政编码，appid是一个当你注册这个服务的时候获得的ID，下一节中会详细讨论。如果你把上面的URL粘贴到浏览器的地址栏中，你将看到如下回复：

```
<Result>
http://img.maps.yahoo.com/mapimage?MAPDATA=ytUWRed6wXWoR2TGzwl3wROg3iyHedtGDvtw ➥
766fmR4iboSayYoDOI4llk594b5QaoMqKvZB5AdndE5FtDXv8lT8apVTrjOY5Zuhrhiugmeogq5t ➥
GHi5&mvt=m
</Result>
<!--
 ws01.search.re2.yahoo.com uncompressed/chunked Sun Dec 10 22:18:44 PST 2006
-->
```

你得到的反馈是指向一个位于Yahoo的服务器中的图像。如果你取出如下的URL：

```
http://img.maps.yahoo.com/mapimage?MAPDATA=ytUWRed6wXWoR2TGzwl3wROg3iyHedtGDvtw ➥
766fmR4iboSayYoDOI4llk594b5QaoMqKvZB5AdndE5FtDXv8lT8apVTrjOY5Zuhrhiugmeogq5t ➥
GHi5&mvt=m
```

然后把它放在一个Web浏览器的地址栏中，你将看到一个图片，它是纽约长岛的Bellport/ Mastic 岸区的地图。

你还可以添加一些参数给那个原始的请求。比如，你可以指定想要返回一个GIF（默认地，你得到的是一个PNG文件），你可以指定想要返回JSON来取代XML。那么URL看起来将会是这样的：

```
http://api.local.yahoo.com/MapsService/V1/mapImage?appid=YahooDemo&location=11719 ➥
&image_type=gif&output=json&callback=myCallback
```

现在响应看起来就像这个：

```
myCallback({"ResultSet":{"Result":"http:\/\/image.maps.yahoo.com\/mapimage? ➥
MAPDATA=cgnWqud6wXUpZCKOcjzKJ3PPgRQkY6thMdXo4raWKRcxvbRSpJ67PGisuDp5Y0829Zi5fd ➥
hWYTOm5mmvfBZCqHKDBG8ePGPcc8AlFAuhbWwd6rPOwZ67&mvt=m"}});
```

现在，所有你需要做的就是写myCallback()函数，并告诉它如何显示图像，如下：

```
function myCallback(inJSON) {
  document.getElementById("someImgTag").src = inJSON.ResultSet.Result;
}
```

如你将要看到的，这些是所有这个应用程序与Yahoo的服务进行交互时需要做的！这个应用程序

中还使用了一些其他的参数，如表5-1中所总结的。

表5-1　一些在怪物混搭项目中使用的Yahoo Map地图服务的参数及其含义

参　数	含　义
width	地图图片的宽度
height	地图图片的高度
zoom	应用于地图的比例尺。数值范围是1~12，1表示街区级别，12表示地区级别（比州稍微大一点点）

5.3.2　Yahoo的注册

大多数API服务需要注册才能使用它们的API，Yahoo也不例外。如你所见，在HTTP请求中，传入了一个appid参数。这个值是你发送请求时必须传入的一个唯一标识。如果不传这个参数，或者传入一个非法值的话，就会导致调用失败。YahooDemo是在Yahoo自己的文档的示例中使用的appid值。然而，在你真的大量使用这个应用程序之前，最好注册并获得自己的appid。你可以通过如下的页面来注册：

```
http://api.search.yahoo.com/webservices/register_application
```

在你开始花时间在应用程序上之前，你要把自己的appid添加到Masher类中（在Masher.js源文件中），这样也是你善待Yahoo的方式。

使用API对请求的数目有一些限制，但是上限大得让你根本不需要在这个应用中考虑这个问题！如果你决心要使用这些服务来创建一个产品级别的应用程序的话，可能会需要询问Yahoo获取允许更大访问量的选项。重复一下，对于我们的目标，允许的请求的数量已经足够多了。

5.4　Google的API

Google Base，简单地说，就是一个在线数据库，用户可以在那里发布各种各样的条目，用不同的属性来描述它们。例如，如果你想罗列一些正在你邻居中发生的事情，可以使用Google Base来做。你想要发布你的姨妈艾玛的独家腌菜秘方，那Google Base就是一个好地方。

如你所期望的，你也可以在发布的信息中进行搜索。你想要查找一个指定地点的酒店吗？你可以在Google Base中完成。并且那确实是我们这个应用程序所需要的。

Google Base的API允许你查询一个条目列表，它还允许你向数据库中添加条目。这里，我们只关心查询，但是两种功能都是可用的。

Google Base API提供一些“料口”[①]，通过它们你可以获得信息。每一个料口都对应一个URL，那个URL是在一个基础URL后面附加特定的料口路径组成的。例如，基础URL是http://www.google.com/base，片段料口指定的料口部分是/feeds/snippets。你只要把它们放到一起就组成了最终的URL，可以用来查询项：

```
http://www.google.com/base/feeds/snippets
```

那么，给一个简单的例子，如果你想要查询“laptop”，就可以使用这个URL：

```
http://www.google.com/base/feeds/snippets?bq=laptop&max-results=1
```

① 原文为feed，此处译作机加工的投料口的料口。——译者注

这将返回一个如下的XML：

```
<?xml version="1.0" encoding="UTF-8" ?>
<feed xmlns="http://www.w3.org/2005/Atom" xmlns:openSearch="http://a9.com/-/spec/ ➥
opensearchrss/1.0/" xmlns:g="http://base.google.com/ns/1.0" xmlns:batch= ➥
"http://schemas.google.com/gdata/batch">
  <id>http://www.google.com/base/feeds/snippets</id>
  <updated>2006-12-16T23:28:51.897Z</updated>
  <title type="text">Items matching query: laptop</title>
  <link rel="alternate" type="text/html" href="http://base.google.com" />
  <link rel="http://schemas.google.com/g/2005#feed" type="application/atom+ ➥
xml" href ="http://www.google.com/base/feeds/snippets" />
  <link rel="self" type="application/atom+xml" href="http://www.google.com/base/ ➥
feeds/snippets?max-results=1&bq=laptop" />
  <link rel="next" type="application/atom+xml" href="http://www.google.com/base/ ➥
feeds/snippets?start-index=2&max-results=1&bq=laptop" />
  <generator version="1.0" uri="http://base.google.com">GoogleBase</generator>
  <openSearch:totalResults>3612575</openSearch:totalResults>
  <openSearch:startIndex>1</openSearch:startIndex>
  <openSearch:itemsPerPage>1</openSearch:itemsPerPage>
  <entry>
    <id>http://www.google.com/base/feeds/snippets/18343852209328178501</id>
    <published>2006-11-10T03:41:07.000Z</published>
    <updated>2006-12-08T03:24:30.000Z</updated>
    <title type="text">Portable Laptop Desk</title>
    <content type="html">Looking for portable laptop desk? See our portable ➥
laptop desk guide.</content>
    <link rel="alternate" type="text/html" href="http://portable-laptop- ➥
desk.info" />
    <link rel="self" type="application/atom+xml" href="http://www.google.com/base ➥
/feeds/snippets/18343852209328178501" />
    <author />
    <g:item_language type="text" />
    <g:customer_id type="int">7048781</g:customer_id>
    <g:target_country type="text" />
  </entry>
</feed>
```

bq参数是你想要执行的查询，在这个例子中，它就是单词laptop。参数max-results是可选的，它表示要返回的项的最大数量。为了在这里打印一个短的结果集，我只请求了一个返回项。

如果你想要查询一个特殊的项类型，例如酒店，你可以向如下的URL添加-/hotels。

```
http://www.google.com/base/feeds/snippets/-/hotels?bq=laptop&max-results=1
```

现在明显地，那是有一些无意义的查询，[①]但尽管如此，它仍是一个合法的查询。

一个稍微实际一点的例子，假设你想要查询一个指定位置的酒店，在一个给定邮政编码的地区的酒店列表（征兆!）。要执行这个查询，你需要使用location参数。location参数可以接受你能想象的任何形式—— 一个完整的街道地址、只是一个邮政编码或者甚至是经度和纬度。该服务通常会帮你

① 注意，你会得到结果，因为酒店会在自己的描述信息里列出笔记本无线上网支持，那个查询就会看到这些。

考虑你发给它的是什么，所以你甚至都不用告诉它！

那查询在90210地区的酒店的URL会是什么样子呢？就是像这样：

```
http://www.google.com/base/feeds/snippets/-/hotels?bq=%5blocation:@%2290210%22%2 ➥
b50mi%5d&max-results=1
```

在浏览器中试试它——这个也能用。为了更加完全地理解它，我需要指出你想要执行的那个查询中bq参数的值事实上是这个：

```
[location: @"90210" + 50mi]
```

意思是说，你想要找在90210邮政编码中心区约50英里的范围内地区的酒店。在URL中看起来有一点点古怪的原因是，我在URL上使用了URL编码：左和右花括吗、at符号（@）、加号和引号。在应用程序中你也会看到这么处理，只是它可能是在一步中把整个东西都编码了。两种方法都可以用。只要URL最后是URL编码的，这才是我们要关心的。

和Yahoo API相同，你可以声明说你需要返回封装在JavaScript函数调用中的JSON，并且你可以指定那个要调用的函数。这是通过alt和callback参数来完成的。当你给alt参数传递json-in-script值时，你获得的就是那个：封装在一个JavaScript调用中的JSON。callback参数自然就是那个发送JSON的函数的名字了。

虽然看起来这里的内容并不多，但这些实际上就是所有你在这个应用程序中要使用的API的信息！

说明　片段料口是一个只读的公共料口，这意味着你不需要获得一个API 认证字就可以访问它。如果你想要处理一些其他的料口，或者试图使用API来完成写或更新的操作，就确实需要一个认证字了，就如同你使用Yahoo所做的一样。你可以在http://code.google.com/apis/base/signup.html获得那个认证字。

5.5　script.aculo.us特效

script.aculo.us完全是关于特效的！包括飞入页面、元素收缩和扩大、实体的部分淡入淡出、循环变色的文本等——所有这些都可以使用script.aculo.us来完成。

特效不是很糟的视觉糖果吗

让我们先来应付一个经常出现在脑海中的问题：到底为什么需要特效？它不就是一些肤浅的视觉糖果吗？除了让人们产生“哦”和“哇”这样的反应，没别的什么作用吗？好的，首先，如果你曾经为其他人设计过应用程序，你就会知道，展现是整个系统中一个很重要的部分。人们越是喜欢应用程序的外观，他们就会越喜欢它的工作方式——无论它是否工作的很好。这是一个相对衡量而言的。当然这个理由相对没那么强，但也是一个不容忽视的理由。

更加重要的原因与我们如何感知事物相关。现在看看你周围，捡起任何你想要的东西并把它移到其他的位置。那个东西就会从起点弹出，然后在新的位置出现吗？不，当然不是！它平滑地从一个位置移动到另一个位置。猜到什么？这就是世界运转的方式！而且，这也是我们的大脑所期盼的事物运转的方式。当事物并非以那种方式工作时，就是不和谐的，就是令人迷惑的，也是令人沮丧的。

人类使用移动作为视觉暗示，暗示将要发生什么。这就是为什么现在操作系统正开始添加各种各样的酷炫的特征，像窗口收缩和打开。它们并不仅仅是视觉糖果。事实上，它们是为一个目的服务的：帮助我们的大脑的焦点处在感兴趣的事物上。

在一个Web应用程序中，有相同的道理。如果你可以使某些东西滑出，其他的一些东西滑入视线，它将会让用户更加愉快，而且更重要的，通过避免让人们失去哪怕是一小段时间的焦点，来帮助他们提高效率。

使用script.aculo.us可以被分解为简单的3步。

(1) 导入所需的JavaScript文件。

(2) 创建一个新的Effect对象，传给它将要执行特效的元素的ID，并且可选择传入特效的参数。

(3) 休息一下享受（你的特效）!

所需要的文件——prototype.js、scriptaculous.js、builder.js、effects.js、dragdrop.js、slider.js和control.js，就像其他外部JavaScript文件一样，通过<script>标签引入即可。一旦它们在页面中存在了，你就可以像这样初始化一个特效：

```
new Effect.Appear("div1");
```

这将开始一个Appear特效，它使一个元素在一个时间段中淡入。假设页面中有一个 ，ID是div1，它就将淡入。这里发生的事情是，一个新的对象被实例化，名为Effect.Appear对象。一个特效的构造函数的第一个参数通常是待操作元素的ID或者一个指向它自己的DOM对象，第二个参数是必须的参数（虽然大多数特效并不需要参数），第三个参数是一个选项集。好的，选项是可选的!如果没有传入任何选项，那么你将得到一些默认值的选项集。

大多数特效都共享一些常用的选项，如表5-2所示。

表5-2 一些常用的script.aculo.us特效选项

选　项	描　述
duration	设置特效持续的时间，浮点数，默认值是1.0
fps	每秒帧数的目标。默认值是25，不能高于100
transition	设置一个修改当前动画的函数，在0和1之间。支持下列的变换：Effect.Transitions.sinoidal（默认）、Effect.Transitions.linear、Effect.Transitions.reverse、Effect.Transitions.wobble和Effect.Transitions.flicker
from	设置变换的起点，一个介于0.0和1.0之间的浮点值，默认值是0.0
to	设置变换的终点，一个介于0.0和1.0之间的浮点值，默认值是1.0
sync	设置特效是否应该自动展现新帧，默认为假。如果为真，你可以通过调用一个特效的render()方法，手工展现帧
queue	设置队列选项。在和字符串一起使用的时候，它可以是front或者end，把特效放在全局特效队列的开头或者结尾，或者是一个可以有{position: "front/end", scope: "scope", limit:1}的队列参数对象
direction	设置变换的方向。值可以是top-left、top-right、bottom-left、bottom-right或者center（默认值是center）。它只适用于展开和收缩特效

所有的特效还支持在选项中使用回调函数。这允许你在一个特效的生命周期内发生特定事件时执行一些函数。表5-3总结了可能的回调函数。

表5-3　script.aculo.us特效可能的回调函数

回调函数	描　述
beforeStart	在主效果展现循环开始之前调用
beforeUpdate	在每次特效呈现循环里，在重画之前调用
afterUpdate	在每次特效呈现循环里，在重画之后调用
afterFinish	在特效的最后一次重画之后调用

在本章的项目中，我们准备用下面这些特效。

- 卷起（BlindUp）和卷落（BlindDown）：这些函数像百叶窗，基本上是把对应元素打开或者关闭。我们用它展开和压缩搜索结果。
- 收缩（Shrink）：这个特效将一个元素缩小到它的左上角，本质上把它压缩到那个角落。在显示一个酒店的扩展信息时，如果你点击一个新酒店，我们将使用收缩特效来隐藏当前展示的信息。
- 展开（Grow）：这个特效将一个元素扩展到视图中。我们将在扩展酒店信息并隐藏所有当前信息时使用它。
- 喷雾（Puff）：这个新特效提供了将元素像一片烟云那样吹走的幻觉。当执行一个新的查询时，我们将使用它来移走地图。

所以，了解这些之后，让我们看看一些样例特效以及如何写它们的代码。 所有这些都假设我们在页面上有一个ID为div1的 ，并且里面有一些文字。首先，让我们看看BlinkUp特效：

```
new Effect.BlindUp("div1",
  {
    afterFinish : function() {
      alert("All done!");;
    }
  }
);
```

这里，我们已经指定了一个回调函数，所以当一个元素完全卷起时，我们将看到一个弹出的警告信息。

下面是另外一个例子：

```
new Effect.Shrink("div1", {duration : 4.0, fps : 60 } );
```

这将在4 s内，把 移出视图，并将尝试以60 fps的速度来完成。假设计算机和浏览器可以达到这个帧速度，那它对于用户来说将非常平滑。记住那句常识，人类的眼睛在24 fps到30 fps的范围内，感知到的动画是平滑的，所以默认的值是25 fps是非常有道理的。[①]

这里有一些关于script.aculo.us特效的更多的备注。

- 所有的特效都是基于时间的，也就意味着如果你想要使用展开特效把一个元素扩展入视图，并且想它花两秒钟的时间来完成，那么特效就将要花费两秒钟，不管浏览器展现每一帧有多快。
- 通常，特效并不在乎提供给它们的元素的类型。通常应该可以给任何东西提供特效。现在，

① 比如，大多数电影的拍摄速率都是每秒24帧。

你将要发现一些例外，这是正常的，因为不同浏览器对CSS的解释是不一样的。不过，大多数时候，你会发现特效是不在乎元素类型的。

- 要想大部分特效能运转，必须至少给元素指定一些内联的样式属性，如果在一个外部样式表中指定的话，它们将不会生效。比如，本章的应用程序中你将看到的一些特效，如果display属性并不是内联的话就不会生效。这并不是什么大事，但是如果你第一次尝试一个特效并发现它什么都没有做的话，并且你想知道为什么，那么请记住这一点。这样很可能给自己节约一些时间！

了解了API和script.aculo.us之后，让我们开始看看应用程序本身以及它是如何组合在一起的。

5.6 怪物混合（搭）的预览

怪物混搭应用的外观乍一看相对简单，直到你实际用它。页面实际上包含下面4个主要的部分。

- 顶部，包含标题图片和输入邮政编码的地方。
- 一个其下部的段，在左边，显示酒店搜索结果（以及酒店信息）。
- 一个其下部的段，在右边，显示酒店所在位置附近的地图。
- 在结果和地图段下头，一个包含地图缩放按钮的段。

图5-1显示了执行一个搜索之后，查阅一堆酒店列表的样子。

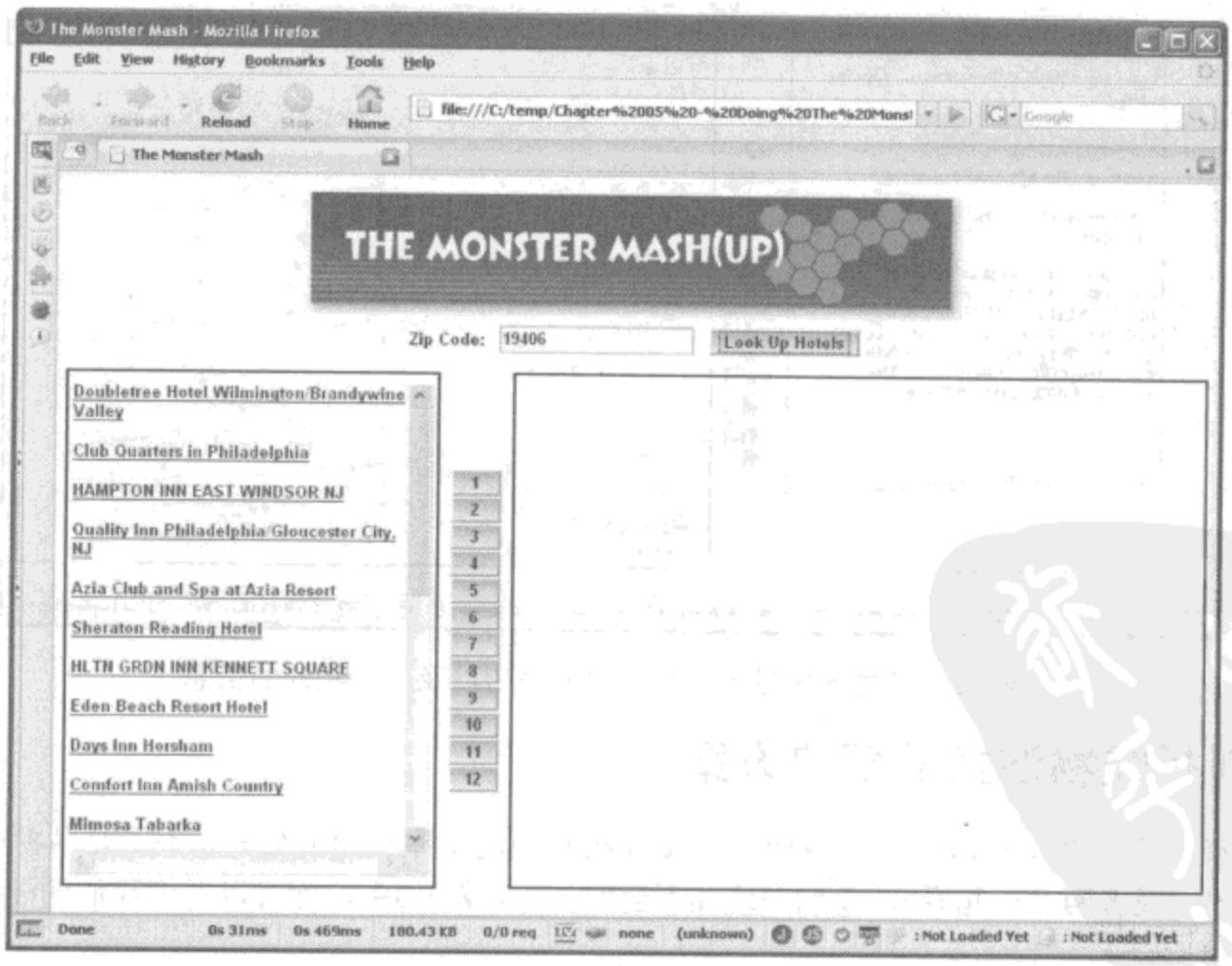


图5-1 怪物混搭显示了一个匹配请求的邮政编码的酒店列表

在你搜索结果的时候，服务器返回的酒店列表会伸展进入视野，显然你没法在这个截屏看到这个

特效！如果已经有结果正在展现，而你发出了另外一个查询，那么现存的结果会收缩出视野。

如果你点击一个酒店名字，那么就可以在点击项的下方获得扩展的酒店信息。这个信息“飞进”视野，当你点击其他的酒店名称的时候，它飞出视野。图5-2显示了查看信息的屏幕。

和扩展信息一起出现的还有该酒店周围的地图，在图5-2中也能看到。然后你可以使用缩放按钮放大或者缩小地图，获取更详细的或者是更大视野的地图。缩放级别越小，你看得越近。级别一是街区级别，级别12是地区/州级别。

如果在显示地图的时候发起一个新的搜索，那么就有一个非常炫的特效出现。你会看到地图“烟消云散”——也就是说，它向你飞去并逐渐淡出。每个人都会说，“多谢script.aculo.us！”

现在，该说兔巴哥和达菲鸭（你应该记得它们周六早晨的歌吧？[①]）的那句经典台词：“开始演出了！”

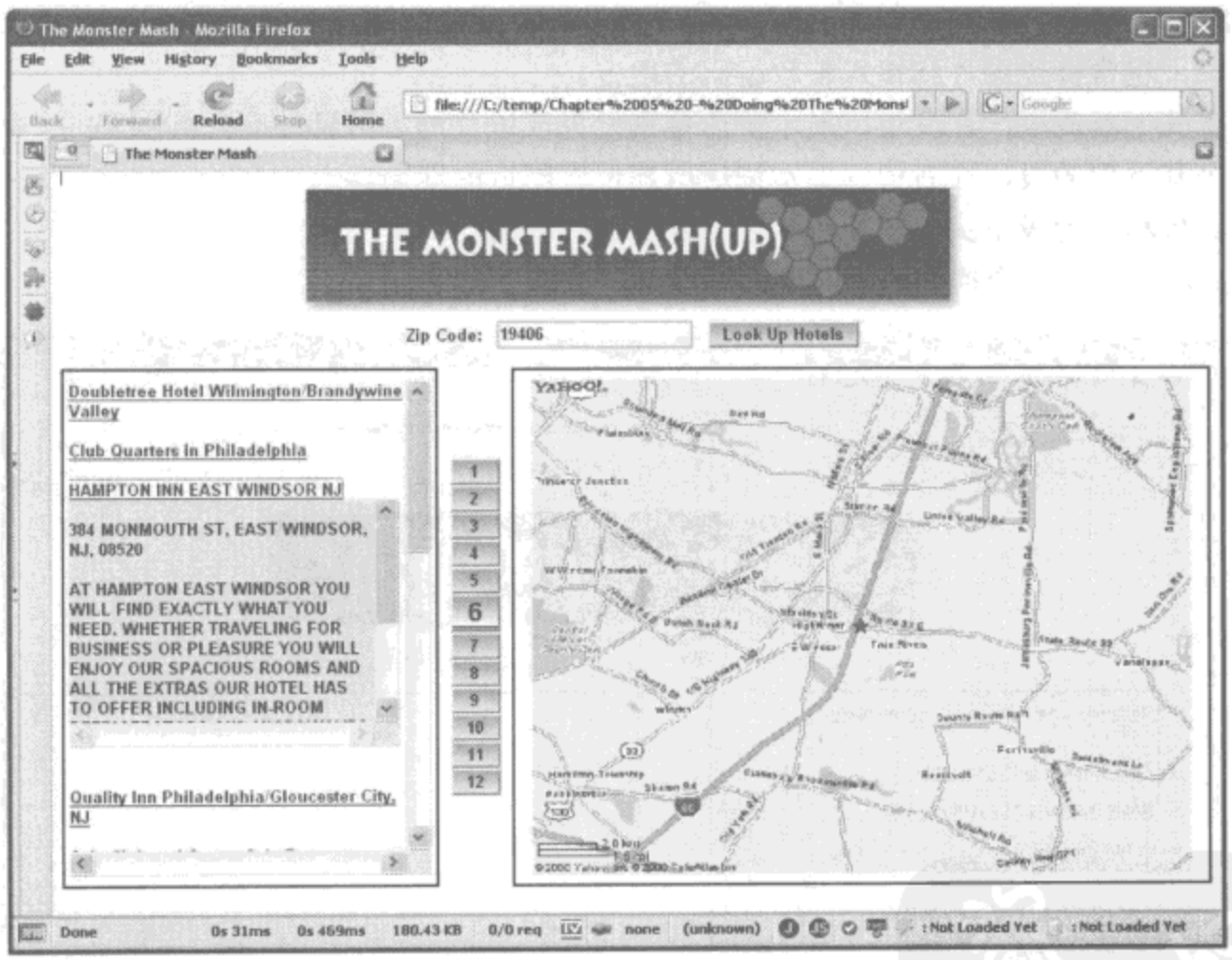


图5-2　怪物混搭显示一个选定的酒店以及一些额外的信息

5.7　剖析怪物混搭的解决方案

理解任何应用似乎都应该从组成应用的文件级别的高层次视角开始——不管是从源代码还是从可执行代码（或者两个一起看），我们在这儿不会例外！在图5-3中，你可以看到我们的混搭应用的目录结构。

根目录里只有一个HTML文件，它构成混搭主干，名字很没有想象力：mashup.html。css目录里只

① 美国每周六晨经常播放系列卡通片“兔巴哥”。——译者注

有一个styles.css样式表。在img目录里是4个图片：buttonBG.gif，是背景图片，给所有按钮金属质感；pixel_of_destiny.gif，也叫单像素透明图片（去搜索一下“pixel of destiny”，就知道好玩的了！）；retrieving_map.gif，在检索图片的时候地图区里显示的图片信息[①]；以及title.gif，是页面顶端的标题条。

js目录是所有组成应用的JavaScript文件所在的地方。其中有些，比如builder.js、controls.js、dragdrop.js、effects.js、prototype.js、scriptaculous.js和slider.js都是script.aculo.us库的文件，我们不会在这儿查阅这些文件。[②]

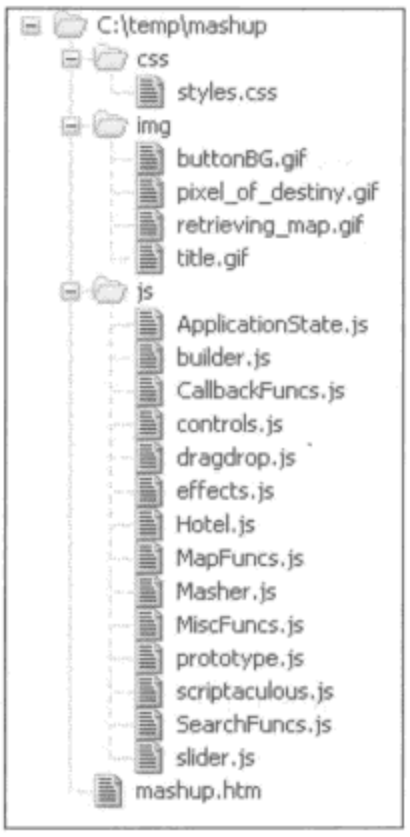


图5-3 混搭应用的目录结构

其他的文件组成混搭应用自身，从ApplicationState.js开始，它定义了一个叫ApplicationState的类。（我敢打赌你能猜出它的用途！）CallbackFuncs.js包含由Yahoo和Google的Web服务“回调”的函数。Hotel.js定义一个Hotel类，用于描述酒店。MapFuncs.js包含处理指定酒店的地图的函数。Masher.js包含那些和Google以及Yahoo的Web服务进行通讯的函数。MiscFuncs.js包含杂项函数，还有什么？最后，SearchFuncs.js包含执行酒店搜索的函数。

既然我们完成了基本介绍，那就开始看看具体的代码！

5.7.1 编写styles.css

styles.css文件在代码清单5-1中显示，是用于定义此应用的唯一的一个样式表文件。

① 其实就是一个等待状态图。——译者注

② 看看这些好代码总是有用且值得进行的练习，所以我强烈建议你至少花几分钟去看看script.aculo.us是如何运转的。理解我们的混搭并不需要了解它，不过你肯定能学到一些技巧。

代码清单5-1　styles.css文件

```
/* Style applied to everything on the page. */
* {
  font-size        : 10pt;
  font-weight      : bold;
  font-family      : arial;
}
/* Style applied to the body of the document. */
.cssBody {
  background-color : #ffffff;
}


/* Style applied to the left and right sections. */
.cssSectionBorder {
  border           : 2px solid #000000;
}


/* Style applied to the search results div. */
.cssSearchResults {
  width            : 100%;
  height           : 400px;
  overflow         : scroll;
}

/* Style applied to the popup area where hotel info is displayed. */
.hotelInfo {
  width            : 98%;
  height           : 200px;
  overflow         : scroll;
  background-color : #eaeaea;
}


/* Style applied to buttons. */
.cssButton {
  width            : 40px;
  border-color     : #ffffff;
  background       : url(../img/buttonBG.gif);
  color            : #000000;
}


/* Style applied to buttons when they are hovered over. */
.cssButtonOver {
  width            : 40px;
  border-color     : #ff0000;
  background       : url(../img/buttonBG.gif);
  color            : #ff0000;
}
```

第一个样式使用通配符应用于文档里面的所有元素的样式。这是一个包含所有东西的有用的技巧，虽然有些元素在使用全局样式的时候通常都有问题，比如表的单元格在某些浏览器里，级联的表单元可能无法获取样式信息。

对于页面体，我们用cssBody类应用白色的背景。cssSectionBorder应用于左边的搜索结果和右边的地图。cssSearchResults类给搜索结果定义样式，以保证它们可以正确放进页面左边的结构里，并且保证结果在比区域大的时候可以滚动（overflow:scroll）。

hotelInfo类给特定的酒店下头出现的信息做样式。在这个类里面，我们定义它不是在整个它所在的内容区里面伸展，这样它自己的滚动条就不会顶到搜索结果的滚动条上。

cssButton类用于给我们的按钮做样式。我们用它声明了一个背景图片，并且确保每个按钮都有一样的大小。背景图片设计成可以适用于任何风格的任意大小的按钮，给按钮金属质感的外观。这么做只是为了美感，但也让按钮有了一个漂亮的、唯一的外观。cssButtonOver类实际上是一样的，只是文本的颜色是红的，并且有个边框，用于在鼠标放在按钮上的时候给它一个动态的视觉。

简而言之，这个样式表相当简单，除了按钮之外没什么有趣的东西，我认为按钮的确是精心雕琢过的。

5.7.2 编写mashup.htm

mashup.htm文件，如代码清单5-2所示，是这个应用的起点，这个文件定义了基本的页面布局。

代码清单5-2 mashup.htm文件

```
<html>
  <head>

    <title>The Monster Mash</title>

    <link rel="StyleSheet" href="css/styles.css" type="text/css">

    <script src="js/prototype.js" type="text/javascript"></script>
    <script src="js/scriptaculous.js" type="text/javascript"></script>

    <script src="js/ApplicationState.js" type="text/javascript"></script>
    <script src="js/Masher.js" type="text/javascript"></script>
    <script src="js/Hotel.js" type="text/javascript"></script>
    <script src="js/CallbackFuncs.js" type="text/javascript"></script>
    <script src="js/SearchFuncs.js" type="text/javascript"></script>
    <script src="js/MapFuncs.js" type="text/javascript"></script>
    <script src="js/MiscFuncs.js" type="text/javascript"></script>
  <script>
    masher = new Masher();
    appState = new ApplicationState();
  </script>

</head>
```

```
<body class="cssBody">

  <center>
    <img src="img/title.gif" vspace="5">
    <br>
    <form onSubmit="search();return false;">
    Zip Code:
     
    <input type="text" id="zipCodeField" value="">
     
    <input type="submit" value="Look Up Hotels"
      class="cssButton" style="width:120px;"
      onMouseOver="this.className='cssButtonOver';"
      onMouseOut="this.className='cssButton';">
    </form>
    <br>
  </center>

  <table border="0" width="100%" cellpadding="4" cellspacing="0">
    <tr>

      <td align="left" height="420" class="cssSectionBorder">
        <img width="1" height="1" src="img/pixel_of_destiny.gif">
        <center>
          <div id="pleaseWait" style="display:none;">
            Please Wait, Retrieving Data...
          </div>
        </center>
        <div id="searchResults" class="cssSearchResults" style="display:none;">
        </div>
      </td>

      <td width="50" align="center">
        <input type="button" value="1" onClick="zoomMap(1);" id="zoomButton1"
          class="cssButton"
          onMouseOver="this.className='cssButtonOver';"
          onMouseOut="this.className='cssButton';"><br>
        <input type="button" value="2" onClick="zoomMap(2);" id="zoomButton2"
          class="cssButton"
          onMouseOver="this.className='cssButtonOver';"
          onMouseOut="this.className='cssButton';"><br>
<input type="button" value="3" onClick="zoomMap(3);" id="zoomButton3"
  class="cssButton"
  onMouseOver="this.className='cssButtonOver';"
  onMouseOut="this.className='cssButton';"><br>
<input type="button" value="4" onClick="zoomMap(4);" id="zoomButton4"
  class="cssButton"
  onMouseOver="this.className='cssButtonOver';"
  onMouseOut="this.className='cssButton';"><br>
```

```
<input type="button" value="5" onClick="zoomMap(5);" id="zoomButton5"
  class="cssButton"
  onMouseOver="this.className='cssButtonOver';"
  onMouseOut="this.className='cssButton';"><br>
<input type="button" value="6" onClick="zoomMap(6);" id="zoomButton6"
  class="cssButton"
  onMouseOver="this.className='cssButtonOver';"
  onMouseOut="this.className='cssButton';"><br>
<input type="button" value="7" onClick="zoomMap(7);" id="zoomButton7"
  class="cssButton"
  onMouseOver="this.className='cssButtonOver';"
  onMouseOut="this.className='cssButton';"><br>
<input type="button" value="8" onClick="zoomMap(8);" id="zoomButton8"
  class="cssButton"
  onMouseOver="this.className='cssButtonOver';"
  onMouseOut="this.className='cssButton';"><br>
<input type="button" value="9" onClick="zoomMap(9);" id="zoomButton9"
  class="cssButton"
  onMouseOver="this.className='cssButtonOver';"
  onMouseOut="this.className='cssButton';"><br>
<input type="button" value="10" onClick="zoomMap(10);" id="zoomButton10"
  class="cssButton"
  onMouseOver="this.className='cssButtonOver';"
  onMouseOut="this.className='cssButton';"><br>
<input type="button" value="11" onClick="zoomMap(11);" id="zoomButton11"
  class="cssButton"
  onMouseOver="this.className='cssButtonOver';"
  onMouseOut="this.className='cssButton';"><br>
<input type="button" value="12" onClick="zoomMap(12);" id="zoomButton12"
  class="cssButton"
  onMouseOver="this.className='cssButtonOver';"
  onMouseOut="this.className='cssButton';">
</td>

<td align="center" width="540" class="cssSectionBorder">
  <img id="mapFiller" style="display:block;" width="520" height="400"
    src="img/pixel_of_destiny.gif">
  <img id="map" style="display:none;" width="520" height="400"
    src="img/pixel_of_destiny.gif">
      </td>

    </tr>
  </table>

</body>

</html>
```

如你所见，mashup.htm以一系列JavaScript文件输入开头。头两个文件，prototype.js和scriptaculous.js是script.aculo.us运转需要的文件。scriptaculous.js负责包含它依赖的其他的JavaScript文件，比如

builder.js和effects.js。剩下的输入文件都是组成应用自身的源代码。

跟在JavaScript输入后头的是很小的一段JavaScript，负责创建Masher和ApplicationState对象实例，我们一会儿讨论它们。这一小段脚本包括文档的<head>。

继续看看<body>，一开始是标题图片，然后是一个你输入需要搜索的邮政编码的文本区域，然后是搜索按钮。这些都是表单的一部分，不过，具有讽刺意味的是，表单之所以需要，只是因为我们需要允许Enter键可以用于Firefox（在IE里，你可以省略<form>，只用简单的<input type="button">即可）。这就是为什么我们在表单的onSubmit时间上调用search()，但仍然返回false，这样就终止了表单的提交。这是Firefox比IE要费劲的少数几个情况之一！

然后我们就看到用于定义屏幕两半的表的开头，两半屏幕分别是左边的搜索结果和右边的地图，中间用地图的缩放按钮分隔。

表的第一列包含两个 元素。第一个包含在处理搜索的时候看到的“请等待（Please Wait）”信息。初始化的时候它是隐藏的。下一个 包含搜索结果。也要注意，它在开始的时候是隐藏的。但是更重要的是，样式是内联的方式声明的，而不是在外头的样式表里。这不可忽视！实际上，这个 准备有一些script.aculo.us特效，为了让特效能运转，我们必须内联一些样式信息。基于同样的原因，在页面的其他一两个位置也是如此。对于将通过script.aculo.us隐藏的元素，你需要在元素上内联地设置显示的style属性来定义它们初始的可视性。就那么简单。

说明 script.aculo.us特效要求初始的可视性声明为内联的原因是script.aculo.us是基于原型库的，然后它用了原型提供的show()函数做元素的可视性切换。这个函数的工作原理是设置元素的display样式属性为一个空字符串（换句话说，未定义）。这儿的概念是给指定的属性设置成默认值（空字符串），意思是该元素可见。不过，也可能会有问题：如果这个属性在CSS里定义为“高于”元素级别。还记得原型是覆盖元素级别的东西的吧？在这种情况下，原型会寻找元素级别的未定义值，即使你的样式表可能是声明为display:none的，但结果还是看上去什么都没发生一样。

跟在这些东西后面的是表里面的第二列，它是地图的缩放按钮的位置。在里面是一系列的12个按钮。它们都共用同样的基本框架：附着在上头的onClick事件处理函数调用zoomMap()函数，这个函数得到对应每个按钮的缩放级别。onMouseOver和onMouseOut事件处理函数也基本和我们前面在搜索按钮上看到的一样。这儿确实没什么好玩的东西。

最后，我们看到在你选定一个酒店查看的时候包含地图的表，或者更准确说，是会包含地图的表。它需要ID mapFiller是因为我们需要在IE里维持一个一致的大小，要不然就很难看了。ID map标签显示地图或者是“请等待（Please Wait）”信息，它自己也是一个图片。这个标签的src会在不同场合更新为指向请等待图片或者是Yahoo生成的地图。这两个<img>标签根据情况轮流隐藏和显示。

这些就是所有HTML标记的说明！

5.7.3　编写ApplicationState.js

毫无意外的，ApplicationState.js文件包括那个ApplicationState类。虽然这个类很简单，但它扮演了一个十分重要的角色，因为它保存了那些在整个应用程序中都将使用到的值。图5-4是

ApplicationState类的UML类图。

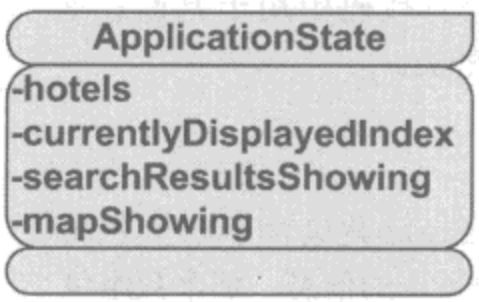


图5-4 ApplicationStat类的UML图

如你在代码清单5-3中看到的，ApplicationState类总共包含了4个字段。

代码清单5-3 ApplicationState.js文件

```
/**
 * This class stores information about the state of the application.
 */
function ApplicationState() {


  /**
   * This is an array of hotels returned from a search. */
  this.hotels = new Array();
  /**
   * This is the index into the hotels array of the hotel that is currently
   * being viewed, i.e., that has info showing and that has a map showing.
   */
  this.currentlyDisplayedIndex = -1;


  /**
   * This is a flag that indicates whether search results are currently
   * showing or not.
   */
  this.searchResultsShowing = false;


  /**
   * This is a flag that indicates whether a map is currently showing or not.
   */
  this.mapShowing = false;


} // End ApplicationState.
```

hotels数组是一个Hotel对象的集合，在执行查询的时候填充。currentlyDisplayedIndex字段存储了当前浏览的那个酒店在hotels数组中的下标。searchResultsShowing字段是个简单的标志位，它告诉我们搜索结果是否当前可见。最后，mapShowing字段是另一个标志位，它告诉我们是否正在展示一个

地图。

你将看到这些字段在后面的所有代码中用于判断在某个点应该做什么，不过大体上说，ApplicationState还是一个非常简单的类。

5.7.4　编写Hotel.js

如你刚看到过了，ApplicationState类包含了一个名叫hotels的数组，它是当前查询得到的酒店的信息。那个数组包含了Hotel对象，分别对应每一个酒店。在图5-5中，你可以看到Hotel类的UML类图。

图5-5　Hotel类的UML图表

像ApplicationState一样，Hotel类并没多少东西，事实上，比ApplicationState类还要少！总共只有3个字段，就提供给我们了在这个应用程序中所需要的关于酒店的所有信息，如代码清单5-4所示。

代码清单5-4　Hotel.js文件

```
/**
 * This class represents a hotel as returned by the Google web service.
 */
function Hotel() {

    /**
     * The name of the hotel.
     */
    this.name = "";


    /**
     * The location (address) of the hotel.
     */
    this.location = "";


    /**
     * The description of the hotel.
     */
    this.description = "";


} // End Hotel.
```

name字段是酒店的名字，就是在结果列表中看到的。location字段是通过Web服务返回的酒店的位置。description字段是那个酒店的完整描述。在一些情况下，这个描述也会包含酒店的名字和位置，所以可能会有一些展示上的冗余。

5.7.5 编写SearchFuncs.js

这个应用程序中的所有东西都是以查询开始的，所以很有理由从代码的最开始看起，也就是由搜索触发的真正行为的代码看起。当点击搜索按钮的时候，会调用search()函数。代码清单5-5展示了这个函数。

代码清单5-5 SearchFuncs.js函数

```
/**
 * Start execution of a search by ZIP code with Google Base.
 */
function search() {

  // Reset things that need to be reset.
  resetZoomButtons();
  appState.currentlyDisplayedIndex = -1;

  // Hide information for all hotels.
  for (var i = 0; i < appState.hotels.length; i++) {
    $("hotelInfo" + i).style.display = "none";
  }

  // If there are search results showing, hide them.
  if (appState.searchResultsShowing) {
    new Effect.BlindUp("searchResults",
      {
        afterFinish : function() {
          // Now do the actual search.
          searchPart2();
        }
      }
    );
    // If a map and hotel info are showing, hide them too.
    if (appState.mapShowing) {
      new Effect.Puff("map",
        {
          afterFinish : function() {
            $("map").style.display = "none";
            $("mapFiller").style.display = "block";
          }
        }
      );
    }

  } else {

    // No results currently showing, so just do the search.
    searchPart2();
```

```
    }

  } // End search().


  /**
   * Continue search after effect.
   */
  function searchPart2() {

    // Show Please Wait.
    $("searchResults").style.display = "none";
    $("pleaseWait").style.display = "block";

    // Do search.
    var zipCode = $("zipCodeField").value;
    masher.doRequest(masher.googleURL,
      {
        "bq" : "%5blocation:@%22" + escape(zipCode) + "%22%2b50mi%5d",
        "alt" : "json-in-script",
        "callback" : "googleCallback"
      }
    );

  } // End searchPart2().
```

首先，在我们继续之前，注意一下$()函数的使用（$是一个合法的JavaScript函数名字，即使它看上去有些不同寻常的地方）。你可能会想起在前面的章节中说的，这个函数是从Prototype来的，Prototype是script.aculo.us的基础。它是document.getElementById()的一种缩写，但$()提供了一些附加功能，例如返回多个元素的能力。在我们查看代码的时候，你会在很多地方看到这样的东西。

继续，代码作了一些重置，把应用程序设置到一个适于查询的状态。最重要的，这意味着重置了缩放按钮，使它们都处于未被选中的状态，并且把currentlyDisplayedIndex的索引值置为–1，意思是当前没有正在浏览的酒店信息。下一个重置是要确定没有正显示的酒店信息。为了完成这个工作，会遍历当前屏幕中所有可见的酒店列表，如果有的话。列表中每一个在酒店下面的信息 都通过设置display样式属性为none隐藏起来。

然后，还要做一个检查来看看当前是否还有被展示的搜索结果。如果有，我们就需要隐藏搜索结果和地图。为了隐藏搜索结果，我们收缩那个区域。script.aculo.us提供了一个BlindUp特效来收缩一个元素，就如同降下百叶窗一样。这段代码告诉script.aculo.us来收缩搜索结果：

```
new Effect.BlindUp("searchResults",
  {
    afterFinish : function() {
      // Now do the actual search.
      searchPart2();
    }
  }
);
```

在完成之前，我们还要说说，将要调用searchPart2()函数（一会儿你就会看到）。

然后，如果还有地图正在展示的话，我们还需要再一次隐藏地图。script.aculo.us为我们提供了一个漂亮的效果。Puff效果将一个元素扩展并同时淡出，就像一股烟一样。这里就是让一个ID为map的图像“消散”掉所需的所有代码：

```
new Effect.Puff("map",
  {
    afterFinish : function() {
      $("map").style.display = "none";
      $("mapFiller").style.display = "block";
    }
  }
);
```

当这个效果完成时，我们隐藏了地图的图像并显示填充图像，这样做只是为了确认单元格元素没有坍塌，并且我们能得到一个边框（它在刚才那个效果中消失）。

现在，如果没有需要展示的搜索结果了，那么就调用searchPart2()，也没有什么需要被隐藏的。

所以，我之前谈过的searchPart2()究竟是什么？它只不过是那个search()函数的一个延续罢了。这个函数先隐藏了搜索结果 ，并在它的位置显示了请等待（Please Wait） 。然后，它抓取用户输入的邮政编码（ZIP），最后，它调用Masher对象的doRequest()函数，传给它Google服务的URL以及那个服务所需要的参数。这些参数分别是：bq，定义了我们想要执行的查询；alt，它告诉我们，想要JSON封装在一个JavaScript调用中返回给我们的那个服务；最后是callback，它指定了将要被调用的那个回调函数的名字。注意bq的值是根据查询服务的规则来编码，并且转义了邮政编码（ZIP），让它可以安全地包含在后面的URL中。

很自然，下一步是看看Masher对象到底是什么！

5.7.6 编写Masher.js

简单地说，Masher对象或者说Masher类是在页面载入时创建的实例，负责与基于JSON的Web API进行通信。它是以一种比较通用的方式写的，所以我们只做少量修改就可以用它访问其他的服务。图5-6展示了这个类的非常简单的UML图。

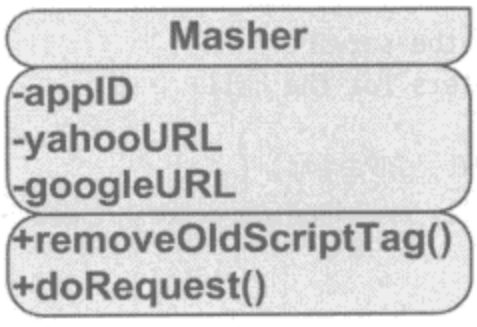


图5-6 Masher类的UML图表

appID字段是访问Yahoo的API时需要的应用程序ID。yahooURL字段是Yahoo Map地图服务的URL。最后，googleURL字段是Google Base 酒店查询服务的URL。

第一个方法，removeOldScriptTag()，是一个用于移除一个动态添加的<script>标签的简单函数。

这样我们在调用服务的时候，就不会随着新标签的添加而导致内存的增长。代码清单5-6展示了这个方法的代码以及Masher类的其余部分。

代码清单5-6　Masher.js文件

```
/**
 * This class allows us to interact with JSON-based web services in the style
 * of Yahoo, that is, JSON wrapped in a function call.
 */
function Masher() {


  // Parameters for Yahoo! services.
  this.appID = "xxxxx";
  this.yahooURL = "http://api.local.yahoo.com/MapsService/V1/mapImage";

  // Parameters for Google services.
  this.googleURL = "http://www.google.com/base/feeds/snippets/-/hotels";


  /**
   * Removes an old script tag used to retrieve JSON.
   */
  this.removeOldScriptTag = function() {
    var scriptTag = $("jsonScriptTag");
      if(scriptTag) {
        scriptTag.parentNode.removeChild(scriptTag);
    }

} // End removeOldScriptTag().


/**
 * Perform an Ajax request using the dynamic script tag approach.
 *
 * @param inURL    The URL of the service.
 * @param inParams The parameters for the call.
 */
this.doRequest = function(inURL, inParams) {
  // First, to avoid continually building up memory, remove any old script
  // tag out there.
  this.removeOldScriptTag();

  // Now build up a query string using the passed in parameters
  var queryString = "";
  for (param in inParams) {
    var paramVal = inParams[param];
    if (queryString == "") {
```

```
        queryString += "?";
      } else {
        queryString += "&";
      }
      queryString += param + "=" + paramVal;
    }

    // Now add a new script tag with the appropriate URL.
    var scriptTag = document.createElement("script");
    scriptTag.setAttribute("id", "jsonScriptTag");
    scriptTag.setAttribute("src", inURL + queryString);
    scriptTag.setAttribute("type", "text/javascript");
    var headTag = document.getElementsByTagName("head").item(0);
    headTag.appendChild(scriptTag);

  } // End doRequest().


} // End Masher.
```

如你在SearchFuncs.js代码中看到的，当需要一个API时，它通过使用在Masher类中的doRequest()方法来调用。这个函数开始先调用removeOldScriptTag()，我们前面说过，用以确定内存并没有不停的被耗费。下一步，在传入的那些参数的基础上创建一个查询字符串。我们将每一个参数都添加到查询字符串。注意在这里并没有转义字符串值，因为它认为传入的所有参数都已经进行了转义（这样避免了潜在的双重转义错误）。邮政编码（ZIP）是传入的唯一一个用户输入值，你可以回想一下，它是在SearchFuncs.js的代码中进行转义的，所以这里就没有问题了。

一旦创建了查询字符串，本质上来说，我们就有了与你已经看到过的添加一个<script>标签相同的代码了：

```
var scriptTag = document.createElement("script");
scriptTag.setAttribute("id", "jsonScriptTag");
scriptTag.setAttribute("src", inURL + queryString);
scriptTag.setAttribute("type", "text/javascript");
var headTag = document.getElementsByTagName("head").item(0);
headTag.appendChild(scriptTag);
```

注意设置在这个标签上的ID要与在removeOldScriptTag()中被移除的那个一样。因此，在页面上，无论何时，都有唯一的一个动态<script>标签。除此之外，这个代码与你之前看到的相同，而且我相信你记得它是如何工作的！

并没有太多东西需要与这些服务交互，就如这个类说明的那样清楚。只剩下一件事情了：回调函数，当从服务返回应答时调用的函数。那正好是我们下一个要看的！

5.7.7 编写CallbackFuncs.js

正如你已经看到的，当Google或Yahoo的API返回了它们的响应时，应答的形式是传递了JSON参数的一个JavaScript函数（更加准确一点，传递的是从JSON解析而来的对象）。回调函数，googleCallback()和yahooCallback()，如代码清单5-7所示。

代码清单5-7　CallbackFuncs.js文件

```
/**
 * This is the function that is called when a Google service returns.
 *
 * @param inJSON The JSON object returned by the service.
 */
function googleCallback(inJSON) {

  var htmlOut = "";
  appState.hotels = new Array();

  // Iterate over the list of hotels.
  for (var i = 0; i < inJSON.feed.openSearch$itemsPerPage.$t; i++) {

    // Construct markup for the list for each hotel, including its information.
    var entry = inJSON.feed.entry[i];
    var hotel = new Hotel();
    hotel.name = entry.title.$t;
    hotel.location = entry.g$location.$t;
    hotel.description = entry.content.$t;
    appState.hotels.push(hotel);
    htmlOut += "<span onClick=\"" +
      "getMap('" + entry.g$location.$t + "');" +
      "showInfo(" + i + ");\" " +
      "onMouseOver=\"this.style.backgroundColor='#ffff00';" +
      "this.style.cursor='pointer';\" " +
      "onMouseOut=\"this.style.backgroundColor='';" +
      "this.style.cursor='';\"" +
      ">";
    htmlOut += entry.title.$t;
    htmlOut += "</span>";    htmlOut += "<div id=\"hotelInfo" + i + "\" style=\" ➥
display:none;\" ";
    htmlOut += "class=\"hotelInfo\">";
    htmlOut += "</div>";
    htmlOut += "<br><br>";

  }

  // Put the generated markup in the search results list div.
  $("searchResults").innerHTML = htmlOut;
  $("pleaseWait").style.display = "none";

  // Ask script.aculo.us to show the search results.
  new Effect.BlindDown("searchResults");

  // Set flags so we know what state the application is in.
  appState.searchResultsShowing = true;
  appState.mapShowing = false;
```

```
} // End googleCallback().


/**
 * This is the function that is called when a Yahoo service returns.
 *
 * @param inJSON The JSON object returned by the service.
 */
function yahooCallback(inJSON) {

  if (inJSON.Error) {
    var msg = "An error occurred retrieving map: ";
    if (inJSON.Error.Message) {
      msg += inJSON.Error.Message;
    }
    appState.mapShowing = false;
    $("map").src = "img/pixel_of_destiny.gif";
    alert(msg);
  } else {
    $("map").src = inJSON.ResultSet.Result;
  }

} // End yahooCallback().
```

googleCallback()处理的第一个重要的任务是，清空任何存储在ApplicationState实例中的已存在的酒店，方法是给hotels字段分配一个新的数组。这些做完之后，就是时候遍历那些返回的酒店了。

要做这个遍历，我们需要知道共返回了多少条目，这可以在JSON对象的feed.openSearch$itemsPerPage.$t属性中找到。拿到那个值之后我们就可以开始循环了。每一次遍历里都通过访问feed.entry数组来获得一个条目，也就是一个酒店，那里我们根据循环计数器，在结尾处附加一个索引值，也就是feed.entry[i]。

只要有了条目，就只剩下为结果列表构建HTML的事了。每一个条目被一个包围，还有一些事件处理函数，处理用户把鼠标放在上面（鼠标的黄色背景）的事件，还有onClick处理函数负责为那个条目（酒店）载入地图以及它的扩展信息。的内容是酒店的名字（通过entry.title.$t 属性获得），它下面有一个 ，那里将展示扩展信息。

同时，我们使用酒店名字(entry.title.$t)、位置(entry.g$location.$t)和描述信息(entry.content.$t)来填充一个新的Hotel对象的实例。然后把这个对象放到ApplicationState的hotels数组中。

这里有两个值得注意的有趣事情。第一，描述信息本身并不包含在这个HTML标签中，而它是在后面动态添加的。我承认这么做除了为了演示一下这是可能的之外，并没有什么其他真正的原因。第二，注意那个接在条目的上的onClick处理函数，它是负责通过getMap()调用来找回酒店的地图，以及使用showInfo()展示它的信息。

Yahoo回调函数比Google回调函数做的工作还少。它的唯一工作就是更新带有地图ID的<img>的src属性，使其指向由Yahoo服务返回的URL。这就是所有用来展现一个地图的工作！不过，动态<script>的Ajax用法也有一个很恶心的弱点，就是错误处理。

通常，错误都是相当容易被处理的，如在yahooCallback()函数中的例子一样。如果在返回的JSON

中发现了一个Error属性，我们就可以捕获那个消息并把它展现给用户。还需要重置mapShowing标签并确定我们展现了空白的“目标像素”。然而，这只包括了一种潜在的错误类型。例如如果发生了网络失败，并没有返回响应，就没有你可以使用的错误处理。应用程序就只是表现为不工作了。

再者，如果服务返回了一个非法数据结构，它也是不能被处理的。Yahoo的函数就有这种问题。比如，尝试邮政编码94505，并选择任何开头的几个酒店。你都会注意到会出现“请等待，地图返回中”的信息，但是并没有出现地图。如果你正在Firefox中运行并打开了Firebug，你会发现，从服务返回的是非法数据，解析时引发了一个JavaScript错误。糟糕的是，我们还没法解决这个问题。这与引用一个非法的.js文件并在解析时引发JavaScript错误非常相似，但在这里更糟糕，因为你无法自己修正这个错误。这绝对是在使用混搭时应该注意的一个问题，更准确一点地说，就是动态<script>标签技术。世上没有免费的午餐！

5.7.8　编写MapFuncs.js

MapFuncs.js文件包含4个函数，处理地图或在地图上执行操作，比如放大或缩小。代码清单5-8展示了这个文件的代码。

代码清单5-8　MapFuncs.js文件

```
/**
 * Retrieve map for address of selected hotel with Yahoo Maps.
 *
 * @param inLocation The address of the hotel to get a map for.
 * @param inZoom     The zoom level, 1-12, of the map.
 */
function getMap(inLocation, inZoom) {

  // The default zoom level is 6.
  if (!inZoom) {
    inZoom = 6;
  }

  // Show the Please Wait message while we request the map.
  $("map").src = "img/retrieving_map.gif";
  $("map").style.display = "block";
  $("mapFiller").style.display = "none";

  // Ask the masher to make the request for us.
  masher.doRequest(masher.yahooURL,
    {
      "appid" : masher.appID,
      "location" : escape(inLocation),
      "image_type" : "gif",
      "output" : "json",
      "width" : "520",
      "height" : "400",
      "zoom" : inZoom,
```

```
      "callback" : "yahooCallback"
    }
  );

  // Set state and reset the zoom buttons, and highlight the current zoom
  // level.
  appState.mapShowing = true;
  resetZoomButtons();
  highlightZoomButton(inZoom);

} // End getMap().


/**
 * Zoom the map according to the zoom button clicked.
 *
 * @param inZoom The zoom level, 1-12, to zoom the map to.
 */
function zoomMap(inZoom) {

  // Obviously this only does something if a map is showing.
  if (appState.mapShowing) {
    var hotel = appState.hotels[appState.currentlyDisplayedIndex];
    getMap(hotel.location, inZoom);
  }
} // End zoomMap().


/**
 * Reset all the zoom buttons so none are highlighted.
 */
function resetZoomButtons() {

  for (var i = 1; i < 13; i++) {
    $("zoomButton" + i).style.fontSize = "10pt"
  }

} // End resetZoomButtons().


/**
 * Highlight the specified zoom button.
 *
 * @param inZoom The zoom level, 1-12, of the button to highlight.
 */
function highlightZoomButton(inZoom) {

  $("zoomButton" + inZoom).style.fontSize = "16pt"

} // End highlightZoomButton().
```

第一个函数，getMap()，当用户点击搜索结果中的某个酒店连接时调用它。点击缩放按钮的时候也调用它。由于这个双重目的，它接收两个参数：inLocation和inZoom。inLocation，你可能已经猜到，就是我们需要的地图的地址。它与Hotel对象的location字段相互关联。inZoom是指缩放级别。因为是Yahoo生成那个地图，所以需要告诉它我们需要什么样的缩放级别。然而，当第一次点击酒店、第一次返回地图时，并不会传入inZoom变量。所以，在这段代码中你看到的第一件事，就是检测是否传入inZoom。如果没有，就使用默认级别6。

在那之后是3行代码，负责在地图区域展示“Please Wait（请等待）”消息。只要把map的<img>标签指向Please Wait图像就显示它，并隐藏mapFiller就好了。

然后我们让Masher实例来获取地图。就这个API调用而言，我们需要传入在注册yahoo服务时所获得的appid。显然，还需要传入location，而且需要声明我们要返回一个GIF（那个image_type参数）。我们声明希望输出的是一个封装在JavaScript函数调用中的JSON（通过输出output参数），这个函数使用callback参数指定。我们需要一个适合地图区域大小的地图，地图区域是520×400像素，因此我们设置适当的width和height参数。最后通过命名得很恰当的zoom参数传递所需的放大级别。

还有最后一些需要关心的细节，也就是设置标志位来表明地图是否正被展示，把缩放按钮重置为默认状态（当前没有任何缩放按钮比其他的大），然后把那个缩放级别按钮设置为当前的。

API调用返回之后，就会落到yahooCallback()里，那个你已经看过的函数。

在点击一个地图缩放按钮时会调用zoomMap()函数。它先做一个简单的检查保证有正在展示的地图；否则，当按钮被点击时什么都不会发生。如果有地图正在被展示，zoomMap()获取对应正在被浏览的那个酒店的Hotel对象，然后用Hotel对象和传入的新的缩放级别作参数传递给getMap()函数。

接着的resetZoomButtons()函数执行了简单地重置所有放大按钮为默认字体大小的工作，效果是没有表示当前的缩放级别的按钮。这仅是在12个按钮中的循环，为每一个构建适当的DOM ID，并在其上设置fontSize样式属性。①

最后是highlightZoomButton()函数。这是当用户点击了一个按钮时调用的，并且，它的任务是将按钮设为“当前”，意思就是，字体更大些。这也是一个对fontSize样式属性的直接操作。

5.7.9　编写MiscFuncs.js

现在我们来到了这个项目中的最后一个源文件，MiscFuncs.js。这个文件包含了一个单独的函数，但它是非常重要的：showInfo()，它是用来为一个点击的酒店展现扩展信息的。代码清单5-9展示了这个代码。

代码清单5-9　MiscFuncs.js文件

```
/**
 * Function to show extended hotel information.
 *
 * @param inIndex Index into the array of hotels in appState.
 */
```

① 我们当然可以通过把按钮切换为不同样式类的方式来实现，而且很多时候这样做更好。不过我想显示一些做同一件事的不同方法，当然了，直接操作风格属性是其中方法之一。

```
function showInfo(inIndex) {

  // Trivial rejection: are we already showing the requested hotel?
  if (inIndex == appState.currentlyDisplayedIndex) {
    return;
  }

  // Shrink the information for the currently showing hotel.
  if (appState.currentlyDisplayedIndex != -1) {
    new Effect.Shrink("hotelInfo" +
      appState.currentlyDisplayedIndex,{duration:1.0});
  }
  // Update application state and insert the new hotel information.
  appState.currentlyDisplayedIndex = inIndex;
  var hotel = appState.hotels[inIndex];
  var htmlOut = "<br>" + hotel.location + "<br><br>";
  htmlOut += hotel.description;
  $("hotelInfo" + inIndex).innerHTML = htmlOut;

  // And finally, have the new info "grow" into view.
  new Effect.Grow("hotelInfo" + inIndex,{duration:1.0});

} // End showInfo().
```

这个函数接受一个参数，是指向需要获取信息的hotels数组中某个元素的下标。所以，首先是一个快速的检查看看刚被点击的那个酒店是不是已经显示了相关信息。如果是，那我们就结束——这儿有毛病！

假设是一个不同的酒店，下一步就检查是否正在显示某个酒店的相关信息，也就是说，在appState中的currentDisplayedIndex字段值不是–1。在那种情况下，我们用script.aculo.us使用Shrink特效把那个信息从视野中收缩出去。

然后代码记录了那个现在要被展示的下标值，获取合适的Hotel对象，并使用它的位置和描述信息创建HTML标签。创建了HTML标签之后，就直接把它插入到位于被点击的酒店下面的 中， 的ID是hotelInfo，后面跟着下标。最后，我们请求script.aculo.us使用Grow特效来把这个新的信息在视野中展开。

注意Shrink和Grow会同时发生，因为，当新的Shrink特效实例化时，就会启动一个计时器来完成这个特效，但是在启动计时器和新的Grow特效实例之间的代码会在Shrink特效结束之前发生。这样，就会有两个计时器，分别对应每一个特效，然后它们看起来是同时发生。我们是有意这样设计的，因为它使整个转变过程看起来更酷。当然， script.aculo.us 为我们处理了所有繁琐的细节。

最后，我们已经到了最后了！我希望你可以享受看到一个真实的混搭的感觉，感谢Yahoo和Google以及其他的Web服务提供者，给我们在自己的应用程序中提供了主要构件。

5.8 练习

对于混搭，你可以继续向你自己的核心内容中添加特性，只要你可以找到适当的API！我把搜寻

API的工作留给你，不过这里还有一些有用的建议，可以试着玩玩：

- 添加为一个特定日期或日期范围获取天气预报的功能。如果你正打算安排一个假期，那么这就非常有用！
- 添加查找饭馆而不是酒店的功能（提示：Google也提供了这个功能）。
- 扩展地图功能。你可以看看Yahoo Maps提供的那些特性并复制它们，因为大多数功能（如果不是全部的话）都是从API开发而来的。

5.9　小结

本章涉及了一个当今最流行的热门词：混搭。你知道了如何绕开大多数Ajax实现里典型的同域限制的问题。我们介绍了一些公共API以及开发者如何使用它们，和应用程序如何与它们交互的内容。你还看到如何使用最简短的代码，利用script.aculo.us制作一些非常漂亮的小UI视觉糖。

总而言之，我们没费多大力气就创建了一个非常有用的小应用程序，当然，也是混搭的目标。同时，你还在过程中看到了一些优雅的JavaScript技巧，又扩展了一点点自己的思维工具箱！

第6章

不要只考虑眼前：客户端的持久对象

对于应用程序而言，有两种可以使用的数据存储类型：持久的和非持久的，或者说是临时的。持久存储是指在程序执行之间给程序提供的数据存储空间的机制，它通常是没有时间期限的（直到明确地从存储中删除为止）。临时存储是指只在程序实际执行时（或者是从那以后一段较短时间内）为保存数据提供空间的机制。通常认为数据库是一个持久存储机制，很明显RAM就不是。向硬盘写数据通常也是持久的，不过会话内存一般不是。术语持久的（durable）也常用来描述持久存储介质。

在Web应用程序中，服务器上的存储状态，不论是在数据库中还是在会话中都是非常普遍的。但是如果你在服务器上并没有一个持久存储机制，或可能由于某些原因不想用服务器端持久存储时，又应该怎样呢？有什么替代品呢？

在讨论一个纯粹的基于JavaScript的客户端应用程序里面的持久存储机制时，就只有非常少的几种可能的方案。[①]无所不在的cookie（在客户端为每个域存储的少量信息）是其中之一。另外一个是现在在大多数浏览器上可以使用的工具，叫做本地共享对象（或者叫Flash共享对象，甚至可以叫Flash cookie，这取决于你所阅读的文档）。这是由Adobe Flash浏览器插件提供的一个存储机制。

本地共享对象获得了相当广泛的应用，并且我们会把这种方法用在本章的项目中，它是一个简单的通讯录。虽然最大的卖点可能是实现这个客户端持久存储所使用的Dojo库，它让我们的生活变得更加轻松。那么，演出开始。

6.1 通讯录的需求和目标

在本章中，我们将创建一个通讯录应用程序，它是完全在客户端运行的。这样非常方便，因为这意味着它实际上可以运行在任何PC上，而不依赖于网络连接（虽然，反过来看，在不同的机器上也不能方便地共享联系数据信息）。很明显，这里需要持久存储，因为如果我们不能实际存储联系信息，这就将是一个非常没用的应用程序。

让我们列出一些在这个项目中将完成的关键部分，以及一些要尝试去实现的特性。

① Java小应用程序和ActiveX控制是其他的选择，但是通常我们认为它们完全是另外一类的可能了，因为，从某种意义上来说，它们可以看作是浏览器扩展（当然，ActiveX控制完全就是了，但是其实小应用程序也可以看作是此类）。不管怎么说，它们都需要HTML和JavaScript之外的东西，这些因素令它们属于其他范畴。

- 每一个联系信息都应该包括许多的数据。除了基本的姓名、电话号码和电子邮箱地址之外，我们还要允许大量的扩展信息，例如生日、配偶的名字、子女的名字等。不过，我们要把这些项尽可能地做成自由格式，这样用户可以依个人喜好来使用不同的数据字段。
- 我们将实现典型的字母选择器标签，使其可以更加容易地找到联系人信息。
- 我们将使用本地共享对象存储联系信息。
- 我们将使用Dojo来提供一些很酷的窗口小部件，把界面做得更加有趣有吸引力。我们还将使用Dojo来处理使用本地共享对象时的一些底层细节。

现在在头脑中已经有了一些目标，那么让我们先看看将在这个项目中使用的库吧。

6.2　Dojo特性

为了辅助创建通讯录应用程序，我们将使用非常流行的Dojo工具包。第2章曾简单地介绍了Dojo，现在让我们了解一些更详细的内容。

在它的官方主页，Dojo（http://dojotoolkit.org）介绍了它是什么。我看没有必要试着解释了，这里就是直接引用了。

Dojo是个开源的JavaScript工具包，它可以帮助你在短时间内建立严肃的应用程序。它填补了JavaScript和浏览器没有很好覆盖到的一些空白，并提供给你功能强大的、可移植的、轻量级的并且是经过测试的工具来构建动态界面。Dojo可以让你很快地原型化交互控件、动画转换以及创建带有功能最强大以及容易使用的封装的Ajax。这个功能是在轻量级打包系统的基础上实现的，所以你根本不需要再关注请求装载脚本文件的顺序。Dojo的包系统和可选的制作工具帮助你快速开发和透明地优化。

Dojo还封装了一个轻松使用的窗口小部件系统。从原型到部署，Dojo窗口小部件自始至终都是HTML和CSS。最好的是，因为Dojo以可移植的JavaScript为核心，所以窗口小部件可以在HTML、SVG以及其他任何主流的语言之间移植。Web是变化的，Dojo可以帮你一直站在前面。

Dojo使专业化的Web开发更迅速、更方便、更快捷。以上面顺序。

没错，总体上看就是这样！Dojo最近获得了不少拥趸，甚至还被集成到了一些很流行的框架里面去，比如OpenSymphony的WebWork（现在是Apache的Strus2，http://www.opensymphony.com/webwork 或者 http://struts.apache.org）。

Dojo是一个相当大的库，包含了无数的包和特性。Dojo并不像其他一些库那样只是关于Ajax的。它提供了一些函数扩展了JavaScript本身，还有一些常见的JavaScript应用程序功能代码。

然而，Dojo也有一个弱点：它现在确实还处于婴儿期。看一下在线文档，你就会意识到使用Dojo在很大程度上意味着要依靠自己的努力。在写这本书的时候，一些包还没有任何文档可参考。同时，例子也很少，而且计划的特性也还没有完全成熟。不过，相比几个月之前，在这方面Dojo已经有了飞速的发展。因此，如果你正考虑使用Dojo，就可以开始有一个相对舒适的环境来使用它了。而且Dojo邮件列表是非常活跃的，它有很多真的非常有帮助的人。任何好的开源项目都是由它的社区的本质和活跃程度来定义的，在这点上，Dojo可以打高分。

在本章中，我们将只使用Dojo众多包中的两个（UI小部件和存储功能）。表6-1展示了Dojo提供的一些包。注意，这只是一小部分，如果你的时间允许，可以去看看它还提供了什么。

表6-1 一些Dojo包

包	描 述
dojo.lang	工具例程，让JavaScript更加易用。包括许多用来操作JavaScript对象、检测数据类型等的函数
dojo.string	字符串操作函数，包括trim()、trimStart()、escape()等
dojo.logging	JavaScript 日志
dojo.profile	JavaScript代码配置
dojo.validate	数据验证函数，例如isNumber()、isText()、isValidDate()等
dojo.crypto	加密例程
dojo.storage	使用Flash的 cookie机制实现了一个持久的客户端缓存的代码。这种在客户端实现的效果类似于服务器端的HttpSession对象
dojo.widget	各种各样非常酷的GUI小部件（例如按钮、对话窗口、照片幻灯等）
dojo.collections	多种数据结构，像Dictionary、ArrayList、Set等

要开始使用Dojo，你有很多选择。Dojo出现了很多“版本”。因此，如果你只对Ajax功能感兴趣，就可以只下载那个IO版本。如果你只对GUI窗口小部件感兴趣，就去下载那个窗口小部件版本。还有一个叫做“洗涤槽”的版本，它包括所有Dojo提供的东西。对于本章的项目，我只使用最小的那个版本，但是我们会稍微谈及一些其他的。

在下载了合适的版本之后，所有需要做的就是给dojo.js文件写一个典型的JavaScrpt导入语句，像这样：

```
<script src="js/dojo.js"></script>
```

然后，导入的工作就完了。Dojo也提供了一个“导入”特性。也就是说，例如，如果你已经下载了IO版本，并决定使用事件系统，那么你可以这样做：

```
<script type="text/javascript">
    dojo.require("dojo.event.*");
</script>
```

然后Dojo就会载入所有依赖包，你就可以用了。你可以在任何时候导入任何想要的特性。换句话说就是，不需要在页面顶部给想要使用的包写一堆导入了，取而代之的是用Dojo导入的办法，就像Java一样，然后让Dojo去处理所有的细节问题（不过那个主要的dojo.js的导入还是需要的）。

6.2.1 Dojo和cookie

使用cookie非常简单，在第3章中当我们给函数创建jscript.storage包时其实已经看到了。Dojo提供了类似的功能，可能还有更多的一些功能。要了解Dojo提供的一些函数，仔细看看在http://dojotoolkit.org/api的在线API手册，它在短期内进步了很多，并且开始让Dojo可以更方便地使用了。图6-1展示了这个页面。

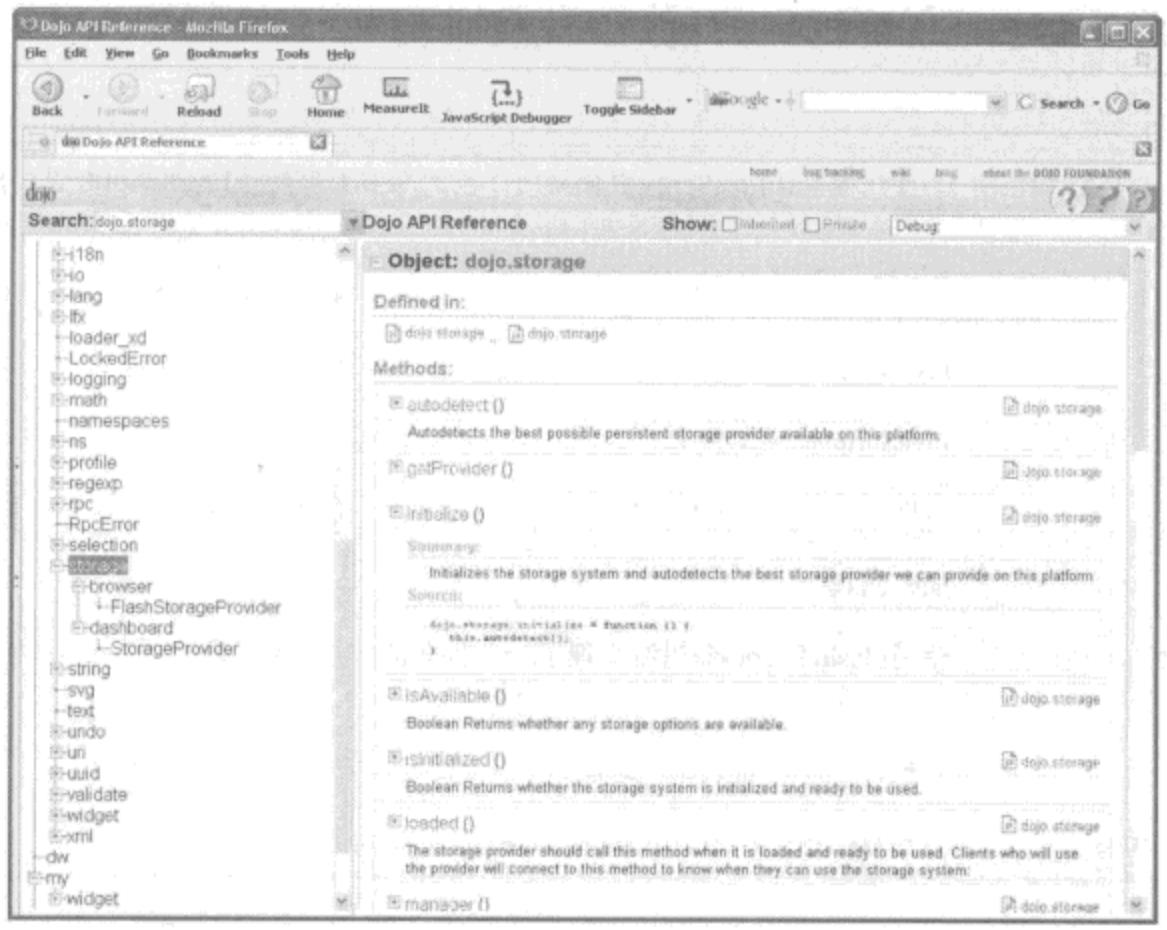


图6-1　Dojo API在线手册

你会在dojo.io.cookie包中找到cookie的相关函数（这个包可能看起来有些多余，因为还有一个叫做dojo.storage的包）。

举个例子，在Dojo中设置一个cookie，就照下面这样做：

```
dojo.io.cookie.setCookie("cookieName", "cookieValue");
```

看起来这和第3章中开发的setCookie()函数并没有太多不同。不过，Dojo在这个函数后面还额外添加了4个（可选的）参数。

- days：决定这个cookie在过期前的有效期是多少天。
- path：设置cookie（对于域名下面的哪些路径有效并将可以被返回）的路径。
- domain：设置cookie的域名（返回哪个域的cookie）。
- secure：设置cookie是否安全（当为true时，只有通过一个SSL通道时才返回cookie）。

你可能已经想到了，Dojo也提供了getCookie(name)和deleteCookie(name)函数。还有两个略微高级一点的函数：setObjectCookie()和getObjectCookie()。这些函数把cookie的值作为对象/数值对的形式来处理，而不是一个简单的数据类型。这些都是很便利的特性，这样你就不用写对象写出之前的分析代码了。

每一天，每个角落都在普遍地使用着cookie。它们简单、快速，并且足以满足大部分工作。不过，它们也还是有自己的局限性的。

- 每个域最多可以有20个cookie。你可能发现有一些浏览器允许更多，但是HTTP规范中声明最少要有20个。所以，为了确保你的代码不会在某些浏览器中崩溃，最好还是假设20就是最大的个数限制，因为大多数都把20作为上限来处理。

- 每一个cookie都被限制最大不超过4kb。口算就能得出，在客户端每个域名下最多可以有80kb的cookie。先别说80kb不能满足很多工作，光是它需要被分配到20个cookie中间，而且你还要自己写这些代码，就已经让cookie无法用于很多用途了。

幸运地是，Adobe的员工已经为我们提供了一个解决方案，并且如你所期望的，Dojo利用了这个机制，让开发者的生活更加美好。所以，在本章的项目中并不使用cookie，我们会简短认识一下将要使用的存储机制，但让我们先看看一些Dojo提供的其他特性。

6.2.2 Dojo窗口小部件和事件系统

通讯录应用程序将使用Dojo提供的UI小部件中的一个，叫做鱼眼列表。如果你已经玩过这个应用程序了，就会认出鱼眼列表就是顶部的一行图标，当鼠标移过其上时，它们就膨胀或收缩，如果你对Apple的操作系统很熟悉的话，它类似于Mac的启动条的效果。Dojo的窗口小部件的一个优点是，因为它们是在一个通用的窗口小部件框架上制作的，所以，它们大部分都是用同样方法使用的，因此了解一个就可以相当不错地帮你了解其他的使用。使用Dojo窗口小部件的细节可能相当冗长，所以会在我们仔细研究应用程序的时候再解释它们。在上下文中了解它们的使用会更清楚些。

我们还会用到Dojo中的事件系统，它允许我们将函数附加到任何可以发生的实际事件上，并不只是像鼠标点击那样的UI事件。Dojo提供一个面向特征的事件系统，比如，它允许你在任何其他函数调用的时候，同时调用另外一个函数。Dojo提供的事件系统是相当大的，在本章中我们将只使用一些表面的东西，但你应该可以感觉到这个系统的潜在能力。再说一遍，我会在仔细分析的时候解释相关的细节。

我们这个演出的真正的明星，或者坦率说是本章的重点，是Dojo的存储系统和本地共享对象，所以现在让我们就看看这部分！

6.2.3 本地共享对象和Dojo存储系统

本地共享对象（也叫Flash共享对象）有点类似于cookie，但是它是Adobe Flash插件的一种机制。它是和Flash MX一起被引入的，所以较早的版本（在版本6之前）不支持它们。它们的使用方法和使用cookie非常相同，但是使用cookie时的大小和每个域下面的数量限制，对本地共享对象来说就不适用了。在本地共享对象中，你可以想存多少数据就存多少，直到用户的硬盘被填满为止。

其实，Flash有比IE更广泛的安装基础：写这本书的时候已经接近97%了。这就意味着我们可以抛开很多认为flash插件是否存在于客户端的顾虑——插件很可能已经比其他很多东西还早就存在了，甚至比JavaScript自己都早。Flash甚至可以用于一些传统的受限的设备，例如PDA和手机（不过，糟糕的是，你可能会发现那些设备中有一些并不支持本地共享对象）。

然而，可能会出现一个速度的障碍，原因很简单：本地共享对象是在Flash短片中使用的，并不是JavaScript。我们怎么克服这个问题呢？当然，让Dojo来为我们解决它。

我们可以自己来写那些代码，不过可能会需要写一两个Flash短片公开一段后面可以与JavaScript进行交互的脚本界面。如果这让你大伤脑筋，那就加入我们吧。虽然我已经用Flash工作了一段时间，但远不是专家，并且自己写这些组件绝对会花费相当大的时间和精力。但既然开发Dojo的人们已经处理了所有麻烦，并且给我们提供了一种简单的方式来使用它，我们为什么还要自找麻烦？有时，懒一点是件好事！

Dojo提供了一个存储包，它号称是客户端持久存储的插件机制。它的结构是一个存储管理器和多

个存储提供者的概念。每一个提供者都可以通过任何它选择的方式来持久存储数据，但是客户端应用程序写到一个所有的提供者都实现的公用接口，从而允许开发者像摘帽子那样在不同的存储机制之间交换，而并不需要改动任何代码。这非常酷。

写这本书的时候，dojo.storage包只提供了一个存储提供者[①]，那就是处理共享对象的。dojo.storage包是个令人惊奇的创造，它给开发者提供了很多能量。它本质上说是一个简单的体系结构，就如图6-2所示，这又一次证明了简单通常就是合适的方法。

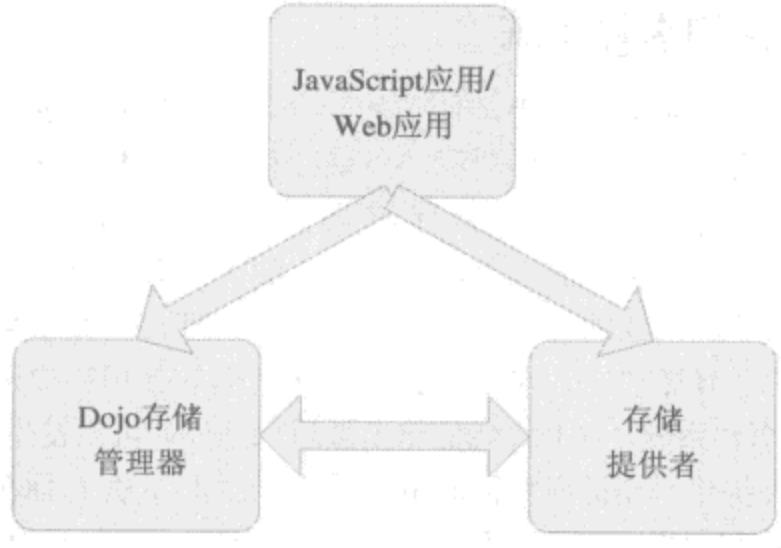


图6-2　dojo.storage包结构

当我们研究本章应用程序后的代码时，你就会看到，代码与dojo.storage进行的主要交互是通过那个存储管理器和存储提供者接口进行的，分别如图6-3和图6-4所示。而且，这里并没有太多东西，实际上，为实现本章应用的目标，我们所使用的接口不超过它提供的一半。

图6-3　Dojo存储管理器接口

图6-4　Dojo存储提供者接口

① cookie函数自然也是这种之一。因此很多人在谈论cookie提供者（到你看到本书的时候，甚至它已经实现了）。我们也可以构造一个直接写入SQL Server数据库的ActiveX提供者，而使用dojo.storage的应用不用知道这些细节。

我前面说过，Dojo开发者已经处理了可以和JavaScript交互的Flash短片的实现的细节。把那些短片想成类似于一个DAO（Data Access Object）的东西，负责实际的存储行为。在某种意义上，它公开的API是用dojo.storage API封装的。所以，当你调用StorageProvider类（嗯，就是这个类实现了那个API）的put()函数时，它就调用一些在Flash 影片中实际用来向存储对象保存数据的函数。这绝对是一个非常一流的解决方案。

那么，现在，你大概了解了实现机制在Dojo中如何工作，让我们看看将要使用它的这个应用程序吧。

6.3 通讯录的预览

图6-5展示了本章中要创建的那个应用程序。我非常推荐你在开始之前先花几分钟玩一下这个程序。我想你会发现它是一个相当简洁且有用的小通讯列表。它不会让Microsoft Outlook开发组的任何人睡不着觉，但是从任何角度看它都不算差。

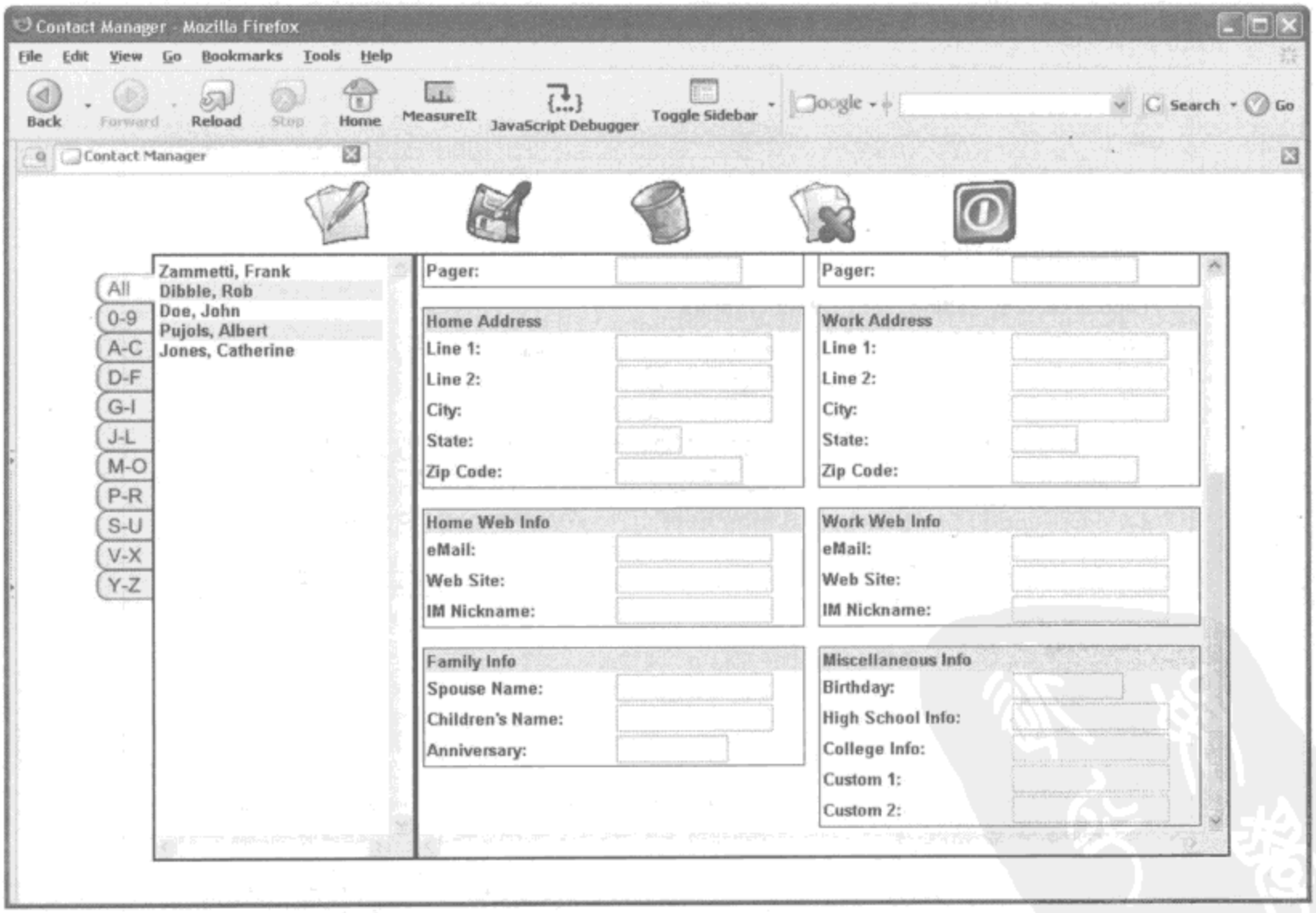


图6-5 JS通讯录：Outlook可能不用担心，但是它不差

当你第一次运行这个应用程序时，很可能看到一个由Flash生成的弹出对话框，类似图6-6那样的。

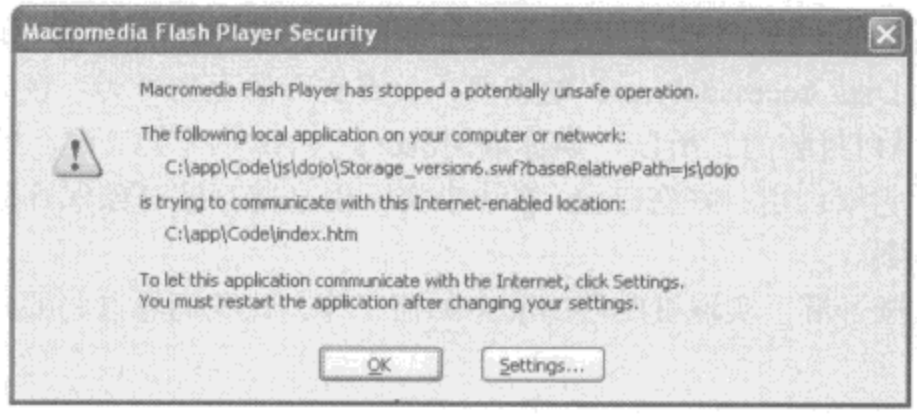


图6-6　第一次运行该应用程序时看到的安全警告

如果你没有看到这个弹出框，那很好，庆幸自己的好运并继续往下看。如果你看到它了（可能性很高），这是在Flash插件中一些安全警告的结果。你需要告诉插件，在本地文件系统中的这个目录（就是你的应用程序运行的）是允许访问的，不应该再询问了。比较麻烦的事情是，你需要因特网连接来修改这个设置。信不信由你，你需要的那个设置对话框，是在Adobe网站的一个网页！图6-7展示了这个页面。

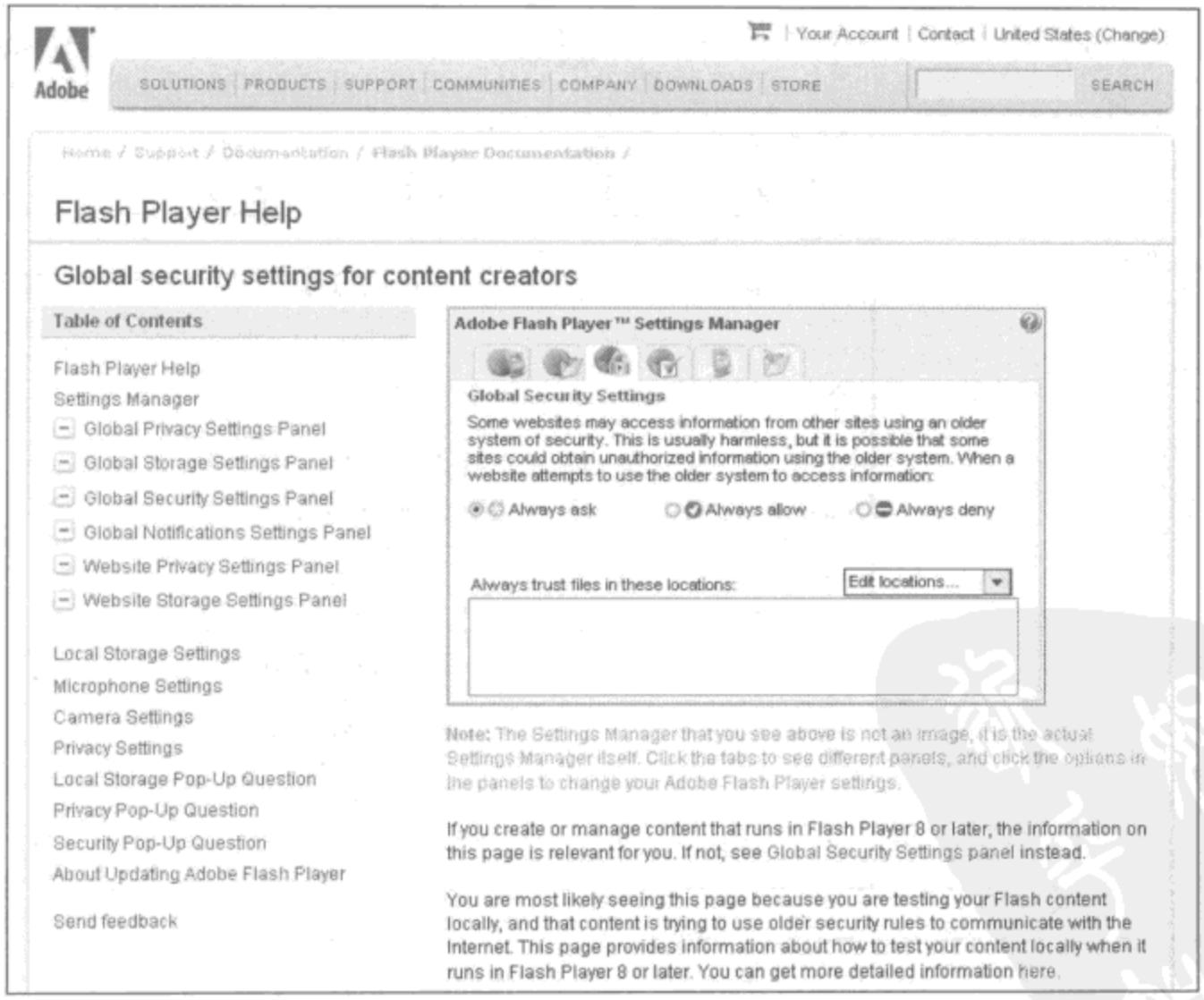


图6-7　Flash安全设置网页

我只能假设这么做是有理由的，而且Adobe知道这个原因，但显然我并不相信！不管怎样，当你

点击安全警告对话框中的Settings按钮时，假定已经连接了因特网，都将被定向到这个页面。在这里，你需要点击Edit Location框，并选择Add Location，然后浏览运行应用程序的那个文件夹，这应该就行了。我还选择了在Always Allow旁的单选按钮。这可能并不是必须的，但是你也许也得这么做。只要你做了这些修改，关闭那个页面，重新载入应用程序，就应该一切正常了。

