

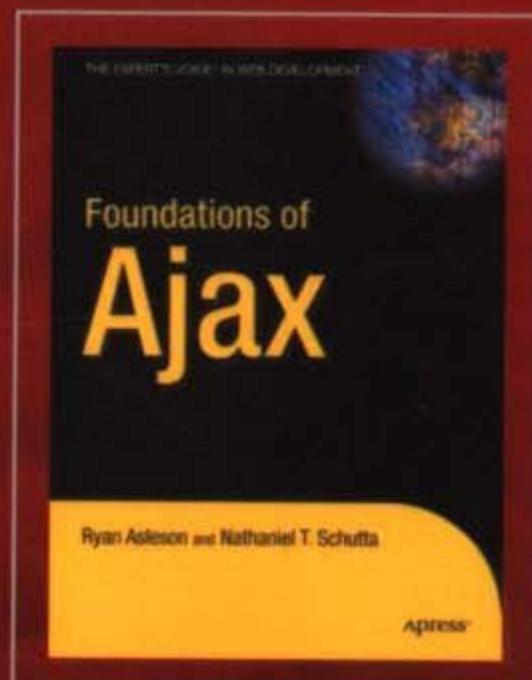
Foundations of **Ajax**

亚马逊计算机  
榜首图书!

# Ajax 基础教程

[美] Ryan Asleson  
Nathaniel T. Schutta 著  
金灵 等译

- 涵盖 Web 2.0 核心技术
- 创建桌面式 Web 应用
- 快速上手，立即给网站带来全新用户体验



人民邮电出版社  
POSTS & TELECOM PRESS

Foundations of **Ajax**

# Ajax 基础教程

这是一本学习 Ajax 的好书，可以为你开发应用打下坚实基础。

—Kishore, ajaxguru.blogspot.com

这本书最打动我的一点是……书中有那么多详细、深入的代码示例和讨论。不过，它并不是简单地罗列代码，而是清楚地说明了该怎么做，以及为什么这样做，这真是太棒了！

—Greg Hughes, 个人网络日志

本书循序渐进，实例极为清晰简洁，后面几章讲述了 JavaScript 与 Ajax 的调试、测试和文档编写，即使有经验的 Ajax 程序员也会获益匪浅。……总之，每一位 Web 开发人员都应该拥有本书。

—Ernest Friedman-Hill, Jess (开源 Java 规则引擎) 的开发者，

美国 Sandia 国家实验室

2005 年，在 Web 2.0 热潮中，Ajax 横空出世，迅速成为最炙手可热的 Web 开发技术。Google、Microsoft、Amazon 和 Yahoo 都已经全面采用 Ajax，新一代的网站如雨后春笋，迅速兴起。

什么是 Ajax？Ajax 为什么会这么热，它到底有什么奇妙之处？Ajax 是少数高手才能使用的尖端技术吗？如何用 Ajax 开发全新用户体验的 Web 应用，如何用 Ajax 赋予原有应用新的生命？本书将给你满意的答案！

书中不仅详细讲述了如何结合使用各种标准 Web 技术如 JavaScript、HTML、CSS 和 XMLHttpRequest 开发 Ajax 应用，而且涵盖了调试、测试、文档、验证等工具，以及相关的模式、框架、应该避免的陷阱。阅读本书，再加上已有的开发经验，你也能在应用中使用超炫的 Ajax 技术，使你的网站立即焕然一新！这一过程将乐趣无穷，我们衷心地希望，有一天能看到你开发的基于 Ajax 的一流应用！



**Ryan Asleson** 资深 Web 开发人员，对 JavaScript 和 Web 开发工具有着丰富的经验。他是基于 J2EE 的 Ajax 开源框架 Taconite (taconite.sf.net) 的创始开发者之一。他的兴趣还包括性能优化和基于标准的开发。



**Nathaniel T. Schutta** 资深 Java Web 开发人员。他拥有明尼苏达大学软件工程硕士学位，并通过了 SCWCD (Sun 认证 Web 构件开发人员) 认证。他特别关注用户界面设计，是 ACM 人机交互特殊兴趣小组的资深成员。

本书相关信息请访问：图灵网站 <http://www.turingbook.com>

读者/作者热线：(010)88593802

反馈/投稿/推荐信箱：[contact@turingbook.com](mailto:contact@turingbook.com)

上架建议

计算机 / 网络开发 / 程序设计

ISBN 7-115-14481-8



9 787115 144812 >

Apress®

ISBN7-115-14481-8/TP·5211

定价：35.00 元

TURING 图灵程序员设计丛书

# Ajax基础教程

## Foundations of Ajax

[美] Ryan Asleson  
Nathaniel T. Schutta 著

金灵 等译



## 图书在版编目 (CIP) 数据

Ajax 基础教程 / (美) 阿斯利森, (美) 舒塔著; 金灵等译.

—北京: 人民邮电出版社, 2006.2

(图灵程序设计丛书)

ISBN 7-115-14481-8

I. A... II. ①阿...②舒...③金... III. 计算机网络—程序设计 IV. TP393.092

中国版本图书馆 CIP 数据核字 (2006) 第 003735 号

## 内 容 提 要

Ajax 技术可以提供高度交互的 Web 应用, 给予用户更丰富的页面浏览体验。本书重点介绍 Ajax 及相关的工具和技术, 主要内容包括 XMLHttpRequest 对象及其属性和方法、发送请求和处理响应、构建完备的 Ajax 开发工具、使用 JsUnit 测试 JavaScript、分析 JavaScript 调试工具和技术, 以及 Ajax 开发模式和框架等。本书中所有例子的代码都可以从 Apress 网站本书主页的源代码 (Source Code) 免费得到。

本书适合各层次 Web 应用开发人员和网页设计人员阅读。

## 图灵程序设计丛书

### Ajax 基础教程

- 
- ◆ 著 [美] Ryan Asleson Nathaniel T.Schutta
  - ◆ 译 金 灵 等
  - 责任编辑 傅志红
  - ◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街 14 号  
    邮编 100061 电子函件 315@ptpress.com.cn  
    网址 <http://www.ptpress.com.cn>  
    北京顺义振华印刷厂印刷  
    新华书店总店北京发行所经销
  - ◆ 开本: 800×1000 1/16  
    印张: 17  
    字数: 356 千字 2006 年 2 月第 1 版  
    印数: 1~5 000 册 2006 年 2 月北京第 1 次印刷

著作权合同登记号 图字: 01-2006-0319 号

ISBN 7-115-14481-8/TP · 5211

定价: 35.00 元

读者服务热线: (010) 88593802 印装质量热线: (010) 67129223

# 译 者 序

见过 Google Maps 的界面吗？也许你还不以为然，如果是桌面应用，这只是雕虫小技而已。不过要知道，你看到的可不是桌面应用，它完全是一个 Web 应用！

也许你又会说，是 Web 应用又怎么样？我一样可以做到。不错，不是有这么一句话吗？“只要功夫深，铁杵也能磨成针。”只要你肯想办法，没有做不到的。但是，可以告诉你，这个应用并没有让开发人员费多大力气。

是什么“高级技术”让开发人员如虎添翼？又是什么“宝贝”使用户欣喜万分？

这就是 Ajax！

Ajax 并不是一个高深的技术，如果说它是宝贝，也只能算是“老宝贝”！曾经有人评价 Ajax 是“新瓶装老酒”。但是这并不妨碍人们对它趋之若鹜。而且 Ajax 这瓶老酒经过时间的积淀和环境的变化，已经更加醇香，再加上美观实用的新包装，它的人气这么旺，可以说是顺理成章。

实际上我们从界面上看到的只是一部分，Ajax 带给我们的远不只是这些。到 Google Maps、Google Suggest 和 Gmail 上亲身感受一下，你会有更深体会。

你是不是有这样一些问题：

- Ajax 是什么？新名词？杂烩汤？还是……
- 用 Ajax 到底能做什么？制作漂亮界面？改善用户体验？还有……
- 怎么实现 Ajax？XMLHttpRequest 对象是什么？怎么避免页面完全刷新？……
- Ajax 最适用哪些情况？完成验证？建立自动刷新页面？还有……
- Ajax 有没有帮手？可以用哪些工具和技术简化 Ajax 开发？
- 能在 Ajax 开发应用测试驱动开发吗？怎么做呢？
- JavaScript 也能轻松地调试吗？有哪些调试工具？这些工具好用吗？

- 有哪些可用的 Ajax 框架？Ajax 框架对 Ajax 开发有多大作用？

本书能够帮助你回答这些问题。相信通过阅读本书，你能轻松地迈进 Ajax 殿堂，让你的老板、用户对你刮目相看！

我们深深地感谢我们的家人和朋友。在翻译过程中，他们给予了我们莫大的关心、支持和帮助。

全书由金灵主译，苏金国、徐阳、刘鑫、蔡洪亮等检查术语，刘晓兵、荆涛、张野、任岗、伊瑞海等提供技术问题支持，在大家的努力下共同完成了本书的翻译工作。

由于时间仓促，且译者的水平有限，译文中难免会出现一些错误，请读者批评指正。

译者

2005.12

# 前　　言

几年前开始构建 Web 应用时，我们感觉这简直就是软件开发的“圣杯”。以前，我们一直开发的都是胖客户应用，公司每次发布这种公司应用的新版本时，总是需要将这种应用部署到分散在全国各地的数百个用户那里去，让我们沮丧的是，这种复杂的安装过程不仅冗长而且很容易出错，不仅让开发人员很头疼，用户也非常不满。

通过浏览器来部署应用，这看上去相当不错，因为这样，就不再需要在客户端上安装软件了。所以，与许多其他公司一样，我们公司也很快转型，开始在 Web 上部署应用。

尽管部署起来相对容易，但 Web 应用也有自己的问题。在用户看来，最突出的问题是用户界面没有了以往丰富的交互性。Web 应用仅限于使用 HTML 提供的一组基本部件，这是很有限的。更糟糕的是，与服务器交互需要完全刷新页面，很多用户已经熟悉了功能强大的客户-服务器应用，对他们来说，这一点很让人不快。

我们曾经一直认为，在 Web 应用中只要刷新页面就必须完全刷新，好像这是在所难免的，所以往往想方设法地避免页面刷新。我们甚至还考虑过编写一个 Java applet，由它处理浏览器和服务器之间的通信。不过，随着越来越多 Web 应用的部署，我们很快发现，用户已经习惯了这种完全页面刷新的方式，这么一来，我们也不再那么强烈地想要另辟蹊径了。

转眼 5 年过去了。由于 Google Suggest 和 Gmail 等应用的出现，甚至在 Ajax 这个术语出现之前，这种使用 XMLHttpRequest 对象在浏览器和服务器之间完成异步通信的方法就已经在开发者社区中产生了很大反响。多年之前，IE 中就已经使用了 XMLHttpRequest 对象，但是如今它得到了更多其他浏览器的支持，取得了重大突破。我们在一个正在开发的应用中增加了 Ajax 功能，结果令我们震撼不已，所以我们都产生了一种想法：“嘿，应该有人来写一本有关 Ajax 的书呀。”本书因此应运而生。

## 本书概述

本书旨在为开发人员介绍为已有的或者将来的应用增加 Ajax 技术所需的所有工具。

在写作中我们牢记：“你需要知道的我们都要介绍，你不需要知道的我们绝口不提。”我们认为，作为本书的读者，你应该已经是一个有经验的 Web 应用开发人员。正因如此，我们会把重点放在你很可能不了解的新内容上：Ajax 及相关的工具和技术。我们不会花大量篇幅来讲述服务器端语言，因为我们认为你会自己选择工具集来开发服务器端功能，在这方面你不需要我们的帮助。另外我们也不会浪费时间来讨论如何构建企业级应用，这样的应用很少使用 Ajax。相反，重申一次，我们的重点只是 Ajax 以及相关的工具和技术。

本书中的示例特意做得很小，而且很紧凑。它们会尽可能简洁地展示一个或两个重要的 Ajax 概念。我们认为，作为一个有经验的 Web 开发人员，你应该能熟练地把我们展示的内容推广到自己的环境中去，因此，我们不会在示例中“堆放”对你没有多大用处的信息。

第 1 章讨论了 Web 应用开发的发展历程，从过去谈到现在，并且预测了将来。如果你了解了 Web 开发技术的过去，就能更容易地认识到它们将来会有怎样的发展。

第 2 章介绍了 XMLHttpRequest 对象。这个 Ajax 概念你可能不太熟悉，所以我们专门用一章来解释 XMLHttpRequest 对象的属性和方法。也许你像我们一样，直到最近才注意到 XMLHttpRequest 对象。不过，要知道早在几年前 IE 中就已经有了 XMLHttpRequest 对象。因此，我们会用充分的笔墨讨论 XMLHttpRequest 对象，并说明它能做什么。

第 3 章开始讲述 Ajax 的具体内容。这一章将讨论 XMLHttpRequest 对象用来与服务器通信的各种方法。我们讨论了可以使用 XML、纯文本甚至 JavaScript Object Notation (JSON) 作为传输介质，并介绍了它们与 XMLHttpRequest 对象结合使用的各种方式。在这一章的最后，你就能熟练地使用 XMLHttpRequest 对象与服务器端通信，而不必让用户苦苦等待页面完全刷新了。

作为开发人员，我们总是花很多时间学习新技术，但并不知道怎么实际应用。第 4 章将针对这个问题展示一系列可以使用 Ajax 技术的场合。如前所述，每个示例都很小、很紧凑，这样你能更好地理解相应内容，而不用在大堆不必要的信息中搜寻。

第 5、6 和 7 章对于 Ajax 新手来说尤其重要。我们不希望你仓促上阵，应该先配备好合适的工具和技术，再考虑在应用中增加 Ajax。利用第 5 章介绍的工具和技术，可以简化 Web 应用的开发，你可以得到高质量、遵循行业标准而且将来更易于维护的代码。

测试驱动开发 (Test-driven development, TDD) 使我们开发应用的方法发生了根本性的改变。在编写代码之前先编写单元测试，这样就能确保所写的代码会正常工作，并能大大提高代码的质量。利用一个单元测试集，可以确保做出修改后所有代码还能按预期的那样工作，这样将来进行修改将更容易。Ajax 当然也应该使用 TDD，而且 TDD 的好处再强调也不为过，所以我们专门用一章来讨论。由于 Ajax 主要是一个基于浏览器的技术，第 6 章将展示如何对 JavaScript 代码应用 TDD。

谈到 JavaScript，如果你想使用 Ajax，就必须至少用 JavaScript 写过一些程序。许多开发人员都不喜欢 JavaScript，认为它缺少一些重要的生产性工具，如调试器，所以没有多大的用处。如今这种说法已经不成立了。第 7 章讨论了一些调试工具和技术，如果出现问题，你可以用这些工具和技术来跟踪，使得问题可以尽快、尽可能容易地加以解决。你不用再担心在使用 JavaScript 时出现问题而不能诊断，也不用因此对 JavaScript 退避三舍。

Ajax 是一个发展迅猛的技术，在写本书期间，Ajax 已经得到了飞速的发展。第 8 章把所有的内容综合在一起，讨论了新兴的 Ajax 开发模式和框架，并提供了一些在线资源。另外，第 8 章还给出了一个完整的示例，不仅显示了一些高级的 Ajax 技术，还展示了使用现成的 Ajax 框架来完成 Ajax 开发是何等容易。通过使用框架，你不用再做 Ajax 开发的一些繁琐任务，这样就能把重点集中在业务逻辑上，而不是 Ajax 的细节上。

最后，附录 A 描述了不同浏览器上 W3C DOM 和 JavaScript 实现中存在的一些特异之处和不一致的地方，并提供了一些方法来解决这些问题。附录 B 总结了最流行的 Ajax 框架和库，这些框架和库有助于简化 Ajax 技术的使用。随着 Ajax 越来越流行，框架也肯定会越来越多，所以让我们对新的框架和其他开发工具拭目以待。

## 本书源代码

本书中的所有示例都可以从 Apress 网站的源代码（Source Code）部分免费得到。在浏览器地址栏中输入 [www.apress.com](http://www.apress.com)，点击 Source Code 链接，在列表中找到 *Foundations of Ajax*，也可以从 Apress 网站的本书主页下载源代码（zip 文件）。源代码是按章组织的。

## 本书的更新信息

尽管我们尽了最大努力，但是你可能还会在书中偶尔发现一些错误，当然我们希望不会这样！如果正文或者源代码中还有错误，对此我们表示歉意。可以从 Apress 网站的本书主页上得到最新的勘误表，在那里还会提供我们的联系信息，如果你发现了错误，可以通知我们。

## 联系我们

我们非常重视你对本书内容和源代码示例的疑问和建议。请把所有问题和建议直接发到 [foundationsofajax@gmail.com](mailto:foundationsofajax@gmail.com)，我们会尽快回复。不过要记住，我们（像你一样！）不可能马上就有所回应。

谢谢你购买了这本书！我们希望你认为这是一本有价值的书，能像我们写书一样兴趣盎然地阅读这本书。

Ryan Asleson 和 Nathaniel T. Schutta

# 致 谢

**感**谢 Apress 提供了编写本书的机会。感谢 Grace Wong 在整个写作过程中一直督促着我们集中精力、保证进度。Keith Harvey 和 Brent Ashley 提供了很有价值的反馈意见，本书能写成这样，离不开他们的帮助。Kim Wimpsett 很是倒霉，他要对付我们成堆的拼写和语法错误，对此我们致以衷心的谢意。我们要感谢 Linda Marousek 指导我们完成了最后的出版过程，看到用电脑打印的手稿变成一本装帧精美的图书，实在让我们欣喜万分。Ewan Buckingham 回答了我们很多问题，让我们能轻松地进入写作过程。还要感谢在整个过程中我们的代理人 Laura Lewin 和 Studio B 的全体人员所提供的支持。

—Ryan Asleson 和 Nathaniel T. Schutta

**我**要感谢整个 Apress 团队，能让我这样一个没有写作经验的新手写这本书。没有他们的信任和支持，我是无法完成这本书的。我要感谢 Gary Cornell，在我最早提议写这本书后，他给我回了一封热情洋溢、充满信任和鼓励的电子邮件。我要特别感谢 Chris Mills，是他帮助我反复修改提纲，终于有了现在这个最终稿。

对于 Nate Schutta，我的朋友、同事和合作者，说再多好话也不为过。我实在钦佩他旺盛的精力、卓越的见识和为这本书付出的辛勤努力。没有他的帮助，这本书就不可能出版。

最重要的，我要感谢妻子 Sara，在我忙于写书的几个月里，她独自承担了很多责任，但却毫无怨言。

—Ryan Asleson

**感**谢我的合作者 Ryan，谢谢你让我帮忙，把你的想法变成一本书，这真是我的荣幸。那一天在会议中间休息的时候，我只是向你扔过去一个球，谁想到会带来这样一个结果呢？说到这里，我还要感谢 Sara 和 Adam！当然，要是没有感谢我可爱的妻子 Christine，

实在是说不过去。在写书的期间，你如此宽容，如果没有你的爱和耐心，我什么也做不到。非常感谢 Nathan Good，感谢你的帮助和建议，否则我不会有今天的成绩（我会怀念与你共饮咖啡的那些时光）。

没有我父母的远见，我可能不会涉足这个领域。多年前的一个夏天，他们给我报名参加了一个暑期计算机学习班，而且给我用了当时最新潮的机器。谢谢你们，对你们的感谢无以言表！另外，还要特别感谢圣约翰大学计算机科学系的 Jim Schnepf、Lynn Ziegler、John Miller、Andy Holey 和 Noreen Herzfeld，你们真是好心，能接受我这样一个化学专业的学生攻读计算机专业的硕士，我才有了这样一个非比寻常的学习历程！我还要感谢很多对我有影响的老师：Karen Sweet、Mary Ellen Briel、Michael Youngberg、Peggy Anderson、Jim Murphy（已故），当然还有 Elizabeth Stoltz。谢谢你们大家，你们让我的生活更加丰富，对此我无以为报。没有你们的指导和监督，我可能不是今天的我。我知道，由于篇幅所限，我肯定还遗漏了很多人，对此我致以深深的歉意。再次感谢这里提到的每一个人，还有那些永远铭刻在我心里的人们。

——Nathaniel T. Schutta

# 关于技术审核

© KEITH HARVEY 是 SCOPE iT 公司 ([www.scopeit.com](http://www.scopeit.com)) 的 CTO 和首席架构师。该公司主要致力于开发基于 Web 的项目预算应用程序，用于 IT 项目的快速估算、预算和规划。SCOPE iT 是 Microsoft 的合作伙伴，拥有 ISV/软件解决方案资格。SCOPE iT 应用程序是在最新的 Microsoft .NET 技术、SQL Server 和 Ajax 的基础上开发的。

Keith 也是一位作者，写过许多关于 Microsoft 技术、数据库、Ajax、软件项目估算等方面的文章。Keith 与妻子 Tricia 和女儿 Hanna 居住在北加州。他的个人网站是 [www.keith-harvey.com](http://www.keith-harvey.com)。

# 目 录

**译者序**

**前言**

**致谢**

**关于技术审校**

<b>第 1 章 Ajax 简介</b>	1
1.1 Web 应用简史	1
1.2 浏览器历史	2
1.3 Web 应用的发展历程	3
1.3.1 CGI	4
1.3.2 applet	4
1.3.3 JavaScript	5
1.3.4 servlet、ASP 和 PHP……哦，太多了！	6
1.3.5 Flash	9
1.3.6 DHTML 革命	10
1.3.7 XML 衍生语言	10
1.3.8 基本问题	12
1.3.9 Ajax	13
1.4 可用性问题	17
1.5 相关技术	18
1.6 使用场合	19
1.7 设计考虑	20
1.8 小结	21

<b>第 2 章 使用 XMLHttpRequest 对象</b>	23
2.1 XMLHttpRequest 对象概述	23
2.2 方法和属性	25
2.3 交互示例	26
2.4 GET 与 POST	28
2.5 远程脚本	29
2.5.1 远程脚本概述	29
2.5.2 远程脚本的示例	29
2.6 如何发送简单请求	31
2.6.1 简单请求的示例	32
2.6.2 关于安全	34
2.7 DOM Level 3 加载和保存规约	35
2.8 DOM	35
2.9 小结	36
<b>第 3 章 与服务器通信：发送请求和处理响应</b>	37
3.1 处理服务器响应	37
3.1.1 使用 innerHTML 属性创建动态内容	37
3.1.2 将响应解析为 XML	40
3.1.3 使用 W3C DOM 动态编辑页面	45
3.2 发送请求参数	52
3.2.1 请求参数作为 XML 发送	59
3.2.2 使用 JSON 向服务器发送数据	64
3.3 小结	70
<b>第 4 章 实现基本 Ajax 技术</b>	71
4.1 完成验证	71
4.2 读取响应首部	75
4.3 动态加载列表框	79
4.4 创建自动刷新页面	85
4.5 显示进度条	90
4.6 创建工具提示	95
4.7 动态更新 Web 页面	101
4.8 访问 Web 服务	110

---

4.9 提供自动完成 .....	116
4.10 小结 .....	123
<b>第 5 章 构建完备的 Ajax 开发工具箱 .....</b>	<b>125</b>
5.1 使用 JSDoc 建立 JavaScript 代码的文档 .....	125
5.1.1 安装 .....	126
5.1.2 用法 .....	126
5.2 使用 Firefox 扩展验证 HTML 内容 .....	129
5.2.1 HTML Validator .....	130
5.2.2 Checky .....	132
5.3 使用 DOM Inspector 搜索节点 .....	133
5.4 使用 JSLint 完成 JavaScript 语法检查 .....	137
5.5 完成 JavaScript 压缩和模糊处理 .....	138
5.6 使用 Firefox 的 Web 开发扩展 .....	140
5.7 实现高级 JavaScript 技术 .....	141
5.7.1 通过 prototype 属性建立面向对象的 JavaScript .....	142
5.7.2 私有属性和使用 JavaScript 的信息隐藏 .....	146
5.7.3 JavaScript 中基于类的继承 .....	148
5.7.4 汇合 .....	149
5.8 小结 .....	152
<b>第 6 章 使用 JsUnit 测试 JavaScript 代码 .....</b>	<b>155</b>
6.1 JavaScript 提出的问题 .....	155
6.1.1 测试先行方法介绍 .....	155
6.1.2 JUnit 介绍 .....	157
6.2 分析 JsUnit .....	158
6.2.1 起步 .....	159
6.2.2 编写测试 .....	159
6.2.3 运行测试 .....	172
6.2.4 使用标准/定制查询串 .....	177
6.2.5 使用 JsUnit 服务器 .....	181
6.2.6 获得帮助 .....	183
6.2.7 还能用什么? .....	184
6.3 小结 .....	184

<b>第 7 章 分析 JavaScript 调试工具和技术</b>	185
7.1 用 Greasemonkey 调试 Ajax 请求	186
7.1.1 Greasemonkey 介绍	186
7.1.2 使用 Greasemonkey XMLHttpRequest 调试用户脚本	186
7.1.3 使用 XMLHttpRequest 调试用户脚本检查 Ajax 请求和响应	186
7.2 调试 JavaScript	188
7.2.1 使用 Firefox JavaScript Console	189
7.2.2 使用 Microsoft Script Debugger	190
7.2.3 使用 Venkman	192
7.3 小结	207
<b>第 8 章 万事俱备</b>	209
8.1 模式介绍	209
8.1.1 实现褪色技术	209
8.1.2 实现自动刷新	210
8.1.3 实现部分页面绘制	210
8.1.4 实现可拖放 DOM	211
8.2 避免常见的陷阱	212
8.3 相关的更多资源	214
8.4 使用框架	216
8.5 Taconite 介绍	216
8.5.1 Taconite 原理	217
8.5.2 解决方案	217
8.5.3 Taconite 怎么处理内容	219
8.6 Dashboard 应用介绍	219
8.7 用 Taconite 构建 Ajax Dashboard	221
8.7.1 一般特性介绍	221
8.7.2 设计特性介绍	222
8.7.3 分析代码	224
8.7.4 分析天气预报组件	225
8.7.5 分析标题新闻组件	232
8.7.6 如何完成自动重新刷新工作	235
8.7.7 构建更好的 autocomplete	237
8.8 小结	240

---

<b>附录 A 开发跨浏览器 JavaScript</b>	241
A.1 向表中追加行	241
A.2 通过 JavaScript 设置元素的样式	242
A.3 设置元素的 class 属性	243
A.4 创建输入元素	243
A.5 向输入元素增加事件处理程序	244
A.6 创建单选钮	245
A.7 小结	246
<b>附录 B Ajax 框架介绍</b>	247
B.1 浏览器端框架	247
B.1.1 Dojo	248
B.1.2 Rico	248
B.1.3 qooxdoo	248
B.1.4 TIBET	249
B.1.5 Flash/JavaScript 集成包	249
B.1.6 Google AJAXSLT	249
B.1.7 libXmlRequest	249
B.1.8 RSLite	250
B.1.9 SACK	250
B.1.10 sarrisa	250
B.1.11 XHConn	251
B.2 服务器端框架	251
B.2.1 CPAINT	251
B.2.2 Sajax	251
B.2.3 JSON/JSON-RPC	251
B.2.4 Direct Web Remoting	252
B.2.5 SWATO	252
B.2.6 Java BluePrints	252
B.2.7 Ajax.Net	252
B.2.8 Microsoft 的 Atlas 项目	253
B.2.9 Ruby on Rails	253

## Ajax 简介

今天的因特网已经历了翻天覆地的重大改变。最早它只有基于文本的简单浏览器，供科学家之间交流研究心得；如今，它已经成为商务和信息的中心。这期间，许多新方法和新技术相继粉墨登场，从早期的图形化浏览器到最近的自由播（podcast，也称播客、随身播）等等。到了今天，因特网已经成为大量应用的首选平台。（还记得你最后一次和旅行社直接打交道是什么时候吗？是不是早已经开始通过网络完成这些交易了？）不过，尽管因特网提供了很大的便利，但很少有人会把 Web 应用与桌面应用相混淆。本章将先对 Web 应用的发展历程做简要的回顾，最后向您介绍明日之星：Ajax。

### 1.1 Web 应用简史

混沌初开，一切都那么简单。为了连接美国的少数几个顶尖研究机构，人们设计了最早的“Internet”，以便共同开展科学的研究。不论是图书馆员、核物理学家，还是计算机科学家，都必须学习一个相当复杂的系统。1962 年，麻省理工学院（MIT）的 J.C.R. Licklider 最早提出他的“Galactic Network”（超大网络）思想时，Firefox 和 IE 之类的便捷工具连概念都未产生。

Licklider 后来继续在美国国防高级研究计划局（DARPA）从事计算机研究，在那里他积极地宣扬网络化思想的重要性。几乎与此同时，MIT 的 Leonard Kleinrock 和 Lawrence G. Roberts 正在开展分组交换理论的研究，这是计算机联网的一个核心概念。在 Thomas Merrill 的帮助之下，Roberts 于 1965 年进而创建了第一个广域网，他通过一个拨号连接使马萨诸塞州的一台 TX-2 连上了加利福尼亚州的一台 Q-32。

1966 年底，Roberts 带着他的实验结果来到 DARPA，在这里他构思了高级研究项目管理网络（Advanced Research Projects Administration Network，ARPANET）的计划。此时，Kleinrock 正在加州大学洛杉矶分校网络测量中心（Network Measurement Center），这里被选作 ARPANET 的第一个节点。正是在这里，1969 年 BBN 公司成功地安装了第一个分组交换

器，称为接口消息处理器（Interface Message Processors, IMP）。斯坦福研究中心被选为第二个节点，1969年10月，在此首次实现了主机到主机的消息交换。此后不久，又将加州大学圣巴巴拉分校和犹他大学增加为节点，这就是我们现在所称因特网的前身。

这个时期小型机刚开始出现，DEC公司推出了PDP-1，其后又相继推出了PDP-8、PDP-11和VAX-11/780，并取得了巨大成功。计算机能力得到了极大增强，而且使用也越来越方便，不像最初只有极少的几台大型机时，人们须排队使用。计算机已经更加平民化；不过，这时个人计算机革命还未到来。

最初，研究人员认为TCP协议只适用于大型系统，因为TCP就是为大型系统设计的。不过，麻省理工学院David Clark的研究小组发现，工作站也可以与大型机互联。Clark的研究，再加上20世纪80年代和90年代个人计算机领域的爆炸式发展，为网络的发展铺平了道路。

20世纪80年代出现了几个大变化。随着主机数量从很少发展到成千上万，需要为主机指定不同的名字，这样人们就不用费劲地去记它们的数字地址。这个变化，以及主机数量的飞速增长，催生了DNS。另外，ARPANET从使用NCP转为使用TCP/IP协议，后者是军方使用的标准协议。到了20世纪80年代中期，因特网已经成为一个连接不同研究人员群体的平台，而且其他网络也开始出现：美国国家航空航天局（NASA）创建了SPAN；美国能源部建立了MFENet来研究磁聚变能源，另外在美国国家科学基金会（National Science Foundation）的资助下，还创建了CSNET来开展计算机科学的研究。

1989年，欧洲粒子物理研究中心（CERN）的Tim Berners-Lee提出了一个很有意思的概念。他认为，与其简单地引用其他人的著作，不如进行实际的链接。读一篇文章时，读者可以打开所引用的其他文章。超文本（hypertext）当时相当流行，Berners-Lee还利用了他先前在文档和文本处理方面的研究成果，发明了标准通用标记语言（Standard Generalized Markup Language, SGML）的一个子集，称为超文本标记语言（HyperText Markup Language, HTML）。HTML的妙处在于，它能将有关文本显示方式的信息与具体显示的实现相分离。Berners-Lee不仅创建了一个称为超文本传输协议（HyperText Transfer Protocol, HTTP）的简单协议，还发明了第一个Web浏览器，叫做WorldWideWeb。

## 1.2 浏览器历史

提到Web浏览器，大多数人都会想到无处不在的Microsoft Internet Explorer，直到最近像Firefox、Safari和Opera之类的浏览器日益兴起，这种情况才稍有改观。许多新手可能会误认为IE是市场上的第一个浏览器，其实不然。实际上，第一个Web浏览器出自Berners-Lee之手，这是他为NeXT计算机创建的（这个Web浏览器原来取名叫WorldWideWeb，后来改名为Nexus），并在1990年发布给CERN的人员使用。Berners-Lee和Jean-Francois Groff

将 WorldWideWeb 移植到 C，并把这个浏览器改名为 libwww。20世纪90年代初出现了许多浏览器，包括 Nicola Pellow 编写的行模式浏览器（这个浏览器允许任何系统的用户都能访问 Internet，从 Unix 到 Microsoft DOS 都涵盖在内），还有 Samba，这是第一个面向 Macintosh 的浏览器。

1993年2月，伊利诺伊大学 Urbana-Champaign 分校美国国家超级计算应用中心的 Marc Andreessen 和 Eric Bina 发布了 Unix 版本的 Mosaic。几个月之后，Aleks Totic 发布了 Mosaic 的 Macintosh 版本，这使得 Mosaic 成为第一个跨平台浏览器，它很快得到普及，并成为最流行的 Web 浏览器<sup>1</sup>。这项技术后来卖给了 Spyglass，最后又归入 Microsoft 的门下，并应用在 Internet Explorer 中。

1993年，堪萨斯大学的开发人员编写了一个基于文本的浏览器，叫做 Lynx，它成为了字符终端的标准。1994年，挪威奥斯陆的一个小组开发了 Opera，到1996年这个浏览器得到了广泛使用。1994年12月，Netscape 发布了 Mozilla 的 1.0 版，第一个盈利性质的浏览器从此诞生。2002 年又发布了一个开源的版本，这最终发展为 2004 年 11 月发布的、现在十分流行的 Firefox 浏览器。

当 Microsoft 发布 Windows 95 时，IE 1.0 是作为 Microsoft Plus! 包的一部分同时发布的。尽管这个浏览器与操作系统集成在一起，但大多数人还是坚持使用 Netscape、Lynx 或 Opera。IE 2.0 有了很大起色，增加了对 cookie、安全套接字层（Secure Socket Layer，SSL）和其他新兴标准的支持。2.0 版还可以用于 Macintosh，从而成为 Microsoft 的第一个跨平台浏览器。不过，大多数用户还是很执着，仍然坚持使用他们习用的浏览器。

不过到了 1996 年夏天，Microsoft 发布了 IE 3.0 版。几乎一夜之间，人们纷纷拥向 IE。当时，Netscape 的浏览器是要收费的，Microsoft 则免费提供 IE。关于浏览器领域谁主沉浮，因特网社区发生了两极分化，很多人担心 Microsoft 会像在桌面领域一样，在 Web 领域也一统天下。有些人则考虑到安全因素——果然不出所料，发布 3.0 版 9 天之后就报告了第一个安全问题。但是到 1999 年发布 IE 5 时，它已经成为使用最广的浏览器。

### 1.3 Web 应用的发展历程

最初，所有 Web 页面都是静态的，用户请求一个资源，服务器再返回这个资源。什么都不动，什么都不闪。坦率地讲，对于许多 Web 网站来说，这样也是可以的，这些网站的 Web 页面只是电子形式的文本，在一处生成，内容固定，再发布到多处。在浏览器发展的最

1. 本书的一位作者这样回忆最初接触 Mosaic 的情形：“那时我刚接触 Lynx，还是个化学专业的新生，我惊讶地发现，居然能在明尼苏达州浏览英国牛津大学的书库（尽管只是基于文本的浏览）。不过使用了 Mosaic 的 Beta 版之后，我觉得实在太慢了，也太不稳定了，所以我还是坚持使用了 Lynx。现在可以自豪地说，我用的是 Firefox。”

初阶段，Web 页面的这种静态性不成问题，科学家只是使用因特网来交换研究论文，大学院校也只是通过因特网在线发布课程信息。企业界还没有发现这个新“渠道”会提供什么商机。实际上，以前公司主页显示的信息通常很少，无非是一些联系信息或者只是一些文档。不过没过多久，Web 用户就开始有新的要求了，希望能得到更动态的网上体验。个人计算机成为企业不可或缺的资源，而且从个人宿舍到住家办公室开始出现越来越多的计算机。随着 Windows 95 的问世，随着人们已经领教了 Corel WordPerfect 和 Microsoft Excel 丰富的功能，用户的期望也越来越高。

### 1.3.1 CGI

要让 Web 更为动态，第一个办法是公共网关接口（Common Gateway Interface，CGI）。与静态的 Web 获取不同，使用 CGI 可以创建程序，当用户发出请求时就会执行这个程序。假设要在 Web 网站上显示销售的商品，你可以利用 CGI 脚本来访问商品数据库，并显示结果。通过使用简单的 HTML 表单和 CGI 脚本，可以创建简单的网上店面，这样别人就可以通过浏览器来购买商品。编写 CGI 脚本可以用多种语言，从 Perl 到 Visual Basic 都可以，这使得掌握不同编程语言的人都能编写 CGI 脚本。

不过，要创建动态的 Web 页面，CGI 并不是最安全的方法。如果采用 CGI，将允许别人在你的系统上执行程序。大多数情况下这可能没有问题，但是倘若某个用户有恶意企图，则很可能会利用这一点，让系统运行你本来不想运行的程序。尽管存在这个缺陷，到如今 CGI 仍在使用。

### 1.3.2 applet

很显然，CGI 可以有所改进。1995 年 5 月，Sun 公司的 John Gage 和 Andreessen（目前在 Netscape 通信公司）宣布一种新的编程语言诞生，这就是 Java。Netscape Navigator 为这种新语言提供了支持，最初是为了支持机顶盒。（你可能原认为最早涉足智能家居的公司是 Microsoft 和 Sony，其实不然。）就像所有革命都机缘巧合一样，Java 和因特网的出现恰到好处，在适当的时间、适当的地点横空出世，Java 在 Web 上发布仅几个月，就已经有成千上万的人下载。由于 Netscape 的 Navigator 支持 Java，动态 Web 页面掀开了新的一页：applet 时代到来了。

applet 允许开发人员编写可嵌入在 Web 页面上的小应用程序。只要用户使用支持 Java 的浏览器，就可以在浏览器的 Java 虚拟机（Java Virtual Machine，JVM）中运行 applet。尽管 applet 可以做很多事情，但它也存在一些限制：通常不允许它读写文件系统，它也不能加载本地库，而且可能无法启动客户端上的程序。除了这些限制外，applet 是在一个沙箱安全模型中运行的，这是为了有助于防止用户运行恶意代码。

对许多人来说，最初接触 Java 编程语言就是从 applet 开始的，当时这是创建动态 Web 应用的一种绝好的方法。applet 允许你在浏览器中创建一个胖客户应用，不过要在平台的安全限制范围内。当时，在很多领域都广泛使用了 applet，但是，Web 社区并没有完全被 applet “征服”<sup>1</sup>。胖客户的开发人员都很熟悉一个问题：必须在客户端上部署适当的 Java 版本。因为 applet 在浏览器的虚拟机中运行，所以开发人员必须确保客户端安装了适当版本的 Java。尽管这个问题也可以解决，但它确实妨碍了 applet 技术的进一步推广。而且如果 applet 写得不好，很可能对客户主机造成影响，这使许多客户对于是否采用基于 applet 的解决方案犹豫不定。如果你还不太熟悉 applet，请看图 1-1，图中显示了 Sun 公司提供的时钟 applet。

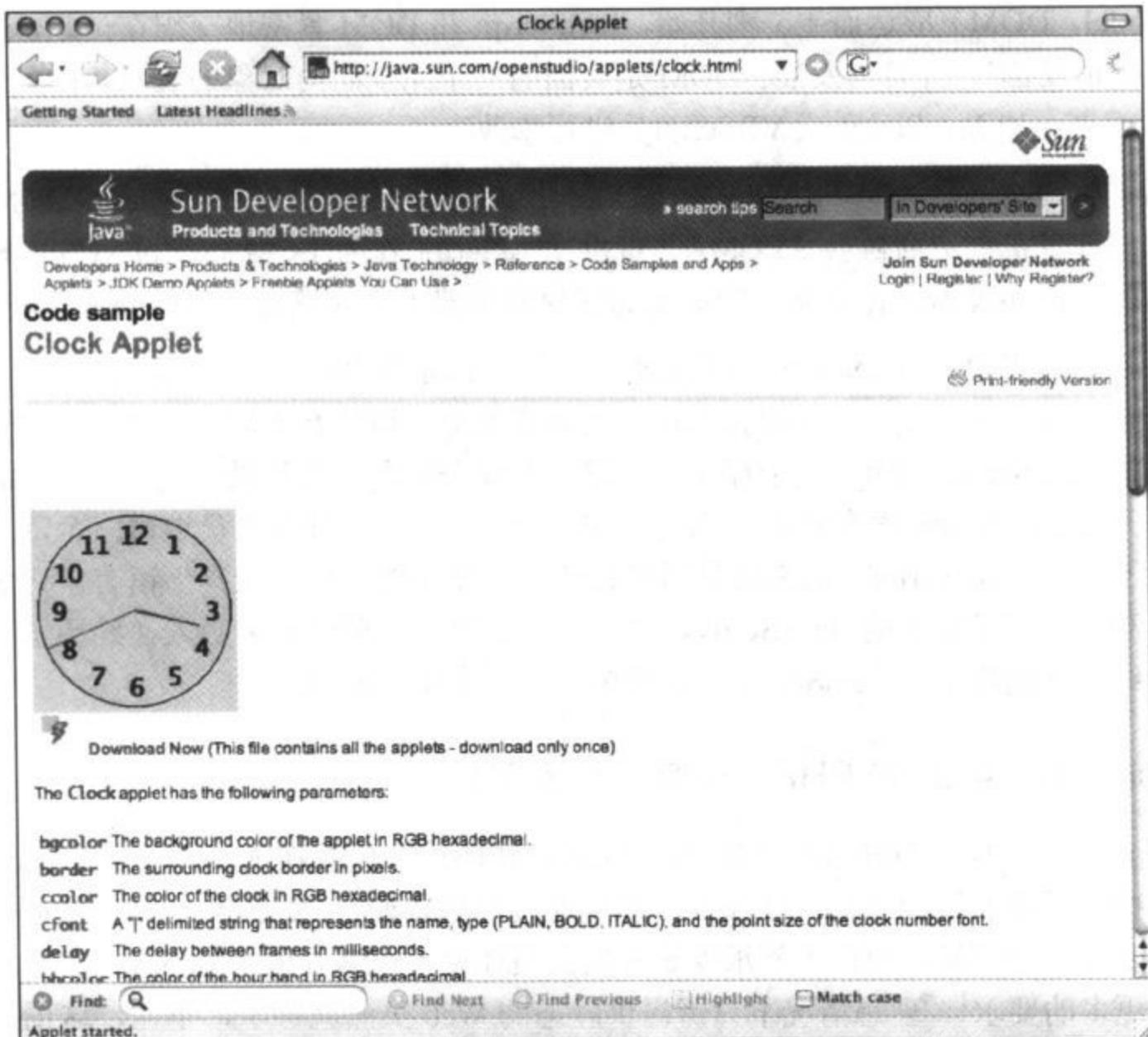


图 1-1 Sun 的时钟 applet

### 1.3.3 JavaScript

与此同时，Netscape 创建了一种脚本语言，并最终命名为 JavaScript（建立原型时叫做

1. 你相信吗？本书的一位作者所在公司的打卡计时软件就是一个 applet。

Mocha，正式发布之前曾经改名为 LiveWire 和 LiveScript，不过最后终于确定为 JavaScript）。设计 JavaScript 是为了让不太熟悉 Java 的 Web 设计人员和程序员能够更轻松地开发 applet（当然，Microsoft 也推出了与 JavaScript 相对应的脚本语言，称为 VBScript）。Netscape 请 Brendan Eich 来设计和实现这种新语言，他认为市场需要的是一种动态类型脚本语言。由于缺乏开发工具，缺少有用的消息和调试工具，JavaScript 很受非议，但尽管如此，JavaScript 仍然是一种创建动态 Web 应用的强大方法。

最初，创建 JavaScript 是为了帮助开发人员动态地修改页面上的标记，以便为客户提供更丰富的体验。人们越来越认识到，页面也可以当作对象，因此文档对象模型（Document Object Model，DOM）应运而生。刚开始，JavaScript 和 DOM 紧密地交织在一起，但最后它们还是“分道扬镳”，并各自发展。DOM 是页面的一个完全面向对象的表示，该页面可以用某种脚本语言（如 JavaScript 或 VBScript）进行修改。

最后，万维网协会（World Wide Web Consortium，W3C）介入，并完成了 DOM 的标准化，而欧洲计算机制造商协会（ECMA）批准 JavaScript 作为 ECMAScript 规约。根据这些标准编写的页面和脚本，在遵循相应原则的任何浏览器上都应该有相同的外观和表现。

在最初的几年中，JavaScript 的发展很是坎坷，这是许多因素造成的。首先，浏览器支持很不一致，即使是今天，同样的脚本在不同浏览器上也可能有不同的表现；其次，客户可以自由地把 JavaScript 关闭，由于存在一些已知的安全漏洞，往往鼓励用户把 JavaScript 关掉。由于开发 JavaScript 很有难度（你会用 alert 吗？），许多开发人员退避三舍，有些开发人员干脆不考虑 JavaScript，认为这是图形设计人员使用的一种“玩具”语言。许多人曾试图使用、测试和调试复杂的 JavaScript，并为此身心俱疲，所以大多数人在经历了这种痛苦之后，最终只能满足于用 JavaScript 创建简单的基于表单的应用。

### 1.3.4 servlet、ASP 和 PHP……哦，太多了！

尽管 applet 是基于 Web 的，但胖客户端应用存在的许多问题在 applet 上也有所体现。在大量使用拨号连接的年代（就算是今天，拨号连接也很普遍），要下载一个复杂 applet 的完整代码，要花很多时间，用户不能承受。开发人员还要考虑客户端上的 Java 版本，有些虚拟机还有更多的要求<sup>1</sup>。理想情况下只需提供静态的 Web 页面就够了，毕竟，这正是设计因特网的本来目的。当然，尽管静态页面是静态的，但是如果能在服务器上动态地生成内容，再把静态的内容返回，这就太好了。

在 Java 问世一年左右，Sun 引入了 servlet。现在 Java 代码不用再像 applet 那样在客户端浏览器中运行了，它可以在你控制的一个应用服务器上运行。这样，开发人员就能充分利用

---

1. Microsoft 虚拟机不支持 1.1 以后版本的 Java，这大大阻碍了 applet 在 Microsoft 平台上的作为。

用现有的业务应用，而且，如果需要升级为最新的 Java 版本，只需要考虑服务器就行了。Java 推崇“一次编写，到处运行”，这一点使得开发人员可以选择最先进的应用服务器和服务器环境，这也是这种新技术的另一个优点。servlet 还可以取代 CGI 脚本。

servlet 向前迈出了很大一步。servlet 提供了对整个 Java 应用编程接口（API）的完全访问，而且提供了一个完备的库可以处理 HTTP。不过，servlet 不是十全十美的。使用 servlet 设计界面可能很困难。在典型的 servlet 交互中，先要从用户那里得到一些信息，完成某种业务逻辑，然后使用一些“打印行”创建 HTML，为用户显示结果。代码清单 1-1 所示的代码就相当常见。

### 代码清单 1-1 简单的 servlet 代码

```
response.setContentType("text/html; charset=UTF-8");
PrintWriter out = response.getWriter();

out.println("<html>");
out.println("<head>");
out.println("<title>Servlet SimpleServlet</title>");
out.println("</head>");
out.println("<body>");
out.println("<h1>Hello World</h1>");
out.println("<p>Imagine if this were more complex.</p>");
out.println("</body>");
out.println("</html>");

out.close();
```

以上这一小段代码可以生成图 1-2 所示的一个相当简单的 Web 页面。

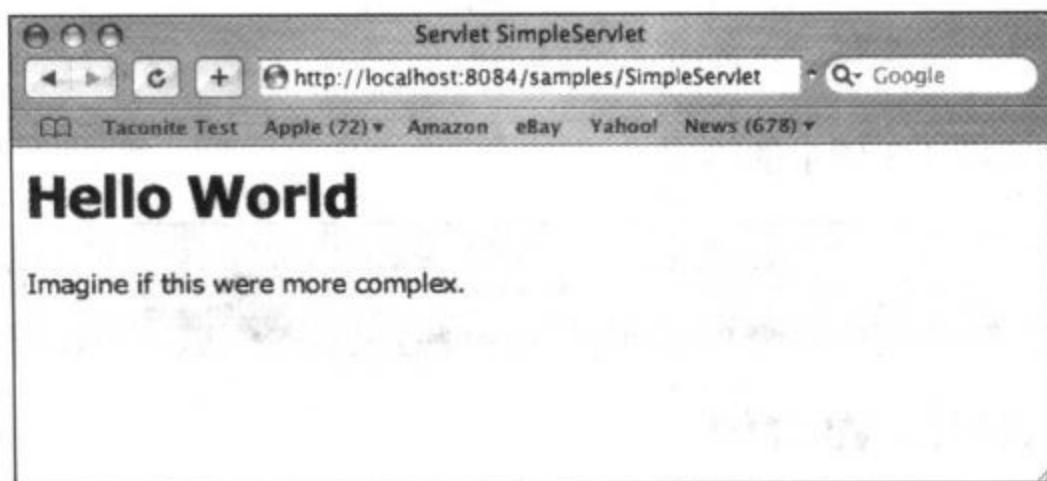


图 1-2 代码清单 1-1 中简单 servlet 的输出

servlet 不仅容易出错，很难生成可视化显示，而且还无法让开发者尽展其才。一般地，编写服务器端代码的人往往是软件开发人员，他们只是对算法和编译器很精通，但不是能设计公司精美网站的图形设计人员。业务开发人员不仅要编写业务逻辑，还必须考虑怎么创建

一致的设计。因此，很有必要将表示与业务逻辑分离。因此 JSP (JavaServer Pages) 出现了。

在某种程度上，JSP 是对 Microsoft 的 Active Server Pages (ASP) 做出的回应。Microsoft 从 Sun 在 servlet 规约上所犯的错误汲取了教训，并创建了 ASP 来简化动态页面的开发。Microsoft 增加了非常好的工具支持，并与其 Web 服务器紧密集成。JSP 和 ASP 的设计目的都是为了将业务处理与页面外观相分离，从这个意义上讲，二者是相似的。虽然存在一些技术上的差别（Sun 也从 Microsoft 那里学到了教训），但它们有一个最大的共同点，即 Web 设计人员能够专心设计页面外观，而软件开发人员可以专心开发业务逻辑。代码清单 1-2 显示了一个简单的 JSP。

### 代码清单 1-2 简单的 JSP

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
 "http://www.w3.org/TR/html4/loose.dtd">

<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Hello World</title>
  </head>
  <body>

    <h1>Hello World</h1>
    <p>This code is more familiar for Web developers.</p>

  </body>
</html>
```

这个代码会生成图 1-3 所示的输出。



图 1-3 简单 JSP 的输出

当然，Microsoft 和 Sun 并没有垄断服务器端解决方案。还有许多其他的方案在这个领域都有一席之地，如 PHP 和 ColdFusion 等等。有些开发人员喜欢新奇的工具，还有一些则倾向于更简单的语言。目前来看，所有这些解决方案完成的任务都是一样的，它们都是要动态生成 HTML。在服务器端生成内容可以解决发布问题。不过，与使用胖客户或 applet 所做的工作相比，用户从原始 HTML 得到的体验就太过单调和苍白了。下面几节将介绍几种力图提供更丰富用户体验的解决方案。

### 1.3.5 Flash

并不是只有 Microsoft 和 Sun 在努力寻找办法来解决动态 Web 页面问题。1996 年夏天，FutureWave 发布了一个名叫 FutureSplash Animator 的产品。这个产品起源于一个基于 Java 的动画播放器，FutureWave 很快被 Macromedia 兼并，Macromedia 则将这个产品改名为 Flash。

利用 Flash，设计人员可以创建令人惊叹的动态应用。公司可以在 Web 上发布高度交互性的应用，几乎与胖客户应用相差无几（见图 1-4）。不同于 applet、servlet 和 CGI 脚本，Flash 不需要编程技巧，很容易上手。在 20 世纪 90 年代末期，掌握 Flash 是一个很重要的特长，因为许多老板都非常需要有这种技能的员工。不过，这种易用性也是有代价的。

The screenshot shows a Flash-based web application for hotel reservations. At the top, there's a navigation bar with links for "Getting Started" and "Latest Headlines". The main header reads "The Broadmoor COLORADO SPRINGS". On the left, a sidebar lists months from June to March. Below it is a legend for room types: "rooms" (blue), "adults" (green), and "children" (orange). A "reset" button is also present. The central part of the screen features a calendar for July and August. A dropdown menu titled "Choose a room below and availability will be displayed on the calendar" lists room types: Classic, Superior, Deluxe, Elite, Premier, and Suite. To the right, a form for completing the reservation is shown, including fields for "check in" (July 7, 2005), "check out" (July 7, 2005), "room type" (superior), "nights" (0), "adults" (1), "rooms" (1), "children" (0), and "amount". Below this is a grid showing room availability. A small image of a room interior is visible. At the bottom, there's a note about room features, a link to "View General Terms & Conditions", and a "comment/requests" section. A "finish reservation" button is at the very bottom.

图 1-4 Flash 应用

像许多解决方案一样，Flash 需要客户端软件。尽管许多流行的操作系统和浏览器上都内置有所需的 Shockwave 播放器插件，但并非普遍都有。虽然能免费下载，但由于担心感染病毒，使得许多用户都拒绝安装这个软件。Flash 应用可能还需要大量网络带宽才能正常地工作，另外，由于没有广泛的宽带连接，Flash 的推广受到局限（因此产生了“跳过本页”之类的链接）。虽然确有一些网站选择建立多个版本的 Web 应用，分别适应于不同的连接速度，但是许多公司都无法承受支持两个或更多网站所增加的开发开销。

总之，创建 Flash 应用需要专用的软件和浏览器插件。applet 可以用文本编辑器编写，而且有一个免费的 Java 开发包，Flash 则不同，使用完整的 Flash 工具包需要按用户数付费，每个用户需要数百美元。尽管这些因素不是难以逾越的障碍，但它们确实减慢了 Flash 在动态 Web 应用道路上的前进脚步。

### 1.3.6 DHTML 革命

当 Microsoft 和 Netscape 发布其各自浏览器的第 4 版时，Web 开发人员有了一个新的选择：动态 HTML (Dynamic HTML, DHTML)。与有些人想像的不同，DHTML 不是一个 W3C 标准，它更像是一种营销手段。实际上，DHTML 结合了 HTML、层叠样式表 (Cascading Style Sheets, CSS)、JavaScript 和 DOM。这些技术的结合使得开发人员可以动态地修改 Web 页面的内容和结构。

最初 DHTML 的反响很好。不过，它需要的浏览器版本还没有得到广泛采用。尽管 IE 和 Netscape 都支持 DHTML，但是它们的实现大相径庭，这要求开发人员必须知道他们的客户使用什么浏览器。而这通常意味着需要大量代码来检查浏览器的类型和版本，这就进一步增加了开发的开销。有些人对于尝试这种方法很是迟疑，因为 DHTML 还没有一个官方的标准。不过，将来新标准有可能会出现。

### 1.3.7 XML 衍生语言

20 世纪 90 年代中期，基于 SGML 衍生出了 W3C 的可扩展标记语言 (eXtensible Markup Language, XML)，自此以后，XML 变得极为流行。许多人把 XML 视为解决所有计算机开发问题的灵丹妙药，以至于 XML 几乎无处不在。实际上，Microsoft 就已经宣布，Office 12 将支持 XML 文件格式。

如今，我们至少有 4 种 XML 衍生语言可以用来创建 Web 应用 (W3C 的 XHTML 不包括在内)：Mozilla 的 XUL；XAMJ，这是结合 Java 的一种开源语言；Macromedia 的 MXML；Microsoft 的 XAML。

XUL: XUL(读作“zool”)代表 XML 用户界面语言 (XML User Interface Language)，由 Mozilla 基金会推出。流行的 Firefox 浏览器和 Thunderbird 邮件客户端都是用 XUL 编

写的。利用 XUL，开发人员能构建功能很丰富的应用，这个应用可以与因特网连接，也可以不与因特网连接。为了方便那些熟悉 DHTML 的开发人员使用，XUL 设计为可以跨平台支持诸如窗口和按钮等标准界面部件。虽然 XUL 本身不是标准，但它是基于各种标准的，如 HTML 4.0、CSS、DOM、XML 和 ECMAScript 等等。XUL 应用可以在浏览器上运行，也可以安装在客户端主机上。

当然，XUL 也不是没有缺点。它需要 Gecko 引擎，而且目前 IE 还没有相应的插件。尽管 Firefox 在浏览器市场中已经有了一定的份额，但少了 IE 的支持还是影响很大，大多数应用都无法使用 XUL。目前开展的很多项目都是力图在多个平台上使用 XUL，包括 Eclipse。

**XAML:** XAML（读作“zammel”）是 Microsoft 即将推出的操作系统（名为 Windows Vista）的一个组件。XAML 是可扩展应用标记语言（eXtensible Application Markup Language）的缩写，它为使用 Vista 创建用户界面定义了标准。与 HTML 类似，XAML 使用标记来创建标准元素，如按钮和文本框等。XAML 建立在 Microsoft 的 .NET 平台上，而且可以编译为 .NET 类。

XAML 的局限应当很清楚。作为 Microsoft 的产品，它要求必须使用 Microsoft 的操作系统。多数情况下特别是在大公司中，这可能不成问题，但是有些小公司使用的不是 Microsoft 的操作系统，总不能削足适履吧，就像是没有哪家公司会因为买家没有开某种牌子的车来就把他拒之门外。Vista 交付的日期一再推迟，与此同时 XAML 也有了很大变化，不再只是一个播放器。据说，在未来几年内，我们可能会看到一个全新的 XAML。

**MXML:** Macromedia 创建了 MXML，作为与其 Flex 技术一同使用的一种标记语言。MXML 是最佳体验标记语言（Maximum eXperience Markup Language）的缩写，它与 HTML 很相似，可以以声明的方式来设计界面。与 XUL 和 XAML 类似，MXML 提供了更丰富的界面组件，如 DataGrid 和 TabNavigator，利用这些组件可以创建功能丰富的因特网应用。不过，MXML 不能独立使用，它依赖于 Flex 和 ActionScript 编程语言来编写业务逻辑。

MXML 与 Flash 有同样的一些限制。它是专用的，而且依赖于价格昂贵的开发和部署环境。尽管将来.NET 可能会对 MXML 提供支持，但现在 Flex 只能在 J2EE 应用服务器上运行，如 Tomcat 和 IBM 的 WebSphere，这就进一步限制了 MXML 的广泛采用。

**XAMJ:** 让人欣喜的是，开源社区又向有关界面设计的 XML 衍生语言领域增加了新的成员。XAMJ 作为另一种跨平台的语言，为 Web 应用开发人员又提供了一个工具。这种衍生语言基于 Java，而 Java 是当前最流行的面向对象语言之一，XAMJ 也因此获得了面向对象语言的强大功能。XAMJ 实际上想要替代基于 XAML 或 HTML 的应用，力图寻找一种更为安全的方法，既不依赖于某种特定的框架，也不需要高速的因特网连

接。**XAMJ** 是一种编译型语言，建立在“clientlet”（小客户端）体系结构之上，尽管基于**XAMJ** 的程序也可以是独立的应用，但通常都是基于 Web 的应用。在撰写本书时，**XAMJ** 还太新，我们还没有听到太多批评的声音。不过，批评是肯定会有的，让我们拭目以待。

当谈到“以 X 开头的东西”时，别忘了 W3C XForms 规约。XForms 设计为支持更丰富的用户界面，而且能够将数据与表示解耦合。毋庸置疑，XForms 数据是 XML，这样你就能使用现有的 XML 技术，如 XPath 和 XML Schema。标准 HTML 能做的，XForms 都能做，而且 XForms 还有更多功能，包括动态检查域值、与 Web 服务集成等等。不同于其他的许多 W3C 规约，XForms 不需要新的浏览器，你可以使用现在已有的许多浏览器去实现。与大多数 XML 衍生语言一样，XForms 是一种全新的方法，所以它要得到采纳尚需时日。

### 1.3.8 基本问题

有了以上了解，你怎么想？即使是最苛刻的客户应用，也已经把 Web 作为首选平台。很显然，基于 Web 的应用很容易部署，这种低门槛正是 Web 应用最耀眼的地方。由于浏览器无处不在，而且无需下载和安装新的软件，用户利用基于浏览器的客户端就能很轻松地尝试新的应用。用户只需点击一个链接就能运行你的应用程序，而不用先下载几兆比特的安装程序才行。基于浏览器的应用也不考虑操作系统是什么，也就是说，不仅使用不同操作系统（如 Linux 和 Mac OS X）的人能运行你的应用程序，而且你也不必考虑针对不同的操作系统开发和维护多个安装包。

既然基于 Web 的应用是前所未有的好东西，那我们为什么还要写这本书？如果回头看看因特网的起源就可以知道，最初因特网实际上就是为了科学家们和学术机构间交换文章和研究成果，这是一种简单的请求/响应模式。那时不需要会话状态，也不需要购物车，人们只是在交换文档。但现在你有很多办法来创建动态的 Web 应用，如果想让应用真正深入人心，赢得大量的用户，就必须在浏览器上大做文章，这说明，因特网以请求/响应模式作为基础，由此带来的同步性对你造成了妨碍。

与 Microsoft Word 或 Intuit Quicken 之类的胖客户应用相比，Web 模型当然只能根据一般用户需要做折中考虑。不过，由于 Web 应用很容易部署，而且浏览器的发展相当迅速，这意味着大多数用户都已经学会了适应。但是，还是有许多人认为 Web 应用只能算“二等公民”，给人的用户体验不是太好。因为因特网是一个同步的请求/响应系统，所以浏览器中的整个页面会进行刷新。最初，这种简单的请求并没有什么问题。如果用户做了一两处修改，就必须向服务器发回整个文档，而且要重新绘制整个页面。尽管这样是可行的，但是这种完全刷新的局限，意味着应用确实还很粗糙。

这并不是说开发人员只是袖手旁观，全然接受这种状况。Microsoft 对于交互式应用有

一定了解，而且对于这种标准请求/响应模式的限制一直都不满意，因此提出了远程脚本（remote scripting）的概念。远程脚本看似神奇，其实很简单：它允许开发人员创建以异步方式与服务器交互的页面。例如，顾客可以从下拉列表中选择状态，这样就会在服务器上运行一个脚本，计算顾客的运费。更重要的是，显示这些运费时无需刷新整个页面！当然，Microsoft 的方案只适用于它自己的技术，而且需要 Java，但有了这个进步，说明更丰富的浏览器应用并不是海市蜃楼。

对于同步页面刷新问题还有其他一些解决方案。针对 Microsoft 的远程脚本，Brent Ashley 在创建 JavaScript 远程脚本（JavaScript Remote Scripting, JSRS）时开发了一个平台中立（独立于平台）的方案。JSRS 依赖于一个客户端 JavaScript 库和 DHTML，可以向服务器做异步的调用。与此同时，许多人利用了 IFRAME 标记，可以只加载页面中的某些部分，或者向服务器做“隐藏”的调用。尽管这是一个可行的方法，而且也为很多人所用，但它肯定不是最理想的，还有待改善。

### 1.3.9 Ajax

终于谈到这里了：客户希望得到一个功能更完备的应用，而开发人员想避开繁琐的部署工作，不想把可执行文件逐个地部署到数以千计的工作站上。我们已经做过很多尝试，但是任何方法都不像它原来标榜的那么完美。不过，最近一个极其强大的工具横空出世了。

是的，我们又有了一个新的选择，新的工具，可以创建的确丰富的基于浏览器的应用。这就是 Ajax。Ajax 不只是一个特定的技术，更应算是一种技巧，不过前面提到的 JavaScript 是其主要组件。我们知道，你可能会说“JavaScript 根本不值一提”，但是由于 Ajax 的出现，人们对这种语言又有了新的兴趣，应用和测试框架再加上更优秀的工具支持，减轻了开发人员肩头的重担。随着 Atlas 的引入，Microsoft 对 Ajax 投入了大力支持，而名声不太好的 Rails Web 框架也预置了充分的 Ajax 支持。在 Java 世界中，Sun 已经在其 BluePrints Solutions Catalog 中增加了许多 Ajax 组件。

坦率地讲，Ajax 并不是什么新鲜玩艺。实际上，与这个词相关的“最新”术语就是 XMLHttpRequest 对象（XHR），它早在 IE 5（于 1999 年春天发布）中就已经出现了，是作为 Active X 控件露面的。不过，最近出现的新现象是浏览器的支持。原先，XHR 对象只在 IE 中得到支持（因此限制了它的使用），但是从 Mozilla 1.0 和 Safari 1.2 开始，对 XHR 对象的支持开始普及。这个很少使用的对象和相关的基本概念甚至已经出现在 W3C 标准中：DOM Level 3 加载和保存规约（DOM Level 3 Load and Save Specification）。现在，特别是随着 Google Maps、Google Suggest、Gmail、Flickr、Netflix 和 A9 等应用变得越来越炙手可热，XHR 也已经成为事实上的标准。

与前面几页提到的方法不同，Ajax 在大多数现代浏览器中都能使用，而且不需要任何

专门的软件或硬件。实际上，这种方法的一大优势就是开发人员不需要学习一种新的语言，也不必完全丢掉他们原先掌握的服务器端技术。Ajax 是一种客户端方法，可以与 J2EE、.NET、PHP、Ruby 和 CGI 脚本交互，它并不关心服务器是什么。尽管存在一些很小的安全限制，你还是可以现在就开始使用 Ajax，而且能充分利用你原有的知识。

你可能会问：“谁在使用 Ajax？”前面已经提到，Google 显然是最早采用 Ajax 的公司之一，而且已经用在很多技术上，随便说几个应用，如 Google Maps、Google Suggest 和 Gmail。Yahoo!也开始引入 Ajax 控件，另外 Amazon 提供了一个简洁的搜索工具，其中大量使用了这个技术，例如，在钻石的某一方面上移动滑块，这会带来动态更新的结果（见图 1-5）。并不是每次改变你的查询标准时都会更新页面，而是在移动滑块时就会查询服务器，从而更快、更容易地缩小你的选择范围。

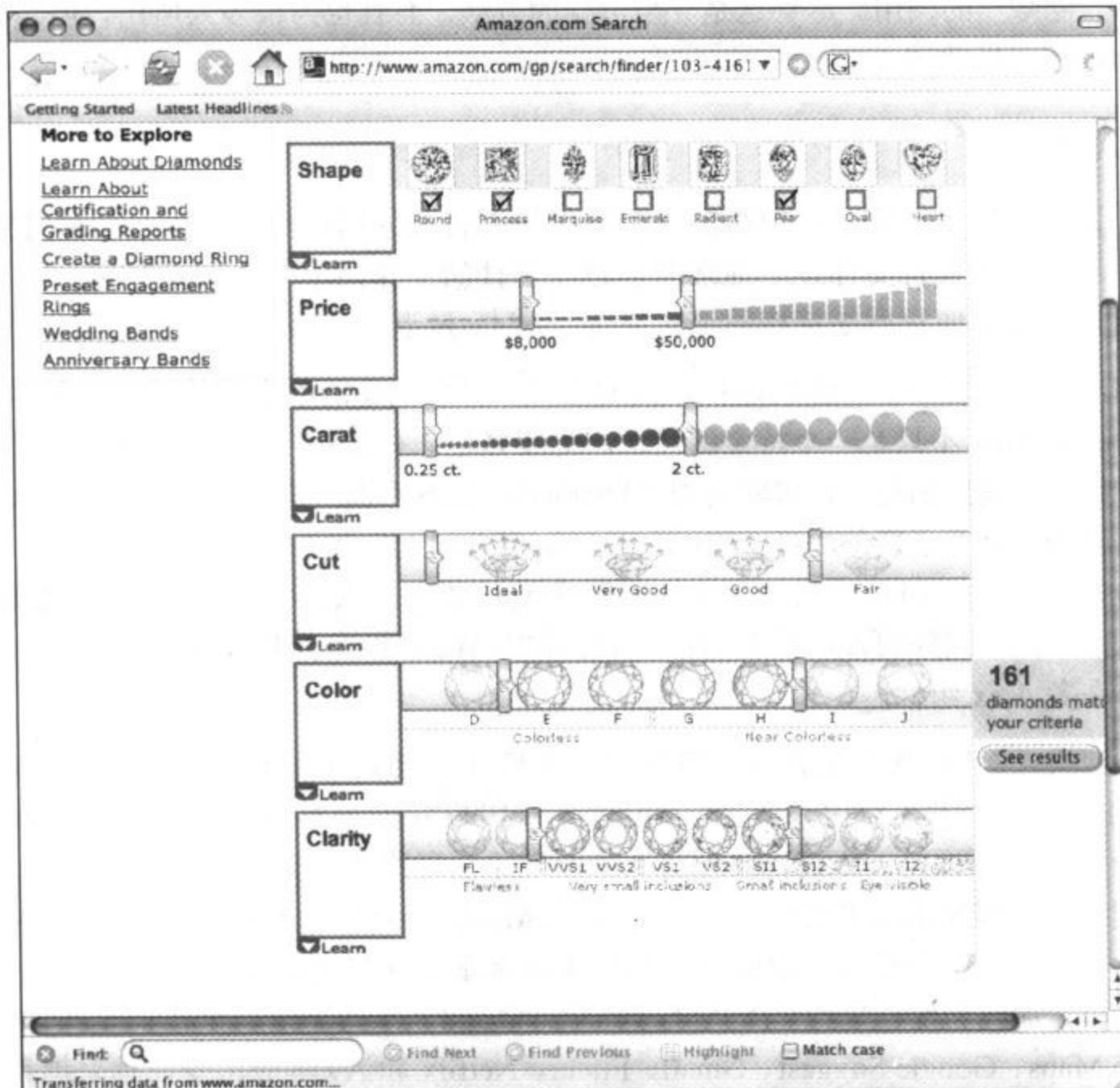


图 1-5 Amazon 的钻石搜索页面

Netflix 是一家很有名的 DVD 租借公司，它也使用了 Ajax。当用户浏览影片时，可以得

到更详细的信息。当顾客把鼠标放在一个影片的图片上时，这个影片的 ID 就会发送到中心服务器，然后会出现一个“气泡”，提供这个影片的更多细节（见图 1-6）。同样，页面并不会刷新，每个影片的详细信息并不是放在隐藏的表单域中。利用这种方法，Netflix 可以提供影片的更多信息，而不会把页面弄乱。顾客浏览起来也更容易，他们不必点击影片，看完影片详细信息后再点击回到影片列表页面；他们只需把鼠标停在影片上，就这么简单！我们想要强调的是，Ajax 并不限于 .com 之类的网站使用，普通公司的开发人员也开始涉足这个技术，有些人已经在使用 Ajax 来改善原来很丑陋的验证方案，或者用于动态地获取数据了。



图 1-6 Netflix 的浏览页面特性

关键在于，因特网默认的请求/响应模式有了重大转变，这正是 Ajax 的核心所在，尽管这并非全新的内容。Web 应用开发人员现在可以自由地与服务器异步交互，这说明他们可以完成许多原本只能在胖客户上完成的任务。例如，当用户输入邮政编码时，可以验证它是否正确，然后自动用相应城市名和州名填充到表单中的其他部分；或者，当用户选择美国时，

可以引入美国各个州的一个下拉列表。以前也可以用其他方式模拟这些工作，但是使用 Ajax 的话，这些工作会更加简单。

那么，是谁发明了 Ajax？要找出真正的源头，总免不了一场争论。不过有一点是确定的，2005 年 2 月，Adaptive Path 的 Jesse James Garrett 最早创造了这个词。在他的文章 *Ajax: A New Approach to Web Applications* (Ajax: Web 应用的一种新方法) 中，Garrett 讨论了如何消除胖客户（或桌面）应用与瘦客户（或 Web）应用之间的界限。当然，当 Google 在 Google Labs 发布 Google Maps 和 Google Suggest 时，这个技术才真正为人所认识，而且此前已经有许多这方面的文章了。但确实是 Garrett 最早提出了这个好名字，否则我们就得啰嗦地不说一大堆：异步（Asynchronous）、XMLHttpRequest、JavaScript、CSS、DOM 等等。尽管原来把 Ajax 认为是 Asynchronous JavaScript + XML (异步 JavaScript + XML) 的缩写，但如今，这个词的覆盖面有所扩展，把允许浏览器与服务器通信而无需刷新当前页面的技术都涵盖在内。

你可能会说：“哦，那有什么大不了的？”这么说吧，使用 XHR 而且与服务器异步通信，就能创建更加动态的 Web 应用。例如，假设你有一个下拉列表，它是根据另外一个域或下拉列表的输入来填写的。在正常情况下，必须在加载第一个页面时把所有数据都发送给客户端，然后使用 JavaScript 根据输入来填写下拉列表。这么做并不困难，但是会让页面变得很臃肿，取决于这个下拉列表到底有多“动态”，页面有可能膨胀得过大，从而出现问题。利用 Ajax，当作为触发源的域有变化，或者失去了输入焦点，就可以向服务器发一个简单的请求，只要求得到更新下拉列表所需的部分信息即可。

来单独考虑一下验证。你写过多少次 JavaScript 验证逻辑？用 Java 或 C# 编写验证逻辑可能很简单，但是由于 JavaScript 缺乏很好的调试工具，再加上它是一种弱类型语言，所以用 JavaScript 编写验证逻辑实在是一件让人头疼的事情，而且很容易出错。服务器上还很有可能重复这些客户端验证规则。使用 XHR，可以对服务器做一个调用，触发某一组验证规则。这些规则可能比你用 JavaScript 编写的任何规则都更丰富、更复杂，而且你还能得到功能强大的调试工具和集成开发环境（IDE）。

你现在可能又会说：“这些事情我早已经用 IFRAME 或隐藏框架做到了。”我们甚至还会使用这种技术来提交或刷新过页面的一部分，而不是整个浏览器（页面）。不能不承认，这确实可行。不过，许多人认为这种方法只是一种修补手段，以弥补 XHR 原来缺乏对跨浏览器的支持。作为 Ajax 的核心，XHR 对象设计为允许从服务器异步地获取任意的数据。

我们讨论过，传统的 Web 应用遵循一种请求/响应模式。如果没有 Ajax，对于每个请求都会重新加载整个页面（或者利用 IFRAME，则是部分页面）。原来查看的页面会放到浏览器的历史栈中（不过，如果使用了 IFRAME，点击“后退”按钮不一定能得到用户期望的历史页面）。与此不同，用 XHR 做出的请求不会记录在浏览器的历史中。如果你的用户习惯于

使用“后退”按钮在Web应用中进行导航，就可能会产生问题。

## 1.4 可用性问题

前面谈到的都是用户的期望，除此以外，可用性也不能不提。Ajax方法相当新，还没有多少成熟的最佳实践。不过，标准Web设计原则还是适用的。随着时间推移，当越来越多的人开始尝试这种方法时，就会发现可能存在哪些限制，并建立适当的指导原则。也就是说，你应该让用户来指导你。根据在应用中使用Ajax的方式，你可能会动态地改变页面中的某些部分，习惯于整个浏览器刷新的用户可能不会注意到与以前相比有什么变化。这个问题引出了一些新的特性，如37signals所普及的黄褪技术(Yellow Fade Technique, YFT)，这个特性已经用在Ajax的招牌应用Basecamp中了。

基本说来，YFT是指“取页面中有变化的部分，并置为黄色”。假设你的应用原本没有大量使用黄色，用户就很可能会注意到这种改变。过一段时间后，再让黄色逐渐褪色，直到恢复为原来的背景色。当然，你也可以选用你喜欢的其他颜色，只要能把用户的注意力吸引到有变化的部分。

可能YTF并不适用于你的应用，你也可以选择用一种不那么张扬但仍很有用的方式来提醒用户。Gmail在右上角显示了一个闪动的红色“Loading”加载记号，提醒用户正在获取数据(见图1-7)。

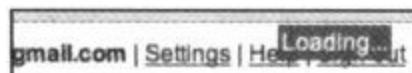


图1-7 Gmail的“Loading”记号

究竟要使用YFT还是其他类似的技术，实际上取决于你的用户。最简单的方法是让一组用户代表来进行测试。可以通过文字问卷，也可以使用基于Web的原型应用，这要看你处在设计过程的哪个阶段。但是不论如何测试，在真正采用Ajax完成复杂设计之前都应该取得一些用户反馈。

而且要从小处做起。在刚开始使用Ajax时，不应该马上就创建一个可调整列的动态门户网站，而是应该先试着处理客户端验证，逐步转向服务器端。待有所了解后，可以再尝试更动态的使用，如填写一个下拉列表，或者设置某些默认文本。

不管你要如何应用Ajax，记住别做稀奇古怪的事情。我们知道，这不算是一个学术性的建议。不过，目前这方面还没有严格的规则。先听听用户怎么说，部署之前一定要先做测试，而且要记住，如果太过古怪，用户很快就会点击“跳过本页”链接跳过你精心设计的这些部分。

要知道使用Ajax时有几个常犯的错误。我们已经讨论过，有变化时如何向用户提供可

视化的提示，不仅如此，Ajax 还会以其他方式改变标准的 Web 方法。首先，不同于 IFRAME 和隐藏框架，通过 XHR 做出请求不会修改浏览器的历史栈。在许多情况下这没有什么问题（你可能会点击后退箭头，只是要看看是不是什么都没有改变，但这么做能有几次呢？），不过，如果你的用户确实想用后退按钮，就有问题了。

其次，与其他基于浏览器的方法不同，Ajax 不会修改地址栏中显示的链接，这表明你不能轻松地为一个页面建立书签，或者向朋友发送一个链接。对于许多应用来说，可能没有这个要求，但是如果你的网站专门为人们提供行车路线之类的东西，就要针对这个问题提供一个解决方案。

有一点很重要，使用 Ajax 不要过度。记住，JavaScript 会在客户端的浏览器上运行，如果有数千行 JavaScript 代码，可能会让用户感觉速度太慢。如果脚本编写不当，就会很快失去控制，特别是当通信量增加时。

Ajax 允许你异步地完成操作，这个最大的优点同时也是它最突出的缺点。我们以前总是告诉用户，Web 应用是以一种请求/响应模式完成操作的，用户也已经接受了这种思想。但是用了 Ajax，就不再有这个限制。我们可以只修改页面的一部分，如果用户没想到这一点，他们很可能会被搞糊涂。所以，你要注意一定要让用户明白这一点，不要想当然地以为他们知道。记住，只要有疑问，就要请用户代表进行测试！

## 1.5 相关技术

当你看到本书时，可能已经了解了在应用中实现 Ajax 所需的大多数技术。重申一句，我们想强调的是，Ajax 是一个客户端技术，不论你现在使用何种服务器端技术，都能使用 Ajax，而不管使用的是 Java、.NET、Ruby、PHP 还是 CGI。实际上，在这本书中我们并不考虑服务器端，而且假设你已经很清楚如何结合日常工作中使用的服务器端技术。在后面的几百页中，我们强调的重点是客户端技术和方法，创建丰富的基于浏览器的应用时需要用到这些技术。

尽管可以使用你喜欢的任何服务器端技术，但当使用 Ajax 时还是需要转变一下思想。在一般的 Web 应用中，服务器端代码会呈现一个完整的页面，并涉及一个完整的工作单元。利用 Ajax，可能只返回一点点文本，而且只涉及一个业务应用的很小子集。对于大多数有经验的 Web 开发人员来说，理解起来没有什么问题，但是一定要记住这一点。

一些新兴的框架有助于开发人员跳出 Ajax 的一些细节。不过，你还是要对 JavaScript 有所了解。我们知道，JavaScript 用起来可能很费劲。但很遗憾，对此没有什么办法。我们大多数人都学过这么一招，把“alert”作为一种系统类型输出来帮助调试，糟糕的是，这种技术使用得还很广。不过，现在我们有了新的利器。

除了 JavaScript，你还要熟悉其他一些与表示相关的技术，如 HTML、DOM 和 CSS。你不必是这方面的专家，但是基本了解还是必要的。本书中我们会谈到你需要知道的大多数内容，没有谈到的内容可以参考网上的资源。

关于测试驱动（你肯定写过单元测试，对不对？），我们会介绍 JsUnit 和 Selenium（见图 1-8）。利用这些工具，可以先开发 JavaScript 测试，并检查浏览器兼容性测试。通常认为，下一代开发环境会对 JavaScript 提供更好的支持，另外一些与 Ajax 相关的技术会进一步减轻开发人员的负担。正在不断出现的脚本和框架也会使开发变得更为简单。

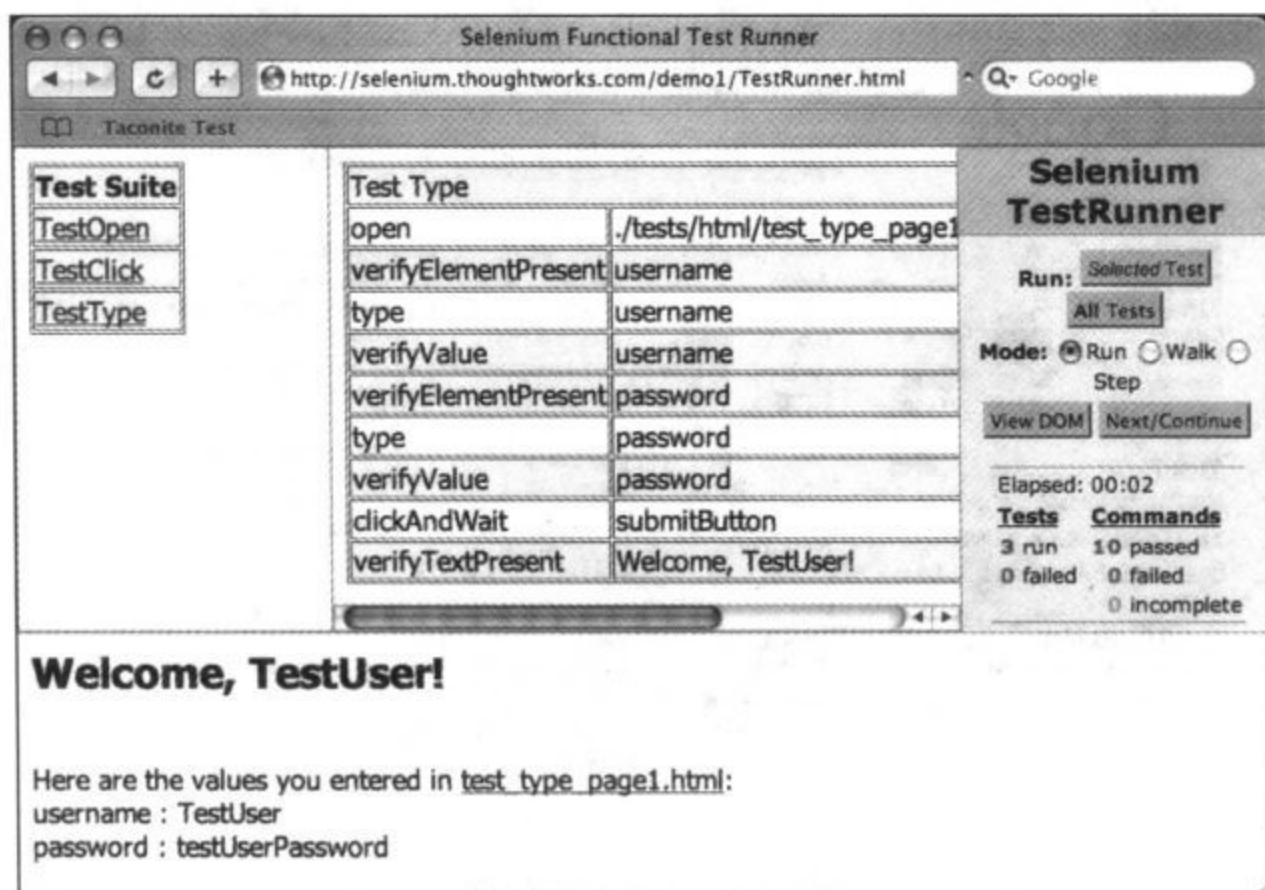


图 1-8 Selenium

## 1.6 使用场合

既然你已经对 Ajax 产生了兴趣，还要知道重要的一点，即什么时候应该使用 Ajax 技术，而什么时候不该用。首先，不要害怕在应用中尝试新的方法。我们相信，几乎每个 Web 应用都能从 Ajax 技术中获益，只不过不要矫枉过正，过于离谱就行了。从验证开始就很合适，但是不要限制你的主动性。你当然可以使用 Ajax 提交数据，但也许不能把它作为提交数据的主要方法。

其次，惟一会影响你应用 Ajax 的就是浏览器问题。如果大量用户（或者特别重要的用户）还在使用比较旧的浏览器，如 IE 5、Safari 1.2 或 Mozilla 1.0 之前的版本，Ajax 技术就不能奏效。如果这是一些很重要的用户，你就要使用针对目标用户的跨浏览器的方法，而放弃 Ajax，或者开发一个可以妥善降级的网站。浏览器支持可能不是一个重要因素，因为

Netscape Navigator 4 在市场上的份额很小。不过，还是应该查看 Web 日志，看看你的应用适用什么技术。

如前所述，验证和表单填写就非常适合采用 Ajax 实现。还可以使用 DOM 的“拖”技术建立真正动态的网站，如 Google 的个性化主页（见图 1-9）。

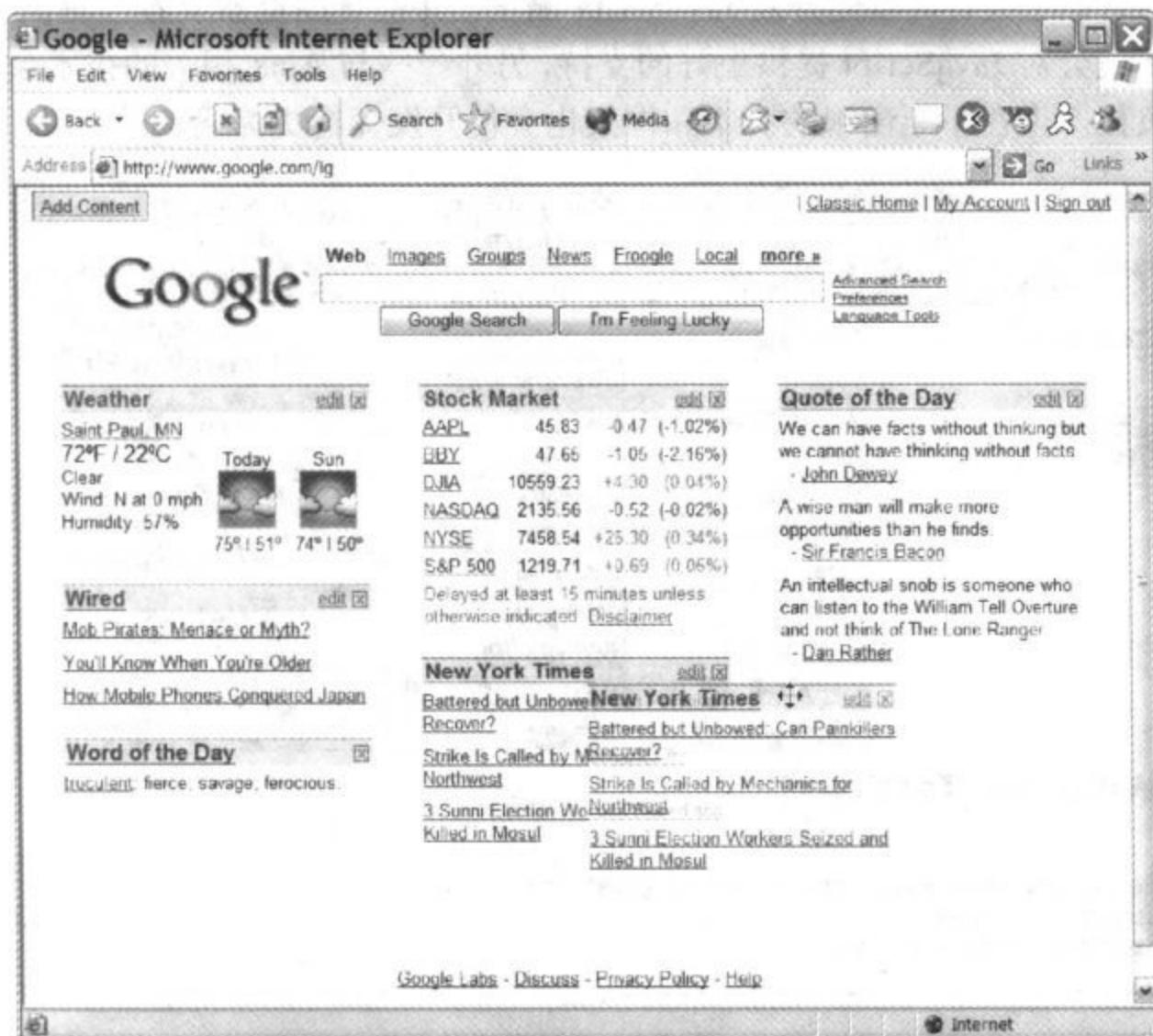


图 1-9 Google 的个性化主页

可以看到，Ajax 为 Web 应用开发提供了新的机会。你不会再因为以往的专用技术或技术折中方案而受到妨碍。利用 Ajax，胖客户与瘦客户之间的界限不再分明，真正的赢家则是你的用户。

## 1.7 设计考虑

既然对在哪里使用 Ajax 已经有所认识，下面再来谈谈应用 Ajax 的一些设计考虑。许多原则与 Web 应用的原则并无不同，不过还是有必要强调一下。要尽力减少客户和服务器之间的通信量。如果应用得当，Ajax 会使你的应用响应更快，但是如果每次用户从一个域移到另一个域时你都来回传递超量的数据，用户肯定不会满意。如果有疑问，按标准约定行事。如果大多数应用都那么做，可能你也应该那么做。如果还有问题，可以看看 Web 桌面应用的有关

标准。为此已经建立了一些模式，而且以后还会有更多的模式 ([www.ajaxpatterns.org](http://www.ajaxpatterns.org))。

在刚开始使用 Ajax 时，你的用户可能不清楚应用的工作机理。多年来我们一直在告诉用户：Web 是以某种（同步）方式工作的，而 Ajax 则增加了异步组件，可能与之背道而驰。简单地说，不要让用户觉得奇怪。当用户用跳格键离开最后一个域时，如果以前的应用（没有使用 Ajax 的应用）没有保存表单，那么使用 Ajax 之后的应用也不要保存表单。

实现 Ajax 时最重要的问题是要力求简单，完全从用户出发，要尽量“傻瓜化”。要把用户放在心上，不要去做“简历驱动的设计”<sup>1</sup>。如果只是想让新老板接受你，并因此在应用中使用 Ajax，这是不合适的；如果使用 Ajax 能让你的用户有更丰富的体验，那就义无反顾地使用 Ajax 吧。但是别忘了，你会做，并不意味着你应该做。要理智一些，先考虑你的用户才对。

我们后面还会更多地谈到安全，但是这里需要先说明一点，Ajax 有一些安全考虑。记住，可以在浏览器中查看源代码，这说明任何人都能知道你是怎么创建小部件的。建立 XHR 对象时必须包含统一资源定位符（uniform resource locators, URL），所以可能会有恶意用户修改你的网站，运行他们自己的代码。谨慎地使用 Ajax 可以降低这种风险。

## 1.8 小结

因特网最初只是为连接研究人员，使他们共享信息，时至今日，因特网已经得到了巨大的发展。因特网开始时只有简单的文本浏览器和静态页面，但是如今几乎每家公司都有一个亮丽的网站，想找到一个粗糙的网站倒是很不容易。最早谁能想得到，有一天人们能在网上共同研究新型汽车，或者购买最新的斯蒂芬·金的小说呢？

胖客户应用的开发人员都饱受部署之苦，因为要把应用部署到数以千计的用户机器上，他们急切地希望 Web 能够减轻他们的负担。多年以来，已经出现了许多 Web 应用技术，有些是专用的，有些需要高超的编程能力。尽管这些技术在用户体验方面各有千秋，但没有哪个技术能使瘦客户应用达到桌面应用的水平。不过，由于很容易部署，有更大的客户群体，而且维护开销更低，这说明尽管浏览器存在一定的局限性，但仍是许多应用的首选目标平台。

开发人员可以使用一些技巧来绕过因特网的麻烦限制。利用各种远程脚本方法和 HTML 元素，开发人员可以与服务器异步地通信，但是直到有主流浏览器对 XMLHttpRequest 对象提供了支持，真正的跨浏览器方法才有可能。Google、Yahoo 和 Amazon 等公司已经走在前面，我们终于看到基于浏览器的应用也能与胖客户应用不相上下。利用 Ajax，你可以尽享这两方面的好处：代码位于你能控制的服务器上，而且只要客户有浏览器就能访问一个能提供丰富用户体验的应用。

---

1. 有的开发人员希望完成一个复杂的设计，以便写在简历里，来证明自己的能力。——译者注



## 使用 XMLHttpRequest 对象

我们已经讨论了动态 Web 应用的发展历史，并简要介绍了 Ajax，下面再来讨论问题的关键：如何使用 XMLHttpRequest 对象。尽管与其说 Ajax 是一种技术，不如说是一种技巧，但如果对 XMLHttpRequest 的广泛支持，Google Suggest 和 Ta-da List 可能不会像我们看到的有今天这样的发展，而你可能也不会看到手上的这本书！

XMLHttpRequest 最早是在 IE 5 中以 ActiveX 组件形式实现的。由于只能在 IE 中使用，所以大多数开发人员都没有用 XMLHttpRequest，直到最近，Mozilla 1.0 和 Safari 1.2 把它采用为事实上的标准，情况才有改观。需要重点说明的是，XMLHttpRequest 并不是一个 W3C 标准，不过许多功能已经涵盖在一个新提案中：DOM Level 3 加载和保存规约（DOM Level 3 Load and Save Specification）。因为它不是标准，所以在不同浏览器上的表现也稍有区别，不过大多数方法和属性都得到了广泛的支持。当前，Firefox、Safari、Opera、Konqueror 和 Internet Explorer 都以类似的方式实现了 XMLHttpRequest 对象的行为。

前面已经说过，如果大量用户还是在使用较旧的浏览器访问网站或应用，就要三思了。第 1 章讨论过，在这种情况下，如果要使用 Ajax 技术，要么需要开发一个候选网站，要么你的应用应当能妥善地降级。大多数使用统计表明，在当前使用的浏览器中只有极少数不支持 XMLHttpRequest，所以一般情况下不会存在这个问题。不过，还是应该查看 Web 日志，确定你的用户在使用什么样的客户端来访问网站。

### 2.1 XMLHttpRequest 对象概述

在使用 XMLHttpRequest 对象发送请求和处理响应之前，必须先用 JavaScript 创建一个 XMLHttpRequest 对象。由于 XMLHttpRequest 不是一个 W3C 标准，所以可以采用多种方法使用 JavaScript 来创建 XMLHttpRequest 的实例。Internet Explorer 把 XMLHttpRequest 实现为一个 ActiveX 对象，其他浏览器（如 Firefox、Safari 和 Opera）把它实现为一个本地 JavaScript 对象。由于存在这些差别，JavaScript 代码中必须包含有关的逻辑，从而使用 ActiveX 技术

或者使用本地 JavaScript 对象技术来创建 XMLHttpRequest 的一个实例。

很多人可能还记得从前的那段日子，那时不同浏览器上的 JavaScript 和 DOM 实现简直千差万别，听了上面这段话之后，这些人可能又会不寒而栗。幸运的是，在这里为了明确该如何创建 XMLHttpRequest 对象的实例，并不需要那么详细地编写代码来区别浏览器类型。你要做的只是检查浏览器是否提供对 ActiveX 对象的支持。如果浏览器支持 ActiveX 对象，就可以使用 ActiveX 来创建 XMLHttpRequest 对象。否则，就要使用本地 JavaScript 对象技术来创建。代码清单 2-1 展示了编写跨浏览器的 JavaScript 代码来创建 XMLHttpRequest 对象实例是多么简单。

### 代码清单 2-1 创建 XMLHttpRequest 对象的一个实例