6.4 剖析通讯录的解决方案

和本书中其他项目一样，我们先感受一下文件布局，也就是说，先看看目录结构。看看图6-8。

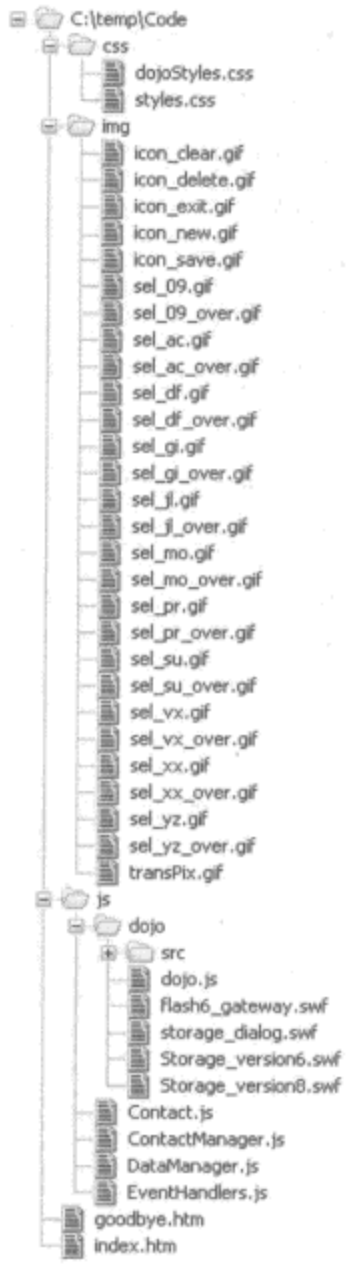


图6-8 JS通讯录目录结构

在根目录下有两个文件。

- index.htm，你不久就会看到，是这个应用程序的心脏和灵魂。
- goodby.htm，一个简单的页面，当用户退出应用程序时显示。

在根下面还有一些目录，从css目录开始。和往常一样，css目录是我们存放样式表的地方。这个程序中，我们有两个样式表。

- styles.css，应用程序的主要样式表。
- dojoStyles.css，它包括为Dojo窗口小部件使用的特别样式。

下一个目录是img，很明显，它包括了整个应用程序中使用到的图片。图片使用非常易读的方式命名，所以这里就没有必要全部看一遍了。

最后是js目录，当然，这里是JavaScript存在的地方。存在着4个.js文件。

- Contact.js，定义了描绘一个联系信息的类。
- ContactManager.js，包括主要的应用程序代码，它或多或少有些类似于一个交通警察，调用所需要的其他类。
- DataManager.js，包括实际用来存储和获取联系信息的代码。
- EventHandlers.js，包括使UI正常工作的UI事件处理程序。

在js目录中，你还会发现一个名叫dojo的子目录，这就是Dojo库存放的地方。Dojo有很多版本，每一个版本都把Dojo的特定部分单独放置在一个.js文件中。你可以选择使用最小的那个版本，也就是说它只在dojo.js中包括了核心的Dojo代码，任何你将用到的其他部分都需要单独引入。这也是我们准备使用的版本，因为它减少了启动的装载时间并且装载了各种我们需要的Dojo部件。这个版本，在我看来，还可以降低我们出问题的可能，原因很简单，因为内存里的代码越少，出错的机会就越小。

在dojo目录中，你将看到一些子目录，它们包含了其他Dojo包的源文件，HTML和CSS资源以及其他所有的Dojo需要运行的东西。虽然这个应用程序只使用了Dojo的一小部分，这里包含全部意味着如果你希望扩展这个程序并使用更多的Dojo时，它们就都在那里了，为你时刻待命。

那么现在，就不再耽误时间了，让我们开始接触一些代码吧。

6.4.1　编写styles.css

代码清单6-1展示了styles.css文件，它是主要的样式表，定义了应用程序的大部分可见样式。

代码清单6-1　styles.css文件

```
/* Generic style applied to all elements */
* {
  font-family      : arial;
  font-size        : 10pt;
  font-weight      : bold;
}

/* Style for body of document */
body {
  margin           : 0px;
}

/* Style for spacer between data boxes */
.cssSpacer {
  height           : 64px;
}
```

```
/* Non-hover state of contacts in the contact list */
.cssContactListNormal {
  background-color : #ffffff;
}

/* Non-hover state of contacts in the contact list, alternate row striping */
.cssContactListAlternate {
  background-color : #eaeaea;
}

/* Hover state of contacts in the contact list */
.cssContactListOver {
  background-color : #ffffa0;
  cursor           : pointer;
}

/* Style for main container box */
.cssMain {
  width            : 100%;
  height           : 580px;
  z-index          : 0;
}

/* Style of the initializing message seen at startup */
.cssInitializing {
  width            : 920px;
  height           : 2000px;
  z-index          : 1000;
}

/* Style of selector tabs */
.cssTab {
  position         : relative;
  left             : 2px;
  _left            : 4px; /* Style for IE */
}

/* Style of the contact list container */
.cssContactList {
  padding          : 4px;
  height           : 480px;
  border           : 2px solid #000000;
  overflow         : scroll;
}
/* Style of the box surrounding the data boxes */
.cssMainOuter {
  padding          : 4px;
  height           : 480px;
  border           : 2px solid #000000;
```

```
  overflow          : scroll;
}

/* Style of a data box */
.cssDataBox {
  width             : 100%;
  border            : 1px solid #000000;
}

/* Style for the header of a data box */
.cssDataBoxHeader {
  background-color : #e0e0f0;
}

/* Style for textboxes */
.cssTextbox {
  border : 1px solid #7f9db9;
}
```

星号选择器，和你在本书中其他项目里看到的一样，我们又想用一招就给页面中的所有东西都设定样式。

大多数样式表都是不需要加说明的，不过有个需要指出的技巧，就是在cssTab选择器。这个样式是用来定位左边那个按照字母顺序来选择的标签的。为了让其中一个标签是当前标签的效果可用，那这个标签可能需要盖住联系列表的边框几个像素大小。很不幸地，这里所需的像素大小在IE和Firefox中的效果并不一致，所以我们对这个区别使用了一些技巧。

当Firefox遇到一个以下划线开头的属性时，它就忽略这个属性。而IE就只是忽略那个下划线，就好像那里没有下划线一样。所以，这里实际的效果是，在两个浏览器中，都给left属性设置了第一个值2px。然后当遇到_left属性时，Firefox忽略它，不过IE删去了下划线，并把它作为left属性来设置，用4px替换了2px的值。用这种方法，你可以不需要写太多的分支代码，就可以处理在IE和Firefox中出现的样式表差别。

说明　如果你使用了太多带有下划线前缀的属性名称，可能就应该重新思考一下使用元素样式的方法了，因为你的做事方法可能与浏览器太相关了。不过不管什么时候，偶尔用用这个方法是一个好技巧。还应该注意到，将来其中的一个或者全部浏览器都可能修改这行为，这样就破坏了使用它（这个技巧）的所有页面。这只是一些应该记在脑子里的事情。

6.4.2　编写dojoStyles.css

dojoStyles.css是一个只包含了很少选择器的样式表，它覆盖了Dojo中的默认样式。代码清单6-2展示了这个相当小的源文件。不管大小怎样，我们都应该看看这个样式表。

代码清单6-2　dojoStyles.css文件

```
/* Style for the fisheye listbar */
.dojoHtmlFisheyeListBar {
```

```
  margin          : 0 auto;
  text-align      : center;
}

/* Style for the fisheye container */
.outerbar {
  text-align      : center;
  position        : absolute;
  left            : 0px;
  top             : 0px;
  width           : 100%;
}
```

虽然小，但是这个样式表提供了相当重要的样式信息。没有它的话，那个很酷的鱼眼图标看起来就不对了。

6.4.3 编写index.htm

index.htm中，我们可以看到组成通讯录应用程序的大部分源代码。它主要使用散置的脚本来展示页面。让我们从<head>部分看起，那里引入了样式表和JavaScript，如代码清单6-3所示。

代码清单6-3 index.htm里输入的JavaScript和样式表

```
<link rel="StyleSheet" href="css/dojoStyles.css" type="text/css">
<link rel="StyleSheet" href="css/styles.css" type="text/css">
<script type="text/javascript">
  var djConfig = {
    baseScriptUri : "js/dojo/",
    isDebug : true
  };
</script>
<script type="text/javascript" src="js/dojo/dojo.js"></script>
<script language="JavaScript" type="text/javascript">
  dojo.require("dojo.widget.FisheyeList");
  dojo.require("dojo.storage.*");
</script>
<script type="text/javascript" src="js/Contact.js"></script>
<script type="text/javascript" src="js/EventHandlers.js"></script>
<script type="text/javascript" src="js/DataManager.js"></script>
<script type="text/javascript" src="js/ContactManager.js"></script>
```

首先，我们引入两个样式表。然后是一小段JavaScript，它设置了一些Dojo属性。djConfig变量是一个关联数组，它是在Dojo启动时要查找的（也就是说这段代码必须在Dojo引入之前出现），它包括了一些各种Dojo设置的选项。在这个例子中，有两个是非常重要的。

- baseScriptUri定义了路径的开始部分，在那里可以找到所有的Dojo资源文件。包括诸如源文件、图像、样式表等。在这个例子中，我们把Dojo安装在js/dojo，所以这就是这个属性应有的值。
- isDebug定义了Dojo是否要打印错误信息。

djConfig还有一些其他可用的值（网上搜索一下，就可以看到它们了），但是按照我们这里的目标，就只有这两个是很重要的。

在那段代码后面是Dojo自身主要部分的引入。紧跟着的是一系列dojo.require()函数调用的代码。Dojo是以包的结构来组织的，像我们在第3章中创建的一样。Dojo还实现了根据需要引入一小部分的功能，以及提供通配符（.*）引入功能。而且，如果你引入的一些功能是依赖于其他未被引入的程序，Dojo会自动为你引入那些依赖。

在引入Dojo之后，是4个更加直白的JavaScript引入语句，它们带来了组成通讯录应用程序的代码。我们一会儿会迅速看看每个文件的细节。

1. 添加引导代码

在<head>部分的结尾，是一段<script>，它包括实际上引导应用程序执行的代码。这个代码如代码清单6-4所示。

代码清单6-4　index.htm <head>的引导用的JavaScript代码

```
<script>

  // Shorthand function to get a reference to a DOM element.
  function $(inID) {
    return document.getElementById(inID);
  } // End $().

  // The contactManager instance that is the core of this application.
  var contactManager = new ContactManager();

  // Connect init() function in ContactManager to onLoad event.
  dojo.event.connect(window, "onload", contactManager.init);

</script>
```

首先你看到的是一个名叫$的函数，你已经在前面章节的项目中见过它了。应该已经知道，它是一个封装了那个到处出现的document.getElementById()调用的函数。它帮我们节省了开发的打字时间，并让代码看起来更加干净。

在那之后的是ContactManager类的实例，这个类基本上是应用程序的核心代码（不久你就会看到这个了）。

最后，你看到了下面这行：

```
dojo.event.connect(window, "onload", contactManager.init);
```

对于新的Dojo开发者（包括我）来说，有一件很棘手的事情，就是Dojo接管了onLoad事件并覆盖了所有你可能已经放在那里的东西。这就意味着，我们不能简单地像其他时候一样调用onLoad来实现应用程序了。替换方法是使用dojo.event包，它是面向方向编程（AOP）来实现的，它允许把JavaScript挂到各种各样的事件上。这听起来简单，不过dojo.event是一个具有令人惊讶的强大功能的包。你不仅仅可以挂在通常的UI事件上，比如onLoad、onClick、onMouseOver等，而且还可以挂在一个任意函数调用这样的事件上。比如，如果你希望在每次函数B被调用的时候都执行函数A，而并不想让函数B明确地调用函数A（为了避免它们知道对方），那么就可以使用dojo.event来实现。

这里，我们正在讨论的是，希望每次窗口对象的onLoad事件被触发时，都调用对象contactManager上的函数init()。这样就给了我们与使用onLoad相同的功能，不过使用的是Dojo的事件系统。

在保持JavaScript不唐突方面，你会注意到，目前为止，这个页面很少有这些考虑。一些人可能会认为，即使是我这里做的这些，也应该写在.js文件中，但是我认为你有时候可能会把不唐突这个观点延伸得太远了。我并不认为一定要把每一丁点儿的脚本都从外部引入，但是确实应该保持脚本最少，我想你应该看得出来我在这里用的是这个观点。

2. 初始化应用

现在我们来到页面的<body>部分了，它像这样开始：

```
<div id="divInitializing" class="cssInitializing">
  <br><br><br><br><br><br><br><br><br><br>
  <center>...Initializing Contact Manager, please wait...</center>
</div>
```

当第一次载入页面的时候，我们并不希望用户可以在完全载入UI之前就触发UI事件（比如，在持久存储的通讯录还没恢复之前就点击保存按钮可不是一件好事）。所以，刚开始的时候，用户会看到一个消息说应用程序正在初始化。一旦万事俱备了，这个 就被隐藏起来，然后就展示主要内容。

3. 添加鱼眼列表

现在我们来看一些有趣的UI。Dojo做的非常好的一件事（可能是它最著名的）就是窗口小部件。它其中一个绝对异乎寻常的窗口小部件（最让我想花钱的一个）就是鱼眼列表。你见过Mac 操作系统中的启动栏吗？

你想知道在移动鼠标到图标上面时，它们是如何扩展并收缩的吗？好的，Dojo的鱼眼列表让你也可以在自己的Web应用程序中拥有和图6-9中同样的效果。当然，在静态截屏中看对它是不公平的。所以运行那个应用程序，然后就把鼠标移到图标上面即可。我想你会很乐意什么都不做就这么玩几分钟。

图6-9 Dojo鱼眼窗口小部件的炫酷

那么使用那个小部件很难吗？当然不！即使不是全部，Dojo也做到了允许你使用HTML就可以定义绝大部分的窗口小部件。代码清单6-5展示了这个程序中应鱼眼列表的HTML标记。

代码清单6-5　定义鱼眼的HTML标记

```
<div class="outerbar">
  <div class="dojo-FisheyeList"
    dojo:itemWidth="64" dojo:itemHeight="64"
    dojo:itemMaxWidth="128" dojo:itemMaxHeight="128"
    dojo:orientation="horizontal" dojo:effectUnits="1"
    dojo:itemPadding="10" dojo:attachEdge="top"
    dojo:labelEdge="bottom" dojo:enableCrappySvgSupport="false">
    <div class="dojo-FisheyeListItem"
      onClick="contactManager.doNewContact();"
      dojo:iconsrc="img/icon_new.gif" caption="New Contact">
    </div>
    <div class="dojo-FisheyeListItem"
      dojo:iconsrc="img/transPix.gif" caption="">
    </div>
    <div class="dojo-FisheyeListItem"
      onClick="contactManager.doSaveContact();"
      dojo:iconsrc="img/icon_save.gif" caption="Save Contact">
    </div>
    <div class="dojo-FisheyeListItem"
      dojo:iconsrc="img/transPix.gif" caption="">
    </div>
    <div class="dojo-FisheyeListItem"
      onClick="contactManager.doDeleteContact()"
      dojo:iconsrc="img/icon_delete.gif" caption="Delete Contact">
    </div>
    <div class="dojo-FisheyeListItem"
      dojo:iconsrc="img/transPix.gif" caption="">
    </div>
    <div class="dojo-FisheyeListItem"
      onClick="contactManager.doClearContacts()"
      dojo:iconsrc="img/icon_clear.gif" caption="Clear Contacts">
    </div>
    <div class="dojo-FisheyeListItem"
      dojo:iconsrc="img/transPix.gif" caption="">
    </div>
    <div class="dojo-FisheyeListItem"
      onClick="contactManager.doExit();"
      dojo:iconsrc="img/icon_exit.gif" caption="Exit Contact Manager">
    </div>
  </div>
</div>
```

在载入页面时，Dojo将解析它，查找那些它识别为定义窗口小部件的标签。然后使用适当的HTML标签替换它们，组成窗口小部件。在这个例子中，把一个 的类设置为dojo-FisheyeList就可以了。你会注意到这个 还包括了一些用户属性。这也是Dojo的一个特点以及你如何配置窗口小部件选

项的方法。每一个图标都是另外一个简单的 ，这次使用的是dojo-FisheyeListItem类(css中的类)。注意，我们在每一个实际的图标之间放置了空白图标。可能还有另外的我不知道的方法，不过这样我可以让图标之间空出一些地方，要不然我觉得它们看起来有一些拥挤。

可以随便修改第二个 。你可以通过操作它们来改变鱼眼列表，比如使放大的图片更大一点、修改离图标多远你就激活等。实践是学习Dojo的一个好方法（事实上，它通常是真正领悟一件事情的唯一途径）。Dojo通常值得尝试，所以我绝对鼓励你花一些时间来玩一会儿，而弄乱鱼眼的属性通常是一个很好的开始。

继续，我们在每个图标上定义一个onClick事件处理函数来处理图表对应的操作。这里又是一个和严格的不唐突使用JavaScript有出入的地方，不唐突建议不要内联事件处理函数。这里我想要声明下面两点。

- 如果事件处理函数只是调用了一些做实际工作的JavaScript函数，我想，把处理函数写成内联的并没有什么问题。事实上，我认为它们在那里更有道理，它们作为元素的一个属性，因此应该与元素一起定义。而且，这样可以帮助提高性能，因为不需要处理与事件处理挂钩的脚本。
- 由于Dojo在这些 元素上生成HTML标记，那么如果想在事后勾上事件可能比想象的要更复杂些。dojo.event包可以帮忙，但它依然不是那么直接。

所以，今晚我不会再为内嵌的事件处理函数而失眠了。

通常，不要盲目地遵从任何手册，不论它看起来多有道理。而是要考虑每一种情况，并作出一个适当的选择。

好，让我们把哲学的论调放在一边，回到具体的代码中来，继续下面的联系列表部分。

4. 添加联系列表

在鱼眼的HTML标记之后，是一大块定义左边的选择器标签的HTML标记。简单地说，我们就只看看第一段，并注意剩下的实际上都是一样的：

```
<img src="img/sel_xx_over.gif" id="sel_XX" class="cssTab"
  onClick="contactManager.eventHandlers.stClick(this);"
  onMouseOver="contactManager.eventHandlers.stOver(this);"
  onMouseOut="contactManager.eventHandlers.stOut(this);"><br>
```

这是All标签，意思是它展示所有的联系信息。标签的ID是用于根据它的当前状态（也就是说，它是已选的标签，还是鼠标悬浮的）判断它应该使用的图片的。比如，在这个例子中，ID是sel_XX，如你所见，它是这个标签初始化图片的源文件名字的开头。当我们看事件处理函数（也是一个内联的事件处理函数）的时候，我想你就会理解更多了。我说过，后面的所有标签都是一样的，只是ID不同而已。例如，下一个标签的ID是sel_09，因为它是用来显示那些以0～9之间的字符开头的联系信息的。

在这个HTML标记之后是非常小的一段代码：

```
<td width="200" valign="top">
  <div class="cssContactList" id="contactList">
  </div>
</td>
```

毫无疑问，contactList 是我们将要插入联系信息的列表。这个列表是被选择器标签过滤的，

每当选择一个新的标签，或者添加删除一个联系信息时，这个 的内容都会重写。

在这之后是我们为填写联系人信息的数据输入框HTML标签的剩余部分。这是一大块看上去非常相似的HTML标记，所以让我们只看看它的一个片段。输入联系信息识别身份（包括个人和商业两部分）的部分，如代码清单6-6所示。

代码清单6-6　数据录入标记段

```
<tr>
  <!-- Contact Identity -->
  <td width="49%" valign="top">
    <div class="cssDataBox">
      <table border="0" cellpadding="1" cellspacing="1"
        width="100%">
        <tr>
          <td colspan="2" class="cssDataBoxHeader">
            Contact Identity
          </td>
        </tr>
        <tr>
          <td width="50%" valign="middle">Title:</td>
          <td width="50%" valign="middle">
            <input type="text" id="title"
              maxlength="3" size="4" class="cssTextbox">
          </td>
        </tr>
<tr>
  <td valign="middle">First Name:</td>
  <td valign="middle">
    <input type="text" id="firstName"
      maxlength="15" size="15" class="cssTextbox">
  </td>
</tr>
<tr>
  <td valign="middle">Middle Name:</td>
  <td valign="middle">
    <input type="text" id="middleName"
      maxlength="15" size="15" class="cssTextbox">
  </td>
</tr>
<tr>
  <td valign="middle">Last Name:</td>
  <td valign="middle">
    <input type="text" id="lastName"
      maxlength="20" size="15" class="cssTextbox">
  </td>
</tr>
<tr>
  <td valign="middle">Suffix:</td>
```

```
          <td valign="middle">
            <input type="text" id="suffix"
              maxlength="3" size="4" class="cssTextbox">
          </td>
        </tr>
      </table>
    </div>
  </td>
  <!-- Divider -->
  <td width="2%"> </td>
  <!-- Work Identity -->
  <td width="49%" valign="top">
    <div class="cssDataBox">
      <table border="0" cellpadding="1" cellspacing="1"
        width="100%">
        <tr>
          <td colspan="2" class="cssDataBoxHeader">
            Work Identity
          </td>
        </tr>
        <tr>
          <td width="50%" valign="middle">Job Title:</td>
          <td width="50%" valign="middle">
            <input type="text" id="jobTitle"
              maxlength="24" size="15" class="cssTextbox">
          </td>
        </tr>
        <tr>
          <td valign="middle">Company:</td>
          <td valign="middle">
            <input type="text" id="company"
              maxlength="25" size="15" class="cssTextbox">
          </td>
        </tr>
        <tr>
          <td valign="middle">Department:</td>
          <td valign="middle">
            <input type="text" id="department"
              maxlength="25" size="15" class="cssTextbox">
          </td>
        </tr>
        <tr>
          <td valign="middle">Manager's Name:</td>
          <td valign="middle">
            <input type="text" id="managerName"
              maxlength="30" size="15" class="cssTextbox">
          </td>
        </tr>
          <tr>
            <td valign="middle">Assistant's Name:</td>
```

```
            <td valign="middle">
              <input type="text" id="assistantName"
                maxlength="30" size="15" class="cssTextbox">
            </td>
          </tr>
        </table>
      </div>
    </td>
  </tr>
```

这是相当典型的HTML。注意，字段的大小是有限制的，这样保证每个联系信息最多占用1024字节。这么做是因为，如果你想要修改代码使之用原始的cookie来存储联系信息（想想，想想），那你就可以在每个cookie中保存4个联系人信息（记住每个cookie都被限制在4 kb以内）。

还要注意cssTextbox样式类的使用。这是用来处理这样一种情况，当字段获取焦点时边框变为嵌入的，但并不变回来。这好像是浏览器的毛病，但是在cssTextbox类中指定的我们想要的边框设置可以解决它。

然后，我们就只剩下一块HTML标记要看了，那就是goodbye.htm页面。

6.4.4　编写goodbye.htm

如代码清单6-7所示，goodbye.htm文件东西很少。

代码清单6-7　goodbye.htm文件（没什么太多可看的）

```
<html>

  <head>

    <title>Contact Manager</title>

  </head>

  <body>
    Thanks for using the contact manager... goodbye!
  </body>

</html>
```

这个页面是个简单的“着陆”界面，当用户点击Exit图标的时候就会看到它。以一种比较礼貌的“感谢你停止”的消息来结束，总是比较好的方式。

让我们从EventHandler类开始来看看JavaScript文件，因为这是一段独立的代码。

6.4.5　编写EventHandlers.js

index.htm中，你看到选择标签调用了这个类的事件处理函数。而且你会看到，所有的输入框所做的事情实际上都是相同的（“什么？”你可能在想，“我不记得在那些输入框上看到过任何事件处理函数啊”，你是对的）。还有，你在屏幕左边看到的联系信息列表也使用了这里的一些函数，但是这些都是简单的UI相关的函数。换句话说，如果你把这些函数拿出来，功能基本还是能完成的，但是UI就不

会像它以前那样反应了。当把鼠标移到它们上面时，选择器不会变红。当输入框获得焦点时也不会突出显示。而且联系信息列表根本就不会有任何鼠标悬浮的效果了。所有的这些动态效果都是在EventHandlers类中的，图6-10展示了这个类图。

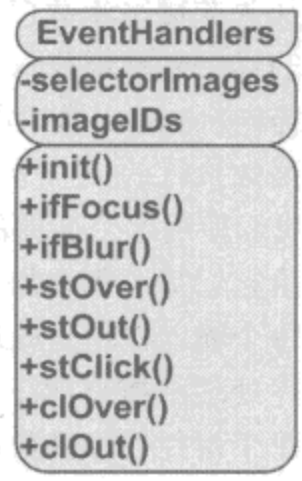


图6-10 EventHandlers类的UML图表

首先，EventHandlers类是在后面我们将会看到的ContactManager类中被实例化的，而且对它的引用也是ContactManager中的一个字段（所以所有index.htm里面的事件处理函数都是contactManagerxxxx()的形式）。

EventHanlder类的第一个字段是selectorImages字段，它是一个存储指向选择器标签页的预先读图片的引用的数组。然后是另一个数组字段，imageIDs，它是选择器标签页的ID列表。每一个标签页的图片名称可以使用这些ID来构成，页面元素的ID也是可以的，我们两个都需要。

如我曾提到过的，ContactManager将要实例化EventHandlers，并使用它上面的init()进行初始化。这个init()函数如下所示。

```
this.init = function() {

  this.selectorImages = new Array();

  // Load images from the above array and store them in selectorImages.
  for (var i = 0; i < this.imageIDs.length; i++) {
    var sid = this.imageIDs[i];
    this.selectorImages[sid] = new Image();
    this.selectorImages[sid].src = "img/" +
      sid + ".gif";
    this.selectorImages[sid + "_over"] = new Image();
    this.selectorImages[sid + "_over"].src = "img/" +
      sid + "_over.gif";
  }

  // Get all input fields and attach onFocus and onBlur handlers.
  var inputFields = document.getElementsByTagName("input");
  for (i = 0; i < inputFields.length; i++) {
    inputFields[i].onfocus = this.ifFocus;
    inputFields[i].onblur = this.ifBlur;
  }

} // End init().
```

这个函数执行的第一个任务是，为那些选择器标签页预读入图片。它遍历imageIDs数组的元素。在每一个上都实例化一个Image对象并把它的src属性设置为图片的文件名。每个标签载入两个图片：一个是鼠标没有位于其上的状态，另一个是鼠标悬浮在上面的状态。然后图片被添加到数组中，以从imageIDs得来的ID作为下标。

我们最后得到关联数组selectorImages，它包含了每个标签的两个状态预读的所有图片，我们可

以使用ID作为关键字来获得每个图片（对于悬浮图片，它的ID就是后缀字符串_over）。这样我们就不用写明确的代码来重新载入每个图片。如果在以后想添加更多的标签，只要它们遵从同样的命名规则，我们就只需向imageIDs数组添加ID就可以了。

这个函数执行的下一个任务是将事件处理函数和输入框挂钩起来。是的，我们正要看看它是如何工作的。我们使用那个简便的函数document.getElementsByTagName()来获得页面上的<input>字段的集合。然后给它们每个都添加onFocus和onBlur事件，指向EventHandlers类的ifFocus()和ifBlur()函数。不需要为每个输入框指定处理函数的感觉挺好吧？

将事件处理函数追加在一个老式的HTML标记上又是另外一个内联的原则。虽然我们并不确信对页面中的每个事件处理函数都这么做是否合适，但在这个场合，在这种一大堆元素都需要附着同样的处理函数的时候，我认为比在HTML标记上内联处理函数的方式更好。

我认为我们应该看看那个ifFocus()和ifBlur()函数，不是吗？好的，这里就是：

```
// ********** Input Field focus.
this.ifFocus = function() {

  this.style.backgroundColor = "#ffffa0";

} // End ifFocus();


// ********** Input Field blur.
this.ifBlur = function() {

  this.style.backgroundColor = "#ffffff";

} // End ifBlur().
```

当然，这些并没有什么特殊。它们只是改变了输入字段的背景颜色：当它获取焦点时是黄色的，失去焦点时是白色的。把将当前字段突出显示是能给大多数用户很好的感觉的特性，所以应该实现。

在ifFocus()和ifBlur()函数中，注意关键字this的使用，它可能让人觉得有些混淆。回忆一下：这些函数都是以事件处理函数的方式附加在页面元素上的。当调用它们时，代码this.style.backgroundColor的关键字this就指向了触发当前事件的那个元素，因为在运行时，关键字this是在它执行的环境中进行计算的。和这个this用法对比的是this.ifBlur那行。在那里，this指向EventHandlers类，因为它是定义在那个类中的。你可以把这看成是对关键字this的静态和动态的解释。静态是指它用来将方法添加到EventHandler类上，动态是指它与事件处理函数一同使用。这个概念通常叫做早绑定与晚绑定。晚绑定是在运行时发生的，而早绑定是在编译时发生的。当然，对于JavaScript来说并没有“编译时”，但是它还是意味着在代码实际运行之前。

在那些函数之后，是3个处理选择器标签页的函数。

```
// ********** Selector Tab mouseOver.
this.stOver = function(inTab) {

  inTab.src = this.selectorImages[inTab.id + "_over"].src;

} // End stOver().
```

```
// ********** Selector Tab mouseOut.
this.stOut = function(inTab) {
  // Only switch state if not the current tab.
  if (contactManager.currentTab != inTab.id.substr(4, 2)) {
    inTab.src = this.selectorImages[inTab.id].src;
  }

} // End stOut().


// ********** Selector Tab click.
this.stClick = function(inTab) {

  // Reset all tabs before setting the current one.
  for (var i = 0; i < this.imageIDs.length; i++) {
    var sid = this.imageIDs[i];
    $(sid).src = this.selectorImages[sid].src;
  }

  inTab.src = this.selectorImages[inTab.id + "_over"].src;

  // Record the current tab, and redisplay the contact list.
  contactManager.currentTab = inTab.id.substr(4, 2);
  contactManager.displayContactList();

 } // End stClick().
```

stOver()和stOut()函数，分别处理onMouseOver和onMouseOut事件。它们使用那些之前讨论过的预载入并存储在selectorImage数组中的图片。这两个函数只是把触发事件标签上的src属性改成数组中合适的图片。当然，如果一个用户鼠标悬浮在当前选定的标签页上，然后再移开鼠标，我们可不想把它重置为未悬浮的状态，因此我们在stOut()中写检查以确保触发事件的标签页不是当前选定的标签页。

stClick()更加有趣一点点。当用户点击一个标签，它变成当前标签，这意味着它还保留着鼠标悬浮的状态，直到另一个标签被选中。为了实现这个效果，我们首先要重置当前的标签为非悬浮状态。我决定先重置所有的标签，然后再设置新的当前标签。[①]在处理完所有标签页之后，我们调用ContactManager对象上的displayContactlist()，把联系信息列表更新为只包含应该在新的当前标签下显示的联系信息。

在EventHandler类的最后，是两个处理屏幕左边联系信息列表中的条目上发生的鼠标事件的函数。

```
// ********** Contact List mouseOver.
this.clOver = function(inContact) {

  inContact.className = "cssContactListOver";
```

① 我可以只重置当前标签页，而不是全部。你可以把这个看作一个可选的方案。

```
} // End clOver().


// ********** Contact List mouseOut.
this.clOut = function(inContact) {

  if (inContact.getAttribute("altRow") == "true") {
    inContact.className = "cssContactListAlternate";
  } else {
    inContact.className = "cssContactListNormal";
  }

} // End clOut().
```

为了示范一个稍微不同的技术，我决定，不同于输入框的处理器那样直接修改目标元素的样式属性，这里我将为目标元素设置合适的样式类。我认为这通常是一种更好的办法，因为样式可以像一般应用那样抽象到样式表中。但是现在你知道也可以使用另外一种方式，只要你觉得那样做更合适。

这里，唯一真正复杂的是在clOut()函数中。在鼠标离开之后，可以用来切换的样式可以是两者之一，因为在联系人列表里面的联系信息是用可选的行抽取方法显示的，很多显示列表经常这么做。为了判断应该设置哪个，我们询问列表所携带的每个联系信息各自的altRow属性。当那个属性被设置为true，就知道它是一个灰色背景的元素（即cssContactListAlternate样式选择器），否则，它是白色背景的（使用cssContactListNormal选择器）。除此之外，clOut()函数是非常直白的。

6.4.6　编写Contact.js

如果你熟悉DTO（Data Transfer Object，数据传输对象）或VO（Value Object，价值对象）的概念，Contact.js源文件对你来说就没什么特别了。因为它就是简单地定义了一个DTO来展现一个联系信息。它的类图如6-11所示。

图6-11　Contact类的UML图表

首先是一个展现联系信息的属性列表

```
this.title = "";
this.firstName = "";
this.middleName = "";
this.lastName = "";
this.suffix = "";
this.jobTitle = "";
this.company = "";
this.department = "";
this.managerName = "";
this.assistantName = "";
```

```
this.homePhone = "";
this.homeCellPhone = "";
this.homeFAX = "";
this.homePager = "";
this.workPhone = "";
this.workCellPhone = "";
this.workFAX = "";
this.workPager = "";
this.homeEMail = "";
this.homeWebSite = "";
this.homeIMNickname = "";
this.workEMail = "";
this.workWebSite = "";
this.workIMNickname = "";
this.spouseName = "";
this.childrenName = "";
this.anniversary = "";
this.birthday = "";
this.highSchoolInfo = "";
this.collegeInfo = "";
this.custom1 = "";
this.custom2 = "";
this.homeAddressLine1 = "";
this.homeAddressLine2 = "";
this.homeCity = "";
this.homeState = "";
this.homeZipCode = "";
this.workAddressLine1 = "";
this.workAddressLine2 = "";
this.workCity = "";
this.workState = "";
this.workZipCode = "";

this.arrayIndex = -1;
```

注意一下最后的那个脱离队伍的家伙，arrayIndex。实际上你可以认为这个字段是暂时的，是存储在DataManager（联系人存储在这里）里的联系人数组的下标。这个值在联系信息被删除时会改变，并且在应用程序初始化时恢复联系人信息的时候会动态计算。注意它并没有在后面将要讨论的fieldsArray中出现，这就是为什么它是暂时的原因（当我们讨论fieldsArray时，它不在列表中的原因就会变得明朗了）。

在这之后，我们发现了一段有意思的代码，它确实需要一些解释。就是那个fieldsArray变量，它是一个包含了这个类中所有字段的名字的数组：

```
this.fieldsArray = [
  "title", "firstName", "middleName", "lastName", "suffix", "jobTitle",
  "company", "department", "managerName", "assistantName", "homePhone",
  "homeCellPhone", "homeFAX", "homePager", "workPhone", "workCellPhone",
  "workFAX", "workPager", "homeEMail", "homeWebSite", "homeIMNickname",
  "workEMail", "workWebSite", "workIMNickname", "spouseName", "childrenName",
```

```
    "anniversary", "birthday", "highSchoolInfo", "collegeInfo", "custom1",
    "custom2", "homeAddressLine1", "homeAddressLine2", "homeCity", "homeState",
    "homeZipCode", "workAddressLine1", "workAddressLine2", "workCity",
    "workState", "workZipCode"
  ];
```

如果你看到这些数据字段，然后对比一下index.htm中的输入框的ID，你将发现它们是对应的。这些应该是关于这个数组是干什么的一条线索，但是，不用担心，我们就要把它全盘托出了！

在应用程序中的各个点，我们都会需要从输入字段的值上构建一个Contact类的实例，或者用Contact实例内部的数据字段来填充屏幕上显示的信息。人们肯定可以按照这些思路来设想所写的代码：

```
this.firstName = $("firstName").value;
this.lastName = $("lastName").value;
```

我们也可以想象在一个大热天跳下布鲁克林大桥，只要活着，于是就凉快了。但是这种跳桥方式的代码并不一定是实现目标的最好方法。有时候，如果我们能写一些填充对象或者输入框的通用代码会更棒，这样代码就不需要知道都有哪些字段。而且这样我们也可以更容易地给联系人列表添加元素。这就是在Contact类里面出现的代码——比如，在populateContact()函数里：

```
this.populateContact = function() {

  for (var i = 0; i < this.fieldsArray.length; i++) {
    var fieldValue = $(this.fieldsArray[i]).value;
    this[this.fieldsArray[i]] = fieldValue;
  }

} // End populateContact();
```

现在，那个fieldsArray成员的作用可能就很清楚了。我们遍历那个数组，因为Contact类实例的成员是与输入框的ID相同的，这样我们就可以用数组中的值来访问它们两个，因此这段代码不用知道在类和屏幕上当前都显示了哪些字段。如果我们想要添加一个字段来记录一个联系人的血型，只要把它添加到fieldsArray中就可以了，不需要修改其他填充数据的代码了。

填充屏幕显示更容易，不过使用的是完全一样的概念：

```
this.populateScreen = function() {

  for (var i = 0; i < this.fieldsArray.length; i++) {
    $(this.fieldsArray[i]).value = this[this.fieldsArray[i]];
  }

} // End populateScreen().
```

我们要看的下一个函数是toString()。回想一下，和Java类似，所有的JavaScript对象都实现了一个toString()函数。从基类Object类（也和Java类似，JavaScript里的所有对象都是从Object上继承下来的）继承下来的基础版本不一定非常有用。不过，我们可以覆盖它，并提供一些更加有用的输出。在这个例子里，输出是用JSON展现的联系信息。下面就是那个toString()函数。

```
this.toString = function() {
```

```
    var json = "";
    json += "{ ";
    // For each field in the fieldsArray, get the value and add it to the JSON.
    for (var i = 0; i < this.fieldsArray.length; i++) {
      if (json != "{ ") {
        json += ", ";
      }
      json += "\"" + this.fieldsArray[i] + "\":\"" +
        this[this.fieldsArray[i]] + "\"";
    }
    json += " }";
    return json;

  } // End toString().
```

你可能想知道，为什么这里没有用一些类似toJSON()的函数。那也可以工作得很好，只不过覆盖toString()的方法来和JSON联系更加简便一点，你在后面的DataManager类中也会看到类似的代码。而且，这样可以使调试更加方便，因为如果你想展示一个指定的Contact实例（例如，是在一个alert()弹出框中）你就会得到比那个默认的toString()提供的更有用一些信息了。这是一件好事。

我们在Contact类中看到的最后一个函数是restoreFromJSON():

```
this.restoreFromJSON = function(inJSON) {

  eval("json = (" + inJSON + ")");
  for (var i = 0; i < this.fieldsArray.length; i++) {
    this[this.fieldsArray[i]] = json[this.fieldsArray[i]];
  }

} // End restoreFromJSON().
```

名称说明一切。这个函数从一个传入的JSON字符串填充了它处理的那个Contact类的实例。我认为使用JSON来存储联系信息的好处是很明显的：代码异常简单而紧凑。我们只是再遍历一次那个简便的fileldsArray，把每个元素的值设置为从JSON对象解析出来的类中合适字段的值。这是非常简单的，也是个好事情。

顺便说一下，还记得那个我曾经提到过的临时arrayIndex字段吗？现在你知道为什么了吧？由于并没有列在fieldsArray中，它就不包含在那个使用toString()生成的JSON中，因此也就不需要使用restoreFromJSON()来恢复。

6.4.7 编写ContactManager.js

定义在ContactManager.js文件中的ContactManager类，是这个应用程序主要的代码。它的类如图6-12所示。

这个类从如下5个数据字段开始。

- eventHandlers：指向EventHandlers类实例的引用。
- dataManager：指向DataManager类实例的引用。
- currentTab：当前选定的标签页的ID。

- currentContactIndex：当前正在编辑的联系人的联系信息数组（存储在DataManager类中）的下标。
- initTimer：应用程序初始化过程中使用的计时器的引用。

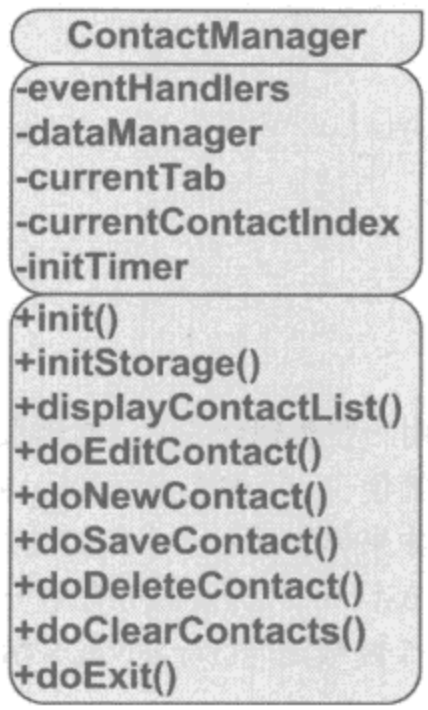


图6-12　ContactManager类的UML图表

1. 初始化

回想一下，在index.htm中，我们使用Dojo事件系统，给一个onLoad事件挂上一个对ContactManager类的init()函数的调用。那么，现在就是看看这个函数中到底有什么的时候了：

```
this.init = function() {

  contactManager.eventHandlers = new EventHandlers();
  contactManager.eventHandlers.init();
  contactManager.dataManager = new DataManager();
  this.initTimer  = setTimeout("contactManager.initStorage()", 500);

} // End init().
```

首先，将EventHandlers类实例化，并把它的一个引用存储在eventHandlers字段中。然后，在这个类上面调用init()。

之后我们对DataManager类做同样的处理，但先不像对EventHandlers的实例那样，马上调用init()。而是开启一个计时器，它每500 ms调用一次ContactManager上面的initStorage()函数。这个函数如下所示：

```
this.initStorage = function() {

  if (dojo.storage.manager.isInitialized()) {
    clearTimeout(this.initTimer);
    contactManager.dataManager.init();
    contactManager.displayContactList();
```

```
    $("divInitializing").style.display = "none";
    this.initTimer = null;
  } else {
    this.initTimer  = setTimeout("contactManager.initStorage()", 500);
  }

} // End initStorage().
```

你可能会问：这是什么？简单地说，如果在存储系统被适当地初始化之前，就尝试使用Dojo的函数来使用共享对象工作的话，将发生坏事。当我们看DataManager类时，你已经看到，在它的init()函数中完成的一件事是，将已保存的联系信息从持久存储中恢复回来（读取本地共享对象）。因此，我们不能像对EventHandlers类的处理方式一样，立即调用那个函数。实际上，直到存储系统被完全初始化后才可以这么做。这就是计时器存在的原因。每500 ms，我们使用Dojo来检测一下，存储系统是否已经完成初始化了。如果还没有，就只能继续保持计时器并等待它完成。

只要存储系统初始化完成，我们就停止计时器，并调用DataManager实例上的init()。在这时候，我们还要展现联系信息列表，这样用户就可以使用任何已存储的联系信息了，最后，我们隐藏早先展示的那个初始化信息。只要DataManager初始化完成，应用程序就为用户交互做好准备了。

2. 生成联系人

回想一下，我们是在那个初始化阶段，从持久存储中恢复的联系信息列表。这里展示了生成联系信息列表的代码：

```
this.displayContactList = function() {
// Get a list of contacts for the current tab.
var contacts = this.dataManager.listContacts(this.currentTab);

// Generate the markup for the list.
var html = "";
var alt = false;
for (var i = 0; i < contacts.length; i++) {
  html += "<div indexNum=\"" + contacts[i].arrayIndex + "\" ";
  html += "onMouseOver=\"contactManager.eventHandlers.clOver(this);\" ";
  html += "onMouseOut=\"contactManager.eventHandlers.clOut(this);\" ";
  html += "onClick=\"contactManager.doEditContact(" +
    "this.getAttribute('indexNum'));\" ";
  if (alt) {
    html += "class=\"cssContactListAlternate\" altRow=\"true\">";
    alt = false;
  } else {
    html += "class=\"cssContactListNormal\" altRow=\"false\">";
    alt = true;
  }
  html += contacts[i].lastName + ", " + contacts[i].firstName;
  html += "</div>";
}

// Display it.
```

```
    $("contactList").innerHTML = html;

  } // End displayContactList().
```

首先，从DataManager中取回联系信息列表。我们给DataManager类的listcontacts()函数传入当前选定的标签页，这样就可以过滤列表。然后，我们在返回的联系信息（返回数组中的每一个元素都是一个Contact对象）中循环，并为列表构建适当的HTML标签。列表中的每个元素都有鼠标事件负责在鼠标悬浮的时候突出显示对应联系信息。每个元素还包括一个onClick事件处理函数，它调用ContactManager类中的doEditContact()，把联系信息载入到屏幕的输入框中，以便修改。一旦完成HTML标签，就把它插入到那个contactlist 中，这个层负责将它展现给用户。

3. 编辑联系人

说到doEditContact()函数，让我们现在就看看它吧，怎么样？

```
this.doEditContact = function(inIndex) {

  // Record contact index, retrieve contact and populate screen.
  this.currentContactIndex = inIndex;
  var contact = this.dataManager.getContact(inIndex);
  contact.populateScreen();

}
```

喔，真的吗，就这些？是的，确实是。注意那个displayContactList()函数，每个列出的联系信息都有一个叫indexNum 的客户化属性附着在它上面。这个值与Contact类的arrayIndex成员的值相符。因为在选定一个标签页之后，我们看到的是联系信息数组的一个子集，使用这个值可以确定某个联系信息的indexNum属性是数组的正确下标。这样，用户点击一个联系信息并编辑它时，我们就可以拽出正确的数据来展示了。

例如，如果你点击了A~C标签，就会看到十分之三的存储信息，DataManager类的listContacts()返回的数组，将含有3个元素。但是那个数组中的脚标(0、1或2)，可能并不与给定的联系信息的contacts数组下表匹配。换句话说，listContacts()返回的首先是数组中的0脚标元素，但是实际上可能是那个主contacts信息数组中下标为9的联系信息。因此，如果我们使用返回数组的下标值作为indexNum的值，就不能在联系信息数组中获得正确的元素了（在这个例子中，是用0代替了9）。我们需要的应该是Contact对象的arrayIndex字段。

4. 添加按钮函数

ContackManager类包含了其他5个方法，每个都与页面顶端5个按钮之一相对应。首先是doNewContact()：

```
this.doNewContact = function() {

  if (this.initTimer == null) {

    if (confirm("Create New Contact\n\nYou will lose any unsaved changes.  " +
      "Are you sure?")) {
      document.forms[0].reset();
      this.currentContactIndex = -1;
```

```
    }

  }

} // End doNewContact().
```

我承认，这儿没做太多事。在这里和下一个函数中，你将注意到的一件事是检查initTimer是否为null。在应用程序开始时，initTimer的值为-1。初始化循环完成后，它就被设置为null。由于我们并不希望用户在应用程序完全初始化之前做任何操作，而Dojo在初始化完成之前就提供了鱼眼图标，所以我们需要确保初始化完成了才能处理任何用户事件。这就是为什么我们需要检查initTime是null之后（也就是所有事情都准备好了）才允许代码执行的原因。

如果initTimer是null，我们只要确保用户是想要创建一个新的联系信息，因为如果他在页面中做了任何编辑都会丢失。然后我们重置那个表单，清空所有输入框，并把currentContactIndex设置为-1，它告诉了其余的代码将创建一条新的联系信息。

紧接着是在点击保存图标时调用的函数，名为doSaveContact()：

```
this.doSaveContact = function() {

  if (this.initTimer == null) {

    // Make sure required fields are filled in.
    if ($("firstName").value == "" || $("lastName").value == "") {
      alert("First Name and Last Name are required fields");
      return false;
    }

    // Create a new contact and populate it from the entry fields.
    var contact = new Contact();
    contact.arrayIndex = this.currentContactIndex;
    contact.populateContact();

    // Save the contact.
    this.dataManager.saveContact(contact, this.currentContactIndex);

    // Redisplay the updated contact list.
    this.displayContactList();

    // Reset the entry fields and currentContactIndex.
    document.forms[0].reset();
    this.currentContactIndex = -1;

  }

} // End doSaveContact().
```

这个函数处理两个不同的保存情景：保存一个新的联系信息或保存对已存在联系信息的修改。所以，在检查了initTimer之后，首先做一个快速的修改检查，来确保名字和姓都有输入。这两个字段对

于一个联系信息来说是必须的，因为它们被用来生成那个联系信息列表，不管是新增联系信息还是编辑已存在的，这都是真的。

完成之后，我们实例化一个新的Contact对象，并通过调用它的populateContact()函数，来告诉它从输入框读取信息填充自身。我们还把arrayIndex字段设置为基于currentContactIndex的值的，那将是-1，如果它是一个新联系信息，就和在doNewContact()设定的一样；而如果是修改一个已存在的信息，就设置为合适的数组下标。

下一步，调用DataManager上的saveContact()，并把联系信息和键值传给它（我们一会儿就看那段代码）。之后，重新生成联系信息列表并展示它，因为我们可能刚添加了一个联系信息，它应该立即就可以看到（联系信息可能自动出现在我们所在的标签，或者是在那个All标签中）。最后我们重置输入框和currentContactIndex的值（这实际上有一点多余，但无关痛痒），这就是一个联系信息是如何被保存的。

下面将要看到的函数是doDeleteContact()，我想让你猜猜它是干什么的：

```
this.doDeleteContact = function() {

  if (this.initTimer == null) {

    if (this.currentContactIndex != -1 &&
      confirm("Are you sure you want to delete this contact?")) {

      // Ask the data manager to do the deletion.
      this.dataManager.deleteContact(this.currentContactIndex);

      // Redisplay the updated contact list.
      this.displayContactList();

      // Reset the entry fields and currentContactIndex.
      document.forms[0].reset();
      this.currentContactIndex = -1;

    }

  }

} // End doDeleteContact().
```

在对initTimer的常规检查和用户真的想要删除当前信息的确认（包括一个确保当前有选定删除的联系人信息的检查）之后，我们要求DataManager来执行删除操作：传给它要删除的那个联系信息的下标。和之前一样，重新生成并展示联系信息列表，重置输入框和currentContactIndex值（是的，这里也还是有点多余，但是没关系，我更喜欢任何时候都不要求已知用户输入状态的变量，因为它们更加方便调试）。

下一个函数是doClearContacts()，它是一个庞大的、发亮的、“按下即毁灭宇宙”的按钮。好，没那么夸张，但它从持久存储中删除全部联系信息，所以它并不是那些你希望用户不分青红皂白就可以点击的东西。因此这里需要有一个双重验证，如你看到的：

```
this.doClearContacts = function() {

  if (this.initTimer == null) {

    if (confirm("This will PERMANENTLY delete ALL contacts from " +
      "persistent storage\n\nAre you sure??")) {
      if (confirm("Sorry to be a nudge, but are you REALLY, REALLY SURE " +
        "you want to lose ALL your contacts FOREVER??")) {
        this.dataManager.clearContacts();
        // Redisplay now empty contact list.
        this.displayContactList();
        // Reset form for good measure.
        document.forms[0].reset();
        this.currentContactIndex = -1;
        alert("Ok, it's done.  Don't come cryin' to me later.");
      }
    }

  }

} // End doClearContacts().
```

只剩下一个函数了——doExit()，它是一段相当没必要讨论的代码：

```
this.doExit = function() {

  if (this.initTimer == null) {

    if (confirm("Exit Contact Manager\n\nAre you sure?")) {
      window.location = "goodbye.htm";
    }

  }

} // End doExit().
```

这里没有什么好玩的——只是一个简单的确认，然后把浏览器重定向到了之前看过的goodbye.htm页面。

6.4.8 编写DataManager.js

在通篇的ContactManager代码里，你都能看到一些对DataManager的调用。那么下面的要点是，那个类中我们需要关注的部分，看看其表面之下发生了什么事情。它的类图如6-13所示。

我写DataManager类的时候是带着这样一个想法写的：就是想着你可以将它换成其他的东西，比如使用一些ActiveX控制（很危险的东西）来持久化联系信息并处理下层的存储机制。因此，它的公开API是相当通用的并且可以换掉的。

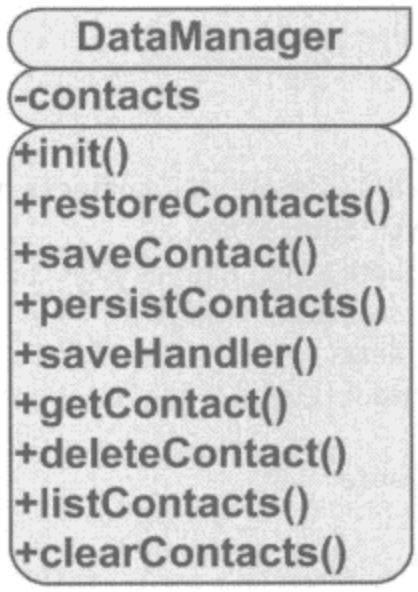


图6-13　DataManager类的UML图表

DataManager类的第一个元素是contacts数组。显然，这是我们的联系信息所存放的数组。它是在应用程序初始化的时候从共享对象中装载的数据，在用户推出之前，数组本身是与持久存储保持同步的。换句话说，当我们添加一个联系信息时，它会被添加到这个数组中，然后数组会持久化到共享对象中。当删除一个信息时，它被从数组中删除，然后数组也是持久化到共享对象中。当编辑已存在信息时，它在数组中更新，然后数组持久化到共享对象中。你看出这里的模式了吗？

在应用程序初始化中，ContactManager类被实例化后，它会初始化DataManager的实例，并保存了一个指向它的引用，就如你在前面看到的，它通过调用init()函数来实例化DataManager。我们现在可以看看这些代码：

```
this.init = function() {

  // Read in existing contacts from the applicable storage mechanism.
  this.contacts = new Array();
  this.restoreContacts();

} // End init().
```

初始化完联系信息数组后，我们调用restortContacts()，请求DataManger从持久存储中恢复联系信息，如下：

```
this.restoreContacts = function() {

  // Retrieve stored contacts.
  var storedContacts = dojo.storage.get("js_contact_manager_contacts");

  // Only do work if there actually were any contacts stored.
  if (storedContacts) {
    // Tokenize the string that was stored.
    var splitContacts = storedContacts.split("~>!<~");
    // Each element in splitContacts is a contact.
    for (var i = 0; i < splitContacts.length; i++) {
      // Instantiate a new Contact instance and populate it.
```

```
        var contact = new Contact();
        contact.restoreFromJSON(splitContacts[i]);
        contact.arrayIndex = i;
        // Add it to the array of contacts.
        this.contacts.push(contact);
      }
    }

  } // End restoreContacts().
```

恢复联系信息是相当简单的过程。首先，我们请求Dojo从本地共享对象中获取联系信息。信息以js_contact_manager_contacts的名称存储。然后代码检查确认我们真的得到了一些返回的东西。打个比方，如果应用程序是第一次在这个机器上运行，那个名字下其实就不会有任何对象，因此也就没有可恢复的联系信息。

假设这里有联系信息，我们从dojo.storage.get()调用返回得到的东西，就是简单的一个大字符串，它由JSON形式的存在联系信息组成，用一个字符序列分隔：~>!<~。我们不能只使用一个字符，比如逗号，因为它很可能出现在用户输入的数据中，那样就无法正确抽取字符串的记号了。所以我们需要一个看起来用户不太可能输入的分隔符来分割联系信息。~>!<~序列是一个相当安全的组合，因为它不大可能真的出现在用户输入中。不过，事实上，如果在一个联系信息在任何字段输入它，都会破坏代码。[①]

把字符串分解成记号之后，我们开始遍历那些记号，我要提醒你每一个记号都代表一个联系信息，都是JSON的形式出现。对于每个联系信息，所有需要我们做的就是实例化一个新的Contact对象，然后把JSON字符串传给Contact的restoreFromJSON()函数，这个函数我们前面看过了。它计算JSON数组并且填充Contact实例，这样就恢复了数据。

只剩下两件事情要做了：设置联系信息的arrayIndex字段，并把它添加到contacts数组中。只要处理完所有的记号（联系信息），restoreContacts()就完成了它的工作，我们现在有了一个联系信息数组，它与最后保存的时候是一样的。

在DataManager类的探险中，我们遇到的下一个函数是saveContact()函数，它是用来旋转页面顶端的广告横幅的。

你一直专心吧？有没有注意到我落下什么？实际上，调用saveContact()函数是用来保存一个联系信息，它看起来是这样的：

```
this.saveContact = function(inContact, inIndex) {

  // Save new contact.
  if (inIndex == -1) {
    inContact.arrayIndex = this.contacts.length;
    this.contacts.push(inContact);
  } else {
    // Update existing contact.
```

① 要想做到真正没有问题，应用程序应该检查所有的输入，保证这个序列不会出现。但是除非有人想故意破坏程序，因为看上去这个序列不太可能出现，所以我决定还是冒这个险（译注：我认为应该选择类似\t这样的单字符，然后扫描用户输入，将里面的所有\t都进行适当的转义。毕竟破坏的可能性是存在的）。

```
    this.contacts[inIndex] = inContact;
  }
  this.persistContacts();

} // End saveContact().
```

我们首先根据是否正在存储一个新的联系信息（inIndex==-1），还是在更新一个已存在的联系信息（inIndex!=-1）做了些简单的分支。在添加一个新的下标时，我们需要做的其实就是设置inContact对象的arrayIndex字段，并把它放到contacts列表中。如果是正在更新联系信息，我们就只是把contacts数组中相应的元素设置到inContact对象中。当上面的事情完成后，我们就调用persistContacts()来把contacts数组保存到共享对象中。

那么，这个persistContacts()函数是什么呢？让我们现在就看看它吧。实际上，saveHandler()是和它一起使用的，saveHandler()与persistContacts()一起来完成那个存入共享对象的工作：

```
this.persistContacts = function() {

  // First, construct a giant string from our contact list, where each
  // contact is separated by ~>!<~ (that delimiter isn't too likely to
  // naturally appear in our data I figure!)
  var contactsString = "";
  for (var i = 0; i < this.contacts.length; i++) {
    if (contactsString != "") {
      contactsString += "~>!<~";
    }
    contactsString += this.contacts[i];
  }

  try {
    dojo.storage.put("js_contact_manager_contacts", contactsString,
      this.saveHandler);
  } catch(e) {
    alert(e);
  }

} // End persistContacts().
// ********** Callback function for Flash storage system save.
this.saveHandler = function(status, keyName){

  if (status == dojo.storage.FAILED) {
    alert("A failure occurred saving contact to Flash storage");
  }

} // End saveHandler().
```

首先，我们用之前讨论restoreContacts()函数的方法，用contacts数组构造那个巨大的字符串。做法是遍历那个contacts数组，并把每个联系信息添加到一个字符串中，这会使用它自己的toString()函数，就像我们之前看Contact类时提到过的。

然后添加特殊的分割字符序列，继续上面的步骤，直到整个contacts数组都追加到这个字符串中

为止。完成之后，把这个字符串传给dojo.storage.put()函数，告诉它将字符串以js_contact_manager_contacts的名字存储。还要传给它那个指向saveHandler()函数的引用。这个函数是一个回调函数，在操作完成时被Dojo调用。我们可以检查操作的结果并做相应处理。这里，其实我们需要注意的只是错误，那种情况下，警告用户。如果发生了错误，我们并没有太多可做的，只有结束处理。如果操作成功了，也可以提示用户，但是我想如果没有任何错误信息，他们也可以猜到了。

下一个是getContact()函数，它绝对是一段简单的代码。实际上，它没什么意思了，所以我都不想展示它了。它所做的所有工作就是接受一个下标，然后返回联系信息数组中相应的那个元素。一行代码，就搞定了。

跟在getContact()之后的是deleteContact()，它还是稍微有点料的（虽然我承认，并不是太多的料）：

```
// ********** Delete a contact.
this.deleteContact = function(inIndex) {

  // Delete from contacts array.
  this.contacts.splice(inIndex, 1);

  // Store the updated contact list.
  this.persistContacts();

  // Finally, renumber all the remaining contacts.
  for (var i = 0; i < this.contacts.length; i++) {
    this.contacts[i].arrayIndex = i;
  }

} // End deleteContact().
```

JavaScript数组有splice()方法，它让我们很容易地从一个数组中删除元素。我们只是指定从哪个元素开始删除，然后指定要删除多少个元素。

联系信息被删除后，就请求DataManager来持久化我们的联系信息，就是更新共享对象。

这里留下的最后一些工作就是给联系信息重新编号。回想一下那个没有与联系信息一起持久化的arrayIndex字段，它是contacts数组中的联系信息所在的下标。它是否准确是很重要的，因为如果用户点击了一个选择标签，我们只返回contacts数组中的一个子集，每个联系信息都需要知道它们所在的是哪个下标，当用户请求它时，我们修改and/or删除相应的联系信息。

不过，假设在contacts数组中有3个信息，我们删除第二个。现在，第一个信息有一个值为0的arrayIndex，第二个信息的值为3，因为那是它之前所处的位置（假设开始时编号正确）。那么，如果用户点击了第二个信息想要编辑时，由于我们尝试访问数组中下标为3的元素，将得到一个错误，它其实并不存在了。所以，在删除一个联系信息时，需要更新arrayIndex的值。幸运地，这是个简单的处理：我们就只需要遍历那个数组，并为每一个联系信息设置arrayIndex就行了。这样将移除任何由删除操作引起的缺口，这样所有的事情又设置正确了。

deleteContact()之后是listContacts()，调用它可以获取一个contacts的子集（或在选择All标签时是整个数组）。虽然它提供了一个重要的功能，不过确实很多东西：

```
this.listContacts = function(inCurrentTab) {

  if (inCurrentTab == "XX") {
    // ALL tab selected, return ALL contact.
    return this.contacts;
  } else {
    // Filter contacts based on current tab.
    var retArray = new Array();
    var start = inCurrentTab.substr(0, 1).toUpperCase();
    var end = inCurrentTab.substr(1, 1).toUpperCase();
    for (var i = 0; i < this.contacts.length; i++) {
      var firstLetter = this.contacts[i].lastName.substr(0, 1).toUpperCase();
      if (firstLetter >= start && firstLetter <= end) {
        retArray.push(this.contacts[i]);
      }
    }
    return retArray;
  }

} // End listContacts().
```

首先，我们检查一下inCurrentTab的值是否为XX，表示选择了All标签。如果是，那只要返回contacts数组就可以了。如果是任何其他的标签，我们还有一些额外的工作要做。第一步，我们创建一个新的数组来存储将要返回的那些联系信息的子集。然后，获取inCurrentTab的第一个字符，转换成大写，这是我们想要返回的信息的范围开始标志。然后对第二个字符做同样的处理，它是结束标志。

那么，让我们假设inCurrentTab的值是AC。这时，想要返回那些姓是以A、B或C开头的所有联系信息。遍历联系人信息，抓取每一个姓氏的首字母。转换为大写后，看看它是否落在了我们指定的范围中。如果是，那就把它添加到数组中。查看了全部联系信息之后，就返回那个数组：现在它已经是适合当前所选标签的联系信息的一个子集了。

现在，我们就快完成DataManager类了。下面是最后一个需要看的函数：

```
this.clearContacts = function() {

  dojo.storage.clear();
  this.contacts = new Array();

} // End clearContacts().
```

如果用户感到沮丧，想要切断所有与外界的联系，他可能会认为自己不再需要任何联系信息了，于是可以点击Clear Contacts图标。那时，就调用clearContacts()函数。清空联系信息时需要做两件事情。首先，我们需要清除持久存储。Dojo为此提供了dojo.storage.clear()。然后，需要清除在DataManager中的联系信息数组，只要把它设置为一个新的、空的数组就可以了。然后，用户就可以去寻求专业的精神病治疗来帮忙解决他的问题了。

6.5　练习

虽然本章的主要目标是强调本项目的持久存储方面，但是并没有理由不给一些涉及其他领域的建

议。下面只是一些你可以开发的点子，它们绝对会是很好的练习。

- 允许在任何字段上的查询。还可以使用一些转换特效让查询面板滑入屏幕视图来，给项目添加一些亮点。[①]
- 允许通过点击联系信息的邮件地址来发送电子邮件。我故意把这部分留下来，因为我想把它作为相对简单的建议写在这里。这个附加功能应该不会花太大力气。
- 给列在指定标签上的联系信息排序。
- 实现用cookie的持久存储。我故意限制一个联系信息的最大容量在1024字节以下。这样应该允许你为每个cookie存储4个联系信息，所以根据每个域名下面20个cookie的限制，你可以存储80个联系信息。

6.6 小结

在本章中，我们看到了几种在客户端持久存储数据的机制。我们关注于由Adobe的Flash插件提供的本地共享对象。你看到了Dojo库是如何帮助我们处理大部分细节工作，把这些事情变得更加简单的。我们创建了一个小通讯录应用程序来演示这项技术，并且在过程中还了解了一些Dojo窗口小部件的魔术。

① 类似于幻灯片效果。——译者注

第7章

JSDigester：消除客户端XML的痛苦

7

本要站出来说一句：在浏览器中解析XML会让人很痛苦。实际上，如果你仔细想想，就知道在任何地方分析XML都有些麻烦。不过，有个库可以让解析XML变得让人能接受，这个库就是Jakarta Commons Digester组件（雅加达的通用摘要组件，http://jakarta.apache.org/commons/digester）。Digester允许你制定一些规则，它们会通过XML文档中不同的元素来触发。这些规则能以很多方式来处理解析操作，包括从XML中创建和填充对象。如果我们可以在JavaScript中做同样的事情，不是很好吗？那么，在本章中，我们就要把那个梦想变成现实，同时，还要使在客户端使用XML不那么痛苦。

7.1 在JavaScript中解析XML

在客户端解析XML，对于大多数人来说，就像“牙疼不是病，疼起来要人命”一样。相信我，我不是瞎比喻（虽然我的牙没疼过，但我已婚且有孩子，所以我很清楚那会是什么感受）。如果你已经在使用JavaScript解析XML的路上做了很多尝试的话，那么代码清单7-1中的代码看起来可能会既熟悉又痛苦。

代码清单7-1 在浏览器里用JavaScript分析XML

```
<html>

  <head>

    <link rel="StyleSheet" href="styles.css" type="text/css">

    <title>Simple JavaScript XML Parsing Example</title>

    <script>

      function doParsing() {
      // This is the XML we will parse.
      var xml = "<messages>";
      xml += "<msg poster=\"Frank\">Hello!</msg>";
      xml += "<msg poster=\"Traci\">I hope all is well with you!</msg>";
```

```
xml += "<msg poster=\"Andrew\">Well, I guess that's it.</msg>";
xml += "<msg poster=\"Ashley\">Have a good day!</msg>";
xml += "</messages>";

// Instantiate an XML parser (or DOM, depending on browser).
var xmlDoc = null;
if (window.XMLHttpRequest){
  var parser = new DOMParser();
  xmlDoc = parser.parseFromString(xml, "application/xml");
} else {
  xmlDoc = new ActiveXObject("Microsoft.XMLDOM");
  xmlDoc.async = false;
  xmlDoc.loadXML(xml);
}

// Now iterate over the DOM created above, and construct an output
// string to display.
var strOut = "Root node = " + xmlDoc.documentElement.nodeName + "<br>";
for (var i = 0; i < xmlDoc.documentElement.childNodes.length; i++) {
  strOut += "nodeName = " +
    xmlDoc.documentElement.childNodes[i].nodeName + ", poster = " +
    xmlDoc.documentElement.childNodes[i].getAttribute("poster") +
    ", text = " +
    xmlDoc.documentElement.childNodes[i].firstChild.nodeValue + "<br>";
}
    document.getElementById("divOut").innerHTML = strOut;

  }

  </script>

</head>

<body class="cssBody">

  <div class="cssTitle">JavaScript XML Parsing Example</div>
  <br><br>
  <input type="button" value="Click me to parse XML"
    onClick="doParsing();" class="cssBody">
  <br><br>
  Info will appear here:
    <br><br>
    <div id="divOut" class="cssLog"></div>

  </body>

</html>
```

我不得不承认这里头可能没什么可看的，图7-1展示了代码清单7-1中代码的输出。

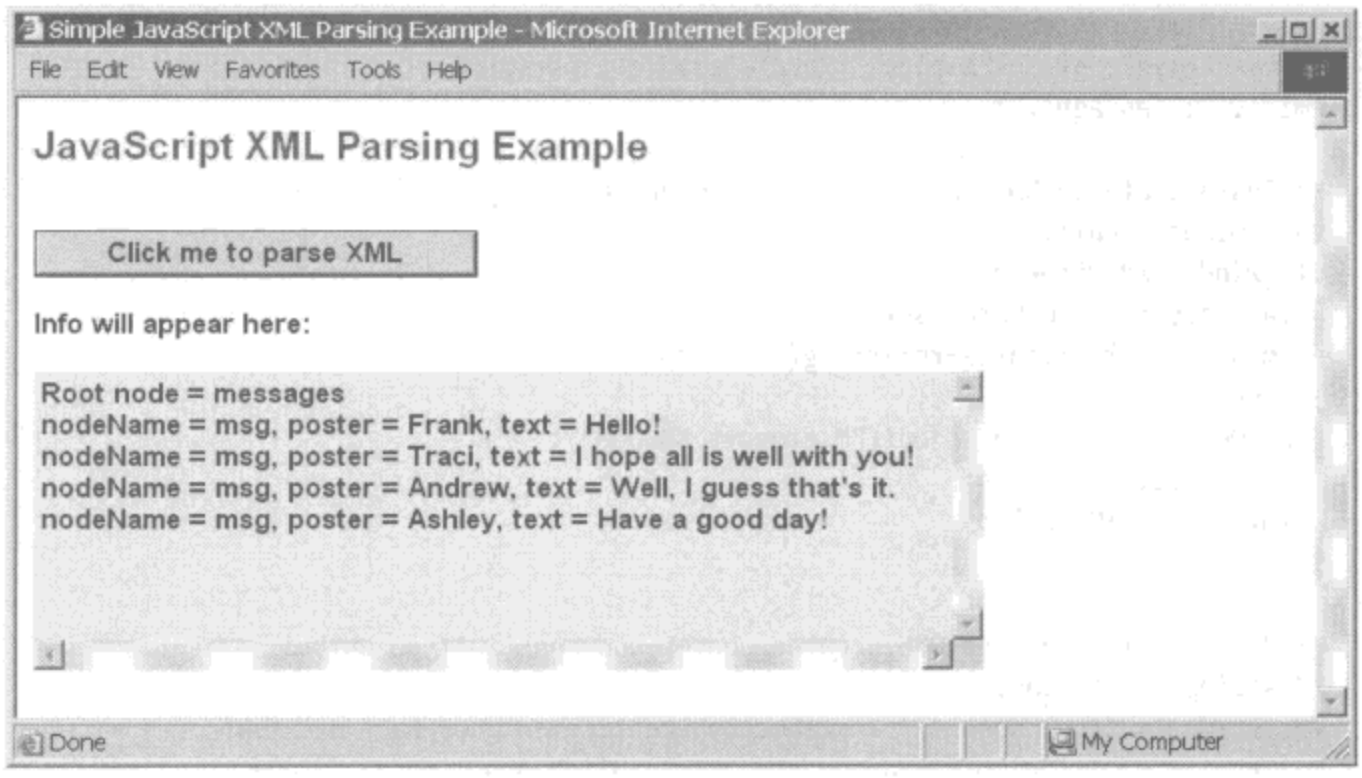


图7-1　简单的解析程序样例的结果

请记住这只是个非常简单的例子。而且在JavaScript中有不止一种方法可以写XML解析代码。不过，我想可以很放心地说，所有的方法都有同一个缺陷：要做的事情实在太多了。不仅这样，你注意到了吗？代码需要知道它正在解析的XML的格式和结构。这通常并不算坏事，当然，有时候也是必要的。但我们还是有必要尽可能多一些限制，那样代码就可以很灵活而且可重用了。在这种情况下，引用指定的元素名可能并不那么理想。

另外，一个非常需要考虑的问题是你到底想要对正在解析的这个XML做什么处理？你只是简单地扫描一遍，并对每一个元素进行操作，比如展示它们吗？或者你的目的是用这个XML填充一些对象——类似于一个对象-XML映射的东西？如果你的目标是前者，那么像代码清单7-1中那样的代码可能不是太坏，而且可能更高效。不过，如果你是要创建和填充对象，那么就需要写很大一堆其他的代码来实现。

7.2　JSDigester需求和目标

本章的项目旨在使我们的生活更加简单。我们将从Apache的Jakarta通用摘要组件项目中获得提示，并创建自己的以JavaScript实现的摘要，我们将把它叫做JSDigester。

下面是我们的目标和需求。

- 与其尝试去实现Digester全部的功能，不如让我们就只涵盖一小部分。不过，像Digester一样，JSDigester也应该是可扩展的，所以如果我们需要沿着这条路走下去，再添加一些规则的话，就可以毫不费力地实现。
- 简单地说，我们应该可以创建一个JSDigester的实例，给它配置一些规则，然后传给它需要解析的XML。结果大概就应该是一个包括了那些从XML解析生成的对象的对象。
- JSDigester看起来、用起来都应该尽可能的像它的大哥哥——Digester，基于这个理解，如果有些东西转换成JavaScript太困难，或者是根本不可能，那么我们就不会尝试去实现。

- 自然地，JSDigester应该是完全浏览器兼容的。

带着这些目标，让我们开始吧，试着把我们自己从使用JavaScript解析XML的那种被虐待的感受中解脱出来吧。

7.3 Digester如何运转

为了复制通用摘要组件，我们需要看看那个项目本身。虽然Digester是一个Java的库，我想一个基本的例子就可以让大部分使用过C语言的开发者理解了。

Digester的主页非常好地描述了什么是Digester（它也不得不描述，对吧？）：

基本上，Digester包让你配置一个XML->Java对象的映射模块，它会在识别出一个特定的套嵌XML元素的模式时触发特定的动作。里面有一大堆你可以使用的预定义的规则，你也可以创建自己的。Digester的高级特性包括以下这些。

- 能插入你自己的模式匹配引擎，如果标准的配置不能满足需求的话。
- 可选的识别命名空间的处理，让你可以定义那些只和某个特定的XML命名空间相关的规则。
- 将规则封装到规则集（RuleSet）中，让它可以在不止一个的、需要统一处理类型的应用程序中简单方便地重用。

在Digester里面的基本想法是，你可以将特定的XML元素映射为一些定义如何处理它们的规则上。然后，解析一个XML文档的时候，Digester遇到相应的元素，它就为这些元素创建对象，设置指定对象的属性或调用一个对象的函数——所有这些都取决于你定义的规则。当XML文档全部分析完时，你就收获了一个对象图，它包含代表XML文档中各种各样元素的新对象。

Digester看上去有些高射炮打蚊子，但是只要你一拿起它，就再也不想使用任何其他的了！让我们现在就看一个小例子吧。

设想我们正在写一个简单的购物车Web应用程序，就像在一些像Amazon.com那样的电子商务网站看到过的一样。假设我们的应用程序有两个类，分别展示在代码清单7-2和代码清单7-3中，它们负责展现那个购物车和在购物车中的任何一个条目。

代码清单7-2 ShoppingCart类

```
package myApp;
public class ShoppingCart {
    public void addItem(Item item);
    public item getItem(int id);
    public Iterator getItems();
    public String getShopperName();
    public void setShopperName(String shopperName);
  }
```

代码清单7-3 Item类

```
package myApp;
public class Item {
  public int getId();
```

```
    public void setId(int id);
    public String getDescription();
    public void setDescription(String description);
}
```

现在让我们假设，之前已经有一个用户开始了购物，可能在给他的朋友买一些圣诞礼物，他在购物车中放弃了一些商品条目，然后离开了。进一步假设，我们想让那个用户有一个很好的购物体验，所以为他把购物车的状态保存了一段时间。最后，假设我们是使用代码清单7-4中展示的XML文档保存的那个状态。

代码清单7-4　保存下来的代表用户的购物车的XML文档

```
<ShoppingCart shopperName="Rick Wakeman">
  <Item id="10" description="Child's bike (boys)" />
  <Item id="11" description="Red Blouse" />
  <Item id="12" description="Reciprocating Saw" />
</ShoppingCart>
```

现在，当用户回到我们的网站时，我们希望从这个XML文档中读取并创建ShoppingCart对象，它包括了3个Item对象，它们所有的属性都被设定为XML文档中相应的数据。我们使用如下的Digester代码来这么做：

```
Digester digester = new Digester();
digester.setValidating(false);
digester.addObjectCreate("ShoppingCart", "myApp.ShoppingCart");
digester.addSetProperties("ShoppingCart");
digester.addObjectCreate("ShoppingCart/Item", "myApp.Item");
digester.addSetProperties("ShoppingCart/Item");
digester.addSetNext("ShoppingCart/Item", "addItem", "myApp.Item");
ShoppingCart shoppingCart = (ShoppingCart)digester.parse();
```

我们分解一下，看看每行代码。第一行实例化了一个Digester对象。第二行告诉Digester我们并不想要XML文档对DTD（Document Type Definition，文档类型定义）做验证。

然后是一系列的addXXX()方法调用，它们每个都向Digester添加一个特定的规则。有不少内置规则可用，如果需要，你也可以自己写。

所有规则共享方法的第一个调用参数：将会触发规则的那个元素的路径。要知道XML文档是层级的树状结构，所以要获得文档中的任意指定元素，就形成了一个从根开始，包含所有该元素祖先节点的路径。换句话说，看看<Item>元素，所有的<Item>元素的父节点都是<ShoppingCart>元素。因此，对于任何<Item>元素来说，完整的路径就是ShoppingCart/Item。按同样的方式，如果<Item>元素自身还有一个内嵌于其下的元素，比如说是<Price>，那么到这个元素的路径就应该是 ShoppingCart/Item/Price。

Digester规则是附加在指定路径上的，任何时候，带该路径的元素都会触发该规则。你可以给某个路径附加多重规则，多重规则可以在任何给定的路径上触发。

在这个例子中，我们的第一个规则（ObjectCreate规则）是用来处理ShoppingCart路径的。意思是，当遇到<ShoppingCart>元素时，就会创建一个myApp.ShoppingCart类的实例。

Digester使用一个栈来实现对它创建的对象的处理操作。例如，当ObjectCreate规则被触发，并且实例化了那个ShoppingCart对象，它就被推入栈。后面所有的子规则都将基于那个对象来工作，直到它从栈中弹出（可能是因为其他规则显示弹出，也可能是因为解析结束）。那么，下一条规则（SetProperties规则）被触发时，它会设置栈顶部对象的所有属性（在这个例子里是ShoppingCart对象）是文档中使用<ShoppingCart>元素的属性。

后面跟着的是另一个ObjectCreate规则，它负责设置<Item>元素，对这个元素还有另外一个规则SetProperties。所以，当遇到第一个<Item>的时候，对象被创建并推入栈里，也就是说，现在它就位于ShoppingCart对象的顶部了。

最后一个规则是SetNext规则。这个规则调用栈中下一个对象的给定的方法——在这个例子里是addItem()，那个对象是ShoppingCart对象，把栈顶部的对象（那个Item对象）传给它。最后，将栈顶部的Item对象弹出，露出ShoppingCart对象，它又一次回到栈的最上面。把这个过程重复3遍，处理每一个<Item>元素。

在最后，Digester弹出并放回栈顶部的那个对象，也就是当时的ShoppingCart对象。我们把返回结果放到shoppingCart变量中，现在就完全是和用户离开时留下的那个购物车状态相同的再生购物车了。

有趣的是，Digester在内部使用的是SAX（Simple API for XML）。SAX是一个基于Java的、事件驱动的、用于解析XML的API，类似Digester（很合理！），但是在更底层，并且也比Digester做的事情多了不少。由于这本书尽可能地使用语言无关（当然除了JavaScript之外）的代码，所以我就不介绍那些直接使用SAX的例子了。不过，我们要知道，SAX在解析一个XML文档的时候，会发生各种各样的事件，这点很重要。事件可能是发生在文档开始或结束时，遇到一个新的标签时，解析一个标签对中的文字时，以及XML标签关闭时等。作为开发人员，你要编写叫做文档处理函数的类，它在这些事件发生时将被调用。Digester其实就是一个文档处理函数类。它在SAX解析事件的基础上创建，并做了很有意义的扩展。

虽然创建我们自己的JavaScript的SAX实现是非常可行的，但如果已经有人做了那我们还麻烦什么？来自Mozilla基金会的JSLib（http://jslib.mozdev.org）就提供了这个实现。顾名思义，JSLib是一个涵盖了一定范围的JavaScript函数库，这个范围就包含SAX，当然，这是因为人们时常需要一些性事（得了吧，我怎么能不在本章用SAX开个玩笑呢）。

一会儿在我们分解JSDigester的时候，你就会看到JSLib的SAX组件是如何使用的。好吧，我们还在等什么呢？

7.4 剖析JSDigester的解决方案

让我们从执行测试代码开始，看看JSDigester实际上做了什么。图7-2展示了当你在浏览器中载入测试页面并点击按钮来初始化测试程序时看到的屏幕。

在我们看JSDigester本身的代码之前，看看测试JSDigester时使用的代码是很自然的事情。

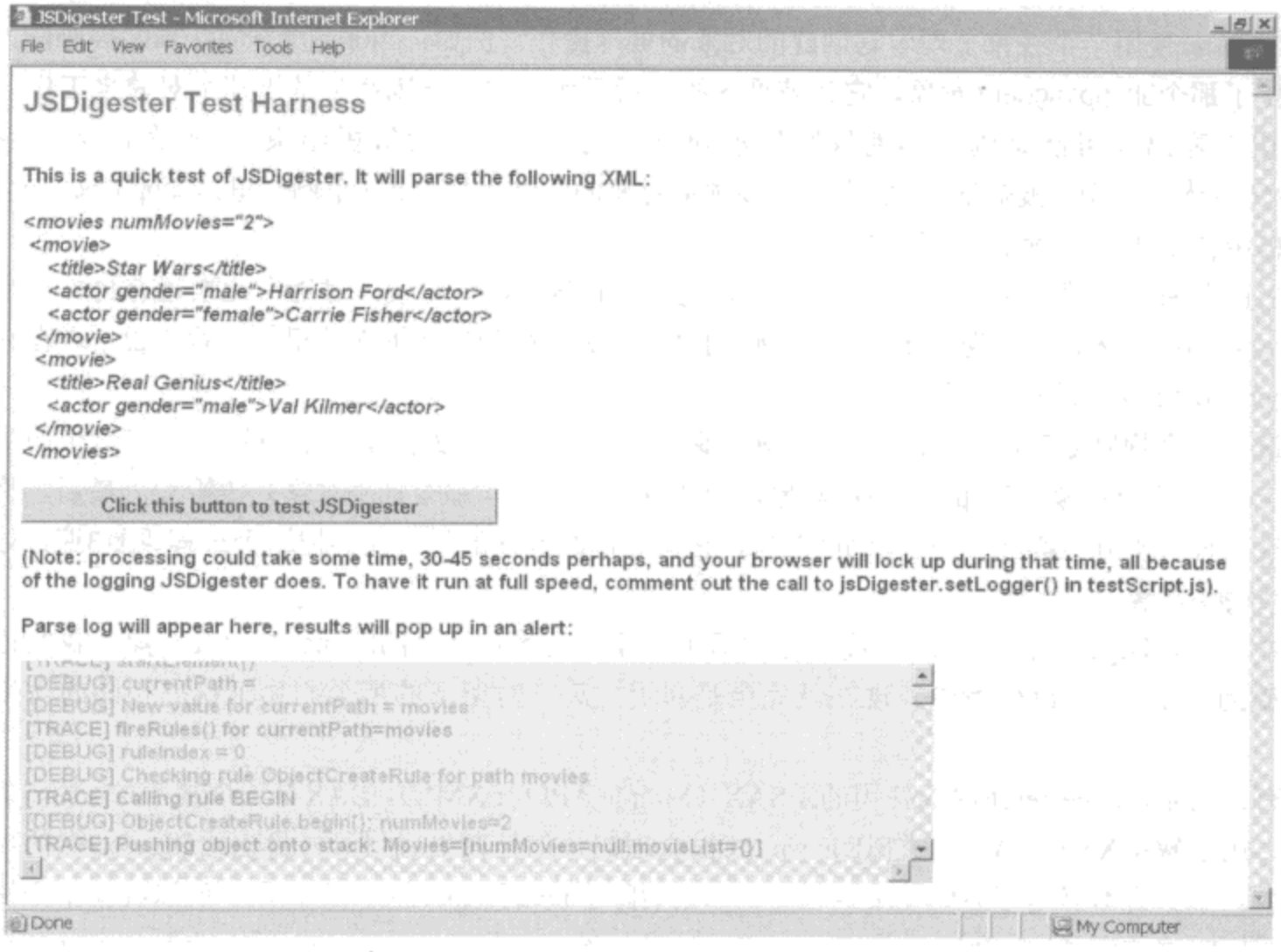


图7-2　测试JSDigester的结果

7.4.1　编写测试代码

JSDigesterTest.htm文件是测试的入口，你将要在浏览器中载入的就是这个文件。为了节省空间，我并不打算在这里展示这段代码，不过你应该去看一眼。它的确只是一些直接的HTML，其中包括一些JavaScript导入命令。在JavaScript文件中发生的事情才是真正有意思的。

首先导入的sax.js是JSLib中的SAX解析器。这里就不详细介绍这个代码如何工作了，因为那并不是我写的代码。但是我非常推荐你自己去看一遍，有个整体的感觉。通常，它并不真的非常难以理解，就只有200多行代码而已，本身并不大。它是我们用的一个库，像我们平常对待库一样，我们很少会涉及它里面究竟是怎么工作的、如何产生出魔力的。我们会更多地关心在自己的代码中应该如何使用它，所以这是我们关注的。

不过，在使用JSLib的时候，你需要知道一个细节，就是你要提供一个DocumentHandler回调。它实际上是一个响应具有良好定义的生命周期的事件的对象。这些事件是在把XML当作一个流解析的时候碰到的。这个对象必须遵守一个已知的接口；也就是，必须实现一系列函数。你一会儿就会看到函数是如何实现的，但是只要我们使用JSLib，这些就是你需要的扩展知识。

继续阅读代码，看看代码清单7-5。它显示了3个JavaScript类：一个类代表一系列电影的集合，另外一个代表一部电影，第三个代表演员。这些代码在testClasses.js文件里。这些类就是JSDigester在工作的时候会初始化和填充的类。

代码清单7-5 JSDigester会初始化和填充的类

```
// This class represents an Actor in a Movie.
function Actor() {
  this.gender = null;
  this.name = null;
}
Actor.prototype.setGender = function(inGender) {
  this.gender = inGender;
}
Actor.prototype.getGender = function() {
  return this.gender;
}
Actor.prototype.setName = function(inName) {
  this.name = inName;
}
Actor.prototype.getName = function() {
  return this.name;
}
Actor.prototype.toString = function() {
  return "Actor=[name=" + this.name + ",gender=" + this.gender + "]";
}


// This class represents a Movie.
function Movie() {
  this.title = null;
  this.actors = new Array();
}
Movie.prototype.setTitle = function(inTitle) {
  this.title = inTitle;
}
Movie.prototype.getTitle = function() {
  return this.title;
}
Movie.prototype.addActor = function(inActor) {
  this.actors.push(inActor);
}
Movie.prototype.getActors = function() {
  return this.actors;
}
Movie.prototype.toString = function() {
  return "Movie=[title=" + this.title + ",actors={" + this.actors + "}]";
}


// This class stores a collection of Movies.
function Movies() {
  this.movieList = new Array();
```

```
  this.numMovies = null;
}
Movies.prototype.setNumMovies = function(inNumMovies) {
  this.numMovies = inNumMovies;
}
Movies.prototype.getNumMovies = function() {
  return this.numMovies;
}
Movies.prototype.addMovie = function(inMovie) {
  this.movieList.push(inMovie);
}
Movies.prototype.getMovieList = function() {
  return this.movieList;
}
Movies.prototype.toString = function() {
  return "Movies=[numMovies=" + this.numMovies + ",movieList={" +
  this.movieList + "}]";
}
```

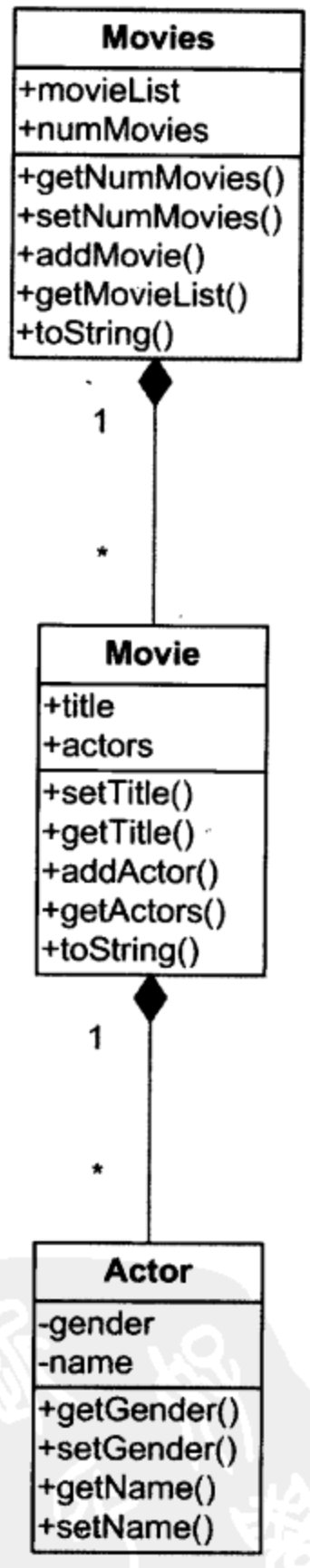


图7-3　用UML表示的测试类的层级结构

图7-3展示了一点点UML图，你可以形象地看到这个小类的层级关系。

这3个类在行为上并没有太多东西，它们多多少少都只是一个存储数据的容器。

Actor类有两个属性：gender和name，同时还有与每一个相连的读写器方法（如果你熟悉Java的话，会发现这是典型的JavaBean）。Actor类还提供了一个覆盖了的toString()方法，默认情况下，JavaScript中所有对象都有这个方法。新版本的toString()方法为Actor对象提供了一个更有意义的字符串形式。

Movie类有两个属性：title和actors。actors是一个与Movie相关的Actor对象的数组。这个类也有一个覆盖了的toString()方法。

最后，Movies类也有两个属性：movieList和numMovie。movieList是一个Movie对象的数组，numMovies是在那个数组中包含的电影的数量。虽然这个信息其实是数组的固有属性，在length属性里，但是我用这种方法是想来举例说明JSDigester的一些解析功能（所以不要在这上面想得太多，因为它并不表示这是最好的实现方法）。和Actor以及Movie类相同，Movies也有它自己的toString()方法，你可以在运行测试代码的时候看到它的输出。

现在，让我们看看执行JSDigester的测试代码吧。这段代码如代码清单7-6中所示，你也可以在testScript.js文件中找到。

代码清单7-6 测试JSDigester的JavaScript代码

```
function testJSDigester() {

  // Create a string of test XML to have JSDigester parse.
  var sampleXML = "";
  sampleXML += "<movies numMovies=\"2\">\n";
  sampleXML += "  <movie>\n";
  sampleXML += "    <title>Star Wars</title>\n";
  sampleXML += "    <actor gender=\"male\">Harrison Ford</actor>\n";
  sampleXML += "    <actor gender=\"female\">Carrie Fisher</actor>\n";
  sampleXML += "  </movie>\n";
  sampleXML += "  <movie>\n";
  sampleXML += "    <title>Real Genius</title>\n";
  sampleXML += "    <actor gender=\"male\">Val Kilmer</actor>\n";
  sampleXML += "  </movie>\n";
  sampleXML += "</movies>";

  // Create a logger for JSDigester to use, and set its level to TRACE, and tell
  // it where to log to.
  var log = new jscript.debug.DivLogger();
  log.setLevel(log.LEVEL_TRACE);
  log.setTargetDiv(document.getElementById("divLog"));

  // Instantiate a JSDigester instance and set up the rules, and logger.
  var jsDigester = new JSDigester();
  jsDigester.setLogger(log);
  jsDigester.addObjectCreate("movies", "Movies");
  jsDigester.addSetProperties("movies");
  jsDigester.addObjectCreate("movies/movie", "Movie");
  jsDigester.addBeanPropertySetter("movies/movie/title", "setTitle");
  jsDigester.addObjectCreate("movies/movie/actor", "Actor");
  jsDigester.addSetProperties("movies/movie/actor");
  jsDigester.addBeanPropertySetter("movies/movie/actor", "setName");
  jsDigester.addSetNext("movies/movie/actor", "addActor");
  jsDigester.addSetNext("movies/movie", "addMovie");

  // Parse the XML, resulting in an instance of the Movies class.
  var myMovies = jsDigester.parse(sampleXML);

  // Construct result string.
  var outStr = "JSDigester processed the specified XML." +
    "\n\nIt created an object graph consisting of a Movies object, " +
    "with a numMovies property, and containing a collection of " +
    "Movie objects." +
    "\n\nEach Movie object has a title property, and " +
    "contains a collection of Actor objects.\n\n" +
    "Each Actor object has two fields, name and gender.\n\n" +
    "Here's the final Movies object JSDigester returned: \n\n" +
```

```
        myMovies;

      // Display results.
      alert(outStr);

    }
```

在JSDigesterTest.htm里的按钮的onClick事件调用了这里看到的testJSDigester()函数。它做的第一件事是创建一个XMLJSDigester将要为我们解析的字符串。我相信那些都不需要加以说明了。

在那之后，是一个DivLogger实例的初始化和配置（DivLogger在第3章中介绍过）。我们把日志级别（logging level）设置为trace，这样就可以非常清楚地看到JSDigester所做的一切。然后我们赋给它一个指向日志输出的那个 的引用。

接下来，我们实例化JSDigester本身，并把刚刚配置好的日志记录程序传给它。注意，如果你没有给JSDigester传日志记录程序，它也还是可以工作的。实际上，如果是在生产环境，我非常推荐你这么做。当你真的尝试那个测试页面的时候，你就会看到，日志大大降低了JSDigester的速度。因此，通常只有在调试时或者可以接受日志记录引发的延迟的情况下，才要把日志记录程序实例传给JSDigester。在不记录日志的时候，JSDigester实际上性能相当不错，只要输入的XML不是太长。打开日志记录的话，它的性能至少是没原来好。

然后，是真正让JSDigester工作的部分：规则。配置的第一个规则是ObjectCreateRule，它将在XML中遇到<movies>的时候使用。不用说，结果是创建了一个Movies类的实例。和它相关的是下一个规则——SetProperties规则。这个规则会为每一个<movies>标签调用最新创建的那个Movies对象上的设置器方法（setter method）——在这个例子中就是numMovies。

之后，我们看到另一个ObjectCreateRule，这次它映射到路径movies/movie。在遇到<movies>元素的任何一个子<movie>元素的时候，它都会创建一个Movie类的实例。接下来的BeanPropertySetter规则与它一起合作。它在遇到路径movies/movie/title时触发，那个路径把一个<title>元素连接在<movie>的子节点。它将调用刚刚创建的那个Movie对象的setTitle()方法，并把这个<title>元素的值传给它。

然后是三个一组的规则，它们处理<actor>元素。首先，可能就像你到现在所期待的那样，是ObjectCreateRule。然后是SetPropertiesRule。所以，在这点，JSDigester知道何时以及如何为那个Actor类创建实例，并且它还知道获得每一个<actor>元素的属性，以及调用相应的设置器方法来设置那些属性。

最后，我们有另外一个BeanPropertySetter规则映射到movies/movie/actor。包含在<actor>的开始标签和结束标签之间的文本是演员的名字。所以，这个规则将取到那个文字值，并调用Actor实例上的setName()方法，就像规则所指定的。

到现在为止，JSDigester已经有足够的信息来为我们创建对象并从XML填充它们。然后是最后一步，让JSDigester知道对象的层级关系，也就是说，如何将Actor对象分配给Movie对象，以及如何给那个最终返回给调用者的Movies对象添加Movie对象。

停一下！稍等——让我们不要掩饰它了。为什么在它处理的最后阶段，JSDigester会给我们返回那个Movies实例？让我们全面地来了解一下这是怎么发生的。

7.4.2 理解JSDigester的整体流程

还记得前面讨论Digester时，我说过它是用了一个栈来实现的吗？好的，下面就是我们最后如何获得那个Movies实例的方法。每次Digester创建一个新的对象时，它都把新的对象推入栈。那是一个先进后出（FILO）的栈，也就是说，后面创建的对象放入栈内时，第一个对象就总是位于底部了。因此，在解析XML时，对象被弹出栈后，最后一个对象最终会暴露出来，那个就是最早被创建的对象，它也就是JSDigester返回的东西。

想想我们配置的第一条规则，它映射为<movies>元素，而<movies>元素恰好是XML文档的根节点。所以，最后一个被返回的对象是根对象很合理，不是吗？所有其他的对象都是包裹在根节点里的，就像XML文档的结构表示的那样。

那么，一个Actor对象是如何添加到一个Moive对象上呢？Movie对象又是如何被添加到Movies对象上的？这两件事情都是SetNext规则的效果，这个规则是JSDigester的最后两个规则。SetNext规则使用栈天生的优点，在栈中的下一个对象调用指定的设置器方法。为了理解这个概念，让我们看看解析测试XML时发生的一系列事件。图7-4展示了事件的流程图。

到这儿，你应该比较熟悉JSDigester运转的方式了，并且了解了测试它的代码。只剩下一个微不足道的细节：查看JSDigester的代码。

7.4.3 编写JSDigester代码

JSDigester有678行代码，[①]我就不在这里全部列出了。在继续后面的内容的过程中，我会点出一些比较有意思的部分，所以打开一个完整版本的代码清单会是很有用的。

1. 准备好（准备处理）……

开头15行左右JSDigester.js文件的实际代码是定义JSDigester（以及规则类）类的地方，那里是一堆类成员。前3个是一些常量，用来判断正在处理的XML中一个特定元素上正在发生什么事件：元素的开始（EVENT_BEGIN）、元素的内部文字（EVENT_BODY）、元素的结束（EVENT_END）。当然，在JavaScript中，文字上，使用“常量”是用词不当的，因为JavaScript并没有真正的常量；代码可以改变这些值的。不过概念上，还是仍然把它们叫做常量，也当作常量来看待。

之后是更常见的变量：currentPath。回想一下我们分析XML的过程，每个元素都可以用一个路径来引用，比如前面例子中的movies/movie/actor就指向了根元素<movies>的子元素<movie>下面的<actor>元素。currentPath是JSDigester在分析XML过程中定位自己的位置的变量。

下一个看到的，是名叫rules的数组。它是加载在这个待运行的JSDigester实例上的所有规则的集合。

然后是我们的objectStack，当然，它将是JSDigester所有行为的核心。

rootObject是接下来要看到的，它指向栈的根对象。这么做是为了我们在需要的时候方便地返回根对象。这将会随着XML的处理发生变化。因为随着元素从栈中弹出来，事实上每个元素都当过根对象——至少是暂时的。虽然到最后，它会指向真正的根对象，然后就可以返回这个对象了。

① 有趣的是，大概一半的代码（准确说是316行）是可执行代码。其他都是空白和注释——我觉得并不是想象的那么多。

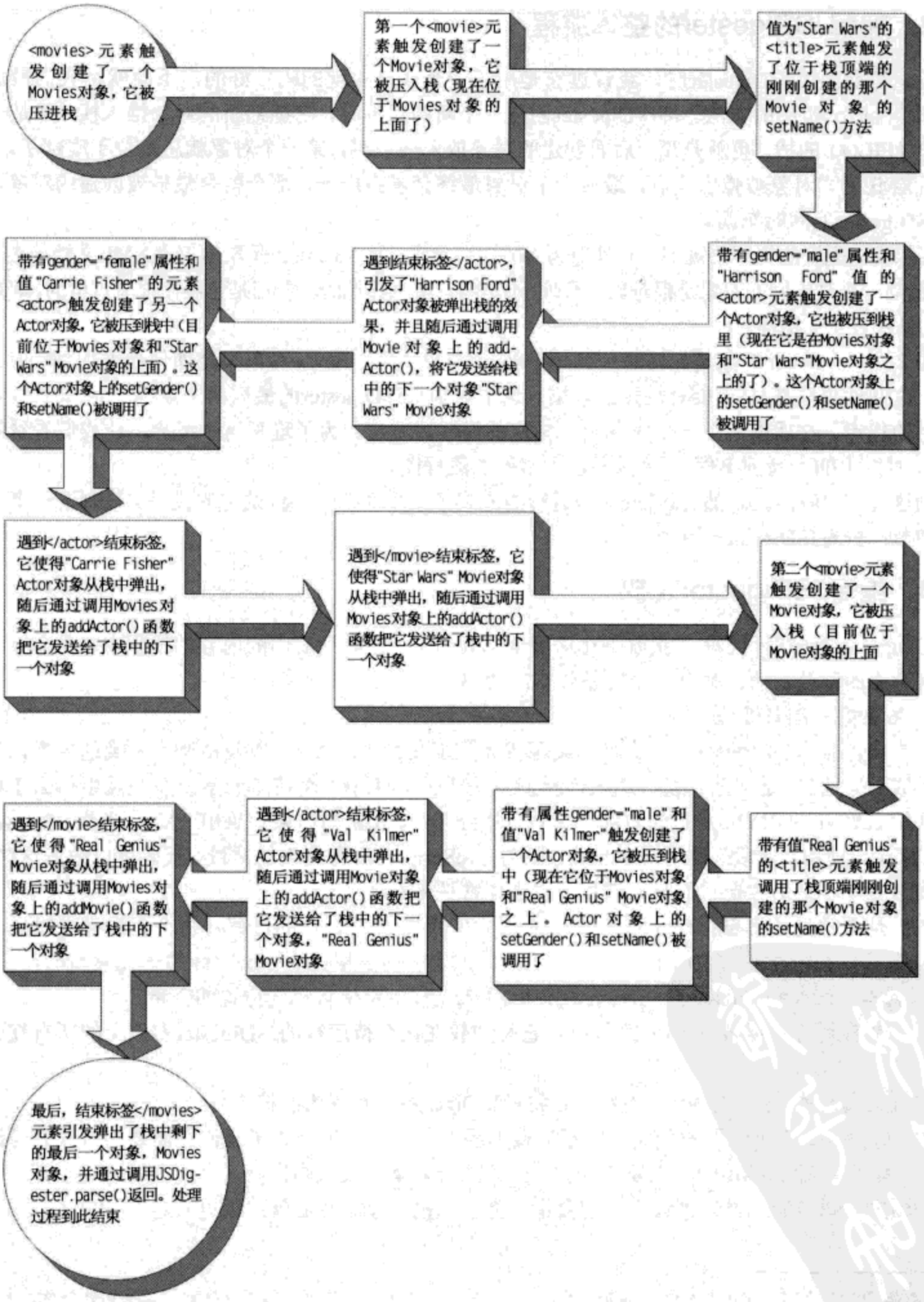


图7-4　JSDigester分析过程的一系列事件

跟在rootObject后面的是变量log。我想，你会推测这个变量是指向JSDigester在处理过程中的日志记录程序的实例的引用。在讨论测试代码的时候，我们已经说过，日志是开销非常大的操作，尤其是在这个实现中，因为我们需要不停地重写某个 元素的innerHTML属性。因此，在默认的情况下，log设置为空，意思是没有日志。也就是说，在默认状态下，JSDigester的实例是配置为性能最优的。

下一个类成员是saxParser，它是一个指向JSLib的SAX解析器实例的引用，它会为我们分析XML文件。请注意，一个新的实例就是此时此地生成并且赋予这个变量的——没理由把它放一边。不过，你会发现这么做的一个明显的问题：如果我们想使用另外一个SAX解析器，那就没有好的办法了。使用JSDigester的代码可以把saxParser设置为有着和SAXParser一样接口的另外一个对象，但这个回答决不能令人满意。就像Shadows[①]的第一次出现一样，你肯定会有某种预感：我肯定要在后面的练习里说点什么。

在所有这些类变量声明和初始化之后，是JSDigester构造器函数的最后两行：一个init()调用。init()函数很简单，也很重要：

```
JSDigester.prototype.init = function() {

  // Tell the SAX parser that this instance of JSDigester is the document
  // handler.
  this.saxParser.setDocumentHandler(this);

} // End init().
```

在JavaScript中，一种创建类的方法是使用prototype。每一个函数其实就是一种对象，而每个对象都含有与其相对应的prototype。通过给已存在的JSDigester函数的prototype分配一个字段——在这个例子中是init，并赋给它一个函数的值（我知道这听起来有点奇怪），我们就实际上在把一个名叫init()的函数添加到JSDigester上。不过，由于我们是给JSDigester的prototype添加函数，意思是说每次实例化新的对象，它都是会包含那个init成员，相当于是一个函数。这是JavaScript中一种普遍的先创建对象然后扩展它们的方法。

这里更加奇怪的是，即便是在初始化一个JSDigester对象之前，你就已经有了一个实例了，这是拜函数定义所赐，函数定义包含了所有成员变量。不过，如果你在代码执行前查看它，你会发现，它其实并不包含那个init()函数。只有那些在这段代码执行后实例化的JSDigester对象才有那个函数。请记住，JavaScript是非常动态化的语言，类似这里发生的这样不寻常的事情在JavaScript中是很常见的。

继续往下看，回想一下，SAX解析器才是真正负责解析XML的。不过，遇到某个元素时应该发生什么，仍然是我们要做的。在SAX的用法中，你需要告诉解析器，哪个类将会扮演文档处理函数的角色。哪个类要在整个处理过程中碰到各种触发事件的时候接收回调函数，比如第一次遇到某个元素、遇到两个标签之间的文字或者遇到一个结束标签。JSDigester本身在这里就是一个文档处理器，因为它将处理那些事件来实现自己的逻辑，所以我们把隐含的this引用传给saxParser实例的setDocumentHandler()。上面这些就是需要为JSDigester做的全部初始化工作。

① Shadows是电视剧《巴比伦5》中古老的、技术更先进的种族。在第一季的*Signs and Portents*里，我们第一次看到了Shadows，了解了它们有多么强大。这个季的《巴比伦5》是真正抓住很多粉丝的心的一季。它是将来临的事件的强烈暗示。可以看看：http://www.imdb.com/title/tt0517690。

在init()方法之后是setLogger()方法。这是一个典型的属性设置方法，它接受一个指向日志记录程序实例的引用，供JSDigester使用。

2. 设置（开始主流程）...

然后是parse()方法，JSDigester所有的企图和目标的主要入口点。让我们看看它吧，如何？代码清单7-7展示了那段代码。

代码清单7-7　JSDigester所有的动作开始的地方：parse()方法

```
JSDigester.prototype.parse = function(inXMLString) {

  if (this.log) {
    this.log.trace("JSDigester.parse()...");
    this.log.debug("inXMLString = " + inXMLString);
  }
  // Remove all items from the object stack, just in case this isn't the
  // first parse this instance of JSDigester has performed.
  this.objectStack.splice(0);
  // Clear current path and root object.
  this.currentPath = "";
  this.rootObject = null;
  // Ask the SAX parser to parse the incoming XML.
  if (this.log) {
    this.log.debug("Calling SAX parser...");
  }
  this.saxParser.parse(inXMLString);
  if (this.log) {
    this.log.debug("SAX parser returned");
  }
  // Return the root object on the stack.
  if (this.log) {
    this.log.debug("Returning root object: " + this.rootObject);
  }
  return this.rootObject;

} // End parse().
```

第一行是一些你会在整个代码中多次看到的东西。还记得我说过，当没有日志记录程序（logger）的时候，JSDigester也还是会继续工作，实际上是会工作得更快？这行代码就是让那个观点变为现实的。这类似于Java中的代码保护，就是你要在尝试记录一些信息之前，先检查某个特定的日志级别是否可用，这种方式通常开销比较大，它一般要使用字符串拼接的方法。这样做就避免了日志调用的开销，在这里也是一样的。除了让它更加有效之外，还很明显可以避免错误。如果我们没有传入日志记录程序实例，在这个例子中，我们就会得到试图引用一个空对象的错误——this.log引用变量为空。

在日志信息的开场白之后，我们发现了下面这行代码：

```
this.objectStack.splice(0);
```

splice()方法是用来从数组中移除某个元素的。在传入一个0的情况下，运行结果就是将整个数组清空。在这里我们这么做是因为如果这个JSDigester实例需要被重用，那么栈就需要预先清空。同样的

逻辑，下面两行重置了currentPath和rootObject到初始状态。如果规则相同的话，重用一个JSDigester可以减少开销，所以我们绝对是想允许这么做的，这三行代码就是用于这个目的的。

下一行开始实际工作了：调用SAX解析器的parse()方法。传入待解析的XML，然后解析器就开始工作了。从那时开始，后面你会看到，JSDigester将会被回调来处理各种各样的事件。parse()的最后一行（是JSDigester的parse()，不是SAX解析器的parse()）返回根元素，就是我们在看测试代码时所希望的。

然后是两个简单工具函数：pop()和push()。这是处理把对象推入栈和弹出栈操作的方法。在JavaScript中，每个数组都提供一个push()和pop()方法，所以我们可以把任何数组都当作栈来对待。把这些方法封装在JSDigester自己的方法中，这样我们就可以处理在这些操作上的附加需求了，比如记录日志。除了pop()之外，它还通常把rootObject字段设置为指向最后一个出栈的那个元素，这就是这些代码做的。

在那些函数之后，我们看到startDocument()方法。这是在XML文档开始处理的时候，SAX解析器调用的回调函数。对于这个事件，JSDigester没什么可做，所以就只记录一个日志信息来表示这个事件被触发了。不过，我们必须提供这个方法，因为这样JSDigester才能实现DocumentHandler回调定义的接口。

代码清单7-8中展示的startElement()方法，是一些实际工作开始发生的地方。

代码清单7-8 JSDigester的startElement()方法

```
JSDigester.prototype.startElement = function(inName, inAttributes) {
  if (this.log) {
    this.log.trace("startElement()");
  }
  // If this is not the first element encountered, start by adding a forward
  // slash (the path separator).
  if (this.currentPath != "") {
    this.currentPath += "/";
  }
  if (this.log) {
    this.log.debug("currentPath = " + this.currentPath);
  }
  // Build up the path of the current element.
  this.currentPath += inName;
  if (this.log) {
    this.log.debug("New value for currentPath = " + this.currentPath);
  }
  // Fire all the rules associated with this element.
  this.fireRules(this.EVENT_BEGIN, inName, inAttributes, null);

} // End startElement().
```

首先，在典型的日志信息之后，我们确定这是否是XML文档的第一个元素——根元素。如果它不是，那么我们就需要给当前路径添加一个斜杠，这样就允许我们来创建一个路径。那么，对于我们的示例XML，currentPath当然是从空白开始的。第一个遇到的元素将会是<movies>，所以，currentPath

就要被设置为movies。当遇到第一个<movie>元素的时候，由于调用startElement()函数时，currentPath不再是空白了，所以就给它添加一个斜杠，也就是说，currentPath变成了movies/。然后是添加那个传入给startElement()的元素的名称，所以currentPath这时将会是movies/movie。在每次调用startElement()的时候，都这样继续附加currentPath（后面你会看到，当调用endElement()的时候，可以说，它就是被截断）。

startElement()的最后一行是最重要的。它调用JSDigester的fireRules()方法。把常量EVENT_BEGIN传给fireRules()，当然，这是由于startElement()是在一个元素开始的时候调用的。它还把元素的名字和元素所包含属性的集合作成一个关联数组传给fireRules()。记一会儿这个逻辑，我们要简单地看看fireRules()的细节部分。

与startElement()脉络相同的是下一个方法——characters()。当解析包含在两个标签之间的文字时会调用这个方法。那么，举例来说，<title>Star Wars</title>就会触发一次对characters()的调用，它的inText参数值为"Star Wars"。它也调用了fireRules()，但是这次传的是常量EVENT_BODY，并且只是把文字部分传入给了startElement()。

如前所述，在characters()之后跟着的是endElement()函数。它的任务，首先是调用fireRules()，这次传入了EVENT_END常量，表明现在应该执行那个应用于闭标签的规则。它的下一个任务是从currentPath中移除最后一个元素，对应于结束的那个元素（记得元素构成一个层级关系，那么一个元素的结束总是发生在它的父节点结束之前）。下面是那段从currentPath移除元素的代码：

```
var i = this.currentPath.lastIndexOf("/");
this.currentPath = this.currentPath.substr(0, i);
```

它简单地找到最后一个斜杠字符，通常是当前元素名称的前缀，然后把currentPath设定为不包含那个斜杠的currentPath的部分。轻巧且干净。

然后我们看到endDocument()方法，它就像startDocument()一样，SAX解析器的DocumentHandler接口约定需要的，但在JSDigester里没有任何用处，所以，这样说说就足够了。

3. 运转（真正的大量工作）

现在，我们最后来到了fireRules()函数。但是在讨论之前，我需要指出一些有关规则的问题，虽然我很希望你已经自己了解它们了。

有些规则在一个元素开始的时候起作用，ObjectCreateRule就是一个很好的例子。其他规则在遇到文字的时候起作用，BeanPropertySetter就是其中之一。还有一些规则仅在一个元素结束的时候起作用，比如SetNextRule就是这么工作的。不过，所有的规则在每个元素上都被调用3次：一次是它开始的时候，一次是如果遇到了主体文字的时候（如果有的话），还有一次是它结束的时候。完成了JSDigester的讲解，我们要看看规则，但是作为预览，你要知道每个规则都有3个方法相对应于这些事件：begin()、body()和end()。你还会看到，通常，三个之中的两个都不会做什么（一个规则可以响应超过一个事件是没有什么不可以的，但是现有的规则没有这么做）。这个信息对于理解fireRules()将非常有用。说到这里，我们就可以看看代码清单7-9中的fireRules()了。

代码清单7-9　JSDigester的真正代码：fireRules()方法

```
JSDigester.prototype.fireRules = function(inEvent, inName, inAttributes,
  inText) {
```

```
if (this.log) {
  this.log.trace("fireRules() for currentPath=" + this.currentPath);
}
var ruleIndex = 0;
if (inEvent != this.EVENT_BEGIN) {
  ruleIndex = this.rules.length - 1;
}
if (this.log) {
  this.log.debug("ruleIndex = " + ruleIndex);
}
for (var ruleCounter = 0; ruleCounter < this.rules.length; ruleCounter++) {
  var rule = this.rules[ruleIndex];
  if (this.log) {
    this.log.debug("Checking rule " + rule.getRuleType() +
      " for path " + rule.getPath());
  }
  if (rule.getPath() == this.currentPath) {
    switch (inEvent) {
      case this.EVENT_BEGIN:
        if (this.log) {
          this.log.trace("Calling rule BEGIN");
        }
        rule.begin(inAttributes);
      break;
      case this.EVENT_BODY:
        if (this.log) {
          this.log.trace("Calling rule BODY");
        }
        rule.body(inText);
      break;
      case this.EVENT_END:
        if (this.log) {
          this.log.trace("Calling rule END");
        }
        rule.end(inName);
      break;
    }
  } else {
    if (this.log) {
      this.log.debug("Rule not applicable to this path, so not firing");
    }
  }
  if (inEvent == this.EVENT_BEGIN) {
    ruleIndex++;
  } else {
    ruleIndex--;
  }
 }

} // End fireRules().
```

那么，这里发生了什么呢？好的，首先我们看看这是哪种类型的事件。我们有一个名叫ruleIndex的变量，它是规则的数组的下标。如果事件不是元素的开始，那么我们就需要反向来看规则，以确认它们是按照应有的顺序触发的。所以我们获取指向数组中最后一个规则的ruleIndex。对于一个对应于一个元素开始的事件，我们需要按照正序来运行规则，那么在这时，ruleIndex就是0。这样使用顺序的原因是，规则必须按照它们操作栈里面出现的对象的顺序来触发。所以，执行SetNext规则应该处理栈中的下一个对象，按照正序遍历栈可不是它想要的。不过，对于一个标签的开始，栈需要初始化创建，因此按照正序处理栈就是它想要的。

当我们开始一个循环，一次性遍历规则数组中的每个规则。在遍历时，我们首先取出下一条规则，用ruleIndex来获得合适的下一条规则。然后检查规则的路径是否与正在处理的元素的当前路径相对应。如果对应，就看看我们正处理的事件是哪种类型的，并调用这个规则的适当方法：begin()，把当前元素的属性传给这个函数；body()，把当前元素所包含的文字部分传给这个函数；或者end()，就把元素的名称传给它。

最后，我们根据需要或者增加或者减小ruleIndex的值。当事件是开始事件时，就增加，否则对于其他任何事件都减小。

看起来简单，做起来也非常容易，这是那个真正使JSDigester工作的函数，如前所述。我从没有说过JSDigester在技术上让人印象深刻，但是就像车轮一样，它确实让生活变得更加容易（好吧，这里的“生活”是由开发者来定义的）。

在JSDigester中的最后4个方法都非常简单，所以我会一起介绍它们。addObjectCreate()、addSetProperties()、addBeanPropertySetter()和addSetNext()是JSDigester的用户用于添加处理规则的方法，就如你在代码清单7-6中看到的示例代码一样。每个都含有一些不同的变量集合，这是因为每个规则都请求不同的信息。它们都共享那个共有的路径inPath，这是指向XML中待处理的那个元素的路径。addObjectCreate()规则还需要待实例化的类的名称，它由inClassName变量提供。addSetProperties()方法只需要路径。除了路径之外，调用addBeanPropertySetter()的时候还需要属性的名字作为参数，用来给对象设置属性，而inMethod提供了属性名。最后，addSetNext()除了路径之外，还需要inMethod，那个在栈中下一个对象要调用的方法的名字。

这些方法为适当的规则类（ObjectCreateRule、SetPropertiesRule、BeanPropertySetterRule或者SetNextRule）实例化了一个对象，并且在构造的时候传入了合适的信息。然后它把这个新规则实例压入到为这个JSDigester实例配置的规则数组中。除了一些日志输出之外，上面就是这些方法所做的全部事情了。

同时，我们还看到了JSDigester需要以代码方式提供的所有东西。就还剩下一件事情，那就是看看那4个类。我想你会惊讶，因为它们只涉及了很少的代码，就像JSDigester本身那样。

7.4.4　编写规则类代码

4个已存在的JSDigester规则类都共享同样的基本结构，出于简洁考虑，我在这里就只展示其中一个的完整代码，然后告诉你其他的3个的区别在哪里。我随机挑选了ObjectCreateRule作为例子。代码

如代码清单7-10所示。

代码清单7-10 JSDigester的Object GreateRule类

```
function ObjectCreateRule(inPath, inClassName, inJSDigester) {

  // Set rule type and path to fire for.
  this.ruleType = "ObjectCreateRule";
  this.path = inPath;
  // Set the JavaScript class to instantiate.
  this.className = inClassName;
  // Record the JSDigester instance that the instance of this class belongs to.
  this.jsDigester = inJSDigester;

} // End ObjectCreateRule().


/**
 * Return the ruleType.
 */
ObjectCreateRule.prototype.getRuleType = function() {

  return this.ruleType;

} // End getRuleType().
```

```
/**
 * Return the path.
 */
ObjectCreateRule.prototype.getPath = function() {

  return this.path;

} // End getPath().
/**
 * Begin an element.
 */
ObjectCreateRule.prototype.begin = function(inAttributes) {

  if (this.jsDigester.log) {
    this.jsDigester.log.debug("ObjectCreateRule.begin(): " + inAttributes);
  }
  var protoObj = eval(this.className);
  this.jsDigester.push(new protoObj());

} // End begin().


/**
```

```
 * Process body text.
 */
ObjectCreateRule.prototype.body = function(inText) {

  if (this.jsDigester.log) {
    this.jsDigester.log.debug("ObjectCreateRule.body(): " + inText);
  }

}


/**
 * Process closing tag.
 */
ObjectCreateRule.prototype.end = function(inName) {

  if (this.jsDigester.log) {
    this.jsDigester.log.debug("ObjectCreateRule.end(): " + inName);
  }
  this.jsDigester.pop();

} // End end().
```

一个JSDigester规则通常包含如下3个字段。

- ruleType：规则的名称。只有在记录日志的时候它才真正派上用场。
- path：应该触发这个规则的XML结构中的路径。
- jsDigester：当规则实例化时，传入到构造函数的那个JSDigester实例。

所有的规则类通常都提供getRuleType()方法，它只返回ruleType值和getPath()方法，getPath()方法返回路径的值。

在那些字段之后的是3个方法：begin()、body()和end()。就像我们在看JSDigester的fireRules()方法时所描述的那样，begin()是在第一次遇到某个元素的时候调用的，也就是一个开始标签。处理两个标签之间的文本内容时，调用body()。end()是在遇到一个结束标签时调用的。

记住，通常来说（特别是对那4个已存在的规则来说），一个规则仅处理这3个事件的其中之一。ObjectCreateRule只负责响应开始事件。好吧，我想在技术上这么说并不准确。在遇到结束标签的时候，它还负责把创建的对象弹出栈，但是那是一件相对来说很小的事情。不过，我想为了准确起见，还是应该说ObjectCreateRule实际上处理了两件事，虽然有一个事件实际上微不足道。

如果看看其他3个规则（除了ObjectCreateRule之外），你就会发现，其中两个方法只有日志语句。剩下的那个做了些实际工作。在ObjectCreateRule中，它非常简单：初始化一个指定名字的新实例，并把它推入栈。实例化过程是通过简单的对className的值做eval()来完成的，这样就给那个类创建了一个新的实例了。

但是现在，它是怎么知道要创建哪个类的实例的？很显然，是通过它的className。不过那是从哪来的？很好，记得所有的规则都共享了3个字段吧：ruleType、path以及jsDigester。虽然那些字段满足某些规则，比如SetPropertiesRule，但是其他的规则，包括我们的例子ObjectCreateRule，还是需要更多的字段。这并不是问题。我们简单地添加字段，并把它加到构造函数的参数列表中，问题就解

决了。在ObjectCreateRule的例子中，我们找到一个className字段，并且看到传给构造函数的第二个参数是inClassName。

对于SetPropertiesRule类，工作是在begin()方法中完成的，它比ObjectCreateRule做得更多些：

```
SetPropertiesRule.prototype.begin = function(inAttributes) {

  if (this.jsDigester.log) {
    this.jsDigester.log.debug("SetPropertiesRule.begin(): " + inAttributes);
  }
  var obj = this.jsDigester.pop();
  for (var i = 0; i < inAttributes.length; i++) {
    var nextAttribute = inAttributes[i];
    var keyVal = nextAttribute.split("=");
    var key = keyVal[0];
    var val = keyVal[1];
    key = "set" + key.substring(0, 1).toUpperCase() + key.substring(1);
    if (this.jsDigester.log) {
      this.jsDigester.log.debug("SetPropertiesRule.begin() - key=" + key +
        "val=" + val);
    }
    obj[key](val);
  }
  this.jsDigester.push(obj);

} // End begin().
```

那么，这段代码完成的事情是：接受一个已经分析成name=value对的描述元素的属性的数组，然后设置栈顶部对象的所有属性。实现方法是先弹出栈顶部的元素，然后遍历传入的属性数组，把每个数组元素都用等号分开。调用nextAttribute.split()生成的数组的第一个元素是等号的左边的，也就是属性的名称；第二个元素是在等号右边的，也就是属性的值。下面我们来看看这几行代码：

```
key = "set" + key.substring(0, 1).toUpperCase() + key.substring(1);
```

它的任务是使用标准JavaBean格式构造要调用的方法的名称，也就是getXXXX，这里XXXX是要设定的属性名称（第一个字母大写，剩下的字母和属性名称中的大小写一样）。所以，对于元素<actot>来说，它含有一个性别（gender）属性，那么组成的方法的名字就是setGender()。一旦构造好了方法名称，就执行下面这行代码：

```
obj[key](val);
```

这其实是非常炫的JavaScript特性。你看，你可以把任何JavaScript对象像关联数组那样处理，数组中元素的名称就是对象的成员。所以，让我们假设obj是指向一个Actor对象的引用。然后如果我们使用obj["setGender"]，这样就给我们获得了一个指向Actor对象成员函数setGender()的引用。同样，把Actor对象也当作一个关联数组来对待，setGender是我们想要引用的那个元素的名字。如果给它添加(val)，其实就是在动态创建一个函数调用。我们可以用一个变量名，而不用写死的setGenter值，而规则类就是使用了变量名。想象一下，我们在Java中需要多么地绞尽脑汁进行思考和反应才能实现这

小小一行代码的功能，然后还说JavaScript不够酷。

在设置完所有属性之后，把对象推回栈顶部，返回给它在规则被触发之前的状态，此时，规则就已经完成了它的任务了。

沿着BeanPropertySetter规则继续往下看，是另外一个附加字段：setMethod字段，它指明了用来给对象赋值的方法。在这个规则中，行为是在body()方法中发生的，而且我们又看到了与SetPropertiesRule类中动态方法调用技术非常类似的东西。栈顶部的对象又被弹出来，并用几乎相同的一行代码来设置其属性：

```
obj[this.setMethod](inText);
```

在这个例子中，只设置了一个属性，它的名字存储在setMethod字段中。所以，这里并没有像BeanPropertySetter中那样的循环，而且也没有需要分解的关联数组。就只有一个简单的调用，并把传给body()方法的文字传给它，我们就完成了，就这么简单。

要看的最后一个规则是SetNextRule。如同BeanPropertySetter一样，它需要那个调用的方法的名称，所以我们在这里也看到了一个setMethod字段，以及相应的构造函数参数inMethod。在这里，工作是在end()方法中完成的，而且它其实有点小技巧（当然，可能用“有趣”来描述它更合适）。

```
SetNextRule.prototype.end = function(inName) {

  if (this.jsDigester.log) {
    this.jsDigester.log.debug("SetNextRule.end(): " + inName);
  }
  var childObj  = this.jsDigester.pop();
  var parentObj = this.jsDigester.pop();
  if (this.jsDigester.log) {
    this.jsDigester.log.debug("SetNextRule.end() - childObj=" + childObj +
      "parentObj=" + parentObj);
  }
  parentObj[this.setMethod](childObj);
  this.jsDigester.push(parentObj);
  this.jsDigester.push(childObj);

} // End end().
```

那么，在一些日志之后，我们从栈中弹出两个对象：第一个是子对象，第二个是父对象。思考一分钟：SetNextRule是用来设置栈中下一个对象的属性的，并给它传递栈顶部的对象。这是用来创建对象的层级关系的，也就是一个对象是另一个的父对象。由于XML的结构，栈顶部的对象通常是子对象。

```
<movies>
  <movie>
    <actor></actor>
  </movie>
</movies>
```

让我们假设上面这些就是待解析的XML，而且我们已经为每个都仅配置了一个ObjectCreateRule规则。在第一次遇到<actor>元素（但并没有结束）时，JSDigester中的对象栈是什么样子的呢？人们说图表胜千言，所以我们就看一看图7-5吧。

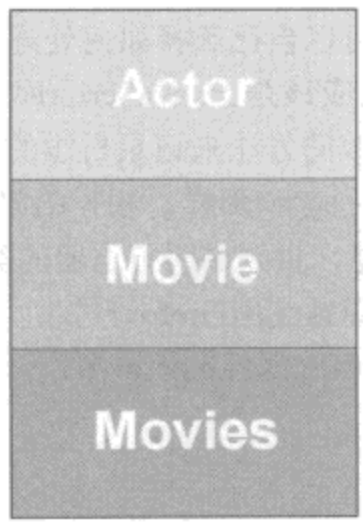


图7-5 在开始一个<actor>元素的时候，JSDigester栈的状态

现在，如果我们有一个使用方法setActor()给路径movies/movie/actor配置的SetNextRule的规则，会怎么样呢？好的，第一个从栈中弹出的对象是顶部对象，Actor对象。它是子对象。然后弹出的还是这个时候的顶部对象，Movie对象，它是父对象。因为它是父对象，所以就调用Movie对象的setActor()方法，用下面这行：

```
parentObj[this.setMethod](childObj);
```

乌啦～这样我们已经创建了父-子层级关系了。现在Movie对象包含Actor对象，就如我们所希望的那样。那么还剩下一个要执行的步骤，就是把这两个对象都推回到栈中，因为可能还有其他规则要在它们身上触发。但是请记住，要保证栈里的顺序没有改变，我们需要用与出栈相反的顺序来把它们压回栈，所以那个Actor对象就会在Movie对象的上面了，你可以在end()方法的最后两行中看到这些。

看到了吧，这些不是很有意思的代码吗？

好的，这很有意思，也没花太长时间。如我说过好几次那样，JSDigester和它的那些关联规则类，在代码长度上并没有什么惊人的。实际上，你甚至可以把它们称作琐碎的。但是，简单且功能强大的东西是好东西，并且这也向Commons Digester的原作者证明，这个东西的确把XML变得更有趣。我希望像我一样，你也可以在日常工作中，发现JSDigester更多的有用之处。

7.5 练习

这里有一些练习，你可以继续把JSDigester做得更有用。

*实现更多规则。*Digester有满满一船可以在JSDigester中实现的规则。有些也许不太可能，但是其他大部分都是可以的。因为JSDigester是可扩展的，所以这将会是熟悉它的非常好的练习。

*添加验证功能。*Digester可以根据DTD来验证XML，这也是可以添加给JSDigester的非常好的附加功能。

*允许可替换的SAX解析器。*提供一个setSAXParser()函数来替换JSDigester可以使用的SAX解析器。作为一个附加的红包，你自己来写写这个代码吧。

*优化代码。*我故意在代码中留下了一个不好的设计，这样我就可以提出这个建议了。你注意到每一个规则类都有一个不同的构造函数吗？当然，这可能不是最佳的，实际上，真的不是。因此我建议你修正它。你可以有不止一种的改进方法，不过我希望是传入一个单独的关联数组作为参数，并且使

用它包含的数值来设置字段。不过，你可以自己选择如何处理，目标就是将规则类的方法一般化。

提高日志记录的性能。这个实际上应该是给JavaScript库（在第3章介绍）提的建议，而不是给JSDigester的。如果你有一个可以用更高效的方法来记录日志的日志记录程序，可能会更好，那样的话，记录日志就不会是性能杀手了，就像DivLogger那样。明显的问题是如何在打开日志的时候仍然能保持较好的性能（比如，你可以缓冲所有消息，直到调用日志器的类似flush()函数，但这样就有在发生错误的时候丢失所有消息的风险）。如果你经常使用Firefox开发，一个建议是使用Firebug扩展来记录日志，我觉得它会比不停更新一个 的innerHTML属性高效得多。

7.6　小结

在本章中，我们非常简要地看了看在JavaScript中如何在不需要其他类或工具包的情况下解析XML。然后效仿Jakarta Commons Digester项目，创建了自己一些代码，它们应该可以在我们需要在浏览器中处理XML的时候更加方便。在这个过程中，你开始了解了Mozilla项目创建的SAX解析器，最后它是JSDigester的基础。

第8章

做正确：JavaScript验证框架

当你在Web应用程序的环境中听到验证（validation）这个词的时候，大概会想到验证表单上的用户输入。这通常让人们回忆起编写事件处理函数，在提交表单之前对表单输入的各种各样检查。这些东西会迅速变得很杂乱。不论你是如何外部化脚本、如何设置基本的事件处理函数来调用函数，结果都会是这些验证散布于代码中——区别只是散布的程度而已。

如果可以真的把那些验证外部化，让我们完全不需要再写任何类似的代码，不是很好吗？如果可以写一个简单的配置文件，假设是XML的，来定义我们想要在什么时候每个字段执行什么样的验证，不是很好吗？这正好是本章的项目要达到的目标！

8.1 JSValidator需求和目标

建立一个JavaScript表单验证框架并不是一个很复杂的事情，但是要想让它真的有用则确实需要一些预先的思考。说到这里，让我们列一些希望在这个项目中达到的目标，我们称这个项目为JSValidator。

- 框架应该用一个XML写的外部配置文件驱动。它将定义对于某个字段应该使用什么验证，可能需要一个给定的验证参数，以及如果验证失败了应该做什么。
- 框架应该是可扩展的。我们应该可以在任何时候添加验证器——就是那些执行一个特定验证功能的类。而且应该不用做任何代码修改，只是配置文件即可。
- 我们应该创建几个常用的验证器类型，并且把它们放到框架中，这样开发人员就知道他们其实可以不用自己开发而直接用。
- 这个框架应该是不唐突的，意思是我们不需要到处放置代码。而且，如果关闭了JavaScript，那个定义了验证功能的表单应该仍然可以提交。
- 使用框架对于开发者来说应该尽可能的简单，只需要一个JavaScript导入，一些最小限度的配置参数，以及一个外部配置文件即可。
- 框架应该以3种方式提供错误报告机制：alert()信息、插入到指定 的信息和字段的突出显示（使用开发者定义的样式）。而且，消息应该可以通过使用一个记号系统来复用，而在发生验证错误的时候，这里的记号可以用配置文件里定义的记号代替。

如果这些听起来需要很多工作，那么我想你会非常惊讶地看到它其实并不像你想的那样。你还会

发现，最终的结果是一些有用的、可扩展的东西，而且马上就可以在项目中使用。不过我还留了几个可以改进的地方，这些就是留给读者的作业了。如果读者能自己完成这些作业，那将是非常宝贵的练习，可以有效地提高你的JavaScript能力。

带着这些目标和需求，让我们一起来看看是如何搞定它们的！

8.2　怎么把它拔下来

在需求中已经提到，我们希望JSValidator使用一个外部的配置文件来定义所有需要在给定HTML表单上执行的验证。更加明确的是，我们希望这个配置文件是XML格式的。

在决定一个配置文件的格式时总是有很多种选择。不过，虽然我觉得XML并不是对于任何事情都非常完美，但对于配置文件，它应该是很合适的。这是因为，它是自描述性的（除非开发者在展现时做得很糟糕）。而且，大小和解析时间通常并不重要，因为通常你都会在开始的时候解析它们，之后就不会频繁的解析了。通常在开始的时候我们都能忍受一点轻微的性能损失，也就是说你并不太关心那个文件有多大。实际上，大多数时候是越冗长越好，只要这样的冗长增加了配置的清晰度。

那么，决定了使用XML来为JSValidator服务之后，下一个问题就是：我们如何去处理它呢？如果你曾经在JavaScript中做过XML解析，你就会知道其实那并不是一个让人十分舒服的经历。我在第7章中说过一点，那么这里就没必要再重温痛处了。在第7章中，我们还找到了让它可以容忍的方法：JSDigester。在JSValidator里你将会看到JSDigester的实际使用。你还会了解到如何使用一些简单的规则来描述一个复杂的XML文档，然后把它快速而简单地解析到JavaScript对象中的。（如果你跳过了第7章里JSDigester的细节的话，我还是建议你现在就回过头去读一下那章的内容。）

既然你已经知道了我们要怎样解析那个配置文件了，那么来回答另外一个重要的问题吧：我们如何载入它？记住JSValidator的另外一个目标，就是使它对一个开发者来说，尽可能地简单且不唐突。举例说，下面就是要在任何给定的页面上使用验证框架时，开发者需要做的全部工作：

```
<script>
  var JSVConfig = {
    pathPrefix : "jsvalidator/",
    configFile : "jsv_config.xml",
    manualInit : false
  };
</script>
<script src="jsvalidator/JSValidator.js"></script>
```

这里需要做的就是导入一个单独的JavaScript文件，以及在此之前定义的一些参数配置。除了这些，开发者就只要写那个配置文件就可以了。在表单字段周围，没有任何垃圾，没有挂在什么东西上的特定属性，也不需要再写什么代码。更好的是，如果禁用了JavaScript，这个页面仍然不会瘫痪——表单依旧可以工作（当然，假设它不会因为其他的原因而崩溃），用户看不出有什么区别。当然，那就没有

进行验证了，不过这也是正常的。[①]

说说那个配置文件吧，它究竟是如何载入的？我们在前面的例子中可以看到，在参数中给出了文件的名字，但是并没有解释如何装载文件。答案就在一个JavaScript库中，它是作为其他库的一个基础，包括Rico（我们在第4章中使用过的）和script.aculo.us（在第5章中使用过的）。在那些章节中，我并没有描述Prototype的细节，但是现在，应该来看看了。

8.3 Prototype库

Prototype现在已经使用非常广泛了，这是因为它简单、轻量级、干净，并且通常非常有用。它基本上是提供JavaScript扩展的，而你用过这些扩展之后，会觉得JavaScript从一开始就包含它们一样。Prototype给基础的JavaScript对象添加了新的方法，同时也为那些像Ajax、DOM操作和循环结构这样的东西提供了新的函数。

Prototype提供的最简单也是最有用的一个功能，就是$()函数。就像你在前面几章中看到的，这其实是document.getElementById()的一个简写方法。$()函数允许引用一个单独的对象，或者同时引用一组对象，就像这样：

```
$("id1", "id2", "id3");
```

这个例子代码会返回指向那3个元素的引用的数组。它比直接使用3次document.getElement- ById()调用要好多了。

Prototype还提供了一些简单的Ajax支持。我这里说的“简单”，并非意味着它是不好的东西。正相反，它非常容易理解和使用，这对我来说是件好事情。在Ajax的领域中，Prototype提供的第一个项是：

```
Ajax.Request(url, { method: 'get', parameters: pars, onComplete: showResponse });
```

是的，这就是使用Prototype来发送一个Ajax请求所需要做的全部工作。只要提供一个URL，设置要使用的方法，传入任何你想要发送请求的参数，并告诉Prototype，当返回结果时应该调用那个函数就可以了。

不过，请等一下，还有更多！

可能最常用的Ajax函数，是把一些从服务器端返回的HTML标签，插入到一个 或其他什么元素中。Prototype也把这个做得非常简单：

```
new Ajax.Updater( "targetID", url, { method: 'get', parameters: pars });
```

这看起来和前面那行代码很相似，除了我们现在传入的是要更新的那个元素的ID，不再传那个回调函数，因为Protorype为我们处理了回调功能。

就像我前面提到过的，Prototype还扩展了一些基础的JavaScript对象。例如，Prototype给在表8-1中的那些对象添加了一些方法。

① 因为禁用了JavaScript。

表8-1　Prototype给内置的JavaScript对象添加的一些方法

方　法	对　象	描　述
extend	Object	提供了一个方法，通过把源对象所有的属性和方法都复制给目标对象来实现继承机制
bind	Function	返回之前绑定在函数（=方法）所有者对象上的那个函数的一个实例。返回的函数将与源函数拥有一样的参数
stripTags	String	返回移除了所有HTML和XML字符的字符串
escapeHTML	String	返回适当转义了HTML代码字符的字符串
toArray	String	将字符串切分成它的字符组成的数组
clear	Array	清空数组并返回它本身
flatten	Array	返回一个一维版本的平面数组。这个平面化（降维）的处理是通过递归地在每个数组的元素中寻找本身也是数组的元素，然后把这些数组的元素也包含在返回的数组中
getElementsByClassName	document	返回与给定CSS类名匹配的所有元素。如果没有给出父节点ID，就会查找整个文档的主体部分
element	Event	返回触发那个事件的元素
pointerX/pointerY	Event	返回页面上鼠标的*x*/*y*坐标

说明　就像我在第2章中曾经说过的，一些人认为，Prototype对JavaScript内置对象的扩展是件坏事。有些实例可以说明这样的扩展并没有和其他的JavaScript代码一起“和谐工作”。我个人从没有被这样的问题困扰过，但是有些人曾经遇到过，所以应该在使用Prototype（以及任何构建在Prototype基础之上的js库）的时候记得这些。但是我觉得，这些不应该是阻止你使用Prototype或者其他一些在它基础上的库的理由，你只要知道这个问题就行了。

正如你看到的，Prototype让生活变得更加方便，这里我只是随便划了几笔而已！我建议你好好看看Prototype。另外，在文档方面，可以看看Prototype的那个“非官方的”指南：http://www.sergio-pereira.com/articles/prototype.js.html。

为什么我总觉得我忘了什么事情呢？哦，对了，让我们去看看JSValidator，怎么样？

8.4　JSValidator的预览

由于JSValidator真的不是一个视觉工具，预览也看不了太多东西，但那并不能阻止我添加一些截

屏。你正看到的是一个使用了JSValidator的示例应用程序。

图8-1展示了当你第一次使用这个演示程序时的界面。它描绘了在5个表单字段中添加了什么样的验证。

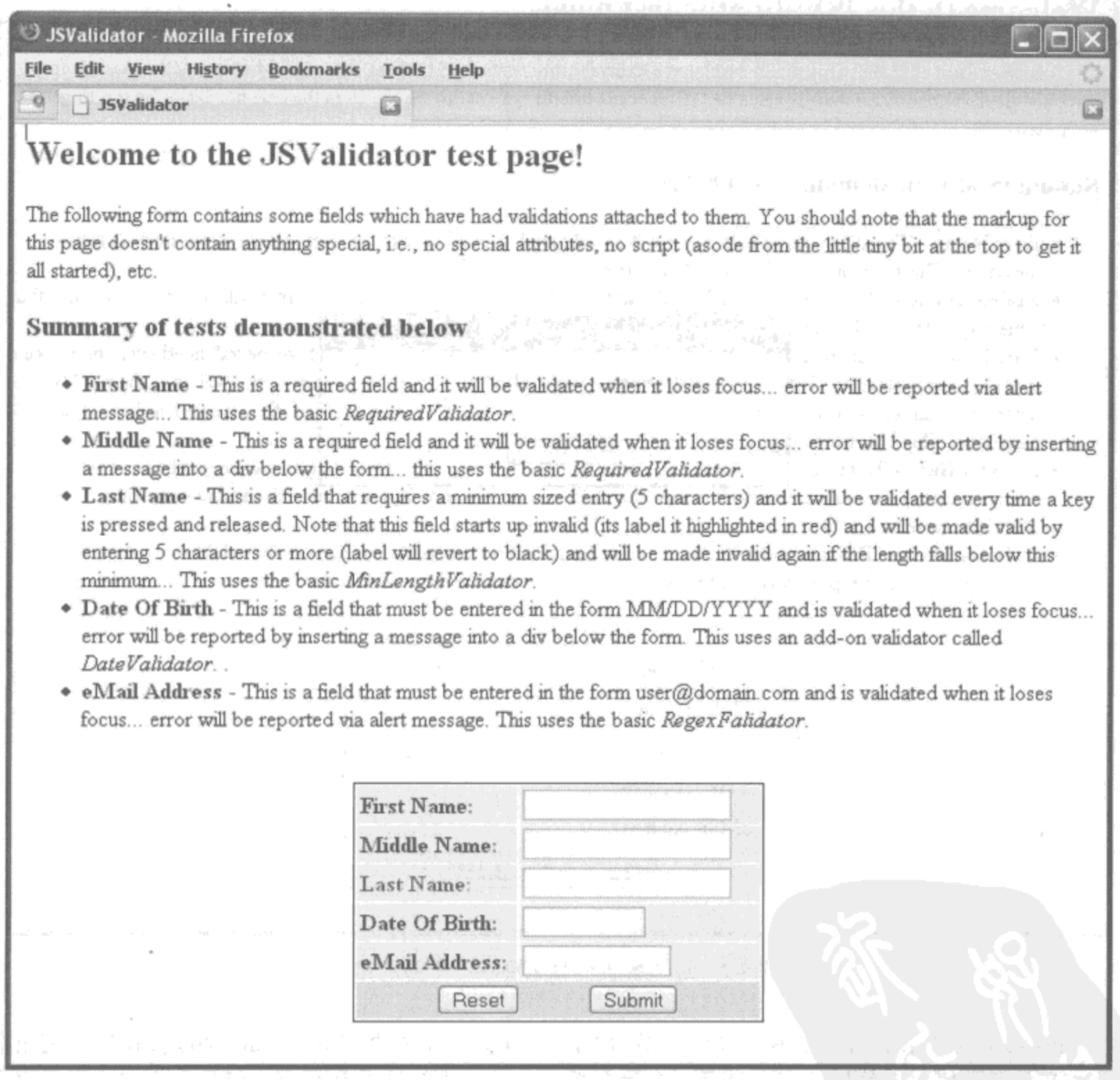


图8-1 JSValidato示例程序

图8-2展示了验证失败时给用户显示信息的一个方法：一个警告信息。这里，当用户使用tab键移出或点击离开那个First Name字段，而并没有在其中输入任何东西的时候（这个字段是必填项），就会弹出一个alert()信息。注意那个信息是可配置、可复用的，并且允许替换其中的记号。

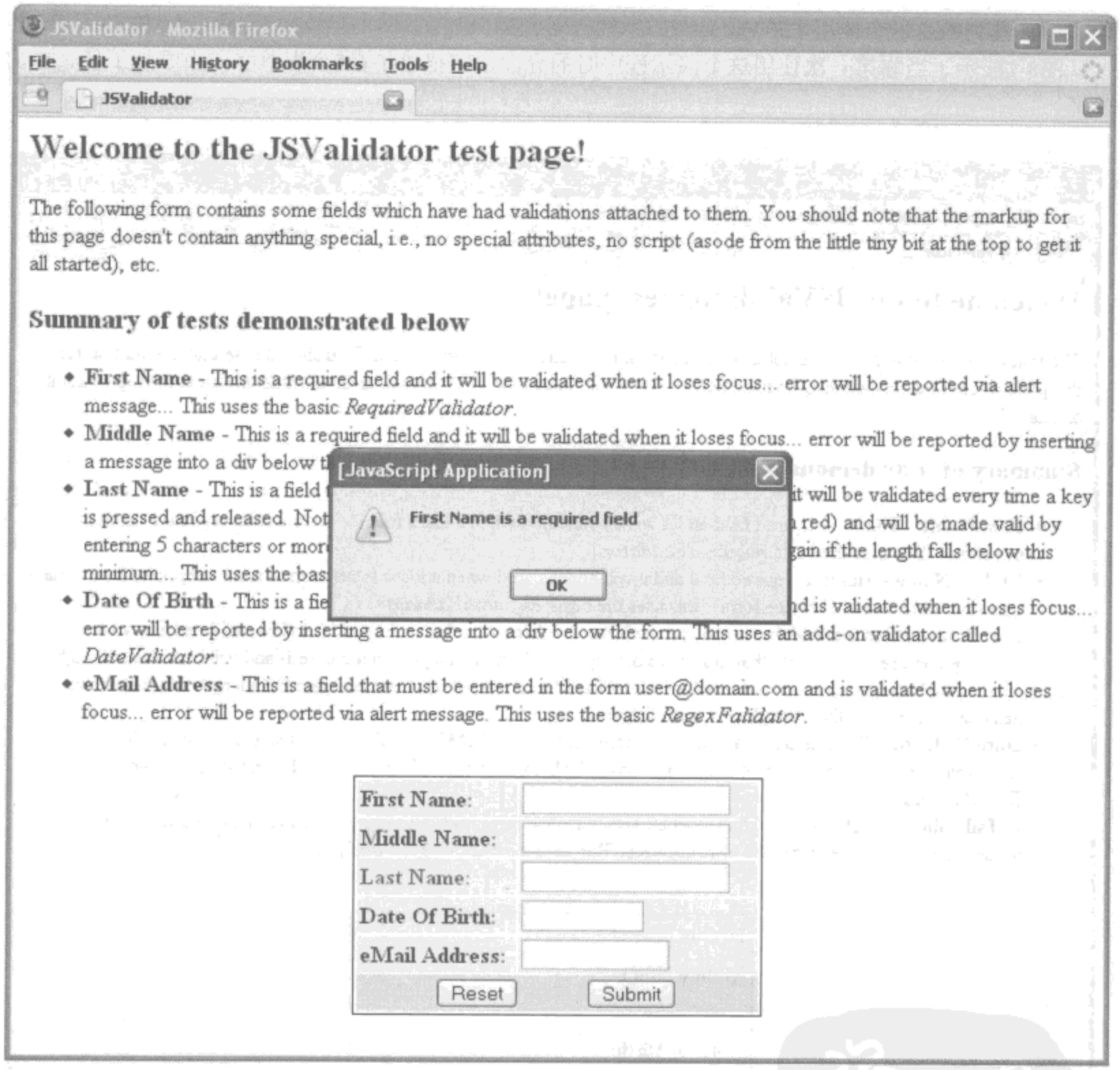


图8-2　通过警告信息来展示一个验证错误

图8-3举例介绍了另一个展示验证错误的方法。当发生错误时，Last Name字段被设置为突出显示。这个字段使用红色标记突出显示（在黑白版本中是几乎无法看到红色，但是当你在应用程序中尝试时，你会发现它非常明显），指出自己被标识为非法输入。实际上，你可以配置页面上的不同元素使用不同的样式。你可以把文本框的背景色变为红色。你也可以在页面上放一个巨大的“ERROR”!

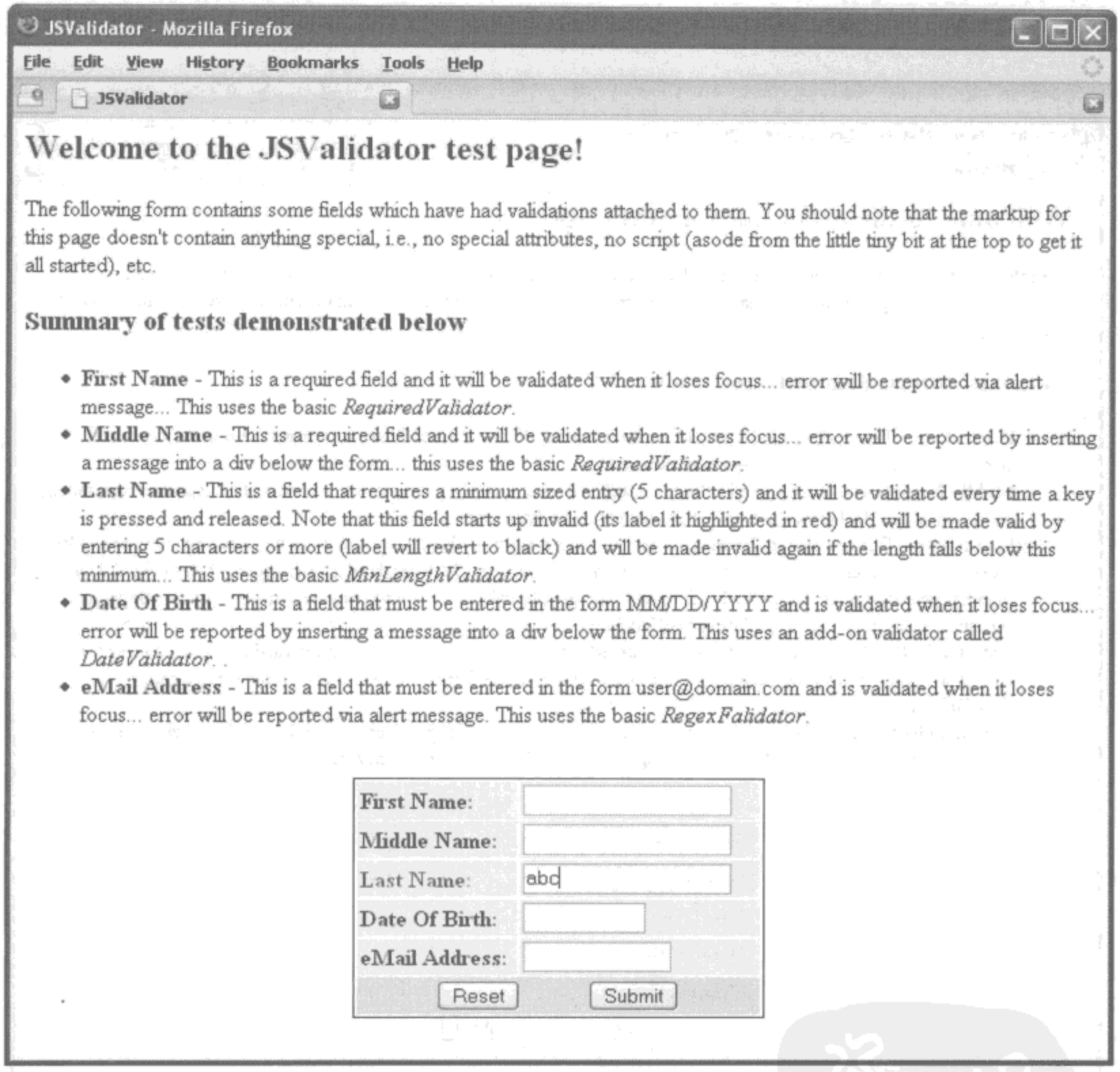


图8-3 通过文本框突出显示来展示一个验证错误

图8-4介绍了第3种可以报告验证错误的方法：通过给页面中的一个元素（通常是一个 ）插入一个错误信息来报错。信息的构造方式与alert()展示相同，但是在这里是某些页面元素的innerHTML属性更新为错误信息。

点击Submit（提交）按钮，就会打开一个新页面，那里就只是一条告诉你提交成功的信息。很显然，我们实际上并没有真的向服务器提交信息，所以这个页面是一些类似于伪服务器的东西。我觉得没有必要在这里展示这个页面了，不过你亲自尝试那个应用程序的时候是会看到它的，所以我要在这里事先说一下。

那么，现在一些预先的介绍已经说完了，我们可以去研究那个解决方法了！

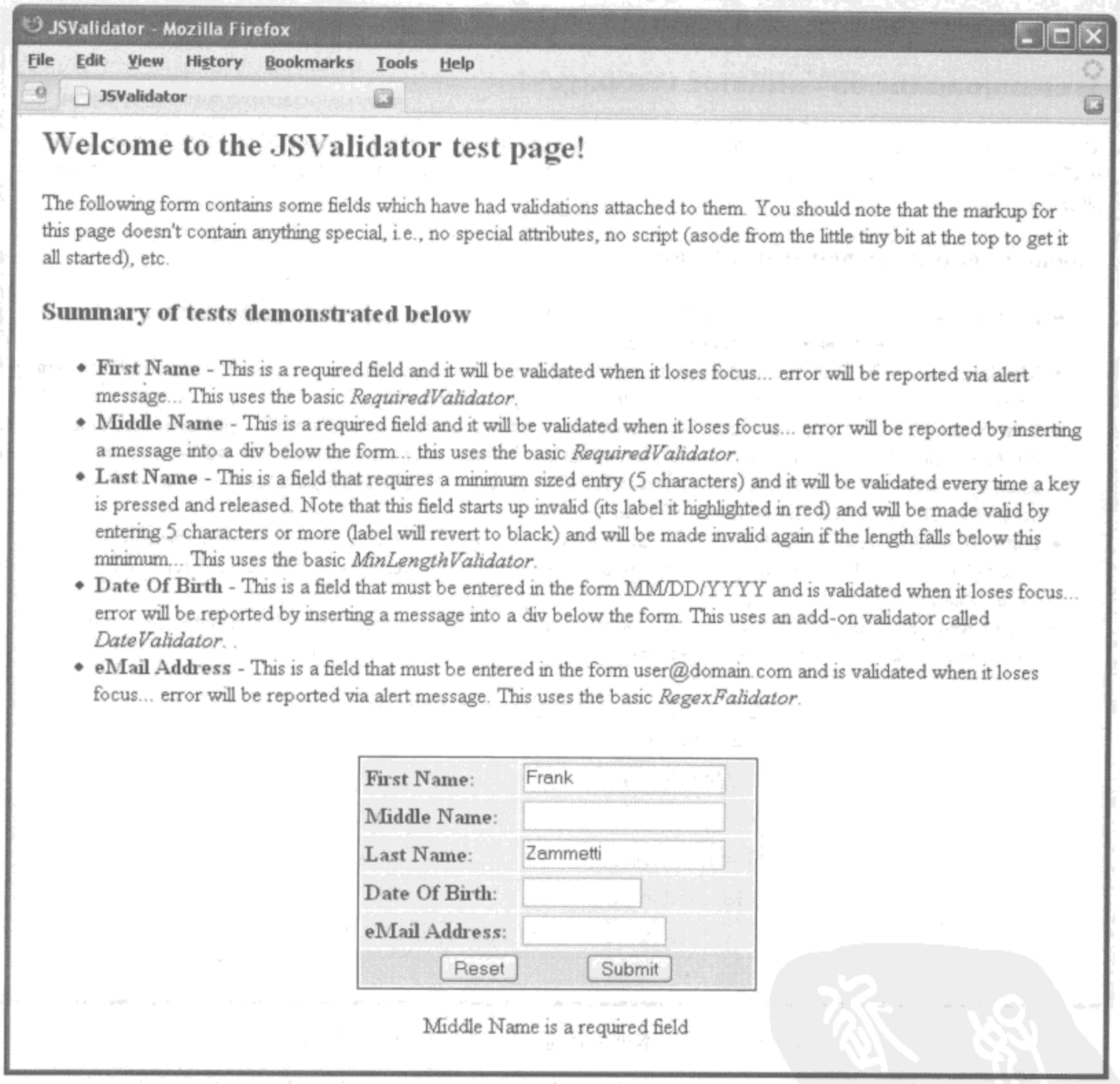


图8-4　通过向一个 元素中添加信息来展示验证错误

8.5　剖析JSValidator的解决方案

我准备将JSValidator框架和它的测试程序一起来解释。首先，让我们看看这些在咱们的硬盘上是如何分布的，就如图8-5所示。

图8-5 JSValidator测试程序的目录结构

8.5.1 编写index.htm

index.htm是这个项目中最没劲的文件，因为它只是非常普通的HTML代码而已。它就只是一些文本，一个非常基本的HTML表单，一个样式表导入语句，以及一些JavaScript。因此，我们就不在这里罗列它的代码了，也不打算看它的任何细节了。不过，你可能应该花5秒钟的时间来看一遍，好对它有个整体的概念。

JavaScript是这里唯一值得一说的，实际上你应该已经看过它了。不过还是让我们再看一次吧，而且这次多说一点细节的东西。

```
<script>
  var JSVConfig = {
    pathPrefix : "jsvalidator/",
    configFile : "jsv_config.xml",
    manualInit : false
  };
</script>
<script src="jsvalidator/JSValidator.js"></script>
```

首先，你看那个JSVConfig结构的定义。这就只是一个关联数组，它有3个元素，正好是目前JSValidator可以理解的几个。

pathPrefix元素指定JSValidator需要用来构造那些载入的验证器和规则文件的所有路径的前缀。这里，我们假设的是所有那些动态创建的路径都以jsvalidator/开始，如果你回过头去看看那个目录结构（图8-5）就明白了。任何指向JSValidator资源的引用都是Web应用程序根目录下的相对路径，因此就有了这个值。

第二个元素，configFile，应该很明显了。它是JSValidator 将要使用的XML配置文件的名字。

最后，manualInit告诉JSValidator，它是应该自动初始化呢，还是由开发者来负责。你已经看到，初始化JSValidator的工作总共也只是一句对jsValidator.init()的调用而已。通常，这是在载入JSValidator的时候自动完成的。不过为了达到这个目的，JSValidator需要覆盖页面的onLoad事件处理函数。由于开发者通常为了自己的目的来利用这种方法，所以简单地覆盖已存在的处理函数其实并不是个好主意。把manualInit设置为true，就意味着开发者将要自己调用init()，并且不会覆盖onLoad处理函数。

8.5.2　编写styles.css

styles.css就像index.htm一样，是相当平凡的。不过，让我们看一下它的全部吧，如代码清单8-1所示。

代码清单8-1　styles.css文件（眨眼就忘！）

```
/* Style for labels on fields that have a validation error. */
.cssErrorField {
  color            : #ff0000;
  background-color : #d0ffd0;
  font-weight      : bold;
}

/* Style for labels on fields that pass validations. */
.cssOKField {
  color            : #000000;
  background-color : #d0ffd0;
  font-weight      : bold;
}

/* Background for the table cells with the entry fields. */
.cssEntryCell {
  background-color : #f0f0f0;
}

/* Style for the table row with the reset and submit buttons. */
.cssButtons {
  background-color : #ffd0d0;
}
```

就像你在图8-3中看到的，JSValidator报告验证错误的一种方式是突出显示一个给定的字段。它可以突出显示页面中的任何一个元素。实际上，它不必“突出显示”任何东西！

真正发生的事情是，当发生一个验证错误的时候，带有指定ID的那个元素的样式类属性就会改变为配置的那个样式类。所以，你真的可以做你想做的任何古怪的事情。想在一个隐藏的 中展示一个图片吗？没问题。想为那些使用老式Netscape浏览器的人把文字变为闪烁的吗？绝对没问题。只要是通过改变样式类就可以实现的效果，你就可以用这个“突出显示”功能实现。那些事只是解释这里的突出显示方法的一个不同路径而已，在这里看到的cssErrorField类是将会用于突出显示样例应用中的字段的标签的类而已。

与cssErrorField一起出现的是cssOKField类，它将会用在通过验证的输入框的标签上。这就是说，JSValidator会用不同的样式来表示是否通过了验证。例如，当验证Last Name框时，在每次敲击和放开一个按键的时候都检查最小长度，你在打字的同时，就得到了动态的合法/非法的突出显示效果。用不着你做额外的工作——就只是把框架指向那两个样式类就可以了，它会自动为你处理。

cssEntryCell样式类就是把输入框的背景变成浅灰色。最后，cssButton是应用于Reset和Submit按钮放置的那一行的。

8.5.3 编写jsv_config.xml

jsv_config.xml是驱动整件事情的部分，所以我们绝对要理解它。图8-6展示了它的图形表示，它可以帮助我们在脑海中把所有关系都弄清楚。

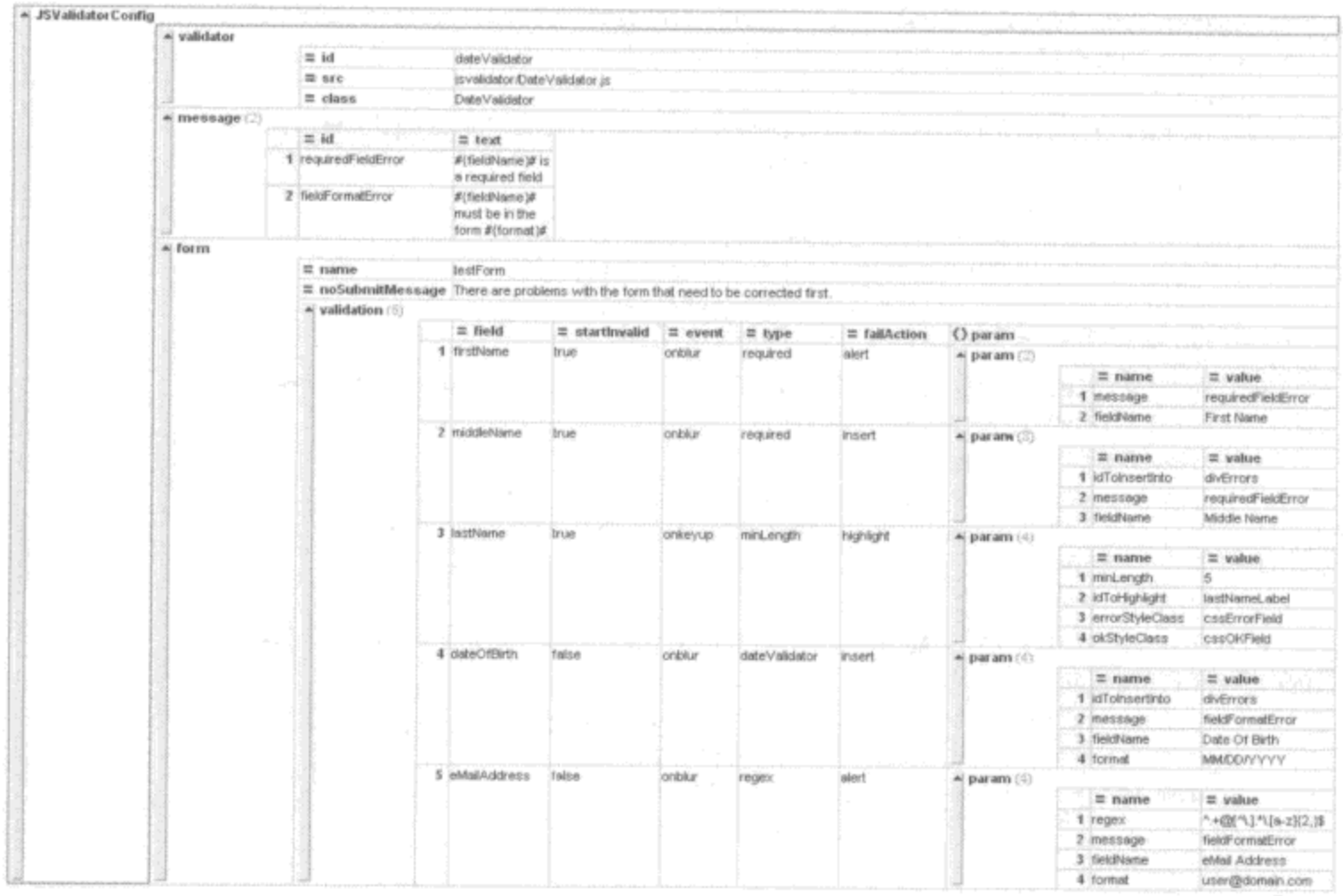


图8-6 jsv_config.xml文件的可视表格展现法

如果图8-6并不特别适合你，代码清单8-2中以原文形式展示了那个配置文件。

代码清单8-2 纯文本格式的JSValidator配置文件

```
<JSValidatorConfig>

  <validator id="dateValidator" src="jsvalidator/DateValidator.js" ➥
class="DateValidator"/>
  <message id="requiredFieldError" text="#{fieldName}# is a required field"/>
  <message id="fieldFormatError" ➥
text="#{fieldName}# must be in the form #{format}#"/>

  <form name="testForm" ➥
noSubmitMessage="There are problems with the form that need to be corrected first.">
    <validation field="firstName" startInvalid="true" event="onblur" ➥
type="required" failAction="alert">
      <param name="message" value="requiredFieldError"/>
```

```
        <param name="fieldName" value="First Name"/>
      </validation>
      <validation field="middleName" startInvalid="true" event="onblur" ➥
type="required" failAction="insert">
        <param name="idToInsertInto" value="divErrors"/>
        <param name="message" value="requiredFieldError"/>
        <param name="fieldName" value="Middle Name"/>
      </validation>
      <validation field="lastName" startInvalid="true" event="onkeyup" ➥
type="minLength" failAction="highlight">
        <param name="minLength" value="5"/>
        <param name="idToHighlight" value="lastNameLabel"/>
        <param name="errorStyleClass" value="cssErrorField"/>
        <param name="okStyleClass" value="cssOKField"/>
      </validation>
      <validation field="dateOfBirth" startInvalid="false" event="onblur" ➥
type="dateValidator" failAction="insert">
        <param name="idToInsertInto" value="divErrors"/>
        <param name="message" value="fieldFormatError"/>
        <param name="fieldName" value="Date Of Birth"/>
        <param name="format" value="MM/DD/YYYY"/>
      </validation>
      <validation field="eMailAddress" startInvalid="false" event="onblur" ➥
type="regex" failAction="alert">
        <param name="regex" value="^.+@[^\.].*\.[a-z]{2,}$"/>
        <param name="message" value="fieldFormatError"/>
        <param name="fieldName" value="eMail Address"/>
        <param name="format" value="user@domain.com"/>
      </validation>
    </form>

  </JSValidatorConfig>
```

让我们仔细剖析一下这个文件，并且看看每个元素和属性吧。首先，为了让它是一个合法的XML，所有东西都是内嵌于根元素JSValidatorConfig之下的。

1. 定义验证器

在根元素之后的，是任何数量的validator元素（也可以是没有）。这些定义了你添加的验证器（那些执行验证逻辑的类）。有一些内置的验证器（validator），你将会简短地看一下，不过任何你自己添加的其他验证器，都需要使用validator元素来定义。validator元素包括下面3个必须属性。

- id：你将通过id来为一个给定的字段验证引用这个验证器。
- src：可以找到那个实现验证器的类的JavaScript源文件。可以是绝对路径或者相对路径——基本上是所有对于<script>标签的src属性合法的任何值都可以。
- class：实现验证器逻辑的JavaScript类的名字。

2. 定义消息

JSValidatorConfig的直接子节点是一个message元素。和validator元素一样，这些是完全可选的，你想要多少就可以定义多少。虽然技术上它们是可选的，不过通常应该至少有一个。因为任何在验证

错误发生时展示信息的验证器，都会引用这些元素的其中之一。

message元素有两个必须属性：id，它与在validator元素上的那个作用相同；text，是信息的文字部分。文本可以包含任何数量的替换记号。这些记号的格式是#{xxx}#，其中xxx表示任何字符串。这些记号将会被给定的字段验证里出现的数值替换。所以，如果你有像代码清单8-2中看到的第一个message那样的那句文字"#{fieldName}# is a required field."，假设一个特定的字段验证定义了fieldName参数，那么当展示信息的时候，这个参数的值将被插入#{fieldname}#中。

3. 定义表单

随后出现的是form元素。它相对应于页面中的一个实际的HTML表单。它有两个属性：name和noSubmitMessage。name属性需要精确匹配页面上那个，要应用验证程序的<form>标签的name属性。noSubmitMessage属性是在用户试图提交表单，并且表单的任意字段非法时展示给用户的信息。

4. 定义字段验证

form元素有任意数量的子元素validation，而且这里就是真正的行为开始的地方。validation元素有下面5个必须的属性。

- field：表单中这个将要应用这个验证的元素的名字。它必须严格匹配表单元素的name属性。
- stratInvalid：值是true或flase。当被设置为true时，最初字段将被标志为非法，如果设置了通过标记突出显示的方式来报错的话，在刚开始的时候它将被标记为突出显示。这对于那些类似Last Name那样有个最小长度限制，并且从空值开始的输入框是很有用的，那样就可以很清楚地知道最初它是没有达到最小长度限制的。
- event：输入框将要执行验证的那个事件。它可以是任何常见的DOM事件，但是onblur可能是最常用的。
- type：一个验证器的ID，可以是你自己配置的，也可以是内置的基本验证器。
- failAction：当输入框验证失败时要执行的操作。这可以有3种值：alert，也就是说通过alert()弹出框来把错误信息展示给用户；insert，就是通过innerHTML属性，把错误信息插入到指定的页面元素中；highlight，即一个指定输入框将被标记为突出显示。

5. 定义验证参数

内嵌在validation元素之下的是param元素。每一个param元素都有两个属性：name，这个参数的名字；value，参数的实际值。

param元素是一个灵活的机制，我们通过它给配置在一个给定元素上面的验证器传递信息以及给JSValidator传递在验证失败的时候使用的信息。由于灵活，参数就可以有各种你所希望的意义，并且每个验证器和验证失败动作都可以请求任何它们想要的信息。

举例来说，内置的MinLengthValidator验证一个输入框是否满足最小长度，它通过查找一个叫做minLength的参数为那个输入框获取最小长度的值。还有，RegexValidator允许一个输入框根据任意的正则表达式来进行验证，它通过regex参数获取预先设置的那个正则表达式。

对于alert和insert失败操作里面使用的那个消息元素的ID，也是通过message变量找到的。如果字符串中有任何需要替换的记号（比如像在代码清单8-2中看到的fieldName的那个信息），也是在参数列表中找到的。对于insert失败操作，要突出显示的元素的ID是在一个名叫idToHighlight的参数里保存的，还有合法和非法的样式类分别可以在errorStyleClass和OKStyleClass中找到。

正如我说过的，当写自己的验证器时，你可以请求任何需要的参数——这里并没有什么真正的限

制。不过，要注意的一点是，替换消息字符串中的记号的功能只发生在处理alert或insert失败的过程中。 不过，你可以使用与这些操作使用的相同的JSValidator里的replaceTokens()，我们一会儿看它。比如，如果你希望一个参数可以动态地插入一个值，就可以使用与处理消息中的记号相同的机制了。

8.5.4 编写JSValidatorObjects.js

JSValidatorObjects.js文件包含了7个不同的类的定义。除了一个之外，其他的都是在解析配置文件的时候填充的对象。这7个类本身并没有提供任何功能。它们基本上就是值对象（Value Object，VO），存储一些数据并提供获取这些数据的公共接口。因此，我相信你会发现它们速度很快、很容易理解。虽然如此，你还是需要知道它们的目的，所以我们会一个一个地看看它们，最后再从整体上看一下它们是如何组合在一起的。

1. JSValidatorValidatorImpl类

首先，看看图8-7，它展示了7个类中唯一的那个不代表配置信息的类的UML图。JSValidatorValidatorImpl类是所有验证器实现类的基类。一个验证器通常会需要从这个类继承并覆盖那个validate()方法，从而成为框架可以使用的一个验证器。

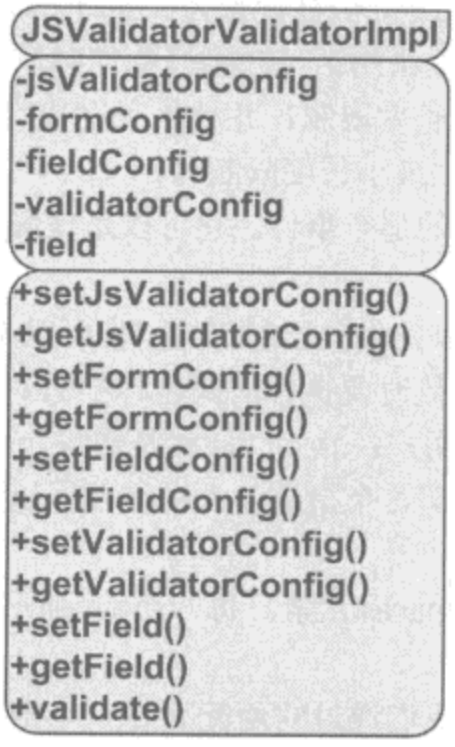


图8-7　JSValidatorValidatorImpl类的UML图

这个类中的5个字段都是在验证器验证输入框之前由框架代码填充的（使用相应的设置器方法）。

- jsValidatorConfig字段保存一个指向JSValidatorConfig实例的引用，这个对象是所有由配置文件解析而来的配置数据的父对象。
- formConfig字段保存了一个指向JSValidatorForm对象的引用，这个对象代表在配置文件中的那个触发了验证的<form>元素。
- fieldConfig字段保存了一个指向JSValidatorFormValidation对象的引用，这个对象是配置文件里的<validation>元素的JavaScript表现形式。它描述了要在指定字段上执行的验证。
- validatorConfig字段保存着一个指向JSValidatorValidatorConfig对象的引用，这个对象是配

置文件里的<validation>元素的JavaScript表现形式。它描述了将在指定字段上使用的验证。

- 最后是field字段，如果你能宽恕我的头韵[①]，它其实是指向那个触发了验证的表单字段本身。

如前所述，一个验证器通常只提供自己的validate()实现就完了。如果通过验证，那么这个方法返回true；如果没有通过验证，就返回false。当然，并没有说这里不能有其他的方法和字段。但是严格地说，这是唯一必须的东西。[②]

为了证明我没有假想这一切，下面是JSValidatorValidatorImpl类的代码。你可以看到，它真的是空的，并且它本身就是一个验证类的实现，虽然是一个相当没意义的实现！

```
function JSValidatorValidatorImpl() {

  this.jsValidatorConfig = null;
  this.formConfig = null;
  this.fieldConfig = null;
  this.validatorConfig = null;
  this.field = null;

  this.setJsValidatorConfig = function(inJsValidatorConfig) {
    this.jsValidatorConfig = inJsValidatorConfig;
  }
  this.getJsValidatorConfig = function() {
    return this.jsValidatorConfig;
  }

  this.setFormConfig = function(inFormConfig) {
    this.formConfig = inFormConfig;
  }
  this.getFormConfig = function() {
    return this.formConfig;
  }

  this.setFieldConfig = function(inFieldConfig) {
    this.fieldConfig = inFieldConfig;
  }
  this.getFieldConfig = function() {
    return this.fieldConfig;
  }

  this.setValidatorConfig = function(inValidatorConfig) {
    this.validatorConfig = inValidatorConfig;
  }
  this.getValidatorConfig = function() {
    return this.validatorConfig;
  }
```

① 抽取所有单词的首字母。

② 实际上，在JSValidatorValidatorImpl类里面有所有方法的空实现。所以，即使你写一个空类扩展JSValidatorValidatorImpl，它也不会破坏JSValidator——它只是不干什么正事而已。

```
    this.setField = function(inField) {
      this.field = inField;
    }
    this.getField = function() {
      return this.field;
    }

    this.validate = function() { }

  } // End JSValidatorValidatorImpl class.
```

2. JSValidatorConfig类

JSValidatorConfig类，如图8-8所示，位于对象实体关系层级结构的顶部，在这里你可以找到从配置文件解析而来的配置信息。3个字段分别代表了3个可以是<JSValidatorConfig>元素的直接子元素的元素：<validator>、<message>和<form>元素。每一个都是从配置文件解析出来的所有对应元素的集合。

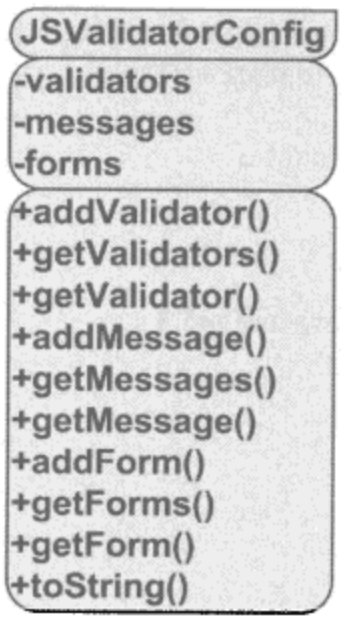


图8-8　JSValidatorConfig类的UML图

注意，为了调试方便，我们覆盖了toString()方法，让它输出更有意义的调试信息。所有剩下的配置文件对象都这么处理了，所以我就不再说一遍了。

现在让我们看看实际的代码吧。

```
function JSValidatorConfig() {

  var validators = new Object();
  var messages = new Object();
  var forms = new Object();
  this.addValidator = function(inValidatorConfig) {
    validators[inValidatorConfig.getId()] = inValidatorConfig;
  }
  this.getValidators = function() {
    return validators;
  }
```

```
this.getValidator = function(inID) {
  return validators[inID];
}

this.addMessage = function(inMessage) {
  messages[inMessage.getId()] = inMessage;
}
this.getMessages = function() {
  return messages;
}
this.getMessage = function(inID) {
  return messages[inID];
}

this.addForm = function(inForm) {
  forms[inForm.getName()] = inForm;
}
this.getForms = function() {
  return forms;
}
this.getForm = function(inName) {
  return forms[inName];
}

this.toString = function() {
  return "JSValidatorConfig=[" +
  "validators=" + validators + "," +
  "messages=" + messages + "," +
  "forms=" + forms + "]";
}

} // End JSValidatorConfig class.
```

除了这3个字段和toString()之外，这个类中包含的就是每个字段的获取方法（getter），以及每个集合的addXXX()方法。JSDigester使用这些方法来给相应的集合添加JSValidatorForm、JSValidator-Message和JSValidatorValidatorConfig实例。注意，除了用于获得集合的获取方法之外，每个都还有一个通过ID或名字从集合中获取特定元素的获取方法，可以根据需要使用ID或者名字来查找元素（对于验证器和消息是ID，对于表单是名字）。

3. JSValidatorValidatorConfig类

图8-9展示了JSValidatorValidatorConfig类，它是使用如下代码来实现的：

```
function JSValidatorValidatorConfig() {

  var id = null;
  var src = null;
  var clazz = null;

  this.getId = function() {
    return id;
```

```
}
this.setId = function(inID) {
  id = inID;
}

this.getSrc = function() {
  return src;
}
this.setSrc = function(inSRC) {
  src = inSRC;
}

this.getClass = function() {
  return clazz;
}
this.setClass = function(inClass) {
  clazz = inClass;
}

this.toString = function() {
  return "JSValidatorValidatorConfig=[" +
  "id=" + id + "," +
  "src=" + src + "," +
  "clazz=" + clazz + "]";
}

} // End JSValidatorValidatorConfig class.
```

代码为在配置文件中的每一个<validator>元素都创建填充了一个此类的实例。JSValidator需要这些信息在你定义的那个验证器上工作。在分解了配置文件之后，这里字段的意思应该很明显了。

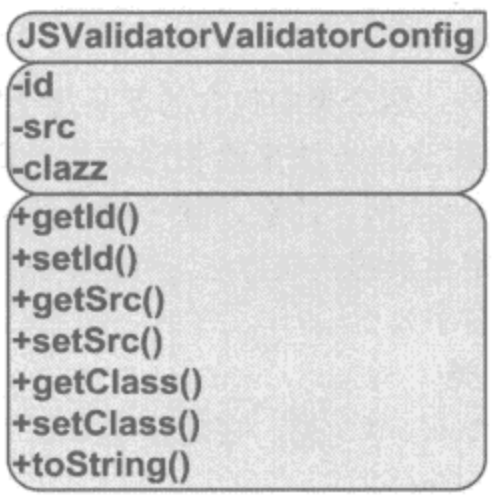


图8-9　JSValidatorValidatorConfig 类的UML图

你可能注意到了，clazz的使用好像是class的反写。IE对class并不十分友善，看起来可能是它的一个保留字。（不过奇怪的是，Firefox好像就没有这个class的问题！）这其实是在学Java，你可能知道，Java中class就是一个保留字。在典型的Java代码中，尤其是处理反应的Java代码，都是使用clazz来代替class的，那么在这个例子里也是同样的道理。所以虽然这并不是一个大问题，但是我还是想让你

注意一下，这里并不是我的一个愚蠢的打字错误——它是有来由的！

4. JSValidatorMessage类

继续我们在这些配置类上的征战，下面遇到的是在图8-10中显示的类：JSValidatorMessage。这个类代表了配置文件中的<message>元素。我还是认为这些字段的意思很明显，因为它们只是对应了<message>元素的属性，方法也只是对应字段的获取方法和设置方法。但是，为了完整性，让我们现在来看看这个类的代码，然后再看别的。

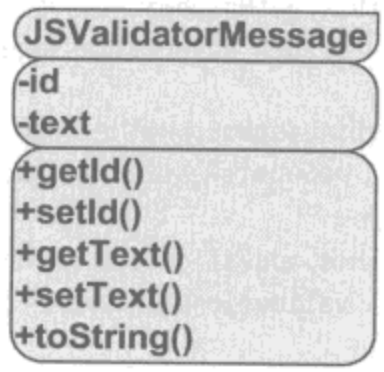


图8-10 JSValidatorMessage类的UML图

```
function JSValidatorMessage() {

  var id = null;
  var text = null;

  this.getId = function() {
    return id;
  }
  this.setId = function(inID) {
    id = inID;
  }

  this.getText = function() {
    return text;
  }
  this.setText = function(inText) {
    text = inText;
  }

  this.toString = function() {
    return "JSValidatorMessage=[" +
    "id=" + id + "," +
    "text=" + text + "]";
  }

} // End JSValidatorMessage class.
```

5. JSValidatorForm 类

在图8-11中，我们遇到了JSValidatorForm类。下面是与那个UML图相匹配的代码：

```
function JSValidatorForm() {

  var name = null;
  var noSubmitMessage = null;
  var validations = new Object;

  this.getName = function() {
    return name;
  }
```

```
this.setName = function(inName) {
  name = inName;
}

this.getNoSubmitMessage = function() {
  return noSubmitMessage;
}
this.setNoSubmitMessage = function(inNoSubmitMessage) {
  noSubmitMessage = inNoSubmitMessage;
}
this.addValidation = function(inValidation) {
  validations[inValidation.getField()] = inValidation;
}
this.getValidations = function() {
  return validations;
}
this.getValidation = function(inField) {
  return validations[inField];
}

this.toString = function() {
  return "JSValidatorForm=[" +
  "name=" + name + ", " +
  "validations=" + validations + "]";
}

} // End JSValidatorForm class.
```

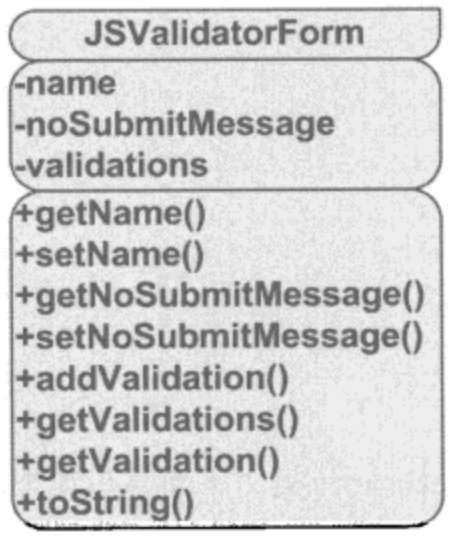


图8-11　JSValidatorForm类的UML图

没错，我又要说，这个类匹配和代表了配置文件中的<form>元素。那么这些字段也都应该是很明显了。不过，我们确实有一些有意思的事情可以说说。

一个应该注意的地方是validations字段并不是一个数组。实际上，如果你回过头去看看JSValidatorConfig类，就会注意到还有3个集合不是数组。为什么这个会值得关注呢？因为，假设你没有使用额外的库，在大多数语言中，一个“集合”对绝大部分人来说都是一个数组的意思。那么我

为什么在这里却使用了一个Object呢？

你可能已经听说过，在JavaScript中对象本身也是一个关联数组。“啊！”我能听到你在喊，“就是这个原因！”实际上，我所使用的正是一个数组，它只不过不是你所想的那个纯粹的Array对象而已。使用一个对象，可以允许你通过名字引用数组的元素，这恰恰就是JSValidator需要的。举个例子来说，一个<form>元素可以有任意数量的内嵌<validation>元素。我们希望可以通过名字把它们拉出来，并把它们设置为一个Object的元素。我们还可以遍历它们，就像JSValidator.js的代码中你将会看到的那样，不过通过名称抓出元素还只是个基本原因。

另外一个值得注意的地方是，除了一些特例之外，这些类中的所有字段都是私有的。这就是为什么需要获取方法和设置方法的原因了（如果它们是公有的，那些方法就是多余的了）。这是很好的、常用的面向对象设计，但是这其实对于绝大部分JavaScript开发者来说，都还是个未知的概念。（坦言之，我也经历过编写面向对象的JavaScript，但却还不知道这么用的时光！）

6. JSValidatorFormValidation类

看看图8-12中的UML图，从提供的信息的角度来看，这是最重要的一个类。JSValidatorFormValidation是这样一个类，我们从中创建那个用来执行发生在一个给定字段上面的验证，并且定义了当验证失败时应该怎么做的对象的类。JSValidatorFormValidation类的代码如下所示：

```
function JSValidatorFormValidation() {

  var field = null;
  var event = null;
  var type = null;
  var failAction = null;
  var startInvalid = null;
  var params = new Object();

  this.getField = function() {
    return field;
  }
  this.setField = function(inField) {
    field = inField;
  }

  this.getEvent = function() {
    return event;
  }
  this.setEvent = function(inEvent) {
    event = inEvent;
  }

  this.getType = function() {
    return type;
  }
  this.setType = function(inType) {
    type = inType;
  }
```

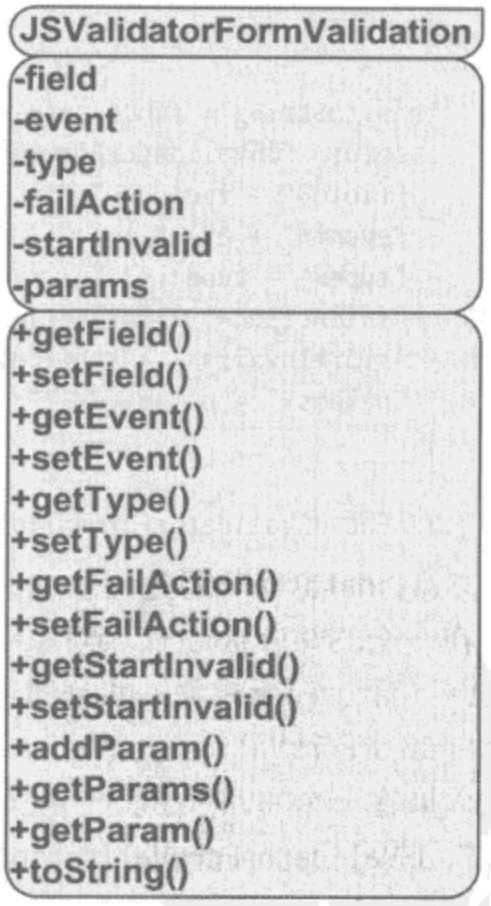


图8-12 JSValidatorFormValidation类的UML图

```
this.getFailAction = function() {
  return failAction;
}
this.setFailAction = function(inFailAction) {
  failAction = inFailAction;
}

this.getStartInvalid = function() {
  return startInvalid;
}
this.setStartInvalid = function(inStartInvalid) {
  startInvalid = inStartInvalid;
}

this.addParam = function(inParam) {
  params[inParam.getName()] = inParam;
}
this.getParams = function() {
  return params;
}
this.getParam = function(inName) {
  return params[inName];
}

this.toString = function() {
  return "JSValidatorFormValidation=[" +
    "field=" + field + "," +
    "event=" + event + "," +
    "type=" + type + "," +
    "failAction=" + failAction + "," +
    "startInvalid=" + startInvalid + "," +
    "params=" + params + "]";
  }

} // End JSValidatorFormValidation class.
```

JSValidatorForm还是一个相关配置信息的存储单元。在一个JSValidatorForm对象中可以内嵌任意数量的JSValidatorForm对象，并且通过扩展，下一个类，JSValidatorFormValidationParam，也可以通过params字段内嵌在一个JSValidatorForm对象中。

7. JSValidatorFormValidationParam类

图8-13展示了JSValidatorFormValidationParam类的UML图，正如你所料，这个类的代码很简单。

图8-13　JSValidatorFormValidationParam类的UML图

```
function JSValidatorFormValidationParam() {

  var name = null;
  var value = null;
```

```
    this.getName = function() {
      return name;
    }
    this.setName = function(inName) {
      name = inName;
    }

    this.getValue = function() {
      return value;
    }
    this.setValue = function(inValue) {
      value = inValue;
    }

    this.toString = function() {
      return "JSValidatorFormValidation=[" +
      "name=" + name + "," +
      "value=" + value + "]";
    }

} // End JSValidatorFormValidationParam class.
```

8. 这些类是如何配合在一起的

现在，你已经看过了每个JSValidatorObjects.js类的UML图和代码了，并且了解了它们与XML配置文件中的元素有什么关系。下一步就是要理解它们是如何在一起合作的。在之前讲解这些类的时候，我在文字上或多或少地描述了一些它们之间的关系，这个关系与我们在配置文件中看到的有些雷同，但是我还是希望给你展示一个图表。一个实体关系图应该可以很好地完成这个任务，这就是图8-14。