```
var xmlhttp;

function createXMLHttpRequest() {
    if (window.ActiveXObject) {
        xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
    }
    else if (window.XMLHttpRequest) {
        xmlhttp = new XMLHttpRequest();
    }
}
```

可以看到，创建 XMLHttpRequest 对象相当容易。首先，要创建一个全局作用域变量 `xmlHttp` 来保存这个对象的引用。`createXMLHttpRequest` 方法完成创建 XMLHttpRequest 实例的具体工作。这个方法中只有简单的分支逻辑（选择逻辑）来确定如何创建对象。对 `window.ActiveXObject` 的调用会返回一个对象，也可能返回 `null`，`if` 语句会把调用返回的结果看作是 `true` 或 `false`（如果返回对象则为 `true`，返回 `null` 则为 `false`），以此指示浏览器是否支持 ActiveX 控件，相应地得知浏览器是不是 Internet Explorer。如果确实是，则通过实例化 `ActiveXObject` 的一个新实例来创建 XMLHttpRequest 对象，并传入一个串指示要创建何种类型的 ActiveX 对象。在这个例子中，为构造函数提供的字符串是 `Microsoft.XMLHTTP`，这说明你想创建 XMLHttpRequest 的一个实例。

如果 `window.ActiveXObject` 调用失败（返回 `null`），JavaScript 就会转到 `else` 语句分支，确定浏览器是否把 XMLHttpRequest 实现为一个本地 JavaScript 对象。如果存在 `window.XMLHttpRequest`，就会创建 XMLHttpRequest 的一个实例。

由于 JavaScript 具有动态类型特性，而且 XMLHttpRequest 在不同浏览器上的实现是兼容的，所以可以用同样的方式访问 XMLHttpRequest 实例的属性和方法，而不论这个实例创建的方法是什么。这就大大简化了开发过程，而且在 JavaScript 中也不必编写特定于浏览器的逻辑。

## 2.2 方法和属性

表 2-1 显示了 XMLHttpRequest 对象的一些典型方法。不要担心，稍后就会详细介绍这些方法。

表 2-1 标准 XMLHttpRequest 操作

方 法	描 述
abort()	停止当前请求
getAllResponseHeaders()	把 HTTP 请求的所有响应首部作为键/值对返回
getResponseHeader("header")	返回指定首部的串值
open("method", "url")	建立对服务器的调用。method 参数可以是 GET、POST 或 PUT。 url 参数可以是相对 URL 或绝对 URL。这个方法还包括 3 个可选的参数
send(content)	向服务器发送请求
setRequestHeader("header", "value")	把指定首部设置为所提供的值。在设置任何首部之前必须先调用 open()

下面来更详细地讨论这些方法。

`void open(string method, string url, boolean asynch, string username, string password)`：这个方法会建立对服务器的调用。这是初始化一个请求的纯脚本方法。它有两个必要的参数，还有 3 个可选参数。要提供调用的特定方法（GET、POST 或 PUT），还要提供所调用资源的 URL。另外还可以传递一个 Boolean 值，指示这个调用是异步的还是同步的。默认值为 `true`，表示请求本质上是异步的。如果这个参数为 `false`，处理就会等待，直到从服务器返回响应为止。由于异步调用是使用 Ajax 的主要优势之一，所以倘若将这个参数设置为 `false`，从某种程度上讲与使用 XMLHttpRequest 对象的初衷不太相符。不过，前面已经说过，在某些情况下这个参数设置为 `false` 也是有用的，比如在持久存储页面之前可以先验证用户的输入。最后两个参数不说自明，允许你指定一个特定的用户名和密码。

`void send(content)`：这个方法具体向服务器发出请求。如果请求声明为异步的，这个方法就会立即返回，否则它会等待直到接收到响应为止。可选参数可以是 DOM 对象的实例、输入流，或者串。传入这个方法的内容会作为请求体的一部分发送。

`void setRequestHeader(string header, string value)`：这个方法为 HTTP 请求中一个给定的首部设置值。它有两个参数，第一个串表示要设置的首部，第二个串表示要在首部中放置的值。需要说明，这个方法必须在调用 `open()` 之后才能调用。

在所有这些方法中，最有可能用到的就是 `open()` 和 `send()`。XMLHttpRequest 对象还有许多属性，在设计 Ajax 交互时这些属性非常有用。

`void abort()`：顾名思义，这个方法就是要停止请求。

`string getAllResponseHeaders()`: 这个方法的核心功能对 Web 应用开发人员应该很熟悉了，它返回一个串，其中包含 HTTP 请求的所有响应首部，首部包括 Content-Length、Date 和 URI。

`string getResponseHeader(string header)`: 这个方法与 `getAllResponseHeaders()` 是对应的，不过它有一个参数表示你希望得到的指定首部值，并且把这个值作为串返回。

除了这些标准方法，`XMLHttpRequest` 对象还提供了许多属性，如表 2-2 所示。处理 `XMLHttpRequest` 时可以大量使用这些属性。

表 2-2 标准 XMLHttpRequest 属性

属性	描述
<code>onreadystatechange</code>	每个状态改变时都会触发这个事件处理器，通常会调用一个 JavaScript 函数
<code>readyState</code>	请求的状态。有 5 个可取值：0 = 未初始化，1 = 正在加载，2 = 已加载，3 = 交互中，4 = 完成
<code>responseText</code>	服务器的响应，表示为一个串
<code>responseXML</code>	服务器的响应，表示为 XML。这个对象可以解析为一个 DOM 对象
<code>status</code>	服务器的 HTTP 状态码（200 对应 OK，404 对应 Not Found（未找到），等等）
<code>statusText</code>	HTTP 状态码的相应文本（OK 或 Not Found（未找到）等等）

## 2.3 交互示例

看到这里，你可能想知道典型的 Ajax 交互是什么样。图 2-1 显示了 Ajax 应用中标准的交互模式。

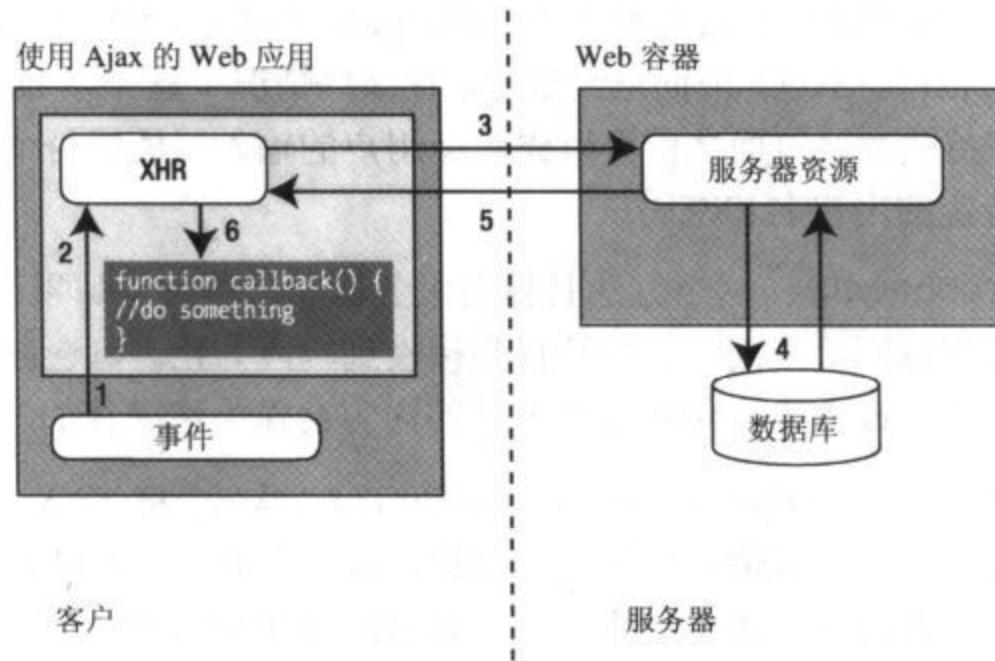


图 2-1 标准 Ajax 交互

不同于标准 Web 客户中所用的标准请求/响应方法，Ajax 应用的做法稍有差别。

- 一个客户端事件触发一个 Ajax 事件。从简单的 onchange 事件到某个特定的用户动作，很多这样的事件都可以触发 Ajax 事件。可以有如下的代码：

```
<input type="text" id="email" name="email" onblur="validateEmail()">
```

- 创建 XMLHttpRequest 对象的一个实例。使用 open() 方法建立调用，并设置 URL 以及所希望的 HTTP 方法（通常是 GET 或 POST）。请求实际上通过一个 send() 方法调用触发。可能的代码如下所示：

```
var xmlhttp;
function validateEmail() {
    var email = document.getElementById("email");
    var url = "validate?email=" + escape(email.value);
    if (window.ActiveXObject) {
        xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
    }
    else if (window.XMLHttpRequest) {
        xmlhttp = new XMLHttpRequest();
    }
    xmlhttp.open("GET", url);
    xmlhttp.onreadystatechange = callback;
    xmlhttp.send(null);
}
```

- 向服务器做出请求。可能调用 servlet、CGI 脚本，或者任何服务器端技术。
- 服务器可以做你想做的事情，包括访问数据库，甚至访问另一个系统。
- 请求返回到浏览器。Content-Type 设置为 text/xml——XMLHttpRequest 对象只能处理 text/html 类型的结果。在另外一些更复杂示例中，响应可能涉及更广，还包括 JavaScript、DOM 管理以及其他相关的技术。需要说明，你还需要设置另外一些首部，使浏览器不会在本地缓存结果。为此可以使用下面的代码：

```
response.setHeader("Cache-Control", "no-cache");
response.setHeader("Pragma", "no-cache");1
```

- 在这个示例中，XMLHttpRequest 对象配置为处理返回时要调用 callback() 函数。这个函数会检查 XMLHttpRequest 对象的 readyState 属性，然后查看服务器返回的状态码。如果一切正常，callback() 函数就会在客户端上做些有意思的工作。以下就是一个典型的回调方法：

---

1. Pragma 和 Cache-Control……它们不是做相同的事吗？是，它们是做相同的事，不过定义 Pragma 是为了保证向后兼容。

```
function callback() {
    if (xmlHttp.readyState == 4) {
        if (xmlHttp.status == 200) {
            //do something interesting here
        }
    }
}
```

可以看到，这与正常的请求/响应模式有所不同，但对 Web 开发人员来说，并不是完全陌生的。显然，在创建和建立 XMLHttpRequest 对象时还可以做些事情，另外当“回调”函数完成了状态检查之后也可以有所作为。一般地，你会把这些标准调用包装在一个库中，以便在整个应用中使用，或者可以使用 Web 上提供的库。这个领域还很新，但是在开源社区中已经如火如荼地展开了大量的工作。

通常，Web 上提供的各种框架和工具包负责基本的连接和浏览器抽象，有些还增加了用户界面组件。有一些纯粹基于客户，还有一些需要在服务器上工作。这些框架中的很多只是刚刚开始开发，或者还处于发布的早期阶段，随着新的库和新的版本的定期出现，情况还在不断发生变化。这个领域正在日渐成熟，最具优势的将脱颖而出。一些比较成熟的库包括 libXmlRequest、RSLite、sarissa、JavaScript 对象注解（JavaScript Object Notation，JSON）、JSRS、直接 Web 远程通信（Direct Web Remoting，DWR）和 Ruby on Rails。这个领域日新月异，所以应当适当地配置你的 RSS 收集器，及时收集有关 Ajax 的所有网站上的信息！

## 2.4 GET与POST

你可能想了解 GET 和 POST 之间有什么区别，并想知道什么时候使用它们。从理论上讲，如果请求是幂等的就可以使用 GET，所谓幂等是指多个请求返回相同的结果。实际上，相应的服务器方法可能会以某种方式修改状态，所以一般情况下这是不成立的。这只是一个标准。更实际的区别在于净荷的大小，在许多情况下，浏览器和服务器会限制 URL 的长度 URL 用于向服务器发送数据。一般来讲，可以使用 GET 从服务器获取数据；换句话说，要避免使用 GET 调用改变服务器上的状态。

一般地，当改变服务器上的状态时应当使用 POST 方法。不同于 GET，需要设置 XMLHttpRequest 对象的 Content-Type 首部，如下所示：

```
xmlHttp.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
```

与 GET 不同，POST 不会限制发送给服务器的净荷的大小，而且 POST 请求不能保证是幂等的。

你做的大多数请求可能都是 GET 请求，不过，如果需要，也完全可以使用 POST。

## 2.5 远程脚本

我们已经介绍了 Ajax，下面来简单谈谈远程脚本。你可能会想：“Ajax 有什么大不了的？我早就用 IFRAME 做过同样的事情。”实际上，我们自己也曾用过这种方法。这在以前一般称为远程脚本（remote scripting），很多人认为这只不过是一种修修补补。不过，这确实提供了一种能避免页面刷新的机制。

### 2.5.1 远程脚本概述

基本说来，远程脚本是一种远程过程调用类型。你可以像正常的 Web 应用一样与服务器交互，但是不用刷新整个页面。与 Ajax 类似，你可以调用任何服务器端技术来接收请求、处理请求并返回一个有意义的结果。正如在服务器端有很多选择，客户端同样有许多实现远程脚本的选择。你可以在应用中嵌入 Flash 动画、Java applet，或者 ActiveX 组件，甚至可以使用 XML-RPC，但是这种方法过于复杂，因此除非你使用这种技术很有经验，否则这种方法不太合适。实现远程脚本的通常做法包括将脚本与一个 IFRAME（隐藏或不隐藏）结合，以及由服务器返回 JavaScript，然后再在浏览器中运行这个 JavaScript。

Microsoft 提供了自己的远程脚本解决方案，并聪明地称之为 Microsoft 远程脚本（Microsoft Remote Scripting, MSRS）。采用这种方法，可以像调用本地脚本一样调用服务器脚本。页面中嵌入 Java applet，以便与服务器通信，.asp 页面用于放置服务器端脚本，并用.htm 文件管理客户端的布局摆放。在 Netscape 和 IE 4.0 及更高版本中都可以使用 Microsoft 的这种解决方案，可以同步调用，也可以异步调用。不过，这种解决方案需要 Java，这意味着可能还需要附加的安装例程，而且还需要 Internet Information Services (IIS)，因此会限制服务器端的选择。

Brent Ashley 为远程脚本创建了两个免费的跨平台库。JSRS 是一个客户端 JavaScript 库，它充分利用 DHTML 向服务器做远程调用。相当多的操作系统和浏览器上都能使用 JSRS。如果采用一些常用的、流行的服务器端实现（如 PHP、Python 和 Perl CGI），JSRS 一般都能在网站上安装并运行。Ashley 免费提供了 JSRS，而且还可以从他的网站([www.ashleyit.com/rs/main.htm](http://www.ashleyit.com/rs/main.htm)) 上得到源代码。

如果你觉得 JSRS 太过笨重，Ashley 还创建了 RSLite，这个库使用了 cookie。RSLite 仅限于少量数据和单一调用，不过大多数浏览器都能提供支持。

### 2.5.2 远程脚本的示例

为了进行比较，这里向你展示如何使用 IFRAME 来实现类似 Ajax 的技术。这非常简单，而且过去我们就用过这种方法（在 XMLHttpRequest 问世之前）。这个示例并没有真正调用

服务器，只是想让你对如何使用 IFRAME 实现远程脚本有所认识。

这个示例包括两个文件：iframe.html（见代码清单 2-2）和 server.html（见代码清单 2-3）。server.html 模拟了本应从服务器返回的响应。

### 代码清单 2-2 iframe.html 文件

```
<html>
  <head>
    <title>Example of remote scripting in an IFRAME</title>
  </head>
  <script type="text/javascript">
    function handleResponse() {
      alert('this function is called from server.html');
    }
  </script>
  <body>
    <h1>Remote Scripting with an IFRAME</h1>

    <iframe id="beforexhr"
name="beforexhr"
style="width:0px; height:0px; border: 0px"
src="blank.html"></iframe>

    <a href="server.html" target="beforexhr">call the server</a>

  </body>
</html>
```

### 代码清单 2-3 server.html 文件

```
<html>
  <head>
    <title>the server</title>
  </head>
  <script type="text/javascript">
    window.parent.handleResponse();
  </script>
  <body>
  </body>
</html>
```

图 2-2 显示了最初的页面。运行这个代码生成的结果如图 2-3 所示。

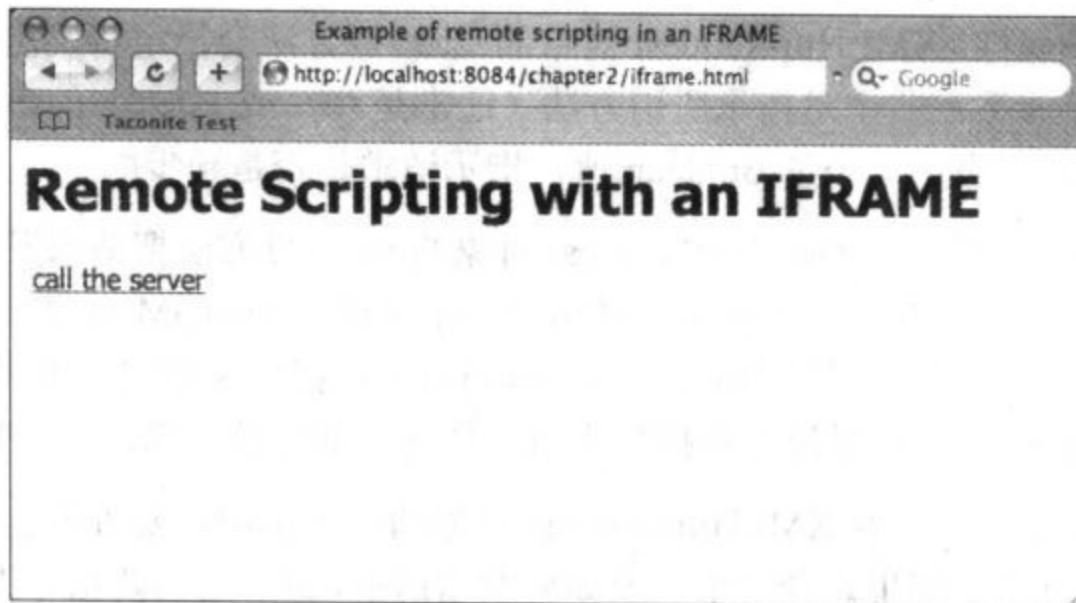


图 2-2 最初的页面

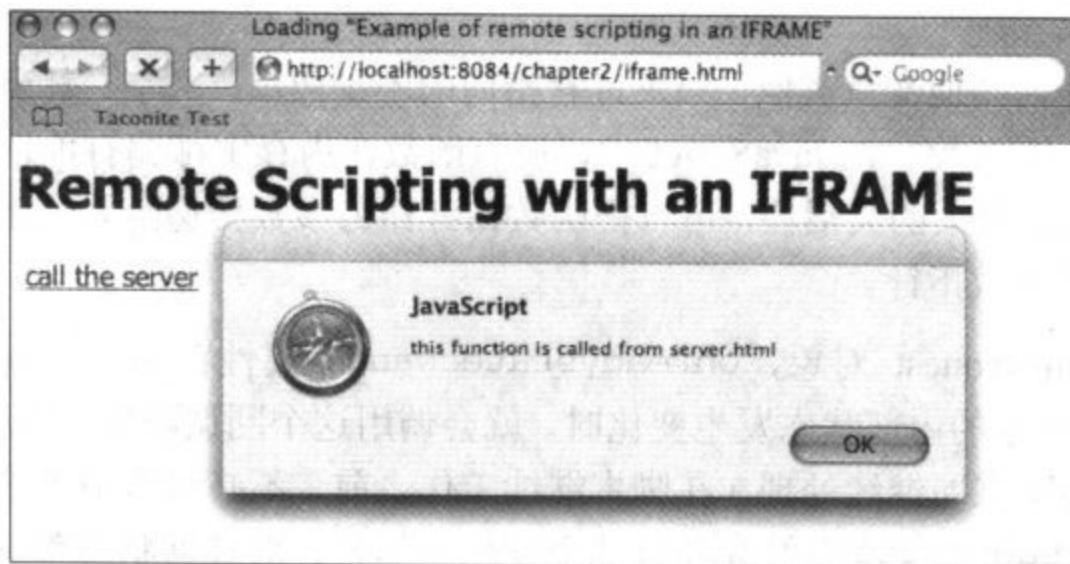


图 2-3 调用“服务器”之后的页面

## 2.6 如何发送简单请求

现在已经准备开始使用 XMLHttpRequest 对象了。我们刚刚讨论了如何创建这个对象，下面来看如何向服务器发送请求，以及如何处理服务器的响应。

最简单的请求是，不以查询参数或提交表单数据的形式向服务器发送任何信息。在实际中，往往都希望向服务器发送一些信息。

使用 XMLHttpRequest 对象发送请求的基本步骤如下：

1. 为得到 XMLHttpRequest 对象实例的一个引用，可以创建一个新的实例，也可以访问包含有 XMLHttpRequest 实例的一个变量。
2. 告诉 XMLHttpRequest 对象，哪个函数会处理 XMLHttpRequest 对象状态的改变，为此要把对象的 `onreadystatechange` 属性设置为指向 JavaScript 函数的指针。

3. 指定请求的属性。XMLHttpRequest 对象的 `open()` 方法会指定将发出的请求。`open()` 方法取 3 个参数：一个是指示所用方法（通常是 GET 或 POST）的串；一个是表示目标资源 URL 的串；一个是 Boolean 值，指示请求是否是异步的。
4. 将请求发送给服务器。XMLHttpRequest 对象的 `send()` 方法把请求发送到指定的目标资源。`send()` 方法接受一个参数，通常是一个串或一个 DOM 对象。这个参数作为请求体的一部分发送到目标 URL。当向 `send()` 方法提供参数时，要确保 `open()` 中指定的方法是 POST。如果没有数据作为请求体的一部分被发送，则使用 `null`。

这些步骤很直观：你需要 XMLHttpRequest 对象的一个实例，要告诉它如果状态有变化该怎么做，还要告诉它向哪里发送请求以及如何发送请求，最后还需要指导 XMLHttpRequest 发送请求。不过，除非你对 C 或 C++ 很了解，否则可能不明白函数指针（function pointer）是什么意思。

函数指针与任何其他变量类似，只不过它指向的不是像串、数字、甚至对象实例之类的数据，而是指向一个函数。在 JavaScript 中，所有函数在内存中都编有地址，可以使用函数名引用。这就提供了很大的灵活性，可以把函数指针作为参数传递给其他函数，或者在一个对象的属性中存储函数指针。

对于 XMLHttpRequest 对象，`onreadystatechange` 属性存储了回调函数的指针。当 XMLHttpRequest 对象的内部状态发生变化时，就会调用这个回调函数。当进行了异步调用，请求就会发出，脚本立即继续处理（在脚本继续工作之前，不必等待请求结束）。一旦发出了请求，对象的 `readyState` 属性会经过几个变化。尽管针对任何状态都可以做一些处理，不过你最感兴趣的状态可能是服务器响应结束时的状态。通过设置回调函数，就可以有效地告诉 XMLHttpRequest 对象：“只要响应到来，就调用这个函数来处理响应。”

### 2.6.1 简单请求的示例

第一个示例很简单。这是一个很小的 HTML 页面，只有一个按钮。点击这个按钮会初始化一个发至服务器的异步请求。服务器将发回一个简单的静态文本文件作为响应。在处理这个响应时，会在一个警告窗口中显示该静态文本文件的内容。代码清单 2-4 显示了这个 HTML 页面和相关的 JavaScript。

**代码清单 2-4 simpleRequest.html 页面**

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Simple XMLHttpRequest</title>
```

```
<script type="text/javascript">
var xmlhttp;

function createXMLHttpRequest() {
    if (window.ActiveXObject) {
        xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
    }
    else if (window.XMLHttpRequest) {
        xmlhttp = new XMLHttpRequest();
    }
}

function startRequest() {
    createXMLHttpRequest();
    xmlhttp.onreadystatechange = handleStateChange;
    xmlhttp.open("GET", "simpleResponse.xml", true);
    xmlhttp.send(null);
}

function handleStateChange() {
    if(xmlhttp.readyState == 4) {
        if(xmlhttp.status == 200) {
            alert("The server replied with: " + xmlhttp.responseText);
        }
    }
}
</script>
</head>

<body>
    <form action="#">
        <input type="button" value="Start Basic Asynchronous Request"
               onclick="startRequest();"/>
    </form>
</body>
</html>
```

服务器的响应文件 simpleResponse.xml 只有一行文本。点击 HTML 页面上的按钮会生成一个警告框，其中显示 simpleResponse.xml 文件的内容。在图 2-4 中可以看到分别在 Internet Explorer 和 Firefox 中显示的包含服务器响应的相同警告框。

对服务器的请求是异步发送的，因此浏览器可以继续响应用户输入，同时在后台等待服务器的响应。如果选择同步操作，而且倘若服务器的响应要花几秒才能到达，浏览器就会表现得很迟钝，在等待期间不能响应用户的输入。这样一来，浏览器好像被冻住一样，无法响应用户输入，而异步做法可以避免这种情况，从而让最终用户有更好的体验。尽管这种改善很细微，但确实很有意义。这样用户就能继续工作，而且服务器会在后台处理先前的请求。

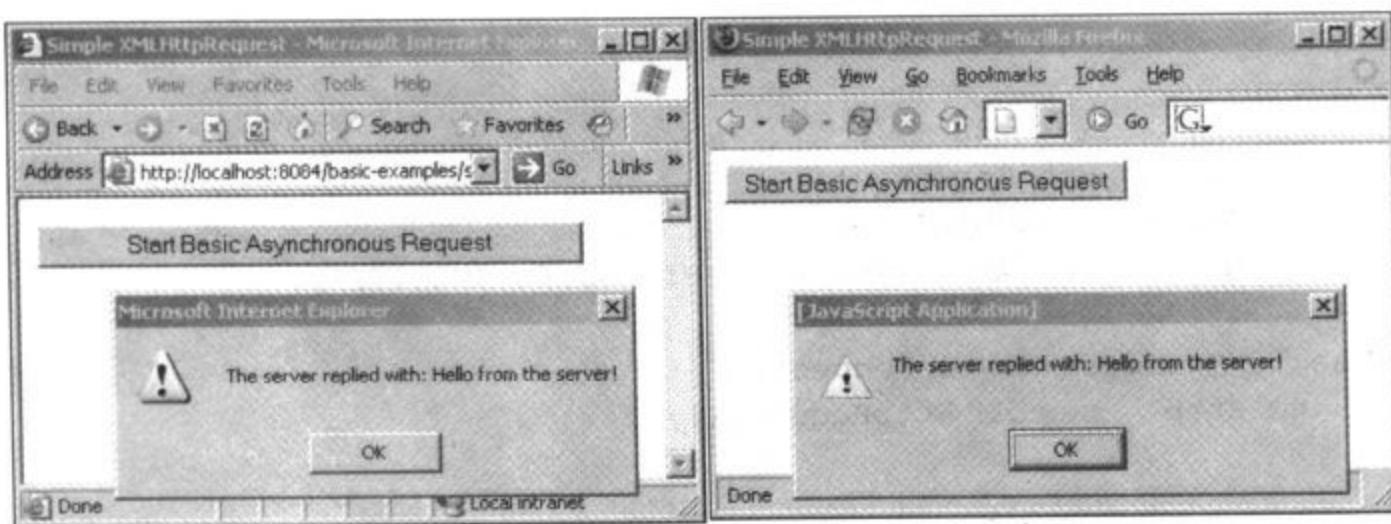


图 2-4 第一个简单的异步请求

与服务器通信而不打断用户的使用流程，这种能力使开发人员采用多种技术改善用户体验成为可能。例如，假设有一个验证用户输入的应用。用户在输入表单上填写各个字段时，浏览器可以定期地向服务器发送表单值来进行验证，此时并不打断用户，他还可以继续填写余下的表单字段。如果某个验证规则失败，在表单真正发送到服务器进行处理之前，用户就会立即得到通知，这就能大大节省用户的时间，也能减轻服务器上的负载，因为不必在表单提交不成功时完全重建表单的内容。

## 2.6.2 关于安全

如果讨论基于浏览器的技术时没有提到安全，那么讨论就是不完整的。`XMLHttpRequest` 对象要受制于浏览器的安全“沙箱”。`XMLHttpRequest` 对象请求的所有资源都必须与调用脚本在同一个域内。这个安全限制使得 `XMLHttpRequest` 对象不能请求脚本所在域之外的资源。

这个安全限制的强度因浏览器而异（见图 2-5）。IE 会显示一个警告，指出可能存在一个潜在的安全风险，但是用户可以选择是否继续发出请求。Firefox 则会断然停止请求，并在 JavaScript 控制台显示一个错误消息。

Firefox 确实提供了一些 JavaScript 技巧，使得 `XMLHttpRequest` 也可以请求外部 URL 的资源。不过，由于这些技术针对特定的浏览器，所以最好不要用，而且要避免使用 `XMLHttpRequest` 访问外部 URL。

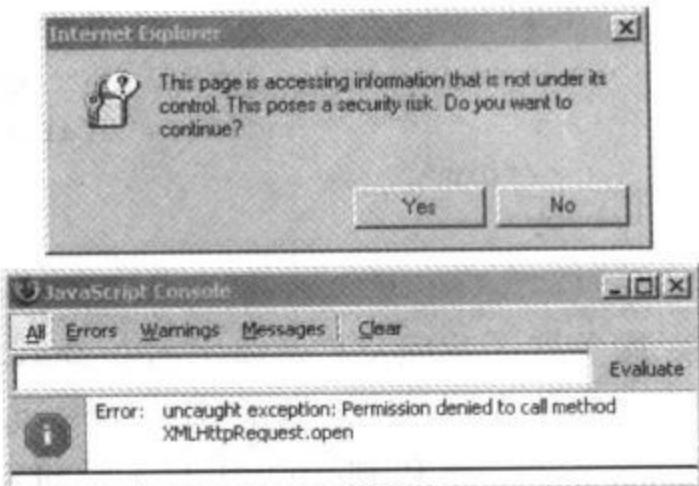


图 2-5 对于潜在的安全威胁，Internet Explorer 和 Firefox 有不同的响应

## 2.7 DOM Level 3 加载和保存规约

到目前为止，我们讨论的解决方案都不是标准。尽管 XMLHttpRequest 得到了广泛支持，但是你已经看到了，创建 XMLHttpRequest 对象的过程会随浏览器不同而有所差异。许多人错误地认为 Ajax 得到了 W3C 的支持，但实际上并非如此。W3C 在一个新标准中解决了这一问题以及其他缺点，这个标准的名字相当长：DOM Level 3 加载和保存规约。这个规约的设计目的是以一种独立于平台和语言的方式，用 XML 内容修改 DOM 文档的内容。2004 年 4 月提出了 1.0 版本，但到目前为止，还没有浏览器实现这个规约。

什么时候加载和保存规约能取代 Ajax？谁也不知道。想想看有多少浏览器没有完全支持现有的标准，所以这很难说，但是随着越来越多的网站和应用利用了 Ajax 技术，可能以后的版本会得到支持。不过，较早的 DOM 版本就花了很长时间才得到采纳，所以你得耐心一点。在一次访谈中，DOM Activity 主席 Philippe Le Hégaret 称，需要花“相当长的时间”才能得到广泛采纳。DOM Level 3 也得到了一些支持，Opera 的 XMLHttpRequest 实现就基于 DOM Level 3，而且 Java XML 处理 API（Java API for XML Processing，JAXP）1.3 版本也支持 DOM Level 3。不过，从出现了相应的 W3C 规约这一点来看，起码可以表明 Ajax 技术的重要性。

从 1997 年 8 月起，人们就一直在为解决浏览器之间的不兼容而努力，加载和保存规约则达到了极致。你可能注意到，标题里写的是“Level 3”，那么 Level 1 和 Level 2 呢？Level 1 在 1998 年 10 月完成，为我们带来了 HTML 4.0 和 XML 1.0。如今，Level 1 已经得到了广泛支持。2000 年 11 月，Level 2 完成，不过它被采纳得比较慢。CSS 就是 Level 2 的一部分。

开发人员能从加载和保存规约得到些什么？在理想情况下，它能解决我们目前遇到的许多跨浏览器问题。尽管 Ajax 很简单，但是你应该记得，仅仅是为了创建 XMLHttpRequest 对象的一个实例，就需要检查浏览器的类型。真正的 W3C 规约可以减少这种编写代码的工作。基本说来，加载和保存规约会为 Web 开发人员提供一个公共的 API，可以以一种独立于平台和语言的方式来访问和修改 DOM。换句话说，不论你的平台是 Windows 还是 Linux，也不论你用 VBScript 开发还是用 JavaScript 开发，都没有关系。还可以把 DOM 树保存为一个 XML 文档，或者将一个 XML 文档加载到 DOM。另外，规约还提供了对 XML 1.1、XML Schema 1.0 和 SOAP 1.2 的支持。这个规约很可能得到开发人员的广泛使用。

## 2.8 DOM

我们一直在说 DOM，如果你没有做过太多客户端的工作，可能不知道什么是 DOM。DOM 是一个 W3C 规约，可以以一种独立于平台和语言的方式访问和修改一个文档的内容和结构。换句话说，这是表示和处理一个 HTML 或 XML 文档的常用方法。

有一点很重要，DOM 的设计是以对象管理组织（OMG）的规约为基础的，因此可以用于任何编程语言。最初人们把它认为是一种让 JavaScript 在浏览器间可移植的方法，不过 DOM 的应用已经远远超出这个范围。

DOM 实际上是以面向对象方式描述的对象模型。DOM 定义了表示和修改文档所需的对象、这些对象的行为和属性以及这些对象之间的关系。可以把 DOM 认为是页面上数据和结构的一个树形表示，不过页面当然可能并不是以这种树的方式具体实现。假设有一个 Web 页面，如代码清单 2-5 所示。

#### 代码清单 2-5 简单的表格

```
<table>
  <tbody>
    <tr>
      <td>Foo</td>
      <td>Bar</td>
    </tr>
  </tbody>
</table>
```

可以画出这个简单表格的 DOM，如图 2-6 所示。

DOM 规约好就好在它提供了一种与文档交互的标准方法。如果没有 DOM，Ajax 最有意思方面也许根本就没有存在的可能。由于 DOM 不仅允许遍历 DOM 树，还可以编辑内容，因此可以建立极为动态的页面。

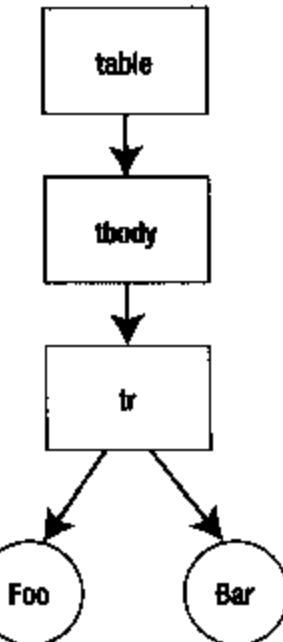


图 2-6 简单的 DOM

## 2.9 小结

尽管 Ajax 风格的技术已经用了许多年，但直到最近 XMLHttpRequest 对象才得到现代浏览器的采纳，而这也为开发丰富的 Web 应用开启了一个新的时代。在本章中，我们讨论了 Ajax 核心（即 XMLHttpRequest 对象）的相关基础知识。我们了解了 XMLHttpRequest 对象的方法和属性，而且展示了使用 XMLHttpRequest 对象的简单示例。可以看到，这个对象相当简单，无需你考虑其中很多的复杂性。适当地使用 JavaScript，再加上基本的 DOM 管理，Ajax 可以提供高度的交互性，而这在此前的 Web 上是做不到的。

第 1 章曾提到，利用 XMLHttpRequest，你不必将整个页面完全刷新，也不限于只能与服务器进行同步会话。在后面的几章中，我们会介绍如何将你已经掌握的服务器端技术与 XMLHttpRequest 的独特功能相结合，来提供高度交互性的 Web 应用。

# 与服务器通信：发送请求和处理响应

**好**戏正式开始！你刚刚了解了 XMLHttpRequest 对象，现在这些新知识就要派上用场了。我们将展示一些简单的例子，说明 XMLHttpRequest 对象怎样向服务器发送请求，以及怎样用 JavaScript 处理服务器响应。

---

**注意** 本章中的例子没有使用动态服务器来处理响应以及提供实时响应。这里的例子只是使用简单的文本文件来模拟服务器的响应。这样一来，可以降低复杂性，使你更专注于浏览器端发生了什么。

---

## 3.1 处理服务器响应

XMLHttpRequest 对象提供了两个可以用来访问服务器响应的属性。第一个属性 `responseText` 将响应提供为一个串，第二个属性 `responseXML` 将响应提供为一个 XML 对象。一些简单的用例就很适合按简单文本来获取响应，如将响应显示在警告框中，或者响应只是指示成功还是失败的词。

第 2 章中的例子就使用了 `responseText` 属性来访问服务器响应，并将响应显示在警告框中。

### 3.1.1 使用 `innerHTML` 属性创建动态内容

如果将服务器响应作为简单文本来访问，则灵活性欠佳。简单文本没有结构，很难用 JavaScript 进行逻辑性的表述，而且要想动态地生成页面内容也很困难。

如果结合使用 HTML 元素的 `innerHTML` 属性，`responseText` 属性就会变得非常有用。`innerHTML` 属性是一个非标准的属性，最早在 IE 中实现，后来也为其他许多流行的浏览器所采用。这是一个简单的串，表示一组开始标记和结束标记之间的内容。

通过结合使用 `responseText` 和 `innerHTML`, 服务器就能“生产”或生成 HTML 内容, 由浏览器使用 `innerHTML` 属性来“消费”或处理。下面的例子展示了一个搜索功能, 这是使用 XMLHttpRequest 对象、其 `responseText` 属性和 HTML 元素的 `innerHTML` 属性实现的。点击 `search` (搜索) 按钮将在服务器上启动“搜索”, 服务器将生成一个结果表作为响应。浏览器处理响应时将 `div` 元素的 `innerHTML` 属性设置为 XMLHttpRequest 对象的 `responseText` 属性值。图 3-1 显示了点击 `search` 按钮而且在窗口内容中增加了结果表之后的浏览器窗口。

第 2 章的例子只是将服务器响应显示在警告框中, 这个例子的代码与它很相似。具体步骤如下:

1. 点击 `search` 按钮, 调用 `startRequest` 函数, 它先调用 `createXMLHttpRequest` 函数来初始化 XMLHttpRequest 对象的一个新实例;
2. `startRequest` 函数将回调函数设置为 `handleStateChange` 函数;
3. `startRequest` 函数使用 `open()` 方法来设置请求方法 (GET) 及请求目标, 并且设置为异步地完成请求;
4. 使用 XMLHttpRequest 对象的 `send()` 方法发送请求;
5. XMLHttpRequest 对象的内部状态每次有变化时, 都会调用 `handleStateChange` 函数。一旦接收到响应 (如果 `readyState` 属性的值为 4), `div` 元素的 `innerHTML` 属性就将使用 XMLHttpRequest 对象的 `responseText` 属性设置。

代码清单 3-1 显示了 `innerHTML.html`。代码清单 3-2 显示了 `innerHTML.xml`, 表示搜索生成的内容。

### 代码清单 3-1 `innerHTML.html`

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Using responseText with innerHTML</title>

<script type="text/javascript">
var xmlhttp;
```

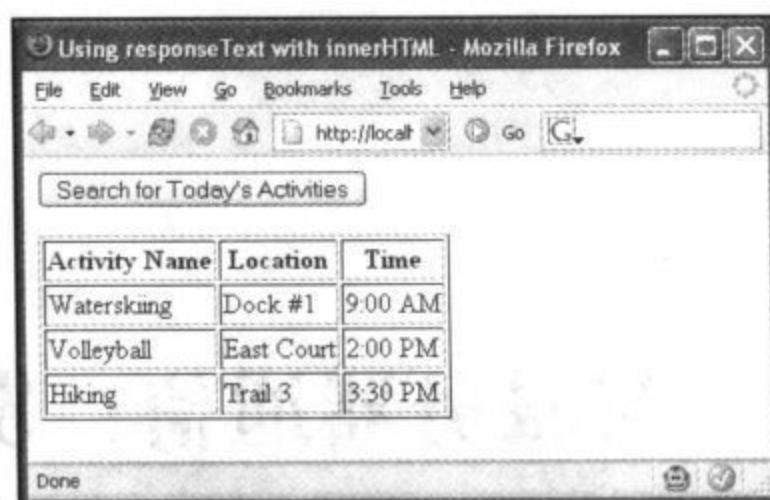


图 3-1 浏览器窗口显示了使用 XMLHttpRequest 获取并使用 `innerHTML` 处理的搜索结果

```
function createXMLHttpRequest() {
    if (window.ActiveXObject) {
        xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
    }
    else if (window.XMLHttpRequest) {
        xmlhttp = new XMLHttpRequest();
    }
}

function startRequest() {
    createXMLHttpRequest();
    xmlhttp.onreadystatechange = handleStateChange;
    xmlhttp.open("GET", "innerHTML.xml", true);
    xmlhttp.send(null);
}

function handleStateChange() {
    if(xmlhttp.readyState == 4) {
        if(xmlhttp.status == 200) {
            document.getElementById("results").innerHTML = xmlhttp.responseText;
        }
    }
}
</script>
</head>

<body>
    <form action="#">
        <input type="button" value="Search for Today's Activities"
               onclick="startRequest();"/>
    </form>
    <div id="results"></div>
</body>
</html>
```

### 代码清单 3-2 innerHTML.xml

```
<table border="1">
    <tbody>
        <tr>
            <th>Activity Name</th>
            <th>Location</th>
            <th>Time</th>
        </tr>
        <tr>
            <td>Waterskiing</td>
            <td>Dock #1</td>
            <td>9:00 AM</td>
        </tr>
```

```
<tr>
    <td>Volleyball</td>
    <td>East Court</td>
    <td>2:00 PM</td>
</tr>
<tr>
    <td>Hiking</td>
    <td>Trail 3</td>
    <td>3:30 PM</td>
</tr>
</tbody>
</table>
```

使用 `responseText` 和 `innerHTML` 可以大大简化向页面增加动态内容的工作。遗憾的是，这种方法存在一些缺陷。前面已经提到，`innerHTML` 属性不是 HTML 元素的标准属性，所以与标准兼容的浏览器不一定提供这个属性的实现。不过，当前大多数浏览器都支持 `innerHTML` 属性。可笑的是，IE 是率先使用 `innerHTML` 的浏览器，但它的 `innerHTML` 实现反而最受限制。如今许多浏览器都将 `innerHTML` 属性作为所有 HTML 元素的读/写属性。与此不同，IE 则有所限制，在表和表行之类的 HTML 元素上 `innerHTML` 属性仅仅是只读属性，从一定程度上讲，这就限制了它的用途。

### 3.1.2 将响应解析为 XML

你已经了解到，服务器不一定按 XML 格式发送响应。只要 Content-Type 响应首部正确地设置为 `text/plain`（如果是 XML，Content-Type 响应首部则是 `text/xml`），将响应作为简单文本发送是完全可以的。复杂的数据结构就很适合以 XML 格式发送。对于导航 XML 文档以及修改 XML 文档的结构和内容，当前浏览器已经提供了很好的支持。

浏览器到底怎么处理服务器返回的 XML 呢？当前浏览器把 XML 看作是遵循 W3C DOM 的 XML 文档。W3C DOM 指定了一组很丰富的 API，可用于搜索和处理 XML 文档。DOM 兼容的浏览器必须实现这些 API，而且不允许有自定义的行为，这样就能尽可能地改善脚本在不同浏览器之间的可移植性。

#### W3C DOM

W3C DOM 到底是什么？W3C 主页提供了清晰的定义：

文档对象模型（DOM）是与平台和语言无关的接口，允许程序和脚本动态地访问和更新文档的内容、结构和样式。文档可以进一步处理，处理的结果可以放回到所提供的页面中。

不仅如此，W3C 还解释了为什么要定义标准的 DOM。W3C 从其成员处收到了大量请求，这些请求都是关于将 XML 和 HTML 文档的对象模型提供给脚本所要采用的方法。提案

并没有提出任何新的标记或样式表技术，而只是力图确保这些可互操作而且与脚本语言无关的解决方案能得到共识，并为开发社区所采纳。简单地说，W3C DOM 标准的目的是尽量避免 20 世纪 90 年代末的脚本恶梦，那时相互竞争的浏览器都有自己专用的对象模型，而且通常都是不兼容的，这就使得实现跨平台的脚本极其困难。

### W3C DOM 和 JavaScript

W3C DOM 和 JavaScript 很容易混淆不清。DOM 是面向 HTML 和 XML 文档的 API，为文档提供了结构化表示，并定义了如何通过脚本来访问文档结构。JavaScript 则是用于访问和处理 DOM 的语言。如果没有 DOM，JavaScript 根本没有 Web 页面和构成页面元素的概念。文档中的每个元素都是 DOM 的一部分，这就使得 JavaScript 可以访问元素的属性和方法。

DOM 独立于具体的编程语言，通常通过 JavaScript 访问 DOM，不过并不严格要求这样。可以使用任何脚本语言来访问 DOM，这要归功于其一致的 API。表 3-1 列出了 DOM 元素的一些有用的属性，表 3-2 列出了一些有用的方法。

表 3-1 用于处理 XML 文档的 DOM 元素属性

属性名	描述
childNodes	返回当前元素所有子元素的数组
firstChild	返回当前元素的第一个下级子元素
lastChild	返回当前元素的最后一个子元素
nextSibling	返回紧跟在当前元素后面的元素
nodeValue	指定表示元素值的读/写属性
parentNode	返回元素的父节点
previousSibling	返回紧邻当前元素之前的元素

表 3-2 用于遍历 XML 文档的 DOM 元素方法

方法名	描述
getElementById(id) (document)	获取有指定惟一 ID 属性值文档中的元素
getElementsByName(name)	返回当前元素中有指定标记名的子元素的数组
hasChildNodes()	返回一个布尔值，指示元素是否有子元素
getAttribute(name)	返回元素的属性值，属性由 name 指定

有了 W3C DOM，就能编写简单的跨浏览器脚本，从而充分利用 XML 的强大功能和灵活性，将 XML 作为浏览器和服务器之间的通信介质。

从下面的例子可以看到，使用遵循 W3C DOM 的 JavaScript 来读取 XML 文档是何等简单。代码清单 3-3 显示了服务器向浏览器返回的 XML 文档的内容。这是一个简单的美国州名列表，各个州按地区划分。

### 代码清单 3-3 服务器返回的美国州名列表

```
<?xml version="1.0" encoding="UTF-8"?>
<states>
    <north>
        <state>Minnesota</state>
        <state>Iowa</state>
        <state>North Dakota</state>
    </north>
    <south>
        <state>Texas</state>
        <state>Oklahoma</state>
        <state>Louisiana</state>
    </south>
    <east>
        <state>New York</state>
        <state>North Carolina</state>
        <state>Massachusetts</state>
    </east>
    <west>
        <state>California</state>
        <state>Oregon</state>
        <state>Nevada</state>
    </west>
</states>
```

在浏览器上会生成具有两个按钮的 HTML 页面。点击第一个按钮，将从服务器加载 XML 文档，然后在警告框中显示列于文档中的所有州。点击第二个按钮也会从服务器加载 XML 文档，不过只在警告框中显示北部地区的各个州（见图 3-2）。

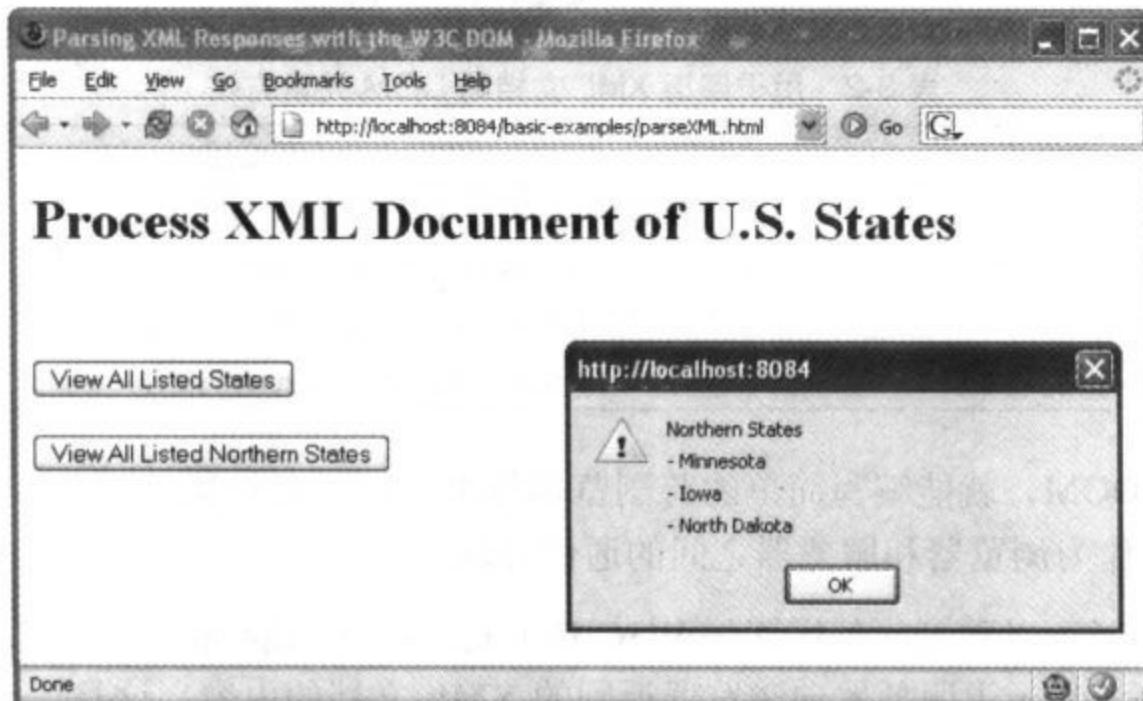


图 3-2 点击页面上的任何一个按钮都会从服务器加载 XML 文档，并在警告框中显示适当的结果

代码清单 3-4 显示了 parseXML.html。

#### 代码清单 3-4 parseXML.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Parsing XML Responses with the W3C DOM</title>

<script type="text/javascript">
var xmlhttp;
var requestType = "";

function createXMLHttpRequest() {
    if (window.ActiveXObject) {
        xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
    }
    else if (window.XMLHttpRequest) {
        xmlhttp = new XMLHttpRequest();
    }
}

function startRequest(requestedList) {
    requestType = requestedList;
    createXMLHttpRequest();
    xmlhttp.onreadystatechange = handleStateChange;
    xmlhttp.open("GET", "parseXML.xml", true);
    xmlhttp.send(null);
}

function handleStateChange() {
    if(xmlhttp.readyState == 4) {
        if(xmlhttp.status == 200) {
            if(requestType == "north") {
                listNorthStates();
            }
            else if(requestType == "all") {
                listAllStates();
            }
        }
    }
}

function listNorthStates() {
    var xmlDoc = xmlhttp.responseXML;
    var northNode = xmlDoc.getElementsByTagName("north")[0];
```

```

var out = "Northern States";
var northStates = northNode.getElementsByTagName("state");

outputList("Northern States", northStates);
}

function listAllStates() {
    var xmlDoc = xmlhttp.responseXML;
    var allStates = xmlDoc.getElementsByTagName("state");

    outputList("All States in Document", allStates);
}

function outputList(title, states) {
    var out = title;
    var currentState = null;
    for(var i = 0; i < states.length; i++) {
        currentState = states[i];
        out = out + "\n- " + currentState.childNodes[0].nodeValue;
    }
    alert(out);
}
</script>
</head>

<body>
    <h1>Process XML Document of U.S. States</h1>
    <br/><br/>
    <form action="#">
        <input type="button" value="View All Listed States"
               onclick="startRequest('all');"/>
        <br/><br/>
        <input type="button" value="View All Listed Northern States"
               onclick="startRequest('north');"/>
    </form>
</body>
</html>

```

以上脚本从服务器获取 XML 文档并加以处理，它与前面看到的例子很相似，不过前面的例子只是将响应处理为简单文本。关键区别就在于 `listNorthStates` 和 `listAllStates` 函数。前面的例子从 XMLHttpRequest 对象获取服务器响应，并使用 XMLHttpRequest 对象的 `responseText` 属性将响应获取为文本。`listNorthStates` 和 `listAllStates` 函数则不同，它们使用了 XMLHttpRequest 对象的 `responseXML` 属性，将结果获取为 XML 文档，这样一来，你就可以使用 W3C DOM 方法来遍历 XML 文档了。

仔细研究一下 `listAllStates` 函数。它首先创建了一个局部变量，名为 `xmlDoc`，并将这个变量初始化设置为服务器返回的 XML 文档，这个 XML 文档是使用 XMLHttpRequest 对象的 `responseXML` 属性得到的。利用 XML 文档的 `getElementsByTagName` 方法可以获取文档中所有标记名为 `state` 的元素。`getElementsByTagName` 方法返回了包含所有 `state` 元素的数组，这个数组将赋给名为 `allStates` 的局部变量。

从 XML 文档获取了所有 `state` 元素之后，`listAllStates` 函数调用 `outputList` 函数，并在警告框中显示这些 `state` 元素。`listAllStates` 方法将迭代处理 `state` 元素的数组，将各元素的相应州名逐个追加到一个串中，这个串最后将显示在警告框中。

有一点要特别注意，即如何从 `state` 元素获取州名。你可能认为，`state` 元素会简单地提供属性或方法来得到这个元素的文本，但并非如此。

表示州名的文本实际上是 `state` 元素的子元素。在 XML 文档中，文本本身被认为是一个节点，而且必须是另外某个元素的子元素。由于表示州名的文本实际上是 `state` 元素的子元素，所以必须先从 `state` 元素获取文本元素，再从这个文本元素得到其文本内容。

`outputList` 函数的工作就是如此。它迭代处理数组中的所有元素，将当前元素赋给 `currentState` 变量。因为表示州名的文本元素总是 `state` 元素的第一个子元素，所以可以使用 `childNodes` 属性来得到文本元素。一旦有了具体的文本元素，就可以使用 `nodeValue` 属性返回表示州名的文本内容。

`listNorthStates` 函数与 `listAllStates` 是类似的，只不过增加了一个小技巧。你只想得到北部地区的州，而不是所有州。为此，首先使用 `getElementsByTagName` 方法获取 `north` 标记，从而获得 XML 文档中的 `north` 元素。因为文档只包含一个 `north` 元素，而且 `getElementsByTagName` 方法总是返回一个数组，所以要用 `[0]` 记法来抽出 `north` 元素。这是因为，在 `getElementsByTagName` 方法返回的数组中，`north` 元素处在第一个位置上（也是唯一的位置）。既然有了 `north` 元素，接下来调用 `north` 元素的 `getElementsByTagName` 方法，就可以得到 `north` 元素的 `state` 子元素。有了 `north` 元素所有 `state` 子元素的数组后，再使用 `outputList` 方法在警告框中显示这些州名。

### 3.1.3 使用 W3C DOM 动态编辑页面

Web 最初只是作为媒介向各处分发静态的文本文档，如今它本身已经发展为一个应用开发平台。遗留的企业系统通常通过纯文本的终端部署，或者作为客户-服务器应用部署，这些遗留系统正在被完全通过 Web 浏览器部署的系统所取代。

随着最终用户越来越习惯于使用基于 Web 的应用，他们开始有了新的要求，需要一种更丰富的用户体验。用户不再满足于完全页面刷新，即每次在页面上编辑一些数据时页面都会完全刷新。他们想立即看到结果，而不是坐等与服务器完成完整的往返通信。

你已经了解了解析服务器发送的 XML 消息是多么容易。W3C DOM 提供了一些属性和方法，使你能轻松地遍历 XML 结构，并抽取所需的数据。

前面的例子对于服务器发送的 XML 响应并没有做多少有用的事情。在警告框中显示 XML 文档的值没有太大的实际意义。你真正想做到的是让用户享有丰富的客户体验，不再遭遇一般 Web 应用中常见的连续页面刷新问题。页面连续刷新不仅使用户不满意，还会浪费服务器上宝贵的处理器时间，因为页面刷新需要重新构建整个页面的内容，而且会不必要地使用网络带宽来传送刷新的页面。

当然，最好的解决办法是根据需要修改页面上已有的内容。如果页面上大多数数据没有改变，则不应刷新整个页面，只需要修改页面中信息有变化的部分。

以往，在 Web 浏览器的限制之下，这一点很难做到。浏览器只是一个工具，它解释特殊的标记（HTML），并根据一组预定的规则显示这些标记。Web 以及 Web 浏览器原来只是为了显示静态的信息，如果不以新页面的形式从服务器请求新的数据，这些信息不会改变。

除了一些例外情况，当前的浏览器都使用 W3C DOM 来表示 Web 页面的内容。这样做可以确保在不同的浏览器上 Web 页面会以同样的方式呈现，同时在不同的浏览器上，用于修改页面内容的脚本也会有相同的表现。Web 浏览器的 W3C DOM 和 JavaScript 实现越来越成熟，这大大简化了在浏览器上动态创建内容的任务。原来总是要苦心积虑地解决浏览器间的不兼容性，如今这已经不太需要。表 3-3 列出了用于动态创建内容的 DOM 属性和方法。

表 3-3 动态创建内容时所用的 W3C DOM 属性和方法

属性/方法	描述
<code>document.createElement(tagName)</code>	文档对象上的 <code>createElement</code> 方法可以创建由 <code>tagName</code> 指定的元素。如果以串 <code>div</code> 作为方法参数，就会生成一个 <code>div</code> 元素
<code>document.createTextNode(text)</code>	文档对象的 <code>createTextNode</code> 方法会创建一个包含静态文本的节点
<code>&lt;element&gt;.appendChild(childNode)</code>	<code>appendChild</code> 方法将指定的节点增加到当前元素的子节点列表（作为一个新的子节点）。例如，可以增加一个 <code>option</code> 元素，作为 <code>select</code> 元素的子节点
<code>&lt;element&gt;.getAttribute(name)</code> <code>&lt;element&gt;.setAttribute(name, value)</code>	这些方法分别获得和设置元素中 <code>name</code> 属性的值
<code>&lt;element&gt;.insertBefore(newNode, targetNode)</code>	这个方法将节点 <code>newNode</code> 作为当前元素的子节点插到 <code>targetNode</code> 元素前面
<code>&lt;element&gt;.removeAttribute(name)</code>	这个方法从元素中删除属性 <code>name</code>
<code>&lt;element&gt;.removeChild(childNode)</code>	这个方法从元素中删除子元素 <code>childNode</code>
<code>&lt;element&gt;.replaceChild(newNode, oldNode)</code>	这个方法将节点 <code>oldNode</code> 替换为节点 <code>newNode</code>
<code>&lt;element&gt;.hasChildnodes()</code>	这个方法返回一个布尔值，指示元素是否有子元素

## 关于浏览器的不兼容性

尽管当前 Web 浏览器中 W3C DOM 和 JavaScript 的实现不断改进，但还是存在一些特异性和不兼容性，这使得应用 DOM 和 JavaScript 进行开发时很是头疼。

IE 的 W3C DOM 和 JavaScript 实现最受限制。2000年初，一些统计称 IE 占据了整个浏览器市场 95% 的份额，由于没有竞争压力，Microsoft 决定不完全实现各个 Web 标准。

这些特异问题大多都能得到解决，不过这样做会让脚本更是混乱不堪而且不合标准。例如，如果使用 `appendChild` 将 `<tr>` 元素直接增加到 `<table>` 中，则在 IE 中这一行并不出现，但在其他浏览器中却会显示出来。对此的解决之道是，将 `<tr>` 元素增加到表的 `<tbody>` 元素中，这种解决办法在所有浏览器中都能正确工作。

关于 `setAttribute` 方法，IE 也有麻烦。IE 不能使用 `setAttribute` 正确地设置 `class` 属性。对此有一个跨浏览器的解决方法，即同时使用 `setAttribute("class", "newClassName")` 和 `setAttribute("className", "newClassName")`。另外，在 IE 中不能使用 `setAttribute` 设置 `style` 属性。最能保证浏览器兼容的技术不是 `<element>.setAttribute("style", "font-weight:bold;")`，而是 `<element>.style.cssText = "font-weight:bold;"`。

本书中的例子会尽可能地遵循 W3C DOM 和 JavaScript 标准，不过如果必须确保大多数当前浏览器的兼容性，可能也会稍稍偏离标准。

下面的例子展示了如何使用 W3C DOM 和 JavaScript 来动态创建内容。这个例子是假想的房地产清单搜索引擎，点击表单上的 Search（搜索）按钮，会使用 XMLHttpRequest 对象以 XML 格式获取结果。使用 JavaScript 处理响应 XML，从而生成一个表，其中列出搜索到的结果（见图 3-3）。



图 3-3 使用 W3C DOM 方法和 JavaScript 动态创建搜索结果

服务器返回的 XML 很简单（见代码清单 3-5）。根节点 properties 包含了得到的所有 property 元素。每个 property 元素包含 3 个子元素：address、price 和 comments。

#### 代码清单 3-5 dynamicContent.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<properties>
    <property>
        <address>812 Gwyn Ave</address>
        <price>$100,000</price>
        <comments>Quiet, serene neighborhood</comments>
    </property>
    <property>
        <address>3308 James Ave S</address>
        <price>$110,000</price>
        <comments>Close to schools, shopping, entertainment</comments>
    </property>
    <property>
        <address>98320 County Rd 113</address>
        <price>$115,000</price>
        <comments>Small acreage outside of town</comments>
    </property>
</properties>
```

具体向服务器发送请求并对服务器响应做出回应的 JavaScript 与前面的例子是一样的。不过，从 handleReadyStateChanged 函数开始有所不同。假设请求成功地完成，接下来第一件事就是调用 clearPreviousResults 函数，将以前搜索所创建的内容删除。

clearPreviousResults 函数完成两个任务：删除出现在最上面的“Results”标题文本，并从结果表中清除所有行。首先使用 hasChildNodes 方法查看可能包括标题文本的 span 元素是否有子元素。应该知道，只有 hasChildNodes 方法返回 true 时才存在标题文本。如果确实返回 true，则删除 span 元素的第一个（也是唯一的）子节点，因为这个子节点表示的就是标题文本。

clearPreviousResults 的下一个任务是在显示搜索结果的表中删除所有行。所有结果行都是 tbody 节点的子节点，所以先使用 document.getElementById 方法得到该 tbody 节点的引用。一旦有了 tbody 节点，只要这个 tbody 节点还有子节点（tr 元素）就进行迭代处理。每次迭代时都会从表体中删除 childNodes 集合中的第一个子节点。当表体中再没有更多的表行时，迭代结束。

搜索结果表在 parseResults 函数中建立。这个函数首先创建一个名为 results 的局部变量，这是使用 XMLHttpRequest 对象的 responseXML 属性得到的 XML 文档。

使用 getElementsByTagName 方法来获得 XML 文档中包含所有 property 元素的数组，

然后将这个数组赋给局部变量 `properties`。一旦有了 `property` 元素的数组，可以迭代处理数组中的各个元素，并获得 `property` 的 `address`、`price` 和 `comments`。

```
var properties = results.getElementsByTagName("property");
for(var i = 0; i < properties.length; i++) {
    property = properties[i];
    address = property.getElementsByTagName("address")[0].firstChild.nodeValue;
    price = property.getElementsByTagName("price")[0].firstChild.nodeValue;
    comments = property.getElementsByTagName("comments")[0].firstChild.nodeValue;

    addTableRow(address, price, comments);
}
```

下面来仔细分析这个循环，因为这正是 `parseResults` 函数的核心。在 `for` 循环中，首先得到数组中的下一个元素，并把它赋给局部变量 `property`。接下来，对于你感兴趣的各个子元素 (`address`、`price` 和 `comments`)，分别获得它们的节点值。

请考虑 `address` 元素，这是 `property` 元素的一个子元素。首先在 `property` 元素上调用 `getElementsByTagName` 方法来得到单个 `address` 元素。`getElementsByTagName` 方法返回一个数组，不过因为你知道有且仅有一个 `address` 元素，所以可以使用 `[0]` 记法来引用这个元素。

沿着 XML 结构继续向下，现在有了 `address` 标记的引用，你需要得到它的文本内容。记住，文本实际上是父元素的一个子节点，所以可以使用 `firstChild` 属性来访问 `address` 元素的文本节点。有了文本节点后，可以引用文本节点的 `nodeValue` 属性来得到文本。

采用同样的办法来得到 `price` 和 `comments` 元素的值，并把各个值分别赋给局部变量 `price` 和 `comments`。再将 `address`、`price` 和 `comments` 传递给名为 `addTableRow` 的辅助函数，它会用这些结果数据具体建立一个表行。

`addTableRow` 函数使用 W3C DOM 方法和 JavaScript 建立一个表行。使用 `document.createElement` 方法创建一个 `row` 对象，之后，再使用名为 `createCellWithText` 的辅助函数分别为 `address`、`price` 和 `comments` 值创建一个 `cell` 对象。`createCellWithText` 函数会创建并返回一个以指定的文本作为单元格内容的 `cell` 对象。

`createCellWithText` 函数首先使用 `document.createElement` 方法创建一个 `td` 元素，然后使用 `document.createTextNode` 方法创建一个包含所需文本的文本节点，所得到的文本节点追加到 `td` 元素。这个函数再把新创建的 `td` 元素返回给调用函数 (`addTableRow`)。

`addTableRow` 函数对 `address`、`price` 和 `comments` 值重复调用 `createCellWithText` 函数，每一次向 `tr` 元素追加一个新创建的 `td` 元素。一旦向 `row` (行) 增加了所有 `cell` (单元格)，这个 `row` 就将被增加到表的 `tbody` 元素中。

就这么多了！你已经成功地读取了服务器返回的 XML 文档，而且动态创建了一个结果表。代码清单 3-6 显示了这个例子完整的 JavaScript 和可扩展 HTML 代码。

### 代码清单 3-6 dynamicContent.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Dynamically Editing Page Content</title>

<script type="text/javascript">
var xmlhttp;

function createXMLHttpRequest() {
    if (window.ActiveXObject) {
        xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
    }
    else if (window.XMLHttpRequest) {
        xmlhttp = new XMLHttpRequest();
    }
}

function doSearch() {
    createXMLHttpRequest();
    xmlhttp.onreadystatechange = handleStateChange;
    xmlhttp.open("GET", "dynamicContent.xml", true);
    xmlhttp.send(null);
}

function handleStateChange() {
    if(xmlhttp.readyState == 4) {
        if(xmlhttp.status == 200) {
            clearPreviousResults();
            parseResults();
        }
    }
}

function clearPreviousResults() {
    var header = document.getElementById("header");
    if(header.childNodes.length) {
        header.removeChild(header.childNodes[0]);
    }

    var tableBody = document.getElementById("resultsBody");
}
```

```
while(tableBody.childNodes.length > 0) {
    tableBody.removeChild(tableBody.childNodes[0]);
}

function parseResults() {
    var results = xmlhttp.responseText;

    var property = null;
    var address = "";
    var price = "";
    var comments = "";

    var properties = results.getElementsByTagName("property");
    for(var i = 0; i < properties.length; i++) {
        property = properties[i];
        address = property.getElementsByTagName("address")[0].firstChild.nodeValue;
        price = property.getElementsByTagName("price")[0].firstChild.nodeValue;
        comments = property.getElementsByTagName("comments")[0]
                    .firstChild.nodeValue;
        addTableRow(address, price, comments);
    }
    var header = document.createElement("h2");
    var headerText = document.createTextNode("Results:");
    header.appendChild(headerText);
    document.getElementById("header").appendChild(header);

    document.getElementById("resultsTable").setAttribute("border", "1");
}

function addTableRow(address, price, comments) {
    var row = document.createElement("tr");
    var cell = createCellWithText(address);
    row.appendChild(cell);

    cell = createCellWithText(price);
    row.appendChild(cell);

    cell = createCellWithText(comments);
    row.appendChild(cell);

    document.getElementById("resultsBody").appendChild(row);
}

function createCellWithText(text) {
    var cell = document.createElement("td");
    var textNode = document.createTextNode(text);
```

```
cell.appendChild(textNode);

return cell;
}
</script>
</head>

<body>
<h1>Search Real Estate Listings</h1>

<form action="#">
Show listings from
<select>
<option value="50000">$50,000</option>
<option value="100000">$100,000</option>
<option value="150000">$150,000</option>
</select>
to
<select>
<option value="100000">$100,000</option>
<option value="150000">$150,000</option>
<option value="200000">$200,000</option>
</select>
<input type="button" value="Search" onclick="doSearch()"/>
</form>

<span id="header">

</span>

<table id="resultsTable" width="75%" border="0">
<tbody id="resultsBody">
</tbody>
</table>
</body>
</html>
```

## 3.2 发送请求参数

到此为止，你已经了解了如何使用 Ajax 技术向服务器发送请求，也知道了客户可以采用多种方法解析服务器的响应。前面的例子中只缺少一个内容，就是你尚未将任何数据作为请求的一部分发送给服务器。在大多数情况下，向服务器发送一个请求而没有任何请求参数是没有什么意义的。如果没有请求参数，服务器就得不到上下文数据，也无法根据上下文数据为客户创建“个性化”的响应，实际上，服务器会向每一个客户发送同样的响应。

要想充分发挥 Ajax 技术的强大功能，这要求你向服务器发送一些上下文数据。假设有一个输入表单，其中包含需要输入邮件地址的部分。根据用户输入的 ZIP 编码，可以使用 Ajax 技术预填相应的城市名。当然，要想查找 ZIP 编码对应的城市，服务器首先需要知道用户输入的 ZIP 编码。

你需要以某种方式将用户输入的 ZIP 编码值传递给服务器。幸运的是，`XMLHttpRequest` 对象的工作与你以往惯用的 HTTP 技术（`GET` 和 `POST`）是一样的。

`GET` 方法把值作为名/值对放在请求 URL 中传递。资源 URL 的最后有一个问号（?），问号后面就是名/值对。名/值对采用 `name=value` 的形式，各个名/值对之间用与号（&）分隔。

下面是 `GET` 请求的一个例子。这个请求向 `localhost` 服务器上的 `yourApp` 应用发送了两个参数：`firstName` 和 `middleName`。需要注意，资源 URL 和参数集之间用问号分隔，`firstName` 和 `middleName` 之间用与号（&）分隔：

```
http://localhost/yourApp?firstName=Adam&middleName=Christopher
```

服务器知道如何获取 URL 中的命名参数。当前大多数服务器端编程环境都提供了简单的 API，使你能很容易地访问命名参数。

采用 `POST` 方法向服务器发送命名参数时，与采用 `GET` 方法几乎是一样的。类似于 `GET` 方法，`POST` 方法会把参数编码为名/值对，形式为 `name=value`，每个名/值对之间也用与号（&）分隔。这两种方法的主要区别在于，`POST` 方法将参数串放在请求体中发送，而 `GET` 方法是将参数追加到 URL 中发送。

如果数据处理不改变数据模型的状态，HTML 使用规约理论上推荐采用 `GET` 方法，从这可以看出，获取数据时应当使用 `GET` 方法。如果因为存储、更新数据，或者发送了电子邮件，操作改变了数据模型的状态，这时建议使用 `POST` 方法。

每个方法都有各自特有的优点。由于 `GET` 请求的参数编码到请求 URL 中，所以可以在浏览器中为该 URL 建立书签，以后就能很容易地重新请求。不过，如果是异步请求就没有什么用。从发送到服务器的数据量来讲，`POST` 方法更为灵活。使用 `GET` 请求所能发送的数据量通常是固定的，因浏览器不同而有所差异，而 `POST` 方法可以发送任意量的数据。

HTML `form` 元素允许通过将 `form` 元素的 `method` 属性设置为 `GET` 或 `POST` 来指定所需的方法。在提交表单时，`form` 元素自动根据其 `method` 属性的规则对 `input` 元素的数据进行编码。`XMLHttpRequest` 对象没有这种内置行为。相反，要由开发人员使用 JavaScript 创建查询串，其中包含的数据要作为请求的一部分发送给服务器。不论使用的是 `GET` 请求还是 `POST` 请求，创建查询串的技术是一样的。唯一的区别是，当使用 `GET` 发送请求时，查询串会追加到请求 URL 中，而使用 `POST` 方法时，则在调用 `XMLHttpRequest` 对象的 `send()` 方法时发送查询串。

图 3-4 显示了一个示例页面，展示了如何向服务器发送请求参数。这是一个简单的输入表单，要求输入名、姓和生日。这个表单有两个按钮，每个按钮都会向服务器发送名、姓和生日数据，不过一个使用 GET 方法，另一个使用 POST 方法。服务器以回显输入数据作为响应。在浏览器在页面上打印出服务器的响应时，请求响应周期结束。

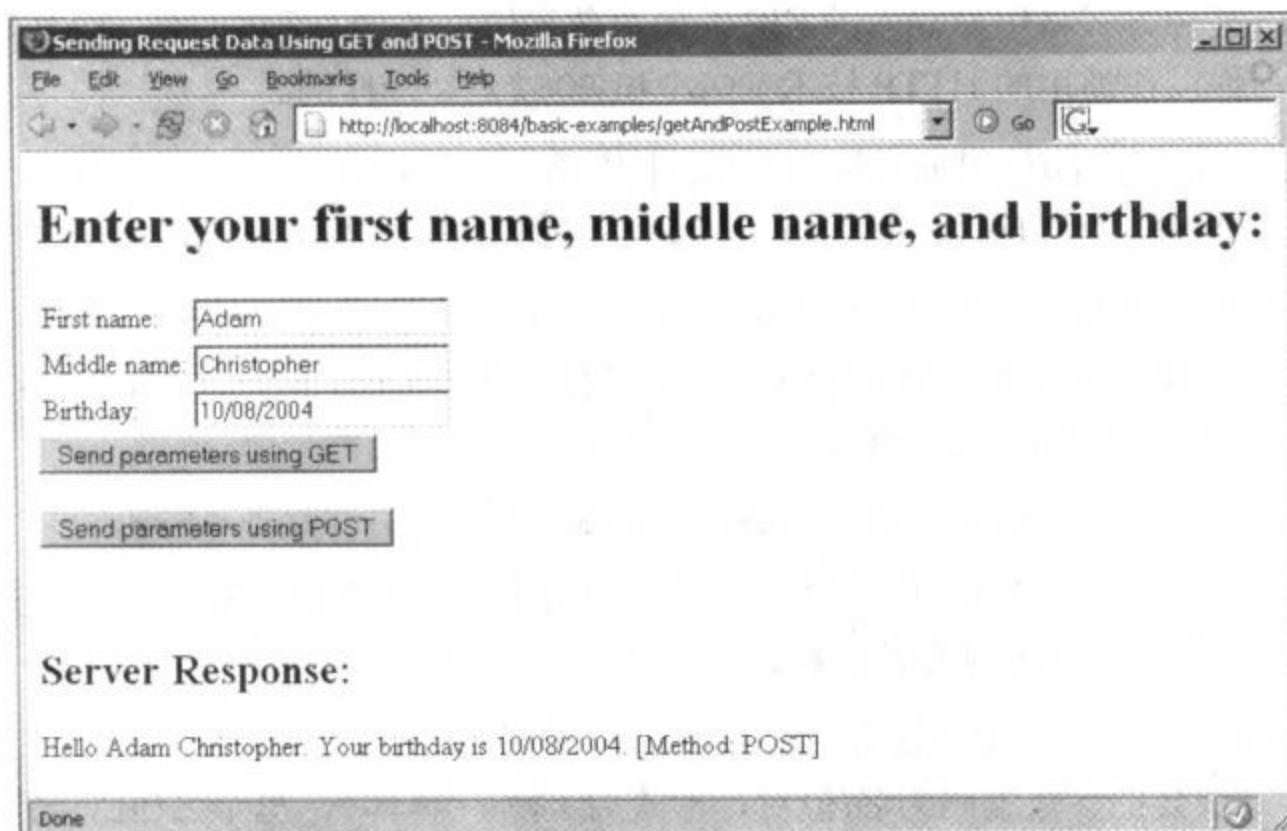


图 3-4 浏览器使用 GET 或 POST 方法发送输入数据，服务器回显输入数据作为响应

代码清单 3-7 显示了 `getAndPostExample.html`，代码清单 3-8 显示了向浏览器回显名、姓和生日数据的 Java servlet。

#### 代码清单 3-7 `getAndPostExample.html`

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Sending Request Data Using GET and POST</title>

<script type="text/javascript">
var xmlhttp;

function createXMLHttpRequest() {
  if (window.ActiveXObject) {
    xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
  }
  else if (window.XMLHttpRequest) {
    xmlhttp = new XMLHttpRequest();
  }
}
```

```
        }
    }

    function createQueryString() {
        var firstName = document.getElementById("firstName").value;
        var middleName = document.getElementById("middleName").value;
        var birthday = document.getElementById("birthday").value;

        var queryString = "firstName=" + firstName + "&middleName=" + middleName
            + "&birthday=" + birthday;

        return queryString;
    }

    function doRequestUsingGET() {
        createXMLHttpRequest();

        var queryString = "GetAndPostExample?";
        queryString = queryString + createQueryString()
            + "&timeStamp=" + new Date().getTime();
        xmlhttp.onreadystatechange = handleStateChange;
        xmlhttp.open("GET", queryString, true);
        xmlhttp.send(null);
    }

    function doRequestUsingPOST() {
        createXMLHttpRequest();

        var url = "GetAndPostExample?timeStamp=" + new Date().getTime();
        var queryString = createQueryString();

        xmlhttp.open("POST", url, true);
        xmlhttp.onreadystatechange = handleStateChange;
        xmlhttp.setRequestHeader("Content-Type",
            "application/x-www-form-urlencoded");
        xmlhttp.send(queryString);
    }

    function handleStateChange() {
        if(xmlhttp.readyState == 4) {
            if(xmlhttp.status == 200) {
                parseResults();
            }
        }
    }

    function parseResults() {
```

```
var responseDiv = document.getElementById("serverResponse");
if(responseDiv.hasChildNodes()) {
    responseDiv.removeChild(responseDiv.childNodes[0]);
}
var responseText = document.createTextNode(xmlHttp.responseText);
responseDiv.appendChild(responseText);
}

</script>
</head>

<body>
<h1>Enter your first name, middle name, and birthday:</h1>

<table>
<tbody>
<tr>
<td>First name:</td>
<td><input type="text" id="firstName"/>
</tr>
<tr>
<td>Middle name:</td>
<td><input type="text" id="middleName"/>
</tr>
<tr>
<td>Birthday:</td>
<td><input type="text" id="birthday"/>
</tr>
</tbody>
</table>

<form action="#">
<input type="button" value="Send parameters using GET"
      onclick="doRequestUsingGET();"/>

<br/><br/>
<input type="button" value="Send parameters using POST"
      onclick="doRequestUsingPOST();"/>
</form>

<br/>
<h2>Server Response:</h2>

<div id="serverResponse"></div>
```

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

### 代码清单 3-8 向浏览器回显名、姓和生日

```
package ajaxbook.chap3;

import java.io.*;
import java.net.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class GetAndPostExample extends HttpServlet {

    protected void processRequest(HttpServletRequest request,
                                  HttpServletResponse response, String method)
    throws ServletException, IOException {

        //Set content type of the response to text/xml
        response.setContentType("text/xml");

        //Get the user's input
        String firstName = request.getParameter("firstName");
        String middleName = request.getParameter("middleName");
        String birthday = request.getParameter("birthday");

        //Create the response text
        String responseText = "Hello " + firstName + " " + middleName
            + ". Your birthday is " + birthday + "."
            + " [Method: " + method + "]";

        //Write the response back to the browser
        PrintWriter out = response.getWriter();
        out.println(responseText);

        //Close the writer
        out.close();
    }

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
        //Process the request in method processRequest
        processRequest(request, response, "GET");
    }

    protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
```

```

    //Process the request in method processRequest
    processRequest(request, response, "POST");
}
}

```

下面先来分析服务器端代码。这个例子使用了 Java servlet 来处理请求，不过也可以使用任何其他服务器端技术，如 PHP、CGI 或.NET。Java servlet 必须定义一个 doGet 方法和一个 doPost 方法，每个方法根据请求方法（GET 或 POST）来调用。在这个例子中，doGet 和 doPost 都调用同样的方法 processRequest 来处理请求。

processRequest 方法先把响应的内容类型设置为 text/xml，尽管在这个例子中并没有真正用到 XML。通过使用 getParameter 方法从 request 对象获得 3 个输入字段。根据名、姓和生日，以及请求方法的类型，会建立一个简单的语句。这个语句将写至响应输出流，最后响应输出流关闭。

浏览器端 JavaScript 与前面的例子同样是类似的，不过这里稍稍增加了几个技巧。这里有一个工具函数 createQueryString 负责将输入参数编码为查询串。createQueryString 函数只是获取名、姓和生日的输入值，并将它们追加为名/值对，每个名/值对之间由与号（&）分隔。这个函数会返回查询串，以便 GET 和 POST 操作重用。

点击 Send Parameters Using GET（使用 GET 方法发送参数）按钮将调用 doRequestUsingGET 函数。这个函数与前面例子中的许多函数一样，先调用创建 XMLHttpRequest 对象实例的函数。接下来，对输入值编码，创建查询串。

在这个例子中，请求端点是名为 GetAndPostExample 的 servlet。在建立查询串时，要把 createQueryString 函数返回的查询串与请求端点连接，中间用问号分隔。

JavaScript 仍与前面看到的类似。XMLHttpRequest 对象的 onreadystatechange 属性设置为要使用 handleStateChange 函数。open() 方法指定这是一个 GET 请求，并指定了端点 URL，在这里端点 URL 中包含有编码的参数。send() 方法将请求发送给服务器，handleStateChange 函数处理服务器响应。

当请求成功完成时，handleStateChange 函数将调用 parseResults 函数。parseResults 函数获取 div 元素，其中包含服务器的响应，并把它保存在局部变量 responseDiv 中。使用 responseDiv 的 removeChild 方法先将以前的服务器结果删除。最后，创建包含服务器响应的新文本节点，并将这个文本节点追加到 responseDiv。

使用 POST 方法而不是 GET 方法基本上是一样的，只是请求参数发送给服务器的方式不同。应该记得，使用 GET 时，名/值对会追加到目标 URL。POST 方法则把同样的查询串作为请求体的一部分发送。

点击 Send Parameters Using POST（使用 POST 方法发送参数）按钮将调用 doRequest-

UsingPOST函数。类似于doRequestUsingGET函数，它先创建 XMLHttpRequest 对象的一个实例，脚本再创建查询串，其中包含要发送给服务器的参数。需要注意，查询串现在并不连接到目标 URL。

接下来调用XMLHttpRequest对象的open()方法，这一次指定请求方法是POST，另外指定了没有追加名/值对的“原”目标URL。onreadystatechange属性设置为handleStateChange函数，所以响应会以与GET方法中相同的方式得到处理。为了确保服务器知道请求体中有请求参数，需要调用setRequestHeader，将Content-Type值设置为application/x-www-form-urlencoded。最后，调用send()方法，并把查询串作为参数传递给这个方法。

点击两个按钮的结果是一样的。页面上会显示一个串，其中包括指定的名、姓和生日，另外还会显示所用请求方法的类型。

### 为什么要把时间戳追加到目标 URL?

在某些情况下，有些浏览器会把多个 XMLHttpRequest 请求的结果缓存在同一个 URL。如果对每个请求的响应不同，这就会带来不好的结果。把当前时间戳追加到 URL 的最后，就能确保 URL 的惟一性，从而避免浏览器缓存结果。

#### 3.2.1 请求参数作为 XML 发送

与几年前相比，当前浏览器上 JavaScript 的兼容性有了长足的进步，已经不可同日而语，再加上越来越成熟的 JavaScript 开发工具和技术，你可以决定把 Web 浏览器作为开发平台。并不只是依赖于浏览器来看待模型-视图-控制器模式中的视图，还可以用 JavaScript 实现部分业务模型。可以使用 Ajax 技术把模型中的变化持久存储到后台服务器。如果模型放在浏览器上，模型的变化可以一齐传递到服务器，从而减少对服务器的远程调用次数，还可能提高性能。

如果只是使用一个包含名/值对的简单查询串，这可能不够健壮，不足以向服务器传递大量复杂的模型变化。更好的解决方案是将模型的变化作为 XML 发送到服务器。怎么向服务器发送 XML 呢？

可以把 XML 作为请求体的一部分发送到服务器，这与 POST 请求中将查询串作为请求体的一部分进行发送异曲同工。服务器可以从请求体读到 XML，并加以处理。

下面的例子展示了对于一个 Ajax 请求如何向服务器发送 XML。图 3-5 显示了这个页面，其中有一个简单的选择框，用户可以选择宠物的类型。这是一个相当简化的例子，但是由此可以了解如何向服务器发送 XML。

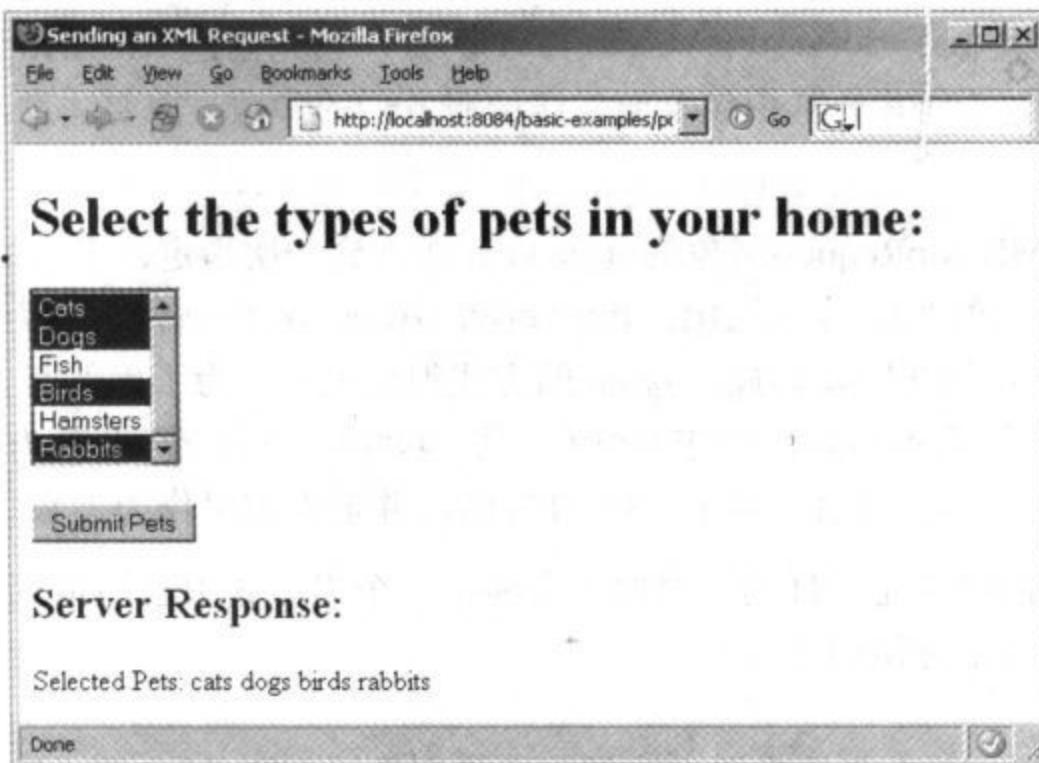


图 3-5 选择框中选中的项将作为 XML 发送到服务器

代码清单 3-9 显示了 postingXML.html。

#### 代码清单 3-9 postingXML.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Sending an XML Request</title>

<script type="text/javascript">

var xmlhttp;

function createXMLHttpRequest() {
    if (window.ActiveXObject) {
        xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
    }
    else if (window.XMLHttpRequest) {
        xmlhttp = new XMLHttpRequest();
    }
}

function createXML() {
    var xml = "<pets>";
    var options = document.getElementById("petTypes").childNodes;
    var option = null;
    for(var i = 0; i < options.length; i++) {
```

```
option = options[i];
if(option.selected) {
    xml = xml + "<type>" + option.value + "</type>";
}
}

xml = xml + "</pets>";
return xml;
}

function sendPetTypes() {
    createXMLHttpRequest();

    var xml = createXML();
    var url = "PostingXMLExample?timeStamp=" + new Date().getTime();

    xmlhttp.open("POST", url, true);
    xmlhttp.onreadystatechange = handleStateChange;
    xmlhttp.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
    xmlhttp.send(xml);
}

function handleStateChange() {
    if(xmlhttp.readyState == 4) {
        if(xmlhttp.status == 200) {
            parseResults();
        }
    }
}

function parseResults() {
    var responseDiv = document.getElementById("serverResponse");
    if(responseDiv.childNodes()) {
        responseDiv.removeChild(responseDiv.childNodes[0]);
    }

    var responseText = document.createTextNode(xmlhttp.responseText);
    responseDiv.appendChild(responseText);
}

</script>
</head>
<body>
<h1>Select the types of pets in your home:</h1>

<form action="#">
```

```

<select id="petTypes" size="6" multiple="true">
    <option value="cats">Cats</option>
    <option value="dogs">Dogs</option>
    <option value="fish">Fish</option>
    <option value="birds">Birds</option>
    <option value="hamsters">Hamsters</option>
    <option value="rabbits">Rabbits</option>
</select>

<br/><br/>
<input type="button" value="Submit Pets" onclick="sendPetTypes() ; " />
</form>

<h2>Server Response:</h2>

<div id="serverResponse"></div>

</body>
</html>