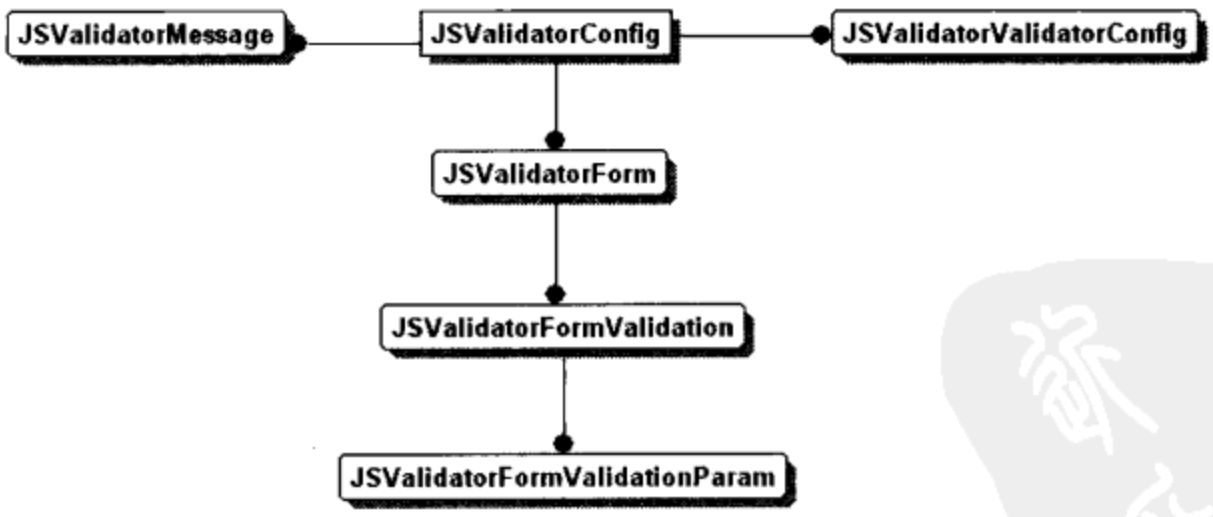


图8-14 展示所有的JSValidatorObject类是如何组合到一起的实体关系层级结构图

8.5.5 编写JSValidator.js

现在，我们来到的部分是可以称作JSValidator框架本身的部分，巧合中的巧合，它就是在JSValidator.js中！

在看代码之前，先来看看这个类的UML图吧。我想在你看到图8-15的时候会非常惊讶地发现，它其实就是那么小的一个类。

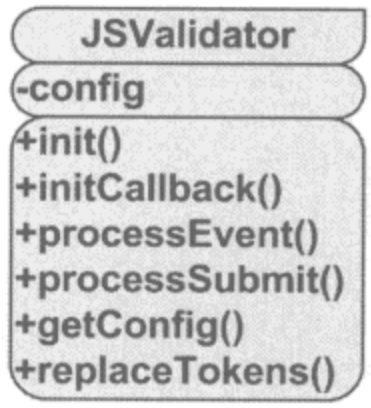


图8-15　JSValidator类的UML图

由于这个类的代码比我们之前看过的都多了不少，我就不在这里把整个代码都列出来了，不过我们还是会去看看要用到的代码片段的。

首先要看的是一个字段：config。config字段是一个JSValidatorConfig对象的引用，我们在前面介绍过这个对象了，它保存了从配置文件解析而来的所有配置信息。

然后我们跳过这个类中的所有方法，去看看这个文件中在JSValidator的定义之后找到的代码：

```
// Include dependencies.
document.write('<script src="' + JSVConfig.pathPrefix +
  'prototype.js"></script>');
document.write('<script src="' + JSVConfig.pathPrefix +
  'sax.js"></script>');
document.write('<script src="' + JSVConfig.pathPrefix +
  'JSDigester.js"></script>');
document.write('<script src="' + JSVConfig.pathPrefix +
  'JSValidatorObjects.js"></script>');
document.write('<script src="' + JSVConfig.pathPrefix +
  'JSValidatorBasicValidators.js"></script>');

// Instantiate JSValidator.
jsValidator = new JSValidator();

// Set onload event to configure JSValidator, unless told not to.
if (!JSVConfig.manualInit) {
  window.onload = function() {
    jsValidator.init();
  };
}
```

我们可以认为5个document.write()调用是为JSValidator引入所需的元素。注意，他们使用在JSVConfig中定义的pathPrefix元素来构建导入的脚本文件的URL。下面是所引入的内容。

- 我们肯定需要Prototype库，所以首先看到的是它。
- 我们都知道，JSDigester是建立在Mozilla项目的SAX解析器的基础之上的，所以第二个引入的是它（SAX解析器）。

- 然后是JSDigester。
- 下一个是引入JSValidatorObjects.js文件，它包含了我们之前看到的所有那些配置类。
- 最后引入的是一个JSValidator提供的基础的、内置的验证器，我们会在JSValidator之后介绍它。

这5个语句是在页面载入时执行的，所以使用这种动态的方法将<script>标签写进页面中，也可以使浏览器下载这些源代码。这就是一个开发者如何在只导入一个.js文件的情况下，就可使使用JSValidator的原因。其余的部分其实是包含在JSValidator.js之中的，因此从开发人员的角度看也是“自动的”。

JavaScript文件用这种方式从其他文件引入更多东西是一种非常便利的机制，它使你可以在保持代码独立的同时，还能够很方便地获得所需要的一切。另外一种方法，就是把所有这些JavaScript文件都整合到一个文件中，从减少对服务器请求的角度看，这样可能更好。但是把不同组件尽量保持在独立的文件中是更强大的一种方法，因而它们可以很容易地独立更新，并且坦率地说这也是全功能语言应该具备的特征。所以我的建议是，除非你明确知道你有这么做的理由，否则就应该让代码保持整洁和独立，这样会更有帮助。

在那5个声明之后，是一个提供了JSValidator实例的语句，我们在后面的代码中都会使用它。随即出现的就是自动初始化JSValidator的代码——如果配置为自动初始化。记住，开发者可以把JSVConfig中的manualInit元素设置为false来控制框架的初始化。这是通过为页面设置onLoad事件处理函数，将其指向JSValidator类的init()方法来完成的。请注意，我们必须这么做，而不是简单地在那里调用init()，因为页面需要被完全载入；否则，那些执行和访问DOM的代码可能失败，因为页面还没有载入完毕。

1. init()方法

这个init()方法看上去像什么？好，这是我们的下一步。这里就是：

```
this.init = function() {

  // Use Prototype to load the configuration file.
  new Ajax.Request(
    JSVConfig.pathPrefix + JSVConfig.configFile,
    { method: "get", onComplete: this.initCallback }
  );

} // End init().
```

这是你希望的吗？肯定不是！这里所做的，只是通过Prototype的Ajax支持载入在JSVConfig结构中指定的配置文件。我们声明在收到服务器应答（也就是，配置文件本身）的时候应该调用JSValidator类的initCallback()方法。那么，就如你所期望的，那个函数就是我们现在要看的。

2. initCallback()方法

由于initCallback()方法稍微有些长，我们将只看看那些有助于我们理解它执行的每个任务的代码片段。这些部分会按照它们出现的顺序来介绍，也就是说，把下面这些代码部分放到一起，就相当于完整的方法。

- 配置JSDigester规则

让我们从这段代码开始：

```
var jsDigester = new JSDigester();

// Create new object when JSValidatorConfig tag encountered.
jsDigester.addObjectCreate("JSValidatorConfig",
  "JSValidatorConfig");

// Create new object when JSValidatorConfig/validator tag encountered,
// populate its properties and add it to the JSValidatorConfig object
// on the top of the stack.
jsDigester.addObjectCreate("JSValidatorConfig/validator",
  "JSValidatorValidatorConfig");
jsDigester.addSetProperties("JSValidatorConfig/validator");
jsDigester.addSetNext("JSValidatorConfig/validator", "addValidator");

// Create new object when JSValidatorConfig/message tag encountered,
// populate its properties and add it to the JSValidatorConfig object
// on the top of the stack.
jsDigester.addObjectCreate("JSValidatorConfig/message",
  "JSValidatorMessage");
jsDigester.addSetProperties("JSValidatorConfig/message");
jsDigester.addSetNext("JSValidatorConfig/message", "addMessage");

// Create new object when JSValidatorConfig/form tag encountered,
// populate its properties and add it to the JSValidatorConfig object
// on the top of the stack.
jsDigester.addObjectCreate("JSValidatorConfig/form",
  "JSValidatorForm");
jsDigester.addSetProperties("JSValidatorConfig/form");
jsDigester.addSetNext("JSValidatorConfig/form", "addForm");
// Create new object when JSValidatorConfig/form/validation tag encountered,
// populate its properties and add it to the JSValidatorForm object
// on the top of the stack.
jsDigester.addObjectCreate("JSValidatorConfig/form/validation",
  "JSValidatorFormValidation");
jsDigester.addSetProperties("JSValidatorConfig/form/validation");
jsDigester.addSetNext("JSValidatorConfig/form/validation", "addValidation");

// Create new object when JSValidatorConfig/form/validation/param tag
// encountered, populate its properties and add it to the
// JSValidatorFormValidation object on the top of the stack.
jsDigester.addObjectCreate("JSValidatorConfig/form/validation/param",
  "JSValidatorFormValidationParam");
jsDigester.addSetProperties("JSValidatorConfig/form/validation/param");
jsDigester.addSetNext("JSValidatorConfig/form/validation/param",
  "addParam");

// Parse config.
config = jsDigester.parse(inRequest.responseText);
```

所有这些，简单地说，就是对应JSValidator配置文件的JSDigester的配置规则。因为我们已经讲过

JSDigester了（在前面的章节中），所以逐一详细介绍这些规则就有些浪费了。不过，如果你觉得这些代码里有任何东西看不懂，那我强烈建议你去看看第7章。

在配置文件上执行这些规则的结果是JSValidator类的config字段成为一个JSValidatorConfig的实例，并且里面所有的子类的也都填充为来自配置文件的信息。这段代码存在于initCallback()函数，Prototype 会给它传递自己制作的用于检索配置文件的Ajax请求的请求对象。因此，你可以在jsDigester.parser()的调用里看到，要被解析的XML可以通过inRequest对象的responseText属性来访问。

● 添加内置的验证器

解析完配置文件之后，我们要执行的下一个任务是添加内置的验证器：Required、Regex和MinLength验证器。实现它们的代码如下：

```
// Add in the basic validators.
var requiredValidatorConfig = new JSValidatorValidatorConfig();
requiredValidatorConfig.setId("required");
requiredValidatorConfig.setSrc("");
requiredValidatorConfig.setClass("RequiredValidator");
config.addValidator(requiredValidatorConfig);
var regexValidatorConfig = new JSValidatorValidatorConfig();
regexValidatorConfig.setId("regex");
regexValidatorConfig.setSrc("");
regexValidatorConfig.setClass("RegexValidator");
config.addValidator(regexValidatorConfig);
var minLengthValidatorConfig = new JSValidatorValidatorConfig();
minLengthValidatorConfig.setId("minLength");
minLengthValidatorConfig.setSrc("");
minLengthValidatorConfig.setClass("MinLengthValidator");
config.addValidator(minLengthValidatorConfig);
```

每一个验证器的代码都分别实例化了一个新的JSValidatorValidatorConfig对象。然后，我们填充它所包含的3个属性：id、src和class。注意，src属性被设置为一个空字符串，是因为不需要引入什么文件。可能你会想起来，这些验证器的代码是在载入.js文件的时候就已经作为全局代码的一部分导入了。实际上，没有什么理由一定要为这些验证器设置src属性，不过我是想让代码尽量明确，即便是并不一定需要这么做。

● 载入客户验证器

对于验证器，我们还有一些工作要做。现在，我们需要导入那些用户附加的验证器的JavaScript文件，比如DateValidator验证器。当这段代码被执行时，页面其实已经载入完毕了，所以我们不能像之前那样用document.write()方法。取而代之地，我们需要稍微更有趣一点的方法：

```
// Add includes for external validators.
var configuredValidators = config.getValidators();
for (var validatorID in configuredValidators) {
  var nextValidatorConfig = configuredValidators[validatorID];
  // Only non-basic validators will have a src value specified.
  if (nextValidatorConfig.getSrc() != "") {
    var scriptTag = document.createElement("script");
    scriptTag.src = nextValidatorConfig.getSrc();
```

```
    var headTag = document.getElementsByTagName("head").item(0);
    headTag.appendChild(scriptTag);
  }
}
```

首先，我们得到从配置文件解析出来的所有验证器的集合，然后遍历这个集合。对于每个元素，我们取得相应的JSValidatorValidatorConfig对象，并看看它的src属性是否有值。（因为，你可能还记得，我们只是添加了基本验证器，它们的src属性都设置成一个空字符串——看看，这就是明确的好处！）得到src属性值之后，就使用DOM的createElement()方法来创建一个新的<scritp>标签。把这个标签的src属性设置为验证器声明的值。然后获得文档的<head>的引用，给它附加上那个新的<script>标签。然后浏览器会立即载入JavaScript源文件，并运行它。那么现在，我们就有空去看看剩下那些配置给验证器的类的代码了。

- 附加事件处理函数

下一步，也是初始化的最后一步，就是把配置好的事件与那些配置好需要验证的表单字段挂钩。这些都在下面的这段代码中完成，它看起来可能有点让人畏缩，不过如果把那些注释都删除了再看，你会发现它其实比你开始想象的要简单得多。

```
// Attach event handlers to fields as defined in config file.
var configuredForms = config.getForms();
// Iterate over forms configured.
for (var formName in configuredForms) {
  var nextFormConfig = configuredForms[formName];
  // Get reference to form being configured.
  var targetForm = document.forms[nextFormConfig.getName()];
  // Attach an onSubmit handler to check if it can submit or not.
  targetForm.onsubmit = jsValidator.processSubmit;
  // Get reference for all validations configured for this form.
  var formValidations = nextFormConfig.getValidations();
  // Iterate over validations defined for this form.
  for (var fieldName in formValidations) {
    // Get the field validation being hooked.
    var nextValidationConfig = formValidations[fieldName];
    // Get the validator definition.
    var validator = config.getValidator(nextValidationConfig.getType());
    // Get the field to hook event to.
    var targetField = targetForm[nextValidationConfig.getField()];
    // Set attribute if this field is initially invalid. and if the field
    // is configured for the highlight action, then highlight it.
    if (nextValidationConfig.getStartInvalid() &&
      nextValidationConfig.getStartInvalid() == "true") {
      targetField.setAttribute("JSValidator_INVALID", "true");
      if (nextValidationConfig.getFailAction() == "highlight") {
        var idToHighlight =
          nextValidationConfig.getParam("idToHighlight").getValue();
        var errorStyleClass =
          nextValidationConfig.getParam("errorStyleClass").getValue();
        $(idToHighlight).className = errorStyleClass;
```

```
        }
      }
      // Set event handler.
      targetField[nextValidationConfig.getEvent()] = jsValidator.processEvent;
    }
  }
```

首先，我们获得配置里的表单集合。然后代码开始遍历那个集合。对于每一个元素，我们通过document.forms[]数组来获取指向它的HTML的引用。把onSubmit事件处理函数附加到表单上。这个事件处理函数是JSValidator对象中的processSubmit()函数，后面会介绍。

然后，我们从JSValidatorConfig对象中得到验证器的集合，并开始遍历它。对于每个元素，我们都有一个指向HTML表单中字段的引用。然后，我们要看看是否没有设置startInvalid属性，或者它是否设置为true了。如果这两个条件之中有真的，那么我们就给这个字段添加一个名叫JSValidator_INVALID的属性，值为true（是什么值其实是没关系的，只要是否有这个属性就可以决定字段是否为非法的）。我们还要看看，那个输入框的failAction是否为highlight。如果是，我们就去找到要突出显示的页面元素的ID和用于突出显示的样式类，我们是用Prototype的$()（document.getElement-ById()的缩写）还获取它们的。

最后，我们给字段设置适当的事件处理函数，指向JSValidator类的processEvent()方法。

所有这些，其实就是以啰嗦的方法说明：给配置文件中每个配置了验证逻辑的字段都设置上合适的事件处理函数和合适的初始状态。这样做的结果是，即使JavaScript被禁用了，表单仍然可以工作，因为它根本不依赖于JSValidator，框架结构就是一个附加的功能而已。这就是好的、分离式编码。

看完了init()和initCallback()之后，我们现在把注意力转移到用来响应字段触发了验证事件的函数上。

3. processEvent()方法

回想一下，在initCallback()中，我们根据配置文件给每一个分配了验证逻辑的字段都附加了合适的事件处理函数，不过JSValidator类总是调用processEvent()。这样就允许框架以同样的方式处理所有的事件。让我们现在就去看看这个事件处理函数到底是怎么做的：

```
this.processEvent = function() {

  // Get reference to form, field and validator config objects for the
  // form element that fired the event that called this callback.
  var formConfig = config.getForm(this.form.name);
  var fieldConfig = formConfig.getValidation(this.name);
  var validatorConfig = config.getValidator(fieldConfig.getType());

  // Get a reference to the class that implements the validator defined
  // for this field.  Then, get a new instance of it and call its validate()
  // method.
  var clazz = eval(validatorConfig.getClass());
  clazz = new clazz;
  clazz.setJsValidatorConfig(config);
  clazz.setFieldConfig(fieldConfig);
  clazz.setValidatorConfig(validatorConfig);
```

```
    clazz.setField(this);
    // Perform the appropriate action for pass and fail.
    var isValid = clazz.validate();
    if (isValid) {
      // When field was valid, might be some cleanup to do in some cases.
      this.removeAttribute("JSValidator_INVALID");
      if (fieldConfig.getFailAction() == "highlight") {
        var idToHighlight = fieldConfig.getParam("idToHighlight").getValue();
        var okStyleClass =
          fieldConfig.getParam("okStyleClass").getValue();
        $(idToHighlight).className = okStyleClass;
      }
      if (fieldConfig.getFailAction() == "insert") {
        var targetID = fieldConfig.getParam("idToInsertInto").getValue();
        $(targetID).innerHTML = "";
      }
    } else {
      // Field NOT valid, so act according to config.
      this.setAttribute("JSValidator_INVALID", "true");
      switch (fieldConfig.getFailAction()) {
        case "alert":
          var whatMessage = fieldConfig.getParam("message");
          var messageConfig =
            jsValidator.getConfig().getMessage(whatMessage.getValue());
          var message = messageConfig.getText();
          message = jsValidator.replaceTokens(message, fieldConfig.getParams());
          alert(message);
        break;
        case "highlight":
          var idToHighlight = fieldConfig.getParam("idToHighlight").getValue();
          var errorStyleClass =
            fieldConfig.getParam("errorStyleClass").getValue();
          $(idToHighlight).className = errorStyleClass;
        break;
        case "insert":
          var whatMessage = fieldConfig.getParam("message");
          var targetID = fieldConfig.getParam("idToInsertInto").getValue();
          var messageConfig =
            jsValidator.getConfig().getMessage(whatMessage.getValue());
          var message = messageConfig.getText();
          message = jsValidator.replaceTokens(message, fieldConfig.getParams());
          $(targetID).innerHTML = message;
        break;
      }
    }

    return isValid;

  } // End processEvent().
```

首先，我们获取表单、字段和字段使用的验证器的config对象。然后，实例化对象里配置的验证器。实现这些的方法是：首先eval()调用validatorConfig.getClass()（它给我们一个对应类的引用）返回的类名，然后使用new关键字创建一个新实例。然后我们为它设置验证器JSValidatorConfig的实例，方法是给它传递setJsValidatorconfig()；以及给它的字段设置JSValidatorFormValidation，方法是给字段实例传递setFieldConfig()，还有就是给JSValidatorValidatorConfig设置实例，方法是把实例传递给setValidatorConfig()。我们还把触发事件的字段传递给setField()设置字段。

现在验证器应该拥有它需要的所有的信息了，所以我们就在验证器上调用validate()。如果字段通过验证，validate()就返回true，否则返回false。如果validate()返回真，我们就需要撤销该字段可能有的所有无效输入的标识。那就是说，如果存在 JSValidator_INVALID标志，那我们首先要删除它，如果该字段的failAction是highlight，我们需要先把该元素的突出显示撤销掉。

最后，如果字段的failAction是insert，那么我们就需要清理目标元素的innerHTML属性。你可以在Last Name字段上看到所有这些。那个字段开始时是无效输入，当敲入超过5个字符之后，这个字段的标签就切换成一个合法输入的条件。这些就是这段代码的作用。

如果一个字段验证没通过怎么办？好的，那样我们就来到了这段代码的else部分。这里，我们检查failAction值，然后根据其值进行分支。如果是alert动作，那么我们就拿来配置里引用的消息，然后调用replaceTokens()做记号替换。最后，用alert()显示结果消息。

如果是highlight，我们只需要获取要突出显示的元素的ID，还有需要使用的样式。然后我们设置相应的className属性，这里用的还是Prototype的$()缩写。

最后，如果是insert动作，那么要做的实际上和alert一样，只是更新目标元素的innerHTML属性，而不是最后调用alert()。

4. processSubmit()方法

需要讨论的另外一个函数是表单自己的onSumit事件。很显然，如果表单上的任意一个字段非法，我们就不应该允许表单的提交。不过，我们如何判断是否应该提交表单呢？在这之后真正的机制是什么？答案可以在下面展示的JSValidator的processSubmit()中找到：

```
this.processSubmit = function() {

  var formValidity = true;
  // If any element of the form has the JSValidator_INVALID attribute, then
  // the form cannot be submitted.
  for (var i = 0; i < this.elements.length; i++) {
    if (this.elements[i].getAttribute("JSValidator_INVALID")) {
      formValidity = false;
    }
  }
  if (!formValidity) {
    // Can't be submitted, show configured message.
    var config = jsValidator.getConfig();
    var formConfig = config.getForm(this.name);
    alert(formConfig.getNoSubmitMessage());
  }
```

```
    return formValidity;

  } // End processSubmit().
```

正如你看到的，我们其实所需要做的就是遍历表单中的所有元素，并检查它们是否含有JSValidator_INVALID属性。如果都没有，那么这个表单就是合法的，可以继续提交。如果有任何元素有那个属性，那我们就获取表单的noSubmitMessage属性，并通过alert()来展示它，告诉用户这个表单上存在一个错误。

注意，这里对this引用的使用（你会注意到processEvent()方法也用了同样的东西）。你可能发现这很有意思，因为既然这些方法都是JSValidator实例的一部分，那么你可能就希望this指向JSValidator实例。但是，回想一下，这两个方法是被附加给页面中的元素的：表单和表单的元素。当它们被调用的时候，其实是在那些元素的环境中，因此this关键字实际上指向了那些元素。请记住，this引用总是指向该方法运行时附着的对象，而不是指向设计的对象。这个很容易弄混，而且要是干活儿干累了，这个东西很可能把你的程序搞砸。

5. replaceTokens()方法

JSValidator提供的最后一点功能是replaceTokens()方法，用于生成在字段验证失效的时候用户看到的消息（如果配置为看消息的话）。那让我们看看这个方法吧。

```
this.replaceTokens = function(inString, inParams) {

  // We're going to scan the text looking for tokens, all the while
  // constructing a new string in a StringBuffer from it, with the
  // data replacing the tokens.
  var finalText = "";
  var i = 0;
  while (i < inString.length) {
    // See if the next character is a hash sign, and if the next
    // character after that is an opening brace, as long as
    // that check doesn't put us beyond the end of the string, then we've
    // found the start of a token.
    if (inString.charAt(i) == '#' && inString.charAt(i + 1) == '{') {
      // Now we get the location of the closing token delimiter.  Note that
      // if the developer forgot to close the token, this will probably
      // blow up with a JS error, and at best it just won't work as
      // expected.  We're going to live with that!
      var lIndex = inString.indexOf("}#", i);
      // Now it's a simple matter to get the token name.
      var tokenName = inString.substring(i + 2, lIndex);
      // Look up the replacement value with that name from inParams.
      var tokenValue = "";
      var param = inParams[tokenName];
      if (param) {
        tokenValue = param.getValue();
      }
      finalText += tokenValue;
      // Set i to take us just past the closing token delimiter, and
```

```
      // we're done with this token
      i = lIndex + 1;
    } else {
      // The current character being checked was NOT part of a token
      // opening delimiter, so just append the character
      finalText += inString.charAt(i);
    }
    i++;
  }
  return finalText;

} // End replaceTokens().
```

replaceTokens()是一段相当直接的字符串操作代码。它一开始是遍历输入字符串里的每个字符，而输入字符串可能是包含形如#{xxx}#的替换记号的某个消息字符串。

检查每个字符，看看它是不是井号（#）。如果是，就检查下一个字符是不是花括弧（{）。如果两个条件之一为假，则执行else块中的代码，就是把字符追加到开始时构造的那个字符串上。不过，如果两个条件都是真，那就是说我们碰到一个要替换的记号了。然后我们需要找到分隔符的结束部分（}#）。找到之后，我们就能很容易地用String对象的substring()方法获取记号的名字。

拿到记号名字之后，我们可以在传递进来的参数集合里把它找出来。请注意，在这个场合下，“参数集合”实际上指向附着了正在进行中的验证器的所有配置参数的Object。然后我们就可以找出记号了。你肯定知道，这是因为用Object而不是Array的要点在于我们可以用[]语法很快就找出那些数值。找到数值之后，只是把其值附着在输出字符串上，然后把循环下标移动到结束分隔符后头。完成循环之后，我们返回构造好的字符串，乌啦！这就是用参数值替换了记号之后的字符串！

看完这些之后，我们就算已经了解了JSValidator的所有细节了！只剩下最后两段代码要看了，这些代码都是验证器的实现类——基本的内置验证器和附加验证器。

8.5.6 编写JSValidatorBasicValidators.js

JSValidatorBasicValidators.js文件，顾名思义，它是能找到JSValidator总是自动包括的3个内置验证器的地方，这3个验证器是：RequiredValidator、RegexValidator和MinLengthValidator。

1. RequiredValidator类

RequiredValidator在图8-16显示，是一个最简单的验证器，代码如下：

```
function RequiredValidator() {

  this.validate = function() {

    var retVal = true;
    if (this.field.value == "") {
      retVal = false;
    }
    return retVal;

  } // End validate().
```

```
} // End RequiredValidator().

// RequiredValidator extends JSValidatorValidatorImpl.
RequiredValidator.prototype = new JSValidatorValidatorImpl;
```

图8-16　RequiredValidator类的UML图表

这个类只是包含对触发验证的字段的一个简单检查，确保它里面有些输入就完了。请注意最后对prototype的设置。这里是继承JSValidatorVAlidatorImpl类的地方，你会在所有验证器上看到这行代码。同样，你自己的验证器也需要包含这行代码。

2. RegexValidator类

然后是RegexValidator，它也没复杂多少。首先，看看图8-17，RegexValidator类的UML图。

图8-17　RegexValidator类的UML图

看到代码的时候，我们会发现，在这个场合下，我们需要从定义这个验证器的参数里获取正则表达式。为了获取正则表达式，我们在这个验证器的fieldConfig对象上调用getParam()方法，然后是调用上面这个方法返回的对象上的getValue()方法，因为返回的是JSValidatorFormVAlidationParam对象，而不是值自身。做完这些之后，我们就只要用典型的JavaScript正则表达式match()函数来判断字段的值是否合法即可。

```
function RegexValidator() {

  this.validate = function() {

    var retVal = true;
    var parm = this.fieldConfig.getParam("regex");
    var regx = parm.getValue();
    if (!this.field.value.match(regx)) {
      retVal = false;
    }
    return retVal;

  } // End validate().

} // End RegexValidator().
// RegexValidator extends JSValidatorValidatorImpl.
RegexValidator.prototype = new JSValidatorValidatorImpl;
```

RegexValidator是那种看上去简单功能却很强大的东西。你拥有了JavaScript正则表达式引擎赐予你的所有功能，而不用自己写一行代码。

3. MinLengthValidator类

要看的最后一个基本验证器是MinLengthValidator，它又是一段很简单的代码。如图8-18所示，这又是一个典型的验证器类。

图8-18 MinLengthValidator类的UML图

看看代码：

```
function MinLengthValidator() {

  this.validate = function() {

    var retVal = true;
    if (this.field.value.length <
      this.fieldConfig.getParam("minLength").getValue()) {
      retVal = false;
    }
    return retVal;

  } // End validate().
} // End MinLengthValidator().

// MinLengthValidator extends JSValidatorValidatorImpl.
MinLengthValidator.prototype = new JSValidatorValidatorImpl;
```

可以看到，和RegexValidator一样，我们做的只是从该验证器的参数中获取输入域的最短长度要求，然后拿输入域的数值和它相比，然后返回合适的布尔值。

8.5.7　编写DateValidator.js

只差最后一段代码，我们就完成这个应用的全部分析了，这段代码是DateValidator类，如图8-19所示。

图8-19　DateValidator类的UML图

让我们看看代码，看看它是如何实现的。

```
function DateValidator() {

  this.validate = function() {

    // Get the configured format of the field.
    var format = this.fieldConfig.getParam("format").getValue();
    // Make sure the value is the same length as the format.
    if (this.field.value.length != format.length) {
      return false;
    }

    // Now iterate over the value.  If any character doesn't match the format,
    // it's a reject.  Note that M, D and Y characters in the format
    // correlate to any numeric character.
    for (var i = 0; i < format.length; i++) {
      if (format.charAt(i).toUpperCase() == "M" ||
        format.charAt(i).toUpperCase() == "D" ||
        format.charAt(i).toUpperCase() == "Y") {
        // Character at this position should be a numeric.
        if (this.field.value.charAt(i) < '0' ||
          this.field.value.charAt(i) > '9') {
          return false;
        }
      } else {
        // Format doesn't specify a numeric value at this position, so just
        // be sure the character matches exactly.
        if (format.charAt(i) != this.field.value.charAt(i)) {
          return false;
        }
      }
    }

    return true;

  } // End validate().

} // End DateValidator().

// DateValidator extends JSValidatorValidatorImpl.
DateValidator.prototype = new JSValidatorValidatorImpl;
```

我们看到的第一件事情是validate()方法从配置里面获取准备使用的格式信息。然后我们就看到一个很简单的拒绝代码：如果输入域的值不匹配格式字符串，那么我们就知道输入域是非法的。

假设输入的数据通过了上面的测验，那我们就遍历字段的数值。对每个字符，我们都看看它是不是格式字符串允许出现在那个位置的。格式字符串中的M、D和Y对应任意0到9的数字。如果格式字符串里的字符是其他的东西，比如通常是斜杠或者划线，那么就要要求输入域数值里的字符要完全一样。如果我们碰到输入字符串里的字符和格式字符串里的字符不匹配的情况，那么这个字段就没通过验证，我们就返回false。

这些就是本书的另外一个项目！

8.6　练习

我留下不少可以提高的地方，可以帮助你理解JSValidator，当然，也可以帮你理解JavaScript。下面是一些建议。

- 第一个是相当简单的一个：当一个字段验证失败时，它应该能自动获取焦点。这是一个非常明显的改进之处，也是我留给你改进的开端，因为增加这个特性应该不难，是一个上手的好办法。
- 另外一个比较明显的是实现更多的验证器：添加一个验证信用卡号码的如何？添加一个输入日期在某个范围之内的如何？添加一个两个输入域的大于和小于比较验证器又如何（这个需要你修改验证器的框架，以便允许验证配置可以给出第二个需要操作的输入域）？只要你能想到的，就实现它吧！
- 允许对一个输入域的多重验证：这个可能不像听上去那么简单，但绝对是一个有价值的目标。
- 添加国际化支持（I18n）。这个目的可以用多个方法实现。一个方法是给<message>元素添加区域标识符——它可以是en代表英语，de代表德语等。然后根据客户端的区域设置获取合适的消息。
- 允许给突出显示的错误域添加以提示标签形式弹出的错误信息。这样，在一个字段突出显示为验证失败的时候，用户可以用鼠标悬浮获取错误信息。

8.7　小结

在本章中，我们开发了一个可扩展的、完全可以外部化的表单验证器。它几乎不需要给我们的页面添加什么代码。你看到了如何给框架添加将来可以用在别处的可重用的验证器。并且你可以以一种漂亮、干净、面向对象（因而它也可以扩展）的方式实现它。

本章的项目中，你看到前面章节的JSDigester是如何帮我们处理XML。你还看到了一些简单Prototype库的使用，以及其他一些跟Ajax相关的简单的东西。我们会在第12章中做一个专门了解Ajax的项目。

第9章

痴迷于窗口小部件：使用GUI窗口小部件框架

在如今的Web开发中，窗口小部件（widget）风靡一时。我们不再满足于一成不变的表单输入域、按钮、下拉列表等。那些平淡无奇的表格更是让一些开发者不能满意！不过，这个趋势并不仅仅来自于对创建一个更酷界面的需求。在现代操作系统中，一些用户界面的表示方法，在Web世界中还没有对应物，至少是没有原生的。

以树状视图（tree view）为例。我敢肯定你之前见过树状视图。的确，如果你使用的是Windows，你几乎无法回避它（它就是在Windows Explorer左边的那个文件夹列表）。那么你在Web上看到过树状视图吗？有可能，不过除非是很新的网页，否则开发者很可能是自己写的，要不大多是从别处粘贴的一些代码。但是它多半都不是一段独立完整的代码，也不是一个较大的UI组件框架的一部分，而这些其实是UI窗口小部件的特征。

窗口小部件可以非常简便地帮你把这些“先进的”功能添加到Web应用程序中，更重要的是，还保留了一致性。后面我们在创建一个发布消息的小应用程序时，你就可以看到这个概念是如何工作的了。在这个项目中，我们将要使用大量的窗口小部件，它是由一个正在迅速流行的库所提供的，那就是：YUI库。

9.1 JSNotes的需求和目标

从现在起，我们叫本章的项目为JSNotes，我们会创建一个类似你每天会在办公桌上使用的，那个黄色可粘贴的便签簿样子的东西。不过在这里，它将是一个基于Web的应用程序，我们可以用它来张贴数字便签。这个应用程序的主要目的除了帮助我们保管每天获得的那些零碎信息以防丢失之外，它还是一个使用YUI库的很好的例子。

那么，JSNotes有什么特殊之处呢？现在我们就逐一地说说。

- 我们可以使用JSNotes创建便签，其中包括主题、添加日期和时间（通常是当前的日期和时间，但你不用知道）。我们还可以把便签分类，标记为个人的或者公务的。
- 我们将会使用类似Windows Explorer的样子来展示便签，在左边是用树状视图，右边放置便签的内容。树状视图中的两个主要分支就是分类：个人便签和公务便签。
- 我们存储每个便签的文字内容、标题以及时间和日期，还有标签的类别。
- 我们可以删除一个标签，也可以“导出”标签，其实就是把它改为适当的格式，可以便于复制粘贴以用于其他程序。

- 添加便签的时候，我们想要用一个弹出的对话框来展现。
- 还有Help和About框，不过它们要使用不同于添加便签时那个弹出框的风格。这些会以覆盖在JSNotes上面的形式来展现。

好消息是，把YUI库掺和进来之后，所有这些都变成了相当简单的练习！

9.2 YUI库

YUI库包括一个UI窗口小部件集合以及大量用于Ajax、鼠标拖放、DOM操作等的工具类。YUI是我最近所见过的文档和样例最充足的一个库。实际上，你可以找到库中每一部分的很好的例子，而且通常还附加了各种各样的说明手册。

既然窗口小部件是我们在这个应用程序中主要使用的东西，并且还会用好几个，那么就让我们来说说它吧。YUI提供了大量窗口小部件，包括AutoComplete、Calendar、Container（其中有Module、Overlay、Panel、Tooltip、Dialog和SimpleDialog）、Logger、 Menu、Slider、TabView以及TreeView。由于这些窗口小部件都是在一个公用的框架之上设计的，所以它们都有一个相对一致的程序接口。

使用YUI就像导入些JavaScript文件一样简单，有时候还会导入一些CSS文件。通常来讲，每一个窗口小部件就是一个单独的JavaScript文件，不过，可能也还是有一些对其他文件的依赖关系，需要手工导入。不过只要完成了这些，你就已经做好使用前的准备了。

例如，让我们假设你想要创建一个菜单。那么你所要做的就是，在简单的html标记中定义你的菜单，就像这样：

```
<div id="basicmenu" class="yuimenu">
  <div class="bd">
    <ul class="first-of-type">
      <li class="yuimenuitem"><a href="page1.htm">Page1</a></li>
      <li class="yuimenuitem"><a href="page2.htm">Page2</a></li>
    </ul>
  </div>
</div>
```

这样就有了菜单的基本架构。最后一步是告诉YUI，基于这个HTML标记为你创建一个菜单。很简单，这样做就可以了：

```
var oMenu = new YAHOO.widget.Menu("basicmenu");
oMenu.render();
```

YUI会在页面中查找带有basicmenu这个ID的元素。然后它就解析其中的内容，查找一个列表，并使用它来创建菜单。原有的HTML标记将会被菜单本身的代码所替换，然后菜单就设置完成了。在上面代码的第一行，实例化了一个菜单，并把它的入口存储在oMenu中，不过这只是在内存中呈现它。要想实际的在屏幕上看见，是它的render()函数要完成的工作。

你将会发现，大多数窗口小部件都遵循一个通用模式：首先实例化一个窗口小部件的实例，给它一些HTML标签的引用，然后调用它的render()方法把它放在屏幕上。有些窗口小部件还支持传入构造函数的参数选项，这些你将会在本章应用程序的代码中看到。它们之中有些创建方法是有些区别的，在我们构建那个应用程序的时候，你就会了解到了。

你也可以从零开始用程序来创建窗口小部件。例如直接使用代码创建一个菜单。可以这样做：

```
var oMenu = new YAHOO.widget.Menu("basicmenu");
oMenu.addItem(new YAHOO.widget.MenuItem("Page1", { url : "page1.htm" } ));
oMenu.addItem(new YAHOO.widget.MenuItem("Page2", { url : "page2.htm" } ));
oMenu.render(document.body);
```

在这个例子中，假设页面里还没有带有basicmenu这个ID的元素，那么我们将创建菜单的DOM结构，赋给它那个ID，并把它添加到文档的body元素上。

如我所说的，YUI还提供了一些工具类的函数。下面就是我们将要在JSNotes应用程序中使用的一个：

```
var obj = YAHOO.util.Dom.get("xxx");
```

它与下面这个常用的函数是等价的：

```
var obj = document.getElementById("xxx");
```

你可能想知道这两个函数有什么不同呢。答案就是，YUI版本可以接收一个单独的ID或者一组ID，并且返回对于一个单独元素或者一组元素的引用。即使不说别的，如果你想要同时获得对不止一个元素的引用时，这都能为你节省大量的打字工作和代码量。

YUI被追逐和热捧是相当有道理的。它在所有的主流浏览器中都可以正常工作，它设计得很好，并且有相当完美的支持文档。我诚挚地推荐你使用它，而且希望你从http：//developer.yahoo.com/yui把它下载下来，进行更为细致的研究。虽然我们将会使用到一些窗口小部件，但是那个库中还有很大一部分是我们没有涉及的。所以我建议你从http：//developer.yahoo.com/yui把它下载下来，更仔细地看看。即使是只花5分钟的时间来看看网站上的那些例子，都会帮你更好地理解它所提供的功能。

9.3 JSNotes的预览

如果你还没有尝试使用过JSNotes，那我建议你现在就去试试。下面我们快速地看一眼。图9-1展示了应用程序启动时候的界面，这里你就已经碰到了两个YUI窗口小部件：菜单（menu）和树状视图（tree view）。

图9-1 启动JSNotes

当你想要添加一个便签时，点击File菜单中的Add Note选项。这时出现一个对话框，你可以在那里添加便签信息。这个对话框示范了另外一些窗口小部件，包括对话框本身、日历以及滑块（slider），正如你在图9-2中可以看到的。

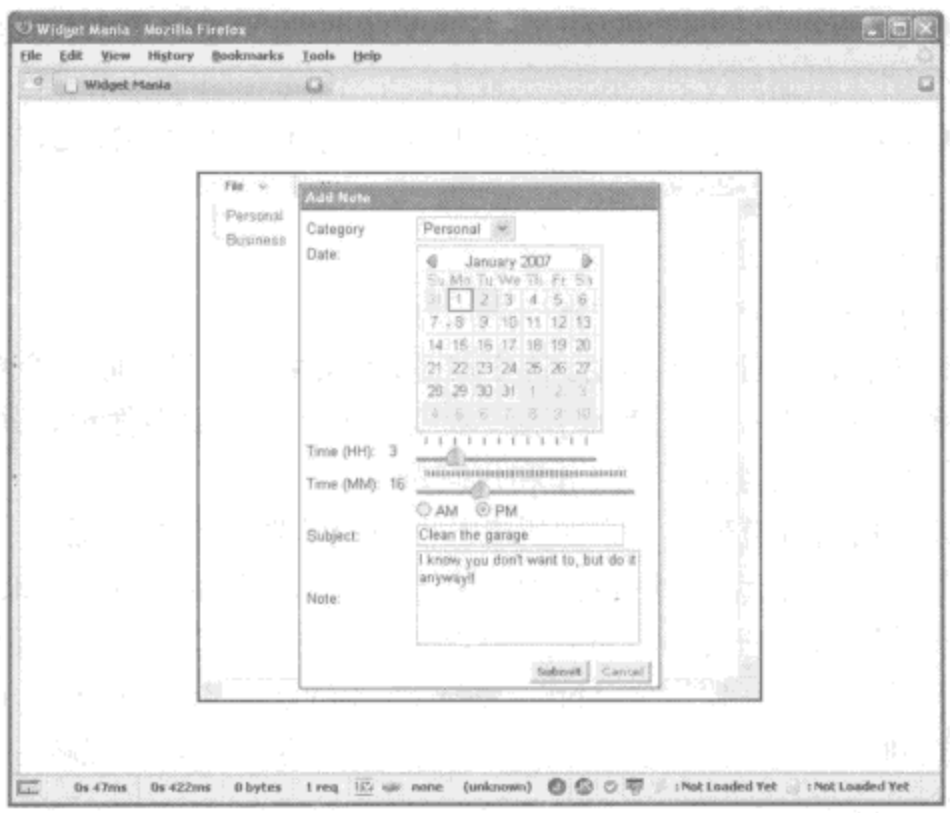


图9-2　添加便签时的JSNotes

在图9-3中，你可以看到另一个窗口小部件：叠加控件（overlay）。的确，它看起来并没有什么不一样。实际上，你甚至不能说你在看一个窗口小部件。

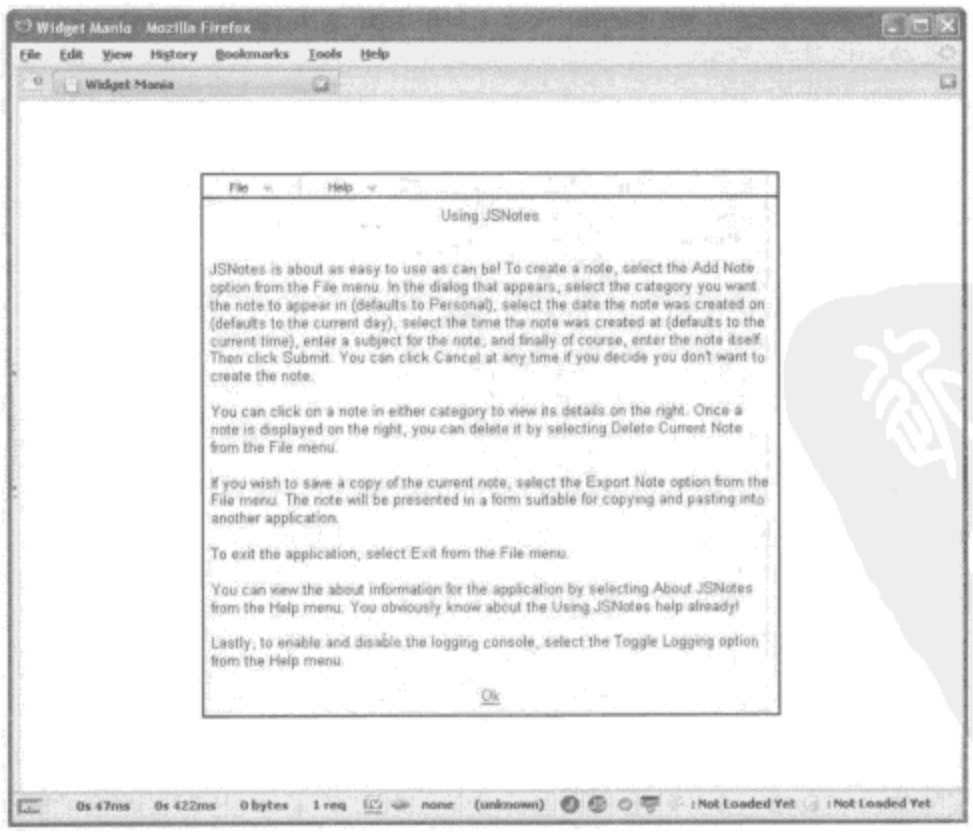


图9-3　使用JSNotes叠加控件

在我们剖析这个应用程序的过程中，你还会看到另外一个或两个屏幕截图。瞧，现在你有了一些值得期待的东西了！那么带着这个小小的欲望，让我们继续表演吧。

9.4 剖析JSNotes的解决方案

如往常一样，让我们从应用程序的文件布局开始看。如果你已经阅读了前面的章节，图9-4中所展示的结构现在对你来说应该是很熟悉的了。

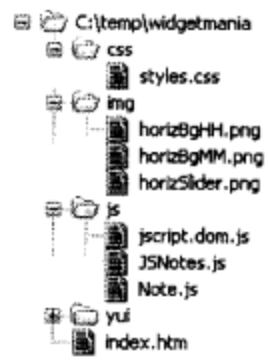


图9-4 JSNotes目录结构

在根目录中放置着一个单独的文件，index.htm，它是这个应用程序的起点。在根目录之下有下面4个子目录。

- css：和往常一样，css目录包含了styles.css，它是这个应用程序的样式表。
- img：img目录中保存着3个文件分别是horizBgHH.png、horizBgMM.png和horizSlider.png。前两个分别是Add Note对话框中，小时和分钟滑动框的背景，它们是你看到的刻度标记。horizSlider.png文件包含了拖放滑动的滑块手柄。
- js：在js目录中，你可以找到3个文件。jscript.dom.js是第3章中的那个DOM包。JSNotes.js是这个应用程序的主要JavaScript代码。Note.js中定义了Note类，它包含一个用户创建标签的数据。
- yui：yui目录是存放Yahoo 库的地方。在那里面，你将看到很多目录，每一个目录都对应于一个部分的YUI功能。它还包含了YUI所需要的所有源文件，比如样式表和图片。

这里并没有太多的文件（除非你把YUI本身也算在内，不过对于本书的目的，我们就不这么做了）。那么，现在就开始蒙头研究它吧。

9.4.1 编写index.htm

index.htm是载入应用程序的文件，也可以说，它是让表单开始运转的东西。这是一个相当大的文件，所以我们在这里就只是看一些重点片段。

一开始，在文档的<head>中，首先看到的是一批样式表的引入语句：

```
<link rel="stylesheet" type="text/css" href="css/styles.css">
<link rel="stylesheet" type="text/css" href="yui/logger/assets/logger.css">
<link rel="stylesheet" type="text/css" href="yui/fonts/fonts.css">
<link rel="stylesheet" type="text/css" href="yui/reset/reset.css">
<link rel="stylesheet" type="text/css" href="yui/menu/assets/menu.css">
<link rel="stylesheet" type="text/css"
  href="yui/container/assets/container.css">
<link rel="stylesheet" type="text/css"
  href="yui/calendar/assets/calendar.css" />
<link rel="stylesheet" type="text/css"
  href="yui/treeview/assets/tree.css" />
```

除了第一句是应用程序自身要使用的样式表之外，其他几个都是YUI窗口小部件所需要的。（如果你感兴趣，可以随意拆分它们。）

在那之后，引入了一批JavaScript源文件：

```
<script type="text/javascript" src="js/jscript.dom.js"></script>
<script type="text/javascript" src="js/JSNotes.js"></script>
<script type="text/javascript" src="js/Note.js"></script>
<script type="text/javascript" src="yui/yahoo/yahoo.js"></script>
<script type="text/javascript" src="yui/event/event.js"></script>
<script type="text/javascript" src="yui/dom/dom.js"></script>
<script type="text/javascript" src="yui/dragdrop/dragdrop.js" ></script>
<script type="text/javascript" src="yui/connection/connection.js" ></script>
<script type="text/javascript" src="yui/logger/logger.js"></script>
<script type="text/javascript" src="yui/container/container.js"></script>
<script type="text/javascript" src="yui/menu/menu.js"></script>
<script type="text/javascript" src="yui/animation/animation.js"></script>
<script type="text/javascript" src="yui/calendar/calendar.js"></script>
<script type="text/javascript" src="yui/slider/slider.js"></script>
<script type="text/javascript" src="yui/treeview/treeview.js"></script>
```

这里也是一样的，除了前3个文件是JSNote本身的一部分之外，剩下的全部都是YUI的组成部分。通常来说，每个窗口小部件都是封装在一个以这个窗口小部件命名的.js文件中的，不过也有可能需要一些支持文件，比如event.js和dom.js。

接下来是实例化JSNotes对象，它是这个应用程序背后真正起作用的代码部分。这个实例化对象是通过变量jsNotes来引用的，后面你会经常看到。

然后，<body>部分开始了，首先是一个名叫divMain的 。它是位于页面中央的一个容器。除了Add Note对话框之外，所有东西都是这个容器的子元素。

下一个出现的是构成菜单栏的HTML标记。这也是我们的第一个YUI窗口小部件——好，某种意义上来说是！你会看到，这个菜单栏会基于HTML标记中这个无序列表自动生成。基本上，我们要给YUI传递的是包含有序列表的 的ID——本例中是divMainMenu。然后YUI会解析列表，并基于它生成一个菜单。你看到的那些类，是应该应用在顶级菜单项和子菜单项上的样式类。每个菜单项都是一个超链接，但我们并不把它指向哪，这就是为什么href的值是javascript:void(0)的原因。onClick事件触发器负责每个菜单项的真正功能。

提示　你可以嵌套列表，它将会正确地呈现为嵌套的菜单项。所以，如果你希望File菜单的一个项本身就是子菜单的话，可以简单地在File的<li>下面添加一个新的列表。简洁！

下面是一个ID为divContent的 。这是相当平凡无奇的东西：两个元素（并没有使用 元素，是因为避免它们不在一行中），一个浮动居左，一个浮动居右（就像你看到的每个样式类一样）。左边的那个，是用于树状视图的，右边的用来展示当前所选择的那个便签的细节。在右边，我们有一系列<td>元素，它们的ID其实是与存储每个便签的字段相对应的。这部分是相当直接的HTML标记。

然后，我们看到的是放置日志控制台的 。YUI将负责创建这个 中的内容，所以开始的时候不用在这里写任何东西。

接下来是3个看起来很类似的 元素，每个元素都是一个YUI叠加控件的内容，通常YUI叠加控件很像 ，但它具有一些很好用的额外特性，例如居中的方法、用户行为的监视，对于IE中

<select>元素可能溢出到具有更高Z index值的元素上的常见问题，它也有内置的解决方案。[①]由于这几个 看起来很相似，我们就只抽取其中的一个来看看，好有个概念：

```
<div id="divAbout" class="cssOverlay">
  <table border="0" cellpadding="0" cellspacing="0" class="cssOverlayTable">
    <tr>
      <td align="center" valign="middle" class="cssPadded">
        JSNotes 1.0
        <br><br>
        Frank W. Zammetti
        <br><br><br><br>
        From the book "Practical JavaScript Projects"
        <br>
        Published by Apress in 2007
        <br><br><br><br>
        <a href="javascript:void(0);" onClick="jsNotes.hideAbout();">Ok</a>
      </td>
    </tr>
  </table>
</div>
```

你可以在叠加控件上面放置的东西是没有限制的（哪怕是其他YUI窗口小部件也可以），而且可以用任何你喜欢的方式来给它设置样式。这里，我们展示了一个相对简单的About，你可以看到，它只是一段非常典型的HTML标记。不过，当你用正确的方法把ID告诉给YUI之后，就得到了一个叠加控件的效果。

index.htm的最后一个要看的部分是Add Note对话框的HTML标记代码。在使用过那个应用程序并看到了一个漂亮的浮动对话框之后，你可能希望这里会有些什么特殊的东西。那么好的，令人惊讶的，这里其实就只是一些无聊的HTML标记！我们看到的只是一个简单的HTML表单，为了保持对齐而使用了表格。在这个例子中，我们实际上并不能在任何地方提交表单，所以没有action和method属性，对于onSubmit，我们也返回了false来确认不会发生提交操作。一个有意思的地方是，这里并没有定义任何提交或取消的按钮。这是因为，当创建对话框时，YUI将会自动为我们完成这些。就像覆盖控件和菜单一样，我们把容器 的ID提交给YUI，然后它就会处理剩下的事情了。

对话框是我们如何在其他窗口小部件中使用YUI窗口小部件的一个好例子。在这个对话框中，我们有（或者更准确地应该说，将要有）一个日历和两个滑动框。这里你可以看到那些小部件的实际占位符：addNoteCalendar的 、divHHSliderBG的 和divMMSliderBG的 。

可能你已经猜到了，我们并不是简单地告诉YUI，“嘿，用这个ID制作一个菜单，”或者“给我把这个 的内容转换成一个对话框。”这其中还涉及了更多的一些工作，不过也并不太多。当我们分析JSNotes.js的时候，你就会看到了。不过到达那里之前，沿途还有一两站，那么，就让我们把火车开出这站，继续驶向前方吧。

9.4.2 编写styles.css

JSNotes的样式表实在是相当简单。这里的重点是cssContentLeft、cssContentRight和cssOverlay

① 基本上，YUI是通过在叠加控件后面放置了一个iFrame来防止<select>元素显示到叠加控件上面的。

类。虽然如此，我还是要简单地介绍一下每个类是干什么的，并指出所有特殊的、有意思的属性。那么，就让我们从样式表自身开始吧，如代码清单9-1所示。

代码清单9-1　style.css文件

```
/* Style for the main DIV. */
.cssMain {
  position         : absolute;
  border           : 2px solid #000000;
  width            : 500px;
  background-color : #ffffff;
}


/* Style for the content container DIV. */
.cssContent {
  background-color : #ffffff;
  width            : 100%;
  height           : 460px;
}


/* Style for the left-hand side content. */
.cssContentLeft {
  width            : 50%;
  height           : 100%;
  float            : left;
  overflow         : scroll;
}


/* Style for the right-hand side content. */
.cssContentRight {
  width            : 50%;
  height           : 100%;
  float            : right;
  overflow         : scroll;
}


/* Style for elements that have padding (overlay contents, etc). */
.cssPadded {
  padding          : 6px;
}

/* Style for the tables that contain overlay contents. */
.cssOverlayTable {
  width            : 100%;
  height           : 100%;
}
```

```
/* Style for the background of the hour slider. */
.cssSliderBGHH {
  position          : relative;
  left              : 0px;
  top               : 0px;
  background        : url(../img/horizBgHH.png) no-repeat;
  height            : 26px;
  width             : 160px;
  zindex            : 5
}


/* Style for the background of the minutes slider. */
.cssSliderBGMM {
  position          : relative;
  left              : 0px;
  top               : 0px;
  background        : url(../img/horizBgMM.png) no-repeat;
  height            : 26px;
  width             : 196px;
  zindex            : 5
}


/* Style for the sliders' handles. */
.cssSliderHandle {
  position          : absolute;
  left              : 0px;
  top               : 8px;
  cursor            : default;
  width             : 18px;
  height            : 18px;
}


/* Style for cells of the table containing the fields of the new note form. */
.cssTDNewNote {
  padding           : 2px;
}

/* Style for overlays. */
.cssOverlay {
  border            : 2px solid #000000;
  position          : relative;
  left              : 0px;
  _left             : -2px; /* IE */
  background-color  : #f7f7ef;
  width             : 100%;
  height            : 460px;
}
```

```
/* Style for the elements where hour and minutes are displayed. */
.cssTimeSpan {
  width             : 30px;
}
```

cssMain是应用于那个包含了页面中所有内容的 的样式类。它使用绝对定位，这样可以居中。它的宽度决定了JSNotes展现的宽度，不过请注意，我们并没有设置它的高度，这是因为高度要由它的内容来决定。

说到它的内容，cssContent类应用于那个封装了菜单栏下面内容的 ，也就是树状视图和便签细节。这就是决定高度的地方。在cssMain里指定了高度和宽度，这样JSNotes就能适合于640×480的屏幕，并且也适合800×600的屏幕。

跟在cssContent之后的，是cssContentLeft和cssContentRight两个样式类。它们是内容区域的左右两半：用于树状视图的cssContentLeft和用于展示便签详细内容的cssContentRight。每一部分都指定为带有cssContent样式的那个 所占的水平区域宽度的一半以及它的全部高度。

你可以使用cssContentLeft和cssContentRight类的float属性来指定一个元素应该浮动在周围文字的左边或者右边。float属性最初的意思是允许图片浮动放置在文字段落的任何一边，不过它并不局限于那个角色。让人感兴趣的是，如何使这两个元素并排。任何float值是left或者right的元素都会被当作一个块级别（block-level）的元素来对待，也就是说，将会忽略它的display属性。比如，你还可以使用这个方法使两个文字段落在页面中并排展现。

这里对我们来说最重要的可能就是下面这个问题了：跟在一个浮动元素后面的元素会相对于第一个浮动元素进行展现。也就是说，正如你在JSNotes中看到的，两个分别带有float:left和float:right的元素，会相应地紧挨着对方排列。这两个浮动元素会被一直推向左边或者右边，直到它们到达了容器元素的边框，或者另外一个块级别元素的页边空白边距为止。（意思就是，每个都会被一直推向左边或者右边，直到碰到容器 的边界。）注意，我用了一个来避免 固有的换行，这个换行会破坏你的布局。还要注意的是， 和元素在使用float属性的时候都必须设置width属性，就像我们这里所做的一样。

继续，cssPadded是作用在覆盖控件内部那个 上的一个简单样式类，这样就可以在边界区域附近留出一些补白。

cssOverlayTable样式类是用在那个展现叠加控件的表格上的。这样做的目的是为了确保浏览器可以按照我们所期望的将表格设置为100%的高度，因为有些浏览器并不遵从<table>标签的height属性。

cssSliderBGMM和cssSliderBGHH样式类，分别定义了在Add Note对话框中，那个分钟和小时滑块的“刻度标记”。你可以看到图片本身是作为背景图片载入的，所以这个样式将被应用在那个将会出现滑块的 元素上。它们必须相对定位，以便YUI可以正确地操作它们。在这个例子中，我们希望它们可以自然地展现，因此对于left和top都使用了0px值。

两个滑块的手柄共享了cssSliderHandle样式类，并且为了可以使用YUI，它们要被绝对定位。left属性将会改变，但top不会，所以，把top适当地设置为相对于刻度标记偏移8px是很安全的。

cssTDNewNote样式类作用在那个展示Add Note对话框时使用的表格的单元格上。因为使用YUI处理HTML标记的时候有一些额外的东西，所以在<table>标签上设置cellpadding和/或cellspacing会被忽略，而通过CSS做同样的事情却可以，因此有这个类。

cssOverlay样式类是用于叠加容器的。一个有意思的地方是，这里使用了浏览器相关技巧，你可能会回想起这是我曾说过要避免的东西。不过，有时你不得不使用它来适应一些浏览器的怪癖，这里就是个例子。带有0px值的left属性是用于所有非IE浏览器的，但是对于IE，第二个left属性的设置中，以下划线开头的那个-2px的值会覆盖之前的值。这是必须的，这样一来，叠加才能合适地对齐而不会产生一个双边框的小毛病（移除那个_left属性，使用IE浏览，就会看见这个问题）。

cssTimeSpan，最后的但并非最不重要的一个样式类，是应用于那个在相应的滑块旁边展示小时和分钟数值的上的。为了任何东西都可以整齐地展示在这里，无论当前值是什么，都需要预留已知大小的一块地方，这就是这个样式类的唯一目的。

解释完index.htm中的CSS和标记后，让我们看看核心内容：JavaScript！

9.4.3 编写Note.js

Note.js文件包含了Note类，它只不过就是个值对象（Value Object，VO），或者，如果你更熟悉C的话，那就是一个结构，它包含了用户输入的用于描述便签的数据。它只含有一些私有的字段以及每个字段的获取方法和设置方法。在我们看代码之前，先来看看图9-5中这个类的UML图吧。

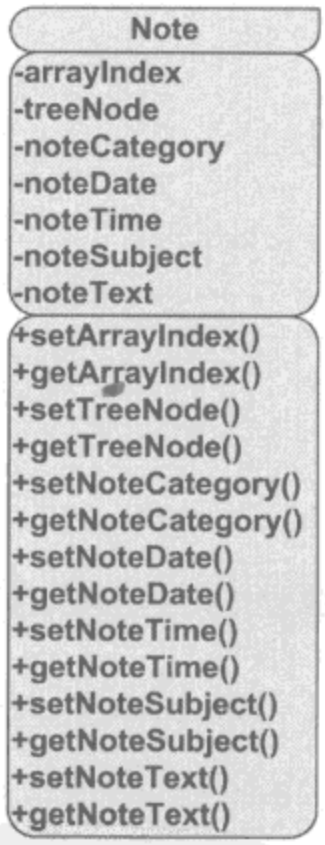


图9-5 Notes类的UML图

我选择并不在这里罗列这个类，基本上是因为它很无聊！它几乎就是一系列的作用于所包含字段的读取器和设置器。实际上，就是这样。除此之外，通常来说，程序中的注释本身就可以告诉你所有你需要知道的东西了，所以如果你自己快速地看了一眼，可能立刻就有了那个完整的图了。

toString()方法覆盖了那个我们从Object基类中得到的默认toString()，它以一种更有意义的方式输出信息，可以方便调试。

注意一下arrayIndex和treeNode字段以及它们的注释。这两个字段是在用户点击树状视图中的一个便签时使用的，不过在那之前，它们的值都是毫无意义的。发生点击操作时，它们就被填上值，并且允许其他的行为请求它们，例如删除操作。

除了这些，这个类绝对是一段简单而且直接了当的代码。现在，如果你想要看些更有趣的东西，马上就来了。

9.4.4 编写JSNote.js

JSNotes.js中包含了JSNotes类，它是这个应用程序的核心和灵魂。所有的功能都在此生存，它也是一段相当长的代码。不过正如你将看到的，其中大部分都非常简单，而且大多都可以认为是很无聊的，实际的功能并不需要你想象的那么多东西。

不过在开始之前，让我们先通过图9-6的UML图站在更高的位置上看看全景。

代码清单9-2是完整的代码，它相当的长。里面有600多行代码，虽然大部分都是注释和空格，但是骨头上还是有很多肉的！

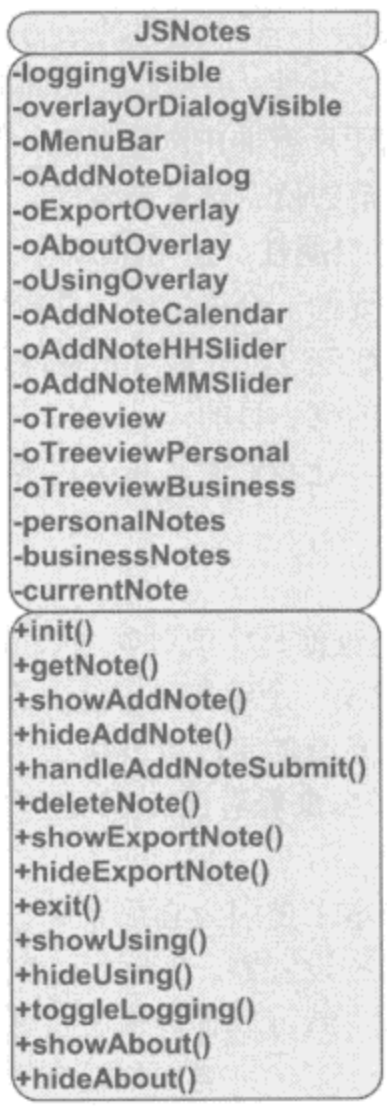


图9-6　JSNotes类的UML图

代码清单9-2　JSNotes类

```
/**
 * The JSNotes class is the main class constituting the application.
 */
function JSNotes() {


  /**
   * Flag: Is the logging div currently visible?
   */
  var loggingVisible = false;

/**
 * Flag: Is an overlay or dialog currently visible?
 */
var overlayOrDialogVisible = false;


/**
 * Reference to the menubar object.
```

```
 */
var oMenuBar = null;


/**
 * Reference to the add notes dialog object.
 */
var oAddNoteDialog = null;


/**
 * Reference to the Export overlay object.
 */
var oExportOverlay = null;


/**
 * Reference to the About overlay object.
 */
var oAboutOverlay = null;


/**
 * Reference to the Using overlay object.
 */
var oUsingOverlay = null;


/**
 * Reference to the date calendar for adding a new note.
 */
var oAddNoteCalendar = null;


/**
 * Reference to the hour slider for adding a new note.
 */
var oAddNoteHHSlider = null;


/**
 * Reference to the minutes slider for adding a new note.
 */
var oAddNoteMMSlider = null;


/**
 * Reference to the treeview for listing categories/notes.
 */
var oTreeview = null;
```

```
/**
 * Reference to the treeview Personal Notes category node.
 */
var oTreeviewPersonal = null;


/**
 * Reference to the treeview Business Notes category node.
 */
var oTreeviewBusiness = null;


/**
 * The collection of Personal notes.
 */
var personalNotes = new Array();


/**
 * The collection of Business notes.
 */
var businessNotes = new Array();


/**
 * This is a reference to the Note object currently being viewed.
 */
var currentNote = null;


/**
 * Call on page load to initialize the application.
 */
this.init = function() {
// Start logging, show logging div if flag set to do so initially.
new YAHOO.widget.LogReader(YAHOO.util.Dom.get("divLog"));
YAHOO.log("init()");
if (loggingVisible) {
  YAHOO.util.Dom.get("divLog").style.display = "block";
}

// Start by centering the main DIV.
jscript.dom.layerCenterH(YAHOO.util.Dom.get("divMain"));
jscript.dom.layerCenterV(YAHOO.util.Dom.get("divMain"));

// Create menubar.
oMenuBar = new YAHOO.widget.MenuBar("divMainMenu");
oMenuBar.render();

// Create About overlay.
```

```
oAboutOverlay = new YAHOO.widget.Overlay("aboutOverlay",
  {
    context : [ "divContent", "tl", "tl" ],
    width : "500px", height : "456px", visible : false
  }
);
oAboutOverlay.setBody(YAHOO.util.Dom.get("divAbout"));
oAboutOverlay.render(document.body);

// Create Export overlay.
oExportOverlay = new YAHOO.widget.Overlay("exportOverlay",
  {
    context : [ "divContent", "tl", "tl" ],
    width : "500px", height : "456px", visible : false
  }
);
oExportOverlay.setBody(YAHOO.util.Dom.get("divExport"));
oExportOverlay.render(document.body);

// Create Using overlay.
oUsingOverlay = new YAHOO.widget.Overlay("usingOverlay",
  {
    context : [ "divContent", "tl", "tl" ],
    width : "500px", height : "456px", visible : false
  }
);
oUsingOverlay.setBody(YAHOO.util.Dom.get("divUsing"));
oUsingOverlay.render(document.body);
// Create Add Note dialog.
oAddNoteDialog = new YAHOO.widget.Dialog("divAddNote",
  {
    close : false,
    width : "320px",
    height : "460px",
    visible : false,
    constraintoviewport : true,
    buttons : [
      {
        text : "Submit",
        handler : jsNotes.handleAddNoteSubmit,
        isDefault : true
      },
      { text : "Cancel", handler : jsNotes.hideAddNote }
    ]
  }
);
oAddNoteDialog.render(document.body);

// Create Add Note calendar.
oAddNoteCalendar = new YAHOO.widget.Calendar("cal1", "addNoteCalendar");
```

```
oAddNoteCalendar.render();

// Create Add Note hour slider.
var bgHH = "divHHSliderBG";
var thumbHH = "divHHSliderThumb";
oAddNoteHHSlider = YAHOO.widget.Slider.getHorizSlider(
  bgHH, thumbHH, 0, 150, 13);
oAddNoteHHSlider.subscribe("change",
  function() {
    YAHOO.util.Dom.get("divHHValue").innerHTML =
      Math.round(oAddNoteHHSlider.getValue() / 13) + 1;
  }
);

// Create Add Note minutes slider.
var bgMM = "divMMSliderBG";
var thumbMM = "divMMSliderThumb";
oAddNoteMMSlider = YAHOO.widget.Slider.getHorizSlider(
  bgMM, thumbMM, 0, 178, 3);
oAddNoteMMSlider.subscribe("change",
  function() {
    var minute = Math.round(oAddNoteMMSlider.getValue() / 3);
    var s = "";
      if (minute < 10) {
        s += "0";
      }
    s += minute;
    YAHOO.util.Dom.get("divMMValue").innerHTML = s;
  }
);

// Create treeview for category/note listing.
oTreeview = new YAHOO.widget.TreeView("divTreeview");
var oRoot = oTreeview.getRoot();
oTreeviewPersonal = new YAHOO.widget.TextNode("Personal",
  oRoot, false);
oTreeviewBusiness = new YAHOO.widget.TextNode("Business",
  oRoot, false);
oTreeview.subscribe("labelClick",
  function(node) {
    var noteSubject = node.data.subject;
    // Only do something when a note is clicked, not a category (only a
    // note would have a subject attribute).
    if (noteSubject) {
      var noteCategory = node.parent.data;
      currentNote = jsNotes.getNote(noteCategory, noteSubject);
      var noteDate = currentNote.getNoteDate();
      YAHOO.util.Dom.get("currentNoteDate").innerHTML =
        noteDate.getMonth() + "/" +
        noteDate.getDate() + "/" +
```

```
            noteDate.getFullYear();
          YAHOO.util.Dom.get("currentNoteTime").innerHTML =
            currentNote.getNoteTime();
          YAHOO.util.Dom.get("currentNoteSubject").innerHTML =
            currentNote.getNoteSubject();
          YAHOO.util.Dom.get("currentNoteText").innerHTML =
            currentNote.getNoteText();
        }
      }
    );
    oTreeview.draw();

    YAHOO.log("init() done");

} // End init().
/**
 * Returns a Note object based on requested category and subject.
 *
 * @param inCategory The category the note belongs to.
 * @param inSubject  The subject of the note to retrieve.
 */
this.getNote = function(inCategory, inSubject) {

  var note = null;

// Determine which array to search based on current category.
var arrayToSearch = null;
if (inCategory == "Personal") {
  arrayToSearch = personalNotes;
} else {
  arrayToSearch = businessNotes;
}

// Search the array and find the match, if any, and return it.
for (var i = 0; i < arrayToSearch.length; i++) {
  var n = arrayToSearch[i];
  if (n.getNoteSubject() == inSubject) {
    note = n;
    note.setArrayIndex(i);
    break;
  }
}

// Now find the note in the treeview for the note.
note.setTreeNode(oTreeview.getNodeByProperty("subject",
  note.getNoteSubject()));

// Not found.
return note;
```

```
} // End getNote();


/**
 * Show the dialog for adding a note.
 */
this.showAddNote = function() {

  YAHOO.log("showAddNote()");
  if (overlayOrDialogVisible) { return; }
  oMenuBar.clearActiveItem();

  overlayOrDialogVisible = true;

  // Reset all form fields.
  var now = new Date();
  var hours = now.getHours();
  var minutes = now.getMinutes();
  YAHOO.util.Dom.get("frmNewNote").reset();
  oAddNoteCalendar.clear();
  oAddNoteCalendar.select(now);
  oAddNoteCalendar.render();
  oAddNoteHHSlider.setValue((hours * 13) - 13, true, true);
  oAddNoteMMSlider.setValue(minutes * 3, true, true);
  YAHOO.util.Dom.get("divHHValue").innerHTML = hours;
  if (minutes < 10) {
    minutes = "0" + minutes;
  }
  YAHOO.util.Dom.get("divMMValue").innerHTML = minutes;
  YAHOO.util.Dom.get("newNotePM").checked = true;

  // Show the dialog amd center it.
  oAddNoteDialog.center();
  oAddNoteDialog.show();

  YAHOO.log("showAddNote() done");

} // End showAddNote().


/**
 * Hide the dialog for adding a note.
 */
this.hideAddNote = function() {

  YAHOO.log("hideAddNote()");

  oAddNoteDialog.hide();
  overlayOrDialogVisible = false;
```

```
    YAHOO.log("hideAddNote() done");

} // End hideAddNote().
/**
 * Handle submit of the add new note form.
 */
this.handleAddNoteSubmit = function() {

    YAHOO.log("handleAddNoteSubmit()");

    // Get entered values.
    var noteCategory = YAHOO.util.Dom.get("newNoteCategorySelect").value;
    var noteDate = oAddNoteCalendar.getSelectedDates()[0];
    var noteHour = YAHOO.util.Dom.get("divHHValue").innerHTML;
    var noteMinute = YAHOO.util.Dom.get("divMMValue").innerHTML;
    var noteMeridian = null;
    if (YAHOO.util.Dom.get("newNoteAM").checked) {
      noteMeridian = "am";
    } else {
      noteMeridian = "pm";
    }
    var noteSubject = YAHOO.util.Dom.get("newNoteSubject").value;
    var noteText = YAHOO.util.Dom.get("newNoteText").value;

    // Now some simple validations.
    if (noteSubject == "") {
      alert("Please enter a subject for this note");
      YAHOO.util.Dom.get("newNoteSubject").focus();
      return false;
    }
    if (noteText == "") {
      alert("Please enter some text for this note");
      YAHOO.util.Dom.get("newNoteText").focus();
      return false;
    }

    // Instantiate a Note object and populate it.
    var note = new Note();
    note.setNoteCategory(noteCategory);
    note.setNoteDate(noteDate);
    note.setNoteTime(noteHour + ":" + noteMinute + noteMeridian);
    note.setNoteSubject(noteSubject);
    note.setNoteText(noteText);

    // Add the note to the appropriate treeview category and storage array.
    if (noteCategory == "Personal") {
      personalNotes.push(note);
      new YAHOO.widget.TextNode({label:noteSubject,subject:noteSubject},
        oTreeviewPersonal, false);
```

```
    } else {
      businessNotes.push(note);
      new YAHOO.widget.TextNode({label:noteSubject,subject:noteSubject},
        oTreeviewBusiness, false);
    }

    // Redraw treeview so it'll show up.
    oTreeview.draw();

    // Hide dialog and we're done!
    jsNotes.hideAddNote();
    YAHOO.log("handleAddNoteSubmit() done");
    return true;

  } // End handleAddNoteSubmit().


  /**
   * Delete the note currently being viewed.
   */
  this.deleteNote = function() {

    YAHOO.log("deleteNote()");

    if (overlayOrDialogVisible) { return; }
    oMenuBar.clearActiveItem();

    if (currentNote &&
      confirm("Are you sure you want to delete the current note?")) {
      // Delete from storage array.
      if (currentNote.getNoteCategory() == "Personal") {
        personalNotes.splice(currentNote.getArrayIndex(), 1);
      } else {
        businessNotes.splice(currentNote.getArrayIndex(), 1);
      }
      // Delete from treeview and redraw.
      oTreeview.removeNode(currentNote.getTreeNode());
      oTreeview.draw();
      // Clear display fields.
      YAHOO.util.Dom.get("currentNoteDate").innerHTML = "";
      YAHOO.util.Dom.get("currentNoteTime").innerHTML = "";
      YAHOO.util.Dom.get("currentNoteSubject").innerHTML = "";
      YAHOO.util.Dom.get("currentNoteText").innerHTML = "";
      // Finally, no more current note.
      currentNote = null;
    }
    YAHOO.log("deleteNote() done");

  } // End deleteNote().
```

```
/**
 * Show the overlay for exporting the current note.
 */
this.showExportNote = function() {

  YAHOO.log("showExportNote()");

  if (overlayOrDialogVisible) { return; }
  oMenuBar.clearActiveItem();

  if (currentNote) {
    var s = "";
    var noteDate = currentNote.getNoteDate();
    s += "Category: " + currentNote.getNoteCategory() + "\n";
    s += "Date: " + noteDate.getMonth() + "/" +
      noteDate.getDate() + "/" +
      noteDate.getFullYear() + "\n";
    s += "Time: " + currentNote.getNoteTime() + "\n";
    s += "Subject: " + currentNote.getNoteSubject() + "\n";
    s += "Note: " + currentNote.getNoteText();
    YAHOO.util.Dom.get("taExport").value = s;
    YAHOO.util.Dom.get("taExport").select();
    overlayOrDialogVisible = true;
    oExportOverlay.show();
  }
  YAHOO.log("showExportNote() done");

} // End showExportNote().


/**
 * Hide the Export Note overlay.
 */
this.hideExportNote = function() {

  YAHOO.log("hideExportNote()");
  oMenuBar.clearActiveItem();

  oExportOverlay.hide();
  overlayOrDialogVisible = false;
  YAHOO.log("hideExportNote() done");

} // End hideExportNote().


/**
 * Exit the application
 */
```

```
this.exit = function() {

  YAHOO.log("exit()");
  if (overlayOrDialogVisible) { return; }

  if (confirm(
    "All notes will be lost!  Are you sure you want to exit?")) {
    window.close();
  }

} // End exit().


/**
 * Toggle the logging div on and off.
 */
this.toggleLogging = function() {

  YAHOO.log("toggleLogging()");
  if (overlayOrDialogVisible) { return; }

  oMenuBar.clearActiveItem();
  if (loggingVisible) {
    YAHOO.util.Dom.get("divLog").style.display = "none";
    loggingVisible = false;
  } else {
    YAHOO.util.Dom.get("divLog").style.display = "block";
    loggingVisible = true;
  }

  YAHOO.log("toggleLogging() done");

} // End toggleLogging().


/**
 * Show the Using (help) overlay.
 */
this.showUsing = function() {
  YAHOO.log("showUsing()");
  if (overlayOrDialogVisible) { return; }
  oMenuBar.clearActiveItem();

  overlayOrDialogVisible = true;
  oUsingOverlay.show();

  YAHOO.log("showUsing() done");

} // End showUsing().
```

```
/**
 * Hide the Using (help) overlay.
 */
this.hideUsing = function() {

  YAHOO.log("showUsing()");
  oMenuBar.clearActiveItem();

  oUsingOverlay.hide();
  overlayOrDialogVisible = false;

  YAHOO.log("showUsing() done");

} // End hideUsing().


/**
 * Show the About overlay.
 */
this.showAbout = function() {

  YAHOO.log("showAbout()");
  if (overlayOrDialogVisible) { return; }

  oMenuBar.clearActiveItem();
  overlayOrDialogVisible = true;
  oAboutOverlay.show();

  YAHOO.log("showAbout() done");

} // End showAbout().
  /**
   * Hide the About overlay.
   */
  this.hideAbout = function() {

    YAHOO.log("hideAbout()");

    oAboutOverlay.hide();
    overlayOrDialogVisible = false;

    YAHOO.log("hideAbout() done");

  } // End hideAbout().


} // End JSNotes class.
```

正如你所见，我们有许多数据元素和一堆方法。首先来看看数据字段。

- loggingVisible：一个布尔值。它决定了日志控制台当前是否可见。开始的时候是不可见的，

除非用户通过Help菜单下面的Toggle Logging选项打开它，它才变得可见。

- overlayOrDialogVisible：当任何叠加控件或者Add Note对话框可见的时候，菜单都不应该有任何操作。这个标志位决定了它在何时是什么都不做的（当这个字段是true的时候）。
- oMenuBar：一个指向由YUI创建的菜单对象的引用。稍后，当点击一个菜单项的时候，我们需要清除那个已选择的项，这样做的话，就需要一个指向菜单栏的引用。把它预先缓存起来，要比每次碰到时都花费开销重新获取更好。
- oAddNoteDialog：一个指向由YUI创建的Add Note对话框的引用。我们也将在很多地方需要它，所以保留这个引用是个好主意。实际上，后面的几个字段也都是为了相同的目的，那就是避免查找的开销，所以后面我就会略过重复这个原因了。
- oExportOverlay：一个指向由YUI创建的Export Note叠加控件的引用。从现在起，如果你不介意的话，我准备不再“由YUI创建的……”！它也适用于后面几个项。
- oAbourOverlay：一个指向About JSNotes叠加控件的引用。（你是否已经开始感觉到这里的一个模式了？）
- oUsingOverlay：是的，你已经猜到了——它是一个指向Using JSNotes叠加控件的引用。
- oAddNoteCalendar：啊，稍微有一些不同，虽然还是一个引用，不过这次是指向Add Note对话框中的日历的。
- oAddNoteMMSlider：引用，Minutes滑块，检查。
- oTreeView：啊，树状视图，啊，树状视图，你的艺术，源自何方？对不起，有点忍不住。没错，它是另外一个对象引用，这次是指向左边展现的那个罗列便签的树状视图的。
- oTreeviewPersonal：一个指向树状视图中个人便签节点对象的引用。
- oTreeviewBusiness：我想让你来猜猜这是干什么的！现在我们说完了那些对象引用了。
- personalNotes：一个包含了所有分类为个人便签对象的数组。
- businessNotes：一个包含了所有分类为业务对象的数组。
- currentNote：最后一个但并非最不重要的，这是一个指向用户当前浏览的那个Note对象的引用。

好，关掉我们的聪明人模式！[①]既然我们已经看过了所有的字段，就来看看它们在代码中是如何使用的吧，从init()方法开始。

1. init()方法

init()方法毫无置疑地是JSNotes中最大的（大约占了总代码量的1/4），也是最复杂的方法。即便如此，它也并不那么可怕！如你前面看到的，它在index.htm页面载入时调用。其主要任务是使用YUI组件来构造UI。

(1) 创建日志控制台。

首先，init()使用下面的代码创建了日志控制台：

```
// Start logging, show logging div if flag set to do so initially.
new YAHOO.widget.LogReader(YAHOO.util.Dom.get("divLog"));
YAHOO.log("init()");
if (loggingVisible) {
```

① 意即后面的都很简单了。——译者注

```
    YAHOO.util.Dom.get("divLog").style.display = "block";
}
```

我们需要做的就是实例化YAHOO.widget.LogReader，把将要保存它的那个DOM对象的引用传给它。通过使用YAHOO.util.Dom.get()来获得那个引用，它基本上是document.getElementById()的一个封装。然后，使用YAHOO.log("init()");这行代码迅速地向日志中写一个信息，就是为了说明一下我们现在在哪里。

回想一下在index.htm中所看到的记录日志的 ，divLog开始是隐藏的。那就是要使用后面的那个if判断的原因。如果你把loggingVisible的默认值改为true，那么这里就会显示那个日志记录控制台。

当日志记录控制台可见时，显示的结果就如我们在图9-7中看到的那样。

然后，需要把那个承载了JSNotes显示部分的主 置中。为实现这个目的，我们用到了在第3章的DOM包中描述过的两个函数：

```
jscript.dom.layerCenterH(YAHOO.util.Dom.get("divMain"));
jscript.dom.layerCenterV(YAHOO.util.Dom.get("divMain"));
```

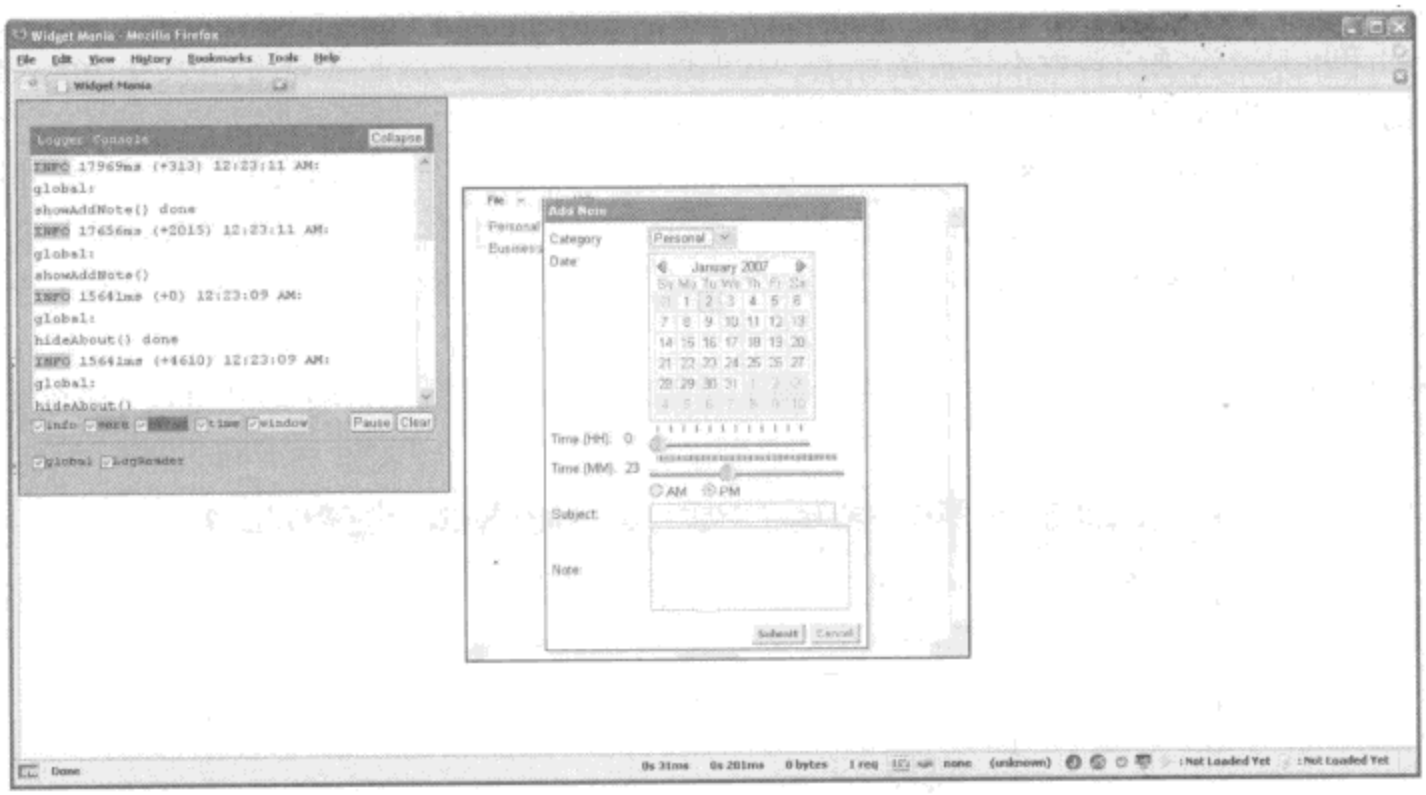


图9-7 日志记录控制台

(2) 创建菜单栏。

下面的任务是创建菜单栏，这绝对是简单到不能再简单的了：

```
oMenuBar = new YAHOO.widget.MenuBar("divMainMenu");
oMenuBar.render();
```

你已经了解了divMainMenu中包含的HTML标记了。YUI使用它们来生成在屏幕中看到的那个菜单。不过，第一行只是简单地在内存中创建了代表菜单的DOM片段。要把它实际放到屏幕上，还需要调用它的render()方法，这个你可以在第二行看到。当这两行都完成时，菜单就可见并可用了。

(3) 创建叠加控件。

然后出现的是创建3个覆盖控件：Export Note、Using JSNotes和About JSNotes。它们其实是一样的，所以我们就只挑一个来说说，Export Note叠加：

```
oExportOverlay = new YAHOO.widget.Overlay("exportOverlay",
  {
    context : [ "divContent", "tl", "tl" ],
    width : "500px", height : "456px", visible : false
  }
);
oExportOverlay.setBody(YAHOO.util.Dom.get("divExport"));
oExportOverlay.render(document.body);
```

我们首先要把YAHOO.widget.overlay这个类实例化。传给构造函数的第一个参数是叠加的名字，在这个例子中就是exportOverlay。第二个参数是包含了一系列配置选项的对象。你看到的第一个选项是context，它用于把这个叠加与其他页面元素对齐。在这个例子里，我们把它和divContent的那个 对齐，这样它就会出现在菜单下方了。我们还要指定叠加的左上角与那个 的左上角对齐。你也可以用其他的方法来对齐它，比如：通过["divContent", "tr", "bl"]传递context，让叠加的右上角和 的左下角对齐。

width和height选项应该就不用再解释了。它们就是字面上的叠加的宽和高。visible选项也是不言而喻的。当值为false时，表示初始时是不显示叠加；当为true时，就显示它。

下一步，是填充叠加的内容。我们通过调用setBody()并交给它一个指向某些已存在元素的引用来完成。在这个例子中，就是index.htm里你看到的那个divExport 。最后，就像对那个菜单所做的一样，我们要呈现叠加。把DOM中要内嵌叠加的那个对象传给render()方法，在这个例子中，就是文档的body对象。

图9-8展示了Export Note叠加。另外两个叠加看起来跟这个很类似（图9-3是Using JSNotes叠加的截屏图）。

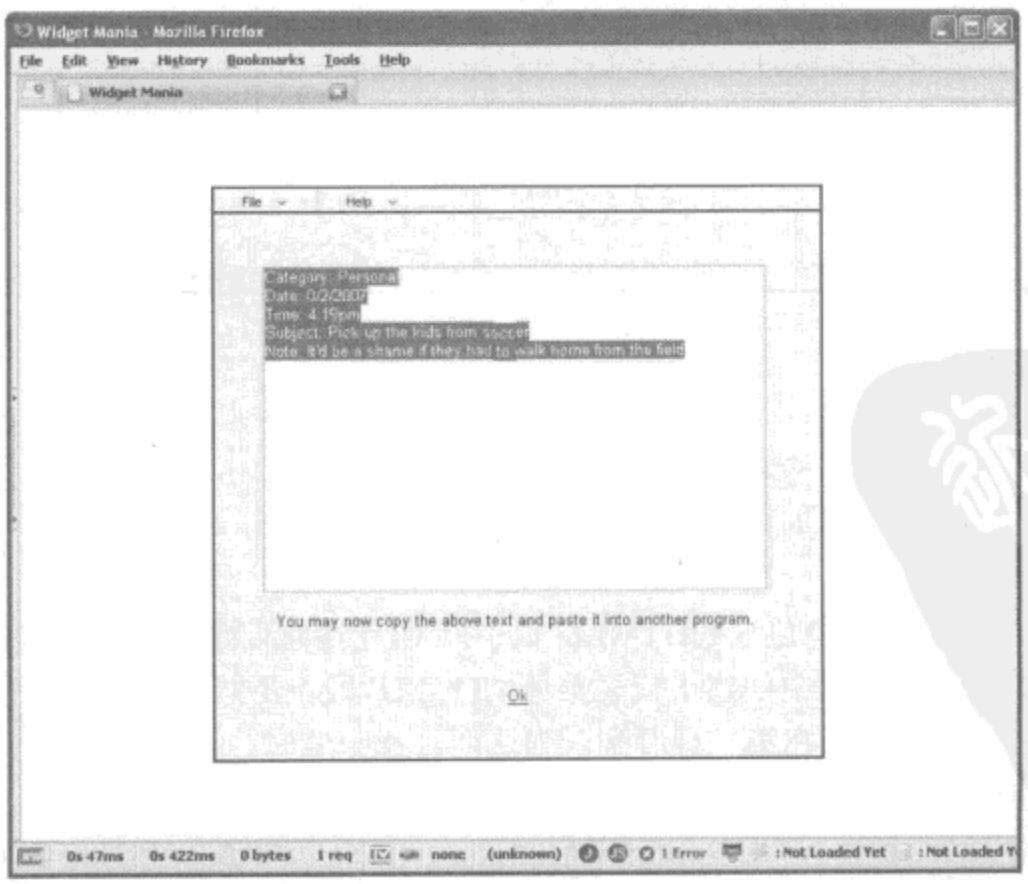


图9-8　从 JSNotes中导出一个便签

(4) 创建Add Note对话框。

下一步是创建Add Note对话框。这与叠加不同，但是大部分还是一样的。通常对话框的意思是承载一个需要填写的表单，并提交给服务器。不过在我们的例子中，并不会提交它，不过说得有点太快了。首先还是让我们来看看是如何创建对话框的：

```
oAddNoteDialog = new YAHOO.widget.Dialog("divAddNote",
  {
    close : false,
    width : "320px",
    height : "460px",
    visible : false,
    constraintoviewport : true,
    buttons : [
      {
        text : "Submit",
        handler : jsNotes.handleAddNoteSubmit,
        isDefault : true
      },
      { text : "Cancel", handler : jsNotes.hideAddNote }
    ]
  }
);
oAddNoteDialog.render(document.body);
```

这次，我们需要的是YAHOO.widget.Dialog，它的第一个参数是为含有那个对话框内容的元素——也就是divAddNote 。第二个参数，像叠加那样，是一个包含了一系列选项的对象。让我们一个个地来看看它们。

- close：这个选项的意思是，是否需要一个关闭按钮，就像在典型的窗口里[①]那个X按钮。这里，我们并不需要它，所以设置为false。
- width、height和visible：这些选项就是你所认为的那样，与叠加控件是相同的。
- constraintoviewport：它表示对话框是否可以被拖入页面。这里并不需要，因为我们想让它停在边上，所以设置了true。
- button：这个选项实际上是一个数组，每个元素都描述了一个按钮。也就是一个包含了各自按钮所需选项的对象。
- text：这个选项是本例中要在Submit和Cancel按钮中显示的内容。
- handler：这个选项定义了当点击一个按钮的时候，应该调用哪个JavaScript函数。这里，它们分别是由参数jsNotes引用的那个JSNotes类的handlerAddNoteSubmit()和hideAddNote()方法。
- isDefault：Submit按钮上的这个选项定义了按下回车键时是否要激活这个按钮，也是本例的使用方式。

正如叠加控件一样，我们需要告诉对话框，它要作为哪个元素的子元素来呈现，并且同样地，答案就是文档的body部分。

① 窗口右上角的。——译者注

下面，我们来创建在Add Note对话框中的窗口小控件。注意，像这样内嵌窗口小部件是没问题的——YUI可以做到！首先，我们需要创建那个用来选择便签日期的日历，通过如下代码来完成：

```
oAddNoteCalendar = new YAHOO.widget.Calendar("cal1", "addNoteCalendar");
oAddNoteCalendar.render();
```

第一个参数是日历将会使用的名字，第二个是当使用第二行代码来呈现那个日历的时候，它的父元素的名称。

在日历之后，我们需要呈现两个滑块，从小时滑块开始：

```
var bgHH = "divHHSliderBG";
var thumbHH = "divHHSliderThumb";
oAddNoteHHSlider = YAHOO.widget.Slider.getHorizSlider(
  bgHH, thumbHH, 0, 150, 13);
oAddNoteHHSlider.subscribe("change",
  function() {
    YAHOO.util.Dom.get("divHHValue").innerHTML =
      Math.round(oAddNoteHHSlider.getValue() / 13) + 1;
  }
);
YAHOO.util.Event.on(bgHH, "keydown",
  function(e) {
    console.log("keydown (HH)");
  }
);
YAHOO.util.Event.on(bgHH, "keypress", YAHOO.util.Event.preventDefault);
```

好的，这里要说的东西确实比较多！我们先建立一些所需的变量。从bgHH开始，这个变量是用来给包含背景图片那个 命名的。下一个是含有滑块手柄的那个元素的名称——divHHSliderThumb，它保存在thumbHH中。

这些完成之后，我们再实例化一个YAHOO.widget.Slider对象，不过使用的是getHorizSlider()方法，它包含3个参数，分别是背景元素和手柄元素的名称，以及3个数字。第一个数字是手柄可以向左移动多少像素，第二个数字是向右可以移动多少像素（基本上定义了手柄可以移动的最大最小像素值），第三个数字是每个刻度标签之间要移动的像素数。

创建滑块需要一些技巧，因为你需要创建一个适合于想要的大小范围的背景图片。例如，对于小时滑块，我们有12个刻度，每一个代表一小时。刻度是1个像素的宽度，间隔为12像素，这样每次移动一个刻度就相当于13个像素（getHorizSlider()方法的第3个数字）。12乘以13得到156，不过我们实际上在getHorizSlider()的第二个数字上使用了150。这是因为刻度标签一般都是从零刻度开始的，也就是说，代表1小时的那个第一个刻度是在0像素的位置上的（好吧，实际上刻度标签距离左边界还有几个像素，但是滑块手柄必须位于零像素的位置上）。因此，我们不希望手柄能操作156像素的位置。所以，我们使用150这个数值，这样实际上就把它约束在最后一个刻度标签的位置。

要理解这些的最好方法，可能就是把背景图片和手柄图片拿出来，放在你熟悉的图片处理工具中（比如Photoshop、Paint Shop Pro、GIMP或者任何你喜欢的工具）。放大背景图片的画布，这样你就有操作的空间了。然后把手柄图片复制进去，作成一个浮动的层，让你可以自由地移动它。把它和第一个刻度标签对齐，并记下x坐标，这时x坐标应该是零。然后把它移动到下一个刻度上，总共移动了13

像素。继续移动直到最后一个刻度标签，你会看到那个位置比150要小。如果你还想再移动13个像素，也就是156，那就会超出背景的刻度范围了。这就证明了你需要用一个比那小的值来限制它。

只要你不再为这个头疼发愁了，（我也是！）我们就可以来看看要如何把滑块和事件钩起来。首先，在移动滑块改变（拖动）时，我们要更新小时的值。我们通过调用滑块上的subscribe()方法来实现。第一个参数是本例中我们要订阅的那个change事件，第二个参数是当那个事件发生时我们要执行的函数，这里使用了内联（in-line）函数。当值发生变化时，我们首先需要获取滑块的新值。这个值基本上就是像素位置，它本身没有什么实际用处。所以我们要先把它除以13，这样就得到了0到11之间的一个数。那么再加上1，就是相对应于那个像素位置的小时的值了。我们取一个指向那个显示小时数字的 层的指针，并且把它的innerHTML属性值设置为刚才计算得到的数字。这样做就实现了小时的数值随着我们拖放滑块而变化的效果了——非常完美！

创建分钟滑块的方法与此类似，只是会有更多的刻度标签，最大值也是不同的——我们这里是178。刻度之间挨得更加紧凑，这样每次递增的数值就变成3了。当然，在change事件的处理上，也还有一些很小的区别。

```
oAddNoteMMSlider.subscribe("change",
  function() {
    var minute = Math.round(oAddNoteMMSlider.getValue() / 3);
    var s = "";
    if (minute < 10) {
      s += "0";
    }
    s += minute;
    YAHOO.util.Dom.get("divMMValue").innerHTML = s;
  }
);
```

这与小时滑块非常相似，不过这里计算时使用的除数是3，因为它是递增的值。而且，最后也没必要再加1，因为这里的范围是从零开始的（对于分钟来说是0到59）。最后，当出现小于10的值时，也就是说，只有一位数字的时候，应该显示带有前导零的数字（leading zero[①]，所以我们会看到一些处理这个情况的逻辑。除了这几个不同的地方，分钟滑块在概念上基本与小时滑块是一样的。

(5) 创建树状视图。

init()方法中的最后一部分是用来创建树状视图的，它将会罗列用户所创建的便签。实现这个功能的代码如下：

```
oTreeview = new YAHOO.widget.TreeView("divTreeview");
var oRoot = oTreeview.getRoot();
oTreeviewPersonal = new YAHOO.widget.TextNode("Personal",
  oRoot, false);
oTreeviewBusiness = new YAHOO.widget.TextNode("Business",
  oRoot, false);
oTreeview.subscribe("labelClick",
  function(node) {
    var noteSubject = node.data.subject;
```

① 例如01，02。——译者注

```
    // Only do something when a note is clicked, not a category (only a
    // note would have a subject attribute).
    if (noteSubject) {
      var noteCategory = node.parent.data;
      currentNote = jsNotes.getNote(noteCategory, noteSubject);
      var noteDate = currentNote.getNoteDate();
      YAHOO.util.Dom.get("currentNoteDate").innerHTML =
        noteDate.getMonth() + "/" +
        noteDate.getDate() + "/" +
        noteDate.getFullYear();
      YAHOO.util.Dom.get("currentNoteTime").innerHTML =
        currentNote.getNoteTime();
      YAHOO.util.Dom.get("currentNoteSubject").innerHTML =
        currentNote.getNoteSubject();
      YAHOO.util.Dom.get("currentNoteText").innerHTML =
        currentNote.getNoteText();
    }
  }
);
oTreeview.draw();
```

好的，确实有很多东西，那么让我们把它拆分来看看吧，如何？

首先，是一行典型的窗口小部件实例化代码。我们告诉它要承载树状视图的那个 。然后，我们得到指向那个树结构根节点的引用，它是在实例化窗口小部件时自动生成的。有了它之后，我们再向上面添加两个节点：一个是个人便签，一个是商用便签。然后，是另外两行事件订阅代码，当点击任何一个标签，也就是便签的时候，都要通知我们。

YUI会负责展开和收缩所有有子节点的便签，也就是我们刚刚创建的那两个。不过我们还要了解一下在这以外还发生了什么，也就是展示所点击的那个便签的细节。

所以，当我们订阅那个事件触发器时，要传给它一个指向该事件回调函数的引用——在本例中，就是一个内联的匿名函数。这个回调函数首先要做的是获取当前便签的主题。在学习如何向树状视图中添加节点的时候，可能会更清楚地了解这一点。现在，我们只要知道在给一个树添加节点的时候，除了可以有文本标签之外，你可以附加任意的信息就可以了。在这儿，我们给它添加一个subject属性。

如果你玩过这个应用，可能已经发现了便签的标签其实就是主题，那你可能会问“再加个标题是不是就多余了？”答案是否定的，下面那行代码就是原因。当点击任何便签的时候，包括那两种节点时，也会调用这个回调函数。不过，当点击的不是那两种节点之一时，我们还要做些其他的事情。那么要怎么分辨呢？很好，我们可以直接检查标签（label），但如果我们曾经修改或添加过分类，那就意味着我们还有一个东西需要修改。因此，对于只代表实际便签的节点，我们附着上标题属性。这样我们就可以简单地判断一下所点击的节点是否定义了那个属性。如果没有，就是分类节点，回调函数也就什么都不做。

否则，我们会发现自己置身于一段if代码段中，需要获得这个便签的分类信息。为了得到这个分类，我们使用node.parent引用，也就是指向所点击的那个节点的node的引用。通过它，我们可以得到节点的标签，默认就是data属性。

在取得分类和主题之后，可以调用getNode()函数，过一会你就会看到这个函数了。它会返回指向所点击的那个Note对象的引用。自此之后，剩下的工作就只是从Node对象的字段中取来数值，填充4个要在页面里显示的数据元素。我们再次使用YAHOO.util.Dom.get()函数来获得这些字段。我们实际上要关心的就只是日期。虽然有JavaScrpit的Date对象，但是我们希望以MM/DD/YYYY的形式来展现便签的日期信息，所以就需要得到日期的各个组成部分，再构造所需的字符串了。

哎呀，这真是一大段代码！比起上面那些来，这个类其余的代码真是相当的没劲。

2. getNote()方法

下面就是我刚刚提到过的那个getNote()方法：

```
this.getNote = function(inCategory, inSubject) {

  var note = null;

  // Determine which array to search based on current category.
  var arrayToSearch = null;
  if (inCategory == "Personal") {
    arrayToSearch = personalNotes;
  } else {
    arrayToSearch = businessNotes;
  }

  // Search the array and find the match, if any, and return it.
  for (var i = 0; i < arrayToSearch.length; i++) {
    var n = arrayToSearch[i];
    if (n.getNoteSubject() == inSubject) {
      note = n;
      note.setArrayIndex(i);
      break;
    }
  }
  // Now find the note in the treeview for the note.
  note.setTreeNode(oTreeview.getNodeByProperty("subject",
    note.getNoteSubject()));

  // Not found.
  return note;

} // End getNote();
```

首先，我们要根据传入的类别判断查找哪个数组，是businessNotes还是personalNotes。然后，检查数组中每个Note对象是否与传入的主题相同。

最后剩下的工作就是得到树状视图中那个便签对应节点的引用。并把它指为Note对象。（还记得treeNode字段的注释吗？要是不记得，就回去看看，因为它们很相关。）树状视图提供了很多种不同的方法来获取一个节点的引用，其中就有getNodeByProperty()方法。回忆一下添加便签时，我曾经说过的，要给便签增加一个叫subject的自定义属性。没错，那就是这里我们指定要查看的属性！然后，返回指向那个便签的引用，如果没有匹配到就返回NULL。这样就搞定了。

3. showAddNote()方法

下面要看的方法，使用户点击Add Note菜单项时调用的，它是用来展示对话框并为用户建立数据入口的：

```
this.showAddNote = function() {

  YAHOO.log("showAddNote()");

  if (overlayOrDialogVisible) { return; }
  oMenuBar.clearActiveItem();

  overlayOrDialogVisible = true;

  // Reset all form fields.
  var now = new Date();
  var hours = now.getHours();
  var minutes = now.getMinutes();
  YAHOO.util.Dom.get("frmNewNote").reset();
  oAddNoteCalendar.clear();
  oAddNoteCalendar.select(now);
  oAddNoteCalendar.render();
  oAddNoteHHSlider.setValue((hours * 13) - 13, true, true);
  oAddNoteMMSlider.setValue(minutes * 3, true, true);
  YAHOO.util.Dom.get("divHHValue").innerHTML = hours;
  if (minutes < 10) {
    minutes = "0" + minutes;
  }
  YAHOO.util.Dom.get("divMMValue").innerHTML = minutes;
  YAHOO.util.Dom.get("newNotePM").checked = true;

  // Show the dialog and center it.
  oAddNoteDialog.center();
  oAddNoteDialog.show();

  YAHOO.log("showAddNote() done");

} // End showAddNote().
```

如你所见，这里并没有太多东西。注意第一个语句是调用YAHOO.log()。这是向日志记录控制台打印日志信息的方法。在这个类里的大多数方法中都可以看到这句调用，后面我就不再说了。

下面的两句代码也会经常看到。回忆一下我之前说过的，当显示对话框或者叠加控件菜单就会失效。这是通过判断overlayOrDialogVisible变量的值来实现的。当值为true时，就立即返回。不过如果不是true，首先要做的就是通过调用菜单控件的clearActiveItem()方法来使子菜单失效。这3行代码我不打算再说了。所以如果它们在后面又冒出来就别太惊讶！

完成这些之后，我们立即设置overlayOrDialogVisible变量，这样，在没有更多注意的情况下，菜单不会做任何事情。之后，就该清理Add Note对话框的表单字段了。不过，在这之前，我们获取一个当前时间的引用，因为我们一会儿要用它。要进行真正的清理，我们从调用表单的rest()方法开始，

它会处理分类、标题和便签文本字段。

然后，我们处理日历，首先通过调用其上的clear()清理当前选择，然后通过调用select()，并给它传递我们刚才初始化的Date对象，然后在其上调用render()。这么做的效果是设置日历为当前月份和年份，并且选择当前日期，这是合理的默认值。

然后我们处理小时和分钟滑块。方法是在它们上面调用setValue()方法。还记得滑块的数值实际上是像素值吗。要把当前时间转换成像素值，我们需要在小时上乘以13（小时的刻度的像素增量），在分钟上乘以3（分钟刻度的像素增量）。我们还通过更新两个 元素的innerHTML更新小时和分钟的文本表现形式，在做这个的时候别忘记对小于10的分钟数增加前导的零。

最后，我们要调用对话框的center()方法将其居中，并且调用它的show()方法将其显示出来。然后我们就有了原始的对话框，可以让用户创建一个新的便签了！我们添加一条简短的日志信息，以表明这个方法已经完成，就好了！

4. hideAddNote()方法

之后，我们看到的是hideAddNote()方法，其名字就可以提醒我们它是在用户点击Submit之后隐藏对话框。它是一段很简单的代码：

```
this.hideAddNote = function() {

  YAHOO.log("hideAddNote()");

  oAddNoteDialog.hide();
  overlayOrDialogVisible = false;

  YAHOO.log("hideAddNote() done");

} // End hideAddNote().
```

我准备把这段省略，假设你不需要我解释它是怎么工作的！而且，更大段的代码才刚刚浮出水面。

5. handleAddNoteSubmit()方法

我们看到的下一个方法是handleAddNoteSubmit()。也许你还记得，在init()中创建Add Note对话框的时候，我们给它的构造函数传递了一个这个函数的引用，指明这是点击Submit按钮的时候要执行的回调函数。它的工作只是简单地保存用户输入的便签，或者是在输入没有通过某些简单的验证的情况下会被拒绝。下面是这个重要的方法的代码：

```
this.handleAddNoteSubmit = function() {

  YAHOO.log("handleAddNoteSubmit()");

  // Get entered values.
  var noteCategory = YAHOO.util.Dom.get("newNoteCategorySelect").value;
  var noteDate = oAddNoteCalendar.getSelectedDates()[0];
  var noteHour = YAHOO.util.Dom.get("divHHValue").innerHTML;
  var noteMinute = YAHOO.util.Dom.get("divMMValue").innerHTML;
  var noteMeridian = null;
  if (YAHOO.util.Dom.get("newNoteAM").checked) {
    noteMeridian = "am";
  } else {
```

```
    noteMeridian = "pm";
  }
  var noteSubject = YAHOO.util.Dom.get("newNoteSubject").value;
  var noteText = YAHOO.util.Dom.get("newNoteText").value;

  // Now some simple validations.
  if (noteSubject == "") {
    alert("Please enter a subject for this note");
    YAHOO.util.Dom.get("newNoteSubject").focus();
    return false;
  }
  if (noteText == "") {
    alert("Please enter some text for this note");
    YAHOO.util.Dom.get("newNoteText").focus();
    return false;
  }

  // Instantiate a Note object and populate it.
  var note = new Note();
  note.setNoteCategory(noteCategory);
  note.setNoteDate(noteDate);
  note.setNoteTime(noteHour + ":" + noteMinute + noteMeridian);
  note.setNoteSubject(noteSubject);
  note.setNoteText(noteText);

  // Add the note to the appropriate treeview category and storage array.
  if (noteCategory == "Personal") {
    personalNotes.push(note);
    new YAHOO.widget.TextNode({label:noteSubject,subject:noteSubject},
      oTreeviewPersonal, false);
  } else {
    businessNotes.push(note);
    new YAHOO.widget.TextNode({label:noteSubject,subject:noteSubject},
      oTreeviewBusiness, false);
  }

  // Redraw treeview so it'll show up.
  oTreeview.draw();

  // Hide dialog and we're done!
  jsNotes.hideAddNote();
  YAHOO.log("handleAddNoteSubmit() done");
  return true;

} // End handleAddNoteSubmit().
```

首先，我们需要获取从用户输入的数值。

- 对于分类、标题和便签字段，我们只需要获取它们的value属性的值。
- 对日期，我们需要调用日历的getSelectedDates()方法。这个方法返回一个Date对象的数组，

不过在这个时候，我们只关心数组里的第一个元素，也就是下标[0]。请注意，默认的时候，日历只允许选择一个日期，但是getSelectedDates()仍然返回一个数组，所以它没有改变我们获取数值的方法。

- 对时间的小时和分钟，我们获取显示该值的 的innerHTML值（没理由使用我们前面看到的像素与真实数值的关系，因为我们已经在 中有最终值）。时间的上午下午的区别，我们可以检查AM单选按钮是否选中，如果是，那么是上午（AM），否则是下午（PM）。

拿到所有数值之后，我们就做一些简单的验证。这些验证只是检查看看用户是否输入了标题和便签文本，如果没有，我们就把输入焦点设置为缺失的字段并且立即返回。

如果通过了验证，那就是保存便签的时候了。第一步是实例化一个Note对象并且填充所有字段。这儿唯一让人感兴趣的东西是时间，你可以在代码里看到，它是以一种合适在详情段里展现的格式存储的，这样用户点击详情段即可显示。除此之外，都是一系列设置器调用。

下一步是给对应的存储数组和树状视图添加便签。首先，根据便签的分类有一个逻辑分支。在得知便签是属于个人还是业务类别之后，我们把它push()到合适的数组里。要把它添加到树状视图里面要求我们实例化一个新的TextNode窗口小部件，就像我们为分类创建节点那样。它接受下面的3个参数。

- 给窗口小部件的第一个参数是一个包含数据元素的对象。label属性是你在树状视图里看到的东西，subject显然是便签的标题。这样就揭开了我们前面看到的一个谜。还记得当用户点击便签的时候，我们是怎样获得标题，然后我们就可以在数组中把它找出来显示？好，这里就是在节点上设置数值的地方。
- 第二个参数是新节点在树里面的父节点：要么是个人便签节点要么是业务便签节点。
- 最后一个参数判断一开始节点是展开（true）还是折叠（false）。不过，展开一个便签节点没什么意义，因为根本不会有子节点。因此，传递false是最可靠的。

这个方法里只剩下几个细节了。首先，比较重要的是更新树状视图。这是通过在其上调用draw()实现的，结果是它被重新绘制，包括新的节点。最后，我们隐藏Add Note对话框，写出一个日志信息，然后返回——这就是新增一个新节点要处理的事情。

6. deleteNote()方法

跟在handlerAddNoteSubmit()后面的方法是deleteNote()。删除一条便签没什么复杂的，如下所示：

```
this.deleteNote = function() {

  YAHOO.log("deleteNote()");

  if (overlayOrDialogVisible) { return; }
  oMenuBar.clearActiveItem();
  if (currentNote &&
    confirm("Are you sure you want to delete the current note?")) {
    // Delete from storage array.
    if (currentNote.getNoteCategory() == "Personal") {
      personalNotes.splice(currentNote.getArrayIndex(), 1);
    } else {
```

```
        businessNotes.splice(currentNote.getArrayIndex(), 1);
      }
      // Delete from treeview and redraw.
      oTreeview.removeNode(currentNote.getTreeNode());
      oTreeview.draw();
      // Clear display fields.
      YAHOO.util.Dom.get("currentNoteDate").innerHTML = "";
      YAHOO.util.Dom.get("currentNoteTime").innerHTML = "";
      YAHOO.util.Dom.get("currentNoteSubject").innerHTML = "";
      YAHOO.util.Dom.get("currentNoteText").innerHTML = "";
      // Finally, no more current note.
      currentNote = null;
    }

    YAHOO.log("deleteNote() done");

  } // End deleteNote().
```

第一步是保证当前有一个正在显示的便签，方法是判断currentNote字段是否为空。如果没有显示便签，显然我们不需要做什么，所以实际上就结束了这个方法。

那么假设有一条便签正在显示，我们首先确信用户想删除它。（尤其是在JSNotes本身还没有存储的时候，这么做就更合理了！）一旦确认了，我们就判断该便签属于哪个类别，这样就知道应该去哪个数组删除。知道类别之后，我们就用在数组上标准的splice()方法，这个方法是JavaScript提供的用于从数组上移除元素的方法。第一个参数是一个要删除元素的数组索引——在本例里，它是便签存储在arrayIndex字段上的数值。第二个参数是要删除的元素的个数，在本例里面是1。

这些代码就可以从数据数组里面删除便签，不过我们现在还要把它从树状视图里面删除。要做这件事，我们只要调用removeNode()方法，给它传递要删除的节点的引用，你应该记得我们在点击某节点显示的时候，把它保存在了Note对象的treeNode字段里。因此，剩下的就只是调用draw()刷新树状视图，然后就完了。

最后，我们确信要清理显示字段，因为原来那个地方的东西显然已经是无效的了，因此设置currentNote字段为空，因为没有需要显示的便签了。现在，便签正式、完全、最终地被删除了！

7. showExportNote()方法

下面要检查的功能是显示Export Note叠加。下面是实现这个的方法：

```
this.showExportNote = function() {

  YAHOO.log("showExportNote()");

  if (overlayOrDialogVisible) { return; }
  oMenuBar.clearActiveItem();

  if (currentNote) {
    var s = "";
    var noteDate = currentNote.getNoteDate();
    s += "Category: " + currentNote.getNoteCategory() + "\n";
    s += "Date: " + noteDate.getMonth() + "/" +
```

```
      noteDate.getDate() + "/" +
      noteDate.getFullYear() + "\n";
    s += "Time: " + currentNote.getNoteTime() + "\n";
    s += "Subject: " + currentNote.getNoteSubject() + "\n";
    s += "Note: " + currentNote.getNoteText();
    YAHOO.util.Dom.get("taExport").value = s;
    YAHOO.util.Dom.get("taExport").select();
    overlayOrDialogVisible = true;
    oExportOverlay.show();
  }

  YAHOO.log("showExportNote() done");

} // End showExportNote().
```

假设当前正在显示一个便签，一个字符串已经构造好了。这个字符串多多少少模拟了在点击一个便签的时候显示的细节，只是用纯文本显示。然后这个字符串就插入到这个作为叠加的一部分的<textarea>中，然后这个文本被文本域的select()方法选中。然后通过调用叠加的show()方法显示这个叠加。就这些。

8. hideExportNote()方法

如果用户在Export Notes叠加中点击OK，那它就通过一个hideExportNote()调用关闭，如下：

```
this.hideExportNote = function() {

  YAHOO.log("hideExportNote()");
  oMenuBar.clearActiveItem();

  oExportOverlay.hide();
  overlayOrDialogVisible = false;

  YAHOO.log("hideExportNote() done");

} // End hideExportNote().
```

这个方法就只是调用其hide()方法，然后设置overlayOrDialogVisible为false，这样我们的菜单就又可以用了。这就完了。

9. exit()方法

我们接近完成了！下一个要看的方法是在点击File菜单的Exit选项的时候调用的：

```
this.exit = function() {

  YAHOO.log("exit()");
  if (overlayOrDialogVisible) { return; }

  if (confirm(
    "All notes will be lost!  Are you sure you want to exit?")) {
    window.close();
  }

} // End exit().
```

这儿没什么可看的。只是一个确认，保证用户的确想离开，因为一旦离开，用户所有的便签就都没了。这儿做的所有事情只是调用窗口对象的close()方法。

10. toggleLogging()方法

下面是toggleLogging()方法，先来看看代码：

```
this.toggleLogging = function() {

  YAHOO.log("toggleLogging()");
  if (overlayOrDialogVisible) { return; }

  oMenuBar.clearActiveItem();
  if (loggingVisible) {
    YAHOO.util.Dom.get("divLog").style.display = "none";
    loggingVisible = false;
  } else {
    YAHOO.util.Dom.get("divLog").style.display = "block";
    loggingVisible = true;
  }

  YAHOO.log("toggleLogging() done");

} // End toggleLogging().
```

这些代码的作用就是在loggingVisible为false的时候显示divLog，并把loggingVisible设置为true，或者当它就是true的时候隐藏divLog，然后在把loggingVisible的值改回false。

11. 其他

现在就只剩下4个方法没说了，不过它们实在是太简单了，我都看不出有什么可讲的了。我肯定你能猜得到，showUsing()、hideUsing()、showAbout()和hideAbout()，是用来显示和隐藏那个Using JSNotes和About JSNotes 覆盖控件的。你也已经看过showExportNode()方法的基本代码了。showXXX()方法就是把overlayOrDialogVisible设置为true，并调用适当的那个覆盖控件对象上面的show()方法。而对于hideXXX()方法来说，它把overlayOrDialogVisible设置为false，然后调用hide()。这几个方法没什么别的更多的了！

最后这一小块砖头之后，我们就到达了旅程的终点！我希望你能感觉到，有了YUI的帮助之后，代码量并不大，并且是相当容易上手的代码。

9.5　练习

如果你观察一个留言标签，会觉得没什么可改进的，除了不同的颜色和更大的编写区域之外。JSNotes也一样，这么一个简单的小应用，我们没什么太多可以添加的高级特性。当然，我肯定不会不给你留点练习的，这里是一些可以帮助你扩展对YUI的认识的练习，当然，它们也能帮助你扩展对JavaScript的认识。

- 增加一次清除整个目录的功能以及一次输出整个目录的功能。
- 用标签页展示界面。YUI提供了一个标签页对话框窗口小部件，我们应该可以看到两个标签页，个人（Personal）和业务（Business），然后在左边有一列留言。实际上我本打算做这个例子，

后来我觉得这是一个让你熟悉YUI的绝佳练习。

- 对于标签页界面，增加一个有两个选项的View菜单：标签视图（Tabbed View）和树状视图（Tree View），这样你可以在两个视图之间切换。我建议在index.htm里面创建两个层：一个包含你现在看到的东西，一个包含标签视图。这样就很容易在两个之间切换。
- 添加一些特效。YUI提供了一些其他特效，你可以很容易加进去。比如把使用（Using）和关于（About）的轮廓扩展到视图里面如何？

9.6 小结

本章里，我们做了一个挺有用的便签本应用。在这个过程中，我给大家介绍了一个顶级的JavaScript库，YUI。你看到了用自定义窗口小部件创建好用的用户界面是多么地轻松，并且你也了解了一些它提供的工具功能。观察这个应用，我们可以发现，在这个优秀的库的帮助下，我们只要编写很少的一点代码，就可以创建相当不错的一些功能。

第 10 章

支持拖放的购物车

大多数曾经在因特网上购物的人都熟悉那个虚拟的购物车。你会看到一个待售的商品项目列表，点击某个按钮，然后就可以把一定数量的商品添加到购物车中。然后可以付账离开，你的订单是基于购物车中内容生成的。这当然很不错，但是我们还可以继续改进。

在本章中，我们将创建一个这样购物车，它允许你把商品条目拖入其中，而不需要通过点击按钮来添加到购物车里面。我们会使用一些特效让这个过程看起来比听起来更酷。不过，同时，我们还要尊重那些关闭了JavaScript的人。对于他们我们要确保购物车柔性衰减并仍然可以工作，即使是在这种“不可思议的”条件下。

我们将会用到一个新的库MochiKit，它用来实现拖放动作。而且，你还会了解到一个很有用的新技巧：模拟服务器，在不用写真实的服务器代码的情况下处理服务需求。

那么，拿出你的钱包和信用卡吧，我们开始购物！

10.1 购物车项目的需求和目标

购物车是一个典型的在电子商务网站使用的应用范例。诸如Amazon（http://www.amazon.com）、Best Buy（http://www.bestbuy.com）、Newegg（http://www.newegg.com）、CD Now（http://www.cdnow.com）和TigerDirect（http://www.tigerdirect.com）等网站（我们只举了几例），它们都是使用购物车让用户购买其产品。这在概念上一点都不难，但是只要再加一点点活力元素，它的使用就会变得更加有意思，并且也会更Web 2.0一点。我们先来列举一下购物车应用程序要实现的目标。

- 用户可以选择一个商品条目，选择该商品的数量，然后把它添加到购物车中。他们可以把商品拖放到购物车里或者使用更手工的方式（柔性衰减的一部分，后面你将会看到的）。
- 用户应该可以在任意时间浏览购物车中的东西，包括当前的总价，并且还能够在浏览时进行修改：添加或删除商品条目以及改变数量。
- 用户可以付款并离开，也就是说，完成他们的购买交易。不过，由于本书大体上是专注于JavaScript和客户端的，所以我们就不实际写那部分了。[①]我们只是在客户端写一个“模拟”服务器。把这个模拟服务器替换成真品会是很简单的练习。
- 购物车应该柔性衰减，也就是说，不管是否打开了JavaScript都可以工作。当打开JavaScript时，就可以使用拖放功能。当关闭时，购物车的功能仍然完好，只是更加手动一些，像大多数典

① 在讲述Ajax的那章（第12章）中，你将看到一些服务器端的代码，不过那是因为并没有很好的纯客户端的方式来模拟服务器。这里，我们可以使用一个模拟服务器来完成。

型购物车那样。不过由于我们使用了仿真服务器，就并不能真正意义上关闭JavaScript，因为需要使用它来仿真服务器。因此，我们会提供一个开关JavaScript的伪开关，这样就可以看到购物车在两个环境中是怎样起作用的。

- 我们将使用一个JavaScript库来帮助我们实现比较复杂的代码片段，尤其是拖放功能。这个库就是MochiKit。

对于这个应用，我们有3个主要的关注点。首先，我们希望用户可以把商品拖放到购物车上，所以需要编写实现这个拖放效果的代码。这并不是一个Web应用程序中最难实现的东西，但它包括了相当多的细节，所以使用MochiKit库帮我们处理这些复杂的事情。

第二，我们希望这个应用程序可以柔性衰减，所以它在JavaScript被禁用的时候下依然可以工作。本书都是关于JavaScript的，并没有重点讲述柔性衰减是怎么实现的，所以这正是一个说明的好机会。你也可以想象，没有JavaScript的话，拖放购物车就不能实现，所以如果有人认为可拖放的版本是最好的话，那不支持拖放的就真的是柔性衰减了。而且，我们还可以提供可用的购物感受，即使在没有JavaScript的情况下。

第三个要考虑的事情是，我们不想在这儿花太多笔墨处理服务器端的事情，但开发过程中是需要一个服务器的，怎么实现呢？可以使用我称之为虚拟服务器[①]的技巧来实现。

让我们从如何得到柔性衰减开始吧。

10.2 柔性衰减，或者说在石器时代工作

目前，如果不使用JavaScript来写Web应用程序，简直就是钻木取火。虽然那样绝对没问题，但是在这个可以去沃尔玛买BIC打火机，然后轻轻动一下手指就点火的时代，你为什么要这么做呢？

多年以前，JavaScript因为几个比较显著的原因，口碑不太好，比如说：安全性、性能和惹人烦（实际上是使用者的问题）等。那时候，人们通常会完全关闭JavaScript，使浏览体验更加愉快些。现在还有一些人会这么做，即使已经远没那么普遍了。而且，我们需要考虑其他浏览设备，以及像手机浏览器那样的受限的设备，它们通常不支持JavaScript功能。因此，让一个应用程序在没有JavaScript的时候，跟有JavaScript时一样（或几乎一样）能用，绝对是件好事情。

那么我们到底要怎么实现它呢？可以总结成：写出在没有JavaScript下也能工作的应用程序，然后添加脚本，“激活”只有在启用JavaScript的环境下才能展示的效果。

举个例子，我们假设想要有这样一个表单：

```
<html>
  <head></head>
  <body>
    <form name="myForm" method="post" action="someAction.do">
      Your name: <input type="text" name="yourName" size="20">
      <br>
      <input type="submit" value="Submit">
    </form>
  </body>
</html>
```

① 虽然我不能声称自己创造了这个词，但老实说，我从没有在别的地方听到这么用的，所以，如果你也这么用它，那就给我的出版商发个忠诚度检查，让他们给我。开个玩笑！

显然，不管有没有打开JavaScript，它都可以工作得挺好。现在来考虑一个我们可能遇到的问题，如果用户没有输入名字会怎么样？提交到的那个服务器会不会歇菜，并返回一些错误呢？非常有可能。所以，我们不得不在服务器端检查这个问题。但是对于这么简单的验证，为什么要让服务器做呢？使用JavaScript，我们可以这么做：

```
<html>
  <head>
    <script>
      function validate(inForm) {
      if (inForm.yourName.value == "") {
        alert("Please enter your name");
      } else {
        inForm.submit();
      }
    </script>
  </head>
  <body>
    <form name="myForm" method="post" action="someAction.do">
      Your name: <input type="text" name="yourName" size="20">
      <br>
      <input type="button" value="Submit" onClick="validate(this.form);">
    </form>
  </body>
</html>
```

现在，如果打开了JavaScript，服务器就不会被卷入到这个简单的验证中来了，减少了服务器的使用和网络的占用。并且，这还提供了更好的用户感受，因为在点击Submit和看到错误信息之间，就不再会有任何延迟。

麻烦在于我们需要让这个东西不管在JavaScript启用还是禁用的情况下都能用。首先，我们需要定义在这个上下文中，“能用”到底是什么意思。很明显，当打开JavaScript时，能用就意味着在客户端做检查，如果必要的话弹出错误信息，并取消表单提交。当禁用JavaScript时，能用的意思就是表单仍然要提交给服务器，不过没有了客户端验证的好处。现在你可能会说，这个例子是有缺陷的，因为如果按钮是一个 type="submit" 的，并且验证就在表单的onSubmit中进行，那我们就已经得到所要的了，你是对的。不过，因为这是一个说明思维的例子，所以请容忍我一下。

除了那个解决方法，我们还有什么办法能让它两种条件下都可以工作呢？看看下面这个：

```
<html>
  <head>
    <script>
      function setup() {
        document.getElementById("btnSubmit").style.display = "none";
        document.getElementById("btnButton").style.display = "block";
      }
      function validate(inForm) {
      if (inForm.yourName.value == "") {
        alert("Please enter your name");
      } else {
        inForm.submit();
```

```
    }
  </script>
</head>
<body onLoad="setup();">
  <form name="myForm" method="post" action="someAction.do">
    Your name: <input type="text" name="yourName" size="20">
    <br>
    <input type="submit" value="Submit" id="btnSubmit">
    <input type="button" value="Submit" id="btnButton"
      onClick="validate(this.form);">
  </form>
</body>
</html>
```

这样，我们就尽可能满足了两边。当禁用JavaScript时，唯一可见的按钮就是Submit按钮。没错，我们真的没有经过客户端验证，但是这就是为什么说衰减要比不能用好，它还能用，只不过是以一种衰减了的方式。当打开JavaScript时，Submit按钮被隐藏了，而正常按钮会出现，这就包含了客户端验证。

本章的应用程序中，我们将做与上面非常类似的事情。在还没有看太多之前，你应该先去玩玩它。在main.js中，可以找到一个变量javaScriptEnabled。我稍后会在剖析这个应用程序的时候讲述它，不过现在，你可以简单的把它设置为true，看看在打开JavaScript的情况下，这个应用程序的样子；或者设置为false来模拟禁用了JavaScript的环境。当你把这个变量设为false时，注意在可以购买的商品条目下方有个Description（描述）链接。但是当打开JavaScript时，这个链接就不见了。默认的时候这些链接是在那里的，然后会被JavaScript移走来提供更丰富的用户感受（把鼠标悬浮在商品条目上，会展示那个链接所提供的描述）。

10.3 MochiKit库

现在我们把注意力转移到MochiKit（http://www.mochikit.com）上来。虽然我曾在第2章中提到过它，但是MochiKit的口号还是绝对值得重声的：

MochiKit 可以让使用JavaScript没那么烂。

我说，绝对是轻松愉快！严肃点说，我不认为世上有人会说JavaScript“烂”，（我当然不希望它烂了，尤其是我还在这儿写这本书！）但前面那句话依然有效：MochiKit绝对可以把很多事变得更简单有趣。

拿拖放来说。当你开发一个富客户端（不管是Java Swing还是Windows、Linux亦或其他什么）的拖放都是非常简单的，通常只是设置一些属性，并用很少的代码来实现。不过在一个浏览器中，它需要多做点工作。

- 当按下鼠标按钮时跟踪，看看它是否是在一个可拖放的元素上按下的（顺便说一下，这是你自己的标准所决定的）。
- 跟踪鼠标的移动，并相应的移动那个可拖放的对象。
- 识别鼠标按钮的释放，并确定当释放按钮时可拖放的对象在哪个对象上头。它是可拖放对象能放进去的对象吗？

- 如果对于被拖放的条目来说，释放的目标对象是可放入拖放对象的，那就处理这个事件。并且处理这个可拖放对象上应该发生什么——可能是克隆了，或者只是返回它的起始点。（这种情况下，你知道是记住这个起始点的，对吗？）
- 同时，还要处理浏览器事件模型之间的区别、CSS差异等。
- 还有，增加点特效怎样？让整个交易看上去更酷。

并不是说这些是做不到的，这些当然是可以做到的。实际上，我们已经看到了一些非常一流的实现[①]，能让我少做很多事情——至少拖的部分如此。不过，那些实现并不处理放的部分，除了字面上结束这个拖的过程。判断放下的对象是否在另外的什么东西上面完全在它们的范围之外。

因为在浏览器中拖放并非是一个简单的东西，我们有必要寻找一个可以更加省时、省力、减少麻烦的实现方法。MochiKit 就给我们提供了这样的东西。

使用MochiKit，可以如此简单地让某些东西可拖放：

```
new MochiKit.DragAndDrop.Draggable("myObject", { revert : resetIt} );
```

使用这行代码，页面中ID为myObject的对象就可以被拖放到任意一个地方！当用户释放鼠标按键时，将会调用resetIt()函数。在那个函数中，我们可以做任何想做的事情。还有比这更简单的吗？

“怎么放呢？”你问。这是非常简单的：

```
new MochiKit.DragAndDrop.Droppable("dropHere", { ondrop : doOnDrop } );
```

我们现在让页面中ID为dropHere的那个对象可以允许其他对象拖放到它上面，并且，当发生拖放时，将调用doOnDrop()函数。再问一遍，你能想象得到还有什么更简单的方法吗？

与这个一起的，MochiKit提供了一堆选项，例如让被拖放的对象返回到它起始位置的功能；让创建的对象有一个“影子”，这样其实你拖放的是一个复制品，等等。它还提供特效，比如让对象在被拖放时略微淡化。那个让对象返回起始点的选项，也并不仅仅是跳回去而已，实际上，它可以漂亮地滑回去！

我个人认为，MochiKit的拖放支持是完美的，而且也是最简单快捷的建立并使用的方法之一。在我们剖析购物车应用程序的过程中，你会看到它的实际应用，不过我想你已经对它所能提供东西垂涎三尺了！

MochiKit给这个应用程序提供的另外一个有意思的地方是它的信号（Signal）包。信号基本上就是事件，但是MochiKit里，信号的好处是它们可以不是用户事件，而可以是实际上的任何东西。例如，当点击页面中的某个对象时，如果你想要调用一个函数，你可以这么做：

```
connect('myID', 'onclick', myClicked);
```

当点击ID为myID的那个对象时，将会调用myClicked()。

MochiKit还提供了其他什么？哦，还有一些，让表10-1来说说吧！

表10-1　MochiKit众多包中的一些

包　名	描　述
MochiKit.Async	异步任务的管理
MochiKit.Base	功能性的程序和有用的比较关系

① 参阅http://www.javascriptkit.com/howto/drag.shtml，那是一个非常简单的浏览器拖放的实现。

（续）

包　名	描　述
MochiKit.DOM	无痛的（Painless）DOM操作函数
MochiKit.Color	CSS3支持的颜色提取
MochiKit.DateTime	与时间相关的函数
MochiKit.Format	格式化字符串的函数
MochiKit.Iter	迭代
MochiKit.Logging	比简单的警告信息更强壮的日志功能
MochiKit.LoggingPane	交互式的日志记录框
MochiKit.Selector	使用CSS选择器语法选择元素
MochiKit.Signal	简单的通用事件处理支持
MochiKit.Style	无痛的CSS操作函数
MochiKit.Sortable	一个可排序的对象，可以让拖放更容易些
MochiKit.Visual	可视化效果

就像本书中介绍的大部分库一样，MochiKit所提供的功能比我们这里涵盖的要多得多，最好的办法就是花些时间在MochiKit的网站上。看看文档，试试例子，感受一下它提供的东西。它具有很好的文档和一些有用的例子，我相信你不会失望的。

说明 我写到这里为止，这个应用里使用的拖放特性在大多数当前发布的MochiKit版本里还不能用，目前其主流版本是1.3.1。我很痛苦地发现这个事实，因为它在网站上的文档是给未发布的1.4版本的（不过在你读到这里的时候，这个版本很可能已经发布了）。因此，为了制作这个应用，我得从源码管理系统里获取这个未发布版本，它们使用Subversion。代码的URL是http://svn.mochikit.com/mochikit/trunk/。你可以在MochiKit网站的下载页上看到更多信息。包括Subversion客户端的建议（如果你和我一样使用Windows，我建议你的是TortoiseSVN）。当然，如果你从Apress的站点下载这个应用的代码（我希望你已经下载过了），那你就已经拿到MochiKit的1.4版本了，立即就可以用了。

10.4 仿真服务器技巧

的确，把仿真服务器当作技术看待的确有点太夸张了，但不用说，这是一个很方便的开发方法。

在你编写完整的服务器代码时，需要花费更多的功夫。显然，你需要一个服务器，还有一个Web服务器和/或一个应用服务器。你可能还需要安装些扩展，例如PHP，如果你选择使用这个技术。在某些情况下，例如Java和C#，除了JSP和ASP之外，还要涉及编译的步骤，这意味着你需要为代码修改的生效花更多的精力在重启某些服务器组件上。这还没有算上任何你可能需要的附加开发工具，比如IDE之类的。

如果我可以在服务器把所有这些都做了，不是挺好？现在，你可能在想“嘿，我有Tomcat或者IIS，或者其他一些在我笔记本电脑上运行的东西，所以我可以做这些事。”你确实可以。我就这么做！但是还涉及到了更多东西——编译、重启，诸如此类。

我说的是完全没有服务器端的开发，除了客户端代码之外，没别的。

是的，你可以这么做！实际上，如果你做得正确，可以用HTML和JavaScript来扮演服务器的部分。这样可以通过去除多余的服务器代码开发环节来大幅提高开发速度。还可以去掉一些潜在的失败点，比如网络延迟等。而且它还大大简化了环境，因为你不需要操心服务器配置是否正确、处理操作系统的权限等。我希望已经说服你了解了，这是个很实用的技巧。

但是究竟如何完成这个超人类代码的壮举呢？其实是出乎意料的简单。它的伪代码是这样的：

```
<html>
  <head>
    <script>
      function process() {
        var function = get_request_parameter;
        switch (function) {
          case "some_operation":
            some_function();
          break;
        }
      }
    </script>
  </head>
  <body onLoad="process();"></body>
</html>
```

这是仿真服务器自身。只不过是一个HTML页面（你可以直接把它保存成.htm或者.html为扩展名的文件）。让我们假定你把它叫做mockServer.htm。那么，如果你想在其他页面上有一个表单，就可以这么写：

```
<form name="myForm" method="get" action="mockServer.htm">
  Your Name: <input type="text" name="yourName">
  <input type="submit" value="Submit">
</form>
```

一个很重要的注意事项是方法必须用GET，因为那是唯一可以访问请求参数的方法。你没法访问任何通过POST传递的参数。

你还要看看mockServer.htm文件，里面有个get_request_parameter。这是获取请求参数的JavaScript伪代码。我们实际要用到的函数是第3章中的jscript.page.getParameter()，不过我们稍后会在合适的时候再看它。这里的重点是，你得到了一些参数的值，然后在那个值的基础上判断应该执行什么代码。这类似于服务器对不同的路径产生不同的处理。调用的函数可以做任何你想做的事情，包括展现它们的内容或者进入到另外一个页面。重申一下，当我们剖析应用程序时，你会看到它的实际应用。这里我只是希望你可以先理解基本概念。

我已经说明这个技术的好处了，不过，当然它也有一些缺点。由于它并不是一个真实的服务器，你就失去了服务器提供的所有功能。还有就是你得注意不要做任何在真正的服务器上没法做的事情。不过，虽然有这些缺点，以我的经验来看，从中得到的简单和速度通常还是值得的。

这些基本知识都不是问题了，那让我们来看看那个应用程序吧。如果你还没有试过那个程序，我建议你从Apress的网站下载源代码，并玩一会儿。

10.5 购物车应用的预览

让我们快速看一下本章的项目。首先，在图10-1中，你可以看到应用程序开始时的样子。这是它在禁用JavaScript模式下的样子。可以在购物车的图片下方看到，我已经向购物车中添加了一些商品条目了。

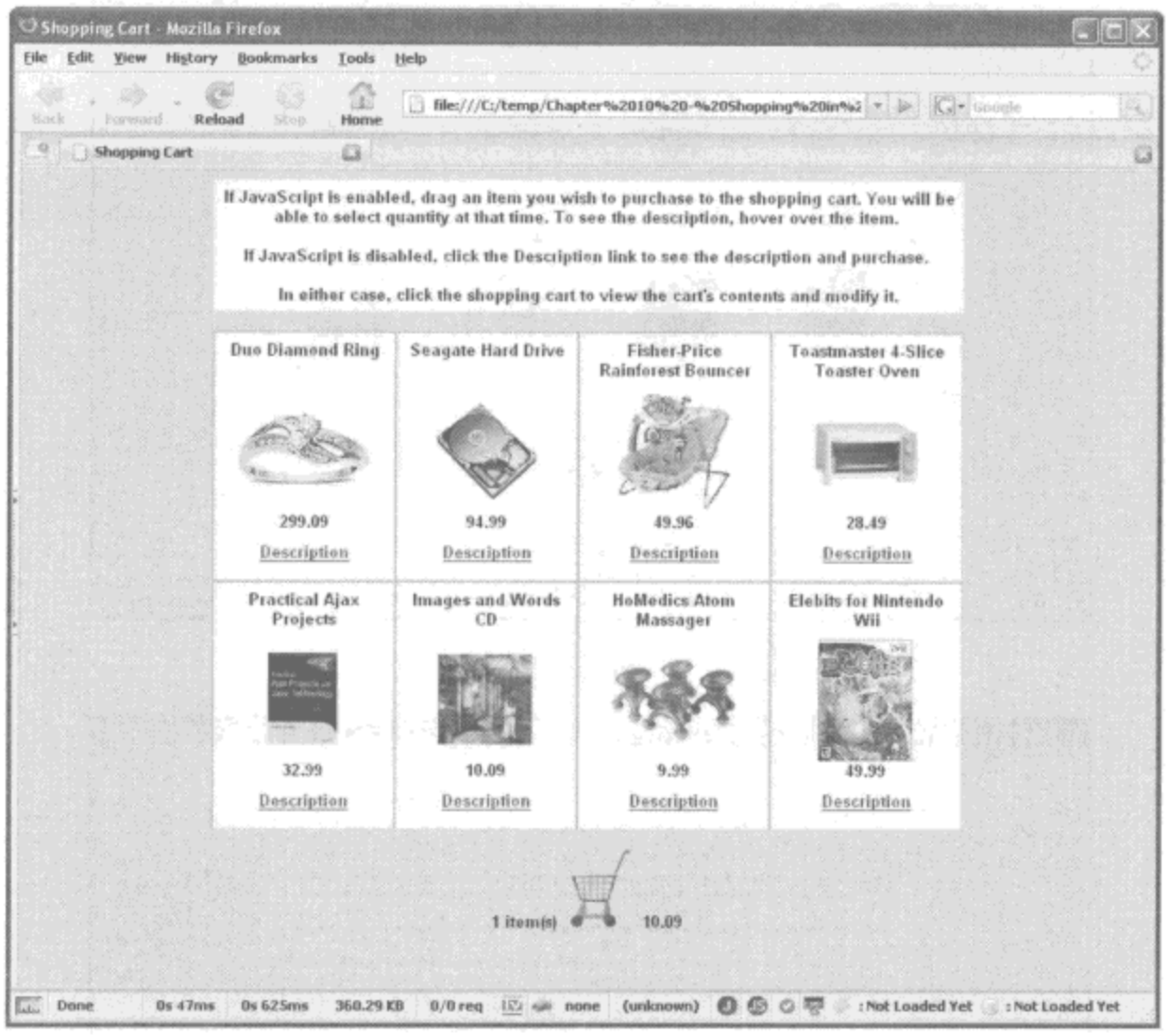


图10-1 禁用JavaScript时的类目浏览页面

与图10-1对照的是图10-2，它是打开JavaScript时的商品类目浏览。就如你所见的，实际的不同就只是没有了Description（描述）链接。现在的商品条目描述是当用户把鼠标悬浮在条目上面时可见的。另外，点击Description链接时，并没有必要去到描述页面，因为你也可以在这里进行交易，只要地把商品拖入到购物车中就可以了。

在图10-3中，你可以看到弹出的描述。当鼠标悬浮在商品条目上时，它会在鼠标的当前位置弹出，在移开图片或者开始拖动商品时，它会消失。

虽然不实际玩一下那个应用程序的话，很难描述拖放商品条目的效果，不过我还是试着做了截屏，如图10-4所示。注意，MochiKit 的确够棒，在拖放的时候，它会把被拖放对象的颜色略微淡化一些，以表明它是在被拖放。

当你把商品条目拖放到购物车中后，下一步就是指定你需要多少商品。实现这个的方法是用一个简单的JavaScript `prompt()`获取该值，如图10-5所示。

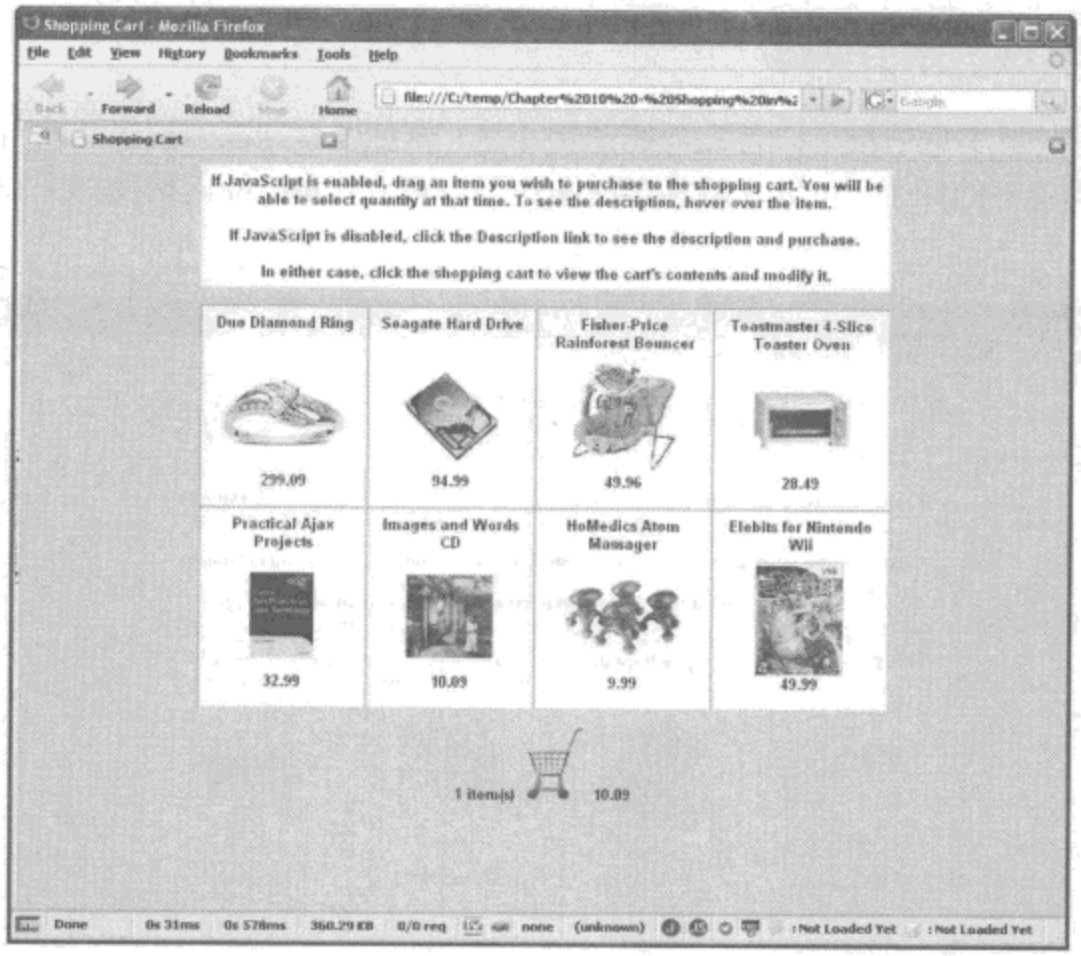


图10-2　打开JavaScript时的类目浏览页面

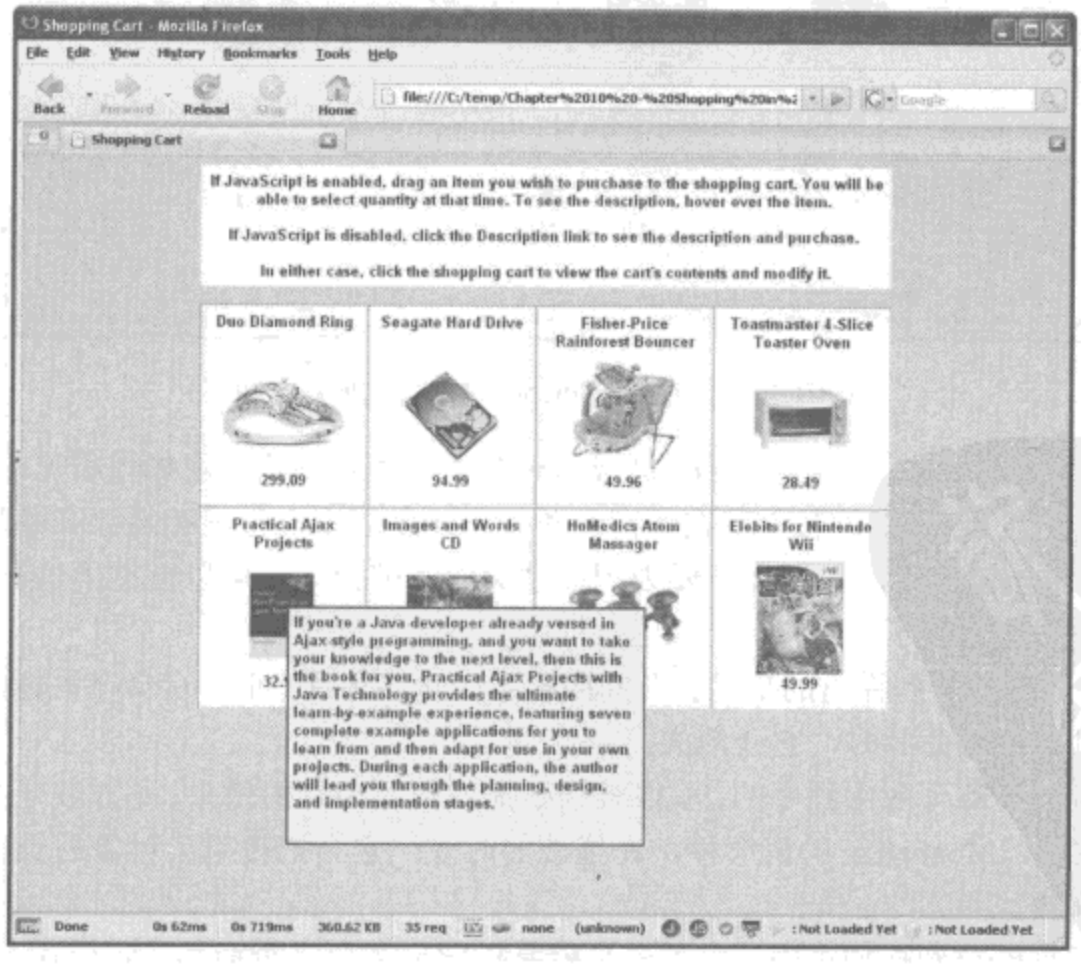


图10-3　打开JavaScript，并且用户把鼠标悬浮在商品条目之上时，看到的描述弹出框

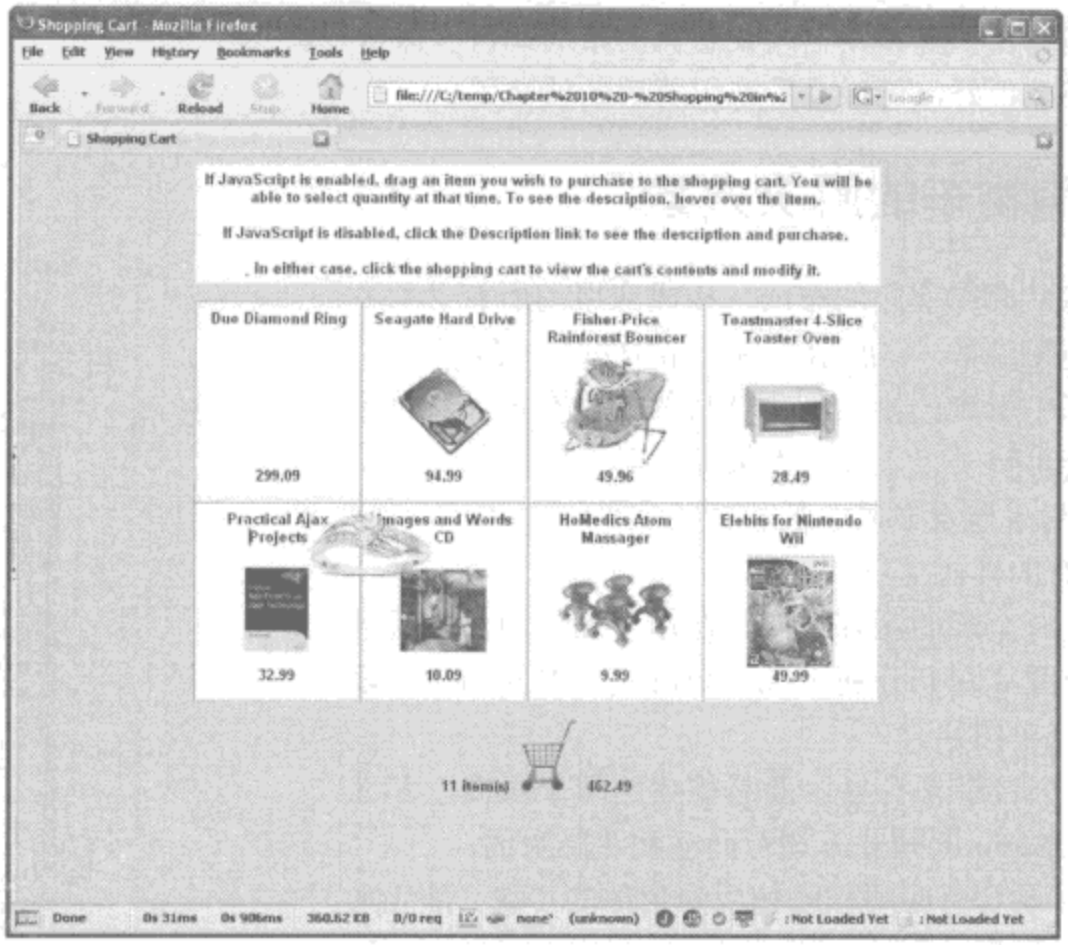


图10-4 一个拖放商品条目的例子（不过你真的应该去看看动态的）

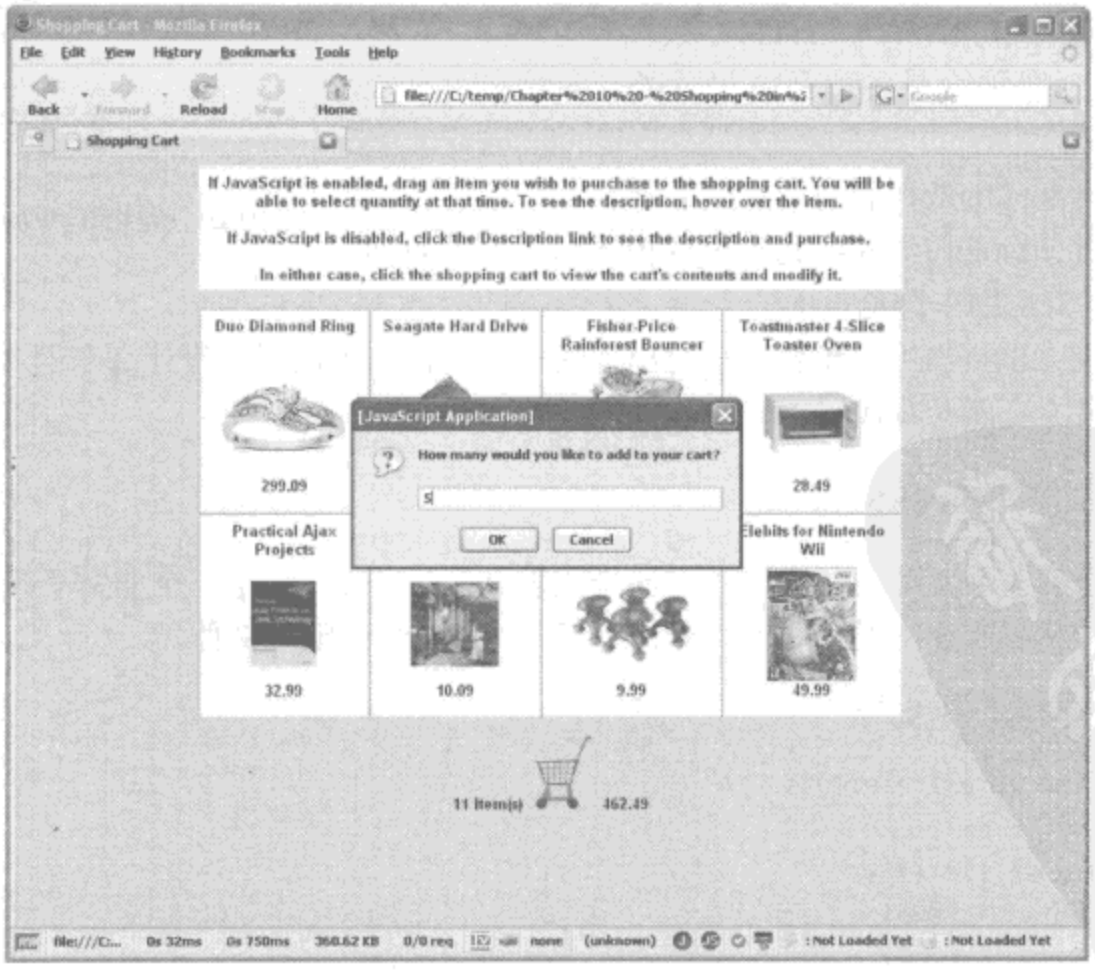


图10-5 把商品拖入购物车后的弹出框

应用中还有其他几个屏幕，不过我想让大家的期待多保留一会儿！

现在我们已经准备好，可以看看应用背后的技巧了。

10.6　剖析购物车的解决方案

照常，我们将从应用程序的布局（构成应用的所有文件）开始剖析。首先是图10-6，它展示了目录结构和文件列表。

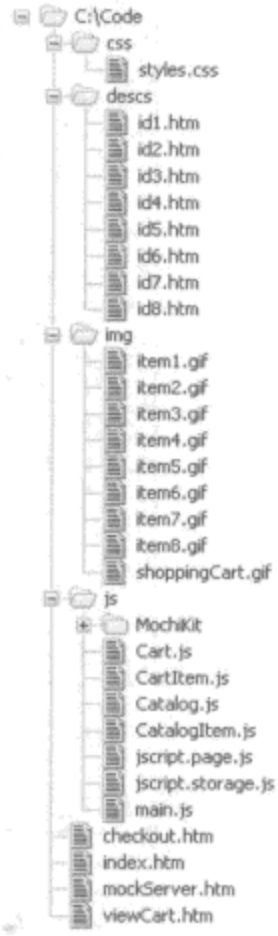


图10-6　购物车应用程序的目录布局

这是你在本书中看到的绝大部分项目的典型布局，不过无论如何我们还是会讲解一下，确保大家不觉得意外。

- css：这个目录包含了唯一的样式表文件styles.css，它封装了所有的样式信息。
- img：这里有全部的图片资源。在这个例子中，本目录包含了8个图片它们分别对应每个可交易的商品条目以及一个购物车图片。
- descs：这个目录含有显示商品描述的那个页面，和允许在非JavaScript的模式下进行商品购买的页面。
- js：这个目录含有应用程序使用的全部的（哦，是几乎全部，就如你将看到的）JavaScript，其中包括MochiKit文件，不奇怪，它在MochiKit子目录里。
- 最后，在根目录里，有4个HTML文件。
  - index.htm是要交易的商品条目分类（我称之为分类浏览页面）。
  - mockServer.htm是我们的仿真服务器。
  - viewCart.htm是在浏览购物车里的内容时看到的页面（我称之为购物车浏览页面）。
  - checkout.htm是当用户想要付款时出现的页面（在这个例子里没什么东西！）。

别再浪费时间了！该看看代码了！

10.6.1　编写styles.css

我们要查阅的第一个文件，是应用程序的样式表styles.css。它是一个非常平凡无奇的样式表，但还是应该看看，快速地看一下。你可以在代码清单10-1中看到整个文件。

代码清单10-1　styles.css文件

```
/* Style applied to all elements. */
* {
  font-family      : arial;
  font-size        : 10pt;
  font-weight      : bold;
}
```

```
/* Style for bodies of pages. */
.cssBody {
  background-color : #d0d0ff;
}


/* Style for instructions, and any other static text display areas. */
.cssInstructionsTable {
  background-color : #ffffff;
}


/* Style for the table used to display the catalog items. */
.cssCatalogTable {
  background-color : #ffffff;
  border           : 0px none #d0d0ff;
}


/* Style for the text on the checkout page. */
.cssCheckoutText {
  font-size        : 12pt;
}

/* Style for the header and footer of the view cart page. */
.cssHeaderFooter {
  background-color : #ffd0d0;
}


/* Row for the alternate row on the view cart page. */
.cssStripRow {
  background-color  : #efefef;
}


/* Style for description in cart view. */
.cssSmallDescription {
  font-size         : 8pt;
  font-weight       : normal;
}
```

如你所见，这里并没有任何需要注意的地方。第一个样式，就像本书中其他应用程序中的一样，是一个类似通吃的样式，它将用在页面中所有的东西上。其他样式的作用如下所述。

- cssBody类是适用于所有页面的<body>的。
- cssInstructionsTable类是作用于页面顶端的那个说明，以及其他一些需要白色区域的地方的，比如那个假的结账页面。
- cssCatalogTable类用在展示类目的那个表格上。它的主要的工作是去除表格边框，保留单元

格边框，这个不用CSS是不可能的。

- cssCheckoutText类用来设定那个假的结账页面的样式，使它的文字稍大一点。
- cssHeaderFooter类是你在购物车浏览页面看到的红色顶部和底部。
- cssStripRow类用于购物车浏览页面的灰行之前的切换，让用户更容易区分不同行的商品条目。
- cssSmallDescription类是用来给购物车浏览页面上的商品描述设定样式的。

现在，让我们继续去看看一些稍微有点料的东西吧。

10.6.2　编写index.htm

index.htm文件是应用程序的起始点，也是我称作类目浏览的页面。它只是简单的标签，还有一些可能有用也可能没用的JavaScript。首先，让我们看看代码，如代码清单10-2所示。

代码清单10-2　index.htm文件

```
<html>

  <head>

    <title>Shopping Cart</title>

    <link rel="StyleSheet" href="css/styles.css" type="text/css">

    <script type="text/javascript" src="js/MochiKit/MochiKit.js"></script>
    <script type="text/javascript" src="js/MochiKit/DragAndDrop.js"></script>
    <script type="text/javascript" src="js/jscript.page.js"></script>
    <script type="text/javascript" src="js/jscript.storage.js"></script>
    <script type="text/javascript" src="js/CatalogItem.js"></script>
    <script type="text/javascript" src="js/Catalog.js"></script>
    <script type="text/javascript" src="js/CartItem.js"></script>
    <script type="text/javascript" src="js/Cart.js"></script>
    <script type="text/javascript" src="js/main.js"></script>

  </head>

  <body class="cssBody" onLoad="init();">

    <div id="divMain">

      <table cellpadding="6" cellspacing="0" width="600" border="0"
        align="center" class="cssInstructionsTable">
        <tr>
          <td align="center" valign="middle">
            If JavaScript is enabled, drag an item you wish to purchase
            to the shopping cart.  You will be able to select quantity at that
            time.  To see the description, hover over the item.
            <br><br>
            If JavaScript is disabled, click the Description link to see the
            description and purchase.
```

```
        <br><br>
        In either case, click the shopping cart to view the cart's contents
        and modify it.
      </tr>
    </td>
  </table>
  <br>

  <table cellpadding="6" cellspacing="0" width="600" border="1"
    align="center" class="cssCatalogTable">

    <tr>

      <td align="center" valign="middle">
        <table border="0" cellpadding="0" cellspacing="0" width="100%">
          <tr><td height="40" align="center" valign="top">
            Duo Diamond Ring
          </td></tr>
          <tr><td height="100" align="center" valign="top" id="td_img_1">
            <img src="img/item1.gif" id="img_1">
          </td></tr>
          <tr><td height="25" align="center" valign="top">
            299.09
          </td></tr>
          <tr id="desc1"><td height="25" align="center" valign="top">
            <a href="mockServer.htm?function=viewDescription&itemID=1">
              Description
            </a>
          </td></tr>
        </table>
      </td>

.... MARKUP FOR THREE OTHER ITEMS REMOVED ....

    </tr>

    <tr>

      <td align="center" valign="middle">
        <table border="0" cellpadding="0" cellspacing="0" width="100%">
          <tr><td height="40" align="center" valign="top">
            Practical Ajax Projects
          </td></tr>
          <tr><td height="100" align="center" valign="top" id="td_img_5">
            <img src="img/item5.gif" id="img_5">
          </td></tr>
          <tr><td height="25" align="center" valign="top">
            32.99
          </td></tr>
          <tr id="desc5"><td height="25" align="center" valign="top">
```

```
            <a href="mockServer.htm?function=viewDescription&itemID=5">
              Description
            </a>
          </td></tr>
        </table>
      </td>

      .... MARKUP FOR THREE OTHER ITEMS REMOVED ....

    </tr>

  </table>

  <br>
  <center>
    <span id="spnItemCount"></span>
    <a href="mockServer.htm?function=viewCart"><img
      src="img/shoppingCart.gif" border="0"
      id="shoppingCart" hspace="6" alt="Click to view cart"></a>
    <span id="spnCartTotal"></span>
  </center>

 </div>

</body>
</html>
```

注意，这里移走了两大段HTML，因为它们与之前的那段非常相似。我说的这些段指的是某个商品条目。比如，你在代码清单10-2中看到的第一个商品条目，就是下面的HTML标记：

```
<td align="center" valign="middle">
  <table border="0" cellpadding="0" cellspacing="0" width="100%">
    <tr><td height="40" align="center" valign="top">
      Duo Diamond Ring
    </td></tr>
    <tr><td height="100" align="center" valign="top" id="td_img_1">
      <img src="img/item1.gif" id="img_1">
    </td></tr>
    <tr><td height="25" align="center" valign="top">
      299.09
    </td></tr>
    <tr id="desc1"><td height="25" align="center" valign="top">
      <a href="mockServer.htm?function=viewDescription&itemID=1">
        Description
      </a>
    </td></tr>
  </table>
</td>
```

正如你可以看到的，它就是HTML。注意描述的那个链接，它的目标是 mockServer.htm文件。这

又是一个要被真正的服务器替换的地方，我们很快就会那么做。你需要了解的另外一个事实是，这个标签所在的表的行定义了一个ID。这个很有意义！

让我们退后一步看看。注意大部分JavaScript文件都是在这个页面引入的，还要注意载入（onLoad）页面时对init()的调用。这个函数可以在main.js中找到，我们一会儿看它。目前要了解的一个重要的事情是，当执行init()时，如果应用程序是在非JavaScript的模式下运行的，那它就什么也不做。这也意味着，在那种模式下，有ID的表格行什么用都没有。

然而，在JavaScript模式下就不同了。这种情况下，这些行的链接是不可用的，也就是为什么每行都有ID——我们需要能够直接定位它们、移除其内容。

这个页面还有个购物车图标，当使用JavaScript模式时，你可以把商品条目拖入其中，以及通常通过点击它来浏览购物车的内容。在那个图片周围是一些元素，有商品数量和购物车中金额总计。注意，它们是元素，而不是 元素，这样它们可以紧挨着购物车。要记住， 元素后都跟着一个换行，所以它们无法挨着购物车展示。元素后不会自动跟着换行，所以这里用它们很合适。

好，我已经作了一些预示了，现在是揭开main.js文件面纱的时候了。

10.6.3 编写main.js

你刚看到的，main.js是被引入到index.htm里面的，它包含了在index.htm页面的onLoad事件调用的init()函数。代码清单10-3展示了main.js的内容。

代码清单10-3 main.js文件

```
/**
 * Set this to true to see the fancy version, false for the plain-jane version.
 */
var javaScriptEnabled = false;


/**
 * Called when the index.htm page loads.
 */
function init() {

  if (javaScriptEnabled) {

    // For each item...
    for (var i = 1; i < 9; i++) {

      // Remove the description link.
      document.getElementById("desc" + i).style.display = "none";

      // Hook up the description hover to it.
      var imgObject = document.getElementById("img_" + i)
      imgObject.onmouseover = cart.hoverDescriptionShow;
      imgObject.onmouseout = cart.hoverDescriptionHide;
```

```
    // Make the image draggable.
    new MochiKit.DragAndDrop.Draggable("img_" + i, { revert : true });

    // Event handler from drag starting (hides description popups).
    connect(Draggables, 'start', cart.onDragStart);

    // Create a description popup for the item.
    var descPopup = document.createElement("div");
    descPopup.setAttribute("id", "desc_" + i);
    descPopup.innerHTML =
      catalog.getItem(i).getItemDescription();
    descPopup.style.width = "300px";
    descPopup.style.height = "200px";
    descPopup.style.position = "absolute";
    descPopup.style.display = "none";
    descPopup.style.border = "2px solid #ff0000";
    descPopup.style.padding = "4px";
    descPopup.style.backgroundColor = "#efefef";
    document.getElementById("divMain").appendChild(descPopup);

}

// Make the shopping cart a drop target.
new MochiKit.DragAndDrop.Droppable("shoppingCart",
  { ondrop : cart.doOnDrop }
  );

}

// Show the cart item count and dollar total.  This only really matters
// if the user goes to the view cart or checkout pages and then goes back to
// the catalog... when the cart is empty this basically has no effect.
// Also note that what this function renders would be done by a server-side
// component if JavaScript was disabled, but we're faking it here.
cart.updateCartStats();

} // End init().
```

首先，让我们来说说那个javaScriptEnabled变量。这个全局变量，是启用和禁用JavaScript模式概念的关键。当设置为true时，这个应用程序就是在使用JavaScript的模式下的。也就是说，它所担当的角色就是说明在这个浏览器中JavaScript是否可用。当设置为false时，就等于是用户禁用了JavaScript。当然，实际上我们需要启用JavaScript，因为需要它来模仿服务器。但是这个简单的变量允许我们做等效于打开或关闭JavaScript的操作。

如果实际情况中，禁用了JavaScript会发生什么呢？那么，文档的onLoad将不再调用init()。去看看那个init()函数，首先出现的是这个：

```
if (javaScriptEnabled) {
```

如果javaScriptEnabled设置为false，那么init()也将不会执行。所以，就真的是相等的，并且你

只要改变这个变量的值，就可以在两种环境下看这个应用程序了，我建议你现在就这么做。

继续，init()中下面是一个遍历。这里的意图是，我们要想办法修改类目中的所有8个条目。首先，它获得一个我前面提到过的带ID的<tr>，并且隐藏它。这就是“移除”那个Discription链接要做的事情，因为我们在打开JavaScript时并不需要显示这个链接。然后，要打开鼠标悬浮在一个商品条目上面时显示描述的能力，这里我们需要做的所有事情只是：

```
var imgObject = document.getElementById("img_" + i)
imgObject.onmouseover = cart.hoverDescriptionShow;
imgObject.onmouseout = cart.hoverDescriptionHide;
```

在获得了指向合适图片的引用之后，我们把onMouseOver和onMouseOut句柄设置为指向购物车对象中相应的方法，这个购物车对象就是后面将会看到的Cart类的一个实例。简而言之，这是购物车本身以及它所封装的所有功能。

随后出现的，是让图像可以被拖动的部分，它是MochiKit为我们做的。非常简单：

```
new MochiKit.DragAndDrop.Draggable("img_" + i, { revert : true });
connect(Draggables, 'start', cart.onDragStart);
```

我们实例化一个MochiKit.DragAndDrop.Draggable对象，并赋给它一个指向某个商品条目图片的引用。同时还传入一些选项，准确地说是一个选项。revert选项可以是一个效果、函数或者简单的true/false的值——在这个例子中是后者。它告诉MochiKit，当拖放可拖动的对象时，不管它是否被放到购物车中（可放对象），我们都希望对象可以回到它的起始位置。默认情况下，MochiKit 就具有处理这方面的天份，它会使用Move（移动）效果把对象（在这里是我们的元素图片）飞回该图片的起点位置。它可以与reverteffect属性一起使用，这个属性是可以在这里传递的另外一个参数，它决定使用哪个效果，不过在本例中，默认的效果就相当不错了。

第二行，以connect()开始的那行，是MochiKit事件系统的一部分。它允许你给页面中各种各样的事件添加处理函数。在这个例子中，我们指的是任何Draggable对象（这也是Draggable的含义，因为所有Draggable对象都包含在Draggables集合里）在开始被拖动时，都会触发执行cart对象的onDragStart()方法。我们需要这个事件，以便在开始拖放时隐藏弹出的描述框。

init()的下一件事情是为我们的商品条目创建弹出的描述，并把它们添加到DOM中去（因为它们一开始不在那里）。这是非常典型的DOM操作代码：创建一个 实例，填充它的属性，通过设置innerHTML赋给它一些内容，并以某个元素的子节点的身份把它添加到DOM中——在这里，那个元素就是divMain ，它包围着页面中所有的内容。

init()的最后一步是让购物车成为可以拖放图片的地方。我们通过下面一行代码来完成这个：

```
new MochiKit.DragAndDrop.Droppable("shoppingCart",
  { ondrop : cart.doOnDrop }
);
```

MochiKit.DragAndDrop.Droppable类是Draggable类的补充，它的调用语法非常相似。这里，我们给它传递了购物车图片的ID，并且又传入了一些选项——在这个例子中，只是当有东西被放在那个对象上的时候，才需要调用的函数。马上你就会看到这个函数，如你所想，它应该负责响应向购物车上实际添加那个放进来的商品条目。

10.6.4　编写idX.htm

在非JavaScript的模式下写idX.htm时，用户点击在商品条目下的Description链接可以浏览描述，也可以购买它。图10-7是此时用户看到的页面。

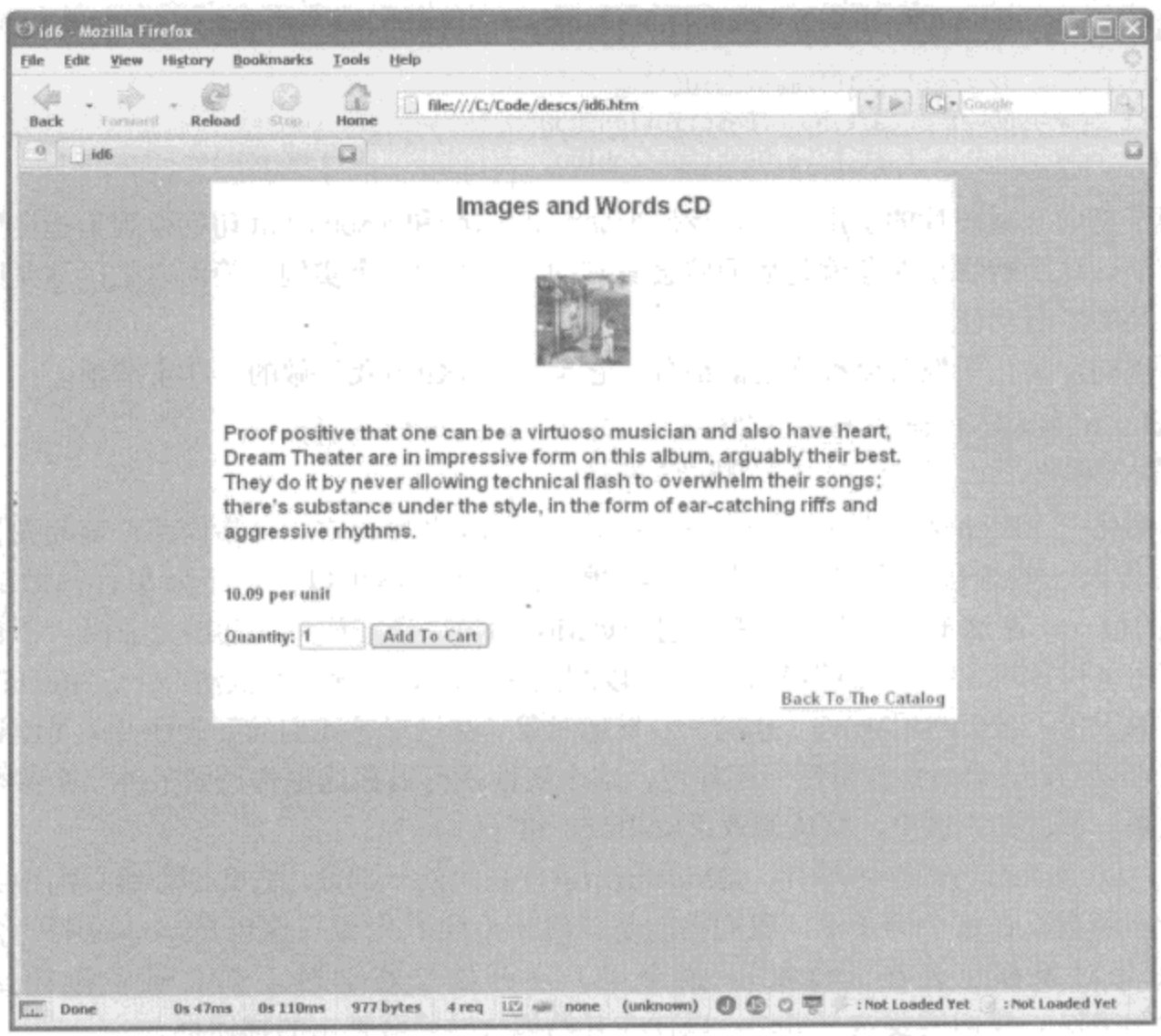


图10-7　非JavaScript版本里的描述/采购页面的例子

每个商品条目都有自己的HTML文档，叫做idX.htm，这里X是商品条目的ID号（1～8）。这些文件放在/descs目录里。因为它们除了商品信息之外全都是相同的，我在这里就只展现一个列表，如代码清单10-4所示。

代码清单10-4　id1.htm文件（其他/descs目录里的文件实际上是一样的）

```
<html>
  <head>
    <title>id1</title>

    <link rel="StyleSheet" href="../css/styles.css" type="text/css">

    <script>
    </script>
```

```
</head>

<body style="background-color:#a0a0ff;">
  <table border="0" cellpadding="10" cellspacing="0" width="600"
    align="center" style="background-color:#ffffff;"><tr><td>
    <center>
      <div style="font-size:14pt;">Duo Diamond Ring</div>
      <br><br>
      <img src="../img/item1.gif">
    </center>
    <br><br>
    <div style="font-size:12pt;">
      This 10K gold Duo ring features two round diamonds in prong settings with
      round diamond accents.  Duo Jewelry is designed to celebrate a couple's
      love.
    </div>
    <br><br>
    299.09 per unit
    <br><br>
    <form name="purchase" method="get" action="../mockServer.htm">
      <input type="hidden" name="function" value="purchase">
      <input type="hidden" name="itemID" value="1">
      Quantity:
      <input type="text" size="3" maxlength="2" name="quantity" value="1">
      <input type="submit" value="Add To Cart">
    </form>
  </tr></tr>
  <tr><td align="right">
    <a href="../mockServer.htm?function=viewCatalog">Back To The Catalog</a>
  </td></tr></table>
</body>

</html>
```

正如你可以看到的，这是相当直白的HTML，并没有涉及JavaScript。实际上，我想这里唯一有意思的地方，就是表单提交的目标以及那个“Back To The Catalog”的链接。注意它们都是指向mockServer.htm文件的，我们一会儿再看其中的细节。现在重要的是，概念上，这个文档充当了真实服务器的角色。注意参数function是作为链接中查询字符串的一部分来传递（还要注意：表单的方法是GET）的，并且作为表单中的一个字段被提交。mockServer.htm文件使用了第3章jscript.page包中的getParameter()函数来获取传递给它的参数。不过，要想使这个方法能够运转，参数必须要作为查询字符串来传递，用POST传递的参数是不能被读取的，这就是表单方法为什么使用GET的原因了。

mockServer.htm文件将会查找那个函数参数，并使用它来判断应该执行哪个操作。目前看来，这些解释已经足够了，我刚才说过，本章的后面你将看到更多细节。

10.6.5 编写CatalogItem.js

CatalogItem类代表在分类中用户可以购买的一个单独的商品条目。它有一大把数据元素来描述商

品，而且这个类还使用了经典的获取方法和设置方法做接口，用很好的面向对象的方式来访问这些元素。图10-8展示了这个类的UML图。

图10-8　CatalogItem类的UML图

CatalogItem类有如下5个字段。

- itemID：商品条目的ID。对于这个应用程序来说，我选择使用简单的数字，1～8。没人说必须这样。我只是遵循KISS原则：保持简洁，呆子！①
- itemTitle：在分类浏览页面看到的商品条目的简短名称。
- itemDescription：在JavaScript模式下，当你把鼠标悬浮在图片上面，或者在非JavaScript模式下点击Description链接时，出现的长一点的商品描述。
- itemImageURL：保存商品条目图片的URL。
- itemPrice：商品的价格。

这个类中的所有方法，都是对各种字段的获取方法和设置方法。这里也有一个覆盖了的toString()方法，这样，我们就可以获取这个类实例的更有意义的表现形式，在调试应用程序时这绝对是很有用。

虽然这是一个简单的类（只是一个DTO），但我们还是应该看看代码，在代码清单10-5中列出。

代码清单10-5　CatalogItem.js里的CatalogItem类

```
/**
 * This class represents one item in the catalog.
 */
function CatalogItem() {


  /**
   * The ID of the item.
   */
  var itemID = "";


  /**
   * The title of the item.
   */
  var itemTitle = "";


  /**
   * The description of the item.
   */
  var itemDescription = "";
```

① KISS原则，就是“Keep It Simple, Stupid!”的缩写。——译者注

```
/**
 * The URL to the image of the item.
 */
var itemImageURL = "";


/**
 * The price for one of the items.
 */
var itemPrice = 0;


/**
 * Setter.
 *
 * @param inItemID New value.
 */
this.setItemID = function(inItemID) {

  itemID = inItemID;

} // End setItemID().


/**
 * Getter.
 *
 * @return The current value of the field.
 */
this.getItemID = function() {

  return itemID;

} // End getItemID().


/**
 * Setter.
 *
 * @param inItemTitle New value.
 */
this.setItemTitle = function(inItemTitle) {

  itemTitle = inItemTitle;

} // End setItemTitle().
/**
 * Getter.
 *
 * @return The current value of the field.
```

```
 */
this.getItemTitle = function() {

  return itemTitle;

} // End getItemTitle().


/**
 * Setter.
 *
 * @param inItemDescription New value.
 */
this.setItemDescription = function(inItemDescription) {

  itemDescription = inItemDescription;

} // End setItemDescription().


/**
 * Getter.
 *
 * @return The current value of the field.
 */
this.getItemDescription = function() {

  return itemDescription;

} // End getItemDescription().


/**
 * Setter.
 *
 * @param inItemImageURL New value.
 */
this.setItemImageURL = function(inItemImageURL) {

  itemImageURL = inItemImageURL;

} // End setItemImageURL().
/**
 * Getter.
 *
 * @return The current value of the field.
 */
this.getItemImageURL = function() {

  return itemImageURL;
```

```
} // End getItemImageURL().


/**
 * Setter.
 *
 * @param inItemPrice New value.
 */
this.setItemPrice = function(inItemPrice) {

  itemPrice = inItemPrice;

} // End setItemPrice().


/**
 * Getter.
 *
 * @return The current value of the field.
 */
this.getItemPrice = function() {

  return itemPrice;

} // End getItemPrice().


/**
 * Overriden toString() method.
 *
 * @return A meaningful string representation of the object.
 */
this.toString = function() {
    return "CatalogItem : [ " +
      "itemID='" + itemID + "', " +
      "itemTitle='" + itemTitle + "', " +
      "itemDescription='" + itemDescription + "', " +
      "itemImageURL='" + itemImageURL + "', " +
      "itemPrice='" + itemPrice + "' ]";

  } // End toString().


} // End CatalogItem class.
```

这段代码很简单，不会给我们赢得“复杂代码”的殊荣，不过它的确能干活儿。当然，一个CatalogItem的实例自身不会有太多用处，它必须作为那个知道如何处理它的类目的一部分时才有意义。的确存在这个东西，毫无疑问，它就是Catalog类！

10.6.6　编写Catalog.js

Catalog类是存储所有那些用来表示用户可以购买的商品条目的CatalogItem实例的地方。这是一个非常简单的类，如你可以在图10-9的UML图中看到的。

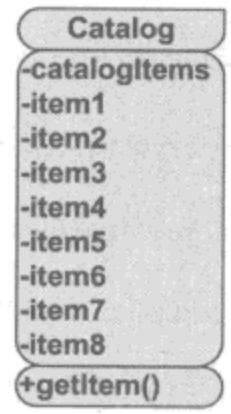


图10-9　Catalog类的UML图

catalogItems字段是CatalogItems的集合，每个元素分别对应一个用户可以购买的商品条目。如果你给getItem()方法传递某个条目的ID，它就返回指定的条目。你肯定能想象出，在很多地方都用了这个方法。

等等，我是不是跳过了什么？哦，是的，那些itemX字段，这里X是1到8。我想知道然后发生了什么？让我们看看那段代码吧，如代码清单10-6所示。

代码清单10-6　Catalog.js里Catalog类背后的代码

```
/**
 * This class represents a catalog of items.
 */
function Catalog() {


  /**
   * The collection of items for sale in the catalog.
   */
  var catalogItems = new Object;


  /**
   * Load some items so the user can play, assuming JavaScript is enabled.
   */
  var item1 = new CatalogItem();
  item1.setItemID("1");
  item1.setItemTitle("Duo Diamond Ring");
  item1.setItemDescription("This 10K gold Duo ring features two round diamonds in ➥
prong settings with round diamond accents.  Duo Jewelry is designed ➥
to celebrate a couple's love.");
  item1.setItemImageURL("img/item1.gif");
  item1.setItemPrice(299.09);
  catalogItems[item1.getItemID()] = item1;

.... CODE FOR SEVEN OTHER ITEMS REMOVED ....

  /**
   * Returns a CatalogItem by ID.
   *
   * @param  inItemID The ID of the item to return.
   * @return          The corresponding item, or null if not found.
```

```
 */
  this.getItem = function(inItemID) {

    return catalogItems[inItemID];

  } // End getItem().


} // End Catalog class.


// The one and only instance of the items catalog.
var catalog = new Catalog();
```

好的，我剪切了一些！

看见那个....CODE FOR SEVEN OTHER ITEMS REMOVED....了吗？它说的意思是有7段其他代码段都和这个注释之前那段非常相似，也就是这个：

```
  var item1 = new CatalogItem();
  item1.setItemID("1");
  item1.setItemTitle("Duo Diamond Ring");
  item1.setItemDescription("This 10K gold Duo ring features two round diamonds in ➥
prong settings with round diamond accents.  Duo Jewelry is designed ➥
to celebrate a couple's love.");
  item1.setItemImageURL("img/item1.gif");
  item1.setItemPrice(299.09);
  catalogItems[item1.getItemID()] = item1;
```

我们在这儿创建了一个CatalogItem实例，给它填充了某个商品条目，然后把它添加到catalogItems集合中。类目中的条目就出自这里。虽然我在UML图中列出了itemX，但实际上它们只是在构造Catalog对象过程中使用的，之后就再也不用了。尽管如此，为了完整性还是把它们列出来了。请注意，这8段代码并不是在Catalog类的什么方法里面，所以在Catalog实例化的时候，它们就会被执行。换句话说，它们是存在于构造函数中的，这是我们想要的结果。我们没必要让这个类的用户明确地调用一些创建方法。

现在已经了解了商品条目分类，那么让我们说说购物车和放入其中的商品吧。

10.6.7 编写CartItem.js

CartItem代表了购物车中的一个商品条目。图10-10是这个小类的UML图。

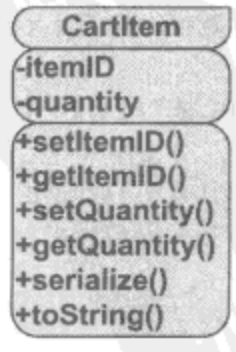


图10-10 CartItem类的UML图

CartItem只有两个重要的信息：itemID字段，它与某个CatalogItem实例的itemID字段相等；quantity，显然它是购物车中指定商品条目的数量。和在CatalogItem类中一样，你还能看到这两个常见的获取和设置方法，以及一个覆盖了的toString()。

还有一个serialize()方法。长话短说，我们想把购物车的内容存储在一个cookie中，这样的话，就需要在购物车中每个CartItem

上面都有个字符串表现形式。serialize()方法返回了这个表现形式。它只不过就是以波浪号（~）分割的itemID和quantity。

现在让我们来看看代码清单10-7，它是CartItem类的完整代码。

代码清单10-7　Cartitem.js里的CartItem类

```
/**
 * This class represents one item in the shopping cart.
 */
function CartItem() {


  /**
   * The ID of the item.
   */
  var itemID = "";


  /**
   * The quantity of the item in the cart.
   */
  var quantity = 0;


  /**
   * Setter.
   *
   * @param inItemID New value.
   */
  this.setItemID = function(inItemID) {

    itemID = inItemID;

  } // End setItemID().


  /**
   * Getter.
   *
   * @return The current value of the field.
   */
  this.getItemID = function() {

    return itemID;
} // End getItemID().


/**
 * Setter.
```

```
 *
 * @param inQuantity New value.
 */
this.setQuantity = function(inQuantity) {

  quantity = inQuantity;

} // End setQuantity().


/**
 * Getter.
 *
 * @return The current value of the field.
 */
this.getQuantity = function() {

  return quantity;

} // End getQuantity().


/**
 * Returns a serialized version of the item suitable for writing out to the
 * cookie.
 */
this.serialize = function() {

  return itemID + "~" + quantity;

} // End serialize().


/**
 * Overriden toString() method.
 *
 * @return A meaningful string representation of the object.
 */
this.toString = function() {
    return "CartItem : [ " +
      "itemID='" + itemID + "', " +
      "quantity='" + quantity + "' ]";

  } // End toString().


} // End CartItem class.
```

这里应该没有一点奇怪的地方。说白了都是非常无聊的代码！尽管如此，没有它，还是什么都做不了，所以它是一个非常重要的无聊代码！

下面，我们要看一些没这么无聊的：Cart类。

10.6.8 编写Cart.js

Cart类包含在Cart.js源文件中，实际上是本应用的大部分所在——所有隐藏在后面的真正功能所在的地方。就像我们查看main.js的时候，你所看到的那样，可以在这个类中找到所有处理各种事件的函数，比如拖放图片等。首先，让我们看看它的UML图，如图10-11所示。