```

这个例子与前面的 POST 例子基本上是一样的。区别在于，不是发送由名/值对组成的查询串，而是向服务器发送 XML 串。

点击表单上的 Submit Pets（提交宠物）按钮将调用 sendPetTypes 函数。类似于前面的例子，这个函数首先创建 XMLHttpRequest 对象的一个实例，然后调用名为 createXML 的辅助函数，它根据所选的宠物类型建立 XML 串。

函数 createXML 使用 document.getElementById 方法获得 select 元素的引用，然后迭代处理所有 option 子元素，对于选中的每个 option 元素依据所选宠物类型创建 XML 标记，并逐个追加到 XML 中。循环结束时，要在返回到调用函数（sendPetTypes）之前向 XML 串追加结束 pets 标记。

一旦得到了 XML 串，sendPetTypes 函数继续为请求准备 XMLHttpRequest，然后把 XML 串指定为 send() 方法的参数，从而将 XML 发送到服务器。

### 在 createXML 方法中，为什么结束标记中斜线前面有一个反斜线？

SGML 规约（HTML 就是从 SGML 发展来的）中提供了一个技巧，利用这个技巧可以识别出 script 元素中的结束标记，但是其他内容（如开始标记和注释）则不能识别。使用反斜线可以避免把串解析为标记。即使没有反斜线，大多数浏览器也能安全地处理，但是根据严格的 XHTML 标准，应该使用反斜线。

聪明的读者可能注意到，根据 XMLHttpRequest 对象的文档，send() 方法可以将串和

XML 文档对象实例作为参数。那么，这个例子为什么使用串连接来创建 XML，而不是直接创建文档和元素对象呢？遗憾的是，对于从头构建文档对象，目前还没有跨浏览器的技术。IE 通过 ActiveX 对象提供这个功能，Mozilla 浏览器则通过本地 JavaScript 对象来提供，其他浏览器可能根本不支持，也可能通过其他途径来支持这个功能。

读取 XML 的服务器端代码如代码清单 3-10 所示，这个代码稍有些复杂。在此使用了 Java servlet 来读取请求，并解析 XML 串，不过你也可以使用其他的服务器端技术。

一旦收到 XMLHttpRequest 对象的请求，就会调用这个 servlet 的 doPost 方法。doPost 方法使用名为 readXMLFromRequestBody 的辅助方法从请求体中抽取 XML，然后使用 JAXP 接口将 XML 串转换为 Document 对象。

注意，Document 对象是 W3C 指定的 Document 接口的一个实例。因此，它与浏览器的 Document 对象有着同样的方法，如 getElementsByTagName。可以使用这个方法来得到文档中所有 type 元素的列表。对于文档中的每个 type 元素，会得到文本值（应该记得，文本值是 type 元素的第一个子节点），并逐个追加到串中。处理完所有 type 元素后，响应串写回到浏览器。

#### 代码清单 3-10 PostingXMLExample.java

```
package ajaxbook.chap3;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;
import org.w3c.dom.Document;
import org.w3c.dom.NodeList;
import org.xml.sax.SAXException;

public class PostingXMLExample extends HttpServlet {

    protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

        String xml = readXMLFromRequestBody(request);
        Document xmlDoc = null;
        try {
            xmlDoc =
                DocumentBuilderFactory.newInstance().newDocumentBuilder()
                    .parse(new ByteArrayInputStream(xml.getBytes()));
        }
        catch(ParserConfigurationException e) {
```

```
        System.out.println("ParserConfigurationException: " + e);
    }
    catch(SAXException e) {
        System.out.println("SAXException: " + e);
    }

    /* Note how the Java implementation of the W3C DOM has the same methods
     * as the JavaScript implementation, such as getElementsByTagName and
     * getNodeValue.
     */
    NodeList selectedPetTypes = xmlDoc.getElementsByTagName("type");
    String type = null;
    String responseText = "Selected Pets: ";
    for(int i = 0; i < selectedPetTypes.getLength(); i++) {
        type = selectedPetTypes.item(i).getFirstChild().getNodeValue();
        responseText = responseText + " " + type;
    }

    response.setContentType("text/xml");
    response.getWriter().print(responseText);
}

private String readXMLFromRequestBody(HttpServletRequest request){
    StringBuffer xml = new StringBuffer();
    String line = null;
    try {
        BufferedReader reader = request.getReader();
        while((line = reader.readLine()) != null) {
            xml.append(line);
        }
    }
    catch(Exception e) {
        System.out.println("Error reading XML: " + e.toString());
    }
    return xml.toString();
}
}
```

### 3.2.2 使用 JSON 向服务器发送数据

做了这么多，你已经能更顺手地使用 JavaScript 了，也许在考虑把更多的模型信息放在浏览器上。不过，看过前面的例子后（使用 XML 向服务器发送复杂的数据结构），你可能会改变主意。通过串连接来创建 XML 串并不好，这也不是用来生成或修改 XML 数据结构的健壮技术。

## JSON概述

XML的一个替代方法是JSON，可以在[www.json.org](http://www.json.org)找到。JSON是一种文本格式，它独立于具体语言，但是使用了与C系列语言（如C、C#、JavaScript等）类似的约定。JSON建立在以下两种数据结构基础上，当前几乎所有编程语言都支持这两种数据结构：

- 名/值对集合。在当前编程语言中，这实现为一个对象、记录或字典。
- 值的有序表，这通常实现为一个数组。

因为这些结构得到了如此众多编程语言的支持，所以JSON是一个理想的选择，可以作为异构系统之间的一种数据互换格式。另外，由于JSON是基于标准JavaScript的子集，所以在所有当前Web浏览器上都应该是兼容的。

JSON对象是名/值对的无序集合。对象以{开始，以}结束，名/值对用冒号分隔。JSON数组是一个有序的值集合，以[开始，以]结束，数组中的值用逗号分隔。值可以是串（用双引号引起）、数值、true或false、对象，或者是数组，因此结构可以嵌套。图3-6以图形方式很好地描述了JSON对象的标记。

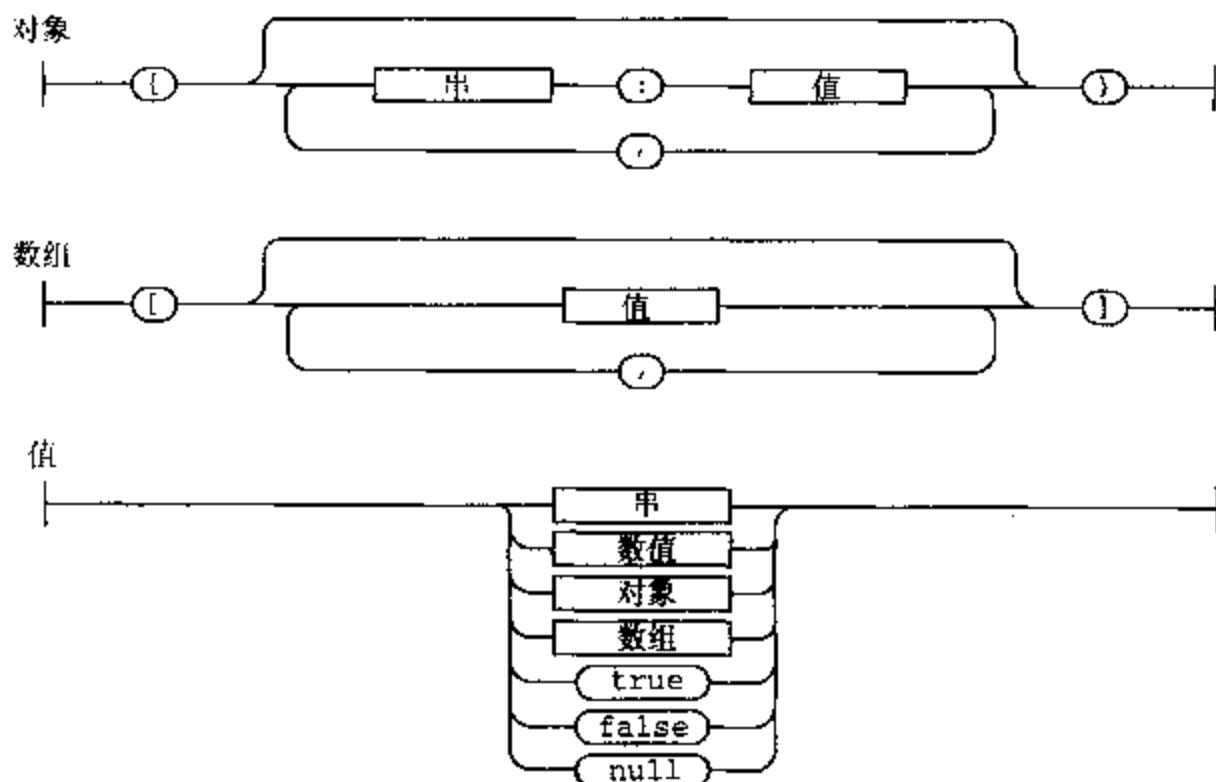


图3-6 JSON对象结构的图形化表示（摘自[www.json.org](http://www.json.org)）

请考虑employee对象的简单例子。employee对象可能包含名、姓、员工号和职位等数据。使用JSON，可以如下表示employee对象实例：

```
var employee = {
    "firstName" : John
    , "lastName" : Doe
    , "employeeNumber" : 123
```

```

        , "title" : "Accountant"
    }

```

然后可以使用标准点记法使用对象的属性，如下所示：

```

var lastName = employee.lastName;      //Access the last name
var title = employee.title;           //Access the title
employee.employeeNumber = 456;         //Change the employee number

```

JSON 有一点很引以为豪，这就是它是一个轻量级的数据互换格式。如果用 XML 来描述同样的 employee 对象，可能如下所示：

```

<employee>
    <firstName>John</firstName>
    <lastName>Doe</lastName>
    <employeeNumber>123</employeeNumber>
    <title>Accountant</title>
</employee>

```

显然，JSON 编码比 XML 编码简短。JSON 编码比较小，所以如果在网络上发送大量数据，可能会带来显著的性能差异。

[www.json.org](http://www.json.org) 网站列出了至少与其他编程语言的 14 种绑定，这说明，不论在服务器端使用何种技术，都能通过 JSON 与浏览器通信。

### 使用JSON的示例

下面是一个简单的例子，展示了如何使用 JSON 将 JavaScript 对象转换为串格式，并使用 Ajax 技术将这个串发送到服务器，然后服务器根据这个串创建一个对象。这个例子中没有业务逻辑，也几乎没有用户交互，它强调的是客户端和服务器端的 JSON 技术。图 3-7 显示了一个“字符串化的”Car 对象。

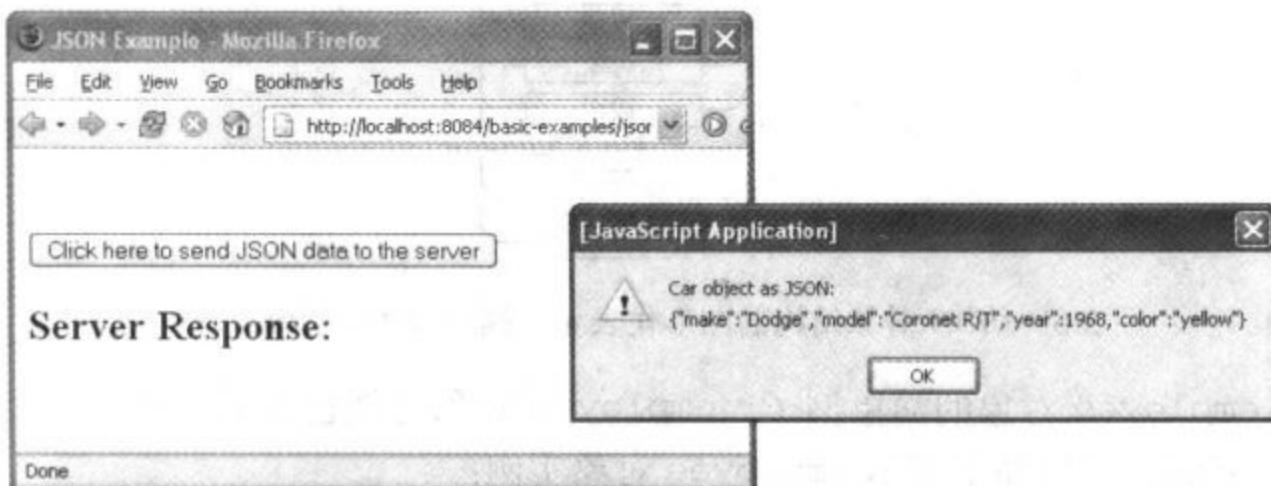


图 3-7 “字符串化的” Car 对象

因为这个例子几乎与前面的 POST 例子完全相同，所以我们只关注 JSON 特定的技术。点击表单上的按钮将调用 doJSON 函数。这个函数首先调用 getCarObject 函数来返回一个

新的 Car 对象实例，然后使用 JSON JavaScript 库（可以从 [www.json.org](http://www.json.org) 免费得到）将 Car 对象转换为 JSON 串，再在警告框中显示这个串。接下来使用 XMLHttpRequest 对象将 JSON 编码的 Car 对象发送到服务器。

因为有可以免费得到的 JSON-Java 绑定库，所以编写 Java servlet 来为 JSON 请求提供服务相当简单。更妙的是，由于对每种服务器端技术都有相应的 JSON 绑定，所以可以使用任何服务器端技术实现这个例子。

JSONExample servlet 的 doPost 方法为 JSON 请求提供服务。它首先调用 readJSONObjectFromRequestBody 方法从请求体获得 JSON 串，然后创建 JSONObject 的一个实例，向 JSONObject 构造函数提供 JSON 串。JSONObject 在对象创建时自动解析 JSON 串。一旦创建了 JSONObject，就可以使用各个 get 方法来获得你感兴趣的对像属性。

这里使用 getString 和 getInt 方法来获取 year、make、model 和 color 属性。这些属性连接起来构成一个串返回给浏览器，并在页面上显示。图 3-8 显示了读取 JSON 对象之后的服务器响应。

代码清单 3-11 显示了 jsonExample.html，代码清单 3-12 显示了 JSONExample.java。



图 3-8 读取 JSON 串之后的服务器响应

### 代码清单 3-11 jsonExample.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>JSON Example</title>

<script type="text/javascript" src="json.js"></script>
<script type="text/javascript">

var xmlhttp;

function createXMLHttpRequest() {
    if (window.ActiveXObject) {
        xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
    }
    else if (window.XMLHttpRequest) {
        xmlhttp = new XMLHttpRequest();
    }
}
```

```
}

function doJSON() {
    var car = getCarObject();

    //Use the JSON JavaScript library to stringify the Car object
    var carAsJSON = JSON.stringify(car);
    alert("Car object as JSON:\n " + carAsJSON);

    var url = "JSONExample?timeStamp=" + new Date().getTime();

    createXMLHttpRequest();
    xmlhttp.open("POST", url, true);
    xmlhttp.onreadystatechange = handleStateChange;
    xmlhttp.setRequestHeader("Content-Type",
        "application/x-www-form-urlencoded");
    xmlhttp.send(carAsJSON);
}

function handleStateChange() {
    if(xmlhttp.readyState == 4) {
        if(xmlhttp.status == 200) {
            parseResults();
        }
    }
}

function parseResults() {
    var responseDiv = document.getElementById("serverResponse");
    if(responseDiv.hasChildNodes()) {
        responseDiv.removeChild(responseDiv.childNodes[0]);
    }

    var responseText = document.createTextNode(xmlhttp.responseText);
    responseDiv.appendChild(responseText);
}

function getCarObject() {
    return new Car("Dodge", "Coronet R/T", 1968, "yellow");
}

function Car(make, model, year, color) {
    this.make = make;
    this.model = model;
    this.year = year;
    this.color = color;
}
```

```
</script>
</head>

<body>

<br/><br/>
<form action="#">
    <input type="button" value="Click here to send JSON data to the server"
           onclick="doJSON();"/>
</form>

<h2>Server Response:</h2>

<div id="serverResponse"></div>

</body>
</html>
```

### 代码清单 3-12 JSONExample.java

```
package ajaxbook.chap3;

import java.io.*;
import java.net.*;
import java.text.ParseException;
import javax.servlet.*;
import javax.servlet.http.*;
import org.json.JSONObject;

public class JSONExample extends HttpServlet {

    protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        String json = readJSONStringFromRequestBody(request);

        //Use the JSON-Java binding library to create a JSON object in Java
        JSONObject jsonObject = null;
        try {
            jsonObject = new JSONObject(json);
        }
        catch(ParseException pe) {
            System.out.println("ParseException: " + pe.toString());
        }

        String responseText = "You have a " + jsonObject.getInt("year") + " "
                           + jsonObject.getString("make") + " " + jsonObject.getString("model")
```

```

        + " " + " that is " + jsonObject.getString("color") + " in color.";

    response.setContentType("text/xml");
    response.getWriter().print(responseText);
}

private String readJSONStringFromRequestBody(HttpServletRequest request) {
    StringBuffer json = new StringBuffer();
    String line = null;
    try {
        BufferedReader reader = request.getReader();
        while((line = reader.readLine()) != null) {
            json.append(line);
        }
    } catch(Exception e) {
        System.out.println("Error reading JSON string: " + e.toString());
    }
    return json.toString();
}
}

```

### 3.3 小结

本章介绍了 XMLHttpRequest 对象与服务器之间相互通信的各种方法。XMLHttpRequest 对象可以使用 HTTP GET 或 POST 方法发送请求，请求数据可以作为查询串、XML 或 JSON 数据发送。处理请求之后，服务器一般会发送简单文本、XML 数据甚至 JSON 数据作为响应。每个格式都有自己最适用的场合。

如果不能根据请求的结果动态更新页面的内容，Ajax 就没有多大的用处。当前的浏览器都把 Web 页面的内容提供为一个遵循 W3C DOM 标准的对象模型。基于这个对象模型，就可以使用 JavaScript 之类的脚本语言在页面上增加、更新和删除内容，而不必与服务器建立往返通信。尽管还是存在一些特异的地方，但如果 Web 页面是根据 W3C 标准编写的，并使用标准 JavaScript 修改，那么在所有与标准兼容的浏览器上这些页面大多都有同样的表现。如今的浏览器还支持非标准的 innerHTML 属性，可以用来更新 Web 页面上的元素。

你现在已经熟悉了 XMLHttpRequest 对象，并且了解了如何使用 XMLHttpRequest 对象与服务器进行无缝通信。你还知道了怎样动态地更新 Web 页面的内容。下面再学些什么呢？

Ajax 的潜力无穷无尽，第 4 章将就此简单地谈一谈。知道如何使用 Ajax 只是一方面，如何在合适的环境中加以应用则是另一方面。下一章会介绍一些常见的情况，在这些情况下，Web 应用就很适合采用 Ajax 技术。

## 实现基本 Ajax 技术

我们已经介绍了 Ajax 技术，也知道了如何使用 XMLHttpRequest 对象，现在要把它结合起来，该怎么做呢？哪些情况下需要应用 Ajax 技术？当然，Ajax 的潜力几乎是无穷尽的，关于 Ajax 的使用，灵感可能源源不断。本章将展示一些例子，在这些情况下，使用 Ajax 技术可以让应用突飞猛进。有些情况是一目了然的，有些则不是。不过无论怎样，对 Ajax 应用积累的经验越多，你就越会找到自己的方法来改善应用。在这些例子中，大多数都使用 Java servlet 作为服务器端组件，其实每个例子也都能很容易地使用.NET、Ruby、Perl、PHP 或任何其他服务器端技术来编写。

### 4.1 完成验证

关于可用性有一句金玉良言，即防患于未然，根本杜绝错误的发生。但是如果真的出现了错误，你就要第一时间通知用户。在 Ajax 之前，基于 Web 的应用必须提交整个页面才能验证数据，或者要依赖复杂的 JavaScript 来检查表单。尽管有些检查确实很简单，可以使用 JavaScript 编写，但另外一些检查则不然，完全靠 JavaScript 编写是办不到的。当然，在客户端编写的每一个验证例程都必须在服务器上以某种方式重写，因为用户有可能禁用 JavaScript。

利用 Ajax，你不用再受这个限制，不再只是编写简单的客户端验证和重复的逻辑。现在，如果你想为用户提供更能体现交互性的体验，可以简单地调用为服务器编写的验证例程。在大多数情况下，这个逻辑编写起来更简单，测试也更容易，而且完全可以借助于现有的框架。

有人问，在应用中应该从哪里开始使用 Ajax，我们一般会建议从验证开始。你很可能要去掉一些 JavaScript，而且可以很容易地加入一些现有的服务器端逻辑。本节将介绍一个例子，这是最常见的验证之一：日期验证。

这个例子的 HTML 很简单（见代码清单 4-1）。其中有一个标准的输入框，相应的 onchange() 事件（当然，可以使用你认为合适的任何事件）会触发验证方法。可以看到，要

调用标准的 `createXMLHttpRequest()` 方法，然后把输入值发送到 `ValidationServlet`。`callback()` 函数从服务器得到结果，然后委托给 `setMessage()` 方法，这个方法会检查值以确定用什么颜色显示消息。

#### 代码清单 4-1 validation.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" >

<html>
  <head>
    <title>Using Ajax for validation</title>

    <script type="text/javascript">
        var xmlhttp;

        function createXMLHttpRequest() {
            if (window.ActiveXObject) {
                xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
            }
            else if (window.XMLHttpRequest) {
                xmlhttp = new XMLHttpRequest();
            }
        }

        function validate() {
            createXMLHttpRequest();
            var date = document.getElementById("birthDate");
            var url = "ValidationServlet?birthDate=" + escape(date.value);
            xmlhttp.open("GET", url, true);
            xmlhttp.onreadystatechange = callback;
            xmlhttp.send(null);
        }

        function callback() {
            if (xmlhttp.readyState == 4) {
                if (xmlhttp.status == 200) {
                    var mes =
                        xmlhttp.responseXML
                            .getElementsByTagName("message")[0].firstChild.data;
                    var val =
                        xmlhttp.responseXML
                            .getElementsByTagName("passed")[0].firstChild.data;
                    setMessage(mes, val);
                }
            }
        }
    </script>
  </head>
  <body>
    <input type="text" id="birthDate" value="1970-01-01" />
    <input type="button" value="Validate" onclick="validate()" />
  </body>
</html>
```

```

        function setMessage(message, isValid) {
            var messageArea = document.getElementById("dateMessage");
            var fontColor = "red";
            if (isValid == "true") {
                fontColor = "green";
            }
            messageArea.innerHTML = "<font color=" + fontColor + ">" +
                message + " </font>";
        }

    </script>
</head>
<body>
    <h1>Ajax Validation Example</h1>
    Birth date: <input type="text" size="10" id="birthDate" onchange="validate()"/>
    <div id="dateMessage"></div>
</body>
</html>

```

服务器端代码也很简单（见代码清单 4-2）。为简单起见，这里把验证代码放在 servlet 中，而在生产环境中很可能会把验证代码委托给验证服务。

#### 代码清单 4-2 ValidationServlet.java

```

package ajaxbook.chap4;

import java.io.*;
import java.text.ParseException;
import java.text.SimpleDateFormat;

import javax.servlet.*;
import javax.servlet.http.*;

public class ValidationServlet extends HttpServlet {

    /**
     * Handles the HTTP <code>GET</code> method.
     * @param request servlet request
     * @param response servlet response
     */
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        PrintWriter out = response.getWriter();

        boolean passed = validateDate(request.getParameter("birthDate"));
        response.setContentType("text/xml");
        response.setHeader("Cache-Control", "no-cache");
        String message = "You have entered an invalid date.";

```

```
if (passed) {
    message = "You have entered a valid date.";
}
out.println("<response>");
out.println("<passed>" + Boolean.toString(passed) + "</passed>");
out.println("<message>" + message + "</message>");
out.println("</response>");
out.close();
}

/**
 * Checks to see whether the argument is a valid date.
 * A null date is considered invalid. This method
 * used the default data formatter and lenient
 * parsing.
 *
 * @param date a String representing the date to check
 * @return message a String representing the outcome of the check
 */
private boolean validateDate(String date) {

    boolean isValid = true;
    if(date != null) {
        SimpleDateFormat formatter= new SimpleDateFormat("MM/dd/yyyy");
        try {
            formatter.parse(date);
        } catch (ParseException pe) {
            System.out.println(pe.toString());
            isValid = false;
        }
    } else {
        isValid = false;
    }
    return isValid;
}
}
```

运行这个例子会得到图 4-1 和图 4-2 所示的结果。

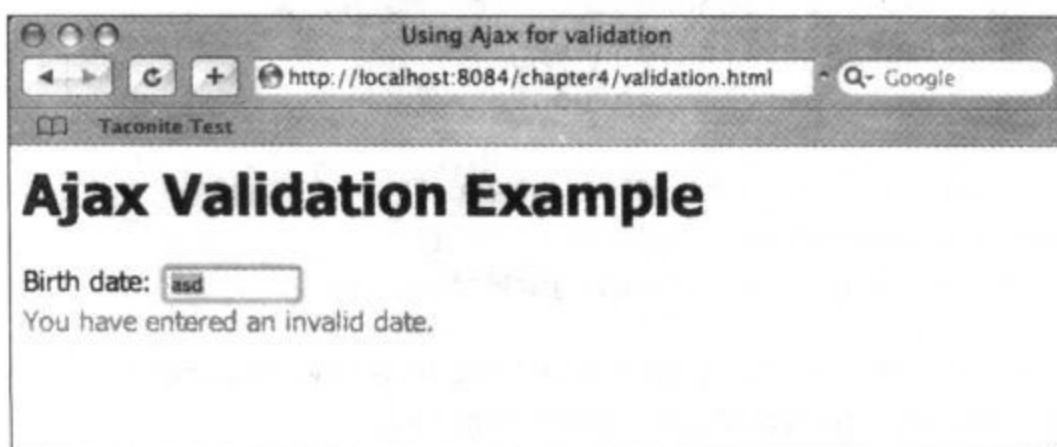


图 4-1 输入非法的日期

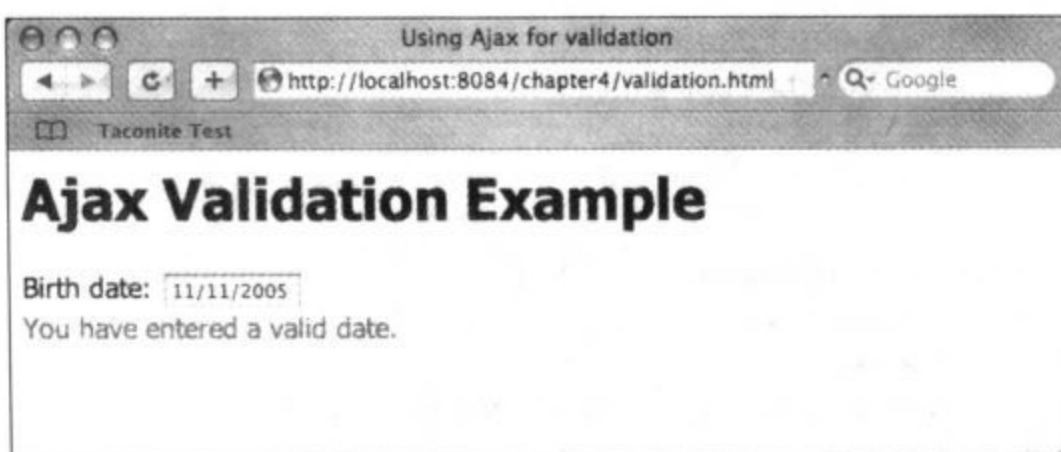


图 4-2 输入合法的日期

## 4.2 读取响应头部

你有时可能需要从服务器获取一些内容，例如，可能想“ping”一下服务器，验证服务器是否正常运行。此时，你也许只想读取服务器发出的响应头部，而忽略内容。通过读取响应头部，可以得出 Content-Type（内容类型）、Content-Length（内容长度），甚至 Last-Modified（最后一次修改）的日期。

如果只关注响应头部，完成这样一个请求的标准做法是使用 HEAD 请求，而不是前面讨论的 GET 或 POST 请求。当服务器对 HEAD 请求做出响应时，它只发送响应头部而忽略内容，即使可以向浏览器返回所请求的内容，也不会真的把内容返回。由于忽略了内容，对 HEAD 请求的响应比对 GET 或 POST 的响应就小得多。

代码清单 4-3 展示了从 XMLHttpRequest 对象获取响应头部的多种方法，并介绍了这些方法在实际中的使用。这个页面有 4 个链接，分别对应从 XMLHttpRequest 对象读取响应头部的各个方法。

### 代码清单 4-3 readingResponseHeaders.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Reading Response Headers</title>

<script type="text/javascript">
var xmlhttp;
var requestType = "";

function createXMLHttpRequest() {
    if (window.ActiveXObject) {
        xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
    }
}
```

```
else if (window.XMLHttpRequest) {
    xmlhttp = new XMLHttpRequest();
}
}

function doHeadRequest(request, url) {
    requestType = request;
    createXMLHttpRequest();
    xmlhttp.onreadystatechange = handleStateChange;
    xmlhttp.open("HEAD", url, true);
    xmlhttp.send(null);
}

function handleStateChange() {
    if(xmlhttp.readyState == 4) {
        if(requestType == "allResponseHeaders") {
            getAllResponseHeaders();
        }
        else if(requestType == "lastModified") {
            getLastModified();
        }
        else if(requestType == "isResourceAvailable") {
            getIsResourceAvailable();
        }
    }
}

function getAllResponseHeaders() {
    alert(xmlhttp.getAllResponseHeaders());
}

function getLastModified() {
    alert("Last Modified: " + xmlhttp.getResponseHeader("Last-Modified"));
}

function getIsResourceAvailable() {
    if(xmlhttp.status == 200) {
        alert("Successful response");
    }
    else if(xmlhttp.status == 404) {
        alert("Resource is unavailable");
    }
    else {
        alert("Unexpected response status: " + xmlhttp.status);
    }
}

</script>
```

```
</head>

<body>
    <h1>Reading Response Headers</h1>

    <a href="javascript:doHeadRequest('allResponseHeaders',
        'readingResponseHeaders.xml');">Read All Response Headers</a>

    <br/>
    <a href="javascript:doHeadRequest('lastModified',
        'readingResponseHeaders.xml');">Get Last Modified Date</a>

    <br/>
    <a href="javascript:doHeadRequest('isResourceAvailable',
        'readingResponseHeaders.xml');">Read Available Resource</a>

    <br/>
    <a href="javascript:doHeadRequest('isResourceAvailable',
        'not-available.xml');">Read Unavailable Resource</a>

</body>
</html>
```

页面上第一个链接展示了 XMLHttpRequest 对象的 `getAllResponseHeaders()` 方法。这个方法只是将所有响应头部获取为一个串。在这个例子中，响应头部显示在警告框中。`getAllResponseHeaders()` 方法的用途很有限，因为它把所有响应头部放在一起作为串返回。要想使用 `getAllResponseHeaders()` 方法来获取单个的响应头部，就需要解析返回的串，查找所关注的响应头部。

`getResponseHeader` 方法可以通过只返回一个响应头部的值解决这个问题。这个方法取一个串参数，该参数表示所需响应头部的名字（就是你想得到这个响应头部的值）。这个例子使用 `getResponseHeader` 方法将 `Last-Modified` 首部显示在警告框中。在实际应用中，`getResponseHeader` 方法很可能用于以某个间隔轮询服务器资源。只有自最近一次轮询服务器资源以来 `Last-Modified` 响应头部发生变化时，浏览器才会根据服务器资源更新其内容。

页面上的后两个链接使用了 XMLHttpRequest 对象的另一种能力，即检查服务器返回的 HTTP 状态码。XMLHttpRequest 对象的 `status` 属性把 HTTP 状态作为一个整数返回。如果状态码为 200，指示这是正常成功服务器响应。相反，如果状态码是 500，则指示服务器处理请求时出现了某种内部错误。

这个例子使用 HTTP 状态码来确定服务器资源是否可用。HTTP 状态码 404 指示没有所请求的资源。页面上的 `Read Available Resource`（读取可用资源）链接请求位于服务器上的简单的 XML 文件。因为服务器上有这个文件，所以 HTTP 状态码为 200，指示这是成功的。

响应。页面上最后一个链接是 Read Unavailable Resource (读取不可用资源)，它请求服务器上没有的文件。服务器会用 HTTP 状态码 404 做出响应。JavaScript 事件处理程序检查服务器响应，看到 404 状态码，显示警告框指示所请求的资源不可用。

代码清单 4-4 显示了 readingResponseHeaders.xml。

#### 代码清单 4-4 readingResponseHeaders.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<readingResponseHeaders>

</readingResponseHeaders>
```

显示所有响应首部的结果如图 4-3 所示。图 4-4 显示了读取 Last-Modified 首部的结果，图 4-5 显示了确定 Web 资源是否可用的结果。

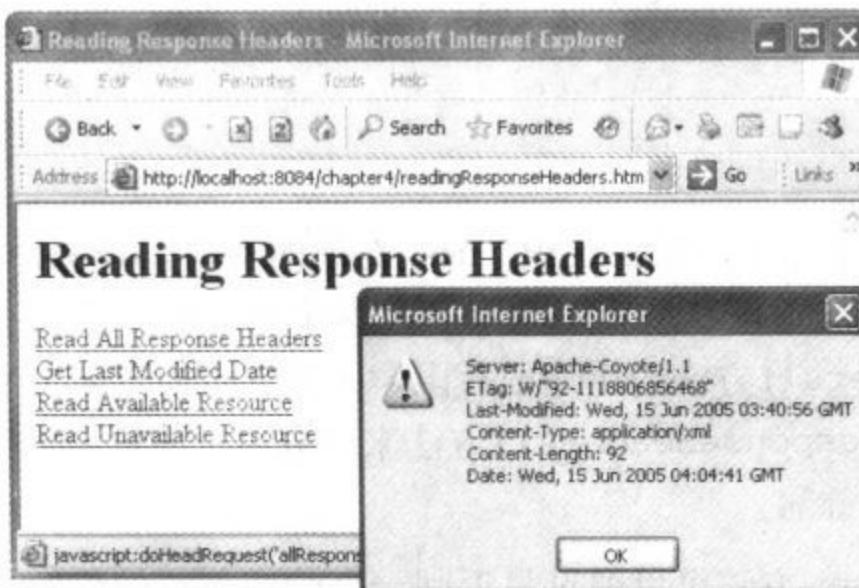


图 4-3 显示所有响应首部

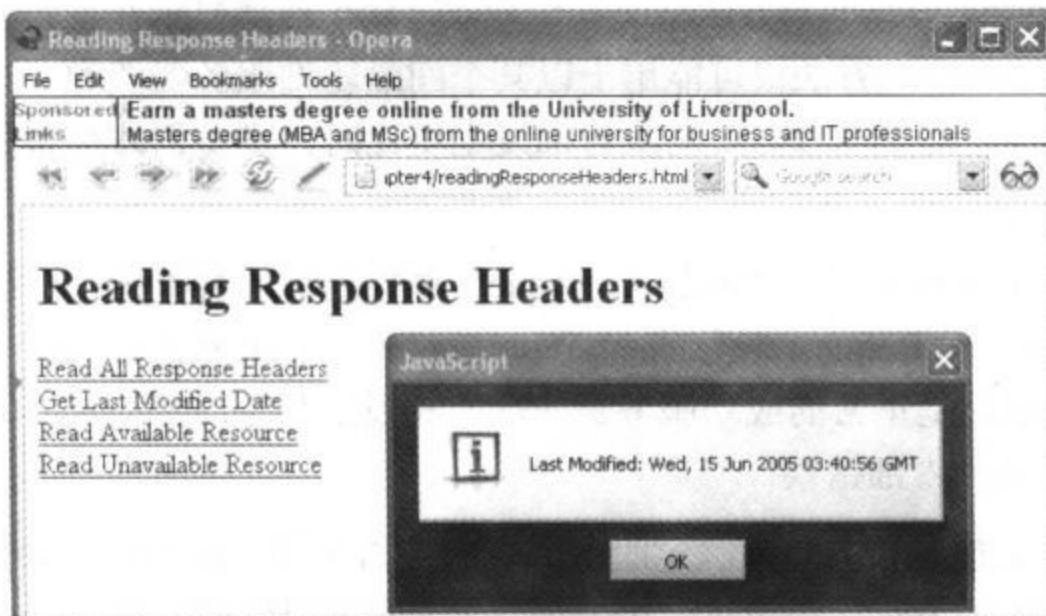


图 4-4 读取一个响应首部，在这里是 Last-Modified 首部

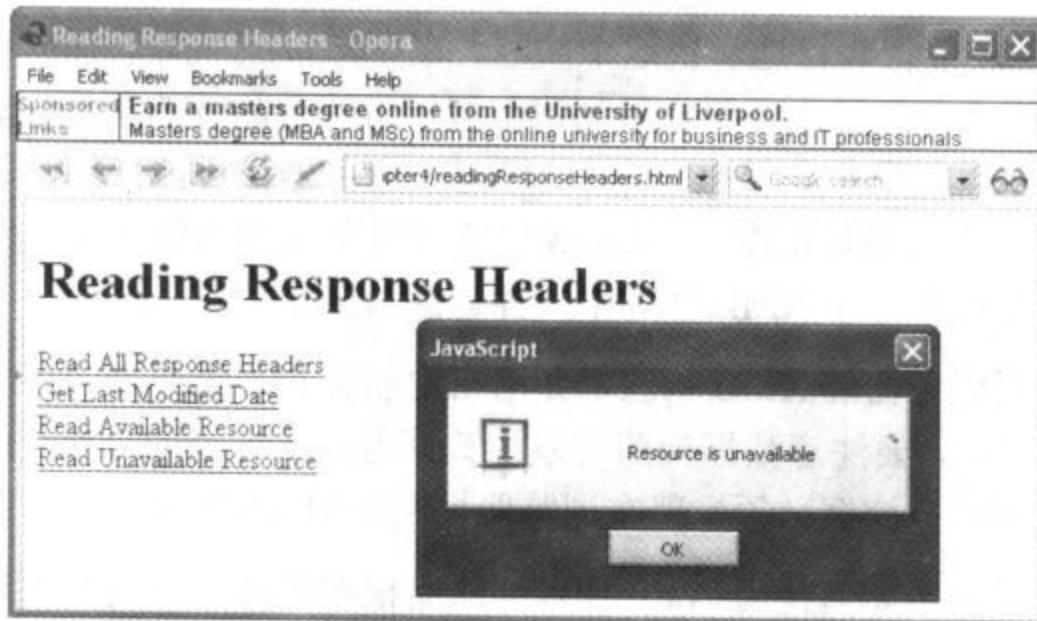


图 4-5 确定 Web 资源是否可用

### 4.3 动态加载列表框

Web 应用通常使用“向导工具”设计原则来构建，即每个屏幕要求用户输入少量的信息，每个后续页的数据都依据前一页的输入来创建。对于某些情况，这个设计模式非常有用，如用户以一种逐步、有序的方式完成任务。遗憾的是，太多的 Web 应用使用了这种方法，因为它们别无选择。在 Ajax 技术出现之前，当基于用户输入修改页面上的某些部分时，动态地更新页面而不刷新整个页面是很难办到的，甚至根本不可能。

避免完全页面刷新的一种技术是在页面上隐藏数据，并在需要时再显示它们。例如，假设选择框 B 的值要根据选择框 A 中所选值来填写，此时选择框 B 的所有可取值就可以放在隐藏的选择框中。当选择框 A 中的所选值有变化时，JavaScript 可以确定要显示哪一个隐藏的选择框，然后将该选择框置为可见，再把前一个选择框置为隐藏。这种技术还可以变化一下，用隐藏列表框中的元素动态填写选择框 B 中的 option 元素。这些技术都很有用，但是它们只在有限的情况下可用，即页面中仅限于根据用户输入对有限的选择进行修改，而且这样的选择必须相对少。

假设你在构建一个在线的汽车分类广告服务。某人想购买汽车，指定了车型年份、品牌和车型，来搜索他想买的汽车。为了避免用户的输入错误，并减少所需的动态验证次数，你决定车型年份、品牌和车型输入字段都应当是选择框，而且要考虑过去 25 年的车型广告。如果车型年份选择框或品牌选择框中的选择发生变化，就必须修改对应该车型年份和品牌的可用车型列表。

要记住，对于每个车型年份，都会出现一些新的品牌，而一些老牌子可能会淡出人们的视线，所以其个数也会有变化。还要记住对于每种品牌来说，每年的车型都可能不同。如果有数十种品牌，每个车型年份每种品牌都有多种车型，那么车型年份、品牌和车型的组合数将是惊人的。由于有这么多的组合，只使用 JavaScript 来填写选择框是不可能的。

使用 Ajax 技术就能很轻松地解决这个问题。车型年份或品牌选择框中的选择每次有变化时，会向服务器发出异步请求，要求得到该车型年份特定品牌的车型列表。服务器负责根据浏览器所请求的品牌和车型年份来确定车型列表。服务器很可能采用一种高速的数据查找组件（可能实现为一个关系数据库），以完成查找可用车型的具体工作。一旦找到可用的车型，服务器把它们打包在一个 XML 文件中，并返回给浏览器。

浏览器负责解析服务器的 XML 响应，并用指定品牌和车型年份的可用车型来填写车型选择框。在这个例子中，要注意数据视图与原始数据得到了很好的分离。浏览器只负责呈现数据视图，服务器则负责挖掘必须呈现在浏览器视图上的原始数据。

代码清单 4-5 展示了如何使用 Ajax 技术，从而根据另外两个列表框的值动态创建一个选择框的内容。这个例子的用例就是以上所述的分类广告服务，在此车型年份选择框和品牌选择框中的所选值决定了车型选择框中的内容。这个例子中只用了 4 个车型年份、3 种品牌，以及对于某个车型年份、特定的品牌的 4 种可用车型。即便如此，车型年份、品牌和车型的组合数也达到了 48。如果采用隐藏的办法，即对应每个车型年份和品牌组件，将相应的车型列表隐藏起来，并根据所选的品牌和车型年份值来显示适当的列表，这是不可行的。

#### 代码清单 4-5 dynamicLists.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Dynamically Filling Lists</title>

<script type="text/javascript">
var xmlhttp;

function createXMLHttpRequest() {
    if (window.ActiveXObject) {
        xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
    }
    else if (window.XMLHttpRequest) {
        xmlhttp = new XMLHttpRequest();
    }
}

function refreshModelList() {
    var make = document.getElementById("make").value;
    var modelYear = document.getElementById("modelYear").value;

    if(make == "" || modelYear == "") {
        clearModelsList();
    }
    else {
        xmlhttp.open("GET", "models.xml?make=" + make +
                    "&modelYear=" + modelYear, true);
        xmlhttp.onreadystatechange = function() {
            if (xmlhttp.readyState == 4 && xmlhttp.status == 200) {
                var modelsList = document.getElementById("modelsList");
                modelsList.innerHTML = xmlhttp.responseText;
            }
        }
    }
}

function clearModelsList() {
    var modelsList = document.getElementById("modelsList");
    modelsList.innerHTML = "";
}
```

```
        return;
    }

    var url = "RefreshModelList?"
        + createQueryString(make, modelYear) + "&ts=" + new Date().getTime();

    createXMLHttpRequest();
    xmlhttp.onreadystatechange = handleStateChange;
    xmlhttp.open("GET", url, true);
    xmlhttp.send(null);
}

function createQueryString(make, modelYear) {
    var queryString = "make=" + make + "&modelYear=" + modelYear;
    return queryString;
}

function handleStateChange() {
    if(xmlhttp.readyState == 4) {
        if(xmlhttp.status == 200) {
            updateModelsList();
        }
    }
}

function updateModelsList() {
    clearModelsList();

    var models = document.getElementById("models");
    var results = xmlhttp.responseXML.getElementsByTagName("model");
    var option = null;
    for(var i = 0; i < results.length; i++) {
        option = document.createElement("option");
        option.appendChild
            (document.createTextNode(results[i].firstChild.nodeValue));
        models.appendChild(option);
    }
}

function clearModelsList() {
    var models = document.getElementById("models");
    while(models.childNodes.length > 0) {
        models.removeChild(models.childNodes[0]);
    }
}
</script>
</head>
```

```
<body>
    <h1>Select Model Year and Make</h1>

    <form action="#">
        <span style="font-weight:bold;">Model Year:</span>
        <select id="modelYear" onchange="refreshModelList();">
            <option value="">Select One</option>
            <option value="2006">2006</option>
            <option value="1995">1995</option>
            <option value="1985">1985</option>
            <option value="1970">1970</option>
        </select>
        <br/><br/>
        <span style="font-weight:bold;">Make:</span>
        <select id="make" onchange="refreshModelList();">
            <option value="">Select One</option>
            <option value="Chevrolet">Chevrolet</option>
            <option value="Dodge">Dodge</option>
            <option value="Pontiac">Pontiac</option>
        </select>

        <br/><br/>
        <span style="font-weight:bold;">Models:</span>
        <br/>
        <select id="models" size="6" style="width:300px;">
        </select>
    </form>

</body>
</html>
```

页面的更新由品牌和车型年份选择框的 `onchange` 事件驱动。只要这两个选择框中任何一个的所选值有变化，浏览器就会向服务器发出异步请求。发送请求时会携带一个查询串，其中包含所选品牌和车型年份的值。

`RefreshModelList` servlet 从浏览器接收到请求，并确定对应指定品牌和车型年份的车型列表。这个 servlet 首先解析查询串，确定所请求的品牌和车型年份。一旦确定了品牌和车型年份，servlet 会迭代处理一个对象集合，其中每个对象分别表示一种车型年份、品牌和车型的组合。如果特定对象的车型年份和品牌属性与所请求的车型年份和品牌匹配，则把这个对象的车型属性增加到响应 XML 串中。找到对应指定品牌和车型年份的所有车型之后，将响应 XML 写回到浏览器。

请注意，在实际实现中，服务器端组件不太可能依赖硬编码的值填写选择框，而是会搜索一个高速数据库，查找所请求车型年份和品牌的相应车型。

代码清单 4-6 显示了 RefreshModelListServlet.java。

**代码清单 4-6 RefreshModelListServlet.java**

```
package ajaxbook.chap4;

import java.io.*;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import javax.servlet.*;
import javax.servlet.http.*;

public class RefreshModelListServlet extends HttpServlet {

    private static List availableModels = new ArrayList();

    protected void processRequest(HttpServletRequest request
                                  , HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");

        int modelYear = Integer.parseInt(request.getParameter("modelYear"));
        String make = request.getParameter("make");

        StringBuffer results = new StringBuffer("<models>");
        MakeModelYear availableModel = null;
        for(Iterator it = availableModels.iterator(); it.hasNext();) {
            availableModel = (MakeModelYear)it.next();
            if(availableModel.modelYear == modelYear) {
                if(availableModel.make.equals(make)) {
                    results.append("<model>");
                    results.append(availableModel.model);
                    results.append("</model>");
                }
            }
        }
        results.append("</models>");

        response.setContentType("text/xml");
        response.getWriter().write(results.toString());
    }
}
```

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    processRequest(request, response);
}

public void init() throws ServletException {
    availableModels.add(new MakeModelYear(2006, "Dodge", "Charger"));
    availableModels.add(new MakeModelYear(2006, "Dodge", "Magnum"));
    availableModels.add(new MakeModelYear(2006, "Dodge", "Ram"));
    availableModels.add(new MakeModelYear(2006, "Dodge", "Viper"));
    availableModels.add(new MakeModelYear(1995, "Dodge", "Avenger"));
    availableModels.add(new MakeModelYear(1995, "Dodge", "Intrepid"));
    availableModels.add(new MakeModelYear(1995, "Dodge", "Neon"));
    availableModels.add(new MakeModelYear(1995, "Dodge", "Spirit"));
    availableModels.add(new MakeModelYear(1985, "Dodge", "Aries"));
    availableModels.add(new MakeModelYear(1985, "Dodge", "Daytona"));
    availableModels.add(new MakeModelYear(1985, "Dodge", "Diplomat"));
    availableModels.add(new MakeModelYear(1985, "Dodge", "Omni"));
    availableModels.add(new MakeModelYear(1970, "Dodge", "Challenger"));
    availableModels.add(new MakeModelYear(1970, "Dodge", "Charger"));
    availableModels.add(new MakeModelYear(1970, "Dodge", "Coronet"));
    availableModels.add(new MakeModelYear(1970, "Dodge", "Dart"));

    availableModels.add(new MakeModelYear(2006, "Chevrolet", "Colorado"));
    availableModels.add(new MakeModelYear(2006, "Chevrolet", "Corvette"));
    availableModels.add(new MakeModelYear(2006, "Chevrolet", "Equinox"));
    availableModels.add(new MakeModelYear(2006, "Chevrolet", "Monte Carlo"));
    availableModels.add(new MakeModelYear(1995, "Chevrolet", "Beretta"));
    availableModels.add(new MakeModelYear(1995, "Chevrolet", "Camaro"));
    availableModels.add(new MakeModelYear(1995, "Chevrolet", "Cavalier"));
    availableModels.add(new MakeModelYear(1995, "Chevrolet", "Lumina"));
    availableModels.add(new MakeModelYear(1985, "Chevrolet", "Cavalier"));
    availableModels.add(new MakeModelYear(1985, "Chevrolet", "Chevette"));
    availableModels.add(new MakeModelYear(1985, "Chevrolet", "Celebrity"));
    availableModels.add(new MakeModelYear(1985, "Chevrolet", "Citation II"));
    availableModels.add(new MakeModelYear(1970, "Chevrolet", "Bel Air"));
    availableModels.add(new MakeModelYear(1970, "Chevrolet", "Caprice"));
    availableModels.add(new MakeModelYear(1970, "Chevrolet", "Chevelle"));
    availableModels.add(new MakeModelYear(1970, "Chevrolet", "Monte Carlo"));

    availableModels.add(new MakeModelYear(2006, "Pontiac", "G6"));
    availableModels.add(new MakeModelYear(2006, "Pontiac", "Grand Prix"));
    availableModels.add(new MakeModelYear(2006, "Pontiac", "Solstice"));
    availableModels.add(new MakeModelYear(2006, "Pontiac", "Vibe"));
    availableModels.add(new MakeModelYear(1995, "Pontiac", "Bonneville"));
    availableModels.add(new MakeModelYear(1995, "Pontiac", "Grand Am"));
    availableModels.add(new MakeModelYear(1995, "Pontiac", "Grand Prix"));
```

```

availableModels.add(new MakeModelYear(1995, "Pontiac", "Firebird"));
availableModels.add(new MakeModelYear(1985, "Pontiac", "6000"));
availableModels.add(new MakeModelYear(1985, "Pontiac", "Fiero"));
availableModels.add(new MakeModelYear(1985, "Pontiac", "Grand Prix"));
availableModels.add(new MakeModelYear(1985, "Pontiac", "Parisienne"));
availableModels.add(new MakeModelYear(1970, "Pontiac", "Catalina"));
availableModels.add(new MakeModelYear(1970, "Pontiac", "GTO"));
availableModels.add(new MakeModelYear(1970, "Pontiac", "LeMans"));
availableModels.add(new MakeModelYear(1970, "Pontiac", "Tempest"));
}

private static class MakeModelYear {
    private int modelYear;
    private String make;
    private String model;

    public MakeModelYear(int modelYear, String make, String model) {
        this.modelYear = modelYear;
        this.make = make;
        this.model = model;
    }
}
}

```

如图 4-6 所示，在任何一个选择框中选择不同的值，就会更新车型列表。

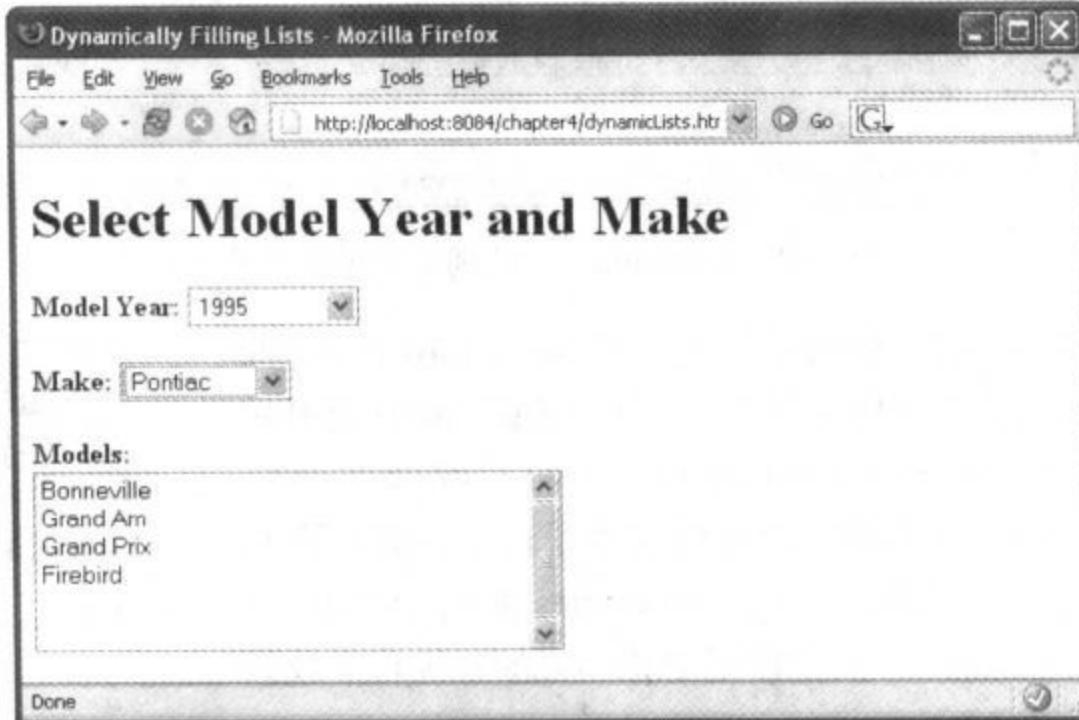


图 4-6 在任何一个选择框中选择不同的值都会更新车型列表

## 4.4 创建自动刷新页面

股票行情、天气数据、标题新闻……这些都是经常改变的数据，但不值得为这些数据的

修改手工地完全刷新页面。尽管 CNN.com 之类的网站确实会定期重新加载，但是，如果只是为了改变一两个标题新闻和几个图就重绘整个页面，这可能很让人扫兴。当然，如果刷新整个页面，可能很难发现到底哪些是新内容！

如果使用 Ajax，用户就不用反复点击 refresh(刷新)按钮。技术新闻网站 Digg (<http://digg.com/spy>) 就使用了这种技术。Digg 采用自动刷新方法不断更新其页面，并使用了很有帮助的褪色技术，以可视化的方式让用户知道哪些新闻是新的（见图 4-7）。

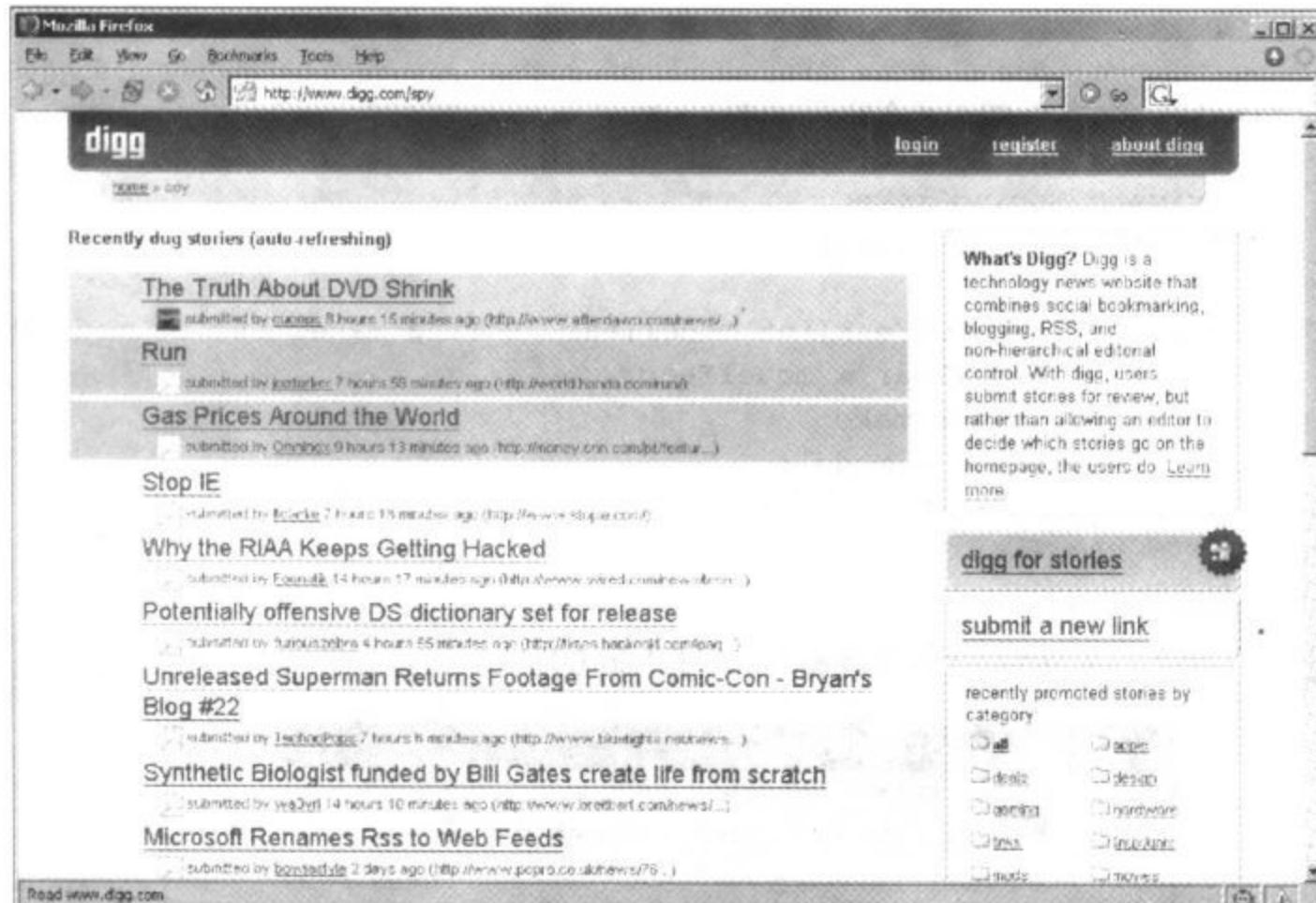


图 4-7 Digg.com，自动刷新页面的例子

如果查看 Apple 的新闻，你可能已经看到 Steve Jobs 在 Apple 的 2005 世界开发人员大会上的主题演讲，其中透露 Apple 将开始转向 Intel 处理器。MacRumors.com (<http://www.macrumors.com/>) 的小组使用 Ajax 技术相当及时地发布了这个信息，而且减轻了其服务器的压力。最近，Apple 的 iTunes 网站 (<http://www.apple.com/itunes/>) 正在使用 Ajax 动态更新其下载数（目标是 5 亿）（见图 4-8）。

自动刷新页面实际上相当简单。对于代码清单 4-7 所示的例子，使用一个按钮开始“轮询”，不过在实际应用中，可能会以 onload 事件代之。doStart() 方法负责启动，不过最有意思的地方是 pollCallback() 方法中的 setTimeout() 方法，它允许以固定的时间间隔（单位是毫秒）执行给定的方法。createRow() 方法只是一个充分利用了 DOM 方法来动态创建



图 4-8 Apple iTunes 采用的基于 Ajax 的计数器

内容的辅助函数，refreshTime()用于刷新定时器值。

#### 代码清单 4-7 dynamicUpdate.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<html>
  <head>
    <title>Ajax Dynamic Update</title>
    <script type="text/javascript">
      var xmlhttp;

      function createXMLHttpRequest() {
        if (window.ActiveXObject) {
          xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
        }
        else if (window.XMLHttpRequest) {
          xmlhttp = new XMLHttpRequest();
        }
      }

      function doStart() {
        createXMLHttpRequest();
        var url = "DynamicUpdateServlet?task=reset";
        xmlhttp.open("GET", url, true);
        xmlhttp.onreadystatechange = startCallback;
        xmlhttp.send(null);
      }

      function startCallback() {
        if (xmlhttp.readyState == 4) {
          if (xmlhttp.status == 200) {
            setTimeout("pollServer()", 5000);
            refreshTime();
          }
        }
      }

      function pollServer() {
        createXMLHttpRequest();
        var url = "DynamicUpdateServlet?task=foo";
        xmlhttp.open("GET", url, true);
        xmlhttp.onreadystatechange = pollCallback;
        xmlhttp.send(null);
      }
    </script>
  </head>
  <body>
    <h1>Ajax Dynamic Update</h1>
    <p>This page demonstrates how to use XMLHttpRequest<br/>to dynamically update content on a web page without<br/>refreshing the entire page.</p>
  </body>
</html>
```

```
function refreshTime(){
    var time_span = document.getElementById("time");
    var time_val = time_span.innerHTML;
    var int_val = parseInt(time_val);
    var new_int_val = int_val - 1;

    if (new_int_val > -1) {
        setTimeout("refreshTime()", 1000);
        time_span.innerHTML = new_int_val;
    } else {
        time_span.innerHTML = 5;
    }
}

function pollCallback() {
    if (xmlHttp.readyState == 4) {
        if (xmlHttp.status == 200) {
            var message =
                xmlHttp.responseXML
                    .getElementsByName("message")[0].firstChild.data;

            if (message != "done") {
                var new_row = createRow(message);
                var table = document.getElementById("dynamicUpdateArea");
                var table_body =
                    table.getElementsByTagName("tbody").item(0);
                var first_row =
                    table_body.getElementsByTagName("tr").item(1);
                table_body.insertBefore(new_row, first_row);
                setTimeout("pollServer()", 5000);
                refreshTime();
            }
        }
    }
}

function createRow(message) {
    var row = document.createElement("tr");
    var cell = document.createElement("td");
    var cell_data = document.createTextNode(message);
    cell.appendChild(cell_data);
    row.appendChild(cell);
    return row;
}

</script>
</head>
<body>
```

```
<h1>Ajax Dynamic Update Example</h1>
This page will automatically update itself:
<input type="button" value="Launch" id="go" onclick="doStart();"/>
<p>
Page will refresh in <span id="time">5</span> seconds.
<p>
<table id="dynamicUpdateArea" align="left">
<tbody>
<tr id="row0"><td></td></tr>
</tbody>
</table>
</body>
</html>
```

服务器代码相当简单，它只是根据简单的计数器返回一组信息（见代码清单 4-8）。

#### 代码清单 4-8 DynamicUpdateServlet.java

```
package ajaxbook.chap4;

import java.io.*;
import java.net.*;

import javax.servlet.*;
import javax.servlet.http.*;

public class DynamicUpdateServlet extends HttpServlet {
    private int counter = 1;

    /**
     * Handles the HTTP <code>GET</code> method.
     * @param request servlet request
     * @param response servlet response
     */
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        String res = "";
        String task = request.getParameter("task");
        String message = "";

        if (task.equals("reset")) {
            counter = 1;
        } else {
            switch (counter) {
                case 1: message = "Steve walks on stage"; break;
                case 2: message = "iPods rock"; break;
                case 3: message = "Steve says Macs rule"; break;
                case 4: message = "Change is coming"; break;
            }
        }
        response.setContentType("text/html");
        response.getWriter().println(res);
    }
}
```

```

        case 5: message = "Yes, OS X runs on Intel - has for years"; break;
        case 6: message = "Macs will soon have Intel chips"; break;
        case 7: message = "done"; break;
    }
    counter++;
}

res = "<message>" + message + "</message>";

PrintWriter out = response.getWriter();
response.setContentType("text/xml");
response.setHeader("Cache-Control", "no-cache");
out.println("<response>");
out.println(res);
out.println("</response>");
out.close();
}
}

```

图 4-9 显示了浏览器中看到的这个动态更新例子。

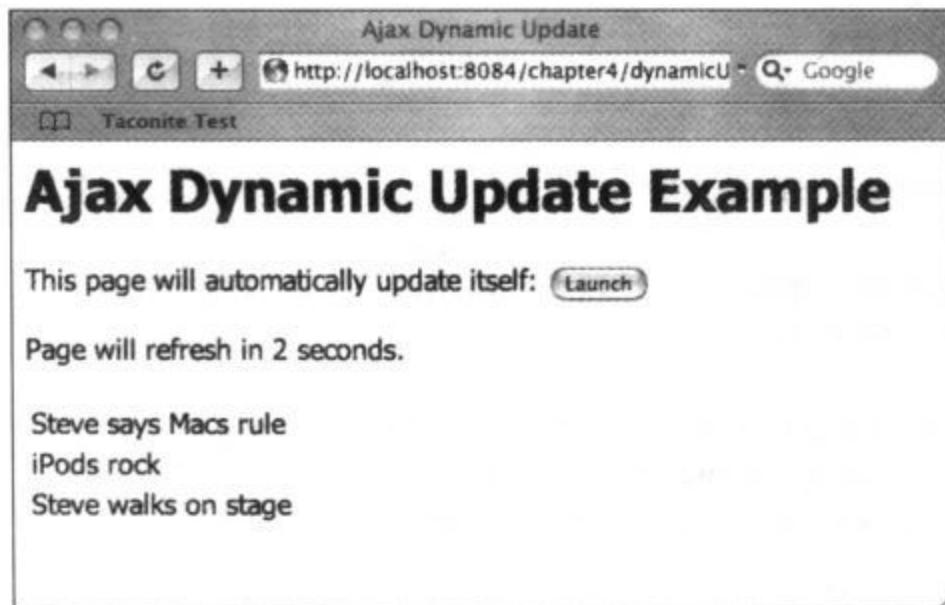


图 4-9 动态更新例子

## 4.5 显示进度条

无一例外地，几乎每个应用都会时不时地调用一个长时间运行的事务。如果你关心系统的可用性，就要确保用户能很容易地看到系统的状态。如果是一个胖客户应用，对于长时间运行事务的问题，解决办法很简单：只需显示一个进度条，以便用户知道目前所处状况。不过，在 Ajax 之前，要在 Web 应用中做到这一点很不容易。本节将使你了解如何使用 Ajax 为 Web 应用建立进度条。

在代码清单 4-9 所示的例子中，再次在 `pollCallback()` 方法中使用了 `setTimeout()`，

从而每隔2秒调用一次服务器。在processResult()方法中，只是从服务器查找已完成比例（百分数）的第一位数字，从而得出要将进度条中的哪些进度块着色（灰色）。

#### 代码清单4-9 progressBar.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<html>
  <head>
    <title>Ajax Progress Bar</title>
    <script type="text/javascript">
      var xmlhttp;
      var key;
      var bar_color = 'gray';
      var span_id = "block";
      var clear = "      "

      function createXMLHttpRequest() {
        if (window.ActiveXObject) {
          xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
        }
        else if (window.XMLHttpRequest) {
          xmlhttp = new XMLHttpRequest();
        }
      }

      function go() {
        createXMLHttpRequest();
        checkDiv();
        var url = "ProgressBarServlet?task=create";
        var button = document.getElementById("go");
        button.disabled = true;
        xmlhttp.open("GET", url, true);
        xmlhttp.onreadystatechange = goCallback;
        xmlhttp.send(null);
      }

      function goCallback() {
        if (xmlhttp.readyState == 4) {
          if (xmlhttp.status == 200) {
            setTimeout("pollServer()", 2000);
          }
        }
      }

      function pollServer() {
        createXMLHttpRequest();
```

```
var url = "ProgressBarServlet?task=poll&key=" + key;
xmlHttp.open("GET", url, true);
xmlHttp.onreadystatechange = pollCallback;
xmlHttp.send(null);
}

function pollCallback() {
    if (xmlHttp.readyState == 4) {
        if (xmlHttp.status == 200) {
            var percent_complete =
                xmlHttp.responseXML
                    .getElementsByTagName("percent")[0].firstChild.data;

            var index = processResult(percent_complete);
            for (var i = 1; i <= index; i++) {
                var elem = document.getElementById("block" + i);
                elem.innerHTML = clear;

                elem.style.backgroundColor = bar_color;
                var next_cell = i + 1;
                if (next_cell > index && next_cell <= 9) {
                    document.getElementById("block" + next_cell)
                        .innerHTML =
                            percent_complete + "%";
                }
            }
            if (index < 9) {
                setTimeout("pollServer()", 2000);
            } else {
                document.getElementById("complete").innerHTML = "Complete!";
                document.getElementById("go").disabled = false;
            }
        }
    }
}

function processResult(percent_complete) {
    var ind;
    if (percent_complete.length == 1) {
        ind = 1;
    } else if (percent_complete.length == 2) {
        ind = percent_complete.substring(0, 1);
    } else {
        ind = 9;
    }
    return ind;
}
```

```
function checkDiv() {
    var progress_bar = document.getElementById("progressBar");
    if (progress_bar.style.visibility == "visible") {
        clearBar();
        document.getElementById("complete").innerHTML = "";
    } else {
        progress_bar.style.visibility = "visible"
    }
}

function clearBar() {
    for (var i = 1; i < 10; i++) {
        var elem = document.getElementById("block" + i);
        elem.innerHTML = clear;
        elem.style.backgroundColor = "white";
    }
}

</script>
</head>
<body>
    <h1>Ajax Progress Bar Example</h1>
    Launch long-running process:
        <input type="button" value="Launch" id="go" onclick="go();"/>
    <p>
        <table align="center">
            <tbody>
                <tr><td>
                    <div id="progressBar"
                        style="padding:2px; border:solid black 2px; visibility:hidden">
                        <span id="block1">&ampnbsp&ampnbsp&ampnbsp</span>
                        <span id="block2">&ampnbsp&ampnbsp&ampnbsp</span>
                        <span id="block3">&ampnbsp&ampnbsp&ampnbsp</span>
                        <span id="block4">&ampnbsp&ampnbsp&ampnbsp</span>
                        <span id="block5">&ampnbsp&ampnbsp&ampnbsp</span>
                        <span id="block6">&ampnbsp&ampnbsp&ampnbsp</span>
                        <span id="block7">&ampnbsp&ampnbsp&ampnbsp</span>
                        <span id="block8">&ampnbsp&ampnbsp&ampnbsp</span>
                        <span id="block9">&ampnbsp&ampnbsp&ampnbsp</span>
                    </div>
                </td></tr>
                <tr><td align="center" id="complete"></td></tr>
            </tbody>
        </table>
    </body>
</html>
```

这个例子的服务器代码“模拟”了一个长时间运行的事务（见代码清单 4-10）。在真实环境中，这期间可能还要同时创建多个新实例并注册，之前还需要客户发出请求。为简单起见，我们忽略了这点，也没有编写线程代码。

#### 代码清单 4-10 ProgressBarServlet.java

```
package ajaxbook.chap4;

import java.io.*;

import javax.servlet.*;
import javax.servlet.http.*;

public class ProgressBarServlet extends HttpServlet {
    private int counter = 1;

    /** Handles the HTTP <code>GET</code> method.
     * @param request servlet request
     * @param response servlet response
     */
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        String task = request.getParameter("task");
        String res = "";

        if (task.equals("create")) {
            res = "<key>1</key>";
            counter = 1;
        }
        else {
            String percent = "";
            switch (counter) {
                case 1: percent = "10"; break;
                case 2: percent = "23"; break;
                case 3: percent = "35"; break;
                case 4: percent = "51"; break;
                case 5: percent = "64"; break;
                case 6: percent = "73"; break;
                case 7: percent = "89"; break;
                case 8: percent = "100"; break;
            }
            counter++;
            res = "<percent>" + percent + "</percent>";
        }
        PrintWriter out = response.getWriter();
    }
}
```

```
response.setContentType("text/xml");
response.setHeader("Cache-Control", "no-cache");
out.println("<response>");
out.println(res);
out.println("</response>");
out.close();
}
}
```

图 4-10 显示了实际工作中的进度条，图 4-11 显示了完成时的情况。

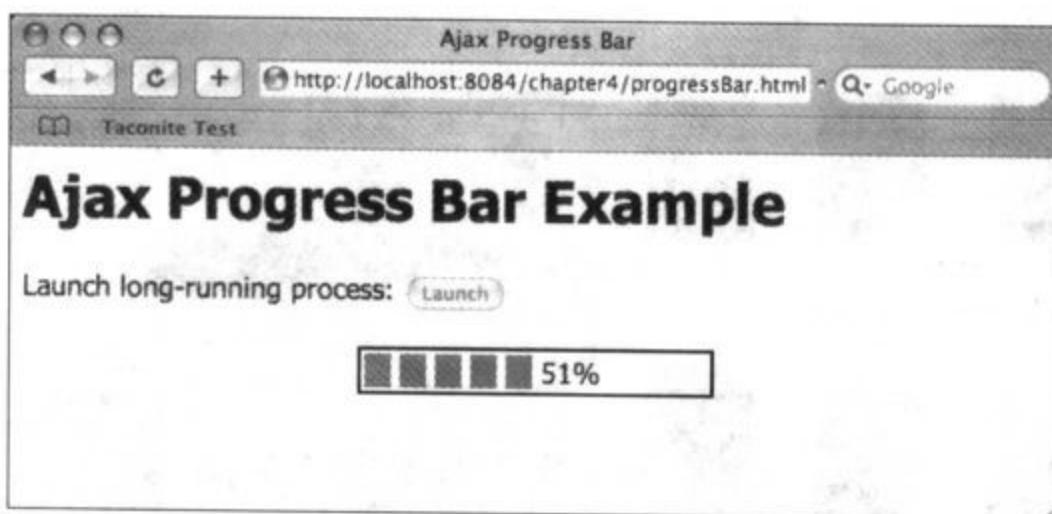


图 4-10 进度条示例

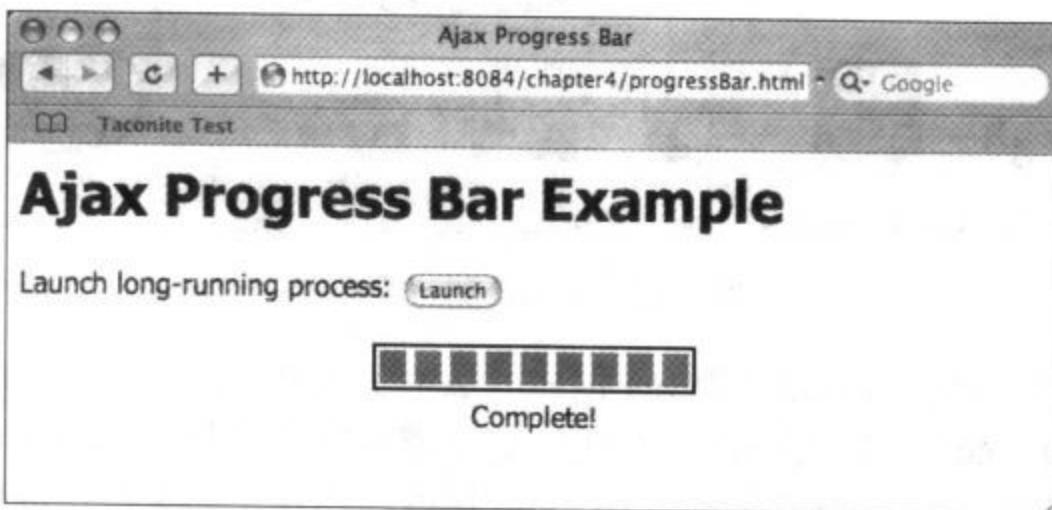


图 4-11 进度条完成

## 4.6 创建工具提示

我们见过许多使用 Ajax 的应用，到目前为止，我们认为最有意思的是 DVD 租借服务 Netflix。当浏览 Netflix 中的各个选择时，会看到各类最新影片的相关图片和文字。当把鼠标停在一个给定影片的图片上时，就会看到更多的信息（见图 4-12）。尽管不使用 Ajax 也能达到这个效果，但第一次获取页面时要纳入大量可能永远也不会用到的信息。通过使用 Ajax，只会在需要时发送所需的信息。

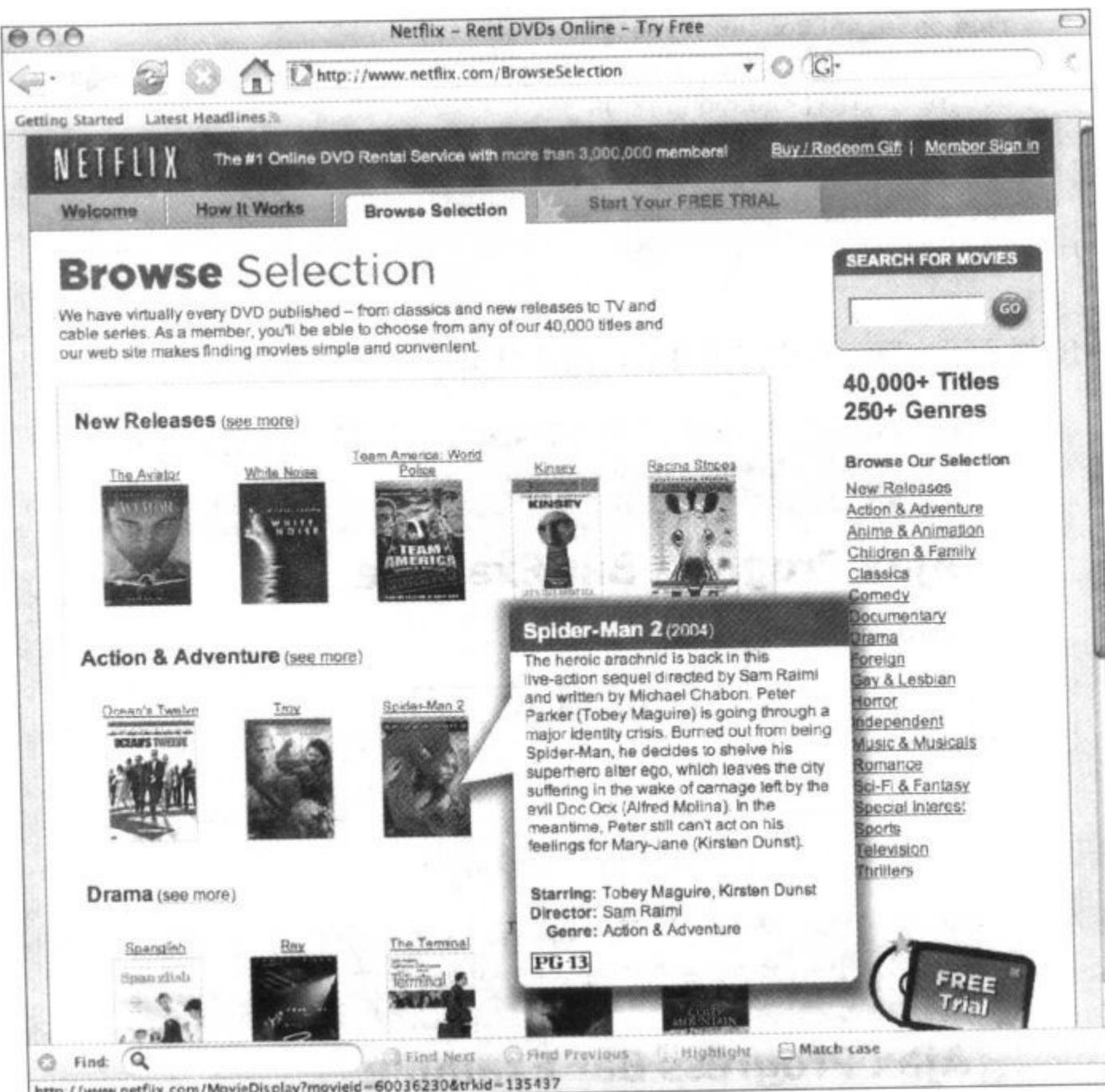


图 4-12 Netflix 浏览器特性

虽然我们的例子做得没有这么漂亮，但你能从中了解到如何提供自己的动态工具提示信息。客户端代码相当简单（见代码清单 4-11）。这里最有意思的是 `calculateOffset()` 方法。在理想情况下，可以依赖于当前元素的 `offset` 属性。不过，如果要跨浏览器，这样做不一定可行，不同浏览器上的偏移量可能不同。但是，你可以访问 DOM 来生成一个准确的偏移量，并使用这个偏移量来放置动态内容。这个例子中有一个简单的表，其中包含著名的高尔夫球场，当用户把鼠标停在表中某个单元格上时，会显示一些额外的信息。

#### 代码清单 4-11 tooltip.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<html>
<head>
```

```
<title>Ajax Tool Tip</title>
<script type="text/javascript">
    var xmlhttp;
    var dataDiv;
    var dataTable;
    var dataTableBody;
    var offsetEl;

    function createXMLHttpRequest() {
        if (window.ActiveXObject) {
            xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
        }
        else if (window.XMLHttpRequest) {
            xmlhttp = new XMLHttpRequest();
        }
    }

    function initVars() {
        dataTableBody = document.getElementById("courseDataBody");
        dataTable = document.getElementById("courseData");
        dataDiv = document.getElementById("popup");
    }

    function getCourseData(element) {
        initVars();
        createXMLHttpRequest();
        offsetEl = element;
        var url = "ToolTipServlet?key=" + escape(element.id);

        xmlhttp.open("GET", url, true);
        xmlhttp.onreadystatechange = callback;
        xmlhttp.send(null);
    }

    function callback() {
        if (xmlhttp.readyState == 4) {
            if (xmlhttp.status == 200) {
                setData(xmlhttp.responseXML);
            }
        }
    }

    function setData(courseData) {
        clearData();
        setOffsets();
        var length =
            courseData.getElementsByTagName("length")[0].firstChild.data;
```

```
var par = courseData.getElementsByTagName("par")[0].firstChild.data;

var row, row2;
var parData = "Par: " + par
var lengthData = "Length: " + length;

row = createRow(parData);
row2 = createRow(lengthData);

dataTableBody.appendChild(row);
dataTableBody.appendChild(row2);

}

function createRow(data) {
    var row, cell, txtNode;
    row = document.createElement("tr");
    cell = document.createElement("td");

    cell.setAttribute("bgcolor", "#FFFAFA");
    cell.setAttribute("border", "0");

    txtNode = document.createTextNode(data);
    cell.appendChild(txtNode);
    row.appendChild(cell);

    return row;
}

function setOffsets() {
    var end = offsetEl.offsetWidth;
    var top = calculateOffsetTop(offsetEl);
    dataDiv.style.border = "black 1px solid";
    dataDiv.style.left = end + 15 + "px";
    dataDiv.style.top = top + "px";
}

function calculateOffsetTop(field) {
    return calculateOffset(field, "offsetTop");
}

function calculateOffset(field, attr) {
    var offset = 0;
    while(field) {
        offset += field[attr];
        field = field.offsetParent;
    }
    return offset;
}
```

```

        }

        function clearData() {
            var ind = dataTableBody.childNodes.length;
            for (var i = ind - 1; i >= 0 ; i--) {
                dataTableBody.removeChild(dataTableBody.childNodes[i]);
            }
            dataDiv.style.border = "none";
        }
    </script>
</head>
<body>
    <h1>Ajax Tool Tip Example</h1>
    <h3>Golf Courses</h3>
    <table id="courses" bgcolor="#FFFFAF" border="1"
           cellspacing="0" cellpadding="2">
        <tbody>
            <tr><td id="1" onmouseover="getCourseData(this);"
                   onmouseout="clearData();">Augusta National</td></tr>
            <tr><td id="2" onmouseover="getCourseData(this);"
                   onmouseout="clearData();">Pinehurst No. 2</td></tr>
            <tr><td id="3" onmouseover="getCourseData(this);"
                   onmouseout="clearData();">
                St. Andrews Links</td></tr>
            <tr><td id="4" onmouseover="getCourseData(this);"
                   onmouseout="clearData();">Baltusrol Golf Club</td></tr>
        </tbody>
    </table>
    <div style="position:absolute;" id="popup">
        <table id="courseData" bgcolor="#FFFFAF" border="0"
               cellspacing="2" cellpadding="2">
            <tbody id="courseDataBody"></tbody>
        </table>
    </div>

    </body>
</html>

```

要记住，在生产环境中，可能会从某种数据库获取额外的信息，而且 servlet 中可能不会有内部类！代码清单 4-12 显示了 ToolTipServlet.java。

#### 代码清单 4-12 ToolTipServlet.java

```

package ajaxbook.chap4;

import java.io.*;

```

```
import java.util.HashMap;
import java.util.Map;

import javax.servlet.*;
import javax.servlet.http.*;

public class ToolTipServlet extends HttpServlet {

    private Map courses = new HashMap();

    public void init(ServletConfig config) throws ServletException {
        CourseData augusta = new CourseData(72, 7290);
        CourseData pinehurst = new CourseData(70, 7214);
        CourseData standrews = new CourseData(72, 6566);
        CourseData baltusrol = new CourseData(70, 7392);
        courses.put(new Integer(1), augusta);
        courses.put(new Integer(2), pinehurst);
        courses.put(new Integer(3), standrews);
        courses.put(new Integer(4), baltusrol);
    }

    /**
     * Handles the HTTP <code>GET</code> method.
     * @param request servlet request
     * @param response servlet response
     */
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        Integer key = Integer.valueOf(request.getParameter("key"));
        CourseData data = (CourseData) courses.get(key);

        PrintWriter out = response.getWriter();

        response.setContentType("text/xml");
        response.setHeader("Cache-Control", "no-cache");

        out.println("<response>");
        out.println("<par>" + data.getPar() + "</par>");
        out.println("<length>" + data.getLength() + "</length>");
        out.println("</response>");
        out.close();
    }

    private class CourseData {
        private int par;
        private int length;

        public CourseData(int par, int length) {
```

```

        this.par = par;
        this.length = length;
    }

    public int getPar() {
        return this.par;
    }

    public int getLength() {
        return this.length;
    }
}
}

```

图 4-13 显示了实际运行的工具提示。

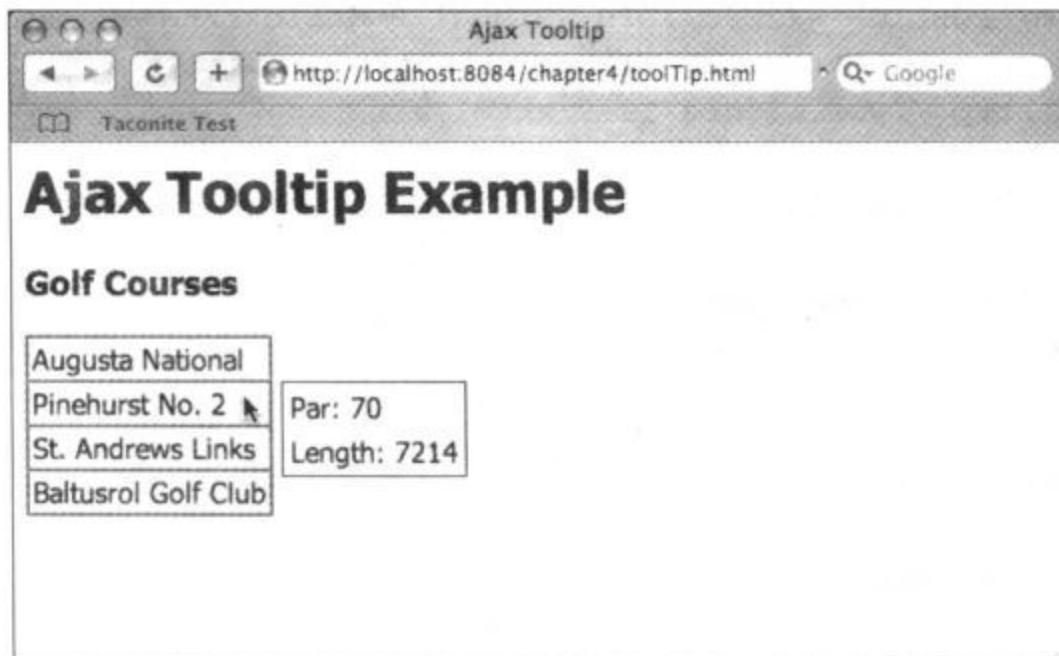


图 4-13 Ajax 工具提示示例

## 4.7 动态更新Web页面

如前所述，如果页面中只有一小部分需要修改，此时 Ajax 技术最适用。换句话说，以前实现一些用例时，为了更新页面中的一小部分总是需要使用完全页面刷新，这些用例就很适合采用 Ajax 技术。

考虑一个有单个页面的用例，用户向这个页面输入的信息要增加到列表中。在这个例子中，你会看到列出某个组织中员工的 Web 页面。页面最上面有 3 个输入框，分别接受员工的姓名、职位和部门。点击 Add（增加）按钮，将员工的姓名、职位和部门数据提交到服务器，在这里将这些员工信息增加到数据库中。

当使用传统的 Web 应用技术时，服务器以重新创建整个页面来做出响应，与前一个页

面相比，唯一的差别只是新员工信息会增加到列表中。在这个例子中，我们要使用 Ajax 技术异步地将员工数据提交到服务器，并把该数据插入到数据库中。服务器发送一个状态码向浏览器做出响应，指示数据库操作是否成功。假设数据库成功插入，浏览器会使用 JavaScript DOM 操作用新员工信息动态更新页面内容。这个例子中还创建了 Delete（删除）按钮，以便从数据库中删除员工信息。

代码清单 4-13 显示了 HTML Web 页面的源代码。这个页面有两部分：第一部分包括一些输入框，分别接受员工姓名、职位和部门的数据，以及启动数据库插入的 Add 按钮；第二部分列出数据库中的所有员工，每个记录有自己的 Delete 按钮，从而能从数据库删除这个记录的信息。

#### 代码清单 4-13 employeeList.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Employee List</title>

<script type="text/javascript">
var xmlhttp;
var name;
var title;
var department;
var deleteID;
var EMP_PREFIX = "emp-";

function createXMLHttpRequest() {
    if (window.ActiveXObject) {
        xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
    }
    else if (window.XMLHttpRequest) {
        xmlhttp = new XMLHttpRequest();
    }
}

function addEmployee() {
    name = document.getElementById("name").value;
    title = document.getElementById("title").value;
    department = document.getElementById("dept").value;
    action = "add";
    if(name == "" || title == "" || department == "") {
        return;
    }
}
```

```
var url = "EmployeeList?"
    + createAddQueryString(name, title, department, "add")
    + "&ts=" + new Date().getTime();

createXMLHttpRequest();
xmlHttp.onreadystatechange = handleAddStateChange;
xmlHttp.open("GET", url, true);
xmlHttp.send(null);
}

function createAddQueryString(name, title, department, action) {
    var queryString = "name=" + name
        + "&title=" + title
        + "&department=" + department
        + "&action=" + action;
    return queryString;
}

function handleAddStateChange() {
    if(xmlHttp.readyState == 4) {
        if(xmlHttp.status == 200) {
            updateEmployeeList();
            clearInputBoxes();
        }
        else {
            alert("Error while adding employee.");
        }
    }
}

function clearInputBoxes() {
    document.getElementById("name").value = "";
    document.getElementById("title").value = "";
    document.getElementById("dept").value = "";
}

function deleteEmployee(id) {
    deleteID = id;

    var url = "EmployeeList?"
        + "action=delete"
        + "&id=" + id
        + "&ts=" + new Date().getTime();
    createXMLHttpRequest();
    xmlHttp.onreadystatechange = handleDeleteStateChange;
    xmlHttp.open("GET", url, true);
}
```

```
        xmlhttp.send(null);
    }

    function updateEmployeeList() {
        var responseXML = xmlhttp.responseXML;

        var status = responseXML.getElementsByTagName("status")
                    .item(0).firstChild.nodeValue;
        status = parseInt(status);
        if(status != 1) {
            return;
        }

        var row = document.createElement("tr");
        var uniqueID = responseXML.getElementsByTagName("uniqueID")[0]
                      .firstChild.nodeValue;
        row.setAttribute("id", EMP_PREFIX + uniqueID);

        row.appendChild(createCellWithText(name));
        row.appendChild(createCellWithText(title));
        row.appendChild(createCellWithText(department));

        var deleteButton = document.createElement("input");
        deleteButton.setAttribute("type", "button");
        deleteButton.setAttribute("value", "Delete");
        deleteButton.onclick = function () { deleteEmployee(uniqueID); };
        cell = document.createElement("td");
        cell.appendChild(deleteButton);
        row.appendChild(cell);

        document.getElementById("employeeList").appendChild(row);
        updateEmployeeListVisibility();
    }

    function createCellWithText(text) {
        var cell = document.createElement("td");
        cell.appendChild(document.createTextNode(text));
        return cell;
    }

    function handleDeleteStateChange() {
        if(xmlhttp.readyState == 4) {
            if(xmlhttp.status == 200) {
                deleteEmployeeFromList();
            }
            else {
                alert("Error while deleting employee.");
            }
        }
    }
}
```

```
        }
    }

}

function deleteEmployeeFromList() {
    var status =
        xmlhttp.responseXML.getElementsByName("status")
        .item(0).firstChild.nodeValue;
    status = parseInt(status);
    if(status != 1) {
        return;
    }

    var rowToDelete = document.getElementById(EMP_PREFIX + deleteID);
    var employeeList = document.getElementById("employeeList");
    employeeList.removeChild(rowToDelete);

    updateEmployeeListVisibility();
}

function updateEmployeeListVisibility() {
    var employeeList = document.getElementById("employeeList");
    if(employeeList.childNodes.length > 0) {
        document.getElementById("employeeListSpan").style.display = "";
    }
    else {
        document.getElementById("employeeListSpan").style.display = "none";
    }
}
</script>
</head>

<body>
    <h1>Employee List</h1>
    <form action="#">
        <table width="80%" border="0">
            <tr>
                <td>Name: <input type="text" id="name"/></td>
                <td>Title: <input type="text" id="title"/></td>
                <td>Department: <input type="text" id="dept"/></td>
            </tr>
            <tr>
                <td colspan="3" align="center">
                    <input type="button" value="Add" onclick="addEmployee();"/>
                </td>
            </tr>
        </table>
    </form>
</body>
```

```

</form>

<span id="employeeListSpan" style="display:none;">
<h2>Employees:</h2>

<table border="1" width="80%">
  <tbody id="employeeList"></tbody>
</table>
</span>
</body>
</html>