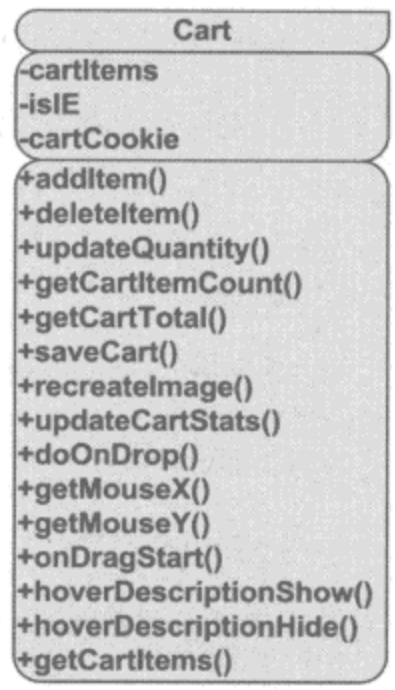


图10-11　Cart类的UML图

由于Cart.js是一个相当长的文件，我就不在这里全部列出了，而是会根据需要列出其中的一些部分。首先，我们应该看看它所包含的3个字段：cartItems、isIE以及cartCookie。

cartItems字段是一个包含CartItem实例的数组。这些是当前在购物车中的商品条目。

isIE字段是在后面用到来处理鼠标事件的。迅速看一眼：

```
var isIE = window.ActiveXObject ? true : false;
```

因为IE是唯一一个支持ActiveX控制的浏览器（忽略可能存在于其他浏览器中的插件），当我们检查window对象的ActiveXObject属性时，只有它会返回true。因此，这是一个检测我们是否在IE中运行的简单方法。

cartCookie字段只是临时需要的，但是为了完整性，我还是把它列在这里了。它是一个字符串，表示用来存储购物车内容的那个cookie的值。

1. 恢复购物车的内容

在这些字段之后，是实例化这个类时将要执行的代码。这些代码的任务就是读取存储购物车内容的那个cookie的值，从中创建CartItem对象，并把它们保存在cartItems字段中。下面是完成这一过程的代码：

```
var cartCookie = jscript.storage.getCookie("js_shopping_cart");
if (cartCookie) {
  var itemsInCart = cartCookie.split("~~");
  for (var i = 0; i < itemsInCart.length; i++) {
    var nextItem = itemsInCart[i];
    var nextItemID = nextItem.split("~")[0];
    var nextItemQuantity = nextItem.split("~")[1];
    var cartItem = new CartItem();
    cartItem.setItemID(nextItemID);
    cartItem.setQuantity(nextItemQuantity);
    cartItems.push(cartItem);
  }
}
```

如你所见，我们在第3章中建立的那个jscript.storage.getCookie()函数被用来获取cookie的值，它的名字是js_shopping_cart。如果找到了那个cookie（当然，用户第一次使用这个应用程序时，应该找不到），那我们需要解析它。

购物车保存的格式是AA~BB~~CC~DD。AA和CC是商品条目ID，BB和DD是那个商品的数量。如你所见，ID和数量通过一个波浪号（~）来分割，不同商品条目之间使用两个波浪号来分割。这样它就非常易于解析。我们只需要使用JavaScript String类的split()方法，在两个波浪号的地方截断，就可以返回给我们商品条目的数组了。然后遍历那个数组，对于每一个商品条目，我们再用一次split()，这次是一个波浪号。结果数组中的第一个元素是ID，第二个元素是数量。非常简单！

我们需要做的全部事情，就是实例化一个CartItem对象，并填充它的内容，最后把它pust()到cartItem数组中。在遍历完成第一个数组中的全部商品条目之后，cartItems字段包含了对应于购物车中每个商品条目的CartItem对象。

在源代码中，你找到的第一个方法是getCartItems()。它只是返回cartItems字段——没做别的。后面你将看到，我们的仿真服务器会需要这个。

2. 添加和删除购物车项目

下一个是addItem()方法：

```
this.addItem = function(inItemToAdd) {

  cartItems.push(inItemToAdd);
  saveCart();

} // End getCartItems().
```

很明显，向购物车添加一个商品条目调用这个方法。它接受的参数是CartItem类的一个实例，本方法认为已经正确填充。addItem()所做的就是把刚来的CartItem推入到cartItems数组中，然后调用saveCart()方法，它是负责把购物车存储为一个cookie的。我们稍候会看到这个方法。

不过在那之前，我们发现了deleteItem()方法：

```
this.deleteItem = function(inItemIndex) {

  cartItems.splice(inItemIndex, 1);
  saveCart();

} // End deleteItem().
```

这个方法和addItem()一样简单。它所做的就是使用cartItems数组的splice()方法来删除传入的商品条目和索引值，然后也要调用saveCart()来更新cookie。

3. 更新购物车中的商品数量

下一个方法是updateQuantity()，它是从浏览购物车页面调用的用来修改购物车中一个特定商品条目的数量：

```
this.updateQuantity = function(inItemIndex, inNewQuantity) {

  var cartItem = cartItems[inItemIndex];
  cartItem.setQuantity(inNewQuantity);
  saveCart();

} // End updateQuantity().
```

首先，它用传入的索引值获取一个指向CartItem对象的引用。然后调用那个对象的setQuantity()

方法，把那个传给updateQuantity()的新的数量值传给setQuantity()。最后，调用saveCart()。

在updateQuantity()之后是getCartItemCount()方法，它只是稍微复杂一点点：

```
this.getCartItemCount = function() {

  var cartItemCount = 0;
  for (var i = 0; i < cartItems.length; i++) {
    cartItemCount += parseInt(cartItems[i].getQuantity());
  }
  return cartItemCount;

} // End getCartItemCount().
```

这是一个对cartItems数组的简单遍历。对于数组中的每一个元素，也就是一个CartItem对象，我们都调用它上面的getQuantity()方法，并累加这个值。记住，我们需要购物车中待购商品的总数，它可能与购物车中CartItems的总数不同。一旦这个遍历完成了，就返回最终总和。

getCartTotal()是下面一个方法：

```
this.getCartTotal = function() {

  var cartTotal = 0;
  for (var i = 0; i < cartItems.length; i++) {
    var nextItem = cartItems[i];
    var nextItemQuantity = nextItem.getQuantity();
    var nextItemID = nextItem.getItemID();
    var catalogItem = catalog.getItem(nextItemID);
    cartTotal += nextItemQuantity * catalogItem.getItemPrice();
  }
  return cartTotal;

} // End getCartTotal().
```

这与getCartItemCount()并没有太多区别。我们又遍历了那个cartItems数组。对于找到的每一个CartItem，我们取得它的数量，并通过调用getItemID()得到它的ID。知道了ID之后，调用catalog对象的getItem()方法，这会返回与正在购买的商品条目相对应的CatalogItem。从中，我们可以得到商品的单价。把单价乘以之前获得的购物车中对应商品的数量。然后计算出的价格的总和，这样，我们就得到了购物车的总金额。[①]

4. 保存购物车

最后，看到的是saveCart()方法，我已经提到它多次：

```
var saveCart = function() {

  // Construct shopping cart string for cookie and store it.
  var shoppingCart = "";
  for (var i = 0; i < cartItems.length; i++) {
    nextItem = cartItems[i];
    if (shoppingCart != "") {
```

① 由于是样例程序，我做了简化，并没有涉及到税收、运费等。嘘，不要告诉联邦政府、UPS或者FedEx！

```
        shoppingCart += "~~";
      }
      shoppingCart += nextItem.serialize();
    }
    var expireDate = new Date();
    expireDate.setDate(expireDate.getDate()+7)
    jscript.storage.setCookie("js_shopping_cart", shoppingCart, expireDate);
  } // End saveCart().
```

我打赌你觉得应该有更多的东西。没有了，就是这些！像其他方法一样，saveCart()遍历cartItems数组。对每个元素都调用CartItem上面的serialiaze()方法，它返回一个X~Y格式的字符串，这里X是商品条目的ID，Y是数量。最终我们建立了X~Y~~X~Y格式的字符串。然后，只需要使用第3章中的jscript.storage.setCookie()函数，就可以把购物车存储在一个cookie中了。

注意，cookie的过期时间设置为7天。所以，如果你离开了购物车，然后在7天之内回来，你的商品还将在那里。这明显比你在实际的购物网站上面使用的时间要长，不过这里，它很好的说明了问题。

5. 更新状态

saveCart()方法之后的是updateCartStats()方法。这个方法是用于在底部的购物车旁边显示购物车中商品总数和总金额。往购物车中添加了商品之后，就会调用它：

```
this.updateCartStats = function() {

  // Put the total item count and dollar amount of the cart on the screen,
  // if and only if there are items in the cart already.
  var spnCartCountValue = "";
  var spnCartTotalValue = "";
  var cartItemCount = cart.getCartItemCount();
  if (cartItemCount != 0) {
    spnCartCountValue = cartItemCount + " item(s)";
    spnCartTotalValue = cart.getCartTotal();
    // Now some math: the total dollar amount has to be rounded for proper
    // display.  The basic logic harkens back to pre-algebra:
    // * Multiply the number by 10^x
    // * Apply Math.round() to the result
    // * Divide the result by 10^x
    spnCartTotalValue = Math.round(spnCartTotalValue * 100) / 100;
  }
  document.getElementById("spnItemCount").innerHTML = spnCartCountValue;
  document.getElementById("spnCartTotal").innerHTML = spnCartTotalValue;

} // End updateCartStats().
```

这个方法一开始是创建两个变量，spnCartCountValue和spnCartTotalValue。第一个变量是购物车中商品的总数，第二个变量是购物车内商品的总金额。请注意，这些是字符串值，看上去有点奇怪，对吧？因为我们知道，概念上它们是数字值。但因为它们都会被当作innerHTML插入到某些标签里，所以用字符串更有逻辑性。（实际上，如果某个变量是值为零的数字，然后我们把它插入到中，它会先转换成字符串。不过我们是想显示一个零呢？还是想什么都不显示——也就是说，显示一

个空字符串？）然后它调用Cart对象的getCartItemCount()方法获取商品总数。如果这个值是非零值，那么我们就在这个值后头附加字符串"item(s)"，把它转换成字符串。我们还会调用购物车的getCartTotal()方法获取总金额。

然后就是把这些值插入到在index.htm中合适的标签中。因为金额里可能带小数点，而且金额真的是数字值，所以它的小数位可能还相当大；所以我们在这儿还要做些手脚。不过，对于金额的数值来说，我们只需要两位小数即可，所以我们要做一些圆整。代码里的注释描述了圆整的基本算法，其实只是基础数学。

最后，代码执行了一个检查，如果数值是零，那么这段代码会给我们返回一个空字符串（当你第一次使用购物车时，它里面什么都没有就会看到这样的返回）。然后是设置元素的innerHTML，这样我们就有了购物车旁边的状态更新信息！

6. 处理拖放下的商品

当把一个商品条目放到购物车上时，调用doOnDrop()方法（你可以回想一下，在main.js中看到的代码）。

```
this.doOnDrop = function(element) {

  // Get the ID of the item dropped in the cart.
  var itemID = element.id.split("_")[1];
  // Find out how many the user wants.
  var quantity =
    parseInt(prompt("How many would you like to add to your cart?"));
  if (!isNaN(quantity) && quantity != 0) {
    // Create a cart item and add it to the cart.
    var cartItem = new CartItem();
    cartItem.setItemID(itemID);
    cartItem.setQuantity(quantity);
    cart.addItem(cartItem);
  }

  // Show the cart item count and dollar total.
  cart.updateCartStats();

} // End doOnDrop().
```

首先，和前面的方法中一样，我们取得商品条目的itemID。然后，用JavaScript的prompt()函数给用户弹出一个对话框，这样他们可以输入所需要的数量。这个调用的返回值，可能是一些非数字的值，或者是零，这两种情况都会导致无法向购物车中添加商品，所以，我们要对这些情况做一个检查。不过，假设输入的是数字，我们就继续实例化一个新的CartItem对象，填充它的属性，也就是itemID和quantity，并把它提交给cart对象的addItem()函数。最后，调用updateCartStats()，这样，新添加的商品条目就可以在购物车旁边的统计中反映出来。

7. 显示和隐藏悬浮的描述

下面是在展现商品条目上的悬浮描述时用的两个函数：getMouseX()和getMouseY()。如名字所示，它们获取一个指定的鼠标事件的X和Y位置。

因为IE和Firefox（以及其他浏览器）以不同方式提供这个信息，所以这里就是用到你之前看到的

那个isIE字段的地方。由于这些方法非常相似，为了尽量简洁，我就只展示其中一个：

```
this.getMouseX = function(inEvent) {

  var x;
  if (isIE) {
    x = (parseInt(event.clientX ) +
      parseInt(document.body.scrollLeft));
  } else {
    x = parseInt(inEvent.pageX);
  }
  return x;

} // End getMouseX().
```

在IE中运行时，事件对象是在页面作用域内提供的，其中有个成员属性是 clientX。当我们使用这个值并加上document.body.srollLeft属性后，就可以获得鼠标事件的绝对X坐标。

在Firefox或其他浏览器中运行时，inEvent被传给这个方法，它包含了pageX属性。这就是鼠标事件的X坐标。注意，我们并不需要像在IE中那样考虑页面水平滚动了多少，因为pageX的值已经算上了那个滚动距离。这就是代码中的不同分支所解决的问题（当然还有包括处理不同鼠标事件和不同的所属对象问题）。getMouseY()也是一样的，区别是：用clientY取代clientX，用scrollTop代替scrollLeft，用pageY代替pageX。

现在我们可以看看使用这两个函数的代码了，它叫做hoverDescriptionShow()：

```
this.hoverDescriptionShow = function(inEvent) {

  var itemID = this.id.split("_")[1];
  var mouseX = cart.getMouseX(inEvent);
  var mouseY = cart.getMouseY(inEvent);
  var descObj = document.getElementById("desc_" + itemID);
  descObj.style.left = mouseX;
  descObj.style.top = mouseY;
  descObj.style.display = "block";

} // End hoverDescriptionShow().
```

注意关键字this的使用。在这个方法的环境中（你应该记得它是附着在指定的商品的图片上），this是一个指向那个图片的引用。这就解释了为什么我们可以用this.id来获取itemID：它是<img>元素的DOM ID。我们再把它split()，然后取得结果数组的第二个元素，也就是所希望的itemID。然后我们调用那些鼠标方法获得鼠标在页面上的当前位置。一旦拿到了坐标和itemID，我们就获取一个指向当前鼠标所悬浮位置的商品简介的 的引用。然后把这个 的left和top样式属性设置成我们刚拿到的鼠标坐标，最后，把display样式属性设置为block，显示出这个 。这样对用户来说，就可以在鼠标指针覆盖的那个图片的位置看到商品的描述了。

Cart类中的最后一个方法是hoverDescriptionHide()方法。它使用与hoverDescriptionShow()相同的方法获得itemID（因为这个方法是挂在图片的onMouseOut事件上面的处理函数），并把display样式属性设置为none，就这么简单。

现在，让我们看看那个展示购物车内容的页面viewCart.htm。（如果你能相信它!）

10.6.9　编写viewCart.htm

viewCart.htm文件是一个信仰的飞跃[①]。其中有一些JavaScript模仿了服务器的行为。实际上，如果这是一个成熟的电子商务网站，客户端其实并不是使用JavaScript来运行服务器的。

在我们深入研究之前，先来看看这个页面，如图10-12所示。

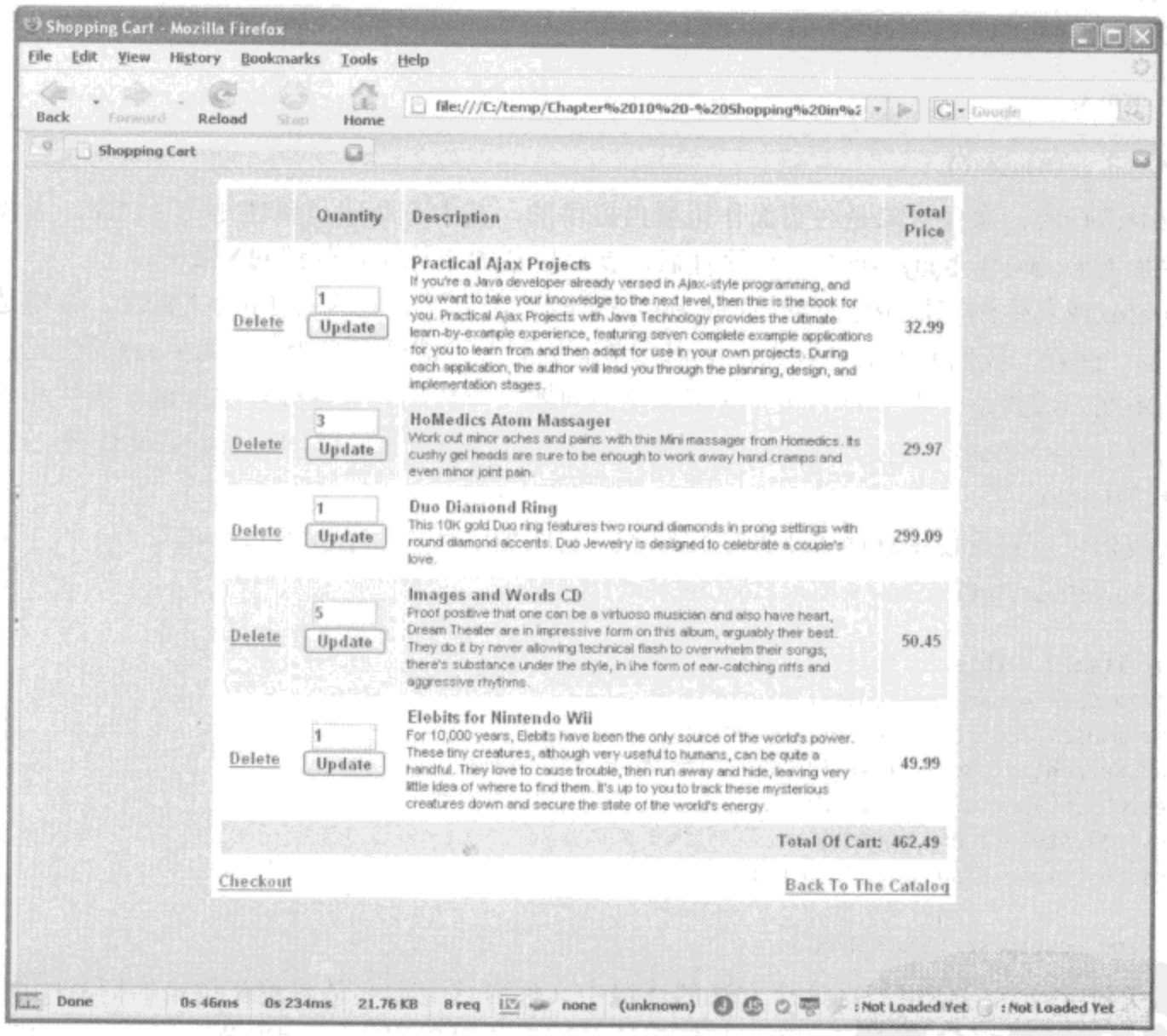


图10-12　viewCart.htm页面展示的浏览购物车的内容

这里我会列出一些必要的代码。如果你看看完整的源文件，就会发现和其他文件一样，是先引入的样式表和JavaScript文件。

1. 显示购物车内容

下面是一些封装在一个页面作用域内函数的JavaScript：viewCart()。这个函数在页面的onLoad中被调用，它就是我提过的要出现的“信仰的飞跃”。当概念化这些代码的时候，要假装它并不在那里。这个函数实际上是应该在服务器端执行的，但是由于我们并没有可用的服务器，于是它就在这里了。

① 原文a leap of faith。——译者注

这个函数负责基于购物车的内容，呈现页面HTML标记。虽然它的代码体积相当大，但还是很简单的，它基本上是由一系列字符串拼接而成的。

viewCart()是从调用cart对象上的getCartItem()开始的。记住，这将返回那个购物车中CartItems的集合，然后检查以确定这个数组的长度不是零。如果是零的话，它就只呈现一个简短的信息：

```
// No items in cart, that's easy!
document.getElementById("divCartContents").innerHTML =
  "<center><br>Your cart is empty<br></center>";
```

如你所见，这个信息被插入到divCartContents的 中，这就是用户在这里看到的所有东西了。如果购物车不是空的，那么viewCart()就开始为显示购物车的内容来构造标记。

```
var htmlOut = "<table width=\"100%\" border=\"0\" ";
htmlOut += "cellpadding=\"6\" cellspacing=\"2\"";
htmlOut += "<tr class=\"cssHeaderFooter\">";
htmlOut += "<td align=\"center\"> </td>";
htmlOut += "<td align=\"center\">Quantity</td>";
htmlOut += "<td>Description</td>";
htmlOut += "<td align=\"right\">Total Price</td></tr>";
var rowStrip = false;
var cartTotal = 0;
```

注意最后那两个变量：rowStrip和cartTotal。rowStrip是用来让表格的相邻两行使用不同颜色的，这样就有了通常在表格展现中的那种斑纹效果。cartTotal是购物车中所有商品条目的总金额。

2. 构造显示购物车内容的标记

下面，我们开始遍历CartItem对象的数组，我们获取每个元素的ItemID和quantity。然后通过调用catalog.getItem(nextItemID)得到相应的CartItem，这里nextItemID就是CartItem的itemID。手里有了那两个对象，构造一些HTML就是小菜一碟了。

```
htmlOut += "<tr";
// If this row should be striped, apply the appropriate class.
if (rowStrip) {
  htmlOut += " class=\"cssStripRow\"";
}
rowStrip = !rowStrip;
htmlOut += ">";
// Now just generate some straightforward markup.
htmlOut += "<td align=\"center\">";
htmlOut += "<a href=\"mockServer.htm?function=delete&" +
  "itemIndex=" + i + "\">Delete</a>";
htmlOut += "</td>";
htmlOut += "<td align=\"center\">";
htmlOut += "<form name=\"updateQuantity\" method=\"get\" " +
  "action=\"mockServer.htm\">";
htmlOut += "<input type=\"hidden\" name=\"function\" " +
  "value=\"updateQuantity\">";
htmlOut += "<input type=\"hidden\" name=\"itemIndex\" value=\"" +
  i + "\">";
htmlOut += "<input type=\"text\" size=\"3\" maxlength=\"2\" " +
  "name=\"quantity\" value=\"" + nextItemQuantity + "\">";
```

```
htmlOut += "<input type=\"submit\" value=\"Update\">";
htmlOut += "</form>";
htmlOut += "</td>";
htmlOut += "<td>" + catalogItem.getItemTitle() + "<br>";
htmlOut += "<div class=\"cssSmallDescription\">";
htmlOut += catalogItem.getItemDescription() + "</div></td>";
// Now some math: the total dollar amount has to be rounded for
// proper display.  The basic logic harkens back to pre-algebra:
// * Multiply the number by 10^x
// * Apply Math.round() to the result
// * Divide the result by 10^x
var itemTotalAmount = nextItemQuantity * catalogItem.getItemPrice();
itemTotalAmount = Math.round(itemTotalAmount * 100) / 100;
htmlOut += "<td align=\"right\">" + itemTotalAmount + "</td>";
htmlOut += "</tr>";
// Add cart amount to cart total.
cartTotal += nextItemQuantity * catalogItem.getItemPrice();
```

这里你可以看到rowStrip的使用。每循环一次，它的值都反转一次，轮换每行上使用的样式类。另外一个有意思的事情是，为每个商品条目计算总价。这只是一些简单的乘法：商品的数量乘以单价。不过，我们需要与你在Cart类中看到的一样的圆整；否则，你可能会得到一个位数很多的小数！你还会看到这个值最后累加到cartTotal上。

3. 显示购物车总额

谈到cartTotal，完成这个循环之后，我们就只剩一件事情要做了，就是呈现表格的底部，在那里，我们可以看到购物车的总额：

```
htmlOut += "<tr class=\"cssHeaderFooter\">";
// Now some math: the total dollar amount has to be rounded for proper
// display.  The basic logic harkens back to pre-algebra:
// * Multiply the number by 10^x
// * Apply Math.round() to the result
// * Divide the result by 10^x
cartTotal = Math.round(cartTotal * 100) / 100;
htmlOut += "<td align=\"right\" colspan=\"4\">Total Of Cart: " +
  " " + cartTotal + "</td>";
htmlOut += "</tr>";
htmlOut += "</table>";
document.getElementById("divCartContents").innerHTML = htmlOut;
```

我们又用了一次数学，都已经是太熟悉了。（提示：如果我们有一个外部函数可以用于圆整数值为指定的位数岂不更好？可以很自然地放在jscript.math，不是吗？）

我们只需要把htmlOut字符串的值添加到divCartContents的 中，就构建了那个页面的HTML标记了。

如我所说，它是非常直白的代码，不过要重复一下，虽然它在这儿，你还是要假装并没有这段代码。某种意义上它不在这里，是因为它本应该在服务器上执行的，但是它确实写在这里，你应该理解它，哪怕是需要假装它是某种意义上的不在这里——对不起，今天太啰嗦了。

10.6.10 编写checkout.htm

解析这个文件让我觉得有点不好意思，因为它完全是些琐碎平淡的代码。尽管如此，我还是喜欢完整性，所以无论如何还是要这么做。在图10-13中，你可以看到checkout.htm文件的结果。

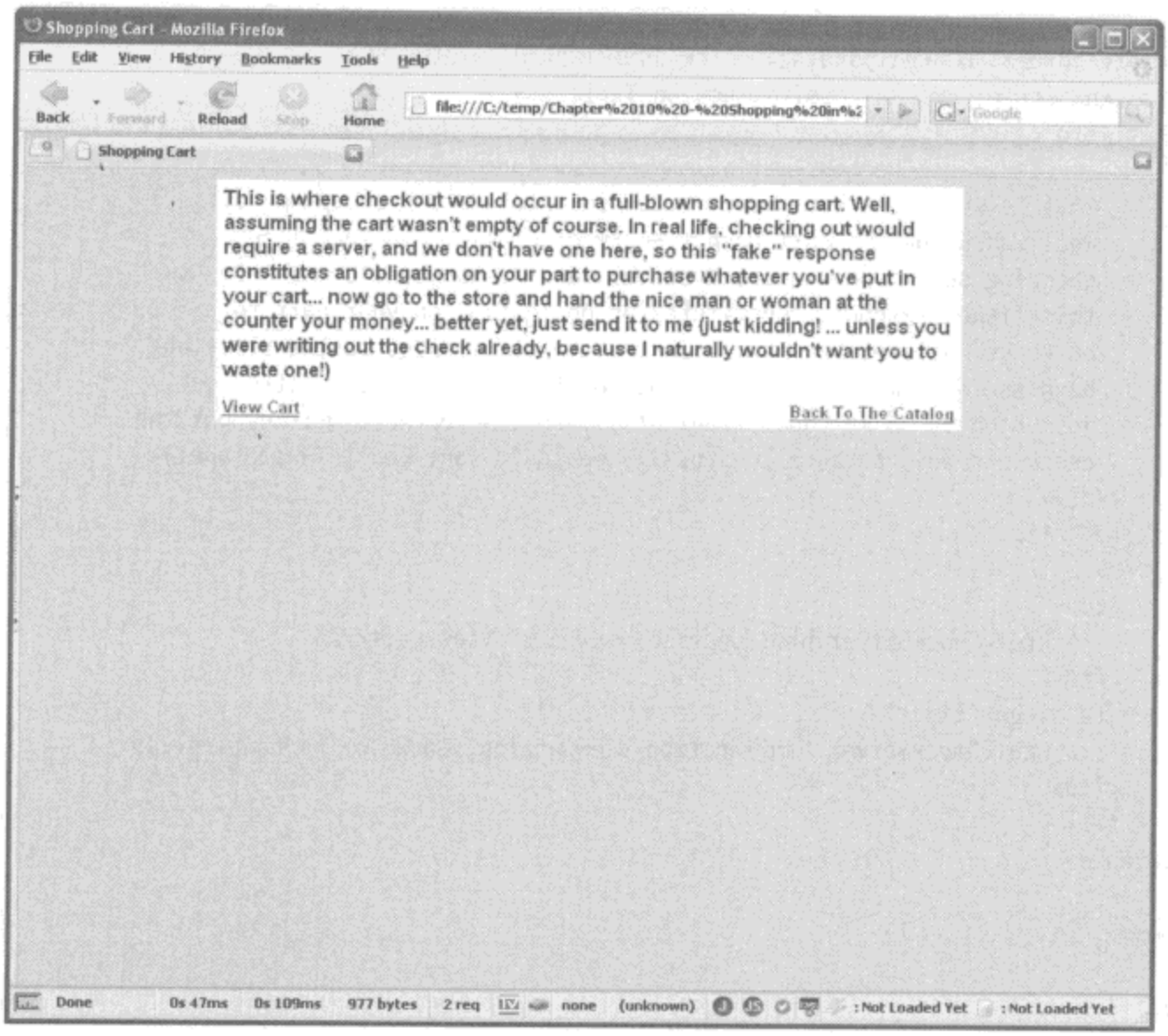


图10-13 结账页面（我承认并没有太多可看的，完全是为了完整性）

checkout.htm文件基本上是展示用购物车买东西的最后一个步骤。当用户点击购物车浏览页面的Check Out链接时，将会访问服务器并完成购物，可能是要获得类似用户的信用卡信息、运送要求等东西。由于我们并没有处理服务器端，取而代之的，我们就使用了这个页面。它只是我徒劳的幽默感和留给服务器处理的位置。

这个页面的代码如代码清单10-8所示。

代码清单10-8 checkout.htm文件——说它不是火箭技术的确有点过！

```
<html>

  <head>

    <title>Shopping Cart</title>
```

```
    <link rel="StyleSheet" href="css/styles.css" type="text/css">

  </head>

  <body class="cssBody">

    <table border="0" cellpadding="6" cellspacing="0" width="600"
      align="center" class="cssInstructionsTable">
      <tr><td colspan="2">
        <div class="cssCheckoutText">
          This is where checkout would occur in a full-blown shopping cart.
          Well, assuming the cart wasn't empty of course.  In real life,
          checking out would require a server, and we don't have one here, so
          this "fake" response constitutes an obligation on your part to
          purchase whatever you've put in your cart... now go to the store and
          hand the nice man or woman at the counter your money... better yet,
          just send it to me (just kidding! ... unless you were writing out the
          check already, because I naturally wouldn't want you to waste one!)
        </div>
      </td></tr>
      <tr>
        <td>
          <a href="mockServer.htm?function=viewCart">View Cart</a>
        </td>
        <td align="right">
          <a href="mockServer.htm?function=viewCatalog">Back To The Catalog</a>
        </td>
      </tr>
    </table>

  </body>

</html>
```

它的字面意思就是些直白的HTML。在底部，你可以看到两个指向虚拟服务器的链接，一个是返回购物车浏览页面的，一个是返回分类浏览页面的。除了那些链接，这个页面就没什么特别的了。

这个应用程序就只剩下一部分需要探究了，而且它还是个关键部分：仿真服务器。不要换台！

10.6.11　编写mockServer.htm

经过长途跋涉，我们终于来到了迷宫的最后一段：包含在mockServer.htm中的仿真服务器。就像之前提过的仿真服务器的想法，它就是一个简单的HTML页面，它是应用程序所有请求的目标，并把这个当作是一个服务器。你已经看过一些关于如何实现这个想法的简单示例，实际上，成熟的mockServer.htm并没多太多东西。

在一些初始的JavaScript引入之后，我们就会看到让一切正常运转的关键，就是那个process()函数：

```
function process() {
```

```
    var func = jscript.page.getParameter("function");
    if (func) {
      func = "process" + func.substr(0, 1).toUpperCase() + func.substr(1);
      if (eval("window." + func)) {
        eval(func + "();");
      } else {
        alert("Unimplemented function received")
      }
    }

  } // End process().
```

这个函数在页面的onLoad中调用。就是它驱动了其他东西，你可以把这个函数看作服务器上用于判断执行哪个功能的组件（比如，Java世界中的FrontServlet或者是Microsfot世界里的切换用的ASP页面）。它接受请求参数里的function，并通过判断它的值，调用适当的函数为调用的那个功能提供服务。做法是获取请求参数中的“function”参数的函数名，把字符串的第一个字母大写，然后给这个字符串值加上前缀“process”组成要调用的那个函数的名称。比如，对于函数viewDescription，我们组成的函数就是processViewDescription（马上你就会看到它），它也是在这个页面上展现的函数之一。一旦我们有了函数的名字，就检查看看它是不是窗口对象的一个子对象，如果是，就使用eval()函数来执行它。如果得到的是一个未知的函数，我们就弹出一个警告，说函数没找到。这肯定不是能想到的最健壮的机制，但已经可以满足我们的需求了。

process()调用的那些函数，用来处理某个特定的服务请求，起点是processViewDescription()。这是在非JavaScript模式下点击Description链接时调用的。它做了一个简单的重定向到/descs中相应的页面：

```
  function processViewDescription() {

    var itemID = jscript.page.getParameter("itemID");
    window.location = "descs/id" + itemID + ".htm";

  } // End processViewDescription().
```

在这个例子中，被调用的函数也会传递一个itemID参数，是jscript.page.getParameter()帮我们早就搞定的了。因为在/descs中的页面叫做idX.htm，这里X就是itemID，所以要构造合适的URL并通过设置window.location重定向到它就很简单了。

下一个是processPurchase()，它只是稍微有一点复杂，当用户在非JavaScript模式下，点击商品描述页面的Add To Cart按钮时，就会调用它：

```
  function processPurchase() {

    // Add new item.
    var newItemID = jscript.page.getParameter("itemID");
    var newItemQuantity = jscript.page.getParameter("quantity");
    var itemToAdd = new CartItem();
    itemToAdd.setItemID(newItemID);
    itemToAdd.setQuantity(parseInt(newItemQuantity));
    cart.addItem(itemToAdd);
```

```
  window.location = "viewCart.htm";

} // End processPurchase().
```

再一次，这里我们获得itemID参数以及quantity参数。然后实例化一个新的CartItem对象，设置相应的itemID和quantity。然后，调用cart.addItem()，并把CartItem传给它。这个函数也负责重写cookie。然后我们重定向回viewCart.htm，这样添加的商品条目就会反映给用户了。

下面看到的是processUpdateQuanitity()，当用户在非JavaScript模式下，从浏览购物车页面修改商品数量时就会用到它:

```
function processUpdateQuantity() {

  var itemIndex = jscript.page.getParameter("itemIndex");
  var newQuantity = jscript.page.getParameter("quantity");
  if (newQuantity == 0) {
    processDelete();
  } else if (newQuantity > 0) {
    cart.updateQuantity(itemIndex, newQuantity);
  }

  window.location = "viewCart.htm";

} // End processUpdateQuantity().
```

开始部分类似processPurchaseItem()，但是后面就有些不同了。首先，它检查用户是否输入了零。如果是的话，就调用processDelete()来从购物车中移除那个商品条目。如果是大于零的值，我们需要做的就是调用cart对象的updateQuantity()方法，把商品条目在cartItems数组中的索引值（这是viewPgae.htm文件中一个隐藏的表单字段）和那个新的数量传给它，就完成设置了。注意，如果用户输入了一个非数字的值，数量就不会被更新，但也不会弹出任何错误。输入像12A这样的东西的话会更有意思：它将记录为12，这比报错要好得多！

说到processDelete()：

```
function processDelete() {

  var itemIndex = jscript.page.getParameter("itemIndex");
  cart.deleteItem(itemIndex);

  window.location = "viewCart.htm";

} // End processDelete().
```

这里我们需要的就是itemIndex参数，把它传递给cert.deleteItem()，你就做好了！一个快速地重定向到viewCart.htm，我们就完成了！

只剩下3个函数了，而且它们基本上是一样的，所以我在这里就介绍其中的一个。这3个函数就是processViewCart()、processViewCatalog()和processCheckout()只是简单的重定向到一些HTML页面，就像这样：

```
function processViewCart() {
```

```
    window.location = "viewCart.htm";

} // End processViewCart().
```

这里选择退出而不是定向到一个最终的HTML页面还是为了模拟一个服务器。如果希望浏览购物车的内容，我们可以直接跳到某个页面去——JSP、ASP、PHP等。但是有些时候，我们首先需要使用一些服务器组件，因为JSP、ASP和PHP是视觉组件，对于现代Web应用程序来说可能还会更多。比如，如果应用程序使用模型-视觉-控制体系结构（Model-View-Controller），[①]它将会有控制和模型层。所以，如果我们要扮演服务器那部分，就需要完整的实现。那也就是说，即使只是简单地重定向，服务器也要做更多东西，比如呈现viewCart.htm页面中的内容。

哇！就是它了。我们已经完成了！希望你很享受这个过程。我想整个仿真服务器技术真的是很有用的东西，它可以帮你节省时间，减少了开发过程中头疼的问题。希望你也了解了一部分MochiKit以及如何真正使用它来做东西。

而且，你有了一个有用的小购物车了！只要实现最终的结账平台，你就能用它了！

10.7 练习

本章中展示的应用程序，可能并不会在未来成为任何人的博士论文，因为它并不都是那么复杂，但是它做了绝大部分购物车应该做的事情。但是我也要说，这里有很多可以提高的地方，可以获得更多经验。

- 边角用圆角：MochiKit提供了把元素边角变圆的效果，如果把这个应用到页面中所有方框（白色区域）上会很好。我故意把这个作为一个修改建议留给你，这样你就需要更加深入研究一下MochiKit的文档。我告诉你，它并不是像它看起来那么容易，而且实现起来需要更多的应用程序修改。（提示：这个特效可以在表上起作用吗，我想知道？）
- 实现服务器端：是的，本书是关注与客户端的，没错，这个特殊的应用程序用它自己的等效方法避免了服务器部分。不过用你选择的技术建立服务器端也是应该的。
- 添加一些效果：这也是一个熟悉MochiKit的好办法。比如拖放到购物车上的商品自动缩入购物车如何？当你从购物车中删除商品时，在提交给服务器之前，先淡出并且折叠表怎么样？

我确定你可以想到更多，不过这些将会给你一个好的开始。玩得开心！

10.8 小结

在本章中，你第一次尝到了MochiKit，一个非常漂亮的小JavaScript库。了解了它的拖放功能是多么强大，但仍然很易用，可以用非常短的代码做很多事情。你还看到了如何创建一个仿真服务器，它允许你直接在浏览器中做所有开发，而不用因为服务器掺和进来而改变自己的方法或者是代码。最后，你还看到了如何把一个购物车这样老式的应用改造、装饰成符合需要下一代Web体验的人们要求的东西。

① 模型-视觉-控制（MVC）架构是实现应用的一种方法，它把用户看到的（视觉层）组件和所使用的数据（模型层）以及操作数据的商业逻辑（通常是模型层的一部分）组件进行分解、分层。这种架构的实现是视觉层绝不会直接和模型层交互，而是通过一个中间（控制）层进行。目标是对其中某个层的修改不会影响其他层。

第 11 章

休息时间：JavaScript游戏

在我出版的第一本书（*Practical Ajax Projects with Java Technology,* Apress, 2006）中，最后一个项目（the apex of the book）是一个名叫“Ajax战士”的探险游戏。它可能是我的一个习惯的开端，从那开始，我所写的每本书都包含一个游戏，因为那也是我们本章中要创建的东西！哦，不，不再是“Ajax战士”了。这次将会是更加带有街机风格的游戏[①]，因为这样网络延迟就不会再来烦我们了。

你将会在这里看到很多有用的小技巧，很多JavaScript和DOM脚本，甚至是相关的一些基础游戏原理。最后，会得到一些可以用于在任何上班时间偷个小懒的东西，或者用在任何有浏览器的地方！我们都知道那句谚语——只会用功不玩耍，聪明孩子也变傻，所以，我们开始游戏，怎么样？

11.1 K&G街机游戏的需求和目标

我们要创建的游戏是PocketPC[②]游戏的一个移植，我把它叫做 K&G 街机。K&G代表Krelmac和Gentoo，是两个决心摧毁地球的爱说笑话的外星人。糟糕的是，它们的智力和青年弱智差不多，但是手里却有量子毁灭枪！

可以在 http://www.omnytex.com/kgarcade 看一下 K&G 游戏，在里面，你扮演Henry，一个温和的、来自古老的17世纪英国的农场主。一天晚上，Krelmac和Gentoo绑架了你，你试图逃出他们的太空舱，太空舱有5个迷宫层，里面有可以远程移动的机器人，一旦碰到它们，你就会被它们杀死。在这5层中的每一层，都有5个你需要通关的迷你游戏（在60 s内达到指定分数），然后才能成功逃脱。你可能还会遇到其他被绑者，要和他们交谈，并试着得到他们的信任，这样他们会给你提供关于某个特定迷你游戏的线索，要不然你可能几乎无法通关。

完整的K&G街机是遮幕电影模式的游戏，还有Krelmac和Gentoo开的玩笑，并且通常自己做害虫。它包括全部的原声道和手绘的卡通图片。K&G街机实际上是第二个有这些特效的电影，第一个是Invasion:Trivia!（http://www.omnytex.com/products_invasion_info.shtml）登录那个站点也会用Flash卡通给你介绍这些特点。

现在，我们的目标并不是把那个完整的K&G街机移植到JavaScript上。实际上，如果可能，那也会困难重重，而且会占用整本书的大小来写它！我的做法是缩小一下规模，只实现迷你游戏部分。实际上，我们将创建25个迷你游戏中的3个。来看些详细介绍。

- 要实现3个迷你游戏——宇宙松鼠，它的概念类似于经典的搬运工、死亡陷阱类游戏，它的灵

① 原文为arcade-style game。——译者注

② PocketPC是与Palm齐名的掌上电脑平台。——译者注

感来自于电影《夺宝奇兵》系列，Refluxive，它很像打砖块系列的游戏。（当然了，并没有真的打碎任何东西！）

- 要实现一个迷你游戏选择画面，其中包含了每个迷你游戏的截屏。
- 应该尽可能地重用已存在的代码。不过，对于这个游戏，我们不使用任何其他的库。这是因为，在写游戏的时候，通常会想要离硬件尽可能的近，这里也是一样的。
- 每个迷你游戏都应该有它们自己的类，并且都从基类继承公用的代码。
- 要优先考虑可扩展性，这样后面就可以毫不费力添加更多的迷你游戏了。
- 总之，我们想要保持全局作用域尽可能干净，并且贯穿始终地使用很好的面向对象设计技术。

在进行游戏编程时，通常你都要尽量进入到底层——尽你所能地接近硬件。这样做的原因很简单：性能。在一个游戏中，很短的时间内要发生并完成很多事情，所以不能有大量多余代码的执行或者附加的工作。最好的实现方法之一就是不要使用那些JavaScript库。现在来说，这并不是绝对的。使用一个写得很好的库通常能获得更好的性能效果。还有一件事是真的，那就是在现代，不管你用不用库，一般来说都不会像以前那样进入到特别底层。在过去，使用汇编语言来写一个游戏的重要部分是很平常的，这样它就可以尽可能得最优也最紧密。现在，这个做法不流行了（是可以做的，但并不常用）。所以，在这个特别的应用程序中，就不使用任何JavaScript库了，我们将全部使用“赤裸裸的”JavaScript。

在JavaScript中进行游戏编程与使用任何其他环境来编程是有根本区别的。细节部分有所不同，不过基本概念是一样的。这里我不会把所有概念一起全列出来，而是看代码的时候逐步讲解它们。

带着这个观点，自己去亲身体验一下游戏，开始我们的K&G Arcade探险吧。

11.2 K&G街机游戏的预览

图11-1展示了游戏标题的截屏，当K&G Arcade第一次被载入到浏览器时，你可以看到这个画面。最初的K&G Arcade是与我自己的一个小型PocketPC软件公司Omnytex Technologies以及Anthony Volpe的公司Crackhead Creations (http://www.planetvolpe.com/crackhead)合作的产品。Anthony Volpe是本书中示例图片的作者，[①]因此就有了屏幕中的那些图标。Anthony也是K&G Arcade和Invasion: Trivia! 的图形作者。

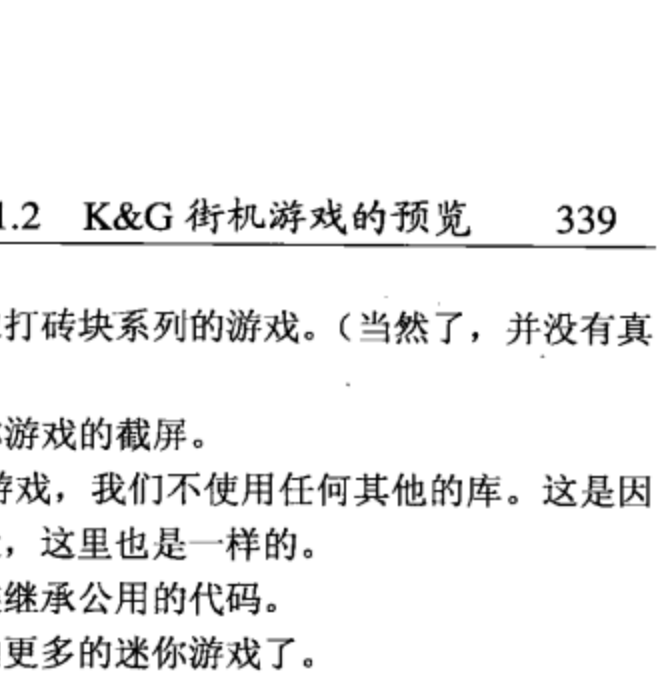


图11-1 K&G Arcade标题画面

图11-2中所示的，是在标题画面之后看到的游戏选择画面。在那里，你可以选择要玩的迷你游戏。它还描述了一些介绍以及迷你游戏的预览和简短的描述。

图11-3中展示的宇宙松鼠是K&G Arcade中的一个迷你游戏。这个游戏是从经典的青蛙过街游戏中得到的灵感，不过变异了一些：你扮演试图得到一个跨星

① 如果你想看看更多Anthony Volpe的作品，和其他我跟它一起做的有关Krelmac 和Gentoo的和其他相关的东西，可以看一下Downtown Uptown网站：http://www.planetvolpe.com/du。那里，你可以找到更多Krelmac 和Gentoo的Flash卡通模式的冒险经历，还有一大群这个宇宙中的其他人物。不要让那些奇怪的东西把你吓跑！拥抱不可思议的东西吧！

系橡果的巨大宇宙松鼠。（我可不知道那个“跨星系橡果”是什么东西！）玩家需要避免外星人、小行星、宇宙飞船、彗星等，最后到达那个果子。

图11-2　游戏选择画面

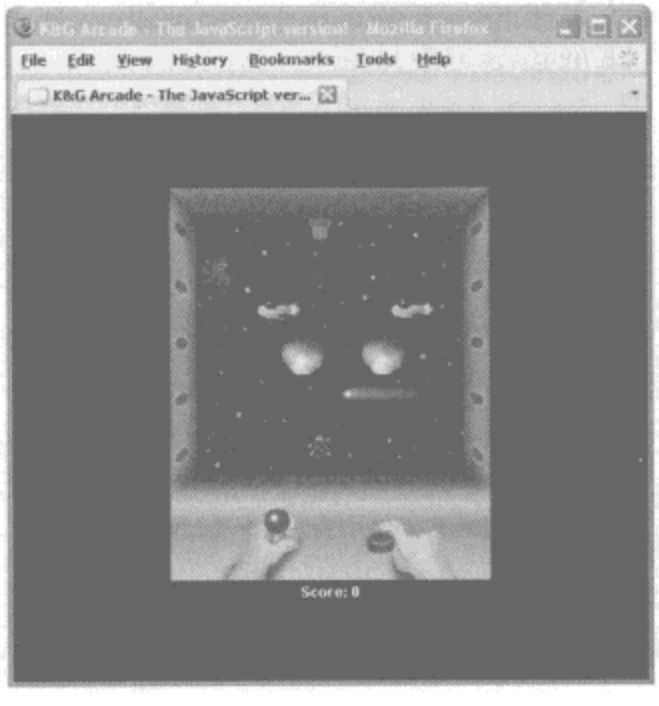


图11-3　宇宙松鼠

图11-4展示了迷你游戏——死亡陷阱，它的灵感来自于电影《夺宝奇兵》系列。你的目标是，通过从这个瓦片到那个瓦片最后到达屏幕顶端的那扇门。问题是，一些瓦片是带电的，如果你碰到了错误的瓦片，就会被电死。

最后，在图11-5中，你可以看到第3个迷你游戏，Refluxive。这与打砖块的概念类似，但是不用真的打破什么东西！我想起来了，实际上，这个游戏非常类似于电影《生死时速》。还记得桑德拉·布洛克和奇努·李唯斯演的那部电影，就是在里面，他们必须让公共汽车保持在某个速度之上，否则炸弹就会爆炸那段？这儿的概念和那部电影类似。某人要求你必须不停地跳，而你必须这么做——不许问问题！

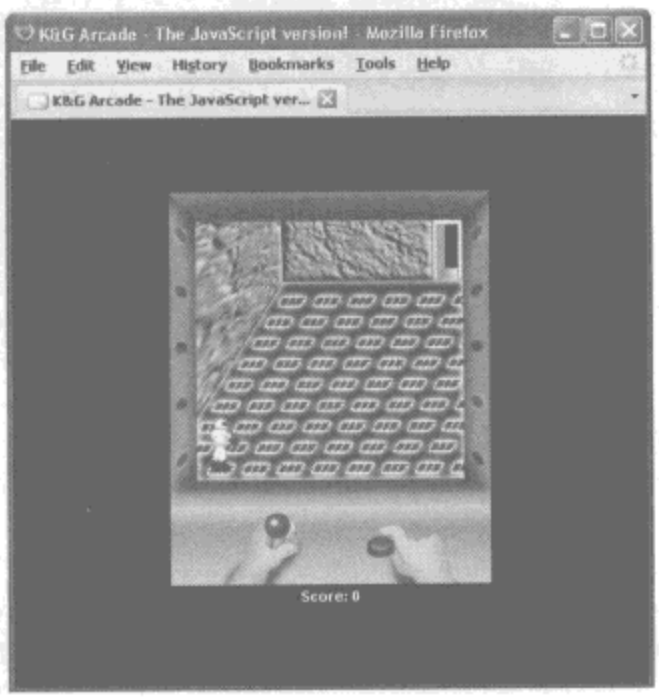


图11-4　死亡陷阱

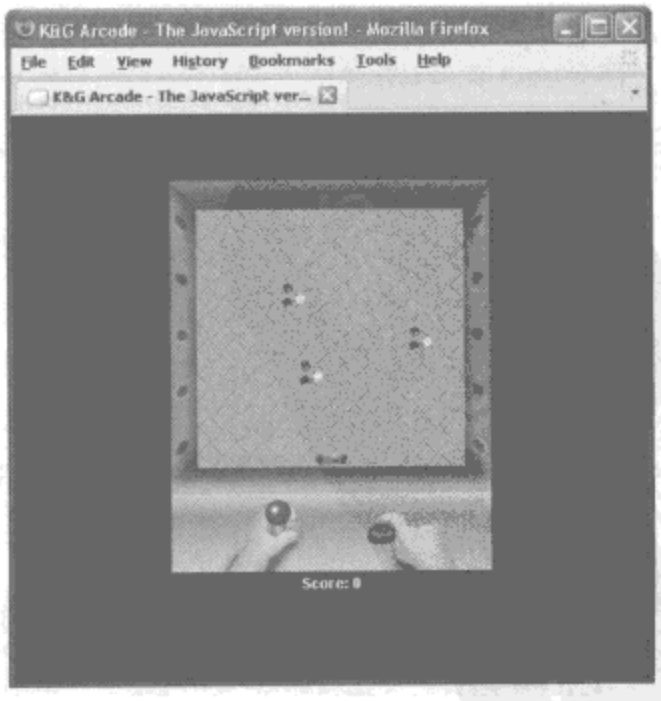


图11-5　Refluxive

到目前为止，你已经很熟悉这个游戏的样子了，让我们来看看如何实现吧。系好安全带，因为这会是一段相当长的旅程！

11.3 剖析K&G街机游戏的解决方案

照常，我们从它的目录结构开始这个项目的探险旅程，如图11-6所示。

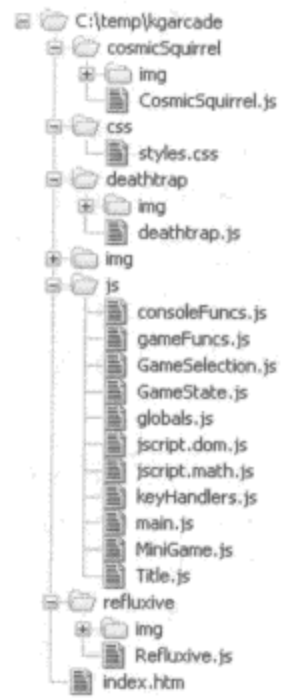


图11-6 K&G Arcade 的目录结构

首先是根目录，我们看到index.htm文件，它是载入启动应用程序的页面。它含有用户所见的那个屏幕的基本标记，并导入了所需的JavaScript。

img目录包含的并不是某个迷你游戏的所有图片，而是诸如标题画面、游戏选择画面和游戏控制画面的图片。

js目录包括我们所有的JavaScript源文件，一共11个。其中2个是我们在第3章中创建的，它们是jscript.math.js和jscript.dom.js。还有4个定义将要使用的类，它们是MiniGame.js、Title.js、GameSelection.js和GameState.js。剩下的5个包含了使用那些类的代码，它们是main.js、keyHandlers.js、globals.js、gameFuncs.js和consoleFuncs.js。

每一个迷你游戏都保存在各自的子目录中，它们的名字与迷你游戏的名字相同。在每个子目录中，都有单个.js文件，例如CosmicSquirrel.js是对应的迷你游戏的代码。子目录中还都含有一个img目录，也就是对应迷你游戏的图片所放置的地方。

现在，让我们来看看代码吧，好吗？

11.3.1 编写index.htm

当我们进入游戏时，index.htm是第一个载入的页面，它定义了整个布局。还“引入”了所有其他的在游戏中需要的源文件。让我们从页面的<head>开始看：

```
<head>

  <title>K&G Arcade - The JavaScript version!</title>

  <link rel="StyleSheet" href="css/styles.css" type="text/css">

  <script src="js/jscript.dom.js"></script>
  <script src="js/jscript.math.js"></script>
  <script src="js/gameFuncs.js"></script>
  <script src="js/consoleFuncs.js"></script>
  <script src="js/keyHandlers.js"></script>
  <script src="js/globals.js"></script>
  <script src="js/GameState.js"></script>
  <script src="js/MiniGame.js"></script>
  <script src="js/Title.js"></script>
  <script src="js/GameSelection.js"></script>
```

```
    <script src="js/main.js"></script>
    <script src="cosmicSquirrel/CosmicSquirrel.js"></script>
    <script src="deathtrap/deathtrap.js"></script>
    <script src="refluxive/refluxive.js"></script>

  </head>
```

首先看到的是链入的样式表（下一步我们就会介绍它）。然后出现的是一整批JavaScript的引入。因为代码剖析的过程将会带领我们去一个一个探究它们，所以现在要说明它们各是什么，就显得有点多余了。这里说一句：它们是使整个应用运转起来的必需品，这样说就足够了。

然后开始的是文档的主体部分，我们看到onLoad调用了一个叫做init()的JavaScript函数。这将初始化应用程序，并为我们建立好一切所需的东西。这个函数可以在main.js中找到，所以我们一会儿看看它。

在<body>的开始标签之后，是下面这部分标记：

```
<!-- The div the title screen is contained in. -->
<div id="divTitle" class="cssTitle">
  <table border="0" cellpadding="0" cellspacing="0" width="98%"
    height="100%" align="center">
    <tr>
      <td align="center" valign="middle">
        <img src="img/title.gif">
        <br/><br/>
        The JavaScript Version, v1.0
        <br/><br/>
        Ported from the original PocketPC version, presented by:
        <br/><br/>
        <img src="img/logoOmnytex.gif"><img src="img/logoCrackhead.gif">
        <br/><br/>
        Press Any Key To Play
      </td>
    </tr>
  </table>
</div>
```

这是非常直白的HTML标记，它呈现了之前在图11-1中展示的那个标题画面。当用户点击一个按键前进到游戏选择画面时，就会立刻隐藏divTitle那个 ，游戏选择画面带给我们的是接下来出现的HTML标记：

```
<!-- The div the game selection screen is contained in. -->
<div id="divGameSelection" class="cssTitleGameSelection">
  <table border="0" cellpadding="0" cellspacing="0" width="98%"
    height="100%" align="center">
    <tr>
      <td align="center" valign="middle">
        Press the LEFT and RIGHT arrow keys to cycle through the
        available games, then press SPACE to play the one you want.
        Once playing a game, press the ENTER key to return here.
        <br/><br/><br/>
        <img src="img/ssCosmicSquirrel.gif" id="ssCosmicSquirrel"
```

```
        style="display:none;">
      <img src="img/ssDeathtrap.gif" id="ssDeathtrap"
        style="display:none;">
      <img src="img/ssRefluxive.gif" id="ssRefluxive"
        style="display:none;">
      <br/>
      <div id="mgsDesc"></div>
      <br/><br/>
      To check out the full PocketPC version of K&G Arcade,
      visit the <a href="http://www.omnytex.com">Omnytex Technologies</a>
      web site.
    </td>
  </tr>
</table>
</div>
```

这又是一堆简单的、几乎是无聊的HTML。注意在中间的截屏图片，开始的时候它们是隐藏的。当用户轮转那些截屏图片，想要选择一个游戏时，每按下一次箭头键都会显示一些图片。mgsDesc 是将要展示游戏描述的地方。

当前面提到过的那个init()函数被触发时，它所做的事情之一就是把divTitle 和divGameSelection 居中。它还把那个包含真正的迷你游戏的 也居中了，也就是divMiniGame，下面可以看到：

```
<!-- The div the game is contained in. -->
<div id="divMiniGame" class="cssMiniGame">
<div id="divGameArea" class="cssGameArea">
</div>

<div id="divStatusArea" class="cssStatusArea">
  Score: 0
</div>

<!-- Game frame -->
<img src="img/gameFrame.gif" id="imgGameFrame"
  class="cssConsoleImage">

<!-- Console left, middle and right -->
<img src="img/consoleLeft.gif" id="imgConsoleLeft"
  class="cssConsoleImage">
<img src="img/consoleMiddle.gif" id="imgConsoleMiddle"
  class="cssConsoleImage">
<img src="img/consoleRight.gif" id="imgConsoleRight"
  class="cssConsoleImage">

<!-- Left hand images -->
<img src="img/leftHandNormal.gif" id="imgLeftHandNormal"
  class="cssConsoleImage">
<img src="img/leftHandUp.gif" id="imgLeftHandUp"
  class="cssConsoleImage">
```

```
<img src="img/leftHandDown.gif" id="imgLeftHandDown"
  class="cssConsoleImage">
<img src="img/leftHandLeft.gif" id="imgLeftHandLeft"
  class="cssConsoleImage">
<img src="img/leftHandRight.gif" id="imgLeftHandRight"
  class="cssConsoleImage">
<img src="img/leftHandDL.gif" id="imgLeftHandDL"
  class="cssConsoleImage">
<img src="img/leftHandDR.gif" id="imgLeftHandDR"
  class="cssConsoleImage">
<img src="img/leftHandUL.gif" id="imgLeftHandUL"
  class="cssConsoleImage">
<img src="img/leftHandUR.gif" id="imgLeftHandUR"
  class="cssConsoleImage">

<!-- Right hand images -->
<img src="img/rightHandUp.gif" id="imgRightHandUp"
  class="cssConsoleImage">
<img src="img/rightHandDown.gif" id="imgRightHandDown"
  class="cssConsoleImage">

<!-- Console light images -->
<img src="img/gameFrameLeftLight1.gif" id="imgGameFrameLeftLight1"
  class="cssConsoleImage">
<img src="img/gameFrameLeftLight2.gif" id="imgGameFrameLeftLight2"
  class="cssConsoleImage">
<img src="img/gameFrameLeftLight3.gif" id="imgGameFrameLeftLight3"
  class="cssConsoleImage">
<img src="img/gameFrameLeftLight4.gif" id="imgGameFrameLeftLight4"
  class="cssConsoleImage">
<img src="img/gameFrameLeftLight5.gif" id="imgGameFrameLeftLight5"
  class="cssConsoleImage">
<img src="img/gameFrameRightLight1.gif" id="imgGameFrameRightLight1"
  class="cssConsoleImage">
<img src="img/gameFrameRightLight2.gif" id="imgGameFrameRightLight2"
  class="cssConsoleImage">
<img src="img/gameFrameRightLight3.gif" id="imgGameFrameRightLight3"
  class="cssConsoleImage">
<img src="img/gameFrameRightLight4.gif" id="imgGameFrameRightLight4"
  class="cssConsoleImage">
<img src="img/gameFrameRightLight5.gif" id="imgGameFrameRightLight5"
  class="cssConsoleImage">

</div>
```

可能这里最重要的元素就是divGameArea了。它是迷你游戏所处的游戏控制台的画面部分。这里可能需要做一些解释。

所有这些图片，都是组成游戏控制台的一部分——操纵杆、迷你游戏周围的框架、灯光，等等。这些图片在迷你游戏之间并不更换，也就是在这里它们为什么是静态的。当玩家在某个迷你游戏中移

动时，放在操纵杆上的左手也会相应地移动。当用户点击action按钮时，右手也会按下那个按钮。每半秒钟，框架上面的灯管就随机改变。所有的这些动作效果，都是通过先隐藏要改变的那个图片，然后显示适当的新图片来完成的。[①]

例如，我们讨论一下动作按钮。开始的时候，展示的是那个按钮并没有被按下的图片——带有imgRightHandUp这个ID的，而不是按下按钮的图片——ID是imgRightHandDown的。当用户点击了那个按钮，首先隐藏起imgRightHandUp，然后再显示imgRightHandDown。这是很简单的例子，不过这也是如何让所有迷你游戏运转的方法，你很快就会看到的。

11.3.2 编写styles.css

如果没有了在styles.css中的样式表的支持，index.htm里面所有的HTML代码也都毫无价值的了。所以让我们来熟悉一下它吧。代码清单11-1中展示了完整的文件。

代码清单11-1 游戏的主样式表styles.css文件

```
/* Generic style applied to all elements. */
* {
  color             : #ffffff;
  font-family       : arial;
  font-size         : 8pt;
  font-weight       : bold;
}

/* Entire page (body element). */
.cssPage {
  background-color : #000000;
}


/* Style for div the title screen and game selection screens are */
/* contained in. */
.cssTitleGameSelection {
  border            : 1px solid #ffffff;
  position          : absolute;
  width             : 240px;
  height            : 320px;
}
/* Style for div the mini-games are contained in. */
.cssMiniGame {
  border            : 1px solid #000000;
  position          : absolute;
  width             : 240px;
  height            : 320px;
}
```

① 其实就是图形处理里面古老的双缓冲技术。——译注者

```
/* Style for the area where a mini-game takes place. */
.cssGameArea {
  position          : absolute;
  left              : 20px;
  top               : 20px;
  width             : 200px;
  height            : 200px;
  overflow          : hidden;
}

/* Style for the status area below the game console. */
.cssStatusArea {
  position          : absolute;
  left              : 0px;
  top               : 302px;
  width             : 240px;
  height            : 20px;
  text-align        : center;
}


/* Style applied to all game console images. */
.cssConsoleImage {
  position          : absolute;
  display           : none;
}
```

如果你面对这个样式表，就会发现它真的没有什么东西。首先，你看到那个通用的“覆盖所有的”选择器，在前面章节的项目中也遇到过它。它只处理字体样式，不过用一个清晰的设置就可以影响所有东西是很好的。

那之后是cssPage，它的唯一使命就是让页面有一个黑色的背景。

下个样式是cssTitleGameSelection，用于标题画面（divTitle）以及游戏选择画面（divGameSelection）。选择器的名字可能有点长，不过非常清楚地描述了它是用来做什么的，不是吗？它还通过把position属性设置为absolute，确保了这些 元素可以居中。这样式还给 元素画了个边框。最后，设置大小。注意，那个240×320的大小，并不是任意的，这是Quarter VGA (QVGA)的决定，在我们开发那个完整版的K&G 街机时，大多数PocketPC设备的默认分辨率就是这个。因为所有图片都是为那个分辨率缩放的，所以在这里的JavaScript版本也使用这个分辨率。

cssMiniGame样式应用在divMiniGame 元素上。注意它与cssTitleGameSelection样式一样，除了那个边框的颜色不同。它把边框设置为黑色，意味着在黑色背景的页面上，就看不到边框了，所以，效果就是游戏控制台将不会再有那个白色边框，但是还是会占用相同的空间，这样可以避免游戏控制台相对于游戏选择的画面发生移动或跳动。

最重要的可能就是cssGameArea样式了。迷你游戏所需要的一个东西就是剪切，就像在“宇宙松鼠”中可以看到的。换句话说，当一个对象移动到边界的时候，它应该看起来像是继续移出屏幕，而不是覆盖框架或者任何东西。为了达到这个效果，我们把overflow属性设置为hidden。这个意思就是任何位于那个 边界之外的内容都将被隐藏，或者准确地说是剪切。如你所见，实际的游戏区域是

200×200像素的，在包含它的那个框架内部，距框架左边和上边20像素的位置，这样就像我们预期的那样，完成了当对象移动到边界之外时就剪切掉的效果。

然后是cssStatusArea样式。我相信你可以猜到，这个是应用在divStatusArea 元素上的，那是游戏控制台下方显示分数的地方。通过把text-align设置为center，这个 中所有的文字都是水平居中的。

最后，看到的是cssConsoleImage样式，它应用在组成游戏控制台的所有图片上。它的主要目标也是确保图片可以被绝对定位，并且开始时是隐藏的。当我们后面看那个blit()函数里那个绝对定位的地方就清楚了。

11.3.3 编写GameState.js

我们来看看在整个代码中使用了很多次的GameState类，也是因此要先来看它。它的功能是保存游戏当前状态的信息。

图11-7展示了这个类的UML图。

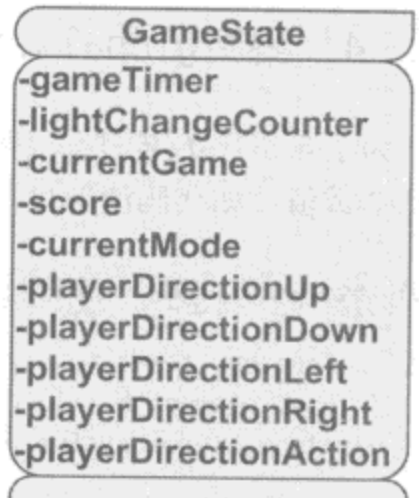


图11-7 GameState类的UML图

GameState类包含了以下这些字段。

- gameTimer：游戏中用作“心跳”的指向JavaScript计时器的引用。
- lightChangeCounter：用来判断何时更换游戏控制台边框的灯光颜色。
- currentGame：指向当前迷你游戏对象（也包括标题画面和游戏选择画面，从剩下的代码角度来看，其实也是迷你游戏）的引用。
- score：我想这个字段的用途已经很明显了。
- currentMode：判断当前是否在玩某个小游戏。
- playerDirectionXXX：5个字段（playerDirectionUp、playerDirectionDown、playerDirectionLeft、playerDirectionRight和playerAction）用来判断玩家正在向那个方向移动以及是否点击了动作按钮。

代码清单11-2完整地展示了这个类。

代码清单11-2 GameState类

```
function GameState() {

  // The main timer, 24 frames a second.
  this.gameTimer = null;

  // Count of how many frames have elapsed since the lights last changed.
  this.lightChangeCounter = null;

  // This is essentially a pointer to the current game.  Note that the term
  // "game" is a little loose here because the title screen and the game
  // selection screen are also "games" as far as the code is concerned.
  this.currentGame = new Title();
```

```
    this.score = 0;

    // Mode the game is currently in: "title", "gameSelection" or "miniGame".
    this.currentMode = null;

    // Flag variables for player movement.
    this.playerDirectionUp = false;
    this.playerDirectionDown = false;
    this.playerDirectionRight = false;
    this.playerDirectionLeft = false;
    this.playerAction = false;

} // End GameState class.
```

11.3.4　编写globals.js

为了与贯穿本书的不浪费全局命名空间的主体保持一致，在global.js文件里，你将看到只有很少的几个数值，如代码清单11-3所示。

代码清单11-3　这个应用里全局作用域内的东西并不对，但是很有用！

```
// Counter, reset to 0 to start each frame, used to set the z-index of
// each element blit()'d to the screen.
var frameZIndexCounter = 0;

// Key code constants.
var KEY_UP = 38;
var KEY_DOWN = 40;
var KEY_LEFT = 37;
var KEY_RIGHT = 39;
var KEY_SPACE = 32;
var KEY_ENTER = 13;

// Structure that stores all game state-related variables.
var gameState = null;

// This is an associative collection of all the images in the game.
// This saves us from having to go to the DOM every time to update one.
var consoleImages = new Object();
```

frameZIndexCounter变量是用来确保当图片被blit()后，有合适的Z轴次序，我们将在下一节讨论blit()。然后是一批伪常量（记住在JavaScript中，并没有真正的常量），它们定义了可以按下的各种按键。我们还可以看到gameState变量，它将是指向通篇代码中唯一的GameState对象的引用。最后，consoleImages数组用来存储指向游戏控制台的那些图片的引用，这个我们一会就会说。

11.3.5　编写main.js

Main.js基本上是K&G街机的心脏和灵魂。你将发现它使用了在其他JavaScript文件中的函数，所

以，可能你会经常读到“我们一会再看这个函数”。放心，我不是在骗你——我们过一会儿确实要接触那些函数！但是，研究main.js所关心的是理解最基本的核心。那么让我们开始吧。

1. init()函数

回忆一下index.htm中看到的，当载入页面时，作为onLoad事件的应答，调用了init()函数。那么现在就是看看关于这个函数的一切的时候了：

```
function init() {

  gameState = new GameState();

  // Get references to all existing images.  This is mainly for the console
  // images so that we don't have to go against the DOM to manipulate them.
  var imgs = document.getElementsByTagName("img");
  for (var i = 0; i < imgs.length; i++){
    consoleImages[imgs[i].id] = imgs[i];
  }

  // Center the three main layers.
  jscript.dom.layerCenterH(document.getElementById("divTitle"));
  jscript.dom.layerCenterV(document.getElementById("divTitle"));
  jscript.dom.layerCenterH(document.getElementById("divGameSelection"));
  jscript.dom.layerCenterV(document.getElementById("divGameSelection"));
  jscript.dom.layerCenterH(document.getElementById("divMiniGame"));
  jscript.dom.layerCenterV(document.getElementById("divMiniGame"));

  // Now hide what we don't need.
  document.getElementById("divGameSelection").style.display = "none";
  document.getElementById("divMiniGame").style.display = "none";

  // Hook event handlers.
  document.onkeydown = keyDownHandler;
  if (document.layers) {
    document.captureEvents(Event.KEYDOWN);
  }
  document.onkeyup = keyUpHandler;
  if (document.layers) {
    document.captureEvents(Event.KEYUP);
  }

  gameState.currentGame.init();

  gameState.gameTimer = setTimeout("mainGameLoop()", 42);

} // End init().
```

首先你看到的是实例化一个新的GameState对象。然后是个钩子，就如注释所说，得到一个指向index.htm页面中所有<img>标签的引用。我们把所有图片引用都保存在consoleImages数组中。这是因为，进行游戏开发时，（大多数时候）特别重要的一点是尽可能地提高性能。

保持帧频（FPS）

后面你会看到，通常游戏都包含一个连续的循环（这里也是一样）。每秒钟这个循环都会调用一些代码来更新几遍展示。每个这样的刷屏就叫一帧，和动画里的帧的概念一样。我相信你知道，1 s包含1000 ms。游戏的循环通常是使用每秒多少帧（fps）的计量方法，这也就是说，展示的画面每秒钟刷新多少次。为了平滑的动作效果，希望每秒钟的帧数越高越好。通常来说，我们希望它不低于24 fps左右，这是一个大概的速度，在这个速度之上，人类的眼睛就无法轻易识别出每一帧了。换句话说，如果你1 s更新10次，那么眼睛可以相当容易地跟踪每一帧，于是动画效果会很慢且很不流畅。在24 fps或更高的速度，可以使人眼认为这是流畅的持续动作，这样，就看起来就相当的平滑了。当然越高越好，不过24 fps是个边界数字。

那么，让我们做些简单的数学运算。如果1 s中有1000 ms，并且因为每一帧都会花费一定的时间来处理和展现，那么我们就可以确定为了达到目标帧频，处理一帧需要花费多少毫秒，也就可以得到每秒钟要有多少帧了。对于24 fps来说，我们把1000除以24，然后发现每帧完成全部处理花费的时间不能超过42 ms。如果某一帧用了更多的时间处理，那我们的帧频就会下降，游戏就会变得有间断，看起来不舒服。所以，不要超过42 ms就变得极为重要，这也就是为什么我们必须要考虑最优的性能。

杀手之一，或者说不只是存在于游戏编程中，而且存在于任何基于浏览器的DOM脚本编程中，就是访问的DOM中的元素。遍历DOM树、查找所需的元素然后返回一个指向它的引用是很花时间的。如果在游戏应用程序中，每帧中做很多次这样的操作，那帧频很快就会下降。优化这个问题的一个好方法是一开始就获得指向那些图片（或其他元素）的引用，然后需要时就访问预先保存的引用。从数组里获取一个是指向某个DOM元素的引用要远比直接得到那个DOM元素快得多。

一个简单的例子可以证明这个观点：

```
<html>

  <head>

    <title>DOM/Array Access Test</title>

    <script>

      function testit() {

        // Time 1000 direct DOM accesses
        var timeStart = new Date();
        for (var i = 0; i < 5000; i++) {
          var elem = document.getElementById("myDiv");
          elem.innerHTML = i;
        }
        var directDOMTime = new Date() - timeStart;
        // Time 1000 accesses via array lookup
        var a = new Array();
        a[0] = document.getElementById("myDiv");
```

```
        timeStart = new Date();
        for (var i = 0; i < 5000; i++) {
          var elem = a[0];
          elem.innerHTML = i;
        }
        var arrayTime = new Date() - timeStart;

        // Display results
        document.getElementById("myDiv").innerHTML =
          "Time for direct DOM access: " + directDOMTime + "<br>" +
          "Time for array access: " + arrayTime;

      }

    </script>

  </head>

  <body>

    <input type="button" onClick="testit();" value="Test">
    <br/>
    <div id="myDiv"></div>

  </body>

</html>
```

运行这个测试，你会发现，数组访问的方法通常都要比直接访问DOM快，虽然我惊讶地发现区别并没有我想象得那么巨大。尽管如此，每次运行的时候，差距通常是在200 ms到300 ms之间，正如我早先做的算术那样，这对于游戏编程来说，是个相当大的数量。

继续init()的学习，我们看到用来把3个主要的 居中的6行代码，这3个主要的 是divTitle，divGameSelection和divMiniGame，分别对应于标题画面、游戏选择画面和实际的迷你游戏画面。我们使用了第3章中创建的jscript.dom.layerCenterH()和jscript.dom.layerCenterV()函数。看，那些代码迟早会派上用场的，不是吗？在它们居中后，立刻隐藏游戏选择 和迷你游戏画面 。这看起来可能有些怪异。为什么不直接把那些 元素的样式设置为display:none呢？答案是，如果先隐藏了元素，居中的操作就不能正确实现，因为在元素隐藏后，浏览器就不能正确地设置那两个函数需要的某些值。

下面的4行代码是与keyDown和keyUp事件挂钩的，这样就可以触发用户处理函数。注意每个事件处理函数都需要两个语句，这是由于IE和基于Mozilla的浏览器中事件处理模型的差异造成的。在IE中仅设置document.keydown和document.keyup就足够了。但是在Firefox以及它的同类浏览器中，还需要一个附加的captureEvents()调用。通过检查是否定义了那个只有非IE浏览器才有的document.layers，我们就可以做到只在支持它的浏览器上面调用captureEvents()。正如前面章节曾经提到过的，利用对象的存在性来基于浏览器有条件地执行代码，是很好的浏览器嗅探方法。

最后，我们调用gameState.currentGame.init()。这是用来请求当前的迷你游戏初始化自身的。有

趣地是，标题画面和游戏选择画面也是被当作迷你游戏来对待的，虽然它们并不是迷你游戏。在init()中的最后一件事情是设置一个过期时间，在42 ms之后触发（又是对应想要的那24 fps），并且当触发时，调用mainGameLoop()函数。说到这儿，让我们暂停一下来说说K&G Arcade的整体结构。

2. 主游戏循环流程

图11-8是一个流程图，它描述了主游戏流的过程以及keyUp和keyDown的事件处理过程。

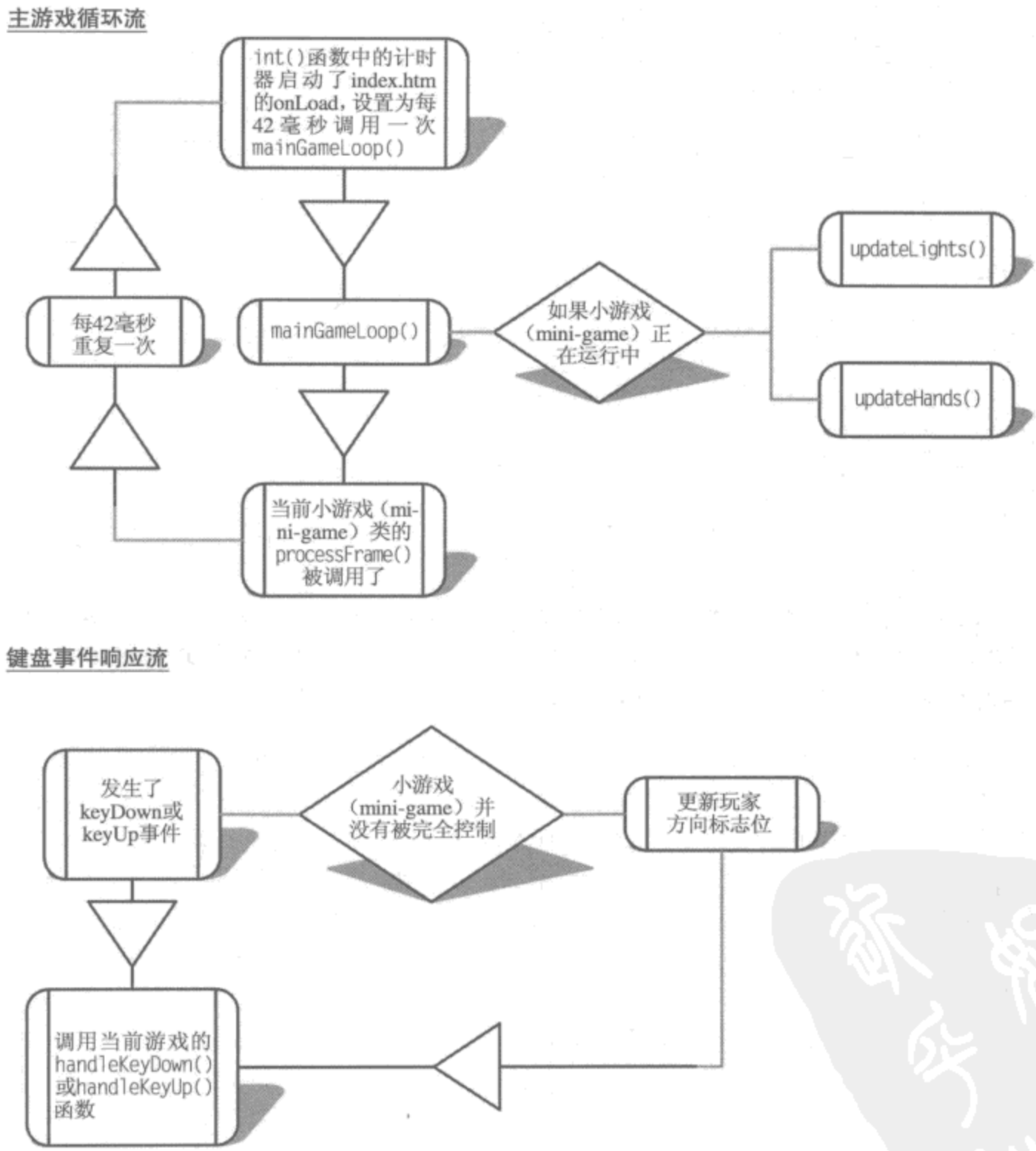


图11-8　较高抽象级别的K&G街机基本流程图

如你所见，init()中的定时器在42 ms之后触发，调用mainGameLoop()。然后mainGameLoop()更新游戏控制台框架的颜色，如果正在处理的是迷你游戏，就还要包括那双手。之后它调用那个指向

gameState.currentGame的对象上的processFrame()。一旦迷你游戏的processFrame()函数返回了，就再一次设定过期时间，并且整个处理过程重新开始。

所有的迷你游戏以及标题和游戏选择画面，实际上都是通过“扩展”MiniGame类的方式来实现一个接口。迷你游戏可以覆盖5个函数：init()、processFrame()、keyDownHandler()、keyUpHandler()和destroy()。它们代表了一个迷你游戏的生命周期。另外，还有3个字段：gameName、gameImages和fullKeyControl。

当用户决定开始玩某个小游戏时，就会调用init()函数。它的任务就是建立那个迷你游戏，主要负责载入图片，不过也会有其他的工作。

游戏的每个帧都会调用一次processFrame()函数。它负责处理所有的游戏逻辑以及更新屏幕。

KeyDownHandler()和keyUpHandler()是用来响应按下键盘和释放键盘事件的。

当用户按下了回车键要退出游戏时，就调用destroy()函数。它的主要工作就是删除在init()中载入的图片，不过也可以用来做其他事情。

gameName字段必须与相应的迷你游戏资源文件所处的目录名字相匹配（我们假设每个迷你游戏都在Web应用程序的根目录下拥有自己的子目录）。gameImages是init()中所载入图片的一个关联数组。它与我们前面曾经接触过的consoleImages数组的目的相同，用于避免可能的直接的DOM访问。最后是fullKeyControl，设置为true，意思是迷你游戏完全控制键盘事件需要处理所有事情。

在基类MiniGame中，所有这些函数的实现都是空的。因此，一个迷你游戏需要只覆盖自己感兴趣的东西。同样，除了gameName字段（这是必须在init()中设置的）之外，其他字段都有默认的值（fullKeyControl默认为false，gameImages是一个空数组）。

3. 启动迷你游戏

回到main.js，我们看看startMiniGame()函数：

```
function startMiniGame(inName) {

  // Reset generic game-related variables.
  gameState.playerDirectionUp = false;
  gameState.playerDirectionDown = false;
  gameState.playerDirectionRight = false;
  gameState.playerDirectionLeft = false;
  gameState.playerAction = false;
  gameState.score = 0;
  document.getElementById("divStatusArea").innerHTML = "Score: " +
    gameState.score;

  // Instantiate mini-game.
  if (inName == "cosmicSquirrel") {
    gameState.currentGame = new CosmicSquirrel();
    gameState.currentGame.init();
  } else if (inName == "deathtrap") {
    gameState.currentGame = new Deathtrap();
    gameState.currentGame.init();
  } else if (inName == "refluxive") {
    gameState.currentGame = new Refluxive();
    gameState.currentGame.init();
```

```
    }

    // Show the game and hide the game selection screen.  Set the mode to indicate
    // mini-game in progress, and draw the console.
    gameState.currentMode = "miniGame";
    drawConsole();
    document.getElementById("divGameSelection").style.display = "none";
    document.getElementById("divMiniGame").style.display = "block";

} // End startMiniGame().
```

当用户在游戏选择画面按下空格键来开始所选的迷你游戏时会调用startMiniGame()。函数的开始是设置gameState里面的5个字段，这5个字段标识了玩家当前的移动方向以及标识出是否按下了action按钮。它还把分数重置为0，并且显示在屏幕上。然后，基于传入的那个迷你游戏的名字，它为这个游戏实例化相应的类，并调用这个类的init()。

最后，这个函数把当前模式设置为“游戏中”，绘制控制台，隐藏游戏选择画面，并展示那个迷你游戏。记住，这个游戏的循环正在连续不断地进行，所以在紧接着的下一次循环时，就会开始了那个小游戏。

4. blit()函数——把东西显示在屏幕上

Main.js中的最后一个函数是那个到处都有的blit()函数：

```
function blit(inImage, inX, inY) {

    inImage.style.left = inX + "px";
    inImage.style.top = inY + "px";
    inImage.style.zIndex = frameZIndexCounter;
    inImage.style.display = "block";
    frameZIndexCounter++;

} // End blit().
```

blit()是用来把一个图片放置到页面上的某个指定坐标位置的。这个函数的第一个参数，是一个指向那个待放置的图片对象的引用。通过设置这个元素的left和top属性，就可以把图片放在所需的位置了。在每个帧开始的时候，都会设置zIndex属性，并重置frameZIndexCounter变量，而且在每次放置图片的时候都会给它加一。这么做的效果是，每个后继的blit()都是在所有前面blit()之上的。这通常就是blit运作的方式。如果display属性设置为block，那么图片就会在屏幕上显示，并且在它指定的位置上可见。

说明　术语blit是一个在图像处理和游戏编程中常见的词汇。简而言之，blit通常是指在屏幕上绘图。在基于浏览器的游戏中，你实际上并不是真的画一幅图像，要做的只是在某个位置放一个图片，如本例所示。（区别只是在于经典的blit是一个像素一个像素地绘图，而本例中只是把图片放在那，并不是真正地绘图。）

11.3.6　编写consoleFuncs.js

consoleFuncs.js包含处理游戏控制台的代码，其中包括在迷你游戏周围的闪光的边框、下面的手

以及状态区域。它包括3个巨大的函数。

1. drawConsole()函数

consoleFuncs.js中的第一个函数是drawConsole()：

```
function drawConsole() {

  // These are the parts of the game console that do not need to be redrawn
  // with each frame.
  blit(consoleImages["imgGameFrame"], 0, 0);
  blit(consoleImages["imgConsoleLeft"], 0, 240);
  blit(consoleImages["imgConsoleMiddle"], 108, 240);
  blit(consoleImages["imgConsoleRight"], 215, 240);

} // End drawConsole().
```

就像你可以在注释中看到的，这4个图片是从来不改变的，并不像那些会改变的图像（比如手和光影）那样会变。因此，就不需要像迷你游戏代码那样，每帧都调用drawConsole()。如果你对游戏设计完全一无所知的话，那么对你来说，谈论每个帧要呈现一些东西可能有点陌生，但是请容忍我说几句。在看main.js的时候，会解释得更详细一些。但现在，我们需要继续看看的，实际上是主处理代码中的支持函数，也就是这个文件（以及其他一些文件）所包含的。所以我们先在这儿唠叨一会儿。

2. updateLights()函数

我们下一个要看的是updateLights()，它的任务是更新游戏控制台周围的灯光，很简单。下面是它的代码：

```
function updateLights() {

  // Every half a second we are going to light some lights and restore others
  gameState.lightChangeCounter++;

  if (gameState.lightChangeCounter > 12) {

    gameState.lightChangeCounter = 0;

    // Hide the frame and lights so we start fresh.
    consoleImages["imgGameFrame"].style.display = "none";
    consoleImages["imgGameFrameLeftLight1"].style.display = "none";
    consoleImages["imgGameFrameLeftLight2"].style.display = "none";
    consoleImages["imgGameFrameLeftLight3"].style.display = "none";
    consoleImages["imgGameFrameLeftLight4"].style.display = "none";
    consoleImages["imgGameFrameLeftLight5"].style.display = "none";
    consoleImages["imgGameFrameRightLight1"].style.display = "none";
    consoleImages["imgGameFrameRightLight2"].style.display = "none";
    consoleImages["imgGameFrameRightLight3"].style.display = "none";
    consoleImages["imgGameFrameRightLight4"].style.display = "none";
    consoleImages["imgGameFrameRightLight5"].style.display = "none";
    // Draw mini-game area frame
    blit(consoleImages["imgGameFrame"], 0, 0);
```

```
// Turn each light on or off randomly.
if (jscript.math.genRandomNumber(0, 1) == 1) {
  blit(consoleImages["imgGameFrameLeftLight1"], 0, 22);
}
if (jscript.math.genRandomNumber(0, 1) == 1) {
  blit(consoleImages["imgGameFrameLeftLight2"], 0, 64);
}
if (jscript.math.genRandomNumber(0, 1) == 1) {
  blit(consoleImages["imgGameFrameLeftLight3"], 0, 107);
}
if (jscript.math.genRandomNumber(0, 1) == 1) {
  blit(consoleImages["imgGameFrameLeftLight4"], 0, 150);
}
if (jscript.math.genRandomNumber(0, 1) == 1) {
  blit(consoleImages["imgGameFrameLeftLight5"], 0, 193);
}
if (jscript.math.genRandomNumber(0, 1) == 1) {
  blit(consoleImages["imgGameFrameRightLight1"], 220, 20);
}
if (jscript.math.genRandomNumber(0, 1) == 1) {
  blit(consoleImages["imgGameFrameRightLight2"], 220, 62);
}
if (jscript.math.genRandomNumber(0, 1) == 1) {
  blit(consoleImages["imgGameFrameRightLight3"], 220, 107);
}
if (jscript.math.genRandomNumber(0, 1) == 1) {
  blit(consoleImages["imgGameFrameRightLight4"], 220, 150);
}
if (jscript.math.genRandomNumber(0, 1) == 1) {
  blit(consoleImages["imgGameFrameRightLight5"], 220, 193);
}
}

} // End updateLights().
```

首先，起码我们不想残害儿童，[①] 并不希望灯光闪烁得太厉害，每半秒一次，看起来是合理的。因为我们的目标是24 fps，所以希望在每12帧的时候更新灯光，这样有了gameState.lightChange-Counter。

一旦确定了改变灯光是安全的，我们就先把它们都隐藏了。通常，当你写游戏的时候，每一帧的开始都是先清空屏幕。因为我们这儿实际上处理的不是那个巨大的像素点阵，所以我们并没有真正的清除什么东西。但是隐藏那些图片的效果是相同的。再说一遍，出于性能考虑，我们并不需要隐藏那些不变的、或者不会被其他图片覆盖的图片。然而，对于灯光来说，它们需要被清除，就像帧一样；否则，你就会看到，灯光打开之后永远不会关闭。

所有东西都清除了之后，我们要确定是不是10盏灯中的每一个都是打开的。实现的方法是使用第

① 1997年12月，有近百日本儿童在观看流行卡通片*Pokemon*的时候发作并发症，详情参阅http://www.cnn.com/WORLD/9712/17/video.seizures.update.

3章中开发的那个jscript.math.genRandomNumber()函数。然后，使用blit处理那些打开的灯，就完成了。

3. updateHands()函数

现在只剩下一个函数了，也就是这里展示的updateHands()：

```
function updateHands() {

  // Clear all images to prepare for proper display.
  consoleImages["imgLeftHandUp"].style.display = "none";
  consoleImages["imgLeftHandDown"].style.display = "none";
  consoleImages["imgLeftHandLeft"].style.display = "none";
  consoleImages["imgLeftHandRight"].style.display = "none";
  consoleImages["imgLeftHandUL"].style.display = "none";
  consoleImages["imgLeftHandUR"].style.display = "none";
  consoleImages["imgLeftHandDL"].style.display = "none";
  consoleImages["imgLeftHandDR"].style.display = "none";
  consoleImages["imgRightHandDown"].style.display = "none";

  // Display appropriate left-hand image.
  if (gameState.playerDirectionUp && !gameState.playerDirectionDown &&
    !gameState.playerDirectionLeft && !gameState.playerDirectionRight) {
    blit(consoleImages["imgLeftHandUp"], 29, 240);
  } else if (!gameState.playerDirectionUp && gameState.playerDirectionDown &&
    !gameState.playerDirectionLeft && !gameState.playerDirectionRight) {
    blit(consoleImages["imgLeftHandDown"], 29, 240);
  } else if (!gameState.playerDirectionUp && !gameState.playerDirectionDown &&
    gameState.playerDirectionLeft && !gameState.playerDirectionRight) {
    blit(consoleImages["imgLeftHandLeft"], 29, 240);
  } else if (!gameState.playerDirectionUp && !gameState.playerDirectionDown &&
    !gameState.playerDirectionLeft && gameState.playerDirectionRight) {
    blit(consoleImages["imgLeftHandRight"], 29, 240);
  } else if (gameState.playerDirectionUp && !gameState.playerDirectionDown &&
    gameState.playerDirectionLeft && !gameState.playerDirectionRight) {
    blit(consoleImages["imgLeftHandUL"], 29, 240);
  } else if (gameState.playerDirectionUp && !gameState.playerDirectionDown &&
    !gameState.playerDirectionLeft && gameState.playerDirectionRight) {
    blit(consoleImages["imgLeftHandUR"], 29, 240);
  } else if (!gameState.playerDirectionUp && gameState.playerDirectionDown &&
    gameState.playerDirectionLeft && !gameState.playerDirectionRight) {
    blit(consoleImages["imgLeftHandDL"], 29, 240);
  } else if (!gameState.playerDirectionUp && gameState.playerDirectionDown &&
    !gameState.playerDirectionLeft && gameState.playerDirectionRight) {
    blit(consoleImages["imgLeftHandDR"], 29, 240);
  } else {
    blit(consoleImages["imgLeftHandNormal"], 29, 240);
  }

  // Display appropriate left-hand image.
  if (gameState.playerAction) {
```

```
    blit(consoleImages["imgRightHandDown"], 145, 240);
  } else {
    blit(consoleImages["imgRightHandUp"], 145, 240);
  }

} // End updateHands().
```

就像在updateLights()函数中一样，我们首先把那两个手的图片都隐藏。然后进入一个庞大的if...else代码段，来决定应该展示哪个左手图片。有4个变量可以帮助我们进行判断，就是在GameState中，玩家的方向字段：playerDirectionUp、playerDirectionDown、playerDirectionLeft和playerDirectionRight。你会注意到，我们需要考虑8个条件：4个主要的方向加4个组合方向（左上、右上、左下以及右下）。else代码段则处理用户当前没有移动的情况。

对于右手的图片也做了同样的判断，但是因为这里只有两种状态：按钮要么按下，要么没有按下。所以代码就紧凑多了。

这就是让底部那两只手工作起来的全部代码。好的，下面来讨论一下按钮状态和4个方向键的按键处理器。

11.3.7　编写keyHandlers.js

某种意思上来说，你已经见过了keyHandlers.js文件里的两个函数，因为它们是在按键按下或抬起的时候调用的函数：keyDownHandler()和keyUpHandler()。代码清单11-4显示了keyHandlers.js文件。

代码清单11-4　keyHandlers.js文件

```
/**
 * ========================================================================
 * Return the keycode of the key firing an event.
 * ========================================================================
 */
function getKeyCode(e) {

  var ev = (e) ? e : (window.event) ? window.event : null;
  if (ev) {
    return (ev.charCode) ? ev.charCode:
      ((ev.keyCode) ? ev.keyCode : ((ev.which) ? ev.which : null));
  }

} // End getKeyCode().


/**
 * ========================================================================
 * Handle key down events.
 * ========================================================================
 */
function keyDownHandler(e) {

  var keyCode = getKeyCode(e);
```

```
    if (!gameState.currentGame.fullKeyControl) {
      switch (keyCode) {
        case KEY_SPACE:
          gameState.playerAction = true;
        break;
        case KEY_UP:
          gameState.playerDirectionUp = true;
        break;
        case KEY_DOWN:
          gameState.playerDirectionDown = true;
        break;
        case KEY_LEFT:
          gameState.playerDirectionLeft = true;
        break;
        case KEY_RIGHT:
          gameState.playerDirectionRight = true;
        break;
      }
    }

    gameState.currentGame.keyDownHandler(keyCode);

} // End keyDownHandler().

/**
 * ========================================================================
 * Handle key up events.
 * ========================================================================
 */
function keyUpHandler(e) {

  var keyCode = getKeyCode(e);

  // Always handle exiting a mini-game, even if the mini-game has full control
  // over key events.
  if (keyCode == 13) {
    if (gameState.currentMode == "miniGame") {
      gameState.currentGame.destroy();
      gameState.currentGame = null;
      document.getElementById("divMiniGame").style.display = "none";
      gameState.currentGame = new GameSelection();
      gameState.currentGame.init();
    }
  }

  if (!gameState.currentGame.fullKeyControl) {
    switch (keyCode) {
      case KEY_SPACE:
        gameState.playerAction = false;
```

```
        break;
        case KEY_UP:
          gameState.playerDirectionUp = false;
        break;
        case KEY_DOWN:
          gameState.playerDirectionDown = false;
        break;
        case KEY_LEFT:
          gameState.playerDirectionLeft = false;
        break;
        case KEY_RIGHT:
          gameState.playerDirectionRight = false;
        break;
      }
    }

    gameState.currentGame.keyUpHandler(keyCode);

  } // End keyUpHandler().
```

在某个键被按下的时候，会调用keyDownHandler()。然后这个函数又会调用getKeyCode()函数。这么做的原因是，在IE里，获取按键键码的方法和在其他浏览器中不同，因为它们的事件模型从基础上就是不一样的。IE的运转方式是为事件生成一个页面作用域内的event对象，而Firefox和其他浏览器直接把对象传递给处理器函数。因此，为了对这种区别做一些封装，我们用getKeyCode()处理它，而不是用两个函数来封装。不过在实质上，这些函数做的事情就是获取传入按键的键码。第一行包含一些逻辑，最后的结果是变量ev包含相关的event对象，而不管这个应用是在什么浏览器里运行的。

if块中的代码行看上去有些复杂（实际上我对这样的三重逻辑很不满，尤其是像这样用字符串串在一起的，不过这么写的确还是比好几个嵌套的if语句更干净些），但实际上它就是通过你使用的浏览器，判断你应该用event对象的哪个属性来获取键码。在某些浏览器里是charCode，在其他一些浏览器里是keyCode，还有一些浏览器里是which。不管是哪种都会返回相关的键码，然后事件处理函数继续运行。

一旦拿到了键码，我们就可以用很简单的switch代码来判断哪个键被按下了，然后在gameState里设置合适的标志。不过，当正在进行的迷你游戏不能全面控制键字事件的时候，才会命中这段switch代码。有些迷你游戏需要这段代码，比如说Deathtrap。

最后，当前迷你游戏的keyDownHandler()函数会被调用，处理特定游戏需要执行的动作（有些游戏里，这个动作可能是空，比如宇宙松鼠）。

onKeyUp()处理函数只是略微复杂一点点。在它的代码里，我们首先检查是否按了回车。如果是，我们需要退出当前的迷你游戏，这就意味着在当前迷你游戏的类上调用destroy()，隐藏divMiniGame ，然后把游戏选择屏幕设置为当前屏幕。然后的动作实际上就和onKeyDown()一样了，区别只是游戏者方向标志设置为未设置（换句话说，就是设置为false）。

11.3.8　编写gameFuncs.js

gameFuncs.js文件包含了几个实际上相当于迷你游戏的“帮助”的函数。首先看到的是load-

GameImage()：

```
function loadGameImage(inName) {

  // Create an img object and set the relevant properties on it.
  var img = document.createElement("img");
  img.src = gameState.currentGame.gameName + "/img/" + inName + ".gif";
  img.style.position = "absolute";
  img.style.display = "none";

  // Add it to the array of images for the current game to avoid DOM access
  // later, and append it to the game area.
  gameState.currentGame.gameImages[inName] = img;
  document.getElementById("divGameArea").appendChild(img);

} // End loadGameImage().
```

回想一下每个迷你游戏，由于它们都是扩展自MiniGame基类（我们下个将会看到它）的，所以都有存储那些指向迷你游戏所使用的图片的gameImages数组。这个数组通过调用loadGameImage()函数来组建。它创建了一个新的<img>元素，把src属性设置为指定的图片（把它载入到内存中），然后把它们设置到指定位置上（poisition:absolute）。并把它作为divGameArea 元素的子节点添加到DOM上，也就是游戏控制台框架中迷你游戏所在的位置。这个函数还添加了指向当前迷你游戏类的gameImages数组的引用。

loadGameImage()的必然结果是，我们有一个destroyGameImage()函数：

```
function destroyGameImage(inName) {

  // Remove it from the DOM.
  var gameArea = document.getElementById("divGameArea");
  gameArea.removeChild(gameState.currentGame.gameImages[inName]);

  // Set element in array in null to complete the destruction.
  gameState.currentGame.gameImages[inName] = null;

} // End destroyGameImage().
```

当调用一个迷你游戏的destroy()函数时，它预期地是使用destroyGameImage()函数来销毁在init()中载入的所有图片。如果不这么做的话，就会引发内存泄露，因为每次开始游戏的时候，都会重新创建图片，但之前的图片副本仍然保留着，却没有被使用。为了销毁图片，我们需要首先把它从DOM中移除，然后把数组中指向它们的引用设置为null。这样，JavaScript的垃圾回收引擎就会负责剩下的工作了。

gameFuncs.js文件中的下一个函数是detectCollision()：

```
function detectCollision(inObj1, inObj2) {

  var left1 = inObj1.x;
  var left2 = inObj2.x;
  var right1 = left1 + inObj1.width;
  var right2 = left2 + inObj2.width;
```

```
    var top1 = inObj1.y;
    var top2 = inObj2.y;
    var bottom1 = top1 + inObj1.height;
    var bottom2 = top2 + inObj2.height;

    if (bottom1 < top2) {
      return false;
    }
    if (top1 > bottom2) {
      return false;
    }
    if (right1 < left2) {
      return false;
    }
    if (left1 > right2) {
      return false;
    }

    return true;

  } // End detectCollision().
```

大多数视频游戏，例如宇宙松鼠，都需要探测两个图片——游戏中的两个对象[通常使用术语精灵（sprite）] ——何时碰撞。比如，我们需要知道玩家的松鼠什么时候被一个小行星挤扁。有无数的碰撞探测算法，但是大多数都没法在浏览器里获得。比如，检查两个图片的每个像素相对的位置，虽然可以得到100%准确的探测效果，但是并不适合于浏览器。这里使用的方法叫做包围盒（bounding box）。它是一个非常简单的方法，基本上就是检测对象的4个角。如果一个对象的角在另一个对象的范围内，那么就是发生了碰撞。

就像图11-9的示例所说明的，每个精灵周围都有一个小方块（或是长方形）区域，叫做它的包围盒，它定义了精灵所占的领地的边界。注意图11-9中，对象1的左上角是如何进入对象2的包围盒的。这就代表一个碰撞。我们可以通过对每个对象所运行的一系列比较测试来检测碰撞。如果所有条件都不为真，那么就没有发生碰撞。例如，如果对象1的底部在对象2的顶部之上，就不可能发生任何碰撞。实际上，由于我们处理的是一个正方形或者长方形，所以只需要检查4个条件，其中任何一个为假的话，就都排除了碰撞的可能性。

这个算法并不能提供完美的结果。例如，在宇宙松鼠中，有时你会看到提示说松鼠撞到了一个对象，但它们明显没有碰到对方。这是因为，包围盒可以在对象本身并没有真实碰撞的时候发生碰撞。[①]可能只能通过像素级别的探测来解决这个问题了，但这对我们并不可行。不过，包围盒提供了一个“足够好的”近似结果，所以就先这么着吧。

最后两个函数是addToScore()和subtractFromScore()，它们都是自释其义的。注意那个subtractFromScore()必须检查分数值不低于零。有些游戏可能真的会有负值，但是我看没必要给玩家造成更大的心理损害。我认为零已经足够令人难堪了。那两个函数都是负责简单地更新gameState中的score字段和状态区域。

① 包围盒比对象大，包围嘛。——译者注

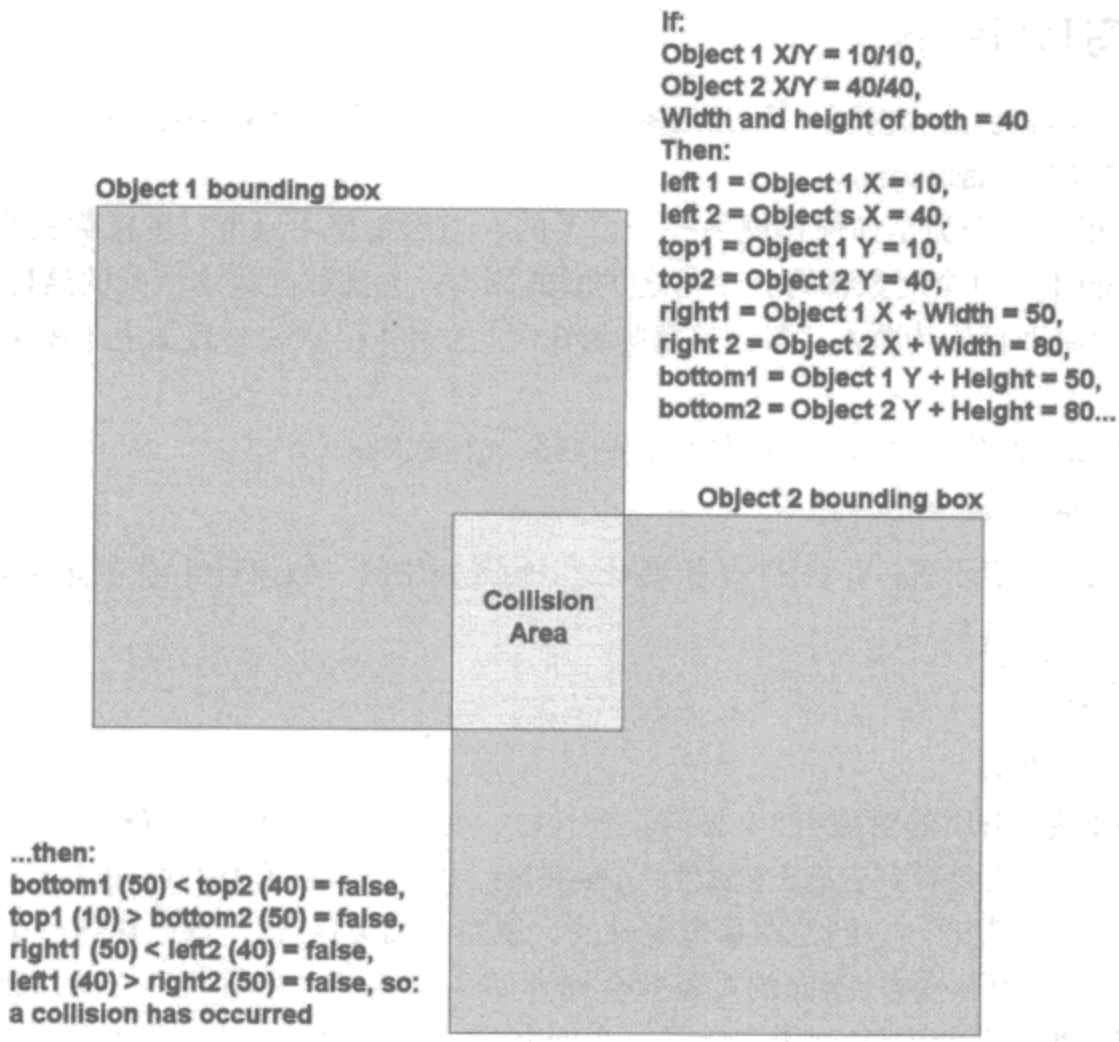


图11-9 基础的包围盒碰撞测探算法的例子

11.3.9 编写MiniGame.js

图11-10 MiniGame类的UML图

MiniGame类是所有迷你游戏以及标题画面、游戏选择画面的基类。图11-10展示了这个类的UML图。它并没有包含任何真正的可执行代码，不过某种意义上来说，它定义了所有迷你游戏都要实现的接口。迷你游戏、标题画面和游戏选择画面，本质上都继承自这个类，覆盖了它所提供的函数的默认实现方法。如果不必覆盖的时候，就使用那个默认的、什么都不做的版本。

MiniGame类中除了成员函数之外，还有3个字段。

- gameName：这个字段必须在实例化的时候由子类设定，而且它要匹配迷你游戏所在目录的名字。这个值是用来为小游戏需要载入的图片构造URL的。它在MiniGame类中的默认值是null。
- gameImage：这个数组在MiniGame中初始化为一个空的数组。这样如果迷你游戏没有载入任何图片也不会报错。不过，没有图片的迷你游戏几乎没意义，所以这也是为什么这个数组不会空太久。
- fullKeyControl：这个字段本质上是一个标志位，它决定迷你游戏是否完全受键盘事件控制，并且这些事件没有默认行为（就像前面在keyHandlers.js中看到的）。默认值是false，所以如果迷你游戏没有覆盖这个值的话，就会发生默认行为。

11.3.10　编写Title.js

我在这儿没有给Title.js也画一个UML图，是因为这张图和MiniGame类的那张看上去完全一样，因为Title类是扩展MiniGame类的。

在JavaScript里，一个类是如何扩展另外一个类的？当然是使用原型！就像你在第1章中所看到的那样，在JavaScript里，每个对象都有一个与之关联的原型。它实际上是这个类的结构原型。比如，如果你把一个类B的原型设置为类A，那么就意味着B看上去和A相似，并且加上了B自己已经定义的东西。

在Title类里面，我们在Title.js的最后看到这么一行代码：

```
Title.prototype = new MiniGame;
```

这句话概念上就相当于说：Title类是扩展MiniGame类而生的。假设Title类的定义只是下面这样：

```
function Title() { }
```

如果你这么写：

```
var t = new Title();
```

那你会发现变量t引用的对象有5个方法：init()、destroy()、processFrame()、keyUpHandler()和keyDownHandler()。还会看到它有3个属性：gameName、gameImages和fullKeyControl()。这些属性实际上都是源于MiniGame类的，它们是在那里定义的。现在，如果Title类自己包含一个init()方法（在这个例子里确实有），那么变量t指向的对象就会拥有Title里面定义的init()方法，而不是在MiniGame中定义的那个空函数。而如果Title类定义了一个叫doSomething()的函数，那么变量t指向的对象就会包含一个叫doSomething()的函数，即使MiniGame类里面没有这个函数也是如此。这些事情在类继承的世界里都是很普通的事情，我们应该觉得很正常。

Title类在代码清单11-5中显示，覆盖了MiniGame中的3个类方法：init()、destroy()和keyUpHandler()。

代码清单11-5　Title类

```
function Title() {


  /**
   * ================================================================================
   * Game initialization.
   * ================================================================================
   */
  this.init = function() {

    document.getElementById("divTitle").style.display = "block";

  } // End init().


  /**
```

```
 * ================================================================
 * Handle key up events.
 * ================================================================
 */
this.keyUpHandler = function(e) {
  gameState.currentGame.destroy();
  gameState.currentGame = null;
  gameState.currentGame = new GameSelection();
  gameState.currentGame.init();

} // End keyUpHandler().


/**
 * ================================================================
 * Destroy resources.
 * ================================================================
 */
this.destroy = function() {

  document.getElementById("divTitle").style.display = "none";

} // End destroy().
} // End Title class.


// Title class "inherits" from MiniGame class (even though, strictly speaking,
// it isn't a mini-game).
Title.prototype = new MiniGame;
```

请记住，我说过，虽然Title类显然不是一个迷你游戏，但它还是被当作迷你游戏看待。因此，在应用启动的时候，一开始先实例化一个Title类，然后在它上面调用init()。在这里，init()要做的唯一一件事就是把包含标题窗口的标记 变成可见的。然后，如果某个键被按下并弹起，就会调用keyUpHandler()。它调用gameState.currentGame字段指向的对象（就是它自己！）的destroy()方法。destroy()只是为屏幕把 再给隐藏起来。然后控制回到keyUpHandler()，它实例化一个GameSelection类的新实例，把gameState.currentGame指向它，然后在它上面调用init()。还要记得，这次，主游戏循环是用超时事件发起的。所以，下次循环开始的时候，循环里会在GameSelection实例上调用processFrame()，因此实际上会切换到该屏幕。

请注意Title类并未覆盖processFrame()函数。它不需要在那边做什么事情，所以没必要包含这个函数。MiniGame里面那个什么也不做的函数每次都会为每帧执行一次，因此就不会有什么问题了。

11.3.11 编写GameSelection.js

毫无疑问了，GameSelection.js是处理选择游戏窗口的类。它和Title屏幕类一样，当作迷你游戏

来看待。图11-11显示了它的UML图表。

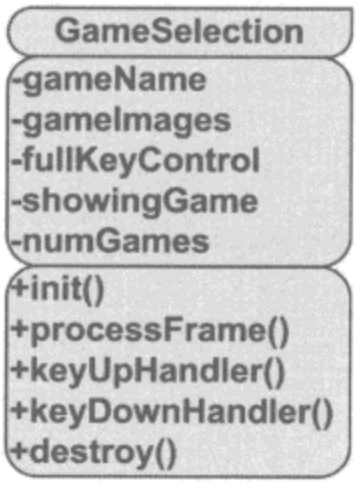


图11-11　GameSelection类的UML图

和Title类一样，如果用户在游戏选择窗口按下并释放一个键，则初始化GameSelection类，并且调用init()。这没做多少事情，不过我们还是看一眼：

```
this.init = function() {

  gameState.currentMode = null;
  document.getElementById("divGameSelection").style.display = "block";

} // End init().
```

首先，我们设置gameState.currentMode为空，表示当前没有处理迷你游戏。记住，在用户退出一个迷你游戏的时候，会创建一个新的GameSelection实例，然后再在它上面调用init()，所以这是一个设置gameState.currentMode值的好地方。然后，就只是很简单地显示游戏选择的 ，我们设置全部。

对每个帧来说，GameSelection还有些事情要做：

```
this.processFrame = function() {

  document.getElementById("ssCosmicSquirrel").style.display = "none";
  document.getElementById("ssDeathtrap").style.display = "none";
  switch (this.showingGame) {
    case 1:
      document.getElementById("ssCosmicSquirrel").style.display = "block";
      document.getElementById("mgsDesc").innerHTML =
        "In space, no one can hear a giant space squirrel buy it";
    break;
    case 2:
      document.getElementById("ssDeathtrap").style.display = "block";
      document.getElementById("mgsDesc").innerHTML =
        "Hop on the tiles to escape the chasm without getting cooked";
    break;
  }

} // End processFrame().
```

它首先把迷你游戏的截屏预览隐藏起来。因为这里图片不多，而且对小游戏而言，速度并不是很重

要的问题，所以我们没有把这些图片的引用装载到一个数组中，来避免前面讨论过的DOM查找。我们用的方法是每次都获取一个指向图片的引用。把它们隐藏起来之后，再判断一下当前正在显示的是哪个迷你游戏的预览，然后我们显示这个游戏的图片对象，同时显示相应的描述。这些就是全部逻辑了。

如你所见，GameSelection类的大部分工作都是在keyUpHandler()里进行的：

```
this.keyUpHandler = function(e) {

  switch (e) {
    case KEY_LEFT:
      this.showingGame--;
      if (this.showingGame < 1) {
        this.showingGame = this.numGames;
      }
    break;
    case KEY_RIGHT:
      this.showingGame++;
      if (this.showingGame > this.numGames) {
        this.showingGame = 1;
      }
    break;
    case KEY_SPACE:
      gameState.currentGame.destroy();
      gameState.currentGame = null;
      switch (this.showingGame) {
        case 1:
          startMiniGame("cosmicSquirrel");
        break;
        case 2:
          startMiniGame("deathtrap");
        break;
        case 3:
          startMiniGame("refluxive");
        break;
      }
    break;
  }

} // End keyUpHandler().
```

如果用户按了左箭头键或者右箭头键，我们就相应地增加或者减少showingGame字段的值。如果是减少，当减少到小于1的时候（1代表第一个迷你游戏），我们就跳转到numGames字段定义的迷你游戏上。类似的是如果增加超过了迷你游戏的数目，我们就跳回1。这样，无论我们有多少迷你游戏，就都能在到达列表的两头时回到另外一头。如果用户按下了空格键，那就可以开始当前选中的迷你游戏了。我们通过调用startMiniGame()函数，给它传递选中的游戏名来启动迷你游戏。你前面已经看到了，这个函数负责所有启动游戏的细节。

最后，destroy()函数很简单：

```
this.destroy = function() {

  document.getElementById("divGameSelection").style.display = "none";

} // End destroy().
```

我们只是隐藏包含游戏选择屏幕的 ，然后这部分的事情就做完了！

11.3.12　编写CosmicSquirrel.js

好，我们现在来看看真正好玩的东西！前面已经完成了所有基础结构的代码。但是今天的最后，我们会把时间都花在迷你游戏上！这里是大多数动作所在的地方，以及所有非通用代码所在的地方。在查阅代码之前，先让我们了解一下它的结构。

在图11-12里，可以看到CosmicSquirrel类的自身。你会发现它扩展了MiniGame类，因此，它输出我们现在已经熟知的5个接口和3个常用属性。你还会发现它包含一些唯一的字段：ObstacleDesc、PlayerDesc、AcornDesc、player、acorn和obstacles。前3个自己也是类，定义在CosmicSquirrel类内部。类似Java里的内联类。它们其实也可以很容易地在CosmicSquirrel类外边定义，但是我觉得它们只被CosmicSquirrel用到，所以定义在其内部更合理些，这样可以把关系搞得更加明确。另外3个字段是3个内联类的实例（实际上player和acorn是实例，而obstacles是ObstacleDesc对象的数组）。

1. 设置Obstacle、Player和Acorn

图11-13显示了ObstacleDesc类的UML图表。这个类的名字是“障碍描述符”的缩写，用于代表屏幕上的障碍物，这些障碍物可能是外星人、飞船、流星或者彗星。每个这样的对象都会定义对象的ID、它的X/Y轴位置、在屏幕上代表它的图片的宽度和高度、它的移动速度和方向。

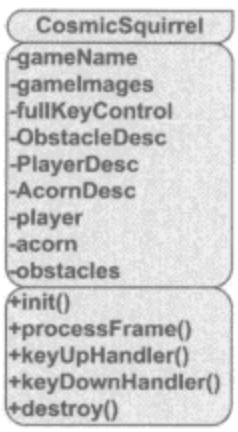


图11-12　CosmicSquirrel类的UML图表

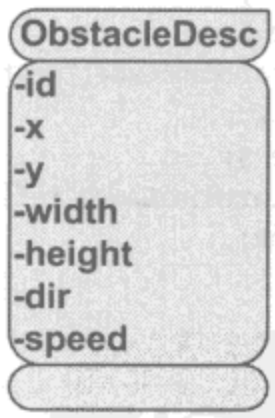


图11-13　ObstacleDesc类的UML图表

ObstacleDesc类的代码如下：

```
function ObstacleDesc(inID, inX, inY, inDir, inSpeed, inWidth, inHeight) {
  this.id = inID;
  this.x = inX;
  this.y = inY;
  this.width = inWidth;
  this.height = inHeight;
  this.dir = inDir;
  this.speed = inSpeed;
} // End ObstacleDesc class.
```

实际上，它只是一个数据结构。请注意它的参数列表（实际上是它的构造器，因为这是在JavaScript里把任何函数定义为类的效果）允许在创建实例的时候设置所有参数。你很快就会看到这些行为了。

图 11-14 PlayerDesc 类的UML图表

PlayerDesc类的UML图表可以在图11-14看到，描述了游戏玩家，实际上就是宇宙松鼠。它包含玩家当前的X/Y坐标以及松鼠图片的宽度和高度。它还定义了一个在玩家死亡或者拿到松果的时候会调用的方法，这个方法会把自己重置为起始位置。

PlayerDesc类的代码和ObstacleDesc类的代码几乎一样简单，只多了一个reset()方法：

```
function PlayerDesc() {
  this.x = null;
  this.y = null;
  this.width = 18;
  this.height = 18;
  this.reset = function() {
    this.x = 91;
    this.y = 180;
  }
  this.reset();
} // End PlayerDesc class.
```

重置玩家的动作，只是将其设置回起点位置。请注意最后的reset()调用。这也是实例化此类要执行的东西——初始化。

往下看，看看AcornDesc类，它在图11-15中显示。当然，这个类描述的是松果。

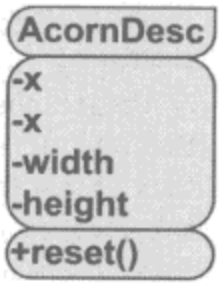


图11-15 AcornDesc类的UML图表

请注意AcornDesc类和PlayerDesc类结构相同，也定义了一样的reset()方法，在玩家到达松果的位置后，它会用jscript.math.genRandomNumber()函数把松果随机地放在屏幕的某个位置，如下所示：

```
function AcornDesc() {
  this.x = null;
  this.y = null;
  this.width = 18;
  this.height = 18;
  this.reset = function() {
    this.x = jscript.math.genRandomNumber(1, 180)
    this.y = 2;
  }
  this.reset();
} // End AcornDesc class.
```

在CosmicSquirrel类里跟在这3个类后面的代码是如下3行：

```
this.player = new PlayerDesc();
this.acorn = new AcornDesc();
this.obstacles = new Array();
```

因为只有一个玩家和一个松果，所以在初始化了CosmicSquirrel之后立刻初始化这3个类。我们还

要填充一个空的障碍物数组，这部分工作是在CosmicSquirrel实例里调用init()的时候做的。

2. 开始游戏

下面让我们来看看init()：

```
this.init = function() {

  // Configure base game parameters.
  this.gameName = "cosmicSquirrel";

  // Load all the images required for this game.
  loadGameImage("background");
  loadGameImage("acorn");
  loadGameImage("squirrelUp");
  loadGameImage("squirrelDown");
  loadGameImage("squirrelLeft");
  loadGameImage("squirrelRight");
  loadGameImage("squirrelStill");
  loadGameImage("alien1");
  loadGameImage("alien2");
  loadGameImage("ship1");
  loadGameImage("ship2");
  loadGameImage("asteroid1");
  loadGameImage("asteroid2");
  loadGameImage("comet1");
  loadGameImage("comet2");

  // Create obstacle descriptors and add to array.
  this.obstacles.push(new ObstacleDesc("alien1", 170, 30, "R", 5, 24, 24));
  this.obstacles.push(new ObstacleDesc("alien2", 80, 30, "R", 5, 24, 24));
  this.obstacles.push(new ObstacleDesc("ship1", 110, 60, "L", 2, 32, 24));
  this.obstacles.push(new ObstacleDesc("ship2", 10, 60, "L", 2, 32, 24));
  this.obstacles.push(new ObstacleDesc("asteroid1", 80, 90, "R", 4, 32, 32));
  this.obstacles.push(new ObstacleDesc("asteroid2", 140, 90, "R", 4, 32, 32));
  this.obstacles.push(new ObstacleDesc("comet1", 240, 130, "L", 3, 64, 14));
  this.obstacles.push(new ObstacleDesc("comet2", 70, 130, "L", 3, 64, 14));

} // End init().
```

首先，设置gameName字段，这是在所有迷你游戏中都要做的。注意这里设置的值是“cosmicSquirrel”，它要与这个代码所在的那个目录相匹配。这样在调用loadGameImage()函数时就没有冲突了，它会使用这个值来构造将被载入的图片的URL。

说说loadGameImage()，后面是一批它的调用。就像你之前看到的，它们每行都装载指定名字的图片，创建一个<img>标签，然后把标签添加到gameImages数组中，这个数组可以在MiniGame基类中找到。

最后，我们还有8行代码，它们是负责创建那些用户必须躲避的障碍物的。每一个障碍物都是一个ObstacleDesc实例，这里你可以看到，我之前提到过的构造函数参数是在哪出现的。我们简单地实例化了一个ObstacleDesc实例，传入一些适当的参数，并把那个对象压入到obstacle数组中。就好了！

3. 处理动作中的一帧

现在我们进入processFrame()函数，这里我要提醒你的是，它会在一秒钟内调用24次，每一帧调用一次。我们看到这里发生的第一件事就是隐藏这个迷你游戏中所用到的所有图片：

```
for (img in this.gameImages) {
  this.gameImages[img].style.display = "none";
}
```

你可以看到，这里使用了gameImages数组，而不是直接的DOM访问。这有利于提高速度！

接下来是这两行代码：

```
// Blit background.
blit(this.gameImages["background"], 0, 0);

// Blit acorn.
blit(this.gameImages["acorn"], this.acorn.x, this.acorn.y);
```

回想一下那个用来把图片放到屏幕中指定位置的blit()函数。这里，我们首先把背景图片放置在游戏区域上，效果是用这个背景填充整个游戏区域。然后，使用存储在acorn变量所指向的AcornDesc实例中的值，把橡树果放在它当前的位置。

然后，对那些障碍物做同样的操作。不过，由于ObstacleDesc对象是存在数组中的，我们需要遍历那个数组，并对每个元素使用blit()，就像这样：

```
for (i = 0; i < this.obstacles.length; i++) {
  var obstacle = this.obstacles[i];
  blit(this.gameImages[obstacle.id], obstacle.x, obstacle.y);
}
```

那么，这时，剩下的唯一一件事情就是显示那个松鼠了；否则，游戏就可能会非常无聊。（我的意思是，看着那些障碍物来回移动是挺酷的，但这可不是游戏！）所以，让我们还是把那只松鼠扔到屏幕上吧：

```
if (gameState.playerDirectionUp) {
  blit(this.gameImages["squirrelUp"], this.player.x, this.player.y);
} else if (gameState.playerDirectionDown) {
  blit(this.gameImages["squirrelDown"], this.player.x, this.player.y);
} else if (gameState.playerDirectionLeft) {
  blit(this.gameImages["squirrelLeft"], this.player.x, this.player.y);
} else if (gameState.playerDirectionRight) {
  blit(this.gameImages["squirrelRight"], this.player.x, this.player.y);
} else {
  blit(this.gameImages["squirrelStill"], this.player.x, this.player.y);
}
```

这里还有一些需要做的事情，因为玩家往哪个方向移动决定了我们要展示的松鼠的样子。所以，就有一系列的if语句询问GameState对象中的4个标志位，获取松鼠正向哪个方向移动的信息，然后blit()适当的图片。如果玩家并没有做任何移动，我们就展示那个默认的脸朝上的松鼠图片。

现在所有的东西实际上都已经画在屏幕上了，我们需要处理这帧中的游戏逻辑。第一步是移动障碍物。由于无论玩家做什么，这个动作都会毫不衰减地持续进行，所以没有检查条件。我们只要更新它们的位置，那些改变就会在下一帧呈现时反映出来的。下面是负责移动的代码：

```
for (i = 0; i < this.obstacles.length; i++) {
  var obstacle = this.obstacles[i];
  if (obstacle.dir == "L") {
    obstacle.x = obstacle.x - obstacle.speed;
  }
  if (obstacle.dir == "R") {
    obstacle.x = obstacle.x + obstacle.speed;
  }
  // Bounds checks (comets handled differently because of their size).
  if (obstacle.id.indexOf("comet") != -1) {
    if (obstacle.x < -40) {
      obstacle.x = 240;
    }
  } else {
    if (obstacle.x < -70) {
      obstacle.x = 240;
    }
  }
  if (obstacle.x > 240) {
    obstacle.x = -40;
  }
}
```

我们检查每个对象中与那个障碍物相关联的ObstacleDesc对象dir属性的值，来确定它正向哪个方向移动，然后使用speed属性作为改变值，相应的更新它的x位置。下一步，我们还做同样的检查，这样当一个障碍物以任何方向完全移出屏幕时，我们就重置它的x位置，让它出现在屏幕的另外一边。注意，由于彗星要比其他障碍物更长，我们需要检查的那个值与其他障碍物使用的值不同，因此就有了相应的分支逻辑。

下面的一段代码是关注于玩家移动的。

```
if (gameState.playerDirectionUp) {
  this.player.y = this.player.y - 2;
  if (this.player.y < 2) {
    this.player.y = 2;
  }
}
if (gameState.playerDirectionDown) {
  this.player.y = this.player.y + 2;
  if (this.player.y > 180) {
    this.player.y = 180;
  }
}
if (gameState.playerDirectionRight) {
  this.player.x = this.player.x + 2;
  if (this.player.x > 180) {
    this.player.x = 180;
  }
}
if (gameState.playerDirectionLeft) {
```

```
    this.player.x = this.player.x - 2;
    if (this.player.x < 2) {
      this.player.x = 2;
    }
  }
```

取决于玩家当前正向哪个方向移动，我们把x或y的位置加2或减2。然后应用一些边界检测，来确认在任何方向上用户都没有移出迷你游戏的屏幕。注意，使用这个逻辑，玩家可以在4个主要方向和4个组合方向上移动。比如，如果gameState.playerDirectionUp和gameState.playerDirection Right都是真的，那么4个if里面就会有两个被执行，这样松鼠就会沿着对角线方向移动。这是完全可以接受的，可以让玩家觉得比只有4个移动方向的时候好很多。

信不信由你，我们几乎快完成游戏代码了！在processFrame()里面只剩下两个任务了。首先，我们判断玩家是否拿到了橡果，于是就有了下面的代码：

```
if (detectCollision(this.player, this.acorn)){
  this.player.reset();
  this.acorn.reset();
  addToScore(50);
}
```

我们已经看了detectCollision()函数，所以这里就不重复了。如果它返回true，那么就调用那个player字段所引用的PlayerDesc实例上的reset()，它把玩家返回到起始位置。然后我们对橡树果做同样的事情，随机地把它放置到屏幕顶部的某个地方。最后，给玩家的分值加50点。

同样，我们还需要为那8个障碍物做碰撞检测，代码也是几乎相同的：

```
for (i = 0; i <  this.obstacles.length; i++) {
  if (detectCollision(this.player, this.obstacles[i])){
    this.player.reset();
    this.acorn.reset();
    subtractFromScore(25);
  }
}
```

我们为每个障碍物调用detectCollision()。如果探测到一个碰撞，就做跟橡树果发生碰撞时相同的重置处理，不过这次要从分值中减去25。所减掉的分值要比吃橡树果得到的分数少是件很好的事情。（如果反过来的话就会显得有点虐待了！）我打赌你以前玩过那种看起来不合理的记分方法的游戏，我知道我确实从中备受挫折，所以我希望这个游戏可以更好一点！

4. 清理

没有了processFrame()挡路，现在只剩下一个任务就可以完成“宇宙松鼠”了，那就是当游戏结束时的清理工作。这是通过实现destroy()函数来完成的，就像这样：

```
this.destroy = function() {

  destroyGameImage("background");
  destroyGameImage("acorn");
  destroyGameImage("squirrelUp");
  destroyGameImage("squirrelDown");
  destroyGameImage("squirrelLeft");
```

```
    destroyGameImage("squirrelRight");
    destroyGameImage("squirrelStill");
    destroyGameImage("alien1");
    destroyGameImage("alien2");
    destroyGameImage("ship1");
    destroyGameImage("ship2");
    destroyGameImage("asteroid1");
    destroyGameImage("asteroid2");
    destroyGameImage("comet1");
    destroyGameImage("comet2");

} // End destroy().
```

init()方法中每个对loadGameImage()的调用都对应了destroy()方法中的一个相应destroyGameImage()调用。这些调用把<img>元素从DOM中移除，并把数组中保存了指向那个元素的引用设置为空，也就是把图片对象标记为通过垃圾回收器删除。不需要再做什么其他的来完成清理工作了，所以说这是个短小精悍的方法。

5. 继承基类

在CosmicSquirrel.js的最后，你可以看到一行不可思议的代码，它实现了继承。这行代码使CosmicSquirrel类不用明确地修改代码就拥有一些函数，这些函数又是游戏剩下的代码需要的，比如keyUpHandler()和keyDownHandler()。这行代码就是：

```
CosmicSquirrel.prototype = new MiniGame;
```

使用这一行代码，我们确保之前看到的那些“铺路”代码（这些代码会在当前的游戏的各个生命周期发生的事件上调用当前迷你游戏类的方法）可用，因为它保证了CosmicSquirrel类（当然了，假设没有哪个类破坏了这个基类!）符合铺路代码预期的接口。

不管你是否相信，这就是这个迷你游戏背后的全部代码!

11.3.13　编写Deathtrap.js

Deathtrap游戏比宇宙松鼠稍微复杂一点。它有大概两倍的代码量，不过相当一部分都是非常普通的、重复性的工作。我不会把它们全部都列出来，但是绝对为你展示了足够感受其功能的部分。

首先，让我们也看看Deathtrap类本身的UML图，如图11-16所示。

1. 设置游戏玩家

就像在“宇宙松鼠”中一样，我们也找到了一个PlayerDesc类，它用来描述玩家特征。就如你可以在图11-17例展示的UML图中所看到的那样，这里的东西稍微多些。我们仍然有玩家的X/Y位置，不过这次还储存了之前的X/Y位置，不久你就会看到这是为什么。另外还有一对X/Y位置，它们用来定义玩家所在的是那个瓦片。这与字面上的屏幕X/Y位置不同，并且这两个信息都是使游戏正确工作所必须的。最后，我们存储了玩家的当前状态：他是活着、死了，还是已经赢了这个游戏。

在PlayerDesc类中也有reset()方法，它的目的与“宇宙松鼠”中的相同。当玩家死了或者赢了的时候，就会调用它。下面是这个迷你游戏的PlayerDesc类的代码：

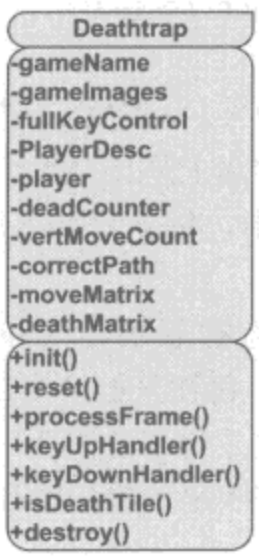


图11-16 Deathtrap类的UML图

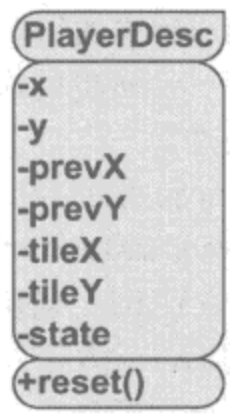


图11-17 PlayerDesc类的UML图

```
function PlayerDesc() {
  this.x = null;
  this.y = null;
  this.prevX = null;
  this.prevY = null;
  this.tileX = null;
  this.tileY = null;
  // State: A=Alive, D=Dead, W=Won
  this.state = null;
  this.reset = function() {
    this.x = 10;
    this.y = 152;
    this.prevX = 0;
    this.prevY = 0;
    this.tileX = 1;
    this.tileY = 8;
    this.state = "A";
  }
  this.reset();
} // End PlayerDesc class.
```

这个内部类定义之后，是Deathtrap类的4个字段：

```
this.player = new PlayerDesc();
this.deadCounter = null;
this.vertMoveCount = null;
this.correctPath = null;
this.regenPath = true;
```

这些字段的用途如下所述。

- player：描述玩家的PlayerDesc实例。
- deadCounter：当玩家死了的时候，用来判断应该过多长时间才可以攻击他。
- vertMoveCount：在玩家从一个瓦片移动到另一个瓦片时使用。
- correctPath：定义10个可能的正确路径中哪个是合法的。

- regenPath：标志位，判断调用reset()时是否要选择一个新的correctPath。

2. 构造死亡矩阵

下一段代码是deathMatrix。deathMatrix是一个多维数组（10×2×2），它展示了玩家必须通过的瓦片格子。对于第一维中的每个元素，都有一个2×2的数组，所以实际上是10个数组。这10个数组中的每一个都定义了一个可能的通过瓦片的正确路径。在2×2的数组中，每个元素不是0就是1，0就代表那是个安全的瓦片。

当调用Deathtrap类的reset()方法时，如果regenPath表示为被设置为true的话，就使用jscript.math.genRandomNumber()函数来从10个路径中选一个。因此，每次开始游戏或者玩家获胜时，都会选择新的correctPath。当玩家死了的时候，就不会选择了。这也是为了游戏公平。如果每次玩家失败都重置路径的话，就太让人灰心了，因为他无法通过反复测试来找出正确的路径，所以每次只能碰运气，这样就不好玩了！

这里是如何定义deathMatrix的一个例子（前两个元素），只是为了完整性而已：

```
this.deathMatrix = new Array(10);
this.deathMatrix[0] = new Array(
  [ 1, 1, 1, 1, 1, 0, 0, 0, 0 ],
  [ 1, 0, 0, 0, 0, 0, 0, 0, 0 ],
  [ 1, 1, 1, 1, 1, 1, 0, 0, 0 ],
  [ 1, 1, 0, 0, 0, 1, 0, 0, 0 ],
  [ 1, 1, 0, 0, 0, 1, 0, 0, 0 ],
  [ 1, 1, 0, 1, 1, 1, 0, 0, 0 ],
  [ 0, 0, 0, 1, 0, 0, 0, 0, 0 ],
  [ 0, 1, 1, 1, 0, 0, 0, 0, 0 ]
);
this.deathMatrix[1] = [
  [ 0, 0, 0, 0, 1, 0, 0, 0, 0 ],
  [ 0, 0, 0, 0, 1, 0, 0, 0, 0 ],
  [ 1, 1, 1, 1, 1, 1, 0, 0, 0 ],
  [ 1, 0, 0, 0, 1, 1, 0, 0, 0 ],
  [ 1, 0, 0, 0, 1, 0, 0, 0, 0 ],
  [ 1, 1, 1, 0, 1, 0, 0, 0, 0 ],
  [ 0, 0, 1, 0, 1, 1, 0, 0, 0 ],
  [ 0, 1, 1, 0, 0, 0, 0, 0, 0 ],
  [ 0, 1, 1, 0, 0, 0, 0, 0, 0 ]
];
```

3. 构造移动矩阵

Deathtrap类中另外一个需要讨论的就是moveMatrix。moveMatrix也是一个多维数组，10×10的大小，其中每个元素都代表一个瓦片。这个数组的目的就是判断玩家可以从某个给定的瓦片上向哪个方向移动。注意，在屏幕上，有些瓦片是不完整的。这是有必要的，因为我们的房间是三角形的。那些残缺的瓦片，应该是玩家不能移到的地方。对于这个数组中的每个元素，都有一个字符串值。这个字符串包括一个或多个U、D、L和R。例如，如果某个特定的瓦片的值为“ULR”，这个意思就是说，玩家可以从这个瓦片向上、向左或向右移动，但不能向下移动。瓦片的那个字符串的值也可以就只是空白的，没有任何一个字母，也就是说它永远不能到达，因此在它上面的任何移动都是非法的。

下面是定义moveMatrix的代码：

```
this.moveMatrix = new Array(10);
this.moveMatrix[0] = [ "RD", "RDL", "RDL", "RDL", "UDL",
  " ", " ", " ", " " ];
this.moveMatrix[1] = [ "URD", "URDL", "URDL", "URDL", "UDL",
  " ", " ", " ", " " ];
this.moveMatrix[2] = [ "URD", "URDL", "URDL", "URDL", "URDL",
  "DL", " ", " ", " " ];
this.moveMatrix[3] = [ "URD", "URDL", "URDL", "URDL", "URDL",
  "UDL", " ", " ", " " ];
this.moveMatrix[4] = [ "URD", "URDL", "URDL", "URDL", "URDL",
  "URDL", "DL", " ", " " ];
this.moveMatrix[5] = [ "URD", "URDL", "URDL", "URDL", "URDL",
  "URDL", "UDL", " ", " " ];
this.moveMatrix[6] = [ "UR", "URDL", "URDL", "URDL", "URDL",
  "URDL", "UDL", " ", " " ];
this.moveMatrix[7] = [ " ", "URD", "URDL", "URDL", "URDL",
  "URDL", "URDL", "DL", " " ];
this.moveMatrix[8] = [ " ", "UR", "URL", "URL", "URL",
  "URL", "URL", "UL", " " ];
```

4. 开始游戏

接下来是init()方法：

```
  this.init = function() {
      // Configure base game parameters.
      this.gameName = "deathtrap";
      this.fullKeyControl = true;

      // Reset the game state.
      this.reset();

      // Load all the images required for this game.
      loadGameImage("background");
      loadGameImage("playerDieing");
      loadGameImage("playerJumping");
      loadGameImage("playerStanding");

  } // End init().
```

设定了迷你游戏的名称之后，要注意把fullKeyControl设置为true。在Deathtrap中，我们需要处理的键盘事件与在“宇宙松鼠”中的略微不同，并且我们前面看到的keyHandler.js里的默认处理函数和这个游戏里会发生的动作有冲突。因此，这个迷你游戏需要自己来处理键盘事件，而并不是把它交给默认的代码。

在这之后是调用reset()，接着是一系列的loadGameImage()调用。（看这个游戏实际所需的图片多么少！）reset()方法真的并没有太多要做的：

```
  this.reset = function() {
```

```
    this.player.reset();
    this.deadCounter = 0;
    this.vertMoveCount = 0;
    if (this.regenPath) {
      this.correctPath = jscript.math.genRandomNumber(0, 9)
    }
    this.regenPath = false;
  } // End reset().
```

首先通过调用player.reset()重置了玩家的初始位置。然后，把deadCounter和vertMoveCount设置为零。然后我们检查regenPath是否为true，如果是true，就选一个新的通过瓦片的当前途径。最后，把regenPath设置为false，这样下一次调用reset()的时候，除非是玩家获胜，否则我们不会选择一个新的通过瓦片的正确路径。

5. 处理游戏玩家状态：赢，死或者活着

继续看processFrame()方法，我们首先遇到与之前在“宇宙松鼠”中看到的相同的屏幕清理循环，它隐藏了gameImages数组中所有的图片。然后blit()了背景。

下面是一个相当大的switch代码快。它是在玩家状态上的选择。第一个情况是，如果玩家获胜了：

```
case "W":
  addToScore(1000);
  this.regenPath = true;
  this.reset();
break;
```

只是加上1000分（我是很慷慨的!），把regenPath设置为true，这样就会选择一个新的通过瓦片的正确路径了，调用reset()使玩家回到起始点，就好了。

下一个情况时，如果玩家死了：

```
case "D":
  blit(this.gameImages["playerDieing"], this.player.x, this.player.y);
  this.deadCounter++;
  if (this.deadCounter > 48) {
    this.reset();
  }
break;
```

在这种情况下，在玩家的当前位置blit()玩家死掉的图片。那个图片是玩家触电而死的动画GIF。我们希望用两秒钟来展示它，也就是48帧（24 fps）。这里就用到了deadCounter。它用来记录已经过了多少帧了。当超过48时，就调用reset()。注意，这时regenPath会被设置为false，这样相同的正确路径依然有效。

现在我们要看的情况是如果玩家活着。这是完成大部分工作的地方。先说第一件事——让我们把游戏玩家放到屏幕上！

```
if (gameState.playerDirectionUp || gameState.playerDirectionDown ||
  gameState.playerDirectionLeft || gameState.playerDirectionRight) {
  blit(this.gameImages["playerJumping"], this.player.x, this.player.y);
} else {
  blit(this.gameImages["playerStanding"], this.player.x, this.player.y);
}
```

玩家跳起时与他站在那里所需要显示的图片是不同的。

下面是4个if代码段，每一个都是可能的移动方向。它们非常相似，所以我们就只看第一个，也就是玩家向上移动的情况：

```
if (gameState.playerDirectionUp) {
  // If movement is done, finish up
  if (this.player.y <= (this.player.prevY - 16)) {
    this.vertMoveCount  = 0;
    this.player.x = this.player.prevX + 10;
    this.player.y = this.player.prevY - 16;
    gameState.playerDirectionUp = false;
    if (this.isDeathTile()) {
      this.player.state = "D";
    }
  } else { // Otherwise, move the player
    this.player.y = this.player.y - 3;
    this.vertMoveCount++;
    if (this.vertMoveCount > 1) {
      this.vertMoveCount = 0;
      this.player.x = this.player.x + 3;
    }
  }
}
```

这个情况（以及向下移动的情况），实际上比左和右更复杂一点，因为水平和竖直的移动都会涉及到。这是由于瓦片处在相对于其他瓦片的对角的位置。所以，首先要确定是否从前一个位置移动了足够的距离，前一个位置就是当它开始移动时所处的地方。如果没有（else分支），玩家就水平移动3个像素，并且每水平移动6个像素就竖直移动3个像素（也就是说，玩家每帧移动一次，每两帧竖直移动一次）。当玩家最终移动了足够的距离时（if分支），当前位置就被设定为之前的位置加上相应的水平位移和竖直位移。这么做是因为对角线移动实际上要求细化的移动是精确的，但在使用整数的时候，总是有些误差。所以为了使玩家最终到达正确的位置，最后的值是基于前面的值的。

最后，是一个对isDeathTile()的调用。这个函数判断了玩家当前所在的那个瓦片是否带电。下面是isDeathTile()的代码：

```
this.isDeathTile = function() {

  if (
    this.deathMatrix[this.correctPath][this.player.tileY][this.player.tileX]
    == 0) {
    return true;
  } else {
    return false;
  }

} // End isDeathTile().
```

把correntPath值作为第一维，玩家的**X/Y**位置作为第二维和第三维值来查找deathMatrix。当值为零时，它就是个带电的瓦片；在这种情况下，玩家的state值被改成“D”，来标识他已经死了。

请再看一下其他3个移动方向的情况。你会发现它们与第一个向上移动的代码非常类似，不过你自己去找出不同绝对是一个好主意。

6. 处理游戏玩家键盘事件

然后是keyDownHandler()函数：

```
this.keyDownHandler = function(inKeyCode) {

  // Although the right hand action button does nothing in this game,
  // it looks like things are broken if it doesn't press, so let's let
  // it be pressed, just to keep up appearances!
  if (inKeyCode == KEY_SPACE) {
    gameState.playerAction = true;
  }

} // End keyDownHandler().
```

请记住，这个迷你游戏要对键盘进行完全的控制，所以没有什么是自动的。在这种情况下，完全控制意味着动作按钮（在游戏控制台里是右边的手柄）在这个游戏里什么也不做，它也不会在对用户点击空格键做出响应。这就让人觉得游戏好像有些东西坏了一样，因为这个按钮应该至少是能按的，即便按下去什么用也没有。所以，keyDownHandler()函数需要对付这个情况。它做的只是把playerAction标志设置为true。

keyUpHandler()要做的事稍微多些。首先，我们检查一下，确信玩家当前没有移动，并且还活着：

```
if (!gameState.playerDirectionUp && !gameState.playerDirectionDown &&
  !gameState.playerDirectionLeft && !gameState.playerDirectionRight &&
  this.player.state == "A") {
```

这样可以避免游戏者开始移动后在屏幕上的位置还没跳转到新瓦片之前就松开按钮。如果我们允许这段代码在上面的情况下执行，那么游戏者的移动标志就会在中间被重置，导致跳跃的动作在中间停止。那可不太好！所以，一旦检查出可以继续了，那么我们先检查一下空格键是否已经弹起，这时，只要设置playerAction为false即可。

然后，我们又碰到对应4个主要方向的情况。和前面一样，我们只看看第一种方向的情况，因为其他3种和这个几乎一样。

```
case KEY_UP:
  if (this.moveMatrix[tileY][tileX].indexOf("U") != -1) {
    if (tileY == 0 && tileX == 4) {
      this.player.state = "W";
    } else {
      this.player.tileY--;
      this.player.prevX = this.player.x;
      this.player.prevY = this.player.y;
      gameState.playerDirectionUp = true;
      gameState.playerDirectionRight  = false;
      gameState.playerDirectionDown = false;
      gameState.playerDirectionLeft  = false;
    }
  }
break;
```

首先，我们搜索一下moveMatrix，找到游戏者当前所在的瓦片。我们看看那块瓦片的字符串值是否包含字母U，表示游戏者可以从那块瓦片上向上移动。如果是，我们就要看看游戏者是否站在正对门口的一块瓦片上。如果是的话，我们就把游戏者的状态改成'W'，表示胜利。如果游戏者还没赢，那我们就给tileY减一，表示他最后站在的瓦片的位置。然后我们设置移动标志，就结束了。

最后一点点让人困惑的地方是destroy()方法，它只是清理在init()里创建的图片。当然，还有prototype那行，像Cosmic Squirrel里一样，确保Deathtrap类是MiniGame类的扩展。

11.3.14 编写Refluxive.js

现在只剩下一个游戏还没看了，那就是Refluxive。和前面一样，我们从看Refluxive类本身的UML图开始，如图11-18所示。

图11-18 Refluxive类的UML图

到现在为止，这些对你来说都已经是老旧的了。这个游戏也是使用典型的从MiniGame继承而来的类以及一些游戏特定的字段。

1. 设置弹球和牌子

首先是另一个内部类，这次它叫作BouncyDesc，如图11-19所示。这里定义了弹球（那个你需要让它保持在空中弹跳的东西）的X和Y坐标以及弹球的宽和高，它们都是用来监测碰撞的。我们还定义了弹球当前移动的方向。另外，有一个标志位可以告诉我们这个弹球是否还在屏幕中。当所有的3个弹球各自的onScreen字段都设置为false之后，游戏就结束了。不同于其他两个游戏的是Refluxive可以真的结束！

另外一个内部类是PaddleDesc类，如图11-20所示。PaddleDesc类描述了那个球拍——换句话说，就是玩家。碰撞监测只需要简单地把X/Y坐标加上宽和高就可以了。

图11-19 BouncyDesc类的UML图

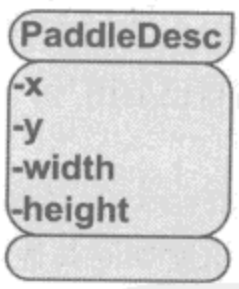


图11-20 PaddleDesc类的UML图

接下来是两个字段：paddle，它是指向一个PaddleDesc实例的指针；bouncies，它是BouncyDesc对象数组。

2. 开始游戏

下面，我们从init()方法开始来继续看些代码：

```
this.init = function() {

  // Configure base game parameters.
  this.gameName = "refluxive";

  // Load all the images required for this game.
  loadGameImage("background");
```

```
    loadGameImage("bouncy1");
    loadGameImage("bouncy2");
    loadGameImage("bouncy3");
    loadGameImage("paddle");
    loadGameImage("gameOver");

    // Initial bouncy positions.
    this.bouncies.push(
      new BouncyDesc(jscript.math.genRandomNumber(1, 180), 10, "SE"));
    this.bouncies.push(
      new BouncyDesc(jscript.math.genRandomNumber(1, 180), 70, "SW"));
    this.bouncies.push(
      new BouncyDesc(jscript.math.genRandomNumber(1, 180), 140, "NE"));

  } // End init().
```

这与你迄今为止曾经看到过的其他init()方法非常相似。我们设置了游戏名称和载入游戏所需要的图片，并且在这个例子中，构建了3个BouncyDesc对象，还把它们都压入到bouncies数组中。它们的水平位置是随机设定的，竖直位置是固定的。初始移动方向也是静态的。后面这两个项使用静态是为了有助于确认它们是从合理的位置出发的，以完全不同的方向运动，并且足够独立，让游戏更有挑战性。

一个有趣的事情是，注意一下每个弹球都有自己的图片，即便它们本质上都是相同的GIF。这是必须的，因为每个弹球都需要可以被独立设定地址。这样有些效率问题，因为我们要把同一个GIF载入到内存3次，浏览器和/或操作系统可能会实际上共用一个文件，但我们可不能依赖它们。对于这种小游戏来说，这谈不上什么问题。对于更大一些的应用，可能这就是一个需要考虑解决的缺点了。一种解决方法是，修改代码执行的方式，让游戏元素从它的图片（们）中抽出来。可能可以有些类似ImageManager那样包含全部图片的类，并且它负责保证任何给定图片的都只存在一个实例，游戏元素要通过这个类引用自己的图片。

3. 玩游戏

下面是processFrame()，它的开始与其他两个相同：清除屏幕中的所有图片。然后我们看到了背景的一个blit()，这就如同其他两个游戏一样。在那之后，是这个游戏独有的一些代码：

```
if (!this.bouncies[0].onScreen && !this.bouncies[1].onScreen &&
  !this.bouncies[2].onScreen) {
  blit(this.gameImages["gameOver"], 10, 40);
  return;
}
```

如我之前提到过的，Refluxive是3个游戏中唯一一个可以在用户决定退出之前结束的。当所有3个弹球都跑到屏幕之外时，它就结束了。所以，我们还要有对这个条件的判断。如果遇到这个情况，就展示“游戏结束”消息。因为processFrame()中剩下的部分在这种情况下就不适用了，所以我们简单地return就可以了。

然而，如果游戏继续进行，我们就从blit()那个球拍和3个弹球开始：

```
if (this.bouncies[0].onScreen) {
  blit(this.gameImages["bouncy1"], this.bouncies[0].x, this.bouncies[0].y);
```

```
}
if (this.bouncies[1].onScreen) {
  blit(this.gameImages["bouncy2"], this.bouncies[1].x, this.bouncies[1].y);
}
if (this.bouncies[2].onScreen) {
  blit(this.gameImages["bouncy3"], this.bouncies[2].x, this.bouncies[2].y);
}
```

对于每个弹球，我们检查它是否在屏幕中。blit()那些用户看不见的东西是毫无意义的！

下面，我们就处理玩家动作：

```
if (gameState.playerDirectionRight) {
  this.paddle.x = this.paddle.x + 4;
  // Stop at edge of screen
  if (this.paddle.x > 174) {
    this.paddle.x = 174;
  }
}
if (gameState.playerDirectionLeft) {
  this.paddle.x = this.paddle.x - 4;
  // Stop at edge of screen
  if (this.paddle.x < 1) {
    this.paddle.x = 1;
  }
}
```

通过一些简单的边界检测来确保玩家没有把球拍移出屏幕的边缘。因为在这个游戏中，玩家只可以左右移动，所以这样就可以了！注意这个游戏并不需要实现keyDownHandler()和keyUpHandler()，因为它们的默认实现就可以了，如同宇宙松鼠中一样。这样也是减少代码量的一个好方法！

然后是弹球的移动。为了实现这个效果，我们遍历bouncies数组，对于每个弹球，首先监测它们是否在屏幕中。如果不在，就继续循环。如果在屏幕中，我们就先基于它当前的dir值来移动它：

```
if (this.bouncies[i].dir == "NE") {
  this.bouncies[i].x = this.bouncies[i].x + 3;
  this.bouncies[i].y = this.bouncies[i].y - 3;
}
if (this.bouncies[i].dir == "NW") {
  this.bouncies[i].x = this.bouncies[i].x - 3;
  this.bouncies[i].y = this.bouncies[i].y - 3;
}
if (this.bouncies[i].dir == "SE") {
  this.bouncies[i].x = this.bouncies[i].x + 3;
  this.bouncies[i].y = this.bouncies[i].y + 3;
}
if (this.bouncies[i].dir == "SW") {
  this.bouncies[i].x = this.bouncies[i].x - 3;
  this.bouncies[i].y = this.bouncies[i].y + 3;
}
```

我相信你可以猜到，“NE”代表东北。相应地，“NW”是西北，“SE”是东南，“SW”是西南。这是弹球可能移动的4个方向。对于每个方向，我们都相应地调整X和Y的坐标。

做完这些之后，我们需要处理弹球弹离屏幕侧边和顶边的情况。使用如下的代码：

```
// Bounce off the frame edges (horizontal).
if (this.bouncies[i].x < 1) {
  if (this.bouncies[i].dir == "NW") {
    this.bouncies[i].dir = "NE";
  } else if (this.bouncies[i].dir == "SW") {
    this.bouncies[i].dir = "SE";
  }
}
if (this.bouncies[i].x > 182) {
  if (this.bouncies[i].dir == "NE") {
    this.bouncies[i].dir = "NW";
  } else if (this.bouncies[i].dir == "SE") {
    this.bouncies[i].dir = "SW";
  }
}
// Bounce off the frame edges (vertical).
if (this.bouncies[i].y < 1) {
  if (this.bouncies[i].dir == "NE") {
    this.bouncies[i].dir = "SE";
  } else if (this.bouncies[i].dir == "NW") {
    this.bouncies[i].dir = "SW";
  }
}
```

当弹球的X坐标小于1时，也就是说它已经碰到了屏幕的左边框了。这时，我们要把它运动的方向改为反向。所以，如果它正向西北移动，我们就把它反向成东北，西南的话就反向成东南。同样地，对于屏幕的右边（横坐标182，因为弹球是18像素的宽度，所以右边框应该在182+18=200的位置），我们也反向运动方向。对于屏幕的上边框，也适用这个基本逻辑。

注意这里并没有对屏幕底部的检测，至少，不像上面那些，因为有一个情况是弹球并不反弹的。如果玩家没有接到弹球，它就只是消失在屏幕底部。这里最后一个需要做的检查就是这种情况：弹球何时离开屏幕。

```
if (this.bouncies[i].y > 200) {
  this.bouncies[i].onScreen = false;
  subtractFromScore(50);
}
```

当玩家没有接到弹球时，它就死了，可以说，是通过把它的onScreen属性设置为false来实现的。这也还消耗了玩家一些分数——本例中是50。

现在还剩下一件事情就可以完成这个游戏了。你能猜到是什么吗？

我们需要让玩家可以反弹那个弹球！实现这个功能的代码如下：

```
if (detectCollision(this.bouncies[i], this.paddle)) {
  // Reverse bouncy direction.
  if (this.bouncies[i].dir == "SE" &&
    this.bouncies[i].x + 9 < this.paddle.x + 12) {
    this.bouncies[i].dir = "NW";
    addToScore(10);
```

```
    }
    if (this.bouncies[i].dir == "SE" &&
      this.bouncies[i].x + 9 > this.paddle.x + 12) {
      this.bouncies[i].dir = "NE";
      addToScore(10);
    }
    if (this.bouncies[i].dir == "SW" &&
      this.bouncies[i].x + 9 < this.paddle.x + 12) {
      this.bouncies[i].dir = "NW";
      addToScore(10);
    }
    if (this.bouncies[i].dir == "SW" &&
      this.bouncies[i].x + 9 > this.paddle.x + 12) {
      this.bouncies[i].dir = "NE";
      addToScore(10);
    }
  } // End collision detected.
```

这也还是在for循环之内的，所以我们其实还是在与某个弹球一起工作。调用detectCollision()函数，如果它返回true，我们就做你之前看到的那种把运动方向反向的操作。这里的一个区别在于，方向决定于弹球撞击了球拍的那一部分。如果弹球正向东北移动，并且玩家使用拍子的左边撞击了它，那么运动方向就变成西北。如果撞击了拍子的右边，就变成东北。如果是向西南运动的球撞击了拍子的左边，它就转变成向西北运动，如果撞击了拍子的右边，那么运动方向就变成东北。

亲爱的朋友们，实际上Refluxive后面的逻辑已经完成了！剩下的代码就是普通的destroy()方法了，它为init()中载入的每个图片和继承自MiniGame的Refluxive类的prototype规范，都调用了destroyGameImage()。

所有的就都完成了！我希望你会同意，这个应用程序提供了很多关于如何使用JavaScript进行面向对象编程的东西。不过更加重要的是，希望你在看这个游戏的时候能够和当初我写它时得到同样多的乐趣！尽量不要浪费太多工作时间来玩它！

11.4 练习

我相信（这些建议的练习）不需要我来告诉你了，不过我的主要建议就是写更多的迷你游戏！如果时间允许的话，我会把成熟的K&G Arcade中更多的迷你游戏移植到JavaScript。我将把它们与本书的其他代码一起发布到Apress下载站点去。添加游戏应该是很简单的，只需要放置适当的目录以及其中的游戏文件，并更新GameSelection类就可以了。你也可以这么做！

提出一两个简单的游戏构思，然后试着实现它们。如果你从来没写过游戏，我相信你会从中获得非常多的快乐，并且学到很多东西。可能你还不能实现像最终幻想、Halo这样的游戏，所以游戏不必太大。瞄准那些自己踮着脚尖才能够得着，同时还乐趣丛生的东西。毕竟，那就是视频游戏的所有……或者是至少应该具备的！

另外一个建议是，使用cookie保存每个迷你游戏的最高分，并创建一个“名人纪念馆”画面来展示它们。这会让你对如何开发游戏中的画面有个很好的感觉，还包括了一些关于cookie的联系。我建议你首先来完成这个任务作为你的起点。

11.5　小结

开始时你可能还不这么认为，但是，在任何平台上，任何语言中，编写游戏都是练习所学技能并深入研究的最好方法之一。游戏设计到了多种领域的专门技术，并且通常需要你更加深入地延伸你的技能，我相信本章已经表现出这个特点了。

在本章中，你了解了JavaScript的面向对象方法，写出干净且可扩展的代码。示例项目说明了在JavaScript中应该如何实现类的继承。你还看到一些最大化性能的技巧，包括避免多余的DOM元素访问以及加速整个应用程序。你甚至拣到了一两个基本游戏理论的珍闻！并且，我相信我们所创建的游戏真的是很好玩的！

第 12 章

Ajax：客户端和服务器端相遇

12

Ajax是当今的潮流。要是你对Ajax一无所知，你甚至不好意思跟人说自己是Web开发人员[①]！在本章里，你会学习一些Ajax的知识，并将其应用在构建一个一对一的聊天程序中，这个应用和你看到过的很多公司网站上为产品提供的在线聊天支持是一样的。这将是本书中一个使用了服务器组件的项目，你可以看到所有这一切是如何相互配合的。另外，你还会接触到一个新的库——Mootools，看看它提供了些什么。

12.1 聊天系统的需求和目标

我们先做一些假设，假设我们是街上一个新开张的公司。公司的名字是Metacusoft Systems，这么叫只是因为这个域名还没人注册，并且用Google搜索它，返回来的是零结果，这样就比较确信我们不会侵犯别人的产权了。第一个注册该域名并建立此公司的读者可是会得奖的哦！（呵呵，开玩笑的，没有奖品给读者，不过你可以随便用这个名字。）

另外，我们销售窗口小部件。没错，就是很俗的窗口小部件。它好玩、有用，人人都离不开它，但我们还是想不出更好的描述性名称。而且，即使我们的窗口小部件是很伟大的产品，用户在使用它的时候仍然不是一帆风顺的。有时候，用户需要我们的帮助才能更好地使用这些窗口小部件，让生活变得更美好（虽然我们从不会在财务报表上承认这点）。所以，我们需要给用户提供一些支持。

我们有几个人通过电话提供支持（当然是在印度，那里叫Bill、Mike、Tom、Sam和Dave的人也不少）。我们还会给在线的人们提供一些支持，这些人，就好像Morlock人一样，[②]不喜欢与人类的接触。我们将提供一个在线的一对一的聊天系统，可以与人们进行实时的交流，而不需要真的和他们对话，也没有发生实际交谈的风险。

这个应用相对简单：一个人打字，另外一个人看，这个就是需求了。好，我们还是适当整理一下。

- 应用看上去应该比较体面。毕竟，它是我们公司的一个公众接口，给那些在我们产品上碰到问题的用户提供帮助。他们很可能已经对我们有些意见了，我们可不想用一个难看的网站进一步惹恼他们。
- 让我们允许用户把聊天内容复制到剪贴板上，粘贴到另外的应用程序中。这样，他们就可以

① 电影《大腕》中的语气。——译者注

② Morlock（中译：莫洛克、没落客）是一种虚构的种族的名字，属于人类的旁支，是著名科幻小说家H.G. Wells在其大作The Time Machine中虚构出来的，存在于未来的很多年之后（当然是Morlock，不是H.G.Wells）。Morlock住在地下，在那个时候，他们甚至都不能被称作人类了（当然指正式称呼）。哦，对了，Morlock会吃一种叫Eloi（中译：埃洛依）的种族，这个种族也是人类的后代。有趣吧。

保存聊天记录，然后在将来有问题的时候可以拿出来冲我们嚷嚷。让我们同时给予他们直接打印聊天内容的功能，这样如果用户想起诉我们，他们就可以手持他们胜诉所需要的文件了（也许我们对我们的客户太好了）。

- 应用当然要是以Ajax为基础的，并且使用Mootools库来实现上述功能（以及我们需要它提供的任何其他功能）。用HTML或者不带服务器组件的JavaScript实现一个一对一的聊天实在是太难了，所以我们在这里需要接合一个服务器。我们将用Java和 Microsoft的技术（就是ASP）实现一个后台，这样至少可以覆盖大部分读者。

面对如此一个相对来说需求很少的应用，那就来看看我们是怎样把东西做起来的，可以开始了吗？好，实际上，先看看我们不应该采用的方法。

12.2　“经典”的Web模型

最开始的时候，Web出现了。它是好东西。各种各样新的字眼、词语和术语涌入词汇表中，而我们在说这些词的时候会觉得更酷。（别笑，你得承认这一点，头几次在交谈中使用“超文本”这个词的时候，你肯定觉得自己特像个教育家！）Web应用诞生了。这些应用，从某种意义上来说，和多年前的“主机”时代很像，应用是在“大机器”上运行的，以一种分时的方式提供服务，因为用户在用于与该应用交流的机器上没有做任何处理。它们也不可能像之后的Visual Basic、PowerBuilder、Delphi和C++“肥客户端”那么“绚丽”（这些客户端今天仍然在用，只是在Web应用的影响下，没那么出彩了）。

Web应用已经有多年历史了，时至今日仍然每天在涌现，它们有个大问题——在很大程度上，每次发生一些事件的时候，Web应用都需要重画整个屏幕。很大程度上，它们的本质是一个服务器中心的引用。当用户做一些动作的时候（除非是一些非常简单的事情，比如鼠标移动效果等），服务器必须进行处理，然后重画界面，让用户看到对应用户动作的数据更新之后的界面。毫无疑问，你也会认为这么做的效率是很低的。

这种模式就是我说的“经典”的Web设计方式，或者说模式。（我还没听说过别人如此使用这个术语，不过我还是不信我是第一个这么用的人！）“经典”的Web模式之于我的含义是这么一个范例：服务器处理几乎所有用户触发的事件，然后重画整个屏幕。这是自从Web出现以来，过去15年中Web应用的开发模式。相对而言，“现代”Web模式指的是开发Web应用的新的方法，这种模式要求客户端承担一部分负载，并且在整个应用中起更重要的作用。

你可能会问自己，“如果我们已经在如此长的时间里使用经典方法开发Web应用，甚至时至今日仍然在用，那它有什么问题呢？”在很多方面，绝对没问题！实际上，经典模式在设计Web应用上仍然价值非凡。经典模式对那些主要是线性应用流的应用依然很有效，而且是用一种可用的方式分发信息的绝佳方案。经典模式让大多数人很容易发布信息，甚至也可以让人们很容易开发那些有简单用户交互的基本应用。经典的Web模式简单、随处可见、并且可以很容易被大多数人使用。

不过，它不是开发那些具有大量用户交互的复杂应用的理想环境。用户之所以能做这些复杂的交互，是工程师的天才和创造性的证明，而非Web作为应用分发的媒介的作用！

如今，我们把网站和Web应用分开是合理的，如表12-1里总结的。它们其实是Web提供的两种不同的用途。一个是分发信息。在这种情况下，最重要的事情是让信息以一种被尽可能多的受众访问到的方式进行分发。这不仅仅是针对那些使用屏幕阅读器和类似设备的残障人士，还包括那些使用更加有局限性设备的人，比如使用手机、移动终端和个人数字助理等设备的用户。在这种场合，除了从一

个静态文档跳转到另外一个静态文档之外，一般都不会有用户交互发生，顶多就是通过简单的表单进行非常有限的交互。用这种方式使用Web的，可以归类为网站。

表12-1 Web应用与网站对比小结

Web应用	网 站
设计的时候充分考虑大量的用户交互	除了在文档之间的指引之外，很少用户交互
主要目的是提供一些功能，通常是基于用户输入的实时功能	主要目的是周期性地分发信息
使用的技术是那种对客户端有大量要求的技术	基于客户端能力的最小公分母思维来创建
可用性放在必要功能之后的位置，而且事实说明很难制作复杂但同时又充分可用的Web应用	可用性的考虑通常是让尽可能多的受众可用
更加基于事件和非线性	通常是线性的，有一个预期用户可以遵循的路径，只有少量的变化

与之相对，Web应用的关注点完全不同。它们关心的不仅仅是展现信息，而且还要基于用户的动作和用户输入的数据执行一些功能。在很多场合下，用户都可能是另外一个自动化的系统，不过，我们通常指的是有血有肉的活人。Web应用通常都更复杂，并且通常对访问它的客户端有更多要求。这里的客户端指的是浏览器。

经典模式有另外一个问题：为了实现最大程度的可用性，你通常需要用最少东西来设计，这样会大大制约你能做的事情。让我们思考一下这里的“最少的东西”的情况。看看为了实现最大程度的可用性，你能用的和不能用的东西。下面是一个我想到的列表。

- 客户端脚本。不，你不能用这个，因为大多数移动设备还不支持脚本，或者是脚本的使用非常受限制。更别说那些使用全功能PC的用户了，他们因为安全或者其他原因关闭了脚本功能。
- CSS：你能用样式表，但是必须非常小心地使用一个老的CSS标准，以确保大多数浏览器都能正确呈现样式——比如，不能用任何CSS 2.0功能。
- 框架：不，框架不是得到全面支持的东西，尤其是在许多移动设备上。即使这些设备支持它，你也得小心，因为框架实际上相当于在内存里运行的另外一个浏览器实例（有时候，它实际上就是另外一个浏览器实例），而这个在移动设备里是要命的事情。
- 图片：在可用性领域，图片的麻烦更多，因为它们实际上承载了比一个alt属性更多的信息。所以，有些图片的含义在一些残障人士面前可能就会丢失了（比如色盲），不管你有多想帮助他们。
- 新HTML标注：许多人都仍然在使用那些甚至连HTML 4.01都不支持的浏览器，所以安全起见，你可能需要用HTML 3.0编码。显然，这么做的话你会失去很多功能。

这里最重要的元素，可能就是缺乏客户端脚本支持了（这也是本书的主题）。如果没有客户端的脚本，你就失去了很多成为开发人员的可能了。最明显的事实就是，你实际上除了让服务器处理每个用户的交互请求，并且响应一个完全重画的视图之外，没有任何其他选择。有时候，你可能可以通过在框架里刷新一些原数据来绕开一些障碍，或者是用其他技巧来换取一定程度的交互，但框架很可能是不被支持的，所以你甚至连这个选择都不具备！

你可能会想，“服务器呈现整个页面有什么问题吗？”当然，这个方法是有自己的优点的，比如，因为具有对应用的运行时状态的完全控制而带来的固有的安全性（用户无法黑掉那些代码），就是其中一大优点。没有因为用户端需要下载代码而导致的延迟是另外一个优点。不过，的确有些场合下，有些问题会完全覆盖这些优点。可能最明显的问题就是服务器端的负载。如果我们要求服务器在一定

的并发量的情况下反复多次代替客户端做这些处理，无疑是要求服务器更加健壮，并且更加强大。从长期来看，所有这些都是成本，因为你需要花钱购买更多服务器的处理能力来对付这些负载。

现在，很多人脑子里想的是“买更多硬件呗”，而且我们实际上处在这样一个时代：那就是大部分时候它都能够工作不过这么说实际上和我们花更多的钱买更大的引擎装在车子里追求更快的速度是一样的，通常我们需要更快速的时候都会这么做。实际上，我们可以通过把小引擎设计得更高效来实现提速的目的，而在很多场合下，这是更需要的——尤其是我们希望能呼吸更多干净的空气的时候！一个更恰当的比喻可能是这样的：我们总是在一个中型车外头添加更多座椅，希望它能在车“里”装载更多的乘客，却没有去寻找让这些乘客到达目的地的更有效的方法。尽管这种贴胶布似的做法可能可以应付一时，但最终会有人栽倒并被跟在后面的18轮大客车给压碎的！

另外一个服务器承担一切的方式的问题是网络开销。网络技术持续不断地以让人眼花缭乱的速度进步和飞跃。现如今我们很多人都拥有了宽带接入，甚至我们可着劲的用都没法全部用完。（我就是其中之一！）但是，这不意味着我们就可以让应用收发比需要的多得多的信息。我们仍该努力节俭，不是吗？

另外一个经典模式的问题是用户对应用如何感受。如果服务器需要重画整个屏幕，那么通常都导致更长的等待时间，更不用说Web应用里发生的多次视觉重画，闪烁等这类几乎所有用户都不喜欢的东西。用户也不喜欢仅仅因为出了错就失去他们所输入的所有东西，这也是经典模式经常会有问题的地方。

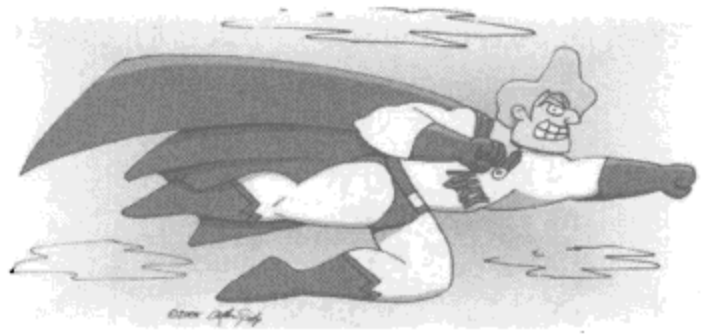
最后，经典模式仍然可以在小范围内扩展得很好，以及那些大多数时候发布静态信息的场合，但是它在更大范围内的扩展就不够好了，而且可以说无法对付今天Web对动态内容的需求。在这种场合下，扩展意味着在应用里增加功能，而不是并发请求的处理能力（尽管很可能在这方面，经典模式也很差）。如果应用不能平滑工作，或者是操作的终端导致太多数据丢失，或者是速度严重下降，那么经典的方法扩展性就很差了。

经典模式在网站的领域里还会继续为我们提供优质服务很长一段时间，但是在Web应用的领域里（如果你在阅读本书，很可能你对此感兴趣）它已经死亡了，屠杀它的就是我们传奇中的英雄：Ajax！

12.3　Ajax

可以说，Ajax是在Web设计咨询企业Adaptive Path 公司用户体验总监耶西·J.加勒（Jesse James Garrett）手中降生的。虽然看似玩笑，但绝对是事实！加勒（Garrett）先生2005年2月写了一篇随笔（你可以在 http://www.adaptivepath.com/publications/essays/archives/000385.php找到这篇随笔），在这篇随笔里，他缔造了术语Ajax。

Ajax的意思是异步JavaScript和XML（Asynchronous JavaScript and XML），我敢拿我的狗当赌注，打赌你已经知道这个含义了（我自己不养狗，不过如果你不知道Ajax是什么意思，我愿意给你买一只——好，玩笑啦）。Ajax有趣的地方在于，它实际上不必是异步的（虽然事实上的确是异步的），不是非用JavaScript不可（但实际上也总是用JavaScript），而且也不一定非用XML不可（不过可能有一半的时候用的都是XML）。实际上，有个最著名的Ajax例子，Google搜索提示（http://www.google.com/webhp?complete= 1&hl=en），就完全不往回传递XML！究其自身而言，实际上它甚至不回传数据，它传回的是包含数据的JavaScript（这个东西，如果你看过第5章，应该已经知道了：Yahoo和Google的Web服务返回了一堆封装在一个JavaScript函数调用里的数据）。



Ajax来救我们来了！（现在你知道把代码或者架构拟人化成超人之后是什么样子了。）

12.3.1 Ajax思维的核心

Ajax自身的核心是一个非常简单的概念，甚至都不需要我们挖空心思去想，这就是：我们没必要为每次用户的的交互（或者说用户“事件”）都刷新整个网页的内容。如果用户点击了一个按钮，我们没必要让服务器重新呈现一个新的页面，就像经典Web模式里面做的那样。与之对照的是，我们可以定义需要更新的页面的范围，并且对用户事件进行颗粒度更小的控制。在点击一个链接的时候，你不再受限于提交一个表单或者是定向到一个新的页面。你现在可以对那些非提交按钮的点击，以及在文本框里输入一个键这样的事件做出响应了——实际上，你可以对任何事件进行响应。

服务器不再是对用户所见的内容负完全责任；有些逻辑现在是在用户的浏览器里执行的。实际上，在很多场合下，返回一堆数据要比返回一堆标签给浏览器来显示明显要更好一些。如果看看前面描述的应用开发的简单历史，就会发现经典Web开发模式从某种意义上已经是背离了我们开始开发之前的意愿的。

最重要的是，Ajax是一种思考方法，一种应用开发的方法，一种才智，而不仅仅是一门技术。有关Ajax最有趣的事情是，它根本不是新东西，除了用来描述它的名字之外。在费城的Java用户组的会议上，一个人的演讲再次提醒了我。他的名字是Steve Banfield，在谈到Ajax的时候，他说（根据我的记忆组合的），“你总是能告诉那些已经开发过Ajax项目的人有关Ajax的概念，因为这一切发生得太快，他们自己都还不知道！”没有更形象的说法了！我就是其中一个好几年前就开始Ajax开发的人。我只是没有想过自己这么做有什么特殊的地方，因此就没有给它一个“合适”的名字。Garrett先生与众不同的地方就是这里。以前我们做的东西也许和我们今天认识的Ajax形式上有所不同，但那是因为技术方法可能有所变化，但是下层的才智是一样的，在我看来，这才是核心要点。

如果你用Ajax的方式思维，也就是我们正在谈到的话题，那你就不再受经典的Web模型的约束了。你现在至少可以拿回来一部分肥客户端提供的功能，而同时还享有Web提供的便利。这些便利条件的首条（也可能是最重要的一条）就是无处不在的Web浏览器。

你是否曾经碰到过这样的事情：有时候你需要在一台自己从没摸过的机器上演示自己新的肥客户端程序（比如，一个Visual Basci应用）？甚至是需要在坐满公司高管的一个会议室里？更甚的是因为一些你没预期到的DLL冲突，你的演示程序很惨地失效了？如果你是一个开发人员，那么所有这些问题的回答很可能都是“是”（除非你是在公众场合演示，那你就不用给高管作演示了，但你应该明白这个意思）。如果你从没开发过Windows应用，可能没有这些经历。但是你可能不得不承认，在很长一段时间里，这种事情发生的频率实在是比我们预期的高得多。对于一个以Web为基础的应用来说，这通常都不是问题。只要保证PC里安装了当前版本的浏览器，那么98%的时候都不会有问题。

Web应用的另外一个主要的便利是分发。你再也不需要3个月的试用期以确保应用不会和现有的企业应用套件冲突。一个在Web浏览器里运行的应用，除安全因素之外，将不会影响或者说不会被其他PC里的

应用影响（我相信我们都有一些例外的导致冲突的故事，不过它们的确是很罕见的“例外”了）。

我们都经历过，现场演示，工程师不能出错

当然，你可能已经知道这些好处了，要不然你根本不会在一开始就对Web开发感兴趣。

听起来不错，是吗？不过，Ajax的世界也并非坦途。Ajax也不是没有自己的问题。有些问题只是感觉上的问题，但还有些的确是很严肃的问题。

12.3.2　可用性以及类似的考虑

第一条也是最重要的一条（至少在我头脑里如此），是可用性的问题。使用Ajax至少会让你失去一些或者一部分可用性，因为像屏幕阅读器这样的设备是设计用于阅读整个页面的，而因为你不再发回完整的页面，那屏幕阅读器就会有问题了。我的理解是有些屏幕阅读器或在某种程度上处理Ajax，很大程度上是取决于Ajax是如何使用的（如果内容是用文本插入到DOM里去的，那就大不一样了）。不管怎样，如果你知道你的目标受众里有残障人士，那么就需要万分细心了，并且你还要很严肃地考虑（以及测试!）Ajax是否可以在你的场合下工作。我相信随着时间的推移，这样的问题会被逐步解决，但是现在，它绝对是一个问题。同时，下面是一些可以用来改善可用性的手段：

- 让用户知道动态更新。在页面顶端放上一条提示，告诉页面会动态地更新。这样就可以让用户在屏幕阅读器上定期重新阅读页面，以便听到动态的更新。
- 使用alert()这样的弹出窗口。根据在页面上使用的Ajax的类型，可能的话尽量使用alert()，因为这些会被屏幕阅读器读出来。对于那些以Ajax为基础的表单提交来说，这个建议是合理的，但显然对那些定时重复的Ajax事件而言，这不是个好建议。
- 增加视觉提示。要知道不光盲人需要可用性帮助，可用性也可以用于帮助有视力的人。对于他们，可能的时候就试着用一些视觉提示。比如，简短地突出显示一些改变了的项就会是一个很大的帮助。有些人称之为“黄色淡出效果”，我在第1章里面提到过，说它是一种可以改进用户体验的特效。这个核心是用黄色突出显示改变的项，然后逐渐淡化成非突出显示的状态。当然，它不一定非要黄色的，也不一定非要淡出，但是里面的概念是一样的，突出显示那些改变了的信息，提供了一个视觉提示，有事情发生了。要记得有些时候Ajax导致的变化是非常细微的，所以不管你做了什么帮助用户注意到变化的事情，都是让人感激的。

Ajax对于很多人来说的另外一个缺点是增加了复杂性。很多商店自己没有Ajax要求的客户端编码专家（使用工具包简化工作不算）。问题是，在客户端发生的错误仍然要比在服务器端发生的问题更难跟踪和排除，Ajax并没有让这件事更简单些。比如，“查看源代码”并不能看到对DOM做的修改。

另外一个问题是，Ajax应用会在很多场合偏离一些长期受人尊重的Web概念，最明显的是前进、

后退按钮和书签。因为不存在整个网页了，而是返回页面片段，在很多场合下，浏览器都不能对这些做书签。而且，前进和后退按钮也失去了作用，因为他们还指着最后请求的URL的位置，而几乎从来不会包含Ajax请求（比如，通过XMLHttpRequest做的请求通常不会加到URL历史里面，因为通常URL都不会改变，尤其是使用的方法是POST的时候）。

12.3.3 Ajax：一个需要大多数人转换的观念

实际上，Ajax代表了一个很多人（甚至是大多数人，基于现在大多数Web应用的状况而言）都需要做的观念转换，因为它可以说是从基本上改变了开发Web应用的方式。更重要的可能是它代表了一个向用户转换的观念，而且，实际上正是用户将会驱动Ajax的采纳。相信我，Ajax这个开发利器很快就会进入你的视野。

把一个非Ajax的Web应用和一个同样的、使用Ajax的应用放在同一个用户面前，你认为他们会选择哪个来作为日常的、十之八九都会用的应用？肯定是Ajax版本的！

用户会马上感觉到Ajax版本的响应速度的提升，并且会发现他们不需要傻傻地盯着浏览器图标的旋转，等待服务器的响应，盼望着发生点儿什么事情。他们会发现如果他们操作错误了，马上就可以得到错误响应，而不是像非Ajax的Web应用里那样要等着服务器告诉他们出错了。他们能看到键击提示、立即排序的表以及实时更新的概要/细节显示——这些都是在非Ajax的Web应用里无法看到的。他们能看到可以拖放的地图应用，就像他们过去花了$80买到的全功能的地图应用一样。所有这些都很明显对用户来说更先进的特性。用户已经习惯于经典的Web应用模式了，但是一旦面对在用户友好和响应性方面都可以和以前熟悉的肥客户端相比的东西的时候，他们会马上意识到他们熟知的Web已经死了，或者是应该死了！

如果你想想近年来出现的许多大的技术，就会发现其中很多都是我们这些技术人员而不是最终用户在驱动它们。你认为一个用户会要求EJB为基础的应用吗？不会，我们只是认为这会是一个好主意。（我们在此事上是多么错误！）Web服务又如何？还记得当初它是如何从基础上改变了应用构造世界的吗？没错，我们现在的确在使用它，但是它们，又和一个企业系统之间的接口有多大的区别呢？通常没有区别。全局描述发现与集成目录（Universal Description, Discovery, and Integration UDDI）和给予应用运行时查找，动态连接并且使用一个已注册的服务又如何？它们够不够好？对我们这些奇客而言，它是下一阶段发生的东西，但是它甚至没有和用户有什么关系。

但是Ajax不一样。用户能看到好处。对他们而言，Ajax的优点是非常真实并且明显的。实际上，作为技术人员的我们，尤其是我们中用Java进行Web开发的人，可能一开始会抵触Ajax，因为客户端做得更多，而这一点和多年来钻进我们脑子里的概念是相冲突的。要知道，我们都认为在JSP（JavaServer Pages）中的小脚本是烂东西，用客户标签来避免使用JSP小脚本。用户并不关心优雅的结构，关注点的分隔，以及通过抽象实现代码重用。用户只是希望能在Google Maps（http://maps.google.com）里四处拖动地图，并且是实时地，而不用等待整个页面重新刷新。

区别是明显的，用户需要它，并且他们现在就需要它。（来吧，我们是成年人！）

RIA

Ajax并非最近流行的用于描述同一个事务的唯一的术语。你可能听说过Web 2.0或者RIA（富因特网应用）。RIA是一个我特别喜欢的术语。尽管它没有我熟悉的正式的定义，但是大多数人都可以

不用上Google搜就了解它的含义。

简单地说，RIA的目标是创建基于Web的“富”应用。应用本身在浏览器中运行，但是界面、感觉和功能都更像一个传统的肥客户端而不像传统的网站。类似部分页面更新这样的事情几乎是一定的，因此Ajax总是和RIA掺和在一起。（尽管具体掺和在一起的Ajax形式会有所变化，你可能根本看不到典型的Ajax解法中的XMLHttpRequest对象出现！）这种类型的应用通常都更加用户友好，并且更为它们服务的用户社区所接受。实际上，你制作RIA的终极目标应该是用户说：“我都不知道它是Web应用！”

Gmail（http://gmail.google.com）是RIA的好例子，当然它还不完美。尽管它绝对比典型的网站好得多，它看上去仍然太像一个网页。另外，Google的开发人员在把Ajax引入人们的脑海里的这件事上，可能做得比任何其他人都多。他们不是最早这么做的，甚至也不是做的最好的，但毫无疑问他们创建了一些最显著的例子，并且实实在在的给人们显示了Ajax所具备的能力。

12.3.4　Ajax的“Hello, World”例子

好了，理论分析已经足够多了。有血有肉的Ajax看上去像什么呢？图12-1在屏幕上显示了一个非常简单的应用。（在这个例子里别期望太多，哥们儿！）

图12-1　请注意在第二个下拉列表中没有内容，因为还没有在第一个下拉列表中选中任何东西

如你所见，在第二个下拉列表里开始的时候没有内容。这个下拉列表将会随着第一个下拉列表的选定而动态的填充，如图12-2所示。

图12-2显示了在选择第一个下拉列表的元素的时候，第二个下拉列表内容动态改变的情况。这种例子里，你看到的是热门电视剧《巴比伦5》。（别争了，你知道我说的是真的。另外，你还有机会把你最喜欢的剧放进去呀！）现在让我们看看这一切都是如何实现的。

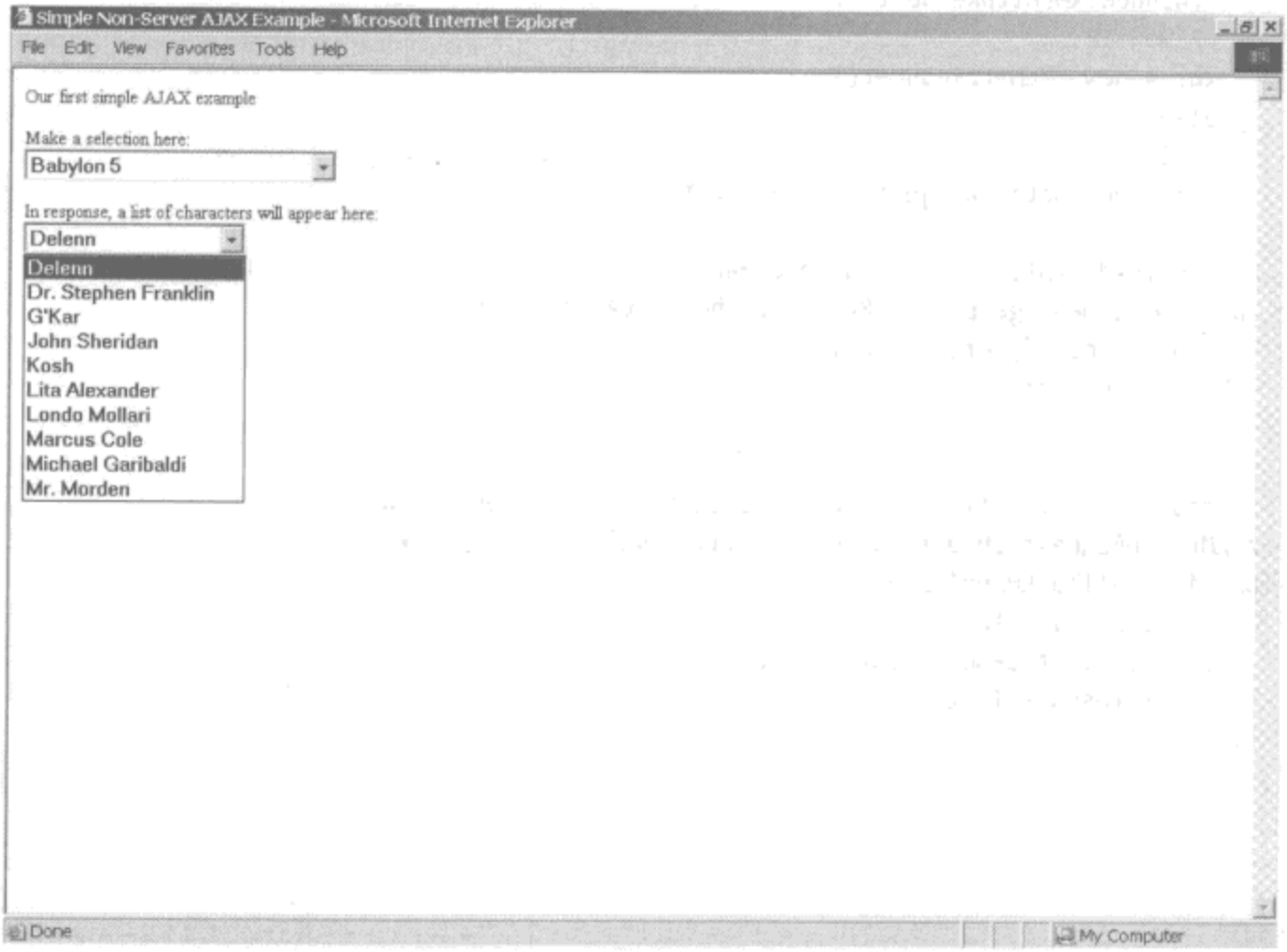


图12-2 在第一个下拉列表里做出选择，然后第二个下拉列表的内容就动态地用“服务器”返回的数据更新

代码清单12-1显示了简单Ajax例子的第一页，它执行了一个相当典型的Ajax类型的函数：基于另外一个选择列表的选定项，填充一个<select>框。这种事情在Web开发过程中经常出现，而“经典”的方法是给服务器提交一个表单——要么是通过用户点击一个按钮来触发，要么是通过JavaScript事件处理函数，然后让它呈现更新了第二个<select>的内容之后的页面。有了Ajax，这一切都不存在了。