```

点击 Add 按钮启动数据库插入操作。基于 Add 按钮的 onclick 事件将调用 addEmployee 函数。addEmployee 函数使用 createAddQueryString 来建立查询串，其中包括用户输入的员工姓名、职位和部门信息。创建 XMLHttpRequest 对象并设置 onreadystatechange 事件处理程序后，请求提交到服务器。

代码清单 4-14 列出了处理请求的 Java servlet，当接收到请求时将调用 servlet 的 doGet 方法。这个方法获取查询串 action 参数的值，并把请求指向适当的方法。如果是增加信息，请求指向 addEmployee 方法。

#### 代码清单 4-14 EmployeeListServlet.java

```

package ajaxbook.chap4;

import java.io.*;
import java.net.*;
import java.util.Random;

import javax.servlet.*;
import javax.servlet.http.*;

public class EmployeeListServlet extends HttpServlet {
    protected void addEmployee(HttpServletRequest request
                               , HttpServletResponse response)
        throws ServletException, IOException {
        //Store the object in the database
        String uniqueID = storeEmployee();

        //Create the response XML
        StringBuffer xml = new StringBuffer("<result><uniqueID>");
        xml.append(uniqueID);
        xml.append("</uniqueID>");
        xml.append("</result>");

        //Send the response back to the browser
    }
}

```

```
        sendResponse(response, xml.toString());
    }

protected void deleteEmployee(HttpServletRequest request
                               , HttpServletResponse response)
throws ServletException, IOException {

    String id = request.getParameter("id");
    /* Assume that a call is made to delete the employee from the database */

    //Create the response XML
    StringBuffer xml = new StringBuffer("<result>");
    xml.append("<status>1</status>");
    xml.append("</result>");

    //Send the response back to the browser
    sendResponse(response, xml.toString());
}

protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    String action = request.getParameter("action");
    if(action.equals("add")) {
        addEmployee(request, response);
    }
    else if(action.equals("delete")) {
        deleteEmployee(request, response);
    }
}

private String storeEmployee() {
    /* Assume that the employee is saved to a database and the
     * database creates a unique ID. Return the unique ID to the
     * calling method. In this case, make up a unique ID.
     */
    String uniqueID = "";
    Random randomizer = new Random(System.currentTimeMillis());
    for(int i = 0; i < 8; i++) {
        uniqueID += randomizer.nextInt(9);
    }

    return uniqueID;
}

private void sendResponse(HttpServletResponse response, String responseText)
throws IOException {
    response.setContentType("text/xml");
    response.getWriter().write(responseText);
}
```

`addEmployee` 函数负责协调数据库插入和服务器响应。`addEmployee` 方法委托 `storeEmployee` 方法完成具体的数据库插入。在实际实现中，`storeEmployee` 方法很可能调用数据库服务，由它处理数据库插入的具体细节。在这个简化的例子中，`storeEmployee` 将模拟数据库插入，其方法是生成一个随机的惟一 ID，模拟实际数据库插入可能返回的 ID。生成的惟一 ID 再返回给 `addEmployee` 方法。

假设数据库插入成功，`addEmployee` 方法则继续准备响应。响应是一个简单的 XML 串，它向浏览器返回一个状态码。XML 通过串连接创建，然后写至响应的输出流。

浏览器通过调用 `handleAddStateChange` 方法来处理服务器的响应。只要 XMLHttpRequest 对象发出信号指出其内部准备状态有变化，就会调用这个方法。一旦 `readystate` 属性指示服务器响应已经成功完成，就会调用 `updateEmployeeList` 函数，然后再调用 `clearInputBoxes` 函数。`updateEmployeeList` 函数负责把成功插入的员工信息增加到页面上显示的员工列表中。`clearInputBoxes` 函数是一个简单的工具方法，它会清空输入框，准备接收下一个员工的信息。

`updateEmployeeList` 函数向表中增加行以列出员工的信息。首先使用 `document.createElement` 方法来创建表行的一个实例。这一行的 `id` 属性设置为包括由数据库插入生成的惟一 ID 的值。`id` 属性值唯一地标识了表行，这样点击 `Delete` 按钮时就能很容易地从表中删除这一行。

`updateEmployeeList` 函数使用名为 `createCellWithText` 的工具函数来创建表单元格元素，其中包含指定的文本。`createCellWithText` 函数分别为用户输入的姓名、职位和部门信息创建表单元格，再把各个单元格增加到先前创建的表行中。

最后要创建的是 `Delete` 按钮以及包含这个按钮的单元格。使用 `document.createElement` 方法创建通用的输入元素，其 `type` 和 `value` 属性分别设置为 `button` 和 `Delete`，这样就能创建 `Delete` 按钮。再创建表单元格，用来放置 `Delete` 按钮，并把 `Delete` 按钮作为子元素增加到表单元格。然后把这个单元格增加到表行，接下来将这一行增加到员工列表中，现在行中已经包含了对应员工姓名、职位、部门和 `Delete` 按钮的单元格。

删除员工与增加员工的工作是一样的。`Delete` 按钮的 `onclick` 事件处理程序调用 `deleteEmployee` 函数，将员工的惟一 ID 传递给这个函数。此时创建一个简单的查询串，指示想做的动作(删除)和要删除的员工记录的惟一 ID。XMLHttpRequest 对象的 `onreadystatechange` 属性设置为所需的事件处理程序后，提交请求。

`EmployeeListServlet` servlet 使用 `deleteEmployee` 方法来处理员工删除用例。这个例子做了简化，在此假设还有另一个方法处理数据库删除的具体细节。如果成功地完成了数据库删除，`deleteEmployee` 方法会准备 XML 串，返回给浏览器。与员工增加用例类似，这个用例向浏览器返回一个状态码。一旦创建 XML 串，则将 XML 串通过响应回对象的输出流写回

到浏览器。

浏览器通过 `handleDeleteStateChange` 函数处理服务器响应，如果响应成功，将转发到 `deleteEmployeeFromList` 方法。`deleteEmployeeFromList` 函数从 XML 响应获取状态码，如果状态码指示删除不成功，这个函数将立即退出。假设成功地完成了删除操作，这个函数则会继续，使用 `document.getElementById` 方法获取表示所删除信息的表行，然后使用表体的 `removeChild` 方法从中删除这一行。

### 为什么不使用 `setAttribute` 方法来设置 DELETE 按钮的事件处理器？

你可能已经注意到，设置 Delete 按钮的事件处理器时采用了何种方法。你可能认为，设置 Delete 按钮的 `onclick` 事件处理器的代码应该如下所示：

```
deleteButton.setAttribute("onclick", "deleteEmployee('" + unique_id + "');");
```

确实，这个代码从理论上是对的，它遵循 W3C 标准，而且在大多数当前浏览器中都可行，只有 IE 例外。幸运的是，对于 IE 也有一个解决办法，它可以在 Firefox、Opera、Safari 和 Konqueror 中适用。

这种解决办法是使用点记法引用 Delete 按钮的 `onclick` 事件处理器，然后使用调用 `deleteEmployee` 函数的匿名函数来设置事件处理器。

图 4-14 显示了实际运行的动态更新例子。

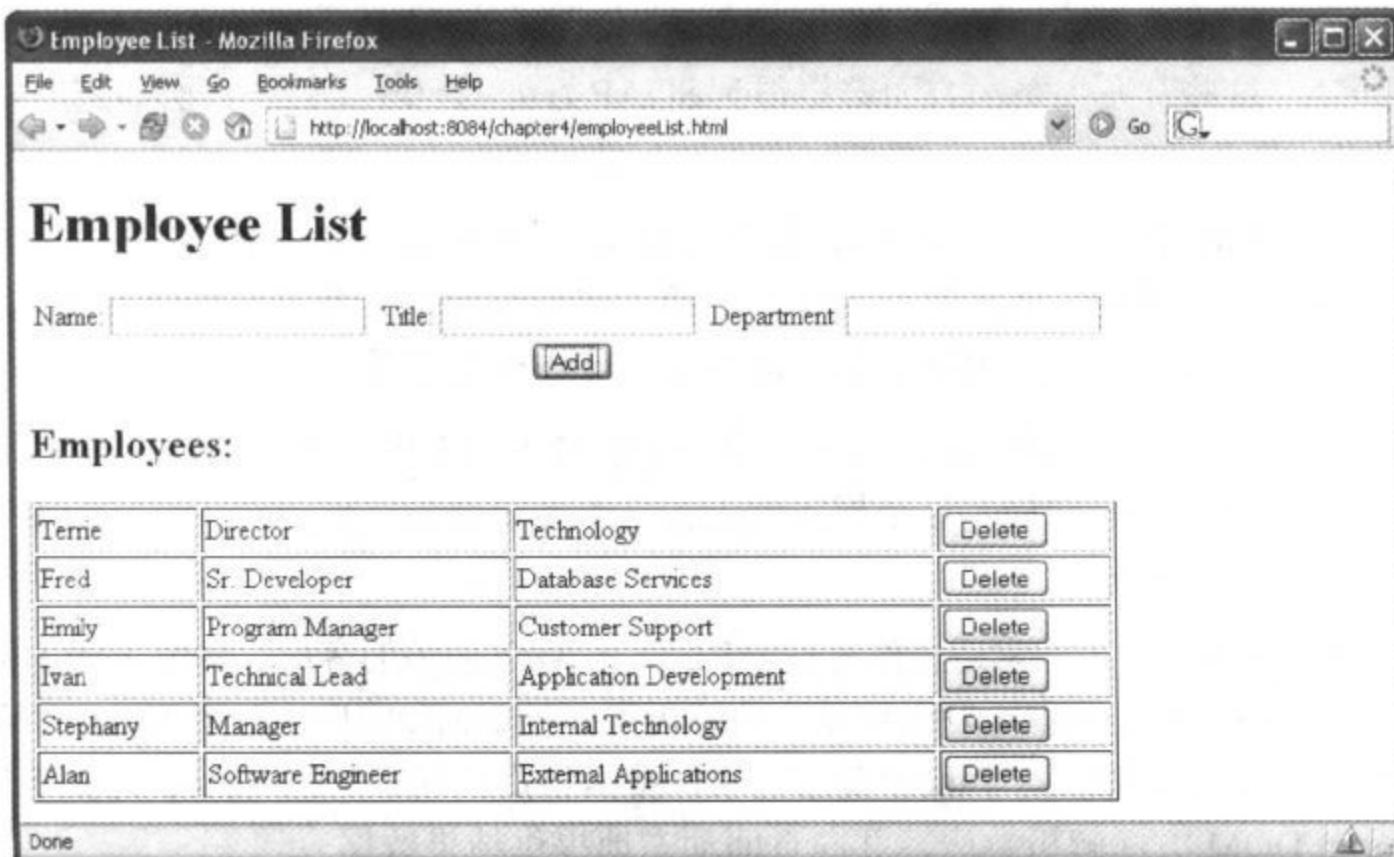


图 4-14 每次点击 Add 按钮时每个姓名动态增加到列表中，而不必每次都刷新页面

## 4.8 访问Web服务

多年以来一直存在一个软件工程问题：从一台机器调用另一台机器上的服务或方法，即使这些机器使用完全不同的硬件或软件。对于这个问题，最近提出的解决方案是 Web 服务。几年前，Web 服务大受吹捧，它的头上围绕着耀眼的光环，有些人认为 Web 服务就是分布式软件开发的“圣杯”。后来，它的光芒逐渐黯淡下来，Web 服务最终找到了自己合适的位置，它是支持异构计算机系统相互操作的一种有用的工具。

Web 服务通常用作为计算机系统之间的通信管道，这与 CORBA（公共对象请求代理体系结构）、RMI（远程方法调用）或 DCOM（分布式组件对象模型）很相似。区别在于，Web 服务独立于具体的开发商，可以采用大量编程工具和平台来实现。为了支持更高层次的互操作性，Web 服务是基于文本的协议，通常在 HTTP 之上实现。由于 Web 服务是基于文本的协议，所以几乎总能使用某种 XML。

最著名的 Web 服务实现是 SOAP（简单对象访问协议）。SOAP 是由 W3C 管理的规约，它是 XML 协议，对于如何调用远程过程给出了定义。

WSDL（Web 服务描述语言）文档也是 XML 文档，描述了如何创建 Web 服务的客户。通过提供 WSDL 文档，Web 服务提供者就能很轻松地为可能的客户创建客户端代码。WSDL 和 SOAP 通常一同使用，不过不一定非得这样，因为这两个规约是分开维护的。

尽管人们在简化 SOAP 实现上做出了很大努力，但 SOAP 还是一个很难使用的技术，因此很受“排挤”，只有在跨平台互操作性确实是一个很重要的需求时才会使用 SOAP。实现 Web 服务还有一种更简单的方法，称为 REST（代表状态传输），它在开发人员中享有越来越高的知名度，这些开发人员一方面希望得到 SOAP 好处的 80%，另一方面只希望付出 SOAP 代价的 20%。

Yahoo!选择 REST 作为其公共 Web 服务的协议。Yahoo!认为基于 REST 的服务很容易理解，而且很推崇 REST 的“平易近人”，因为当前大多数编程语言都可以访问 REST。实际上，Yahoo!相信，与 SOAP 相比，REST 的门槛更低，使用也更容易。

通过使用 REST，建立请求时可以先指定一个服务入口 URL，再向查询串追加搜索参数。服务将结果返回为 XML 文档。这个模式听上去是不是很熟悉？你说对了，它与本书中你见过的 Ajax 例子是一样的。

XMLHttpRequest 对象非常适合作为基于 REST 的 Web 服务的客户。使用 XMLHttpRequest 对象，可以向 Web 服务异步地发出请求，并解析得到的 XML 响应。对于 Yahoo! Web 服务，XMLHttpRequest 对象可以向 Yahoo!发出请求，搜索指定的项。一旦 Yahoo!返回响应，则使用 JavaScript DOM 方法解析响应，并向页面动态地提供结果数据。

代码清单 4-15 展示了如何使用 Ajax 技术访问 Yahoo! Web 服务，并向页面提供结果。

页面上的文本字段允许用户指定搜索项。用户可以使用选择框来指定需要显示多少个结果。点击 Submit（提交）按钮就能启动搜索。

不过，先等等！第2章我们曾经说过，`XMLHttpRequest` 对象只能访问发起文档（即调用脚本）所在域中的资源。如果试图访问其他域的资源，可能因为浏览器的安全限制而失败。怎么解决呢？

解决办法有好几个。在第2章已经了解到，浏览器实现安全沙箱的方式各有不同。IE会询问用户是否允许访问另一个域中的资源。Firefox则会报告错误，自动失败，虽然可以用专用于 Firefox 的 JavaScript 代码避免这种行为。

还有一个选择，这也是本例中要采用的方法，就是建立 Yahoo! 的网关，它与 `XMLHttpRequest` 脚本在同一个域中。由网关接收来自 `XMLHttpRequest` 对象的请求，并把它转发到 Yahoo! Web 服务。Yahoo! 做出响应返回结果时，网关再把结果路由传送到浏览器。通过使用这种方法，就能避免使用浏览器特定的 JavaScript。另外，这种方法也更加健壮，因为你还可以扩展网关，让它支持其他的 Web 服务提供者。

#### 代码清单 4-15 yahooSearch.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml11-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Yahoo! Search Web Services</title>

<script type="text/javascript">
var xmlhttp;
function createXMLHttpRequest() {
    if (window.ActiveXObject) {
        xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
    }
    else if (window.XMLHttpRequest) {
        xmlhttp = new XMLHttpRequest();
    }
}

function doSearch() {
    var url = "YahooSearchGateway?" + createQueryString()
              + "&ts=" + new Date().getTime();
    createXMLHttpRequest();
    xmlhttp.onreadystatechange = handleStateChange;
    xmlhttp.open("GET", url, true);
    xmlhttp.send(null);
}
```

```
function createQueryString() {
    var searchString = document.getElementById("searchString").value;
    searchString = escape(searchString);

    var maxResultsCount = document.getElementById("maxResultCount").value;

    var queryString = "query=" + searchString + "&results=" + maxResultsCount;
    return queryString;
}

function handleStateChange() {
    if(xmlHttp.readyState == 4) {
        if(xmlHttp.status == 200) {
            parseSearchResults();
        }
        else {
            alert("Error accessing Yahoo! search");
        }
    }
}

function parseSearchResults() {
    var resultsDiv = document.getElementById("results");
    while(resultsDiv.childNodes.length > 0) {
        resultsDiv.removeChild(resultsDiv.childNodes[0]);
    }

    var allResults = xmlHttp.responseXML.getElementsByTagName("Result");
    var result = null;
    for(var i = 0; i < allResults.length; i++) {
        result = allResults[i];
        parseResult(result);
    }
}

function parseResult(result) {
    var resultDiv = document.createElement("div");

    var title = document.createElement("h3");
    title.appendChild(document.createTextNode(
        getChildElementText(result, "Title")));
    resultDiv.appendChild(title);

    var summary = document.createTextNode(getChildElementText(result, "Summary"));
    resultDiv.appendChild(summary);
}
```

```

resultDiv.appendChild(document.createElement("br"));
var clickHere = document.createElement("a");
clickHere.setAttribute("href", getChildElementText(result, "ClickUrl"));
clickHere.appendChild(document.createTextNode
    (getChildElementText(result, "Url")));
resultDiv.appendChild(clickHere);

document.getElementById("results").appendChild(resultDiv);
}

function getChildElementText(parentNode, childTagName) {
    var childTag = parentNode.getElementsByTagName(childTagName);
    return childTag[0].firstChild.nodeValue;
}
</script>
</head>

<body>
<h1>Web Search Using Yahoo! Search Web Services</h1>

<form action="#">
    Search String: <input type="text" id="searchString"/>

    <br/><br/>
    Max Number of Results:
    <select id="maxResultCount">
        <option value="1">1</option>
        <option value="10">10</option>
        <option value="25">25</option>
        <option value="50">50</option>
    </select>
    <br/><br/>
    <input type="button" value="Submit" onclick="doSearch();"/>
</form>

<h2>Results:</h2>
<div id="results"/>

</body>
</html>

```

点击页面上的 Submit(提交)按钮将调用 doSearch 函数。这个函数使用 createQueryString 函数来创建目标 URL, createQueryString 函数负责把搜索项和显示的最大结果数(即最多显示多少个结果)放在查询串中。需要注意, 参数名(query 和 results)都是 Yahoo! Search API 定义的。

createQueryString 函数创建的查询串发送给 Yahoo! Search 网关。在这个例子中, 网

关实现为名为 YahooSearchGatewayServlet 的 Java servlet(见代码清单 4-16)。这个 servlet 的目的很简单，就是转发对 Yahoo! Search URL 的所有请求，并把结果传给浏览器。当然，这个网关也可以用其他语言(而不是 Java)来实现。这个网关很简单，在 XMLHttpRequest 对象需要访问其他域中的资源时，这个网关确实能解决问题。

#### 代码清单 4-16 YahooSearchGatewayServlet.java

```
package ajaxbook.chap4;

import java.io.*;
import java.net.HttpURLConnection;
import java.net.URL;

import javax.servlet.*;
import javax.servlet.http.*;

public class YahooSearchGatewayServlet extends HttpServlet {
    private static final String YAHOO_SEARCH_URL =
        "http://api.search.yahoo.com/WebSearchService/V1/webSearch?"
        + "appid=your_app_id" + "&type=all";

    protected void processRequest(HttpServletRequest request
                                  , HttpServletResponse response)
        throws ServletException, IOException {

        String url = YAHOO_SEARCH_URL + "&" + request.getQueryString();

        HttpURLConnection con = (HttpURLConnection)new URL(url).openConnection();
        con.setDoInput(true);
        con.setDoOutput(true);
        con.setRequestMethod("GET");

        //Send back the response to the browser
        response.setStatus(con.getResponseCode());
        response.setContentType("text/xml");

        BufferedReader reader =
            new BufferedReader(new InputStreamReader(con.getInputStream()));
        String input = null;
        OutputStream responseOutput = response.getOutputStream();

        while((input = reader.readLine()) != null) {
            responseOutput.write(input.getBytes());
        }

    }
}
```

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    processRequest(request, response);
}
}
```

Yahoo! Search 把结果返回给网关，而且网关将结果再转发给浏览器后，会调用 parseSearchResults 函数。这个函数从 XMLHttpRequest 对象获取得到的 XML 文档，并查找所有标记名为 Result 的元素。

各 Result 元素传给 parseResult 函数。这个函数使用 Result 元素的子元素 Title、Summary、ClickUrl 和 Url 创建内容，并增加到页面。

可以看到，与基于 REST 的 Web 服务结合使用时，Ajax 技术相当强大。如果想在你自己的域中访问 Web 服务，用 JavaScript 就可以完成。否则，当访问其他域的资源时，就要创建外部资源的某种网关，这样就能避免浏览器安全沙箱问题。

图 4-15 显示了结合使用 Yahoo! Search Web 服务和 Ajax 的搜索结果。

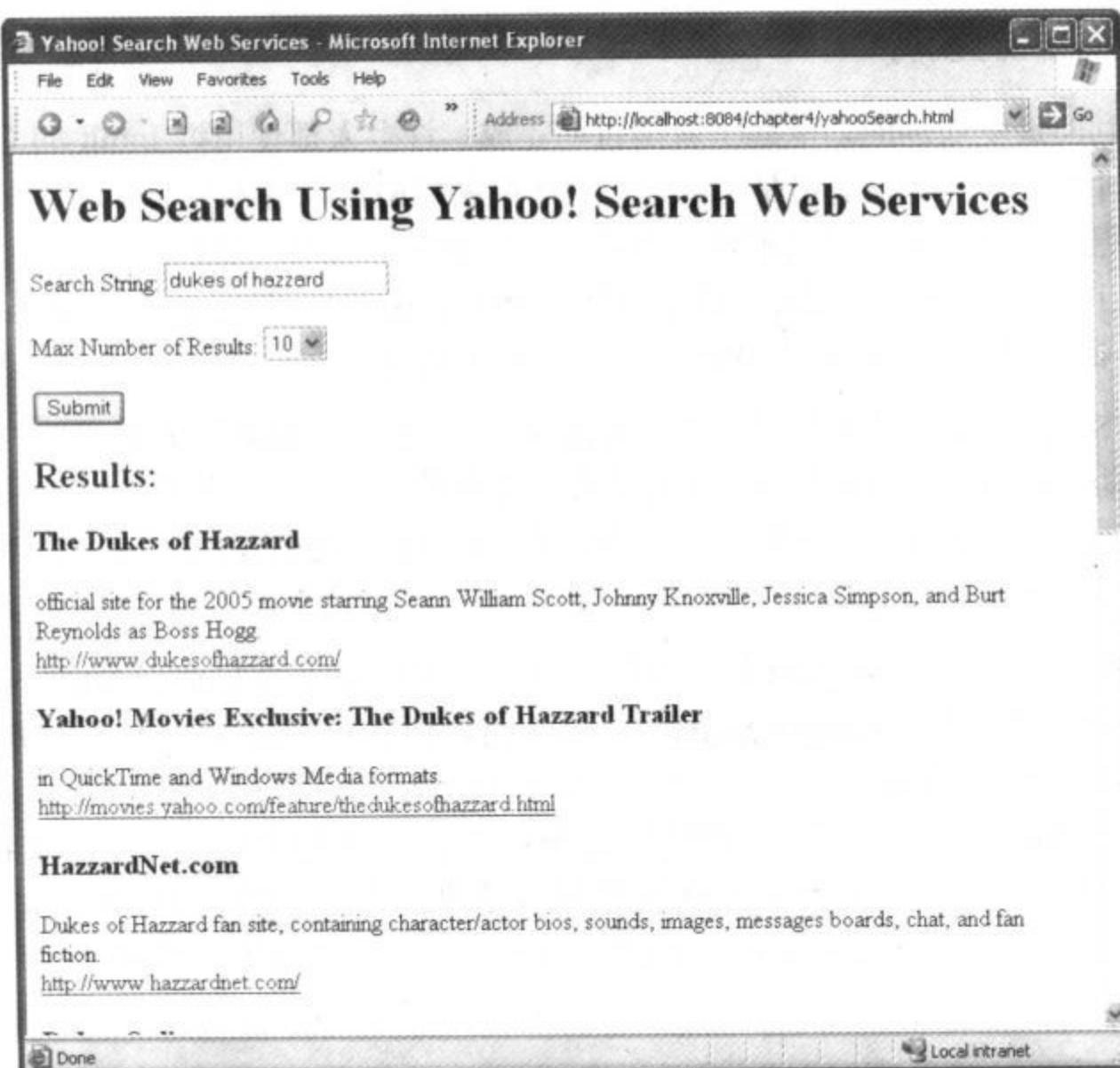


图 4-15 结合使用 Yahoo! Search Web 服务和 Ajax 的搜索结果

### Ajax 也能与 SOAP 一同使用吗？

可以结合使用 Ajax 和 SOAP 吗？答案很简单——可以。Ajax 技术和基于 SOAP 的 Web 服务可以一同使用，不过，与使用基于 REST 的 Web 服务相比，这需要做更多的工作。

REST 和 SOAP 都把响应返回为 XML 文档。二者之间最显著的差异是，REST 将请求作为带查询串参数的简单 URL 发送，而 SOAP 请求是具体的 XML 文档，通常通过 POST 而不是 GET 发送。

结合使用 SOAP 和 Ajax 时，要求以某种方式创建 SOAP 请求的 XML，这可能并不容易。一种做法是使用串连接来创建请求 XML。尽管概念上讲很简单，但这种方法有些混乱，而且很容易出错，如很容易这儿忘了双引号，那儿忘了加号。

还有一种选择是使用一个 XMLHttpRequest 请求从网站加载静态 XML 文档，文档是 SOAP 请求的模板。一旦加载了模板，就可以使用 JavaScript DOM 方法来修改模板，使之满足特定的请求。请求准备好后，再用第二个 XMLHttpRequest 请求发送新创建的 SOAP 请求。

## 4.9 提供自动完成

我们遇到的最受欢迎的功能之一就是自动完成。许多人都使用过 Intuit 的 Quicken 之类的工具，并对其注册表的功能很是着迷，它的注册表能根据以前的注册项填入信息。这就使得数据输入更快、更容易，而且不容易出错。对于胖客户应用，增加这个功能可能很容易，但是 Web 应用长期以来一直都没有这个特性<sup>1</sup>。不过，Google 在其 beta 实验区推出 Google Suggest 后，证明了自动完成对于 Web 应用并非遥不可及。

Google Suggest 实在让人赞叹不已（见图 4-16）。它不仅很好地放置了下拉区，还会在输入框中自动插入最有可能的答案，并将非用户键入的部分置灰，在下拉区中甚至还能使用向上和向下箭头。由于为给定项提供了一些结果，用户就能更清楚地认识到具体完成搜索时可能会得到的结果。

许多网站都对 Google Suggest 做了剖析（可以在 google 上查 Google Suggest!）。代码清单 4-17 所示的例子比不上 Google Suggest 那么丰富，但你确实能从中了解到利用 Ajax 可以做些什么。需要注意，在这个例子中，callback() 函数除了查找一般的返回码 200，还会查找返回码 204。204 响应码指示服务器没有发回任何信息，利用个提示可以清空名字下拉区。你还会注意到，通过点记法为单元格设置鼠标事件，这一点在前面的“为什么不能用 setAttribute 方法来设置 Delete 按钮的事件处理程序？”旁注中已经解释过。重申一次，

1. 本书的一位作者曾有过一个客户，他死活不相信他那么漂亮的新 Web 应用居然没有自动完成特性。像许多请求一样，这个请求已经列入有待实现的特性列表中。

使用 calculateOffset() 方法来确定应该把数据放在哪个位置。

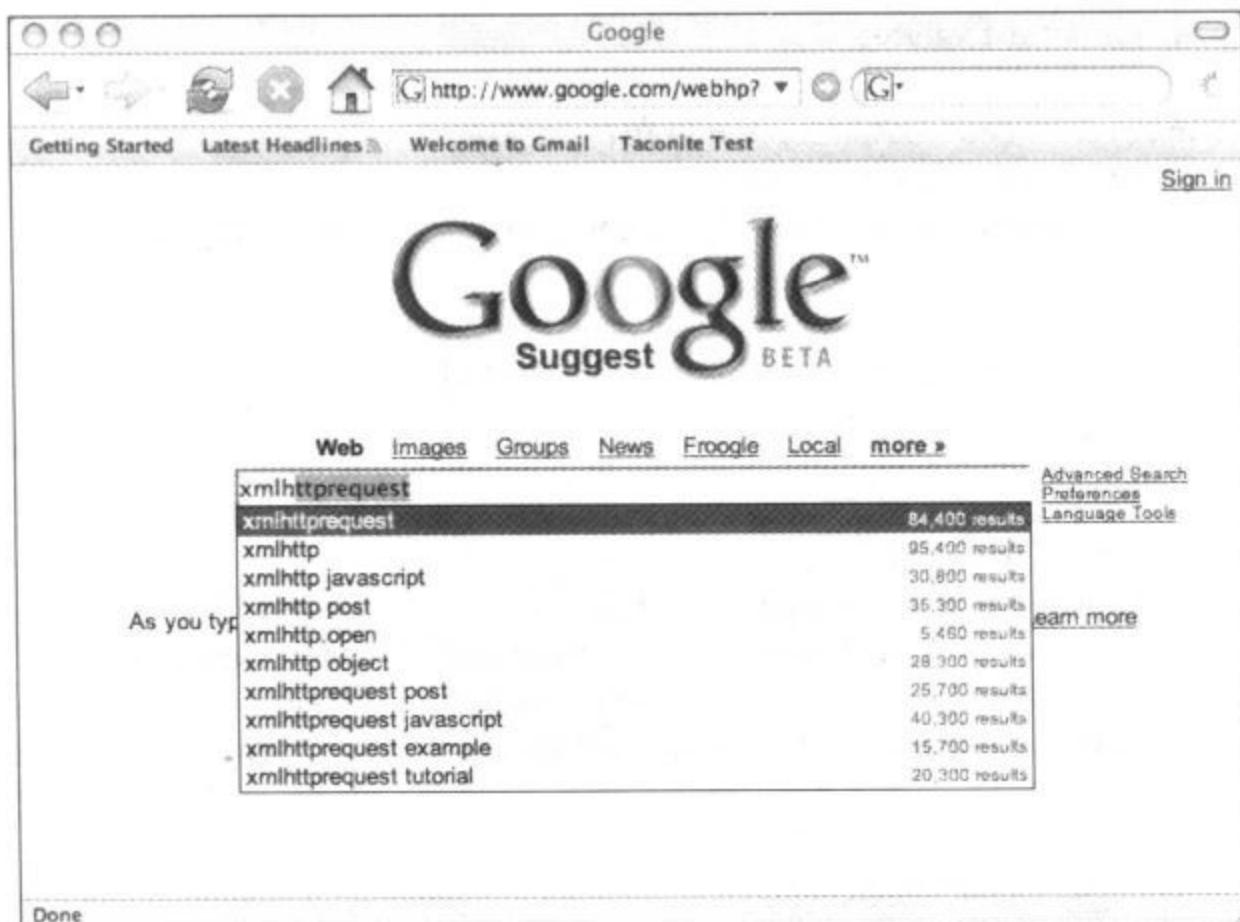


图 4-16 Ajax 为 Google Suggest 增色不少

#### 代码清单 4-17 autoComplete.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<html>
  <head>
    <title>Ajax Auto Complete</title>
    <style type="text/css">

      .mouseOut {
        background: #708090;
        color: #FFFAFA;
      }

      .mouseOver {
        background: #FFFAFA;
        color: #000000;
      }
    </style>
    <script type="text/javascript">
      var xmlhttp;
      var completeDiv;
```

```
var inputField;
var nameTable;
var nameTableBody;

function createXMLHttpRequest() {
    if (window.ActiveXObject) {
        xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
    }
    else if (window.XMLHttpRequest) {
        xmlhttp = new XMLHttpRequest();
    }
}

function initVars() {
    inputField = document.getElementById("names");
    nameTable = document.getElementById("name_table");
    completeDiv = document.getElementById("popup");
    nameTableBody = document.getElementById("name_table_body");
}

function findNames() {
    initVars();
    if (inputField.value.length > 0) {
        createXMLHttpRequest();
        var url = "AutoCompleteServlet?names=" + escape(inputField.value);
        xmlhttp.open("GET", url, true);
        xmlhttp.onreadystatechange = callback;
        xmlhttp.send(null);
    } else {
        clearNames();
    }
}

function callback() {
    if (xmlhttp.readyState == 4) {
        if (xmlhttp.status == 200) {
            var name =
                xmlhttp.responseXML
                    .getElementsByTagName("name")[0].firstChild.data;
            setNames(xmlhttp.responseXML.getElementsByTagName("name"));
        } else if (xmlhttp.status == 204) {
            clearNames();
        }
    }
}

function setNames(the_names) {
```

```
clearNames();
var size = the_names.length;
setOffsets();

var row, cell, txtNode;
for (var i = 0; i < size; i++) {
    var nextNode = the_names[i].firstChild.data;
    row = document.createElement("tr");
    cell = document.createElement("td");

    cell.onmouseout = function() {this.className='mouseOver';};
    cell.onmouseover = function() {this.className='mouseOut';};
    cell.setAttribute("bgcolor", "#FFFAFA");
    cell.setAttribute("border", "0");
    cell.onclick = function() { populateName(this); } ;

    txtNode = document.createTextNode(nextNode);
    cell.appendChild(txtNode);
    row.appendChild(cell);
    nameTableBody.appendChild(row);
}

function setOffsets() {
    var end = inputField.offsetWidth;
    var left = calculateOffsetLeft(inputField);
    var top = calculateOffsetTop(inputField) + inputField.offsetHeight;

    completeDiv.style.border = "black 1px solid";
    completeDiv.style.left = left + "px";
    completeDiv.style.top = top + "px";
    nameTable.style.width = end + "px";
}

function calculateOffsetLeft(field) {
    return calculateOffset(field, "offsetLeft");
}

function calculateOffsetTop(field) {
    return calculateOffset(field, "offsetTop");
}

function calculateOffset(field, attr) {
    var offset = 0;
    while(field) {
        offset += field[attr];
        field = field.offsetParent;
```

```

        }
        return offset;
    }

    function populateName(cell) {
        inputField.value = cell.firstChild.nodeValue;
        clearNames();
    }

    function clearNames() {
        var ind = nameTableBody.childNodes.length;
        for (var i = ind - 1; i >= 0 ; i--) {
            nameTableBody.removeChild(nameTableBody.childNodes[i]);
        }
        completeDiv.style.border = "none";
    }

</script>
</head>
<body>
    <h1>Ajax Auto Complete Example</h1>
    Names: <input type="text" size="20" id="names"
                  onkeyup="findNames();" style="height:20;" />
    <div style="position:absolute;" id="popup">
        <table id="name_table" bgcolor="#FFFFAFA" border="0"
              cellspacing="0" cellpadding="0"/>

        <tbody id="name_table_body"></tbody>
    </table>
</div>
</body>
</html>

```

服务器端代码模拟了命名服务的动态名字搜索功能。servlet 中设置了一组名字，搜索委托给包含有查找逻辑的另一个类。注意，如果没有找到任何数据，要向客户返回响应，指示没有内容。代码清单 4-18 显示了 AutoCompleteServlet.java，代码清单 4-19 是 NameService.java。

#### 代码清单 4-18 AutoCompleteServlet.java

```

package ajaxbook.chap4;

import java.io.*;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

```

```
import javax.servlet.*;
import javax.servlet.http.*;

public class AutoCompleteServlet extends HttpServlet {

    private List names = new ArrayList();

    public void init(ServletConfig config) throws ServletException {
        names.add("Abe");
        names.add("Abel");
        names.add("Abigail");
        names.add("Abner");
        names.add("Abraham");
        names.add("Marcus");
        names.add("Marcy");
        names.add("Marge");
        names.add("Marie");
    }

protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    String prefix = request.getParameter("names");
    NameService service = NameService.getInstance(names);
    List matching = service.findNames(prefix);
    if (matching.size() > 0) {
        PrintWriter out = response.getWriter();

        response.setContentType("text/xml");
        response.setHeader("Cache-Control", "no-cache");

        out.println("<response>");
        Iterator iter = matching.iterator();
        while(iter.hasNext()) {
            String name = (String) iter.next();
            out.println("<name>" + name + "</name>");
        }
        out.println("</response>");
        matching = null;
        service = null;
        out.close();
    } else {
        response.setStatus(HttpServletResponse.SC_NO_CONTENT);
    }
}
}
```

代码清单 4-19 NameService.java

```
package ajaxbook.chap4;
```

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class NameService {
    private List names;

    /** Creates a new instance of NameService */
    private NameService(List list_of_names) {
        this.names = list_of_names;
    }

    public static NameService getInstance(List list_of_names) {
        return new NameService(list_of_names);
    }

    public List findNames(String prefix) {
        String prefix_upper = prefix.toUpperCase();
        List matches = new ArrayList();
        Iterator iter = names.iterator();
        while(iter.hasNext()) {
            String name = (String) iter.next();
            String name_upper_case = name.toUpperCase();
            if(name_upper_case.startsWith(prefix_upper)){
                boolean result = matches.add(name);
            }
        }
        return matches;
    }
}
```

图 4-17 显示了实际运行的自动完成示例。

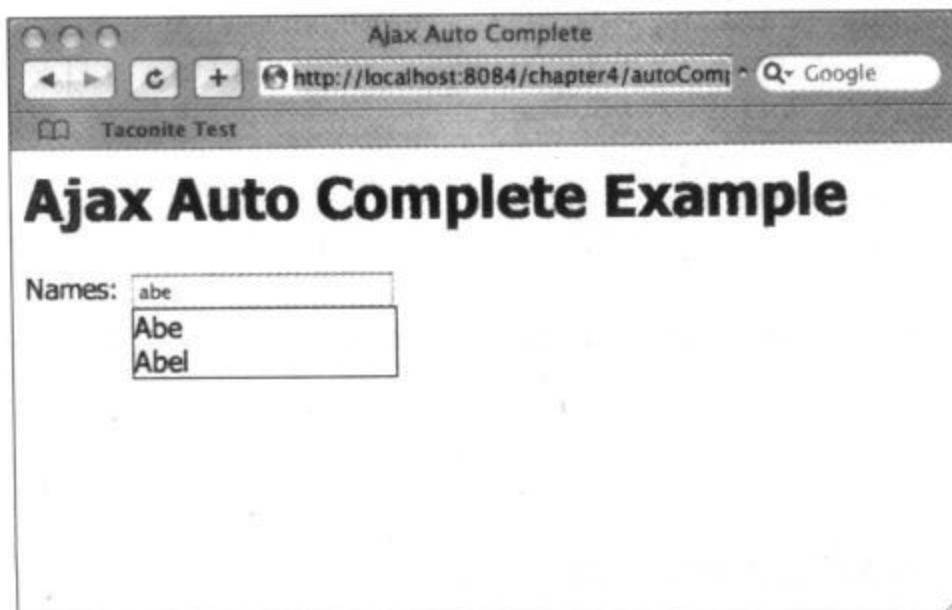


图 4-17 自动完成示例

## 4.10 小结

本章我们提供了许多例子，展示了应用 Ajax 技术改善用户体验的方法。在许多情况下，可以把 Ajax 技术应用到现有的应用中，以 Ajax 请求取代完全页面刷新，从而与服务器无缝地通信，并更新页面内容。用户可能不会注意到在以另外一种方式完成应用，但是过一段时间之后，他们就会认识到应用变得“更好”了。Ajax 技术也可以在开始设计新 Web 应用时就采用。现在，Ajax 工具包应该在你的开发工具箱里有一席之地了，这些工具能帮助你构建更好的 Web 应用，而且与胖客户应用的表现不相上下，这会让最终用户欣喜万分。

现在你已经了解了 Ajax 技术的有关细节，也知道了 XMLHttpRequest 对象的具体内容，以及如何使用 JavaScript 来处理标准 W3C DOM，从而动态地更新 Web 页面。我们展示了很多常见的例子，在这些情况下都可以使用 Ajax 技术来取代完全页面刷新。

下面稍稍换个角度，从更高的层面来看问题。以前你很可能把 JavaScript 看作是一流的编程语言，而且对它退避三舍。在接下来的几章中，你会了解到如何在 JavaScript 开发中结合公认的软件工程技术，从而尽可能地提高生产力，减少错误，并消除以往 JavaScript 开发中存在的压力。



## 构建完备的 Ajax 开发工具箱

**作**为一名有经验的 Web 应用开发人员,你也许可以熟练地应用某种服务器端技术(或者,应用多种服务器端技术)来构建 Web 应用。我们已经看到,在过去几年中,服务器端技术有了长足的发展,服务器端软件开发越来越容易,也越来越健壮,相比之下,客户端技术基本上被抛在了一边。Ajax 技术的横空出世使这种状况有所改观,因为开发人员现在有了一个更丰富的客户端工具箱,有大量工具可以使用。你可能不习惯使用大量的 HTML、JavaScript 和 CSS,但是如果要实现 Ajax 技术,你就必须这么做。本章将介绍的工具和技术会使得开发 Ajax 应用更为容易。本章不是深入全面的教程,只能作为这些有用工具和技术的快速入门。

### 5.1 使用JSDoc建立JavaScript代码的文档

像其他的许多编程语言一样,在一般的软件开发人员看来,JavaScript 也有一个基本的缺陷:编写(或者重新编写)一个功能通常相对容易,但是要阅读现有的代码并明确它是如何工作的,就不那么轻松了。编写代码时可以适当地增加注释,这样当其他开发人员要理解代码如何工作,特别是要修改代码的功能时,就能减轻他们的负担,节省他们的时间和精力。

Java 语言引入了一个工具,名为 javadoc。这个工具可以根据源代码中的文档注释以 HTML 格式生成 API 文档。所生成的 HTML 文档在任何 Web 浏览器上都能阅读,而且由于它是以 HTML 格式生成的,所以可以在线发布,这样开发人员就能很容易地访问这些文档。以一种可以轻松浏览的格式来提供 API 文档,这种方法使得开发人员不必仔细地查看源代码才能了解某个类或方法会有怎样的行为,以及该如何使用。

JSDoc 是面向 JavaScript 的一个类似的工具 ([jsdoc.sourceforge.net](http://jsdoc.sourceforge.net))。JSDoc 是一个开源工具,是采用 GPL (GNU Public License) 协议发布的。JSDoc 用 Perl 编写,这意味着 Windows 用户必须安装一个 Perl 运行时环境。(而对于大多数 Linux 和 Unix 操作系统,Perl 是其中的一个标准部分)。

### 5.1.1 安装

要使用 JSDoc, Windows 用户必须安装一个 Perl 环境, 如 ActivePerl ([www.activeperl.com](http://www.activeperl.com))。还必须安装一个非标准的 Perl 模块, 名为 HTML::Template ([www.cpan.org](http://www.cpan.org))。JSDoc 项目网页提供了有关说明, 如果需要帮助可以参考。

JSDoc 发布为一个压缩的 tarball。要安装 JSDoc, 你只需从 JSDoc 项目网页下载 tarball, 把它解开到指定的目录, 进入 JSDoc 目录, 输入以下命令, 就能测试 JSDoc 了:

```
perl jsdoc.pl test.js
```

JSDoc 将所得到的 HTML 文件保存到名为 `js_docs_out` 的目录。打开这个文件夹中的 `index.html` 文件, 就可以浏览根据 `test.js` 文件生成的文档。

### 5.1.2 用法

既然对 JSDoc 已经有所了解, 你可能想知道如何使用 JSDoc 来为你的 JavaScript 代码生成文档。表 5-1 列出了可以创建 HTML 文档的一些特殊 JSDoc 标记。这些标记对于曾在 Java 代码中编写过 javadoc 注释的人员并不陌生。包含在生成文档中的每个注释块都必须以`/**`开头, 并以`*/`结束。

表 5-1 JSDoc 命令属性

命令名	描述
<code>@param</code>	指定参数名和说明来描述一个函数参数
<code>@argument</code>	
<code>@return</code>	描述函数的返回值
<code>@returns</code>	
<code>@author</code>	指示代码的作者
<code>@deprecated</code>	指示一个函数已经废弃, 而且在将来的代码版本中将彻底删除。要避免使用这段代码
<code>@see</code>	创建一个 HTML 链接, 指向指定类的描述
<code>@version</code>	指定发布版本
<code>@requires</code>	创建一个 HTML 链接, 指向这个类所需的指定类
<code>@throws</code>	描述函数可能抛出的异常的类型
<code>@exception</code>	
<code>{@link}</code>	创建一个 HTML 链接, 指向指定的类。这与 <code>@see</code> 很类似, 但 <code>{@link}</code> 能嵌在注释文本中
<code>@fileoverview</code>	这是一个特殊的标记。如果在文件的第一个文档块中使用这个标记, 则指定该文档块的余下部分将用来提供这个文件的概述
<code>@class</code>	提供类的有关信息, 用在构造函数的文档中
<code>@constructor</code>	明确一个函数是某个类的构造函数
<code>@type</code>	指定函数的返回类型
<code>@extends</code>	指示一个类派生了另一个类。JSDoc 通常自己就可以检测出这种信息, 不过, 在某些情况下则必须使用这个标记

续表

命令名	描述
@private	指示一个类或函数是私有的。私有类和函数不会出现在 HTML 文档中，除非运行 JSDoc 时提供了--private 命令行选项
@final	指示一个值是常量值。要记住 JavaScript 无法真正保证一个值是常量
@ignore	JSDoc 忽略有这个标记的函数

JSDoc 发布包中包括一个名为 test.js 的文件，这是一个很好的参考例子，可以从中了解如何使用 JSDoc。你应该记得，第一次测试 JSDoc 安装是否成功时就是根据这个文件来创建文档文件的。如果对如何使用 JSDoc 标记还有疑问，可以参考这个文件。

代码清单 5-1 是一个小示例，展示了 JSDoc 的用法。jsDocExample.js 定义了两个类：Person 和 Employee。Person 类有一个属性 name，还有一个方法 getName。Employee 类继承自 Person 类，并增加了 title 和 salary 属性，另外还增加了一个方法 getDescription。

### 代码清单 5-1 jsDocExample.js

```
/**
 * @fileoverview This file is an example of how JSDoc can be used to document
 * JavaScript.
 *
 * @author Ryan Asleson
 * @version 1.0
 */

/**
 * Construct a new Person class.
 * @class This class represents an instance of a Person.
 * @constructor
 * @param {String} name The name of the Person.
 * @return A new instance of a Person.
 */
function Person(name) {
    /**
     * The Person's name
     * @type String
     */
    this.name = name;
    /**
     * Return the Person's name. This function is assigned in the class
     * constructor rather than using the prototype keyword.
     * @returns The Person's name
     * @type String
     */
    this.getName = function() {
```

```
        return name;
    }
}

/***
 * Construct a new Employee class.
 * @extends Person
 * @class This class represents an instance of an Employee.
 * @constructor
 * @return A new instance of a Person.
 */
function Employee(name, title, salary) {
    this.name = name;

    /**
     * The Employee's title
     * @type String
     */
    this.title = title;

    /**
     * The Employee's salary
     * @type int
     */
    this.salary = salary;
}

/* Employee extends Person */
Employee.prototype = new Person();

/***
 * An example of function assignment using the prototype keyword.
 * This method returns a String representation of the Employee's data.
 * @returns The Employee's name, title, and salary
 * @type String
 */
Employee.prototype.getDescription = function() {
    return this.name + " - "
        + this.title + " - "
        + "$" + this.salary;
}
```

虽然不像 JSDoc 发布包中的 test.js 文件那么完备,这个示例同样很好地展示了 JSDoc 最常见的用法(见图 5-1)。@fileoverview 标记提供了 jsDocExample.js 的概述。@class 标记描述了两个类, @constructor 标记将适当的函数标记为对象的构造函数。@param 标记描述了函数的输入参数, @returns 和@type 标记描述了函数的返回值和返回类型。这些标记是你最有可能用到的,而且对于浏览文档的其他开发人员,这些标记也最有用。

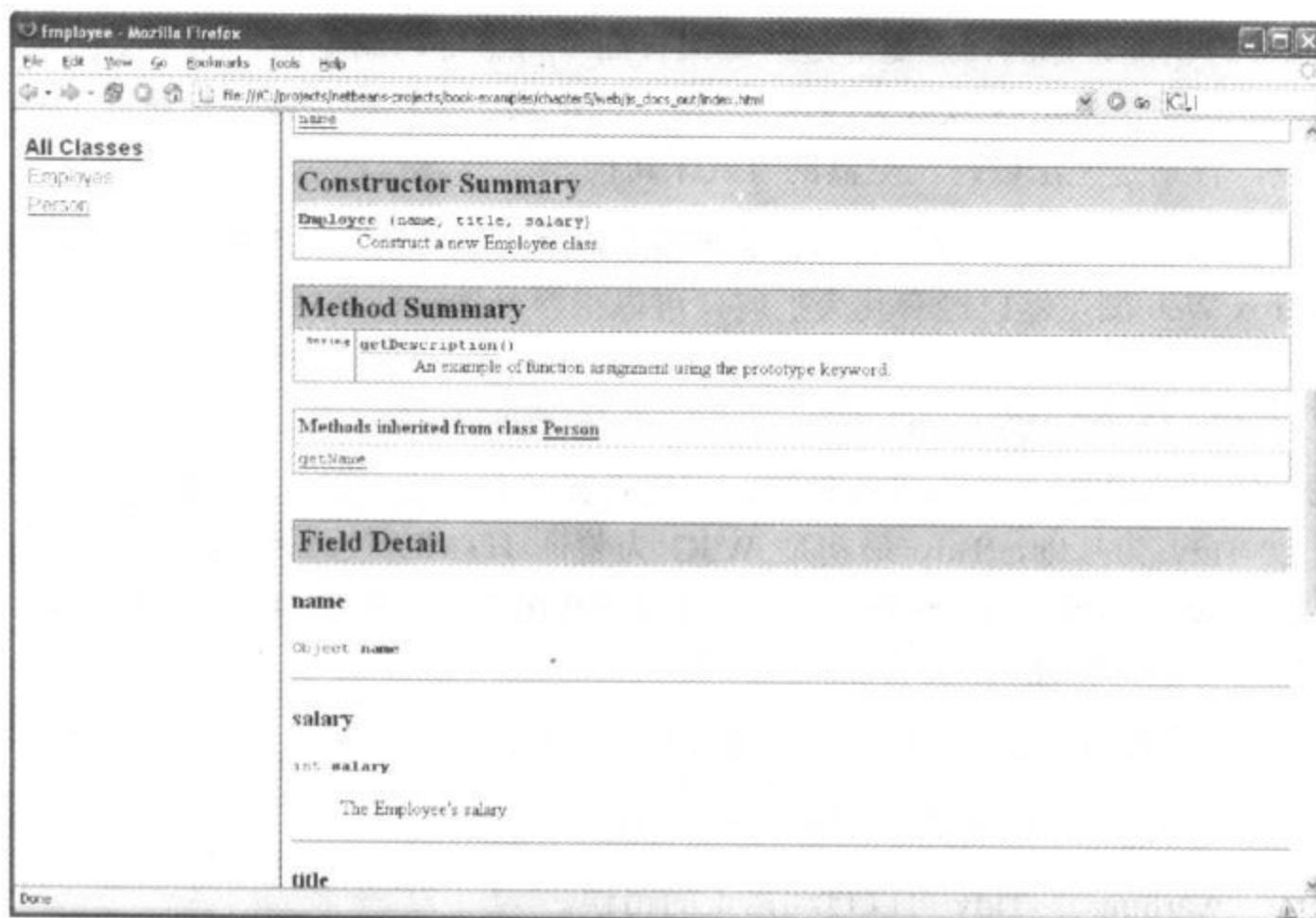


图 5-1 JSDoc 根据 jsDocExample.js 文件生成的文档

## 5.2 使用Firefox扩展验证HTML内容

当前的浏览器都能很好地实现 W3C DOM 标准。只要创建的内容能遵循标准 HTML 或 XHTML，就几乎能得到所有浏览器的支持。

不过通常说起来简单，做起来就不那么容易了。不同于 C++ 或 Java 这样的编译语言，HTML 没有编译器可以将人可读的代码翻译为机器可读的二进制代码，要由 Web 浏览器将人可读的 HTML 或 XHTML 代码解释成 DOM 的内部表示，并适当地将这个内容展现在屏幕上。

20 世纪 90 年代末，浏览器之争使得浏览器开发商（如 Microsoft 和 Netscape）纷纷增加了一些专用的 HTML 标记，以扩大自己的市场份额。出于这个原因，再加上 HTML 没有严格的编译器，这就导致了大量非标准 Web 页面的出现。当前的浏览器尽管支持最新的 W3C 标准，不过也会尽可能地“通融”写得不好的 HTML 页面。根据 HTML 页面的 doctype（如果有的话），大多数浏览器都有两种呈现模式：strict（严格）和 quirks（怪异）。当 doctype 指示 Web 页面应遵循某个 W3C 推荐规约（如 HTML 4.1 或 XHTML 1.0）编写时，Web 浏览器就会使用 strict 模式；当没有 doctype，或者页面与指定的 doctype 有很多冲突时，Web 浏览器就使用 quirks 模式。

作为一名开发人员，应当尽力创建遵循某种 W3C 标准的页面。这么做不仅使你的 Web 页面在所有现代 Web 浏览器上可访问，而且由于浏览器可以根据 HTML 代码创建准确的

DOM 表示，这也能让你的日子更好过。如果页面写得不好，浏览器可能无法创建 DOM 的准确表示，就会使用 quirks 模式来呈现页面。DOM 表示不正确，就很难通过 JavaScript 来访问和修改，特别是无法以跨浏览器的方式来访问。

由于 HTML 没有严格的编译器，怎么确保你写的 HTML 代码遵循 W3C 标准呢？幸运的是，Firefox Web 浏览器已经有几个扩展，可以很容易地验证你的 Web 页面。

### 5.2.1 HTML Validator

HTML Validator<sup>1</sup>是一个 Firefox 扩展，它能查找并标志出 HTML 页面上的错误。HTML Validator 以 Tidy 为基础，Tidy 最初是 W3C 为验证 HTML 代码开发的一个工具。HTML Validator 把 Tidy 工具嵌入在 Firefox 中，这样就能在浏览器中验证页面的源代码，而不必把代码发给第三方进行验证。

Tidy 会查找 HTML 错误，并把这些错误归为 3 类：

- 错误 (error)：Tidy 无法修正或理解的问题。
- 警告 (warning)：Tidy 可以自动修正的错误。
- 可访问性警告 (accessibility warning)：这些 HTML 警告对应 W3C Web 可访问性计划 (Web Accessibility Initiative, WAI) 定义的 3 个优先级。

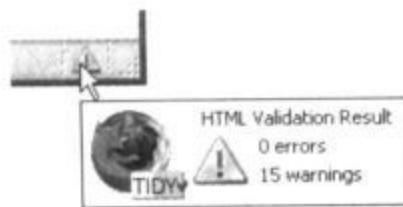


图 5-2 HTML Validator 使用状态条上的图标来总结页面上的错误

HTML Validator 在浏览器的右下角显示页面的状态以及错误个数，从而能在开发周期中提供很快的反馈(见图 5-2)。

如果选择 View→Page Source 菜单项查看 Web 页面的源代码，HTML Validator 还能提供更多的帮助。Firefox 的 view-source (查看源代码) 窗口会正常打开，不过还将启用 HTML Validator，这个窗口中包括两个新的窗格 (见图 5-3)。

HTML Errors and Warnings (HTML 错误和警告) 窗格列出在页面中找到的所有错误。点击列表中的任何错误项，源代码主窗口就会显示 HTML 源代码中有问题的代码行。Help (帮助) 窗格详细描述了这个问题，并提供一些修正这个问题的建议。

Firefox 的 view-source 窗口的底部包括一个 Clean up the Page (清理页面) 按钮。点击这个按钮后将打开一个窗口，这个窗口中显示的内容能进一步帮助你修正页面上的错误 (见图 5-4)。Clean up the Page 窗口打开后，窗口的最上面有 4 个标签：Cleaned Html (清理后的 HTML)、Original Html (原来的 HTML)、Cleaned Browser (清理后的浏览器) 和 Original Browser (原来的浏览器)。

<sup>1</sup>. <https://addons.mozilla.org/extensions/moreinfo.php?application=firefox&category=Developer%20Tools&numpg=10&id=249>。

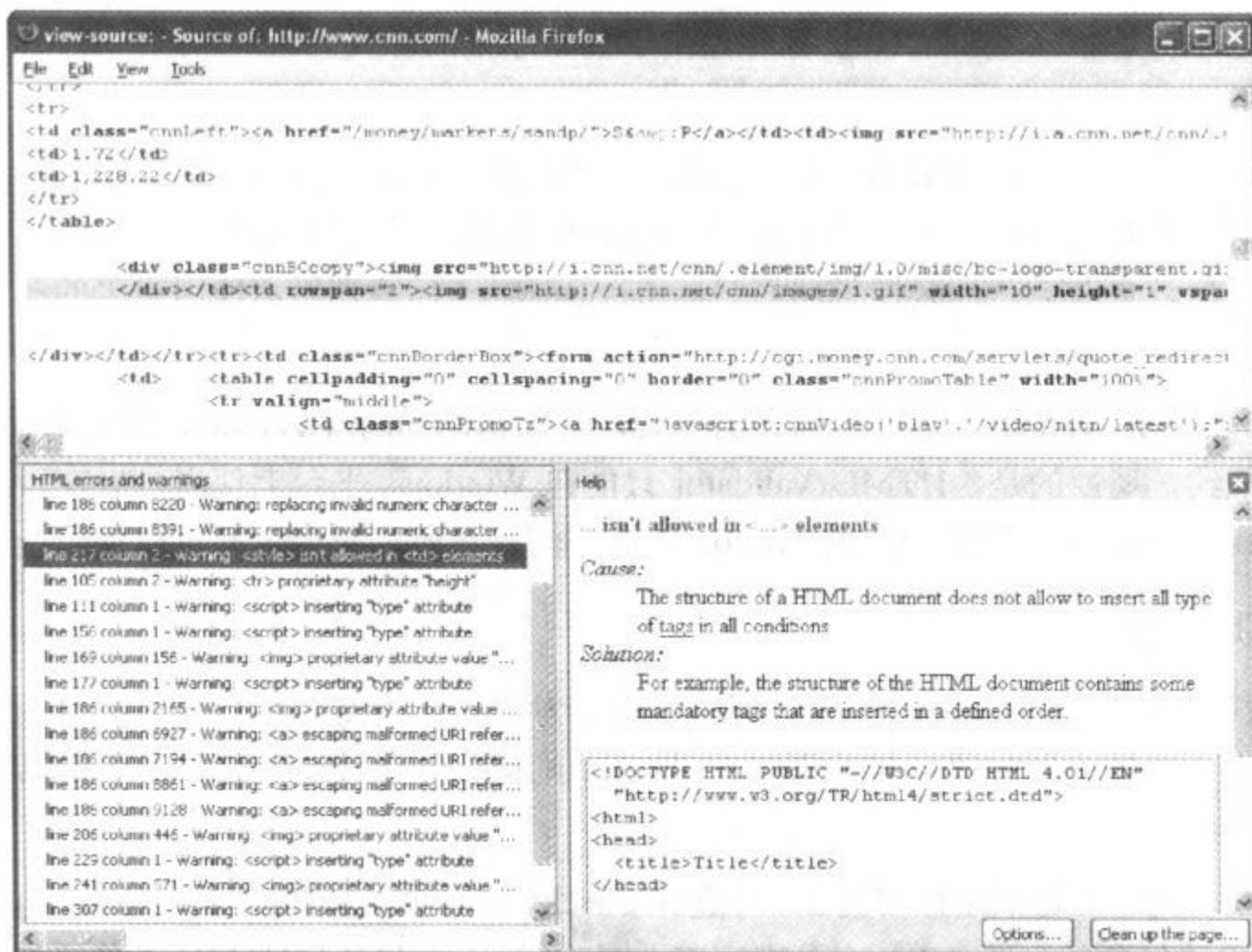


图 5-3 当查看页面的源代码时，HTML Validator 会列出 HTML 源代码中的错误，并提出修正问题的建议

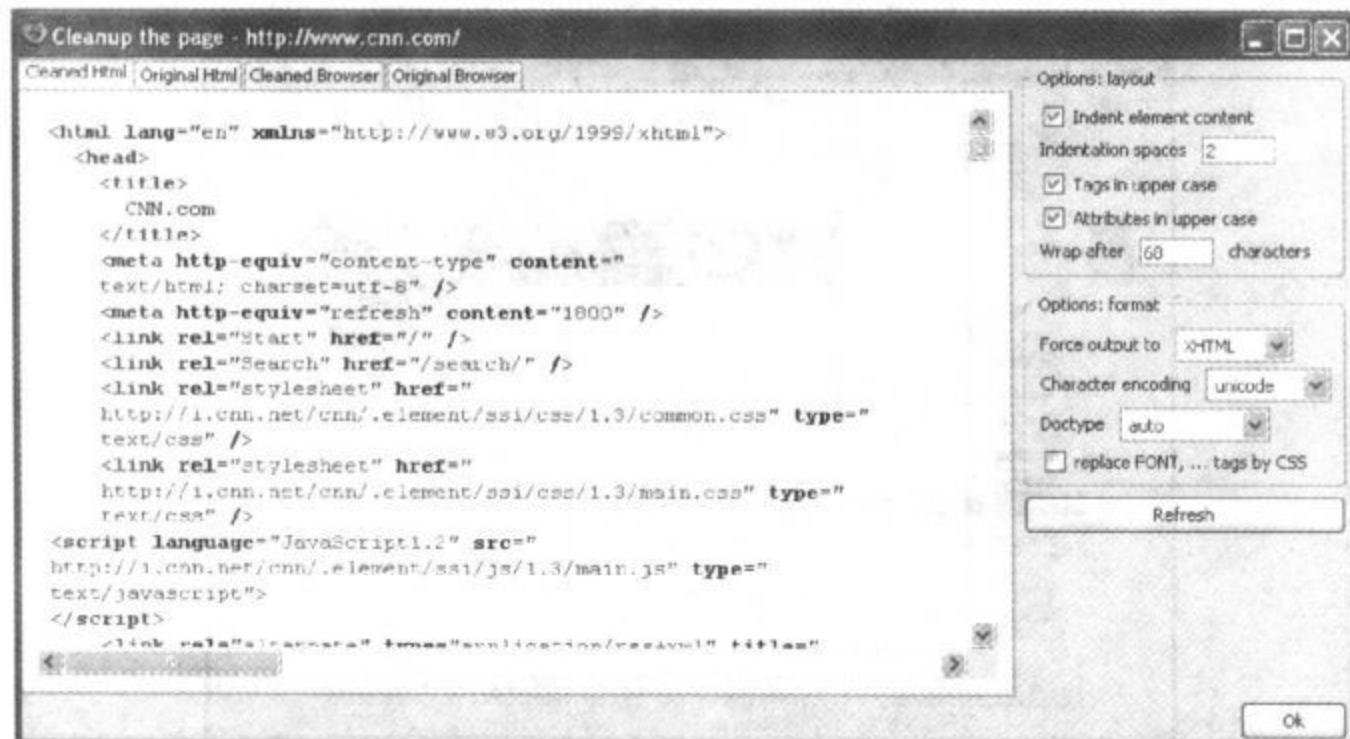


图 5-4 HTML Validator 的 Cleanup the Page 对话框给出了新的源代码，这个源代码已经修正了在原来的源 HTML 代码中发现的错误

Cleaned Html 标签对 Web 开发人员最有用。这个标签列出了通过 HTML Validator 进行修正后的页面源代码。HTML Validator 会尽其所能自动修正页面上的所有错误，修正后的输

出就列在这个标签下。Original Html 标签列出了页面原来的源代码，也就是在 HTML Validator 处理之前的形式。

有时，修正页面上的 HTML 错误可能会改变浏览器呈现页面的方式，这可能是我们需要的，也可能不是我们希望的。Cleaned Browser 标签显示了使用 HTML Validator 提供的已修正源代码后，页面会是什么样子，而 Original Browser 标签则显示使用原来的源代码时相应的页面。

总之，HTML Validator 是一个强大的工具，可以帮助你清理 HTML，使之遵循 W3C 标准和推荐规约。遗憾的是，HTML Validator 只能在 Windows 平台上使用。好在，还有另一个与 HTML Validator 有类似功能的 Firefox 扩展，而且在所有平台上都可以使用。

### 5.2.2 Checky

Checky<sup>1</sup>是另一个 Firefox 扩展，可以帮助开发人员编写更好的 HTML 页面。HTML Validator 在本地验证源代码，与此不同，Checky 则把页面源代码发送给不同的第三方网站来完成 HTML 验证。

在 Firefox 中，右键点击任何页面，并选择 Checky 菜单项（见图 5-5）就可以访问 Checky。Checky 菜单项包含多个子菜单项，分别完成不同的任务。HTML/XHTML 菜单项列出了可以提供 HTML 验证服务的多个网站，点击此列表中的任何网站就会在 Firefox 中打开一个新的标签，该标签指向这个验证网站。Checky 自动地填入进行验证的页面地址，并开始验证过程。

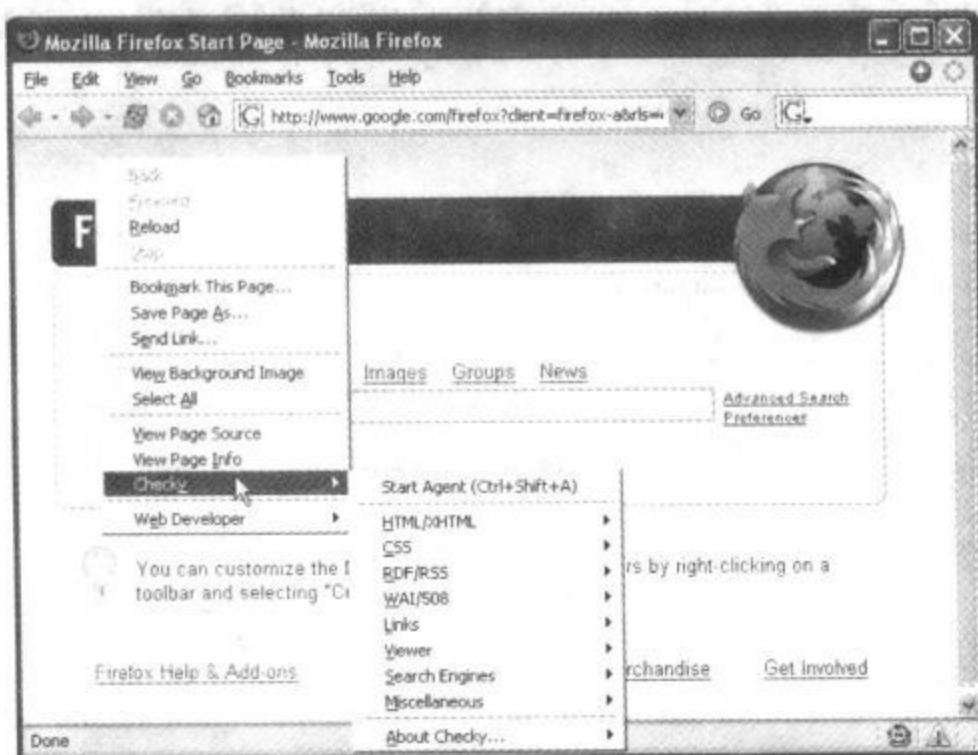


图 5-5 可以在 Firefox 中通过上下文菜单来访问 Checky

1. <https://addons.mozilla.org/extensions/moreinfo.php?application=firefox&category=Developer%20Tools&numpg=10&id=165>。

如图 5-6 所示，要验证的代码必须能够在因特网上公开得到，这样验证网站才能访问到其 HTML。

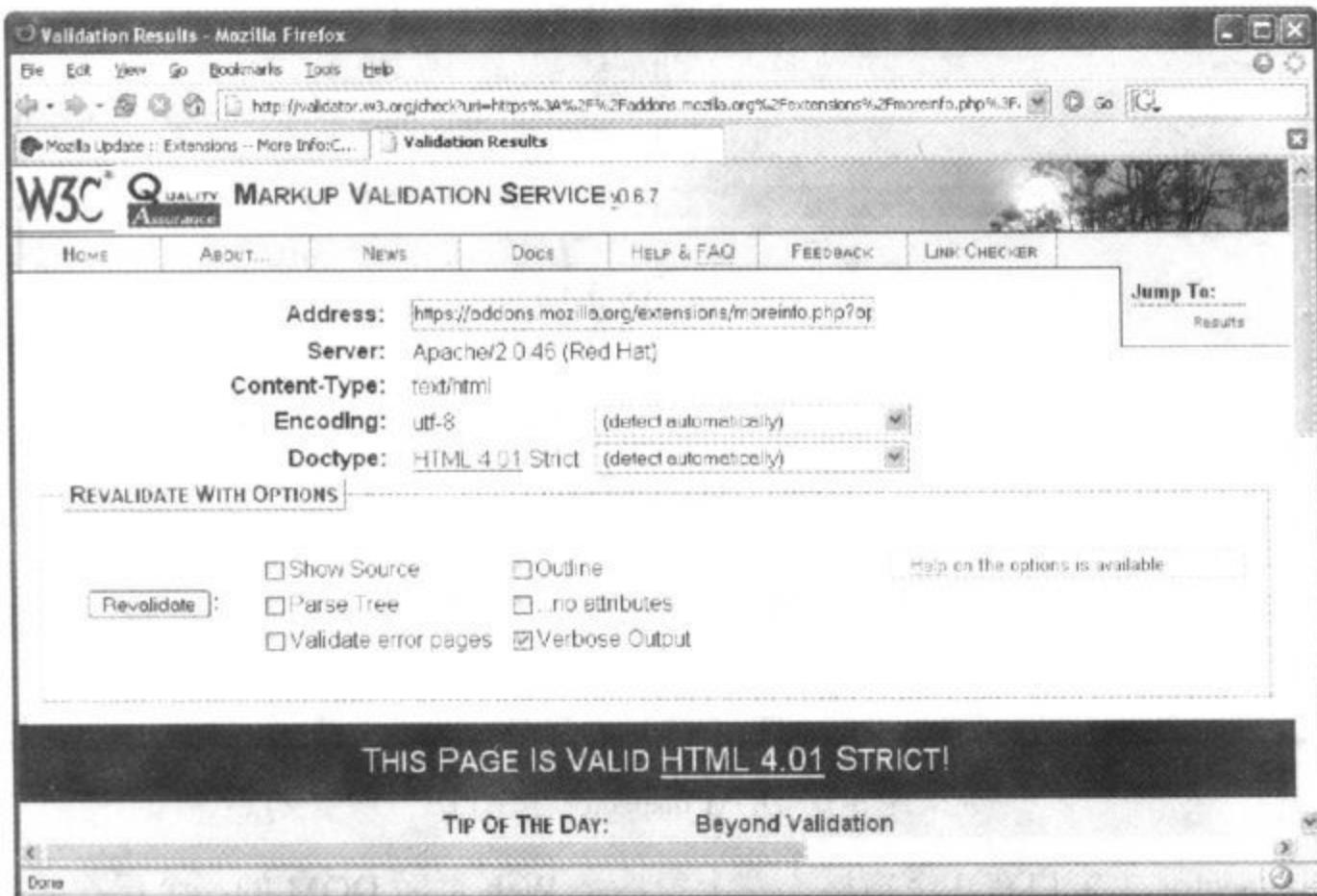


图 5-6 通过 Checky 访问，使用 W3C 的在线验证器得到的 HTML 验证结果

通过 Checky 还可以访问其他网站，这些网站不仅限于验证 HTML。Links（链接）菜单会列出能验证页面上所有链接的网站，以确保所有链接连接的 URL 都确实存在。CSS 菜单列出的网站能验证页面上使用的所有 CSS 文件，以确保这些 CSS 文件遵循标准 CSS 规则。

你应当花点时间来测试 Checky 提供的验证网站。通过使用这些验证工具，可以使你的代码更能与标准兼容，而且可以减少手工跟踪问题花费的时间。

### 5.3 使用DOM Inspector搜索节点

在 Mozilla Suite 和 Firefox 浏览器中打包了一个 DOM Inspector 工具。如果利用 DOM Inspector，则可以查看 Web 页面的结构化表示，甚至能搜索某些特定的节点，并自动更新 DOM 中的节点。在 Firefox 中，你可以通过 Tools 菜单项来访问 DOM Inspector。要使用 DOM Inspector 来检查一个 Web 页面，需要在文本框中输入所需的 URL，并点击 Inspect 框；或者也可以从 File→Inspect a Window 菜单选择一个窗口，这就会列出当前在浏览器中打开的 Web 页面（见图 5-7）。

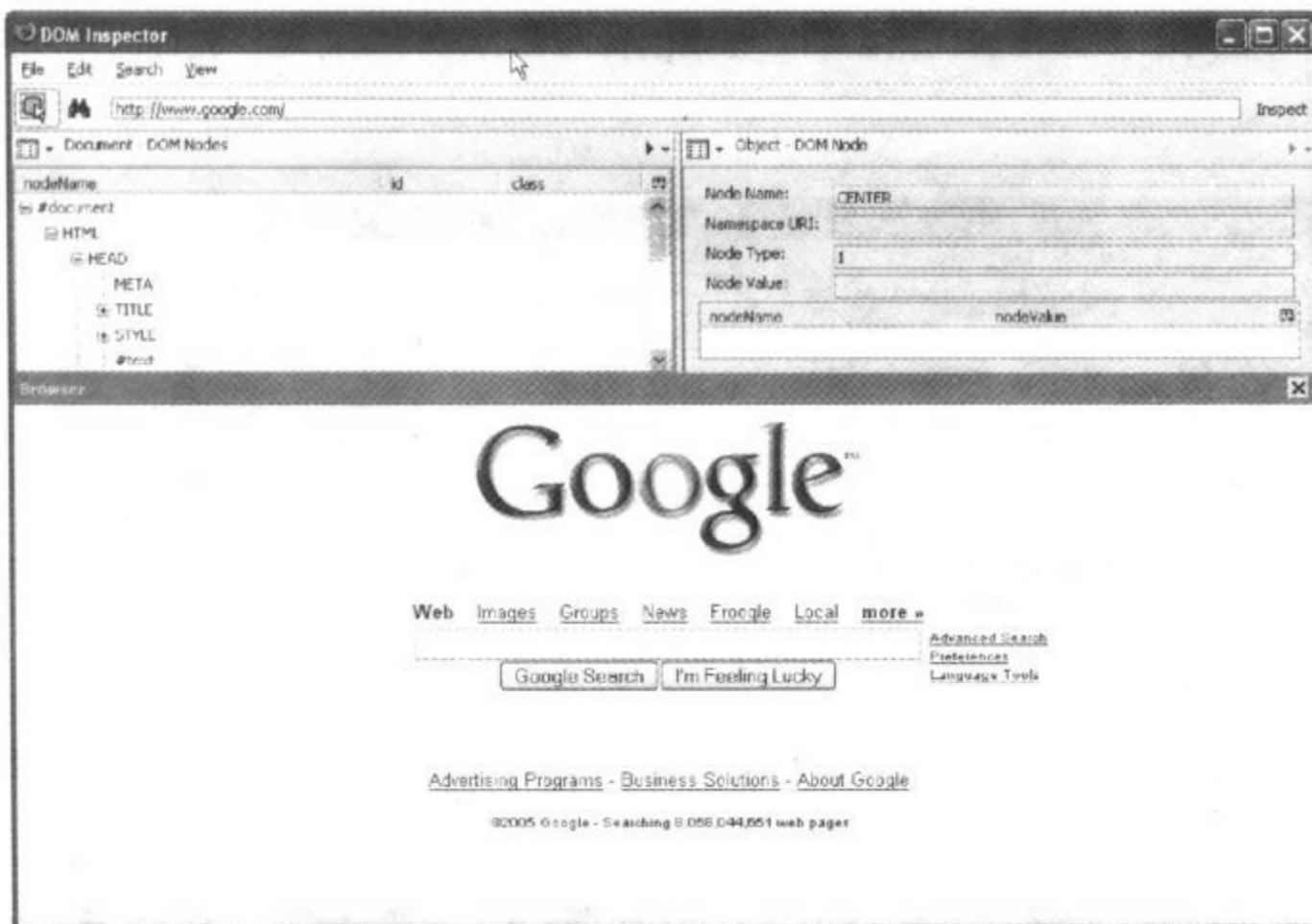


图 5-7 DOM Inspector 主窗口

DOM Inspector 主窗口有 3 个窗格。左上窗格是 Web 页面 DOM 的一个层次结构视图。根元素往往是文档本身，Web 页面中的每个节点都列在这个根元素下面。对于大多数 Web 页面，根节点几乎都是 HTML。如果在结构化视图窗格中选择了一个节点，右上窗格会给出这个节点的详细信息。如果主窗口下部没有打开一个浏览器窗口，则可以选择 View→Browser 菜单项打开一个浏览器窗口。

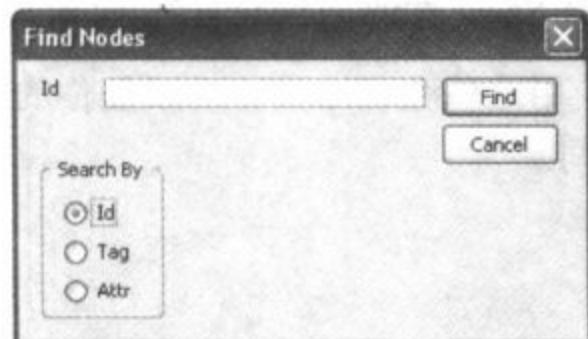


图 5-8 DOM Inspector 的 Find Nodes 对话框

DOM Inspector 是一个功能强大的工具，利用这个工具，你可以快速地遍历给定 Web 页面的结构和视图，并修改 Web 页面 DOM 中的各个节点。通常，可以通过结构化视图中的菜单项手工地查找节点，也可以使用 Search → Find Nodes 菜单项来查找各个节点。利用这个搜索功能，你可以根据 ID 属性、标记名或属性名和值来查找节点（见图 5-8）。

要在 DOM Inspector 中查找节点，最容易的方法是使用鼠标。在结构化视图中查找一个节点时，可以选择 Search→Select Element by Click 菜单项，并点击浏览器窗口中的这一项。所选项会以红色边框突出显示，而且在结构化视图窗格中选中相应的节点。

一旦在结构化视图窗格中选中一个特定节点，你就可以开始检查和修改它的属性了。例

如，可以右键点击一个节点，从上下文菜单中选择 Cut，再选择结构化视图窗格中的另一个节点，右键点击，从上下文菜单中选择 Paste，这样就能在 DOM 中将所选节点有效地从一处移到另一处。图 5-9 显示了使用这种方法可以将 Google 搜索页面上的主图片移到页面的另一个部分。

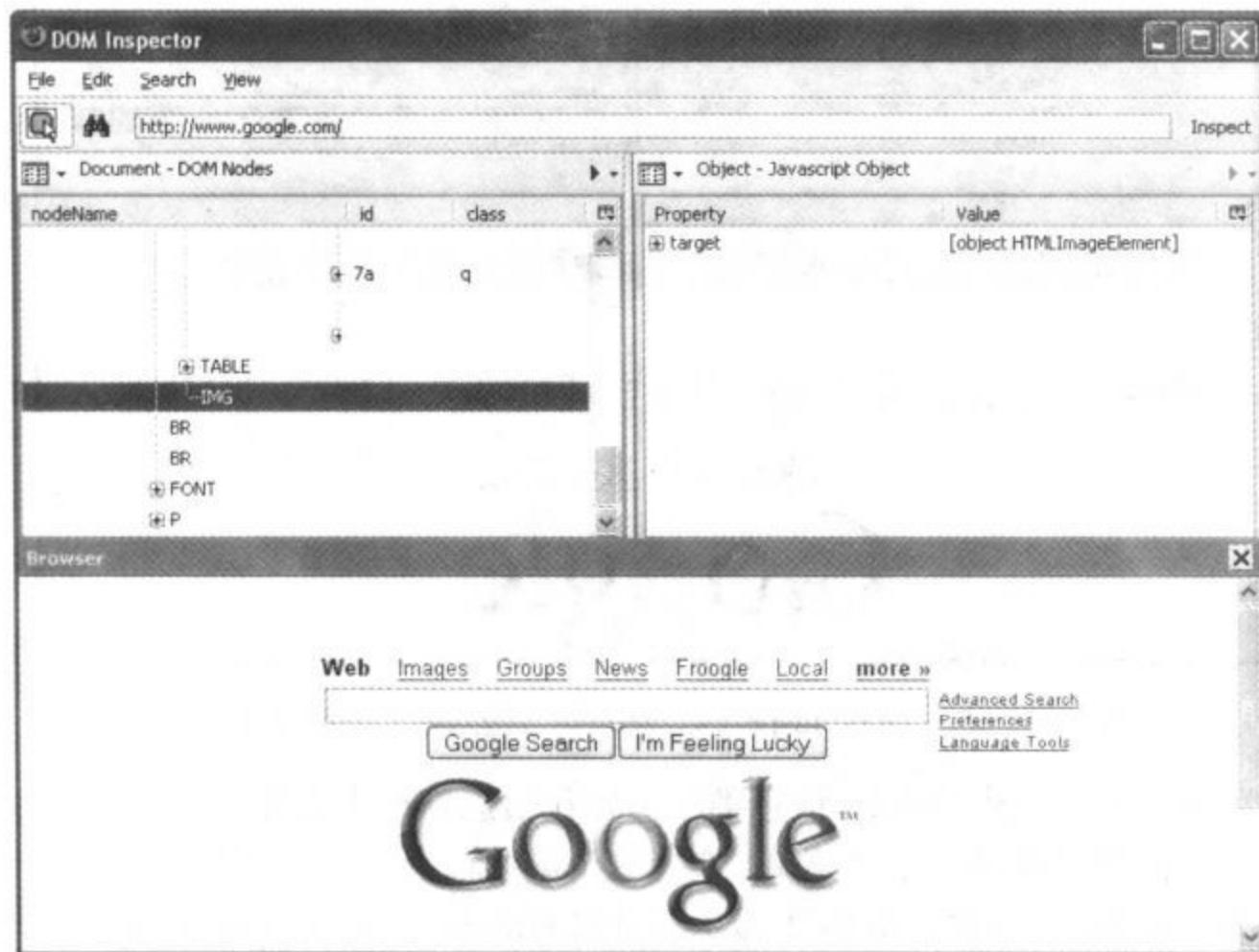


图 5-9 使用 DOM Inspector 移动 Google 搜索页面主图片的结果

你还能在右上方的信息窗格中发现更多功能。对于结构化视图窗格中选中的节点，这个窗口会显示有关该节点的各类信息。上方标题区中的下拉列表图标用于切换信息的类型，可选的信息类型包括 DOM Node、Box Model、XBL Bindings、CSS Style Rules、Computed Style 和 JavaScript Object。当使用 Mozilla 的 XML 用户界面语言（XML User Interface Language, XUL）工具包开发应用时，Box Model 和 XBL Bindings 信息类型更有用。

DOM Node 信息类型会显示有关节点的基本信息，如其标记名、节点值，以及节点的属性。右键点击一个节点会显示一个上下文菜单，选择其中的 Edit 菜单项就可以修改节点属性的值。例如，可以选择一个 font（字体）节点，修改 size（大小）属性。如图 5-10 所示，使用这种技术可以增大 Google 搜索页面中输入框上方的字体大小。

JavaScript Object 信息类型会列出所选节点可用的 DOM 属性和方法。如果要确定一个特定 DOM 节点有哪些可用的属性和方法，这就是一个很有用的特性。例如，除了一般的正常方法外（如 appendChild），对于表格节点还会列出诸如 insertRow 和 deleteRow 的方法。

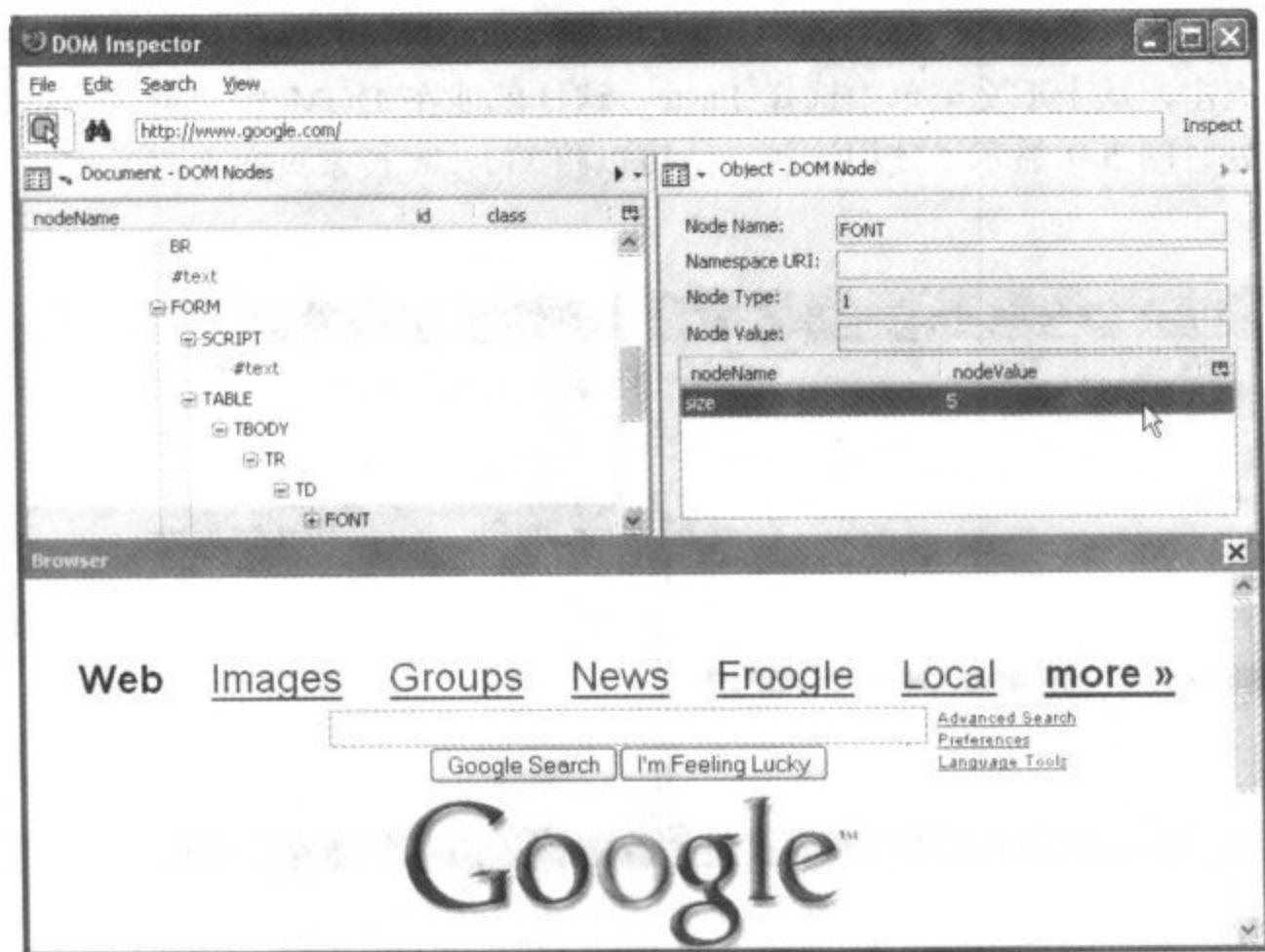


图 5-10 使用 DOM Inspector，动态修改输入框上方的字体大小

如果设置为 JavaScript Object 信息类型，则在信息窗格中右键点击就会显示一个带有 Evaluate JavaScript 菜单项的上下文菜单。选择这个菜单项会弹出一个窗口，可以针对所选节点计算一个 JavaScript 表达式。图 5-11 显示了针对 Google 搜索页面的 body（体）节点打开的 JavaScript 计算窗口，可以看到，如果执行计算窗口所示的 JavaScript 表达式，就会在页面的最后追加指定的文本。注意 target 作为变量名，它指示所选的节点，在这里就是 body 元素。

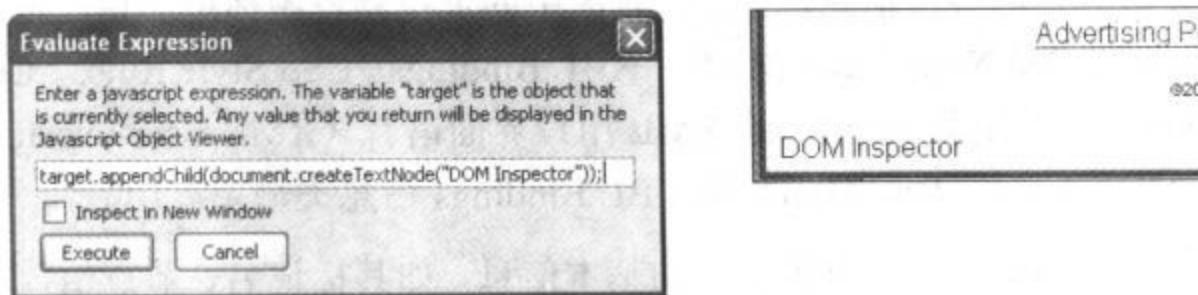


图 5-11 使用 JavaScript 计算窗口在页面的体中动态增加一个文本节点（左图），以及浏览器窗格中的结果（右图）

CSS Style Rules 和 Computed Style 信息类型会显示所选节点样式规则的有关信息。Computed Style 信息类型会列出浏览器的呈现引擎所看到的所有与样式相关的属性，包括使用 style 属性显式设置的样式，在外部 CSS 文件中指定的样式，或者从父节点继承的样式。