代码清单12-1 我们第一个真正的Ajax应用

```
<html>

  <head>

    <title>Simple Non-Server AJAX Example</title>

    <script>

      // This is a reference to an XMLHttpRequest object.
      xhr = null;
```

```
// This function is called any time a selection is made in the first
// <select> element.
function updateCharacters() {
  // Instantiate an XMLHttpRequest object.
  if (window.XMLHttpRequest) {
    // Non-IE.
    xhr = new XMLHttpRequest();
  } else {
    // IE.
    xhr = new ActiveXObject("Microsoft.XMLHTTP");
  }
  xhr.onreadystatechange = callbackHandler;
  url = document.getElementById("selShow").value + ".htm";
  xhr.open("post", url, true);
  xhr.send(null);
}

// This is the function that will repeatedly be called by our
// XMLHttpRequest object during the life cycle of the request.
function callbackHandler() {
  if (xhr.readyState == 4) {
    document.getElementById("divCharacters").innerHTML =
      xhr.responseText;
  }
}
  </script>

</head>

<body>

  Our first simple AJAX example
  <br><br>

  Make a selection here:
  <br>
  <select onChange="updateCharacters();" id="selShow">
    <option value=""></option>
    <option value="b5">Babylon 5</option>
    <option value="bsg">Battlestar Galactica</option>
    <option value="sg1">Stargate SG-1</option>
    <option value="sttng">Star Trek The Next Generation</option>
  </select>
  <br><br>
    In response, a list of characters will appear here:
    <br>
    <div id="divCharacters">
      <select></select>
    </div>
```

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

我们一起浏览一下代码，看看发生了什么事情。要注意的是我们并不认为这个代码很健壮，可以达到生产标准。它只是想给你一些基本的Ajax技术的了解。没必要给我发邮件说你找到的毛病！

先说第一件事：标记自己。在<body>里，我们只是放了点文本和两个<select>元素。请注意它们不是<form>的一部分。你会发现在Ajax的世界里，表单的作用少了很多。很多时候，你都会把自己的表单UI元素和其他页面元素一起当作顶层的元素看待（当然是在<body>里了）。

我们给第一个<select>元素一个ID selShow。它会成为该页面的DOM的一个节点。你会注意到JavaScript事件处理函数附着在这个元素上。在<select>的数值改变的时候，我们都会调用JavaScript函数，名字是updateCharacters()。所有“魔术”都是发生在这个函数里的。其他元素都没什么特别的。我创建了一个<option>用于列出我自己喜欢的电视剧。

然后就是另外一个<select>元素——嗯，姑且这么叫它。它实际上是一个空的<select>元素，但是封装在一个 里。你会发现Ajax函数最常见的功能就是替换一些 的内容。这也是我们这儿要做的事情。在这里，“服务器”返回的东西（稍后详述）是给<select>元素的标记，用<option>元素填充，列出选出的电视剧的所有角色。因此，当你选择一个剧的时候，演员列表就会得到填充，然后在真实的Ajax表单里，不会刷新整个页面，而只是刷新改变的部分——在这里是第二个<select>元素（或者，准确地说是封装它的 ）。

让我们迅速看一眼模拟服务器。第一个<select>里的每个电视剧都有一个HTML文件对应，这个HTML文件实际上代表了服务器处理。在这儿你必须假装认为服务器重新呈现了响应，结果就是这些HTML页面。它们看上去很像，所以我只拿一个当例子。看看代码清单12-2。

代码清单12-2 服务器返回的样例：最伟大的电视剧（《巴比伦5》）的角色列表

```
<select>
  <option>Delenn</option>
  <option>Dr. Stephen Franklin</option>
  <option>G'Kar</option>
  <option>John Sheridan</option>
  <option>Kosh</option>
  <option>Lita Alexander</option>
  <option>Londo Mollari</option>
  <option>Marcus Cole</option>
  <option>Michael Garibaldi</option>
  <option>Mr. Morden</option>
</select>
```

如我们所说的，它除了给第二个<select>元素的标记之外没别的东西。

现在我们来看看真正工作的部分：所有JavaScript函数。首先是updateCharacters()函数。如果你已经有了一些Ajax的开发经验，那么基本的代码会像印在你眼睛里一样，因为它是Ajax函数的原型。让我们把它分解开吧，可以了吗？

我们需要的第一个东西，如大家所预期的一样，是一个XMLHttpRequest对象，可能大多数人都已经知道了，它是Ajax代码的核心对象。这个对象是Microsoft首创的，（信不信由你！）只是一个socket的代理。它有区区几个（很少）方法和属性，但这也是优点之一。它实在是一个很简单的东西。

要注意这里的分支逻辑。我们发现在IE里获取XMLHttpRequest对象的方法和其他浏览器不太一样。还有，在你挽起袖子准备对Microsoft发起反对的怒火之前，先记住是Microsoft发明了这个对象，而世界的其他部分只是跟随的。所以，虽然如果Microsoft的开发人员能更新他们的API，以匹配其他人的接口是个好事，但我们仍然不能说这个分支逻辑是Microsoft的开发人员造成的！其他人其实也可以很容易复制Microsoft的接口，所以我们别在这事上扔石头了……我们都在同一间玻璃屋子里。

最新的消息：IE 7的XMLHttpRequest实现是本生对象，甚至可以在禁用ActiveX的时候使用！这就意味着我在这儿说的代码分支，理论上是不必要的。不过，我读到过很多有关这方面的问题，并且怀疑IE团队是否真的实现了一个本生的版本，还是很聪明地把ActiveX版本封装在了JavaScript外表下。还有一些有关这方面的性能的讨论。所以，今天我的建议是，仍然使用已经证明有效的老办法。实际上，大多数情况下你可能会使用某些库来编写Ajax功能，所以很大程度上你是和这些考虑隔离开的，因此也并不那么关心它的实现是真的本生还是假的。不过，这些信息自然还是值得一提。

现在说XMLHttpRequest已经是事实的标准可能正是时候了。它还在争取成为真正的W3C标准，不过现在还不是。假设任何“现代”的浏览器（只要是在一两个版本之内的桌面浏览器）都支持这个对象是靠谱的。更受限的设备（比如PDA、手机等类似的东西）可能还没有这个对象。但不管怎么说，XMLHttpRequest的确是相当普及的代码了。

继续阅读代码，在拿到XMLHttpRequest对象实例之后，我们在全局的页面作用域里，把它的引用赋与xhr变量。让我们先思考一小会儿。如果超过一个onChange事件几乎同时产生会怎样？实际上，第一个会被丢掉，因为会派生出一个新的XMLHttpRequest对象，然后xht会指向它。更糟糕的是，因为XMLHttpRequest是异步的，如果第一个请求的回调函数正在执行，而引用是空，那么就可能会发生问题，这里的问题实际上就意味着回调函数因为引用了一个空对象而抛出错误。如果这个情况还不够糟糕，那么这个现象还只会在几种浏览器中发生，而不是全部浏览器（当然，我的研究表明，几乎所有浏览器都会抛出错误），所以这个问题甚至都无法稳定复现。别忘了，我说过这个不是健壮的、生产质量的代码！这个例子解释了为什么人们常说，在很多时候，派生一个新的实例并且发送一个新的请求实际上是完全可以接受的。

考虑一下常用的肥客户端。你能举出这么一个例子来么？就是这样的：你发起了一个新的事件，这个事件实际上取消了前面一个正在执行的事件？比如，在你的浏览器里，你能在装载一个页面的时候，点击主页按钮，导致一个页面没下载完就开始现在一个新的吗？是的，你能，而实际上那个就是用同一个引用变量发起一个新的Ajax请求的时候发生的事情。这并不算罕见的应用工作方式，有时候还是完全正确的呢。

下一步我们需要完成的是告诉XMLHttpRequest实例应该使用什么回调函数处理函数。Ajax请求有一个定义得很好的、明确的生命周期，这点和其他HTTP请求一样（记住，我们今天要讲的就是一个Ajax请求！）。这个过程定义为准备状态之间的转换过程（所以才有那个属性名：onreadystatechange）。在这个生命周期中间的某个地方，会调用你指定为回调处理函数的JavaScript函数。比如，在请求开始的时候，函数将被调用。当请求逐块返回给浏览器的时候，至少大部分浏览器（很不幸的是，IE是个例外）里，都会对每个块调用一下你的回调函数（想象一下，如果服务器端没有复杂的排队和回调代码，你能怎样修改你的超酷的状态条）。在我们这个例子里，最重要的是，当请求结束的时候，会调用我们的回调函数。我们一会儿来看看这个函数。

下一步可能是相当明显的：我们需要告诉该对象我们想调用的URL是什么。我们通过该对象的

open()方法来做这个事情。这个方法接受3个参数：执行的HTTP方法，要连接的URL和我们是否希望调用是异步的（true）还是同步（false）。因为这是一个很简单的例子，每个电视节目都使用自己的HTML文件作为伪服务器。HTML文件的名字就是<select>元素值加上.htm扩展名。所以，用户每选择一个选项都会调用一个不同的URL。这很明显不是一个真正的方案的运转方式。真实世界的做法很可能是调用同一个URL，然后里面有些参数声明所选择的元素。不过我们在这儿需要作出一点点牺牲，以保持这个例子相对简单，以便我们不用做任何服务器端的事情。

HTTP方法可以是任何标准的HTTP方法：GET、POST、HEAD，等等。大多数时候，你需要传递GET或者POST。URL自己是自解释的，只有一个细节例外：如果用GET，你必须自己构造查询串，并且附着在URL上头。这是XMLHttpRequest的一个缺点之一。你必须对收发的数据的格式变换负全责。要记住它实际上只是在socket对象上的一个很薄的封装。这里也是任何Ajax工具包所能体现自己便利的地方，在本章稍后我们制作聊天应用的时候，我们会用到Mootools库，那个时候你就会了解这一点了。

当然，如果你实际上需要发送一些内容，你可以在这儿做。你可以给这个方法传递一个字符串当数据，然后这个字符串就会放在HTTP请求体里面发送。很多时候，你可能想发送真正的参数，然后可以构造一个传统形式的var1=val1&var1=val1这样的查询串，只是没有开头的问号。另外，你可以传递一个XML DOM对象进来，然后它会被串行化成一个字串并发送出去。最后，你可以发送任意想发送的数据。如果一个逗号分隔的列表能满足你的需求，你可以发送它。任何非参数字符串的东西都需要你自己处理，参数字符串将会像你预期的那样变成请求参数。

到现在为止，我们已经描述了如何发送请求。挺简单的，不是吗？好的，下一部分是既可以变得更简单，也可以变得更复杂的部分，在我们的例子里，它是前者。我指的是回调函数处理函数。我们的回调函数处理函数做的事情很少。首先，它检查XMLHttpRequest对象的readystate。记得我说过这个回调会在请求的生命期里头调用很多次吗？是的，在生命期中的每个循环事件里，readystate的码值都会变化。在我们的例子里，我们感兴趣的码值是4，表示请求已经完成。请注意！我可没说“成功”完成！不管服务器返回是什么，readystate都会是4。因为这是一个简单的例子，我们不会在乎服务器返回什么。如果收到一个HTTP404错误（没有找到页面），那我们在这个例子里并不关心。如果是HTTP 500错误（服务器处理错误）发生，我们还是不关心。这个函数会在任何情况下都做处理。我重复一下我前面说的：这个不是一个生产级别的例子！

在请求结束调用回调函数的时候，我们只是把页面上ID为divCharacters的 元素的innerHTML属性设置为返回的文本。在我们的例子里，返回的文本是填充<select>的标签，而最终的结果就是第二个<select>被填充成所选的电视剧的角色。

看上去不错，不是吗？

提示 布置一个小的趣味联系，只是想告诉你自己实际上发生的事情，我建议在第一个<select>里添加一两个你自己喜欢的剧集，然后创建合适的HTML文件以呈现第二个<select>需要的标记。

我需要指出的另外一点是，在前面的章节里，比如在第5章里，你看到我说的东西实际上是Ajax。但是，在看到这个简单的小例子之后，你可能会有些困惑。让我在这儿澄清一下。还记得我强调过Ajax更多意义上是一种方法而不是一个实现。没看到XMLHttpRequest的出现不意味着不是Ajax。在我看来，

动态<script>标签技巧就是本章你能看到的最能代表Ajax的东西了。Ajax的意义在于客户端做更多的事情，而用不着服务器每次都呈现全部视图。因此，虽然本章并非你在本书中第一次看到Ajax的应用，但它的确是代表着大多数人理解的Ajax的意义的第一个例子。我想虽然这是概念/语意上的一个玩笑，但是还是值得一提。

如果这些东西看上去很像是鼓吹Ajax是什么以及Ajax多么多么好的洗脑企图，那我就要告诉你，从某种意义上来说，它真的好！对某些人来说，Ajax可能是个坏主意，但是他们大多都是只看问题不看优点的人。因为我认为Ajax更多的是哲学和思维过程，而不只是具体的技术，所以跟你推销它下层的概念是很重要的事情。只是给你看一些代码，然后希望你同意是不够的！Ajax令Web有可能完成很多人期待的许多事情，所以我认为Web开发人员应该理解其重要性，并且善用其下层的概念。

12.4　JSON

还记得我说过Ajax并不要求与服务器之间通过XML传递数据吗？事实是，XML甚至都不是最常用的数据格式。这个区别通常都会导致一个叫JSON的东西的使用，或者说是JavaScript对象表示法，我在第2章里介绍过。

我认为，JSON的缩写有些误导，因为，虽然它可以表示一个对象，但通常它表示的都不是对象，不过这点实在只是命名的问题。它蕴含的基本概念是：它是一种用于构造返回给调用者的数据的方法。

JSON是一种轻量的、系统无关的数据交换格式，容易被人类阅读，容易被计算机分析以及也容易被计算机生成。任何有C族语言（包括Java和JavaScript）经验的程序员都会很快熟悉它。它构建在两个在编程世界里相当通用的的概念之上：一个是名字/数值的集合（映射、键字列表、关联数组等）和一个是有序数值列表（列表或数组）。

好了，计算机科学的泛泛介绍已经足够多了！让我们看看JSON看上去像什么。

```
{"firstName":"Frank","lastName":"Zammetti","age":"34"}
```

真的吗？就是这样？我真希望能用我“高深”的知识给你更深刻的印象，告诉你JSON还蕴含更多的东西，但是我不能，因为这就是JSON所有的东西！如你所见，它看上去像Java里面的一个数组，但是不尽相同，因为每个分隔符之间定义了两个元素。冒号左边的项是键字，右边的是数值。每个键字/数值对都用一个逗号分隔，然后所有东西都用一对花括弧包围。它很简单！

最酷的地方是当你想在JavaScript里处理JSON返回的时候。你需要做的只是：

```
var json = eval("(" + myJSONString + ")");
```

这么做的结果是（假设myJSONString包含一些有效的类似前面讨论的那样的JSON数据）创建一个新的变量，就是json，你可以在自己的脚本里使用它。然后，如果你想获取返回中的名字（first name）部分，你只要这么做就行了：

```
alert(json.firstName);
```

没错，就是这样！这里发生的事情是eval()调用创建了json变量，给予它返回的数值。json变量是在JavaScript中的关联数组。所以你可以像访问其他关联数组那样访问里面的成员。很方便吧！

虽然本章的项目没有这么做，但是你也可以向服务器发送JSON数组。如果你访问http://www.json.org，你会发现一些用于不同语言的库来帮助你生成和分析JSON数组。当然，我们这里实际上只是说说生成和分析一个字符串。重要的是，它绝对不是造火箭，稍后我们看到代码的时

候你就会更明白了。不过，要记住：虽然JSON在核心上是很简单的，但是因为我们可以在元素中嵌套元素，并且使用元素的数组，那么有时候手工生成JSON数组也会相当痛苦。比如，想象一下把整个对象图串行化成JSON。尽管JSON自身理解起来可能并不那么难，但是要编写生成它的代码可能还是有些难度的。在这种场合下，看看一些库或者是现成的代码是否能帮助自己降低实现难度会更明智一些。

我应该强调一下的是，JSON是通用目的的消息格式，正因为此，你可以在Ajax工作范围之外同样有效地利用它。很多人实际上用JSON已经比XML更多了，因为它的信息没那么冗长，而同时也同样容易为人们阅读。我相信我们都看过那些很难理解的“坏”XML代码。同样的事情也会发生在JSON上。比如，在我前面的一本有关Ajax的书里，我列出了一个基于Ajax的游戏所返回的JSON片段，形如：

```
{"dm":"false","pn":"Aragorn The Weak","ht":"100","hp":"1","gp":"10","iu":"true",
"vu":"true","di":"false","wn":"false","ec":"false","mo":"o","es":"false","md":"g
gggggssss[[gggggggggggggggggg([[[[[[[[[[gggggggg[gggggggGgg[[[[ggggggggg[[[[
[(ggggggg[[[[[[[[[[[[[[[[[[[g^ggggggg[[[[ggggggggg[[gggggggggggfgggggggggggfgggg
gggggggg"}
```

这些数据看上去挺不好理解的。元素的名字显然不是给人类阅读的。虽然你可以通过了解游戏来猜出其中不少的元素的含义（欢迎购买该书：*Practical Ajax Projects with Java Technology*，ISBN：1-59059-695-1），但你可能猜不出全部元素的含义。这么做的原因是，对于游戏来说，你通常希望速度越快越好。因此，我们的选择是要么让JSON消息容易为人类阅读，但是实际上人们几乎不会关心它，除非是为了调试，另外一个方面是把数组做得尽可能小，这样(a)不用花太多时间去生成或者分析，(b)也不用花过多时间在网上传递。

当然，大多数应用都不像游戏那样对时间很敏感，所以我总是建议把自己的JSON（或者说是XML，一样的概念！）做得尽可能是人类可读的。比如，用displayMessage代替dm，还有playerName代替pn就是我的建议。

到现在为止，你已经准备好使用JSON了，信不信由你！开始使用它吧，充分发挥它的潜力！

12.5 Mootools

既然你现在已经有了如何制作Ajax应用的足够的基础知识了，那就让我们看看优秀的Mootools库（http://mootools.net）是如何把这一切变得更容易、更干净以及更少错误（对比我们在代码清单12-1里面的简单例子）。

既然叫Mootools，你可能会觉得它发明了很多嘲讽，但这么说就距离它应受的尊重太远了！Mootools是一个轻量的、模块化的JavaScript框架，覆盖了大多数现代的JavaScript开发需要的基本方面。Mootools是由一堆模块构造起来的，包括核心模块（基本Mootools）、本生模块（你可以找到基本的JavaScript扩展和工具）、远程模块（类似Ajax的那些东西）和特效模块（可以找到UI FX等方面的东西）。

Mootools最酷的地方之一就是它的下载页面。如果登录上去，你会看到一个列出所有可用的模块和插件的页面。你只需要选择自己需要的东西，然后就会自动生成仅包含你选择的部分的自定义的包。这个特性极其有用，并且确保你只接收自己需要的代码，这种做法甚至说是给予“轻量”这个词最新

的含义！更重要的是，在选择模块的时候，任何其他依赖的模块都会自动被选中。要说，即使你不想下载Mootools，Mootools的下载页面也够有趣的。

说明　本章里的Mootools.js包括写作此书的时候可用的所有东西（Mootools v1.0）。所以如果你想试验一些东西，如果你不想做自己的设置，那你就没必要去下载自己的客户化包——只要在下载源代码的位置下载这个文件，然后就可以开始试验了。

Mootools显然有很多东西，不幸地是，聊天应用只是用到些皮毛。所以，还是让我们马上就开始了解些皮毛吧。

用Mootools发起一个Ajax请求，你要做的只是：

```
new Ajax("<URL>", {
  postBody :
    Object.toQueryString(
      { "parm1" : $("someID1").value, "parm2" : $("someID2").value }
    ),
  onComplete : function(inResponse) {
    // Do something.
  }
}).request();
```

就是这样！初始化一个新的Ajax对象，给它的构造函数传递几个参数，然后你就进入赛道了！第一个参数是需要调用的URL。第二个参数是postBody，它是会POST到URL上的内容。你会注意到Mootools扩展了一些JavsScript对象，在本例里面是Object，提供给我们一个toQueryString()方法。使用这个方法，我们可以传递给它一个参数名和参数值的列表，然后就会为我们创建合适的查询串。你也会注意到$()操作符的使用，你在其他章里已经看到过了。这个操作符给出我们传递给它的DOM ID的引用。Mootools自己也提供了一个这个函数的实现，你可以在这里看到我们用它获取一个文本字段的值（我们这么猜测是因为我们在找value属性）。

最后一个参数是onComplete，它是一个在请求成功返回的时候会被执行的JavaScript函数。你可以在这儿做任何事情，并且它不必像我在这里写的那样是内联的。你也可以引用一个已经在某个地方存在的函数。不管怎样，所有这些就是用Mootools做一个Ajax调用所需要的东西！

其他可用的选项包括evalScripts，它会计算返回的任何JavaScript；update，它可以自动向指定的页面元素中插入返回值；还有evalResponse，它会执行整个返回值。这些都是一些非常有用的函数，可以不用自己写这些东西绝对是一大享受！

你绝对应该翻阅一遍Mootools的文档，（说实话，相当棒！）看看都有什么可用的。它比其他的库出现的时间晚一些，但是它绝对是从其他库里学到了一些教训，因为它做了很多正确的事情。

既然我们已经了解了我们做事的所有前提条件。那么，东风吹，战鼓擂，让我们开始聊天应用的编码吧！

12.6　聊天应用的预览

开始之前，让我们看看这个应用。图12-3显示了初始的登录屏幕。这个应用实际上有两个不同的登录屏幕（一个是给用户的，一个是给客服的），但是它们看上去很像（只有提示信息有所区别）。

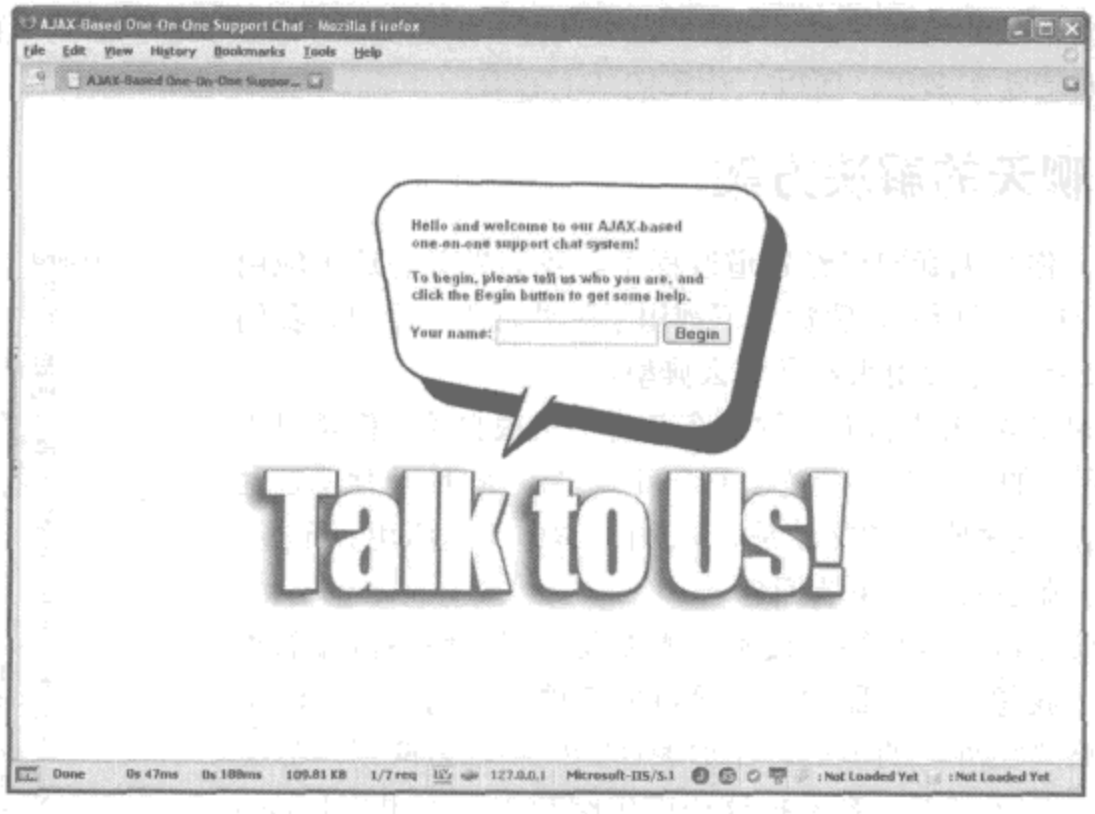


图12-3 聊天系统的登录窗口

一旦登录了，你就可以看到聊天系统的主屏幕，如图12-4所示。这个屏幕包含若干元素。首先，在接近顶部的位置，我们有一个问候语，给出一些与人接触的感觉。它的左边一点是一个短的聊天头，给客户一些变化的感觉。下面是聊天区域，聊天的内容会出现在这个位置。它的左边是聊天者可以操作的菜单，比如复制聊天记录或者是退出聊天等。在最下面是持续更新的当前日期和时间。你知道用户为了等待我们的帮助会等多久，聊天系统也不例外，所以让他们把握时间是一个挺有用的功能（不过，在我们这个实现里，除非有客服在线并处于空闲，否则客户是无法登录的，不过我们先忽略这个因素，以便让用户跟踪时间有意义）。

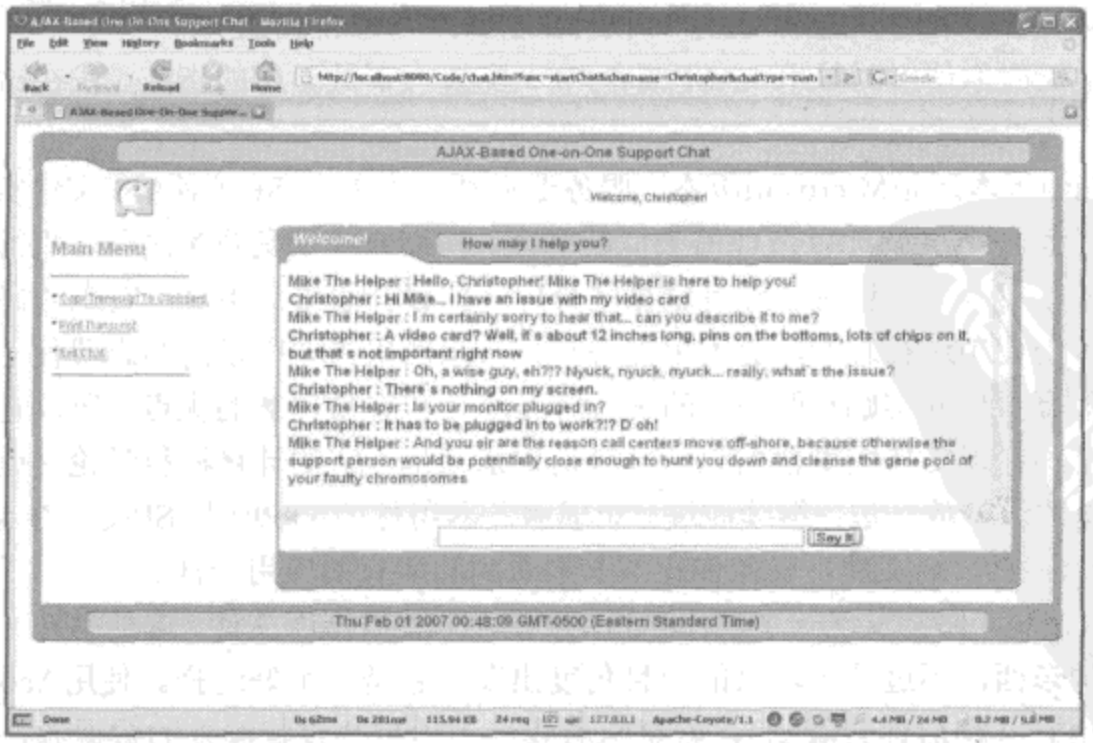


图12-4 主要的聊天窗口，所有“行为”（对话）发生的地方

这两张图相当全面地覆盖了该应用的大部分功能。如我前面所说的，它里面的东西并不多。不过，在它们背后的东西就实在多了，所以，让我们马上看看。

12.7　剖析聊天的解决方案

到现在为止，你毫无疑问已经知道过程了。我们先看看这个应用的目录结构，如图12-5里所示，感觉一下都用了哪些部件。不过，我们这最后一个应用比起其他应用来不是那么典型。

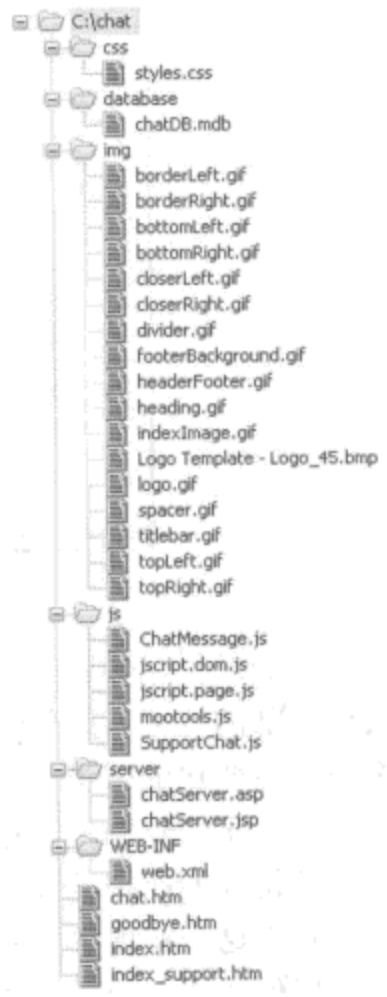


图12-5　聊天应用的目录结构

如果你是Java的Web开发人员，这些东西看上去很自然。但是对于那些不是Java世界里的人，就让我先解释一下，WEB-INF目录意思是这是Java的Web应用。这里唯一的文件是web.xml，它定义了应用。不过，这个应用也适用于那些Microsoft的拥趸。

因为本书主要是讨论客户端的技术，所以我不想在服务器端的事情上花太多口舌。我想把服务器端做得尽可能简单、容易，这实际上意味着我就会忽略“额外”的步骤，比如编译代码和部署应用。如果我们这么看问题，那么可以很容易把这个应用跑起来。如果你是Java开发人员，那么只要把整个目录复制到常用的应用服务器的应用部署目录即可。比如，如果你使用Tomcat（顺便说一句，我非常建议用这个），那就把它复制到/webapps目录里。启动服务器，然后你就马上可以访问应用了。（它的URL看上去会像http://localhost:8080/chat，这里假设你复制了chat目录，并且假设Tomcat安装是监听8080端口的。）

如果你是使用Microsoft技术的开发人员，你可能更熟悉IIS。如果这样，要把这个应用跑起来，你只要把该目录拷贝到INetpub/wwwroot目录下就可以了。

实际上不管用哪个应用服务器都还有一个配置步骤，不过相当简单。具体来说就是根据你使用的技术（ASP或者JSP）配置一下数据库的指向。我在这儿说这些有些超前了，不过我们一会儿就会谈到。

不管我们说Java版本还是Microsoft版本，服务器端都是用单个JSP或者ASP文件实现的。意思是说不用编译，也不用考虑classpath以及任何类似的东西。在你向我扔臭鸡蛋之前，我得承认这种做法在现实世界里可能并不可取。这种做法没有遵循区分应用的设计思维。不过，它也的确代表着你可以在区区几秒钟之内就把应用运行起来，而不用担心有什么东西出错。所以请容忍我使用这样的结构，我老实承认它不可能赢得任何奖励！但是它的确有容易运行和容易理解的优点，所以它在这儿能做得挺好。我们也没理由不在现实环境里使用它——只是你别写基于它的计算机科学论文就是了！

好了，把虚伪的道歉放一边，继续前进。组成服务器组件的ASP或者JSP文件在server目录里。在css目录里是单个styles.css文件，就像你在本书中看到的所有的应用一样。

在database目录里，你会发现一个叫chatDB.mdb的Access数据库文件。这是另外一个在现实世界里你可能不会选用的东西，不过从一本书的项目角度出发，它是一个好选择。使用Access数据库，然后在代码里直接引用它（稍后你会看到），就意味着我们不用担心设置额外的数据源，这样也会让架设

系统、运行例子更快些。

在img目录里是本应用的图片资源。在js目录里是所有JavaScript文件，包括Mootools和一些这个应用使用的类。你还会发现我们用了两个来自第3章的包：jscript.dom包和jscript.page包。

最后，在root目录里是4个HTML文件。chat.htm是应用的主体。index.htm是你（以用户身份）首先接触聊天系统的文件。index_support.htm是客服的入口。goodbye.htm是在你退出聊天的时候看到的页面。

好，我这里做了挺多铺垫了，让我们进入细节吧！虽然在其他项目里，我通常都是从HTML和样式表文件开始，但是这次，让我们直接进入JavaScript。我会帮助你理解前面说的几个有关声明应该使用哪个版本的提示。

12.7.1 编写SupportChat.js

包含在SupportChar.js文件里面的SupportChar类是主要的客户端类，它构成了应用的主体。如图12-6的UML图表所示，它包含4个字段，1个用于判断版本，3个是透明的（包含在应用执行过程中使用的数值）。

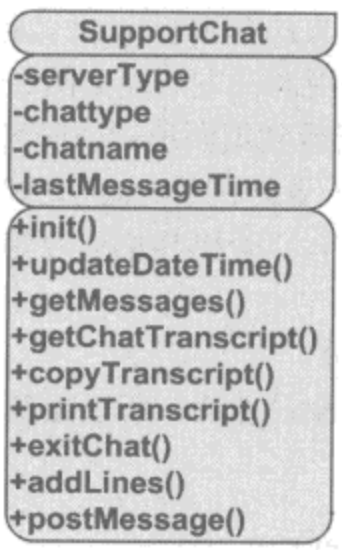


图12-6 SupportChat类的UML图表

- serverType：用于判断我们使用的是ASP版本还是JSP版本的变量。其值要么是asp，要么是jsp。用于正确引用服务器端合适的页面。因此，如果你想把应用放到IIS里面运行，你需要在这儿把它的数值改成asp。要在Tomcat或者其他服务脚本容器里运行，就把这个值设置成jsp。这就是针对不同版本要做的设置。
- chattype：判断聊天对象是客户还是客服。两个可能的值是customer和support。
- chatname：聊天者登录后的聊天名。
- lastMessageTime：保留引用从服务器请求新消息的最后的时间。我们稍后再看看它的机制，我们这儿说该机制运行必须用此变量就够了。

类里面有不少方法，我们逐个看它们。

1. init()方法：启动

当主页面chat.htm转载到浏览器里的时候会调用init()方法：

```
this.init = function() {

  // Get the chatter type
  this.chattype = jscript.page.getParameter("chattype");

  // Get the chatter's name
  this.chatname = jscript.page.getParameter("chatname");

  // Insert greeting.
  $("spnChatname").innerHTML = this.chatname;

  // Set a timer to fire to update the time at the bottom.
  setTimeout(updateDateTime, 0);

  // Set a timer to look for new messages on the server
  // (once every 2 seconds).
  setTimeout(getMessages, 2000);

} // End init().
```

首先，这个方法获取聊天者的类型。方法是用jscript.page对象的getParameter()方法，我们在第3章里写了这个方法。聊天者的名字也是用同样方法检索的。一旦完成了这些，我们就往spnChatname 标签插入聊天者的名字，然后你就看到页面顶部的问候。

然后，就设置两个超时。第一个用于更新在页面底部的时间。注意，这里的时间间隔设置为0，意思是它会立即运行，这也是我们想要的。[①]否则我们就要等一秒钟才能看到时间显示在底部。第二个超时是周期地从服务器请求新的信息。

2. updateDateTime()方法：运行时钟

第一个超时设置调用了updateDateTime()方法，所以让我们看看它。

```
var updateDateTime = function() {

  $("pDateTime").innerHTML = new Date();
  setTimeout(updateDateTime, 1000);

} // End updateDateTime().
```

这个几乎是你能猜到的东西。我们把一个Date对象的字符串形式插入到了pDateTime元素里面，然后再次设置超时为一秒之后。很简单嘛！

3. getMessages()方法：与服务器交谈

第二个超时调用getMessages()方法，当然了，这儿的东西看上去要多些。

```
var getMessages = function() {

  new Ajax("server/chatServer." + chat.serverType, {
    postBody :
```

① 你可能会问，为什么不直接调用函数，然后设置超时？我的回答是：这只是另外一种做法。两种方法都很好。我自己感觉我的做法更干净些，因为它只用一行代码，而不用两行，而且重复调用函数的机制和一开始就调用函数的机制是一样的。这个实现上没有正误之分，只是个人选择罢了。

```
      Object.toQueryString(
        { "func" : "getMessages", "chatname" : chat.chatname,
        "lastMessageTime" : chat.lastMessageTime }
      ),
    onComplete : function(inResponse) {
      // Parse JSON response.
      var messageJSON = eval("(" + inResponse.trim() + ")");
      chat.lastMessageTime = messageJSON.lastMessageTime;
      var lines = new Array();
      // Iterate over messages received.
      for (var i = 0; i < messageJSON.messages.length; i++) {
        var nextMessage = messageJSON.messages[i];
        // Construct a new ChatMessage and add to array.
        var chatMessage = new ChatMessage();
        chatMessage.setTimestamp(nextMessage.timestamp);
        chatMessage.setChatname(nextMessage.chatname);
        chatMessage.setMessage(nextMessage.message);
        lines.push(chatMessage);
      }
      // Display new message lines.
      addLines(lines);
    }
  }).request();

  // Kick off the timer again.
    setTimeout(getMessages, 2000);

  } // End getMessages().
```

在这里，我们有了第一个实际的Ajax使用以及第一次Mootools的调用。如你所见，URL是用我们前面描述的serverType字段的值进行构造的。然后我们就使用Object.toQueryString()方法构造POST内容体。那些内容有：参数func，告诉服务器应该执行哪个功能；chatname是提交信息的聊天者的名字，它是存储在SupportChat类的chatname字段中的值；lastMessageTime，它是最后一个信息请求发生的时间。聊天对话双方之间的任何发生在lastMessageTime之内的信息都会被返回。因为我们只是每3秒钟检查一次信息，所以在这段时间里很可能会有多条信息贴上来，因此我们会一次地把它们都返回，声明一下。

我们还定义了一个在数据返回的时候要执行的一个内联函数。这个回调函数首先分析返回的JSON数组，然后遍历收到信息（当然，也可能是一条都没有，我们这个实现不会因此出错）。然后对出现的每条信息，我们都构造一个ChatMessage对象，它实际上就是一个DTO，代表一条信息。你很快就会看到那个类，它其实只是一个消息属性的存储容器，没别的。消息属性包括timestamp（何时提交的）、聊天者的名字（chatname）以及消息本身。

在创建并填充了ChatMessage之后，程序就把它压入lines数组，它是在循环开始遍历消息之前创建的。在遍历结束之后，我们把数组传递给addLines()方法，它负责具体显示所有数组里面的消息。

4. addLines()方法：显示一些消息

谈到addLines()方法，让我们先看看它：

```
var addLines = function(inLines) {

  for (var i = 0; i < inLines.length; i++) {
    var message = inLines[i];
    var styleClass = "cssChatterText";
    if (message.getChatname() != chat.chatname) {
      styleClass = "cssSupportText";
    }
    htmlOut = "<div class=\"" + styleClass + "\">" +
      message.getChatname() + " : " +
      message.getMessage() +
      "</div>";
    $("divChat").innerHTML = $("divChat").innerHTML + htmlOut;
  }

} // End addLines().
```

这段代码相当直白。它把ChatMessage对象传递进来的数组看作inLines数组的一部分开始遍历。对于数组中的每个元素，它先检查消息是聊天者贴的还是聊天对象贴的。styleClass变量保存着针对两种不同消息的CSS类，这样我们可以令聊天双方的消息颜色不同。然后，针对每条消息都会创建一个 ，然后把聊天者的名字和他说的话插入其中做它的内容。最后，这个 附加在divChat这个 的后头，然后就可以在屏幕上看到消息了。

5. postMessage()方法：告诉世界

你已经知道如何从服务器检索消息并展现出来。等式的另外一边是张贴信息，那是通过postMessage()方法实现的。

```
this.postMessage = function(inLines) {

  new Ajax("server/chatServer." + chat.serverType, {
    postBody :
     .Object.toQueryString(
        { "func" : "postMessage", "chatname" : chat.chatname,
        "messagetext" : $("postMessage").value }
      )
  }).request();
  $("postMessage").value = "";

} // End addPostMessage().
```

到了学习点Ajax的时候了！在这里，我们做的事情实际上和你在getMessage()里看到的一样，但是这次func参数的值是“postMessage”，合理吧！在这里，我们提供了一个messagetext参数，给它传递postMessage文本框的内容。最后，我们清空那个文本框，这样用户可以继续敲入新的消息。这就是这个方法做的全部事情。

聪明的读者可能会问：聊天者是如何看到他们自己的消息的？在这儿显然是没有处理显示这些信息的代码。答案是聊天者自己的信息将作为下一个getMessage()循环的一部分进行显示。没错，这里就意味着聊天者从提交信息到他们在自己的屏幕上看到自己说的话之间会有最多3 s的延时。这样也许还算凑合，不过可能最好的还是让用户立即在自己的屏幕上看到。

在这个类里面还剩下的东西就是处理3个菜单项的函数和处理退出的函数，让我们现在就看看它们。

6. getChatTranscript()方法：为了后代

首先是getChatTranscript方法。这个方法被copyTranscript()和printTranscript()内部使用，后两个方法由对应的菜单项调用。getChatTranscript()收集所有聊天文本，返回给调用它的人。

```
var getChatTranscript = function() {

  // Get the text of the chat.
  var chatTranscript = $("divChat").innerHTML;

  // Now we need to go through the text and remove the HTML components so
  // we are left with nothing but text.  Then, for each line, we make sure
  // there's no trailing or leading whitespace, and we build up a string
  // containing all the lines, separated by linebreaks.
  var transcriptLines = chatTranscript.split(">");
  chatTranscript = "";
  for (var i = 0; i < transcriptLines.length; i++) {
    if (transcriptLines[i].toLowerCase().indexOf("</div") != -1) {
      transcriptLines[i] = transcriptLines[i].replace("</div", "");
      transcriptLines[i] = transcriptLines[i].replace("</DIV", "");
      chatTranscript += transcriptLines[i].trim() + "\r\n";
    }
  }

  return chatTranscript;

} // End getChatTranscript().
```

它一开始就获取两个divChat 的innerHTML。这里是你聊天的时候看到的所有文本。然后会分裂文本，用的是String类的split()方法，分隔符是大于号或者HTML的结束标签。这样生成的就是一个数组，数组里每个元素都是聊天里的一条信息以及 开始标签组成的行。然后我们用一个循环遍历这个数组。我们检查每个元素的值是否包含“</div”。只有来自聊天的信息才包含这样的标志，所有其他元素都只包含 的开始标签。请注意，我们需要和小写的字符串进行对比。这是因为在IE里，</div会表示为</DIV，那样我们就没法用indexOf()获得匹配了。如果找到了匹配</div的行，那我们就把</div删除。然后我们把元素添加到chatTranscript字符串变量里，用trim()去掉空白，并且在最后添加一个回车换行序列。trim()函数是Mootools给String类添加的，它的作用是删除字符串前后的空白。

这些处理的最后结果是变量chatTranscript包含聊天对话的纯文本信息，每行一条，没有任何空白行，也没有多余的空白。最后返回这个字符串，这个方法的工作就完成了。

7. printTranscript()方法：为了维持你的记忆而破坏地球

我们说过，getChatTranscript()方法被printTranscript()方法使用，它的样子是：

```
this.printTranscript = function() {

  // Get the transcript of the chat.
```

```
    var chatTranscript = getChatTranscript();

    // Open a new window for it.
    var newWindow = window.open();
    newWindow.document.open();
    newWindow.document.write("<pre>" + chatTranscript + "<pre>");
    newWindow.document.close();
    newWindow.print();

  } // End printTranscript().
```

没什么好说的。先调用getChatTranscript()。然后我们打开一个新的窗口，把我们从getChat-Transcript()里拿到的字符串写在一个<pre>标签中，然后我们调用那个窗口的print()方法。然后浏览器和操作系统就从这儿开始接手，就这些。

8. copyTranscript()方法：直接到你的操作系统

最后一个方法是copyTranscript()，它是把聊天记录复制到操作系统的剪贴板上的东西。这儿涉及到的东西肯定比你刚开始想的时候多。

```
  this.copyTranscript = function() {

    // Get the transcript of the chat.
    var textToCopy = getChatTranscript();

    // Branch based on browser capabilities...
    if (window.clipboardData) {

      // Internet Explorer is easy!
      window.clipboardData.setData("Text", textToCopy);

      // Let the chatter know we're done.
      alert("Chat transcript has been copied to the clipboard");

    } else if (window.netscape) {

      // Netscape/Firefox is hard!  First, ask it for permission to do this.
      try {
        netscape.security.PrivilegeManager.enablePrivilege('UniversalXPConnect');
      } catch (exception) {
        alert(exception);
        return;
      }

      // Instantiate a clipboard object.
      var clip =
        Components.classes['@mozilla.org/widget/clipboard;1'].createInstance(
        Components.interfaces.nsIClipboard);
      // Instantiate a transferrable object and set it's "flavor."
      var trans =
        Components.classes['@mozilla.org/widget/transferable;1'].createInstance(
```

```
      Components.interfaces.nsITransferable);
    trans.addDataFlavor('text/unicode');
    // Instantiate a string object and set its value.
    var str =
      Components.classes["@mozilla.org/supports-string;1"].createInstance(
      Components.interfaces.nsISupportsString);
    str.data = textToCopy;
    // Set the value of the transferrable using the string.
    trans.setTransferData("text/unicode", str, textToCopy.length * 2);
    // Finally, put the text onto the clipboard.
    clip.setData(trans, null,
      Components.interfaces.nsIClipboard.kGlobalClipboard);

    // Let the chatter know we're done.
    alert("Chat transcript has been copied to the clipboard");

  } else {

    // Unsupported browser.
    alert("Unable to copy chat transcript to clipboard.\n\nOnly Internet " +
      "Explorer and Netscape-based browsers (including Firefox) " +
      "are supported.");

  }
} // End copyTranscript().
```

如你所猜测的那样，我们首先调用getChatTranscript()获取这次聊天会话的文本。然后，我们检查window对象上是否有clipboardData属性。如果存在，说明它是IE，那就很容易了：调用一下window.clipboardData.setData()，给它传递getChatTranscript()返回的字符串，然后我们就完成了。

然后，如果浏览器是以Netscape为基础的，那事情就有趣多了。如果window对象有一个netscape属性，那么我们要做的第一件事情是请求向剪贴板复制数据的权限。这就是你看到的调用netscape.security.PrivilegeManager.enablePrivilege()。结果是给用户的一个询问，如图12-7所示。

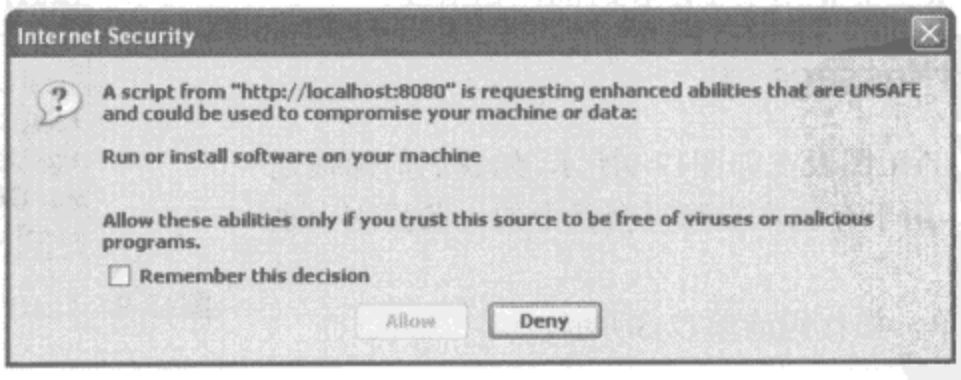


图12-7 Firefox安全权限询问对话框

只要回答“Allow”，或者另外再加上勾上记住设置的对话框的话，内容就会允许复制到剪贴板上。

要注意的是如果因特网安全对话框没有出现，而是出现了一个类似图12-8那样的对话框，那就意味着你需要去设置高级配置选项。在地址栏里输入about:config然后按回车。你就会看到一长串选项。

你要看的是signed.applets.codebase_principal_support。确保这项设置为true。在设置了该选项之后，你就应该看到图12-7那样的对话框了。

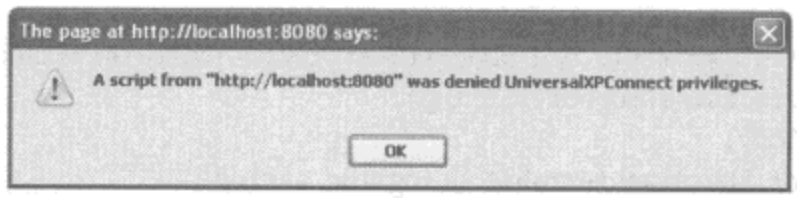


图12-8　如果signed.applets.codebase_principal_support设置为假，你会看到这个

一旦我们有了权限，或者提醒用户了权限被拒绝，我们就可以继续了。第一件事是创建一个剪贴板对象。这就是定义clip变量那行代码的目的。

请注意这里看上去有些可笑的语法，以及下面几个对象的实例化代码，这些是Netscape风格的实例化本生浏览器对象的方法。它基本上相当于指出你需要实例的类的名字，然后请求Components对象给你一个实例。然后，你就像对待大多数其他对象那样使用它。

下一步是创建可转换的对象，由trans变量引用。我们还必须告诉这个对象，它“优选”的是Unicode，这样才能确信我们从聊天会话中正确地得到所有信息。

然后，我们初始化一个String对象，在str变量里面保存一个指向这个对象的引用，然后把它的值设置为我们前面获取的聊天记录。然后我们通过调用setTransferData()把String传递给可转换对象。最后，我们通过调用剪贴板对象的setData()方法把对象复制到剪贴板。

在最后我们还有一个else块，处理那些浏览器不支持的类型。我不知道什么样的浏览器会走到这个分支里，因为我没有全部浏览器可进行测试，但是IE和Firefox是绝对支持的，因此大多数用户应该都没有问题了。

9. exitChat()方法：离开这儿，我说

最后是exitChat()方法。它只是一个简单地弹出确认对话框。如果用户同意退出。window.location会设置为从模拟服务器上取退出页面，它只是一个跟客户说再见的页面。

这些就是客户端代码的主体。下一个需要看看的东西就是我前面说过的ChatMessage类。也用不了多少时间，相信我！

12.7.2　编写ChatMessage.js

ChatMessage类的UML图表在如图12-9所示。如我们前面描述的，它只是一个DTO，用于存储本次聊天会话中的一条提交上来的数据。

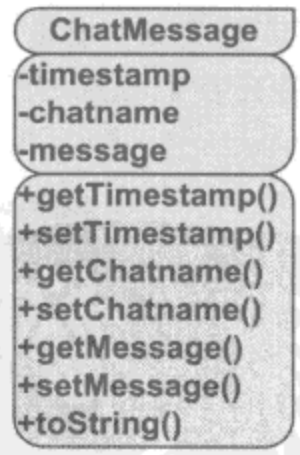


图12-9　ChatMessage类的UML图表

它的代码也很简单，如代码清单12-3所示。

代码清单12-3　ChatMessage类

```
function ChatMessage() {


  /**
```

```
   * The time this message was posted.
   */
  var timestamp = "";


  /**
   * The chatname of the chatter who posted it.
   */
  var chatname = "";
/**
 * The text of the message
 */
var message = "";


/**
 * Mutator.
 *
 * @param inTime The new field value.
 */
this.setTimestamp = function(inTimestamp) {

  timestamp = inTimestamp;

} // End setTimestamp().
```

```
/**
 * Accessor
 *
 * @return The value of the time field.
 */
this.getTimestamp = function() {

  return timestamp;

} // End getTimestamp().


/**
 * Mutator.
 *
 * @param inChatname The new field value.
 */
this.setChatname = function(inChatname) {

  chatname = inChatname;

} // End setChatname().
```

```
/**
 * Accessor
 *
 * @return The value of the chatname field.
 */
 this.getChatname = function() {

   return chatname;

 } // End getChatname().


 /**
  * Mutator.
  *
  * @param inMessage The new field value.
  */
 this.setMessage = function(inMessage) {

   message = inMessage;

 } // End setMessage().


 /**
  * Accessor
  *
  * @return The value of the message field.
  */
this.getMessage = function() {

  return message;

} // End getMessage().


/**
 * Overriden toString() method.
 *
 * @return A meaningful string representation of the object.
 */
this.toString = function() {

  return "ChatMessage : [ " +
    "timestamp='" + timestamp + "', " +
    "chatname='" + chatname + "', " +
    "message='" + message + "' ]";

} // End toString().
```

```
} // End ChatMessage class.
```

看见了吧，我没开玩笑！只有3个字段，timestamp、chatname和message。它们包含了存储某条信息需要的所有数据，以及每个属性的必要的访问方法和修改方法。还有，我们覆盖了toString()，一如你在这本书里看到的那样。这样就允许我们以一种有意义的方式显示一个此类的给出的实例，也是我强烈建议常用的方法，因为它可以让调试更简单。

12.7.3 编写styles.css

到本书的这个时候，你已经多次在其他项目里看到很多样式表了，所以你可能不需要独立地看这个了。我能告诉你的是这个项目的样式表没什么特殊的，每个类的注释基本上就告诉你了它们是什么样式的。

我要指出的一件事请是，针对特定的HTML元素使用样式表，要指出这点是因为在我做的所有项目中，这是第一次。其他项目的样式表通常看上去像下面这样：

```
.cssHeader {
    color : #ff0000;
}
```

你知道这样就定义了一个叫cssHeader的CSS样式类。你可以通过设置元素的class属性把它施加于任何页面元素上。不过，如果我们想给一个页面的所有的<h1>元素都设置一个样式，而且又不想给每个特定的样式单独设置又如何？你可以用下面的声明实现：

```
h1 {
  color : #ff0000;
}
```

这样，页面里任何<h1>元素都会是红色的。在这个项目的样式表里，我们对若干个元素进行了处理，你可以看到下面这样的：

```
/* Style applied to tables. */
table {
  font-size         : 9pt;
  font-family       : arial;
}


/* Style applied to links. */
a:link {
  color             : #6a78a7;
}
/* Style applied to visited links. */
a:visited {
  color             : #6a78a7;
}


/* Style applied to links that are being hovered over. */
a:link:hover {
  color             : #33305b;
```

```
  background-color  : #f0f0f0;
  font-weight       : bold;
}


/* Style applied to h1 elements. */
h1 {
  font-size         : 20pt;
  color             : #000000;
  font-weight       : bold;
}


/* Style applied to h2 elements. */
h2 {
  font-size         : 15pt;
  color             : #8c9cd5;
  margin-top        : 0px;
  margin-left       : 20px;
  margin-bottom     : 0px;
  font-weight       : bold;
}
```

另外一个要提到的事情是链接的样式。对于链接（还有很多其他元素），实际上你可以选择元素的版本。比如，如果你想让所有未曾点击的链接显示为红色，那么你可以设置a:link选择器的color属性。如果想让那些已经访问过的链接显示蓝色，那么可以设置a:visited选择器的color属性。最后，如果你希望所有链接在鼠标悬浮的时候显示绿色，那么你可以设置a:linkhover选择器的color属性。你可以看到这样的样式表给菜单（实际上是链接）鼠标悬浮的效果。

除了这几点之外，样式表是自解释的，所以请看看它们，保证自己知道它们都在干什么。现在，让我们继续。

12.7.4　编写index.htm和index_support.htm

index.htm是应用给客户的入口，index_support.htm是客服登录的入口。我在这儿就只显示index.htm了，在代码清单12-4中。index_support.htm基本上是一样的东西，只有文字上的一点点区别。

代码清单12-4　index.htm文件（基本上也是index_support.htm）

```
<html>

  <head>

    <title>AJAX-Based One-On-One Support Chat</title>

    <link rel="StyleSheet" href="css/styles.css" type="text/css">

    <script src="js/mootools.js" type="text/javascript"></script>
```

```
    <script src="js/jscript.dom.js" type="text/javascript"></script>
    <script src="js/SupportChat.js" type="text/javascript"></script>

    <script>

      /**
       * Called on page load to do some setup.
       */
      function init() {

        var divObj = $('divOuter');
        jscript.dom.layerCenterH(divObj);
        jscript.dom.layerCenterV(divObj);
        $("logonForm").action = "server/chatServer." + chat.serverType;

      } // End init().

    </script>

  </head>

  <body onLoad="init();">

    <div id="divOuter" class="cssDivOuter">
      <img src="img/indexImage.gif" class="cssIndexImg">
      <div class="cssDivInner">
        Hello and welcome to our AJAX-based one-on-one support chat system!
        <br><br>
        To begin, please tell us who you are, and click the Begin button to
        get some help.
        <br><br>
        <form id="logonForm">
          <input type="hidden" name="func" value="logon">
          <input type="hidden" name="chattype" value="customer">
          Your name: <input type="text" name="chatname" size="20">
          <input type="submit" value="Begin" class="cssButton">
        </form>
      </div>
    </div>

  </body>

</html>
```

这个文件和大多数HTML文档一样，以输入样式表和一些JavaScript开始。我们也输入了Mootools，这样我们就可以使用$()函数了。也输入了jscript.dom.js，这样我们就可以使用layerCenterH()和layerCenterV()函数。最后，输入了SupportChat.js，这样就可以使用前面描述的serverType字段。

在<head>里是一个JavaScript函数init()。在装载页面的时候会运行它。它的任务有两个。首先，它用layerCenterH()和layerCenterV()把页面内容居中，这些内容在divOuter 中。它还把页面的

表单的动作指向合适的chatServer版本：JSP或者ASP。

然后就是页面的标记。标记是由单个叫divOuter的 组成的，它就是在init()里居中的 。它包括一些文本和一个HTML表单。这个表单接受聊天者想用的名字。它还包括一个隐藏的字段func，值是“logon”。这个字段告诉我们的服务器进行提交上来的参数值指定的处理。还有chattype参数，在index.htm里，设置为“customer”。在index_support.htm里，设置为“support”，以便告诉服务器不同类型的聊天者的登录。

12.7.5　编写chat.htm

现在我们看看chat.htm，它包含实际的页面布局。你可以在代码清单12-5中看到完整的代码。

代码清单12-5　chat.htm文件

```
<html>

  <head>

    <title>AJAX-Based One-On-One Support Chat</title>

    <link rel="StyleSheet" href="css/styles.css" type="text/css">

    <script src="js/mootools.js" type="text/javascript"></script>
    <script src="js/jscript.page.js" type="text/javascript"></script>
    <script src="js/ChatMessage.js" type="text/javascript"></script>
    <script src="js/SupportChat.js" type="text/javascript"></script>

  </head>


  <body onLoad="chat.init();">
<table width="100%" cellspacing="0" cellpadding="0"><tr><td>

  <table width="98%" align="center" cellpadding="0" cellspacing="0">

    <tr>
      <td>
        <table width="100%" cellspacing="0" cellpadding="0">
          <tr>
           <td width="46"><img src="img/topLeft.gif"></td>
            <td width="64"><img src="img/spacer.gif"></td>
            <td align="center" class="cssTopText"
              background="img/headerFooter.gif">
              AJAX-Based One-on-One Support Chat
            </td>
            <td width="55"><img src="img/topRight.gif"></td>
          </tr>
        </table>
```

```
  </td>
</tr>

<tr>
  <td>
    <table width="100%" cellspacing="0" cellpadding="0">
      <tr>
        <td class="cssLeftColumn">
          <div align="center"><img src="img/logo.gif" vspace="10"></div>
        </td>
        <td class="cssRightColumn">
          <table width="100%" border="0" cellspacing="0"
            cellpadding="0">
            <tr>
              <td class="toptext" align="center">
                Welcome, <span id="spnChatname"></span>!
              </td>
            </tr>
          </table>
        </td>
      </tr>
      <tr>
        <td width="220" valign="top" class="cssLeftColumn">
          <div style="width:220px;height:300px;overflow:hidden;"
            <br>
            <h2>Main Menu</h2>
            <br>
            <div class="cssDividers"><img src="img/divider.gif"></div>
            <br>
                  
    * <a href="javascript:void(0);"
      onClick="chat.copyTranscript();">
      Copy Transcript To Clipboard
    </a><br><br>
          
    * <a href="javascript:void(0);"
      onClick="chat.printTranscript();">
      Print Transcript
    </a><br><br>
          
    * <a href="javascript:void(0);" onClick="chat.exitChat();">
      Exit Chat
    </a><br>
    <br>
    <div class="cssDividers"><img src="img/divider.gif"></div>
  </div>
</td>
 <td class="cssRightColumn">
  <table width="100%" cellspacing="0" cellpadding="0">
    <tr>
```

```
<td class="cssRightColumn">
  <table width="90%" align="center" cellpadding="0"
    cellspacing="0">
    <tr>
      <td>
        <table width="100%" cellpadding="0"
          cellspacing="0">
          <tr>
            <td class="cssLeftColumn">
              <div><img src="img/titlebar.gif"></div>
            </td>
            <td><img src="img/spacer.gif"></td>
            <td width="100%" class="cssHeaderFooter"><img
              src="img/heading.gif"></td>
            <td class="cssRightColumn"><img
              src="img/topRight.gif"></td>
          </tr>
        </table>
      </td>
    </tr>
    <tr>
      <td class="cssRightColumn">
        <table width="100%" cellspacing="0"
          cellpadding="0">
          <tr>
<td class="cssLeftColumn"> </td>
<td>
          <div class="cssChatDiv" id="divChat"></div>
          <span class="cssChatterEntryDiv">
            <table border="0" cellpadding="0"
              cellspacing="0" width="100%"><tr>
              <td valign="middle" class="cssEntry"
                align="center">
                <input type="text" size="62"
                  id="postMessage">
                <input type="button" value="Say It"
                  class="cssButton"
                  onClick="chat.postMessage();">
              </td>
            </tr></table>
          </span>
        </td>
      </tr>
    </table>
  </td>
</tr>
<tr>
  <td>
    <table width="100%" cellspacing="0"
      cellpadding="0">
```

```
                        <tr>
                          <td width="55">
                            <img src="img/bottomLeft.gif">
                          </td>
                          <td class="cssBoxBottom"> </td>
                          <td width="55">
                            <img src="img/bottomRight.gif">
                          </td>
                        </tr>
                      </table>
                    </td>
                  </tr>
                </table>
              </td>
            </tr>
          </table>
          <br>
        </td>
      </tr>
    </table>
  </td>
</tr>
        <tr>
          <td>
            <table width="100%" cellspacing="0" cellpadding="0">
              <tr>
                <td width="55"><img src="img/bottomLeft.gif"></td>
                <td class="cssBoxBottom">
                  <table width="100%" cellspacing="0" cellpadding="0">
                    <tr>
                      <td width="23"><img src="img/closerLeft.gif"></td>
                      <td valign="middle" class="cssHeaderFooter">
                        <div align="center">
                          <div class="cssDividers" id="pDateTime"></div>
                        </div>
                      </td>
                      <td><img src="img/closerRight.gif"></td>
                    </tr>
                  </table>
                </td>
                <td width="55"><img src="img/bottomRight.gif"></td>
              </tr>
            </table>
          </td>
        </tr>
        <tr>
          <td> </td>
        </tr>
</table>
```

```
    </td></tr></table>

  </body>

</html>
```

我会把这些东西留作你的作业，由你自己去阅读。因为它们只是典型的HTML。在输入了样式表和JavaScript之后，你可以看到我们在装载页面的时候调用了chat.init()。变量chat在SupportChat.js里定义，它是唯一的一个SupportChat类的实例。在调用之后，它实际上只是简单的标记。

菜单项用了一个很有用的技巧。链接的href是JavaScript语句javascript:void(0);，这样就导致这个链接在被点击的时候不会执行其通常的动作。取而代之的是我们处理onClick事件并且在ChatSupport里面调用合适的方法。

在页面代码2/3左右的地方，在postMessage文本框的正下方是一个按钮，点击的时候调用chat.postMessage()。当然，这就是聊天者张贴聊天信息的地方。

请花个区区几分钟看看代码清单12-5的代码，确保的确没有什么东西能证明还有别的东西在这些代码里。不过这个现象其实是好事。应用的呈现几乎完全和后面的功能分离了，这正是我们需要的东西。因此，虽然我们不再详细解释这些HTML看上去像逃避责任，但实际上我们要关注的是JavaScript和Ajax，所以我们解释一堆相当简单的HTML并不能帮我们理解JavaScript和Ajax，不是吗？

12.7.6　编写goodby.htm

还有最后一点点标记语言，那就是goodby.htm。实际上这个文件没什么东西，只是在聊天者登出的时候跟他们说再见而已。看看代码吧。不过要是你发现逗留时间超过30 s，那就去冲杯茶，然后清醒点再回来。

12.7.7　创建数据库

然后我们做一件稍微有趣点的事情，这就是数据库结构。我前面说过，为了简化，我用了一个简单的Access数据库文件。这就意味着这个应用只能在Windows系统上运行。这是因为，在JSP版本里，使用的是JDBC:ODBC桥（任何新近版本的Windows都应该带有合适的ODBC驱动），而ASP版本使用ADO，同样也使用ODBC（虽然两个都不需要数据源，因为使用的是直接访问MDB文件）。

数据库里的结构也很简单，总共才两个表，两者之间没有任何键字关系（概念上两个表之间有关系，但是没有使用任何外键之类的东西）。图12-10是一个两个表的图表。

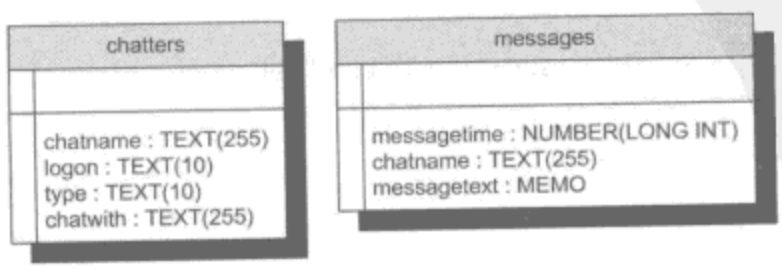


图12-10　如果这个数据库模式把你吓着了，那你就可以找份新职业了

chatters表保存当前正在聊天的聊天者列表。chatname字段是聊天者登录的时候给出的名字。logon字段保存聊天者登录的时候的日期/时间。type字段不是customer就是support，取决于聊天者通过哪个

HTML文件登录。最后，chatwith是聊天者的聊天对象。

message表保存聊天者提交的消息。messagetime字段是消息提交的时间。这个字段用于判断在周期的Ajax消息请求里，应该返回什么消息。chatname字段是提交消息的聊天者的名字。它对应chatters表的chatname字段（这里实际上没有真正的键字关系，只是概念上的）。最后，messagetext字段是消息文本。

这是一个很简单的数据库系统，不需要更多的东西了。请注意，由于变化的速度很快，我认为没必要给这些表加索引。我不认为给这些表加索引会有任何收益。[①] 我也没在表上放任何约束——没有必须的字段等。这些规则，如果有的话，都在与数据库交流的服务器端代码里施加。[②]

在使用ASP版本的应用的时候，有一点需要指出。这个时候，你必须保证运行IIS的用户账号在MDB文件所在的目录上有全部权限，就是说IIS能读写这个目录。如果你在尝试IIS版本的应用，可能你已经知道这点了，但我觉得还是值得一提。如果你试验应用的时候，得到ADO错误0x80004005，或者任何其他的，只要里面有这个，那就请检查权限，因为很可能是这个原因。

现在我们看看正雄赳赳走来的服务器端代码！

12.7.8 编写服务器代码

这部分活儿可能稍微有些棘手，因为我们有JSP和ASP两个版本要评阅。不过，它们在结构和逻辑上都是一样的，只是在语法和其他一些细微的地方有些不同。为了可以合理地审阅这些代码，我会把每个文件都分片，然后描述每一片都做了什么，而不是进入更具体的细节。我相信你可以很容易理解其中一个版本。我会在需要的地方指出细节，但我们会细致地研究一下代码。

1. 启动

启动的时候，两个版本都以变量声明开始。

Asp

```
filename = "C:\Inetpub\wwwroot\Code\database/chatDB.mdb"
```

Jsp

```
String filename = "K:/tomcat5029/webapps/Code/database/chatDB.mdb";
```

filename变量指向Access数据库。如前所述，你需要更新这个变量以便应用可以运行。

然后是一些打开数据库连接的代码。

Asp

```
' variables needed for database work.
Set conn = Server.CreateObject("ADODB.Connection")

' Open connection to database.
conn.Open "DRIVER={Microsoft Access Driver (*.mdb)}; DBQ=" & filename
Set rs = Server.CreateObject("ADODB.Recordset")
rs.CursorLocation = 3
rs.CursorType = 3
rs.LockType = 4
```

① 显然原作者在这里是欠考虑的，如果聊天人数相当多，比如几千人同时聊天，索引还是很有益处的。

② 典型的Java开发人员风格。

Jsp

```
// variables needed for database work.
Connection conn = null;
Statement stmt = null;

// Load JDBC driver.
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
String database = "jdbc:odbc:Driver={Microsoft Access Driver " +
  "(*.mdb)};DBQ=" + filename + ";DriverID=22;READONLY=false}";
conn = DriverManager.getConnection( database ,"","");
stmt = conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
  ResultSet.CONCUR_UPDATABLE);
```

两个版本都用各自技术的典型代码（ASP用ADO，JSP用JDBC）与数据库连接。在ASP版本里，我们还得到一个RecordSet对象，我们会在整个页面里重用它。在JSP版本里，我们得到一个Statement对象，也会在整个页面里使用。

记住这些页面是所有Ajax调用的目标，以及表单提交的目标。还要记住所有那些调用都会提供一个func参数，说明需要执行的服务器端函数是什么。所以，可能你已经想到了，我们下面要做的就是获取该参数的值。

Asp

```
Func = trim(Request("func"))
```

Jsp

```
String func = request.getParameter("func").trim();
```

然后，我们需要构造一个表示这个请求到来的时间戳的字符串形式。这个值会在后面的若干的处理中用到。这个字符串的格式是HHMMSSLLL，HH是小时（00～23），MM是分（00～59），SS是秒（00～59），LLL是毫秒（000～999）。我们对数值小于10（对毫秒而言是小于100）的值做一些字符串处理，使其前缀正确个数的0，以保证我们总是能拿到一个9位数的字符串。

Asp

```
hh = CStr(Hour(Now()))
If Len(hh) = 1 Then hh = "0" & hh End If
mm = CStr(Minute(Now()))
If Len(mm) = 1 Then mm = "0" & mm End If
ss = CStr(Second(Now()))
If Len(ss) = 1 Then ss = "0" & ss End If
ms = "000"
timeStamp = hh + mm + ss + ms
```

Jsp

```
gregoriancalendar calendar = new GregorianCalendar();
String hh = Integer.toString(calendar.get(Calendar.HOUR_OF_DAY));
if (hh.length() == 1) { hh = "0" + hh; }
String mm = Integer.toString(calendar.get(Calendar.MINUTE));
if (mm.length() == 1) { mm = "0" + mm; }
```

```
String ss = Integer.toString(calendar.get(Calendar.SECOND));
if (ss.length() == 1) { ss = "0" + ss; }
String ms = Integer.toString(calendar.get(Calendar.MILLISECOND));
if (ms.length() == 1) { ms = "0" + ms; }
if (ms.length() == 2) { ms = "0" + ms; }
String timeStamp = hh + mm + ss + ms;
```

做完这些事情之后，我们就检查一下得到的func参数。如果它是空，或者是空白——意味着没有传递它进来（或者可能是拼写错误），那么我们就发回一个HTML响应，指出请求了一个未知的函数（这意味着很可能是一个入侵企图，所以我们可以批评聊天者太淘气了，开个玩笑）。

如果我们看到请求参数里有func，就可以判断这是什么函数请求。这就是一堆if条件检查，没别的。

2. 登录

第一个可能的函数是登录，由func值“logon”表示。这是最长、最复杂的代码，但也可以分解成几个逻辑步骤。让我们先看看代码，然后讨论逻辑步骤。

Asp

```
if func = "logon" Then

  ' Processing a logon.
  chatType = Trim(Request("chattype"))

  If chatType = "customer" Then

    ' It's a customer logon.  See if the name is already in use.
customerChatName = Trim(Request("chatname"))
rs.Open "select chatname from chatters where " & _
  "chatname='" & customerChatName & "'", conn
If rs.RecordCount <> 0 Then
  ' Name is already in use, have the chatter select a new one.
  rs.Close
  %>
  <html><head><title>Name already in use</title></head><body>
    I'm sorry but that name is already in use.  Please click
    <a href="../index.htm">HERE</a> and select a new name.
  </body></html>
  <%
   Else
     ' Name is available, so now we have to see if there are any available
     ' support personnel to chat with.
     rs.Close
     rs.Open ("select chatname from chatters " & _
       "where type='support' and chatwith='none'")
     If rs.RecordCount <> 0 Then
       ' Ok, we got someone.  Now log the chatter into the database and
       ' send them to the chat page.
       supportChatName = rs("chatname")
       rs.Close
       conn.Execute "insert into chatters (chatname, logon, type, " & _
```

```
    "chatwith) values (" & _
    "'" & customerChatName & "', " & _
    "'" & timeStamp & "', " & _
    "'customer', '" & supportChatName & "')"
' We also need to mark the support person as chatting with this
' chatter.
conn.Execute "update chatters set chatwith='" & _
  customerChatName & "' where chatname='" & supportChatName & "'"
// Lastly, add a message to the messages table so both chatters see
// who they are chatting with.
conn.Execute "insert into messages (messagetime, chatname, " & _
  "messagetext) values ('" & timeStamp & "', '" & _
  supportChatName & "', 'Hello, " & customerChatName & "!  " & _
  supportChatName & " is here to help you!')"
%>
<html><head><title>Starting chat</title><script>
  function startChat() {
    window.location =
      "../chat.htm?func=startChat&" +
      "chatname=<%=customerChatName%>&chattype=customer&" +
      "chatwith=<%=supportChatName%>"
  }
  </script></head>
  <body onLoad="startChat();">Starting chat...</body>
</html>
<%
          Else
            ' No support personnel available.  Give the chatter the bad news.
            rs.Close
            %>
            <html><head>
            <title>No support personnel available</title>
            </head><body>
              There are currently no support personnel available.  Please click
              <a href="<%=request.ServerVariables("URL")%>?func= ➥
logon&chattype=customer&chatname=<%=customerChatName%>">HERE</a>
          to check for someone again.
        </body></html>
        <%
      End If
    End If

  Else

    ' It's a support personnel logon.  See if the name is already in use.
    supportChatName = Trim(Request("chatname"))
    rs.Open "select chatname from chatters where " & _
      "chatname='" & supportChatName & "'", conn
    If rs.RecordCount <> 0 Then
```

```
    ' Name is already in use, have the chatter select a new one.
    rs.Close
    %>
    <html><head><title>Name already in use</title></head><body>
      I'm sorry but that name is already in use.  Please click
      <a href="../index.htm">HERE</a> and select a new name.
    </body></html>
    <%
  Else
    ' Name is available, so now log the chatter in.
    rs.Close
    conn.Execute "insert into chatters (chatname, logon, type, " & _
      "chatwith) values (" & _
      "'" & supportChatName & "', " & _
      "'" & timeStamp & "', " & _
      "'support', 'none')"
    %>
    <html>
      <head>
        <title>Starting chat</title>
        <script>
          function startChat() {
            window.location =
              "../chat.htm?func=startChat&" +
              "chatname=<%=supportChatName%>&chattype=support";
          }
        </script>
      </head>
      <body onLoad="startChat();">Starting chat...</body>
    </html>
    <%
  End If
End If

End If ' End "logon" function handling.
```

Jsp

```
if (func.equalsIgnoreCase("logon")) {

   // Processing a logon.
   String chatType = request.getParameter("chattype");

   if (chatType.equalsIgnoreCase("customer")) {

     // It's a customer logon.  See if the name is already in use.
     String customerChatName = request.getParameter("chatname");
     ResultSet rs = stmt.executeQuery(
       "select chatname from chatters where " +
       "chatname='" + customerChatName + "'");
     if (rs.first()) {
```

```
    // Name is already in use, have the chatter select a new one.
    rs.close();
    %>
    <html><head><title>Name already in use</title></head><body>
      I'm sorry but that name is already in use.  Please click
      <a href="../index.htm">HERE</a> and select a new name.
    </body></html>
    <%
  } else {
    // Name is available, so now we have to see if there are any available
    // support personnel to chat with.
    rs.close();
    rs = stmt.executeQuery("select chatname from chatters " +
      "where type='support' and chatwith='none'");
    if (rs.first()) {
      // Ok, we got someone.  Now log the chatter into the database and
      // send them to the chat page.
      String supportChatName = rs.getString(1);
      rs.close();
      stmt.executeUpdate("insert into chatters (chatname, logon, type, " +
        "chatwith) values (" +
        "'" + customerChatName + "', " +
        "'" + timeStamp + "', " +
        "'customer', '" + supportChatName + "')");
      // We also need to mark the support person as chatting with this
// chatter.
stmt.executeUpdate("update chatters set chatwith='" +
  customerChatName + "' where chatname='" + supportChatName + "'");
// Lastly, add a message to the messages table so both chatters see
// who they are chatting with.
stmt.executeUpdate("insert into messages (messagetime, chatname, " +
  "messagetext) values ('" + timeStamp + "', '" + supportChatName +
  "', 'Hello, " + customerChatName + "!  " + supportChatName +
  " is here to help you!')");
   %>
   <html><head><title>Starting chat</title><script>
     function startChat() {
     window.location =
       "../chat.htm?func=startChat&" +
       "chatname=<%=customerChatName%>&chattype=customer&" +
       "chatwith=<%=supportChatName%>";
     }
     </script></head>
     <body onLoad="startChat();">Starting chat...</body>
   </html>
   <%
 } else {
   // No support personnel available.  Give the chatter the bad news.
   rs.close();
   %>
```

```
            <html><head>
            <title>No support personnel available</title>
            </head><body>
              There are currently no support personnel available.  Please click
              <a href="chatServer.jsp?func=logon&chattype=customer&chatname= ➥
<%=customerChatName%>">HERE</a>
              to check for someone again.
            </body></html>
            <%
          }
        }

      } else {

        // It's a support personnel logon.  See if the name is already in use.
        String supportChatName = request.getParameter("chatname");
        ResultSet rs = stmt.executeQuery(
          "select chatname from chatters where " +
          "chatname='" + supportChatName + "'");
        if (rs.first()) {
          // Name is already in use, have the chatter select a new one.
          rs.close();
          %>
          <html><head><title>Name already in use</title></head><body>
            I'm sorry but that name is already in use.  Please click
            <a href="../index.htm">HERE</a> and select a new name.
          </body></html>
          <%
        } else {
          // Name is available, so now log the chatter in.
          rs.close();
stmt.executeUpdate("insert into chatters (chatname, logon, type, " +
  "chatwith) values (" +
  "'" + supportChatName + "', " +
  "'" + timeStamp + "', " +
  "'support', 'none')");
%>
<html>
  <head>
    <title>Starting chat</title>
    <script>
      function startChat() {
        window.location =
          "../chat.htm?func=startChat&" +
          "chatname=<%=supportChatName%>&chattype=support";
      }
    </script>
  </head>
  <body onLoad="startChat();">Starting chat...</body>
</html>
```

```
        <%
    }
  }

} // End "logon" function handling.
```

函数以获取chattype参数开始。然后就以这个参数的值做了几个分支。第一个分支是客户登录。这个时候，我们通过获取chatname参数开始。然后我们查询一下数据库，看看这个名字是不是已经被占用了。如果是，就返回一条信息说已经被用了，然后给出一个回到登录界面的链接，这样聊天者可以尝试其他的名字。

如果这个名字还没被使用，那就再查询一下数据库看看是否有空闲的客服。具体是通过在chatters表里搜索type为support的聊天者，并且chatwith字段的值为none来实现的。如果找到这么一个，那么我们就更新chatters表，首先，我们把新聊天者插入该表，然后更新那个有空的客服，把他的聊天对象更新为刚进来的聊天者。最后，在message表里添加一条记录，给两个聊天者一个简单的问候。

然后，我们呈现一个响应，这是一个简单的HTML页面，它在装载的时候会重定向到chat.htm，并且传递给它必要的请求参数。

如果还没有客服有空，我们就给聊天者返回一个说明这个问题的标记，并且给他提供一个链接，用于检查是否有其他有空的客服。聊天者可以不停地点击这个链接，直到有空闲的客服出现，这个时候她会批准登录。（没错，这个做法并非最高效的，这也是为什么本章末尾的练习中有修改这部分的一条！）

然后我们会看到一个else分支，这里是我们处理客服登录的地方。我们也是先检查一下，看看支持的客服名字是否可用，然后跟处理用户一样进行两种情况的处理（名字可用以及名字不可用）。如果名字可用，那么就只有一个简单的更新，就是把这个聊天者插入chatters表中。这就是这个函数最后的部分！

3. 获取张贴的信息

服务器端代码的下一个函数是处理周期性提交的Ajax，返回获取自上次检查以后新张贴信息的函数。

Asp

```
if func = "getMessages" Then

  chatname = Trim(Request("chatname"))
  lastMessageTime = Trim(Request("lastMessageTime"))
   ' First, find out who this chatter is chatting with.
   rs.Open "select chatwith from chatters where " & _
     "chatname='" & chatname & "'", conn
   chatwith = rs("chatwith")
   rs.Close
   ' Now, get all messages posted by this chatter, or by who they were
   ' chatting with, since the time of the last message passed in.
   rs.Open "select messagetime, chatname, messagetext from messages " & _
     "where (chatname='" & chatname & "' or " & "chatname='" & chatwith & _
     "') and messagetime >= " & lastMessageTime, conn
```

```
firstMessage = true
%>
{ "lastMessageTime" : "<%=timeStamp%>",
  "messages" : [
<% Do While Not rs.EOF
  If firstMessage = true Then
    firstMessage = false
  Else
    response.write ", "
  End If
%>
     { "timestamp" : "<%=rs("messagetime")%>",
       "chatname" : "<%=rs("chatname")%>",
       "message" : "<%=rs("messagetext")%>"
     }
<%
     rs.MoveNext
   Loop
%>
  ] }
<%
rs.close()

End If ' End "getMessage" function handling.
```

Jsp

```
if (func.equalsIgnoreCase("getMessages")) {

  String chatname = request.getParameter("chatname");
  String lastMessageTime = request.getParameter("lastMessageTime");
  // First, find out who this chatter is chatting with.
  ResultSet rs = stmt.executeQuery(
    "select chatwith from chatters where " +
    "chatname='" + chatname + "'");
  rs.first();
  String chatwith = rs.getString(1);
  rs.close();
  // Now, get all messages posted by this chatter, or by who they were
  // chatting with, since the time of the last message passed in.
  rs = stmt.executeQuery(
    "select messagetime, chatname, messagetext from messages where " +
    "(chatname='" + chatname + "' or " + "chatname='" + chatwith +
    "') and messagetime >= " + lastMessageTime);
  boolean firstMessage = true;
  %>
  { "lastMessageTime" : "<%=timeStamp%>",
    "messages" : [
  <% while (rs.next()) {
    if (firstMessage) {
      firstMessage = false;
```

```
    } else {
      out.print(", ");
    }
%>
      { "timestamp" : "<%=rs.getString(1)%>",
        "chatname" : "<%=rs.getString(2)%>",
        "message" : "<%=rs.getString(3)%>"
      }
<% } %>
  ] }
<%
rs.close();

} // End "getMessage" function handling.
```

这个函数以获取两个请求参数开始：chatname和lastMessageTime。chatname参数是请求消息的聊天者的名字，lastMessageTime是上次这种请求发生的时间。

下一件事情是找出谁正在和这个聊天者聊天（请注意，这段代码并不关心提交请求的人是客户还是客服，因为这两种角色的逻辑都是一样的）。这么做是因为我们需要获取对话两边的聊天信息，但是这个参数只给了我们所需的一半信息。因此，我们一拿到两个聊天者的名字，就查询数据库，找出两个聊天者在lastMessageTime之后张贴的所有消息。这个动作的结果就是找出两个人自上次消息检查以来张贴的所有消息（如果有的话）的集合。

然后，假设我们至少找到一条记录，程序就开始遍历这个结果集。遍历的同时，它构造一个JSON字符串，它包含一个消息数据组成的数组，也就是消息张贴的时间（timestamp）、张贴的人（chatname）、以及消息文本（message）。这个JSON也包含了新的lastMessageTime数值，是你前面看到的timestamp的值。这个值将存储在客户端，用于下一次getMessages。最后，JSON写入到响应中，（技术上实际上是边生成JSON边发回去的，不过理解了就好。）然后就完了。前面已经在SupportChat.js里看过如何处理JSON数组了，现在你也知道它是如何构造的了！

4. 张贴消息

下一个要看的函数是张贴消息，它很紧凑。

Asp

```
if func = "postMessage" Then

  chatname = Trim(Request("chatname"))
  messagetext = Trim(Request("messagetext"))
  messagetext = Replace(messagetext, "'", "''")
  conn.Execute "insert into messages (messagetime, chatname, " & _
    "messagetext) values (" & _
    timeStamp & ", " & _
    "'" & chatname & "', " & _
    "'" & messagetext & "')"

End If ' End "postMessage" function handling.
```

Jsp

```
if (func.equalsIgnoreCase("postMessage")) {

  String chatname = request.getParameter("chatname");
  String messagetext = request.getParameter("messagetext");
  messagetext = messagetext.replace('\'', '`');
  stmt.executeUpdate("insert into messages (messagetime, chatname, " +
    "messagetext) values (" +
    timeStamp + ", " +
    "'" + chatname + "', " +
    "'" + messagetext + "')");

} // End "postMessage" function handling.
```

张贴信息没什么太多要做的事情。我们先获取到来的请求参数chatname，它表示张贴者，以及messagetext，它是消息本身。然后快速扫描一下这个字符串，用双引号替换所有出现的单引号。这么做是为了避免你下面要看到的SQL构造语句（用于向message表插入消息）。这些就是这个函数要做的全部事情了。

5. 登出

只剩下一个函数了，它就是处理登出的函数。这里要做的处理可能比想象得要多一些，但也多不了多少！

Asp

```
if func = "exitChat" Then

chatname = Trim(Request("chatname"))
' First, find out who this chatter is chatting with.
rs.Open "select chatwith from chatters where " & _
  "chatname='" & chatname & "'", conn
chatwith = rs("chatwith")
rs.Close
' Now, delete all messages the chatter posted, as well as messages
' posted by who they were chatting with.  After this query, the
' "conversation" is effectively deleted from the database.
conn.Execute "delete from messages where chatname='" & chatname & _
  "' or chatname='" & chatwith & "'"
' Next, delete the chatter from the chatters table.
conn.Execute "delete from chatters where chatname='" & chatname & "'"
' Finally, if we find any records in the chatters table where this
' chatter is the value of the chatwith field, update that field of
' that record to "none".  This covers when the chatter logging off is a
' customer, it makes the support personnel available again.  If it's
' a support personnel logging off, it does no harm to the chatter,
' although the chatter is effectively "orphaned", i.e., their messages
' will not be seen by a support personnel, and they will see messages
' from no support personnel.
conn.Execute "update chatters set chatwith='none' where " & _
  "chatwith='" & chatname & "'"
```

```
    ' Finally, say goodbye to the chatter.
    %>
    <html>
      <head>
        <title>Exiting chat</title>
        <script>
          function exitChat() {
            window.location = "../goodbye.htm";
          }
          </script>
        </head>
        <body onLoad="exitChat();">Exiting chat...</body>
      </html>
      <%

  End If ' End "exitChat" function handling.


End If ' End function handling section.
```

Jsp

```
if (func.equalsIgnoreCase("exitChat")) {

  String chatname = request.getParameter("chatname");
  // First, find out who this chatter is chatting with.
  ResultSet rs = stmt.executeQuery(
    "select chatwith from chatters where " +
    "chatname='" + chatname + "'");
  rs.first();
  String chatwith = rs.getString(1);
  // Now, delete all messages the chatter posted, as well as messages
  // posted by who they were chatting with.  After this query, the
  // "conversation" is effectively deleted from the database.
  stmt.executeUpdate("delete from messages where chatname='" + chatname +
    "' or chatname='" + chatwith + "'");
  // Next, delete the chatter from the chatters table.
  stmt.executeUpdate("delete from chatters where chatname='" + chatname +
    "'");
  // Finally, if we find any records in the chatters table where this
  // chatter is the value of the chatwith field, update that field of
  // that record to "none".  This covers when the chatter logging off is a
  // customer, it makes the support personnel available again.  If it's
  // a support personnel logging off, it does no harm to the chatter,
  // although the user is effectively "orphaned", i.e., their messages
  // will be seen no support personnel, and they will see messages from no
  // support personnel.
  stmt.executeUpdate("update chatters set chatwith='none' where " +
    "chatwith='" + chatname + "'");
  // Finally, say goodbye to the chatter.
  %>
```

```
<html>
  <head>
    <title>Exiting chat</title>
    <script>
      function exitChat() {
          window.location = "../goodbye.htm";
        }
      </script>
    </head>
    <body onLoad="exitChat();">Exiting chat...</body>
  </html>
  <%

} // End "exitChat" function handling.


} // End function handling section.
```

首先，通过到来的chatname请求参数判断是哪个聊天者试图登出。然后查询chatters表，看看他们在和谁聊天。然后，在messages表里面删除全部该聊天者张贴的消息，或者是他的聊天对象张贴的消息。这只是一点点清理工作，这样可以避免聊天记录保存起来，占用太多不必要的空间。然后从chatters表里删除这个聊天者。然后，更新chatters表，把所有chatwith字段等于我们这里拿到的chatname的记录都更新掉。这样就可以让刚才和这个聊天者聊天的客服解放出来了。如果是客服登出，那么这个客户就会变成孤儿了，这是我们现在必须面对的缺点（可以在getMessage里添加一些代码，这样发生了这种情况之后，你可以告诉用户已经没有谁在和他聊天了，以及也许还可以把他重定向到另外的页面上）。

最后一步是返回一个页面，这个页面会立即把聊天者重定向到你前面看到的goodbye.htm。然后从这个点开始，这个用户就正式登出了。

6. 清理

两个版本的服务器端文件最后做的事情都是清理数据库连接。

Asp

```
Set conn = Nothing
```

Jsp

```
if (stmt != null) {
  stmt.close();
}
if (conn != null) {
  conn.close();
}
```

不管哪个版本，RecordSet对象都应该已经被关闭了，这就是为什么你在两个版本里都没有看到提及此事的原因。

这样，我们就完成了这个应用的代码讲解！我希望这是一个演示Ajax、JSON和Mootools的好例子，还希望它显示了在Ajax世界里，客户端和服务器是如何交互的。说实话，这个应用是本书中我有意地留下不少需要改进的地方（你可能会说是缺点）的项目，我准备提出一些建议，这样可以增强你的“小

脑”对新知识的印象。让我们看看这些建议。

12.8　练习

这个应用可以稍微修改一下，做得更好，下面是一些建议的练习，练习它们可以让你对Ajax和Mootools熟悉起来。

- 写个PHP版本如何？只需要把Java代码转换成PHP的代码就行了，如果你熟悉PHP，这个活儿应该不难。
- 如果能允许聊天者先登录，然后再等待有空的客服和她聊天会更好些。
- 在任何你认为合理的地方添加工具提示。Mootools有一个提示插件，可以做这个事情。
- 用一些Mootools的特效。比如，可能可以在聊天者登出的时候折叠页面，或者，如果你实现了先登录再等人的特性，则在客服加入聊天的时候来点淡出效果。
- 像我们前面暗示过的那样，在聊天者张贴一条消息的时候，立即在他自己的屏幕上显示这条消息，这样他就不用等待下一次getMessage()运行了。
- 给显示的每条消息前头都加上张贴的时间戳应该是一个不错的小改进。

这是一个很短的列表。我相信你能想出来更多觉得需要改进的东西，但这些应该是个好的开端。

12.9　小结

在我们旅程的最后一步（呃，最后一章）里，我给你们介绍了可能是最近两年来Web开发最著名的一个术语：Ajax。你看到它的使用，见证了它是如何让一些应用成为可能，或者至少是更好些。你还看到了Ajax等式中的服务器组件是如何运转的。另外，你还了解了优秀的Mootools JavaScript库是如何在你的Ajax项目以及其他方面帮助你的。在这个过程中，你创建了一个小小的聊天项目用于在线客户，如果你愿意，这个项目甚至可以投入实用，支持你的客户。

“本书并不像一本教材，畅读之后，你会觉得它就是一位讲课生动的老师，带领你遨游JavaScript海洋，把你推向技术风浪的前沿，同时又给了你一个功能完备的冲浪板，接下来就是你在风口浪尖上享受JavaScript高潮的时候了。”

——本书译者

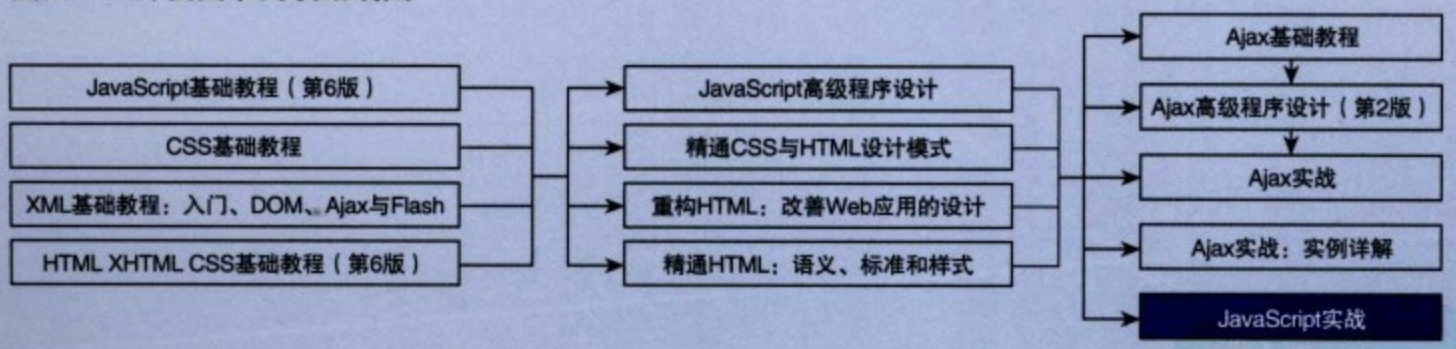
Practical JavaScript, DOM Scripting, and Ajax Projects
JavaScript实战

随着Ajax的兴起，JavaScript迅速地从改进网站的配角晋升为开发专业级高质量应用的主角，成为了Web开发中不可缺少的一员。

本书主要通过10个具体项目，包括构建可扩展的JavaScript库、使用GUI窗口小部件框架、开发支持拖放的购物车和编写JavaScript游戏等，讲述JavaScript最佳实践、Ajax技术，以及一些流行的JavaScript库，如Rico、Dojo、script.aculo.us、YUI等。读者在理解的基础上可以方便地将所学知识应用到自己的项目中。书中项目也非常实用，读者可以直接参考利用。

Frank W. Zammetti 世界知名的Web开发专家，Omnytex公司的创始人和首席软件架构师。他是多个开源项目的领导者，包括扩展Struts的AjaxTag库、StrutsWS和Java Web Parts等。除本书外，他还撰写了Dojo和JavaScript等方面的多部畅销书。

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)51095186
反馈/投稿/推荐信箱：contact@turingbook.com

分类建议 计算机/网络技术/程序设计

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-18915-8

ISBN 978-7-115-18915-8/TP

定价：59.00 元