前面已经简要地了解了 DOM Inspector 的特性，可以想像，在你的开发环境中，这必将

是一个非常有用的工具。你可以使用 DOM Inspector 来检查通过 `document.createElement` 方法动态创建的 DOM 节点，以确保具有所需的属性值。如果一个特定节点没有应用你希望的样式规则，也可以使用 DOM Inspector 来找出原因。随着越来越熟悉 DOM Inspector 的功能，你肯定会发现 DOM Inspector 在 Web 开发过程中将是一个举足轻重的强大工具。

## 5.4 使用JSLint完成JavaScript语法检查

JSLint 是一个 JavaScript 验证工具 ([www.jslint.com](http://www.jslint.com))，可以扫描 JavaScript 源代码来查找问题。如果 JSLint 发现一个问题，JSLint 就会显示描述这个问题的消息，并指出错误在源代码中的大致位置。有些编码风格约定可能导致未预见的行为或错误，JSLint 除了能指出这些不合理的约定，还能标志出结构方面的问题。尽管 JSLint 不能保证逻辑一定正确，但确实有助于发现错误，这些错误很可能导致浏览器的 JavaScript 引擎抛出错误。

JSLint 定义了一组编码约定，这比 ECMA 定义的语言更为严格。这些编码约定汲取了多年来的丰富编码经验，并以一条年代久远的编程原则作为宗旨：能做并不意味着应该做。JSLint 会对它认为有风险的编码实践加标志，另外还会指出哪些是明显的错误（见图 5-12），从而促使你养成好的 JavaScript 编码习惯。

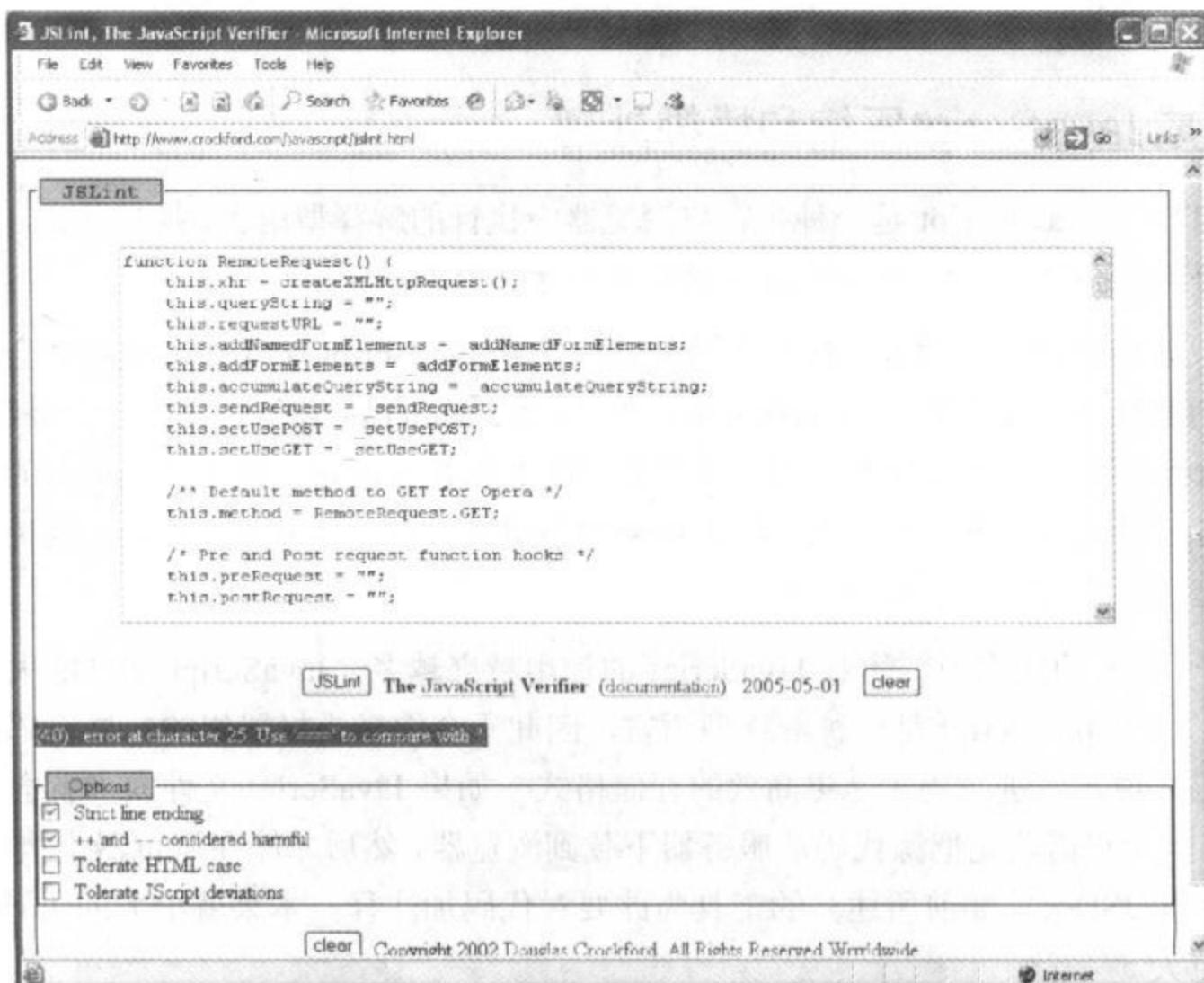


图 5-12 JSLint 会检查错误以及不好的编码风格，以此提供 JavaScript 验证

JSLint 可能会把一些结构方面的错误标志为可疑的编码实践，以下列出了其中一部分（完整的列表可以参考 JSLint 的文档）。

- JSLint 要求所有代码行都以分号结束。尽管 JavaScript 确实允许将换行符作为行结束符，但一般认为这种做法是不明确的，而且是不好的编码风格。
- 使用 if 和 for 的语句必须使用大括号把语句块括起来。
- 不同于其他编程语言，在 JavaScript 中，块不会作为变量的作用域。JavaScript 只支持函数级作用域。因此，JSLint 只接受作为 function、if、switch、while、for、do 和 try 语句一部分的块，其他的块都会标志为错误。
- var 只能声明一次，而且在使用之前必须声明。
- JSLint 会把出现在 return、break、continue 或 throw 语句后面的代码标志为不可达的代码。这些语句后面必须紧跟一个结束大括号。

对于 JavaScript 程序员新手来说，JSLint 是一个非常好的工具，因为它会教你一些好的 JavaScript 编码实践。由于 JSLint 能把可能导致逻辑错误或其他未预见行为的部分标出来，因此可以减少调试时间。如果你调试一段 JavaScript 代码时遇到困难，可以试试 JSLint。

## 5.5 完成JavaScript压缩和模糊处理

我们都知道，JavaScript 是一种在客户浏览器中执行的解释型语言。换句话说，JavaScript 会以明文下载到浏览器，再由浏览器根据需要执行这个 JavaScript 代码。

用户只要使用浏览器的查看源代码功能就能读到 JavaScript 源代码，该功能会显示出页面的完整 HTML 标记，包括所有 JavaScript 块。即使 JavaScript 源代码放在一个外部文件中，并用 script 标记的 src 属性来引用，用户也可以下载并阅读它。由于查看页面的人都能得到 JavaScript 源代码，所以不要把专用或机密的逻辑算法放在 JavaScript 中。这种逻辑最好放在服务器上，在那儿会更安全一些。

在基于 Ajax 的应用中，随着 JavaScript 的使用越来越多，JavaScript 文件的大小可能会成为问题。由于 JavaScript 是一种解释型语言，因此不会编译为机器级的二进制码，而对于可执行代码来说，二进制码才是更高效的存储格式。如果 JavaScript 文件太多就会使应用的速度减慢，因为它需要先把源代码从服务器下载到浏览器，然后才能在浏览器上执行。另外，如果使用诸如 JSDoc（如前所述）的工具为此要对代码加注释，本来就很大的 JavaScript 代码会变得更大。

你可能看到了，JavaScript 缺少二进制的可执行包，这会带来两个问题：安全性差，以

及需要下载大量的源代码。有没有办法避开这些问题呢？

JavaScript 日益普及，因此也产生了许多工具，这些工具有助于解决这些问题。最简单的压缩工具会简单地去除 JavaScript 源代码中的所有注释和换行符，这样可以减小下载的源代码的大小。删除注释行和换行符能使 JavaScript 文件的大小缩小 30%甚至更多，这要依具体情况而定。需要说明的是，JavaScript 源代码中的所有语句必须正确地以分号结束，只有这样才能用这种工具对源代码进行压缩。如果没有做到这一点，你就会接收到错误或者未预料行为的消息。所以，在压缩 JavaScript 源代码之前，一定要使用 JSLint 确保所有语句都以分号结束！

还有一些工具则更进一步，可以提供模糊服务。模糊（Obfuscation）是一种过程，指全面扫描源代码，将字段和函数原来的名字改成经编码的无意义的名字，以防止其他人了解源代码的含义和内部工作。对于能编译为机器级二进制指令的语言来说（如 C++），一般不需要这种模糊处理。即使 Java 和 C#这样能够编译为中间字节码而不是二进制指令的现代语言，也需要模糊工具来保证最大程度的安全。JavaScript 作为一个完全解释型语言同样需要这样一种工具。

有一个能同时提供压缩和模糊服务的免费工具，就是 MemTronic 的 HTML/JavaScript Cruncher-Compressor ([hometown.aol.de/\\_ht\\_a/memtronic/](http://hometown.aol.de/_ht_a/memtronic/))。这个工具支持多个层次的 JavaScript 压缩。最低层次的压缩在这个工具中称为挤压（crunching），只是简单地删除所有注释和换行符。这个工具的相关文档称，这样可以节省 20%~50%的带宽。使用“crunch”模式，可以看到 JavaScript 文件的大小缩小了 30%。

最高层次的压缩在这个工具中称为压缩（compressing），是用一种真正的压缩机制实际压缩 JavaScript 源代码，并向文件增加自动解压缩功能。这个工具的相关文档称，当使用这种模式时，带宽可以节省 40%~90%，而且压缩后的输出已经在当前版本的 IE、Netscape、Mozilla 和 Opera 等浏览器上成功通过测试。使用同一个 JavaScript 文件，应用“compressing”模式和“crunch”模式进行测试，发现使用“compress”模式使得文件大小的缩小幅度超过了 65%（见图 5-13）。

在写本书时，MemTronic 工具的文档称，JavaScript 的模糊工具还不算完备。不过，可以看看图 5-13 所示的输出窗口，这里显示了对 JavaScript 文件执行“压缩”操作的结果。这个输出中包含了许多奇怪的字符，难于阅读。尽管这可能不是真正意义上的模糊处理，但确实足以防止有不良企图的用户查看（甚至窃取）你的 JavaScript 源代码。

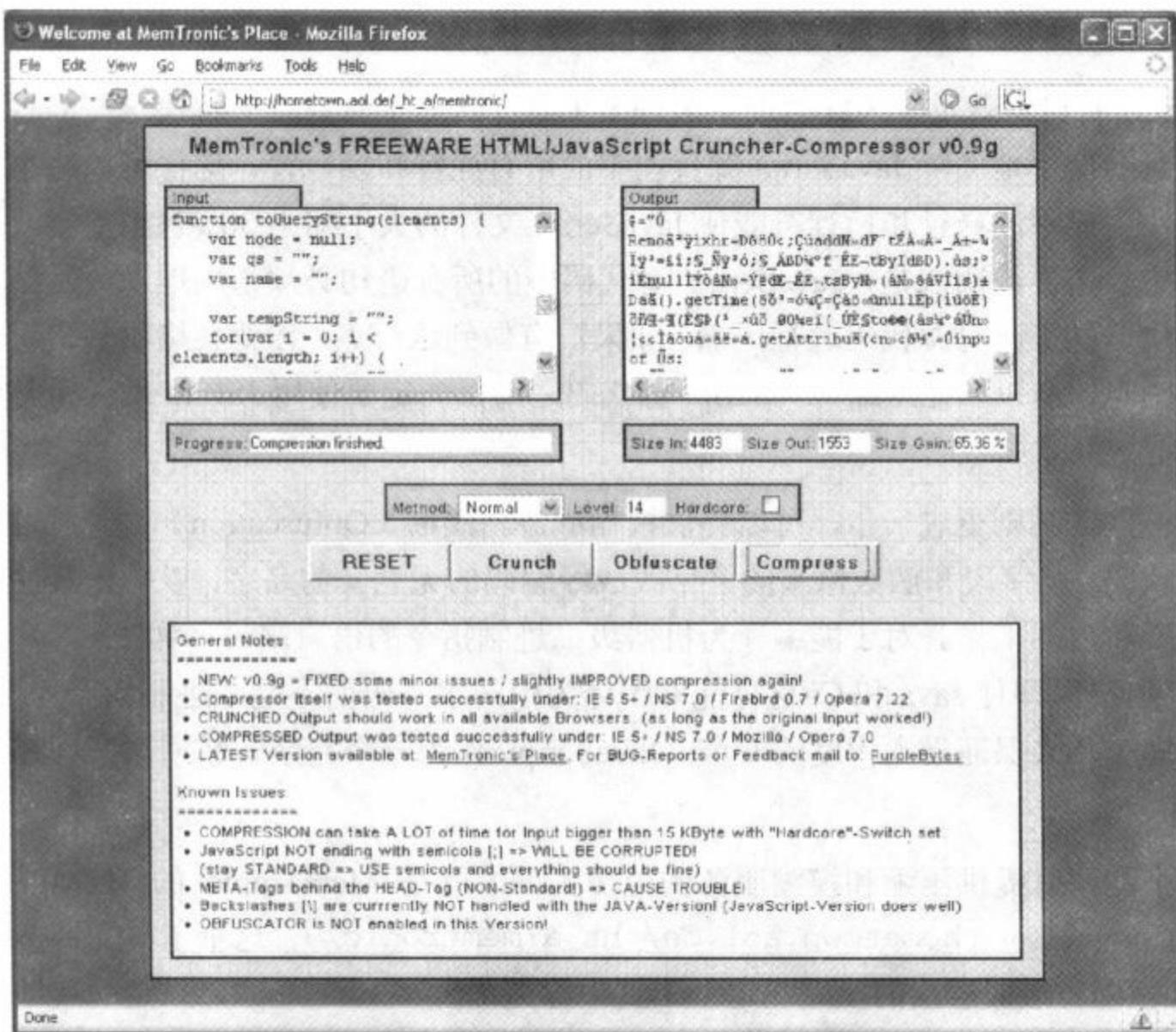


图 5-13 MemTronic 的 HTML/JavaScript Cruncher-Compressor 可以大大缩小 JavaScript 源代码的大小，并且难于读懂

## 5.6 使用Firefox的Web开发扩展

Firefox 的 Web 开发扩展为 Firefox 浏览器增加了大量有用的 Web 开发工具。一旦安装了这个扩展，你就可以通过一个工具条来访问为浏览器增加的这些工具（见图 5-14）。在目前能够运行 Firefox 的所有平台上都能使用这个扩展包，这就意味着在 Windows、OS X 和 Linux 都能顺利地使用这个扩展包。Firefox 的 Web 开发扩展包可以从 [chrispederick.com/work/firefox/webdeveloper/](http://chrispederick.com/work/firefox/webdeveloper/) 获得。

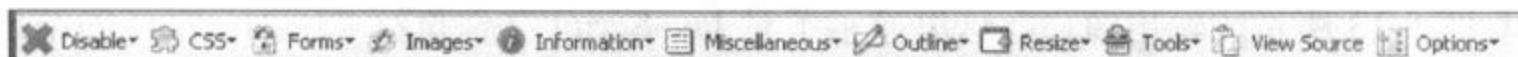


图 5-14 Web 开发扩展为 Firefox 增加的工具条

Web 开发扩展包提供了至少 80 个独立的工具，可以完成，从把 GET 请求转换为 POST 请求（或反之），到允许动态编辑页面的 CSS 规则的各种任务。由于存在太多的工具，因此不便于一一列出，以下只是列出了总的工具种类：

- Disable 菜单用于禁用浏览器的一些功能，如 JavaScript、CSS、cookies 和动画图片。
- CSS 菜单包含了与 CSS 规则和样式表相关的工具。
- Forms 菜单用于把 GET 请求转换为 POST 请求（或反之），自动填写表单值，以及删除输入元素的最大长度。
- Images 菜单用于显示图片缩略图或隐藏图片。
- Information 菜单用于检查与页面相关的各种信息，如 cookie 信息、链接信息和响应头部。
- Miscellaneous 菜单提供的工具可以清空浏览器的缓存、历史和会话 cookie，还可以放大或缩小页面。
- Outline 菜单用于概要列出表格、表单元、框架、块级元素等。
- Resize 菜单会在标题栏中显示当前窗口的大小，以及其他可以调整当前窗口大小的工具。
- Tools 菜单用于寻找到验证 CSS、HTML 和下载速度的第三方网站的快速链接。
- View Source 按钮可以很容易地查看页面的源代码。
- Options 菜单用于定制编辑 Web 开发扩展的颜色、快捷键和行为。

有些 Web 开发人员对 Web 开发扩展包提供的工具和功能大加赞赏，称之为“不可缺少的”、“最好的”和“至关重要的”。可以安装这个扩展，尝试一下各种工具，看看它对你的开发和调试过程是否有帮助。

## 5.7 实现高级JavaScript技术

我们假设本书的读者对 JavaScript 至少有基本的实践知识。如果要提供 JavaScript 的全面教程，这本身就需要一本完整的书才能讲清楚，所以在这里我们不打算详细介绍这种语言。相反，本节只是讨论 JavaScript 的一些可能鲜为人知的高级特性，并说明如何在你的 Ajax 开发中结合使用这些特性。

我们先来简单地谈谈 JavaScript 的历史，以便你了解它原来是什么样子，又是怎么发展到今天的。Netscape 的 Brendan Eich 于 1995 年开发了 JavaScript。他的任务本来是开发一种方法，使得创建和维护 Web 网站的非专业 Web 设计人员能够更容易地使用 Java applet。Eich 认为，适当的选择是开发一种不需要编译器的弱类型语言。

最初，Eich 开发的这个语言有过很多名字，不过，后来由于 Java 在市场上大获成功，因此借着这股东风，最后命名为 JavaScript。JavaScript 很快成为 Web 上最流行的脚本语言，

这要归功于它的低门槛，另外还因为它能够把 JavaScript 脚本从一个页面复制到另一个页面。在 JavaScript 和 Navigator DOM 的早期修订版本基础上，产生了 DOM Level 0 标准，该标准将表单元素和图像定义为 DOM 元素的子元素。

Microsoft 迎头赶上，创建了自己的脚本语言 VBScript。VBScript 在功能上与 JavaScript 类似，但采用了类 Visual Basic 语法，而且只能用于 IE。Microsoft 还提供了 JavaScript 的一个实现 JScript（现在已由 ECMA 标准化并称为 ECMAScript）。尽管不同 JavaScript 的语法几乎是一样的，但不同浏览器上 DOM 实现却大相径庭，以至于几乎不可能创建跨浏览器的脚本。使用“最小公分母”方法得到的脚本通常只能完成最简单的任务。

到了 1998 年，Netscape 开放了其浏览器的源代码，决定开始重写浏览器，并把重点放在遵循 W3C 标准上。那时，IE 5 是 W3C DOM 和 ECMAScript 的最佳实现。开源版本的 Netscape 以 Mozilla 为名于 2002 年问世。由此开始，浏览器领域形成了一个趋势：越来越多的浏览器开始努力遵循 W3C 和 ECMA 维护的 Web 标准。如今，现代浏览器（如 Firefox、Mozilla、Opera、Konqueror 和 Safari）都严格遵循 Web 标准，这就大大简化了编写跨浏览器的 HTML 和 JavaScript 等的任务。IE 6 与 1998 年的 IE 5 并没有太大差别，它严格禁止了最不合标准的行为。

### 5.7.1 通过 **prototype** 属性建立面向对象的 JavaScript

JavaScript 通过一种链接机制来支持继承，而不是通过完全面向对象语言（如 Java）所支持的基于类的继承模型。每个 JavaScript 对象都有一个内置的属性，名为 **prototype**。**prototype** 属性保存着对另一个 JavaScript 对象的引用，这个对象作为当前对象的父对象。

当通过点记法引用对象的一个函数或属性时，倘若对象上没有这个函数或属性，此时就会使用对象的 **prototype** 属性。当出现这种情况时，将检查对象 **prototype** 属性所引用的对象，查看是否有所请求的属性或函数。如果 **prototype** 属性引用的对象也没有所需的函数或属性，则会进一步检查这个对象（**prototype** 属性引用的对象）的 **prototype** 属性，依次沿着链向上查找，直到找到所请求的函数或属性，或者到达链尾，如果已经到达链尾还没有找到，则返回 **undefined**。从这个意义上讲，这种继承结构更应是一种“**has a**”关系，而不是“**is a**”关系。

如果你习惯于基于类的继承机制，那么可能要花一些时间来熟悉这种 **prototype** 机制。**prototype** 机制是动态的，可以根据需要在运行时配置，而无需重新编译。你可以只在需要时才向对象增加属性和函数，而且能动态地把单独的函数合并在一起，来创建动态、全能的对象。对 **prototype** 机制的这种高度动态性可谓褒贬不一，因为这种机制学习和应用起来很不容易，但是一旦正确地加以应用，这种机制则相当强大而且非常健壮。

这种动态性与基于类的继承机制中的多态概念异曲同工。两个对象可以有相同的属性和

函数，但是函数方法（实现）可以完全不同，而且属性可以支持完全不同的数据类型。这种多态性使得 JavaScript 对象能够由其他脚本和函数以统一的方式处理。

图 5-15 显示了实际的 prototype 继承机制。这个脚本定义了 3 类对象：Vehicle、SportsCar 和 CementTruck。Vehicle 是基类，另外两个类由此继承。Vehicle 定义了两个属性：wheelCount 和 curbWeightInPounds，分别表示 Vehicle 的车轮数和总重量。JavaScript 不支持抽象类的概念（抽象类不能实例化，只能由其他类扩展），因此，对于 Vehicle 基类，wheelCount 默认为 4，curbWeightInPounds 默认为 3000。

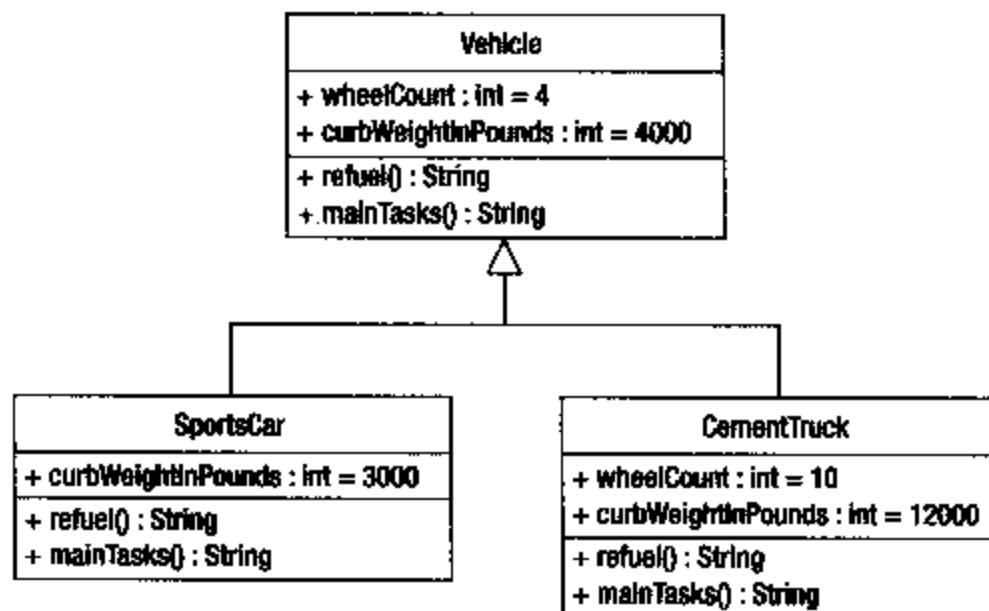


图 5-15 Vehicle、SportsCar 和 CementTruck 对象之间的关系

注意，这个 UML 图展示了 SportsCar 和 CementTruck 对象覆盖了 Vehicle 的 refuel 和 mainTasks 函数，因为一般的 Vehicle、SportsCar（赛车）和 CementTruck（水泥车）会以不同的方式完成这些任务。SportsCar 与 Vehicle 的车轮数相同，所以 SportsCar 的 wheelCount 属性不会覆盖 Vehicle 的 wheelCount 属性。CementTruck 的车轮数和重量都超过了 Vehicle，所以 CementTruck 的 wheelCount 和 curbWeightInPounds 属性要覆盖 Vehicle 的相应属性。

代码清单 5-2 包含了定义这 3 个类的 JavaScript 代码。要特别注意如何在对象定义中对属性和函数附加 prototype 关键字，还要注意每个对象由一个构造函数定义，构造函数与对象类型同名。

### 代码清单 5-2 inheritanceViaPrototype.js

```

/* Constructor function for the Vehicle object */
function Vehicle() {}

/* Standard properties of a Vehicle */
Vehicle.prototype.wheelCount = 4;
Vehicle.prototype.curbWeightInPounds = 4000;
  
```

```
/* Function for refueling a Vehicle */
Vehicle.prototype.refuel = function() {
    return "Refueling Vehicle with regular 87 octane gasoline";
}

/* Function for performing the main tasks of a Vehicle */
Vehicle.prototype.mainTasks = function() {
    return "Driving to work, school, and the grocery store";
}

/* Constructor function for the SportsCar object */
function SportsCar() { }

/* SportsCar extends Vehicle */
SportsCar.prototype = new Vehicle();

/* SportsCar is lighter than Vehicle */
SportsCar.prototype.curbWeightInPounds = 3000;

/* SportsCar requires premium fuel */
SportsCar.prototype.refuel = function() {
    return "Refueling SportsCar with premium 94 octane gasoline";
}

/* Function for performing the main tasks of a SportsCar */
SportsCar.prototype.mainTasks = function() {
    return "Spirited driving, looking good, driving to the beach";
}

/* Constructor function for the CementTruck object */
function CementTruck() { }

/* CementTruck extends Vehicle */
CementTruck.prototype = new Vehicle();

/* CementTruck has 10 wheels and weighs 12,000 pounds*/
CementTruck.prototype.wheelCount = 10;
CementTruck.prototype.curbWeightInPounds = 12000;

/* CementTruck refuels with diesel fuel */
CementTruck.prototype.refuel = function() {
    return "Refueling CementTruck with diesel fuel";
}

/* Function for performing the main tasks of a SportsCar */
CementTruck.prototype.mainTasks = function() {
    return "Arrive at construction site, extend boom, deliver cement";
}
```

代码清单 5-3 是一个很小的 Web 页面，展示了这 3 个对象的继承机制。这个页面只包含 3 个按钮，每个按钮创建一个类型的对象（Vehicle、SportsCar 或 CementTruck），并把对象传递到 describe 函数。describe 函数负责显示各个对象的属性值，以及对象函数的返回值。注意，describe 方法并不知道它描述的对象是 Vehicle、SportsCar，还是 CementTruck，它只是认为这个对象有适当的属性和函数，并由这个对象返回自己的值。

### 代码清单 5-3 inheritanceViaPrototype.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>JavaScript Inheritance via Prototype</title>

<script type="text/javascript" src="inheritanceViaPrototype.js"></script>

<script type="text/javascript">

function describe(vehicle) {
    var description = "";
    description = description + "Number of wheels: " + vehicle.wheelCount;
    description = description + "\n\nCurb Weight: " + vehicle.curbWeightInPounds;
    description = description + "\n\nRefueling Method: " + vehicle.refuel();
    description = description + "\n\nMain Tasks: " + vehicle.mainTasks();
    alert(description);
}

function createVehicle() {
    var vehicle = new Vehicle();
    describe(vehicle);
}

function createSportsCar() {
    var sportsCar = new SportsCar();
    describe(sportsCar);
}

function createCementTruck() {
    var cementTruck = new CementTruck();
    describe(cementTruck);
}
</script>
</head>

<body>
<h1>Examples of JavaScript Inheritance via the Prototype Method</h1>
```

```

<br/><br/>
<button onclick="createVehicle();">Create an instance of Vehicle</button>

<br/><br/>
<button onclick="createSportsCar();">Create an instance of SportsCar</button>

<br/><br/>
<button onclick="createCementTruck();">Create an instance of CementTruck</button>

</body>
</html>

```

分别创建 3 个对象，并用 `describe` 函数描述，结果如图 5-16 所示。

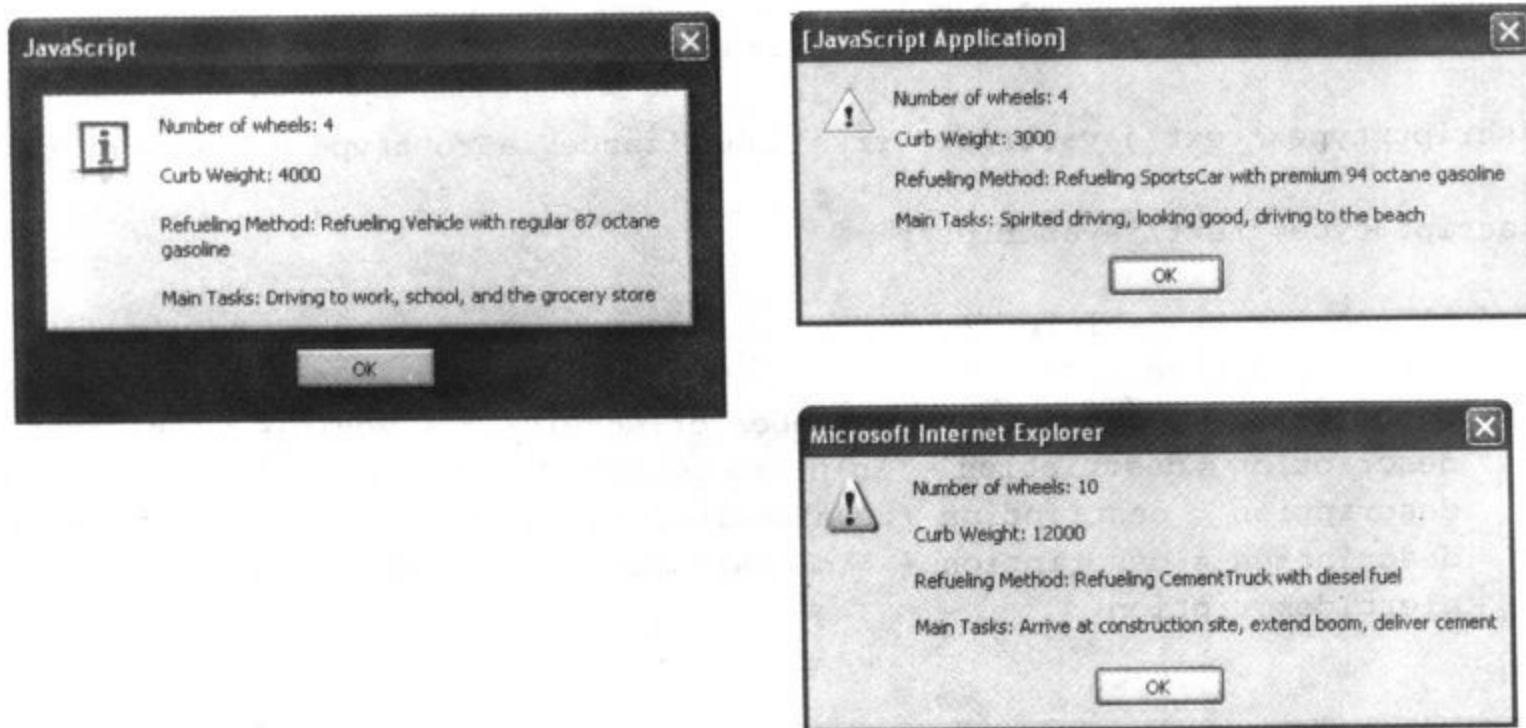


图 5-16 创建 `Vehicle`、`SportsCar` 和 `CementTruck` 对象并使用 `describe` 函数分别描述的结果

### 5.7.2 私有属性和使用 JavaScript 的信息隐藏

铁杆的面向对象设计支持者会注意到，当使用 `prototype` 方法向 JavaScript 对象增加属性和函数时，所增加的属性和函数都是公用的，所有其他对象都能访问。对于函数来说，这通常没有问题，因为大多数函数都确实应当提供给外部客户。但是对于属性，面向对象设计的支持者就会指出，公有属性违反了信息隐藏的概念，对象的属性应当是私有的，因此外部客户不能直接访问。外部客户只能通过公用可用的函数来访问对象的私有属性。

对于 JavaScript，同样有可能创建外部客户不能访问的私有属性，而只能通过对象的（公用）方法来访问，但这一点很少有人知道。Douglas Crockford<sup>1</sup>提出了一种在 JavaScript 中创

1. [http://www.crockford.com/。](http://www.crockford.com/)

建私有属性的方法。这种方法非常简单，总结如下：

- 私有属性可以在构造函数中使用 var 关键字定义。
- 私有属性只能由特权函数（privileged function）公用访问。特权函数就是在构造函数中使用 this 关键字定义的函数。外部客户可以访问特权函数，而且特权函数可以访问对象的私有属性。

下面来考虑前一个示例中的 Vehicle 类。假设你想让 wheelCount 和 curbWeightInPounds 属性是私有的，并只能通过公用方法访问。新的 Vehicle 对象如代码清单 5-4 所示。

#### 代码清单 5-4 重写后的 Vehicle 对象

```
function Vehicle() {  
    var wheelCount = 4;  
    var curbWeightInPounds = 4000;  
  
    this.getWheelCount = function() {  
        return wheelCount;  
    }  
  
    this.setWheelCount = function(count) {  
        wheelCount = count;  
    }  
  
    this.getCurbWeightInPounds = function() {  
        return curbWeightInPounds;  
    }  
  
    this.setCurbWeightInPounds = function(weight) {  
        curbWeightInPounds = weight;  
    }  
  
    this.refuel = function() {  
        return "Refueling Vehicle with regular 87 octane gasoline";  
    }  
  
    this.mainTasks = function() {  
        return "Driving to work, school, and the grocery store";  
    }  
}
```

注意，wheelCount 和 curbWeightInPounds 属性都在构造函数中使用 var 关键字定义，这就使得这两个属性是私有属性。属性不再是公用的，如果想通过点记法访问 wheelCount 属性的值，如下：

```
var numberOfWorks = vehicle.wheelCount;
```

就会返回 undefined，而不是 wheelCount 实际的值。

由于属性现在是私有的，因此需要提供能访问这些属性的公用函数。getWheelCount、setWheelCount、getCurbWeightInPounds 和 setCurbWeightInPounds 函数就是作此使用的。现在 Vehicle 对象可以保证只能通过公用函数访问私有属性，因此满足了信息隐藏的概念。

### 5.7.3 JavaScript 中基于类的继承

JavaScript 中基于 prototype 的继承机制可以很好地工作，但是对于一些已经习惯于 C++ 和 Java 等语言中基于类的继承机制的人来说，JavaScript 的 prototype 继承机制不是一种自然的编程方法。如果你不想用基于 prototype 的继承，而想用一种基于类的继承方法，那就继续读下去吧。

Netscape 的 Bob Clary<sup>1</sup>也提出了一个方法，它可以使一个对象使用一个通用的脚本从另一个对象继承属性和函数。这个脚本只是将“父”对象的属性和函数简单地复制到“子”对象。为此，我们将说明如何对脚本稍加修改，从而只是将子对象中不存在的属性和函数复制到子对象；这样一来，子对象中的函数就能覆盖父对象的函数。在两个对象之间创建继承关系的通用函数如下：

```
function createInheritance(parent, child) {
    var property;
    for(property in parent) {
        if(!child[property]) {
            child[property] = parent[property];
        }
    }
}
```

createInheritance 函数有两个参数，父对象和子对象。这个函数只是迭代处理父对象的所有成员（成员就是属性或函数），如果某个成员在子对象中不存在，则复制到子对象。

使用 createInheritance 函数相当简单：首先创建子对象的一个实例，然后使用 createInheritance 函数，为它传递子对象以及父对象的一个实例，如下：

```
var child = new Child();
createInheritance(new Parent(), child);
```

父对象中有而子对象中没有的所有属性和方法将复制到子对象。

---

<sup>1</sup>. [http://devedge-temp.mozilla.org/toolbox/examples/2003/inheritFrom/index\\_en.html](http://devedge-temp.mozilla.org/toolbox/examples/2003/inheritFrom/index_en.html).

#### 5.7.4 汇合

前面已经了解到, JavaScript 中也可以实现私有属性, 而且 JavaScript 也能像 C++ 和 Java 一样支持基于类的继承方法。为了展示这些是怎样实现的, 下面说明如何转换前面使用 Vehicle、SportsCar 和 CementTruck 对象的示例, 从而使用信息隐藏和继承的新模式。代码清单 5-5 列出了新的对象定义。

代码清单 5-5 classicalInheritance.js

```
function Vehicle() {
    var wheelCount = 4;
    var curbWeightInPounds = 4000;

    this.getWheelCount = function() {
        return wheelCount;
    }

    this.setWheelCount = function(count) {
        wheelCount = count;
    }

    this.getCurbWeightInPounds = function() {
        return curbWeightInPounds;
    }

    this.setCurbWeightInPounds = function(weight) {
        curbWeightInPounds = weight;
    }

    this.refuel = function() {
        return "Refueling Vehicle with regular 87 octane gasoline";
    }

    this.mainTasks = function() {
        return "Driving to work, school, and the grocery store";
    }
}

function SportsCar() {
    this.refuel = function() {
        return "Refueling SportsCar with premium 94 octane gasoline";
    }

    this.mainTasks = function() {
        return "Spirited driving, looking good, driving to the beach";
    }
}
```

```

        }
    }

    function CementTruck() {
        this.refuel = function() {
            return "Refueling CementTruck with diesel fuel";
        }

        this.mainTasks = function() {
            return "Arrive at construction site, extend boom, deliver cement";
        }
    }
}

```

需要注意，SportsCar 和 CementTruck 对象没有定义自己的 wheelCount 和 curbWeightInPounds 属性，也没有相关的存取函数，因为这些属性和函数会从 Vehicle 对象继承。

与前面一样，需要一个简单的 HTML 页面来测试这些新对象。代码清单 5-6 列出了测试这些新对象的 HTML 页面。要特别注意 createInheritance 函数，看看如何使用这个函数在 Vehicle 和 SportsCar 对象之间以及 Vehicle 和 CementTruck 对象之间创建继承关系。还要注意 describe 函数有所修改，以试图直接访问 wheelCount 和 curbWeightInPounds 属性。这样做会返回一个 undefined 值。

#### 代码清单 5-6 classicalInheritance.html

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Classical Inheritance in JavaScript</title>

<script type="text/javascript" src="classicalInheritance.js"></script>

<script type="text/javascript">
function createInheritance(parent, child) {
    var property;
    for(property in parent) {
        if(!child[property]) {
            child[property] = parent[property];
        }
    }
}

function describe(vehicle) {
    var description = "";

```

```
description = description + "Number of wheels (via property): "
                + vehicle.wheelCount;
description = description + "\n\nNumber of wheels (via accessor): "
                + vehicle.getWheelCount();
description = description + "\n\nCurb Weight (via property): "
                + vehicle.curbWeightInPounds;
description = description + "\n\nCurb Weight (via accessor): "
                + vehicle.getCurbWeightInPounds();
description = description + "\n\nRefueling Method: " + vehicle.refuel();
description = description + "\n\nMain Tasks: " + vehicle.mainTasks();
alert(description);
}

function createVehicle() {
    var vehicle = new Vehicle();
    describe(vehicle);
}

function createSportsCar() {
    var sportsCar = new SportsCar();
    createInheritance(new Vehicle(), sportsCar);
    sportsCar.setCurbWeightInPounds(3000);
    describe(sportsCar);
}

function createCementTruck() {
    var cementTruck = new CementTruck();
    createInheritance(new Vehicle(), cementTruck);
    cementTruck.setWheelCount(10);
    cementTruck.setCurbWeightInPounds(10000);
    describe(cementTruck);
}
</script>
</head>

<body>
    <h1>Examples of Classical Inheritance in JavaScript</h1>

    <br/><br/>
    <button onclick="createVehicle();">Create an instance of Vehicle</button>

    <br/><br/>
    <button onclick="createSportsCar();">Create an instance of SportsCar</button>

    <br/><br/>
    <button onclick="createCementTruck();">Create an instance of CementTruck</button>
```

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

分别点击页面上的各个按钮会得到图 5-17 所示的结果。正如所料，试图直接访问私有属性就会返回 undefined。

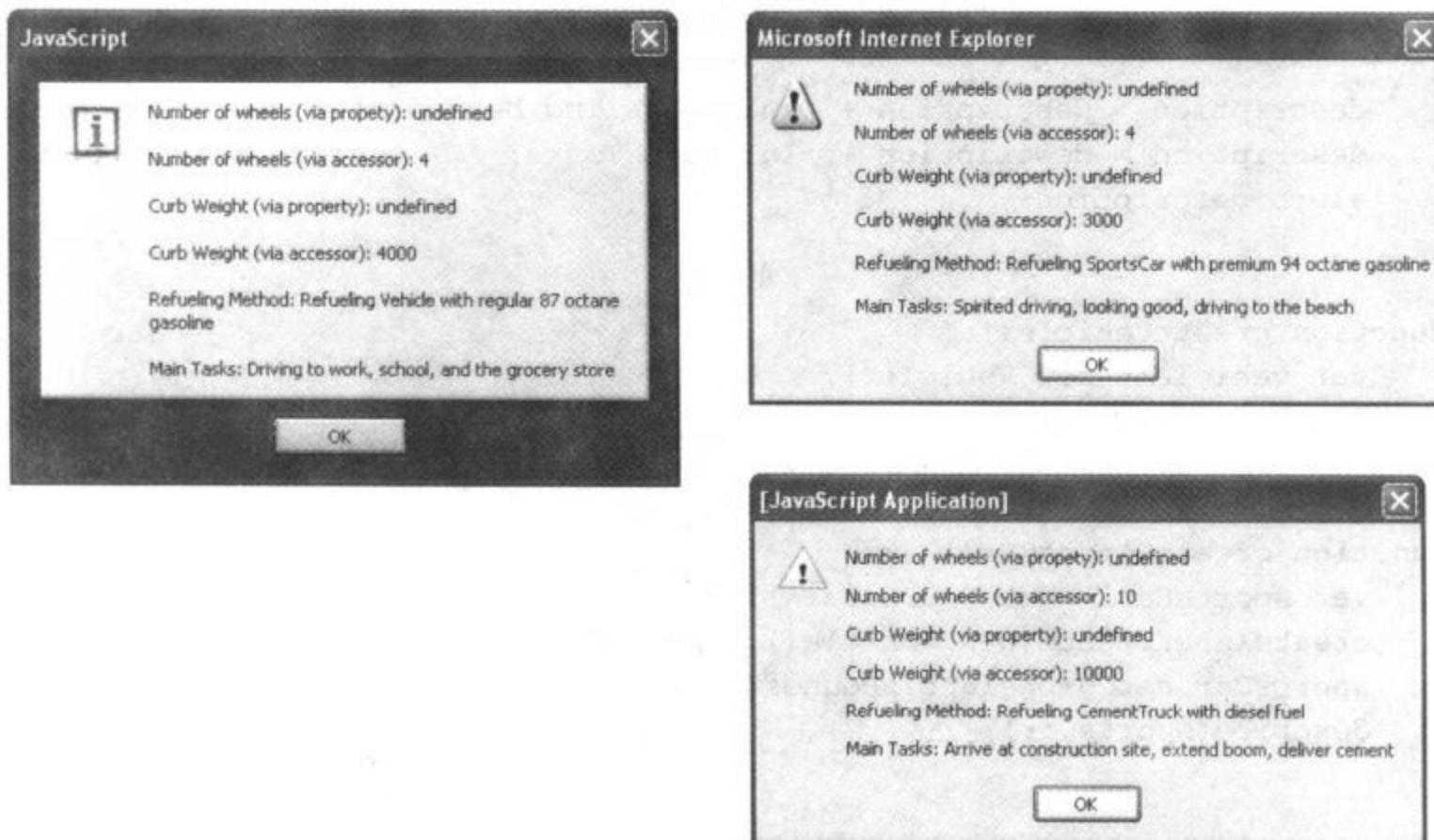


图 5-17 创建 Vehicle、SportsCar 和 CementTruck 对象，并使用 describe 函数描述它们的结果。私有属性不能直接访问，见警告框中的 undefined 值

## 5.8 小结

本章我们介绍了一些工具和技术，采用这些工具和技术，会让你的开发过程更加愉快。JSDoc 可以帮助建立 JavaScript 代码的文档，从而使其他开发人员能够更容易地理解和使用代码。如果你开始经常使用 Ajax 技术，肯定会编写一些你自己可重用的 JavaScript 库，而且会用 JSDoc 为代码建立文档，以便其他人更轻松地使用这些库。

HTML Validator 和 Checky 等工具可以帮助你确保所写的 HTML 代码是合法的 HTML。不合法的 HTML 会导致未预见的行为，所以使用合法的 HTML 或 XHTML 能消除可能导致错误的一些因素。另外，如果 XHTML 或 HTML 是合法的，更有可能在多个浏览器平台上表现相同。

在 Firefox 和其他 Mozilla 浏览器中打包提供了 DOM Inspector 工具，利用这个工具可以将 HTML 文档作为结构化树来检查其节点。DOM Inspector 允许你查看每个节点及其属性值，甚至可以动态修改属性值。你可以动态地将节点从页面中的一个位置移到另一个位置，而不

必重写 HTML。如果要检查通过 JavaScript 动态创建的节点，DOM Inspector 就很有用。

JSLint 是一个 JavaScript 验证工具。尽管它不能确定 JavaScript 的逻辑是否正确，但确实能帮助找出语言语法中存在的错误，还能发现由于编码风格不好而可能出错的部分。

删除 JavaScript 中的注释行和回车换行符可以大大缩小 JavaScript 文件的大小，相应地，将 JavaScript 文件下载到客户浏览器的时间也会减少。MemTronic 的 HTML/JavaScript Cruncher-Compressor 不仅能删除注释行和回车换行符，还能真正压缩 JavaScript 代码，从而加快下载速度。压缩还有一个很好的副作用，就是能使 JavaScript 更难读，从而有助于保护 JavaScript 代码的内部工作不外泄，不会被别人窃取。

Firefox 的 Web 开发扩展包为 Web 开发人员提供了许多有用的工具。利用这些工具，可以调整图片的大小，动态地编辑 CSS 样式规则，将表单方法从 GET 改为 POST（或反之），除此以外还有很多。

我们也介绍了一些高级的 JavaScript 技术，如面向对象编程。首先，你了解了 JavaScript 如何使用一种基于 prototype 的机制来模拟继承。然后了解了 JavaScript 如何使用只能通过公共方法访问的私有属性支持信息隐藏的概念。最后，我们介绍了使 JavaScript 可以模拟基于类的继承机制的技术，这种继承机制类似于 C++ 和 Java 中使用的继承机制。对于那些习惯于完全面向对象语言的人来说，这种技术是一种更自然的编码风格。

作为 Ajax 开发人员，采用这些工具和技术会使开发更为轻松、工作心情更加愉快。你可以把这些工具都拿来试试挑出你喜欢的工具，另外，你还可能在 Web 上碰到其他有用的工具。



## 使用 JsUnit 测试 JavaScript 代码

**读**到这里，应该很清楚了，要想真正很好地使用 Ajax，你就要写一些 JavaScript 代码。尽管框架和工具包能减轻一些负担，但是最终你可能会得到比平常更多的 JavaScript 代码。因为我们自己写过不少，所以很清楚写 JavaScript 代码绝不是轻而易举的，不过，本章我们还是要在你筋疲力尽的肩头再压上几块石头。

具体地，我们将介绍测试驱动开发(test-driven development, TDD)，并展示开发 JavaScript 代码时如何应用 TDD。尽管这种方法不能马上解决你的所有编程问题，但至少能帮助你尽快完成工作，能按时回家与家人共进晚餐。我们先对 TDD 和广泛使用的 JUnit 做一个简要介绍。打好基础之后，我们将讨论 JsUnit，并说明如何编写和运行测试。

### 6.1 JavaScript 提出的问题

如果你参与过 Web 应用的开发，可能已经写过一些 JavaScript 代码；当然，如果你只是写了一些最简单的函数，那么对 JavaScript 的看法可能不会太好。浏览器不兼容，缺少优秀的开发工具，没有代码完成（code completion）之类的生产力工具，没有调试工具——这么多的缺点，足以让大多数开发人员更乐于使用 vi<sup>1</sup>。

我们很清楚你的这些苦衷。在第 5 章中，已经讨论了许多工具，它们能让你的日子更轻松。本章将介绍如何让开发 JavaScript 尽可能地容易（至少，在工具开发商迎头赶上之前，这种方法很合适<sup>2</sup>）。采用测试先行（test-first）的方式来编写 JavaScript，能大大简化整个开发过程。

#### 6.1.1 测试先行方法介绍

是的，现在肯定有读者会这样说了：“我只在产品发品之前写测试。”有些人可能会窃笑，

- 
1. 是的，对于有些人，vi 可能更方便，但是别因此而迟疑，哪怕是遇到了 JavaScript 的问题！
  2. 我们真地希望在不久的将来，那些主要的工具开发商能解决这个问题。

对质量保证部门说三道四。还有一些人作为项目经理可能会添油加醋地说：“我们可不会浪费时间写测试代码；我们还得写真正的代码呢。”那么，采用 TDD 到底是什么意思呢？

TDD 产生于敏捷开发运动，特别是极限编程（extreme programming, XP），而且 TDD 正是 XP 的一个核心原则。推崇 TDD 的人认为，不应该完成开发之后再写测试，这通常只是“马后炮”，而应当在写代码之前先写测试。测试本质上相当于设计文档，而不是花大量时间去摆弄一个复杂的图形化工具，你要直接在代码中“拟画”一个类。开始时先为一些小功能块编写测试。在有些语言下，你写的测试甚至不能编译，因为所引用的类尚不存在。一旦建立了测试，就可以运行这个测试（当然，此时运行测试会失败）。然后再编写最少量的代码，以便测试通过。接下来再重构代码，并增加更多的测试。

通常可以使用测试框架来帮助编写自动化测试。最著名的框架是 JUnit，不过现在已经有很多 xUnit 项目，可以简化在各种语言下测试的创建。一般地，这些框架都建立在断言（assert）基础上。开发人员编写测试方法，将调用方法的实际结果与期望结果进行比较。当然，可以人工地检查一个日志文件或用户界面来完成这些比较，但是，用计算机完成数据比较不仅速度快，也更准确。另外，就算是让计算机把同样的测试运行上 1500 次，它们也不会嫌烦，如果是人可做不到这一点。

有了 JUnit 和其他测试框架，编写和运行测试变得相当简单。这就能鼓励开发人员创建大量测试（往往能更完备地覆盖各种测试情况），而且会乐于经常运行这些测试（可以更快地帮助开发人员找到 bug）。在许多情况下，如果项目中采用了 TDD，测试代码往往与生产代码一样多！

使用 TDD 可以带来许多重要的好处：

**提供明确的目标：**你很清楚，一旦结束（也就是一旦测试通过），你的工作就完成了（假设你的测试写得很好）。测试会为代码建立一个自然的边界，使你把重点集中在当前任务上。一旦测试通过，就有确切的证据证明你的代码能工作。相对于人工地测试用户界面或者比较日志文件中的结果，在一个 xUnit 框架中运行自动化测试，速度要快几个数量级。大多数 xUnit 测试的运行只需几微秒，而且大多数采用 TDD 的人都会一天运行数次测试。在许多开发小组中，将代码签入源代码树之前，代码必须成功地通过测试。

**提供文档。**你是不是经常遇到看不懂的代码？这些代码可能没有任何文档说明，而且开发代码的人可能早就走了（或者度假去了）。当然，看到这种代码的时间往往也很不合时宜，可能是凌晨 3 点，也可能有位副总在你旁边大声催促着赶快解决昨天的问题，这样要想花些时间来明白原作者的意图就更困难了。我们见过一些好的单元测试文档，它们会指出系统要做什么。测试就像原开发人员留下的记号，可以展示他们的类具体是怎么工作的。

**改善设计：**编写测试能改善设计。测试有助于你从界面的角度思考，测试框架也是

代码的客户。测试能让你考虑得更简单。如果你确实遵循了“尽量简单而且行之有效”的原则，就不会写出篇幅达几页的复杂算法。要测试的代码通常依赖性更低，而且相互之间没有紧密的联系，因为这样测试起来更容易。当然，还有一个额外的作用，修改起来也会更容易！

**鼓励重构：**利用一套健壮的测试集，你能根据需要进行重构。你是不是经常遇到一些不知是否该修改的代码？种种顾虑让你行动迟缓，过于保守，因为你不能保证所做的修改会不会破坏系统。如果有一套好的单元测试集，就能放心地进行重构，同时能保证你的代码依然简洁。

**提高速度：**编写这么多测试会不会使开发速度减慢呢？人们经常会以速度（或开发开销）作为反对进行TDD和使用xUnit框架的理由。所有新的工具都会有学习曲线，但是一旦开发人员适应了他们选择的框架（通常只需很短的时间），开发速度实际上会加快。一个完备的单元测试集提供了一种方法对系统完成回归测试，这说明，增加一个新特性之后，你不必因为怀疑它会不会破坏原系统而寝食难安。

**提供反馈：**单元测试还有一个经常被忽略的优点，即开发的节奏。尽管看上去好像无关紧要，但通过测试之后你会有一种完成任务的成就感！你不会成天地修改代码而没有任何反馈，这种测试—代码—测试的方法会鼓励你动作幅度小一些，通常修改一次代码的时间仅几分钟而已。这样你不会一下子看到冒出一大堆的新特性，而只是让代码基一次前进一小步。

从我们的经验看，测试是会传染的，你可能会慢慢上瘾。一开始，许多开发人员都心存疑虑，但最终几乎每个开发人员都迷上了运行测试后的绿条<sup>1</sup>。测试第一次“抓住”bug或者增加一个新特性，只需几分钟而不是几个小时，往往就是在这样一些时候，开发人员会欣喜地认识到测试确实很有意义。

### 6.1.2 JUnit介绍

由于JsUnit的出现源自JUnit的启发，所以我们先对JUnit做一个简单介绍，然后再深入地分析JsUnit。关于JUnit有一些非常好的书，若要想详细了解JUnit，可以参考一下。虽然JUnit不是测试的唯一选择（TestNG和Fit/FitNesse也很值得研究），但是它与JsUnit有着密切的联系，实际上后者相当于为了测试JavaScript，而开发的JUnit“移植版”，这是我们首先讨论JUnit的原因。

JUnit是使用最广泛的xUnit测试框架之一。JUnit是Erich Gamma和Kent Beck编写的，通常用于测试基于Java的开源软件，而且最常用的IDE都对JUnit提供了充分的支持。用JUnit编写测试相当简单，只需创建一个实现TestCase的类，编写一些以test开头的方法，

1. JUnit有一个方便的“红条指示失败，绿条指示成功”的方法。——译者注

其中设置一些断言，然后用你最喜欢的工具来运行这些测试。默认情况下，JUnit 会自动运行以 `test` 开头的方法，不过，你也可以根据需要改变这种行为。

编写第二个或第三个测试时，你会发现有些公共的代码可以重构。你可能已经读过 Andrew Hunt 和 David Thomas 所著的 *The Pragmatic Programmer* (Addison-Wesley 公司 1999 年出版)，应该知道要避免重复，所以会把一些公共的代码抽出到一个固定件 (fixture) 中，为此要覆盖 `setUp()` 和 `tearDown()` 方法，这些方法会分别在运行每个测试之前和之后调用。

刚开始，你可能只有几个测试，但是慢慢地，测试会越来越多，而且需要某种方法来组织这些测试。在 JUnit 中，可以创建 `TestSuite`，其中包括一个测试方法集合，甚至是整个测试类。`(TestSuite 可以包含实现了 Test 接口的任何类。)` 如果你想对测试有更多的控制，可以手工地把测试增加到 `TestSuite`，或者可以告诉 JUnit 来为你完成这个工作，为此要把 `TestCase` 作为参数传递给 `TestSuite` 构造函数。

JUnit 支持许多测试运行工具。有些 IDE 有自己的专用运行工具，而且只要你愿意，还可以开发你自己的运行工具。JUnit 提供了一个文本运行工具，还提供了一个图形化运行工具，它能报告运行测试所得到的结果。(图形化工具有一个方便的“红条失败”/“绿条通过”方法。) JUnit 测试通常由提交或构建过程启动。

## 6.2 分析JsUnit

2001年初，Edward Hieatt 开始“移植”JUnit，目的是在浏览器中测试 JavaScript。从那以后，JsUnit 的下载次数已近 10000 次，大约 300 人加入了 JsUnit 的新闻组。JsUnit 支持一般的 xUnit 功能，完全用 JavaScript 编写，如果你习惯使用 JUnit 或者类似的 xUnit 框架，就会发现 JsUnit 使用起来相当简单直观。

JsUnit 也有一些不同的地方：这里也有 `setUp()` 和 `tearDown()`，不过现在作为函数，而不是方法；测试函数（而不是测试方法）分成多个测试页（而不是测试用例）；另外 JsUnit 提供了自己的基于 HTML 的测试运行工具。表 6-1 对这两个框架做了比较。

表 6-1 JUnit 与 JsUnit 的比较

JUnit	JsUnit
Test 类扩展 <code>TestCase</code>	测试页包含 <code>jsUnitCore.js</code>
测试方法	测试函数
<code>Test</code> 类	基于 HTML 的测试页
<code>TestSuites</code>	基于 HTML 的测试集
多个测试运行工具	基于 HTML/JavaScript 的测试运行工具
<code>setUp()</code> 和 <code>tearDown()</code> 方法	<code>setUp()</code> 和 <code>tearDown()</code> 函数
在虚拟机中运行	在浏览器中运行
用 Java 编写	用 JavaScript 编写

### 6.2.1 起步

对于 JsUnit，起步很简单，只需从 JsUnit 网站 ([www.edwardh.com/jsunit/](http://www.edwardh.com/jsunit/)) 下载 JsUnit zip 文件。把这个压缩文件解开，会得到一个 `jsunit` 文件夹，可以把 Web 服务器放在这里，这样整个团队或者整个组织就能更容易地使用 JsUnit。JsUnit 的大部分“核心”都在 `jsunit/app` 目录中，在这里可以看到 `jsUnitCore.js`、`jsUnitTracer.js` 和 `jsUnitTestManager.js`，另外还有其他一些文件。如果你想运行具体的 JsUnit 测试，可以使用 `testRunner.html` 来运行 `jsunit/tests` 目录中找到的任何测试页。如果你在使用 IntelliJ，而且想具体使用 JsUnit，`jsunit/intelliJ` 目录中包含了需要的所有适当文件。

### 6.2.2 编写测试

用 JsUnit 编写测试与用 JUnit 编写测试很相似。测试函数不能有任何参数，必须有一个前缀 `test`，例如 `testDateValidation()`。测试函数包含在一个测试页（*test page*）中，这类似于 JUnit 中的一个 `Test` 类。测试页必须包含 `jsUnitCore.js` 文件，解开 JsUnit zip 文件后，就会在 `jsunit/app` 目录中找到这个文件。包含这个 JavaScript 文件实际上就是把一个外部 JavaScript 文件增加到页面中；只需使用脚本元素`<script language="JavaScript" src="jsUnitCore.js"></script>`来引用这个文件，要记住，如果你的当前目录不是 `jsunit/app` 目录，则还需要提供 `jsUnitCore.js` 文件的相关路径信息。当然，在测试页中可以包含任意多个其他函数或 JavaScript；实际上，把多个 JavaScript 函数放在分开的文件中，是一个很好的做法。测试函数也可以放在单独的 JavaScript 文件中；不过，如果这样做，就需要使用 `exposeTestFunctionNames()` 方法，这样 JsUnit 才能找到测试函数。实际上，如果需要针对不同的页面内容建立测试，可以把测试函数放在一个单独的文件中，这样能避免复制-粘贴问题带来的痛苦。

一般地，JsUnit 会自动发现测试函数，就像 JUnit 会发现所有测试方法一样。不过，有些操作系统/浏览器不能合作。如果你发现不能如你所愿地发现测试函数，使用 `exposeTestFunctionNames()` 方法就能解决这个问题。

#### 断言方法

现在你对测试函数和测试页有一定的了解了，下面需要写一些实际的测试！与用 JUnit 一样，你可以使用断言方法（*assert method*）。断言方法是任何单元测试的基本模块，它们只是一些简单的布尔表达式，可以指示一个给定语句为 `true` 还是 `false`。断言失败时，就会产生一个错误，这样将得到众所周知的红条。与 JUnit 不同，JsUnit 没有提供那么丰富的断言方法，但是已经足够你测试 JavaScript 代码了。注意，除了 `fail()` 方法的注释外，其他断言方法的注释都是可选的（这与 JUnit 类似，甚至也“不正确”地把可选参数放在最前面，而不是最后）。

```

assert([comment], booleanValue)
assertTrue([comment], booleanValue)
assertFalse([comment], booleanValue)
assertEquals([comment], value1, value2)
assertNotEquals([comment], value1, value2)
assertNull([comment], value)
assertNotNull([comment], value)
assertUndefined([comment], value)
assertNotUndefined([comment], value)
assertNaN([comment], value)
assertNotNaN([comment], value)
fail(comment)

```

要看这些方法怎么用于测试（象征性地从字面了解），只需看 JsUnit 下载包提供的测试页就行了。JsUnit 还提供了一个变量：JSUNIT\_UNDEFINED\_VALUE，它映射到 JavaScript 中的 `undefined` 变量。

我们说的已经够多的了！下面来看一个简单的测试！这个例子中有一个简单的函数，会让两个数相加，而且有两个测试：一个用于正整数的相加，另一个用于负整数相加。要测试这个函数，先创建一个简单的 Web 页面，如代码清单 6-1 所示，其中包含了 `jsUnitCore.js` 文件，另外包含了要测试的函数和测试函数<sup>1</sup>。当然，在生产代码中，可能不能把测试代码与所测试的函数混在一起，但是作为第一次尝试这样做未尝不可。

### 代码清单 6-1 简单的测试页

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<html>
  <head>
    <title>A Simple Test Page</title>
    <script language="JavaScript" src="../jsunit/app/jsUnitCore.js"></script>
    <script language="JavaScript">
      function addTwoNumbers(value1, value2) {
        return value1 + value2;
      }

      function testValidArgs() {
        assertEquals("2 + 2 is 4", 4, addTwoNumbers(2, 2));
      }

      function testWithNegativeNumbers() {
        assertEquals("negative numbers: -2 + -2 is -4", -4,
                    addTwoNumbers(-2, -2));
      }
    </script>
  </head>
  <body>
    <h1>Test Results</h1>
    <ul>
      <li>testValidArgs passed</li>
      <li>testWithNegativeNumbers passed</li>
    </ul>
  </body>
</html>

```

1. 原文此处为“*test methods*”，应当是“*test functions*”。——译者注

```
</script>
</head>
<body>
    This is a simple test page for addTwoNumbers(value1, value2).
</body>
</html>
```

运行这些测试会得到图 6-1 所示的结果。(后面将更详细地介绍测试运行工具。)

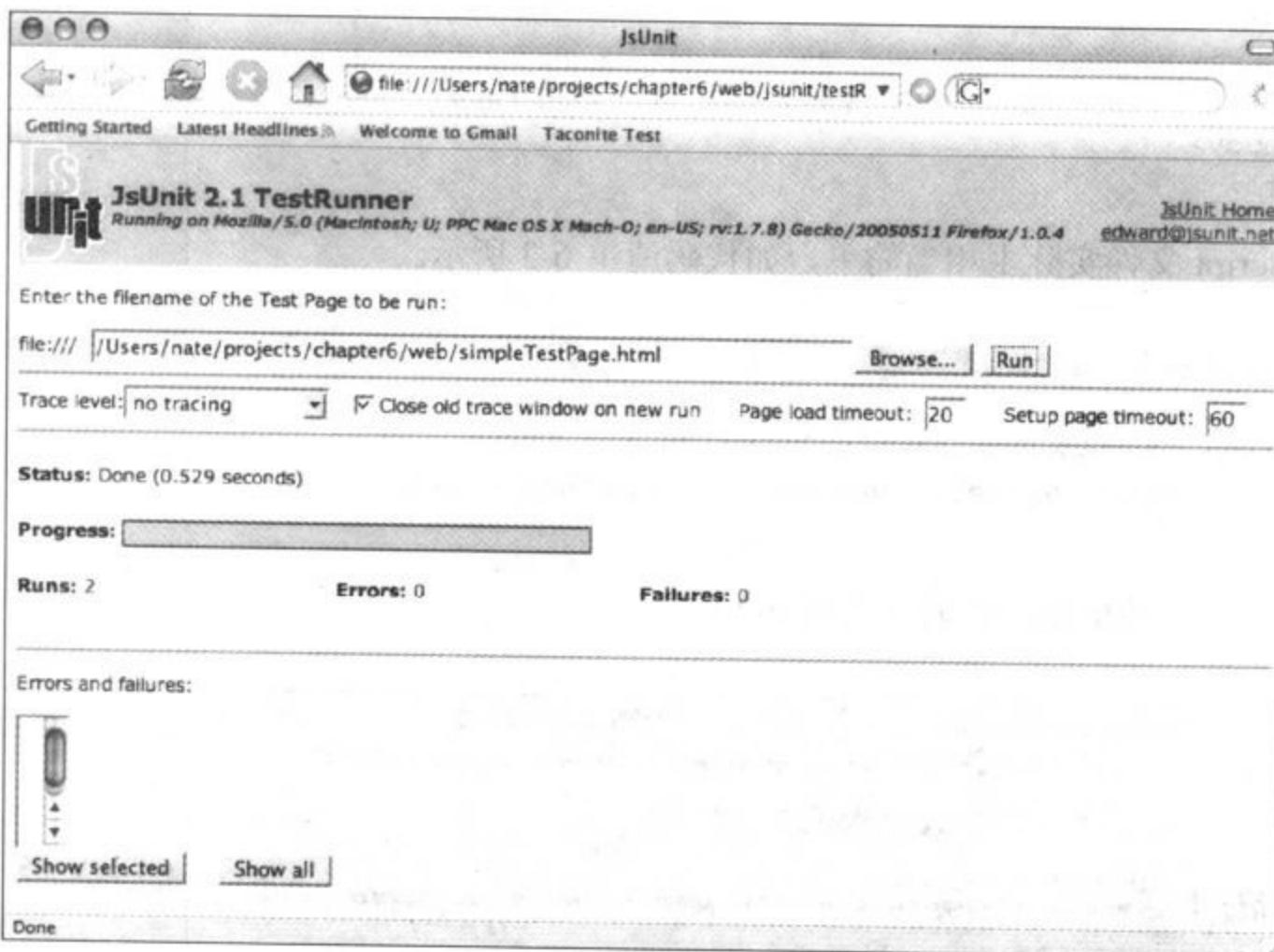


图 6-1 简单测试的结果

显然，不太可能把生产代码（函数）与测试函数混在同一个测试页中。你可能会把生产代码放在一个单独的 JavaScript 文件中，然后在测试页中包含这个文件。代码清单 6-2 就采用了这种方法。

### 代码清单 6-2 一种更典型的方法

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<html>
    <head>
        <title>Another Test Page</title>
        <script language="JavaScript" src="../jsunit/app/jsUnitCore.js"></script>
        <script language="JavaScript" src="simpleJS.js"></script>
        <script language="JavaScript">
```

```

        function testValidArgs() {
            assertEquals("2 + 2 is 4", 4, addTwoNumbers(2, 2));
        }
        function testWithNegativeNumbers() {
            assertEquals("negative numbers: -2 + -2 is -4", -4,
                        addTwoNumbers(-2, -2));
        }
    </script>
</head>
<body>
    This is a simple test page for the simpleJS file.
</body>
</html>

```

JavaScript 文件实际上相当简单，如代码清单 6-3 所示。

### 代码清单 6-3 simple.js

```

function addTwoNumbers(value1, value2) {
    return parseInt(value1) + parseInt(value2);
}

```

不出所料，结果是一样的（见图 6-2）。

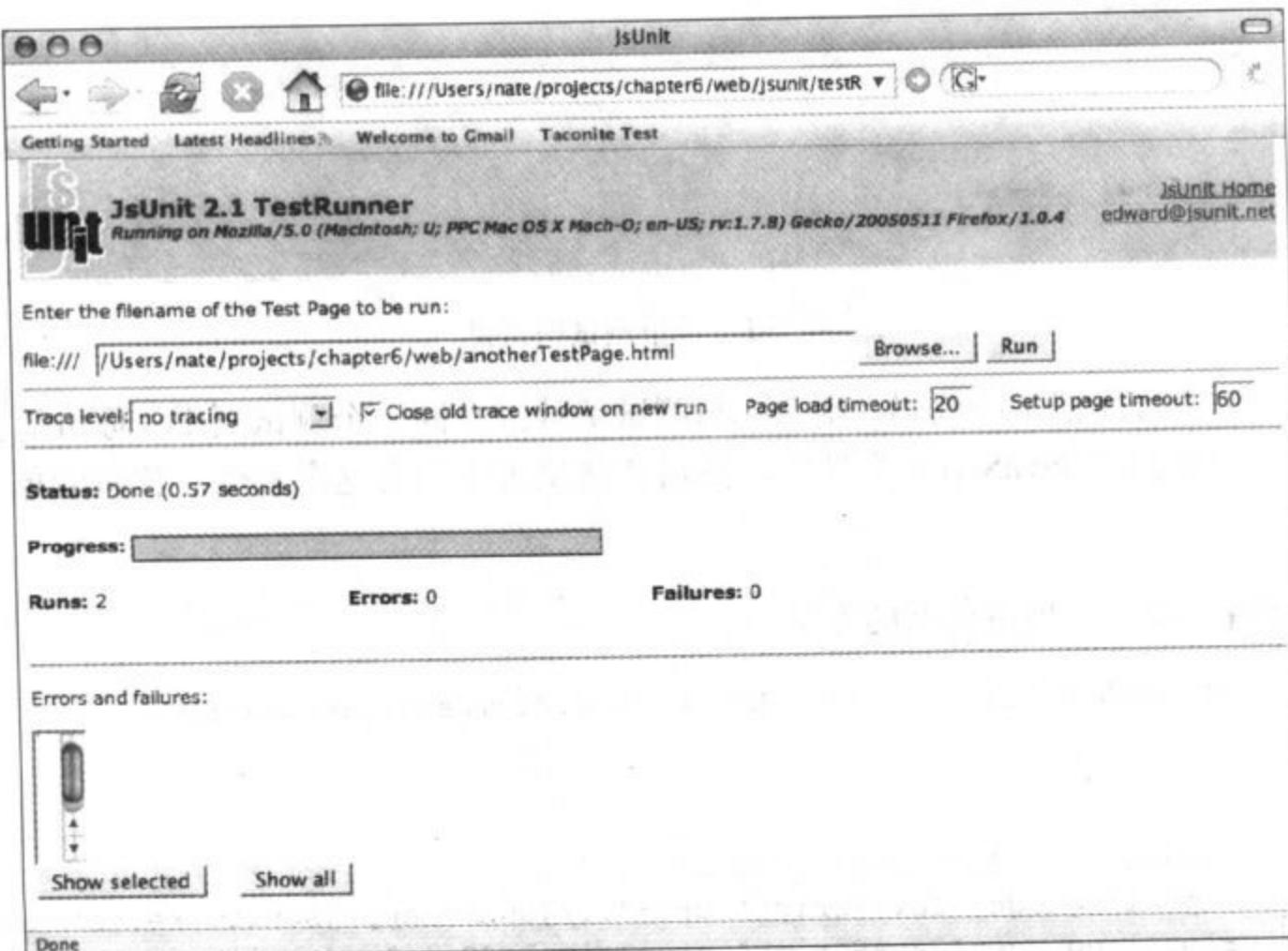


图 6-2 采用典型方法的结果

可以看到，两个测试函数会自动被发现，而且通常都是这样。不过，如果打开测试页，点击 Run 之后什么也没有发生，可能就需要使用 `exposeTestFunctionNames()`，以确保 JsUnit 能找到你的测试，如代码清单 6-4 所示。

#### 代码清单 6-4 使用 `exposeTestFunctionNames()`

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<html>
  <head>
    <title>A Test Page With exposeTestFunctions</title>
    <script language="JavaScript" src="../jsunit/app/jsUnitCore.js"></script>
    <script language="JavaScript" src="simpleJS.js"></script>
    <script language="JavaScript">

      function testValidArgs() {
        assertEquals("2 + 2 is 4", 4, addTwoNumbers(2, 2));
      }

      function testWithNegativeNumbers() {
        assertEquals("negative numbers: -2 + -2 is -4", -4,
                    addTwoNumbers(-2, -2));
      }

      function exposeTestFunctionNames() {
        var tests = new Array(2);
        tests[0] = "testValidArgs";
        tests[1] = "testWithNegativeNumbers";
        return tests;
      }
    </script>
  </head>
  <body>
    This is a simple test page that uses exposeTestFunctionNames.
  </body>
</html>
```

如你所愿，这样就能工作了（见图 6-3）。

如果看到如图 6-4 中所示的错误消息，说明你可能忘了在测试页中包含 `jsUnitCore.js`，或者文件的路径不对。请检查测试页，再运行一次。

点击 OK 时，会提示你重试或取消这个测试，如图 6-5 所示。

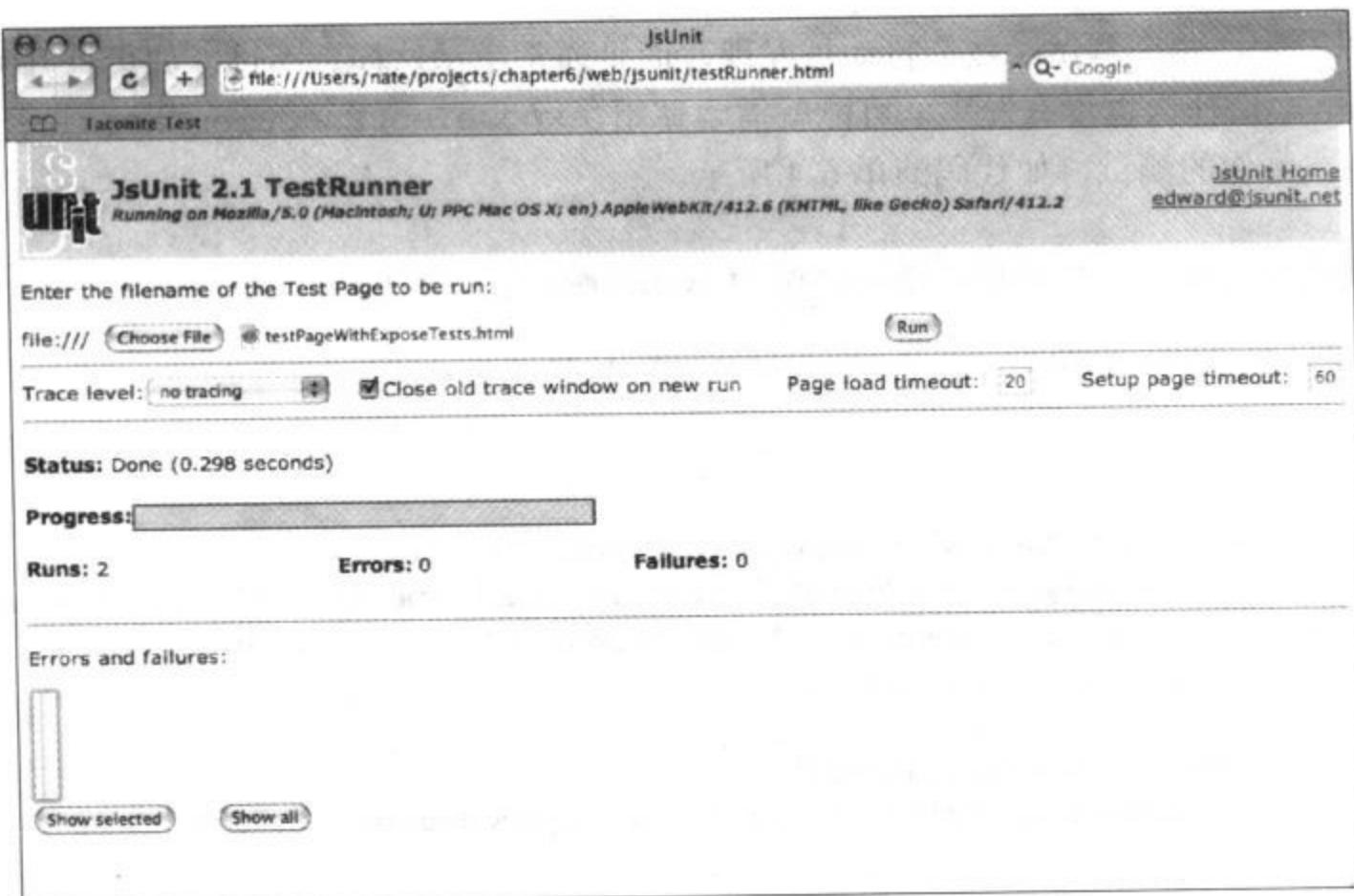


图 6-3 运行使用 exposeTestFunctionNames() 的测试

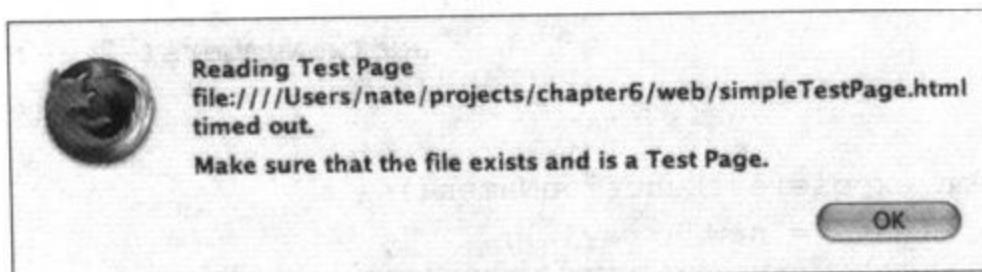


图 6-4 JsUnit 错误消息

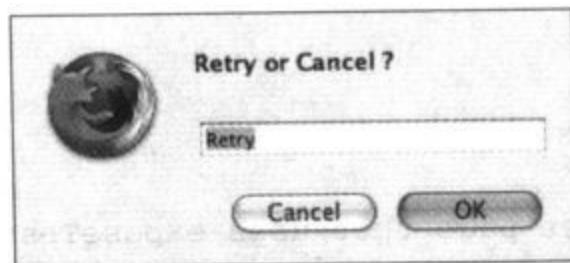


图 6-5 重试或取消

### **setUp() 和 tearDown()**

像 JUnit 一样, JsUnit 也支持 `setUp()` 和 `tearDown()`。JsUnit 与 JUnit 有一点是一样的, 即 `setUp()` 和 `tearDown()` 是可选的, 而且 `setUp()` 会在每个测试之前调用, `tearDown()` 会在每个测试之后调用。在大量使用 `setUp()` 和 `tearDown()` 之前, 需要了解 JUnit 与 JsUnit 中 `setUp()` 和 `tearDown()` 方法的实现有两个重要区别。在 JUnit 中, 每次测试运行会导致创建 `Test` 类的一个新实例, 这说明, 声明的所有实例变量在下一次测试运行时会“重置”。

不过，JsUnit 有所不同，它不会为每次测试运行重新加载测试页，所以变量状态会在多次测试之间保留。还有一个重要区别与测试顺序有关，使用 JUnit 的话，测试执行的顺序是不能保证的。在 JsUnit 中，测试会按测试页中声明的顺序执行，先从最上面的测试开始<sup>1</sup>。

代码清单 6-5 显示了一个相当复杂的例子，其中使用了 `setUp()` 和 `tearDown()` 方法。这里同样以前面创建的 `add` 方法为基础，但是这一次会增加一个表单。你要使用 `setUp()` 填写这个表单，然后使用 `tearDown()` 方法自行清空。

### 代码清单 6-5 使用 `setUp()` 和 `tearDown()`

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<html>
  <head>
    <title>Using setUp and tearDown</title>
    <script language="JavaScript" src="jsunit/app/jsUnitCore.js"></script>
    <script language="JavaScript" src="simpleJS.js"></script>
    <script language="JavaScript">

      function setUp() {
        document.getElementById("value1").value = "2";
        document.getElementById("value2").value = "2";
      }

      function testValidArgs() {
        assertEquals("2 + 2 should equal 4", 4, addNumbers());
      }

      function addNumbers() {
        var val1 = document.getElementById("value1").value;
        var val2 = document.getElementById("value2").value;
        return addTwoNumbers(val1, val2);
      }

      function tearDown() {
        document.getElementById("value1").value = "";
        document.getElementById("value2").value = "";
      }
    </script>
  </head>
  <body>
```

<sup>1</sup> 不过，这里同样适用有关测试依赖性的警告。尽管测试会按预定的顺序执行，但这并不意味着编写测试时应当依赖于执行顺序！

```

<form id="test">
    <input type="text" size="3" id="value1"/>
    <input type="text" size="3" id="value2"/>
    <input type="button" value="Add" onclick="addNumbers()"/>
</form>
</body>
</html>

```

不出意外，结果是很典型的（见图 6-6）。

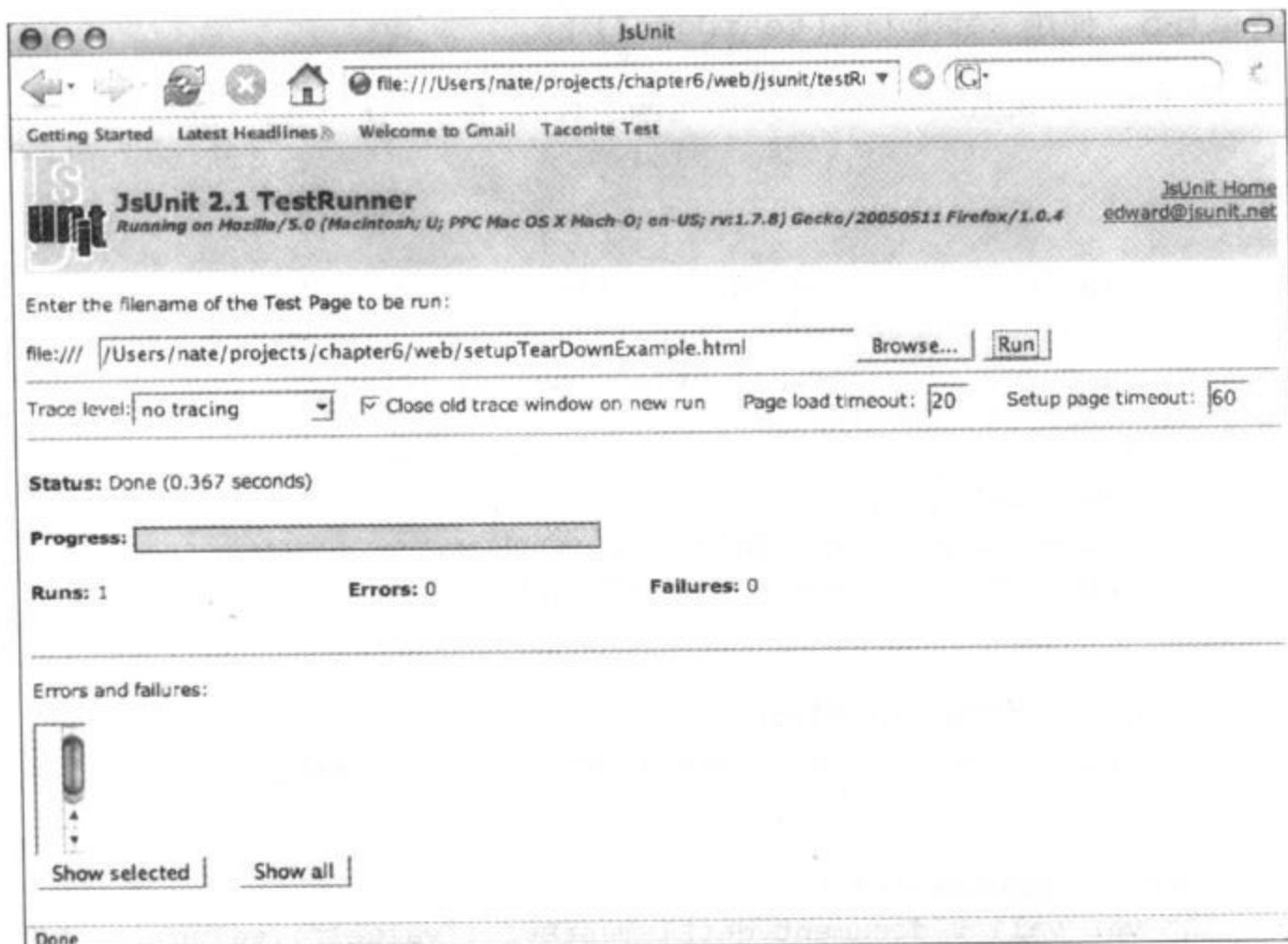


图 6-6 运行 `setUp()`/`tearDown()` 的例子

JsUnit 还包含另外一个特性：`setUpPage()` 函数，这是 JUnit 中所没有的。JUnit 社区的有些人认为 JUnit 缺乏“一次性启动”和“一次性关闭”功能是设计中一个败笔。有些人则更进一步，尝试扩展 JUnit 来包含这个特性，或者干脆创建新的测试框架；另外，JUnit 的 FAQ 网站甚至还介绍了一种方法来模拟这种行为<sup>1</sup>。大多数有关 JUnit 的书也讨论了解决这个问题的方法。

不过，JsUnit 则不同，它确实包含了一次性启动方法：`setUpPage()` 函数只对每个测试页调用一次，即在所有测试函数调用之前调用。现在，你可能已经发现，这里很适合完成预处理，特别是在运行测试之前如果需要向页面加载一些数据，`setUpPage()` 函数就非常有用。

1. 见 [http://junit.sourceforge.net/doc/faq/faq.htm#organize\\_3](http://junit.sourceforge.net/doc/faq/faq.htm#organize_3)。

不同于 `setUp()` 和 `tearDown()` 函数的是，使用 `setUpPage()` 不只是把处理放在这个函数中就行了的。如果确实选择使用这个特性，一定要保证函数完成时要把 `setUpPageStatus` 变量设置为 `complete`，这就告诉 JsUnit 可以继续，接下来可以执行测试页上的测试了。

想看个例子？那好，我们再来看前面的 simpleJS.js 文件，再增加 3 个函数，补充更多的数学特性。在此包括减法、乘法和除法函数，如代码清单 6-6 所示。

### 代码清单 6-6 simpleJS2.js

```
function addTwoNumbers(value1, value2) {
    return parseInt(value1) + parseInt(value2);
}

function subtractTwoNumbers(value1, value2) {
    return parseInt(value1) - parseInt(value2);
}

function multiplyTwoNumbers(value1, .value2) {
    return parseInt(value1) * parseInt(value2);
}

function divideTwoNumbers(value1, value2) {
    return parseInt(value1) / parseInt(value2);
}
```

下面使用 `setUpPage()` 函数建立一些简单的测试数据，如代码清单 6-7。请注意函数最后一行，必须告诉 JsUnit 你已经建好了测试页。

**代码清单 6-7 使用 `setUpPage()` 函数**

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<html>
  <head>
    <title>Using setUp and tearDown</title>
    <script language="JavaScript" src="jsunit/app/jsUnitCore.js"></script>
    <script language="JavaScript" src="simpleJS2.js"></script>
    <script language="JavaScript">
      var arg1;
      var arg2;

      function setUpPage() {
        arg1 = 2;
        arg2 = 2;
        setUpPageStatus = "complete";
      }
    </script>
  </head>
  <body>
    <h1>setUp and tearDown Test</h1>
    <p>This test demonstrates the use of the setUp and tearDown methods in jsUnit. The setUp method is called before each test case, and the tearDown method is called after each test case. In this example, the setUpPage method is used instead of the standard setUp method, demonstrating how to set up multiple variables for a test case.
  </body>
</html>
```

```

        function testAddValidArgs() {
            assertEquals("2 + 2 should equal 4", 4, addTwoNumbers(arg1, arg2));
        }

        function testSubtractValidArgs() {
            assertEquals("2 - 2 should equal 0", 0, subtractTwoNumbers(arg1, arg2));
        }

        function testMultiplyValidArgs() {
            assertEquals("2 * 2 should equal 4", 4, multiplyTwoNumbers(arg1, arg2));
        }

        function testDivideValidArgs() {
            assertEquals("2 / 2 should equal 1", 1, divideTwoNumbers(arg1, arg2));
        }

    </script>
</head>
<body>
    This is an example of using setUpPage.
</body>
</html>

```

这里还是得到一个绿条（见图 6-7），你可能有点厌倦了吧。

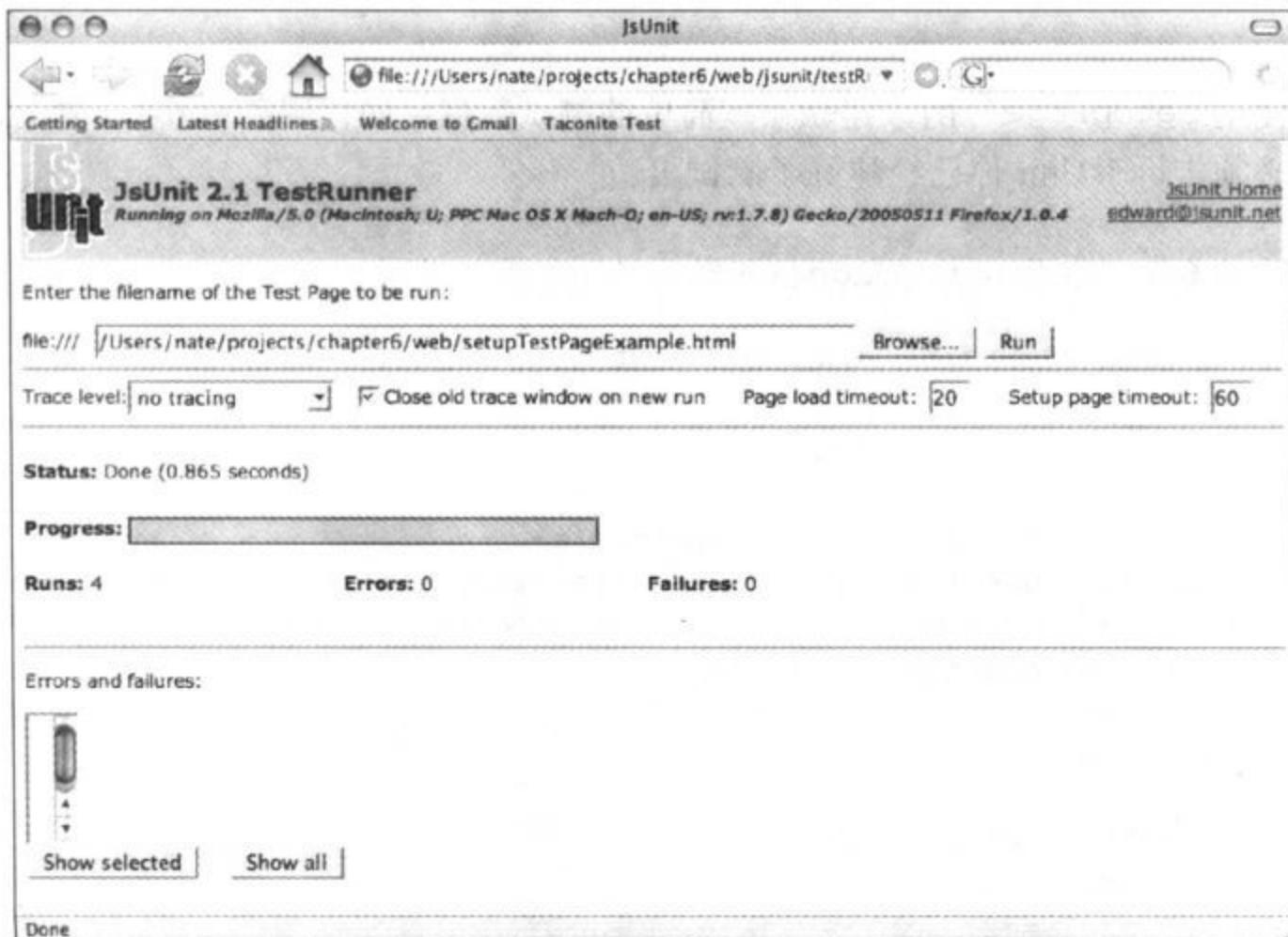


图 6-7 运行 setUpPage() 测试

## 测试集

有了一些测试页之后，你可能想把它们组织为测试集（test suite，也称测试套件），这与JUnit中的TestSuite很相似。测试集把不同的测试页分组组织，这样只需运行一个测试集就能一次运行类似的一组测试。测试集其实就是一些特殊的测试页，其中包含的测试页或其他测试集（相应地就有了一个主测试集）会按顺序运行。

可以采用定义测试页的方式来定义测试集，不过有两个例外。首先，测试集中不能包含任何测试函数；其次，你的测试集必须包含一个返回JsUnitTestSuite对象的suite()函数。可以使用两个方法向测试集中增加测试页或子测试集：addTestPage(testPage)和addTestSuite(testSuite)。前者向测试集中增加单个的测试页；后者向测试集中增加另一个测试集。要记住，向测试集中增加一个测试页时，需要提供一个完全限定名，或者要提供测试页文件相对于测试运行工具的相对路径名。换句话说，如果jsunit文件夹在测试页所在的目录下，那么测试运行工具就比你的测试更深一个层次，在下一层文件夹中。如果看到图6-8所示的一个错误，要确保提供的路径确实是相对于测试运行工具的相对路径。如果想向测试集中增加其他的测试集，要记住，提供给addTestSuite的参数类型必须为JsUnitTestSuite，它要在suite函数所在的同一个页面中声明。你可在jsunit/tests目录中看到一个测试集例子，希望对你有所帮助。代码清单6-8也显示了一个例子。

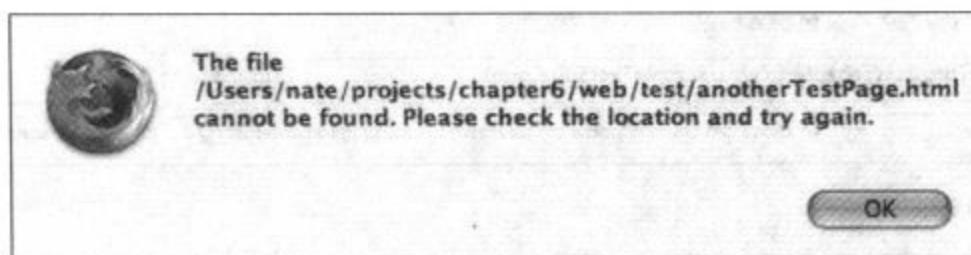


图6-8 运行一个有路径错误的测试集

### 代码清单6-8 一个简单的测试集

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<html>
  <head>
    <title>Sample Test Suite</title>
    <script language="JavaScript" src="jsunit/app/jsUnitCore.js"></script>
    <script language="JavaScript">

      function sampleSuite() {
        var sampleSuite = new top.jsUnitTestSuite();
        sampleSuite.addTestPage("../anotherTestPage.html");
        sampleSuite.addTestPage("../simpleTestPage.html");
        return sampleSuite;
      }
    </script>
  </head>
  <body>
    <h1>Sample Test Suite</h1>
    <p>This page contains a sample test suite for JsUnit. To run the tests, click the 'Run Tests' button below.</p>
    <button type="button" onclick="runTests();">Run Tests</button>
  </body>
</html>
```

```

function suite() {
    var testSuite = new top.jsUnitTestSuite();
    testSuite.addTestSuite(sampelSuite());
    testSuite.addTestPage("../setUpTearDownExample.html");
    return testSuite;
}

</script>
</head>
<body>
    This is a simple test suite.
</body>
</html>

```

正如你所想，这会得到一个绿条（见图 6-9）。

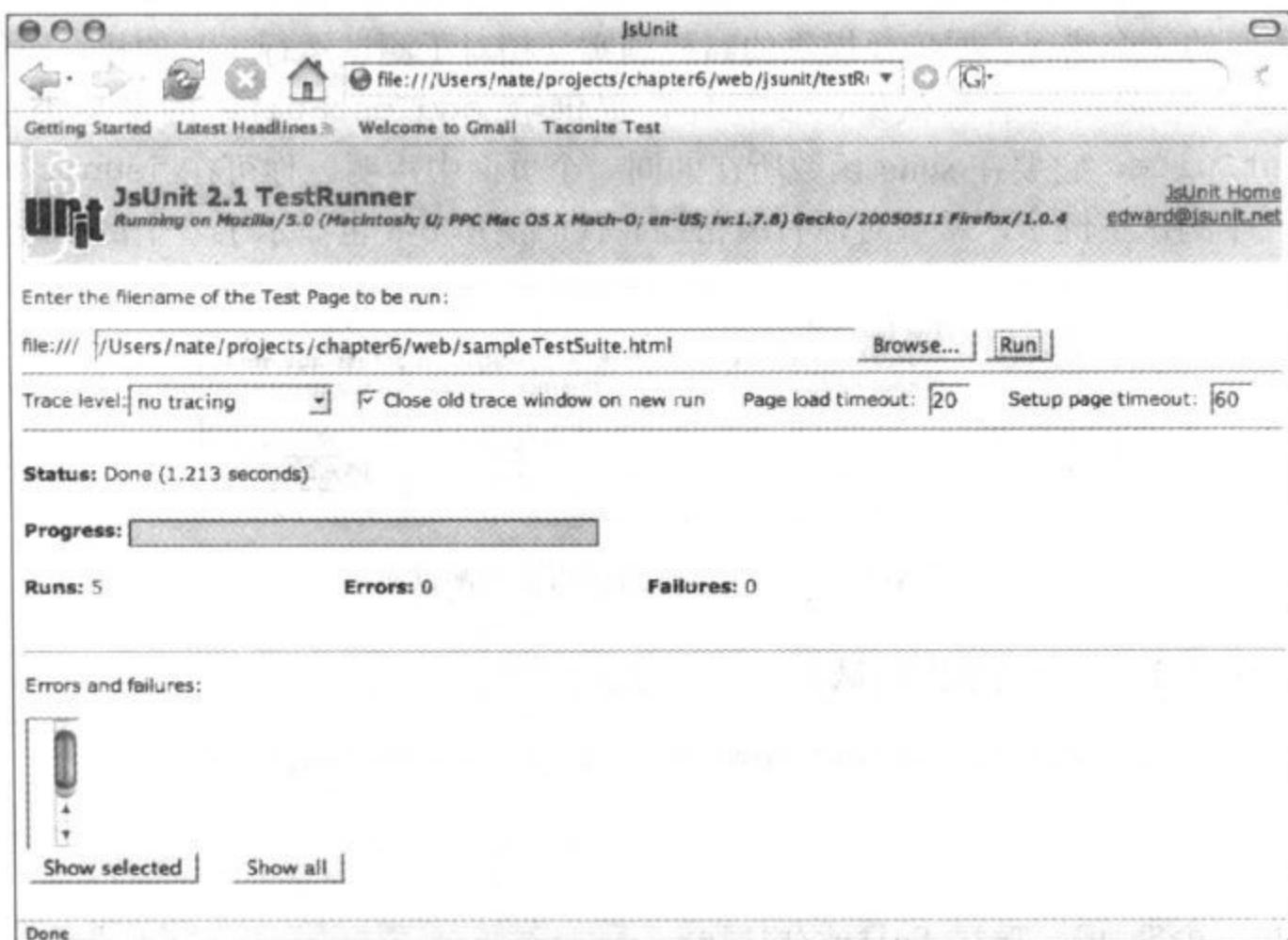


图 6-9 运行测试集

## 跟踪和日志

编写 JavaScript 时最困难的一部分就是跟踪代码。因为 JavaScript 不同于其他许多语言，没有一个得力的日志库来帮助你，因此无法以一种一致的方式打印语句，相反，你必须使用 `alert()`。当然，`alert()`也不是不行，但它肯定不是最理想的方法。为了查找一个问题，要“沿路”布下一大堆 `alert()` 函数，这是很讨厌的，而且一旦修正了 bug，还要再把这一

大堆 alert() 代码统统去掉。当然，等你删除掉所有额外的 alert() 函数后，没准又会出现另一个 bug，而且就出现在上一个 bug 的附近，这就要求你又得把所有 alert() 函数再加上。你现在应该知道，为什么没有多少人喜欢 JavaScript 了吧！

为了让 JavaScript 开发人员的日子更好过，JsUnit 支持跟踪！JsUnit 包含以下 3 个函数，任何测试都可以调用（注意，在每个函数中，value 参数是可选的）：

```
warn(message, [value])
inform(message, [value])
debug(message, [value])
```

JsUnit 支持 3 个跟踪级别：warn（警告）、info（信息）和 debug（调试）。运行测试时，要指定你想在哪个级别上输出。这 3 个级别按以下顺序层叠：warn, info, debug。这说明，如果运行测试时选择 debug，就会看到 warn()、inform() 或 debug() 函数发出的所有消息。如果选择 warn，则只会显示由 warn() 函数发出的消息，选择 info 则会显示由 warn() 和 inform() 发出的消息。默认值为 no tracing（不跟踪）。下面向这个简单例子增加一些跟踪函数<sup>1</sup>，来看看会发生什么（见代码清单 6-9）。

### 代码清单 6-9 向一个测试增加跟踪函数

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<html>
  <head>
    <title>A Simple Test Page with Tracing</title>
    <script language="JavaScript" src="jsunit/app/jsUnitCore.js"></script>
    <script language="JavaScript">
      function addTwoNumbers(value1, value2) {
        warn("this is a warning message");
        warn("this is a warning message with a value", value1);
        return value1 + value2;
      }

      function testValidArgs() {
        inform("this is an inform message");
        assertEquals("2 + 2 is 4", 4, addTwoNumbers(2, 2));
      }

      function testWithNegativeNumbers() {
        debug("this is a debug message");
        assertEquals("negative numbers: -2 + -2 is -4", -4,
                    addTwoNumbers(-2, -2));
      }
    </script>
  </head>
  <body>
    <h1>JsUnit Test Page</h1>
    <p>This page contains a simple test of the addTwoNumbers function.</p>
    <p>The test passes if the result is 4.</p>
    <p>The test also passes if the result is -4 when negative numbers are used.</p>
  </body>
</html>
```

<sup>1</sup> 原文是“方法”，有误。——译者注

```

        }
    </script>
</head>
<body>
    This is a simple test page for addTwoNumbers(value1, value2) with tracing.
</body>
</html>

```

要看跟踪函数得到的输出，需要在测试运行工具中启用跟踪，并选择适当的跟踪级别。如果选择 debug，可以看到来自这 3 个函数的全部消息，“Close Old Trace Window on New Run”（下一次运行测试时关闭上一个跟踪窗口）复选框的作用是，如果你愿意，可以保留以前运行测试的跟踪结果（见图 6-10）。

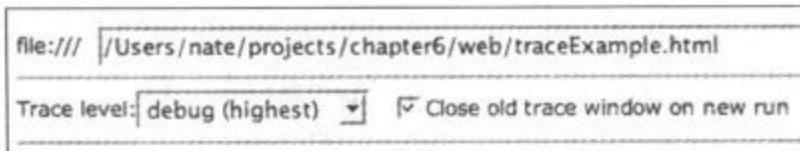


图 6-10 设置跟踪级别

这一回不再是在测试运行工具中千篇一律地显示一个绿条了。图 6-11 显示了跟踪函数的输出。

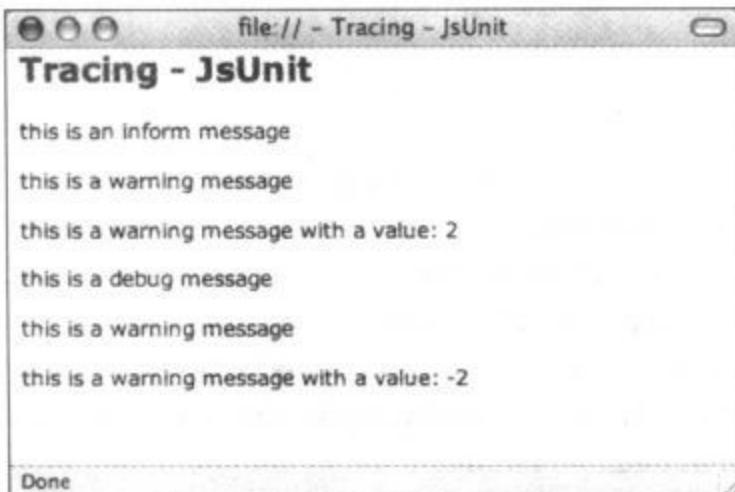


图 6-11 跟踪输出

### 6.2.3 运行测试

你已经写了一些测试，下面需要运行它们，为此可以使用 JsUnit 测试运行工具。你已经看到启动测试运行工具的许多例子，不过这是怎么做到的呢？为了访问这个运行工具，要把浏览器指向 jsunit 文件夹中的 testRunner.html 文件。这个测试运行工具如图 6-12 所示。

这个测试运行工具非常类似于 JUnit 中常用的图形化运行工具。（不过，有意思的是，我们期盼已久的 JUnit 4 并没有包括图形测试运行工具，而且以后也不会增加。）要运行一个测试，可以点击 Choose File（选择文件），来选择要运行的文件。不出所料，在测试运行工具

发现失败之前，进度条一直是绿的，如图6-13所示。Runs字段指示测试函数的总数，还可能报告错误或失败。

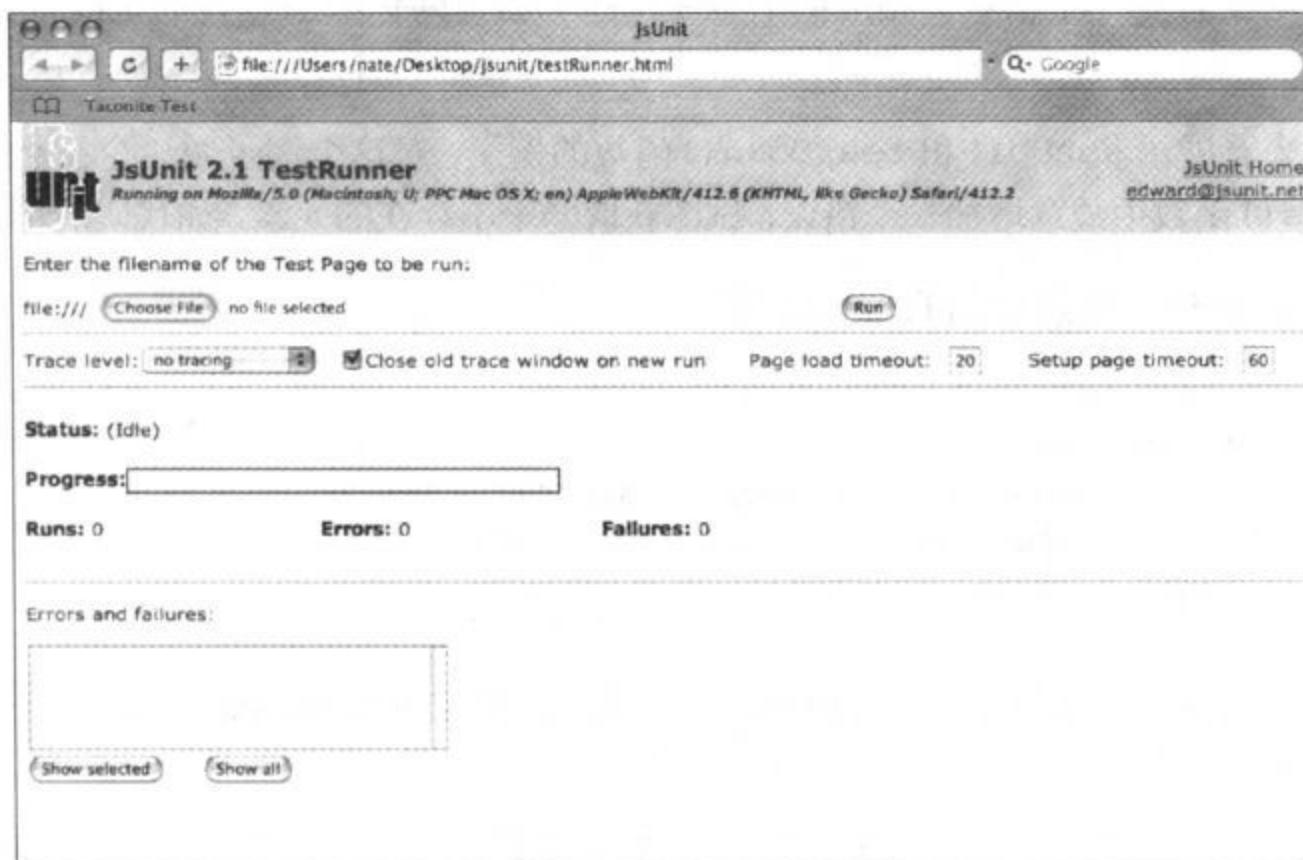


图6-12 JsUnit 测试运行工具

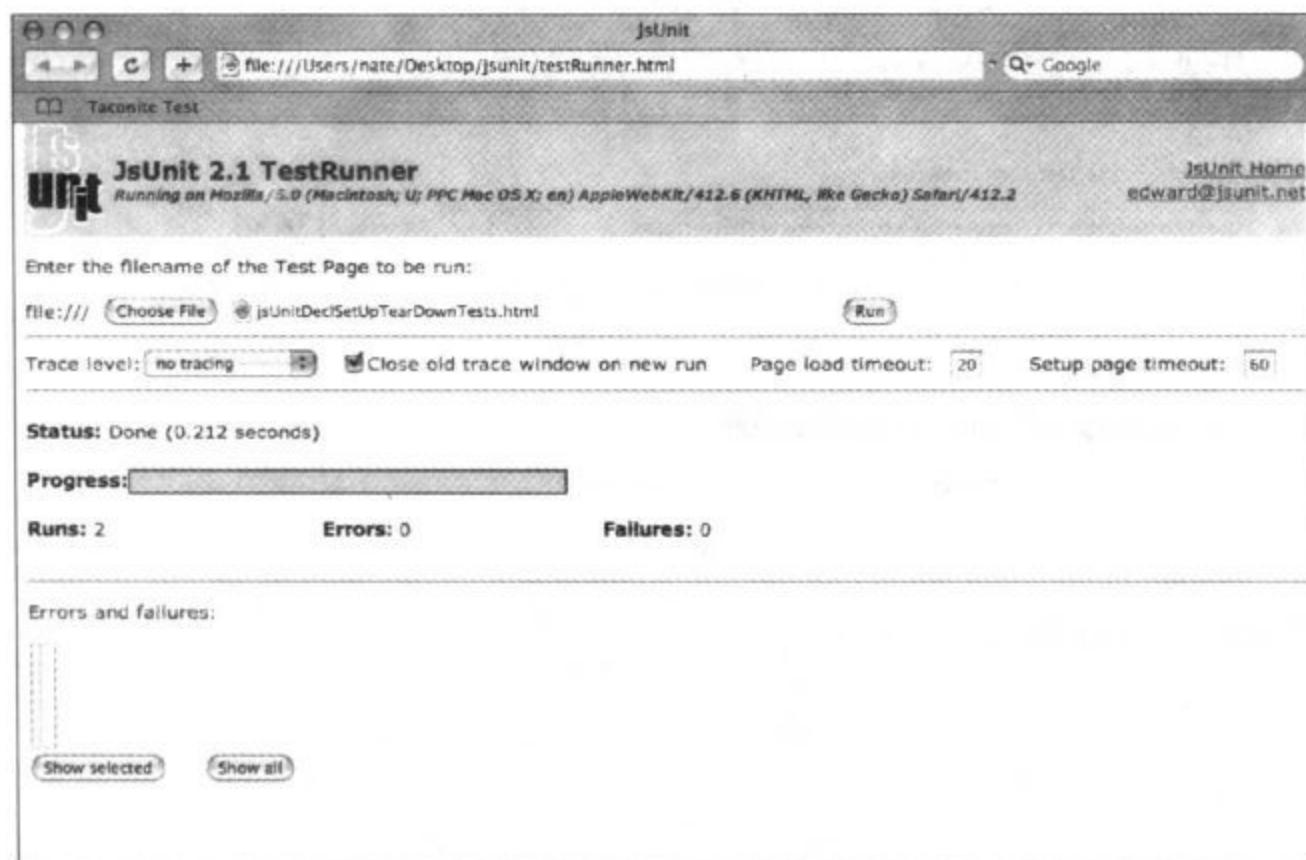


图6-13 成功运行的 JsUnit 测试运行工具

错误(error)来自浏览器，指示测试页出现了某个问题，失败(failure)指示你的某个断言失败。特定的错误或失败会显示在 Errors and Failures 文本框中。要了解一个错误或失败

的更详细的信息，可以双击相应测试函数。或者，选中这个测试函数，再选择 Show Selected。如果是失败，就会出现一个警告，显示出期望值和实际值，另外还会显示你在断言中增加的所有消息。如果看到一个错误，相应的消息（可能）会帮助你缩小查找的范围，更快地找到问题所在。

下面再来看展示 `setUp()` 和 `tearDown()` 函数的例子，对这个例子做一个小小的调整。在 `addNumbers()` 函数中故意增加一个错误，试图获取一个不存在的元素，如代码清单 6-10 所示。

### 代码清单 6-10 故意增加的一个错误

```
function addNumbers() {
    //arg1 doesn't exist!
    var val1 = document.getElementById("arg1").value;
    var val2 = document.getElementById("value2").value;
    return addTwoNumbers(val1, val2);
}
```

运行这个测试，可以想像到，会得到一个红条！注意 Errors and Failures 文本框中显示了一个错误（见图 6-14）。

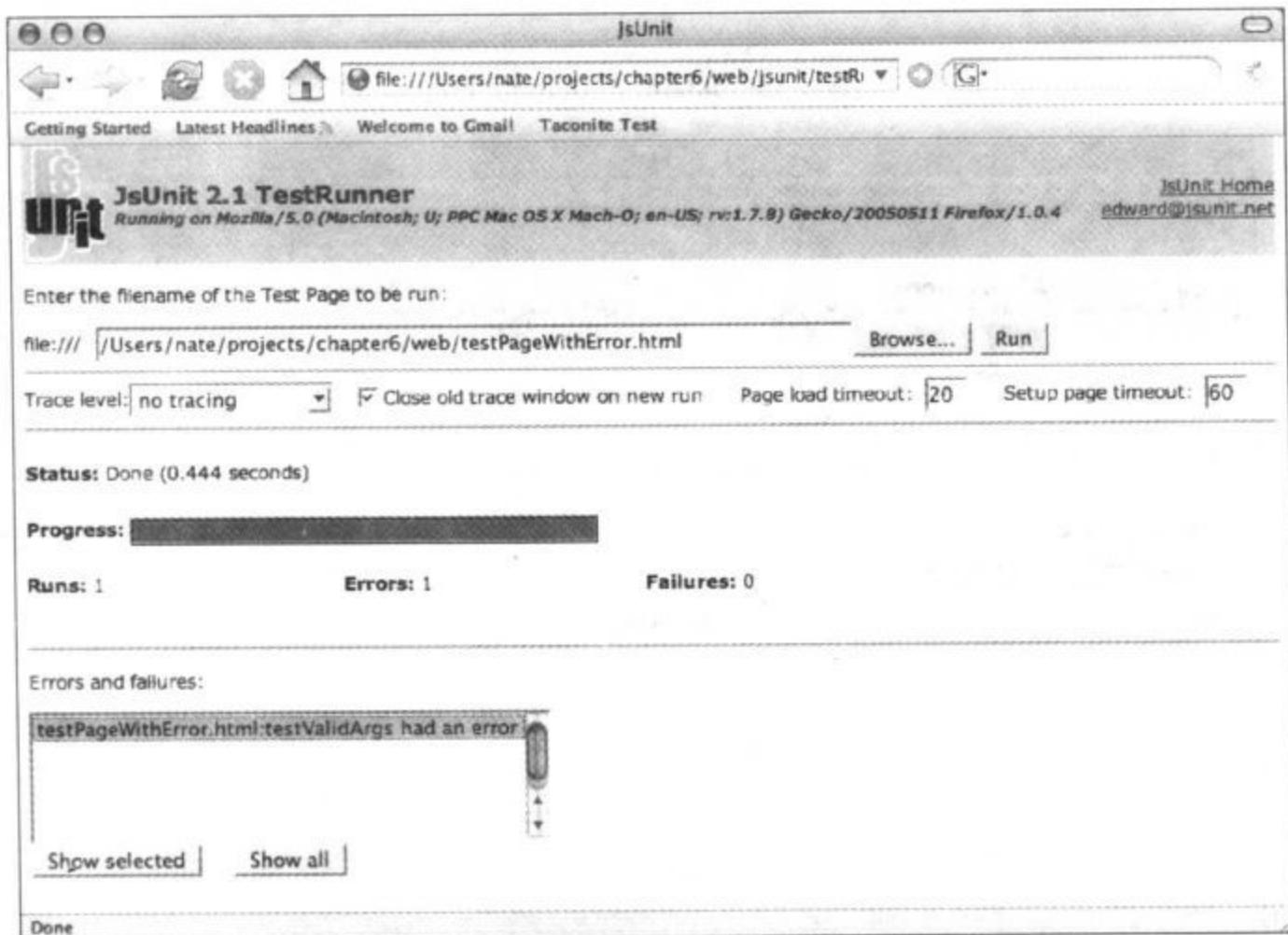


图 6-14 JsUnit 测试运行工具错误

通过进一步观察，可以看到如图 6-15 显示的详细信息。由此得知，`arg1` 没有任何属性，你应该查查 `addNumbers()` 方法来看发生了什么问题。

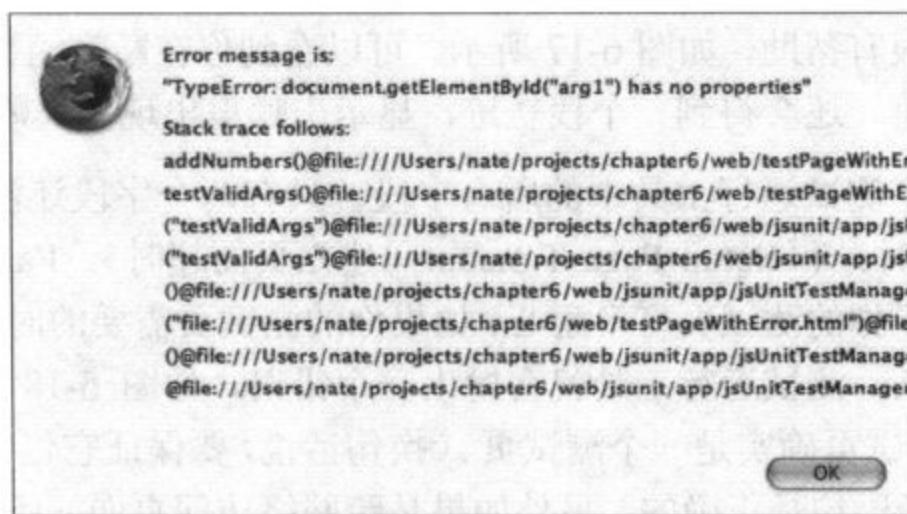


图 6-15 错误详细信息

在运行工具中，失败看上去都是一样的（都会得到一个红条），但是详细信息不同。如果你熟悉 JUnit，会发现这里的失败信息看上去非常熟悉。下面再回到 simpleJS.js 文件的测试。假装你希望  $2 + 2$  等于 5，如代码清单 6-11 所示，来看看会发生什么。

### 代码清单 6-11 另一个故意的错误

```
function testValidArgs() {  
    assertEquals("we really do know that 2 + 2 is 4", 5, addTwoNumbers(2, 2));  
}
```

不出所料，测试运行工具会显示一个红条，但是提供详细信息的警告有所不同，见图 6-16。

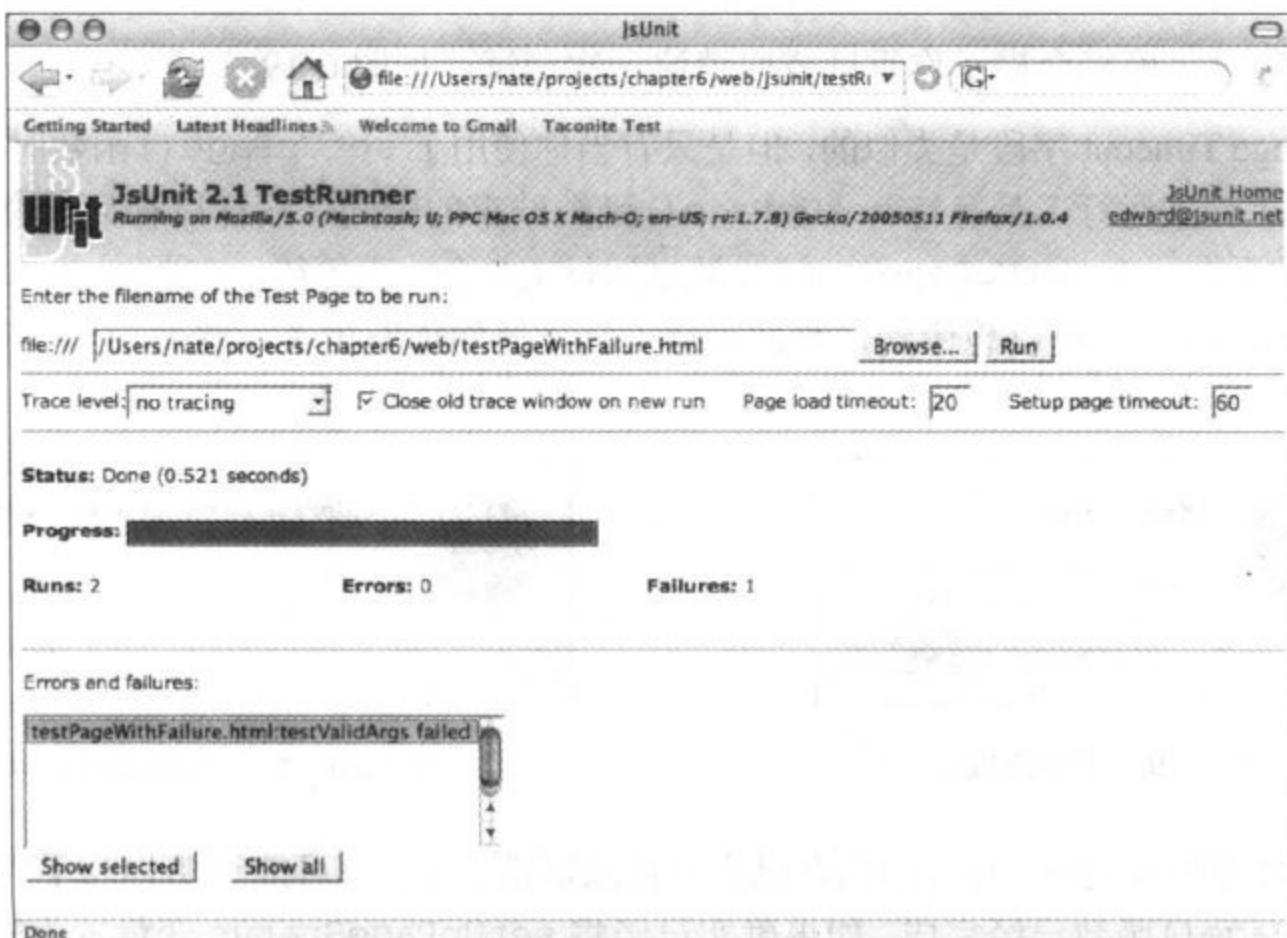


图 6-16 JsUnit 测试运行工具失败

失败的详细信息很有帮助，如图 6-17 所示。可以看到你在断言函数中键入的所有消息，会得到期望值和实际值，还会得到一个栈轨迹，显示出哪里出现了失败。

以上基本上介绍了测试运行工具中的所有字段，只有两个字段还没有谈到：Page Load Timeout（页面加载超时）和 Setup Page Timeout（建立页面超时）。Page Load Timeout 字段是指测试运行工具对于你的测试页有多耐心，如果你的测试页需要的时间比这个框中指定的时间（以秒为单位）长，测试运行工具就会抛出一个错误，如图 6-18 所示。如果看到这个错误，就要确保你的测试页确实是一个测试页。（换句话说，要保证它包含了 `jsUnitCore.js` 文件，而且要确保路径语句是正确的，另外如果是跨网络访问页面，还要保证可以得到这个页面）。当然，如果你在做分布式测试，测试需要的时间就会比较长，这时可能需要增加 Page Load Timeout 字段中的时间。

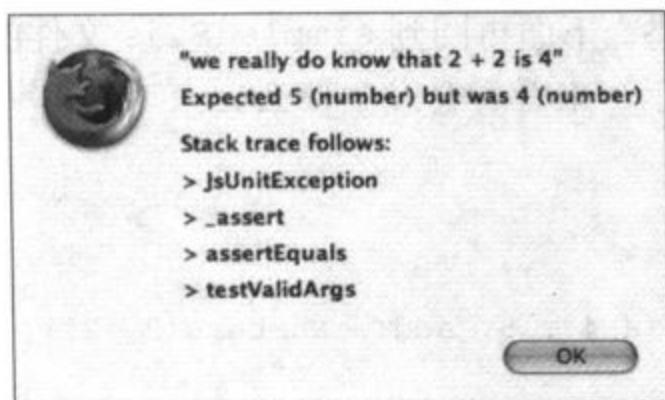


图 6-17 失败详细信息

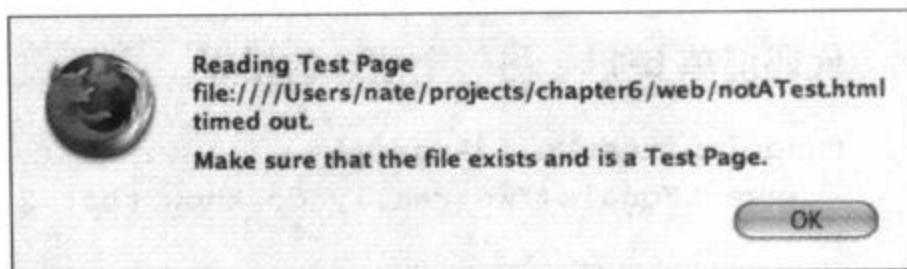


图 6-18 页面超时警告

点击 OK 时，可能会提示你重试或取消这一次测试，如图 6-19 所示。

Setup Page Timeout 字段是类似的，但是只有当你使用了 `setUpPage()` 函数时才适用。它表示了 JsUnit 测试运行工具等待 `setUpPage()` 函数结束的时间（同样以秒为单位）。不要忘了，如果测试页有一个 `setUpPage()` 函数，测试运行工具会一直等待，直到 `setUpPageStatus` 变量置为 `complete`。可以想像到，测试运行工具会通知你发生了一些意想不到的事情，如图 6-20 所示。

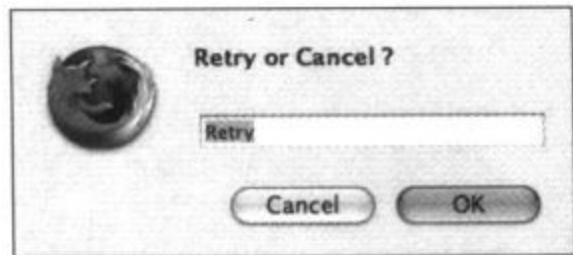


图 6-19 重试或取消

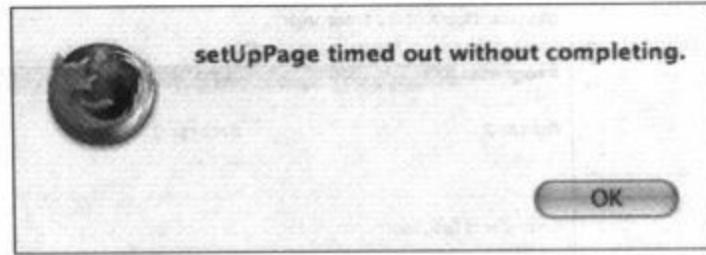


图 6-20 建立页面超时错误

类似于页面加载超时，此时你也有机会重试或取消。这很有可能说明你忘了告诉 JsUnit：你的 `setUpPage()` 函数已经完成。如果确实已经将 `setUpPageStatus` 变量置为 `complete`，可能得查看一下在 `setUpPage()` 函数里做了些什么。你可能需要增加 Setup Page Timeout 字

段的设置值，或者想办法加快建立页面的速度。

#### 6.2.4 使用标准/定制查询串

如此说来，测试运行工具是很强大的，但是每天都得打开这个测试运行工具，并浏览你的测试页或测试集，这可能非常烦人。好在可以使用一些查询串来预置要运行的文件，让测试运行工具自动地运行测试，甚至可以向测试传递参数！

下面先介绍一些基础知识。假设你有一个测试集，每天早上都要运行，然后你才能喝上早茶。当然，你完全可以打开测试运行工具文件，浏览要运行的测试集，然后点击 Run；不过，如果你最近读过 Mike Clark 所著的 *Pragmatic Project Automation* (Pragmatic Programmers 公司 2004 年出版)，可能希望有一种办法能自动地完成这种重复性的任务。好在测试运行工具支持 `testPage1` 查询串。在浏览器地址栏中键入以下地址（当然，根据具体环境可能要做相应调整），就会在浏览器中启动测试运行工具，并且预置了给定的测试：

```
file:///Users/nate/projects/chapter6/web/jsunit/testRunner.html?testPage=/Users/nate/projects/chapter6/web/sampleTestSuite.html
```

就像魔法一般，你会看到测试运行工具运行了起来，你传递给 `testPage` 的实参会显示在 file (文件) 框中，如图 6-21 所示。注意，现在没有 Browse (浏览) 按钮了，因为你已经告诉测试运行工具要运行什么，它已经知道了。



图 6-21 使用 `testPage` 查询串

1. 有意思的是，`testpage` 也可以。

当然，聪明的开发人员可能会对经常运行的测试集建立书签。如果你觉得 testPage 还不错，可能还想让 autoRun 试试身手！顾名思义，可以结合 autoRun<sup>1</sup>和 testPage 来加载你最喜欢的测试页或测试集，然后自动运行。这样一个示例查询串如下所示：

```
file:///Users/nate/projects/chapter6/web/junit/testRunner.html?testPage=/Users/nate/projects/chapter6/web/testPageWithExposeTests.html&autoRun=true
```

你应该能看到一个漂亮的绿条（见图 6-22）。注意这里有 Run 按钮。（既然还是不能浏览其他测试集或测试页，为什么还要一个 Run 按钮呢？）因此，你可以很轻松地重新运行你的测试。同样地，能干的程序员很可能在浏览器上为这个查询增加书签。

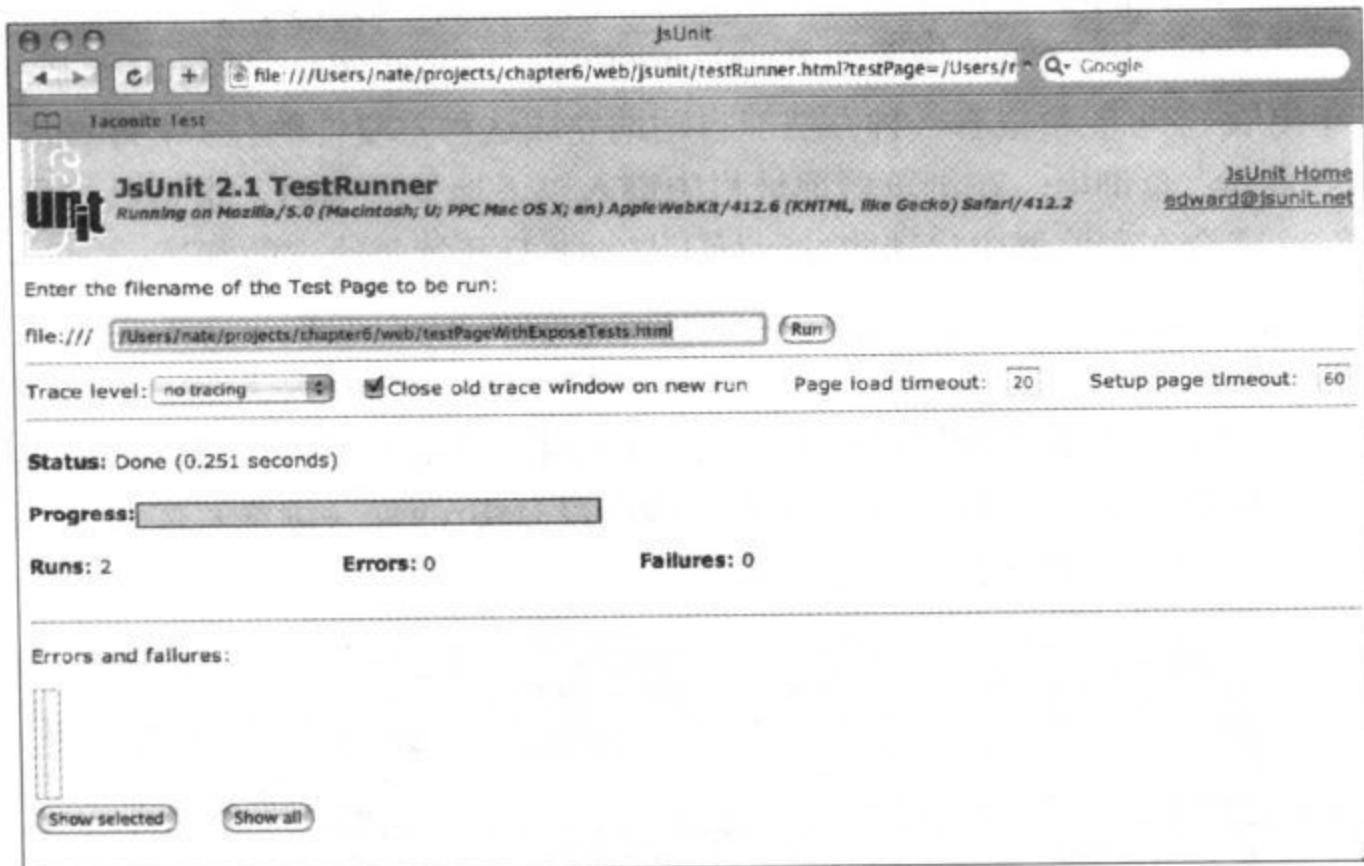


图 6-22 使用带 autoRun 的 testPage 的例子

你可能认为开发人员只需要 testPage 和 autoRun，但实际上还不只如此！在后台，测试运行工具会把各个测试页增加到一个不可见的帧。大多数情况下这样是可以的，但是如果有些代码必须在一个可见的帧中运行，而且你使用的是 IE，就可以告诉测试运行工具要在一个可见帧中显示你的页面。如果在查询串中加入了 showTestFrame，JsUnit 就会在一个可见的帧中显示你的页面。如果你输入了 showTestFrame=true，测试页会在一个默认高度的帧中显示。如果这个默认值不合适，可以传入一个整数值，将帧高度指定为参数值（以像素为单位）。

有时，你可能想向测试页或测试集传递特定的值。可以在查询串中增加任意的行参/实参对，在测试页或测试集中使用 top.jsUnitParmHash.parameterName 或 top.jsUnit-

1. 想到了吧，autorun 也是可以的。

ParmHash['parameterName']来获取这些值。假设按以下路径打开测试运行工具: file:///Users/nate/projects/chapter6/web/jsunit/testRunner.html?key=1, 就可以使用 top.jsUnitParmHash.key 或 top.jsUnitParmHash['key']来访问参数 key。如果你只想运行某些测试, 这就很有用。代码清单 6-12 显示了一个复杂的例子。

### 代码清单 6-12 使用定制查询的测试集例子

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<html>
  <head>
    <title>Sample Test Suite</title>
    <script language="JavaScript" src="jsunit/app/jsUnitCore.js"></script>
    <script language="JavaScript">

      function sampleSuite() {
        var suiteToRun = top.jsUnitParmHash.suite;
        var sampleSuite = new top.jsUnitTestSuite();
        if(suiteToRun == "other") {
          sampleSuite.addTestPage("../anotherTestPage.html");
        } else {
          sampleSuite.addTestPage("../simpleTestPage.html");
        }
        return sampleSuite;
      }

      function suite() {
        var testSuite = new top.jsUnitTestSuite();
        testSuite.addTestSuite(sampleSuite());
        testSuite.addTestPage("../setUpTearDownExample.html");
        return testSuite;
      }

    </script>
  </head>
  <body>
    This is a simple test suite that uses custom queries.
  </body>
</html>
```

可以想见, 在测试运行工具中为路径追加?suite=other, 这会导致运行 anotherTestPage.html, 而不运行 simpleTestPage.html (见图 6-23)。当然, 针对当前的流控制逻辑, 去掉这个参数, 则会正好相反: 只运行 simpleTestPage.html, 而跳过 anotherTestPage.html。

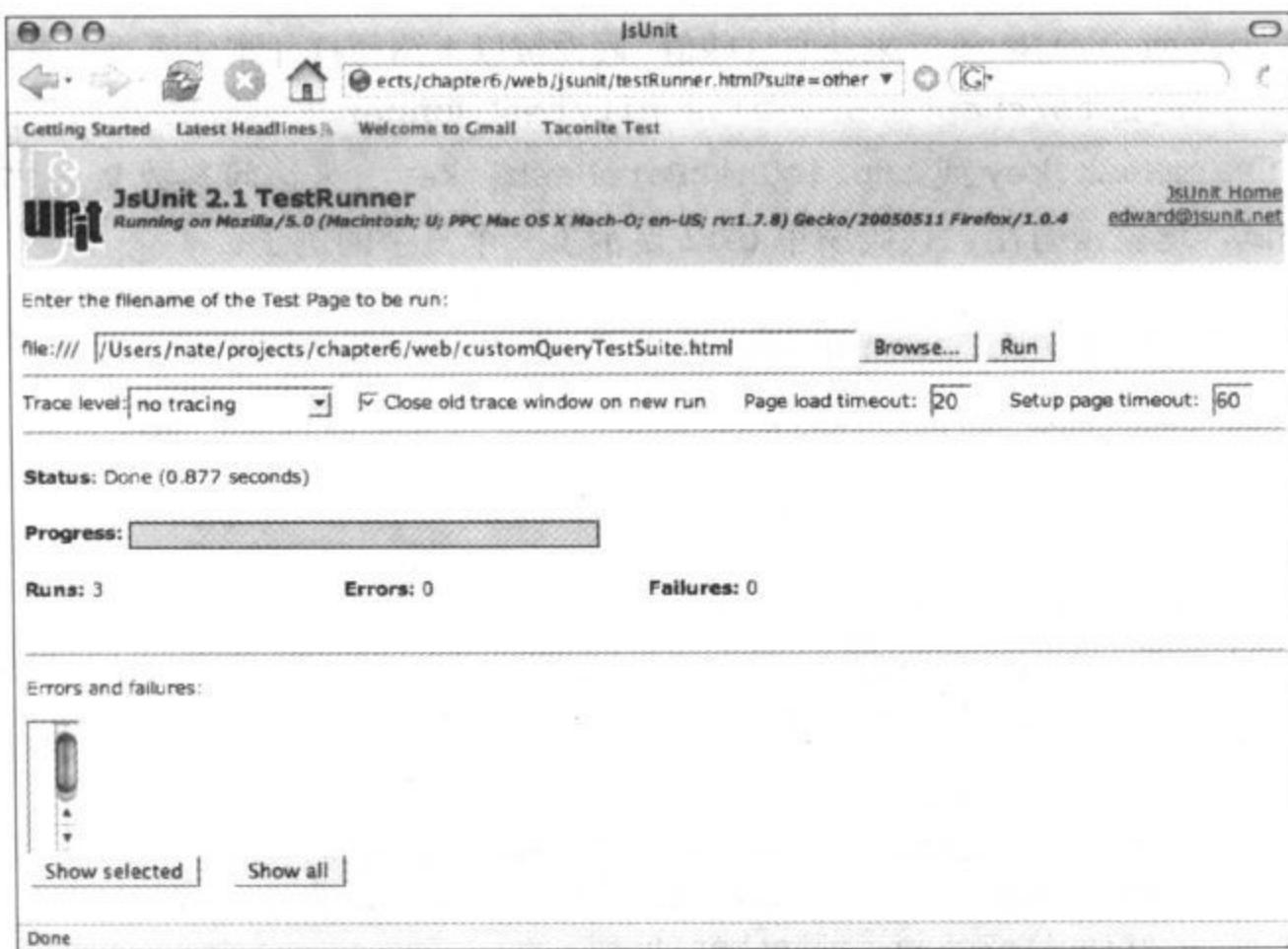


图 6-23 基于一个定制查询参数运行测试集

尽管这显然是一个简单的例子，但是从中可以看出，在不同场合下，使用定制参数来运行测试集或测试页是非常有用的。当然，你可能想为浏览器的书签文件夹增加这些特殊的查询串，以便以后使用。

还有一个有意思的标准参数是 `debug`。不要把这个参数与 `debug` 跟踪级别混为一谈，它

们完全不同。`debug` 参数会在使用 JsUnit 的开发工作中用到。对于这个查询串，你可能不会做太多工作，但是如果你很好奇，可以知道有这样一个参数（见图 6-24）。

另外还有两个标准参数，只有与 JsUnit 服务器结合时才有意义：`submitResults` 和 `resultId`。我们稍后会讨论 JsUnit 服务器，到时你会更清楚地了解这两个参数，不过要知道，在测试运行工具的浏览器路径中增加 `submitResults=true`，这会告诉 JsUnit：将测试的结果发送到

The screenshot shows a browser window titled 'about:blank' displaying a series of JavaScript statements. These statements are part of a debug query used to inspect the internal state of the JsUnit TestManager. The code includes various `enter` and `exit` statements, as well as calls to `loadPage`, `setStatus`, and `_setTextOnLayer` methods. The entire sequence is designed to check the behavior of the test manager when navigating between pages.

```

enter window.jsUnitTestManager._nextPage()
enter window.jsUnitTestManager._currentSuite()
exit window.jsUnitTestManager._currentSuite() == [jsUnitTestSuite]
enter window.jsUnitTestSuite.hasMorePages()
exit window.jsUnitTestSuite.hasMorePages() == true
enter window.jsUnitTestManager._currentSuite()
exit window.jsUnitTestManager._currentSuite() == [jsUnitTestSuite]
enter window.jsUnitTestSuite.nextPage()
exit
window.jsUnitTestSuite.nextPage() == "file:///Users/nate/projects/chapter6/web/a
enter
window.jsUnitTestManager.loadPage("file:///Users/nate/projects/chapter6/web/a
enter window.jsUnitTestManager.setStatus("Opening Test Page
"file:///Users/nate/projects/chapter6/web/anotherTestPage.html")
enter window.jsUnitTestManager._setTextOnLayer("mainStatus", "Status:
Opening Test Page
"file:///Users/nate/projects/chapter6/web/anotherTestPage.html")
exit window.jsUnitTestManager._setTextOnLayer("mainStatus", "Status:
Opening Test Page
"file:///Users/nate/projects/chapter6/web/anotherTestPage.html") == null

```

图 6-24 `debug` 查询的输出

JsUnit 的“acceptor” servlet。这对你意味着什么？这可是个秘密，不过可以告诉你，使用这个参数的话，测试运行的结果会创建结果的一个 XML 表示（与 JUnit 的 XML 输出有同样的结构），以便以后获取。

只有当使用 `submitResults=true` 时，`resultId` 查询才有意义。我们只对 JsUnit 服务器的工作稍有涉及，根据我们介绍的这一点点内容，你可能可以猜出：提交到 JsUnit 服务器的各个测试应该有各自唯一的标识符。可以让 JsUnit 来提供自己的 ID，但是如果必须使用你喜欢的编号，也可以通过 `resultId` 参数传递这样一个数。

### 6.2.5 使用 JsUnit 服务器

虽然向测试运行工具传递各种参数可以较容易地完成自动化测试，但是你很快就会厌倦这样以常规方式手工地运行测试，特别是要考虑到多个操作系统上的多个浏览器时，更是如此。你可能想跟踪以前运行的结果，以便进行审计或完成质量保证。为解决这些问题，JsUnit 服务器会提供测试结果的 XML 日志，从 JUnit 或 Ant 脚本运行测试，以及在远程主机上从 JUnit 或 Ant 脚本运行测试。

利用 JsUnit 服务器，你只需点击一个按钮，就能基于你的操作系统/浏览器运行整个测试集。另外，只需在一个重要步骤中把 JavaScript 的测试增加到 Ant 脚本中，就能使之成为构建过程的一部分。JsUnit 服务器包括一组 Java servlet，它们在可嵌入的 Jetty 开源 Web 服务器上运行，这样你就无需在打算测试的每个主机上都配置一个 Web 服务器/servlet 容器。

一旦完成配置，这个过程就很简单了，只是按下一个按钮而已！

#### 配置服务器

在利用 JsUnit 服务器之前，需要先进行配置。为此可以修改 `build.xml` 文件，这个文件在 `jsunit` 文件夹中。这个文件的最前面有一组属性，可以修改这些属性来满足你的需要。这些变量都很好理解，详细内容请见表 6-2。

表 6-2 服务器配置

环境变量	内 容
<code>browserFileNames</code>	你想测试的一组浏览器可执行文件，这是一个完全路径列表，各个路径之间用逗号分隔
<code>url</code>	测试运行工具的 URL，包括适当的查询串来自动运行适当的测试集
<code>port</code>	运行 JsUnit 服务器的端口，如果没有这个变量，会就使用端口 8080
<code>resourceBase</code>	定义 JsUnit 服务器的文档根。如果是空值（一般设置），就会使用 <code>jsunit</code> 安装目录
<code>logsdirectory</code>	运行测试的结果会写到这个目录。如果是空值，则默认为 <code>jsunit/logs</code>
<code>remoteMachineURLs</code>	指定你想在哪些远程主机上运行测试，即这些远程主机的 URL 列表，各 URL 之间用逗号分隔。这些远程主机需要已经安装配置了一个 JsUnit 服务器

一旦配置了测试集（或测试页）和浏览器组合，只需运行 `standalone_test` 目标。在 NetBeans 开发环境中，可以得到如图 6-25 所示的结果。

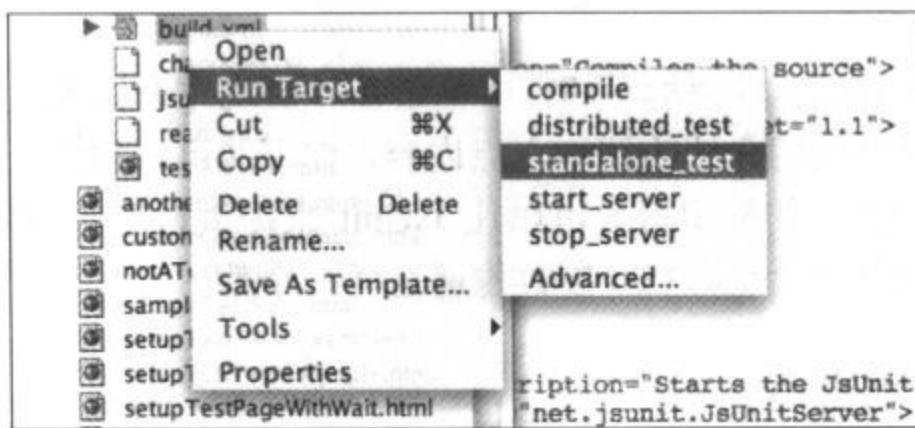


图 6-25 从 NetBeans 运行 `standalone_test` 目标

运行这个目标，会在你指定的端口上启动 Jetty 服务器，而一旦服务器开始运行，就会启动你指定的浏览器，而且会运行你配置的测试。当然，不必特别注意结果，如果出现一个失败或错误，它会显示在 Ant 任务的输出中，指出任务失败。

```
-----
testPageWithError.html:testValidArgs had an error:
Error message is: "TypeError: document.getElementById("arg1") has no properties"
Stack trace follows:
addNumbers()@file:///Users/nate/projects/chapter6/web/testPageWithError.html:21
testValidArgs()@file:///Users/nate/projects/chapter6/web/testPageWithError.html:16
("testValidArgs")@file:///Users/nate/projects/chapter6/web/jsunit/
app/jsUnitTestManager.js:359
("testValidArgs")@file:///Users/nate/projects/chapter6/web/jsunit/
app/jsUnitTestManager.js:359 ()@file:///Users/nate/projects/chapter6/web/jsunit/
app/jsUnitTestManager.js:166 ("file:///Users/nate/projects/chapter6/web/
testPageWithError.html")
@file:///Users/nate/projects/chapter6/web/jsunit/app/jsUnitTestManager.js:104
()@file:///Users/nate/projects/chapter6/web/jsunit/app/jsUnitTestManager.js:338
@file:///Users/nate/projects/chapter6/web/jsunit/app/jsUnitTestManager.js:335
-----
```

假设在查询串上增加了 `submitResults=true`，还可以查看日志文件来检查结果。日志文件采用 JUnit 结果同样的 XML 格式，所以像自动化 JUnit 测试集一样，可以用同样的转换很容易地加以处理。可以直接查看 XML 文件，也可以使用 JsUnit 内置的“`displayer`”servlet。

要使用“`displayer`”servlet，首先确保 JsUnit 服务器已经运行。如果未运行，只需运行 `start_server` 目标。一旦 JsUnit 服务器开始运行，打开你最喜欢的浏览器，指向 `localhost:8080/jsunit/displayer?id=×××`，这里的×××是你想查看的结果日志的 ID。你会看到测试运行的结果，如图 6-26 所示。

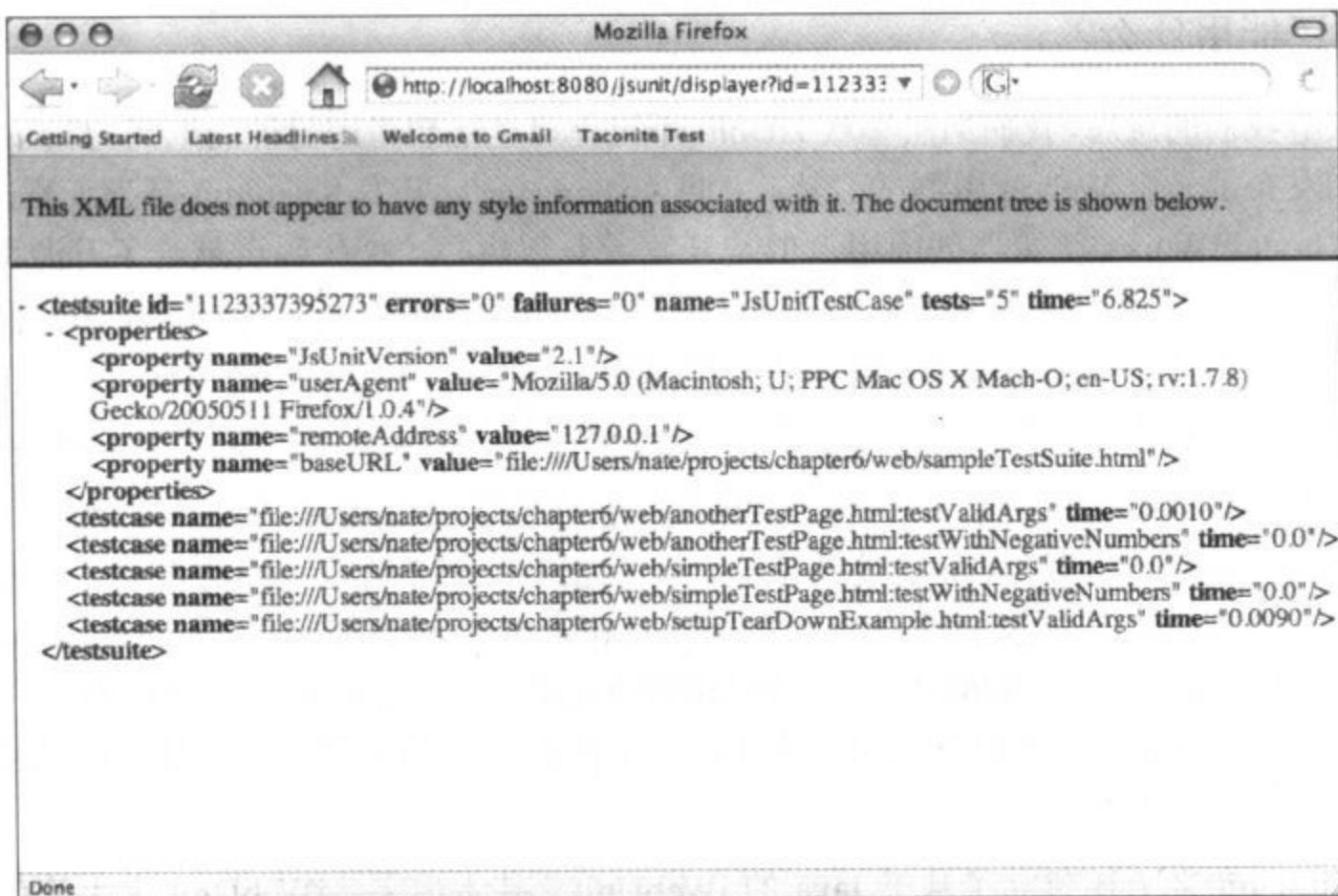


图 6-26 查看测试运行的结果

### 在远程主机上运行测试

如果想在分布式主机上运行测试集，需要使用 `distributed_test` Ant 任务。当然，必须在要用的每个远程主机上配置一个 JsUnit 服务器，并配置适当的测试和浏览器。运行 `distributed_test` Ant 任务会调用一个 JUnit 测试 (`net.jsunit.DistributedTest`)，它会进一步在配置中提供的每个远程主机上调用一个 servlet。这个 servlet 运行的基于 JUnit 的测试与本地运行的测试 (`net.jsunit.StandaloneTest`) 是一样的。然后，会在远程主机上配置的浏览器中运行测试，结果发送回发起者。

#### 6.2.6 获得帮助

在这一章中，我们只是触及 JsUnit 的皮毛，所以如果你想到别处了解更多信息，也没有关系，我们不会难过的。当然，你的第一站应当是 JsUnit 网站 ([www.edwardh.com/jsunit/](http://www.edwardh.com/jsunit/))。实际上，本章许多内容都是从这里参考得来的。如果你有具体的问题，可以通过电子邮件询问 JsUnit 的创始人 Edward Hieatt，或者可以加入新闻组 [groups.yahoo.com/group/jsunit/](http://groups.yahoo.com/group/jsunit/)。你不会被大量邮件淹没，2005 年每个月的邮件不会超过 10 封。不过，对于常见的问题，这是一个提供解答的绝好资源。如果你发现 bug，可以提交给 SourceForge bug tracker ([https://sourceforge.net/tracker/?group\\_id=28041&atid=391976](https://sourceforge.net/tracker/?group_id=28041&atid=391976))。除此以外，还可以在 Google 上查看能找到哪些资源。

### 6.2.7 还能用什么？

别误解我们的意思，我们当然喜欢 JsUnit，不过你可能还想考虑另外的选择。ThoughtWorks 的高手们发布了一个 Web 应用测试工具，名叫 Selenium<sup>1</sup>。开发 Selenium 是为了测试一个基于内部 ThoughtWorks 浏览器的应用，但是开发者非常热心，把代码贡献给了开源世界，所以我们也能有幸享用。Selenium 可以在所有主要浏览器上运行，而且可以在常用的操作系统上运行。类似于 JsUnit，Selenium 测试也在浏览器上运行，因此非常适合测试系统功能和浏览器兼容性。要了解更多的有关内容，请参见 [selenium.thoughtworks.com/index.html](http://selenium.thoughtworks.com/index.html)。

如果你在服务器端使用了 Java，还有很多其他的选择。HttpUnit 是用 Java 编写的，可以用来模拟浏览器。利用 HttpUnit，能模拟提交一个表单，测试请求返回的页面，并检查基本 JavaScript。一般地，你可能会结合使用 HttpUnit 和 JUnit。HtmlUnit 类似于 HttpUnit，不过它对页面建模，而不像 HttpUnit 那样对请求和响应建模。HtmlUnit 模拟了浏览器，并与 JUnit 结合使用。HttpUnit 有一个很有意思的特性，它能模拟特定的浏览器，因此你可以测试任何特定于浏览器的逻辑。

在 HttpUnit 之上还建立了基于 Java 的 jWebUnit。基本说来，jWebUnit 大大简化了导航规则，并提供了一些预置的断言。这个工具同样要与 JUnit 结合使用。JUnit 再向前一步是验收测试框架 FitNesse，这是 Object Mentor 的一些人写的。FitNesse 之所以独树一帜，是因为你的客户可以使用这个框架定义应用应该做什么。你（甚至你的客户）可以创建输入表，并指定运行应用的期望结果。正如你所料，成功的测试会有绿的结果，失败则显示红色。FitNesse 是 xUnit 测试的一个补充，Object Mentor 有一个精辟的说法：xUnit 可以确保你正确地建立了代码，而 FitNesse 可以确保你建立了正确的代码！

这一节很短，主要思想是说明你有很多选择来完成测试。尽管我们介绍的都是免费的工具，但是当然也存在需要花钱的工具。最后我们建议了一种综合的方法，可以充分利用所有这些工具的功能。你可以拿我们在这里谈到的工具小试牛刀，还可以在网上搜索我们没有提到的其他工具，你很快就会找到最适用的方法。

## 6.3 小结

如果你要使用 Ajax，就要使用 JavaScript。尽管一些框架和工具可以简化开发，但是测试还是很困难的一个重要环节。很多人对服务器端代码测试可能很“着迷”，对于 JavaScript，当然更应如此。我们希望以上对 JsUnit 的简短介绍能使你对有关的背景有一定了解，以便着手使用这个测试框架。如果你是一个开发人员，则这不仅能够让你更轻松地完成工作，而且肯定能改善代码的质量。

---

1. 从名字里能看出来什么吗？据一个古老的传说称，Selenium（硒）元素可以治疗汞中毒。

## 分析 JavaScript 调试工具和技术

“**没**有编译器来捕捉错误！没有好的 IDE 提供代码完成之类的生产力工具！没有好的调试环境！”

人们不愿意进行 JavaScript 开发，为此找出了种种理由，以上只是其中的部分原因。从 20 世纪 90 年代末的浏览器之争开始，JavaScript 就有了一个不好的名声，人们一直认为这是一种容易出错、使用困难的开发语言。这个坏名声主要归咎于 JavaScript 早期版本中存在大量的 bug，而那时运行 JavaScript 的 Netscape Navigator 和 IE 早期版本也同样存在大量 bug。更糟糕的是，JavaScript 代码通常都由非程序员编写，其做法是逐个地尝试一遍，直到得到看上去能工作的代码为止。而且网上有很多免费可用的脚本，其中许多脚本的质量相当差，这就进一步败坏了 JavaScript 的名声，让人们越发相信这是一种没水准的编程环境。

与几年前相比，如今的 JavaScript 实现已经让人刮目相看了，这得益于 ECMA 完成了 JavaScript 语法和行为的标准化，一个稳定的 JavaScript 版本，以及更好的 Web 浏览器实现，现在使用 JavaScript 编程已经让人很享受了。另外，一些工具纷纷登场，如完备的调试环境、Ajax 特定的调试工具，以及 JavaScript 错误控制台等等，这为开发人员提供了更多的开发工具选择。

但还是存在一些问题。例如，在写本书时，还缺少有用的 JavaScript IDE 以提供诸如代码完成之类的生产性工具。另外，通常还是那些没有编程背景的人编写 JavaScript 代码。不过，如今可用的工具确实更多了，这能大大缓解原来 JavaScript 开发带来的痛苦，进一步说，可以减轻 Ajax 开发的负担。

本章我们介绍一些调试工具。如果出现了问题，不能如期运行，你就可以使用这些工具，称为调试（debug）。有这样一些工具来帮助你诊断问题，这就成功了一半。一旦真正掌握了这些工具，你就会发现调试实在是乐趣无穷！

## 7.1 用Greasemonkey调试Ajax请求

Ajax 请求大大改善了用户的体验，因为它们出现在后台，不会让浏览器停滞不动，也不会只是指示正在向服务器发出请求。这种行为也有一个不好的副作用，就是页面将更难调试。假设 Ajax 请求或响应存在一个问题，开发人员很难知道请求是在哪里失败的。是因为服务器无法正确地读取请求吗？还是浏览器无法正确地处理服务器的响应？真的向服务器做出了请求吗？

好在有了 Firefox 扩展 Greasemonkey，以及一个相关的用户脚本，这使得调试 Ajax 请求变得容易得多。

### 7.1.1 Greasemonkey 介绍

Greasemonkey 是一个 Firefox 扩展，允许你向任何 Web 页面增加一些 JavaScript 来改变页面的行为，这些 JavaScript 称为用户脚本（user script）。Greasemonkey 与其他的一些 Firefox 扩展类似（例如，有些 Firefox 扩展允许你修改页面的 CSS 样式规则）。Greasemonkey 实际上就是负责完成具体工作的用户脚本的代理。你可以很容易地从 [greasemonkey.mozdev.org](http://greasemonkey.mozdev.org) 主页下载和安装 Greasemonkey。

### 7.1.2 使用 Greasemonkey XMLHttpRequest 调试用户脚本

XMLHttpRequest 调试是 Julien Couvreur 编写的一个 Greasemonkey 用户脚本。这个脚本是一个 Ajax 调试控制台，它能很方便地显示每一个 Ajax 请求及相关的响应。

要安装 XMLHttpRequest 调试用户脚本，首先要确保已经成功地为 Firefox 安装了 Greasemonkey。接下来，让 Firefox 指向 [blog.monstuff.com/archives/images/XMLHttpRequest-Debugging.v1.0.user.js](http://blog.monstuff.com/archives/images/XMLHttpRequest-Debugging.v1.0.user.js)。这就是构成该用户脚本的 JavaScript 文件。从 Firefox 的菜单条选择 Tools→Install User Script（安装用户脚本）。当安装脚本时，你要在 Included Pages 框中输入想在哪个域中完成 Ajax 调试。如果没有正确地设置域，这个用户脚本在希望完成 Ajax 调试时就不会正确启动。这个窗口也可以稍后使用 Firefox 的 Tools→Manage User Scripts（管理用户脚本）菜单项打开。

### 7.1.3 使用 XMLHttpRequest 调试用户脚本检查 Ajax 请求和响应

用户脚本已经安装了，现在将浏览器指向包含你想调试的 Ajax 请求的页面，这就会立即打开 XMLHttpRequest 调试窗口。点击窗口最上面的“help”链接将显示脚本的帮助模块（见图 7-1）。点击 XMLHttpRequest 调试窗口的标题，把它拖动到期望的位置，这样就能在页面上任意移动 XMLHttpRequest 调试窗口。再点击一次“help”链接，就能关掉帮助内容。

保持 XMLHttpRequest 调试窗口打开，接下来初始化一个 Ajax 请求。在初始化请求后，窗口会自行刷新，显示这个 Ajax 请求和相应响应的有关信息。

请求和响应的信息都会在 XMLHttpRequest 调试窗口中显示。信息窗口中的第一行指示了请求方法，如 GET 或 POST，下一行详细列出了请求的 URL，以及构成请求的查询串。如果你把请求参数作为查询串的一部分发送，可以仔细检查这一行，就能很容易地注意到是否有错误。

当使用 XMLHttpRequest 调试时，通过点击“body”链接就能查看请求体。请求体通常为 NULL，除非将一个串或 XML 文档作为参数传递给 XMLHttpRequest 对象的 send() 方法。

请求信息窗口的下半部分详细显示了服务器响应。点击“headers”链接会显示由服务器发送的所有响应首部。

当点击“response”链接时，会显示服务器响应的完整内容。最后，点击“callback”链接，就会显示响应完成时 XMLHttpRequest 所调用的回调方法的文本。如果窗口太小，只需点击“export”链接，请求和响应信息部分的整个内容就可以导出到一个更大的浏览器窗口中。如图 7-2 所示，XMLHttpRequest 调试窗口显示了请求和响应值，并显示了 XMLHttpRequest 对象的回调函数。



图 7-1 XMLHttpRequest 调试窗口的帮助内容

This screenshot displays two panels of the XMLHttpRequest debugging window. The left panel shows the same request and response details as the previous screenshot. The right panel shows the 'callback' section, which contains the JavaScript code for the XMLHttpRequest object's onreadystatechange event handler. The code checks the readyState and status properties, and if successful (status == 200), it calls a function named 'parseSearchResults()'.

图 7-2 显示请求和响应（左）以及 XMLHttpRequest 对象的回调函数（右）

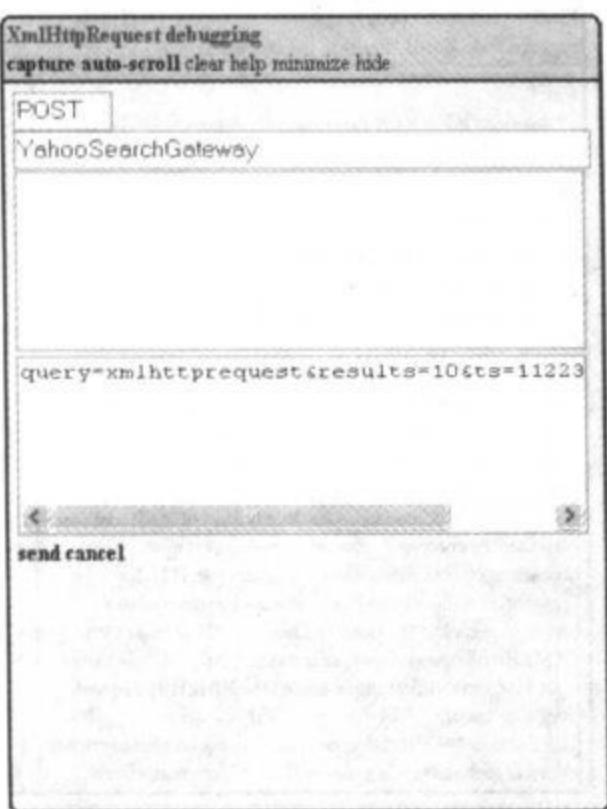


图 7-3 前一个例子的请求修改为使用 POST 方法，查询串从 URL 移到请求体中

XMLHttpRequest 调试有一个功能强大的方面，它允许基于前一个请求创建一个新的 Ajax 请求。点击“edit& replay”链接打开窗口，如图 7-3 所示。可以看到，窗口中列出了最初请求的数据，不过现在所有字段都是可编辑的。你可以修改请求方法，更新 URL，增加一些请求首部，甚至能增加一个数据串或 XML（作为请求体的一部分发送）。点击“send”链接，发送新创建的请求。如果你想对不同的请求类型和数据进行实验，这个工具就很有用，而且无需手工地更新脚本，再在 Web 服务器上部署并重复测试。

使用 Firefox 浏览器的 Greasemonkey 扩展，再结合 XMLHttpRequest 调试用户脚本，这就成为一个强大的 Ajax 调试和开发工具。它有助于准确跟踪浏览器中所发生的事情，如果出现问题，你能得到快速调试问题所需的信息。有了它的帮助，你可以清楚地区分是浏览器端客户脚本出现了问题，还是为请求提供服务的服务器端代码出现了问题。

## 7.2 调试JavaScript

在 Ajax 开发期间，你的 JavaScript 代码迟早都会出现 bug。有些 bug 很容易发现，因为这些错误本质上讲只是语法错误。但另外一些错误查找起来就要困难得多，因为它们是业务逻辑中的一些很微妙的错误。此时，调试工具就能派上用场了，利用调试工具，你就能逐行地跟踪代码，验证脚本是否按预想的流程工作，以及变量是否有正确的值。

可以使用 3 个工具来调试 JavaScript：

**Firefox JavaScript Console:** Firefox JavaScript Console 记录 JavaScript 中出现的所有错误和警告。这个工具使用起来很容易，使用 JavaScript Console，你能很快诊断出大多数错误，而不必依赖完备的调试环境。

**Microsoft Script Debugger:** 这个工具集成在 IE 中，提供了基本的调试功能，例如可以设置断点，还能在运行时检查和修改变量值。这是一个相当原始的工具，不过在某些情况下很适用，如有时可能使用了许多 IE 特定的行为，或者调试错误只在 IE 中出现。

**Venkman:** 作为基于 Mozilla 的浏览器（如 Firefox）的一个扩展，Venkman 是一个功能强大的 JavaScript 调试工具。Venkman 是一个功能很丰富的环境，对断点和对象检

查提供了充分的支持。Firefox 遵循 Web 标准，因此可以确信，如果脚本能在 Firefox 中工作，那么在与标准兼容的任何其他浏览器中也同样能运行。

由于调试在开发过程中可能是很重要的一部分，我们将分别介绍上述工具，并说明它们能做什么（以及不能做什么）。

### 7.2.1 使用 Firefox JavaScript Console

作为解释语言，JavaScript 缺少捕捉一些小错误，如丢失引号，或者大括号不匹配的编译器。JSLint 之类的工具可以提供帮助，但是这些工具会漏掉很多错误。

通常，在浏览器中执行脚本之前，这些错误不会暴露出来。关于 JavaScript 开发有一个常见的抱怨，这就是：当出现 JavaScript 错误时，浏览器不能提供描述性的错误消息。

Firefox 有一个很棒的工具，称为 JavaScript Console，在 Tools 菜单下就能找到。所有 JavaScript 错误都会记录到 JavaScript Console，它描述错误，并指示有错误的源文件的位置<sup>1</sup>。

如图 7-4 所示，JavaScript Console 显示了许多错误。第一个错误是在一个串直接量的最后缺少一个匹配的引号。请注意 JavaScript Console 是如何描述这个错误的，它指出了错误的位置（第 11 行），甚至指示了不匹配的引号。第二个错误出现在一个对象实例中，这里引用了对象的一个不存在的属性。

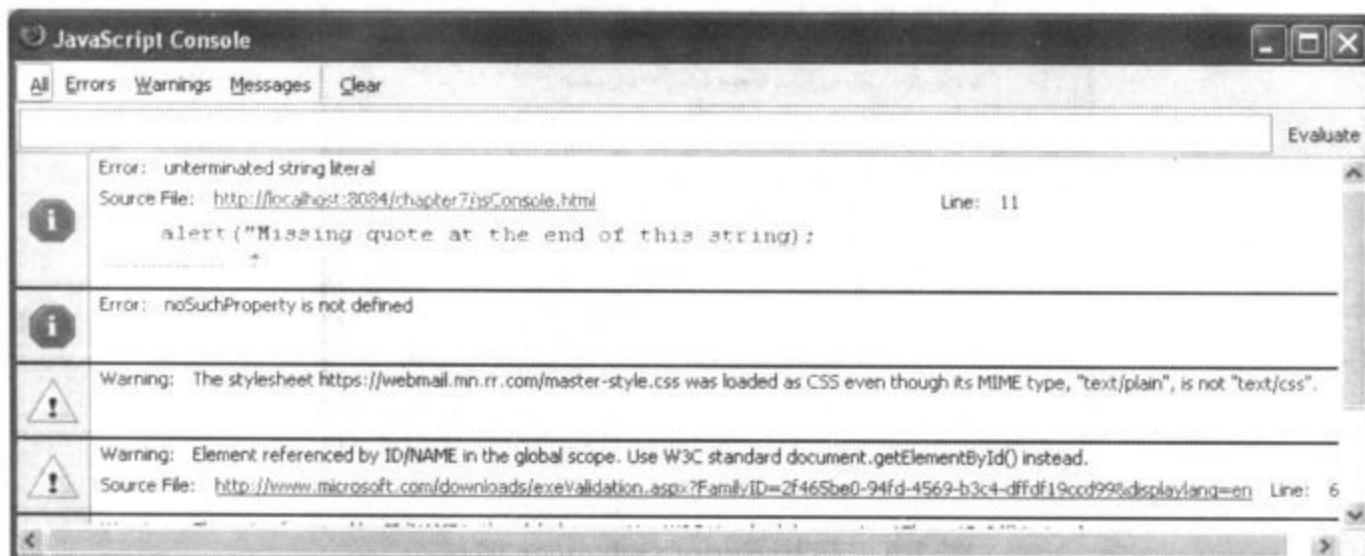


图 7-4 Firefox JavaScript Console 描述错误及错误在源文件中的位置

图 7-4 中的最后两项是信息性警告。有些实例从技术上讲是不正确的，但是 Firefox 并不会恢复，最后两项就是这样的例子。第一个信息性消息描述了一个样式表，其 MIME 类型设置为 text/plain 而不是正确的 text/css。如果能保证提供页面的服务器把 CSS 文件的 MIME

1. JavaScript Console 不但会指示出现错误的源文件在哪里，而且还会更确切地指出源文件中错误所在的位置。——译者注

类型正确地设置为 text/css，就能修正这个错误。图 7-4 显示的最后一个消息提示开发人员引用对象时应该使用 W3C 标准方法，而不要使用专用方法。

## 7.2.2 使用 Microsoft Script Debugger

Microsoft Script Debugger<sup>1</sup> 是 IE 4（或更高版本）的一个扩展，可以用于调试客户端脚本，还能调试在 Microsoft IIS 上运行的服务器端脚本。本节我们将重点强调如何用 Microsoft Script Debugger 调试客户端脚本。

安装 Script Debugger 非常简单，只需下载安装程序，然后运行就行了。完成安装后，需要确保 IE 启用了调试。为此，选择 Tools→Internet Options 菜单，打开 Internet Options（Internet 选项）窗口。在 Advanced(高级)标签 Browsing(浏览)子菜单下，确保 Disable Script Debugging（禁止脚本调试）项没有选中，如图 7-5 所示。

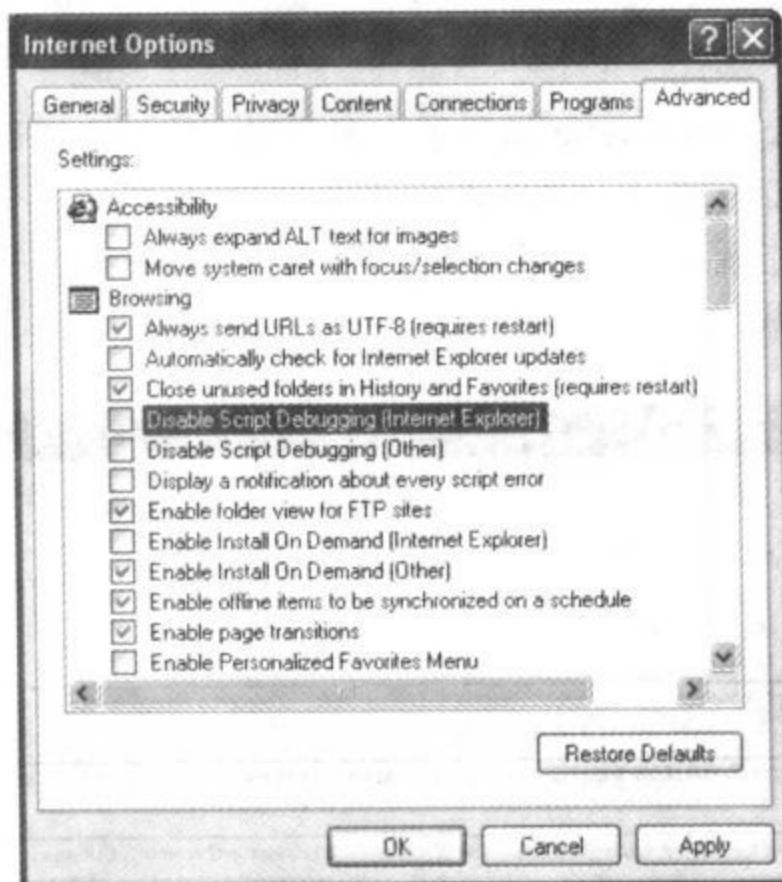


图 7-5 启用 Script Debugger 调试

重启 IE，现在 View 菜单下就有了 Script Debugger（脚本调试工具）菜单。接下来就可以使用 Script Debugger 了。导航到含有一些你想调试的 JavaScript 的某个 Web 页面。要打开 Script Debugger，可以选择 View→Script Debugger→Open 菜单项。这样打开的 Script Debugger 就会显示你目前查看的 HTML 页面的代码，如图 7-6 所示。

1. 可以从 <http://www.microsoft.com/downloads/details.aspx?familyid=2f465be0-94fd-4569-b3c4-dffdf19ccd99&displaylang=en> 获得。

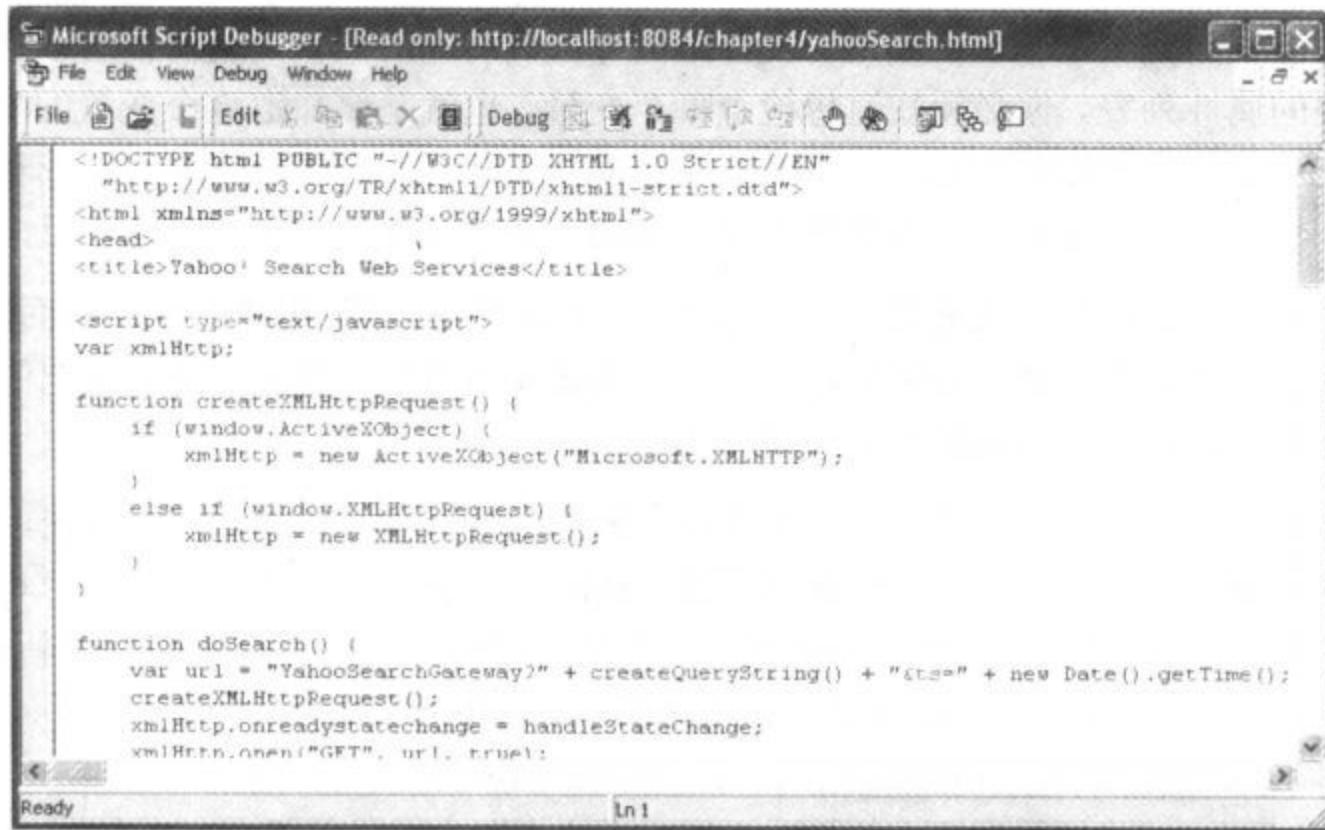


图 7-6 Microsoft Script Debugger 显示当前 HTML 页面

当前正在查看的 HTML 页面是活动页面，相应文档会以只读模式打开。换句话说，你可以查看这个文档，但不能对这个文档做任何修改。包含内嵌脚本的 HTML 文件的整个内容都会显示在页面内。如果脚本并非直接嵌在 HTML 中，也可以打开查看。有些脚本可能作为单独的.js 文件包含在 HTML 文件中，可以使用 Running Documents（运行文档）窗口来查看这些脚本，选择 View→Running Documents 菜单就可以打开这个窗口，见图 7-7。

打开当前文档后，现在可以开始调试了。选择你想设置断点的一行代码。把光标放在这一行上，再按 F9 或者点击菜单条上的 Toggle Breakpoint（切换断点）按钮。这一行就会加亮显示，在窗口左侧边栏上出现一个点。

在页面上完成导致脚本运行的一个动作，就会启动调试过程。当遇到所选的断点时，脚本会暂停执行。

脚本现在停止运行了，但是你能对它做什么呢？第一个简单任务是查看调用栈。点击菜单条上的 Call Stack（调用栈）窗口，可以查看嵌套函数调用的执行过程。双击调用栈列表中的各项，就会显示相应的函数调用。如果运行了多个线程，那么每个线程都有自己的调用栈。

Script Debugger 最有用的特性是能够在运行时查看和修改变量值。Command Window（命令窗口）是使用 Script Debugger 检查和修改变量值的入口。可以点击菜单条上的按钮打开命令窗口，也可以选择 View→Command Window 菜单项打开命令窗口。

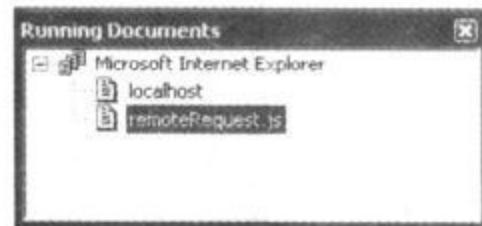


图 7-7 在 Running Documents 窗口中找到当前 Web 页面以及其中包含的所有脚本文件

命令窗口界面完全是基于文本的，没有提供可检查对象的列表，也没有在每个对象下面提供可折叠的属性列表。你必须知道想检查哪个变量，也就是要知道这个变量的名字，它可以是一个 JavaScript 对象或对象属性，或者是页面中出现的某一项的属性。要查看变量值，只需在命令窗口中键入变量名，再按回车键，此时变量的值就会显示在下一行上。

更新变量值的过程与查找变量值是类似的。要更新变量的值，需要在变量后面键入等号，并在其后提供该变量的新值。如果值是一个串，就必须用引号引起。还可以把值设置为其他对象，为此只需在等号后面键入对象变量名。

图 7-8 显示了这样一个例子，在此先查看变量值，然后对它进行修改。首先，我们在命令窗口中输入变量名 `queryString`，按回车键，这样会在下一行上显示 `queryString` 的值。接下来，输入下一行，为 `queryString` 赋一个新值，接回车键之后会回显这个新值。最后，再次检查 `queryString` 的值，确保确实正确地设置了新值。

```

function createQueryString() {
    var searchString = document.getElementById("searchString").value;
    searchString = escape(searchString);

    var maxResultsCount = document.getElementById("maxResultCount").value;

    var queryString = "query=" + searchString + "&results=" + maxResultsCount;
    return queryString;
}

function handleStateChange() {
    if(xmlHttp.readyState == 4) {
        if(xmlHttp.status == 200) {
            parseSearchResults();
        }
        else {
            alert("Error occurred");
        }
    }
}

```

Command Window

```

queryString
"query=xmIhttprequest&results=10"
queryString = "query=xmIhttprequest&results=14"
"query=xmIhttprequest&results=14"
queryString
"query=xmIhttprequest&results=14"

```

图 7-8 使用命令窗口检查和修改变量值

检查并（可能）修改了变量值之后，可以继续逐行地单步跟踪脚本。按下 F8 键或者点击 Step Into 工具条按钮，就能步进跟踪到下一行代码，如果下一行代码中调用了其他函数，这就会进入所调用的函数单步跟踪；按下 Shift+F8 键或者点击 Step Over 工具条按钮，也会执行下一行代码，但是不会进入到所调用的函数中。按下 F5 键或点击 Run 按钮，会继续运行余下的脚本，只有遇到下一个断点或者脚本结束时才会停止。

Microsoft Script Debugger 是一个很简单的工具，但对于调试客户端脚本很有用。因为命令窗口的界面相当原始（仅基于文本），所以 Script Debugger 可能更适合那些完全采用 IE 环境的人使用，甚至最好由使用 VBScript 而不是 JavaScript 的人使用。不过，开发人员还能得到另一个工具，它能提供更大的帮助。

### 7.2.3 使用 Venkman

基于 Mozilla 的浏览器（如 Firefox）有一个 JavaScript 调试环境，称为 Venkman。Venkman

是这些浏览器的一个扩展，可以从 [www.hacksrus.com/~ginda/venkman/](http://www.hacksrus.com/~ginda/venkman/) 得到并安装。Venkman 开发最早是由 Robert Ginda 于 2001 年 4 月开始的。Venkman 基于 Mozilla JavaScript 调试 API（称为 js/jsd）。js/jsd API 构成了 Netscape JavaScript Debugger 1.1 的基础，Netscape 浏览器 4.x 系统都提供了这个调试工具。

一旦完成安装，可以从 Firefox 主菜单条上选择 Tools→JavaScript Debugger 启动 Venkman。图 7-9 显示了 Venkman 的默认布局。

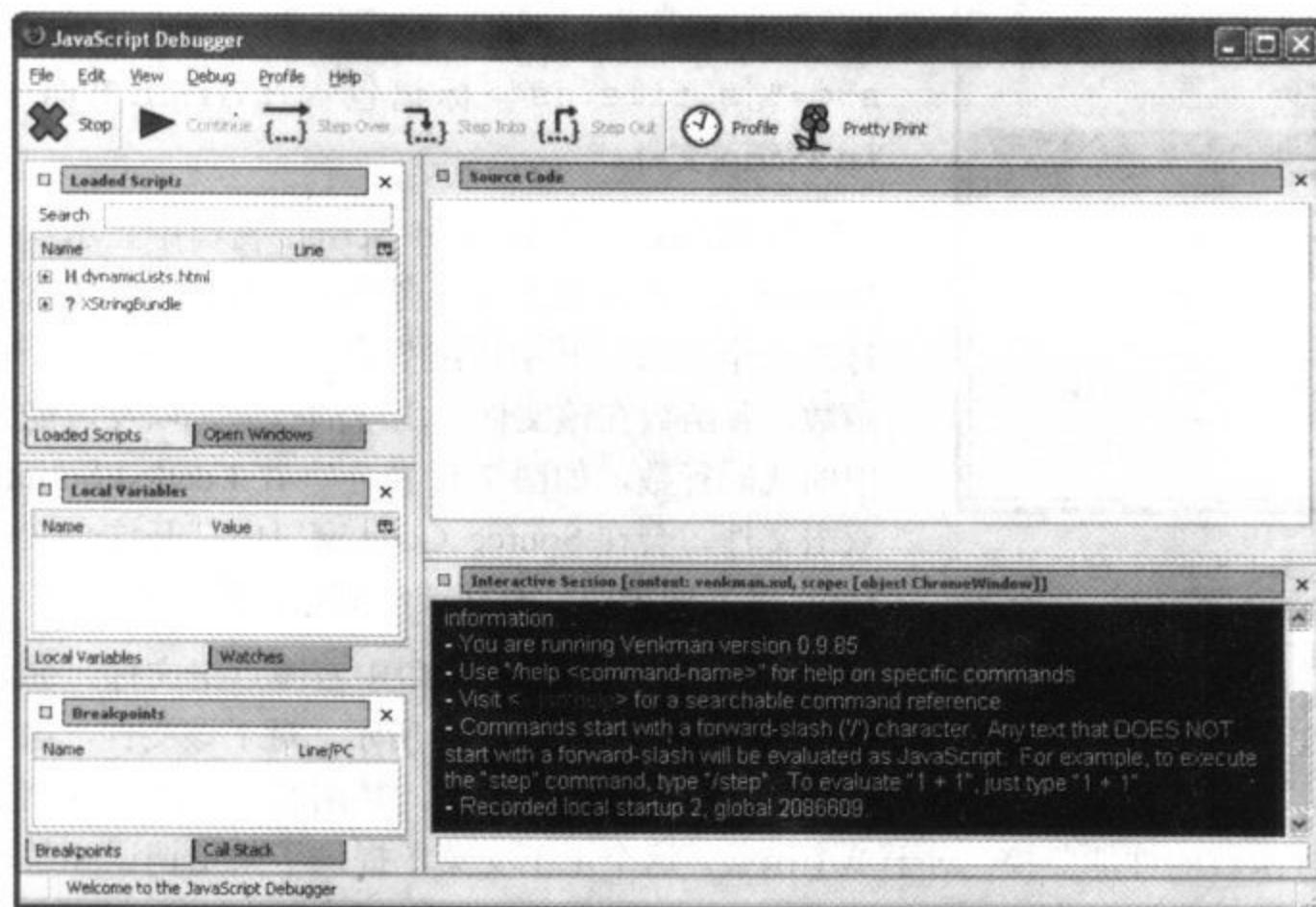


图 7-9 Venkman 的默认窗口布局

Venkman 提供了大量信息，这些信息划分到 8 个窗格。默认布局包括一个较大的窗格，其中显示所选的源代码，另外一些较小的窗格垂直放在该窗格的左边。Venkman 的命令行界面放在 Source Code（源代码）窗格的下面。

可以用鼠标拖动各个窗格，把它们放在主窗口中的其他位置。还可以把各个窗格作为单独的标签增加到一个现有窗格中。例如，要让 Loaded Scripts（已加载脚本）标签成为 Local Variables（局部变量）窗格中的标签，只需把 Loaded Scripts 标签拖放到 Local Variables 标签中就可以。点击窗格标题栏左侧的 docking（停靠）按钮，还可以取消小窗格的停靠，如图 7-10 所示。把窗格放回主窗口很容易，只需再点击一次 docking（停靠）按钮就行了。

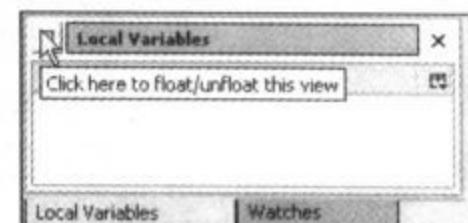


图 7-10 使用 docking 按钮自由放置各个窗格

使用 Venkman 时与你最常使用的窗口没有两样。要关闭不常用的窗口，可以点击窗口标题栏右侧的×按钮。选择 View→Show/Hide 菜单项就能重新打开窗口。如果某个时刻你想把窗口布局还原为默认设置，只需在命令行界面使用 /restore-layout factory 选项。

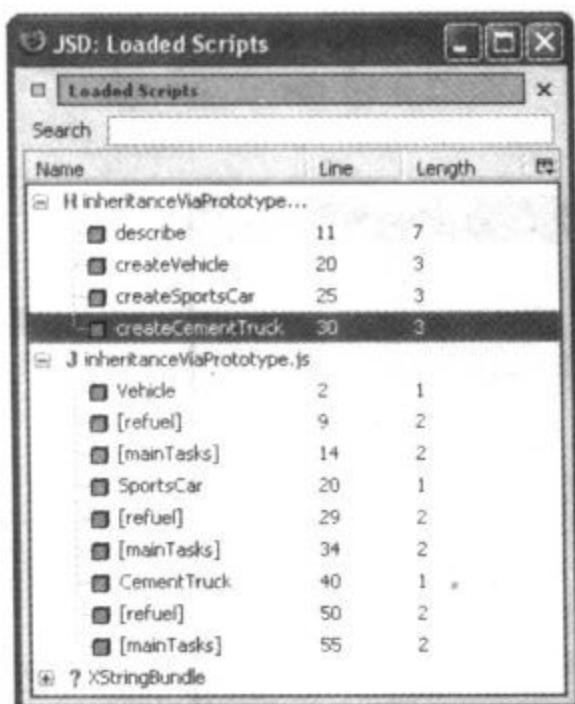


图 7-11 Loaded Scripts 窗口显示当前浏览器窗口中的所有可用脚本

本身以及文件所包含 JavaScript 函数的许多选项，如图 7-12 所示。对于该文件，这个弹出菜单允许你完成以下任务，如禁用 eval 和 timeout 语句的调试，禁用所包含函数的调试，以及禁用所包含函数的性能监控。对于单个函数，这个上下文菜单提供了禁用调试和禁用性能测评分析等功能。

### 查看已加载的脚本

当 Venkman 打开时，它会识别浏览器窗口中当前页面可用的所有 JavaScript。Venkman 能识别使用<script>标记嵌在 HTML 页面中的 JavaScript，还能识别使用<script src="js\_file.js">标记包含在 HTML 页面中的外部 JavaScript 文件。

Venkman 会在 Loaded Scripts 窗口中显示当前可用的 JavaScript。点击每个文件旁边的加号，就会在这个文件下打开一个列表，其中详细列出该文件中的所有 JavaScript 函数，各函数在该文件中出现的行号，以及可能列出函数中的代码行数，如图 7-11 所示。在 Loaded Scripts 窗口中双击文件，会在 Source Code 窗口中打开这个文件，而且 Source Code 窗口会直接滚动到该函数所在位置。

在 Loaded Scripts 窗口中右键点击文件，会显示文件

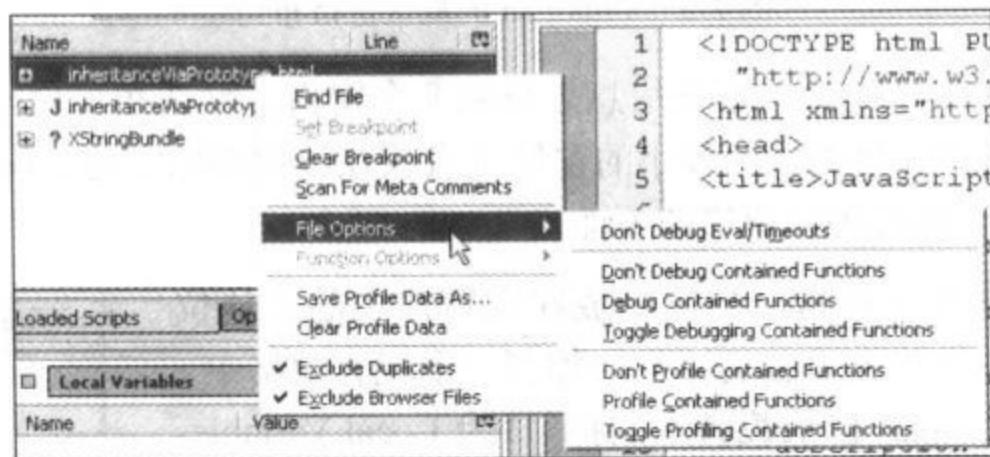


图 7-12 包含 JavaScript 的文件的上下文菜单项

### 源代码

Source Code（源代码）窗口列出了当前打开文件的源代码，文件可以是 HTML 文件、XHTML 文件或 JavaScript 文件。Source Code 窗口采用了一种分页样式，这样就能同时打开

多个文件，每个文件分别放在自己的页中。代码做了彩色处理，以增加可读性。JavaScript关键字（如 `function` 和 `var`）采用粗体，串直接量采用不同颜色的字体。窗口左侧是文件源代码的行号。Source Code 窗口最左侧有一个边栏，可以在这里设置调试断点。

### 断点，第1部分

有了断点（Breakpoint）特性，调试工具才真正有用。断点允许在所选位置暂停代码的执行，这样就有机会检查各个变量和属性，来确定 bug 的根本原因。Venkman 支持两种断点：硬（hard）断点和将来（future）断点。这与大多数调试环境有所不同，所以我们将讨论这两种断点之间的区别。

硬断点就是使用当前编程语言（如 Java）时经常看到的那种断点，它指示 Venkman 在断点处暂停处理，在用户指示能继续之前不能继续执行。在 Venkman 中，硬断点总是放在函数体中。

将来断点与硬断点有一点很相似，它也指示 Venkman 在断点处暂停 JavaScript 的执行。二者的区别是，将来断点设置在函数体之外的代码行上。一旦这些代码行加载到浏览器上就会立即执行。与之相反，只有当为响应某个动作或事件执行了函数时，位于函数体中的代码才会执行。

图 7-13 显示了加载 JavaScript 代码的示例页面，其中的 JavaScript 代码有两个断点。内嵌的 JavaScript 代码有一个包含代码的函数，以及位于函数体之外的代码。函数中的代码有一个硬断点，函数外的代码有一个将来断点。

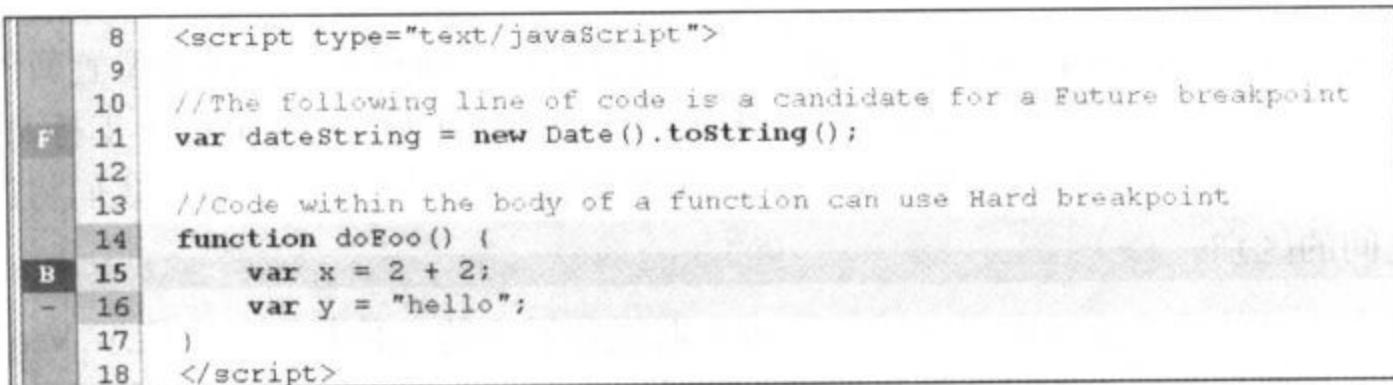
```
8 <script type="text/javascript">
9
10 //The following line of code is a candidate for a Future breakpoint
11 var dateString = new Date().toString();
12
13 //Code within the body of a function can use Hard breakpoint
14 function doFoo() {
15     var x = 2 + 2;
16     var y = "hello";
17 }
18 </script>
```

图 7-13 Source Code 窗口显示了 `breakpoints.html` 的源代码，可以看到其中的硬断点和将来断点

要在所需的代码行上设置一个断点，点击指定代码行左侧的边栏即可。每次点击这一行时，这一行就会轮流地切换为以下 3 种断点设置：无断点、硬断点和将来断点。硬断点由一个红色的 B 指示，将来断点由橙色的 F 指示。位于函数体之外的代码行只可能切换为无断点或将来断点两种设置；函数体中的代码会在无断点、硬断点和将来断点这三种设置中轮流切换。

设置了断点后，可以启动一个可能导致遇到断点的动作。`breakpoints.html` 页面有一

个按钮。点击这个按钮会调用 doFoo 方法，这个方法只是创建两个变量。在浏览器窗口中点击 doFoo 按钮时，就会调用 doFoo 方法，然后遇到第 15 行的断点。当遇到这个断点时，JavaScript 的执行会暂停，直到得到通知可以继续。Venkman 窗口显示在浏览器窗口的前面，断点加亮显示，如图 7-14 所示。



```

8 <script type="text/javascript">
9
10 //The following line of code is a candidate for a Future breakpoint
11 var dateString = new Date().toString();
12
13 //Code within the body of a function can use Hard breakpoint
14 function doFoo() {
15     var x = 2 + 2;
16     var y = "hello";
17 }
18 </script>

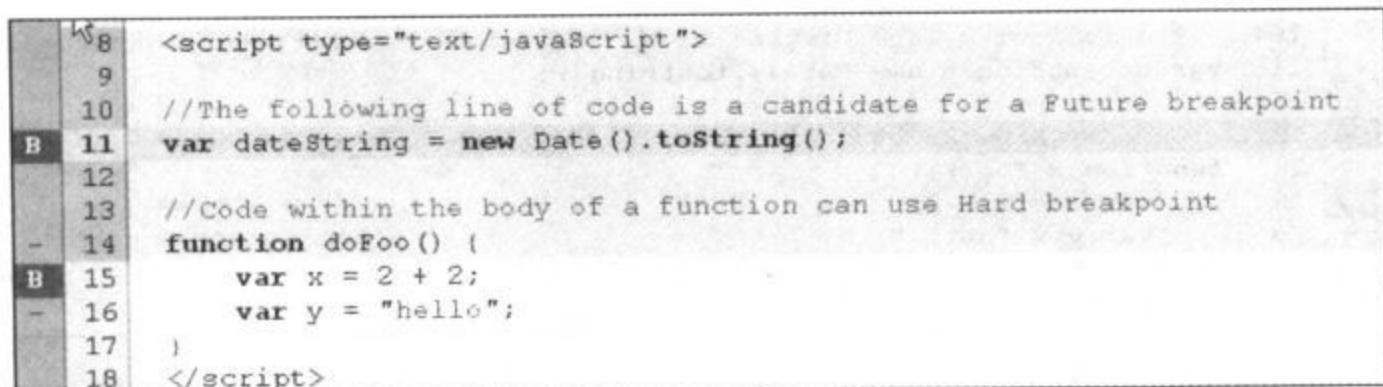
```

图 7-14 Venkman Source Code 窗口中遇到的断点会加亮显示

既然执行已经暂停，就可以自由地完成许多任务，如检查变量值，查看调用栈，从而确定执行的确切顺序。

到目前为止，硬断点的工作与你预期的完全一样。那么，将来断点呢？你怎么在将来断点处暂停执行呢？要知道，你并不能控制导致执行将来断点的动作或事件。

应该记得，函数体之外的 JavaScript 代码会在页面加载时立即执行。所以，要想遇到第 11 行的断点，需要切换到浏览器窗口，点击 Reload（重新加载）按钮。这样就会重新加载页面，并暂停第 11 行代码的执行。当遇到将来断点时，橙色的 F 图标会变成红色的 B 图标，如图 7-15 所示。



```

8 <script type="text/javascript">
9
10 //The following line of code is a candidate for a Future breakpoint
11 var dateString = new Date().toString();
12
13 //Code within the body of a function can use Hard breakpoint
14 function doFoo() {
15     var x = 2 + 2;
16     var y = "hello";
17 }
18 </script>

```

图 7-15 重新加载一个设置了将来断点的页面，这会导致遇到将来断点，并暂停脚本的执行

Venkman 提供了一个断点窗口，可以列出当前设置的所有断点。如果要调试在多个文件中设置了多个断点的页面，这个窗口就能提供很大方便。例如，你可能要调试一个 HTML 页面，其中包含特别为这个页面编写的内嵌 JavaScript；一个外部 JavaScript 文件，外部 JavaScript 文件是你所在组织维护的一个 JavaScript 函数库；以及由第三方开发商或开源项目提供的一个外部 JavaScript 文件。每个设置了断点的文件都会列在 Breakpoints（断点）窗口

中，在每个文件下面还列出了该文件中的所有断点，如图 7-16 所示。

### 单步跟踪代码

设置了断点后，现在可以具体调试代码了。一旦遇到断点，Venkman 会自动暂停执行，此时要由你来控制脚本的执行。可以检查变量值，修改变量值，继续脚本执行（可以一次只跟踪一步，也可以重新开始执行，并让代码一直运行到结束）。

一旦遇到断点，Venkman 会为开发人员提供许多单步跟踪的选择。遇到断点后，可以选择 Continue（继续）、Step Over（跳步跟踪）、Step Into（步进跟踪）或 Step Out（步出跟踪），如图 7-17 所示。

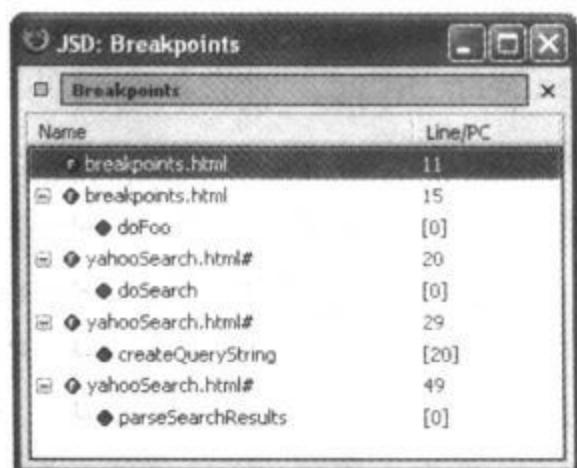


图 7-16 Venkman 的 Breakpoints 窗口按文件列出所有断点

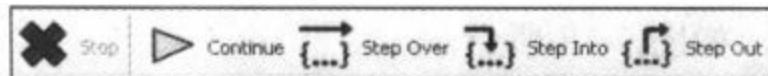


图 7-17 Venkman 的工具条提供了很多单步跟踪脚本的选择

Continue（继续）选项重新开始执行脚本。在遇到下一个断点或者脚本结束之前，执行不会停止。如果要跟踪出现问题的位置，Continue 就很有用。可以沿着执行链设置断点，每次遇到一个断点时，检查变量值，查看问题是否已经出现。一旦出现了问题，你就知道错误肯定出现在当前断点和前一个断点之间的某个地方，接下来可以在这之间进一步缩小范围。如果你在调试一个循环，Continue 也很有用。可以在循环中的某一点设置断点，使用 Continue 选项来加快循环代码的执行（而不是一次只执行一个代码行），检查每次执行暂停时是否出现问题。

如果你不想单步跟踪当前函数中调用的函数，Step Over 就很有用。所调用的函数可能已经得到充分的调试，而且你很清楚问题并不出现在那里，或者你可能就是想跳过这个函数的单步跟踪，因为你只关心当前函数。要记住，跳步跟踪一个函数并不意味着不执行这个函数，这只是意味着你不会逐行地跟踪这个函数的执行。相反，这个函数会像一条语句一样执行。

Step Into 与 Step Over 刚好相反。Step Into 会步进跟踪所调用的函数，从而能调试这个函数。如果你想跟踪到错误的准确位置，Step Over 和 Step Into 结合使用会更好。考虑这样一种情况，你要调试函数 sendAjaxRequest。这个函数本身包含多行逻辑，另外 sendAjaxRequest 又调用了 createXmlHttp。第一次单步跟踪 sendAjaxRequest 时，可以仔细地跟踪每一行，监视什么时候最早暴露错误。当遇到对应 createXmlHttp 的断点时，使用 Step Over 快速执行 createXmlHttp，而不是逐行地单步跟踪。等跳步跟踪 createXmlHttp 后，再查看是否已经出现错误。如果没有，可以知道错误不在 createXmlHttp 中；如果有错误，

说明错误就在 `createXmlHttp` 中。下一次调试 `sendAjaxRequest` 时，就应该步进跟踪 `createXmlHttp`，对它做细致的调试。

可以把 **Step Out** 看作是 **Step Into** 的“补救”。**Step Out** 允许你退出当前函数的调试，返回到调用栈中的前一个函数。如果你目前正在单步跟踪函数 `createXmlHttp`，它由 `sendAjaxRequest` 调用，则使用 **Step Out** 就会返回到 `sendAjaxRequest` 中调用 `createXmlHttp` 的下一行代码。如果你正处在一个函数中，但是不想完整地单步跟踪这个函数，而是想返回到调用栈中的前一个函数，此时 **Step Out** 就非常有用。

你现在可能注意到 Venkman 的工具条上还有一个 **Stop** 按钮，这个按钮是做什么的呢？点击 **Stop** 按钮，会激活 Venkman 的另一个功能，即下一次执行任何 JavaScript 时都会立即暂停 JavaScript。一旦点击 **Stop** 按钮，可以相信，下一次执行任何 JavaScript 时，Venkman 就会暂停其执行。使用 `setTimeout` 或 `setInterval` 自动运行的脚本通常很难调试，因为脚本的入口点不好确定。通过激活 **Stop** 函数，Venkman 就能捕获到脚本的入口点，如果需要，可以从这里开始调试，如图 7-18 所示。



图 7-18 Venkman 的 Stop 按钮置无效（左）和激活（右）

## 断点，第2部分

Venkman 还有一个强大的特性，可以编写定制代码，并在每次执行断点时运行这些定制代码。在 **Breakpoints** 窗口中，右键点击一个断点，选择 **Breakpoint Properties**（断点属性）菜单项，如图 7-19 所示。

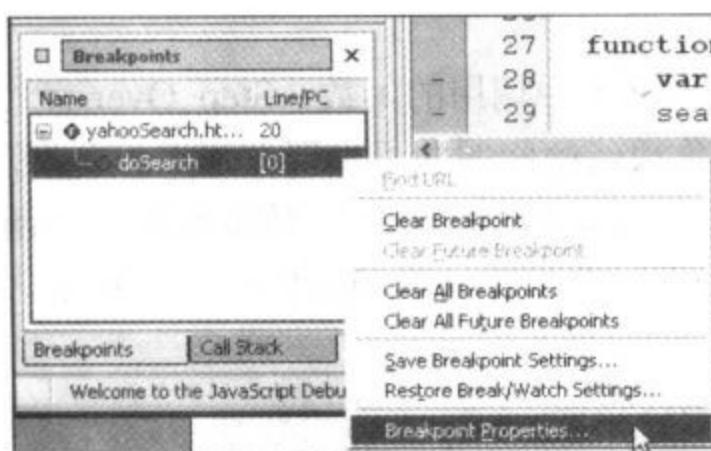


图 7-19 从 Breakpoints 窗口的上下文菜单选择 Breakpoint Properties 菜单项

这样会打开 **Breakpoint Properties** 对话框，这是一个强大的工具，允许你修改断点的行为，即定制断点的行为来满足你自己的需要。

在这个窗口最上面，有两个复选框：**Enable Breakpoint**（启用断点）和 **Clear Breakpoint After First Trigger**（首次触发后清除断点）。这些复选框的行为不言自明。**Breakpoints Properties**

窗口的真正强大之处在于“**When Triggered, Execute**”（如果触发，则执行……）复选框，如图 7-20 所示。

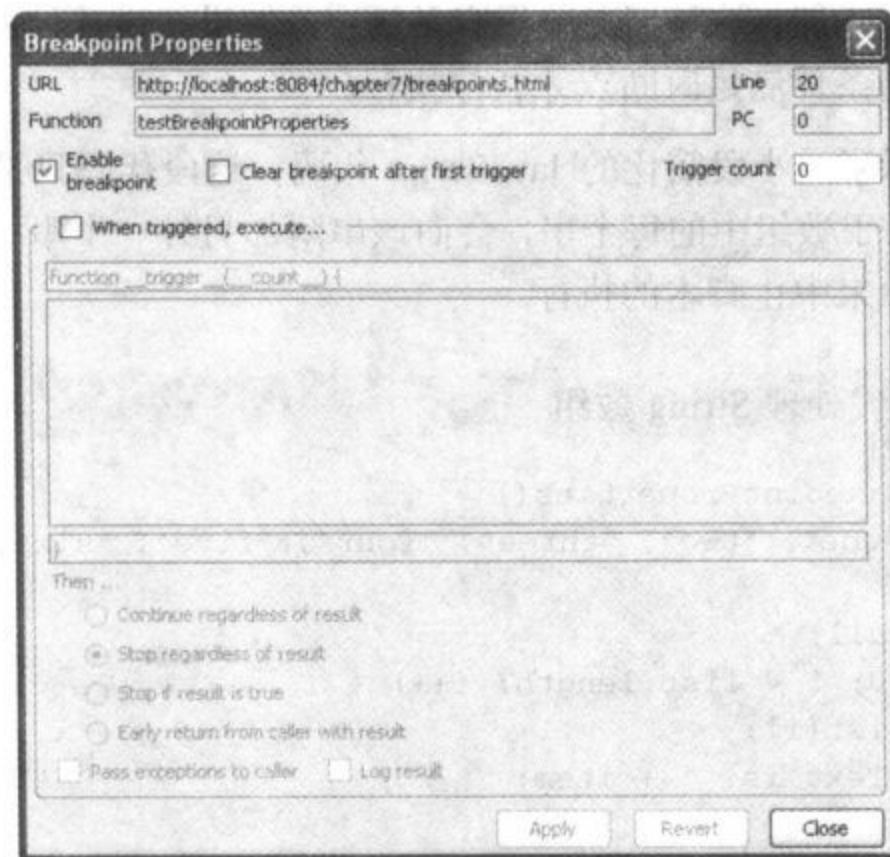


图 7-20 Breakpoint Properties 窗口的默认外观

选中“**When Triggered, Execute**”复选框，会置一个文本域有效。可以在这个文本域中编写 JavaScript 代码，每次遇到断点时都会执行此代码。向这个定制脚本传递的参数名为 `_count_`，它表示遇到断点的次数。

在文本域下面，可以指定每次遇到断点时该采取怎样的行为。具体行为可以根据文本域中定制代码的返回值决定。断点的行为有以下选择：

- Continue Regardless of Result（不论结果如何都继续）
- Stop Regardless of Result（不论结果如何都停止）
- Stop If Result Is True（如果结果为 true 则停止）
- Early Return from Caller with Result（带着结果从调用函数提前返回）

这些选项大多都不言自明。功能最强大的可能是“**Stop If Result Is True**”选项。选择这个选项的话，意味着只有当定制代码的返回值为 true 时断点才会暂停执行。

想像一下，如果能够根据给定条件暂停执行，这会有多方便。考虑以下情况，假设你在迭代处理一个很大的对象列表，并对每个对象完成某种计算。在开发过程中，你注意到，计算时在一个特定对象处失败了。如果能独立出数据源，只处理单个的对象，或者只是一个很小的对象列表，则没有任何问题。但是，如果只有当使用一个很大的对象列表时错误才可能

再次出现，在这种情况下，倘若失败的对象刚好在列表的最后，就必须设置一个断点，每次单步跟踪，查找会出现错误的那个特定对象。

如果使用条件断点功能，这种情况就很容易解决。如果知道某种情况会失败，可以编写一个条件语句，只有当条件满足时断点才暂停执行。

代码清单 7-1 显示了一小段简化的 JavaScript 代码，这段代码只是创建一个串数组，并迭代处理这个数组，对于数组中的每个串，会向该串附加另外一个串。这段代码显示了如何根据你定义的一个条件来中止脚本的执行。

### 代码清单 7-1 迭代处理 String 数组

```
function testBreakpointProperties() {
    var list = ["one", "two", "three", "four", "five", "six", "seven", "eight"];

    var item = null;
    for(var i = 0; i < list.length; i++) {
        item = list[i];
        item = "Text is: " + item;
    }
}
```

对于这个例子，你可能希望只对列表中第 7 项暂停执行，即 `seven`。如果只是在循环中设置断点，那么每次遇到这行代码时都会暂停执行，而你每次都必须手工地重新开始执行，直至到达列表中你感兴趣的那一项。在这个例子中，这个列表不算长，但是如果列表中有数百项，只在满足特定条件时才中止执行就会很方便。

要想只对列表中第 7 项中止执行，首先在所需的行上设置一个断点，然后右键点击这一行上的任意位置，选择 `Breakpoint Properties`，打开 `Breakpoint Properties` 对话框。

假设有一个简单的场景：你希望只在处理列表中第 7 项（也就是串 `seven`）时断点才真正暂停执行。选中“Triggered, Execute”复选框，启用条件式断点。在所提供的方法体中输入一行代码：

```
return item == "seven";
```

这样一来，只有处理列表中第 7 项时这个方法才会返回 `true`。然后选中“Stop If Result Is True”单选钮，指示 Venkman 只有当处理的项是串 `seven` 时才暂停代码的执行，这样就大功告成了。图 7-21 显示了设置好的 `Breakpoint Properties` 对话框。

图 7-21 中所示的函数执行时设置了一个条件式断点，这个断点只在第 7 次迭代时才暂停执行，也就是 `item` 变量处于串 `seven` 时才中止执行。

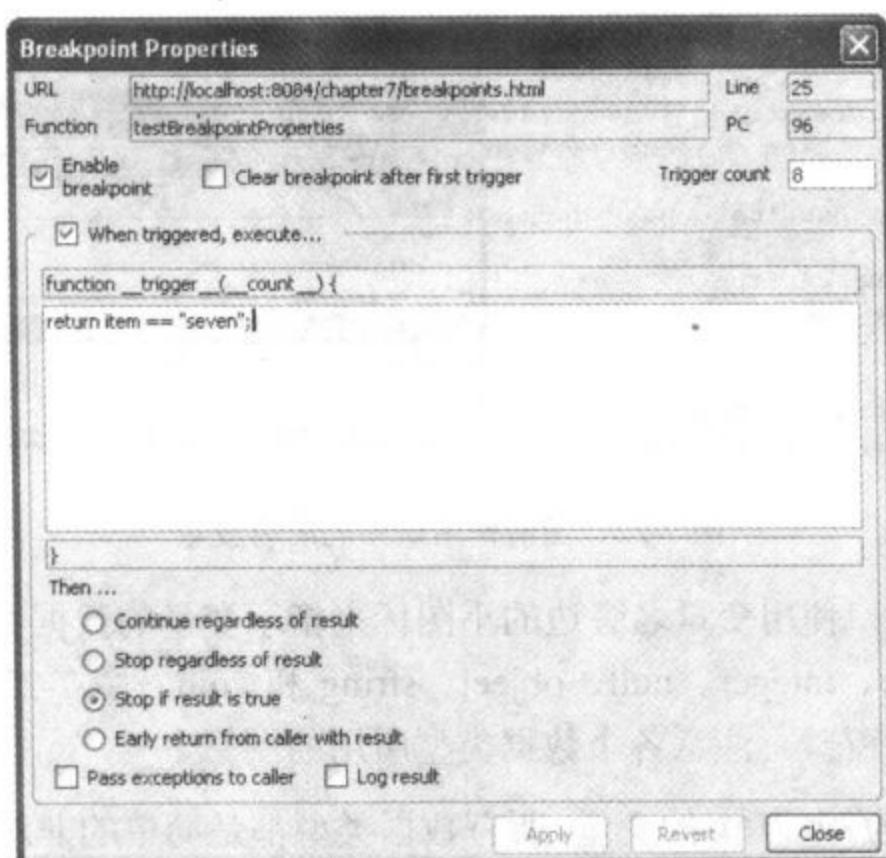


图 7-21 配置一个断点，从而只基于一个条件暂停执行

这是一个功能很强大的技术，在很多情况都很方便。除了前面的例子外，你可能希望每隔一次遇到断点时暂停执行。要实现这个功能，需要使用取模（%）操作符确定`_count_`参数是奇数还是偶数，而且要设置“Stop If Result Is True”选项。再举一个使用“Continue Regardless of Result”设置的例子，比如每次遇到断点时都执行某种日志记录功能。甚至可以使用 Ajax 技术向服务器发送日志信息！

### 局部变量表

**Local Variables**（局部变量）窗口允许你在脚本执行期间查看甚至修改变量值。无论遇到断点还是暂停了脚本的执行，**Local Variables** 窗口总会显示作用域中的所有变量。

**Local Variables** 窗口总是有两个顶层项：`scope` 和 `this`。`scope` 指示当前作用域中的所有变量。由于大多数 JavaScript 代码都写为函数，所以当前作用域通常就是函数作用域。例如，如果遇到一个函数中的断点，那么 **Local Variables** 窗口中的 `scope` 项就会指示该函数作用域中的所有变量，具体说来，就是在该函数中用关键字 `var` 定义的所有变量。全局作用域中定义（即在函数体之外定义）的变量从技术上讲可以在函数中访问，但是不会显示在当前变量作用域中（见图 7-22）。

**Local Variables** 窗口中第二个顶层项是 `this`。`this` 指示关键字 `this` 引用的任何对象。如果遇到函数中的一个断点，而且这个函数是对象的一部分，`this` 引用的就是当前对象实例。`this` 一般会引用浏览器的 `window` 对象。注意，全局作用域中定义的所有变量都会出现在 `this` 项下面。

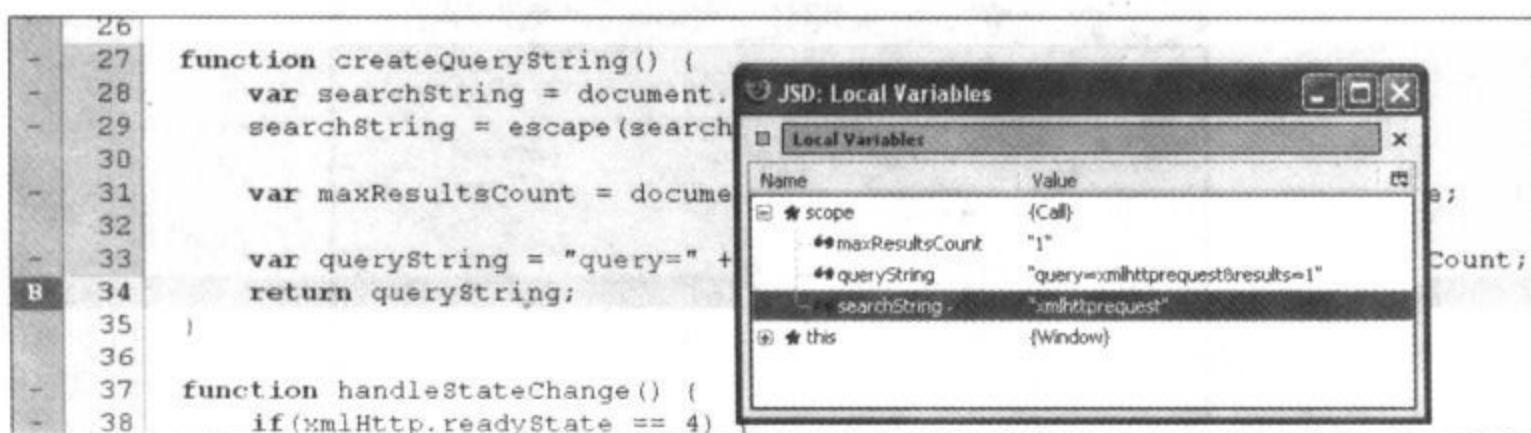


图 7-22 当前作用域中的所有变量

Local Variables 窗口使用变量名旁边的小图标来指示变量的数据类型。可用的数据类型包括: boolean、double、integer、null、object、string 和 void。图 7-23 显示了包括各种数据类型的 Local Variables 窗口。注意各个数据类型所用的图标。

除了图 7-23 中名为 objVar 的变量,所有数据类型都是简单的非对象数据类型,在 Local Variables 窗口的 Value 列中可以很容易地看到它们的值。注意 objVar 变量旁边有一个加号,这说明可以打开这个对象查看它的属性。图 7-24 显示了使用 object 变量类型的 Local Variables 窗口。注意在这个例子中,断点设置在对象的成员函数中,这说明 Local Variables 窗口中 this 项引用的是对象实例本身,而 scope 项没有任何子项。

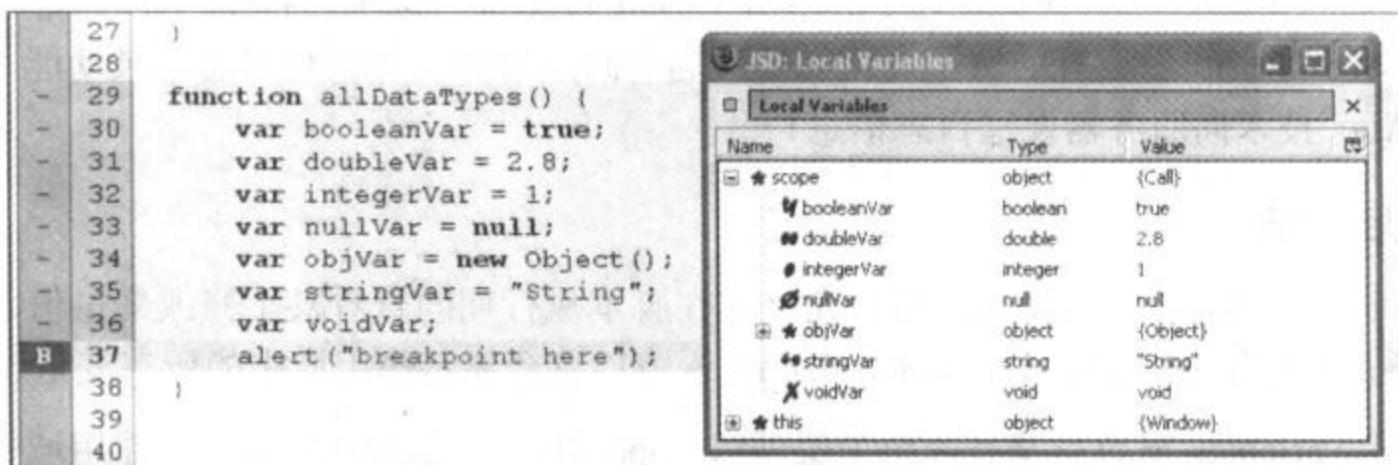


图 7-23 Local Variables 窗口显示了每种可用的数据类型

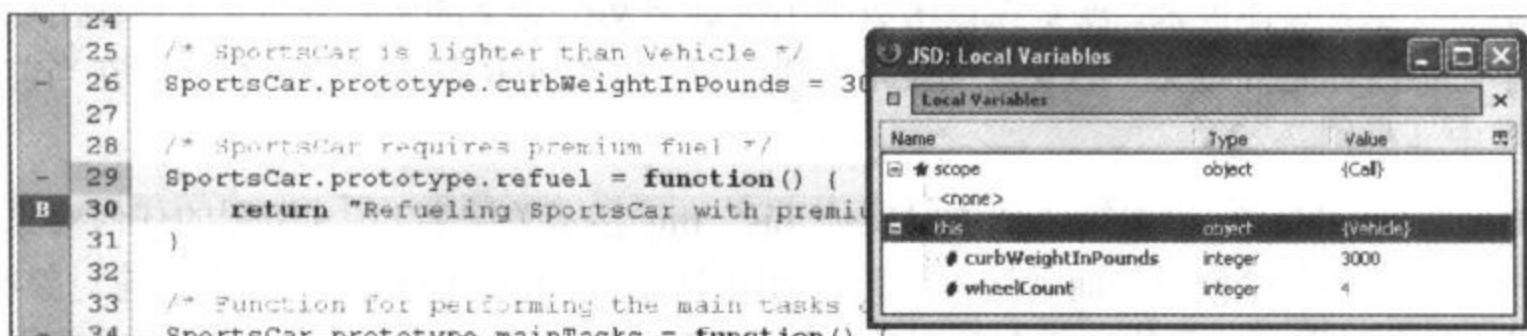


图 7-24 在对象的成员函数中暂停执行, Local Variable 窗口显示了对象的属性

在这里，这个对象有两个属性：curbWeightInPounds 和 wheelCount，它们各自的值分别显示在 Local Variables 窗口中。

在此之前，你可能使用过警告框以便在运行时显示变量的值。这种技术当然是可以的，但是你要花时间在脚本中到处“洒上”警告，而且每个警告通常一次只显示一个变量值。脚本调试后，则需要删除所有这些警告框，如果你想再次调试，只能把这些警告框再加上。利用 Venkman，你只需设置一个断点，并使用 Local Variables 窗口来查看变量的值，而且你能很快地得到所有变量的值。

回想一下，正是有了 Local Variables 窗口，才使 Venkman 成为一个极其强大的调试工具。只需快速一瞥就能查看 JavaScript 引擎认识的所有对象和变量的值。通过使用断点，你可以有选择地暂停脚本的执行，并检查各个对象和变量，确保它们的值是对的。由于你已经使用了 TDD 技术（你肯定用了，对不对？），知道脚本的期望输出应当是什么。如果你的测试失败了，可以使用 Venkman 单步跟踪脚本，结合使用断点和 Local Variables 窗口，就能很快地诊断出错误的根源。

Local Variables 窗口不仅是功能强大的调试工具，它还是一个很不错的学习工具。利用这个窗口，你可以检查任何对象的属性或 JavaScript 解释器可用的变量。请考虑 XMLHttpRequest 对象，这方面的文档很多。你知道它有一些可以公共访问的属性，如 responseText、responseXML 和 status。但是，如果你不知道怎么办？有没有办法来查呢？图 7-25 显示了一个 XMLHttpRequest 对象的两个视图。左边的图显示了刚创建的对象，还没有设置任何属性。右边的图显示了做出一个成功请求后的对象属性。

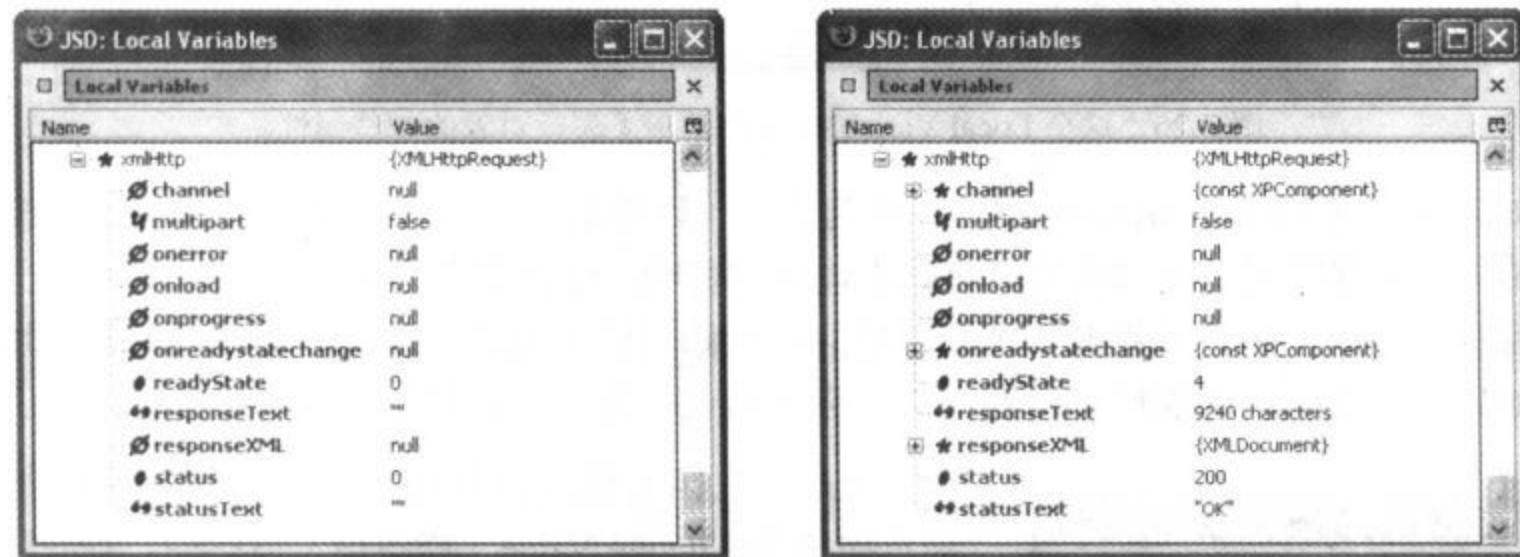


图 7-25 刚创建的 XMLHttpRequest 对象（左）和做出成功请求之后的 XMLHttpRequest 对象（右）

对于 XMLHttpRequest 对象，这样是可以的，因为 XMLHttpRequest 对象有充分的文档。如果是一个没有完备文档的对象或者文档很难得到，会怎么样呢？

请考虑一个表行 DOM 对象。我们曾经提到过，所有 DOM 元素都有诸如 firstChild 的属

性，但是除此以外还有其他的属性吗？可以使用 Venkman 和 Local Variables 窗口看一看。图 7-26 显示了 Local Variables 窗口，其中列出了表行对象的一些可用属性。你知道它还有 offsetHeight、offsetLeft、parentNode 和 previousSibling 属性吗？如果知道，那祝贺你。如果以前不知道，起码现在应该了解了，而且这个知识没准以后能派上大用场。你可能不了解所有这些属性的含义，不过既然已经知道存在这些属性，也知道它们的名字，就可以到网上查一查，了解有关的更多信息。

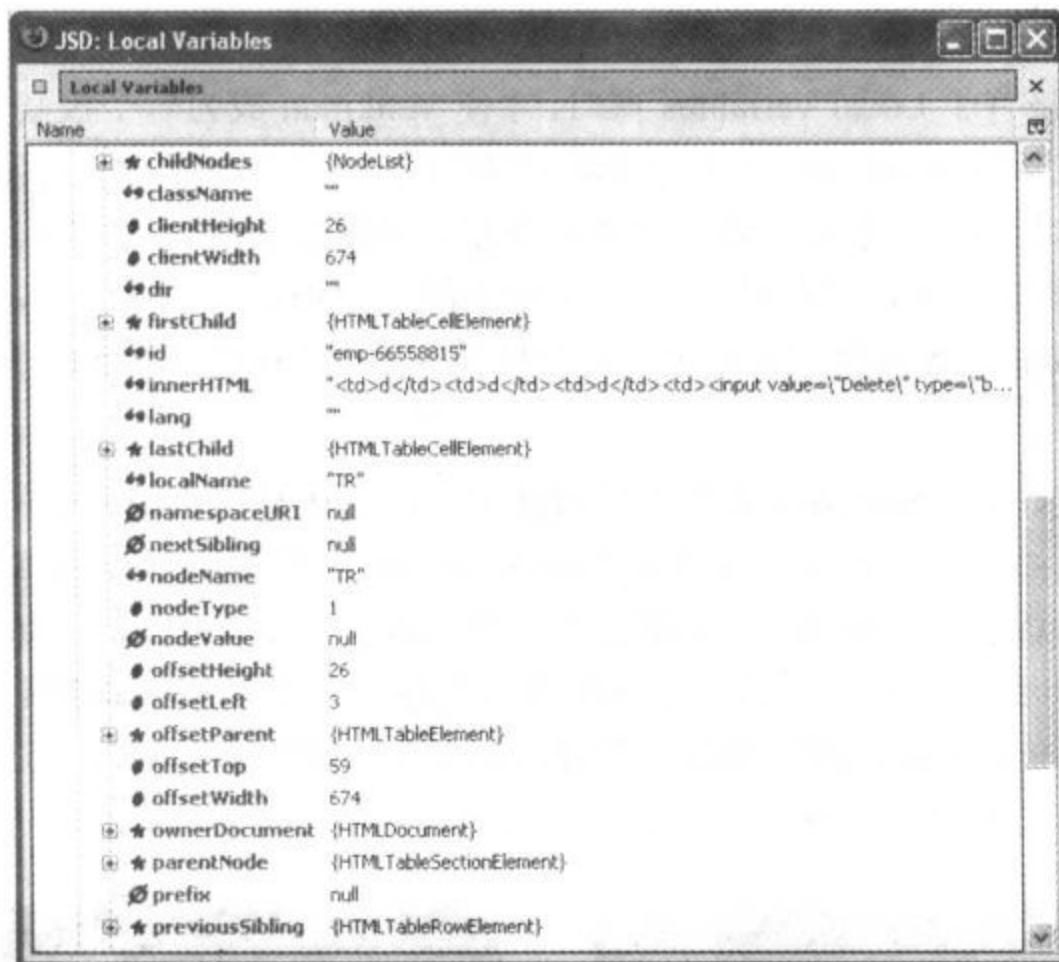


图 7-26 这个 Local Variables 窗口显示了表行对象的一些属性

如果你还对 Local Variables 窗口不以为然，可以考虑一下：它还允许你在运行时修改变量的值。如果你想测试不同变量值对脚本输出的影响，这就非常有意义。如果你认为已经发现了哪里有问题，想查看修改一个变量值能不能修正这个问题，Local Variables 窗口也很有用。如果修改变量值就能修正问题，那么你只需查看为什么最初变量值是错的。

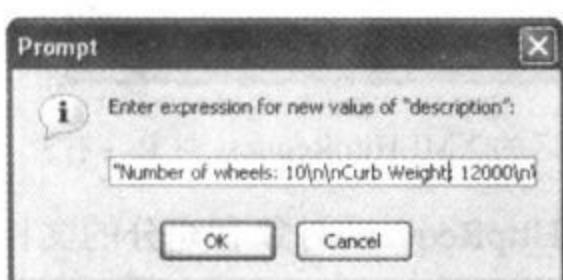


图 7-27 使用 Change Value 提示窗口修改一个变量值

为此，只需右键点击你想修改的变量值，从上下文菜单中选择 Change Value（修改值）。这会打开一个小的提示窗口，可以在其中修改变量的值，如图 7-27 所示。在提示窗口中可以输入任何合法的 JavaScript 表达式，包括诸如 new Object() 等表达式。要保证串直接量一定要用双引号或单引号引起来。记住，在提示窗口中还可以使用变量名来引用其他变量。

## 监视列表

Watches list (监视列表) 与 Local Variables 窗口几乎是一样的，它也显示运行在当前作用域中变量的有关信息。Watches list 和 Local Variables 窗口之间的区别在于，开发人员可以决定监视列表中显示哪些变量。与此不同，Local Variables 窗口会显示当前执行脚本可用的所有变量。可以把监视列表认为是缩水后的局部变量列表。

要向监视列表增加一个变量，可以在 Watches 窗口的任意位置右键点击，从上下文菜单中选择 Add Watches Expression (增加监视表达式)，这时会出现一个提示窗口，允许你在其中输入一个合法的 JavaScript 表达式来标识要监视的变量。在大多数情况下，你会输入一个变量名。图 7-28 显示了上下文菜单和结果窗口。

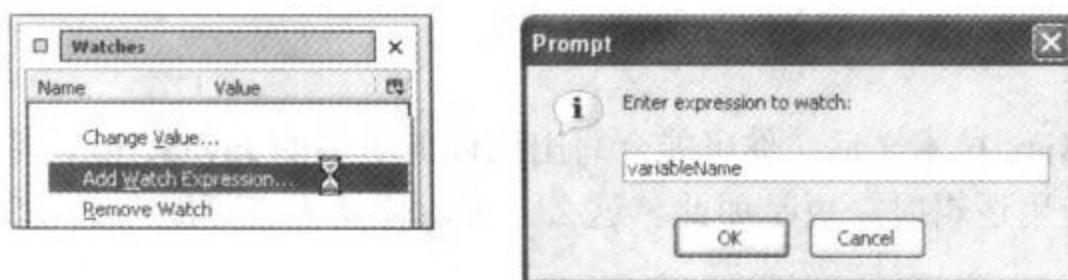


图 7-28 右键点击 Watches 窗口（左），然后输入想监视的变量的名字（右）

记住，Watches 窗口会显示你想监视的所有变量。不过，有时你想监视的变量不在脚本的当前作用域中。例如，如果设置要监视 doThis 函数中的变量 foo，而执行在函数 doThat 中的一个断点处暂停，doThis 中的变量 foo 就不在作用域中。如果是这样，Venkman 只会报告变量值为 {Error}。不要担心，这不会带来任何问题，这只是 Venkman 的一种对策，想让你知道这个变量不在当前作用域中。

在许多情况下，你可能想更多地使用 Watches 窗口而不是 Local Variables 窗口，这是因为你能确定在窗口里显示什么。因此能得到一个简洁得多的窗口，而且不用花那么多时间来查找你真正想跟踪的几个变量。

图 7-29 显示了 3 个变量。变量 url 定义在另一个函数中，所以它的值报告为 {Error}。变量 xmlhttp 是全局变量，所以可以显示当前值。变量 queryString 定义在当前函数中，目前就是在这个函数中暂停了脚本的执行。

## 调用栈

Venkman 会在 Call Stack (调用栈) 窗口跟踪当前调用栈 (见图 7-30)。这个窗口只显示

JSD: Watches	
Name	Value
★ url	{Error}
★ xmlhttp	{XMLHttpRequest}
★ channel	{const XPCOMPONENT}
multip...	false
onerror	null
onload	null
onpro...	null
onrea...	null
ready...	4
respo...	<?xml version='1.0'?><Result>
★ respo...	{XMLDocument}
status	200
status...	"OK"
queryString	"query=xmlhttprequest&results=1"

图 7-29 Watches 窗口显示了 3 个变量

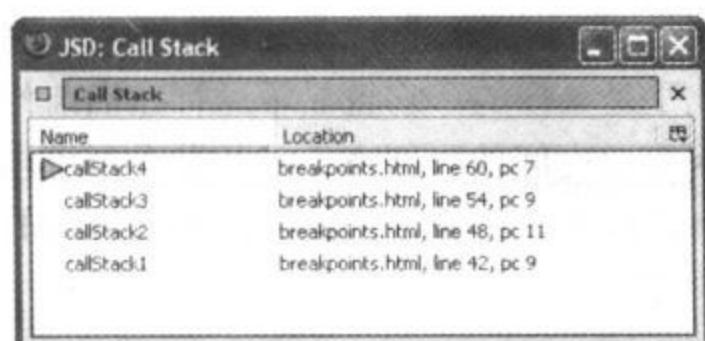


图 7-30 Venkman 的 Call Stack 窗口

一个栈，这是一个函数调用列表。栈顶的函数就是当前函数，列表中的下一个函数就是栈顶函数调用了的函数，以此类推。如果你有一个复杂的调用结构，而且想了解函数调用的顺序，这个窗口就非常有用。

双击 Call Stack 窗口中的一项，Source Code 窗口中就会显示调用栈中所选项相应函数的源代码。Local Variables 列表会自行更新，以显示该函数中的变量。注意，这并不会真正改变执行的顺序，你只是“旧地重游”，但这并不会改变你将来要去哪里。

### 性能测评分析

越来越熟悉 Ajax 技术之后，你可能会写出比以前更多的 JavaScript，特别是当你决定由浏览器执行更多应用逻辑时，更是如此。这么一来，你就更有可能在 JavaScript 代码中遭遇性能瓶颈。

合理软件开发实践指出，不要因为性能优化代码，除非确实已经存在性能问题。比起未优化的代码来说，优化代码几乎总是更难读，也更难维护，所以除非你真的需要，否则不要为性能优化代码。

编写 JavaScript 也是一样。在大多数情况下，一般都没必要因为性能原因优化代码。不过，有时可能确实需要做一些性能调整。但在优化代码之前，首先需要确定哪些代码需要优化。Venkman 有一个内置的性能测评分析工具，在这方面会很有帮助。

Venkman 中的性能测评分析工具会自动监视脚本，并跟踪每个函数的执行时间，另外还会记录每个函数的调用次数。性能测评分析工具将其数据导出为 HTML、XML、CSV 和文本格式。

要启用性能测评分析工具，只需点击 Venkman 工具条上的 Profile 按钮（见图 7-31）。

一旦激活，Profile 按钮中就会有一个小的绿色复选框。在此之后，性能测评分析工具将记录当前浏览器窗口中运行的所有 JavaScript 的性能度量。再点击这个按钮就会停止收集性能数据。

一旦收集到一些性能数据，就可以看看哪里出现了瓶颈。选择 Profile→Save Profile Data As（将测试数据另存为）菜单项，如图 7-32 所示，会打开 Save As（另存为）窗口，你可以选择文件名和文件格式。文本格式看上去

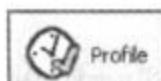


图 7-31 激活时的 Profile 工具条按钮

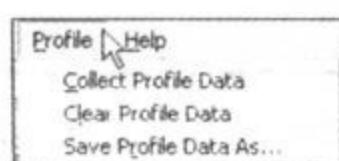
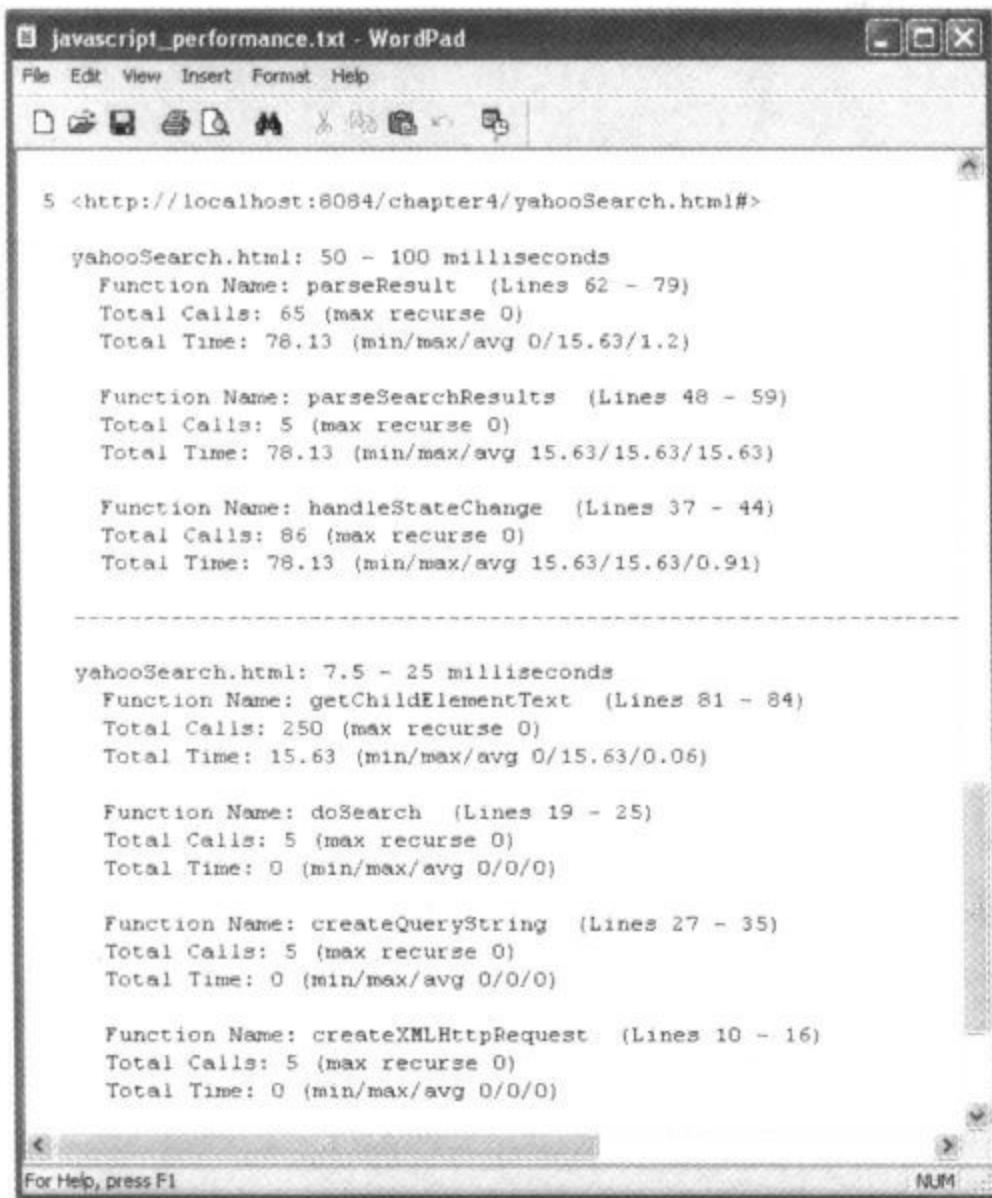


图 7-32 查看 Profile 菜单

是最易读的格式。输入所需的数据，点击OK。保存了性能数据之后，要确保选择Profile→Clear Profile Data（清除测评数据）菜单项，将数据缓冲区清空，以备下次使用。

图7-33显示了性能测评分析工具的输出例子。对于文件中的每个函数，测评工具记录函数调用的次数，函数中花费的总时间，以及每个函数调用所花费的最大、最小和平均时间。要记住，所有性能度量都以毫秒为单位。输出很容易看懂，你应该能很快决定哪一部分可能需要进一步性能调优。



The screenshot shows a Windows WordPad application window titled "javascript\_performance.txt - WordPad". The content of the document is a performance analysis report for the file "yahooSearch.html". The report is structured as follows:

```
5 <http://localhost:8084/chapter4/yahooSearch.html#>

yahooSearch.html: 50 - 100 milliseconds
  Function Name: parseResult (Lines 62 - 79)
  Total Calls: 65 (max recurse 0)
  Total Time: 78.13 (min/max/avg 0/15.63/1.2)

  Function Name: parseSearchResults (Lines 48 - 59)
  Total Calls: 5 (max recurse 0)
  Total Time: 78.13 (min/max/avg 15.63/15.63/15.63)

  Function Name: handleStateChange (Lines 37 - 44)
  Total Calls: 86 (max recurse 0)
  Total Time: 78.13 (min/max/avg 15.63/15.63/0.91)

-----
yahooSearch.html: 7.5 - 25 milliseconds
  Function Name: getChildElementText (Lines 81 - 84)
  Total Calls: 250 (max recurse 0)
  Total Time: 15.63 (min/max/avg 0/15.63/0.06)

  Function Name: doSearch (Lines 19 - 25)
  Total Calls: 5 (max recurse 0)
  Total Time: 0 (min/max/avg 0/0/0)

  Function Name: createQueryString (Lines 27 - 35)
  Total Calls: 5 (max recurse 0)
  Total Time: 0 (min/max/avg 0/0/0)

  Function Name: createXMLHttpRequest (Lines 10 - 16)
  Total Calls: 5 (max recurse 0)
  Total Time: 0 (min/max/avg 0/0/0)
```

图7-33 Venkman 性能测评分析工具的输出示例

### 7.3 小结

不论你采用哪种开发环境，都肯定会犯一些错误。据某些研究估计，开发人员几乎会花一半的时间来看懂其他人的代码，并找出和修正错误。

本章介绍的工具和技术能帮助你跟踪代码，并在尽可能短的时间内尽可能轻松地找出bug。Firefox的Greasemonkey扩展允许运行用户脚本，从而帮助你调试从浏览器发至服务器的Ajax请求，以及从服务器发回的响应。这个工具有助于确定bug究竟出现在浏览器端

还是服务器端。Microsoft Script Debugger 是一个集成到 Microsoft Internet Explorer 的 JavaScript 调试工具，它提供了基本的调试功能，如断点以及变量检查和编辑。Venkman 也是 JavaScript 调试工具，集成到基于 Mozilla 的浏览器中（如 Firefox），而且提供了更高级的调试特性，如智能断点和性能测评分析。

你不用再因为没有可用的生产性工具而躲避 Ajax 或 JavaScript 开发。使用本章的工具和技术，你就能信心百倍地扩展 Ajax 和 JavaScript 开发，因为你很清楚，就算是出现了麻烦，也有合适的工具来查找问题并加以修正。

## 万事俱备

好了，这本书里讲的内容已经很多了！现在，对于 Ajax 能够为你做什么应该有充分的认识了，而且你已经看到很多例子可以助你上路。我们介绍了许多工具，利用这些工具，开发 Ajax 应用将更为轻松，而且测试也会相当简单，因此你没有理由跳过测试。在这一章中，我们会介绍另外一些主题，如模式和框架，并提供一个更复杂的 Ajax 例子。

### 8.1 模式介绍

如今，如果一本技术书没有提到模式就好像是不完整似的。如前所述，Ajax 还相当新，所以这个领域还有很多不明确的地方。随着越来越多的网站利用 Ajax 提供的功能，你对 Ajax 的了解就会越来越深入。不过，我们还是要在后面的几节中简要地谈谈几种基本模式。要更全面地了解有关的模式，可以访问 [ajaxpatterns.org](http://ajaxpatterns.org)。另外，Christian Gross 所著的 *Ajax Patterns and Best Practices* (Apress 将于 2006 年出版) 也会对快速编写实际应用所需的各种模式提供全面的介绍。

#### 8.1.1 实现褪色技术

Ajax 的亮点之一就是，你可以只修改 Web 页面的一部分。不用重新绘制整个视图，而是只更新有变化的部分。尽管这是一种很方便的技术，但是如果用户想刷新整个页面，就会被搞糊涂。考虑到这一点，37signals 公司在其旗舰产品 Basecamp 中使用了黄褪技术 (Yellow Fade Technique, YFT)，这项技术能够巧妙地提示用户哪些部分有所变化。YFT 顾名思义：页面中有变化的部分会用黄色重新绘制，而且会慢慢地褪成原来的背景色。

褪色技术 (Fade Anything Technique, FAT) 模式在原理上与 YFT 是类似的。实际上，惟一的不同是你使用哪一种颜色褪色；毕竟，对你来说黄色可能不是最佳的选择。实现这种技术不太难；使用你最喜欢的搜索引擎就能找到相关的示例代码。图 8-1 显示了这个模式的一个例子。有变化的部分用灰色突出显示，这样用户就能很容易地发现最新的内容。

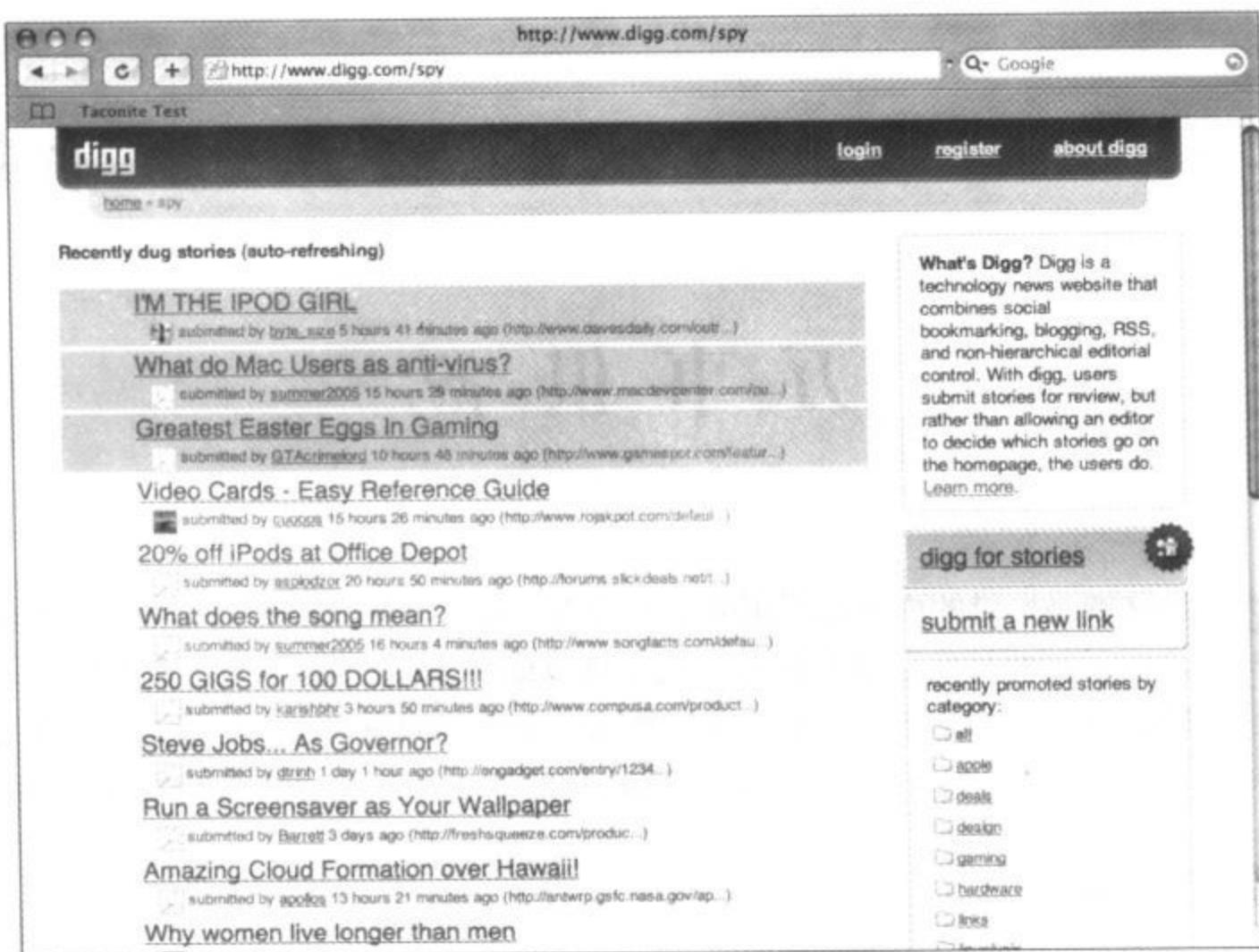


图 8-1 FAT（和自动刷新）的例子

### 8.1.2 实现自动刷新

第 4 章已经介绍过如何实现自动刷新（Auto Refresh）模式；能够自动刷新页面中的某些部分，这是很有用的。就拿天气、新闻或其他可能随时间变化的信息流来说，只重绘变化的部分会更有意义，而不应为几个小变化就刷新整个页面。当然，如果用户习惯于点击刷新（refresh）按钮，可能就不会很明显地看出这个模式有什么意义，正是因为这个原因，自动刷新模式通常与 FAT 成对出现。

自动刷新模式不仅能使你的用户少做工作，还有一个很突出的优点：它还能减少服务器上的负载。不用让数以千计的用户不断地点击刷新按钮，你可以设置一个特定的轮询周期，从而使请求能更平均地分布。（图 8-1 显示了自动刷新模式的一个例子。）

### 8.1.3 实现部分页面绘制

对于这个模式我们已经讨论过不少，不过要知道，Ajax 的一个强大之处在于，你不必再重新绘制整个页面；相反，只需修改有变化的部分。显然，这个模式可以与 FAT 和自动刷新模式一同使用。实际上，这对于 Web 应用很有帮助。

现在已经有许多框架可以帮助你修改页面的一部分，而且由于当前的浏览器中提供了可靠的 DOM 支持，因此，这种方法比你想像的要容易得多。图 8-2 显示了 A9 的 BlockView 特性，这就是部分页面绘制（Partial Page Paint）模式的一个例子。当你在左侧地图中选择不同部分时，相应的街区图就会自动改变，以反映你在地图上所指的位置（假设有相应的街区图）。

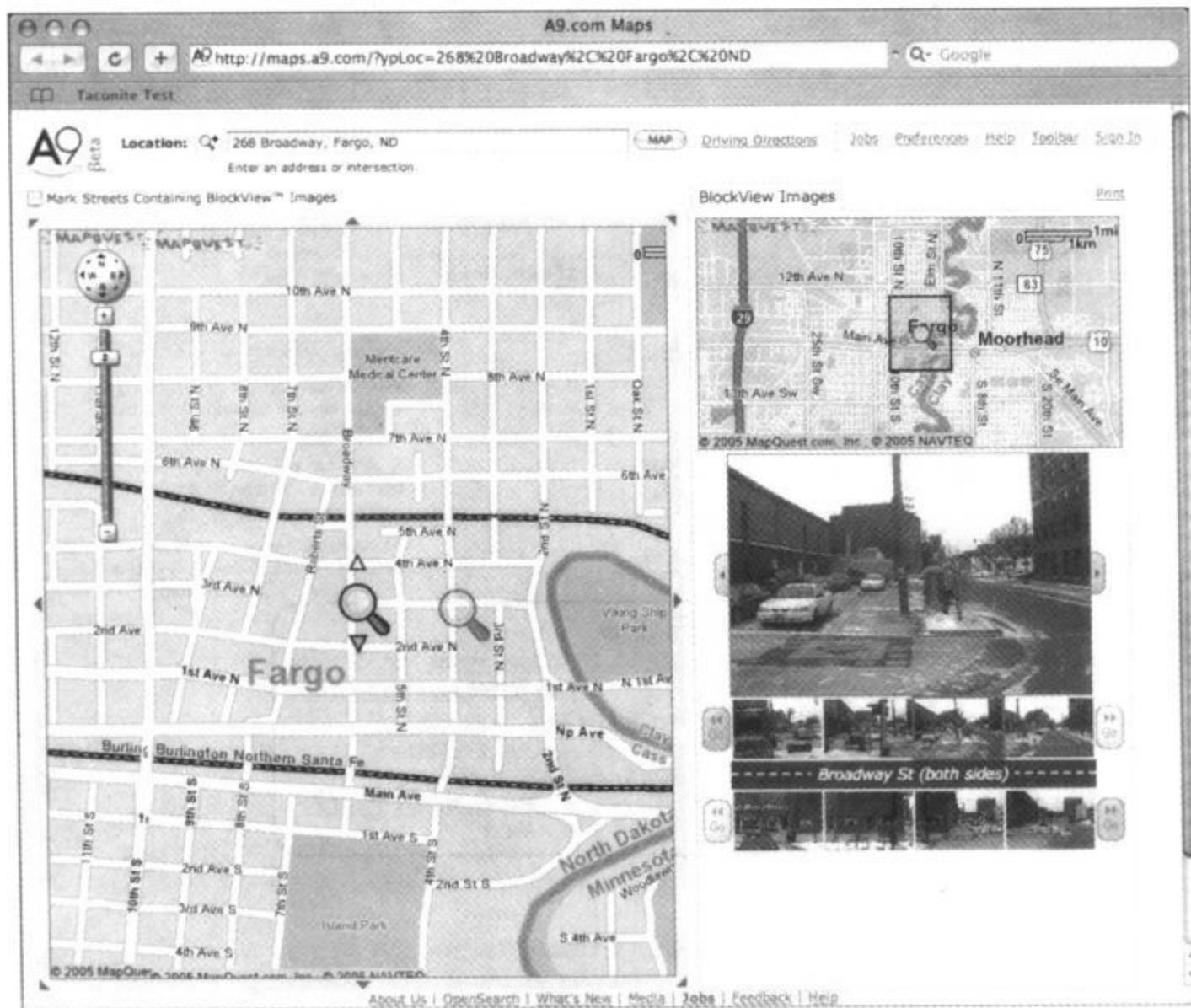


图 8-2 A9 的 BlockView 特性

#### 8.1.4 实现可拖放 DOM

原来人们总认为，门户网站可以解决所有问题。公司内部网站设计为“一站式购物”方式，员工可以轻松地得到所需的所有信息，最常用的应用、重要报告的相关链接、业界新闻等等，为此，往往认为答案就是实现一个定制的门户网站。遗憾的是，公司内部网从来没有发展到这种程度，至少部分原因在于增加新内容和移动现有部分的界面太过笨拙。通常，你必须到一个单独的管理页面进行修改（一个全页面刷新），保存你做的修改，再返回到主页（另一个页面刷新）。尽管这种方法也可以，但肯定不是最理想的。

Ajax 为门户网站赋予了新的生机，特别是使用可拖放 DOM 模式（Draggable DOM pattern）。采用这种方法，在主页上就可以编辑各个部分，并且，只需要用鼠标选中要移动的部分，把它们拖到新的位置上，就可以定制页面。很多网站都使用了这种模式，包括个性化 Google 网站 ([www.google.com/ig](http://www.google.com/ig))，如图 8-3 所示。

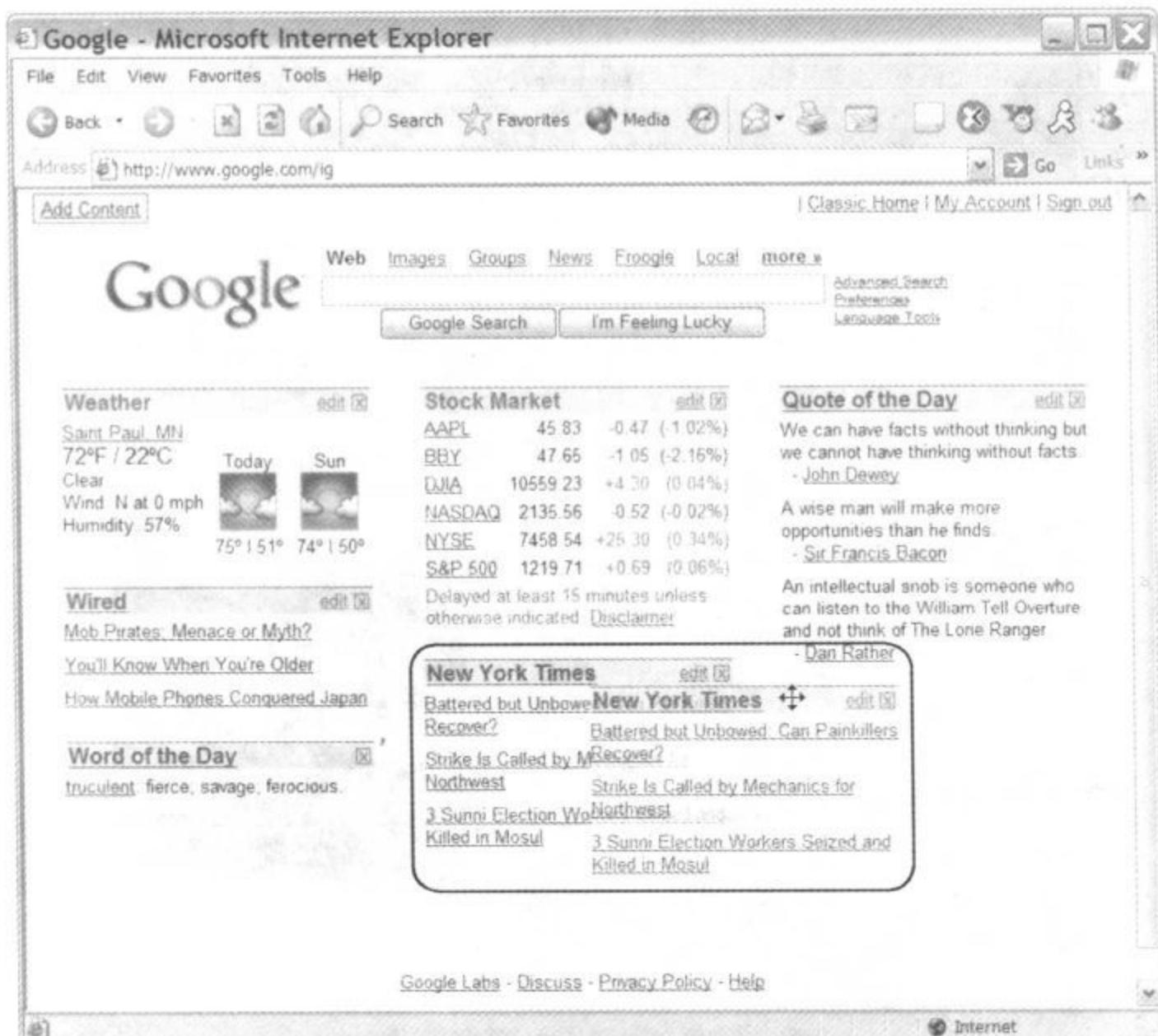


图 8-3 Google 个性化网站使用了可拖放 DOM 模式

## 8.2 避免常见的陷阱

我们已经介绍了能使 Ajax 开发更为容易的一些工具，不过除此之外，你还要知道一些常见的陷阱。你可能不会碰到太多这样的问题，但是在开始广泛使用 Ajax 之前，一定要记住以下几点。

**不可链接的页面：**你可能已经注意到了，在我们显示的大多数图中，尽管页面有变化，但是地址栏并没有变化。使用 XMLHttpRequest 对象与服务器通信时，不需要修改地址栏中显示的 URL。对于有些 Web 应用，这确实是可以“加分”的优点，但这也意

意味着你的用户不能对页面建书签，也不能向他们的朋友发送 URL（可以考虑一下地图或者有行驶方向的情况）。这个问题并非无法解决；实际上，Google Maps 现在就包括一个 Link to This Page（链接到此页面）链接（见图 8-4）。如果链接对你的应用或网站至关重要，就要注意 Ajax 会带来一点难度。

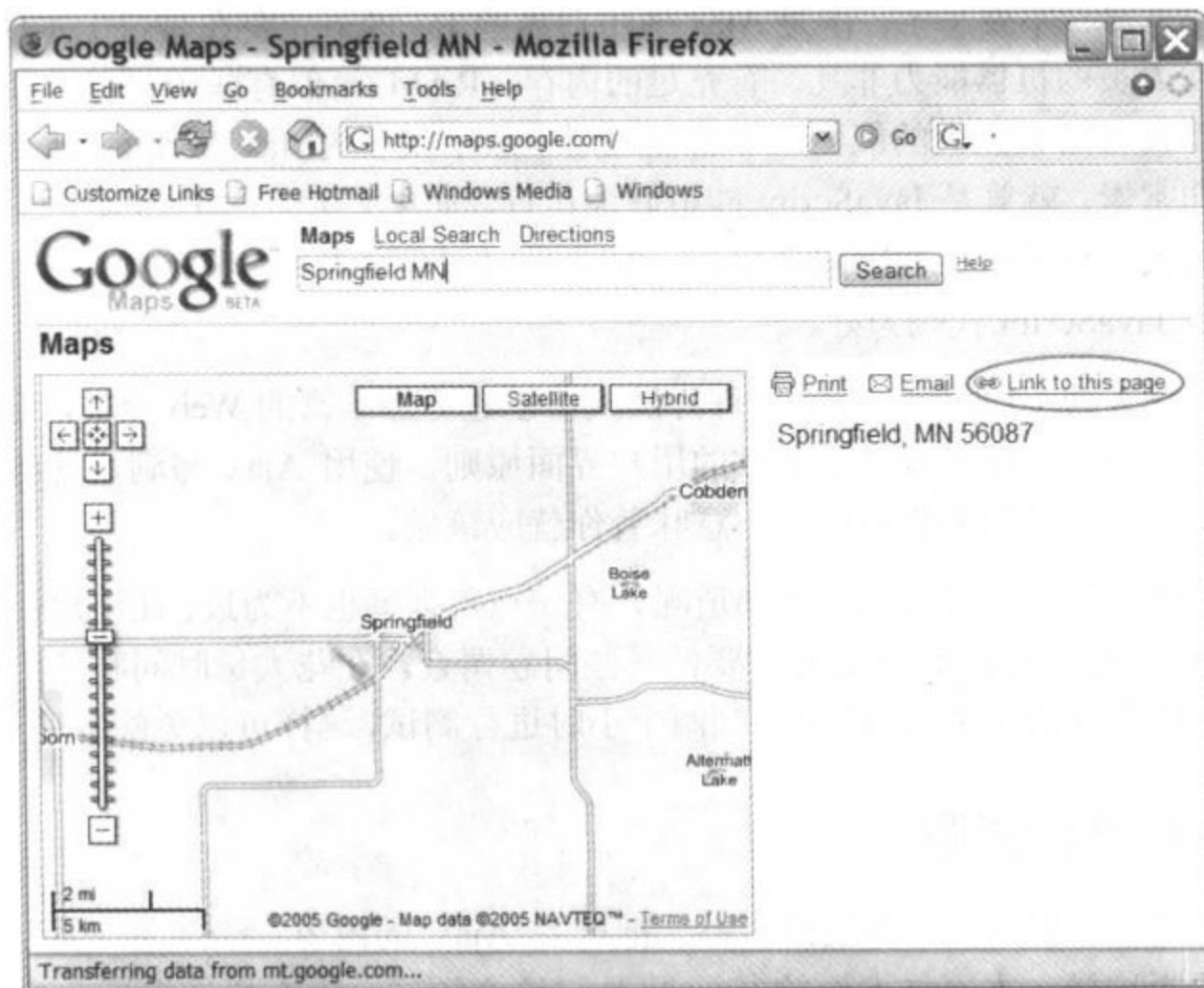


图 8-4 Google Maps 的 Link to This Page 链接

**异步修改：**异步地与服务器通信，这是 Ajax 所带来的一大进步；但也并非全无问题。这一点我们已经谈到过很多次，但还是有必要再做一些讨论：用户已经有一个常识，认为只要有变化就会重新绘制整个页面，所以如果你只更新了页面的某些部分，他们可能不会注意到。能够只重新加载页面的某些部分，这种能力固然很好，但对你的整个应用来说并不一定合适的方法，所以要审慎地使用这种方法。

**缺少可视化提示：**由于不会重新绘制整个页面，因此用户可能不知道发生了改变。最终，这就促使了 FAT 的诞生，不过你确实还有其他选择。例如，Gmail 就使用了一个“Loading”（加载中）图标，来指示它正在做某项工作（见图 8-5）。取决于你的具体应用，可能还必须增加某种提示，这样你的用户才能知道发生了什么。

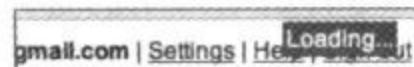


图 8-5 Gmail 的“Loading”图标

**断开的后退按钮：**有些 Web 应用故意禁用了浏览器的后退按钮，不过采用这种做法的网站很少。当然，如果使用 Ajax，点击后退按钮就没有任何意义，什么都不会做。如果你的用户希望后退按钮能起作用，而且你是在使用 Ajax 来管理页面的各个部分，可能就会出现一些问题需要解决。

**代码膨胀：**不要忘了，作为 Ajax 应用后盾的 JavaScript 要在客户上本地运行。尽管许多开发人员的机器能力非凡，有充足的内存（RAM），但有些用户仍旧使用比较老的机器，不具备这么强大的能力。如果在应用中放了太多的 JavaScript，就会发现客户端的响应非常慢。就算是 JavaScript 能很好地运行，如果有太多的 JavaScript，也意味着页面越来越大，这说明下载时间会更长。除非我们都有宽带和双 CPU 计算机，否则还是尽量减少 JavaScript 代码为好。

**违反现有的 UI 约定：**Ajax 允许开发人员创建更加丰富的 Web 应用，这是他们以前做不到的。不过，还是要遵循正常的用户界面原则，使用 Ajax 与满足这种需求并不矛盾。要记住，只是因为你能做并不意味着你就应该做。

如果你遭遇了这些陷阱又怎么能知道呢？有一点再强调也不为过：让用户代表测试你的设计。在采用全新的 Ajax 特性之前，要做一些问卷调查，在花大量时间和精力开发之前，一定要让一些用户先尝试运行。花上一到两个小时进行测试，这样可以免除日后更大的问题。

### 8.3 相关的更多资源

本书还只是一个起点——我们没有办法把有关 Ajax 的内容一一涵盖（因为我们只能用有限的资金出版这样一本不能太厚的书）。另外，这个领域一直在快速发展。好在，有很多非常棒的资源可以帮助你跟上 Ajax 的发展脚步。

第一个要提到的资源是 Ajaxian ([www.ajaxian.com](http://www.ajaxian.com))，我们几乎每天都会访问这个网站。Ajaxian 声称自己是“Ben 和 Dion 的 AJaX 行动”，它关注着 Ajax 世界中的每一个新事物。（Ben 或 Dion 有时一天之内就会提交多项新内容，这种情况时常发生。）Ajaxian.com 由业界专家 Ben Galbraith 和 Dion Almaer 维护，他们还经常在“*No Fluff Just Stuff*”等大会上发表演说。如果你对他们还不太信任，可以告诉你，他们都参加了第一届 O'Reilly/Adaptive Path Ajax 峰会，而且单从个人来讲，我们可以保证，他们的知识水平以及对这个领域投入的热情，肯定能让你信服。如果你要了解 Ajax 领域的新闻和有关事件，毫无疑问，Ajaxian.com 是再合适不过的第一站。

前面已经提到过，Ajax Patterns ([ajaxpatterns.org](http://ajaxpatterns.org)) 也是一个很不错的资源。Ajax Patterns 由 Michael Mahemoff 维护，尽管这个网站的更新不像 Ajaxian 那么频繁，但对于这么一个新领域能有如此深度确实难能可贵。不要被网站名中的模式（Pattern）吓住，这个网站提供了有关 Ajax 基础、框架和常见陷阱的大量信息。

Ajax Matters ([ajaxmatters.com/r/welcome](http://ajaxmatters.com/r/welcome)) 收集了很多与 Ajax 有关的文章和书。在这里不仅能找到讨论 XMLHttpRequest 对象的文章，还能找到有关 JavaScript 和 CSS 的绝好资源。Ajax Matters 里有一个 blog，不过更新得不太频繁。

还有一个 blog 值得一看，它的名字恰如其分：Ajax Blog ([ajaxblog.com](http://ajaxblog.com))。这个 blog 的参与者众多，而且如你所料，其中涵盖了 Ajax 领域的大量主题。与 Ajax Matters 类似，这个 blog 很好地讨论了有关 Ajax 的一些内容（如浏览器兼容性），还提供了很不错的教程。

除了可以使用 Google 来搜索有关 Ajax 的信息之外，还可以经常访问 Google Labs ([labs.google.com](http://labs.google.com))。Google Labs 在其 beta 版增加的每一个应用或特性并非都基于 Ajax，但正是因为有了 Google Maps 和 Google Suggest 之类的特性，才使得 Ajax 得到如此的关注。如果 Google 采用了确实很酷的东西，几天之内就会有人将其进行解剖，分析得通透透彻。

Adaptive Path 公司 ([www.adaptivepath.com](http://www.adaptivepath.com)) 中有些人在这个领域影响很大。Ajax 这个词就是 Jesse James Garrett (Adaptive Path 公司的创始人) 提出的；另外，Ajax 峰会就是这个公司与 O'Reilly 公司共同组织的。毫无疑问，Adaptive Path 公司是 Web 设计领域真正的领头人，肯定对 Ajax 有更多的发言权。

如果你还没听说过 Rails 或 Ruby，可以先访问 [www.rubyonrails.org/](http://www.rubyonrails.org/) 和 [www.ruby-lang.org/en/](http://www.ruby-lang.org/en/)。Rails 是一个开源的 Web 框架，由 37signals 公司的 David Heinemeier Hansson 开发，那时他正在开发一个项目管理工具 Basecamp，而且这个工具是用面向对象的脚本语言 Ruby 编写的。所有好的框架都来自于有生命力的应用，Rails 也应验了这一点，它结合了大量的优秀特性。人们之所以对 Rails 趋之若鹜，就是因为它采用了“配置上约定 (convention-over-configuration)”方法，而且内置了相关功能，可以生成典型应用的基本架构，比如为一个关系数据库创建一个 Web 前端，这种架构在目前相当时兴。尽管坦率地讲，Ruby on Rails 并不是很完美，但很多人都报告称基于 Ruby on Rails 大大提高了生产力，并且这个网站上提供了很多教程，按照任何一个教程，你都能在几分钟之内就把应用建起来，并顺利运行<sup>1</sup>。

对于本书来讲，Rails 有意思的地方是：它对 Ajax 提供了极其充分的支持。Rails 包括了一些库，可以处理拖放动作和其他常用的 Ajax 方法，另外还有一些辅助包能减轻完成某些任务的负担，如使用自动完成 (autocomplete)、调用服务器，以及在后台提交表单。Rails 对 Ajax 如此大力支持并不奇怪，原先用 Rails 构建的应用都成为展示 Ajax 作用的闪亮例子。可以看看 Basecamp、Backpack 和 Ta-da List，为你的下一个应用获得一些灵感。37signals ([www.37signals.com](http://www.37signals.com)) 很值得关注，是它真正推动了传统 Web 应用向前发展。

1. 作者就曾这样做过，而且发现，下载所有组件所需的时间甚至比让应用程序运行所需的时间更长。一位作者听说，有一个项目使用最新的 Java 工具开发用了 4 个月时间，而改用 Ruby on Rails 重写，只花了 4 个晚上（不是白天，确实是晚上）。

每周（甚至每天）都会出现新的文章和教程。只要访问以上所列的网站，就能了解到有关的几乎所有工作，尽管如此，还是很有必要看看 O'Reilly ([www.oreilly.com](http://www.oreilly.com)) 和 Apple Developer Connection ([developer.apple.com](http://developer.apple.com))。这两个网站提供了大量有关 Ajax 主题的最新信息，本书就参考了这两个网站。

## 8.4 使用框架

在本书中，我们已经提供了许多 Ajax 例子。如果你很仔细，就会注意到存在很多重复的代码。例如，代码清单 8-1 所示的代码是不是就曾经见过很多次？<sup>1</sup>

**代码清单 8-1** 本书中重复最多的代码段

```
var xmlhttp;

function createXMLHttpRequest() {
    if (window.ActiveXObject) {
        xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
    }
    else if (window.XMLHttpRequest) {
        xmlhttp = new XMLHttpRequest();
    }
}
```

当然，在一个实际应用中，我们会对这个小例程进行抽象。实际上，还可能更进一步，创建一个特殊的库来封装 Ajax 的麻烦、重复的细节。而且，如果你再做一个快速的 Google 搜索，就能发现有关 Ajax 框架的很多资源。要想快速了解有哪些可用的框架，请参考附录 B。

## 8.5 Taconite介绍

自豪地插一句：本书的作者与人合作创建了一个开源的 Ajax 框架：Taconite。我们承认，这有点自吹自擂，但说实在的，我们确实认为 Taconite 非常棒。尽管原来 Taconite 是为基于 Java 企业版的应用构建的，不过我们后来又将 Taconite 的核心重构为一个客户端库，可以与任何服务器端技术一同使用。除此以外，把 Taconite 服务器端组件移植到其他技术（如.NET）中也不难。

Taconite 为什么这么特殊？在 Web 应用的发展历程中，Ajax 是迈出的很大一步。不过，多年以来我们一直在与浏览器间的不一致性做斗争，而且开发大量的 JavaScript 存在“与生俱来”的困难，这也是我们着力要克服的。因为我们很懒，希望只需“构建一次”，这样我

---

1. 说实在的，到底有多少次？我们数了一下，包括这一次，一共 19 次。

们就能轻松地重用以往艰苦得来的工作。(另外，可以告诉老板完成某个工作要花3个星期，但是实际上一个星期就做完，剩下的时间就可以自由安排、尽情享受了。)

我们知道，你可能会问：为什么我要了解 Taconite？利用 Taconite，你就不用处理 JavaScript，而且更重要的是，可以确保能跨浏览器运行。Taconite 的核心是一个解析器，可以将正常的 HTML 代码转换为一系列 JavaScript 命令，这些 JavaScript 命令会动态地创建浏览器上的内容。

### 8.5.1 Taconite 原理

要生成动态的 HTML，一直以来这都是一项很困难的工作。我们已经讨论过，在 Web 的早期岁月，可以使用 CGI，然后出现了 servlet。当然，这两种解决方案也是可行的，但是究其本质，都是通过串连接（string concatenation）来创建 HTML 内容。所有的 servlet 开发人员都会告诉你，串连接很难开发，很麻烦，很容易出错，而且不便于维护。针对这些问题，开发人员创建了 JSP 和 ASP 等模板模型来减轻开发的负担，只要使用正常的 HTML 再加上一些特殊的标记就能开发动态的网页。与采用“串连接”方法相比，这些网页的构建和维护都要容易得多。

在前面的一些例子中已经看到，使用 Ajax 来生成动态内容时，也会出现同样的一些问题：可以使用 DOM 方法来创建内容，但是这种方法可能很麻烦，而且通常会带来大量多余的代码。作为这本书的读者，你应该不只是一个一般水平的开发人员；不过，有些东西你可能并不知道，如果我们说一些 DOM 方法在不同浏览器上有不同的表现，请相信我们所说的。

很多人决定在应用程序中采用一个新的 Ajax 特性时，都会依赖于 innerHTML，并结合使用串连接。尽管在服务器上建立 innerHTML 内容（使用你最喜欢的语

言）确实有帮助，但 innerHTML 并非没有问题。虽然 innerHTML 得到了广泛支持，但它不是一个 W3C 标准。利用 innerHTML 可以做很多事情，不过它也有局限。例如，不能在其他的表行之间插入一个表行。当然，最大的难题是，在不同的浏览器上，innerHTML 的行为存在差异，相信你肯定会对此感到震惊。

### 8.5.2 解决方案

Taconite 提供了一个定制解析器来解决这个核心问题，这个解析器能把 HTML 转换为与之等价的 W3C DOM 方法，来创建指定的内容。这说明，可以使用你喜欢的任何 HTML 生成技术来编写动态内容。这样就可以鱼和熊掌兼得：既能得到新的 Ajax 特性，而且无需处理串连接，也不用使用标准 W3C DOM 调用。而且，作为 Taconite 的开发人员，我们都很负责，所有跨浏览器的繁琐问题我们都已经为你解决了！

这个解析器由一组定制 JSP 标记使用，使你能以一种自然的方式写内容，即在 JSP 中嵌入

HTML。你的页面中不必再充斥着大量的`document.createElement`和`document.appendChild`命令来动态创建新内容。JSP定制标记会使用这个解析器负责所有这些工作，而你只需坐在一边，享受新鲜的加冰穆哈咖啡！所得到的JavaScript作为响应的一部分返回给浏览器，在浏览器上，客户端JavaScript库会接管工作，并执行这些命令，生成所需的输出。

Taconite 并不想完全从头开始；实际上，它把目光对准了 Ajax 开发的致命弱点：动态更新 Web 页面可能需要编写大量的 JavaScript，这个任务相当麻烦。不同于其他的一些工具包，Taconite 并不打算发明一种新的方法向服务器发送请求数据；它只是依赖于经验证的名/值对，这些名/值对要么嵌在 GET 请求的查询串中，要么安全地放置在 POST 的体中。客户端库甚至会自动创建名/值对查询串，我们甚至还为你完成了数据的转义处理。你要做的只是告诉 Taconite 应当向服务器发送什么值，剩下的工作交给它就行了。

因为所有 JavaScript 都是解析器根据你指定的 HTML 创建的，所以你不用操心处理 JavaScript 在不同浏览器上的不兼容性。这个工作会由解析器来完成。记住了吗？Taconite 会负责处理不兼容性。说实在的，你确实不用为此担心；我们都已经为你做好了。你要做的就是在 JSP 中指定动态内容（指定为 HTML），余下的事情解析器会负责。

Taconite 设计的一大妙处在于，它鼓励你把所有业务逻辑都放在服务器端（坦率地讲，在这里编写业务逻辑更容易），而浏览器则能司其所长：它会显示一个用户界面。Taconite 简化了从浏览器向服务器做出 Ajax 请求的工作，这样你就能很容易地重用领域逻辑。当然，不光是能去除冗余，避免在浏览器中放置业务逻辑还有其他原因。如果把领域逻辑放在客户端上，就会大开安全之门，滋生脆弱性。如果你的逻辑在浏览器上实现，倘若一个恶意用户使用诸如 Greasemonkey 等工具来更新 JavaScript，并插入一些特殊的处理（比如让你的新产品打 9 折），你又如何防范呢？利用 Taconite，可以很容易地在服务器端访问你的业务例程，而且，要开发、调试业务逻辑，并保证其安全，在服务器端完成是最容易的。

Taconite 的服务器端库是轻量级的，只需要你最喜欢的 JSP/servlet 容器就行了，别的都不需要。配置是最简单的（没有大量 XML！）；你要做的就是把 Taconite 标记库描述文件（tag library descriptor, TLD）的位置放在 WEB-INF.xml 文件中。归功于它的小规模和零依赖性，Taconite 可以很好地与任何 J2EE Web 框架合作。不论你使用 Struts、Spring、MVC、WebWork、Tapestry，或者其他的技术，都可以使用 Taconite，而无需任何额外的工作，也没有太陡的学习曲线。就以 Taconite 与 Struts 集成为例：客户端 JavaScript 库自动完成创建查询串的任务，查询串将发送到服务器。Struts 根据查询串的值自动创建和填写一个 ActionForm 对象，并把 ActionForm 传递到一个 Action 类，在这里开始“真正”的处理工作。处理了领域逻辑之后，Action 类将响应转发给一个 JSP，这个 JSP 使用 Taconite 的定制 JSP 标记来呈现内容，它会动态地更新页面的内容。

最近，我们已经把这个解析器移植到一个 JavaScript 库，以便非 Java 开发人员也可以使

用。要使用基于 JavaScript 的解析器，动态内容必须是嵌在一个特殊 Taconite 标记中的合法 XML。使用这种方法，内容将作为 XHTML 返回，它会解析为 JavaScript，然后在浏览器上执行。

由于 JavaScript 是在一处（Taconite 解析器）集中生成，所以可以确保 Taconite 解析器能处理浏览器之间所有已知的不兼容性。这些工作由我们来做，你就能集中精力，以自然的方式写内容（写为 HTML）。出现新问题时，我们可以更新解析器，而不会影响任何客户代码！

### 8.5.3 Taconite 怎么处理内容

既然 innerHTML 不是答案，那么在客户端到底要做什么呢？如果查看 W3C 的加载和保存规约，就会看到 ACTION\_APPEND\_AS\_CHILD 和 ACTION\_REPLACE 之类的东西。Taconite 支持这样一些动作，这就为你提供了充分的灵活性，可以自由地放置内容。如果遵循加载和保存规约的思想，就会使从 Taconite 到 W3C 规约的迁移更为容易。Taconite 当前支持以下动作，假设这些动作都针对一个指定的上下文元素：

- 追加为子元素
- 追加为第一个子元素
- 删除
- 插在后面
- 插在前面
- 替换子元素
- 替换

Taconite 不是一个工具包，并不提供部件，但它相当灵活。不管你给它什么，它都能呈现出来，而不是像一些开发人员所想的那样，只是像文本框一样处理。当然，如果你非得要这样一个部件，也能很容易地在 Taconite 之上建立。另外，利用基于 JavaScript 的解析器，就能结合使用你喜欢的任何服务器端技术。

要记住，因为 Taconite 是开源的（得到 Apache license 许可），你可以自由地查看代码，加以调整，从中学习，甚至可以把它移植为你喜欢的语言。要好好看看 Taconite 主页 ([sourceforge.net/projects/taconite](http://sourceforge.net/projects/taconite)) 来了解更多详细内容。

## 8.6 Dashboard 应用介绍

Mac OS X Tiger 的引入带来了数百个新特性，其中包括一个新的浏览器，它提供了改进的“确实简单合成”（Really Simple Syndication, RSS）支持；另外还提供了一个可以简化常见任务的自动化工具，以及一个速度极快的桌面搜索功能。对 Tiger 谈及最多的特性之一是

Dashboard，这实际上是一些小应用的集合（见图 8-6）。如果你没有 Mac（或者所有朋友也没有 Mac），可以把 Dashboard 应用看作是一种替代做法，通常你可能会每天数次打开浏览器来做一些小事情，如检查股票、在字典里查个单词，或者看看今天晚上有什么电视节目，Dashboard 应用就可以取代这些小事情。你不必频频登录你爱去的天气网站，只需访问 Dashboard，看一下你的本地天气预报就可以了！



图 8-6 Mac OS X Tiger 中的 Dashboard

Dashboard 部件就是一些 HTML 和 JavaScript，并夹杂一点 CSS 来得到更精致的外观。好的 Dashboard 部件只有一个单纯的用途，规模都很小，只做一件事，并且把它做好。原先 Mac OS X Tiger 内置了少量小部件（日历、航班记录器、股票记录器，天气部件，等等），不过，如今已经有上千个这样的部件。Dashboard 部件连接到 Internet 来交付实时信息；这意味着只要一次点击就能保证跟踪到最新信息。Dashboard 最大的亮点是：很多人可以创建自己的部件，实际上，大多数可以下载的部件就不是用 Apple 编写的。

遗憾的是，我们不一定都有 Mac，至少现在是这样，在 Intel 主机上无法运行 Mac OS X。不过，这并不是说你无法享受到 Dashboard 的特性。实际上，在下一节中，我们就会在浏览器中模拟 Dashboard，为此将使用 Ajax，再结合一些免费的 Web 服务。

## 8.7 用Taconite构建Ajax Dashboard

到目前为止，本书中的所有例子都很小，而且点到为止，只展示一个或两个特定的主题。这些例子并不是很漂亮，但是确实能解开 Ajax 谜题中的重要部分。

这一章的例子是一个 Ajax Dashboard，它是一个完全跨浏览器的应用，模拟了 Mac OS X Tiger Dashboard 特性。虽然这个例子不像 Mac OS X Tiger Dashboard 那么光鲜夺目，但确实证明了 Ajax 很容易使用，而且可以使可用性大幅攀升。Ajax Dashboard 也可以作为 Taconite 框架的一个演示测试床，由此你能看到 Taconite 框架能省去你多少工作。Ajax Dashboard 的完整源代码可以从 Apress 网站 ([www.apress.com](http://www.apress.com)) 的 Source Code (源代码) 部分得到。在那里可以下载源代码，还可以预先构建 WAR 文件，以便在当前的 servlet 容器（如 Tomcat）中运行。

### 8.7.1 一般特性介绍

图 8-7 显示了 Ajax Dashboard。它包括 4 个小窗口，每个窗口包含一个不同的小应用。第一个窗口提供了一个简单的 7 天天气预报，对于任何合法的美国地区邮政编码 (ZIP)，会显示 7 天的预报最高温度和最低温度，并显示当天的云量情况。只要 ZIP 编码有变更，天气预报就会自动改变，只要浏览器中这个窗口是打开的，Ajax 就会自动刷新天气预报。

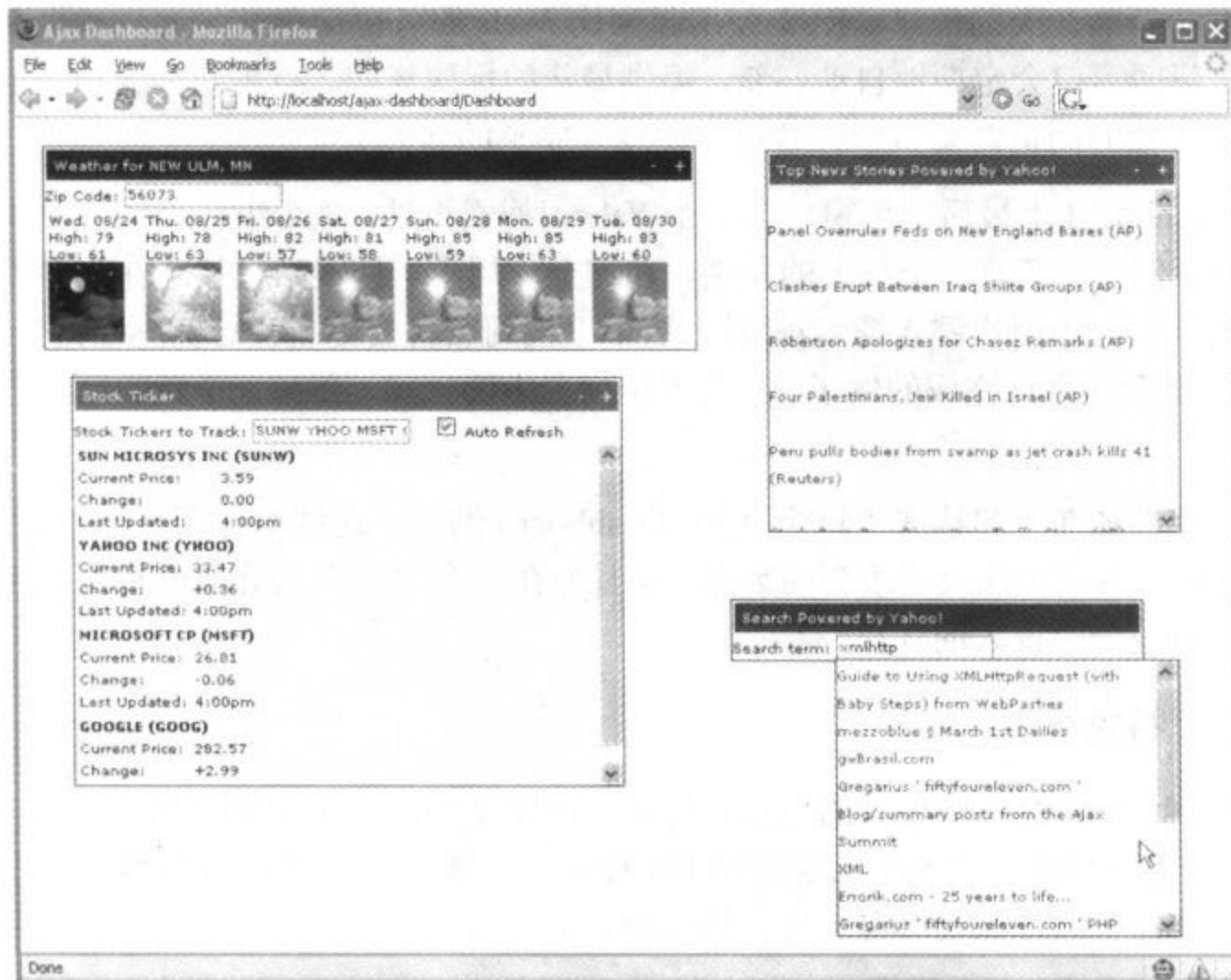


图 8-7 Ajax Dashboard 示例应用

天气预报数据由一个 Web 服务提供，其 WSDL 文档位于 [www.webservicex.net/WeatherForecast.asmx?WSDL](http://www.webservicex.net/WeatherForecast.asmx?WSDL)。美国国家海洋和大气管理局（National Oceanic & Atmospheric Administration, NOAA）在 [weather.gov/forecasts/xml/](http://weather.gov/forecasts/xml/) 上发布了自己的一个更完备的天气预报 Web 服务。每次打开 Ajax Dashboard 时，都会访问我们使用的 Web 服务来得到默认 ZIP 编码所对应的天气预报。一旦打开，就会定期访问这个 Web 服务，或者当 ZIP 编码有变化时访问 Web 服务，来得到更新的天气预报。在这些更新期间，Ajax 会动态地更新天气预报，所以更新是无缝进行的，而且不需要完全刷新页面。

股票行情（Stock Ticker）窗口会跟踪你最关心的股票在股市上的表现。可以用它来看看你的投资会以多快的速度增长，相应地你是不是能更早地退休（或者，也可能很不幸，你会看到你的投资会以多快的速度缩水，相应地是不是还得工作更长时间才能退休）。在文本框里输入所关心的股票，选中 Auto Refresh 复选框，以启用自动刷新功能，观察一下一天内股票价格怎么变化，在此根本不需要手工地刷新页面。类似于天气预报，股票记录器使用了 Ajax，并且会定期地自动更新，以确保信息总是最新的。股票记录器从一个 Web 服务获取数据，其 WSDL 位于 [www.swanandmokashi.com/HomePage/WebServices/StockQuotes.asmx?WSDL](http://www.swanandmokashi.com/HomePage/WebServices/StockQuotes.asmx?WSDL)。

新闻窗口是一个简单的窗口，它会显示 Yahoo! 提供的当前最新新闻。这里没有使用 Web 服务，而是把 Yahoo! 作为一个 RSS 输入源来提供新闻。RSS 是一种简单的基于 XML 的格式，可以发布新闻之类的信息。RSS 使用起来比 Web 服务更简单。与天气预报和股票行情窗口一样，新闻窗口会定期地自动刷新，提供最新的标题新闻。点击新闻窗口中的一项，会在另外一个单独的浏览器窗口中打开这篇文章（完整的新闻）。

Ajax Dashboard 上最后一个窗口是一个 Yahoo! 搜索组件。这个组件类似于第 4 章介绍的 Yahoo! 搜索示例。二者有一个很大的差别，在这个应用中，搜索结果显示在文本框下面的一个下拉列表框中，当用户键入搜索项<sup>1</sup>时，这个下拉列表就会更新。这实际上结合了第 4 章中的 Yahoo! 搜索和自动完成的例子。点击下拉列表中的一项，会在一个新的浏览器窗口中打开这个资源。

这 4 个组件加在一起构成了这个 Ajax Dashboard 例子。这些组件都很适合使用 Ajax 技术，因为它们要表示可能快速改变的数据，而且能在同一个窗口中无缝地更新数据而无需刷新整个页面，这是一个很大的优点。

### 8.7.2 设计特性介绍

Ajax Dashboard 有一些很好的特质，这使它区别于传统的基于浏览器的应用。对于这 4 个组件，每个组件都在一个有绝对位置的 div 标记中呈现，并且，基于 DOM-Drag 库的帮助，可以用鼠标把各个组件拖到屏幕上任意的位置。

1. 原文此处为“search time”，应当是“search term”。——译者注

DOM-Drag 库可以从 [www.youngpup.net/2001/domdrag](http://www.youngpup.net/2001/domdrag) 得到, 它的作者称之为面向现代 DHTML 浏览器的轻量级、易于使用的拖放 API (见图 8-8)。每个组件的建立只需几行 JavaScript 代码, 因此很适合使用 DOM-Drag, 对此应该没有什么争议。这里惟一的 JavaScript 文件只有 121 行代码, 大小大约是 8KB。每个 Ajax Dashboard 组件都有一个彩色的标题栏, 这是拖动组件的“手柄”。要移动一个组件, 只需点击彩色标题栏区域中的任何位置, 并按住不放, 把组件拖到期望的位置上, 然后松开鼠标键。利用 DOM-Drag, 用户可以把组件放到他们希望的任何位置。这种改进尽管看似微小, 但使得 Ajax Dashboard 更像是一个多文档界面 (multiple document interface, MDI) 客户应用, 而不只是一个 Web 页面。

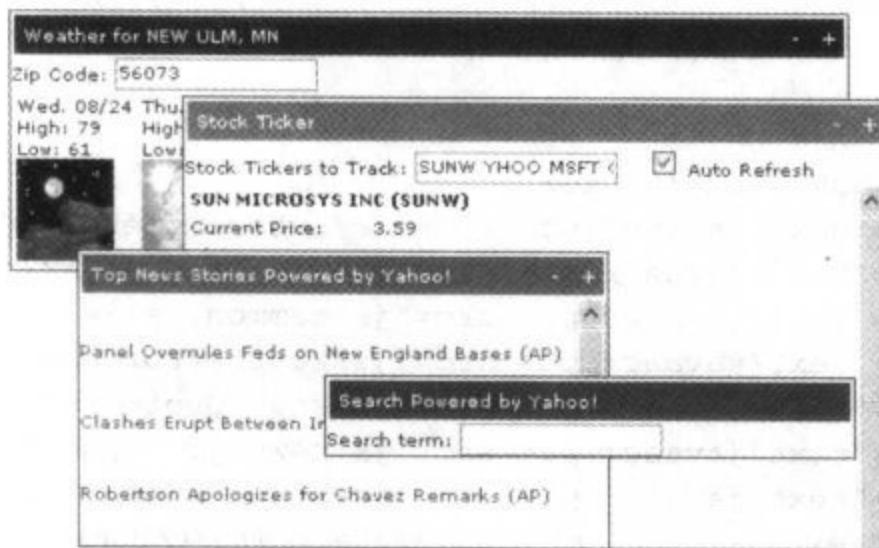


图 8-8 DOM-Drag 允许将独立的组件拖到屏幕的任何地方

你可能还注意到天气预报、股票行情和新闻组件标题栏的右侧有小的加减号。这些加减号能分别将组件的内容最大化和最小化。最小化组件的内容会把数据叠起来, 从而只能看到标题栏, 这样可以节省宝贵的屏幕空间 (见图 8-9)。在这里看起来这个特性好像不太重要, 因为 Ajax Dashboard 只有 4 个组件, 但是想像一下, 如果应用扩展到包括更多的组件, 会怎么样呢? 屏幕上充斥着大量的组件, 看上去就会很乱, 不便于阅读, 把使用较少的组件最小化, 就能使应用的外观更加简洁, 而且只需一点鼠标, 又能看到最小化组件中的数据。

通过使用一个 `div` 元素及其 `style` 属性的 `display` 性质, 可以得到最大化和最小化的效果。下一节将更具体地介绍这一点。

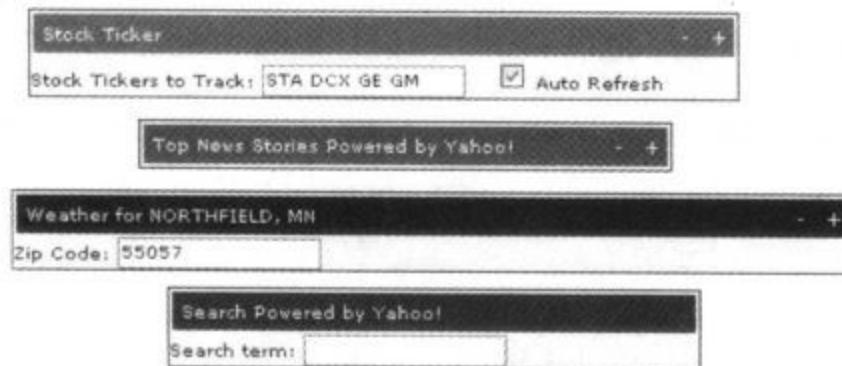


图 8-9 最小化状态的 Ajax Dashboard 组件

### 8.7.3 分析代码

下面开始分析构成 Ajax Dashboard 的代码，我们会介绍所有这些组件如何集中在 1 个 Web 页面上。代码清单 8-2 显示了呈现 Ajax Dashboard 主页的 JSP。

**代码清单 8-2 ajaxDashboard.jsp**

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html>
<head>
    <title>Ajax Dashboard</title>
    <script type="text/javascript" src="js/autocomplete.js"></script>
    <script type="text/javascript" src="js/dom-drag.js"></script>
    <script type="text/javascript" src="js/common.js"></script>
    <script type="text/javascript" src="js/weatherForecast.js"></script>
    <script type="text/javascript" src="js/stockQuote.js"></script>
    <script type="text/javascript" src="js/news.js"></script>
    <script type="text/javascript" src="js/search.js"></script>
    <script type="text/javascript" src="js/taconite/taconite-client.js"></script>
    <script type="text/javascript" src="js/taconite/taconite-parser.js"></script>
    <link type="text/css" rel="stylesheet" title="style" href="css/styles.css" >
        </link>
</head>

<body>

    <%@ include file="weather/weatherForecast.jsp" %>

    <%@ include file="stockQuote/stockQuote.jsp" %>

    <%@ include file="news/newsItems.jsp" %>

    <%@ include file="search/search.jsp" %>

    <script type="text/javascript" src="js/ajaxDashboard.js"></script>

    <div style="position:absolute;overflow:auto;display:none;background-color:white"
        id="popup">

    </div>

</body>

</html>
```

这个 JSP 很大程度上只是用来包含构成各个组件的 JSP。文件最上面是对各组件相应 JavaScript 文件的引用。以这种方式把 JavaScript 划分开，可以使文件更可管理，更易于修改。所有组件都公用的 JavaScript 函数放在 common.js 文件中。构建这个例子中用到了 Taconite 框架，taconite-client.js 和 taconite-parser.js 文件是这个 Taconite 框架特定的 JavaScript 文件。

每个组件都有自己的 JSP，均包含在主 JSP 的 body（体）部分中。让组件有自己的 JSP 文件，这样能使维护更加容易，而且倘若要增加组件，或者要从主 JSP 文件中删除组件，这样做还能简化增加或删除组件的过程。在 body 标记中还有一个名为 ajaxDashboard.js 的 JavaScript 文件的引用。对应搜索组件的下拉列表，其绝对位置 div 标记的 id 属性值为 popup，有关内容请见“建立一个更好的 Autocomplete”小节。

ajaxDashboard.js 文件包括了加载页面时必须运行的 JavaScript。放在函数体之外的代码一旦由浏览器加载就会执行。大部分代码都是 DOM-Drag 库初始化组件窗口的可拖放行为所必需的代码。ajaxDashboard.js 中的 initDomDrag 函数封装了初始化组件可拖放行为的代码，如下所示：

```
function initDomDrag(handleID, rootID) {  
    var handle = document.getElementById(handleID);  
    var root = document.getElementById(rootID);  
    Drag.init(handle, root);  
}
```

handle 变量引用了构成组件彩色标题栏的 div 元素。这是一个“手柄”，可由鼠标点击，并用来拖动组件的余下部分。root 变量引用了包括所有组件内容的父 div 元素。handle 元素必须是 root 元素的子元素。最后一点，最后一行把 root 和 handle 变量传递给 DOM-Drag 库，使得组件可以拖放。例如，下面的 initDomDrag 函数就会使股票报价组件可以拖放：

```
initDomDrag("stockQuoteHandle", "stockQuoteRoot");
```

#### 8.7.4 分析天气预报组件

代码清单 8-3 详细列出了显示天气预报窗口的代码。这里的父元素是 id 属性为 root 的一个 div 元素。最外层 div 元素的第一个子元素是另一个 div 元素，其 id 属性为 handle。这个子 div 表示组件的彩色标题栏，这个区域可以点击，而且可以拖动，从而能在屏幕上移动整个组件。Ajax Dashboard 中包括的所有组件都按照这种模式：一个最外层的 div 元素表示组件窗口，其后是另一个 div 元素，定义了组件的彩色标题栏。

对于天气预报组件，显示标题栏的 HTML 后面是一个 div 元素，它包括了一个文本框，所需的 ZIP 编码就要输入到这个文本框中。最后，会有一个 id 属性为 weatherContent 的 div 元素，它包含了具体的天气预报。

**代码清单 8-3 weatherForecast.jsp**

```

<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

<div id="root" style="left:20px; top:20px;">
    <div id="handle">

        <table width="100%" border="0" class="textbox">
            <tr>
                <td align="left" class="controls">
                    <span id="forecastLocation">
                        <%@ include file="weatherLocation.jsp" %>
                    </span>
                </td>
                <td align="right">
                    <a class="controls"
                        href="javascript:minimize('weatherContent');">
                        -
                    </a>
                    &nbsp;
                    <a class="controls"
                        href="javascript:maximize('weatherContent');">
                        +
                    </a>
                </td>
            </tr>
        </table>

    </div>

    <div class="normalText">
        Zip Code:
        <input type="text" name="forecastZipCode" id="forecastZipCode"
            onkeyup="handleZipCodeChange();" class="normalText"
            value="<%=\!ajaxdashboard.Constants.DEFAULT_WEATHER_ZIP_CODE%>" />
    </div>

    <div id="weatherContent">
        <%@ include file="weatherTable.jsp" %>
    </div>
</div>
```

可以注意到，这个 JSP 包含了来自另外两个 JSP 文件的内容：weatherLocation.jsp（见代码清单 8-4）和 weatherTable.jsp（见代码清单 8-5）。这两个 JSP 分别显示天气预报的位置和具体的预报情况。在分析为什么分别为这两部分建立不同的 JSP 文件之前，先来看

一下它们的源代码。这些源代码应该与你预想的差不多。`weatherLocation.jsp` 文件只是输出当前预报位置的名字，`weatherTable.jsp` 文件会显示一个一行 7 列的表，详细显示 7 天的天气预报。

#### 代码清单 8-4 `weatherLocation.jsp`

```
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

Weather for ${forecastData.placeName}, ${forecastData.stateCode}
```

#### 代码清单 8-5 `weatherTable.jsp`

```
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>



|                                                                                                                                                                        |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| \${forecast.day}  <br/> High: \${forecast.maxTemperatureF}  <br/> Low: \${forecast.minTemperatureF}  <br/>  |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------|


```

`weatherLocation.jsp` 文件的代码相当简单。它使用了 Java 标准标记库 (Java Standard Tag Library, JSTL)，动态填入预报位置所在的城市名和州名。在 `weatherTable.jsp` 中，JSTL 标记迭代处理了一个天气数据对象数组，其中每个天气数据对象都封装了某一天的预报信息。对于每一个天气数据对象，会显示一个表单元格，其中显示这一天的最高温度和最低温度，还会显示一张说明云量情况的图片。

现在你可能想知道：Ajax 用在哪里呢？下面来看如何向服务器发送 Ajax 请求。以下函数出现在 `weatherForecast.js` 文件中，它使用了 Taconite 框架来发送请求，要求得到对应一个特定 ZIP 编码的天气预报：

```

function updateWeatherForecast() {
    var ajaxRequest = new AjaxRequest("UpdateWeatherForecast");
    ajaxRequest.addFormElementsById("forecastZipCode");
    ajaxRequest.sendRequest();
}

```

要注意的第一件事是，`updateWeatherForecast` 方法体中只有 3 行代码。与之对照，第 4 章中也有这样一个例子，要把一个参数作为请求的一部分发送给服务器，但在那个方法体中，只把具体创建 XMLHttpRequest 对象、创建查询串和发送响应的代码算在内，就有 28 行代码之多。

`AjaxRequest` 对象是 Taconite 框架中浏览器端功能的核心。`AjaxRequest` 对象封装了它自己的 XMLHttpRequest 对象实例，因此 `AjaxRequest` 的多个实例不会共享同一个 XMLHttpRequest 对象实例。`AjaxRequest` 对象定义了一些方法，这样在构建作为 Ajax 请求一部分的查询串时能够有所简化，包括累加指定的表单元素值，甚至对值应用适当的编码。`AjaxRequest` 对象还会处理服务器的响应。

`updateWeatherForecast` 函数的第一行创建了 `AjaxRequest` 对象的一个实例，为 `AjaxRequest` 构造函数传递了要处理这个 Ajax 请求的 URL。第二行调用 `AjaxRequest` 对象的 `addFormElementsById` 方法，为之传递了 ZIP 编码文本框的 `id` 属性。这个方法会把用户输入的 ZIP 编码自动增加到将发送给服务器的查询串中。`addFormElementsById` 方法可以取任意多个参数。如果要把两个表单元素的值作为 Ajax 请求的一部分发送给服务器，而不是把 `addFormElementsById` 方法调用两次，就可以只调用一次，并为这个方法提供两个元素 ID，两个元素 ID 之间用一个逗号隔开。`updateWeatherForecast` 方法的最后一行指示 `AjaxRequest` 对象将 Ajax 请求发送给指定的 URL。

更新天气预报的 servlet (`UpdateWeatherForecastServlet.java`) 的源代码如代码清单 8-6 所示，它会处理 Ajax 请求来更新天气预报。如果不考虑访问 Web 服务的细节，这个 servlet 只是把 `request` 对象的 ZIP 编码取出，并传给 `WeatherForecastService` 对象。这个对象返回 `WeatherForecasts` 对象的一个实例，它封装了所有的预报数据。`WeatherForecasts` 对象作为一个属性附加到 `request` 对象，最后 servlet 把 `request` 和 `response` 转发到 `weatherForecastAjax.jsp` 文件。

#### 代码清单 8-6 UpdateWeatherForecastServlet.java

```

package ajaxdashboard.servlet;

import ajaxdashboard.service.WeatherForecastService;
import java.io.*;
import java.util.Date;

```

```

import javax.servlet.*;
import javax.servlet.http.*;

public class UpdateWeatherForecastServlet extends HttpServlet {
    protected void processRequest(HttpServletRequest request
                                  , HttpServletResponse response)
        throws ServletException, IOException {

        String zipCode = request.getParameter("forecastZipCode");
        WeatherForecastService forecastService = new WeatherForecastService();
        request.setAttribute("forecastData"
                            , forecastService.getForecastFor(zipCode));

        System.out.println("Weather updated at: " + new Date().toString());
        request.getRequestDispatcher("/jsp/weather/weatherForecastAjax.jsp")
            .forward(request, response);
    }

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        processRequest(request, response);
    }

    protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        processRequest(request, response);
    }
}

```

在前面的例子中，响应总会以一种 XML 或 JSON 的格式返回给浏览器，再由浏览器负责读取响应，并生成新的内容，或者根据结果更新现有的内容。这通常需要几行代码来写出 `document.createElement()` 或 `document.appendChild()` 调用，写起来可能很麻烦。例如，第 4 章的 Yahoo! 搜索例子就需要 37 行代码来显示结果。

可以与代码清单 8-7 做个比较，这里显示了 `weatherForecastAjax.jsp` 文件的内容。这个文件只有 19 行（其中 8 行都是空行），同样能处理天气预报组件的更新，从而对 Ajax 请求做出响应。

#### 代码清单 8-7 weatherForecastAjax.jsp

```

<%@ taglib uri="http://taconite.sf.net/tags" prefix="tac" %>

<tac:taconiteRoot>

    <tac:replaceChildren contextNodeID="forecastLocation" parseOnServer="true">

        <%@ include file="weatherLocation.jsp" %>

```

```

</tac:replaceChildren>

<tac:replaceChildren contextNodeID="weatherContent" parseOnServer="true">
    <%@ include file="weatherTable.jsp" %>
</tac:replaceChildren>

</tac:taconiteRoot>

```

多亏 Taconite 框架的神奇魔力，在处理一个 Ajax 请求的响应时要想动态地更新页面，现在不用再花时间写 W3C DOM 方法了。相反，Taconite 会为你生成 JavaScript，生成的 JavaScript 将返回给浏览器，并在浏览器上执行来生成所需的结果。

下面一行一行地分析 weatherForecastAjax.jsp 的代码，以便了解 Taconite 框架是怎么工作的。文件的第一行只是一个 JSP taglib 声明，声明了 Taconite taglib。

tac:taconiteRoot 标记是所有 Taconite 响应的根标记。在后台，Taconite 总是向浏览器返回一个 XML 文档。在 XML 中可能嵌有 JavaScript（如果解析在服务器端完成）或 XHTML（如果解析在浏览器端完成）。tac:taconiteRoot 要作为所返回的 XML 的根标记。使用 Taconite 标记库的所有 JSP 文件都必须使用 tac:taconiteRoot 标记。

每次做出 Ajax 请求来更新天气预报时，都必须更新天气预报组件中的两项：位置（万一预报位置的相应 ZIP 编码有变）和具体的预报。tac:taconiteRoot 标记有两个 tac:replaceChildren 标记作为其直接子标记。第一个 tac:replaceChildren 标记更新预报位置，第二个标记更新天气预报。

具体的工作就是在 tac:replaceChildren 标记中完成，来生成最新的更新内容。tac:replaceChildren 标记是指：“对于有指定 id 属性值的元素，要用以下内容替换该元素的所有子元素”。tac:replaceChildren 标记有一个必要的属性：contextNodeID。contextNodeID 指定了要用 tac:replaceChildren 标记的内容替换哪个节点的子元素。

在这个例子中，第一个 tac:replaceChildren 标记的 contextNodeID 属性值为 forecastLocation。再看看代码清单 8-3，可以看到有一个 span 的 id 属性值为 forecastLocation。所以，这个 tac:replaceChildren 标记就是在告诉 Taconite 框架：forecastLocation span 元素的所有子元素都应当用 tac:replaceChildren 标记中指定的内容替换。指定的内容来自 weatherLocation.jsp 文件，前面已经看到了，这个文件只是显示要预报的位置。

第二个 tac:replaceChildren 标记会重建具体的天气预报。这个标记中的 contextNodeID 属性值为 weatherContent，再回头看看代码清单 8-3，可以看到一个 id 属性值为 weatherContent 的 div 标记。这个 weatherContent div 标记的所有子标记都会用第二个 tac:replaceChildren 标记指定的内容替换。其内容来自 weatherTable.jsp 文件（代

码清单 8-5), 这个JSP只是建立了包含天气预报的一个表。

在这个例子中, 每个`tac:replaceChildren`标记都有一个名为`parseOnServer`的属性。这是一个可选的属性, 如果没有这个属性, 其默认值为`true`。如果值为`true`, 则指示Taconite要在服务器端将指定内容解析为JavaScript, 并把JavaScript嵌在响应XML中发回给浏览器。如果值为`false`, 则指示把内容解析为JavaScript的工作会在浏览器上完成。

现在应该很清楚了, 为什么要把天气预报分成这么多单独的JSP文件。第一次呈现整个Ajax Dashboard页面时, 主`weatherForecast.jsp`文件会显示天气预报组件。`WeatherForecast-Ajax.jsp`文件处理预报组件的Ajax更新。另外两个JSP用于显示预报位置和具体预报。你可能不想写重复的代码, 所以预报位置和预报表分别放在自己单独的JSP文件中, 并包含在`weatherForecast.jsp`和`weatherForecastAjax.jsp`文件中。这样一来, 不论是Ajax Dashboard第一次加载还是根据一个Ajax请求更新, 预报位置或预报表的更新都是一样的。

`weatherForecastAjax.jsp`文件的输出到底是什么? 代码清单 8-8 显示了一个 XML 响应的例子, `weatherForecastAjax.jsp`会生成这样一个响应并返回给浏览器。

#### 代码清单 8-8 `weatherForecast.jsp`

```
<taconite-root>

    <taconite-replace-children contextNodeID="forecastLocation"
                                parseInBrowser="false">
        <![CDATA[
            var element0 = document.createDocumentFragment();
            element0.appendChild(document.createTextNode
                ("Weather for NEW ULM, MN"));
            while (document.getElementById("forecastLocation")
                .childNodes.length > 0) {
                document.getElementById("forecastLocation").removeChild
                    (document.getElementById("forecastLocation").childNodes[0]);
            }
            document.getElementById("forecastLocation").appendChild(element0);
        ]]>
    </taconite-replace-children>

    <taconite-replace-children contextNodeID="weatherContent"
                                parseInBrowser="false">
        <![CDATA[
            var element0 = document.createDocumentFragment();
            var element1 = document.createElement("table");
            element0.appendChild(element1);
            var element2 = document.createElement("tbody");
            element1.appendChild(element2);
        ]]>
    </taconite-replace-children>
```

```

var element3 = document.createElement("tr");
element2.appendChild(element3);
var element4 = document.createElement("td");
element3.appendChild(element4);
element4.appendChild(document.createTextNode("Sun. 08/28"));
var element5 = document.createElement("br");
element4.appendChild(element5);
element4.appendChild(document.createTextNode("High: 78"));
element5 = document.createElement("br");
element4.appendChild(element5);
element4.appendChild(document.createTextNode("Low: 54"));
element5 = document.createElement("br");
element4.appendChild(element5);
element5 = document.createElement("img");
element5.setAttribute("src",
    "http://www.nws.noaa.gov/weather/images/fcicons/sct.jpg");
element5.setAttribute("alt", "forecast image");
element4.appendChild(element5);

<!-- Other cells omitted for brevity -->
while (document.getElementById("weatherContent").childNodes
.length > 0) {
    document.getElementById("weatherContent").removeChild
    (document.getElementById("weatherContent").childNodes[0]);
}
document.getElementById("weatherContent").appendChild(element0);
}]]>
</taconite-replace-children>

</taconite-root>

```

注意, XML 最前面有一个 `taconite-root` 标记, 它有两个 `taconite-replace-children` 标记, 这类似于代码清单 8-7 中的标记结构。每个 `taconite-replace-children` 有一个 CDATA 部分, 其中包含用于适当更新页面内容的内嵌 JavaScript。如果仔细阅读代码清单 8-8 中的内嵌 JavaScript, 可以很容易地看到这个 JavaScript 在做什么。实际上, 如果你自己手工编写 JavaScript 来更新页面的话, 可能与这里嵌入的 JavaScript 是一样的! 接收到来自服务器的这个 XML 响应之后, Taconite 浏览器端的库会使用 `eval` 函数自动抽出并执行嵌入的 JavaScript。

### 8.7.5 分析标题新闻组件

标题新闻组件是 Ajax Dashboard 中最简单的组件。这个组件会按一定间隔定期轮询服务器, 并用最新的标题新闻更新其内容。不同于其他三个组件, 新闻组件不会向服务器发送任何数据。相反, 它只是要求得到最新的新闻, 并显示这些新闻。由于不会向服务器发送数据, 具体发送请求所需的代码就会少一些, 如下所示:

```
function updateNewsItems() {  
    var ajaxRequest = new AjaxRequest("UpdateNewsItems");  
    ajaxRequest.sendRequest();  
}
```

这个组件的实现与天气预报组件稍有区别。它们都依赖于 Taconite 框架来动态地创建 JSP 体中指定的内容。天气预报组件使用 Taconite 的 JSP 标记来构建返回给服务器的响应。与之不同，新闻组件没有使用 Taconite 的 JSP 标记，而是依赖于开发人员来构建返回给客户的 XML 响应。在这种情况下，把 HTML 解析为 JavaScript 会在浏览器上完成，而不是在服务器上进行。

不使用 Taconite 的 JSP 标记有什么好处呢？主要的优点在于，这种方法展示了 Taconite 框架可以做到与服务器端技术独立。只要服务器上使用的工具可以生成 XML 文档，就可以采用这种方法与 Taconite 框架一同使用。

使用浏览器端解析器而不是服务器端解析器还有一个很好的副作用，就是可以降低带宽使用。使用 JavaScript 解析器时，所需的 HTML 内容会原样返回给浏览器，即只是作为 HTML 返回。使用服务器端 Java 解析器时，HTML 内容会翻译成多个 JavaScript 命令，并把这些命令返回给浏览器执行。

JavaScript 命令与它们所表示的 HTML 内容相比，JavaScript 命令要详细得多，所以如果带宽很紧张，可能就需要使用 JavaScript 解析器来完成解析。不过，要记住，Java 解析器可能比 JavaScript 解析器快得多，所以如果性能很重要（而且是在 Java 环境下），可能就要使用 Java 解析器。

代码清单 8-9 显示了呈现新闻组件的 newsItems.jsp 文件。类似于天气预报组件，它有一个表示组件窗口的外层 div 元素，后面是另一个 div 元素，用来显示组件的彩色标题栏。

#### 代码清单 8-9 newsItems.jsp

```
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>  
  
<div id="newsItemsRoot" style="left:400px; top:300px;">  
    <div id="newsItemsHandle">  
        <table width="100%" border="0" class="textbox">  
            <tr>  
                <td align="left" class="controls">  
                    Top News Stories Powered by Yahoo!  
                </td>  
                <td align="right">  
                    <a class="controls"  
                        href="javascript:minimize('newsItemsContent');">  
                        -</a>  
                </td>  
            </tr>  
        </table>  
    </div>  
</div>
```

```

        &nbsp;
        <a class="controls"
            href="javascript:maximize('newsItemsContent');">
        +
        </a>
    </td>
</tr>
</table>
</div>

<div id="newsItemsContent" class="newsItemsContent">

    <%@include file="newsItemDetail.jsp"%>

</div>

</div>

```

要注意id属性值为newsItemContent的div元素。在这个div元素中有一个JSP include，包含了一个名为newsItemDetail.jsp的JSP文件。与天气预报组件类似，这个新闻组件也分成多个单独的JSP文件，并包含在适当的位置，以避免重复代码。

代码清单 8-10 显示了 newsItemDetail.jsp 文件的内容。这个 JSP 的工作就是迭代处理所有新闻项，对于每个新闻项，JSP 会为该新闻项创建一个链接，并把这个链接放在其自己的 div 标记中。

#### 代码清单 8-10 newsItemDetail.jsp

```

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

<c:forEach var="newsItem" items="${newsItems}">
    <div>
        <br/>
        <a href="${newsItem.link}" class="newsLink" target="blank">
            ${newsItem.title}
        </a>
    </div>
</c:forEach>

```

现在可以看到，把代码放在多个单独的 JSP 文件中组织确实很有好处。每个 JSP 文件表示一个可以轻松重用的集中的工作单元，只需把它包含在其他 JSP 文件中，这样当这些工作单元有变化时就能减轻维护的负担。

只要浏览器做出一个 Ajax 请求来更新新闻项，就可以重用 newsItemDetail.jsp 文件。

不需要重新绘制整个组件。所要更新的只是放置新闻列表的 div 元素的内容。为此单独建立了 newsItemsAjax.jsp 文件。

浏览器提交一个 Ajax 请求，要求更新新闻项时，就要用到 newsItemsAjax.jsp 文件。这个文件只关心更新新闻项，而不是重新绘制整个新闻组件。与天气预报组件类似，它会使用 Taconite 框架，但是这一次为了展示 Taconite 框架可以与任何服务器端技术一同使用，所以没有使用 Taconite 的 JSP 标记库。

代码清单 8-11 显示了 newsItemsAjax.jsp 文件。由于这个例子没有使用 Taconite 的任何定制 JSP 标记，因此必须手工地将响应的 Content-Type 首部设置为 text/xml。注意它与代码清单 8-8 的相似性（代码清单 8-8 是 Taconite 的 JSP 标记库生成的具体 XML）。

#### 代码清单 8-11 newsItemsAjax.jsp

```
<%@page contentType="text/xml"%>

<taconite-root>
    <taconite-replace-children contextNodeID="newsItemsContent"
                                parseInBrowser="true">
        <%@include file="newsItemsDetail.jsp"%>
    </taconite-replace-children>
</taconite-root>
```

这里没有使用 tac:taconiteRoot 和 tac:replaceChildren JSP 标记，这个例子只是用了 XML 标记，而没有依赖于 Taconite 的定制 JSP 标记库来完成。注意，就像 tac:replace-Children JSP 标记一样， taconite-replace-children 标记有一个 contextNodeID 属性，它指定了页面上现有的哪个元素要把其内容替换为指定的内容。parseInBrowser 属性值为 true 则是指示 Taconite JavaScript 客户：首先要把 HTML 内容解析为 JavaScript 命令，然后才能执行命令。

在 taconite-replace-children 标记中嵌有一个 JSP include 元素，包含了 newsItemsDetail.jsp 文件。前面已经看到，这个文件会具体完成 HTML 内容生成以呈现新闻项链接。

newsItemsAjax.jsp 文件呈现的 XML 会作为 Ajax 请求的响应返回给浏览器。Taconite 客户读取响应，并注意到 taconite-replace-children 标记的 parseInBrowser 属性设置为 true。因此，Taconite 的 taconite-replace-children 标记不包含要执行的 JavaScript。实际上，其内容是 HTML，首先需要解析为 JavaScript。taconite-parser.js 文件包含将 HTML 解析为相应 JavaScript 命令的 JavaScript 代码。

#### 8.7.6 如何完成自动重新刷新工作

Ajax Dashboard 上的天气预报、股票行情和标题新闻组件会以预定间隔自动重新刷新。

另外，每次用户编辑其属性时，天气预报和股票行情组件也会自行更新。你可能想知道如何实现这种自动更新。

要完成自动更新，需要某种定时器，从而能以预定间隔执行一个指定的 JavaScript 函数。幸运的是，在 window 对象中已经预置了这样一个定时器。虽然官方标准中没有明确指定，但所有当前的浏览器都支持 window.setInterval 方法。

window.setInterval 方法有两个参数。第一个参数是一个串，指明了应当按预定间隔调用哪个 JavaScript 函数。第二个参数是一个整数，指明了调用指定函数的时间间隔（毫秒数）。window.setInterval 方法返回一个整数，表示赋给相应间隔的惟一 ID。可以使用这个惟一 ID 作为 window.clearInterval 方法的参数，这个方法用于停止计算间隔。

代码清单 8-12 显示了 stockQuote.js 文件，这个文件会处理所有用户输入，并管理股票行情组件的自动更新。

#### 代码清单 8-12 stockQuote.js

```
var stockTickerUpdateIntervalID = 0;

function handleStockTickersChange() {
    clearStockTickerUpdateInterval();
}

function clearStockTickerUpdateInterval() {
    if(stockTickerUpdateIntervalID != 0) {
        window.clearInterval(stockTickerUpdateIntervalID);
    }
    document.getElementById("trackFlag").checked = false;
}

function updateStockQuote() {
    var ajaxRequest = new AjaxRequest("UpdateStockQuote");
    ajaxRequest.addFormElementsById("stockTickers");
    ajaxRequest.sendRequest();
}

function startUpdateStockQuoteInterval() {
    stockTickerUpdateIntervalID = window.setInterval("updateStockQuote()", 30000);
}

function handleTrackFlagClick() {
    var trackFlag = document.getElementById("trackFlag");

    if(trackFlag.checked) {
        updateStockQuote();
    }
}
```

```

        startUpdateStockQuoteInterval();
    }
    else {
        clearStockTickerUpdateInterval();
    }
}

```

这个文件的开头是一个全局变量 `stockTickerUpdateIntervalID`。这个变量保存的是更新股票行情的间隔的唯一 ID。这个变量初始化为 0。

用户在显示所关心股票的文本框中编辑内容时会禁用自动刷新。文本框的 `onkeyup` 事件触发时，会调用 `handleStockTickersChange` 函数。`handleStockTickersChange` 函数调用了 `clearStockTickerUpdateInterval` 函数，它使用 `stockTickerUpdateIntervalID` 来清除间隔，相应地停止股票行情的自动刷新。然后取消复选框的选择，指示用户自动刷新已经临时禁用。

`updateStockQuote` 函数是具体完成 Ajax 请求的函数。与本章前面的例子类似，它使用 Taconite JavaScript 客户端来完成大量麻烦的工作，包括创建 XMLHttpRequest 对象、向查询串增加所需的参数，以及发送请求。

`startUpdateStockQuoteInterval` 函数使用 `window.setInterval` 方法来启动股票行情的自动刷新。`window.setInterval` 调用的第一个参数是一个表示 `updateStockQuote`<sup>1</sup> 函数的串，第二个参数是表示时间延迟的一个整数，即指定函数调用之间的间隔时间。要注意 `window.setInterval` 方法返回了这个特定间隔的唯一 ID，它保存在 `stockTickerUpdateIntervalID` 变量中，这个变量后面还将用于清除此间隔。

最后，`handleTrackFlagClick` 函数会处理复选框的 `onclick` 事件。如果选中复选框，会调用 `updateStockQuote` 函数立即刷新股票行情，然后调用 `startUpdateStockQuoteInterval` 函数启用自动刷新。如果因为用户点击取消了复选框的选择，就会调用 `clearStockTickerUpdateInterval` 函数禁用自动刷新。

与股票行情组件的自动刷新类似，天气预报和标题新闻组件的自动刷新实现也如上所述。可以看到，自动刷新并不难实现。关键是把任务分解为小的、可重用的函数，然后在这些可重用的函数之上完成任务。整个文件不超过 40 行代码。更好的一点是，多亏了 Taconite 框架，完成 Ajax 请求只需 5 行 JavaScript 代码！

### 8.7.7 构建更好的 autocomplete

第 4 章展示了如何构建一个简单的 `autocomplete` 特性，每次用户在一个文本框中键入时，

---

1. 原文此处为 `startUpdateStockQuoteInterval`，有误。——译者注

下拉列表会以 Google Suggest 的方式显示一组建议。

Ajax Dashboard 的搜索组件就是以这个例子为基础。第 4 章的例子使用了硬编码的搜索“结果”，与之不同，这里的例子将使用 Yahoo! Search API 来完成搜索。Ajax Dashboard 同样使用了 Taconite 框架来建立搜索结果。

代码清单 8-13 是从第 4 章的 autoComplete.html 例子中抽出的。代码清单 8-13 中所列的函数负责构建结果下拉列表。

### 代码清单 8-13 第 4 章 autoComplete.html 例子的片段

```
function setNames(the_names) {
    clearNames();
    var size = the_names.length;
    setOffsets();

    var row, cell, txtNode;
    for (var i = 0; i < size; i++) {
        var nextNode = the_names[i].firstChild.data;
        row = document.createElement("tr");
        cell = document.createElement("td");

        cell.onmouseout = function() {this.className='mouseOver';};
        cell.onmouseover = function() {this.className='mouseOut';};
        cell.setAttribute("bgcolor", "#FFFAFA");
        cell.setAttribute("border", "0");
        cell.onclick = function() { populateName(this); } ;

        txtNode = document.createTextNode(nextNode);
        cell.appendChild(txtNode);
        row.appendChild(cell);
        nameTableBody.appendChild(row);
    }
}
```

set\_names 函数负责读取服务器返回的 XML，并构建结果。set\_names 函数中的 JavaScript 并不难。只不过以这种方式写 HTML 内容很麻烦，也不太自然。不过，要记住，构建 onmouseover、onmouseout 和 onclick 事件处理器需要非标准的代码；否则在所有主流浏览器上都无法工作。如果你不知道如何解决这些问题，就会不明白为什么这个例子在某些浏览器上能运行，而在另外一些浏览器上不能工作。

下面，将代码清单 8-13 中的代码与代码清单 8-14 中的代码做一个比较，代码清单 8-14 显示了 searchAutocomplete.jsp。代码清单 8-14 展示了如何使用 Taconite 框架来构建结果下拉列表的内容。

**代码清单 8-14 searchAutocomplete.jsp**

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ taglib uri="http://taconite.sf.net/tags" prefix="tac" %>

<tac:taconiteRoot>

    <tac:replaceChildren contextNodeID="popup" parseOnServer="true">

        <c:forEach var="result" items="${results}">
            <div onmouseover="hilite(this); onmouseout="unhilite(this);>
                <a href="${result.url}" class="autocomplete" target="_blank">
                    ${result.title}
                </a>
            </div>
        </c:forEach>

    </tac:replaceChildren>

</tac:taconiteRoot>
```

代码清单 8-13 和代码清单 8-14 中的代码完成的任务是一样的。它们都会构建在结果下拉列表中显示的结果。(代码清单 8-13 中的 JavaScript 把各个结果放在一个表行中, 代码清单 8-14 中的 JSP 则把各个结果放在一个 div 元素中)。JSP/Taconite 方法不光更为简短(18 行, 而代码清单 8-13 有 23 行), 还可以保证代码清单 8-14 中的代码更易于编写, 而且自然得多。你可能想直接使用 HTML 标记来编写 HTML 标记, 而不是通过 JavaScript 由程序来创建 HTML 标记。

更妙的是, 还记得那些麻烦的 onmouseover、onmouseout 和 onclick 事件处理器吗? 在 searchAutocomplete.jsp 中, 这些事件处理器只是编写为 HTML 元素(div)的简单属性。Taconite 会自动生成跨浏览器的 JavaScript 来创建这些事件处理器, 所以你不必自己记住该怎么解决这些问题。

相对于第 4 章的 autocomplete 例子, 搜索组件有了许多改进。在第 4 章的 autocomplete 例子中, 用户每次在文本框中键入一个字符时都会完成一个 Ajax 请求。如果用户敲得快, Ajax 请求的频率就会比较高, 这就增加了服务器的负载, 对用户来说并没有好处, 因为敲得太快会使击键之间的间隔太短, 不足以显示更新的结果。

这里的搜索组件会以另外一种方式处理这个问题。并非对文本框中的每个击键都做出响应, 一旦用户开始键入, 就会以预定的间隔做 Ajax 请求来更新结果下拉列表, 然后只有当文本框的内容自上一次发出请求后有所变化时才会发出 Ajax 请求。

## 8.8 小结

我们已经了解了 Ajax 是多么简单，希望从现在开始就能在你的应用中使用这些技术。在本章中，我们介绍了一个更复杂的例子，来展示 Ajax 的一些好处。我们还提供了一些资源，以便你进一步加深理解。根据我们展示的一些常用模式，希望你已经了解了如何将 Ajax 用于你的工作，记住，还要避免本章中谈到的一些常见陷阱。最后一点，Taconite 之类的框架可以大大简化 Ajax 开发工作，你应该已经有所体会，这些框架确实值得一用。

# 开发跨浏览器 JavaScript

自 1999 年以来，浏览器已经走了很长的路，这归功于一些标准机构（如 W3C 和 ECMA），也多亏浏览器市场重又恢复了竞争的氛围，当前浏览器从很大程度上消除了专用的扩展和特异的行为，而这些曾在 20 世纪 90 年代末让浏览器备受困扰。为使应用在不同的浏览器上都能正常工作，开发人员花费了大量的时间去调整 HTML 布局和 JavaScript 代码，这种日子将一去不复返。如今，开发人员编写代码时只要遵循标准，就能保证所写的代码在与标准兼容的所有浏览器上都能正确地工作。

在应用中实现 Ajax 技术，意味着要使用 JavaScript 动态地更新页面内容，可以是创建新内容、删除现有内容，或者修改现有内容。遗憾的是，在某些浏览器中确实还存在着一些特异性，这就会导致奇怪的行为，让开发人员头疼不已。

这其中最不守规矩的就是 IE。IE 的 HTML 呈现引擎和 JavaScript 环境在过去几年中已历经数次更新，所以 IE 是当前与标准最不兼容的浏览器。不过，由于 IE 还持有着浏览器市场中的大部分份额，所以编写的 JavaScript 代码不仅要在其他浏览器上有正常表现，在 IE 上也要能有效地工作。

本附录将介绍在哪些情况下很可能出现浏览器不兼容性。这里所列的并不完全，我们谈到的只是实现 Ajax 技术时最有可能遇到的一些情况。相应的解决方法在当前所有浏览器上都能正常工作。

## A.1 向表中追加行

在以往使用 Ajax 的经历中，你很可能会使用 JavaScript 向现有的表中追加行，或者从头创建包含表行的新表。`document.createElement` 和 `document.appendChild` 方法可以使这很容易做到，只需使用 `document.createElement` 创建表单元格，再使用 `document.appendChild` 方法将这些表单元格增加到表行。接下来的编辑步骤是使用 `document.appendChild` 将表行增加到表中。

在 Firefox、Safari 和 Opera 等当前浏览器中，这样做是可以的。不过，如果使用的是 IE，表行则不会在表中出现。更糟糕的是，IE 甚至不抛出任何错误，或对表行确实已经追加到表中却不会显示出来的问题提供任何线索。

在这种情况下，解决方法很简单。IE 允许将 `tr` 元素增加到 `tbody` 元素，而不是直接增加到 `table` 元素。例如，如果定义一个空表如下：

```
<table id="myTable">
  <tbody id="myTableBody"></tbody>
</table>
```

向这个表中增加行的正确做法是把行增加到表体，而不是增加到表，如下所示：

```
var cell = document.createElement("td").appendChild(document.createTextNode("foo"));
var row = document.createElement("tr").appendChild(cell);
document.getElementById("myTableBody").appendChild(row);
```

幸运的是，这种方法在所有当前浏览器上都能用，也包括 IE。如果你养成习惯，总是使用表中的表体，就不用担心这个问题了。

## A.2 通过JavaScript设置元素的样式

利用 Ajax 技术，开发人员创建的 Web 应用可以与服务器无缝地通信，而无需完全页面刷新。这种应用的用户可能认为与服务器来回传递数据时页面会闪一下。但对于 Ajax 请求，是不会出现这种页面闪烁的，所以用户可能不知道页面上有些数据已经更新。你可能想修改某些元素的样式，指示页面上一些数据已经改变。例如，如果股票的价格已经通过 Ajax 请求得到了无缝更新，可以加亮显示这支股票的名字。

可以通过 JavaScript 使用元素的 `setAttribute` 方法设置元素的样式。例如，要把 `span` 元素中的文本修改为采用红色粗体显示，可以使用 `setAttribute` 方法如下：

```
var spanElement = document.getElementById("mySpan");
spanElement.setAttribute("style", "font-weight:bold; color:red;");
```

除了 IE，这种方法在当前其他浏览器上都是行得通的。对于 IE，解决方法是使用元素 `style` 对象的 `cssText` 属性来设置所需的样式，尽管这个属性不是标准的，但得到了广泛支持，如下所示：

```
var spanElement = document.getElementById("mySpan");
spanElement.style.cssText = "font-weight:bold; color:red;";
```

这种方法在 IE 和大多数其他浏览器上都能很好地工作，只有 Opera 除外。为了让代码在所有当前浏览器上都可移植，可以同时使用这两种方法，也就是既使用 `setAttribute` 方法，又使用元素 `style` 对象的 `cssText` 属性，如下所示：

```
var spanElement = document.getElementById("mySpan");
spanElement.setAttribute("style", "font-weight:bold; color:red;");
spanElement.style.cssText = "font-weight:bold; color:red;";
```

这样一来，在当前所有浏览器上都能正常地设置元素的样式了。

### A.3 设置元素的**class**属性

读过前一节后，了解到可以通过 JavaScript 来设置元素的内联样式，你可能想当然地认为，简单地设置元素的 **class** 属性应该是最容易的了。遗憾的是，并不如此。与设置元素内联样式类似，通过 JavaScript 动态地设置元素的 **class** 属性时也存在一些特异性。

你可能已经猜到了，IE 是当前浏览器中的一个异类，不过解决方法倒也相当简单。使用 Firefox 和 Safari 之类的浏览器时，可以使用元素的 **setAttribute** 方法来设置元素的 **class** 属性，如下所示：

```
var element = document.getElementById("myElement");
element.setAttribute("class", "styleClass");
```

奇怪的是，如果使用 **setAttribute** 方法，并指定属性名为 **class**，IE 并不会设置元素的 **class** 属性。相反，只使用 **setAttribute** 方法时 IE 会自己识别 **className** 属性。

对于这种情况，完备的解决方法是：使用元素的 **setAttribute** 方法时，将 **class** 和 **className** 都用作属性名，如下所示：

```
var element = document.getElementById("myElement");
element.setAttribute("class", "styleClass");
element.setAttribute("className", "styleClass");
```

当前大多数浏览器都会使用 **class** 属性名而忽略 **className**，IE 则正好相反。

### A.4 创建输入元素

输入元素为用户提供了与页面交互的手段。HTML 本身有一组有限的输入元素，包括单行文本框、多行文本区、选择框、按钮、复选框和单选钮。你可能想使用 JavaScript 动态地创建这样一些输入元素作为 Ajax 实现的一部分。

单行文本框、按钮、复选框和单选钮都可以创建为输入元素，只是 **type** 属性的值有所不同。选择框和文本区有自己特有的标记。通过 JavaScript 动态地创建输入元素很简单（但单选钮除外，这在“创建单选钮”一节中再做解释），只要遵循一些简单的规则就行。使用 **document.createElement** 方法可以很容易地创建选择框和文本区，只需向 **document.createElement** 传递元素的标记名，如 **select** 或 **textarea**。

单行文本框、按钮、复选框和单选钮稍难一些，因为它们都有同样的元素名 **input**，只是

`type` 属性的值不同。所以，要创建这些元素，不仅需要使用 `document.createElement` 方法，后面还要调用元素的 `setAttribute` 方法来设置 `type` 属性的值。这并不难，但确实要多加一行代码。

考虑在哪里把新创建的输入元素增加到其父元素中，必须注意 `document.createElement` 和 `setAttribute` 语句的顺序。在某些浏览器中，只有创建了元素，而且正确地设置了 `type` 属性时，才会把新创建的元素增加到其父元素中。例如，以下代码段在某些浏览器中可能有奇怪的行为：

```
document.getElementById("formElement").appendChild(button);
button.setAttribute("type", "button");
```

为了避免奇怪的行为，要确保创建输入元素后设置其所有属性，特别是 `type` 属性，然后再把它增加到父元素中，如下：

```
var button = document.createElement("input");
button.setAttribute("type", "button");
document.getElementById("formElement").appendChild(button);
```

遵循这个简单的规则，有助于消除以后可能出现的一些难于诊断的问题。

## A.5 向输入元素增加事件处理程序

向输入元素增加事件处理程序应该与使用 `setAttribute` 方法并指定事件处理程序的名字和所需函数处理程序的名字一样容易，对吗？错。设置元素的事件处理程序的标准做法是使用元素的 `setAttribute` 方法，它以事件名作为属性名，并把函数处理程序作为属性值，如下所示：

```
var formElement = document.getElementById("formElement");
formElement.setAttribute("onclick", "doFoo();");
```

除了 IE，上面的代码在所有当前浏览器中都能工作。如果在 IE 中使用 JavaScript 设置元素的事件处理程序，必须对元素使用点记法来引用所需的事件处理程序，并把它赋为匿名函数，这个匿名函数要调用所需的事件处理程序，如下所示：

```
var formElement = document.getElementById("formElement");
formElement.onclick = function() { doFoo();};
```

注意，是如何通过点记法从 `formElement` 引用 `onclick` 事件处理程序。`onclick` 事件处理程序赋为一个匿名函数，这个匿名函数只是调用了所需的事件处理程序，在这个例子中事件处理程序就是 `doFoo`。

幸运的是，这种技术得到了 IE 和所有其他当前浏览器的支持，所以完全可以通过 JavaScript 动态地设置表单元素的事件处理程序。

## A.6 创建单选钮

最好的总是留在最后。通过 JavaScript 动态地创建单选钮是很费劲的，因为 IE 中创建单选钮的方法与其他浏览器所用的方法大相径庭。

除了 IE，当前所有其他浏览器都允许使用以下方法创建单选钮（这些方法应该能想得到）：

```
var radioButton = document.createElement("input");
radioButton.setAttribute("type", "radio");
radioButton.setAttribute("name", "radioButton");
radioButton.setAttribute("value", "checked");
```

这样就能在除 IE 以外的所有当前浏览器中创建单选钮，而且能正常工作。在 IE 中，单选钮确实会显示出来，但是无法将其选中，因为点击单选钮并不按我们预想得那样使之选中。在 IE 中，创建单选钮的方法与其他浏览器所用的方法完全不同，而且根本不兼容。对于前面建立的单选钮，在 IE 中可以如下建立：

```
var radioButton = document.createElement("<input type='radio' name='radioButton' value='checked'>");
```

好在，IE 中确实可以通过 JavaScript 动态地创建单选钮，只不过难一些，而且与其他浏览器不兼容。

如何克服这个限制呢？答案很简单，这就是需要某种浏览器嗅探（browser-sniffing）机制，使得在创建单选钮时脚本就能知道该使用哪个方法。幸运的是，你不用检查大量不同的浏览器，假设只使用现代浏览器，脚本只需要在 IE 和其他浏览器间进行区分就行了。

IE 能识别出名为 uniqueID 的 document 对象的专用属性，名为 uniqueID。IE 也是唯一能识别这个属性的浏览器，所以 uniqueID 很适合用来确定脚本是不是在 IE 中运行。

使用 document.uniqueID 属性来确定脚本在哪个浏览器中运行时，可以结合 IE 特定的方法和标准兼容的方法，这就会得到以下代码：

```
if(document.uniqueID) {
    //Internet Explorer
    var radioButton = document.createElement("<input type='radio' name='radioButton' value='checked'>");
}
else {
    //Standards Compliant
    var radioButton = document.createElement("input");
    radioButton.setAttribute("type", "radio");
    radioButton.setAttribute("name", "radioButton");
    radioButton.setAttribute("value", "checked");
}
```

## A.7 小结

使用 JavaScript 开发 Web 应用不再像在几年前开发时那么痛苦了。如今，当前浏览器的 W3C DOM 标准和 JavaScript 的实现都惊人地相似。不过还是存在一些特异性（主要是因为 IE），因此仍然需要一些解决方法。本附录中给出的提示可以帮助你编写出“通用”的 JavaScript，从而在大多数当前浏览器上都能正确地工作。通过采用这里介绍的技术，你能节省大量调试和测试的时间。脚本万岁！

## Ajax 框架介绍

到此为止，你可能已经注意到，使用 Ajax 编程时有很多麻烦事。如果你要支持多个浏览器（现在还有谁只支持一个浏览器呢？），无疑会遭遇不兼容问题。单看一个简单的动作，比如说创建 XMLHttpRequest 对象的一个实例，这需要先进行浏览器测试。一旦开始尝试使用 Ajax 技术，你很快就会注意到要反复地完成同样的一些任务。当然，你可以收集一些常用代码库，甚至创建自己的框架。不过，做这个工作之前，需要先了解一下现在已经有些什么了。

与所有优秀技术一样，Ajax 已经催生出大量框架，有了这些框架，开发人员的日子好过多了。我们要强调一点，Ajax 还很新，而且还在发展，框架领域也同样如此。几乎每天都有新来者，目前还看不出谁是最后的赢家。2003 年 6 月之前，这方面的框架还不多，所以在以后的几个月可能还会有巨大变化。

有些框架基于客户端，有些基于服务器端；有些专门为特定语言设计，另外一些则与语言无关。其中绝大多数都有开源实现，但也有少数是专用的。我们不会面面俱到地谈到每一个框架，而且也不可能深入分析提到的每个框架。我们的出发点很明确，就是让你对现在有些什么有所认识。在你读到本附录时，我们提到的一些工具包可能已经销声匿迹，另外的则可能刚刚创建。哪个框架最适合你？对于这个问题，只有你自己有发言权；不过，在框架领域稳定之前，你可以持一种保守的态度。甚至还有人在着力将各种框架合并在一起，等这个工作结束时应该会有好戏看！当你读到本书时，情况应该会更加明朗，但也许你还想了解一下目前的情况<sup>1</sup>。

### B.1 浏览器端框架

下面几节介绍了一些浏览器端框架。

---

1. 对你来说，这可能要算是“过去”的情况了。——译者注

### B.1.1 Dojo

Dojo 是最老的框架之一，于 2004 年 9 月开始开发。这个项目的目标是建立充分利用 XHR 的 DHTML 工具包，并把重心放在可用性问题上。Dojo 只有几个文件，不用处理 XHR 的建立，只需调用 `bind` 方法，并传入想调用的 URL 和回调方法即可。就这么简单。还可以使用 `bind` 方法来提交整个表单。

Dojo 有一个特性使它独树一帜，这就是它支持向后和向前按钮。尽管这个特性不一定在每个浏览器上都能用（遗憾的是，Safari 就是一个异类），但你确实可以注册一个回调方法，在用户点击了向后按钮或向前按钮时触发这个方法。Dojo 还提供了 `changeURL` 标记，力图解决使用 Ajax 所固有的书签问题。

Dojo 看上去是相对成熟的工具包之一，它把重点放在可用性上，这一点很不错。Dojo 表现得相当稳定，在它身后还有一些支撑力量。Dojo 的邮件列表相当活跃，多看一些文档可能更有帮助。可以在 [dojotoolkit.org](http://dojotoolkit.org) 得到更多相关信息。

### B.1.2 Rico

Rico 是市场上最新的框架之一，由 Sabre Airline Solutions 开发，随后又成为开源实现。当然，`rico` 在西班牙语里就是 `rich`，说明这个项目的总目标是提供一组组件来开发丰富的因特网应用。它得到了广泛的浏览器支持，不过让人不解的是 Safari<sup>1</sup>并不支持 Rico。

与 Dojo 关注可用性不同，Rico 则是针对拖放动作、数据网格和所谓的电影效果（移动部件、淡入淡出等等）而设计。Rico 网站上有很多有意思的演示版（DEMO），并且提供了代码。如果开发人员想尽快了解 Rico，并且运行起来，这是一个很好的起点。相关的文档不多，不过随着这个框架的日渐成熟，这种情况会有所改观。

Rico 可以作为单个文件下载，不过你可能还需要 Prototype JS 库。更多有关的信息请访问 [openrico.org/home.page](http://openrico.org/home.page)。

### B.1.3 qooxdoo

qooxdoo 也是 Ajax 框架领域的一个新成员，它提供了一个基于 JavaScript 的工具包来弥补 HTML 的不足。尽管还处在早期的 alpha 阶段，但 qooxdoo 确实提供了一些相当引人注目的部件。使用 qooxdoo，可以模拟标准胖客户应用中的一些特性，如菜单条、工具提示、网格布局和拖放支持。

---

1. 在使用 Ajax 技术时，IE 最不合群，总是有一些奇怪的行为，很不合标准，人们对这一点颇多诟病，IE 也明智地接受了大多数抱怨，问问本书作者就知道了！可是，写这本书时，我们还发现 Safari 也有如此之多的“奇特”行为（这让最近转向 Mac 的人大为失望）。

qooxdoo 确实有一些有用的文档，还对底层细节提供了很有帮助的解释。qooxdoo 的魅力显然体现在它的复杂部分上。如果你的目标是创建瘦应用，并希望它与胖客户应用相差无几，就可以试试 qooxdoo。更多有关的信息请访问 [qooxdoo.oss.schlund.de](http://qooxdoo.oss.schlund.de)。

#### B.1.4 TIBET

你觉得 Ajax 最早是什么时候出现的？根据对此的解释，也许会认为 TIBET 可能是现存最老的框架。根据文档所述，TIBET 小组从 1997 年就开始开发这个工具包，他们的目标是提供企业级 Ajax 支持。TIBET 看上去不只是包装了 XMLHttpRequest 对象，它还对 Web 服务和底层协议提供了支持，并且提供了 Google、Amazon 和许多其他常用服务的预置包装器。

真正让 TIBET 卓而不群的是，它有一个完全交互式的基于浏览器的 IDE，这能大大简化开发、调试和单元测试。更多有关的信息请访问 [www.technicalpursuit.com](http://www.technicalpursuit.com)。

#### B.1.5 Flash/JavaScript 集成包

在 Ajax 之前，Flash 很是风行，很多 Web 网站都建立在 Flash 平台上。那些曾对 Flash 狠下一番功夫的人不想完全放弃 Flash，利用这个开源项目就能同时利用 Ajax 技术。这个工具包在所有主要浏览器上都能用，使得 JavaScript 能够调用 ActionScript，ActionScript 也能调用 JavaScript。可以来回传递大量对象，包括日期、串和数组。

Flash/JavaScript 集成包的安装涉及一些 JavaScript 文件，以及两个用于 Flash 的库函数。从页面上调用 ActionScript 函数只需几行代码而已。有关的文档相当少，不过，如果你想使用 Ajax 访问 Flash，这个工具包就很值得研究。更多有关的信息请访问 [weblogs.macromedia.com/flashjavascript/](http://weblogs.macromedia.com/flashjavascript/)。

#### B.1.6 Google AJAXSLT

基于 Google Maps 的工作，Google AJAXSLT 是使用 XPath 的 XSL 转换（XSLT）的 JavaScript 实现。XSLT 可以把 XML 文档转换为其他语言，如 HTML。AJAXSLT 允许使用 JavaScript 在浏览器上直接完成这些转换。

Google AJAXSLT 在所有主要浏览器上都能工作，它是在 BSD 许可证下发布的。这个工具包很小，包括几个 JavaScript 文件，还有一些方便的测试页。Google AJAXSLT 不是十全十美的，不过，如果 Google Suggest 有所提示，我们希望 Google AJAXSLT 的缺点能很快解决。因为 Google 是最先使用 Ajax 的网站之一，我们会很有兴致地看到在未来几个月它还会有所增加。更多有关的信息请访问 [goog-ajaxslt.sourceforge.net](http://goog-ajaxslt.sourceforge.net)。

#### B.1.7 libXmlRequest

libXmlRequest 框架也是比较老的一个框架，早在 2003 年就已经发布了。这个框架包括

一个 JavaScript 文件，它相当于 XMLHttpRequest 对象的一个包装器，提供了两个重载的请求函数：`getXml` 和 `postXml`。另外，它有一些处理缓冲池和缓存的属性，还有一些工具函数处理常见的任务，如解析来自服务器的 XML 以及修改 DOM。

这个工具包能在哪些浏览器上运行，这一点还不是很清楚，而且有关的文档相当少。这个工具包版权归其作者 Stephen W. Cote 所有，其中没有提到许可问题。因此，只能用它帮助你产生灵感。更多有关的信息请访问 [www.whitefrost.com/index.jsp](http://www.whitefrost.com/index.jsp)。

### B.1.8 RSLite

RSLite 是远程脚本的一个实现，由 Brent Ashley 编写。从技术上讲，它没有利用作为 Ajax 核心的 XMLHttpRequest 对象，但是得到了更广泛的浏览器支持。如果你需要支持原来的浏览器，而这些浏览器不支持 XMLHttpRequest 对象，就可以试试 RSLite。RSLite 是相当轻量级的，已从 2000 年发展至今<sup>1</sup>。更多有关的信息请访 [www.ashleyit.com/rs/rslite/](http://www.ashleyit.com/rs/rslite/)。

### B.1.9 SACK

SACK（简单 Ajax 代码包）开发为一个瘦包装器，包装了 XMLHttpRequest 对象。其作者 Gregory Wild-Smith 认为，其他的许多框架太过复杂，而且做了许多本不该它们完成的任务。所以他创建了 SACK 来简化 Ajax 的开发。SACK 包括几个可以简化服务器调用的方法。比起具体创建适当的 XMLHttpRequest 对象实例来说，用更少的代码就能向服务器发送数据，并处理响应。

SACK 由一个 JavaScript 文件组成，其中包含很少的代码。SACK 底层软件的发布得到了修改 X11 许可（也称为 MIT 许可），与大多数开源项目一样，它的文档并不多，不过，入门肯定还是绰绰有余的。SACK 的真正强大之处在于它的简单性，如果你要找的是一个基本包装器，可以试试 SACK。更多有关的信息请访问 [twilightuniverse.com/projects/sack/](http://twilightuniverse.com/projects/sack/)。

### B.1.10 sarissa

sarissa 有一点是 Ajax 做不到的，它以一种独立于浏览器的方式对 XML API 提供了包装支持。利用这个框架，创建和使用 XMLHttpRequest 对象实在是小菜一碟（不用检查浏览器，它已经为你处理好了）。另外，sarissa 还对使用 DOM 提供了支持。类似于 Google AJAXSLT，sarissa 也支持 XSLT，它模拟了 IE 上的 Mozilla 处理器。

sarissa 只包括几个类，在 GPL 协议下发布。Mozilla/Firefox 和 IE 都充分支持 sarissa，只在 Opera、Konqueror 和 Safari 浏览器上有些函数不能用。更多有关的信息请访问 [sarissa.sourceforge.net](http://sarissa.sourceforge.net)。

1. 前面提到，Ajax 框架是从 2003 年发展到现在，不过 RSLite 确实从 2000 年就开始发展。RSLite 是关于远程脚本的，而远程脚本出现在 Ajax 之前。

[sourceforge.net/doc/。](http://sourceforge.net/doc/)

### B.1.11 XHConn

XHConn 类似于 SACK，它相当于 XMLHttpRequest 对象的一个简单包装器。你不用直接使用 XMLHttpRequest 对象，只需首先启动一个 XHConn 实例，与使用 XHR 同样的方法加以处理。也就是说，无需浏览器检查，并提供了一种简单的方法来确定浏览器是否支持 XHR（这对于需要妥善降级的网站尤其方便）。

XHConn 在 Safari、IE、Mozilla、Firefox 和 Opera 上都能工作。类似于大多数 Ajax 框架，这是一个开源实现，在 Creative Commons License 协议下发布。XHConn 是一个代码不多的文件，不过它确实做到了该做的事情——简化 Ajax。更多有关的信息请访问 [xkr.us/code/javascript/XHConn/](http://xkr.us/code/javascript/XHConn/)。

## B.2 服务器端框架

以下介绍服务器端的框架。

### B.2.1 CPAINT

CPAINT（跨平台异步接口工具包）在服务器端实现 Ajax，它向客户返回文本或 DOM 文档对象，以便用 JavaScript 处理。CPAINT 在大多数主要浏览器上都能用，而且支持远程脚本，在 GPL 协议下发布。这个项目的文档相当完备，不过，CPAINT 只支持 PHP 和 ASP。更多有关的信息请访问 [sourceforge.net/projects/cpaint/](http://sourceforge.net/projects/cpaint/)。

### B.2.2 Sajax

利用 Sajax，可以直接从 JavaScript 调用服务器端代码。Sajax 支持 Perl、Python、Ruby 和 ASP 等语言（不过奇怪的是，目前并不支持 Java）。安装 Sajax 相当简单，只涉及针对特定服务器语言的简单的库。Sajax 的开发社区极其活跃。已经确认的只有 IE 6 和 Mozilla/Firefox 提供 Sajax 支持，不过本书作者认为它在 Safari 上也能很好地使用。更多有关的信息请访问 [www.modernmethod.com/sajax](http://www.modernmethod.com/sajax)。

### B.2.3 JSON/JSON-RPC

JavaScript 对象注解（JSON）是一种文本格式，与 XML 很相似，可以用于交换数据。JSON 的设计要保证两方面，一方面便于人阅读，另一方面便于机器解析，它使用了 C 系列语言类似的约定。与 JSON 相关的还有 JSON-RPC，这是一个远程过程调用（RPC）协议，类似于 XML-RPC，但面向的是 JSON 语言。作为规约，JSON-RPC 在许多语言中都有实现，包括 Java、Ruby、Python 和 Perl。

由于 JSON-RPC 是规约，你需要知道哪个特定实现适用于你的环境，还要充分了解特定的实现。取决于具体的实现，有些实现的文档相当完备，有些则根本没有。开发人员的参与程度也有很大不同。关于 JSON-RPC 规约的讨论已经有些少了。更多有关的信息请访问 [www.crockford.com/JSON/index.html](http://www.crockford.com/JSON/index.html)。

### B.2.4 Direct Web Remoting

利用 Direct Web Remoting (DWR)，你能从 JavaScript 直接调用 Java 方法，就好像它们是浏览器的本地方法一样。尽管后台严格限制为 Java，但 DWR 仍然是最流行的框架之一。DWR 的文档是最棒的，还有一些有用的例子可以帮助你入门。

安装并不难，不过还要编辑 Web 应用的部署描述文件，另外要编辑 DWR 特定的文件。DWR 配置文件指定了可以远程创建和调用的类，而且文档中警告用户：从浏览器调用服务器确实存在一些安全问题。除了包含服务器端代码的 JAR 文件，另外还有两个 JavaScript 文件包含了一些辅助函数。DWR 适用于一些常见的 Web 框架，如 Struts 和 Tapestry，在 Apache 协议下发布。如果想从 Web 页面调用 Java 方法，DWR 能助你一臂之力。更多有关的信息请访问 [getahead.ltd.uk/dwr/index](http://getahead.ltd.uk/dwr/index)。

### B.2.5 SWATo

Shift Web Applications TO (SWATO) 也是一个基于 Java 的 Ajax 框架解决方案。SWATO 在所有 Servlet 2.3 或更高版本的容器中都能工作，类似于 DWR，它也需要对配置文件做一些更新。有意思的是，SWATO 充分利用了 JSON 来完成客户和服务器之间数据的编组，与本附录中讨论的其他一些框架相似，它也允许从浏览器调用服务器端 Java。为了帮助开发人员，SWATO 包括许多可复用的组件，如自动完成文本框等。

与使用其他框架相比，使用 SWATo 要相对复杂一些，要访问的类需要实现一个 SWATo 接口。不过，其文档相当完备，对于入门来讲绰绰有余。SWATO 设计为使用 Spring 来打包服务，但是不一定非得如此。更多有关的信息请访问 <https://swato.dev.java.net/doc/html/>。

### B.2.6 Java BluePrints

Sun 的 BluePrints 小组一直忙于将 Ajax 纳入他们的解决方案目录(Solutions Catalog)中。Solutions Catalog 包括一些很好的文档，描述了如何使用基本 Ajax，如何实现自动完成，如何创建一个进度条以及如何验证表单。它还包括 JavaServer Faces 组件。为 BluePrints Solutions Catalog 开发的代码可以从 [www.java.net](http://www.java.net) 网站得到。

### B.2.7 Ajax.Net

Ajax.Net 之于 Microsoft .NET 就相当于 Sajax、DWR 和 SWATo 之于 Java。利用 Ajax.Net，

你能从 JavaScript 客户端调用.NET 方法。Ajax.NET 包括一个 DLL，可以与 VB .NET 或 C#一同使用。Ajax.NET 的文档很好地展示了针对各种场景的解决方案，而且能得到相关的源代码。不过，Ajax.NET 的许可协议很不明确。更多有关的信息请访问 [ajax.net](http://ajax.net)。

### B.2.8 Microsoft 的 Atlas 项目

Microsoft 在 Ajax 领域涉足的时间已经不短了，毕竟，XMLHttpRequest 对象是 Microsoft 发明的，而且从 1998 年开始就已经用在 Web 版本的 Outlook 中。Microsoft 把重点放在提供一个更加健壮的开发环境上，从而让开发人员的工作更轻松。Microsoft 的着眼点还不只这些，还力图提供客户端脚本框架、ASP.NET 控件和 Web 服务集成。Microsoft 还发布了 Atlas 项目，作为其 ASP.NET 2.0 预览版的一部分。有 Microsoft 的介入，开发人员的工具包可能会比今天充实得多。更多有关的信息请访问 [beta.asp.net/default.aspx?tabindex=7&t-abid=47](http://beta.asp.net/default.aspx?tabindex=7&t-abid=47)。

### B.2.9 Ruby on Rails

Rails 是一个令人兴奋的新 Web 框架，建立在 Ruby 语言基础上。如今，Rails 已经得到了大量关注（在 Google 上查一下 Rails，可以找到更多信息），这是因为使用 Rails 能够快速开发基于 Web 的应用。开发 Basecamp 时，37signals 小组提出名为 Rails 的框架。Basecamp 正是 Ajax 应用的主要示例，所以看到 Rails 对 Ajax 提供如此充分的支持，我们不应感到奇怪。Rails 有许多内置的 JavaScript 库，其中包装了很多常用的特性，它还包含一个模块，其中包装了 Ruby 的 JavaScript 调用。如果你在使用 Rails，就会发现 Ajax 非常简单。更多有关的信息请访问 [www.rubyonrails.org](http://www.rubyonrails.org)。